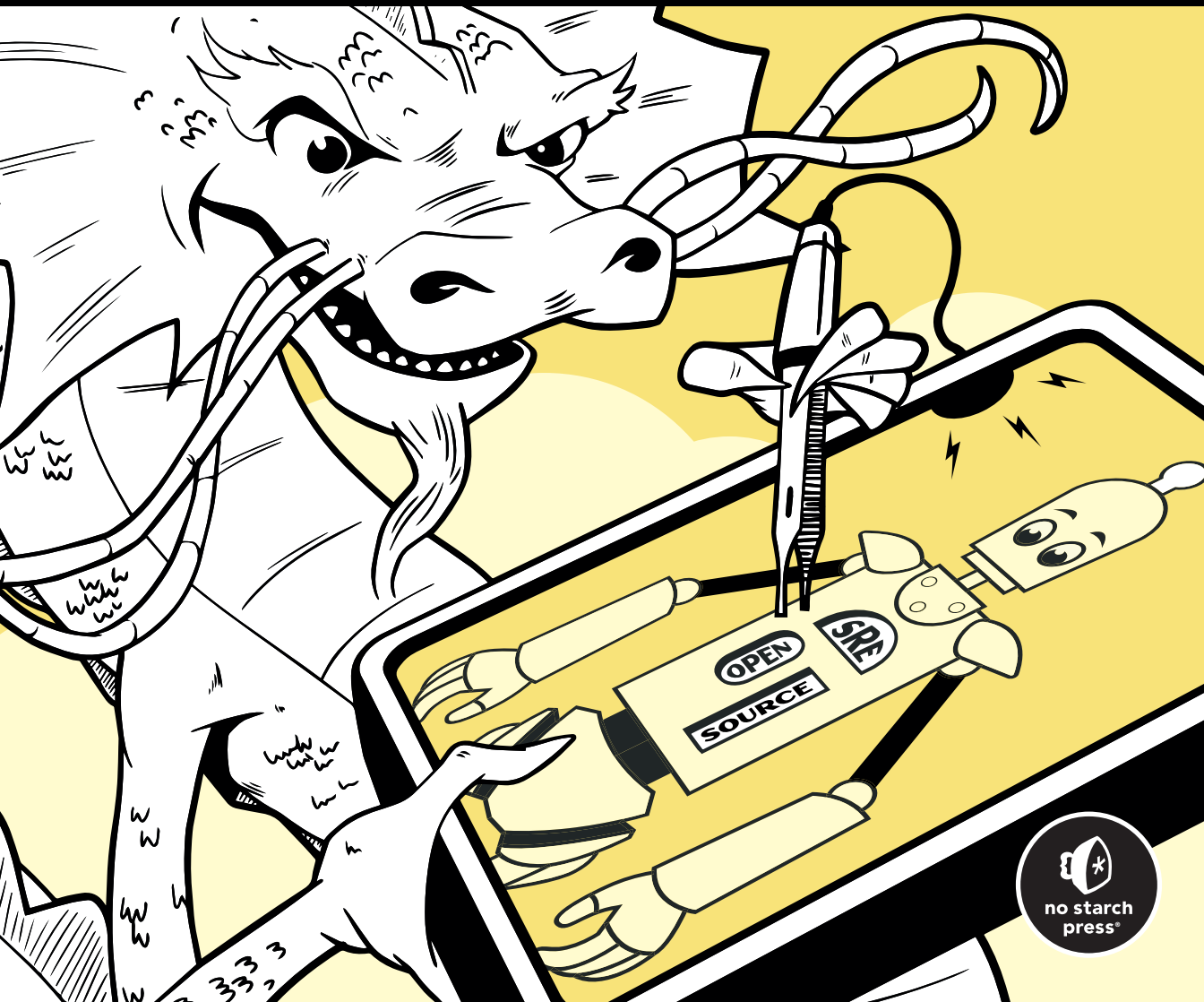


2ND
EDITION

THE GHIDRA BOOK

THE DEFINITIVE GUIDE

KARA NANCE AND CHRIS EAGLE



PRAISE FOR *THE GHIDRA BOOK*

“Chris Eagle and Kara Nance have—once more—delivered a very readable and hands-on book on reverse engineering using a publicly available tool. This second edition . . . will continue to make the black art of reverse engineering more accessible.”

—SVEN DIETRICH, BOOK REVIEW EDITOR,
IEEE CIPHER

“*The Ghidra Book*’s second edition is a must read for any (aspiring) reverse engineer to learn more about the analyst’s mindset, as well as Ghidra! ”

—MAX “LIBRA” KERSTEN, MALWARE ANALYST

“From an introduction to disassembly and working with the basics of Ghidra to scripting in Ghidra to extend its capabilities, this book covers it all.”

—TYLER REGULY, TRIPWIRE BOOK CLUB

“Rather than simply being a Ghidra user guide, the authors did an exceptional job of laying out many of the fundamental concepts involved in software reverse engineering.”

—CRAIG YOUNG, PRINCIPAL SECURITY
RESEARCHER, TRIPWIRE

“I encourage anyone that has an interest in reverse engineering or who just wants to investigate cool open-sourced tools to give *The Ghidra Book* a read.”

—MATTHEW JERZEWSKI, SECURITY
RESEARCHER, TRIPWIRE

THE GHIDRA BOOK

2nd Edition

The Definitive Guide

by Kara Nance and Chris Eagle



**no starch
press®**

San Francisco

THE GHIDRA BOOK, 2ND EDITION. Copyright © 2026 by Kara Nance and Chris Eagle.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

30 29 28 27 26 1 2 3 4 5

ISBN-13: 978-1-7185-0468-4 (print)

ISBN-13: 978-1-7185-0469-1 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Miles Bond
Developmental Editor: Frances Saux
Cover Illustrator: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Brian Hay
Copyeditor: George Hale
Proofreader: Scout Festa

Library of Congress Control Number for the first edition: 2020938508

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the authors nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

To Hazel and all young minds passionate about
investigating and understanding technology,
and the men and women who support and encourage them.
Dream big and never stop exploring!

About the Authors

Kara Nance is a private security consultant. She has been a professor of computer science for many years. She serves on the HoneyNet Project Board of Directors and has given numerous talks at security conferences around the world. She enjoys building Ghidra extensions and regularly provides Ghidra training.

Chris Eagle has been reverse engineering software for 45 years. He is the author of *The IDA Pro Book* (No Starch Press, 2011) and is a highly sought-after provider of reverse engineering training. He has published numerous reverse engineering tools and given talks at conferences such as Blackhat, DEF CON, and ShmooCon.

About the Technical Reviewer

Brian Hay is a reverse engineer, professor, and software developer. He has spoken and taught at many conferences and is currently a senior researcher for a boutique security research company. He specializes in designing and developing virtualized environments for training and testing exciting tools like Ghidra.

BRIEF CONTENTS

Acknowledgments	xxiii
Introduction	xxv

PART I: GETTING STARTED

Chapter 1: Introduction to Disassembly	3
Chapter 2: Reversing and Disassembly Tools	15
Chapter 3: Meet Ghidra	31

PART II: BASIC GHIDRA USAGE

Chapter 4: Beginning Your Analysis	41
Chapter 5: Exploring Ghidra's Data Displays	57
Chapter 6: Making Sense of a Disassembly	91
Chapter 7: Refining a Disassembly	119
Chapter 8: Working with Data Types and Data Structures	147
Chapter 9: Understanding Cross-References	185
Chapter 10: Using Graph Views	199

PART III: CUSTOMIZING AND EXTENDING GHIDRA

Chapter 11: Using Ghidra Collaboratively	221
Chapter 12: Customizing Ghidra	245
Chapter 13: Extending Ghidra's Worldview	279
Chapter 14: Basic Scripting with Ghidra and PyGhidra	303
Chapter 15: Integrated Scripting with Eclipse and GhidraDev	337
Chapter 16: Running Ghidra in Headless Mode	365

PART IV: A DEEPER DIVE

Chapter 17: Loaders	387
Chapter 18: Processors	427

Chapter 19: The Decompiler	453
Chapter 20: Compiler Variations	469

PART V: REAL-WORLD APPLICATIONS

Chapter 21: Obfuscation and Emulation	495
Chapter 22: Patching Binaries	515
Chapter 23: BSim and Other Comparison Tools	539
Appendix: Ghidra for IDA Users	571
Index	579

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

xxiii

INTRODUCTION

xxv

Who Should Read This Book?	xxv
What's New in This Edition?	xxvi
What's in This Book?	xxvi

PART I GETTING STARTED

1 INTRODUCTION TO DISASSEMBLY

3

Disassembly Theory	4
The What of Disassembly	4
The Why of Disassembly	5
Malware Analysis	6
Vulnerability Analysis	6
Software Interoperability	6
Compiler Validation	7
Debugging Displays	7
The How of Disassembly	7
A Basic Disassembly Algorithm	8
Linear Sweep Disassembly	9
Recursive Descent Disassembly	10
Summary	14

2 REVERSING AND DISASSEMBLY TOOLS

15

Classification Tools	15
file	16
PE-bear	18
Detect-It-Easy	18
Summary Tools	19
nm	19
ldd	21
objdump	23

otool	24
dumpbin	24
c++filt	25
Deep Inspection Tools	26
strings	26
Disassemblers	28
Summary	29

3

MEET GHIDRA

31

Licenses	31
Versions	32
Support Resources	32
Downloading Ghidra	33
Installing Ghidra	34
Exploring the Directory Layout	34
Starting Ghidra	37
Summary	37

PART II

BASIC GHIDRA USAGE

4

BEGINNING YOUR ANALYSIS

41

Launching Ghidra	41
Creating a New Project	43
Loading Files	44
Using the Raw Binary Loader	47
Analyzing Files	48
Auto Analysis	49
Desktop Behavior During Initial Analysis	53
Saving and Exiting Options	54
Ghidra Desktop Tips and Tricks	55
Summary	55

5

EXPLORING GHIDRA'S DATA DISPLAYS

57

CodeBrowser	57
CodeBrowser Windows	60
The Listing Window	63
Additional Disassembly Windows	67

The Function Graph Window	68
The Program Trees Window	73
The Symbol Tree Window	74
The Data Type Manager Window	77
The Console Window	78
The Decompiler Window	78
Other Ghidra Windows	81
The Bytes Window	81
The Defined Data Window	82
The Defined Strings Window	84
The Symbol Table and Symbol References Windows	85
The Memory Map Window	88
The Function Call Graph Window	89
The Function Call Trees Window	89
Summary	90

6

MAKING SENSE OF A DISASSEMBLY

91

Disassembly Navigation	92
Names and Labels	92
Navigation in Ghidra	93
The Go To Dialog	94
Navigation History	95
Stack Frames	96
Function Call Mechanics	96
Calling Conventions	97
Application Binary Interfaces	101
Local Variable Layout	102
Ghidra's Stack Frame Analysis	103
Setting Up the Stack	104
Representing Stack Frame Information	108
Searching	114
Searching Program Text	115
Searching Memory	116
Summary	118

7

REFINING A DISASSEMBLY

119

Manipulating Names and Labels	120
Renaming Parameters and Local Variables	121
Renaming Labels	125
Adding a New Label	126
Editing Labels	127
Removing a Label	128
Navigating Labels	129

Comments	129
End-of-Line Comments	130
Pre and Post Comments	131
Plate Comments	131
Repeatable Comments	132
Parameter and Local Variable Comments	133
Annotations	133
Basic Code Transformations	134
Changing Code Display Options	135
Formatting Instruction Operands	136
Manipulating Functions	138
Converting Data to Code (and Vice Versa)	141
Basic Data Transformations	142
Specifying Data Types	142
Working with Strings	144
Defining Arrays	145
Summary	146

8

WORKING WITH DATA TYPES AND DATA STRUCTURES

147

Making Sense of Data	148
Recognizing Data Structure Use	150
Array Member Access	150
Structure Member Access	160
Creating Structures with Ghidra	166
Creating a New Structure	167
Editing Structure Members	169
Applying Structure Layouts	171
C++ Reversing Primer	172
The this Pointer	173
Virtual Functions and Vtables	173
The Object Life Cycle	177
Name Mangling	180
Runtime Type Identification	180
Inheritance Relationships	181
Summary	183

9

UNDERSTANDING CROSS-REFERENCES

185

Referencing Basics	186
Cross-References (Back References)	187
References in Practice	190
Reference Management Windows	195
XRefs	195
References To	196

Symbol References	196
Advanced Reference Manipulation.....	197
Summary	198

10 USING GRAPH VIEWS 199

Basic Blocks	200
Function Graphs.....	200
Using Graph Toolbars	202
Splitting Blocks	205
Interacting with Function Graphs.....	206
Function Call Graphs	207
Trees	210
The Graph Menu	212
Graph Services	212
The Graph Menu in Practice	214
Summary	217

PART III CUSTOMIZING AND EXTENDING GHIDRA

11 USING GHIDRA COLLABORATIVELY 221

Starting Ghidra Server	222
Configuration.....	222
Server Administration.....	225
Creating a Shared Project	226
Project Window Menus	228
File	229
Edit	232
Project	233
Project Repositories	236
Version Control	237
An Example Scenario	240
Summary	244

12 CUSTOMIZING GHIDRA 245

Customizing the CodeBrowser	246
Rearranging Windows	246
Editing Tool Options	247
Editing the Tool	251
Accessing Special Tool Editing Features	251
Saving the CodeBrowser Layout	253

Customizing the Project Window	254
Creating New Tools	259
Example 1: Building a Debugger Tool	259
Example 2: Performing Dynamic Analysis with the Debugger Tool	265
Saving Workspaces	277
Summary	278

13 EXTENDING GHIDRA'S WORLDVIEW 279

Importing Files	280
Analyzers	283
Word Models	284
Data Types	286
Loading Additional Data Type Archives	287
Creating New Data Type Archives	288
Function IDs	290
The Function ID Plugin	292
Example: Identifying UPX Functions	293
Example: Profiling a Static Library	297
Summary	301

14 BASIC SCRIPTING WITH GHIDRA AND PYGHIDRA 303

The Script Manager	304
The Script Manager Window	304
The Script Manager Toolbar	306
Script Development	307
Scripting in Java	307
Editing a Script: Regex Search	309
Scripting in Python	314
Scripting with PyGhidra	316
Example: Using PyGhidra to Add Warnings	317
An Introduction to the Ghidra API	321
The Address Interface	321
The Symbol Interface	322
The Reference Interface	322
The GhidraScript Class	322
The Program Class	327
The Function Interface	328
The Instruction Interface	329

Ghidra Scripting Examples	329
Example 1: Enumerating Functions.....	329
Example 2: Enumerating Instructions	330
Example 3: Enumerating Cross-References	330
Example 4: Finding Function Calls	331
Example 5: Emulating Assembly Language Behavior.....	333
Summary	335

15 INTEGRATED SCRIPTING WITH ECLIPSE AND GHIDRADEV 337

Eclipse	338
Integrating Ghidra and Eclipse	338
Starting Eclipse	338
Editing Scripts with Eclipse	340
The GhidraDev Menu	340
The GhidraDev New Menu	341
The Package Explorer	346
Example: Building a Ghidra Analyzer Module	352
Step 1: Defining the Problem	353
Step 2: Creating the Eclipse Module	353
Step 3: Building the Analyzer	353
Step 4: Testing the Analyzer in Eclipse	360
Step 5: Adding the Analyzer to Ghidra	360
Step 6: Testing the Analyzer in Ghidra	361
Summary	363

16 RUNNING GHIDRA IN HEADLESS MODE 365

Getting Started with the Headless Analyzer	366
Step 1: Launching Ghidra	367
Steps 2, 3, and 4: Creating a New Ghidra Project and Importing a File	367
Steps 5 and 6: Auto Analyzing the File, Saving, and Exiting	368
Options and Parameters	371
General Options	371
Server Options	376
Script Options	377
Writing Scripts	378
Example 1: Headless ROP Gadget Identification	378
Example 2: Automated Function ID Database Creation	383
Summary	383

PART IV A DEEPER DIVE

17

LOADERS

387

Unknown File Analysis	388
Using the Data Type Manager	389
Creating Segments	394
Finalizing the Memory Map	396
Identifying Code	398
Example 1: Simple Shellcode Loader Module	400
Step 0: Taking a Step Back	401
Step 1: Defining the Problem	404
Step 2: Creating the Eclipse Module	404
Step 3: Building the Loader	404
Step 4: Adding the Loader to Ghidra	410
Step 5: Testing the Loader in Ghidra	410
Example 2: Simple Shellcode Source Loader	412
Step 1: Modifying the Response to the Importer Poll	413
Step 2: Finding the Shellcode in the Source Code	413
Step 3: Converting Shellcode to Byte Values	414
Step 4: Loading the Converted Byte Array	414
Steps 5: Testing the Loader	414
Example 3: Simple ELF Shellcode Loader	416
Step 1: Performing Housekeeping	417
Step 2: Researching the ELF Header Format	417
Step 3: Finding Supported Load Specifications	418
Step 4: Loading File Content into Ghidra	420
Step 5: Formatting Data Bytes and Adding an Entry Point	421
Step 6: Understanding Language Definition Files	422
Step 7: Creating the Opinion Files	423
Step 8: Observing the Results	424
Summary	426

18

PROCESSORS

427

Understanding Ghidra Processor Modules	428
Eclipse Processor Modules	428
The SLEIGH Language	430
Processor Manuals	431
Modifying Processor Modules	433
Problem Statement	434
Example 1: Adding an Instruction to a Processor Module	435

Example 2: Modifying an Instruction in a Processor Module	442
Example 3: Adding a Register to a Processor Module	451
Summary	452

19 THE DECOMPILER 453

Decompiler Analysis	453
Analysis Options	454
Eliminating Unreachable Code	455
Simplifying Predication	456
The Decompiler Window	456
Example 1: Editing in the Decompiler Window	457
Example 2: Non-Returning Functions	462
Example 3: Automated Structure Creation	463
Summary	468

20 COMPILER VARIATIONS 469

switch Statements	470
Jump Tables	471
Binary Search	472
Example: Comparing gcc to the Microsoft C/C++ Compiler	476
Compiler Build Options	478
Example 1: Modulo Operator	479
Example 2: The Ternary Operator	481
Example 3: Function Inlining	483
Compiler-Specific C++ Implementations	485
Function Overloading	485
RTTI Implementations	485
Locating the main Function	490
Example 1: _start to main with gcc on Linux x86-64	491
Example 2: _start to main with Microsoft's C/C++ compiler	491
Summary	492

PART V REAL-WORLD APPLICATIONS

21 OBFUSCATION AND EMULATION 495

Anti-Reverse Engineering	496
Obfuscation	496
Anti-Static Analysis Techniques	496

Static Deobfuscation	498
Unpacking Binaries	499
Navigating Obfuscation	499
Resolving Indirect Jumps and Obfuscated Calls	500
Recovering Data and Strings	500
Emulating Code	501
Emulation vs. Live Debugging	502
The Ghidra Emulator	502
Use Cases	503
Building an Emulator	505
Step 1: Defining the Problem	507
Step 2: Creating the Eclipse Script Project	507
Step 3: Setting Up the Emulator	508
Step 4: Selecting the Address Range to Emulate	509
Step 5: Checking for an Instruction	509
Step 6: Performing Emulation	509
Step 7: Writing Memory Back to the Program	510
Step 8: Cleaning Up Resources	511
Step 9: Testing the Script	511
Summary	513

22

PATCHING BINARIES

515

Planning Your Patch	516
Finding Program Content to Change	516
Memory	517
Direct References	518
Instruction Patterns	518
Specific Behaviors	522
Applying Your Patch	523
Making Basic Changes	523
Making Oversized Patches	529
Exporting Patched Files	532
Ghidra Export Formats	532
The Original Format Export	533
Example: Evading Anti-Debugger Checks	534
Summary	537

23

BSIM AND OTHER COMPARISON TOOLS

539

Comparing Files with the Program Diff Tool	540
Example: Spotting a Patched Instruction	540
Program Diff Options	542
Example: Merging Two Analyzed Files	545
Comparing Functions with the Function Comparison Provider	549
Example: Revisiting Patched Functions	550
The Function Comparison Toolbar	551
Example: Comparing Crypto Routines	553
Comparing Behaviors with BSim	557
How BSim Works	558
The BSim Workflow	558
Example: Investigating Ransomware Evolution	559
Comparing Versions with Version Tracking	567
Correlators	568
Sessions	569
Summary	569

APPENDIX: GHIDRA FOR IDA USERS

571

Database Creation	572
Basic Windows and Navigation	574
The Listing View	574
The Graph View	575
The Decompiler	575
The Symbol Tree	575
The Data Type Manager	576
Scripting	576
Summary	577

INDEX

579

ACKNOWLEDGMENTS

This second edition was greatly strengthened by the dedicated team at No Starch Press. We are deeply grateful to Jill Franklin, Sabrina Plomitullo-González, Frances Saux, George Hale, Scout Festa, and Miles Bond for their thoughtful guidance, careful attention, and steady support throughout the revision process. We also extend our thanks to Bill Pollock for his continued belief in this project and for supporting its evolution from the first edition to this one. This edition stands on the strong groundwork established by the No Starch Press first-edition team: Barbara Yien, Athabasca Witschi, Laurel Chun, Katrina Taylor, Barton D. Reed, Sharon Wilkey, and Danielle Foster. We remain grateful for their efforts.

We would like to thank our technical editor, Brian Hay, for reviewing our many words and examples. His knowledge and experience with Ghidra has helped ensure that the technical content in this book is solid, and his teaching experience guided our presentation so that the material is presented in a way that appeals to both new and experienced reverse engineers. We are especially grateful for his contributions across both editions, which have strengthened the book in countless ways.

Our appreciation also extends to the entire Ghidra development team, past and present, at the National Security Agency and beyond for building Ghidra and sharing it with the world as an open source project.

Kara would like to thank Ben for his patience while she learned about technology and Katie for her patience while she wrote about it. In preparing this updated edition, she is especially grateful to Wesley McGrew, whose generosity in sharing his knowledge of Ghidra and reverse engineering has been both guiding and inspiring, and to Casey Smith, whose steady flow of ideas never fails to be entertaining and welcome. She also thanks Brian for his humor and unwavering support. Without the encouragement and contributions of all of you, this book would not have been possible.

INTRODUCTION



Ghidra is a complex, open source reverse engineering tool suite, and our goal in writing this book is to provide a resource that introduces it to both current and future reverse engineers.

In the hands of a skilled reverse engineer, Ghidra streamlines the analysis process and allows users to customize its capabilities to suit their individual needs. Ghidra is also very accessible to new reverse engineers, particularly with its included decompiler, which can help them more clearly understand the relationships between high-level language and disassembly listings as they begin exploring the world of binary analysis.

Who Should Read This Book?

This book is intended for aspiring and experienced software reverse engineers. If you don't already have reverse engineering experience, that's okay, as the early chapters provide the background material necessary to introduce you to reverse engineering and enable you to explore and analyze binaries with Ghidra.

We assume a basic understanding of programming because you will need to be familiar with concepts such as functions, variables, loops, and data structures to understand what you see when analyzing binaries. You do not need to be an expert programmer, but you should be comfortable reading code snippets and reasoning through program logic.

Experienced reverse engineers who want to add Ghidra to their toolkits might choose to move quickly through the first two parts to gain a basic understanding of Ghidra and then jump to specific chapters of interest. Advanced Ghidra users and developers may choose to focus on the later chapters, where they will learn how to create new Ghidra extensions and apply their experience to contribute new content to the Ghidra project.

What's New in This Edition?

The previous edition of this book was the first comprehensive book about Ghidra ever published. However, it describes a moving target, as the Ghidra community continues to improve and extend its capabilities. As with many open source projects, Ghidra began its public life with a rapid string of evolutionary releases. As Ghidra evolves, we aim to ensure that the book's content continues to help readers effectively use Ghidra to address their reverse engineering challenges.

This updated edition reflects the continued evolution of the open source project and brings readers up to speed with expanded real-world examples, new tools like PyGhidra and BSim, enhanced debugger and emulator capabilities, new graph functionality, and seamless IDE integration. Clear, practical examples are woven into every chapter, helping you leverage Ghidra's growing ecosystem and modern development environments to their fullest potential.

As much as possible, we have tried to keep the book version agnostic. Fortunately, new releases of Ghidra are well documented, with detailed listings of changes that provide version-specific guidance should you encounter any differences between the book and your version of Ghidra.

What's in This Book?

The book is divided into five parts. Part I introduces disassembly, reverse engineering, and the Ghidra project. Part II covers basic Ghidra usage. Part III demonstrates ways you can customize and automate Ghidra to make it work for you. Part IV takes a deeper dive into explaining specific types of Ghidra modules and supporting concepts. Part V demonstrates how you can apply Ghidra to real-world situations a reverse engineer is likely to encounter.

Part I: Getting Started

Chapter 1: Introduction to Disassembly This introductory chapter walks you through the theory and practice of disassembly and discusses some of the pros and cons associated with the two common disassembly algorithms.

Chapter 2: Reversing and Disassembly Tools This chapter discusses the major categories of tools available for reverse engineering and disassembly.

Chapter 3: Meet Ghidra In this chapter, you get to meet Ghidra, learn a little bit about its origin, and discover how you can obtain and start using this free, open source tool suite.

Part II: Basic Ghidra Usage

Chapter 4: Beginning Your Analysis Your journey using Ghidra begins in this chapter. You'll get your first glimpse of Ghidra in action as you create a project, analyze a file, and begin to understand the Ghidra graphical user interface (GUI).

Chapter 5: Exploring Ghidra's Data Displays In this chapter, you'll be introduced to the CodeBrowser, Ghidra's main tool for file analysis. You'll also explore the primary CodeBrowser display windows.

Chapter 6: Making Sense of a Disassembly This chapter explores the concepts that are fundamental to understanding and navigating Ghidra disassemblies.

Chapter 7: Refining a Disassembly In this chapter, you'll learn to supplement Ghidra's analysis and manipulate a Ghidra disassembly as part of your own analysis process.

Chapter 8: Working with Data Types and Data Structures In this chapter, you will learn how to manipulate and define simple and complex data structures found within compiled programs.

Chapter 9: Understanding Cross-References This chapter provides a detailed look at cross-references, how they support graphing, and the critical role they play in understanding a program's behavior.

Chapter 10: Using Graph Views This chapter introduces you to Ghidra's graphing capabilities and the use of graphs as binary analysis tools.

Part III: Customizing and Extending Ghidra

Chapter 11: Using Ghidra Collaboratively This chapter presents a unique capability within Ghidra: using Ghidra as a collaborative tool. You will learn how to configure a Ghidra server and share projects with other analysts.

Chapter 12: Customizing Ghidra In this chapter, you will begin to see how you can customize Ghidra by configuring projects and tools to support your individual analysis workflows and apply these skills to create a new debugger tool.

Chapter 13: Extending Ghidra's Worldview This chapter teaches you how to generate and apply library signatures and other specialized content so that Ghidra can recognize new binary constructs.

Chapter 14: Basic Scripting with Ghidra and PyGhidra In this chapter, you'll be introduced to the basic Ghidra scripting capabilities in Python and Java using Ghidra's inline editor as well as PyGhidra.

Chapter 15: Integrated Scripting with Eclipse and GhidraDev This chapter takes your Ghidra scripting to a whole new level by integrating IDEs into Ghidra and exploring the powerful scripting capabilities they provide, including a worked example of building a new analyzer.

Chapter 16: Running Ghidra in Headless Mode In this chapter, you'll be introduced to the use of Ghidra in headless mode, which requires no GUI, and explore the advantage of this mode for common large-scale and repetitive tasks.

Part IV: A Deeper Dive

Chapter 17: Loaders In this chapter, you'll take a deep dive into how Ghidra imports and loads files. You will have the opportunity to build new loaders to handle previously unrecognized file types.

Chapter 18: Processors This chapter introduces you to Ghidra's SLEIGH language for defining processor architectures. You will explore the process of adding new processors and instructions to Ghidra.

Chapter 19: The Decompiler In this chapter, you'll be provided with a closer look at one of Ghidra's most popular features: the Ghidra Decompiler. You will see how it works behind the scenes and how it can contribute to your analysis process.

Chapter 20: Compiler Variations This chapter helps you understand the variations you can expect to see in code compiled using different compilers and targeting different platforms.

Part V: Real-World Applications

Chapter 21: Obfuscation and Emulation You'll learn how to use Ghidra to analyze obfuscated code in a static context so that the code doesn't need to be executed.

Chapter 22: Patching Binaries This chapter teaches you some methods for using Ghidra to patch binaries during analysis, both within Ghidra itself and to create new patched versions of the original binaries.

Chapter 23: BSim and Other Comparison Tools This final chapter provides an overview of the Ghidra features that allow you to identify physical and behavioral differences between binaries, including a brief introduction to Ghidra's advanced version tracking capabilities.

Appendix: Ghidra for IDA Users If you are an experienced IDA user, this appendix will provide you with some tips and tricks for mapping IDA terminology and usage to similar functionality in Ghidra.

NOTE

Please visit the companion sites, <https://nostarch.com/ghidra-book-2e> and <https://ghidrabook.com>, to access the code listings contained in this book.

PART I

GETTING STARTED

1

INTRODUCTION TO DISASSEMBLY



You may be wondering what to expect in a book dedicated to Ghidra. While obviously Ghidra-centric, this book is not intended to come across as “The Ghidra User’s Manual.”

Instead, we intend to use Ghidra as the enabling tool for discussing reverse engineering techniques that you will find useful in analyzing a wide variety of software, ranging from vulnerable applications to malware.

When appropriate, we will provide detailed steps in Ghidra for performing specific actions related to the task at hand. As a result, we will take a rather roundabout walk through Ghidra’s capabilities, beginning with the basic tasks you will want to perform upon initial examination of a file and leading up to advanced uses and customization of Ghidra for more challenging reverse engineering problems. We make no attempt to cover all of Ghidra’s features; however, we do cover the features you will find most useful in meeting your reverse engineering challenges. This book will help make Ghidra the most potent weapon in your arsenal of tools.

Prior to diving into any Ghidra specifics, we will cover some of the basics of the disassembly process and review other tools available for reverse engineering compiled code. While these tools may not match the complete range of Ghidra’s capabilities, each does address specific subsets of Ghidra

functionality and offers valuable insight into specific Ghidra features. The remainder of this chapter is dedicated to understanding the disassembly process from a high level.

Disassembly Theory

Anyone who has spent any time at all studying programming languages has probably learned about the various generations of languages, but they are summarized here for those who may have been sleeping:

First-generation languages These are the lowest form of language, generally consisting of ones and zeros or a shorthand form, such as hexadecimal, and readable only by hex ninjas. Distinguishing data from instructions is difficult at this level because all of the content looks the same. First-generation languages may also be referred to as *machine languages*, and in some cases *byte code*, while machine language programs are often referred to as *binaries*.

Second-generation languages Also called *assembly languages*, second-generation languages are a mere table lookup away from machine language and generally map specific bit patterns, or operation codes (opcodes), to short but memorable character sequences called *mnemonics*. These mnemonics help programmers remember the instructions with which they are associated. An *assembler* is a tool used by programmers to translate their assembly language programs into machine language suitable for execution. In addition to instruction mnemonics, a complete assembly language generally includes *directives* to the assembler that help dictate the memory layout of code and data in the final binary.

Third-generation languages These languages take another step toward the expressive capability of natural languages by introducing keywords and constructs that programmers use as the building blocks for their programs. Third-generation languages are generally platform independent, though programs written using them may be platform dependent as a result of using features unique to a specific operating system. Often-cited examples of third-generation languages include FORTRAN, C, and Java. Programmers generally use compilers to translate their programs into assembly language or all the way to machine language (or some rough equivalent such as byte code).

Fourth-generation languages These exist but are not relevant to this book and are not discussed.

The What of Disassembly

In a traditional software development model, compilers, assemblers, and linkers are used by themselves or in combination to create executable programs. To work our way backward (or reverse engineer programs), we use tools to undo the assembly and compilation processes. Not surprisingly, such tools are called *disassemblers* and *decompilers*, and they do pretty much what their names indicate. A disassembler undoes the assembly process, so

we should expect assembly language as the output (and therefore machine language as input). Decompilers aim to produce output in a high-level language when given assembly or even machine language as input.

The promise of “source code recovery” will always be attractive in a competitive software market, and thus the development of usable decompilers remains an active research area in computer science. The following are just a few of the reasons that decompilation is difficult:

The compilation process is lossy. At the machine language level, there are no variable or function names, and variable type information can be determined only by how the data is used rather than explicit type declarations. When you observe 32 bits of data being transferred, you will need to do some investigative work to determine whether those 32 bits represent an integer, a 32-bit floating point value, or a 32-bit pointer.

Compilation is a many-to-many operation. This means that a source program can be translated to assembly language in many different ways, and machine language can be translated back to source in many different ways. As a result, compiling a file and immediately decompiling it commonly yields a source file that is vastly different from the original.

Decompilers are language and library dependent. Processing a binary produced by a Delphi compiler with a decompiler designed to generate C code can yield very strange results. Similarly, feeding a compiled Windows binary through a decompiler that has no knowledge of the Windows application programming interface (API) may not yield anything useful.

A nearly perfect disassembly capability is needed to accurately decompile a binary. Any errors or omissions in the disassembly phase will almost certainly propagate through to the decompiled code. The correctness of the disassembled code can be verified against the appropriate processor reference manuals; however, no canonical reference manuals are available to use to verify the correctness of a decompiler’s output.

The Why of Disassembly

The purpose of disassembly tools is often to facilitate understanding of programs when source code is unavailable. Common situations in which disassembly is used include the following:

- Analysis of malware
- Analysis of closed source software for vulnerabilities
- Analysis of closed source software for interoperability
- Analysis of compiler-generated code to validate compiler performance or correctness
- Display of program instructions while debugging

The subsequent sections explain each situation in more detail.

Malware Analysis

Unless you are dealing with script-based malware, malware authors seldom do you the favor of providing the source code for their creations. Lacking source code, you are faced with few options for discovering exactly how the malware behaves. The two main techniques for malware analysis are dynamic analysis and static analysis. *Dynamic analysis* involves allowing the malware to execute in a carefully controlled environment (sandbox) while recording every observable aspect of its behavior using any number of system instrumentation utilities. In contrast, *static analysis* attempts to understand a program's behavior by reading through the program code, which, in the case of malware, generally consists solely of a disassembly listing and possibly a decompiler listing.

Vulnerability Analysis

For the sake of simplification, let's break the entire security-auditing process into three steps: vulnerability discovery, vulnerability analysis, and exploit development. The same steps apply whether you have source code or not; however, the level of effort increases substantially when all you have is a binary. The first step in the process is to discover a potentially exploitable condition in a program. This is often accomplished using dynamic techniques such as fuzzing, but it can also be performed (usually with much more effort) via static analysis. Once a problem has been discovered, further analysis is often required to determine whether the problem is exploitable at all and, if so, under what conditions.

NOTE

Fuzzing is a vulnerability discovery technique that relies on generating a large number of unique inputs for programs in the hope that one of those inputs will cause the program to fail in a manner that can be detected, analyzed, and ultimately exploited.

Identifying variables that can be manipulated to the attacker's advantage is an important early step in vulnerability discovery. Disassembly listings provide the level of detail required to understand exactly how the compiler has chosen to allocate program variables. For example, it might be useful to know that a 70-byte character array declared by a programmer was rounded up to 80 bytes when allocated by the compiler. Disassembly listings also provide the only means to determine exactly how a compiler has chosen to order all of the variables declared globally or within functions. Understanding the spatial relationships among variables is often essential when attempting to develop exploits. Ultimately, by using a disassembler and a debugger together, an exploit may be developed.

Software Interoperability

When software is released in binary form only, it is very difficult for competitors to create software that can interoperate with it or to provide plugin replacements for that software. A common example is driver code released

for hardware that is supported on only one platform. When a vendor is slow to support or, worse yet, refuses to support the use of its hardware with alternative platforms, developing software drivers to support the hardware may require substantial reverse engineering effort. In these cases, static code analysis is almost the only remedy and often must go beyond the software driver to understand embedded firmware.

Compiler Validation

Since the purpose of a compiler (or assembler) is to generate machine language, good disassembly tools are often required to verify that the compiler is doing its job in accordance with any design specifications. Analysts may also be interested in locating additional opportunities for optimizing compiler output and, from a security standpoint, ascertaining whether the compiler itself has been compromised to the extent that it may be inserting backdoors into generated code.

Debugging Displays

Perhaps the single most common use of disassemblers is to generate listings within debuggers. Unfortunately, disassemblers embedded within debuggers tend to lack sophistication. They are generally incapable of batch disassembly and sometimes balk at disassembling when they cannot determine the boundaries of a function. This is one of the reasons it is best to use a debugger in conjunction with a high-quality disassembler to provide better situational awareness and context during debugging.

The How of Disassembly

Now that you are well versed in the purposes of disassembly, it is time to move on to how the process actually works. Consider a typical daunting task faced by a disassembler: *Take these 100KB, distinguish code from data, convert the code to assembly language for display to a user, and please don't miss anything along the way.* We could tack on any number of special requests, such as asking the disassembler to locate functions, recognize jump tables, and identify local variables, making the disassembler's job that much more difficult.

To accommodate all of our demands, any disassembler will need to pick and choose from a variety of algorithms as it navigates through the files we feed it. The quality of the generated disassembly listing will be directly related to the quality of the algorithms utilized and how well they have been implemented.

In this section, we discuss two of the fundamental algorithms in use today for disassembling machine code. As we present these algorithms, we also point out their shortcomings in order to prepare you for situations in which your disassembler appears to fail. By understanding a disassembler's limitations, you will be able to manually intervene to improve the overall quality of the disassembly output.

A Basic Disassembly Algorithm

For starters, let's develop a simple algorithm for accepting machine language as input and producing assembly language as output. In doing so, you will gain an understanding of the challenges, assumptions, and compromises that underlie an automated disassembly process:

1. The first step in the disassembly process is to identify a region of code to disassemble. This is not necessarily as straightforward as it may seem. Instructions are generally mixed with data, and it is important to distinguish between the two. In the most common case, disassembly of an executable file, the file will conform to a common format for executable files such as the *Portable Executable (PE)* format used on Windows and the *Executable and Linkable Format (ELF)* common on many Unix-based systems. These formats typically contain mechanisms (often in the form of hierarchical file headers) for locating the sections of the file that contain code and entry points into that code.
2. Given the address of an instruction, the next step is to read the value or values contained at that address (or file offset) and perform a table lookup to match the binary opcode value to its assembly language mnemonic. Depending on the complexity of the instruction set being disassembled, this may be a trivial process, or it may involve several additional operations such as understanding any prefixes that may modify the instruction's behavior and determining any operands the instruction requires. For instruction sets with variable-length instructions, such as the Intel x86 instruction set, additional instruction bytes may need to be retrieved in order to completely disassemble a single instruction.
3. Once an instruction has been fetched and any required operands decoded, its assembly language equivalent is formatted and output as part of the assembly listing. It may be possible to choose from more than one assembly language output syntax. For example, the two predominant formats for x86 assembly language are the Intel format and the AT&T format.
4. Following the output of an instruction, we need to advance to the next instruction and repeat the previous process until we have disassembled every instruction in the file.

Various algorithms and heuristics exist for determining where to begin a disassembly, how to choose the next instruction to be disassembled, how to distinguish code from data, and how to determine when the last instruction has been disassembled. The two predominant disassembly algorithms are linear sweep and recursive descent.

X86 ASSEMBLY SYNTAX: AT&T VS. INTEL

Two main syntaxes are used for x86 assembly source code: AT&T and Intel. Even though they are second-generation languages, the two vary greatly in syntax—from variable, constant, and register access, to segment and instruction size overrides, to indirection and offsets. The AT&T assembly syntax is distinguished by its use of the % symbol to prefix all register names, the use of \$ as a prefix for literal constants (also called *immediate operands*), and its operand ordering in which the source operand appears on the left and the destination operand appears on the right. Using AT&T syntax, the instruction to add 4 to the EAX register would be `add $0x4,%eax`. The GNU Assembler (*as*) and many other GNU tools, including *gcc* and *gdb*, utilize AT&T syntax by default.

Intel syntax differs from AT&T in that it requires no register or literal prefixes, and the operand ordering is reversed such that the source operand appears on the right and the destination appears on the left. The same `add` instruction using the Intel syntax would be `add eax,0x4`. Assemblers that utilize the Intel syntax include the Microsoft Assembler (MASM) and the Netwide Assembler (NASM).

Linear Sweep Disassembly

The *linear sweep* disassembly algorithm takes a very straightforward approach to locating instructions to disassemble: Where one instruction ends, another begins. As a result, the most difficult decisions faced are where to begin and when to stop. The usual solution is to assume that everything contained in sections of a program marked as code (typically specified by the program file's headers) represents machine language instructions. Disassembly begins with the first byte in a code section and moves, in a linear fashion, through the section, disassembling one instruction after another until reaching the end of the section. No effort is made to understand the program's control flow through recognition of nonlinear instructions such as branches.

During the disassembly process, a pointer can be maintained to mark the beginning of the instruction currently being disassembled. As part of the disassembly process, the length of each instruction is computed and used to determine the location of the next instruction to be disassembled. Instruction sets with fixed-length instructions (MIPS, for example) are somewhat easier to disassemble, as locating subsequent instructions is straightforward.

The main advantage of the linear sweep algorithm is that it provides complete coverage of a program's code sections. One of the primary disadvantages of the linear sweep method is that it fails to account for data that may be commingled with code. This is evident in Listing 1-1, which shows the output of a function disassembled with a linear sweep disassembler.

40123f:	55	push ebp
401240:	8b ec	mov ebp,esp
401242:	33 c0	xor eax,eax
401244:	8b 55 08	mov edx,DWORD PTR [ebp+8]
401247:	83 fa 0c	cmp edx,0xc

```

40124a: 0f 87 90 00 00 00      ja 0x4012e0
401250: ff 24 95 57 12 40 00 ❶ jmp DWORD PTR [edx*4+0x401257]
❷ 401257: e0 12                  ❸ loopne 0x40126b
401259: 40                      inc eax
40125a: 00 8b 12 40 00 90      add BYTE PTR [ebx-0x6ffffbfee],cl
401260: 12 40 00                adc al,BYTE PTR [eax]
401263: 95                      xchg ebp,eax
401264: 12 40 00                adc al,BYTE PTR [eax]
401267: 9a 12 40 00 a2 12 40    call 0x4012:0xa2004012
40126e: 00 aa 12 40 00 b2      add BYTE PTR [edx-0x4dffbfef],ch
401274: 12 40 00                adc al,BYTE PTR [eax]
401277: ba 12 40 00 c2         mov edx,0xc2004012
40127c: 12 40 00                adc al,BYTE PTR [eax]
40127f: ca 12 40                lret 0x4012
401282: 00 d2                  add dl,dl
401284: 12 40 00                adc al,BYTE PTR [eax]
401287: da 12                  ficom DWORD PTR [edx]
401289: 40                      inc eax
40128a: 00 8b 45 0c eb 50      add BYTE PTR [ebx+0x50eb0c45],cl
401290: 8b 45 10                mov eax,DWORD PTR [ebp+16]
401293: eb 4b                  jmp 0x4012e0

```

Listing 1-1: Linear sweep disassembly

This function contains a switch statement, and the compiler used in this case has elected to implement the switch by using a jump table to resolve case label targets. Furthermore, the compiler has elected to embed the jump table within the function itself. The `jmp` statement ❶ references an address table ❷. Unfortunately, the disassembler treats the address table as if it were a series of instructions and incorrectly generates the assembly language representation starting at address 401257 ❸ in the listing.

The x86 is a little-endian architecture, meaning that the least significant byte of a multibyte data value is stored first, at a lower memory address than each subsequent byte of that data item. Big-endian data is stored in reverse order, with the most significant byte of a data value being stored at a lower memory address than each subsequent byte. If we treat successive 4-byte groups in the jump table ❷ as little-endian values, we see that each represents a pointer to a nearby address that is in fact the destination for one of the various jumps (004012e0, 0040128b, 00401290, ...). Thus, the `loopne` instruction ❸ is not an instruction at all. Instead, it indicates a failure of the linear sweep algorithm to properly distinguish embedded data from code.

Linear sweep is used by the disassembly engines contained in the GNU debugger (`gdb`), Microsoft's WinDbg debugger, and the `objdump` utility.

Recursive Descent Disassembly

The *recursive descent* disassembly algorithm takes a different approach to locating instructions: It focuses on the concept of control flow, which determines whether an instruction should be disassembled based on whether it

is referenced by another instruction. To understand recursive descent, it is helpful to classify instructions according to how they affect the instruction pointer.

Sequential Flow Instructions

Sequential flow instructions pass execution to the instruction that immediately follows. Examples of sequential flow instructions include simple arithmetic instructions, such as `add`; register-to-memory transfer instructions, such as `mov`; and stack-manipulation operations, such as `push` and `pop`. For such instructions, disassembly proceeds as with linear sweep.

Conditional Branching Instructions

Conditional branching instructions, such as the x86 `jnz`, offer two possible execution paths. If the condition evaluates to true, the branch is taken, and the instruction pointer must be changed to reflect the target of the branch. However, if the condition is false, execution continues in a linear fashion, and a linear sweep methodology can be used to disassemble the next instruction. As it is generally not possible in a static context to determine the outcome of a conditional test, the recursive descent algorithm disassembles both paths, deferring disassembly of the branch target instruction by adding the address of the target instruction to a list of addresses to be disassembled at a later point.

Unconditional Branching Instructions

Unconditional branches do not follow the linear flow model and therefore are handled differently by the recursive descent algorithm. As with sequential flow instructions, execution can flow to only one instruction; however, that instruction does not need to immediately follow the branch instruction. In fact, as seen in Listing 1-1, there is no requirement at all for an instruction to immediately follow an unconditional branch. Therefore, there is no reason to immediately disassemble the bytes that follow an unconditional branch.

A recursive descent disassembler attempts to determine the target of the unconditional jump and continues disassembly at the target address. Unfortunately, some unconditional branches can cause problems for recursive descent disassemblers. When the target of a jump instruction depends on a runtime value, it may not be possible to determine the destination of the jump by using static analysis. The x86 instruction `jmp rax` demonstrates this problem. The `rax` register contains a value only when the program is actually running. Since the register contains no value during static analysis, we have no way to determine the target of the jump instruction, and, consequently, we have no way to determine where to continue the disassembly process.

Function Call Instructions

Function call instructions operate similarly to unconditional jump instructions (including the inability of the disassembler to determine the target of instructions such as `call rax`), with the additional expectation that execution usually returns to the instruction immediately following the call instruction after the

function completes. In this regard, they are similar to conditional branch instructions in that they generate two execution paths. The target address of the call instruction is added to a list for deferred disassembly, while the instruction immediately following the call is disassembled in a manner similar to linear sweep.

Recursive descent can fail if programs do not behave as expected when returning from called functions. For example, code in a function can deliberately manipulate the return address of that function so that upon completion, control returns to a location different from the one the disassembler expects. The following incorrect listing shows a simple example where the function `badfunc` simply adds 1 to the return address before returning to the caller:

```
badfunc proc near
48 FF 04 24  inc qword ptr [rsp] ; increments saved return addr
C3          retn
badfunc endp
; -----
label:
E8 F6 FF FF  call badfunc
❶ 05 48 89 45 F8 add eax, F8458948hu
```

As a result, control does not actually pass to the `add` instruction ❶ following the call to `badfunc`. A proper disassembly appears next:

```
48 FF 04 24  inc qword ptr [rsp]
C3          retn
badfunc endp
; -----
label:
E8 F6 FF FF  call badfunc
05          db 5          ; formerly the first byte of the add instruction
❶ 48 89 45 F8  mov [rbp-8], rax
```

This listing more clearly shows the flow of the program in which the function `badfunc` actually returns to the `mov` instruction ❶. It is important to understand that a linear sweep disassembler will also fail to properly disassemble this code, though for slightly different reasons.

Return Instructions

In some cases, the recursive descent algorithm runs out of paths to follow. A function *return instruction* (x86 `ret`, for example) offers no information about which instruction will be executed next. If the program were actually running, an address would be taken from the top of the runtime stack, and execution would resume at that address. Disassemblers do not have the benefit of access to a stack. Instead, disassembly abruptly comes to a halt. It is at this point that the recursive descent disassembler turns to the list of addresses it

has been setting aside for deferred disassembly. An address is removed from this list, and the disassembly process is continued from this address. This is the recursive process that lends the disassembly algorithm its name.

One of the principal advantages of the recursive descent algorithm is its superior ability to distinguish code from data. As a control flow-based algorithm, it is much less likely to incorrectly disassemble data values as code. The main disadvantage of recursive descent is the inability to follow indirect code paths, such as jumps or calls, which utilize tables of pointers to look up a target address. However, with the addition of some heuristics to identify pointers to code, recursive descent disassemblers can provide very complete code coverage and excellent recognition of code versus data. Listing 1-2 shows the output of Ghidra's recursive descent disassembler used on the same switch statement shown earlier in Listing 1-1.

```
0040123f  PUSH   EBP
00401240  MOV    EBP,ESP
00401242  XOR    EAX,EAX
00401244  MOV    EDX,dword ptr [EBP + param_1]
00401247  CMP    EDX,0xc
0040124a  JA     switchD_00401250::caseD_0
                switchD_00401250::switchD
00401250  JMP    dword ptr [EDX*0x4 + ->switchD_00401250::caseD_0] = 004012e0
                switchD_00401250::switchdataD_00401257
00401257  addr   switchD_00401250::caseD_0
0040125b  addr   switchD_00401250::caseD_1
0040125f  addr   switchD_00401250::caseD_2
00401263  addr   switchD_00401250::caseD_3
00401267  addr   switchD_00401250::caseD_4
0040126b  addr   switchD_00401250::caseD_5
0040126f  addr   switchD_00401250::caseD_6
00401273  addr   switchD_00401250::caseD_7
00401277  addr   switchD_00401250::caseD_8
0040127b  addr   switchD_00401250::caseD_9
0040127f  addr   switchD_00401250::caseD_a
00401283  addr   switchD_00401250::caseD_b
00401287  addr   switchD_00401250::caseD_c
                switchD_00401250::caseD_1
0040128b  MOV    EAX,dword ptr [EBP + param_2]
0040128e  JMP    switchD_00401250::caseD_0040128E
```

Listing 1-2: Recursive descent disassembly

Ghidra has recognized this section of the binary as a switch statement and has formatted it accordingly. An understanding of the recursive descent process will help us recognize situations in which Ghidra may produce less-than-optimal disassemblies and allow us to develop strategies to improve Ghidra's output.

Summary

Is a deep understanding of disassembly algorithms essential when using a disassembler? No. Is it useful? Yes! Battling your tools is the last thing you want to spend time doing while reverse engineering. One of the many advantages of Ghidra is that, as an interactive disassembler, it offers you plenty of opportunities to guide and override its decisions. The net result is quite often a disassembly that is both thorough and accurate.

In the next chapter, we review a variety of existing tools that prove useful in many reverse engineering situations. Although not directly related to Ghidra, many of these tools have influenced Ghidra and help explain the wide variety of informational displays available in the Ghidra user interface.

2

REVERSING AND DISASSEMBLY TOOLS



With some disassembly background under our belts, and before we begin our dive into the specifics of Ghidra, it will be helpful to understand some of the other tools used to reverse engineer binaries. Many of these tools predate Ghidra and continue to be useful for taking quick glimpses into files, as well as for double-checking the work that Ghidra does. As we will see, Ghidra rolls many of these tools' capabilities into its user interface to provide a single, integrated environment for reverse engineering.

Classification Tools

When first confronted with an unknown file, it is often useful to answer simple questions such as “What is this thing?” The first rule of thumb when attempting to answer that question is to *never* rely on a file extension to determine what a file actually is. That is also the second, third, and fourth rules of thumb. Once you have become an adherent of the “file extensions

are meaningless” line of thinking, you may wish to familiarize yourself with one or more of the following utilities.

file

The `file` command is a standard utility, included with most *nix-style operating systems, as well as the Windows Subsystem for Linux (WSL). The `file` command attempts to identify a file’s type by examining specific fields within the file. In some cases, `file` recognizes common strings such as `#!/bin/sh` (a shell script) and `<html>` (an HTML document).

THE WSL ENVIRONMENT

The Windows Subsystem for Linux (WSL) enables users to run a full GNU/Linux environment directly on Windows without needing a virtual machine or dual-boot setup. WSL supports running Linux distributions with access to a wide range of command line tools (`grep`, `awk`), compilers (`gcc`, `g++`), interpreters (Perl, Python, Ruby), networking utilities (`curl`, `ssh`), and many others. With WSL 2, users gain enhanced performance and full system call compatibility, enabling many Linux-native applications to be compiled and executed seamlessly on Windows systems with the bonus of a choice between a command line or GUI.

Files containing non-ASCII content present somewhat more of a challenge. In such cases, `file` attempts to determine whether the content appears to be structured according to a known file format. In many cases, it searches for specific tag values known to be unique to specific file types, often referred to as magic numbers.

A *magic number* is a unique sequence of bytes in a file that can be used to identify its format. This allows classification tools to easily identify a file’s type regardless of the file’s extension. In some cases, magic numbers are selected for humorous reasons. The MZ byte sequence in MS-DOS executable file headers represents the initials of Mark Zbikowski, one of the original architects of MS-DOS, while the hex value `0xcafebabe`, the well-known magic number associated with Java `.class` files, was chosen because the sequence of hex digits spells an easily remembered word.

The following hex listings show several examples of magic numbers used to identify some common file types:

Windows PE executable file

```
00000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....
00000010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 .....@.....
```

Jpeg image file

```
00000000 FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 60 .....JFIF....`
00000010 00 60 00 00 FF DB 00 43 00 0A 07 07 08 07 06 0A .`.....C.....
```

Java `.class` file

```
00000000 CA FE BA BE 00 00 00 32 00 98 0A 00 2E 00 3E 08 .....2.....>.
00000010 00 3F 09 00 40 00 41 08 00 42 0A 00 43 00 44 0A .?..@.A..B..C.D.
```

The `file` command has the ability to identify many file formats, including several types of ASCII text files and various executable and data file formats. The magic number checks performed by `file` are governed by rules contained in a *magic file*. The default magic file varies by operating system, but common locations include `/usr/share/file/magic`, `/usr/share/misc/magic`, and `/etc/magic`. Please refer to the documentation for `file` for more information concerning magic files.

In some cases, `file` can distinguish variations within a given file type. The following listing demonstrates `file`'s ability to identify not only several variations of ELF binaries but also information pertaining to how the binary was linked (statically or dynamically) and whether the binary was stripped.

```
ghidrabook# file ch2_ex_*
ch2_ex_x64:      ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                 dynamically linked, interpreter /lib64/ld, for GNU/Linux
                 3.2.0, not stripped
ch2_ex_x64_dbg:  ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                 dynamically linked, interpreter /lib64/ld, for GNU/Linux
                 3.2.0, with debug_info, not stripped
ch2_ex_x64_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
                 statically linked, for GNU/Linux 3.2.0, not stripped
ch2_ex_x64_strip: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                 dynamically linked, interpreter /lib64/ld, for GNU/Linux
                 3.2.0, stripped
ch2_ex_x86:      ELF 32-bit LSB shared object, Intel 80386, version 1
                 (SYSV), dynamically linked, interpreter /lib/ld-, for
                 GNU/Linux 3.2.0, not stripped
ch2_ex_x86_dbg:  ELF 32-bit LSB shared object, Intel 80386, version 1
                 (SYSV), dynamically linked, interpreter /lib/ld-, for
                 GNU/Linux 3.2.0, with debug_info, not stripped
ch2_ex_x86_static: ELF 32-bit LSB executable, Intel 80386, version 1
                 (GNU/Linux), statically linked, for GNU/Linux 3.2.0,
                 not stripped
ch2_ex_x86_strip: ELF 32-bit LSB shared object, Intel 80386, version 1
                 (SYSV), dynamically linked, interpreter /lib/ld-, for
                 GNU/Linux 3.2.0, stripped
ch2_ex_Win32:    PE32 executable (console) Intel 80386, for MS Windows
ch2_ex_x64:      PE32+ executable (console) x86-64, for MS Windows
```

The `file` utility and similar utilities are not foolproof. It is quite possible for a file to be misidentified simply because it happens to bear the identifying marks of a particular file format. You can see this for yourself by using a hex editor to modify the first 4 bytes of any file to the Java magic number sequence: `CA FE BA BE`. The `file` utility will incorrectly identify the newly modified file as *compiled java class data*. Similarly, a text file that contains only

the two characters MZ will be identified as *MS-DOS executable*. A good approach to take in any reverse engineering effort is to never fully trust the output of any tool until you have correlated that output with several tools and manual analysis.

STRIPPING BINARY EXECUTABLE FILES

Stripping a binary is the process of removing symbols from the binary file. Binary object files contain symbols as a result of the compilation process. Some of these symbols are utilized during the linking process to resolve references between files when creating the final executable file or library. In other cases, symbols may be present to provide additional information for use with debuggers. Following the linking process, many of the symbols are no longer required. Options passed to the linker can cause the linker to remove the unnecessary symbols at build time. Alternatively, a utility named `strip` may be used to remove symbols from existing binary files. Although a stripped binary will be smaller than its unstripped counterpart, the behavior of the stripped binary will remain unchanged.

PE-bear

PE-bear, a tool available for Windows and Linux, is useful for quickly looking at 32-bit and 64-bit Windows executable files. You can find it at <https://github.com/hasherezade/pe-bear>. It provides a graphical display of a PE file's structure, including listing and graphical views of each file section, hex dumps, and disassembly listings of section content.

By using its included signatures, PE-bear can also identify whether an executable was processed by any known obfuscation utilities. Finally, PE-bear includes a PE editor, allowing you to easily modify header values and section content. You'll often need to modify PE headers when attempting to reconstruct a valid PE from an obfuscated version of that file.

Detect-It-Easy

Detect-It-Easy (DiE) is a tool designed primarily for feature extraction and basic identification of executable files, and it's available at <https://github.com/horsicq/Detect-It-Easy>. DiE uses signatures to identify common executable and container file formats, such as PE, ELF, ZIP, and APK. Further, DiE attempts to identify the compiler used to build the executable and any obfuscation utilities that may have been used on the file.

Features extracted by DiE include lists of strings, hashes of individual sections, and entropy measures across all portions of the binary. Many of the tool's additional capabilities overlap with those of PE-bear, including the ability to summarize file headers and perform basic disassembly.

BINARY FILE OBFUSCATION

Obfuscation is any attempt to obscure the true meaning of something. When applied to executable files, obfuscation is any attempt to hide the true behavior of a program. Programmers may employ obfuscation for a number of reasons. Commonly cited examples include protecting proprietary algorithms and obscuring malicious intent. Almost all forms of malware utilize obfuscation in an effort to hinder analysis. Tools to help program authors generate obfuscated programs are widely available. We further discuss obfuscation tools and techniques and their associated impacts on the reverse engineering process in Chapter 21.

Summary Tools

Once we've performed the initial classification of a file, we will need more sophisticated tools to extract detailed information. These highly specialized tools are typically designed to work with specific file formats and often won't function correctly outside of that narrow scope. Each tool understands the structure of a particular file type and is capable of parsing it to extract targeted, format-specific information.

nm

When source files are compiled to object files, compilers must embed information regarding the location of any global (external) symbols into the object file so that the linker can resolve references to those symbols when it combines object files to create an executable. Unless instructed to strip symbols from the final executable, the linker generally carries symbols from the object files over into the resulting executable. According to the man page, the `nm` utility lists symbols from object files.

When used to examine an object file (a compiled file that typically has a `.o` extension and is not yet linked into a final executable), `nm`'s default output yields the names of any functions and global variables declared in the file. Here is sample output of the `nm` utility:

```
ghidrabook# gcc -c ch2_nm_example.c
ghidrabook# nm ch2_nm_example.o
                 U exit
                 U fwrite
000000000000002e t get_max
                 U _GLOBAL_OFFSET_TABLE_
                 U __isoc99_scanf
00000000000000a6 T main
0000000000000000 D my_initialized_global
0000000000000004 C my_uninitialized_global
                 U printf
                 U puts
```

```

U rand
U srand
U __stack_chk_fail
U stderr
U time
0000000000000000 T usage
ghidrabook#

```

We see that `nm` lists each symbol, along with information about the symbol. The letter codes indicate the type of symbol being listed. In this example, we see the following letter codes:

- U** An undefined symbol (usually an external symbol reference).
- T** A symbol defined in the text section (usually a function name).
- t** A local symbol defined in the text section. In a C program, this usually equates to a static function.
- D** An initialized data value.
- C** An uninitialized data value.

NOTE

Uppercase letter codes are used for global symbols, whereas lowercase letter codes are used for local symbols. More information, including a full explanation of the letter codes, can be found in the man page for `nm`.

The utility displays somewhat more information when used to list symbols from an executable file. During linking, symbols are resolved to virtual addresses when possible, which makes more information available when `nm` is run. Here is truncated sample output from `nm` used on an executable:

```

ghidrabook# gcc -o ch2_nm_example ch2_nm_example.c
ghidrabook# nm ch2_nm_example
...
U fwrite@@GLIBC_2.2.5
0000000000000938 t get_max
0000000000201f78 d _GLOBAL_OFFSET_TABLE_
w __gmon_start__
0000000000000c5c r __GNU_EH_FRAME_HDR
0000000000000730 T _init
0000000000201d80 t __init_array_end
0000000000201d78 t __init_array_start
0000000000000b60 R _IO_stdin_used
U __isoc99_scanf@@GLIBC_2.7
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
0000000000000b50 T __libc_csu_fini
0000000000000ae0 T __libc_csu_init
U __libc_start_main@@GLIBC_2.2.5
00000000000009b0 T main
0000000000202010 D my_initialized_global
000000000020202c B my_uninitialized_global

```

```

                                U printf@@GLIBC_2.2.5
                                U puts@@GLIBC_2.2.5
                                U rand@@GLIBC_2.2.5
0000000000000870 t register_tm_clones
                                U srand@@GLIBC_2.2.5
                                U __stack_chk_fail@@GLIBC_2.4
0000000000000800 T _start
0000000000202020 B stderr@@GLIBC_2.2.5
                                U time@@GLIBC_2.2.5
0000000000202018 D __TMC_END__
000000000000090a T usage
ghidrabook#

```

At this point, some of the symbols (`main`, for example) have been assigned virtual addresses, new ones (such as `__libc_csu_init`) have been introduced as a result of the linking process, some (such as `my_uninitialized_global`) have had their symbol type changed, and others remain undefined as they continue to reference external symbols. In this case, the binary we are examining is dynamically linked, and the undefined symbols are defined in the shared C library.

ldd

When creating an executable, the linker must resolve the location of any library functions that executable references. The linker has two methods for resolving calls to library functions: *static linking* and *dynamic linking*. Command line arguments provided to the linker determine which of the two methods is used. An executable may be statically linked, dynamically linked, or both.

When static linking is requested, the linker combines an application's object files with a copy of the required library to create an executable file. At runtime, there is no need to locate the library code because it is already contained within the executable. Static linking has a few advantages: It results in slightly faster function calls and makes it easier to distribute binaries because no assumptions need to be made regarding the availability of library code on users' systems.

Disadvantages of static linking include larger resulting executables and greater difficulty upgrading programs when library components change. Programs are more difficult to update because they must be relinked every time a library is changed. From a reverse engineering perspective, static linking also complicates matters somewhat. If we are faced with the task of analyzing a statically linked binary, there is no easy way to answer the questions "Which libraries are linked into this binary?" and "Which of these functions is a library function?" Chapter 13 discusses the challenges encountered while reverse engineering statically linked code.

Dynamic linking differs from static linking in that the linker does not need to make a copy of any required libraries. Instead, the linker simply inserts references to any required libraries (often `.so` or `.dll` files) into the final

executable, usually resulting in much smaller executable files. Upgrading library code is much easier in these cases; the vendor maintains a single copy of a library, and many binaries reference that copy, replacing the outdated library with a new version.

One of the disadvantages of using dynamic linking is that it requires a more complicated loading process. All of the necessary libraries must be located and loaded into memory, as opposed to loading one statically linked file that contains all of the library code. Another disadvantage of dynamic linking is that vendors must distribute not only their own executable file but also all library files upon which that executable depends. Attempting to execute a program on a system that does not contain all the required library files will result in an error.

The following output demonstrates the creation of dynamically and statically linked versions of a program, the size of the resulting binaries, and the manner in which file identifies those binaries:

```
ghidrabook# gcc -o ch2_example_dynamic ch2_example.c
ghidrabook# gcc -o ch2_example_static ch2_example.c -static
ghidrabook# ls -l ch2_example_*
-rwxrwxr-x 1 ghydrabook ghydrabook 12944 Nov  7 10:07 ch2_example_dynamic
-rwxrwxr-x 1 ghydrabook ghydrabook 963504 Nov  7 10:07 ch2_example_static
ghidrabook# file ch2_example_*
ch2_example_dynamic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld, for GNU/Linux 3.2.0,
BuildID[sha1]=e56ed40012accb3734bde7f8bca3cc2c368455c3, not stripped
ch2_example_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=430996c6db103e4fe76aea7d578e636712b2b4b0, not stripped
ghidrabook#
```

For dynamic linking to function properly, dynamically linked binaries must indicate which libraries they depend on, along with the specific resources required from each of those libraries. As a result, unlike statically linked binaries, it is quite simple to determine the libraries on which a dynamically linked binary depends. The `ldd` (*list dynamic dependencies*) utility is a tool used to list the dynamic libraries an executable requires. In the following example, we run `ldd` to determine the libraries on which the Apache web server depends:

```
ghidrabook# ldd /usr/sbin/apache2
linux-vdso.so.1 => (0x00007ffffc1c8d000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fbeb7410000)
libaprutil-1.so.0 => /usr/lib/x86_64-linux-gnu/libaprutil-1.so.0 (0x00007fbeb71e0000)
libapr-1.so.0 => /usr/lib/x86_64-linux-gnu/libapr-1.so.0 (0x00007fbeb6fa0000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fbeb6d70000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbeb69a0000)
libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1 (0x00007fbeb6760000)
libexpat.so.1 => /lib/x86_64-linux-gnu/libexpat.so.1 (0x00007fbeb6520000)
libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007fbeb6310000)
```



```
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fbeb6100000)
/lib64/ld-linux-x86-64.so.2 (0x00007fbeb7a00000)
ghidrabook#
```

The `ldd` utility is available on Linux and BSD systems. On macOS systems, similar functionality is available using the `otool` utility with the `-L` option: `otool -L filename`. On Windows systems, the `dumpbin` utility, part of the Visual Studio tool suite, can be used to list dependent libraries: `dumpbin /dependents filename`.

BEWARE YOUR TOOLS!

While `ldd` may appear to be a simple tool, the `ldd` man page states that “you should never employ `ldd` on an untrusted executable, since this may result in the execution of arbitrary code.” While this is unlikely in most cases, it provides a reminder that running even apparently simple software reverse engineering (SRE) tools may have unintended consequences when examining untrusted input files. While we hope it is obvious that executing untrusted binaries is unlikely to be safe, it is wise to take precautions even when statically analyzing untrusted binaries and to assume that the computer on which you perform SRE tasks, along with any data on it or other hosts connected to it, may be compromised as a result of SRE activities.

objdump

Whereas `ldd` is fairly specialized, `objdump` is extremely versatile. Its purpose is to display information from object files. This is a fairly broad goal, and to accomplish it, `objdump` responds to more than 50 command line options tailored to extract various pieces of information. You can use this tool to display the following data (and much more) related to object files:

Section headers Summary information for each of the sections in the program file.

Private headers Program memory layout information and other information required by the runtime loader, including a list of required libraries, such as that produced by `ldd`.

Debugging information Any debugging information embedded in the program file.

Symbol information Symbol table information, dumped in a manner similar to the `nm` utility.

Disassembly listing The `objdump` tool performs a linear sweep disassembly of sections of the file marked as code. When disassembling x86 code, `objdump` can generate either AT&T or Intel syntax, and the disassembly can be captured as a text file. This type of text file is called a disassembly *dead listing*, and while these files can certainly be used for reverse engineering, they are difficult to navigate effectively and even more difficult to modify in a consistent and error-free manner.

The `objdump` tool is available as part of the GNU binutils tool suite and can be found on Linux, FreeBSD, and Windows (via WSL). Note that `objdump` relies on the *Binary File Descriptor library* (*libbfd*), a component of binutils, to access object files and thus is only capable of parsing file formats supported by *libbfd* (including ELF and PE, among others). For ELF-specific parsing, you could also use a utility named `readelf`, which offers most of the same capabilities as `objdump` but does not rely upon *libbfd*.

otool

The `otool` utility is most easily described as an `objdump`-like option for macOS, and it is useful for parsing information about macOS Mach-O binaries. The following listing demonstrates how `otool` displays the dynamic library dependencies for a Mach-O binary, thus performing a function similar to `ldd`:

```
ghidrabook# file ch2_ex_macOS_arm
ch2_ex_macOS_arm: Mach-O 64-bit executable arm64
ghidrabook# otool -L ch2_ex_macOS_arm
ch2_ex_macOS_arm:
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1351.0.0)
```

The `otool` utility can be used to display information related to a file's headers and symbol tables and to perform disassembly of the file's code section. You can find more information about the tool's capabilities on the associated man page.

dumpbin

Like `otool` and `objdump`, the `dumpbin` utility is capable of displaying a wide range of information related to Windows PE files. It's included in Microsoft's Visual Studio suite of tools. The following listing shows how `dumpbin` displays the dynamic dependencies of the Windows Notepad program in a manner similar to `ldd`:

```
$ dumpbin /dependents C:\Windows\System32\notepad.exe
Microsoft (R) COFF/PE Dumper
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Dump of file c:\Windows\System32\notepad.exe
```

```
File Type: EXECUTABLE IMAGE
```

```
Image has the following dependencies:
```

```
GDI32.dll
USER32.dll
api-ms-win-crt-string-l1-1-0.dll
api-ms-win-crt-runtime-l1-1-0.dll
...
```

Image has the following delay load dependencies:

```
ADVAPI32.dll
COMDLG32.dll
PROPSYS.dll
SHELL32.dll
WINSPOOL.DRV
urlmon.dll
```

Summary

```
3000 .data
1000 .didat
...
```

Additional dumpbin options offer the ability to extract information from various sections of a PE binary, including symbols, imported function names, exported function names, and disassembled code. You can find more information related to the use of dumpbin on the Microsoft website.

c++filt

Languages that allow function overloading must have a mechanism for distinguishing between the many overloaded versions of a function since each version has the same name. The following C++ example shows the prototypes for several overloaded versions of a function named `demo`:

```
void demo(void);
void demo(int x);
void demo(double x);
void demo(int x, double y);
void demo(double x, int y);
void demo(char* str);
```

As a general rule, two functions cannot have the same name in an object file. So, compilers derive unique names for overloaded functions by incorporating information that describes the type sequence of the function arguments. The process of deriving unique names for functions with identical names is called *name mangling*. If we use `nm` to dump the symbols from the compiled version of the preceding C++ code, we might see something like the following (filtered to focus on versions of `demo`):

```
ghidrabook# g++ -o ch2_cpp_example ch2_cpp_example.cc
ghidrabook# nm ch2_cpp_example | grep demo
000000000000060b T _Z4demod
0000000000000626 T _Z4demodi
0000000000000601 T _Z4demoi
0000000000000617 T _Z4demoid
```

```
0000000000000635 T _Z4demoPc
00000000000005fa T _Z4demov
```

The C++ standard does not define a standard name mangling scheme, leaving compiler designers to develop their own. To decipher the mangled variants of `demo` shown here, we need a tool that understands the name mangling scheme of our compiler (g++, in this case).

This is precisely the purpose of `c++filt`. This utility treats each input word as if it were a mangled name and then attempts to determine the compiler used to generate that name. If the name appears to be a valid mangled name, it outputs the demangled version of the name. When `c++filt` does not recognize a word as a mangled name, it simply outputs the word with no changes.

If we pass the results of `nm` from the preceding example through `c++filt`, the tool recovers the demangled function names, as shown here:

```
ghidrabook# nm ch2_cpp_example | grep demo | c++filt
000000000000060b T demo(double)
0000000000000626 T demo(double, int)
0000000000000601 T demo(int)
0000000000000617 T demo(int, double)
0000000000000635 T demo(char*)
00000000000005fa T demo()
```

It is important to note that the mangled names contain additional information about functions that `nm` does not normally provide. This information can be extremely helpful in reverse engineering situations. In more complex cases, it may include data regarding class names or function-calling conventions.

Deep Inspection Tools

So far, we have discussed tools that perform a cursory analysis of files based on minimal knowledge of those files' internal structures. We have also seen tools capable of extracting specific pieces of data from files based on very detailed knowledge of a file's structure. In this section, we discuss tools designed to extract specific types of information independently of the type of file being analyzed.

strings

It is occasionally useful to ask more generic questions regarding file content that do not necessarily require any specific knowledge of a file's structure. One such question is "Does this file contain any embedded strings?" Of course, we must first answer the question "What exactly constitutes a string?"

Let's loosely define a *string* as a consecutive sequence of printable characters. This definition is often augmented to specify a minimum length and a specific character set. Thus, we could specify a search for all sequences of at least four consecutive ASCII printable characters and print the results to the console. Searches for such strings generally do not depend on the file's structure. You can search for strings in an ELF binary just as easily as you can search for strings in a Microsoft Word document.

The strings utility is designed specifically to extract string content from files, often without regard for the format of those files. Using strings with its default settings (which look for 7-bit ASCII sequences of at least four characters) might yield something like the following:

```
ghidrabook# strings ch2_example
/lib64/ld-linux-x86-64.so.2
libc.so.6
exit
srand
__isoc99_scanf
puts
time
__stack_chk_fail
printf
stderr
fwrite
__libc_start_main
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
usage: ch4_example [max]
A simple guessing game!
Please guess a number between 1 and %d.
Invalid input, quitting!
Congratulations, you got it in %d attempt(s)!
Sorry too low, please try again
Sorry too high, please try again
GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
...
```

Unfortunately, while some of these strings look like they might be program outputs, other strings appear to be function names and library names. We should be careful not to jump to any conclusions; analysts often fall into the trap of attempting to deduce the behavior of a program based on the output of strings, but the presence of a string within a binary in no way indicates that the binary ever uses the string in any manner.

Here are some final notes on the use of strings:

- By default, strings gives no indication of where, within a file, a string is located. You can use the `-t x` command line argument to have strings print file offset information for each string found.
- In addition, strings searches for ASCII printable sequences. The strings man page describes the options available for searching for strings made up of other character encodings such as Unicode.

WHY DID STRINGS CHANGE?

Historically, when strings was used on executable files, it would, by default, search only for character sequences in the loadable, initialized data sections of the binary file. This required strings to parse the binary file to find those sections, using libraries such as libbfd. When it was used for parsing untrusted binaries, vulnerabilities in libraries could potentially result in arbitrary code execution. As a result, the default behavior for strings was changed to examine the entire binary file without parsing for loadable initialized data sections (synonymous with the use of the `-a` flag). The historical behavior can be invoked using the `-d` flag.

Disassemblers

Earlier, we mentioned that dedicated tools can generate dead listing-style disassemblies of binary object files. You can disassemble PE, ELF, and Mach-O binaries using `dumpbin`, `objdump`, and `otool`, respectively. However, none of those tools can deal with arbitrary blocks of binary data. You will occasionally be confronted with a binary file that does not conform to a widely used file format, in which case you will need tools capable of beginning the disassembly process at user-specified offsets.

Two examples of such *stream disassemblers* for the x86 instruction set are `ndisasm` and `diStorm3`. The utility `ndisasm` is included with Netwide Assembler (NASM). The following example illustrates the use of `ndisasm` to disassemble a piece of shellcode generated using the Metasploit framework:

```
ghidrabook# msfvenom -p linux/x64/shell_find_port -f raw > findport
ghidrabook# ndisasm -b 64 findport
00000000 4831FF          xor rdi,rdi
00000003 4831DB          xor rbx,rbx
00000006 B314           mov bl,0x14
00000008 4829DC          sub rsp,rbx
0000000B 488D1424        lea rdx,[rsp]
0000000F 488D742404      lea rsi,[rsp+0x4]
00000014 6A34           push byte +0x34
00000016 58             pop rax
00000017 0F05           syscall
00000019 48FFC7         inc rdi
```

0000001C	66817E024A67	cmp word [rsi+0x2],0x674a
00000022	75F0	jnz 0x14
00000024	48FFCF	dec rdi
00000027	6A02	push byte +0x2
00000029	5E	pop rsi
0000002A	6A21	push byte +0x21
0000002C	58	pop rax
0000002D	0F05	syscall
0000002F	48FFCE	dec rsi
00000032	79F6	jns 0x2a
00000034	4889F3	mov rbx,rsi
00000037	BB412F7368	mov ebx,0x68732f41
0000003C	B82F62696E	mov eax,0x6e69622f
00000041	48C1EB08	shr rbx,byte 0x8
00000045	48C1E320	shl rbx,byte 0x20
00000049	4809D8	or rax,rbx
0000004C	50	push rax
0000004D	4889E7	mov rdi,rsi
00000050	4831F6	xor rsi,rsi
00000053	4889F2	mov rdx,rsi
00000056	6A3B	push byte +0x3b
00000058	58	pop rax
00000059	0F05	syscall

ghidrabook#

The flexibility of stream disassembly is useful in many situations. One scenario involves the analysis of computer network attacks in which network packets may contain shellcode. Stream disassemblers can disassemble the portions of the packet that contain shellcode to analyze the payload's behavior. Another situation involves the analysis of ROM images for which no layout reference can be located. Portions of the ROM will contain data, while other portions will contain code. Stream disassemblers can be used to disassemble just those portions of the image thought to be code.

Summary

The tools discussed in this chapter are not necessarily the best of their kind. However, they do represent tools commonly available for anyone who wishes to reverse engineer binary files. More importantly, they represent the types of tools that motivated much of the development of Ghidra. In future chapters, we occasionally highlight stand-alone tools that provide functionality similar to that integrated into Ghidra. An awareness of these tools will greatly enhance your understanding of the Ghidra user interface and the many informational displays that Ghidra offers.

3

MEET GHIDRA



Ghidra is a freely available open source tool suite developed for software reverse engineering (SRE). Originally a National Security Agency (NSA) project, it is now supported by a growing community of Ghidra enthusiasts.

The platform-independent Ghidra environment includes an interactive disassembler and decompiler, as well as a plethora of related tools that work together to help you analyze code. It supports a wide variety of instruction set architectures and binary formats and can be run in both stand-alone and collaborative SRE configurations.

Perhaps the best feature of Ghidra is that it allows you to customize your work environment and develop your own plugins and scripts to enhance your SRE workflow and to share your innovations with the Ghidra community at large.

Licenses

Ghidra is distributed free of charge and is licensed under the Apache License, Version 2.0. This license provides a lot of freedom to individuals to use Ghidra, but it does have some associated restrictions. All individuals

downloading, using, or editing Ghidra are encouraged to read the Ghidra User Agreement (*docs/UserAgreement.html*) and the license files in the *GPL* and *licenses* directories to ensure that they comply with all licensing agreements, as third-party components within Ghidra have their own licenses. In case you ever forget anything in this paragraph, Ghidra helpfully displays the licensing information every time you start Ghidra or select About Ghidra from the Help menu.

Versions

Ghidra is available for Windows, Linux, and macOS. While Ghidra is highly configurable, most new users will likely download a Ghidra release and choose to start with the most current version of Ghidra Core, which includes traditional reverse engineering functionality. This book focuses on the Ghidra Core functionality for nonshared projects; however, we'll also discuss shared projects and headless Ghidra, as well as the Developer, Function ID, BSim, Debugger, and Experimental configurations.

Support Resources

Working with a new software suite can be daunting, especially when the intention is to approach a challenging real-world problem using reverse engineering. As a Ghidra user (or potential developer), you may wonder where to turn for help when you have Ghidra-related questions. If we do our job well enough, this book will suffice in many situations, but when you need additional guidance, here are some additional resources you can turn to:

Official help documentation Ghidra contains a detailed help system that you can activate through the menu or by pressing F1. The help system provides a hierarchical menu as well as search functionality.

README files In some cases, the Ghidra Help menu will refer you to additional content on a particular topic. Various README files supplement specific plugins, extend topics in the Help menu (such as *support/analyzeHeadlessREADME.html*), assist with various installations (such as *GettingStarted.html*), and aid your evolution as a developer (such as *Extensions/Eclipse/GhidraDev/README.html*).

The Ghidra site The Ghidra project home page (<https://github.com/NationalSecurityAgency/ghidra>) provides options for potential users, current users, developers, and contributors to further their knowledge about Ghidra. When Ghidra was first released, the original project page (<https://www.ghidra-sre.org>) introduced some basic information about Ghidra, including the option to download the current version, and provided a link to the Ghidra GitHub repository. As the repository has become increasingly populated with information, the original URL now takes you directly to the Ghidra GitHub repository.

The Ghidra docs directory Your Ghidra installation includes a directory containing helpful Ghidra-related documentation, including a printable guide to menus and hotkeys (*docs/CheatSheet.html*) that can greatly ease your introduction to Ghidra, and much more. Tutorials covering beginner, intermediate, and advanced features of Ghidra can be found in *docs/GhidraClass*.

Downloading Ghidra

As the project grew to involve more contributors, new features, multiple releases, and an expanding community, it became clear that users needed a more flexible and transparent way to manage Ghidra's evolution. The Ghidra GitHub repository now serves as the gateway for the growing Ghidra community and is where all releases, updates, issues, and documentation are maintained.

You can obtain your free copy of Ghidra from the Ghidra GitHub repository by navigating to <https://github.com/NationalSecurityAgency/ghidra>. If you are already familiar with GitHub, you should be able to navigate through a variety of interesting paths to locate the version of Ghidra that you would like to download. Perhaps the easiest is to locate the “Releases” section in the right-hand column of Ghidra's GitHub landing page and click the Latest button.

For those who are new to GitHub, it can feel a bit technical or overwhelming at first. We encourage you to scroll down below the list of folders and files on the landing page, where you will find the latest information about Ghidra including installation instructions. After ensuring you have the required version of Java installed, you can click “Download a release file” and scroll through the list of available Ghidra releases. For most users, the most recent release is the appropriate version. A quick click on the ZIP file that starts with *ghidra* (under *Assets*) sends a copy of Ghidra to your *Downloads* directory.

The following options are for those of you who want something different from the traditional starter pack:

- If you want to install a different release or multiple releases, simply scroll and select other versions as desired. You can obtain additional information about each release by exploring the links under the release name. While the functionality will vary, the basics of Ghidra should remain the same.
- If you wish to support collaborative work using Ghidra, hold on until Chapter 11 to find out how to make that important change to your installation (or feel free to jump ahead and give it a try using the information in the *server* directory).
- The truly brave at heart may wish to build Ghidra from source. The Ghidra source code for each release is available on GitHub under *Assets*, in the same location where you found the Ghidra release ZIP file.

We encourage you to continue visiting and exploring the GitHub Ghidra project page as your experience with Ghidra increases. This is where the project lives and evolves. Through this site, you can:

- Check out the entire Ghidra source code repository
- Access the latest updates and new releases
- Browse documentation and usage guides
- Review or open issues (such as bug reports, feature requests, and questions)
- Engage with the Ghidra community

Now that you have your copy of Ghidra, let's move ahead with the traditional installation process.

Installing Ghidra

The name of the file you download tells you a lot about the file contents. For the original Ghidra release, the ZIP filename was *ghidra_9.0_PUBLIC_20190228.zip*. We can easily break down the naming convention. First, *ghidra* is the product and *9.0* is the version number. Next, *PUBLIC* is the type of release. (There are other types of releases, such as *BETA_DEV* releases.) Finally, we have the release date, followed by the *.zip* file extension.

This ZIP file is actually a collection of more than 7,500 files that make up the Ghidra framework. When you are happy with the location you saved the file to, unzipping it (for example, by right-clicking and selecting Extract All in Windows) will provide access to the Ghidra hierarchical directory. Note that Ghidra needs to compile some of its internal data files, so a Ghidra user will typically need write access to all Ghidra program subdirectories.

Exploring the Directory Layout

Familiarity with the contents of your Ghidra installation is by no means a requirement for starting to use Ghidra, but since your attention is on the extracted download, let's take an initial look at the basic layout. Understanding the Ghidra directory structure will become more important as you progress to using the more advanced features covered in later chapters. For now, it is sufficient to be aware of the top-level directory contents shown in Figure 3-1.

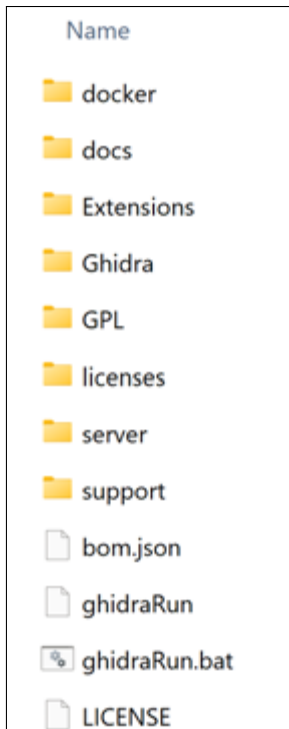


Figure 3-1: The Ghidra directory layout

Here is a brief description of each of the subdirectories:

docker Provides the files required to build and run Ghidra as a Docker container, which can be particularly useful for running Ghidra in the various headless and server modes (although it can also be run in a GUI mode if necessary). We discuss these use cases in more depth in Chapters 11, 16, and 23.

docs Contains general support documentation about Ghidra and how to use it. Included in this directory are two subdirectories that bear mentioning. First, the *GhidraClass* subdirectory provides educational content to help you learn about Ghidra. Second, the *languages* subdirectory describes Ghidra's processor specification language, SLEIGH. We discuss SLEIGH extensively in Chapter 18.

Extensions Contains useful prebuilt extensions, important content, and information for writing Ghidra extensions. We cover this directory more thoroughly in Chapters 15, 17, and 18.

Ghidra Contains the code for Ghidra. You will learn more about the resources and content in this directory as we begin customizing Ghidra in Chapter 12 and building new capabilities in Chapters 13 through 18.

GPL Some of the components that make up the Ghidra framework were not developed by the Ghidra team but consist of other code distributed under the GNU General Public License (GPL). The *GPL* directory contains files associated with this content, including licensing information.

licenses Contains files that outline the appropriate and legal usage of Ghidra's various third-party components.

server Supports the installation of the Ghidra Server, which facilitates collaborative SRE. We discuss this directory in depth in Chapter 11.

support Serves as a catchall for a variety of specialized Ghidra capabilities and functionalities. As a bonus, this is also where you will find the Ghidra icon (*ghidra.ico*), which can be useful for customizing your work environment to, for example, create a shortcut to your preferred Ghidra startup script. We discuss other files in this directory throughout this text as we introduce various Ghidra capabilities.

WHO'S THE BOM?

Recent releases of Ghidra have a *bom.json* file in the root directory in addition to the anticipated files for starting Ghidra and providing licensing information. A *bom.json* file is basically a shopping list for your software. Short for “bill of materials,” it lays out everything that went into your project: libraries, dependencies, versions, licenses, and more. Think of it as the ingredients label on your favorite snack, but for code. It's especially useful when you need to keep track of what you're using (and where it came from), whether for security audits, license checks, or so you can remember why that obscure library is in your build.

After the log4j vulnerability reminded the entire tech world (including Ghidra) that we might not always know what's lurking in our software cabinets, *bom.json* files got a big boost in visibility. The log4j vulnerability, hidden deep inside a widely used logging library, was quietly embedded in thousands of applications, often without the developers realizing it. It exposed just how tricky it can be to trace what your software is really made of. Since the vulnerability's discovery in 2021, having this information readily available in a *bom.json* file has become less of a “nice-to-have” and more like wearing your seatbelt. It's a smart, practical move that helps you stay safe when the unexpected happens.

Starting Ghidra

In addition to the subdirectories, there are several important files in the root directory: yet another license file (*LICENSE*), and more importantly, two of the scripts that actually launch Ghidra.

Double-clicking *ghidraRun.bat* (or running the equivalent *ghidraRun* script from the command line on Linux or macOS) starts Ghidra in its standard mode. You can find other modes of starting Ghidra (including CPython 3 and headless mode) in the *support* directory. We will look at some of these options in detail in future chapters.

When you first start Ghidra, you may be required to agree to a license and provide the path to your Java installation. If you do not have Java installed, see the Installation Guide in the *docs* subdirectory, which provides supporting documentation in the Java Notes section to let you know the Java Development Kit (JDK) version requirements and has additional requirements to enable Python 3 support. Python versions are tightly coupled to Ghidra functionality, so the Python 3 version requirement can vary depending on the Ghidra tools you plan to use. You will find these requirements in the Ghidra Installation Guide, and we will discuss them as we present individual tools.

Summary

Once you have successfully opened Ghidra, you are ready to use it to do something useful. Over the course of the next few chapters, you will discover how to use Ghidra to perform basic file analysis, learn about Code-Browser and the many common Ghidra display windows, and see how to configure and manipulate those displays to further your understanding of a program's behavior.

PART II

BASIC GHIDRA USAGE

4

BEGINNING YOUR ANALYSIS



It's about time we got down to actually using Ghidra. The remainder of this book is dedicated to its various features and how you can leverage them to best meet your reverse engineering needs. In this chapter, we begin by covering the options you are presented with when you launch Ghidra, and then describe what happens when you open a single binary file for analysis. Finally, we present a quick overview of the user interface to lay the groundwork for the remaining chapters.

Launching Ghidra

Anytime you launch Ghidra, you will be greeted briefly by a splash screen that displays the Ghidra logo, build information, the Ghidra and Java version numbers, and the licensing information. If you wish to thoroughly read the splash screen to learn more about your versions, you can display it at

any time by choosing Help ► About Ghidra from the Ghidra Project window. Once the splash screen clears, Ghidra displays the Ghidra Project window behind a Tip of the Day dialog, as shown in Figure 4-1. You can scroll through the tips by clicking the Next Tip button. When you are ready to start working, close the Tip of the Day dialog.

If you prefer not to see the daily tips, feel free to uncheck the Show Tips on Startup? checkbox at the bottom of the dialog. If you uncheck the box and find yourself missing the Tip of the Day dialog, you can easily restore it through the Ghidra Help menu.

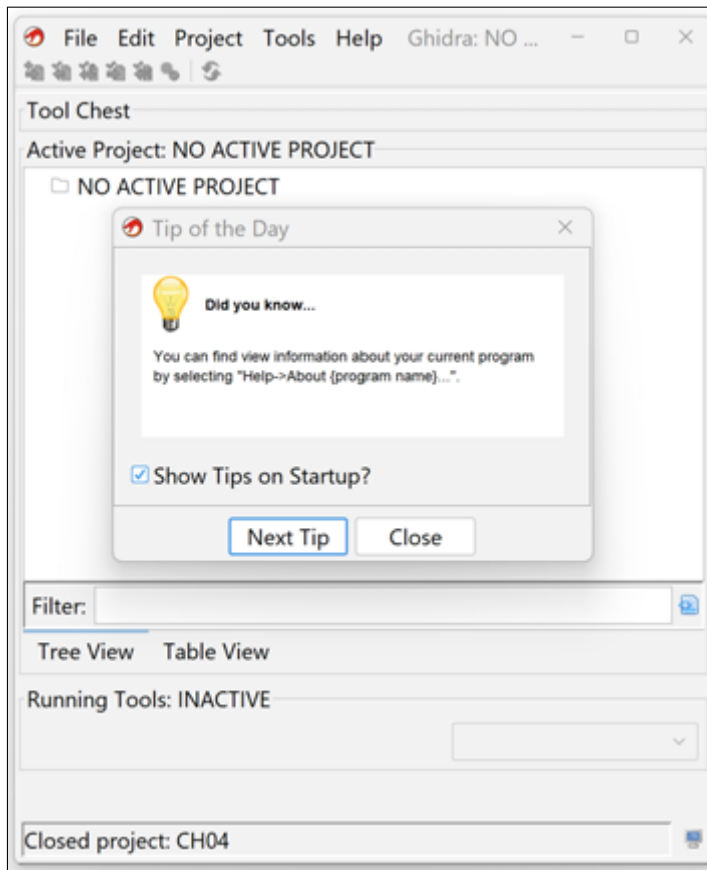


Figure 4-1: The Ghidra Project window and Tip of the Day dialog

If you close the Tip of the Day dialog or uncheck the box and restart Ghidra, you will be presented with the Ghidra Project window. Ghidra uses a project environment to allow you to manage and control the tools and data associated with a file or group of files while working with them. This initial introduction focuses on a single file as a component of a nonshared project. We discuss more complex project capabilities in Chapter 11.

Creating a New Project

If this is your first time launching Ghidra, you will need to create a project. If you have previously launched Ghidra, the active project will be the one you used most recently. Choosing File ► New Project allows you to specify characteristics of the environment associated with the project.

The first step is to choose between a nonshared project and a shared project. In this chapter, we begin with a nonshared project. With that choice out of the way, you will be presented with the dialog in Figure 4-2. Nonshared projects require you to specify a project directory and name.

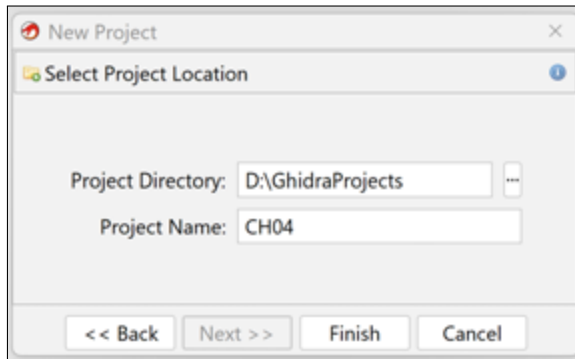


Figure 4-2: Creating a Ghidra project

Once you have entered the project location information, click Finish to complete the project creation process. This will return you to the Project window with the newly created project selected, as shown in Figure 4-3.

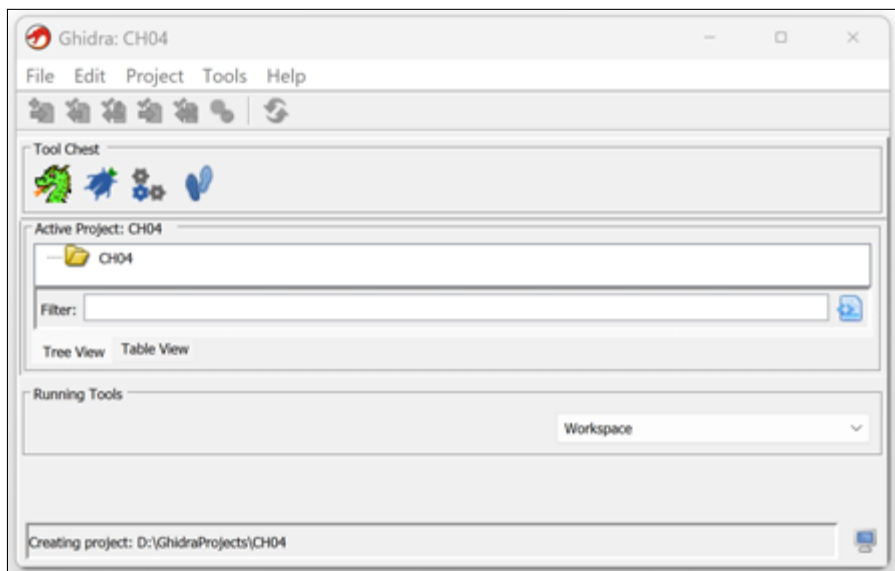


Figure 4-3: The Ghidra Project window

To do any useful work, you will need to add at least one file to your new project.

Loading Files

You can add a file either by choosing File ► Import File and browsing to the file or by dragging and dropping a file directly into a folder in the Project window. Once you have selected a file, you will be presented with the Import dialog shown in Figure 4-4.

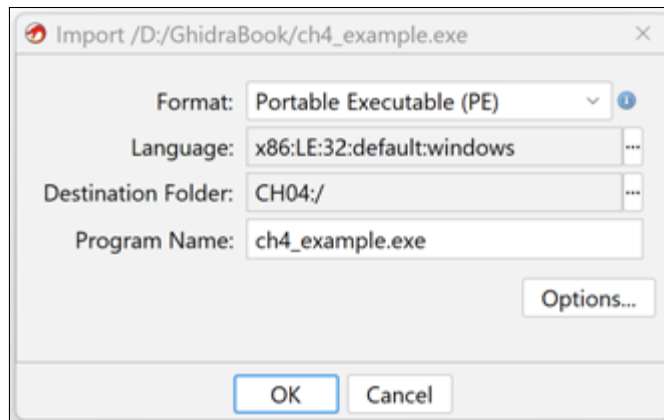


Figure 4-4: The Import dialog

Ghidra generates a list of potential file types and provides these to you in the Format picklist at the top of the dialog. Clicking the Information button to the right of the Format field will list the supported formats for that file, which we describe in Chapter 17.

In this example, the Format picklist provides three options: Portable Executable (PE), Old-style DOS Executable (MZ), and Raw Binary. The Raw Binary option is always present because it is Ghidra’s default for loading files that it does not recognize, and it provides the lowest-level option for loading any file. When offered the choice of several loaders, it is not a bad strategy to accept the default selections unless you possess specific information that contradicts Ghidra’s determination.

The Language field allows you to specify which processor module to use during the disassembly process. A Ghidra language/compiler specification can consist of a processor type, an endianness specification (LE/BE), a bitness value (16/32/64), a processor variant, and a compiler ID (such as ARM:LE:32:v7:default). In most cases, Ghidra will choose the proper processor based on information it reads from the executable file’s headers.

The Destination Folder field lets you select the project folder in which to display the newly imported file. By default, it shows the top-level project folder, but you can add subfolders to organize imported programs within a project. Select the browse buttons to the right of the Language and Destination Folder fields to view other options for each.

You can also edit the text in the Program Name field. “Program Name” is the term Ghidra uses to refer to the imported binary within the project, including for display in the project window. It defaults to the name of the imported file, but you could change it to something more descriptive, such as “Malware from Starship Enterprise.”

You can access other options for controlling the loading process via the Options button. These options depend on the selected format and processor. Figure 4-5 shows those for *ch4_example.exe*, a PE file for x86, with the defaults selected. While moving ahead with the default options is generally a good approach, you may choose other options as you gain experience. For example, you could include the Load External Libraries option if you wanted to add dependent libraries to your project.

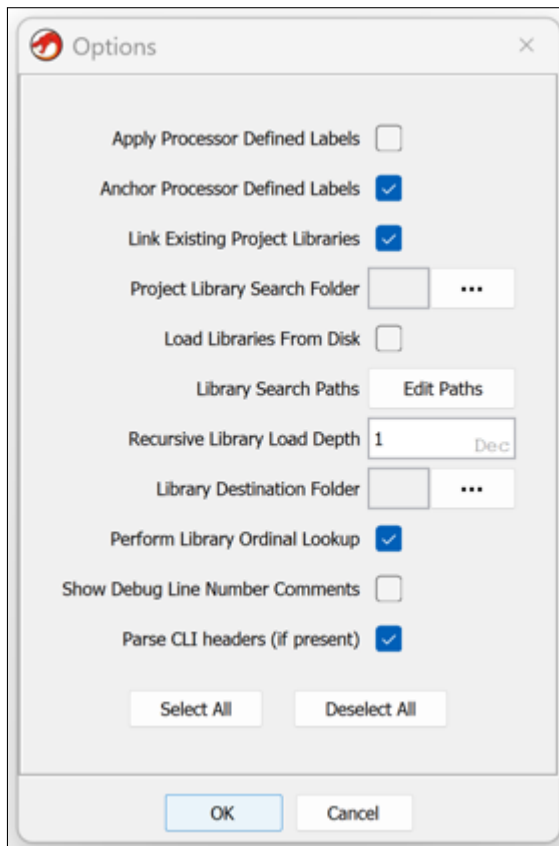


Figure 4-5: The PE file-loading options

When you are satisfied with your loading options, click OK to close the dialog, and Ghidra will present you with an Import Results Summary window, as shown in Figure 4-6.

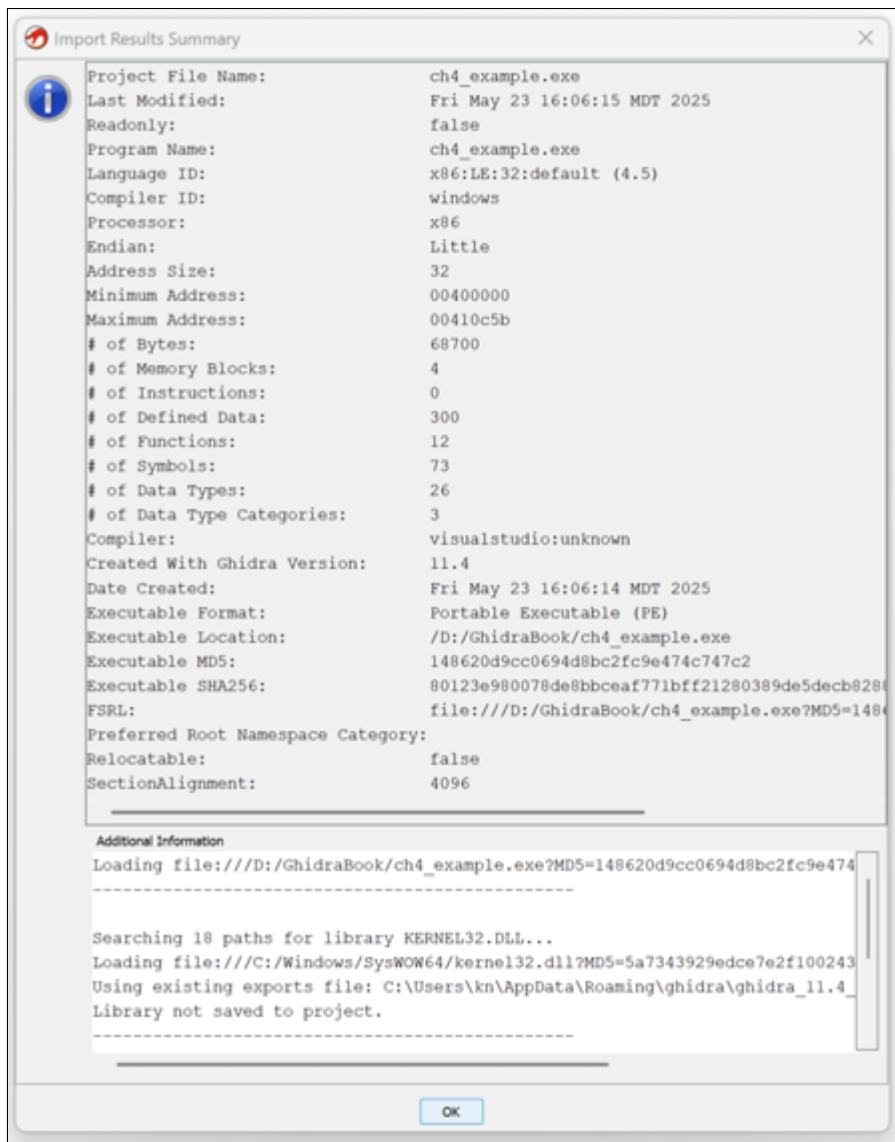


Figure 4-6: The Import Results Summary window

This summary provides you with an opportunity to review the selected import options along with the basic information the loader has extracted from your chosen file. In “Importing Files” on page 280, we discuss ways to modify some of the import results prior to analysis if you have additional information not reflected in the Import Results Summary window.

Using the Raw Binary Loader

At times, Raw Binary will be the only entry in the Format picklist. This is Ghidra's way of telling you that none of its loaders recognize the chosen file. For instance, say you were analyzing custom firmware images or exploit payloads extracted from network packet captures or logfiles. In these cases, Ghidra would not recognize any file header information to guide the loading process, so it would be up to you to perform tasks that loaders often do automatically, like specifying the processor, the bit size, and, in some cases, a particular compiler.

For example, if you know the binary contains x86 code, you can choose from among the many choices available in the Language dialog, as shown in Figure 4-7.

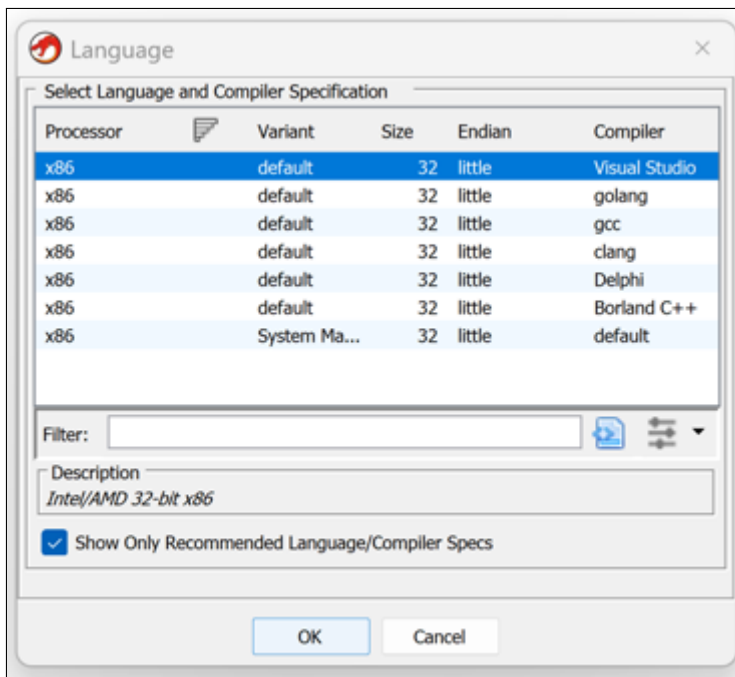


Figure 4-7: Language and Compiler Specification options

Narrowing your language choices to something that will work for your binary may require research and, occasionally, trial and error. Any information you can obtain about the device on which the file was designed to run will be useful. If you are confident that the file is not intended for a Windows system, you should select gcc or default (if available) for the Compiler setting.

If the binary file contains no header information Ghidra can work with, Ghidra also will not recognize the file's memory layout. If you know the base address, file offset, or length of the file, you can enter those values into the corresponding loader option fields shown in Figure 4-8.

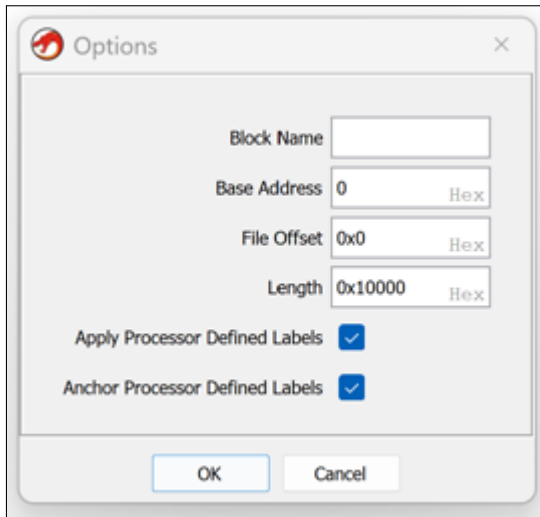


Figure 4-8: The Ghidra Raw Binary loader options

You could also continue to load the file without entering this additional information, and you can provide or adjust these details at any point before or after analysis through the Memory Map window, discussed on page 88. Chapter 17 provides a more detailed discussion of manually loading and organizing unrecognized binary files.

Analyzing Files

At its heart, Ghidra is essentially a database application controlled by a library of plugins, each with its own functionality. It stores all project data using a custom database that evolves as the user adds information to the project. Ghidra's various displays are simply views into the database that reveal information in formats useful to the software reverse engineer. Any modifications made to the database are reflected in the views and saved to the database, but the changes have no effect on the original executable file.

When you double-click a file in the Ghidra Project window, it will present you with the CodeBrowser window, shown in Figure 4-9.

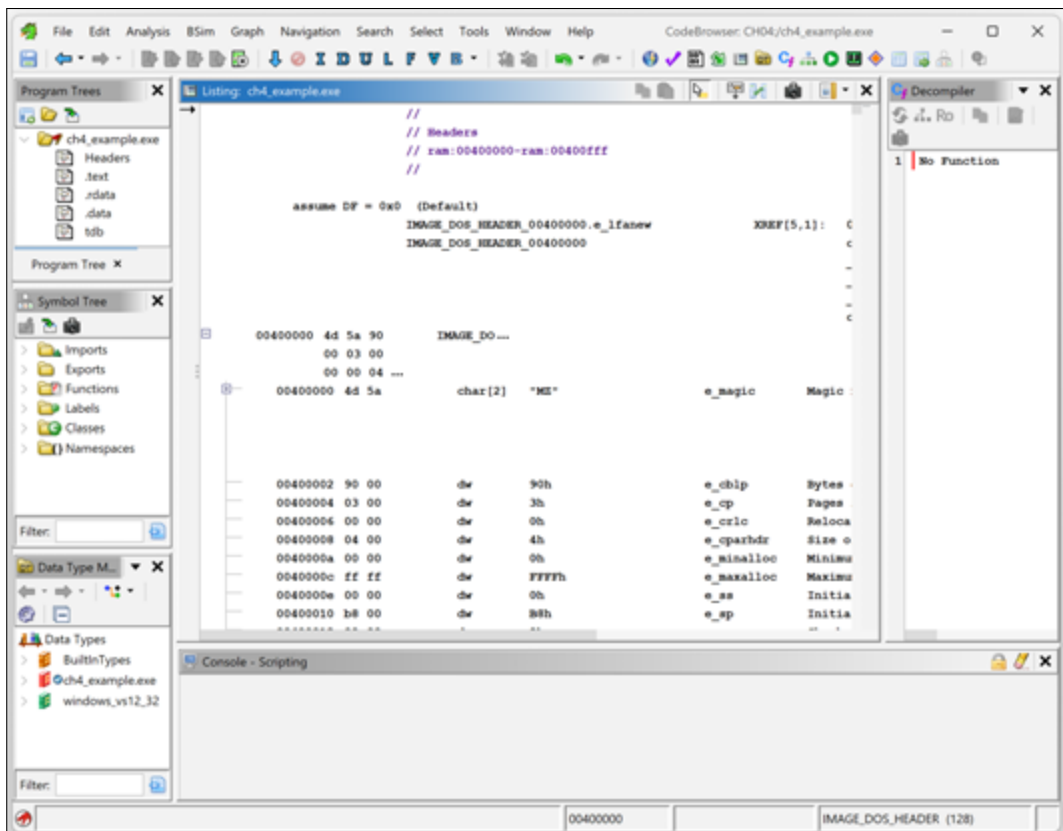


Figure 4-9: The Ghidra CodeBrowser window

The CodeBrowser anchors the many tools available in Ghidra and has unique functionality to help you keep your windows organized, add and delete tools, rearrange content, and document your process.

Auto Analysis

If this is your first time selecting one of the imported files, the CodeBrowser window will present you with an option to allow Ghidra to auto analyze the file. Figure 4-10 shows an example of auto analysis using the Analysis Options dialog.

Ghidra carries out its auto analysis by running each of the selected analyzers over your newly loaded binary. The Analysis Options dialog, as well as Ghidra Help, offers descriptions of each analyzer. As defaults, Ghidra chooses the analyzers Ghidra users have historically found to be the most useful across a wide range of file types.

In the majority of cases involving binaries taken from common platforms and built with commonly available compilers, auto analysis is the correct first choice. You can halt the auto analysis process at any time by clicking the red stop button at the bottom-right corner of the CodeBrowser window. (The button is visible only during auto analysis.)

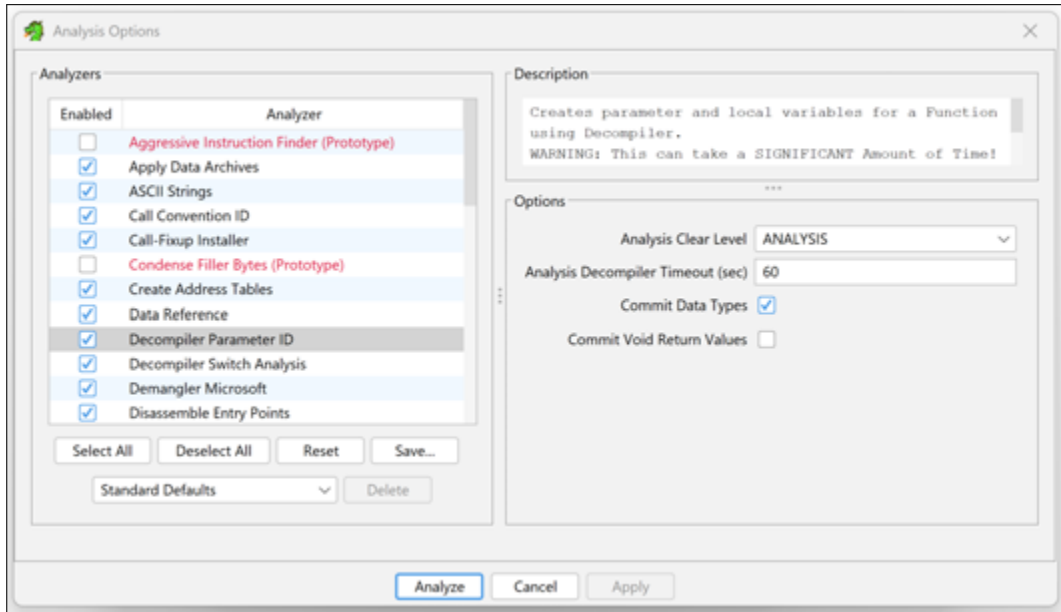


Figure 4-10: The Analysis Options dialog

If you are not happy with Ghidra’s auto analysis, you can always discard your work by closing the CodeBrowser and electing not to save your changes. Then, you can reopen the file and try a different combination of auto analysis options. The most common reasons for modifying these involve unusually structured files such as obfuscated binaries or binaries built with compilers or on operating systems that may be unknown to Ghidra.

Note that if you are working with an extremely large binary (perhaps 10MB or larger) and complex analyzers, Ghidra may take minutes to hours to perform its auto analysis. In such cases, you may opt to disable or set an analysis time-out for some of the more demanding analyzers (for example, Decompiler Switch Analysis, Decompiler Parameter ID, and Stack).

As shown in Figure 4-10, highlighting an analyzer will display a description of the analyzer, which may include useful warnings about the amount of time the analyzer will take to run. In addition, you will see the Options frame, which provides you with an opportunity to control some behavioral aspects of the individual analyzers. Any analysis that you opt to disable or that times out can always be run later using the options available under Ghidra’s Analysis menu.

AUTO ANALYSIS WARNINGS

Once a loader begins to analyze a file, it may encounter issues that it deems important enough to warn you about. One example occurs when you attempt to analyze a program with conflicting analyzers enabled.

Since Ghidra's initial release, its developers have significantly improved its analyzers for Program Database (PDB) files, which contain information to facilitate debugging in binaries compiled with Microsoft compilers.

While Ghidra's initial release had only one PDB analyzer, the current version has two. PDB Universal, the current default run as part of auto analysis, requires significantly less effort on your part than the original. The other option, not enabled by default, is PDB MSDIA.

Enabling both of these analyzers introduces a conflict. In such cases, once analysis is complete, you will likely be presented with an Auto Analysis Summary dialog similar to the one shown in Figure 4-11 that includes a message about any issues encountered.

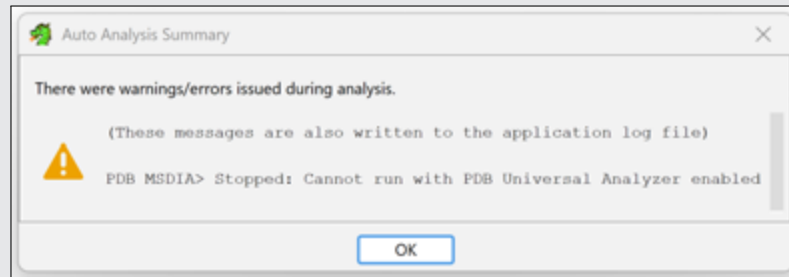


Figure 4-11: The Auto Analysis Summary dialog

In most cases, the messages are simply informational. In some cases, the messages are instructional, offering you suggestions for ways to resolve an issue, perhaps by installing an optional, third-party utility for Ghidra to make use of in the future.

When tools such as Ghidra have access to debugging information from a PDB, they can provide a much more detailed picture of the binary's internal structure, including information like local variable names and types, function prototypes, and layouts for programmer-defined data structures. While PDB files are not often distributed with their associated binaries, Microsoft makes many Windows-related PDBs available through their public symbol servers. Unfortunately, malware authors don't tend to be nearly as generous.

After Ghidra has auto analyzed the file, you can select the filename under Help and you should see the import summary information supplemented with new information about your file, as shown in Figure 4-12.

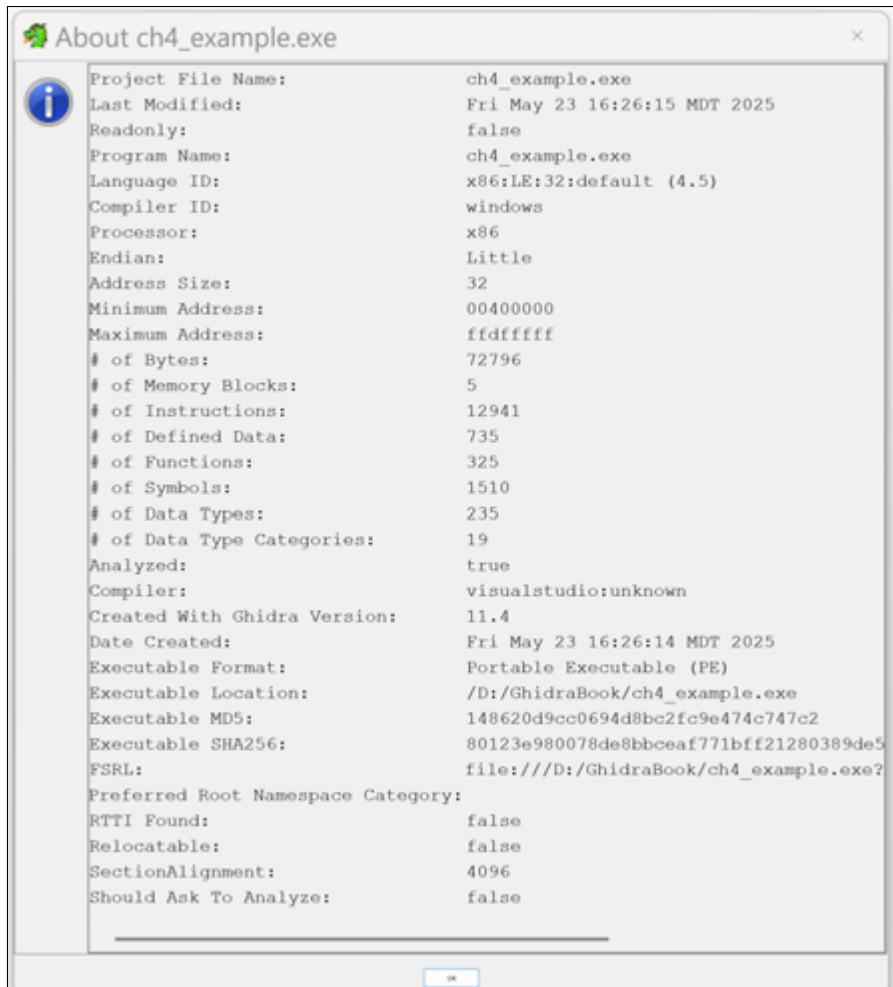


Figure 4-12: A view of import summary information

In the sections that follow, we discuss some of the most useful information extracted from a binary file during its initial loading and subsequent auto analysis.

Compiler Identification

Identifying the compiler used to build a piece of software can help you understand function-calling conventions used in a binary, as well as determine which libraries the binary may be linked with. If the compiler can be identified when a file is loaded, Ghidra's auto analysis will incorporate knowledge of behaviors specific to the identified compiler. You may observe differences when using different compilers and different compile time options; these are the focus of Chapter 20.

Function Argument and Local Variable Identification

Within each identified function (identified from symbol table entries and addresses that are targets of call instructions), Ghidra performs a detailed analysis of the behavior of the stack pointer register in order to recognize accesses to variables located within the stack and understand the layout of the function's stack frame. Names are automatically generated for such variables based on their use either as local variables within the function or as stack-allocated arguments passed into the function as part of the function call process. Chapter 6 discusses stack frames further.

Data Type Information

Ghidra uses its knowledge of common library functions and their associated parameters to identify functions, data types, and data structures used within each function. It adds this information to the Symbol Tree and Data Type Manager windows, as well as the Listing window. This process saves you a tremendous amount of time by providing information that you would otherwise need to manually retrieve and apply from various application programming interface (API) references. We discuss detailed information about Ghidra's handling of library functions and associated data types in Chapter 8.

Desktop Behavior During Initial Analysis

A tremendous amount of activity takes place within the CodeBrowser window during the initial analysis of a newly opened file. You can gain an understanding of this analysis by watching the updates in the bottom right of the CodeBrowser window. These updates also notify you of the progress of the analysis.

If you are not an expert in speed reading, you can open the associated Ghidra logfile and peruse the activities at a more leisurely pace. To do so, select Help ► Show Log. (Note that the Show Log menu option is available only in the Ghidra Project ► Help menu, not in the CodeBrowser ► Help menu.)

The following output is taken from the logfile generated by Ghidra during the auto analysis of *ch4_example.exe* and is representative of messages generated during the auto analysis process. The messages form a narrative of the analysis process and offer insight into the sequence of operations performed by Ghidra as well as the time required for each task during that analysis:

INFO (AutoAnalysisManager) -----	
ASCII Strings	0.003 secs
Apply Data Archives	0.253 secs
Call Convention ID	0.003 secs
Call-Fixup Installer	0.028 secs
Create Address Tables	0.005 secs
Create Address Tables - One Time	0.009 secs
Create Function	0.025 secs

Data Reference	0.009 secs
Decompiler Parameter ID	2.550 secs
Decompiler Switch Analysis	6.264 secs
Demangler Microsoft	0.007 secs
Disassemble	0.013 secs
Disassemble Entry Points	0.087 secs
Embedded Media	0.003 secs
External Entry References	0.000 secs
Function ID	0.101 secs
Function Start Pre Search	0.006 secs
Function Start Search	0.016 secs
Function Start Search After Code	0.004 secs
Function Start Search After Data	0.002 secs
Function Start Search delayed - One Time	0.001 secs
Non-Returning Functions - Discovered	0.011 secs
Non-Returning Functions - Known	0.002 secs
PDB Universal	0.000 secs
Reference	0.014 secs
Scalar Operand References	0.022 secs
Shared Return Calls	0.022 secs
Stack	0.224 secs
Subroutine References	0.015 secs
Subroutine References - One Time	0.001 secs
Windows x86 PE Exception Handling	0.002 secs
Windows x86 PE RTTI Analyzer	0.001 secs
Windows x86 Thread Environment Block (TEB) Analyzer	0.003 secs
WindowsResourceReference	0.265 secs
X86 Function Callee Purge	0.006 secs
x86 Constant Reference Analyzer	0.209 secs

Total Time	10 secs

Even before the auto analysis has completed, you can begin navigating through the various data displays. When the auto analysis is complete, it is safe to make any changes you like to your project file.

Saving and Exiting Options

When you need to take a break from your analysis, it is a good idea to save your work. This is easy to accomplish in any of the following ways:

- Use one of the Save options within the CodeBrowser File menu.
- Click the Save icon in the CodeBrowser toolbar.
- Close the CodeBrowser window.
- Save the project in the Ghidra Project window.
- Choose File ► Exit Ghidra in the Ghidra Project window.

In each case, you will be prompted to save any modified files. We discuss more detailed information about changing the appearance and functionality of CodeBrowser and other Ghidra tools in Chapter 12.

Ghidra Desktop Tips and Tricks

Ghidra displays a tremendous amount of information, and its desktop can quickly become cluttered. Here are some quick tips for making the best use of your desktop:

- The more screen real estate you dedicate to Ghidra, the happier you will be. Use this fact to justify the purchase of a king-size monitor (or four)!
- Don't forget to use the Window menu in the CodeBrowser as a means of opening new views or restoring data displays you have inadvertently closed. You can also open many windows using tool buttons on the CodeBrowser toolbar.
- When you open a new window, it may appear in front of an existing window. When this happens, look for tabs at the top or bottom of the windows that allow you to switch between them.
- You can close any window, reopen it as needed, and drag it to a new location on the CodeBrowser desktop.
- You can control the appearance of displays using Edit ► Tool Options and locating the associated Display options.

While these pointers are just the tip of the iceberg, they should be helpful as you begin to navigate the Ghidra CodeBrowser desktop. We discuss additional CodeBrowser tips and tricks, including shortcuts and toolbar options, in Chapter 5.

Summary

Familiarity with the CodeBrowser tool will greatly enhance your Ghidra experience. Reverse engineering binary code is difficult enough without having to struggle with your tools. The options you choose during the initial loading phase and the associated analysis performed by Ghidra set the stage for all the analysis you will do later.

At this point, you may be content with the work that Ghidra has accomplished on your behalf, and for simple binaries, this may be all that you need. But if you are wondering how you can gain additional control over your reverse engineering process, you are now ready to dive deeper into the functionality of Ghidra's many data displays. In the coming chapters, we will introduce you to each of the primary displays, the circumstances under which you will find each one useful, and how to gain mastery of the tools and displays to optimize your workflow.

5

EXPLORING GHIDRA'S DATA DISPLAYS



At this point, you should have some confidence creating projects, loading binaries into projects, and initiating auto analysis.

Once Ghidra's initial analysis phase is complete, it is time for you to take control.

As discussed in Chapter 4, when you launch Ghidra, your adventure starts in the Ghidra Project window. When you open a file within one of your projects, a second window opens. This is the Ghidra CodeBrowser, and it's your home base for much of your SRE efforts.

You have already used the CodeBrowser to auto analyze your file; now we will take a deeper dive into the CodeBrowser menu, windows, and basic options to increase your awareness of Ghidra's capabilities and allow you to create an SRE analysis environment that is consistent with your personal workflow. Let's begin with the principal Ghidra data displays.

CodeBrowser

You can open the CodeBrowser window by selecting Tools ► RunTool ► CodeBrowser from the Ghidra Project window. Although you will generally open CodeBrowser by selecting a file for analysis, we are opening an empty

instance in this case to demonstrate its functionality and configuration options without specific file-related content influencing the display, as shown in Figure 5-1. In its default configuration, CodeBrowser has six subwindows. Before we get into the details associated with each of these displays, let's spend a little time looking at the CodeBrowser menu and its associated functionality.

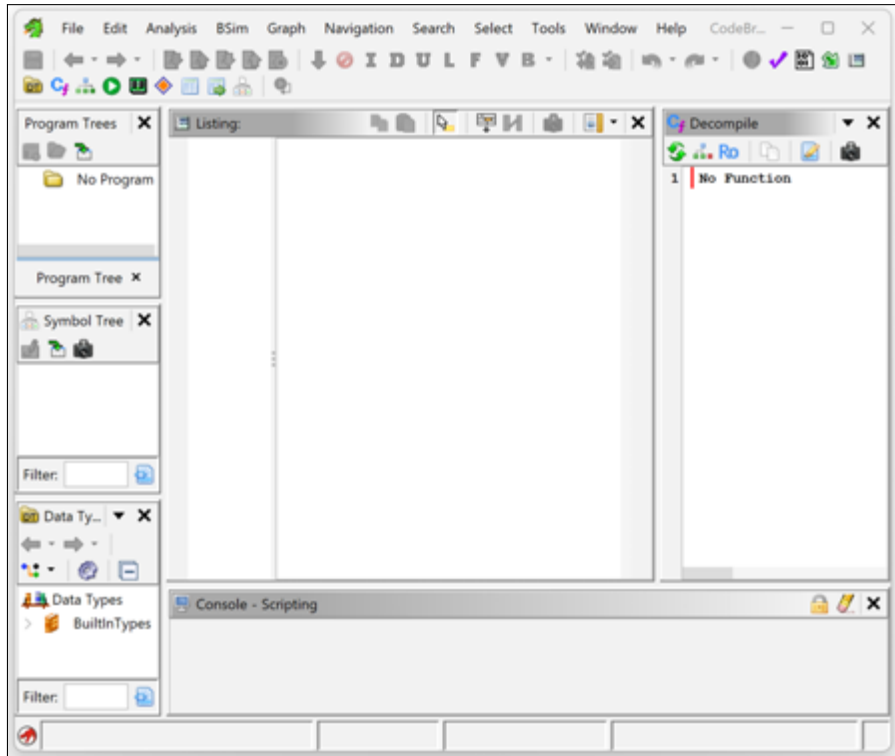


Figure 5-1: The unpopulated default CodeBrowser window

At the top of the CodeBrowser window is the main menu, with a toolbar immediately below. The toolbar provides one-click shortcuts to some of the most commonly used menu options. As we do not currently have a file loaded, we will focus on the menu options that are not associated with a loaded file in this section. We will explain other menu actions in context when we discuss their applicability to the SRE process.

File Provides the basic functionality expected in most file manipulation menus, including options for Open/Close, Import/Export, Save, and Print. In addition, some options are specific to Ghidra, such as the three tool options, which allow you to save and manipulate the CodeBrowser tool, and Parse C Source, which can aid in the decompilation process by extracting data type information from C header files. (See “Parsing C Header Files” on page 288.)

Edit Includes one command that is applicable outside individual sub-windows: the Edit ► Tool Options command, which opens a new window that allows you to control parameters and options associated with the many tools available from the CodeBrowser. Figure 5-2 shows the options related to the console. The Restore Defaults button (revert to default settings) is always available at the bottom right.

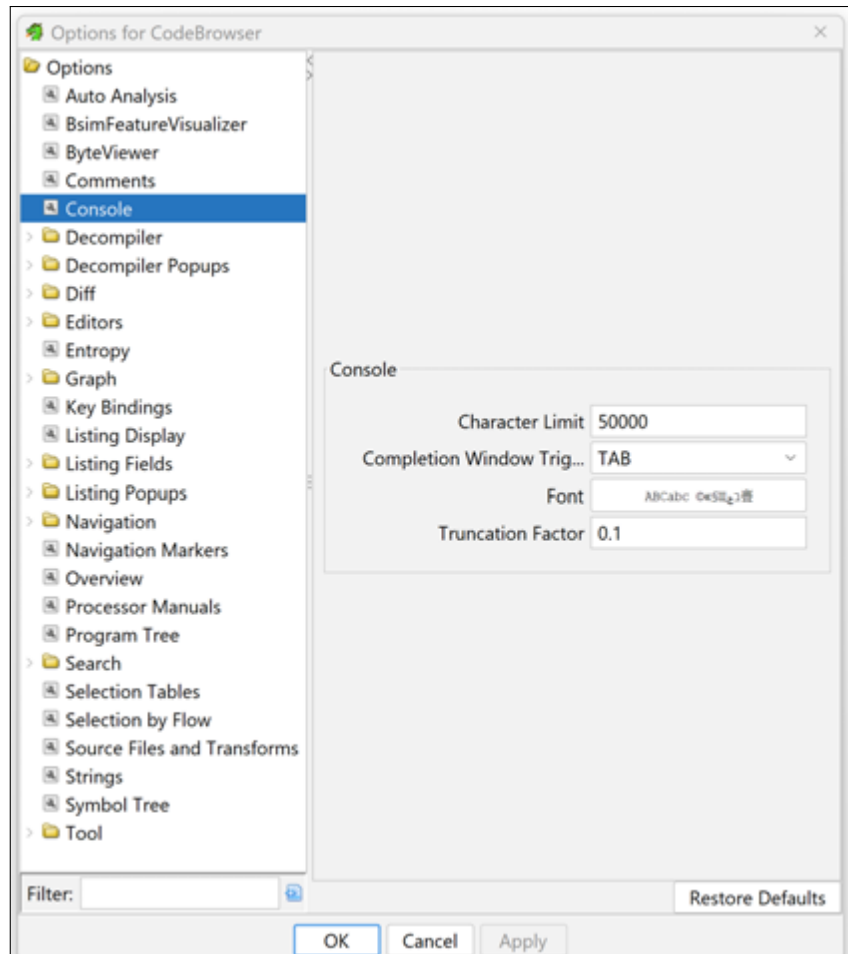


Figure 5-2: CodeBrowser Console Edit options

Analysis Allows you to reanalyze a binary or selectively perform individual analysis tasks. We introduced the basic analysis options in “Analyzing Files” on page 48.

Navigation Facilitates navigation within files. This menu provides the basic keyboard functionality supported by many applications and adds special navigation options for binaries. While the menu provides one method for moving through a file, you will likely use toolbar options or

shortcuts (listed to the right of each menu option) after you gain experience with the many options available for navigation.

Search Provides search capabilities for memory, program text, strings, address tables, direct references, instruction patterns, and much more. We introduce the basic search functionality in “Searching” on page 114. We will present more specialized search concepts in context as part of the many examples in subsequent chapters.

Select Provides the capability to identify a portion of the file to consider for a specific task. Selections can be based on subroutines, functions, or control flows, or made simply by highlighting a desired portion of the file.

Tools Includes some interesting features that allow you to place additional SRE resources on your desktop. One of the most useful is the Processor Manual option, which brings up the processor manual associated with the current file. Owing to copyright restrictions, most processor manuals are not distributed directly with Ghidra. If you attempt to open a missing processor manual, you will be provided with a method to include the manual, as shown in Figure 5-3.

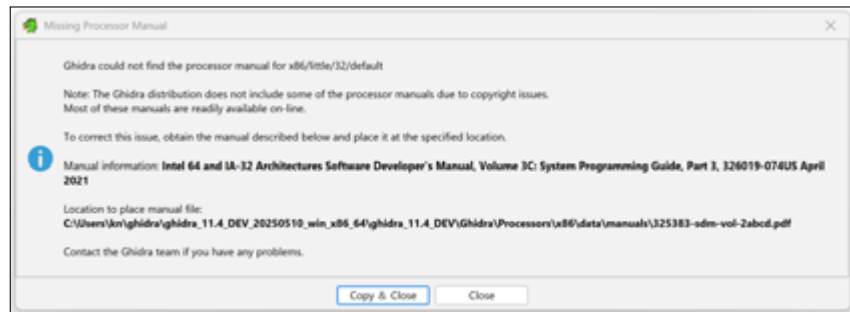


Figure 5-3: The Missing Processor Manual message

Window Allows you to configure your Ghidra work environment for your workflow. We spend most of this chapter introducing and investigating the default Ghidra windows as well as some others that you will find helpful.

Help Provides rich, well-organized, and very detailed options. The Help window supports searching, different views, favorites, and zooming in/out, as well as printing and page setup options.

CodeBrowser Windows

The expanded Window menu can be seen running down the center of Figure 5-4. By default, six of the available windows are opened when CodeBrowser is launched: Program Trees, Symbol Tree, Data Type Manager, Listing, Console, and Decompiler. The name of each window is displayed on the top left of the associated window. Each of these windows appears as

an option on the Window menu, and some also have associated icons on the toolbar directly below the menu. (As an example, we have used arrows in Figure 5-4 to highlight the toolbar option and menu option for opening and accessing the Decompiler window.)

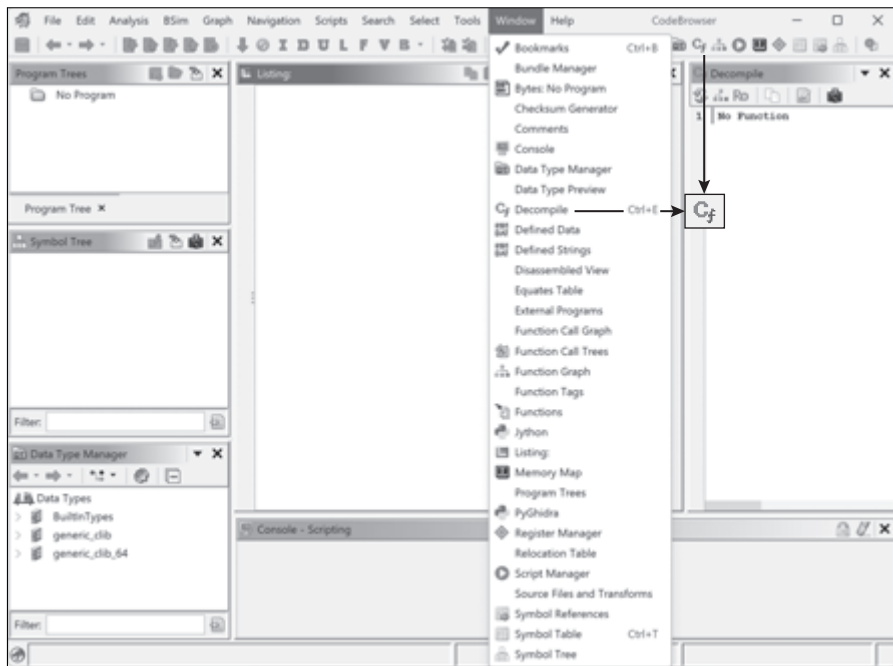


Figure 5-4: The CodeBrowser window with options to display the Decompiler window emphasized

HOTKEYS AND BUTTONS AND BARS, OH MY!

Almost all commonly used actions within Ghidra have an associated menu item, hotkey, and toolbar button. If they don't, you have the power to create them! The Ghidra toolbar is highly configurable, as is the mapping of hotkeys to menu actions. (See CodeBrowser Edit ► Tool Options ► Key Bindings or just hover over a command and press F4.) As if this weren't enough, Ghidra also offers good context-sensitive menu actions in response to right mouse clicks. While these context-sensitive menus do not offer an exhaustive list of permissible actions at a given location, they do serve as good reminders of the most common actions you will be performing. This flexibility allows you to perform an action using the means most comfortable to you and to customize the environment as you discover how Ghidra can work for you.

Let's dive into the six default windows to understand their fundamental importance in the SRE process.

WINDOWS INSIDERS AND OUTSIDERS

As you begin exploring the various Ghidra windows, you will notice that, by default, some windows open within the CodeBrowser desktop and others open as new floating windows outside the CodeBrowser desktop. Let's take a minute to talk about these "insiders" and "outsiders" in the context of the Ghidra environment.

The "outsider" windows float outside the CodeBrowser environment and may be connected or independent. These windows allow you to explore their contents side by side with CodeBrowser. Examples of these windows are Function Graph, Comments, and Memory Map. Next, there are three distinct classes of "insider" windows:

- Windows that open by default in CodeBrowser (for example, Symbol Tree and Listing)
- Windows that are stacked with a default CodeBrowser window (for example, Bytes)
- Windows that create or share space with other CodeBrowser windows (for example, Data Type Preview and Disassembled View)

When you open a window that shares a space with another open window, it appears in front of the existing window. All windows sharing the same space are tabbed to allow rapid navigation between windows. If you want to view two windows that share a space simultaneously, you can click the title bar of the window and drag it outside the CodeBrowser window.

But be careful! Getting windows back into the CodeBrowser window is not as easy as moving them out. (See "Rearranging Windows" on page 246 for more details.)

WHERE'S MY WINDOW?

Ghidra has a lot of windows, and it can be challenging to keep track of where they are at any particular time. This becomes even more complicated as you open more windows and others disappear behind them in CodeBrowser or on your desktop. Ghidra has a unique feature to help you locate those missing windows. Clicking the associated toolbar icon or menu item will move the selected window to the front, but that might not be enough. If you continue clicking the toolbar icon for the window, your missing window will try to catch your attention by vibrating, changing font size or colors, zooming, spinning, and other exciting motions that are sure to catch your eye to help you find it. If you are bored, you can wave back.

The Listing Window

Also known as the Disassembly window, the Listing window will be your primary tool for viewing, manipulating, and analyzing Ghidra-generated disassemblies. The text display presents the entire disassembly listing of a program and provides the primary means for viewing the data regions of a binary.

The CodeBrowser display for *ch5_example1.exe* is shown in its default configuration in Figure 5-5. The margin to the left of the Listing window provides important information about the file as well as your location within the file. An additional marker area on the right side of the Listing window (immediately to the right of the vertical scroll bar) also provides important information and navigational capabilities. The scroll bar indicates your location within the file and can be used for navigation. To the immediate right of the scroll bar, informational displays, including bookmarks, provide additional insight into the file.

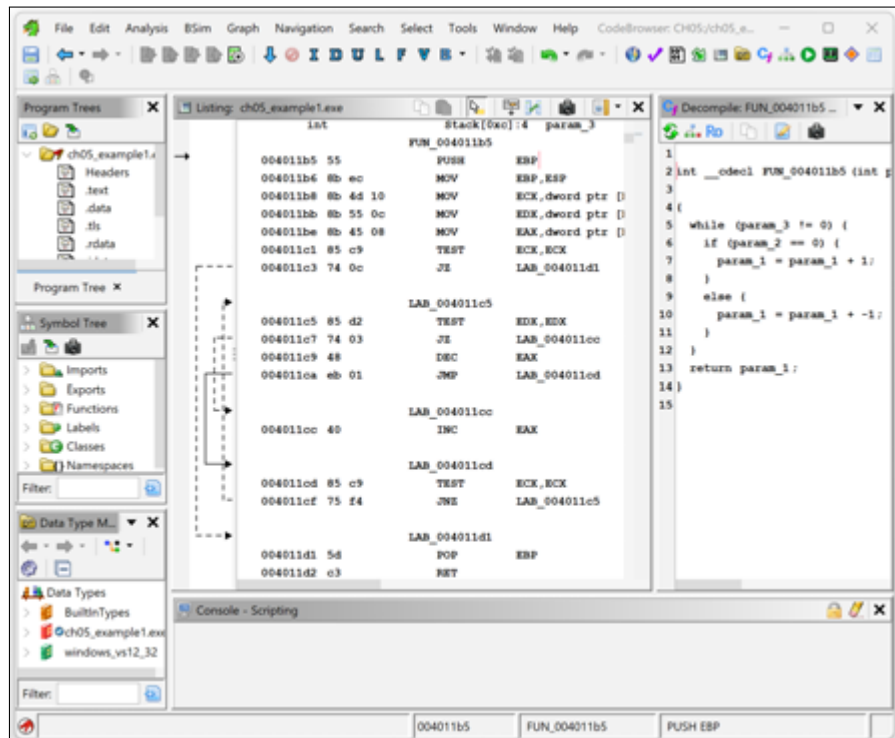


Figure 5-5: The default CodeBrowser window with *ch5_example1.exe* loaded

YOUR FAVORITE BARS

After a file is auto analyzed, you can use informational margin bars to help you navigate and further analyze the file. By default, only the Navigation bar is displayed. You can choose to add (or hide) the Overview bar and the Entropy bar by using the Toggle Overview Margin Displays tool button at the top right of the Listing window (see Figure 5-6). Regardless of which bars are displayed, a navigation marker to the left of all of the bars reminds you of where you are in the file. Left-clicking any location in any of the bars will move you to that location in the file and update the contents of the Listing window.

Now that you know how to control the appearance (and disappearance) of the bars, let's investigate what each bar shows and how you might use it in your SRE process:

Navigation Marker area Allows you to move through the file, but it also has another very important function: If you right-click the Navigation Marker area, you will see the classes of markers and bookmarks that can be associated with your file. By selecting and deselecting marker types, you can control what is displayed in the Navigation bar. This allows you to easily move through particular types of markers (such as highlights).

Overview bar Provides you with important visual information about the contents of a file. The horizontal bands in the Overview bar represent color-coded regions of the program. While Ghidra provides default colors associated with common categories, such as functions, external references, data, and instructions, you can control the color scheme through the Edit ► Tool Options menu. By default, if you hover over a region, you can view detailed information about that region, including the region type and an associated address, if applicable.

Entropy bar Provides the unique Ghidra functionality of “stereotyping” file content based on the file content around it. If there is very little variation within a region, it is assigned a low entropy value. If there is a high degree of randomness, the corresponding entropy value is high. Hovering your mouse over a horizontal band in the Entropy bar will give you the entropy value (between 0.0 and 8.0), a type (for example, `.text`), and the associated address in the file. The highly configurable Entropy bar can be used to help determine the most likely content in the band. You can discover more information about this capability and the mathematics behind it in the Ghidra Help system.

Figure 5-6 provides a breakdown of tool buttons specific to the Listing window.

	Copy to Ghidra Clipboard	This functionality is available in the number of Ghidra windows and varies based on the window in which they appear as well as the content that is selected when the operation is activated. In some cases with incompatible content, you will see an error message.
	Paste from Ghidra Clipboard	
	Toggle Mouse Hovers	This button allows you to choose whether you want mouse hovers to display information or not.
	Browser Field Formatter	This allows you to format the Listing window. (See Chapter 12.)
	Open Diff View	This allows you to compare two files. (See Chapter 23.)
	Snapshot	This button creates and opens a disconnected copy of the Listing window.
	Toggle Overview Margin Displays	This toggle allows you to choose if the entropy and overview bars are displayed.

Figure 5-6: The Listing window tool buttons

In Figure 5-7, we have expanded and zoomed in on the Listing window to investigate what is shown. The disassembly is presented in linear fashion, with the leftmost column displaying virtual addresses by default.

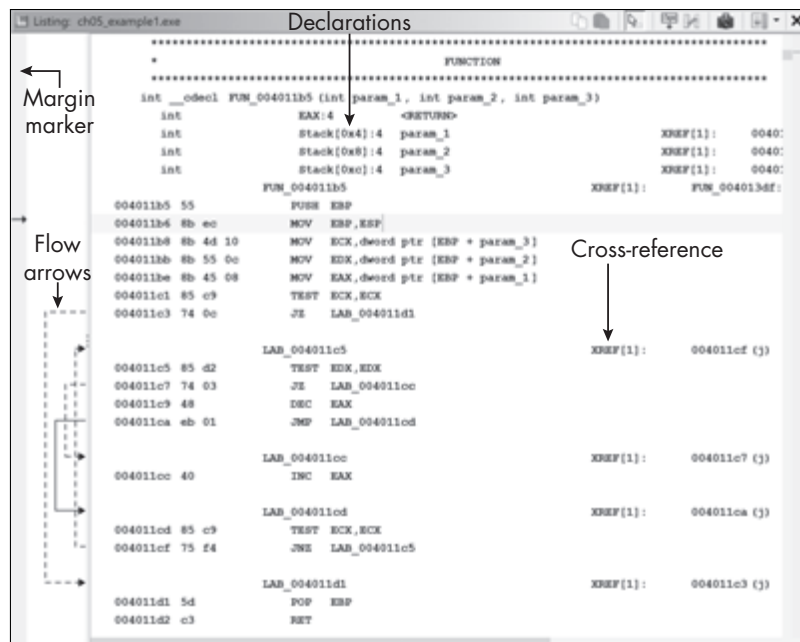


Figure 5-7: The Listing window with labeled example artifacts

Within the Listing window are several items that merit your attention. The gray band to the far left of the window is the margin marker. It is used to indicate your current location in the file and includes point markers and area markers, which are described in Ghidra Help. In this example, the current file location (004011b6) is indicated in the margin marker by the small black arrow.

The region immediately to the right of the margin marker is used to graphically depict nonlinear flow within a function. Ghidra uses the term *flow* to indicate how execution can continue from a given instruction. A *normal* (also called *ordinary*) flow indicates default sequential execution of instructions. A *jump* flow indicates that the current instruction jumps (or may jump) to a nonsequential location. A *call* flow indicates that the current instruction calls a subroutine.

When the source or target address for a control flow instruction is visible in the Listing window, associated flow arrows appear. Solid arrows represent unconditional jumps, while dashed arrows represent conditional jumps. Hovering over a flow line opens a tool tip that displays the start and end addresses of the flow, along with the flow type. When a jump (conditional or unconditional) transfers control to an earlier address in the program, it is often indicative of a loop. This is demonstrated in Figure 5-7 by the flow arrow from address 004011cf to 004011c5. You can easily navigate to the source or destination of any jump by double-clicking the associated flow arrow.

The declarations at the top of Figure 5-7 show Ghidra's best estimate concerning the layout of the function's stack frame. Ghidra computes the structure of a function's stack frame (local variables) by performing detailed analysis of the behavior of the stack pointer and any stack frame pointer used within a function. We discuss stack displays further in Chapter 6.

NOTE

A stack frame (or activation record) is a block of memory, allocated in a program's runtime stack, that contains the function's local variables, any registers the function elects to save on behalf of the caller, and potentially some or all of the arguments passed into the function. Stack frames are allocated upon entry into a function and released as the function exits. We discuss stack frames in more detail in Chapter 6.

Listings generally have numerous data and code cross-references indicated by XREF, shown on the right side of Figure 5-7. A *cross-reference* is created anytime one location in the disassembly refers to another location in the disassembly. For example, an instruction at address A jumping to an instruction at address B would result in the creation of a cross-reference from A to B. Hovering over a reference address causes a reference pop-up to appear with a preview of the location being referred to. The reference pop-up is in the same layout as the Listing window but has a yellow background (similar to a tool tip pop-up). The pop-up window allows you to view the content but does not allow you to follow the references. Cross-references are the subject of Chapter 9.

CONFIGURING LISTING WINDOWS

A single disassembly line is composed of potentially many fields, such as a mnemonic field, operand fields, an address field, and a comment field. The listings we have seen so far have consisted of a default set of fields that provide important information about the file and its contents. However, sometimes the default view provides either too little or too much information. Enter the Browser Field Formatter.

The Browser Field Formatter provides you with the ability to customize over 30 fields to ensure you have ultimate control over the appearance of your Listing windows. You can activate the Browser Field Formatter by clicking its button in the Listing toolbar (refer to Figure 5-6). This opens a powerful submenu and layout editor, shown in Figure 5-8, at the top of the listing.

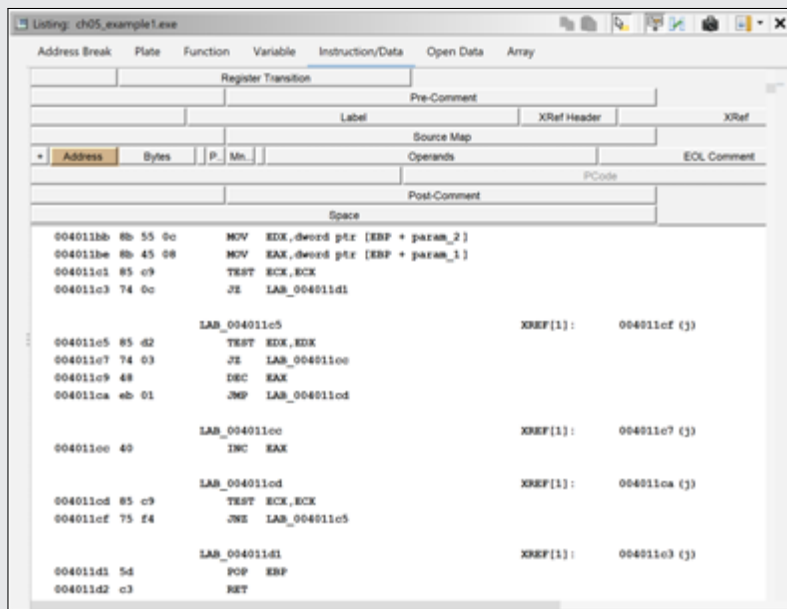


Figure 5-8: The Listing window with the Browser Field Formatter activated

The Browser Field Formatter allows you to control the appearance of address breaks, plate comments, functions, variables, instructions, data, structures, and arrays. Within each of these categories are fields that you can adjust, tune, and control to create the perfect listing format for you. We stick primarily with the default formats for listings, but you should explore the Browser Field Formatter to determine whether any options improve your understanding of the Listing window contents.

Additional Disassembly Windows

If you ever find yourself wanting to view a listing of two functions simultaneously, all you need to do is open another disassembly window by using

the Snapshot icon in the Listing toolbar (refer to Figure 5-6). The first disassembly window opened has the prefix *Listing:* before the filename. All subsequent disassembly windows are titled [*Listing:* <filename>] to indicate that they are disconnected from the primary display. The snapshots are disconnected so you can navigate freely through them without affecting other windows.

The Function Graph Window

While assembly listings are interesting and informative, the flow of the program might be easier to understand by viewing a graph-based display. You can open a Function Graph window associated with the CodeBrowser by choosing Window ► Function Graph or clicking the associated icon in the CodeBrowser toolbar. Figure 5-9 shows a Function Graph window corresponding to the listing in Figure 5-7.

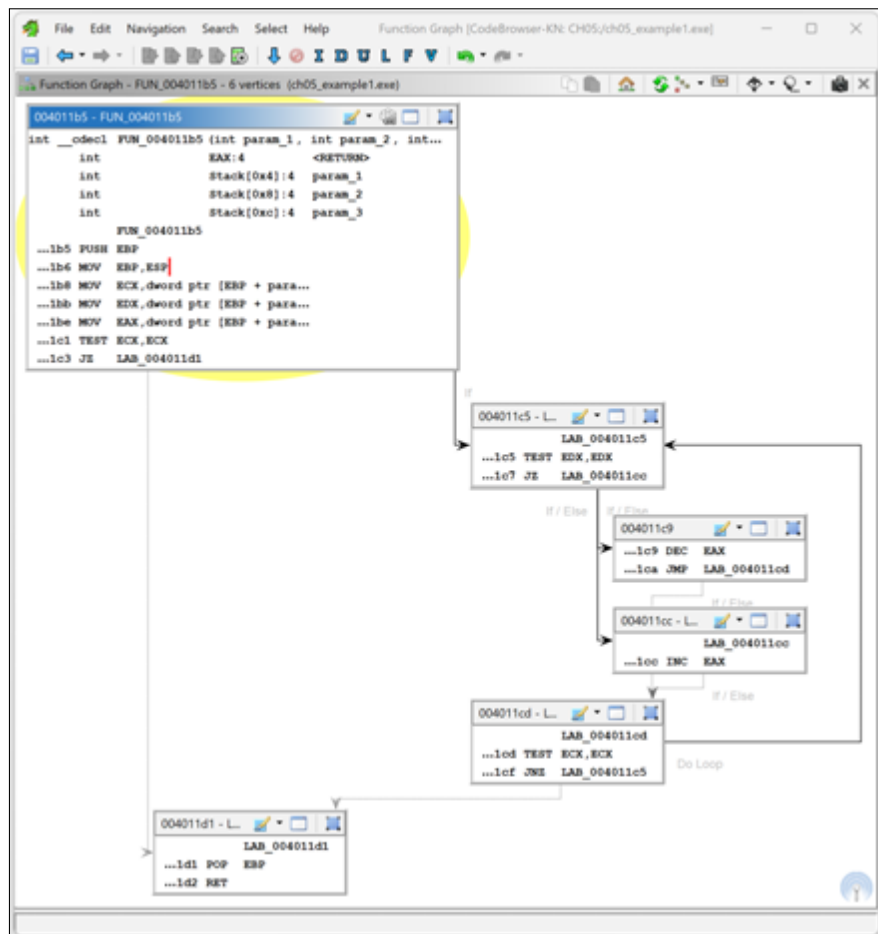


Figure 5-9: A Graph view of the listing from Figure 5-7

Graph views are somewhat reminiscent of program flowcharts in that they break a function into basic blocks so you can visualize the function's control flow from one block to another.

NOTE

A basic block is a maximal sequence of instructions that executes, without branching, from beginning to end. Each basic block has a single entry point (the first instruction in the block) and a single exit point (the last instruction in the block). The first instruction in a basic block is often the target of a branching instruction, while the last instruction in a basic block is often a branch instruction.

Onscreen, Ghidra uses different-colored arrows to distinguish various types of flows between the blocks of a function. In addition, the flows become animated as you mouse over them to indicate direction. Basic blocks that terminate with a conditional jump generate two possible flows: the *Yes edge* arrow (meaning yes, the tested condition was met) is green by default, and the *No edge* arrow (meaning no, the tested condition was not met) is red by default. Basic blocks that terminate with one potential successor block use a *Normal edge* (blue by default) to point to the next block to be executed. You can click any arrow to see the associated transition from one block to another. Since the graph and listing tools are synchronized by default, your file location will generally remain consistent when switching between and navigating within the Listing view and Graph view. You can find exceptions discussed in Chapter 10 as well as in Ghidra Help.

In graph mode, Ghidra displays one function at a time and lets you navigate around the graph by using traditional image interaction techniques such as pan and zoom. Large or complex functions may cause the graph to become extremely cluttered, making it difficult to navigate, which is where the Satellite View can help you. By default, the Satellite View is positioned at the bottom right of the graph window and can be a valuable aid to provide some situational awareness.

TOOLS MAKING CONNECTIONS

Tools can operate together or independently. We have seen how the Listing window and the Function Graph window share data and how events that occur in one window affect the other. If you select a particular block in the Function Graph window, the corresponding code will be highlighted in the Listing window. Conversely, navigating between functions in the Listing window will cause the Function Graph window to be updated. This is one of the many tool connections that happens automatically and is bidirectional. Ghidra also has capabilities for unidirectional connections, as well as the ability to manually connect and disconnect tools using a producer/consumer model associated with tool events. In this book, we focus on the bidirectional automatic tool connections that Ghidra provides.

SATELLITE NAVIGATION

The Satellite View always displays the complete block structure of the graph, along with a highlighted frame that indicates the region of the graph currently being viewed in the disassembly window. Clicking any block in the Satellite View centers the graph around that block. The highlighted frame acts as a lens and can be dragged around the overview window to rapidly reposition the Graph view to any location on the graph. In addition to providing a means to navigate the Function Graph window, this magical window has other powers that can work for or against you as you examine files.

This window consumes valuable space in your Function Graph window and can hide important blocks and contents just when you want to see them. There are two approaches to remedy this situation. You can right-click the Satellite View and uncheck the Dock Satellite View checkbox. This will move the Satellite View and its full functionality outside the Function Graph window. Rechecking the option at any time will move it back to its original location in the Function Graph window.

A second option is to hide the Satellite View, provided that you don't need to use it to navigate. This is another checkbox available in the right-click context menu. When you hide the Satellite View, a small icon will appear in the bottom right of the Function Graph window. Clicking this icon will restore the Satellite View.

When visible, the Satellite View can cause the primary view to behave more slowly than desired. Hiding the Satellite View can help make it more responsive.

In addition to navigating with the Satellite View, you can manipulate the view within the Function Graph window in many ways to suit your needs:

Panning In addition to rapidly repositioning the graph using the Satellite View, you can do so by clicking and dragging the background to change the Graph view.

Zooming You can zoom in and out using traditional keyboard methods such as CTRL/COMMAND, a mouse scroll, or associated key bindings. If you zoom out too far, you may pass the *painting threshold*, where the block contents are no longer displayed. Each block just becomes a colored rectangle. In some cases, particularly when working side by side with the Listing window, this might be advantageous, as it improves the speed at which the function graph can be rendered.

Rearranging blocks Individual blocks within the graph can be dragged to new positions by clicking the title bar for the desired block and dragging it to a new position. All links between blocks are preserved as you move the blocks. If at any point you find yourself wishing to revert to the default layout for your graph, you can do so by selecting the Refresh icon in the Function Graph toolbar.

Grouping and collapsing blocks Blocks can be grouped, either individually or together with other blocks, and collapsed to reduce the clutter in the display. Grouping causes a block to collapse. Collapsing blocks is an easy method to keep track of the blocks you have analyzed. You can collapse any block by choosing the Group icon in the far right of the block toolbar. If you choose this option with multiple blocks selected, they will be collapsed, and the list of associated blocks will be displayed in the stacked window. Some nuances are associated with forming/unforming groups as well as performing actions on the newly formed groups that are explained in Ghidra Help.

CUSTOMIZING YOUR GRAPH DISPLAY

To help you with your analysis, Ghidra provides a menu bar at the top of each node in the Function Graph display that allows you to control the display for that particular node. You can control background/text color for the node, jump to an XREF, view a full window listing of the graph node, and use grouping functionality to combine and collapse nodes. (Note that changing the background for a block in the Function Graph also changes the background in the Listing window.) Some of these features might be unnecessary if you are actively using the Listing window in conjunction with the Function Graph window, but the customization options may be helpful and are certainly worth investigating. We discuss these options further in Chapter 10.

As the graph-based display opens in a window external to CodeBrowser, you can view the two displays side by side. Because the windows have a connection, changing locations in one of the windows moves the location marker in the other window. While many users tend to prefer one view over the other to visualize program flow, you don't have to choose only one. Also, keep in mind that your control over the graph and text views extends far beyond these examples. We cover additional Ghidra graphing capabilities in Chapter 10 and you can find more information on the manipulation of Ghidra's view options in Ghidra Help.

For the next five chapters, we primarily focus on the listing display for examples, supplemented with the graph display in cases where it adds significant clarity. In Chapter 6 we will focus on understanding a Ghidra disassembly, and in Chapter 7 we cover the specifics of manipulating the listing display in order to clean up and annotate a disassembly.

MOVING AROUND

In addition to traditional means of navigating a file (up arrow, down arrow, page up, page down, and so on), Ghidra provides navigation tools specific to the SRE process. The icons in the Navigation toolbar (shown in Figure 5-10) make it easy to move through the program. Let's meet the icons that serve the reverse engineer.



Figure 5-10: The CodeBrowser Navigation toolbar

On the far left is the Direction icon. This arrow toggles between up and down and controls the direction for all of the other navigation icons. Beside the Direction icon is the Invert Search Logic icon. This icon inverts the meaning of the seven icons to its right, which allow you to advance through the various targets shown in Figure 5-11.

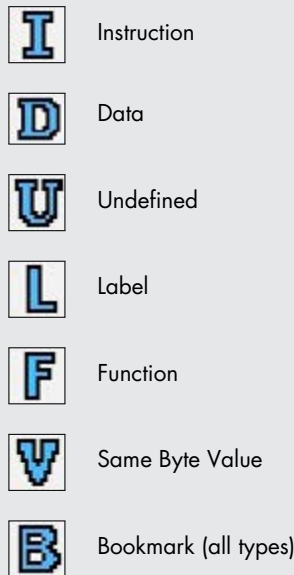


Figure 5-11: Navigation toolbar definitions

Rather than just advancing you to the next data in the listing, choosing the Data option skips over adjacent data and takes you to the start of the next nonadjacent data. Instruction and Undefined demonstrate the same behavior.

The drop-down arrow at the far right of the Navigation toolbar displays a list that allows you to select among specific bookmark types for quick navigation. While used primarily with the Listing window, these navigation shortcuts work in all windows that are connected to the Listing window. Navigating within any of these windows results in synchronous navigation in all connected windows.

The Program Trees Window

Let's return to our discussion of the default CodeBrowser windows by taking a brief look at the Program Trees window, shown in Figure 5-12.

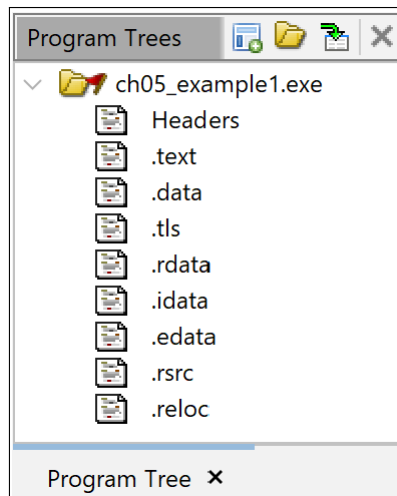


Figure 5-12: The Program Trees window

This window presents a view of your program organized into folders and fragments and provides you with the ability to refine the organization that takes place during auto analysis. *Fragment* is a Ghidra term for a contiguous range of addresses. Fragments may not overlap each other. A more traditional name for a fragment is a *program section* (for example, *.text*, *.data*, and *.bss*). Program tree-related operations include the following:

- Create folder/fragment
- Expand/Collapse/Merge folders
- Add/Remove folders/fragments
- Identify content in Listing window and move to a fragment
- Sort by name/address
- Select addresses
- Copy/Cut/Paste fragment/folders
- Reorder folders

The Program Trees window is a connected window, so double-clicking a fragment in the window navigates you to that location in the Listing window. You can find more information about the Program Trees window in Ghidra Help.

The Symbol Tree Window

When you import a file into a Ghidra project, a Ghidra loader module is selected to load the file content. When present in the binary, the loader is capable of extracting symbol table information (discussed in Chapter 2) for display in the Symbol Tree window shown in Figure 5-13. The Symbol Tree window includes the imports, exports, functions, labels, classes, and namespaces associated with a program. We discuss each of these categories and associated symbol types in the sections that follow.

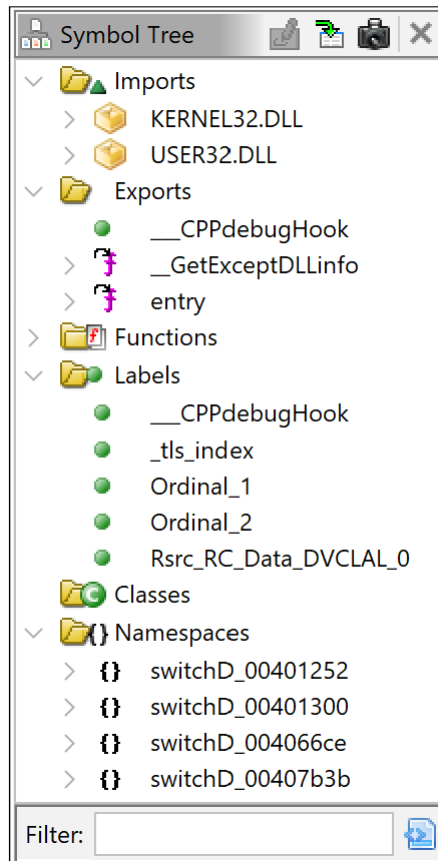


Figure 5-13: The CodeBrowser Symbol Tree window

All six of the Symbol Tree folders can be controlled by the filter at the bottom of the Symbol Tree window. This functionality will become more valuable as you get to know the file you are analyzing. In addition, you will find that the Symbol Tree window offers functionality similar to command line tools such as `objdump (-T)`, `readelf (-s)`, and `dumpbin (/EXPORTS)`.

Imports

The *Imports* folder in the Symbol Tree window lists all functions imported by the binary being analyzed. It is relevant only when a binary makes use of shared libraries—statically linked binaries have no external dependencies and, therefore, no imports. The *Imports* folder lists imported libraries with entries for each item (function or data) imported from that library.

Clicking any symbol within the Symbol Tree view jumps all connected displays to the selected symbol. In our sample Windows binary, clicking the `GetModuleHandleA` in the *Imports* folder would jump the disassembly window to the import address table entry for `GetModuleHandleA`, which in this example resides at address `0040e108`, as shown in Figure 5-14.

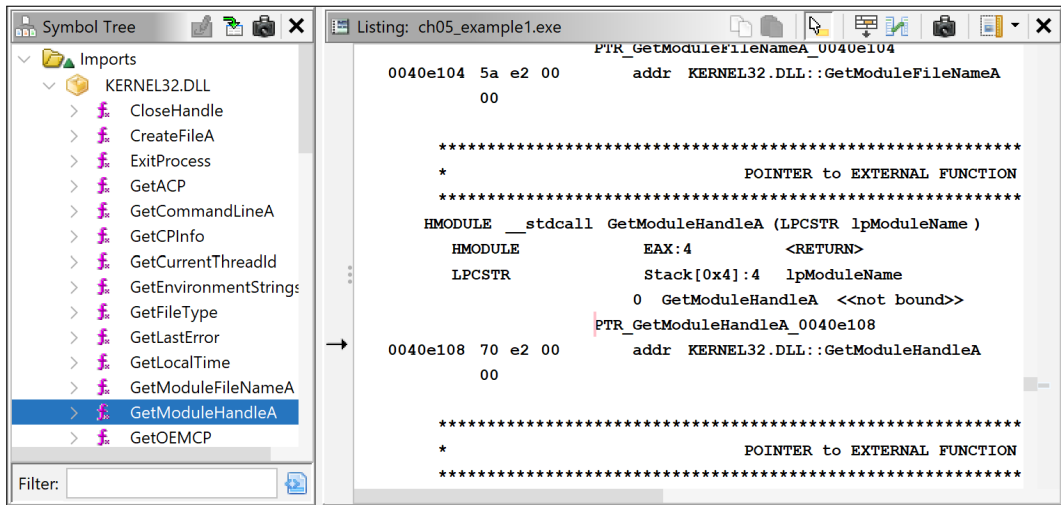


Figure 5-14: An import address table entry and associated location in the Listing window

An important point to remember about the Imports category is that it displays only the symbols named in the binary's import table. Symbols that a binary chooses to load on its own using a mechanism such as `dlopen/dlsym` or `LoadLibrary/GetProcAddress` will not be listed in the Symbol Tree window.

Exports

The *Exports* folder lists the entry points into the file. These include the program's execution entry point, as specified in its header section, along with any functions and variables that the file exports for use by other files. Exported functions are commonly found in shared libraries such as Windows DLL files. Ghidra lists exported entries by name and highlights the corresponding virtual address in the Listing window when the export is selected. For executable files, the *Exports* folder always contains at least one entry: the program's execution entry point. Ghidra may name this symbol entry or `_start`, depending on the binary's type.

Functions

The *Functions* folder contains a list of every function that Ghidra has identified in the binary. Hovering over any function name in the Symbol Tree window generates a pop-up with detailed information about the function, as shown in Figure 5-15.

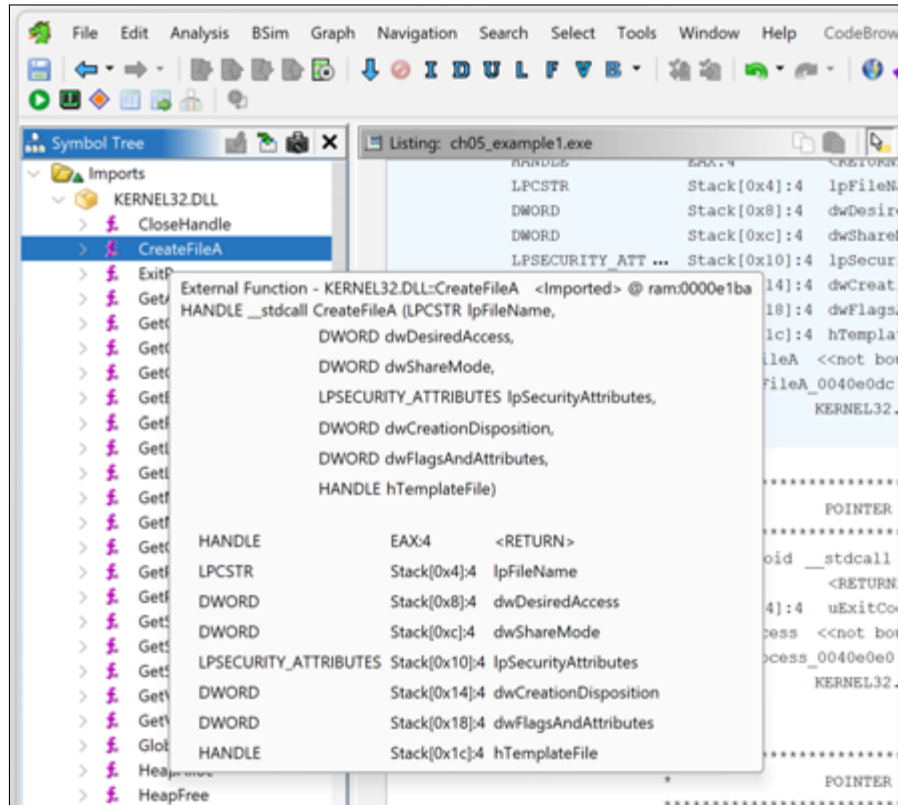


Figure 5-15: A Symbol Tree imported function pop-up

As part of the loading process, the loader utilizes various algorithms, including file structure analysis and byte sequence matching, to infer the compiler used to create the file. During the analysis phase, the *Function ID* analyzer uses this compiler identification information to perform hash-based function body matching and identify the presence of library function bodies that may have been linked into the binary. When a hash match is made, Ghidra retrieves the matched function's name from the hash database (contained in Ghidra *.fidbf* files) and adds the name as a function symbol. Hash matching is particularly useful on stripped binaries, as it provides a means of symbol recovery that is independent of the presence of a symbol table. We discuss this functionality in more depth in “Function IDs” on page 290.

Labels

The *Labels* folder is the data equivalent of the *Functions* folder. Any data symbols contained in a binary's symbol table will be listed in the *Labels* folder. In addition, anytime you add a new label name to a data address, that label will be added to the *Labels* folder.

Classes

The *Classes* folder contains an entry for each class identified by Ghidra during its analysis phase. Under each, Ghidra lists the identified data and methods that may assist you in understanding the behavior of the class. We discuss C++ classes and the structures that Ghidra uses to populate the *Classes* folder in more detail in Chapter 8.

Namespaces

In the *Namespaces* folder, Ghidra may create new namespaces to provide organization and ensure that the assigned names do not conflict in the binary. For example, it may create a namespace for each identified external library or for each switch statement that uses jump tables (allowing jump table labels to be reused in other switch statements without conflicting).

The Data Type Manager Window

The Data Type Manager window, located in the bottom left of the Code-Browser window, allows you to locate, organize, and apply data types to your file by using a system of data type archives. The archives represent Ghidra's accumulated knowledge of predefined data types gleaned from header files included with most popular compilers. By processing header files, Ghidra has learned the data types expected by common library functions and can annotate your disassembly and decompiler listings accordingly. Similarly, from these header files, Ghidra understands both the size and the layout of complex data structures. All of this information is collected into archive files and applied anytime a binary is analyzed.

Referring back to Figure 5-4, you can see the root of the BuiltInTypes tree within the Data Type Manager window. The BuiltInTypes tree contains primitive types like `int` that cannot be changed, renamed, or moved within a data type archive. (This tree is displayed even without a program loaded.) In addition to the built-in types, Ghidra supports the creation of user-defined data types, including structures, unions, enums, and typedefs. It also supports arrays and pointers as derived data types.

Each file you open has an associated entry in the Data Type Manager window. You can see the entry for *ch5_example1.exe* in Figure 5-5, which matches the name of the loaded binary. The folder shares the name of the current file, and entries within the folder are specific to the current file.

The Data Type Manager window displays nodes for each of the data type archives that are open. Archives can be opened automatically, such as when a program references an archive, or manually by the user. We cover data types and the Data Type Manager in more detail in Chapters 8 and 13.

The Console Window

The Console window at the bottom of the CodeBrowser window serves as Ghidra's output area for plugins and scripts, including those you develop yourself, and is the place to look for information on tasks Ghidra is performing as you work with a file. We introduce developing scripts and plugins in Chapters 14 and 15.

The Decompiler Window

The Decompiler window allows you to simultaneously view and manipulate assembly and C representations of your binary through connected windows. The C representation that is generated by the Ghidra decompiler isn't always perfect, but it can be very useful in helping you to understand a binary. Basic functionality provided by the decompiler includes recovery of expressions, variables, function parameters, and structure fields. The decompiler is also often capable of recovering a function's block structure, which can be obscured in assembly language. (Assembly language makes extensive use of statements equivalent to goto, which can be challenging to follow.)

The Decompiler window displays a C representation of a function selected in the Listing window, as shown in Figure 5-16. Depending on your experience with assembly language, the decompiled code may be much easier to understand than the code in the Listing window. Even beginning programmers should be able to identify the infinite loop in the decompiled function. (The while loop condition is dependent on the value of param_3, which is not modified within the loop.)

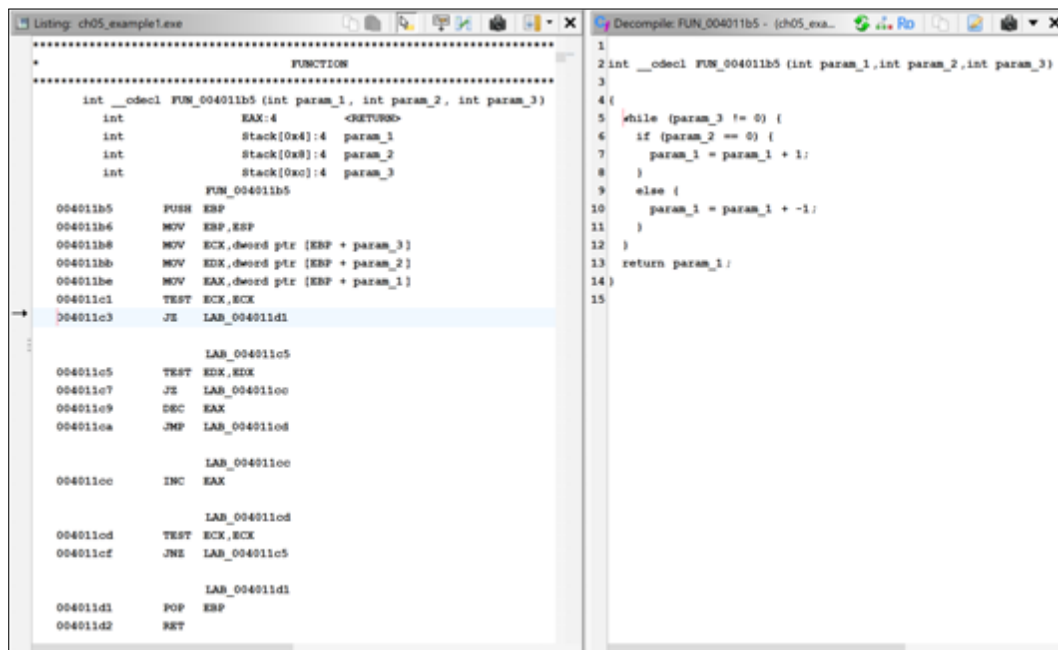


Figure 5-16: The Listing and Decompiler windows

Figure 5-17 shows the Decompiler window icons. You can use the Snapshot icon to open additional (disconnected) Decompiler windows if you want to compare the decompiled version of multiple functions or continue viewing a particular function while moving elsewhere in the Listing window. The Export icon allows you to save the decompiled function to a C file.

		
	Re-decompile	This button re-decompiles the listing when selected.
	Eliminate Unreachable Code	Toggle this button to change the decompiler setting eliminating unreachable code.
	Respect Read-Only Flags	Toggle this button to change the decompiler setting respecting read-only flags.
	Copy to Ghidra Clipboard	This button copies the selected contents from the Decompiler window to the Ghidra clipboard.
	Export	This button exports the decompiled function and lets you choose a file destination.
	Snapshot	This button creates and opens a disconnected copy of the Decompiler window.
	Additional Options	This button displays a submenu including Debug Function Decompilation and options for generating data flow and control flow graphs.

Figure 5-17: The Decompiler window toolbar

Within the Decompiler window, you can right-click a highlighted item to open context menus that allow you to perform actions associated with the item. Figure 5-18 shows the options associated with one of the function parameters, `param_1`.

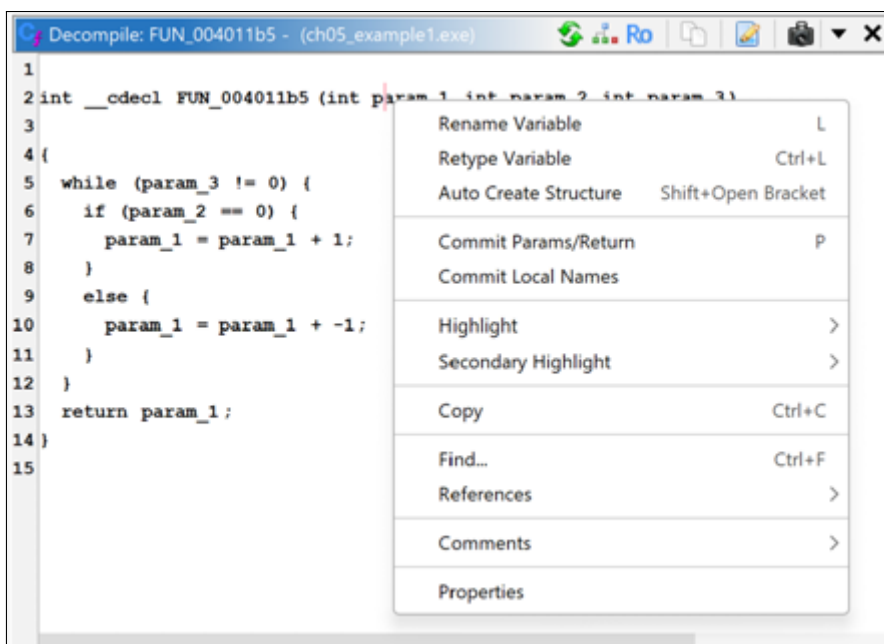


Figure 5-18: The Decompiler window options for function parameters

Decompilation is an extraordinarily complicated process, and decompiler theory remains an active research area. Unlike disassembly, whose accuracy can be verified against manufacturers' reference manuals, no reference manuals provide canonical translations of assembly language back to C (or from C to assembly, for that matter). In fact, while Ghidra's decompiler always generates C source code, it may be the case that the binary the decompiler is analyzing was originally written in a language other than C, and many of the decompiler's C-oriented assumptions may not hold at all.

As with most complex plugins, the decompiler has idiosyncrasies, and the quality of its output depends, to a large extent, on the quality of its input. Many of the issues and irregularities in the Decompiler window can be traced back to issues with the underlying disassembly, so if the decompiled code does not make sense, you may need to spend time improving the quality of the disassembly. In most cases, this involves annotating the disassembly with more accurate data type information, a practice we discuss in Chapters 8 and 13. We will continue to explore the decompiler's capabilities in subsequent chapters and discuss it in depth in Chapter 19.

Other Ghidra Windows

In addition to the six default windows, you can open other windows to support your SRE process with alternate or specialized views of the file. The CodeBrowser ► Window menu displays the list of available windows, as shown previously in Figure 5-4. The utility of these displays depends on both the characteristics of the binary you are analyzing and your skill with Ghidra. Several of these windows are sufficiently specialized to require more detailed coverage in later chapters, but we introduce some common ones here.

The Bytes Window

The Bytes window provides a raw look at the byte-level content of the file. By default, the Bytes window opens on the upper right side of the CodeBrowser and provides a standard hex dump display of the program contents with 16 bytes per line. The window doubles as a hex editor and can be configured to display a variety of formats by using the Settings tool in the Bytes window toolbar. In many cases, it might be helpful to add the ASCII display to the Bytes window, as shown in Figure 5-19. The figure also shows the Byte Viewer Options dialog and toolbar icons to edit or snapshot the byte view.

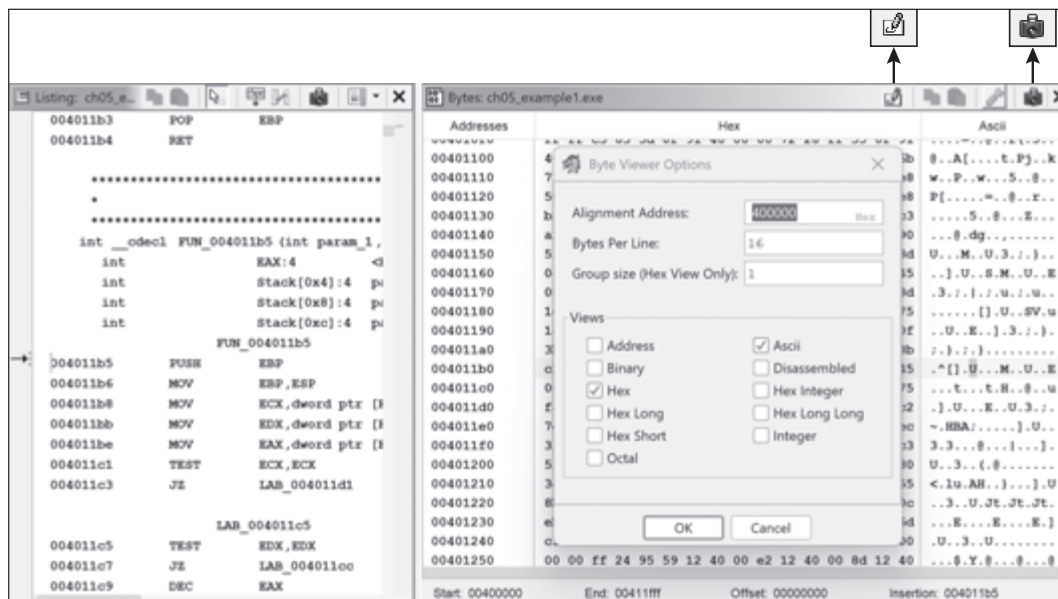


Figure 5-19: The synchronized hex and disassembly views with the Toggle and Snapshot icons emphasized

As with the Listing window, you can open several Bytes windows simultaneously using the Snapshot icon (see Figure 5-19) in the Bytes window toolbar. By default, the first Bytes window has a connection to the Listing window, so scrolling in one window and clicking an element causes the other window to scroll to the same virtual address. Subsequent Bytes windows are disconnected, which allows you to scroll through them independently. When a window is disconnected, the window name appears within square brackets in the window title bar.

To turn the Bytes window into a hex (or ASCII) editor, simply toggle the pencil icon highlighted in Figure 5-19. While you will not be able to edit anything at addresses that contain an existing code item, such as an instruction, the font will be red for any changes you do make. When you are finished editing, toggle the icon again and you will be back in read-only mode. (Note that any changes will not be reflected in disconnected Bytes windows.)

Although we speak of using the Bytes window as an editor, it is important to understand that you are editing only the content that Ghidra has loaded into the database representing the file under analysis. Edits made within the Bytes window are never propagated to the original binary file, which remains unchanged by Ghidra following the initial loading process.

If the Hex column displays question marks rather than hex values, Ghidra is telling you that it is not sure what values might occupy a given virtual address range. Such is the case when a program contains a bss section, which typically occupies no space within a file but is expanded by the loader to accommodate the program's static storage requirements.

NOTE

A bss section is created by a compiler to house all of a program's uninitialized, static variables. Since no initial value is assigned to these variables, there is no need to allocate space for them in the program's file image, so the section's size is noted in one of the program's headers. When the program is executed, the loader allocates the required space and initializes the entire block to zero.

The Defined Data Window

The Defined Data window displays a string representation of data defined in the current program, view, or selection, along with the associated address, type, and size, as shown in Figure 5-20. As with most of the columnar windows, you can sort by any column in ascending or descending order by clicking the column header. Double-clicking any row in the Defined Data window causes the Listing window to jump to the address of the selected item.

When used with cross-references (discussed in Chapter 9), the Defined Data window provides the means to quickly spot an interesting item and to track back to any location in the program that references that item with only a few clicks. For example, you might see the string "SOFTWARE\Microsoft\Windows\Current Version\Run" listed and wonder why an application is referencing this particular key within the Windows registry, and then discover that the program is setting that registry key to automatically start itself when Windows boots.

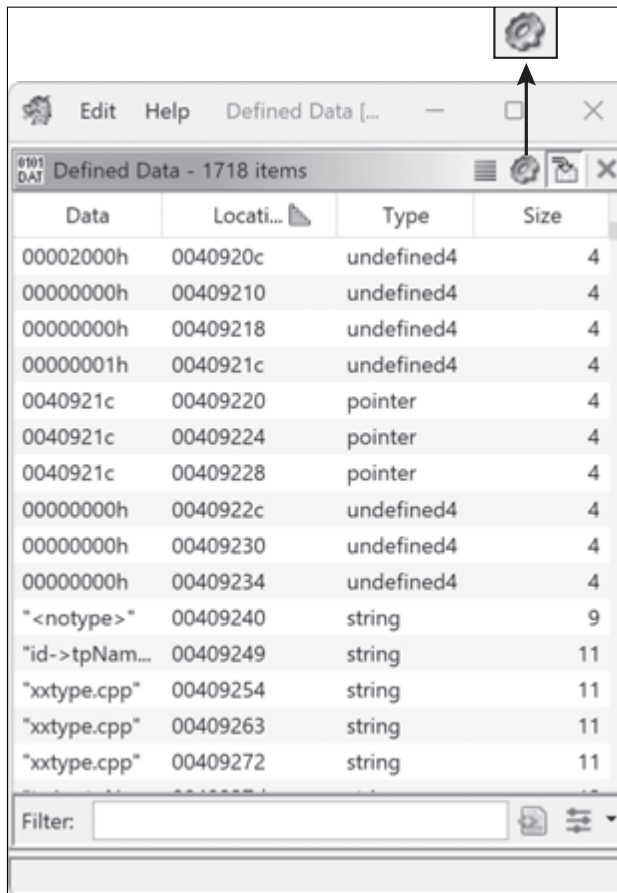


Figure 5-20: The Defined Data window with the Filter icon emphasized

The Defined Data window has extensive filtering capabilities. In addition to the Filter bar at the bottom of the window, a Filter icon at the top right (emphasized in Figure 5-20) allows you to control additional data type filter options, as shown in Figure 5-21.

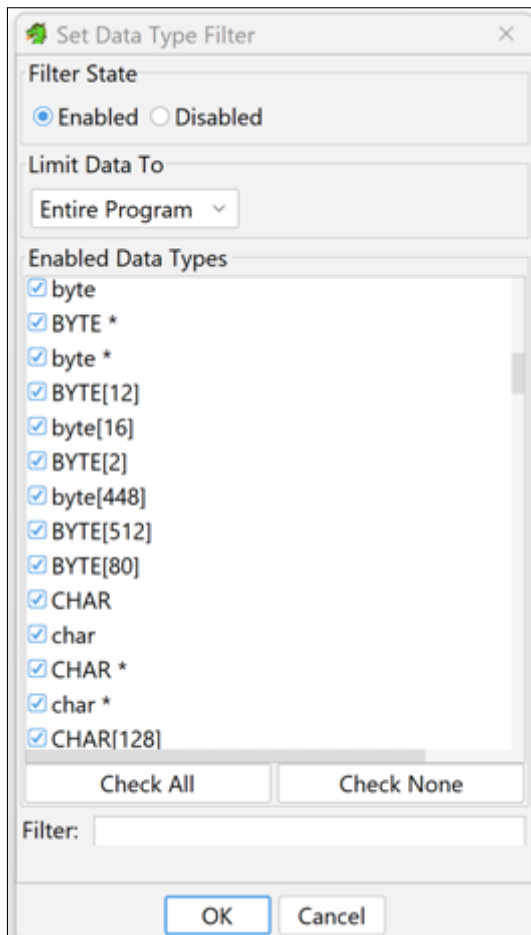


Figure 5-21: Defined data type filter options

Every time you close the Set Data Type Filter dialog by clicking OK, Ghidra will regenerate the Defined Data window contents in accordance with the new settings.

The Defined Strings Window

The Defined Strings window displays any strings defined in the binary. Figure 5-22 shows an example of this window.

Locati...	String Value	String Repres...	Data Type
00409294	tp2->tpName	"tp2->tpName"	ds
004092a0	xctype.cpp	"xctype.cpp"	ds
004092ab	IS_STRUC(bas...	"IS_STRUC(ba...	ds
004092c2	xctype.cpp	"xctype.cpp"	ds
004092cd	IS_STRUC(der...	"IS_STRUC(de...	ds
004092e4	xctype.cpp	"xctype.cpp"	ds
004092ef	derv->tpClass...	"derv->tpClas...	ds
00409315	xctype.cpp	"xctype.cpp"	ds
00409320	((unsigned __f...	"((unsigned __...	ds
00409347	xctype.cpp	"xctype.cpp"	ds
00409352	<notype>	"<notype>"	ds
0040935b	topTypPtr != ...	"topTypPtr != ...	ds
00409389	xctype.cpp	"xctype.cpp"	ds
00409394	tgtTypPtr != 0...	"tgtTypPtr != ...	ds
004093c2	xctype.cpp	"xctype.cpp"	ds
004093cd	srcTypPtr == ...	"srcTypPtr ==...	ds
004093fb	xctype.cpp	"xctype.cpp"	ds
00409406	__isSameType...	"__isSameTyp...	ds
00409430	xctype.cpp	"xctype.cpp"	ds

Figure 5-22: The Defined Strings window

In addition to the default columns displayed in the figure, you can add columns by right-clicking the row of column titles. Perhaps one of the most interesting available columns is the Has Encoding Error flag, which can indicate an issue with the character set or a misidentification of a string. In addition to this window, Ghidra has substantial string search functionality, which we discuss in Chapter 6.

The Symbol Table and Symbol References Windows

The Symbol Table window provides a summary listing of all global names within a binary. By default, it displays seven columns, as shown in Figure 5-23. The window is highly configurable, granting users the ability to add and delete columns in the display and sort the contents of any column in ascending or descending order. The first two default columns are Name and Location. A *name* is nothing more than a symbolic description given to a symbol defined at a *location*.

The Symbol Table is connected to the Listing window but provides the capability to control its interaction with the Listing window. The emphasized icon on the right in Figure 5-23 is a toggle that determines whether a single click on a location in the window causes a related move in the Listing window. Regardless of the toggle selection, double-clicking any Symbol Table location entry will immediately jump the Listing view to the selected entry. This provides a useful tool for rapidly navigating to known locations within a program listing.

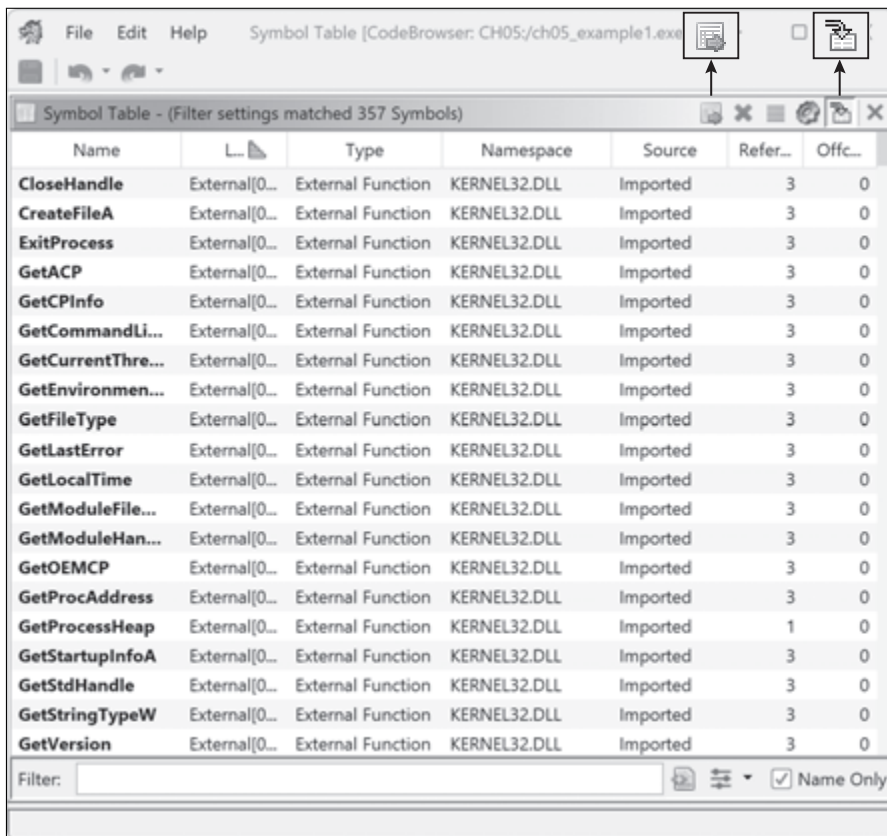


Figure 5-23: The Symbol Table window with Display Symbol References and Navigation Toggle icons emphasized

The Symbol Table window offers extensive filtering capabilities and several ways to access the filtering options. The cog icon in the toolbar opens the Symbol Table Filter dialog. Figure 5-24 shows this dialog (with the Use Advanced Filters box checked). In addition to this dialog, you can use the Filter options at the bottom of the window. You can find thorough discussions of the symbol table filtering options in Ghidra Help.

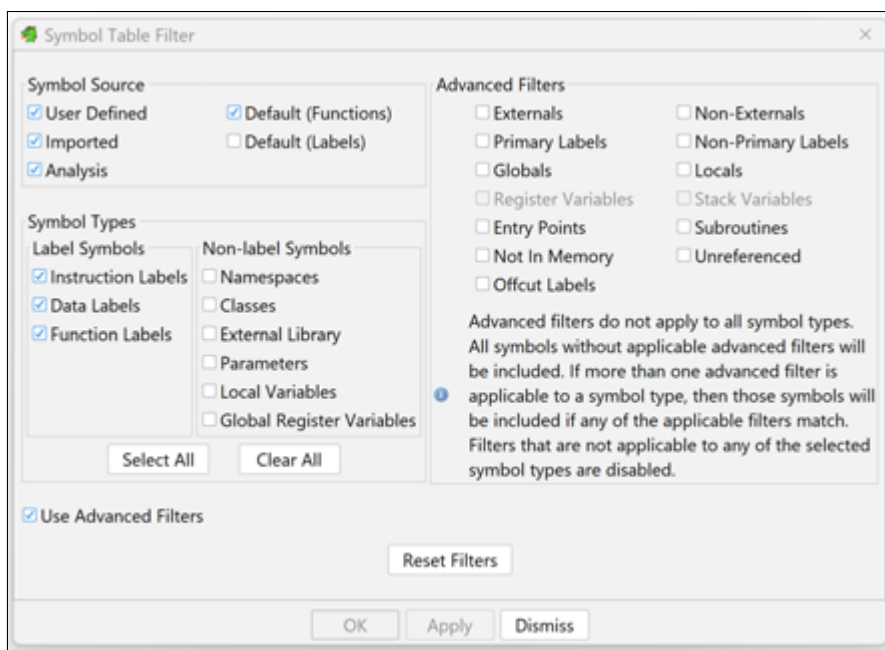


Figure 5-24: The Symbol Table Filter dialog

The emphasized icon on the left in Figure 5-23 is the Display Symbol References icon. Clicking this icon adds the Symbol References window to the Symbol Table window. By default, these two tables will appear side by side. To improve readability, you can drag the Symbol References window below the Symbol Table window, as shown in Figure 5-25. The connection between these two tables is unidirectional; the Symbol References table is updated when you make a selection in the Symbol Table.

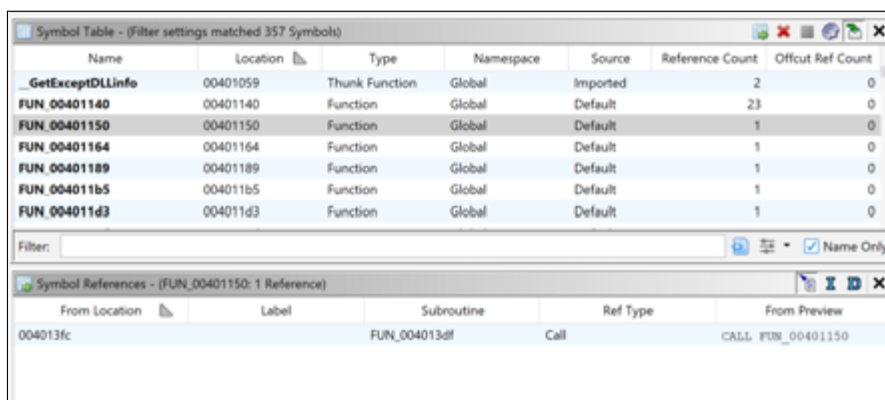


Figure 5-25: The Symbol Table with Symbol References

Like the Symbol Table window, the Symbol References window has the same column organization controls. In addition, the content of the Symbol References window is controlled by the three icons (S, I, and D) at the top right of the Symbol References toolbar. These options are mutually exclusive, meaning only one can be selected at a time:

S icon When this icon is selected, the Symbol References window will display all *references to* the symbol that you have selected in the Symbol Table. Figure 5-25 shows a Symbol References window with this option selected.

I icon When this icon is selected, the Symbol References window will display all instruction references from the function that you have selected in the Symbol Table. This list will be empty if you did not select a function entry point.

D icon When this icon is selected, the Symbol References window will display all data references from the function that you have selected in the Symbol Table. This list will be empty if you did not select a function entry point or if the function makes no reference to any data symbols.

The Memory Map Window

The Memory Map window displays a summary listing of the memory blocks present in the program, as shown in Figure 5-26. Note that what Ghidra terms *memory blocks* are frequently called *sections* when discussing the structure of binary files. Information presented in the window includes the memory block (section) name, the start and end addresses, the length, permission flags, the block type, the initialized flag, and a space for the source filename and user comments. The start and end addresses represent the virtual address range to which the program sections will be mapped at runtime.

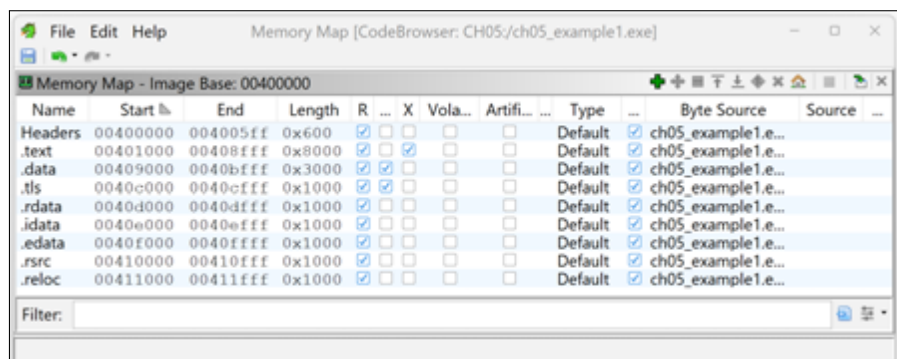


Figure 5-26: The Memory Map window

Double-clicking any start or end address in the window jumps the Listing window (and all other connected windows) to the specified address. The Memory Map window toolbar provides options to add/delete blocks, move blocks, split/merge blocks, edit addresses, and set a new image base. These

features are particularly useful when reverse engineering files with nonstandard formats, as Ghidra's loader may not have detected the binary's segment structure.

The Function Call Graph Window

In any program, a function can both call and be called by other functions. The Function Call Graph window shows the immediate neighbors of the function in which the cursor is positioned. For our purposes, we will call Y a neighbor of X if Y directly calls X or if X directly calls Y.

For example, Figure 5-27 shows a function named FUN_0040198c that is called from FUN_00401edc and, in turn, calls six other functions. Double-clicking any function in the window immediately jumps the Listing window and other connected windows to the selected function. Ghidra cross-references (XREFs) underlie the generation of the Function Call Graph window. We cover XREFs in more detail in Chapter 9.

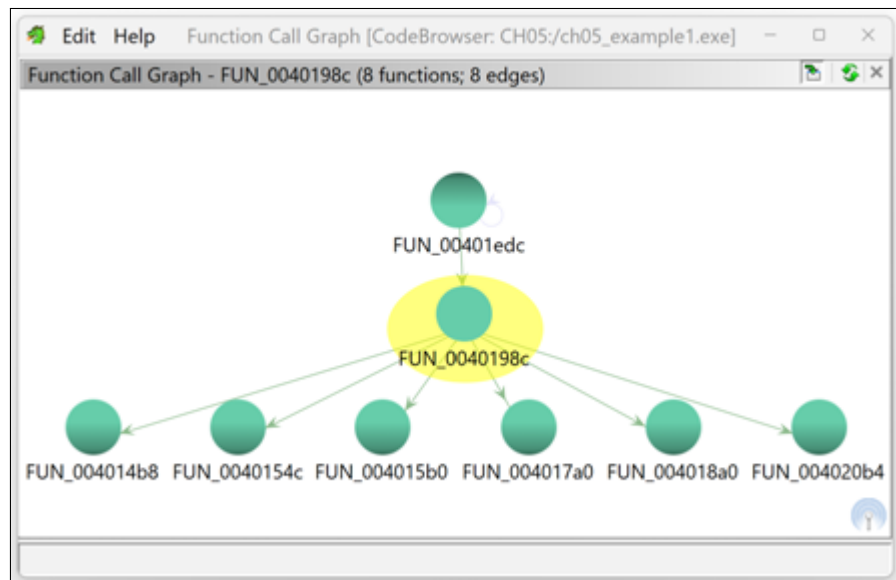


Figure 5-27: The Function Call Graph window

The Function Call Trees Window

While the Function Call Graph window is helpful, sometimes you need the big picture, or at least a bigger picture. The Function Call Trees window (Window ► Function Call Trees) allows you to see all calls to and from a selected function. As shown in Figure 5-28, the Function Call Trees window has two sections: one for incoming calls and one for outgoing calls. You can expand and collapse both incoming calls and outgoing calls as desired.

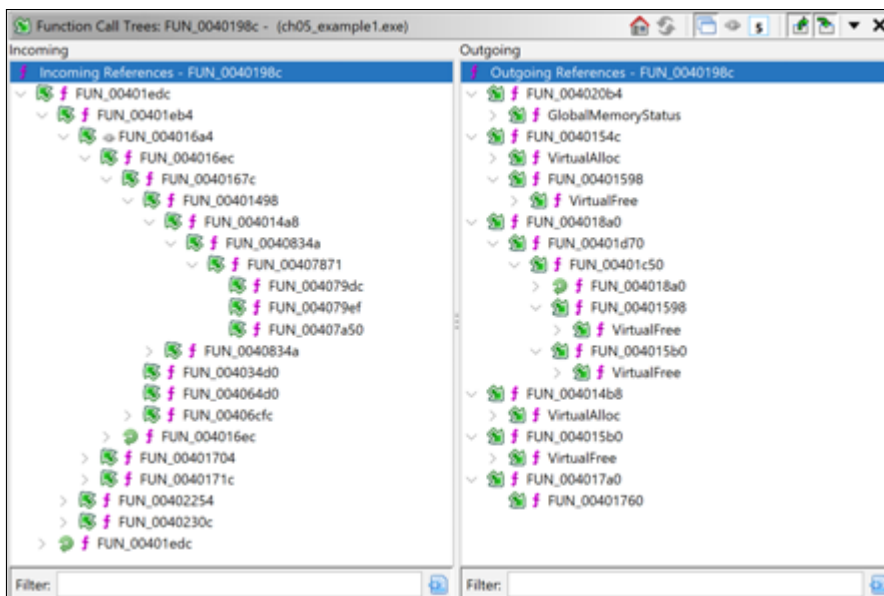


Figure 5-28: The Function Call Trees view

If you open the Function Call Trees window with the entry function selected, you can view a hierarchical representation of the program’s function calls.

Summary

At first glance, the number of displays that Ghidra offers can seem overwhelming. You may find it easier to stick to the default displays until you are comfortable enough to begin exploring the additional display offerings. In any case, you should certainly not feel obligated to use everything Ghidra throws at you. Not every window will be useful in every reverse engineering scenario.

One of the best ways to familiarize yourself with Ghidra’s displays is simply to browse around the various tabbed subwindows that Ghidra populates with data about your binary and also open a few of the other available windows. The efficiency and effectiveness of your reverse engineering sessions will improve as your level of comfort with Ghidra increases.

Ghidra is a very complex tool. In addition to the windows covered in this chapter, you may encounter additional dialogs as you endeavor to master Ghidra. We introduce key dialogs as they become relevant throughout the remainder of the book.

At this point, you should start to feel more comfortable with the Ghidra interface and the CodeBrowser desktop. In the next chapter, we begin to focus on the many ways that you can manipulate a disassembly with Ghidra to enhance your understanding of its behavior and to generally make your life easier.

6

MAKING SENSE OF A DISASSEMBLY



In this chapter, we introduce the foundational skills that will help you better understand the Ghidra disassembly. We start with basic navigational techniques that allow you to move through the assembly and examine the artifacts you encounter.

As you navigate from function to function, you will find that you need to decode each function's prototype by using only clues available in the disassembly. Accordingly, we will discuss techniques for understanding how many parameters a function receives and how we might decode the data types of each parameter we encounter.

Since much of the work that a function performs is associated with local variables maintained by the function, we will also discuss how functions use the stack for local variable storage and how you can, with Ghidra's help, understand exactly how a function makes use of any stack space it may reserve for itself. Whether you find yourself debugging code, analyzing malware, or developing exploits, understanding how to decode a function's stack-allocated variables is an essential skill for understanding the behavior of any program.

In addition, we will look at the options Ghidra provides for searching and how that can contribute to understanding the disassembly.

Disassembly Navigation

Static disassembly listings, such as those provided by tools like `objdump`, offer no inherent navigational capability other than the ability to scroll up and down the listing. Even with the best text editors offering an integrated, `grep`-style search, such *dead listings* are very difficult to navigate.

Ghidra, on the other hand, provides exceptional navigational features. In addition to offering the fairly standard search features you are probably accustomed to from your use of text editors or word processors, Ghidra develops and displays a comprehensive list of cross-references that behave like web page hyperlinks. The end result is that, in most cases, navigating to locations of interest requires nothing more than a double-click.

Names and Labels

When a program is disassembled, every location in the program is assigned a virtual address. As a result, we can navigate to any location within a program by providing its virtual address.

Unfortunately for us, maintaining a catalog of addresses in our heads is not a trivial task. This fact motivated early programmers to assign symbolic names to program locations that they wished to reference. The assignment of symbolic names to program addresses was not unlike the assignment of mnemonic instruction names to program opcodes; programs became easier to read and write as identifiers became easier to remember.

Ghidra continues this tradition by creating labels for virtual addresses and allowing the user to modify and expand the set of labels. We have already seen the use of names in relation to the Symbol Tree window. Recall that clicking a name caused the Listing view to jump to the referenced location. While there are usage differences between the terms *name* and *label* (for example, functions have names and appear in a separate branch of the Ghidra Symbol Tree from labels), in a navigational context the terms are largely interchangeable because both represent navigational targets.

Ghidra generates symbolic names during the auto analysis phase by using an existing name from the binary (if available) or by automatically generating a name based on how a location is referenced within the binary. In addition to its symbolic purpose, any label displayed in the disassembly window is a potential navigation target similar to a hyperlink on a web page. The two major differences between these labels and standard hyperlinks are that the labels are not highlighted in any way to indicate that they can be followed, and that Ghidra generally requires a double-click to follow them rather than the single-click required by a traditional hyperlink.

YOU ARE INVITED TO THE NAMING CONVENTION!

Ghidra provides the user with a lot of flexibility when assigning labels, but certain patterns have a special meaning and are reserved for Ghidra. These include the following prefixes when they are followed by an underscore and an address: EXT, FUN, SUB, LAB, DAT, OFF, and UNK. When you create a label, avoid these patterns. In addition, spaces and nonprintable characters are not allowed in labels. On the plus side, labels can be up to 2,000 characters long. Count carefully if you think you are in danger of exceeding that limit!

Navigation in Ghidra

In the listing shown in Figure 6-1, each of the symbols indicated by a solid arrow represents a named navigational target. Double-clicking any of them in the Listing window will cause Ghidra to relocate the Listing display (and all connected windows) to the selected location.

LAB_0040132b			XREF[1]: 004012ef(j)
0040132b	81 fa cd	CMP	EDX,0xcd
00401331	7f 1b	JG	LAB_0040134eu
00401333	0f 84 86	JZ	LAB_004013bfu
00401339	83 ea 0a	SUB	EDX,0xa
0040133c	74 69	JZ	LAB_004013a7u
0040133e	81 ea c1	SUB	EDX,0xc1
00401344	74 69	JZ	LAB_004013afu
00401346	4a	DEC	EDX
00401347	74 6e	JZ	LAB_004013b7u
LAB_0040134			XREF[1]: 00401331(j)

Figure 6-1: A listing showing navigational targets

For navigational purposes, Ghidra treats two additional display entities as navigational targets. The first is cross-references, indicated by dashed arrows in Figure 6-1. Double-clicking the bottom cross-reference address will jump the display to the referencing location (00401331 in this case). Hovering over any of these navigable objects will display a pop-up that shows the destination code.

The second type of display entity that Ghidra treats as a navigational target is one that uses hexadecimal values. If a displayed sequence of hexadecimal values represents a valid virtual address within the binary, then Ghidra will display its associated virtual address to the right, as shown in Figure 6-2.

00409013	04	??	04h		
00409014	b0	??	80h	?	-> 004037b0
00409015	37	??	37h	7	
00409016	40	??	40h	@	
00409017	00	??	00h		
00409018	00	??	00h		
00409019	0a	??	0Ah		
0040901a	90	??	90h	?	-> 00404590
0040901b	45	??	45h	E	
0040901c	40	??	40h	@	
0040901d	00	??	00h		
0040901e	00	??	00h		
0040901f	0a	??	0Ah		
00409020	a8	??	A8h	?	-> 00404da8

Navigation
targets

Figure 6-2: A listing showing hexadecimal navigational targets

Double-clicking the displayed value will reposition the disassembly window to the associated virtual address. For example, in Figure 6-2, double-clicking any of the values indicated by a solid arrow will jump the display because each is a valid virtual address within this particular binary. Double-clicking any of the other values will have no effect.

The Go To Dialog

When you know the address or name you want to navigate to (for example, when you want to navigate to *main* in an ELF binary to begin your analysis), you could scroll through the listing to look for the address, scroll through the Functions folder in the Symbol Tree window to find the desired name, or use Ghidra's search features (which we discuss later in this chapter) to locate it.

Ultimately, however, the easiest way to get to a known address or name is to use the Go To dialog, shown in Figure 6-3. You can access it via Navigation ► Go To or by using the G hotkey while the disassembly window is active.

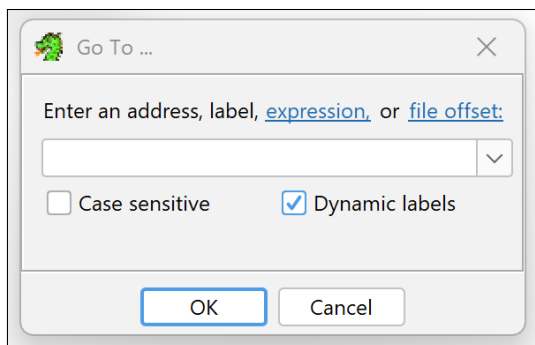


Figure 6-3: The Go To dialog

To navigate to any location in the binary using this dialog, specify a valid address (a case-sensitive symbol name or hex value) and click OK. This will immediately jump the display to the desired location.

The Go To dialog has two newer options: expression and file offset. These options allow you to enter a destination in relative terms, rather than as a label or specific address. Clicking those links in the Go To dialog opens a Ghidra Help window to provide you with more information about the formatting requirements for each of those options. Values entered into the dialog are made available on subsequent use via a drop-down history list, which simplifies returning to previously requested locations.

Navigation History

As a final navigational feature, Ghidra supports forward and backward navigation based on the order in which you navigate the disassembly. Each time you navigate to a new location within a disassembly, Ghidra appends your current location to a history list. You can traverse this list using the left and right arrow icons in the CodeBrowser toolbar.

In the Go To window, the arrow on the right side of the text box opens a picklist that allows you to choose from previous locations you have entered in the Go To dialog. The CodeBrowser toolbar buttons, shown near the top left in Figure 6-4, provide familiar browser-style forward and backward behavior. Each button is associated with a detailed drop-down history list that provides you with instant access to any location in the navigation history without requiring you to retrace your steps through the entire list. Figure 6-4 shows a sample drop-down list associated with the back arrow.

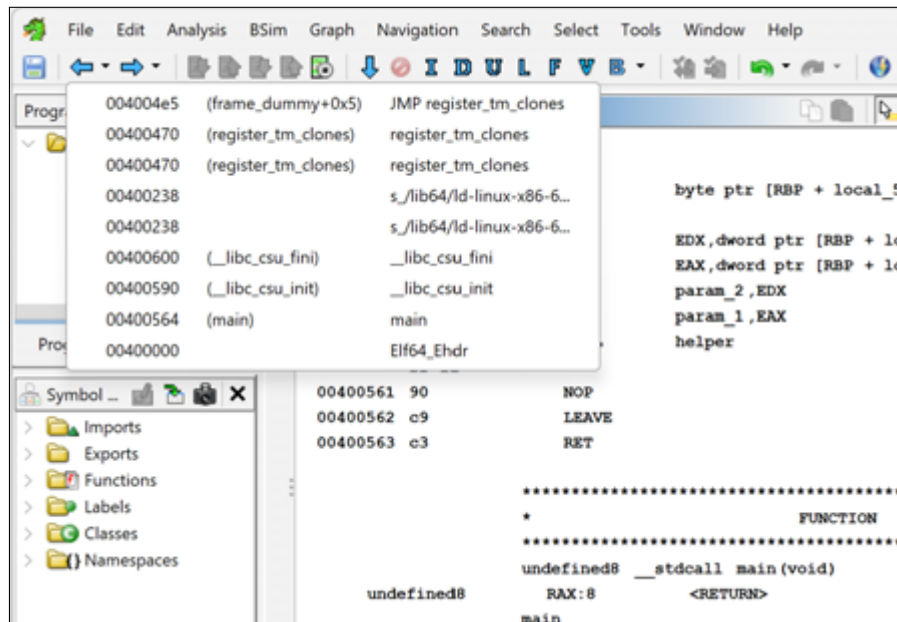


Figure 6-4: Forward and backward navigation arrows with address list

ALT-left arrow (or OPTION-left arrow on a Mac), for backward navigation, is one of the most useful hotkeys you can commit to memory. Backward navigation is extremely handy when you have followed a chain of function calls several levels deep and you decide that you want to navigate back to your original position within the disassembly. ALT-right arrow (OPTION-right arrow on a Mac) moves the disassembly window forward in the history list.

While we now have a much clearer picture regarding navigating a disassembly in Ghidra, we still have not attached meaning to the various destinations we have visited. The next section investigates what makes functions in general, and stack frames in particular, such important navigational targets for a reverse engineer.

Stack Frames

Because Ghidra is a low-level analysis tool, many of its features and displays expect the user to have some familiarity with the low-level details of compiled languages, which focus on the specifics of generating machine language and managing the memory used by a high-level program. In particular, Ghidra pays close attention to the manner in which compilers handle local variable declarations and accesses.

You may have noticed that most function listings begin with a significant number of lines dedicated to local variables. These lines result from the detailed stack analysis that Ghidra performs on each function using its Stack analyzer. This analysis is necessary because compilers place a function's local variables (and in some circumstances, the function's incoming arguments) in blocks of memory allocated on the stack.

To better understand how Ghidra represents program behavior during analysis, it's helpful to first look at how compilers structure function calls, organize local variables, and lay out stack frames. This background will provide important context for interpreting disassembly and decompiled output throughout the reverse engineering process.

While the concepts and examples presented here reflect the most common patterns you are likely to encounter, the reality of working with compiled binaries includes plenty of variation. Real-world code can include compiler-specific behavior, unexpected optimizations, and other edge cases that fall outside the scope of what we cover in this foundational material.

Function Call Mechanics

When a program runs, it needs a way to keep track of where it is, where it's going, and how to get back when a function finishes. This is the job of the call stack. A *call stack* is a structured region of memory that helps manage function calls, arguments, return addresses, and local variables.

Every time a function is called, a stack frame is created. *Stack frames*, also known as *activation records*, are blocks of memory allocated within a program's runtime stack and are dedicated to a specific invocation of a function. Compilers use stack frames to make the allocation and deallocation

of function parameters and local variables transparent to the programmer. You can think of a stack frame as a temporary workspace for a function. It is a small section of the stack where the function stores its return address, local variables, and sometimes parameters. When the function completes, its stack frame is cleaned up and removed, and control returns to the calling function.

Let's consider an example stack use for the Intel x64 architecture, starting with the `CALL` instruction. This special instruction does two things: It pushes the address of the next instruction (where the program should return to) onto the stack, and it jumps to the start of the target function. The called function usually begins with a prologue, a short series of instructions that sets up the stack frame. This often includes saving the base pointer (RBP) and adjusting the stack pointer (RSP) to allocate space for local variables.

NOTE

This is a somewhat simplified (and very common) view of parameter passing, as the number of registers are limited and the parameter passing rules are rather flexible. In reality, there are a limited number of registers, and these registers are of fixed size. If we ventured into passing larger values, we would have to accommodate them in other ways. For example, if a function took an int, a pointer, a float, and another int, they would be passed in RCX, RDX, XMM2L, and R9.

When the function finishes, it uses a function epilogue to restore the previous stack frame. Typically, this includes restoring the saved RBP, deallocating any stack space that was used, and using the `RET` instruction to pop the return address from the stack and jump back to the caller.

This consistent setup and teardown process ensures that each function operates in its own isolated workspace on the stack and that the program can safely move from one function to the next (and back again) without losing track of execution flow. Understanding how this mechanism works lays the groundwork for analyzing real binaries, interpreting disassembly, and eventually manipulating stack frames within tools like Ghidra.

Calling Conventions

Calling conventions are the standardized rules that determine how functions receive parameters, where return values are placed, which registers must be preserved, and who is responsible for cleaning up the stack. You can think of them as an agreement between functions: The caller agrees to pass arguments a certain way, and the callee agrees to handle them accordingly. These conventions can vary across platforms, operating systems, and processor architectures, so understanding the key differences is essential when analyzing binaries.

NOTE

Every calling convention has its own quirks, exceptions, and edge cases that can show up in real binaries. The generalized descriptions and examples we cover here represent what you are most likely to see in practice, but keep in mind that there will always be unusual scenarios that go beyond the scope of this introductory material.

Classic x86 32-Bit Calling Conventions

While 64-bit systems are now widely used, understanding 32-bit x86 calling conventions is valuable when analyzing older software, reverse engineering legacy programs, or investigating binaries built for embedded environments. The three most common conventions found in legacy 32-bit systems are `cdecl`, `stdcall`, and `fastcall`, each with unique characteristics that affect how arguments are passed, how the stack is managed, and how return values are handled.

The `cdecl` convention Short for “C declaration,” this convention is widely used in 32-bit binaries compiled for both Windows and Linux. In this convention, function arguments are pushed onto the stack in right-to-left order. Once the function returns, the calling function is responsible for cleaning up the stack. This convention is flexible and especially useful for functions that take a variable number of arguments, such as `printf`. Return values are typically placed in the `EAX` register.

The `stdcall` convention This convention also passes arguments on the stack in right-to-left order. However, in this case, the called function is the one that cleans up the stack after execution. This approach simplifies the caller’s code and is often used by the Windows API.

The `fastcall` convention This convention is designed to improve performance by passing some of the first arguments directly in CPU registers rather than placing them on the stack. Typically, the first argument is passed in `ECX` and the second in `EDX`. If there are additional arguments, they are pushed onto the stack. This convention reduces memory access during function calls and can speed up performance in certain situations. It is worth noting that different compilers implement variations of `fastcall`, so the exact behavior may vary.

While these conventions are less common in modern 64-bit applications, they remain essential for understanding 32-bit code. Being able to recognize and interpret these calling conventions can help you reconstruct a function’s logic and behavior, especially when working with stripped binaries or performing low-level static analysis.

NOTE

While most contemporary desktop and server environments use 64-bit architectures, 32-bit platforms are still prevalent in certain domains. For example, 32-bit ARM is widely used in embedded systems, mobile devices, and IoT applications, where smaller code size and lower power consumption are priorities. 32-bit ARM calling conventions are included alongside 64-bit models, rather than grouped solely with legacy x86, to reflect their ongoing importance in current software and hardware ecosystems.

Current Calling Conventions

As we move from 32-bit to 64-bit architectures, the conventions used to pass arguments and manage function calls evolve, but they still follow structured patterns. Understanding these patterns is crucial when analyzing compiled binaries on modern systems.

The Microsoft x64 calling convention On 64-bit Windows systems, this calling convention is defined by the Microsoft x64 Application Binary Interface (ABI) and is used consistently across all user-space programs. In this convention, the first four integer arguments are passed using the RCX, RDX, R8, and R9 registers. Any additional arguments are placed on the stack. The calling function always reserves 32 bytes of stack space, known as shadow space, regardless of whether the called function uses all four registers. This space allows the callee to save those register values if needed. Return values are typically placed in the RAX register. The convention also specifies that certain registers, including RBX, RBP, and R12 through R15, must be preserved by the callee. If the callee modifies them, it must restore their original values before returning to the caller.

The System V AMD64 ABI This ABI defines the standard calling convention on Intel 64-bit Linux and Unix systems. In this model, the first six function arguments are passed in the RDI, RSI, RDX, RCX, R8, and R9 registers. If a function needs more than six parameters, the additional arguments are passed on the stack and the stack must be aligned to 16 bytes. Return values typically reside in RAX, although for 128-bit return values, the upper 64 bits of the return value are placed into RDX. The callee is required to preserve the values of certain registers, including RBX, RBP, and R12 through R15. The stack pointer (RSP) must always remain 16-byte aligned, and the calling function is responsible for cleaning up the stack once the function returns.

The ARM Procedure Call Standard (AAPCS) This standard for 32-bit ARM systems defines how arguments and return values are handled. The first four arguments are passed using registers R0 through R3, and any remaining arguments are pushed onto the stack. Return values are placed in R0, and R1 may be used if the return value requires more space. The callee must preserve the values of registers R4 through R11, and the stack must maintain 8-byte alignment throughout execution. This convention is especially common in embedded systems, mobile applications, and Linux-based ARM32 environments.

The AAPCS64 convention On 64-bit ARM platforms such as those found in modern iOS, macOS, and Android devices, this convention defines how functions receive arguments and return values. Although there are exceptions, in general the first eight arguments are passed using registers X0 through X7. If a function receives more than eight arguments, the extras are passed on the stack. Return values are generally placed in the X0 register. The callee must preserve registers X19 through X28. Additionally, the stack is required to remain aligned to a 16-byte boundary. This convention ensures performance and consistency across Apple Silicon and ARM-based Linux systems.

Understanding these calling conventions gives you a map to follow when analyzing function calls in disassembly or when interpreting Ghidra's output. They help you determine where to find arguments, how return values

are passed back, and how the stack is being managed from one function to another.

ARE THEY REALLY GONE?

When we talk about “removing” items from the stack, as well as removing entire stack frames, we mean that the stack pointer is adjusted so it points to data lower on the stack and the removed content is no longer accessible through the POP operation. But until that content is overwritten by a PUSH operation, it is still there. From a programming perspective, that qualifies as removal. From a digital forensics perspective, you just have to look a little harder to find the contents. And from a variable initialization standpoint, it means that any uninitialized local variables within a stack frame may contain stale values that remain in memory from the last use of a particular range of stack bytes.

Other Calling Conventions

Complete coverage of every calling convention would require a book in its own right. Calling conventions are often operating system, language, compiler, or processor specific, so you may have to conduct some research if you encounter code generated by less-common compilers. Still, a few additional situations deserve special mention: optimized code, custom assembly language code, and system calls.

When functions are exported for use by other programmers (as in the case of library functions), it is important that they adhere to well-known calling conventions so that programmers can easily interface to those functions. On the other hand, if a function is intended for internal program use only, the calling convention used by that function need be known only within the program.

In such cases, optimizing compilers may choose to use alternate calling conventions to generate faster code. For example, the use of the `/GL` option with Microsoft C/C++ instructs it to perform “whole program optimization,” which may result in the optimized use of registers across function boundaries, and the use of the `regparm` keyword with GNU `gcc/g++` allows the programmer to dictate that up to three arguments be passed to registers.

When programmers go to the trouble of writing in assembly language, they gain complete control over how parameters will be passed to any functions that they create. Unless they wish to make their functions available to other programmers, assembly language programmers are free to pass parameters in any way they see fit. As a result, take extra care when analyzing custom assembly code, such as obfuscation routines and shellcode.

A *system call* is a specialized form of function invocation that allows a user-mode program to request services from the operating system kernel. This transition from user mode to kernel mode is tightly controlled and varies depending on the platform and processor architecture. The specific mechanism used to initiate a system call depends on both the operating system and the instruction set.

On modern 64-bit Linux systems using x86-64 architecture, system calls are typically made using the `syscall` instruction, with parameters passed in specific registers according to the System V AMD64 ABI. Older 32-bit systems may use legacy mechanisms such as `int 0x80` or `sysenter`. Windows systems, both 32-bit and 64-bit, employ the `syscall` or `sysenter` instructions internally, though these are often abstracted away behind higher-level API calls rather than being used directly by application code.

In general, the calling convention for system calls differs from regular function calls. Parameters are usually passed in a defined set of registers, and the system call number (used to identify the specific service being requested) is typically placed in a dedicated register (such as `EAX` or `RAX`, depending on architecture). When the number of parameters exceeds the number of available registers, additional arguments may be passed via memory. These conventions are part of what makes reverse engineering system call-heavy binaries distinct from analyzing user-space function calls.

Application Binary Interfaces

On any processor, registers are a finite resource, and all functions within a program must share them cooperatively. Yet when some function, `func1`, is executing, its worldview is that it has complete control over all processor registers. When `func1` calls another function, `func2`, that function may wish to adopt this same view and make use of all available processor registers according to its own needs, but if `func2` makes arbitrary changes to the registers, it may destroy values that `func1` depends on.

To address this problem, all compilers follow well-defined rules for register allocation and use. These rules are generally referred to as a platform's *application binary interface (ABI)*. An ABI divides registers into two categories: caller-saved and callee-saved. When one function calls another, the caller needs to save only registers in the caller-saved category to prevent values from being lost. The called function (the callee) must save any registers in the callee-saved category before that function is allowed to use any of those registers for its own purposes. This saving typically takes place as part of the function's prologue sequence, and the caller's saved values are restored within the function's epilogue immediately prior to returning. We refer to caller-saved registers as *clobber* registers because a called function is free to modify their contents without first saving any of them. We call callee-saved registers *no-clobber* registers.

In calling conventions defined by the System V ABI for modern platforms, registers are categorized as either caller-saved or callee-saved. This distinction determines who is responsible for preserving the contents of a register across a function call. For example, in the 64-bit System V AMD64 ABI used on Linux and other Unix-like systems, caller-saved registers include `RAX`, `RCX`, `RDX`, `RSI`, `RDI`, and `R8` through `R11`. Callee-saved registers include `RBX`, `RBP`, `R12` through `R15`, and the stack pointer `RSP`, which must remain aligned.

In practice, compilers tend to favor caller-saved registers for temporary or short-lived values within a function because they do not need to preserve their contents across function boundaries. This helps reduce overhead, as

the responsibility for saving those values lies with the calling function, not the one being called. Understanding this behavior can help explain register usage patterns observed during disassembly and static analysis across different architectures.

A GAMER'S GUIDE TO API AND ABI

An application programming interface (API) is what you, the programmer, interact with when writing code. It's the set of function names, parameters, return type, and other data structures that a library or service makes available. You can think of it as a game controller. It defines the buttons you can press, what each one does, and how you interact with the game, with rules like "press A to jump" or "press X to reload."

An application binary interface (ABI) defines how things happen behind the scenes, including how functions are called, how data is passed around in memory, and how compiled code pieces can interact. You can think of the ABI as the internal wiring and interactions that take place between the controller and the console. It determines how those button presses are translated into actions in the game and on the screen, down to the electrical signals and timing.

The API helps you write code that sends the right commands; the ABI ensures that your compiled code and the system it's running on understand each other. Break the API, and the game won't respond to your button mashing. Break the ABI, and pressing "jump" might actually empty your inventory of hard-won loot or even crash the game entirely.

Local Variable Layout

When a function is called, the stack frame it creates not only holds the return address and any saved registers, it also serves as temporary storage for local variables. Understanding how these variables are laid out in memory is critical for reading assembly code, debugging, and working with reverse engineering tools like Ghidra.

In the x86-64 architecture, most local variables are stored below the base pointer (RBP), using negative offsets. The stack grows downward (toward lower memory addresses), so as a function allocates space for variables, it subtracts from the stack pointer (RSP). To manage this layout, compilers often preserve the current base pointer at the start of a function and then use it as a fixed reference point for accessing both arguments and local variables.

Compilers ensure that the stack maintains proper alignment, often 16-byte alignment on x86-64. This affects how much space is reserved, even if a function only uses a few variables. Additionally, modern compilers may reuse stack space between variables or even optimize some variables into registers, especially when optimization flags are enabled during compilation. That means not every variable declared in source code has a dedicated memory address in the stack.

Some functions do not use a base pointer at all. This is known as frame pointer omission (FPO). In those cases, all local variable accesses are made relative to the stack pointer (RSP), which moves during function execution. This can make reverse engineering more difficult, especially when trying to track stack locations over time.

In reverse engineering, recognizing stack variable layout helps you:

- Identify local buffers, counters, and pointers
- Spot security vulnerabilities (like buffer overflows)
- Reconstruct higher-level logic from low-level memory references

Whether you're working in Ghidra or reading raw disassembly, understanding the structure of a stack frame makes unfamiliar code feel far more approachable.

RBP AND RSP: GUARDIANS OF THE FUNCTION QUEST

A function call is like pausing your main quest (func1) in a video game so you can tackle a side quest (func2) for extra loot or experience. To keep track of exactly where you are and what you have packed, you rely on two magical items: a mystical checkpoint marker (RBP) and an enchanted supply tracker (RSP). These help you remember your main quest's progress and your current gear. At the start of the side quest (func2), you record your main quest's home base (func1's RBP) and the state of your inventory (func1's RSP) so you can pick up right where you left off when you complete your quest.

With these magical items in hand, you enter the side quest (func2) using the same magical tools (RBP and RSP) to track your new progress. While you explore, fight enemies, and collect loot, you rely on a small stash that exists only for this side quest, including what you brought with you and whatever you find along the way. When the side quest ends, you then restore your checkpoint marker (RBP) and supply tracker (RSP) exactly as they were for the main quest (func1). Your main quest adventure continues right where you left off, with nothing missing or out of place.

In the world of the stack, there is always just one base pointer (RBP) and one stack pointer (RSP) in play at any moment. Each side quest (func2) borrows these magical items and keeps your main quest's details safe so nothing is lost when you complete the side quest. By restoring the main quest's values when you return, your main quest adventure picks up smoothly with no missing gear, no lost progress, and no confusion about what happens next.

Ghidra's Stack Frame Analysis

Now that you understand the theory behind function call mechanics, calling conventions, and local variable layout, let's look at some concrete examples of stack frames in action and investigate how Ghidra sets up the stack. These examples will help you visualize how arguments, local variables, return addresses, and saved registers are arranged and accessed during function execution.

Setting Up the Stack

When you look at a function in Ghidra, remember that you are seeing the stack frame as it is set up by the Ghidra analyzers. Since CodeBrowser is designed for static analysis, Ghidra does not actually run the program. Instead, it predicts what the stack should look like for that function based on how the compiler organized it. This means that no matter where you look inside that function, the stack frame you see in the Stack window stays the same. If you were running the program for real, the stack pointer could change as the code pushes or pops data, makes other function calls, or uses space for temporary values. But in static analysis, you work with one consistent snapshot of how Ghidra thinks the stack frame is arranged.

So, when you see local variables and parameters in the Stack view, remember that you are seeing a fixed model of the stack for that function. Ghidra gives you this steady view so you can figure out how data is stored and accessed, even though the real stack could shift at runtime. Knowing this helps you connect what you see in the decompiler and the listing with how memory is laid out, which makes you a stronger reverse engineer.

Example 1: Analyzing a Simple Function Stack Frame

For this first example, let's assume we have a program that calls a single function:

```
void example1() {
    int x = 42;
}
int main() {
    example1();
    return 0;
}
```

To analyze the binary, Ghidra has a lot of work to do, including analyzing the stack. The following shows the Ghidra disassembly for the function `example1()`:

```
void example1(void)
    undefined4    Stack[-0xc]:4    local_c
00401106  PUSH      RBP
00401107  MOV       RBP, RSP
0040110a  MOV       dword ptr [RBP + local_c], 42
00401111  NOP
00401112  POP       RBP
00401113  RET
```

Although the program never uses the variable `x` after assignment and the value does not appear in the decompiled output, the compiler still produces instructions that allocate stack space and store a value. We see the local variable at the top of the listing, named `local_c`. Ghidra has also provided an associated address in the stack, `Stack[-0xc]:4`.

You might have some questions about what Ghidra has done here, and why. We'll discuss naming conventions in the next chapter, but a short explanation of why Ghidra named the variable `local_c` is that it names function variables by appending their address in the stack to an appropriate term. Since this is a local variable, its name is `local_c`.

A logical next question would be, "Why is this variable stored in this stack location?" This decision is the result of the Ghidra analysis process. After analyzing the function, Ghidra has determined that the function prologue will need space to do its job, so it has selected the next available location for the variable (relative to the stack pointer value when the function starts).

Let's look at the common x64 function prologue and determine why it needs space:

00401106	PUSH	RBP
00401107	MOV	RBP, RSP

The first instruction saves the current base pointer (RBP) onto the stack. This is standard prologue behavior for establishing a new stack frame. It preserves the caller's frame pointer so the frame pointer can be restored when the function exits. This will allow the current function to safely use RBP as its own frame pointer. The second instruction, which is also part of the common x64 function prologue, establishes the current value of RSP as the new RBP, defining the base of this function's stack frame. The seemingly empty space that Ghidra left in the stack was to provide space for these values.

Now let's take a look at the actual work that the function needs to do, contained in the following instruction:

0040110a	MOV	dword ptr [RBP + local_c], 42

This instruction stores the value 42 in 4 bytes at the specified location, `RBP + local_c`. (The offset `local_c` is negative, which means it's stored below the new base pointer.) Although Ghidra's Stack Editor window will not reflect the changes during static analysis, we know that the stack would have the following contents at this point when executing:

- The return address of the caller function
- The base address of the caller function
- The callee function's local variable (in this case, `int x` with a value of 42)

If we continue with the function, we see the following sequence:

00401111	NOP	
00401112	POP	RBP
00401113	RET	

The `POP/RET` sequence is a common x64 function epilogue that performs cleanup and returns control to the caller.

NOTE

The default appearance of a disassembly can differ between Ghidra versions. If your view does not exactly match the examples shown here, it could be due to version-specific defaults. It's also possible we tuned some listing contents to make it easier to demonstrate a particular point. In the next chapter, you will learn how to change aspects of your Listing and Decompiler windows to achieve the same results.

Example 2: Analyzing a Stack Frame with Parameters and a Return Value

In this next example, our function is a little more complex and actually accomplishes something:

```
int example2(int a, int b) {
    int sum = a + b;
    return sum;
}
int main() {
    int result = example2(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

The function `example2()` takes two integer parameters, adds them, stores the result in a local variable, and returns it. The function is compiled without optimizations, so we can clearly observe how arguments and local variables are handled in the stack frame.

Before we look at the disassembly, let's take a quick peek at what we see in Ghidra's Decompiler window:

```
int example2(int a, int b)
{
    return b + a;
}
```

Despite allocating space for a local variable (`sum`), the decompiler simplifies the function down to its essential behavior: returning the sum of `a + b`. Still, the disassembly and stack editor give us more insight into what's actually happening behind the scenes. The following shows the Ghidra disassembly for this example:

```
int example2(int a, int b)

    int          EAX:4          <RETURN>
    int          EDI:4          a
    int          ESI:4          b
    undefined4   Stack[-0xc]:4  local_c
    undefined4   Stack[-0x1c]:4 local_1c
    undefined4   Stack[-0x20]:4 local_20
00401126  PUSH    RBP
00401127  MOV     RBP, RSP
0040112a  MOV     dword ptr [RBP + local_1c], a
```

```

0040112d  MOV     dword ptr [RBP + local_20], b
00401130  MOV     EDX, dword ptr [RBP + local_1c]
00401133  MOV     EAX, dword ptr [RBP + local_20]
00401136  ADD     EAX, EDX
00401138  MOV     dword ptr [RBP + local_c], EAX
0040113b  MOV     EAX, dword ptr [RBP + local_c]
0040113e  POP     RBP
0040113f  RET

```

Let's walk through the disassembly for this example. We'll skip the function prologues and move straight to the subsequent instructions:

```

0040112a  MOV     dword ptr [RBP + local_1c], a
0040112d  MOV     dword ptr [RBP + local_20], b

```

The function moves the two incoming arguments (a and b) into stack-local copies. Even though the calling convention passes them in registers (EDI and ESI on Linux x64), this function saves them in local stack slots. This is common in nonoptimized code because it simplifies variable access and debugging, even though it is not strictly necessary:

```

00401130  MOV     EDX, dword ptr [RBP + local_1c]
00401133  MOV     EAX, dword ptr [RBP + local_20]
00401136  ADD     EAX, EDX

```

Now we see the actual computation of the sum. The saved value of a is loaded into EDX, b is loaded into EAX, and the two are added. The result is left in EAX, which aligns with the convention for returning integers:

```

00401138  MOV     dword ptr [RBP + local_c], EAX

```

Rather than returning the value immediately, the function stores the result in another stack-local variable, local_c, located at RBP + local_c. This extra step may seem redundant, but it reflects how the compiler keeps intermediate variables accessible for debugging or further use, a side effect of compiling without optimization:

```

0040113b  MOV     EAX, dword ptr [RBP + local_c]

```

The result is then retrieved from the stack and placed back into EAX to serve as the return value:

```

0040113e  POP     RBP
0040113f  RET

```

The old RBP is restored, which tears down the local stack frame and returns control to the caller, using the return address from the common x64 function epilogue we saw in the previous example.

This example highlights how even simple arithmetic operations can lead to multistep stack behavior when viewed in low-level terms. By walking

through the disassembly and understanding the corresponding stack layout, we gain insight into compiler decisions, calling conventions, and memory usage. In the next example, you will see how to view (and edit) Ghidra's stack using Ghidra's Stack Editor. This will let us visualize and interact directly with the stack, setting the stage for deeper analysis in more complex functions or obfuscated binaries.

Representing Stack Frame Information

Now that the structure of stack frames is familiar, from the mechanics of function calls to calling conventions and the layout of local variables, we can shift focus to how Ghidra represents this information. This section explores how Ghidra identifies, visualizes, and organizes stack frame details as part of its analysis process.

You will see how function parameters and local variables are shown in the Stack window, how offsets such as `[RBP - 8]` are translated into meaningful labels, and how Ghidra interprets function setup and teardown using the Decompiler and Listing views. Ghidra performs much of this analysis automatically, but its output may need refinement when working with optimized or symbol-stripped binaries.

As you progress, you will learn how to adjust Ghidra's interpretation by editing variable names, correcting memory storage locations, and applying accurate data types. These tools allow you to examine a function's stack structure with greater clarity and uncover how a program operates, even when source code is unavailable. To begin, let's take a closer look at how Ghidra presents stack frames during binary analysis.

Visualizing Stack Frames

At this point, you've been introduced to Ghidra's Decompiler and Listing windows, which provide a readable view of functions and low-level instructions. Now, we will use those same views, together with the Stack Editor window, to develop a deeper understanding of how functions allocate and manage local variables and memory. These tools can be used together to reveal how stack frames are structured and how data moves during execution. Let's begin by building on what you already know.

The Listing window shows the raw disassembled instructions, but it does more than just translate bytes into opcodes. It often includes annotations that reference stack variables by name or offset. You might see instructions like the following after Ghidra has analyzed a binary:

```
MOV EAX, [RBP - 0x8]
```

This tells you that the instruction is accessing a local variable stored on the stack. The `[RBP - offset]` notation corresponds directly to stack space allocated during the function prologue. These offsets help you understand where each variable lives in memory during function execution.

The Decompiler takes that stack access and transforms it into a C-like expression. For instance:

```
int sum = a + b;
```

Behind the scenes, this might correspond to multiple instructions that operate on `[RBP - 0x4]` and `[RBP - 0x8]`. The Decompiler window abstracts that for you, but it's still using those stack references to determine the logic. Since the Listing Window and the Decompiler window are linked, clicking content in one moves you to the corresponding content in the other. You can apply the skills from the start of this chapter to rename stack variables directly in either window or change their data types based on your analysis. These edits automatically update the related entries in other windows, including the Stack Editor window.

Selecting Function ► Edit Stack Frame from the right-click context menu opens the Stack Editor window, giving you a structured table view of the stack frame for the current function showing the following:

Variable names Such as `local_c` or renamed identifiers like `x`

Offsets Stack locations such as `[RBP - 0x8]`, relative to the stack pointer before the start of the function

Data types Either Ghidra's inferred types or the ones you assign manually

The Stack Editor window is the best place to get a “map” of the stack layout all in one place. If a function uses a lot of stack space or has confusing variable references, the Stack Editor window can help you see how everything fits together. This is also where you can make the stack easier to understand. You can double-click entries to rename them, apply new data types, or add comments to clarify how the variable is used.

When used together, these three windows offer a complete stack frame analysis experience:

- The Decompiler window gives you readable logic from which you can infer how the stack is used.
- The Listing window shows you the exact instructions and stack offsets in use.
- The Stack Editor window provides a structured layout of local variables and their memory locations.

The ability to navigate these three windows will make you a much more efficient reverse engineer, especially when Ghidra's auto analysis needs a helping hand.

Understanding Stack Offsets

Stack offsets are the key to understanding where a variable lives in memory during a function's execution. When you're reverse engineering, you'll constantly run into references like `[RBP - 0x4]` or `[RSP + 0x20]`. Understanding

how those offsets work and what they mean will make your analysis much stronger, especially when working with stripped or optimized binaries.

Most local variables are stored below the base pointer, which means they use negative offsets like `[RBP - 0x4]`. These represent memory addresses further down the stack as the stack grows downward. Here is an example:

```
MOV DWORD PTR [RBP + local_c], 42
```

This instruction stores the value 42 in a local variable located 12 bytes below the saved base pointer. The instruction would typically appear in the Stack Editor window as follows:

Offset	length	DataType	Name
-0xc	0x4	undefined4	local_c

You can rename `local_c` to something more meaningful in the Stack Editor window by editing the field. When you click the “Apply editor changes” icon, it will update in the Listing and Decompiler windows.

When parameters are passed via the stack (including when there are more parameters than can be stored in registers), they often appear above the base pointer, using positive offsets like `[RBP + 0x10]`.

This happens less frequently in 64-bit binaries due to the use of registers for parameter passing, but when parameters are saved to the stack (either explicitly or during optimization), they show up with positive offsets.

IN FRAME OR OUT OF FRAME

Most stack-based access in well-behaved, unoptimized code is relative to RBP, which serves as a stable frame pointer. Compilers preserve RBP at the beginning of a function and restore it at the end, allowing easy reference to fixed stack offsets throughout the function.

However, optimized code may omit the frame pointer entirely, relying instead on RSP (the current stack pointer) for all stack operations. This is more efficient but harder to follow, especially in disassembly.

In those cases, Ghidra will still try to infer variable locations based on RSP, but the layout might be less obvious and more prone to error. You’ll need to track how RSP changes throughout the function to understand what lives where.

Walking Through a Ghidra Stack Frame

When reverse engineering a binary, you will need to understand how arguments are passed, how local variables are stored, and how values are returned. Even a simple arithmetic function can illustrate the layers of activity happening within a stack frame. In this walkthrough, we’ll analyze a short C program that computes the area of a rectangle and applies an adjustment factor. By tracing the compiled machine code in Ghidra, we’ll observe how the program stores, modifies, and returns values, while exploring the function’s stack frame in detail.

Consider the following C program:

```
int compute_area(int width, int height) {
    int area = width * height;
    int adjustment = 2;
    area += adjustment;
    return area;
}

int main() {
    int result = compute_area(5, 10);
    printf("Computed area: %d\n", result);
    return 0;
}
```

Although this program is simple from a C programming perspective, the compiled binary reveals important patterns in function prologues, stack allocation, and register usage. We'll follow execution from the beginning of `main`, step through the call to `compute_area`, examine how it computes and stores intermediate values, and then return to `main` to finish printing the result. Along the way, we'll pay close attention to stack frame structure, Ghidra's Stack Editor output, and how the decompiler simplifies the underlying operations.

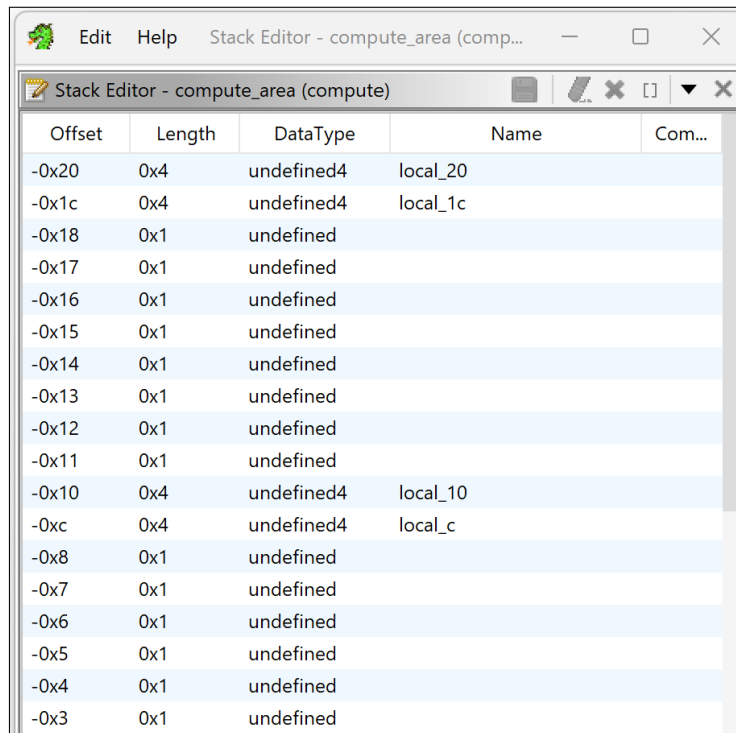
Execution begins in `main`. The following is the annotated disassembly for `main` up to the call to `compute_area`, showing the stack frame interaction:

0040114c	PUSH	EBP	Save caller's base pointer
0040114d	MOV	EBP, ESP	Establish new base pointer
00401150	SUB	ESP, 0x10	Allocate 16 bytes for local vars
00401154	MOV	ESI, 0xa	Second argument: height = 10
00401159	MOV	EDI, 0x5	First argument: width = 5
0040115e	CALL	compute_area	Call function with args in EDI/ESI

Control is now passed to the `compute_area` function, which results in the following disassembly in the Listing window (annotated for clarity):

00401126	PUSH	EBP	Save caller's RBP
00401127	MOV	EBP, ESP	Set new base pointer
0040112a	MOV	dword ptr [EBP + local_1c], param_1	Store width in stack
0040112d	MOV	dword ptr [EBP + local_20], param_2	Store height in stack
00401130	MOV	EAX, dword ptr [EBP + local_1c]	
00401133	IMUL	EAX, dword ptr [EBP + local_20]	Multiply width * height
00401137	MOV	dword ptr [EBP + local_c], EAX	Store result as area
0040113a	MOV	dword ptr [EBP + local_10], 0x2	Store adjustment = 2
00401141	MOV	EAX, dword ptr [EBP + local_10]	
00401144	ADD	dword ptr [EBP + local_c], EAX	Add adjustment to area
00401147	MOV	EAX, dword ptr [EBP + local_c]	Move to return register
0040114a	POP	EBP	Restore caller's RBP
0040114b	RET		Return to `main`

Figure 6-5 shows the resulting Stack Editor window (Window ► Stack Editor from CodeBrowser).



Offset	Length	DataType	Name	Com...
-0x20	0x4	undefined4	local_20	
-0x1c	0x4	undefined4	local_1c	
-0x18	0x1	undefined		
-0x17	0x1	undefined		
-0x16	0x1	undefined		
-0x15	0x1	undefined		
-0x14	0x1	undefined		
-0x13	0x1	undefined		
-0x12	0x1	undefined		
-0x11	0x1	undefined		
-0x10	0x4	undefined4	local_10	
-0xc	0x4	undefined4	local_c	
-0x8	0x1	undefined		
-0x7	0x1	undefined		
-0x6	0x1	undefined		
-0x5	0x1	undefined		
-0x4	0x1	undefined		
-0x3	0x1	undefined		

Figure 6-5: The default Stack Editor window

The following shows some of the content annotated for clarity:

```

-0x20  0x4 undefined4 local_20  (height)
-0x1c  0x4 undefined4 local_1c  (width)
-0x18  0x1 undefined
-0x17  0x1 undefined
-0x16  0x1 undefined
-0x15  0x1 undefined
-0x14  0x1 undefined
-0x13  0x1 undefined
-0x12  0x1 undefined
-0x11  0x1 undefined
-0x10  0x4 undefined4 local_10  (adjustment = 2)
-0xc   0x4 undefined4 local_c   (area)
-0x8   0x1 undefined

```

We can observe that the Stack Editor does not show two important components of the stack frame: that the function prologue stores the return address at `RBP + 0x0` and the saved RBP is at location `RBP + 0x8`. We are free to update this information in the Stack Editor by double-clicking the field we

wish to change. Updating the Stack Editor window with these changes gives us the updated window shown in Figure 6-6.

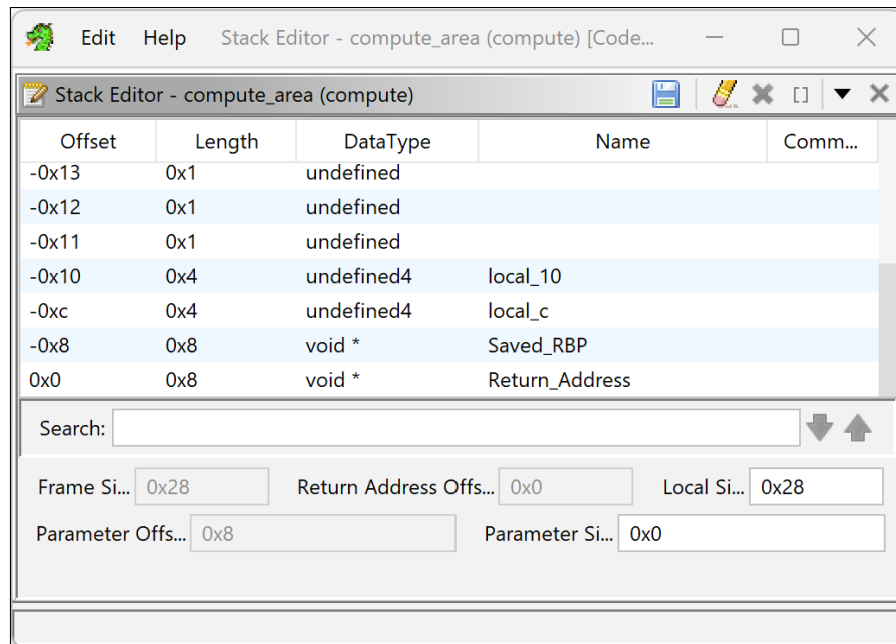


Figure 6-6: The edited Stack Editor window

If we had changed just the Name field in the Stack Editor window, and not the DataType field, the Offset fields between -0x7 and -0x1 would have each displayed undefined. Ghidra helpfully provides you with valid options as you update the DataType field. Since both of the fields we renamed in this example are pointers, choosing the “64-bit void *” option updated the Stack Editor window to accurately reflect the stack space.

Returning to the analysis, the result of the arithmetic (width * height + adjustment) is placed into EAX, ready to be returned to main. When the function returns control to main, it continues with the next sequential instruction after the original call to compute_area. The EAX register holds the return value (computed to be 52), and the (annotated) function continues execution:

00401163	MOV	dword ptr [RBP + local_c],EAX	Save return value to local_c
00401166	MOV	EAX,dword ptr [RBP + local_c]	Reload for printf
00401169	MOV	ESI,EAX	Argument to printf
0040116b	LEA	RAX,[s_Computed_area]	Load string for printf
00401172	MOV	RDI, RAX	Format string for printf
00401175	MOV	EAX,0x0	Clear EAX for vararg calling
0040117a	CALL	<EXTERNAL>::printf	Print the result
0040117f	MOV	EAX, 0	Prepare to return 0 from main
00401184	LEAVE		Restore RBP, reset stack
00401185	RET		Exit `main`

The result (52) is printed to the console. After that, the program prepares to exit cleanly.

In summary, you can make the following observations:

Argument setup `main` uses registers to pass arguments, consistent with the 64-bit calling convention seen in the binary.

Stack frames Both `main` and `compute_area` establish standard RBP-based frames using `PUSH RBP` and `MOV RBP, RSP`.

Stack local variables Ghidra's Stack Editor shows local variables like `local_c`, `local_10`, `local_1c`, and `local_20`, matching the function's temporary variables.

Compiler behavior Despite the simplicity of the function, the compiled program pushes all values to stack variables and then reloads them (likely because optimization was disabled during compilation).

Return Values The return value of `compute_area` is placed in `EAX` and moved to a local variable in `main` before printing.

This section has focused on understanding how compiled programs use stack frames and how Ghidra presents that information. You examined how local variables and parameters are organized in memory, how to interpret stack offsets, and how to refine Ghidra's analysis to make disassembled functions easier to understand. By working with the Stack, Listing, and Decompiler views together, you gained a clearer picture of how data moves within a function.

As analysis continues, there will be times when the information you need is not confined to a single function. You may need to locate references to specific strings, identify how and where functions are called, or track down particular values across a large binary. At that point, being able to search effectively becomes essential.

The next section focuses on how to use Ghidra's search tools to find meaningful patterns in both code and data. These capabilities will help you move beyond isolated functions and begin to analyze programs as a whole.

Searching

As shown at the beginning of the chapter, Ghidra makes it easy to navigate through the disassembly to locate artifacts that you know about and to discover new artifacts. It also designs many of its data displays to summarize specific types of information (names, strings, imports, and so on), making them easy to find as well. However, effective analysis of a disassembly listing often requires the ability to search for new clues to inform the disassembly analysis.

Fortunately for us, Ghidra has a Search menu that allows us to conduct searches to locate items of interest. Figure 6-7 shows the default search menu options.

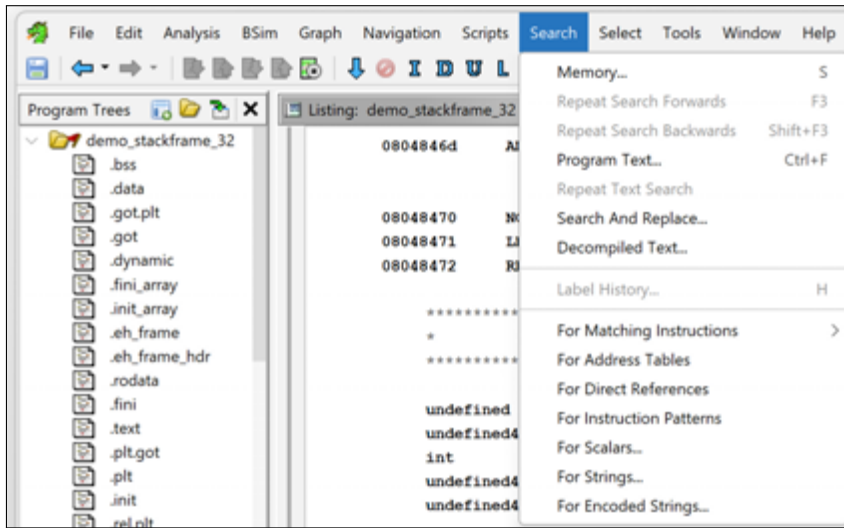


Figure 6-7: The Ghidra Search menu options

In this section, we investigate methods of searching the disassembly by using both text and byte search functionality provided in the CodeBrowser.

I DUB THEE . . .

Search windows are one of the window types within Ghidra that you can rename at will, which will help you keep track of search windows as you experiment. To rename a window, just right-click the title bar and provide a name that is meaningful to you. A handy trick is to include the search string along with a mnemonic to help you remember the settings you have chosen.

Searching Program Text

Ghidra text searches are equivalent to substring searches through the disassembly Listing view. You can initiate one by navigating to Search ► Program Text, which opens the dialog shown in Figure 6-8.

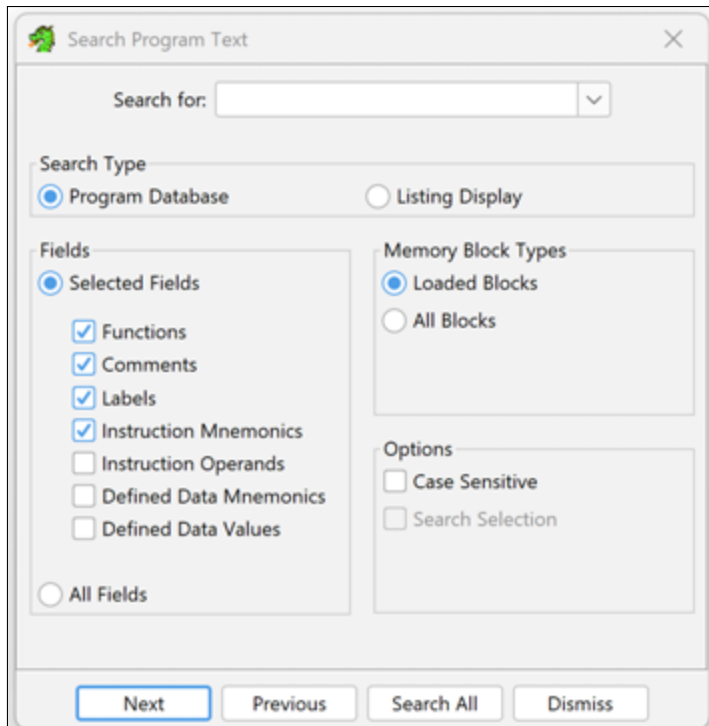


Figure 6-8: The Search Program Text dialog

This dialog offers two search types: one that includes artifacts in the program database, which extends beyond what you see in the CodeBrowser window, and one that is restricted to the listing display within the CodeBrowser. Beyond the search type, several self-explanatory options let you select how and what to search.

To navigate between matches, use the Next and Previous buttons at the bottom of the Search Program Text dialog or select Search All to open the search results in a new window, which allows easy navigation to any match.

Searching Memory

If you need to search for specific binary content, such as a known sequence of bytes, text searches are not the answer. Instead, you need to use Ghidra's memory search functionality. You can initiate a memory search using Search ► Memory or the associated hotkey S. Figure 6-9 shows the Search Memory dialog.

To search for a sequence of hex bytes, specify the search string as a space-separated list of case-insensitive two-digit hex values, such as c9 c3, as shown in Figure 6-9. If you are not sure of the hex sequence, you can use wildcards (* or ?).

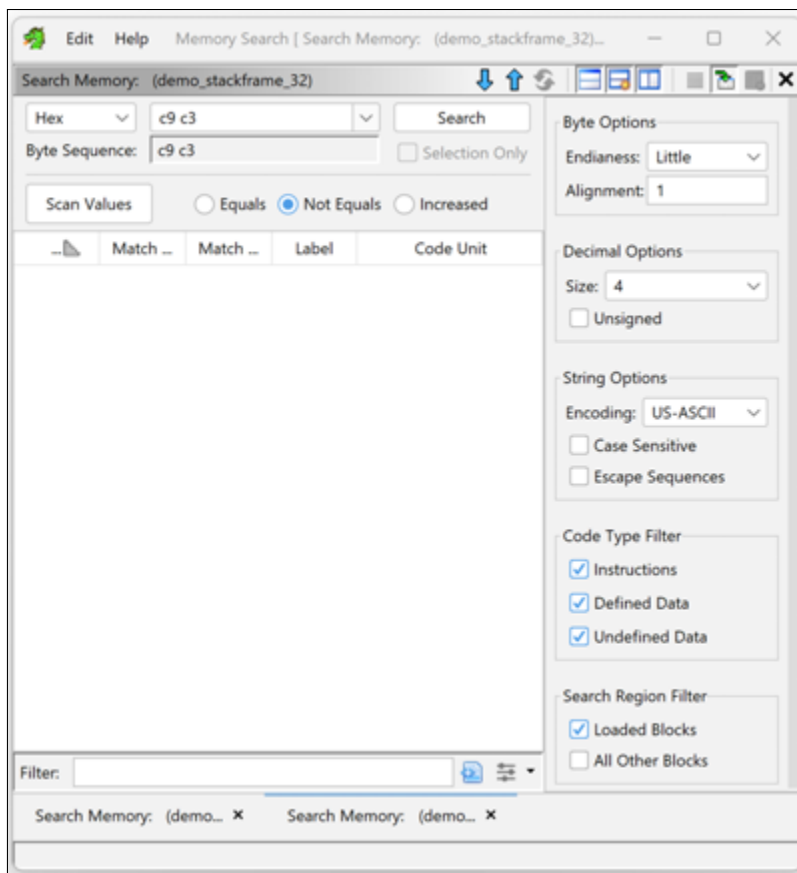


Figure 6-9: The Search Memory dialog

Figure 6-10 shows the Search Memory results for the bytes c9 c3 run with the Search All option. In this dialog, you can sort on any column, rename the window, or apply a filter. This window also offers some right-click options, including the ability to delete rows and manipulate selections.

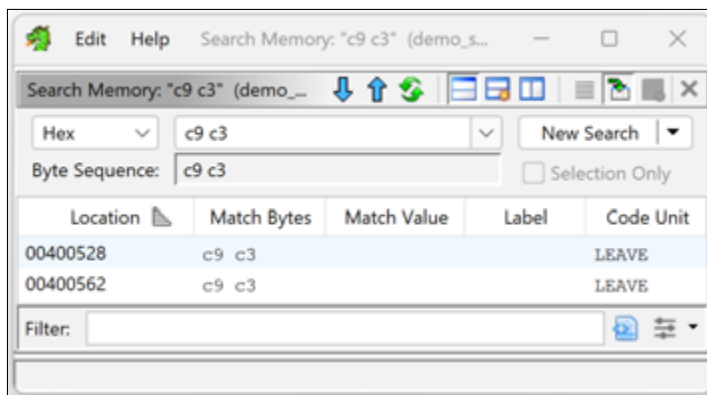


Figure 6-10: The Search Memory results

You can enter search values in string, decimal, and binary formats, as well as in regular expression formats. String, decimal, and binary each provide context-appropriate format options. Regular expressions let you search for a particular pattern, but only in the forward direction because of restrictions on how they are processed. Ghidra uses Java's built-in regular expression syntax, which Ghidra Help describes in significant detail.

Summary

The intent of this chapter was to provide you with the minimum essential skills for effectively interpreting Ghidra's disassembly listings and navigating your way around them. The overwhelming majority of your interactions with Ghidra will involve the operations that we have discussed so far, but the ability to perform basic navigation, understand important disassembly constructs like the stack, and search the disassembly are just the tip of the iceberg for a reverse engineer.

With these skills safely under your belt, the logical next step is learning how to use Ghidra to suit your particular needs. In the next chapter, we begin to look at how to make the most basic changes to a disassembly listing to add new information to the listing based on our understanding of the content and behavior of a binary.

7

REFINING A DISASSEMBLY



After navigation, disassembly modification is Ghidra's next most significant feature. Ghidra offers the ability to easily manipulate disassemblies to add new information or reformat a listing to suit your particular needs, and it will propagate any changes you make to a disassembly to all associated Ghidra views to maintain a consistent picture of your program.

Ghidra automatically handles operations such as context-aware search-and-replace when it makes sense to do so, and it makes trivial work of re-formatting instructions as data, and data as instructions. Perhaps the best feature of all is that almost anything you do can be undone!

Manipulating Names and Labels

At this point, we have encountered two categories of identifiers in Ghidra disassemblies: labels (which are identifiers associated with locations) and names (which are identifiers associated with stack frame variables). For the most part, we will refer to both as *names*, as Ghidra is somewhat loose in this distinction. (If you want to be really precise, labels actually have associated names, addresses, histories, and so on. The name of the label is how we generally reference the label.) We use more specific terms when the distinction makes a critical difference.

To review, stack variable names have one of two prefixes based on whether the variable is a parameter (`param_`) or a local variable (`local_`), and locations are assigned names/labels with helpful prefixes during auto analysis (for example, `LAB_`, `DAT_`, `FUN_`, `EXT_`, `OFF_`, and `UNK_`). In most cases, Ghidra will automatically generate names and labels based on its best guess about the use of the associated variable or address, but you will need to analyze the program yourself to understand the purpose of a location or variable.

As you begin to analyze any program, one of the first and most common ways to manipulate a disassembly listing is to change default names into more meaningful names. Fortunately, Ghidra allows you to easily change any name, and it intelligently propagates name changes throughout the entire program. To open a name-change dialog, select the name by clicking it and then use the L hotkey or the Edit Label option on the right-click context menu. From there, the process for stack variables (names) and named locations (labels) varies, as detailed in the following sections.

I WISH I HADN'T DONE THAT

Part of being good at software reverse engineering is the ability to explore, experiment, and, when necessary, backtrack and retrace your steps. Ghidra's powerful Undo capability allows you the flexibility to undo (and redo) actions during the SRE process. There are multiple ways to access this magical power: the appropriate arrow icons on the top right of the CodeBrowser toolbar, as shown in Figure 7-1 ❶ ❷; Edit ► Undo from the CodeBrowser menu; and the hotkeys CTRL-Z to undo and CTRL-SHIFT-Z to redo.



Figure 7-1: Undo and Redo icons in the CodeBrowser toolbar

Renaming Parameters and Local Variables

Names associated with stack variables are not associated with a specific virtual address. As in most programming languages, such names are restricted to the scope of the function to which a given stack frame belongs. Thus, every function in a program can have its own stack variable named `param_1`, but no function may have more than one variable named `param_1`, as shown in Figure 7-2.

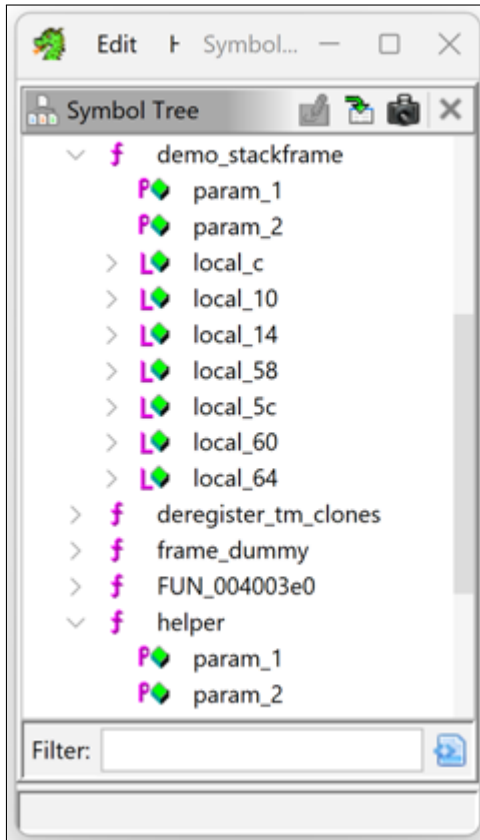


Figure 7-2: The Symbol Tree showing reuse of parameter names (`param_1`)

When you choose to rename a variable in the Listing window, the informative dialog shown in Figure 7-3 will pop up. The type of entity you are changing (variable, function, and so on) appears in the title bar of the window, and the current (about to be changed) name appears in the editable text box and the title bar.

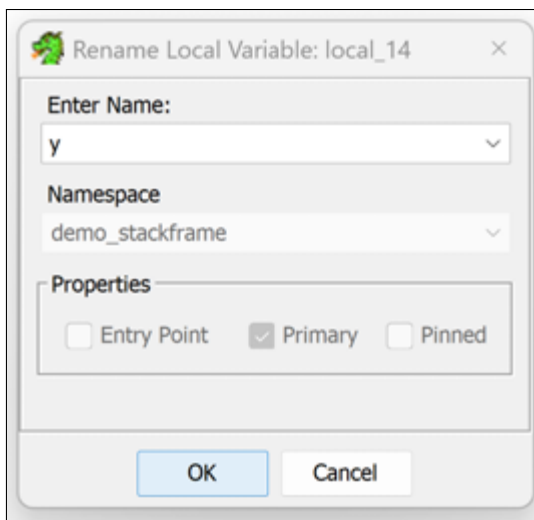


Figure 7-3: Renaming a stack variable (*local_14* to *y*)

Once you have provided a new name, Ghidra changes every occurrence of the old name in the current function. The following listing shows the result of renaming *local_14* to *y* in *demo_stackframe*:

```

*****
*                                     *
*                               FUNCTION                               *
*                                     *
*****
undefined demo_stackframe(undefined param_1, undefined4
    undefined    <UNASSIGNED>    <RETURN>
    undefined    Stack[0x4]:1    param_1
    undefined4   Stack[0x8]:4    param_2
    undefined4   Stack[0xc]:4    param_3
    undefined4   Stack[-0x10]:4   local_10
    ❶ undefined4   Stack[-0x14]:4   y
    undefined4   Stack[-0x18]:4   local_18
    undefined1   Stack[-0x58]:1   local_58
demo_stackframe
08048473 55      PUSH    EBP
08048474 89 e5    MOV     EBP,ESP
08048476 83 ec 58   SUB     ESP,0x58
08048479 8b 45 10    MOV     EAX,dword ptr [EBP + param_3]
0804847c 89 45 f4    MOV     dword ptr [EBP + local_10],EAX
0804847f 8b 45 0c    MOV     EAX,dword ptr [EBP + param_2]
❷ 08048482 89 45 f0    MOV     dword ptr [EBP + y],EAX
08048485 c7 45 ec    MOV     dword ptr [EBP + local_18],0xa
    0a 00 00 00

```

```

0804848c c6 45 ac 41 MOV    byte ptr [EBP + local_58],0x41
08048490 83 ec 08      SUB    ESP,0x8
❸ 08048493 ff 75 f0      PUSH   dword ptr [EBP + y]
08048496 ff 75 ec      PUSH   dword ptr [EBP + local_18]
08048499 e8 88 ff      CALL   helper
      ff ff
0804849e 83 c4 10      ADD    ESP,0x10
080484a1 90            NOP
080484a2 c9            LEAVE
080484a3 c3            RET

```

These changes ❶ ❷ ❸ are also reflected in the Symbol Tree, as shown in Figure 7-4.

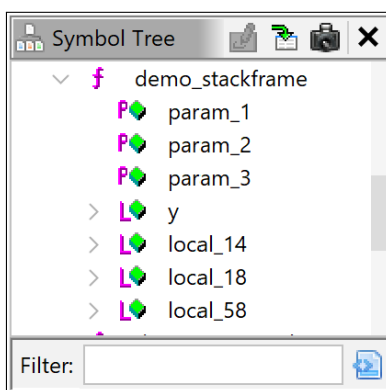


Figure 7-4: The Symbol Tree view of the renamed stack variable, *y*

There are a lot of places where you can change variable names, including the Listing, Symbol Tree, and Decompiler windows; the outcome is the same regardless, but the dialog accessed from the Listing window presents more information. Ghidra enforces all rules associated with naming variables when you use any of these methods.

We changed many of the example parameter names in this book in the Listing window using the dialog shown on the left in Figure 7-5. To change a name in the Symbol Tree, right-click the name, select *Rename* from the context menu, and edit the name right there. In the Decompiler window, use the hotkey *L* or the *Rename Variable* context menu option; the corresponding dialog is shown on the right in Figure 7-5. While the two dialogs provide the same functionality, the right dialog does not include information about the namespace or properties associated with the parameter.

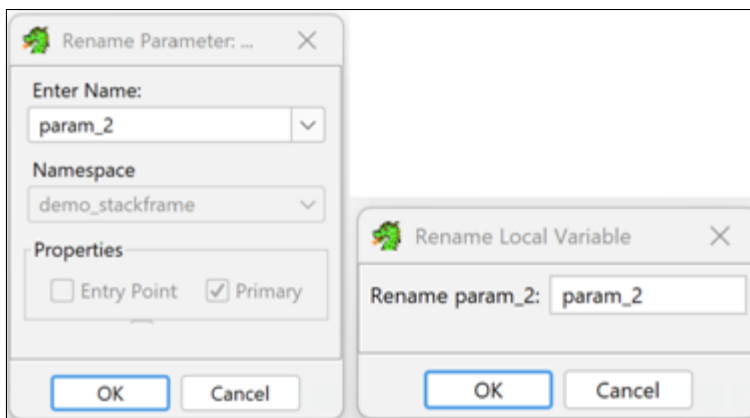


Figure 7-5: Renaming a variable from the Listing window (left) or the Decompiler window (right)

In Ghidra, a *namespace* is simply a named scope. Within a namespace, all symbols are unique. The global namespace contains all symbols within a binary. Function namespaces are nested within the global namespace. Within a function namespace, all variable names and labels are unique. Functions may themselves contain nested namespaces, such as a namespace associated with a switch statement (which allows case labels to be reused in separate namespaces; for example, when a function contains two switch statements that each have a case 10).

THE FORBIDDEN NAMES

Some interesting rules restrict what you can name variables within a function. Here are the most relevant rules for parameters:

- You *can't* use the prefix `param_` followed by an integer in a name, even if the resulting name does not conflict with an existing parameter name.
- You *can* use the prefix `param_` followed by other characters.
- You *can* use the prefix `Param_` followed by an integer, as names are case sensitive (but doing so might not be advisable).
- You *can* restore the name of a parameter to its original Ghidra-assigned name by entering `param_` followed by an integer value. If you use the original integer value, Ghidra will revert the name with no complaints. If you use any integer other than the original value, Ghidra will warn "Rename failed." At this point, clicking Cancel in the Rename Parameter dialog will restore the original name.
- You *can* have two parameters with the names `param_1` (named by Ghidra) and `Param_1` (named by you). Names are case sensitive, but it might not be advisable to reuse them.

Local variables are also case sensitive, and you can use the prefix `local_` with a nonnumeric suffix.

For all types of variables, you *can't* use a variable name that is already used in that scope (for example, in the same function). Your attempt will be rejected with a reason in the dialog.

Finally, if you are thoroughly confused by your labels, you can see the label history for a variable by pressing the hotkey H, choosing Show All History, and entering the current name (or a past name) of the variable into the text box. (This option is also available through Search ► Label History in the main menu.)

Renaming Labels

A label is a default or user-assigned name associated with a location. As with stack variables, you can change their names by opening the name-change dialog with the hotkey L or context option Edit Label. When you change a location's name, you can also change its namespace and properties, as shown in Figure 7-6.

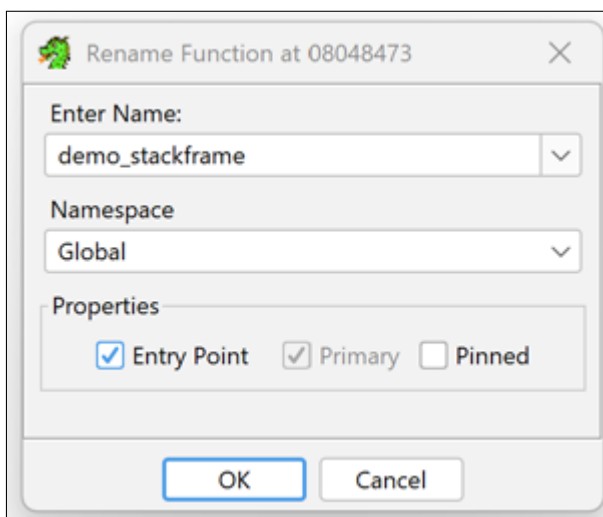


Figure 7-6: Renaming a function

This enhanced dialog shows the entity type and the virtual address of the location in the title bar. Under Properties, you can identify the address as an entry point or pin the address (see “Editing Labels” on page 127). As mentioned in Chapter 6, Ghidra allows names as large as 2,000 characters, so feel free to use meaningful names or even embed a narrative about the address (without any spaces). The Listing window will display only a portion of the name if the length is excessive, but the Decompiler window shows the entire thing.

Adding a New Label

While Ghidra generates many default labels, you can also add new labels and associate them with any address in the listing. These can be used to annotate your disassembly, although in many cases *comments* (discussed later in this chapter) are a more appropriate mechanism for this. To add a new label, open the Add Label dialog (hotkey L), shown in Figure 7-7, for the address associated with the cursor location. The drop-down list for the name includes a list of names you have used recently, and the Namespace drop-down list lets you choose an appropriate label scope.

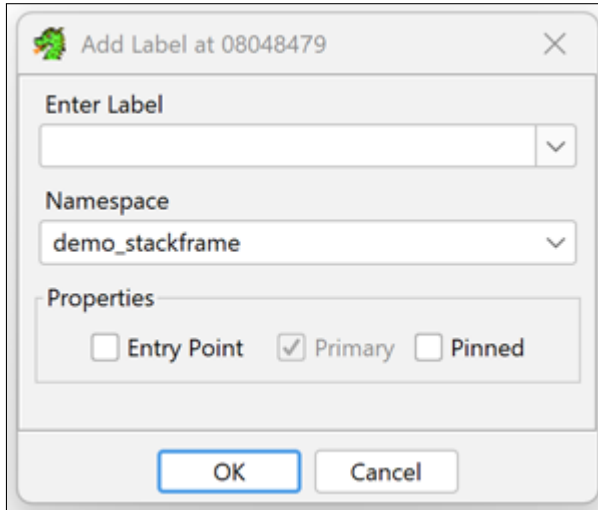


Figure 7-7: The Add Label dialog

You can run into conflicts if you attempt to use one of Ghidra's reserved prefixes when entering a name. If you insist on using a reserved prefix, Ghidra might reject your new label if it believes that a name conflict could arise. This occurs only when Ghidra determines that your suffix looks like an address. (In our experience, this means it contains four or more hex digits.) For example, Ghidra will allow `FUN_zone` and `FUN_123`, but might reject `FUN_12345`. Also, if you attempt to add a label at the same address as a function that has a default label (for example, `FUN_08048473`), Ghidra might rename the function rather than adding a second label at that location.

FUN_ WITH PREFIXES

When Ghidra creates labels during auto analysis, it uses meaningful prefixes followed by an address to let you know what to expect at that location. These prefixes are listed next with very general descriptions. You can find more information about the meaning of each prefix in Ghidra Help.

LAB_*address* Code: an auto-generated label (usually a jump target within a function)

DAT_*address* Data: an auto-generated global variable name

FUN_*address* Function: an auto-generated function name

SUB_*address* Target of a call (or equivalent): probably not a function

EXT_*address* External entry point: probably an exported entry point

OFF_*address* An offcut (inside existing data or code): probably a disassembly error

UNK_*address* Unknown: the purpose of the data here can't be determined

Function labels have the following specific behaviors associated with them:

- If you delete a default function label (such as FUN_08048473) in the Listing window, the FUN_ prefix will be replaced by the SUB_ prefix (in this case, resulting in SUB_08048473).
- Adding a new label to an address that has a default FUN_ label changes the function name rather than creating a new label.
- Labels are case sensitive, so you can use Fun_ or fun_ as a valid prefix if your desire is to create a confusing disassembly.

Editing Labels

To edit a label, use the hotkey L or context menu option Edit Label. Editing a label presents you with the same dialog as adding a label, except that Ghidra initializes the fields in the dialog with the current values of the existing label. Note that editing labels can affect other labels that share the same address, whether or not they share the same namespace. For example, if you identify a label as an entry point, Ghidra will identify all labels associated with that location as entry points.

The Primary checkbox in Figure 7-7 indicates that Ghidra will display this label when it displays the address. By default, this checkbox is disabled for the primary label, so you cannot deselect the primary name. This is necessary to ensure that there is always a name to display. If another label were chosen as the primary, its checkbox would be disabled and checkboxes for other labels at the same address would be enabled.

Although we have, up to now, associated labels with addresses, in reality labels are most commonly associated with content that happens to have an address. For example, the label `main` typically denotes the beginning of the block of code that is the main function in a program. Ghidra assigns an address to this location based on file header information. If we were to relocate the entire content of the binary to a new address range, we would expect that the label `main` would continue to correctly associate with the new address of `main` and its corresponding, unchanged byte content.

However, when a label is *pinned*, the association of the label with the content at its address is severed. If you were to then relocate the binary's content to a new address range, any pinned labels would not move accordingly, but remain fixed to the address that you pinned them to. The most common use of pinned labels is to name reset vectors and memory mapped I/O locations that exist at specific addresses designated by the processor/system designers.

IS IT A BUG OR A FEATURE?

In the course of experimenting with function names, you may notice that Ghidra is perfectly content to allow you to give two functions the same name. This may elicit flashbacks to overloaded functions, which can be distinguished by the parameters they are passed. Ghidra's capability extends beyond this: You can give two functions the exact same name even if this results in duplicate function prototypes within the same namespace. This is possible because a label is not a unique identifier (primary key in the database sense) and thus does not uniquely identify a function, even when considered with its associated parameters. Duplicate names can be used to tag functions (for example, to classify them for further analysis or eliminate them from consideration). Recall that all names are preserved in the function history (hotkey H) and can easily be reverted.

Removing a Label

To remove a label at the cursor, you can use the right-click context option (or hotkey `DELETE`). Be warned that not all labels are removable. First, it is impossible to delete a default, Ghidra-generated label. Second, if you have renamed a default label and later decide to delete the new label, Ghidra will replace the name you are deleting with the originally assigned, default label (as a direct result of the previous statement). The finer details associated with removing labels are discussed in Ghidra Help.

Navigating Labels

Labels are associated with navigable locations, so double-clicking a reference to a label will navigate you to that label. Remember that you can add labels to any location you wish to navigate to in the disassembly. Sometimes, a label (particularly with its 2,000-character allowance) is the quickest way to accomplish this goal. We discuss this feature more thoroughly in Chapter 9.

Comments

Embedding comments into your disassembly and decompiler listings is a particularly useful way to leave notes regarding your progress and discoveries as you analyze a program. Ghidra offers five categories of comments, each suited for a different purpose. We begin by looking at comments that we can add directly to the disassembly in the Listing window.

You can navigate to the Set Comment dialog (shown in Figure 7-8) through the right-click context menu, but the quickest way is to use the hotkey for comments, which is the semicolon (;) key. (This is a logical choice, as the semicolon is the comment indicator in many flavors of assembly.)

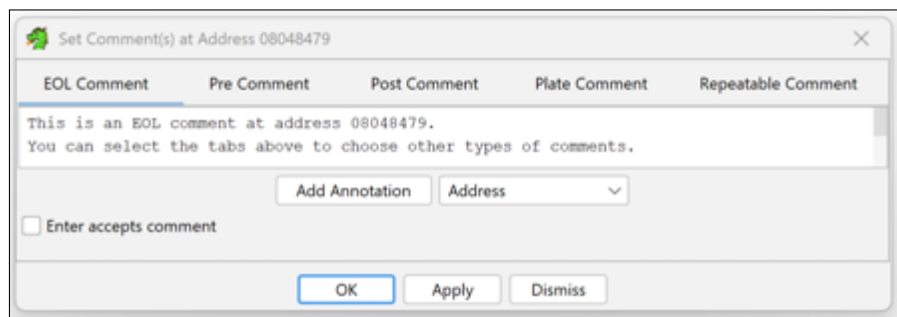


Figure 7-8: The Set Comment dialog

The Set Comment dialog opens in association with a particular address: 08048479 in Figure 7-8, as displayed in the title bar. Any content entered into any one of the five comment category tabs (EOL, Pre, Post, Plate, and Repeatable Comments) is associated with that address.

You can enter one or more lines of content in the text box, including carriage returns, create a comment that is one or more lines long, and then click Apply or OK to post the comment. Apply allows you to see the comment in context and keeps the Set Comment dialog open for continued editing.

To save time when entering short comments, select the “Enter accepts comment” checkbox in the lower left of the dialog, which automatically saves the comments when you hit ENTER. (You can always temporarily deselect the box if you are writing a particularly informative plate comment.)

To delete a comment, clear a comment's text in the Set Comment dialog or use the hotkey `DELETE` when the cursor is on a comment in the Listing window. You can right-click Comments ► Show History for Comment to recall the comments associated with a particular address and re-create them as needed.

THOSE THREE BUTTONS

Of the three buttons at the bottom of the Set Comment dialog (Figure 7-8), the OK and Apply buttons behave as you might expect. Clicking OK closes the dialog and commits your changes. When you click Apply, the listing updates so that you can examine your changes, approve them, or continue editing your comment.

Dismiss, however, is not the same as Cancel, which would exit the dialog with no effect on your listing! The unique term is consistent with the unique behavior. Clicking the Dismiss button exits the window immediately if you have not modified any comments, but lets you decide whether you want to save changes if you did modify comments. Closing the window using the X in the top right exhibits the same behavior. You'll encounter this Dismiss functionality in other places within Ghidra.

End-of-Line Comments

Perhaps the most commonly used type of comment is the *end-of-line (EOL) comment*, placed at the end of existing lines in the Listing window. To add one, open the Set Comment dialog with the semicolon hotkey and select the EOL Comment tab. By default, EOL comments are displayed as blue text and will span multiple lines if you enter multiple lines in the comment text box. Each line will be indented to align at the right side of the disassembly, and existing content will be moved down to make space for the new comments. You can edit your comments at any time by reopening the Set Comment dialog. The quickest method to delete a comment is to click the comment in the Listing window and press `DELETE`.

Ghidra itself adds many EOL comments during auto analysis. For example, when you load a PE file, Ghidra inserts descriptive EOL comments to describe the fields in the `IMAGE_DOS_HEADER` section, including the comment Magic number. Ghidra is able to do this only when it has this information associated with a particular data type. This information is typically contained within type libraries, which are displayed in the Data Type Manager window and discussed in depth in Chapter 8 and Chapter 13. Among all the comment types, EOL comments are the most configurable through the Edit ► Tool Options ► Listing Fields options for each comment type.

Pre and Post Comments

Pre and *post* comments are full-line comments that appear either immediately before or after a given disassembly line. The following listing shows a multi-line pre comment and a truncated single-line post comment, associated with address 08048476:

```
08048473  PUSH   EBP
08048474  MOV     EBP,ESP
          ***** Pre Comment - This is a multiline comment.
          ***** The following statement allocates 88 bytes of local
          ***** variable space in the stack frame.
08048476  SUB     ESP,0x58
          ***** Post Comment - Now that we have allocated the space...
08048479  MOV     EAX,dword ptr [EBP + param_3]
```

Hovering over a truncated comment will display the complete comment. By default, Ghidra displays pre comments in purple and post comments in blue so you can easily associate them with the correct address in the listing.

Plate Comments

Plate comments allow you to group comments for display anywhere in the Listing window. Ghidra centers each plate comment and places it within an asterisk-bounded rectangle. Many of the listings we have examined include a simple plate comment with the word `FUNCTION` inside the bounding box, as shown in Figure 7-9. As you can see on the right side of the figure, no corresponding comment exists in the Decompiler window.

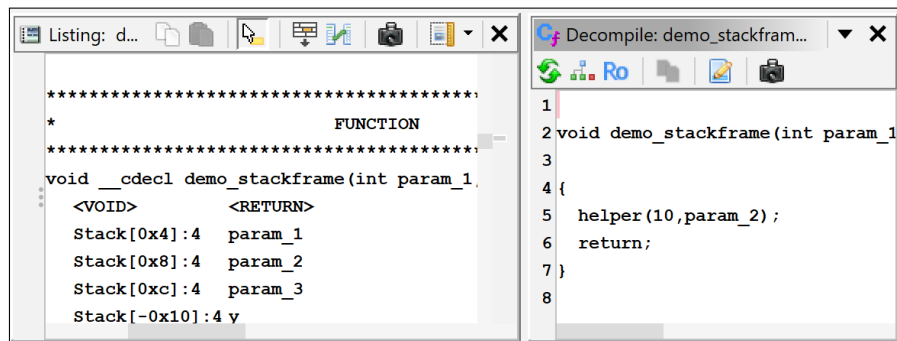


Figure 7-9: A plate comment example

When you open the comment dialog with the first address in the selected function, you have the option to replace this general plate comment with your own, more informative one, as shown in Figure 7-10. In addition to replacing the default plate comment, Ghidra adds your comment as a C-style comment at the top of the Decompiler window. If the cursor were at the top of the Decompiler window when the plate comment was created, the result would have been the same.

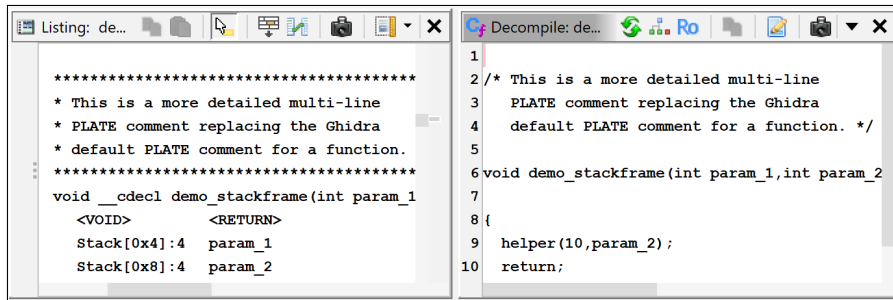


Figure 7-10: Custom plate comment example

NOTE

By default, the Decompiler window displays only plate and pre comments. You can change this behavior using options in **Edit ► Tool Options ► Decompiler ► Display**.

Repeatable Comments

A *repeatable comment* gets entered once but may appear automatically in many locations throughout the disassembly. This behavior is tied to the concept of cross-references, which we discuss in depth in Chapter 9.

Basically, a repeatable comment entered at the target of a cross-reference is echoed at the source of a cross-reference. As a result, echoes of a single repeatable comment may appear at many locations. In a disassembly listing, Ghidra uses orange for repeatable comments and light blue for echoed comments by default, making them easily distinguishable from other types of comments. The following listing demonstrates the use of a repeatable comment:

```

08048432 JGE LAB_08048446 Repeatable comment at 08048446 ❶
08048434 SUB ESP,0xc
08048437 PUSH s_The_second_parameter_is_larger
0804843c CALL puts
08048441 ADD ESP,0x10
08048444 JMP LAB_08048470
LAB_08048446
08048446 MOV EAX,dword ptr [EBP + param_2] Repeatable comment at 08048446 ❷

```

In the listing, a repeatable comment set at 08048446 ❷ is echoed at 08048432 ❶ because the instruction at 08048432 refers to address 08048446 as a jump target. Thus, a cross-reference exists from 08048432 to 08048446.

When an EOL comment and a repeatable comment share the same address, only the EOL comment is visible in the listing, but you can view and edit both comments in the Set Comment dialog. If you delete the EOL comment, the repeatable comment will become visible in the listing.

Parameter and Local Variable Comments

To associate a comment with a stack variable, select the stack variable and use the semicolon hotkey. Figure 7-11 shows the resulting minimal comment window.



Figure 7-11: A stack variable comment

Ghidra will display the comment next to the stack variable in a truncated format similar to an EOL comment. Hovering over the comment will display it in its entirety. The color of the comment matches the default color of the variable type, rather than defaulting to blue, for EOL comments.

Annotations

Ghidra's Set Comment dialog provides the powerful ability to annotate comments with links to programs within your Ghidra project, URLs, addresses, symbols, and executables. For example, the annotation on a plate comment in Figure 7-12 provides a hyperlink to an address in the listing. You can find additional information about the power of annotations in Ghidra Help.

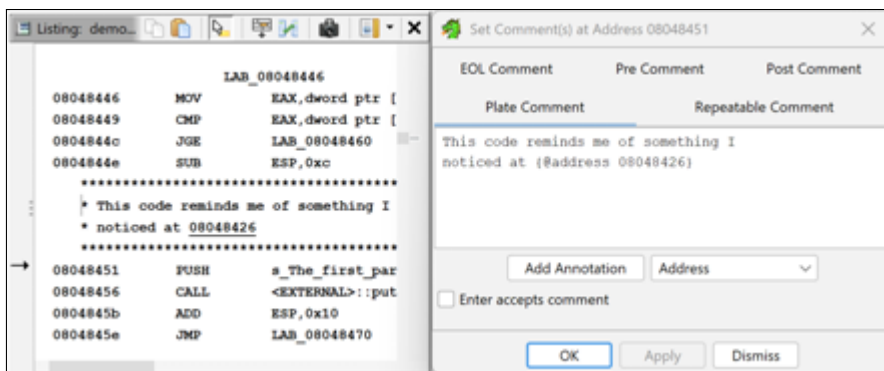


Figure 7-12: An address annotation example

Symbol information in comments will automatically update when symbol names are changed.

When you use an annotation to launch an executable, you can provide an optional list of arguments to supply the executable when it's launched. (Yes, that sounds dangerous to us, too.)

Basic Code Transformations

You may not always be satisfied with the disassembly listings Ghidra generates. As the types of files that you analyze diverge further and further from ordinary executables generated with common compilers, you may need to take control of the disassembly analysis and display processes. This is especially true if you analyze obfuscated code or files that use a custom file format unknown to Ghidra.

Ghidra facilitates many types of code transformations, including these:

- Changing code display options
- Formatting instruction operands
- Manipulating functions
- Converting data into code
- Converting code into data

In general, if a binary is very complex, or if Ghidra is not familiar with the code sequences generated by the compiler used to build the binary, it will encounter more problems during the analysis phase, and you will need to make manual adjustments to the disassembled code.

Changing Code Display Options

Ghidra gives you fine-grained control over the formatting of lines in the Listing window.

The Browser Field Formatter (introduced in Chapter 5) controls the Listing window's layout. Selecting the Browser Field Formatter icon opens a tabbed display of all fields associated with your listing, as displayed in Figure 5-8. You can add, delete, and rearrange fields by using a simple drag-and-drop interface that allows you to immediately observe the changes in your listing.

The tight association between an item in the listing field and in the associated Browser Field Formatter is very useful. Anytime you move the cursor to a new location in the Listing window, the Browser Field Formatter moves the appropriate tab and associated field so that you can immediately identify options associated with a particular item. See “Accessing Special Tool Editing Features” on page 251 for additional discussion of the Browser Field Formatter.

To control the appearance of individual elements within the Listing window, you can select Edit ► Tool Options, as described in Chapter 4. The unique submenus for each field in the Listing window allow you to fine-tune each field to your liking. While the capabilities associated with each field vary, in general you can control display colors, associated default values, configurations, and formats. For example, users who love assembly code and read it in their spare time may choose to adjust the default parameters in the EOL Comments Field area, shown in Figure 7-13, to activate the Show Semicolon at Start of Each Line option so they can view the assembly comments in a familiar format.

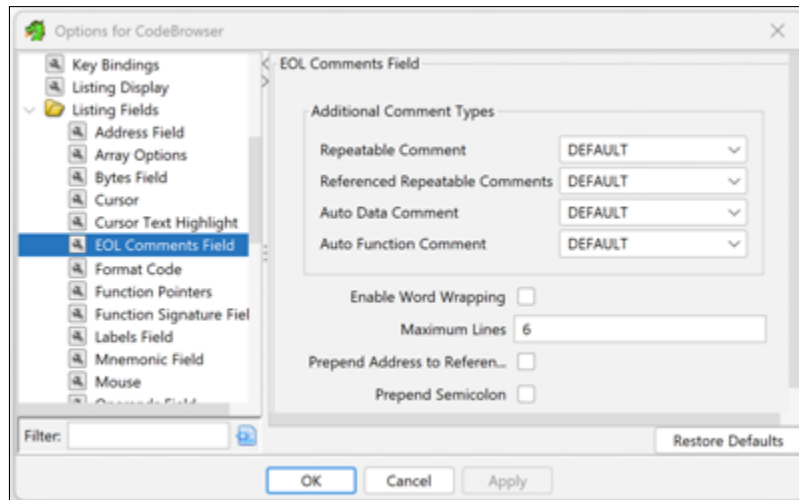


Figure 7-13: The Tool Options menu for the EOL comments field

To color the background of individual lines or larger selections in the Listing window, select the Colors option through the right-click context menu and choose a color. The range of available colors is extensive, and a quick pick option lets you reselect recently used colors. Through the same menu, you can also clear the background color for a line, a selection, or an entire file.

NOTE

Clearing options do not appear if no colors are currently set for the listing.

Formatting Instruction Operands

During the auto analysis process, Ghidra makes many decisions regarding how to format operands associated with each instruction, especially various integer constants used by a wide variety of instruction types. Among other things, these constants can represent relative offsets in jump or call instructions, absolute addresses of global variables, values to be used in arithmetic operations, or programmer-defined constants. To make a disassembly more readable, Ghidra attempts to use symbolic names rather than numbers whenever possible.

In some cases, Ghidra makes its formatting decisions based on the context of the instruction being disassembled (such as a call instruction); in other cases, it bases the decision on the data being used (such as access to a global variable or an offset into a stack frame or structure). Often, Ghidra is not able to discern the exact context in which a constant is used. When this happens, it typically formats the constant as a hexadecimal value.

If you are not one of the few people in the world who eat, sleep, and breathe hex, then you will welcome Ghidra's operand-formatting features. Assume that your disassembly listing contains the following:

08048485	MOV	dword ptr [EBP + local_18],0xA
0804848c	MOV	byte ptr [EBP + local_58],0x41

Right-clicking the hex constant 0x41 opens the context-sensitive menu shown in Figure 7-14. You can reformat the constant in the various numeric representations displayed on the right side of the figure, or as a character constant (since this value also falls within the ASCII printable range). This menu can be very helpful, as you may not realize the many possible representations associated with a given constant. In all cases, the menu displays the exact text that will replace the operand text should you select a particular option.

Convert	>	Char	'A'
Set Equate...	E	Double:	... 3.211426697968103E-322
Fallthrough	>	Float:	9.1084400E-44
References	>	Unsigned Binary:	01000001b
		Unsigned Decimal:	65
		Unsigned Hex:	0x41
		Unsigned Octal:	101o

Figure 7-14: Formatting options for constants

In many cases, programmers use named constants in their source code. Such constants may be the result of `#define` statements (or their equivalent), or they may belong to a set of enumerated constants. Unfortunately, by the time a compiler is finished with the source code, it is no longer possible to determine whether the source used a symbolic constant or a literal, numeric constant. Fortunately, Ghidra maintains a large catalog of named constants associated with many common libraries, such as the C standard library or the Windows API. You can access this catalog via the Set Equate option (hotkey E) in the context-sensitive menu associated with any constant value. For example, selecting this option for the constant `0xa` opens the Set Equate dialog shown in Figure 7-15.

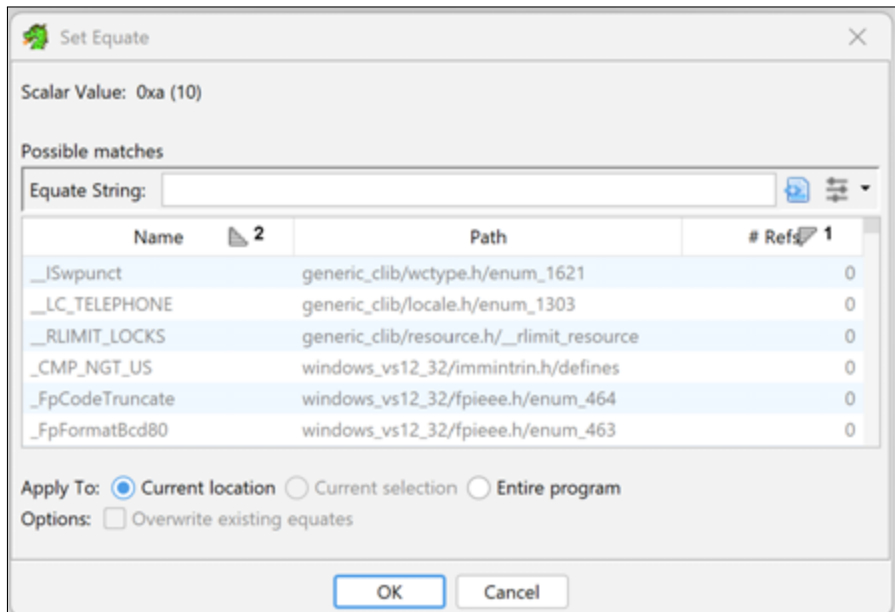


Figure 7-15: The Set Equate dialog

Ghidra populates the dialog using its internal list of constants, filtered according to the value of the constant we are attempting to format. In this case, we can scroll to see all of the constants that Ghidra knows to be equated with the value 0xA. If we determined that code was using the value in conjunction with the creation of an X.25-style network connection, we might select AF_CCITT and end up with the following disassembly line:

```
08048485  MOV     dword ptr [EBP + local_18],AF_CCITT
```

The list of standard constants is useful for determining whether a particular constant may be associated with a known name, and can save you a lot of time spent reading through API documentation in search of potential matches.

Manipulating Functions

Ghidra allows you to manipulate functions in the disassembly (for example, to correct which code Ghidra identifies as belonging to functions, or to change function attributes), which is especially helpful when you disagree with the results of the auto analysis. In some cases, such as when Ghidra fails to locate a call to a function, it may not recognize a function at all because there may be no obvious way to reach it.

In other cases, Ghidra may fail to properly locate the end of a function, requiring you to correct the disassembly. This may occur if the compiler has split the function across several address ranges or when, in the process of optimizing code, a compiler merges common end sequences of two or more functions to save space.

Creating New Functions

To create new functions from existing instructions that do not already belong to a function, you can right-click the first instruction to be included in the new function and select Create Function (or hotkey F). If you selected a range, that range will become the function body. If you did not, Ghidra will follow the control flow to try to determine the bounds of the function body.

Deleting Functions

You can delete existing functions by placing the cursor within the function signature and using the hotkey DELETE. You may wish to delete a function if you believe Ghidra has erred in its auto analysis or if you mistakenly created a function. Note that while the function and its associated attributes will no longer exist, no change occurs to the underlying byte content, so the associated assembly language statements continue to be displayed, and the function can be re-created if desired.

Editing Function Attributes

Ghidra associates several attributes with each function that it recognizes, and you can view these by selecting the Window ► Functions option from the CodeBrowser menu. (While Ghidra displays only five attributes by default, you can add any of 16 additional attributes by right-clicking in a column heading.) To edit the attributes, open the Edit Function dialog from the right-click context menu when the cursor is positioned in the region between a function's plate comment and the last local variable listed before the beginning of the function's disassembled code. Figure 7-16 is an example of the Edit Function dialog.

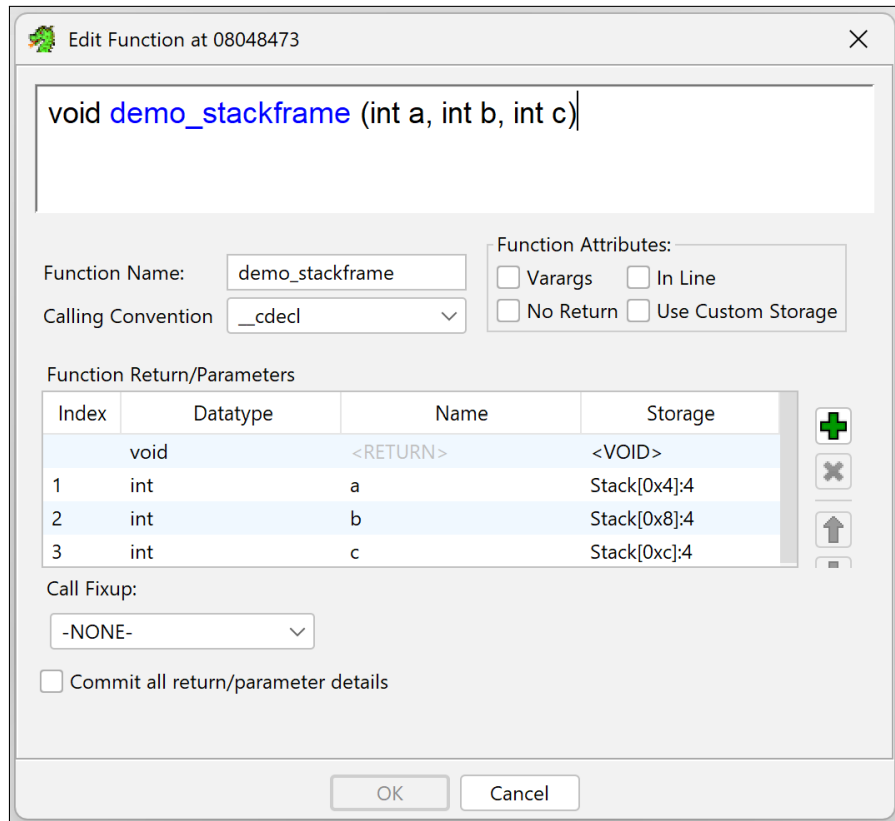


Figure 7-16: The Edit Function dialog

Collectively, these attributes form the function's prototype. As you edit individual attributes, you will notice that Ghidra updates the function's prototype accordingly (as we have done for clarity in many examples). Here is an explanation of each attribute you can modify through this dialog:

Function Signature

The untitled edit box at the top of the dialog displays Ghidra's current understanding of the function's prototype using C-style syntax. You are free to update the prototype here, though once you begin to edit, all of the controls listed below will be disabled until you TAB or ENTER to commit your changes or ESC to discard them.

Function Name

You can modify the name within the text box at the top of the dialog or within the Function Name field.

Function Attributes

You can enable five optional function attributes in this area. The first four attributes, Varargs, In Line, No Return, and Use Custom Storage, are checkboxes and unchecked by default. The fifth optional attribute, Call Fixup, appears in the bottom-left of the dialog, defaults to none, and provides a drop-down menu from which you can choose a value. If you modify any of the function's attributes, Ghidra automatically propagates the function's updated prototype to all locations at which it may be displayed throughout the disassembly.

The Varargs option indicates that a function takes a variable number of arguments (as in the case of `printf`, for example). Varargs is also enabled if you edit the function's parameter list in the text field at the top of Figure 7-16 such that the last argument has an ellipsis (...).

The In Line option has no effect on disassembly analysis other than to include the `inline` keyword in the function's prototype. (Keep in mind that if a function were actually inlined by a compiler, you would not see that function as a distinct entity in a disassembly because its body would have been embedded within the body of the functions that call it.)

The No Return option indicates that a function is known to never return (for example, if it uses `exit` or an opaque predicate to jump to another function). When a function is tagged as No Return, Ghidra will not assume that the bytes following a call to that function are reachable unless it has other evidence to support their reachability, such as a jump instruction targeting those bytes.

The Use Custom Storage option allows you to override Ghidra's analysis of parameter and return value storage locations and sizes.

Calling Convention

The Calling Convention drop-down allows you to modify the calling convention used by the function. Modifying the calling convention may change Ghidra's stack pointer analysis, so it is important to get this information right.

Function Return/Parameters

The Function Return/Parameters area allows you to edit function arguments with guidance. As you modify the data in the four columns associated with the arguments, Ghidra will provide information to help you change things appropriately. For example, attempts to change the Storage for param_1 will result in a message saying Enable 'Use Custom Storage' to allow editing of Parameter and Return Storage. The four icons on the right allow you to add, delete, and navigate through the variables.

Converting Data to Code (and Vice Versa)

During the automatic analysis phase, Ghidra may incorrectly classify data bytes as code bytes and disassemble them into instructions, or it may incorrectly classify code bytes as data bytes and format them as data values. This happens for many reasons, including because some compilers embed data into the code section of programs and because some code bytes are never directly referenced as code, so Ghidra opts not to disassemble them. Obfuscated programs in particular tend to deliberately blur the distinction between code and data. (See Chapter 21.)

To reformat data or code, the first option is to remove its current classification. You can undefine functions, code, or data by right-clicking the item you wish to undefine and selecting Clear Code Bytes (hotkey C). Undefining an item reformats the underlying bytes as a list of raw byte values.

You can undefine large regions by using a click-and-drag operation to select a range of addresses prior to performing the clearing the bytes. As an example, consider this simple function listing:

004013e0	PUSH	EBP
004013e1	MOV	EBP, ESP
004013e3	POP	EBP
004013e4	RET	

Undefining this function would yield the series of uncategorized bytes shown here, which we could reformat in virtually any manner:

004013e0	??	55h	U
004013e1	??	89h	
004013e2	??	E5h	
004013e3	??	5Dh	]
004013e4	??	C3h	

To disassemble a sequence of undefined bytes, right-click the first byte to be disassembled and select Disassemble. This will cause Ghidra to start the recursive descent algorithm at that point. You can convert large regions to code by using click-and-drag to select a range of addresses prior to performing the code-conversion operation.

Converting code to data is a little more complex. You cannot directly convert code to data by using the context menu unless you first undefine the instructions that you wish to convert to data and then format the bytes appropriately. We discuss basic data formatting in the following section.

Basic Data Transformations

Properly formatted data can be as important to understanding a program's behavior as properly formatted code. Ghidra takes information from a variety of sources and uses an algorithmic approach to determine the most appropriate way to format data in a disassembly. Here are some examples:

- It can infer data types and/or sizes from the manner in which registers are used. An instruction that loads a 32-bit register from memory implies that the associated memory location holds a 4-byte data type (though we may not be able to distinguish between a 4-byte integer and a 4-byte pointer).
- It can use function prototypes to assign data types to function parameters. Ghidra maintains a large library of function prototypes for exactly this purpose. It performs analysis on the parameters passed to functions in an attempt to tie a parameter to a memory location. If it can uncover such a relationship, it can apply a data type to the associated memory location. Consider a function whose single parameter is a pointer to a `CRITICAL_SECTION` (a Windows API data type). If Ghidra can determine the address passed in a call to this function, it can flag that address as a `CRITICAL_SECTION` object.
- Analysis of a sequence of bytes can reveal likely data types. This is precisely what happens when a binary is scanned for string content. When long sequences of ASCII characters are encountered, it is not unreasonable to assume that they represent character arrays.

In the next few sections, we discuss some basic transformations you can perform on data in your disassemblies.

Specifying Data Types

Ghidra offers data size and type specifiers. The most commonly encountered specifiers are `byte`, `word`, `dword`, and `qword`, representing 1-, 2-, 4-, and 8-byte data, respectively. You can set or change data types by right-clicking any disassembly line that contains data (that is, not an instruction), choosing Set Data Type from the right-click context menu, and then selecting the desired type from the submenu shown in Figure 7-17.

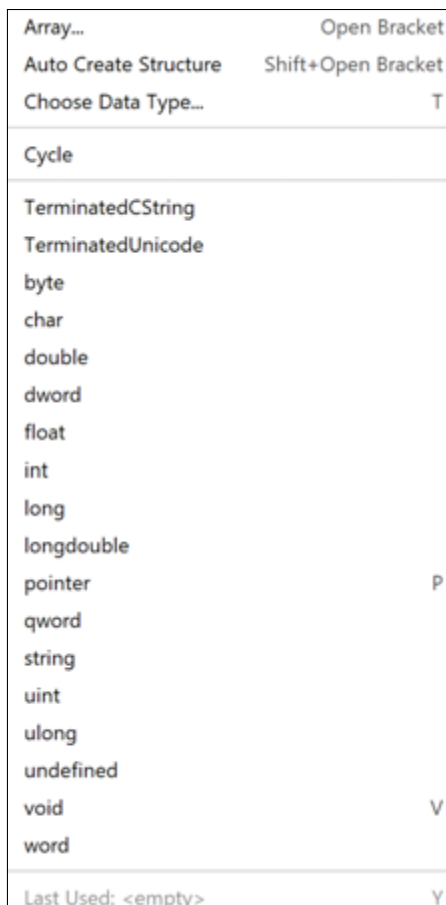


Figure 7-17: The Data submenu

This list allows you to immediately change the formatting and data size of the currently selected item by choosing a data type. The Cycle option allows you to quickly cycle through a group of associated data types, such as numeric, character, and floating point types, as shown in Figure 7-18, along with the associated hotkeys. For example, repeatedly pressing F would cycle you between float and double, as they are the only items in that cycle group.

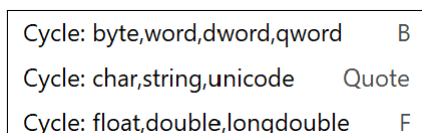


Figure 7-18: Cycle groups

Toggling through data types causes data items to grow, shrink, or remain the same size. Figure 7-19 shows an example involving array dimensioning.

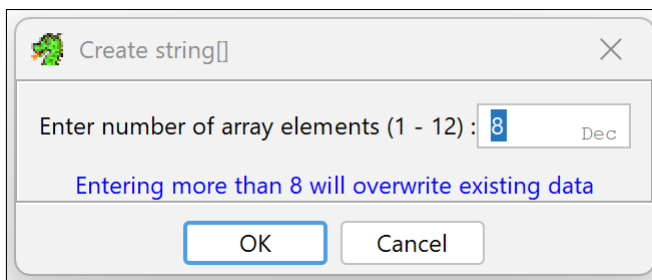


Figure 7-19: An example of an array declaration and warning

If an item's size remains the same, the only observable change is in the way the data is formatted. If you reduce the size of an item, from `ddw` (4 bytes) to `db` (1 byte), for example, any extra bytes (3 in this case) become undefined. If you increase the size of an item, Ghidra will warn you of any conflict and guide you through resolving it.

Working with Strings

Choosing Search ► For Strings brings up the dialog shown in Figure 7-20, where you can set and control the search criteria for a specific string search. While most of the fields in this window are self explanatory, a unique feature of Ghidra is the ability to associate a *word model* with a search. A word model can help determine whether a particular string is considered a word in a given context. We discuss word models in Chapter 13.

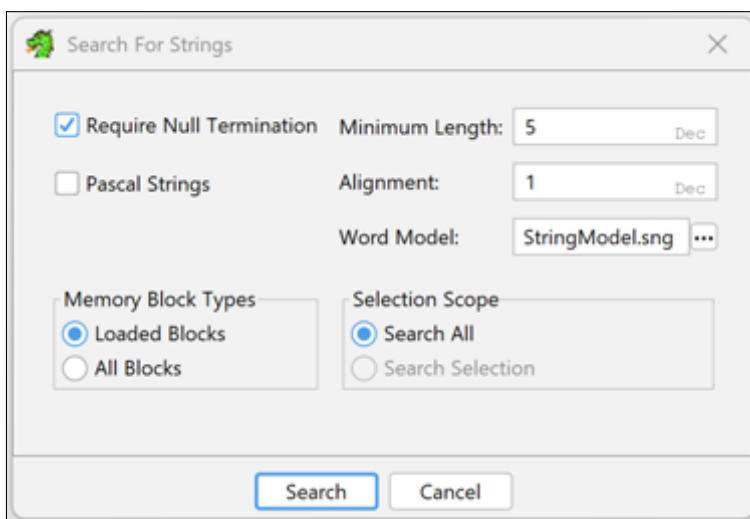


Figure 7-20: The Search For Strings dialog

Once you have performed a search, Ghidra will present the results in a String Search window (Figure 7-21). The window will create new tabs for subsequent searches, and the window title bar will include timestamps for each search so you can easily order them.

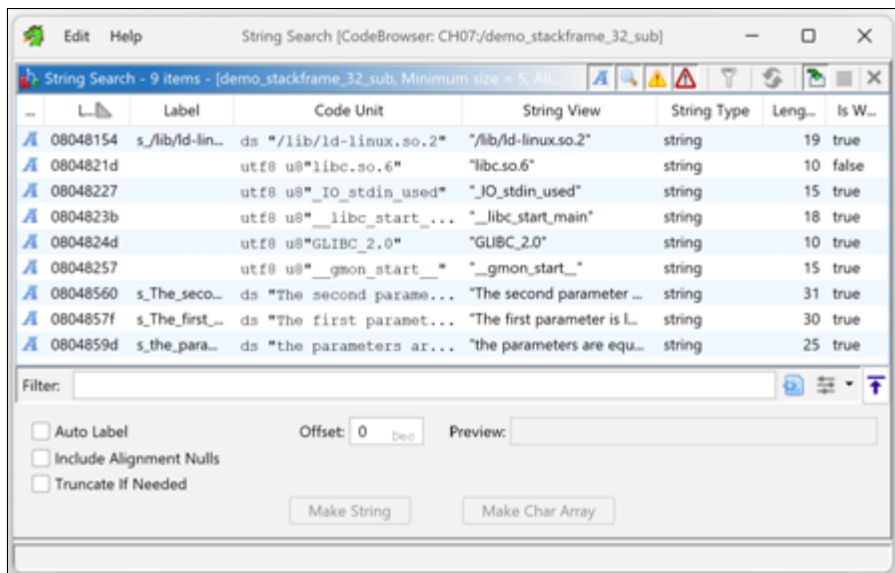


Figure 7-21: The String Search window showing search results

The leftmost column of the String Search window contains icons that indicate the string definition status (from undefined to conflicting). Figure 7-22 shows the meanings for these icons. To show or hide strings in any of the categories, toggle the corresponding icons in the title bar.

Icon	Definition
	The string is already defined (and thus appears in the Defined Strings window). Such strings are usually the target of a cross-reference.
	The string is not defined. The string is not the target of a cross-reference, and the bytes generally appear as individual hex values.
	Part of the string has been defined. Usually this is a string that has a defined string as a substring.
	The string conflicts (overlaps) with something already defined such as existing instructions or data.

Figure 7-22: String toggle icon definitions

Using the icons allows you to easily identify the items in the listing that are not already defined as strings and then make a string or character array from these entries by selecting them and clicking the Make String or Make Char Array button, as appropriate. These newly defined entities will be displayed in the Defined Strings window, discussed in “The Defined Strings Window” on page 84.

Defining Arrays

One of the drawbacks of disassembly listings derived from higher-level languages is that they provide very few clues regarding the size of arrays, and specifying each item in an array on its own line can require a tremendous amount of space.

For example, the following listing shows a sequence of items in a data section. The fact that only the first item in the listing is referenced by any instructions suggests that it may be the first element in an array. Rather than being referenced directly, additional elements within arrays are often referenced using index computations relative to the beginning of the array.

	DAT_004195a4		XREF[1]: main:00411727(W)
004195a4	undefined4	??	
004195a8	??	??	
004195a9	??	??	
004195aa	??	??	
004195ab	??	??	
004195ac	??	??	
004195ad	??	??	
004195ae	??	??	
004195af	??	??	
004195b0	??	??	
004195b1	??	??	
004195b2	??	??	
004195b3	??	??	
004195b4	??	??	
004195b5	??	??	
004195b6	??	??	

Ghidra can group consecutive data definitions into a single array definition. To create an array, select the array's first element and use the Data ► Create Array option in the context menu (hotkey []). You will be prompted for the number of elements in the array or you can accept the default that Ghidra suggests. (If you have selected a range of data, rather than a single value, Ghidra will use your selection as the array bounds.)

By default, Ghidra bases the data type and size associated with the array elements on the data type of the first element in the selection. It presents the array in a collapsed format, but you can expand it to view the individual elements and control the number of elements displayed per line in the Edit ► Tool Options menu of the CodeBrowser window. We discuss arrays more thoroughly in Chapter 8.

Summary

This and the previous chapter encompass the most common operations Ghidra users will need to perform. Disassembly manipulation lets you combine your knowledge with the knowledge imparted by Ghidra during its analysis phase to produce valuable information. As in source code, the effective use of names, detailed comments, and the assignment of data types will not only assist you in remembering what you have analyzed but will also greatly assist others who make use of your work. In the next chapter, we take a look at how to deal with more complex data structures, such as the C struct, and examine some of the low-level details of compiled C++.

8

WORKING WITH DATA TYPES AND DATA STRUCTURES



It's important that you understand the data types and data structures you encounter as you analyze a binary. The data passed into a function will help you reverse engineer the function's signature (the number, type, and sequence of parameters required by the function). Beyond that, the data types and data structures declared and used within functions provide valuable clues about what each function is doing.

In this chapter, we devote significant time to these critical topics. We demonstrate how to recognize data structures used in a disassembly, how to model those structures in Ghidra, and how Ghidra's rich collection of structure layouts can save you time during your analysis. Because C++ classes are a complex extension of C structures, the chapter concludes with a discussion of reverse engineering compiled C++ programs. Let's begin by covering the manipulation and definition of simple and complex data types and structures found within compiled programs.

Making Sense of Data

The simplest way to associate a specific data type with a variable is to observe the use of the variable as a parameter to a function that you already know something about. During its analysis phase, Ghidra makes every effort to annotate data types when they can be deduced based on a variable's use in a function for which Ghidra possesses a prototype.

When it comes to imported library functions, Ghidra will often already know the function's prototype. In such cases, you can easily view the prototype by hovering over the function name in the Listing window or the Symbol Tree window. The low-hanging fruit in understanding the behavior of binary programs lies in cataloging the library functions that the program calls. A C program that calls the `connect` function is creating a network connection. A Windows program that calls `RegOpenKey` is accessing the Windows registry. However, additional analysis is required to gain an understanding of how and why these functions are called.

Discovering how a function is called requires learning about the parameters associated with the function. Let's consider a C program that calls the `connect` function as part of retrieving an HTML page. To call `connect`, the program needs to know the IP address and destination port of the server that hosts the page, information provided by a library function called `getaddrinfo`. Ghidra recognizes this as a library function and updates the Listing window by prefacing the operand with `<EXTERNAL>` and adding an EOL comment to the call to provide us with additional information, as shown here:

```
00100a30 CALL <EXTERNAL>::getaddrinfo    int getaddrinfo(char * __name, c...
```

You can obtain more information about this call in several ways. Hovering over the abbreviated comment to the right of the instruction shows that Ghidra has provided the complete function prototype, which should help you understand the parameters being passed in the function call. Hovering over the function name in the Symbol Tree displays the function prototype and variables in a pop-up window. Alternatively, choosing Edit Function from the right-click menu provides the same information in an editable format, as shown in Figure 8-1.

If you want even more information, you can use the Data Type Manager window to find details about specific parameters, such as the `addrinfo` data type. If you had clicked `getaddrinfo` in the preceding listing, you would have seen that the content shown in Figure 8-1 is replicated within the listing. (This occurs within a `thunk` function, which we discuss in the “Thunk” box on page 211.)

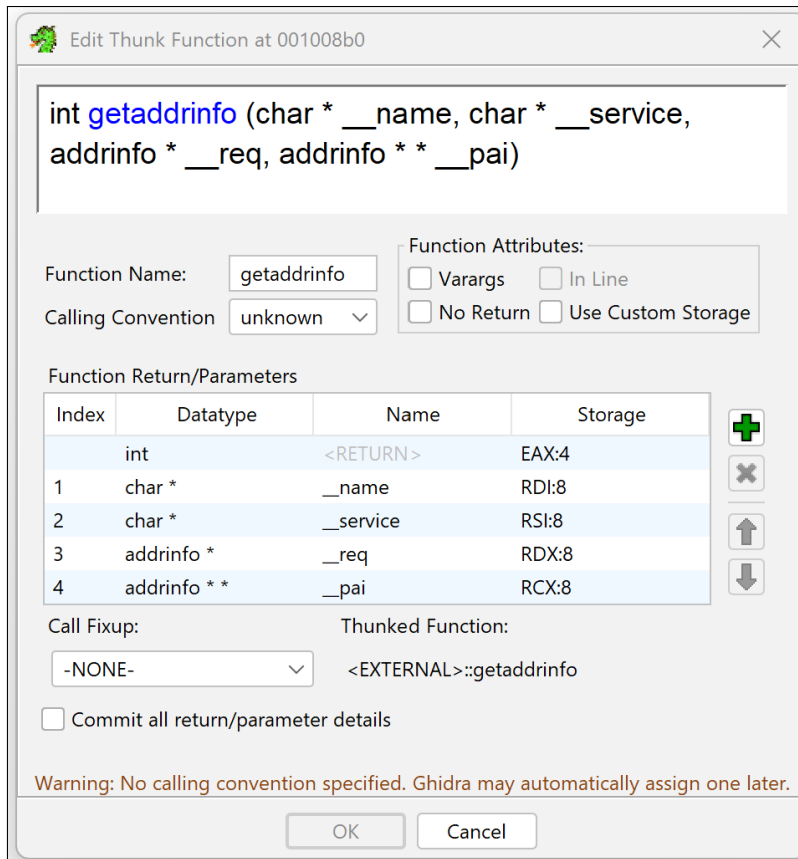


Figure 8-1: The Edit Function window for the `getaddrinfo` function

Finally, you are not required to navigate through the Symbol Tree and Data Type Manager windows to make these observations, as the decompiler has already applied this information in the Decompiler window. There, you will see that Ghidra has already incorporated member names for the fields contained within the `addrinfo` structure by using information from loaded-type libraries. In the following excerpt of code from the decompiler, you can see that the member names `ai_family` and `ai_socktype` help us understand that `local_48` is a structure used when getting the information needed for connect:

```
local_48.ai_family = 2;
local_48.ai_socktype = 1;
local_c = getaddrinfo(param_1,"www",&local_48,&local_18);
```

In this case, the assignment `ai_family` indicates the use of an IPv4 address (2 equates to the symbolic constant `AF_INET`) and `ai_socktype` indicates the use of a stream socket (1 equates to the symbolic constant `SOCK_STREAM`).

When Ghidra has no knowledge of a function's parameter sequence, it should, at a minimum, know the name of the library from which the function was imported. (See the *Imports* folder in the Symbol Tree window.) When this happens, your best option for learning the signature and behavior of the function is to consult any associated man pages or other available API documentation.

Recognizing Data Structure Use

While primitive data types often fit in a processor's registers or instruction operands, accessing the individual data items of composite data types, such as arrays and structures, typically requires more complex instruction sequences. Before we can discuss Ghidra's features for improving the readability of code that uses complex data types, we need to review what that code looks like.

Array Member Access

Arrays are the simplest composite data structures in terms of memory layout. Traditionally, they are contiguous blocks of memory that contain consecutive elements of the same data type.

The size of an array is the product of the number of elements in the array and the size of each element. Using C notation, we can compute the minimum number of bytes consumed by declaring the integer array `int array_demo[100]`; as follows:

```
int bytes = 100 * sizeof(int);    // Or 100 * sizeof(array_demo[0])
```

We can access individual array elements by supplying an index value, either as a variable or as a constant, as shown in these valid array references:

```
❶ array_demo[20] = 15;           // Fixed index into the array
for (int i = 0; i < 100; i++) {
    ❷ array_demo[i] = i;         // Varying index into the array
}
```

Assuming, for the sake of example, that `sizeof(int)` is 4 bytes, the first array reference ❶ accesses the integer value that lies 80 bytes into the array, while the second array reference ❷ accesses integers at offsets 0, 4, 8, . . . , 96 bytes into the array. The offset for the first array access can be computed at compile time as $20 * 4$. In most cases, the offset for the second array access must be computed at runtime because the value of the loop counter, `i`, is not fixed at compile time. Thus, the product $i * 4$ is computed on each pass through the loop to determine the exact offset into the array.

Ultimately, how an array element is accessed depends not only on the type of index used but also on where the array is allocated within the program's memory space.

WHAT IS C REALLY EXPECTING?

For simplicity, we said that C expects an integer index as either a variable or a constant. In reality, any expression that can be evaluated to an integer or interpreted as an integer will do. The general guideline is that anywhere you can use an integer, you can use an expression that evaluates to an integer. Of course, this is not limited to just integers. C is perfectly happy to evaluate any expression you provide and try to make it work for the expected variable type. What if the values are outside the bounds of the array? You have the makings of numerous exploitable vulnerabilities, of course! Values will be read from or written to the resulting out-of-bounds memory region, or the program will simply crash if the computed target address is not valid within the program.

Globally Allocated Arrays

When an array is allocated within the global data area of a program (within the *.data* or *.bss* section, for example), the compiler knows the base address of the array at compile time, enabling it to compute fixed addresses for any array element that is accessed using a fixed index. Consider the following trivial program, which accesses a global array using both fixed and variable indices:

```
int global_array[3];
int main(int argc, char **argv) {
    int idx = atoi(argv[1]);    // Not bounds checked for simplicity
    global_array[0] = 10;
    global_array[1] = 20;
    global_array[2] = 30;
    global_array[idx] = 40;
}
```

If we disassemble a stripped version of the corresponding binary, the main function contains the following code:

```
...
00100657 CALL    atoi
0010065c MOV     dword ptr [RBP + local_c],EAX
❶ 0010065f MOV     dword ptr [DAT_00301018],10
❷ 00100669 MOV     dword ptr [DAT_0030101c],20
❸ 00100673 MOV     dword ptr [DAT_00301020],30
0010067d MOV     EAX,dword ptr [RBP + local_c]
00100680 CDQE
❹ 00100682 LEA     RDX,[RAX*4]
❺ 0010068a LEA     RAX,[DAT_00301018]
❻ 00100691 MOV     dword ptr [RDX + RAX*1]=>DAT_00301018,40
...

```

While this program has only one global variable (the global array), the disassembly lines ❶ ❷ ❸ seem to indicate three global variables: DAT_00301018, DAT_0030101c, and DAT_00301020. However, the LEA instruction ❺ loads the address of a global variable seen previously ❶. In this context, when combined with the computation of an offset (RAX*4) ❹ and scaled memory access ❻, DAT_00301018 is most likely the base address of a global array. The annotated operand =>DAT_00301018 ❻ provides us with the base of the array into which 40 will be written.

WHAT'S A STRIPPED BINARY?

When compilers generate object files, they must include enough information for the linker to be able to do its job. One of the linker's jobs is to resolve references between object files, such as a call to a function whose body resides in a different file, utilizing information from a compiler-generated symbol. In many cases, the linker combines all of the symbol table information from the object files and includes the consolidated information in the resulting executable file. This information is not necessary for the executable to run properly, but it is very useful from a reverse engineering perspective, as Ghidra (and other tools like debuggers) can use the symbol table information to recover function and global variable names and sizes.

Stripping a binary means removing portions of an executable file that are not essential to the runtime operation of the binary. This can be accomplished by using the command line strip utility to post-process an executable, or by providing build options to the compiler and/or linker (-s for gcc/ld) to have them generate a stripped binary themselves. In addition to symbol table information, strip can remove any debugging symbol information, such as local variable names and type information, that were embedded in a binary when it was built. Lacking symbol information, reverse engineering tools must have algorithms for both identifying and naming functions and data.

Based on the names assigned by Ghidra, we know that the global array starts with the 12 bytes beginning at address 00301018. During compilation, the compiler used the fixed indices (0, 1, and 2) to compute the actual addresses of the corresponding elements in the array (00301018, 0030101c, and 00301020), which are referenced using the global variables ❶ ❷ ❸. Based on the values being moved into these locations, we can surmise that we are moving 32-bit integer (dword) values into this array. If we navigate to the associated data in the listing, we see the following content:

```

    DAT_00301018
00301018    ??    ??
00301019    ??    ??
0030101a    ??    ??
0030101b    ??    ??
    DAT_0030101c
0030101c    ??    ??
```

0030101d	??	??
0030101e	??	??
0030101f	??	??
DAT_00301020		
00301020	??	??
00301021	??	??
00301022	??	??
00301023	??	??

The question marks indicate that this array is probably allocated within the program's bss section and that no initialization values are present within the file image.

It is easier to recognize an array in disassembly when it is accessed using variable indices. When constant indices are used to access global arrays, the corresponding array elements appear as global variables in the disassembly. However, the use of variable index values reveals the base address of the array ❹ and the size of the individual elements ❺ because the offset into the array must be computed using the index. (Such scaling operations are required to convert an integer array index from C to a byte offset for the correct array element in assembly language.)

Using Ghidra's type- and array-formatting operations discussed in Chapter 7 (Data ► Create Array), we can format DAT_00301018 as a three-element integer array, yielding disassembly lines with a named array accessed with indices rather than offsets:

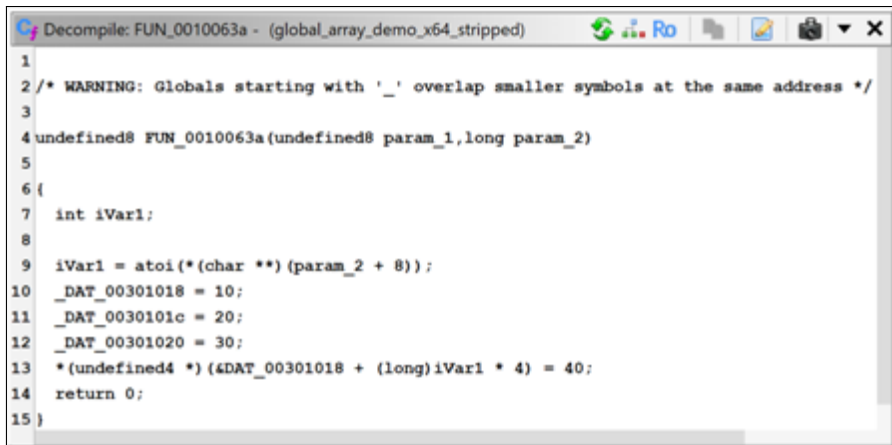
00100660	MOV	dword ptr [INT_ARRAY_00301018],10
0010066a	MOV	dword ptr [INT_ARRAY_00301018[1]],20
00100674	MOV	dword ptr [INT_ARRAY_00301018[2]],30

The default array name assigned by Ghidra, INT_ARRAY_00301018, includes the array type as well as the starting address of the array.

UPDATING SYMBOL INFORMATION IN COMMENTS

As you begin identifying data types, changing symbol names, and so on, you can make sure that the valuable comments you have added to your listing don't become outdated, or challenging to follow, by using comment annotations that update automatically. The `Symbol` annotation option lets you include references to symbols that will be updated as you change the symbols to accurately reflect your findings. (See "Annotations" on page 133.)

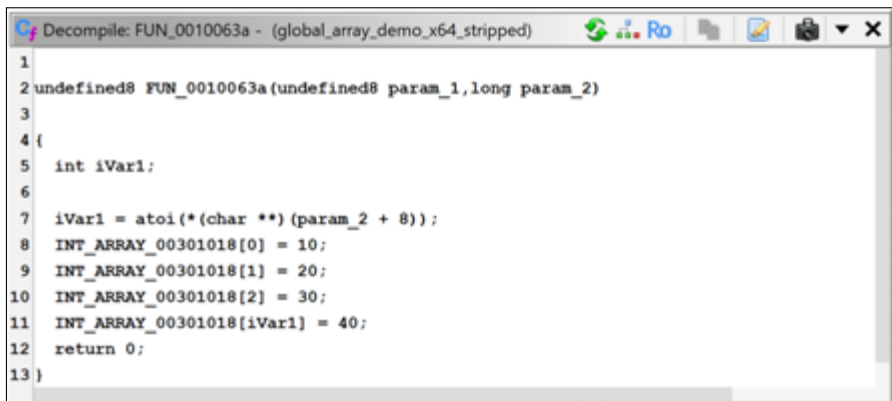
Let's look at the Decompiler window before (Figure 8-2) and after (Figure 8-3) the array has been created. In Figure 8-2, the important warning on line 2 is another clue that you might be looking at an array, and the assignment of integer values supports the assumption that the array type is integer.



```
Decompile: FUN_0010063a - (global_array_demo_x64_stripped)
1
2 /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4 undefined8 FUN_0010063a(undefined8 param_1,long param_2)
5
6 {
7     int iVar1;
8
9     iVar1 = atoi(*(char **) (param_2 + 8));
10    _DAT_00301018 = 10;
11    _DAT_0030101c = 20;
12    _DAT_00301020 = 30;
13    *(undefined4 *) (&DAT_00301018 + (long) iVar1 * 4) = 40;
14    return 0;
15 }
```

Figure 8-2: A Decompiler view indicating a potential array

After the integer array is created, the code in the Decompiler window is updated to use the new array variable, as shown in Figure 8-3.



```
Decompile: FUN_0010063a - (global_array_demo_x64_stripped)
1
2 undefined8 FUN_0010063a(undefined8 param_1,long param_2)
3
4 {
5     int iVar1;
6
7     iVar1 = atoi(*(char **) (param_2 + 8));
8     INT_ARRAY_00301018[0] = 10;
9     INT_ARRAY_00301018[1] = 20;
10    INT_ARRAY_00301018[2] = 30;
11    INT_ARRAY_00301018[iVar1] = 40;
12    return 0;
13 }
```

Figure 8-3: A Decompiler view after declaring an array

Stack-Allocated Arrays

The compiler cannot know the absolute address of an array allocated on the stack as a local variable in a function at compile time, so even accesses that use constant indices require some computation at runtime. Despite the differences, compilers often treat stack-allocated arrays almost identically to globally allocated arrays.

The following program is a variation of the previous example that uses a stack-allocated array rather than a global array:

```
int main(int argc, char **argv) {
    int stack_array[3];
    int idx = atoi(argv[1]);    // Bounds check omitted for simplicity
```

```

stack_array[0] = 10;
stack_array[1] = 20;
stack_array[2] = 30;
stack_array[idx] = 40;
}

```

The address at which `stack_array` will be allocated is unknown at compile time, so the compiler cannot precompute the address of `stack_array[2]` as it did for `global_array[2]`. However, the compiler can compute the relative location of any element within the array. For example, element `stack_array[2]` begins at offset $2 * \text{sizeof}(\text{int})$ from the beginning of the array, and the compiler is well aware of this at compile time. If the compiler elects to allocate `stack_array` at offset `RBP - 0x10` within the stack frame, it can compute the value `RBP - 0x10 + 2 * sizeof(int)`, which reduces to `RBP - 0x8` at compile time and avoids the need for additional arithmetic at runtime to access `stack_array[2]`. This becomes evident in the following listing:

```

int main(void)
    int          EAX:4          <RETURN>
❶ undefined4    Stack[-0xc]:4  local_c
    undefined4    Stack[-0x10]:4 local_10
    undefined4    Stack[-0x14]:4 local_14
    undefined4    Stack[-0x18]:4 local_18
    undefined4    Stack[-0x1c]:4 local_1c
    undefined8    Stack[-0x28]:8 local_28
0010063a  PUSH    RBP
0010063b  MOV     RBP,RSP
0010063e  SUB     RSP,0x20
00100642  MOV ❷ dword ptr [RBP + local_1c],EDI
00100645  MOV    qword ptr [RBP + local_28],RSI
00100649  MOV    RAX,qword ptr [RBP + local_28]
0010064d  ADD    RAX,0x8
00100651  MOV    RAX,qword ptr [RAX]
00100654  MOV    RDI,RAX
00100657  CALL   <EXTERNAL>::atoi
0010065c  MOV ❸ dword ptr [RBP + local_c],EAX
00100661  MOV ❹ dword ptr [RBP + local_18],10
00100664  MOV    dword ptr [RBP + local_14],20
0010066b  MOV    dword ptr [RBP + local_10],30
00100672  MOV    EAX,dword ptr [RBP + local_c]
00100679  CDQE
0010067c  MOV ❺ dword ptr [RBP + RAX*0x4 + -0x10],40
0010067e  MOV    EAX,0x0
00100686  LEAVE
0010068b  RET

```

It is even more difficult to detect this array than the global array. This function appears to have six unrelated variables ❶ (`local_c`, `local_10`, `local_14`, `local_18`, `local_1c`, and `local_28`), rather than an array of three integers and an integer index variable. Two of these locals (`local_1c` and `local_28`) are the function's two parameters, `argc` and `argv`, being saved for later use ❷.

The use of constant index values tends to hide the presence of a stack-allocated array because you see only assignments to separate local variables ❸. Only the multiplication ❹ hints at the existence of an array with individual elements that are 4 bytes each. Let's break down that statement further: `RBP` holds the stack frame base address, `RAX*4` is the array index (converted by `atoi` and stored in `local_c` ❺) multiplied by the size of an array element, and `-0x10` is the offset to the start of the array from `RBP`.

The process used to convert local variables to an array is a little different from the one used to create an array in the data section of the listing. Because the stack structure information is associated with the first address in the function, you cannot select a subset of the stack variables. Instead, place the cursor on the variable at the start of the array, `local_18`, select **Set Data Type** followed by the **Array** option from the right-click context menu, and then specify the number of elements in the array. Ghidra will display a warning message about a possible conflict with the local variables that we are absorbing into the array, as shown in Figure 8-4.

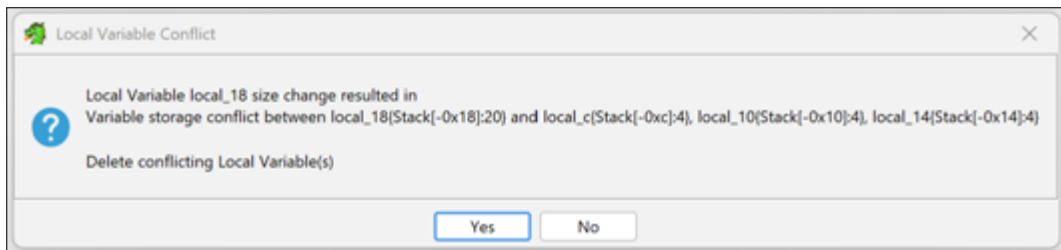


Figure 8-4: A warning about a potential conflict when defining a stack array

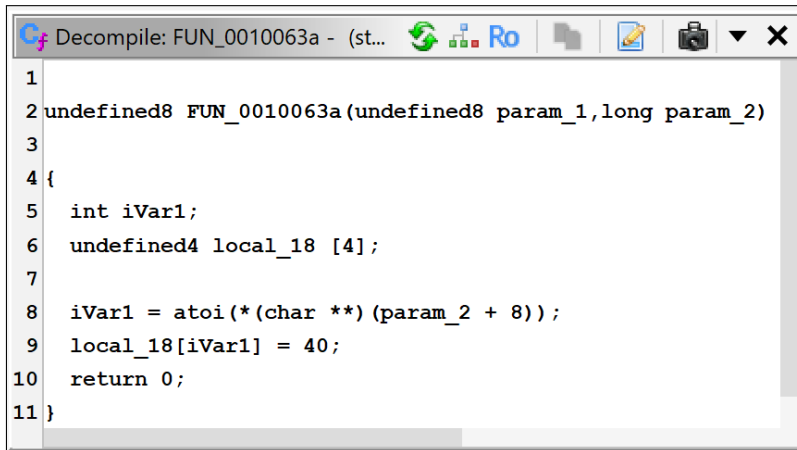
If you proceed despite the potential conflict, you will see the array in the Listing window, as shown here:

```

...
00100664  MOV     dword ptr [RBP + local_18[1]],20
0010066b  MOV     dword ptr [RBP + local_18[2]],30
...

```

Even after the array is defined, the decompiler listing in Figure 8-5 does not resemble the original source code. The decompiler has omitted the static array assignments because it believes they do not contribute to the result of the function.



```
Decompile: FUN_0010063a - (st...  
1  
2 undefined8 FUN_0010063a(undefined8 param_1, long param_2)  
3  
4 {  
5     int iVar1;  
6     undefined4 local_18 [4];  
7  
8     iVar1 = atoi(*(char **) (param_2 + 8));  
9     local_18[iVar1] = 40;  
10    return 0;  
11 }
```

Figure 8-5: The Decompiler view of the function with all stack variables after the array is defined

The call to `atoi` and resulting assignment remain because Ghidra cannot compute the side effects of calling `atoi`, but Ghidra mistakes the saved result of `atoi` as the fourth element of the array (`local_c` in the disassembly is named `iVar1` in the decompiler listing).

Heap-Allocated Arrays

Heap-allocated arrays are those created using a dynamic memory allocation function such as `malloc` (C) or `new` (C++). From the compiler's perspective, the primary way they differ from stack-allocated arrays is that the compiler must generate all references into the array based on the address returned from the memory allocation function. The following C program allocates a small array in the program heap:

```
int main(int argc, char **argv) {  
    int *heap_array = (int*)malloc(3 * sizeof(int));  
    int idx = atoi(argv[1]);          // Bounds check omitted for simplicity  
    heap_array[0] = 10;  
    heap_array[1] = 20;  
    heap_array[2] = 30;  
    heap_array[idx] = 40;  
}
```

The corresponding disassembly is a little more complex than the two previous examples:

```

        undefined main()
            undefined      <UNASSIGNED>      <RETURN>
            undefined8     Stack[-0x10]:8     heap_array
            undefined4     Stack[-0x14]:4     local_14
            undefined4     Stack[-0x1c]:4     local_1c
            undefined8     Stack[-0x28]:8     local_28
0010068a  PUSH    RBP
0010068b  MOV     RBP,RSP
0010068e  SUB     RSP,0x20
00100692  MOV     dword ptr [RBP + local_1c],EDI
00100695  MOV     qword ptr [RBP + local_28],RSI
❶ 00100699  MOV     EDI,0xc
0010069e  CALL    <EXTERNAL>::malloc
❷ 001006a3  MOV     qword ptr [RBP + heap_array],RAX
001006a7  MOV     RAX,qword ptr [RBP + local_28]
001006ab  ADD     RAX,0x8
001006af  MOV     RAX,qword ptr [RAX]
001006b2  MOV     RDI,RAX
001006b5  CALL    <EXTERNAL>::atoi
001006ba  MOV     dword ptr [RBP + local_14],EAX
001006bd  MOV     RAX,qword ptr [RBP + heap_array]
❸ 001006c1  MOV     dword ptr [RAX],10
001006c7  MOV     RAX,qword ptr [RBP + heap_array]
❹ 001006cb  ADD     RAX,0x4
001006cf  MOV     dword ptr [RAX],20
001006d5  MOV     RAX,qword ptr [RBP + heap_array]
❺ 001006d9  ADD     RAX,0x8
001006dd  MOV     dword ptr [RAX],30
001006e3  MOV     EAX,dword ptr [RBP + local_14]
001006e6  CDQE
❻ 001006e8  LEA     RDX,[RAX*0x4]
001006f0  MOV     RAX,qword ptr [RBP + heap_array]
❼ 001006f4  ADD     RAX,RDX
001006f7  MOV     dword ptr [RAX],40
001006fd  MOV     EAX,0x0
00100702  LEAVE
00100703  RET

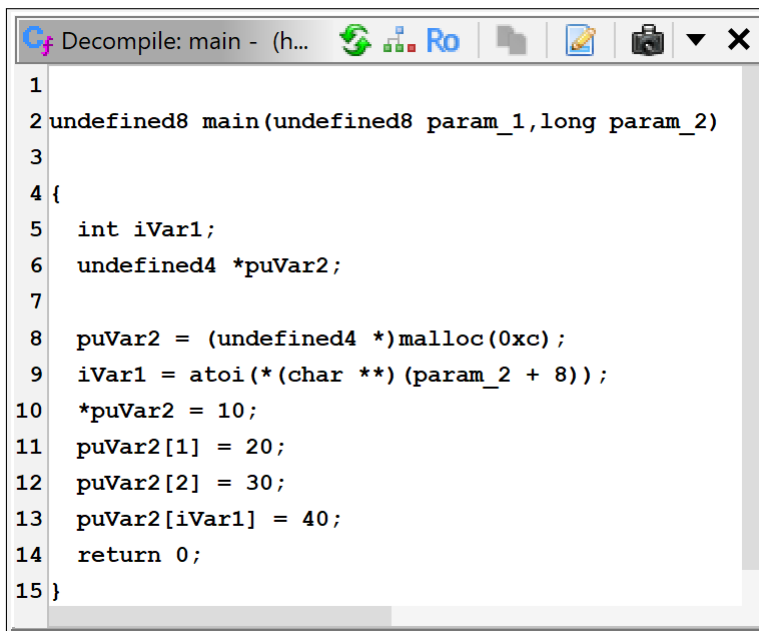
```

The starting address of the array (returned from `malloc` in the `RAX` register) is stored in the local variable `heap_array` ❷. In this example, unlike the previous examples, every access to the array begins with reading the contents of `heap_array` to obtain the array's base address. The references to `heap_array[0]`, `heap_array[1]`, and `heap_array[2]` require offsets of 0 ❸, 4 ❹, and 8 bytes ❺, respectively. The variable index array access `heap_array[idx]` is implemented with multiple instructions that compute the offset into the array

by multiplying the array index by the size of an array element ⑥ and adding the result to the base address of the array ⑦.

Heap-allocated arrays have one particularly nice feature: The number of elements allocated to the array can be computed from the total size of the array and the size of each element. The parameter passed to the allocator function (12 passed to `malloc` ① in this case) tells you the number of bytes allocated to the array. Dividing this by the size of an element (4 bytes here, as observed from the offsets ③ ④ ⑤, which step by 4, and the scale factor ⑥) tells us the number of elements in the array. In this example, a three-element array was allocated.

The decompiler was also able to recognize the array, as shown in Figure 8-6. (The name of the array pointer, `puVar2`, likely indicates that it is a pointer to an unsigned 4-byte value using the prefix `pu`.)



```
1
2 undefined8 main(undefined8 param_1,long param_2)
3
4 {
5     int iVar1;
6     undefined4 *puVar2;
7
8     puVar2 = (undefined4 *)malloc(0xc);
9     iVar1 = atoi(*(char **) (param_2 + 8));
10    *puVar2 = 10;
11    puVar2[1] = 20;
12    puVar2[2] = 30;
13    puVar2[iVar1] = 40;
14    return 0;
15 }
```

Figure 8-6: A Decompiler view of a heap array function

In this function, unlike in the stack-allocated array function, the decompiler listing shows the constant index array assignments. Normally, it would exclude them because the array is not used in other operations or returned from the function. This case is different because the assignments are not just manipulating stack variables: The stack variable is actually a pointer to memory that `malloc` requested from the heap. Writing via that variable does not write to the local stack variable, but instead uses the stack variable to locate the allocated memory. The program may lose the pointer (address of the start of the heap array) when the function exits, but the values persist in memory. (This particular example is actually a demonstration of a memory leak. While not a good programming practice, it does allow us to demonstrate the concept of a heap array.)

In conclusion, arrays are easiest to recognize when a variable is used as an index into the array. The array-access operation, which requires the index to be scaled by the size of an array element before adding the resulting offset to the base address of the array, stands out in a disassembly listing.

Structure Member Access

C-style structs, here generically referred to as *structures*, group collections of (often heterogeneous) data items into a composite data type. In source code, the data fields within a structure are accessed by name rather than by index. Unfortunately, the compiler converts these informative field names to numeric offsets, so by the time you are looking at a disassembly, structure field access looks remarkably similar to accessing array elements using constant indices.

The following examples will use this structure definition containing five heterogeneous fields:

struct ch8_struct {	// Size	Minimum offset	Default offset
int field1;	// 4	0	0
short field2;	// 2	4	4
char field3;	// 1	6	6
int field4;	// 4	7	8
double field5;	// 8	11	16
};	// Minimum total size: 19	Default size: 24	

When a compiler encounters a structure definition, it maintains a running total of the number of bytes consumed by the fields of the structure to determine the offset of each field within the structure. The resulting sum determines the minimum space required for the structure.

You should never assume that a compiler utilizes the minimum required space to allocate a structure, however. By default, compilers align structure fields to memory addresses that allow for the most efficient reading and writing of those fields. For example, a compiler will align 4-byte integer fields to offsets that are divisible by four, while it will align 8-byte doubles to offsets that are divisible by eight.

Depending on the composition of the structure, the compiler may insert padding bytes to meet alignment requirements, meaning the actual size of a structure may be larger than the sum of its component fields. You can view the default offsets and resulting structure size for the sample structure in the commented Default offset column in the preceding structure definition; notice that they sum to 24 rather than the minimum required size, 19.

You can pack structures into the minimum required space by using compiler options to request specific member alignments. Microsoft C/C++ and GNU gcc/g++ both recognize the pack pragma for controlling structure field alignment. The GNU compilers additionally recognize the packed attribute for controlling structure alignment on a per-structure basis. If you request a 1-byte alignment for structure fields, compilers will squeeze the structure into the minimum required space. In the sample structure definition, you can see

the offsets and structure size for the packed version in the Minimum offset column. (Note that some processors perform better when data is aligned according to their types, while other processors may generate exceptions if data is *not* aligned on specific boundaries.)

With these facts in mind, let's look at how structures are treated in compiled code. As with arrays, accessing structure members involves adding the base address of the structure to the offset of the desired member. But while array offsets can be computed at runtime from a provided index value because each element in an array has the same size, structure offsets are almost always computed at compile time and will turn up in compiled code as fixed offsets into the structure, looking nearly identical to array references that make use of constant indices.

Creating structures in Ghidra is more involved than creating arrays, so we cover this task in the next section, after we show several examples of disassembled and decompiled structures.

Globally Allocated Structures

As with globally allocated arrays, the addresses of globally allocated structures are known at compile time. This allows the compiler to compute the address of each member of the structure at compile time and eliminates the need to do any math at runtime. Consider the following program, which accesses a globally allocated structure:

```
struct ch8_struct global_struct;
int main() {
    global_struct.field1 = 10;
    global_struct.field2 = 20;
    global_struct.field3 = 30;
    global_struct.field4 = 40;
    global_struct.field5 = 50.0;
}
```

If this program is compiled with default structure alignment options, we can expect to see something like the following when we disassemble it:

	int	EAX:4	<RETURN>
001005fa	PUSH	RBP	
001005fb	MOV	RBP,RSP	
001005fe	MOV	dword ptr [DAT_00301020],10	
00100608	MOV	word ptr [DAT_00301024],20	
00100611	MOV	byte ptr [DAT_00301026],30	
00100618	MOV	dword ptr [DAT_00301028],40	
00100622	MOVSD	XMM0,qword ptr [DAT_001006c8]	
0010062a	MOVSD	qword ptr [DAT_00301030],XMM0	
00100632	MOV	EAX,0x0	
00100637	POP	RBP	
00100638	RET		

This disassembly contains no math whatsoever to access the members of the structure, and, without access to the source code, it would not be possible to state with any certainty that a structure is being used at all. Because the compiler has performed all of the offset computations at compile time, this program appears to reference five global variables rather than five fields within a single structure.

You should be able to observe similarities with the example we considered earlier of globally allocated arrays that use constant index values. In Figure 8-2, the uniform offsets coupled with the values allowed us to surmise (accurately) that we were dealing with an array. In this example, we can conclude that we are not dealing with an array because the size of the variables is nonuniform (dword, word, byte, dword, and qword, respectively), but we lack sufficient evidence to assert that we are dealing with a struct.

Stack-Allocated Structures

Like stack-allocated arrays, stack-allocated structures are challenging to recognize based on stack layout alone, and the decompiler does not provide additional insight. Modifying the preceding program to use a stack-allocated structure declared in `main` yields the following disassembly:

```

    int main(void)
        int          EAX:4          <RETURN>
        undefined8    Stack[-0x18]:8  local_18
        undefined4    Stack[-0x20]:4  local_20
        undefined1    Stack[-0x22]:1  local_22
        undefined2    Stack[-0x24]:2  local_24
        undefined4    Stack[-0x28]:4  local_28
001005fa  PUSH    RBP
001005fb  MOV     RBP,RSP
001005fe  MOV     dword ptr [RBP + local_28],10
00100605  MOV     word ptr [RBP + local_24],20
0010060b  MOV     byte ptr [RBP + local_22],30
0010060f  MOV     dword ptr [RBP + local_20],40
00100616  MOVSD   XMM0,qword ptr [DAT_001006b8]
0010061e  MOVSD   qword ptr [RBP + local_18],XMM0
00100623  MOV     EAX,0x0
00100628  POP     RBP
00100629  RET

```

In the disassembly, it is easy to see that the 8-byte doubles are aligned to offsets that are divisible by eight, since the structure is not packed. Again, no math is performed to access the structure's fields, since the compiler can determine the relative offsets for each field within the stack frame at compile time, and we are left with the same, potentially misleading picture that five individual variables are being used rather than a single variable that happens to contain five distinct fields. In reality, `local_28` should be the start of a 24-byte structure, and each of the other variables should somehow be formatted to reflect the fact that they are fields within the structure.

Heap-Allocated Structures

Heap-allocated structures reveal much more about the size of the structure and the layout of its fields than do the structures we just discussed. When a structure is allocated in the program heap, the compiler has no choice but to generate code to compute the proper field address whenever a field is accessed because the structure's address is unknown at compile time.

For globally allocated structures, the compiler is able to compute a fixed starting address. For stack-allocated structures, the compiler can compute a fixed relationship between the start of the structure and the frame pointer for the enclosing stack frame. But when a structure has been allocated in the heap, the only reference to the structure available to the compiler is the pointer to the structure's starting address.

To demonstrate heap-allocated structures, let's modify the sample program to declare a pointer within main and assign it the address of a block of memory large enough to hold the structure:

```
int main() {
    struct ch8_struct *heap_struct;
    heap_struct = (struct ch8_struct*)malloc(sizeof(struct ch8_struct));
    heap_struct->field1 = 10;
    heap_struct->field2 = 20;
    heap_struct->field3 = 30;
    heap_struct->field4 = 40;
    heap_struct->field5 = 50.0;
}
```

Here is the corresponding disassembly:

	int	EAX:4	<RETURN>
	undefined8	Stack[-0x10]:8	heap_struct
	0010064a	PUSH	RBP
	0010064b	MOV	RBP,RSP
	0010064e	SUB	RSP,16
❶	00100652	MOV	EDI,24
	00100657	CALL	malloc
	0010065c	MOV	qword ptr [RBP + heap_struct],RAX
	00100660	MOV	RAX,qword ptr [RBP + heap_struct]
❷	00100664	MOV	dword ptr [RAX],10
	0010066a	MOV	RAX,qword ptr [RBP + heap_struct]
❸	0010066e	MOV	word ptr [RAX + 4],20
	00100674	MOV	RAX,qword ptr [RBP + heap_struct]
❹	00100678	MOV	byte ptr [RAX + 6],30
	0010067c	MOV	RAX,qword ptr [RBP + heap_struct]
❺	00100680	MOV	dword ptr [RAX + 8],40
	00100687	MOV	RAX,qword ptr [RBP + heap_struct]
	0010068b	MOVSD	XMM0,qword ptr [DAT_00100728]
❻	00100693	MOVSD	qword ptr [RAX + 16],XMM0

```

00100698  MOV    EAX,0x0
0010069d  LEAVE
0010069e  RET

```

In this example, we can discern the exact size and layout of the structure. The size of the structure can be inferred to be 24 bytes based on the amount of memory requested from `malloc` ❶. The structure contains the following fields at the indicated offsets:

- A 4-byte (dword) field at offset 0 ❷
- A 2-byte (word) field at offset 4 ❸
- A 1-byte field at offset 6 ❹
- A 4-byte (dword) field at offset 8 ❺
- An 8-byte (qword) field at offset 16 ❻

Based on the use of floating point instructions (`MOVSD`), we can further deduce that the `qword` field is actually a `double`.

The same program compiled to pack structures with a 1-byte alignment yields the following disassembly:

```

0010064a  PUSH   RBP
0010064e  SUB    RSP,16
00100652  MOV    EDI,19
00100657  CALL   malloc
0010065c  MOV    qword ptr [RBP + local_10],RAX
00100660  MOV    RAX,qword ptr [RBP + local_10]
00100664  MOV    dword ptr [RAX],10
0010066a  MOV    RAX,qword ptr [RBP + local_10]
0010066e  MOV    word ptr [RAX + 4],20
00100674  MOV    RAX,qword ptr [RBP + local_10]
00100678  MOV    byte ptr [RAX + 6],30
0010067c  MOV    RAX,qword ptr [RBP + local_10]
00100680  MOV    dword ptr [RAX + 7],40
00100687  MOV    RAX,qword ptr [RBP + local_10]
0010068b  MOVSD  XMM0,qword ptr [DAT_00100728] =
00100693  MOVSD  qword ptr [RAX + 11],XMM0
00100698  MOV    EAX,0x0
0010069d  LEAVE
0010069e  RET

```

The only changes are the smaller structure size (now 19 bytes) and the adjusted offsets to account for the realignment of each structure field.

Regardless of the alignment used when compiling a program, finding structures allocated and manipulated in the program heap is the fastest way to determine the size and layout of a given data structure.

However, keep in mind that many functions will not do you the favor of immediately accessing every member of a structure to help you understand the structure's layout. Instead, you may need to follow the use of the pointer

to the structure and make note of the offsets used whenever that pointer is dereferenced, and eventually piece together the complete layout of the structure. In “Example 3: Automated Structure Creation” on page 463, you will see how the decompiler can automate this process for you.

Arrays of Structures

Some programmers say that the beauty of composite data structures is that they allow you to build arbitrarily complex structures by nesting smaller structures within larger structures: arrays of structures, structures within structures, and structures that contain arrays as members, for example. The preceding discussions regarding arrays and structures apply just as well to such nested types. As an example, consider the following simple program, in which `heap_struct` points to an array of five `ch8_struct` items:

```
int main() {
    int idx = 1;
    struct ch8_struct *heap_struct;
    heap_struct = (struct ch8_struct*)malloc(sizeof(struct ch8_struct) * 5);
    heap_struct[idx].field1 = 10;
}
```

Under the hood, accessing `field1` involves multiplying the index value by the size of an array element (in this case, the size of the structure) and then adding the offset to the desired field. The corresponding disassembly is shown here:

	int	EAX:4	<RETURN>
	undefined4	Stack[-0xc]:4	idx
	undefined4	Stack[-0x18]:8	heap_struct

```

0010064a PUSH    RBP
0010064b MOV     RBP,RSP
0010064e SUB     RSP,16
00100652 MOV     dword ptr [RBP + idx],1
00100659 MOV     ❶ EDI,120
0010065e CALL    <EXTERNAL>::malloc
00100663 MOV     qword ptr [RBP + heap_struct],RAX
00100667 MOV     EAX,dword ptr [RBP + idx]
0010066a MOVSXD  RDX,EAX
0010066d MOV     ❷ RAX,RDX
00100670 ADD     RAX,RAX
00100673 ADD     RAX,RDX
00100676 SHL     ❸ RAX,3
0010067a MOV     RDX,RAX
0010067d MOV     RAX,qword ptr [RBP + heap_struct]
00100681 ADD     ❹ RAX,RDX
00100684 MOV     ❺ dword ptr [RAX],10
0010068a MOV     EAX,0

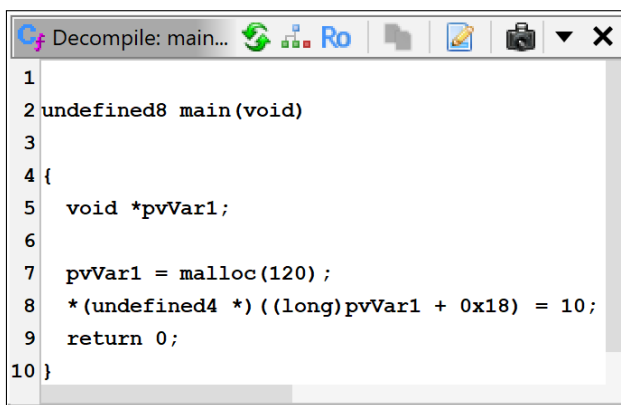
```

```
0010068f  LEAVE
00100690  RET
```

The function allocates 120 bytes ❶ in the heap. The array index in RAX is multiplied by 24 using a series of operations ❷, ending with `SHL RAX, 3` ❸ before being added to the start address of the array ❹. (If it is not readily apparent to you that the series of operations starting at ❷ is equivalent to multiplication by 24, don't worry. We discuss code sequences such as this in Chapter 20.) Because `field1` is the first member of the struct, no additional offset is required in order to generate the final address for the assignment into `field1` ❺.

From these facts, we can deduce the size of an array item (24), the number of items in the array ($120 / 24 = 5$), and the fact that there is a 4-byte (dword) field at offset 0 within each array element, although this short listing does not offer enough information to draw any conclusions about how the remaining 20 bytes within each structure are allocated to additional fields.

You can even more easily deduce the size of the array using the same formula from the decompiler listing in Figure 8-7.



```
1
2 undefined8 main(void)
3
4 {
5     void *pvVar1;
6
7     pvVar1 = malloc(120);
8     *(undefined4 *) ((long)pvVar1 + 0x18) = 10;
9     return 0;
10 }
```

Figure 8-7: The Decompiler view of a function with a heap-allocated struct array

(Note that 0x18 hex is 24 decimal.)

Creating Structures with Ghidra

In the preceding chapter, you saw how to use Ghidra's array-aggregation capabilities to collapse long lists of data declarations into a single disassembly line representing an array. The next few sections explore Ghidra's facilities for improving the readability of code that manipulates structures. Our goal is to move away from cryptic structure references such as `[EDX + 10h]` and toward something more readable, like `[EDX + ch8_struct.field_e]`.

Whenever you discover that a program is manipulating a data structure, you need to decide whether you want to incorporate structure field names into your disassembly or whether you can make sense of all the numeric offsets sprinkled throughout the listing. In some cases, Ghidra may recognize

the use of a structure defined as part of the C standard library or the Windows API and use its knowledge of the exact layout of the structure to convert numeric offsets into symbolic field names. This is the ideal case, as it leaves you with a lot less work to do. We will return to this scenario once you understand a little more about how Ghidra deals with structure definitions in general.

STATE OF THE UNION

A union is a construct that is similar to a structure. The major difference between structures and unions is that structure fields have unique offsets and their own dedicated memory space, whereas union fields all overlap one another beginning at offset 0. The result is that all union fields share the same memory space. The Union Editor window in Ghidra looks similar to the Structure Editor window, and the functionality is basically the same.

Creating a New Structure

When Ghidra has no layout knowledge for a structure, you can create the structure by selecting the data and using the right-click context menu. When you select **Data ► Create Structure** (or use the hotkey **SHIFT-[]**), you will see the Create Structure window shown in Figure 8-8. Since you have highlighted a block of data (which could be defined or undefined), Ghidra will try to identify existing structures that have a matching format or the same size. You can select one of the existing structures from the window or create a new structure.

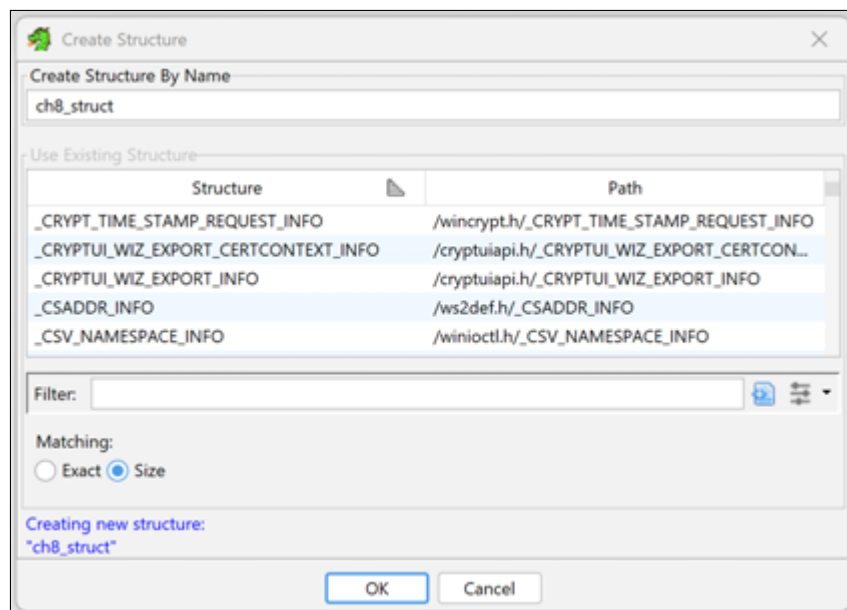


Figure 8-8: The Create Structure window

In this example, we are using the globally allocated structure sample code discussed previously and are creating a new structure called `ch8_struct`. As soon as you click OK, the structure becomes an official type in the Data Type Manager window, and the information is propagated to other Code-Browser windows. (To make it easier to track the disassembly and decompiler, we have used the right-click context window to convert the hex values to decimal.)

Let's look at the effect of this creation on the associated CodeBrowser windows, starting with the Listing window. As shown earlier in the chapter, the disassembly listing gives you few hints that you might be dealing with a structure because the code modifies a series of seemingly unrelated global variables:

```

001005fa  PUSH   RBP
001005fb  MOV     RBP,RSP
001005fe  MOV     dword ptr [DAT_00301020],10
00100608  MOV     word ptr [DAT_00301024],20
00100611  MOV     byte ptr [DAT_00301026],30
00100618  MOV     dword ptr [DAT_00301028],40
00100622  MOVSD   XMM0,qword ptr [DAT_001006c8]
0010062a  MOVSD   qword ptr [DAT_00301030],XMM0
00100632  MOV     EAX,0
00100637  POP     RBP
00100638  RET

```

When you navigate to the associated data items, select the range (00301020 through 00301037), and create the associated structure, you see that the individual data items in the structure are now associated with a structure called `ch8_struct_00301020`, and that each item in the structure has the name `field_` concatenated with its offset from the first element in the structure:

```

00401035  POP     EBP
001005fb  MOV     RBP,RSP
001005fe  MOV     dword ptr [ch8_struct_00301020],10
00100608  MOV     word ptr [ch8_struct_00301020.field4_0x4],20
00100611  MOV     byte ptr [ch8_struct_00301020.field6_0x6],30
00100618  MOV     dword ptr [ch8_struct_00301020.field8_0x8],40
00100622  MOVSD   XMM0,qword ptr [DAT_001006c8]
0010062a  MOVSD   qword ptr [ch8_struct_00301020.field16_0x10],XMM0
00100632  MOV     EAX,0
00100637  POP     RBP
00100638  RET

```

This is just one of the windows that changes with the creation of the structure. Recall that the Decompiler window gave us a helpful warning that we might be working with a structure or array. After we create the structure, the warning disappears, and the decompiled code more closely resembles the original C code, as shown in Figure 8-9.

```

1
2 undefined8 main(void)
3
4 {
5     ch8_struct_00301020._0_4_ = 10;
6     ch8_struct_00301020._4_2_ = 20;
7     ch8_struct_00301020._6_1_ = 30;
8     ch8_struct_00301020._8_4_ = 40;
9     ch8_struct_00301020._16_8_ = 0x4049000000000000;
10    return 0;
11 }

```

Figure 8-9: The Decompiler view after a struct is created

The new structure also now appears as an entry in the Data Type Manager window in the CodeBrowser, and a right-click context menu allows you to find all uses of the structure. Figure 8-10 shows the new entry in the Data Type Manager window and the associated window showing all uses of `ch8_struct`.

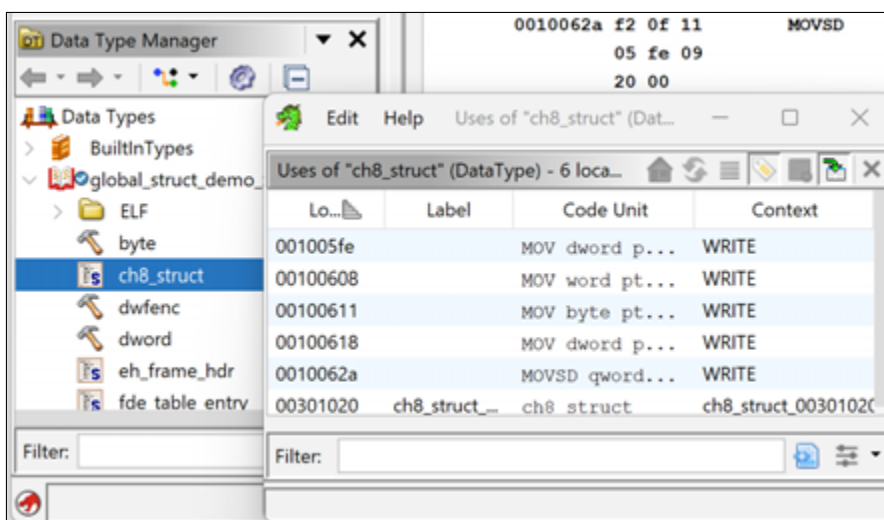


Figure 8-10: The newly declared structure in the Data Type Manager and References windows

Editing Structure Members

At this point, Ghidra presents the newly created structure as a contiguous collection of undefined bytes with cross-references at each offset accessed by the example program, instead of a collection of defined data types (which you have identified from the size of each item and the way it is being used). To define the type of each field, you can edit the structure from the Listing

window by right-clicking and selecting the appropriate Data option. Alternatively, you can edit the structure from within the Data Type Manager by double-clicking the structure.

If you double-click the newly created structure in the Data Type Manager window (shown in Figure 8-10), the Structure Editor window (shown in Figure 8-11) opens to show 24 elements of an undefined type, all with a length of 1. To determine the number, sizes, and types of the individual elements within the structure, you could study the disassembly, or you could let the decompiler listing (from Figure 8-9) provide the answers.

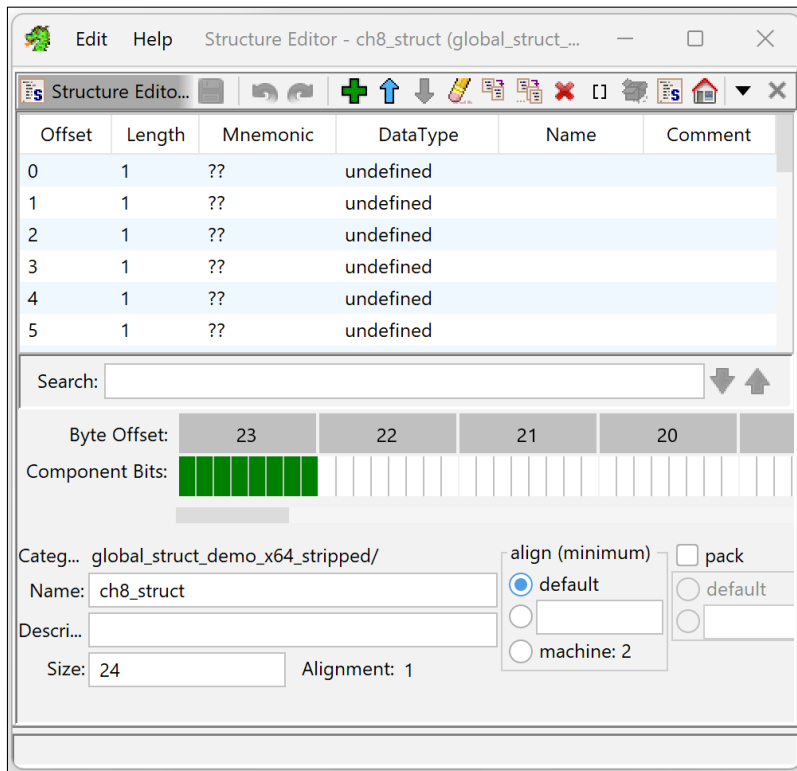


Figure 8-11: The Structure Editor window (with “Show numbers in Hexadecimal” deselected)

The original decompiler listing associated with our new structure shows that five items are referenced within the same structure, `ch8_struct_00301020`, using field names containing two integers. The first integer represents the offset from the base address of the structure. The second shows the number of bytes used, which is a good indicator of the size of the item. Using this information (and some meaningful field names), you can update the Structure Editor window, as shown in Figure 8-12.

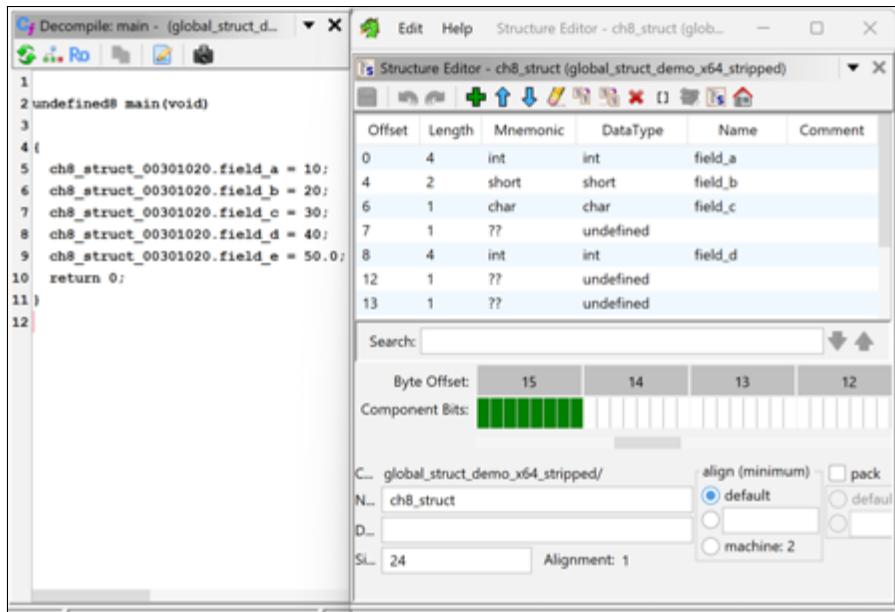


Figure 8-12: Decompiler and Structure Editor windows after editing structure

The Byte Offset / Component Bits scroll bar within the Structure Editor provides a visual representation of the structure. When a structure is edited, the Decompiler window (on the left of Figure 8-12), the Listing window, and other associated windows are also updated.

In this example, we have defined `field_c` as a character within the structure, but we used the right-click context menu to select to display it as a decimal at the start of this example. Since the ASCII character represented by 30 (0x1e) is an unprintable control character (RS), the Decompiler window would display `'\x1e'` if we used the same menu option to convert it to a character. (If the ASCII character were a printable character, you would see the ASCII character in the Decompiler window.) You can also see that the padding bytes (indicated by the mnemonic `??`) have been included for proper field alignment, and that the offsets to each field and the overall size (24 bytes) of the structure match the values seen in the earlier examples.

Applying Structure Layouts

While understanding how to identify and define structures is important, it isn't always necessary. Ghidra has a library of known structure layouts populated with information gathered by parsing common C header files. The layout of a structure defines its total size, the name and size of each field, and the starting offset of each field within the structure. You can use structure layouts even without associated content in the data section, which is especially helpful when dealing with structure pointers.

Anytime you encounter a memory reference of the form `[reg+N]` (for example, `[RAX+0x12]`), where `reg` is a register name and `N` is a small constant, `reg` is being used as a pointer and `N` represents an offset into the memory that `reg` points to. This is a common pattern for structure member access, with `reg` pointing to the beginning of the structure and `N` selecting the field at structure offset `N`. Under some circumstances, and with your assistance, Ghidra can clean up this type of memory reference to reflect both the type of structure being pointed to and the specific field within that structure that is being referenced.

Let's look at the 32-bit version of the example from the beginning of the chapter, where we were requesting an HTTP page from a server. The request is made by a function named `get_page`. In this version of the binary, Ghidra asserts that the function receives three stack-allocated parameters. These parameters appear in the Decompiler window as follows:

```
ssize_t get_page(undefined4 param_1,undefined4 param_2,int param_3)
```

The Decompiler window shows that `param_3` is used with some offsets in a call to `connect`:

```
iVar1=connect(local_14,*(sockaddr **)(param_3+20),*(socklen_t*)(param_3+16));
```

Tracing through the calling sequence and the return values from the called functions, we can conclude that `param_3` is a pointer to an `addrinfo` struct, and can retype `param_3` as an `addrinfo*` (using CTRL-L from the Listing or Decompiler window). The decompiled statement using `param_3` will be replaced with the far more informative statement shown here:

```
iVar1 = connect(local_14, param_3->ai_addr, param_3->ai_addrlen);
```

You can see that pointer arithmetic has been replaced by structure field references. Pointer arithmetic in source code is rarely self explanatory. Any effort you spend updating data types for program variables will be well worth it. You will have saved your colleagues the time required to deduce the type of `param_3` themselves, and you will be thankful, upon returning from two weeks at the beach, that you don't need to reanalyze the code to relearn the type of that variable that you forgot to update.

C++ Reversing Primer

C++ classes are the object-oriented extensions of C structs, so it is somewhat logical to wrap up our discussion of data structures by reviewing the features of compiled C++ code. Detailed coverage of C++ is beyond the scope of this book. Here, we attempt to cover the highlights and a few of the differences between Microsoft's C++ compiler and GNU's `g++`.

That said, a solid, fundamental understanding of the C++ language will greatly assist you in understanding compiled C++. Object-oriented concepts such as inheritance and polymorphism are difficult enough to master at the source level. Attempting to dive into these concepts at the assembly

level without understanding them at the source level can be an exercise in frustration.

The this Pointer

The `this` pointer is available in all nonstatic C++ member functions. Whenever such a function is called, `this` is initialized to point to the object used to invoke the function. Consider the following function calls in C++:

```
// object1, object2, and *p_obj are all the same type.
object1.member_func();
object2.member_func();
p_obj->member_func();
```

In the three calls to `member_func`, `this` takes on the values `&object1`, `&object2`, and `p_obj`, respectively.

It is easiest to view `this` as a hidden first parameter passed in to all nonstatic member functions. The Microsoft C++ compiler utilizes the `thiscall` calling convention and passes `this` in the ECX register (x86) or the RCX register (x64). The GNU g++ compiler treats `this` exactly as if it were the first (leftmost) parameter to nonstatic member functions. On 32-bit Linux x86, the address of the object used to invoke the function is pushed as the topmost item on the stack prior to calling the function. On Linux x86-64, this is passed in the first register parameter, RDI.

From a reverse engineering point of view, moving an address into the ECX register immediately before a function call is a probable indicator of two things. First, the file was compiled using Microsoft's C++ compiler. Second, the function is possibly a member function. When the same address is passed to two or more functions, we can conclude that those functions all belong to the same class hierarchy.

Within a function, the use of ECX prior to initializing it implies that the caller must have initialized ECX and is a possible sign that the function is a member function (although the function may simply use the `fastcall` calling convention). In addition, when a member function passes `this` to additional functions, those functions can be inferred to be members of the same class as well.

For code compiled using GNU g++, calls to member functions stand out somewhat less because `this` looks a lot like any other first parameter. However, any function that does not take a pointer as its first argument can certainly be ruled out as a member function.

Virtual Functions and Vtables

Virtual functions enable polymorphic behavior in C++ programs. For each class (or subclass, through inheritance) that contains virtual functions, the compiler generates a table containing pointers to each virtual function in the class. Such tables are called *vtables* or *vtables*.

Every instance of a class that contains virtual functions is given an additional data member that points to the class's vtable. The *vtable pointer* is allocated as the first data member within the class instance, and when an object is created at runtime, its constructor function sets its vtable pointer to point at the appropriate vtable. When that object invokes a virtual function, the correct function is selected by performing a lookup in the object's vtable. Thus, vtables are the underlying mechanism that facilitates runtime resolution of calls to virtual functions.

Some examples may help clarify the use of vtables. Consider the following C++ class definitions:

```
class BaseClass {
public:
    BaseClass();
    ❶ virtual void vfunc1() = 0;
    virtual void vfunc2();
    virtual void vfunc3();
    virtual void vfunc4();
private:
    int x;
    int y;
};
❷ class SubClass : public BaseClass {
public:
    SubClass();
    ❸ virtual void vfunc1();
    virtual void vfunc3();
    virtual void vfunc5();
private:
    int z;
};
```

In this case, SubClass inherits from BaseClass ❷. BaseClass contains four virtual functions ❶, while SubClass contains five ❸ (four from BaseClass, two of which it overrides, plus the new vfunc5).

Within BaseClass, vfunc1 is a *pure virtual function*, indicated by = 0 in its declaration. Pure virtual functions have no implementation in their declaring class and *must* be overridden in a subclass before the class is considered concrete. In other words, there is no function named BaseClass::vfunc1, and until a subclass provides an implementation, no objects can be instantiated. SubClass provides such an implementation, so SubClass objects can be created. In object-oriented terms, BaseClass::vfunc1 is an *abstract function*, which makes BaseClass an *abstract base class* (that is, an incomplete class that cannot be directly instantiated since it is missing an implementation for at least one function).

At first glance, BaseClass appears to contain two data members and SubClass has three data members. However, any class that contains virtual functions, either explicitly or because they are inherited, also contains a

vtable pointer. As a result, the compiled implementation of `BaseClass` has three data members, while instantiated `SubClass` objects have four data members. In each case, the first data member is the vtable pointer.

Within `SubClass`, the vtable pointer is actually inherited from `BaseClass` rather than being introduced specifically for `SubClass`. You can see this in the simplified memory layout in Figure 8-13, in which a single `SubClass` object has been dynamically allocated. During the creation of the object, the new object's vtable pointer is initialized to point to the correct vtable (`SubClass`'s in this case).

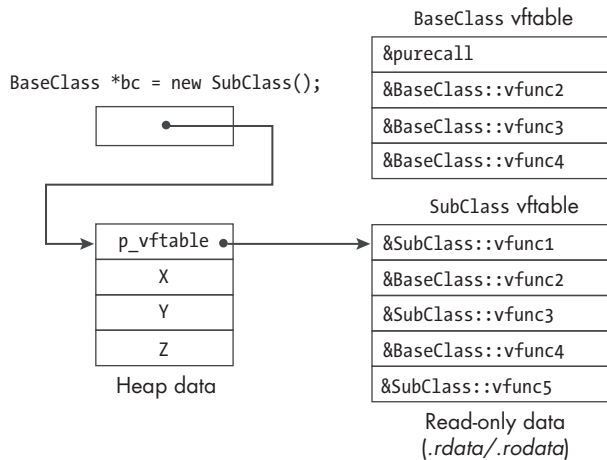


Figure 8-13: A simple vtable layout

The vtable for `SubClass` contains two pointers to functions belonging to `BaseClass` (`BaseClass::vfunc2` and `BaseClass::vfunc4`) because `SubClass` does not override either of these functions and instead inherits them from `BaseClass`. The vtable for `BaseClass` shows how pure virtual functions are handled. Because there is no implementation for the pure virtual function `BaseClass::vfunc1`, no address is available to store in the `BaseClass` vtable slot for `vfunc1`. In such cases, compilers insert the address of an error-handling function, dubbed `purecall` in Microsoft libraries and `__cxa_pure_virtual` in GNU libraries. In theory, these functions should never be called, but in the event that they are, they cause the program to be terminated abnormally.

You must account for the vtable pointer when you manipulate classes within Ghidra. Because C++ classes are extensions of C structures, you can use Ghidra's structure definition features to define the layout of C++ classes. With polymorphic classes, you must include a vtable pointer as the first field within the class and account for the vtable pointer in the total size of the object. This is most apparent when observing the dynamic allocation of an object using the `new` operator, where the size value passed to `new` includes the space needed by all explicitly declared fields in the class (and any super-classes) as well as any space required for a vtable pointer.

In the following example, a SubClass object is created dynamically and its address saved in a BaseClass pointer. The pointer is then passed to a function (call_vfunc), which uses the pointer to call vfunc3:

```
void call_vfunc(BaseClass *bc) {
    bc->vfunc3();
}

int main() {
    BaseClass *bc = new Subclass();
    call_vfunc(bc);
}
```

Since vfunc3 is a virtual function and bc points to a SubClass object, the compiler must ensure that SubClass::vfunc3 is called. The following disassembly of a 32-bit Microsoft C++ version of call_vfunc demonstrates how the virtual function call is resolved:

	undefined	__cdecl	call_vfunc(int * bc)
	undefined	<UNASSIGNED>	<RETURN>
	int *	Stack[0x4]:4	bc
004010a0	PUSH	EBP	
004010a1	MOV	EBP,ESP	
004010a3	MOV	EAX,dword ptr [EBP + bc]	
004010a6	MOV	❶ EDX,dword ptr [EAX]	
004010a8	MOV	❷ ECX,dword ptr [EBP + bc]	
004010ab	MOV	❸ EAX,dword ptr [EDX + 8]	
004010ae	CALL	❹ EAX	
004010b0	POP	EBP	
004010b1	RET		

The vtable pointer (the address of SubClass's vtable) is read from the structure and saved in EDX ❶. Next, the this pointer is moved into ECX ❷. Then, the vtable is indexed to read the third pointer (the address of SubClass::vfunc3 in this case) into the EAX register ❸. Finally, the virtual function is called ❹.

The vtable indexing operation ❸ looks very much like a structure reference operation. In fact, it is no different, and it is possible to define new structures for the class and its vtable (by right-clicking in the Data Type Manager window) and then use the defined structures (see Figure 8-14) to make the disassembly and decompilation more readable.

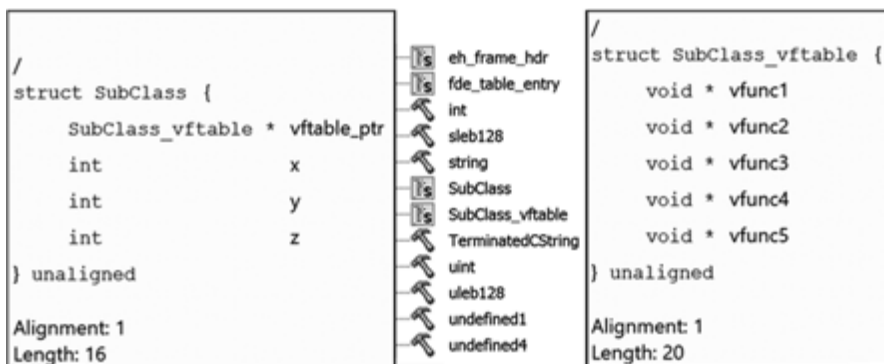


Figure 8-14: The Data Manager Window showing the new *SubClass* and *SubClass_vftable*

Figure 8-15 shows the Decompiler window with references to the new structures.

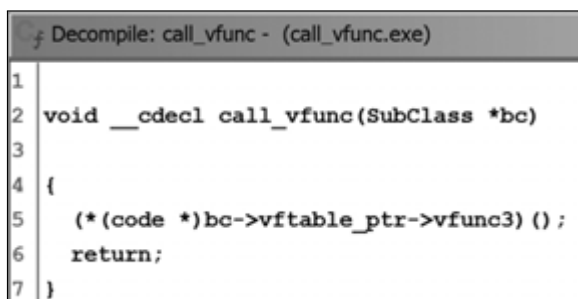


Figure 8-15: The Decompiler window reflecting defined structures for *SubClass*

A class's vftable is referenced directly only within the class's constructor(s) and its destructor(s). When you locate a vftable, you can utilize Ghidra's data cross-referencing capabilities (see Chapter 9) to quickly locate all constructors and destructors for the associated class.

The Object Life Cycle

Understanding the mechanism by which objects are created and destroyed can help reveal object hierarchies and nested object relationships and quickly identify class constructor and destructor functions.

WHAT'S A CONSTRUCTOR?

A *class constructor function* is an initialization function that is invoked when a new object of that class is created. Constructors provide an opportunity to initialize variables within the class. The inverse of a constructor, a *destructor*, is called when an object goes out of scope or a dynamically allocated object is explicitly deleted. Destructor functions perform cleanup activities, such as releasing resources like open file descriptors and dynamically allocated memory. Properly written destructors mitigate the potential for memory leaks.

The storage class of an object determines when its constructor is called. A variable's storage class roughly defines its lifetime during program execution. The two most common storage classes in C are static and automatic. The storage space for static variables exists for the duration of the program. Automatic variables are associated with function invocations and exist only for the duration of a specific function call.

For global and statically allocated objects (which are of the static storage class), constructors are called during program startup prior to entry into the program's main function. Constructors for stack-allocated objects (which are of the automatic storage class) are invoked when the object comes into scope within the function in which it is declared. In many cases, this will be immediately upon entry to that function. However, when an object is declared within a nested block statement, its constructor is not invoked until that block is entered, if it is entered at all. When an object is allocated dynamically in the program heap, its creation is a two-step process: The `new` operator is invoked to allocate the object's memory, and then the constructor is invoked to initialize the object. Microsoft C++ ensures that the result of `new` is not null prior to invoking the constructor, but GNU's `g++` does not.

WHAT'S NEW?

The `new` operator is used for dynamic memory allocation in C++ in much the same way that `malloc` is used in C. It is used to allocate memory from the heap and allows a program to request space as needed during execution. The `new` operator is built into the C++ language, while `malloc` is merely a standard library function. Recall that C is a subset of C++, so you might see either in a C++ program. The most notable difference between `malloc` and `new` is that invocations of `new` for object types will result in an implicit invocation of the object's constructor, whereas memory returned by `malloc` is not initialized before it is made available to the caller.

When a constructor executes, the following sequence of actions takes place:

1. If the class has a superclass, the superclass constructor is invoked.
2. If the class has any virtual functions, the vtable pointer is initialized to point to the class's vtable. This may overwrite a vtable pointer

that was initialized in the superclass constructor, which is exactly the desired behavior.

3. If the class has any data members that are themselves objects, the constructor for each of those data members is invoked.
4. Finally, the class constructor is executed. This is the C++ constructor code specified by the programmer of the class.

From a programmer's perspective, constructors do not specify a return type or allow a value to be returned. Some compilers actually return this as a result that they may further utilize in the caller, but this is a compiler implementation detail and C++ programmers cannot access the returned value.

Destructors, as their name implies, are called at the end of an object's lifetime. For global and static objects, destructors are called by cleanup code that is executed after the `main` function terminates. Destructors for stack-allocated objects are invoked as the objects go out of scope. Destructors for heap-allocated objects are invoked via the `delete` operator immediately before the memory allocated to the object is released.

The actions performed by destructors mimic those performed by constructors, with the exception that they are performed in roughly reverse order:

1. If the class has any virtual functions, the vtable pointer for the object is restored to point to the vtable for the associated class. This is required in the case where a subclass had overwritten the vtable pointer as part of its creation process.
2. The programmer-specified code for the destructor executes.
3. If the class has any data members that are themselves objects, the destructor for each of those members is executed.
4. Finally, if the object has a superclass, the superclass destructor is called.

By understanding when superclass constructors and destructors are called, it is possible to trace an object's inheritance hierarchy through the chain of calls to its related superclass functions.

I THINK YOU ARE OVERLOADED

Overloaded functions are functions that share the same name but have different parameters. C++ requires that each version of an overloaded function differs from every other version in the sequence and/or quantity of parameter types that the function receives. In other words, while they share the same function name, each function prototype must be unique, and each overloaded function body can be uniquely identified within the disassembled binary. This should not be confused with functions, such as `printf`, which take a variable number of arguments but have a single function body.

Name Mangling

Also called *name decoration*, *name mangling* is the mechanism that C++ compilers use to distinguish among overloaded versions of a function. To generate unique internal names for overloaded functions, compilers decorate the function name with additional characters that encode various pieces of information about the function: the namespace to which the function (or its owning class) belongs (if any), the class to which the function belongs (if any), and the parameter sequence (type and order) required to call the function.

Name mangling is a compiler implementation detail for C++ programs and, as such, is not part of the C++ language specification. Not unexpectedly, compiler vendors have developed their own, often incompatible conventions for name mangling. Fortunately, Ghidra understands the name mangling conventions employed by Microsoft's C++ compiler and GNU g++ v3 (and later), as well as those of some other compilers.

Ghidra provides names of the form `FUN_address` in place of the mangled name. Mangled names do carry valuable information regarding the signature of each function, and Ghidra includes this information in the Symbol Table window in addition to propagating the information to the disassembly and other related windows. (To determine the signature of a function without a mangled name, you might need to conduct time-consuming analysis of the data flowing into and out of the function.)

Runtime Type Identification

C++ provides operators to determine (`typeid`) and check (`dynamic_cast`) an object's data type at runtime. To support these operations, C++ compilers must embed type-specific information, for each polymorphic class, within a program binary. When a `typeid` or `dynamic_cast` operation is performed at runtime, library routines reference the type-specific information to determine the exact runtime type of the polymorphic object being referenced. Unfortunately, as with name mangling, *Runtime Type Identification (RTTI)* is a compiler implementation detail rather than a language issue, and there is no standard means by which compilers implement RTTI capabilities.

We will take a brief look at the similarities and differences between the RTTI implementations of Microsoft's C++ compiler and GNU g++. Specifically, we will describe how to locate RTTI information and, from there, how to learn the name of the class with which the information is associated.

Consider the following simple program, which uses polymorphism:

```
#include <iostream>
class abstract_class {
public:
    virtual int vfunc() = 0;
};
class concrete_class : public abstract_class {
public:
    concrete_class(){};
    int vfunc();
};
```

```

};
int concrete_class::vfunc() {return 0;}
❶ void print_type(abstract_class *p) {
    std::cout << typeid(*p).name() << std::endl;
}
int main() {
    ❷ abstract_class *sc = new concrete_class();
    print_type(sc);
}

```

The `print_type` function ❶ prints the type of the object being pointed to by the pointer `p`. In this case, it must print "concrete_class" since the `main` function ❷ creates a `concrete_class` object. How does `print_type`, and more specifically `typeid`, know what type of object `p` is pointing to?

The answer is surprisingly simple. Since every polymorphic object contains a pointer to a vtable, compilers leverage that fact by colocating class-type information with the class vtable. Specifically, the compiler places a pointer that points to a structure containing information about the class that owns the vtable immediately prior to the class vtable. In GNU g++ code, this pointer points to a `type_info` structure, which contains a pointer to the name of the class. In Microsoft C++ code, the pointer points to a Microsoft `RTTICompleteObjectLocator` structure, which in turn contains a pointer to a `TypeDescriptor` structure. The `TypeDescriptor` structure contains a character array that specifies the name of the polymorphic class.

RTTI information is required only in C++ programs that use the `typeid` or `dynamic_cast` operator. Most compilers provide options to disable the generation of RTTI in binaries that do not require it; therefore, you should not be surprised if you encounter compiled binaries that contain no RTTI information even though vtables are present.

For C++ programs built with Microsoft's C++ compiler, Ghidra contains an RTTI analyzer that is enabled by default and that is capable of identifying Microsoft RTTI structures, annotating those structures (if present) in the disassembly listing, and utilizing class names recovered from those RTTI structures in the Symbol Tree's *Classes* folder.

Ghidra has no RTTI analyzer for non-Windows binaries. When Ghidra encounters an unstripped, non-Windows binary, and if it understands the name mangling scheme employed in the binary, then Ghidra utilizes available name information to populate the Symbol Tree's *Classes* folder. If a non-Windows binary has been stripped, Ghidra will not be able to automatically recover any class names or identify vtables or RTTI information.

Inheritance Relationships

It is possible to unravel inheritance relationships by using a compiler's particular implementation of RTTI, but RTTI may not be present when a program does not utilize the `typeid` or `dynamic_cast` operators. Lacking RTTI information, what techniques can we employ to determine inheritance relationships among C++ classes?

The simplest method of determining an inheritance hierarchy is to observe the chain of calls to superclass constructors that are called when an object is created. The single biggest hindrance to this technique is the use of inline constructors. In C/C++, a function declared as `inline` is usually treated as a macro by the compiler, and the code for the function is expanded in place of an explicit function call. Inline functions hide the fact that a function is being used, since no assembly language call statement will be generated. This makes it challenging to understand that a superclass constructor has in fact been called.

The analysis and comparison of vtables can also reveal inheritance relationships. For example, in comparing the vtables shown in Figure 8-13, we note that the vtable for `SubClass` contains two of the same pointers that appear in the vtable for `BaseClass`, and we conclude that `BaseClass` and `SubClass` must be related in some way. To understand which one is the base class and which one is the subclass, we can apply the following guidelines, singly or in combination:

- When two vtables contain the same number of entries, the two corresponding classes *may* be involved in an inheritance relationship.
- When the vtable for class X contains more entries than the vtable for class Y, class X *may* be a subclass of class Y.
- When the vtable for class X contains entries that are also found in the vtable for class Y, then one of the following relationships must exist: X is a subclass of Y, Y is a subclass of X, or X and Y are both subclasses of a common superclass Z.
- When the vtable for class X contains entries that are also found in the vtable for class Y and the vtable for class X contains at least one `purecall` entry that is not also present in the corresponding vtable entry for class Y, then class Y is likely to be a subclass of class X.

NOTE

An exception to the third item above may occur when a compiler optimizes multiple instances of identical function bodies (often empty, placeholder member functions) into a single instance and points all vtable entries for these empty bodies, across all classes, to the lone code block for the optimized function.

While the preceding list is by no means all-inclusive, we can use these guidelines to deduce the relationship between `BaseClass` and `SubClass` in Figure 8-14. In this case, the last three rules all apply, but the last rule specifically leads us to conclude, based on vtable analysis alone, that `SubClass` inherits from `BaseClass`.

Summary

You can expect to encounter complex data types in all but the most trivial programs. Understanding how data within data structures is accessed and recognizing clues as to the layout of those data structures are essential reverse engineering skills. Ghidra provides a wide variety of features designed specifically for dealing with data structures. Familiarity with these features will greatly enhance your ability to comprehend what data is being manipulated and enable you to spend more time understanding how and why that data is being manipulated. In the next chapter, we continue our discussion of Ghidra's basic capabilities with an in-depth look at cross-references.

9

UNDERSTANDING CROSS-REFERENCES



Two common questions you will ask while reverse engineering a binary are “Where is this function called from?” and “Which functions access this data?” These and other similar questions seek to identify and catalog the references to and from various resources in a program.

To understand the usefulness of such questions, consider the following two examples:

Example 1

While you are reviewing the large number of ASCII strings in a particular binary, you see a string that seems particularly suspicious: “Pay within 72 hours or the recovery key will be destroyed and your data will remain encrypted forever.” On its own, this string is just circumstantial evidence. It in no way confirms that the binary has the capability or intent to execute a crypto ransomware attack. The answer to the question “Where is this string referenced in the binary?” would help you to quickly track down the program locations that make use of the string. This information, in turn, should assist you in locating any related crypto ransomware code, or help you demonstrate that, in this context, the string is benign.

Example 2

You have located a function containing a stack-allocated buffer that can be overflowed, and you want to determine whether it's possible to exploit this vulnerability. Before you can use this function to demonstrate an exploit, you must get it to execute. This leads to the question “Which functions call this vulnerable function?” as well as additional questions regarding the nature of the data that those functions may pass to the vulnerable function. This line of questioning must continue as you work your way back up potential call chains to find one that you can influence to demonstrate that the overflow is exploitable.

Ghidra can help you analyze both of these cases (and many others) through its extensive mechanisms for displaying and accessing reference information. In this chapter, we discuss the types of references that Ghidra makes available, the tools for accessing reference information, and ways to interpret that information. In Chapter 10, we will use Ghidra's graphing capabilities to examine visual representations of reference relationships.

Referencing Basics

All references obey the same general traffic rules. Associated with each reference is the notion of a direction; all references are made from one address to another address. If you are familiar with graph theory, you can think of addresses as nodes (or *vertices*) in a directed graph and references as the *edges* that identify directed connections between the nodes. Figure 9-1 provides a quick refresher on basic graph terminology. In this simple graph, three nodes—A, B, and C—are connected by two directed edges.

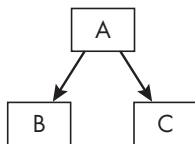


Figure 9-1: A directed graph with three nodes and two edges

The arrows indicate the allowable direction of travel along the edge. In Figure 9-1, travel from A to B is possible, but travel from B to A is not, similar to a one-way street.

Ghidra has two basic categories of references: forward references and back references, each with subcategories. The back references are the less complex of the two types and the ones you are likely to use most frequently in reverse engineering. Also referred to as *cross-references*, they provide a means to navigate between locations in the listing, such as code and data.

Cross-References (Back References)

Ghidra often refers to back references simply as *XREFs*, which is a mnemonic for the term *cross-reference*. Let's start by looking at specific examples of XREFs in Ghidra before moving on to a more comprehensive example.

NOTE

Within this text, we use the term XREF only when referring to the specific sequence of characters (XREF) in a Ghidra listing, menu item, or dialog. We stick to the more general term cross-reference when referring to back references.

Example 1: Basic XREFs

Let's examine some basic XREFs in a function that accepts three parameters and declares four local variables. We have edited the following listing for clarity using the Edit Function dialog:

```
*****
*                                     *
*                                     FUNCTION                                     *
*                                     *
*****

void demo_xrefs(undefined param_1, undefined4. . .
    void          <void>          <RETURN>
    undefined      Stack[0x4]:4      param_1
    undefined4     Stack[0x8]:4      param_2 ❶ XREF[1]:    0804847f(R)
    undefined4     Stack[0xc]:4      param_3   XREF[1]:    08048479(R)
    undefined4     Stack[-0x10]:4     local_10  XREF[1]:    0804847c(W)
    undefined4     Stack[-0x14]:4     local_14  XREF[2]:    08048482(W),
                                                08048493(R)
    undefined4     Stack[-0x18]:4     local_18  XREF[2]:    08048485(W),
                                                08048496(R)
    undefined1     Stack[-0x58]:1     local_58  XREF[1]:    0804848c(W)
demo_xrefs                                XREF[4]:    Entry Point(*),
                                                ❷ main:080484be(c),
                                                080485e4, 08048690(*)
```

Ghidra notes that there is a cross-reference with the indicator XREF ❶ and shows the number of cross-references with an index value in brackets. We call this part of the cross-reference (for example, XREF[2]) the *XREF header*.

Examining the headers in the listing, we can see that most of the cross-references have only one referring address, but a few have more. After the header is the address associated with the cross-reference, which is a navigable object. Following the address is a type indicator in parentheses.

Data cross-references (like those in this example) have these valid types: R (indicating that the variable is read at the corresponding XREF address), W (indicating that the variable is being written), and * (indicating that an address of a location is being taken as a pointer). In summary, Ghidra identifies data cross-references in the listing where the data is declared, and associated XREF entries provide links to the locations where the data is referenced.

The listing also contains a code cross-reference, used to indicate that an instruction transfers or may transfer control to another instruction ❷. Code

cross-references are a very important concept, as they facilitate Ghidra's visualization of function calls, which are the focus of Chapter 10.

FORMATTING XREFS

As with most items you encounter in the Listing window, you can control the attributes associated with the cross-reference display. Selecting Edit ► Tool Options opens the editable options for the CodeBrowser. Since an XREF is part of the Listing window, you can find the XREFs Field within the Listing Fields folder. When selected, it will open the dialog shown in Figure 9-2 (here with the default options).

If you were to change the Maximum Number of XREFs to Display to 2, the header for all cross-references exceeding this number would be displayed as XREF[more]. The option to display nonlocal namespaces allows you to quickly identify all of the cross-references that are not within the current function's body. Ghidra Help explains all of these options.

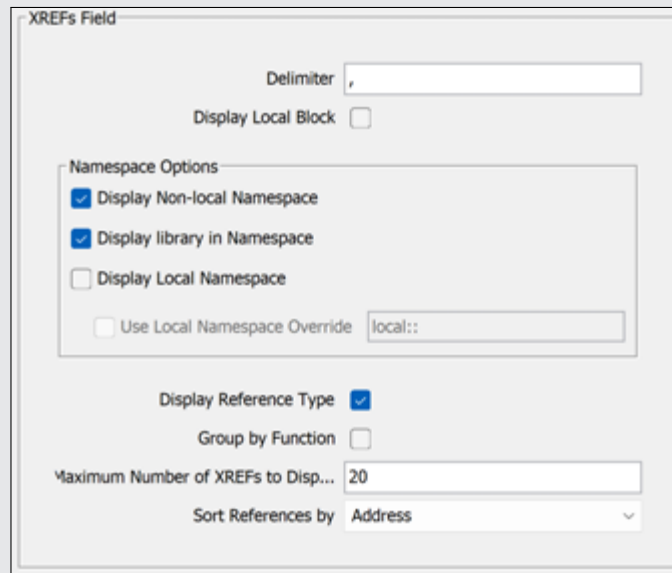


Figure 9-2: The XREFs Field edit window showing defaults

We refer to the manner in which instructions transfer control as a *flow*. Flows may be any of three basic types: sequential, jump, or call. A *sequential flow* is the simplest flow type, as it represents linear transfer of control from one instruction to the next. This is the default execution flow for all non-branching instructions, such as ADD. There are no special display indicators for sequential flows other than the order in which instructions are listed in the disassembly: If instruction A has a sequential flow to instruction B, then instruction B will immediately follow instruction A in the disassembly listing.

We can further classify jump and call flows according to whether the target address is a near or far address. Let's take a quick look at a new example containing code cross-references that demonstrate jumps and calls.

Example 2: Jump and Call XREFs

As with data cross-references, code cross-references also have an associated XREF entry in the Listing window. This next example shows a different listing containing information associated with the function `main`:

```
*****
*                               *
*                               FUNCTION                               *
*                               *****
undefined4 __stdcall main(void)
    undefined4 EAX:4             <RETURN>
    undefined4 Stack[-0x8]:4     ptr      ❶ XREF[3]: 00401014(W),
                                                0040101b(R),
                                                00401026(R)
main                               ❷ XREF[1]: entry:0040121e(c)
```

You can clearly identify the three XREFs associated with the stack variable ❶ as well as the XREF associated with the function itself ❷.

Let's decode the meaning of the XREF entry:0040121e(c). The address or identifier before the colon indicates the referring (or source) entity. In this case, control is transferred from `entry`. To the right of the colon is the specific address within `entry` that is the source of the cross-reference. The suffix (c) indicates that this is a CALL to `main`. Stated simply, the cross-reference says, "main is called from address 0040121e within entry."

If we double-click the cross-reference address to follow the link, we are taken to the specified address within entry where we can examine the call. While the XREF is a unidirectional link, we can quickly return to `main` by double-clicking the function name (`main`) or using the backward navigation arrow in the CodeBrowser toolbar:

```
0040121e CALL    main
```

In the following listing, the (j) suffix on the XREF indicates that this labeled location is the target of a jump instruction:

```
004011fe JZ      ❶ LAB_00401207
00401200 PUSH    EAX
00401201 CALL    __amsgr_exit
00401206 POP     ECX
                LAB_00401207                XREF[1]:❷ 004011fe(j)
00401207 MOV     EAX,[DAT_0040acf0]
```

As in the previous example, we can double-click the XREF address ❷ to navigate to the statement that transferred control and return by double-clicking the associated label ❶.

References in Practice

Let's walk through an example to demonstrate many types of cross-references. The following program, *simple_flows.c*, contains various operations that exercise Ghidra's cross-referencing features, as noted in the comment text:

```
int read_it;           // Integer variable read in main
int write_it;          // Integer variable written 3 times in main
int ref_it;            // Integer variable whose address is taken in main
void callflow() {}     // Function called twice from main

int main() {
    int *ptr = &ref_it; // Results in a "pointer" style data reference (*)
    *ptr = read_it;     // Results in a "read" style data reference (R)
    write_it = *ptr;    // Results in a "write" style data reference (W)
    callflow();         // Results in a "call" style code reference (c)
    if (read_it == 3) { // Results in a "jump" style code reference (j)
        write_it = 2;  // Results in a "write" style data reference (W)
    }
    else {              // Results in a "jump" style code reference (j)
        write_it = 1;  // Results in a "write" style data reference (W)
    }
    callflow();         // Results in a "call" style code reference (c)
}
```

We'll demonstrate how each of these features of the source code shows up as cross-references in the disassembly in Ghidra.

Code Cross-References

Listing 9-1 shows a disassembly of the preceding program. We have renamed the disassembly variables to be consistent with source code.

undefined4 __stdcall main(void)		
undefined4 EAX:4 <RETURN>		
undefined4 Stack[-0x8]:4 ptr		XREF[3]: 00401014(W), 0040101b(R), 00401026(R)
main		XREF[1]: entry:0040121e(c)
00401010	PUSH EBP	
00401011	MOV EBP,ESP	
00401013	PUSH ECX	
❶ 00401014	MOV dword ptr [EBP + ptr],ref_it	
0040101b	MOV EAX,dword ptr [EBP + ptr]	
❷ 0040101e	MOV ECX,dword ptr [read_it]	
00401024	MOV dword ptr [EAX]=>ref_it,ECX	
00401026	MOV EDX,dword ptr [EBP + ptr]	
00401029	MOV EAX=>ref_it,dword ptr [EDX]	
0040102b	MOV [write_it],EAX	


```

③ 00401030 CALL    callflow
    00401035 CMP     dword ptr [read_it],3
    0040103c JNZ     LAB_0040104a
    0040103e MOV     dword ptr [write_it],2
④ 00401048 JMP     LAB_00401054
        LAB_0040104a                                XREF[1]: ③ 0040103c(j)
    0040104a MOV     dword ptr [write_it],1
        LAB_00401054                                XREF[1]: ③ 00401048(j)
    00401054 CALL    callflow
    00401059 XOR     EAX,EAX
    0040105b MOV     ESP,EBP
    0040105d POP     EBP
⑦ 0040105e RET

```

Listing 9-1: Disassembly of simple_flows.c

Every instruction other than JMP ④ and RET ⑦ has an associated sequential flow to its immediate successor. Instructions used to invoke functions, such as the x86 CALL instruction ③, are assigned a call flow, indicating transfer of control to the target function. Ghidra indicates call flows with XREFs at the target function (the destination address of the flow). Listing 9-2 shows the disassembly of the callflow function referenced in Listing 9-1.

```

        undefined __stdcall callflow(void)
        undefined <UNASSIGNED> <RETURN>
        callflow                                XREF[4]: 0040010c(*),
                                                004001e4(*),
                                                main:00401030(c),
                                                main:00401054(c)

00401000 PUSH    EBP
00401001 MOV     EBP,ESP
00401003 POP     EBP
00401004 RET

```

Listing 9-2: Disassembly of the callflow function

In this example, we see that callflow is called twice from main: once from address 00401030 and again from address 00401054. Cross-references resulting from function calls are distinguished by the suffix (c). The source location displayed in the cross-references indicates both the address from which the call is being made and the function that contains the call.

A jump flow is assigned to each unconditional and conditional branch instruction. Conditional branches are also assigned sequential flows to account for control flow when the branch is not taken; unconditional branches have no associated sequential flow because the branch is always taken. Jump flows are associated with jump-style cross-references displayed at the targets of the

JNZ ⑤ and JMP ⑥ instructions in Listing 9-1. As with call-style cross-references, jump cross-references display the address of the referring location (the source of the jump). Jump cross-references are distinguished by the (j) suffix.

EXTRA XREFS?

Every now and again, you see something in a listing that seems anomalous. In Listing 9-2, we can easily trace two of the XREFs back to the calls to `callflow` in `main`, but the two other pointer XREFs, `0040010c(*)` and `004001e4(*)`, are not easily explained. What are they? It turns out that these are an interesting artifact of this particular code. This program was compiled for Windows, which results in a PE file, and the two anomalous XREFs take us to the PE header in the Headers section of the listing. The two reference addresses (including the associated bytes) are shown here:

0040010c	00 10 00 00 ibo32	callflow	BaseOfCode
	. . .		
004001e4	00 10 00 00 ibo32	callflow	VirtualAddress

Why is this function referenced in the PE header? A quick online search can help us understand what is happening: `callflow` just happens to be the very first thing in the text section, and the two PE fields indirectly reference the start of the text section, hence the unanticipated XREFs associated with `callflow`.

There are logical limits on what can be cross-referenced, primarily in the context of basic blocks. A *basic block* is a maximal sequence of instructions that executes, without branching, from beginning to end. Each basic block therefore has a single entry point (the first instruction in the block) and a single exit point (the last instruction in the block). The first instruction in a basic block is often the target of a branching instruction, while the last instruction is often a branch instruction. The first instruction may be the target of multiple code cross-references. Other than the first instruction in the block, no other instruction within a basic block can be the target of a code cross-reference. The last instruction of a basic block may be the source of multiple code cross-references, such as a conditional jump, or it may flow into an instruction that is the target of multiple code cross-references (which, by definition, must begin a new basic block). We will investigate basic blocks further in Chapter 10.

Data Cross-References

Data cross-references track how data is accessed within a binary. The three most commonly encountered types of data cross-references indicate when a location is being read, when a location is being written, and when the address of a location is being taken. Listing 9-3 shows the global variables from the previous sample program, as they provide several examples of data cross-references.

	read_it		XREF[2]:	main:0040101e(R), main:00401035(R)
0040b720	undefined4	??		
	write_it		XREF[3]:	main:0040102b(W), main:0040103e(W), main:0040104a(W)
0040b724	??	??		
0040b725	??	??		
0040b726	??	??		
0040b727	??	??		
	ref_it		XREF[3]:	main:00401014(*), main:00401024(W), main:00401029(R)
0040b728	undefined4	??		

Listing 9-3: Global variables referenced in simple_flows.c

A *read cross-reference* indicates that the content of a memory location is being read. Read cross-references can originate only from an instruction address but may refer to any program location. The global variable `read_it` is read twice in Listing 9-1. The associated cross-reference comments shown in this listing indicate exactly which locations in `main` are referencing `read_it` and are recognizable as read cross-references from the (R) suffix. The read performed on `read_it` ❷ in Listing 9-1 is a 32-bit read into the ECX register, which leads Ghidra to format `read_it` as an `undefined4` (a 4-byte value of unspecified type). Ghidra often attempts to infer the size of a data item based on how the item is manipulated by code throughout a binary.

The global variable `write_it` is referenced three times in Listing 9-1. Associated *write cross-references* are generated and displayed as comments for the `write_it` variable, indicating the program locations that modify the contents of the variable. Write cross-references utilize the (W) suffix. In this case, Ghidra did not format `write_it` as a 4-byte variable even though there seems to be enough information to do so. As with read cross-references, write cross-references can originate only from a program instruction but may reference any program location. Generally, a write cross-reference that targets a program instruction byte is indicative of self-modifying code and is frequently encountered in malware deobfuscation routines.

The third type of data cross-reference, a *pointer cross-reference*, indicates that the address of a location is being used (rather than the content of the location). The address of global variable `ref_it` is taken ❶ in Listing 9-1, resulting in the pointer cross-reference at `ref_it` in Listing 9-3, as indicated by the suffix (*). Pointer cross-references are commonly the result of address derivations either in code or in data. As you saw in Chapter 8, array access operations are typically implemented by adding an offset to the base address of the array, and the first address in most global arrays can often be recognized by the presence of a pointer cross-reference. For this reason, most string literals (strings being arrays of characters in C/C++) are the targets of pointer cross-references.

Unlike read and write cross-references, which can originate only from instruction locations, pointer cross-references can originate from either instruction locations or data locations. An example of pointers that can originate from a program's data section is any table of addresses (such as a vtable, which results in the generation of a pointer cross-reference from each entry in the table to the corresponding virtual function). Let's see this in context using the SubClass example from Chapter 8. The disassembly for the vtable for SubClass is shown here:

```

                SubClass::vtable          XREF[1]: SubClass_Constructor:00401062(*)
00408148 void * SubClass::vfunc1 vfunc1
❶ 0040814c void * BaseClass::vfunc2 vfunc2
00408150 void * SubClass::vfunc3 vfunc3
00408154 void * BaseClass::vfunc4 vfunc4
00408158 void * SubClass::vfunc5 vfunc5

```

Here you see that the data item at location 0040814c ❶ is a pointer to BaseClass::vfunc2. Navigating to BaseClass::vfunc2 presents us with the following listing:

```

*****
*                                     *
*                                     FUNCTION                                     *
*                                     *
*****
undefined __stdcall vfunc2(void)
    undefined    <UNASSIGNED>    <RETURN>
    undefined4   Stack[-0x8]:4   local_8      XREF[1]:    00401024(W)
    BaseClass::vfunc2              XREF[2]: ❶ 00408138(*),
                                      ❷ 0040814c(*)
00401020 PUSH    EBP
00401021 MOV     EBP,ESP
00401023 PUSH    ECX
00401024 MOV     dword ptr [EBP + local_8],ECX
00401027 MOV     ESP,EBP
00401029 POP     EBP
0040102a RET

```

Unlike most functions, this function has no code cross-references. Instead, we see two pointer cross-references indicating that the address of the function is derived in two locations. The second XREF ❷ refers back to the SubClass vtable entry discussed earlier. Following the first XREF ❶ would lead us to the vtable for BaseClass, which also contains a pointer to this virtual function.

This example demonstrates that C++ virtual functions are rarely called directly and are usually not the target of a call cross-reference. Because of the way vtables are created, all C++ virtual functions will be referred to by at least one vtable entry and will always be the target of at least one pointer cross-reference. (Recall that overriding a virtual function is not mandatory.)

When a binary contains sufficient information, Ghidra is able to locate vtables for you. Any vtables that Ghidra finds are listed as an entry under the vtable's corresponding class entry within the *Classes* folder of the Symbol Tree. Clicking a vtable in the Symbol Tree window navigates you to the vtable location in the program's data section.

Reference Management Windows

By now, you have probably noticed that XREF annotations are quite common in the Listing window. This is no accident, as the links formed by cross-references are the glue that holds a program together. Cross-references tell the story of intra- and interfunctional dependencies, and most successful reverse engineering efforts demand a comprehensive understanding of their behavior. The sections that follow move beyond the basic display and navigational usefulness of cross-references to introduce several options for managing cross-references within Ghidra.

XRefs

While clicking an XREF address will take you to that address, you can use XREF headers to learn more about a particular cross-reference, as shown in the following listing:

undefined4 Stack[-0x10]:4 local_10	XREF[1]:	0804847c(W)
undefined4 Stack[-0x14]:4 local_14	❶ XREF[2]:	08048482(W), 08048493(R)

Double-clicking the second XREF header ❶ will bring up the associated XRefs window shown in Figure 9-3 with a more detailed listing of the cross-references.

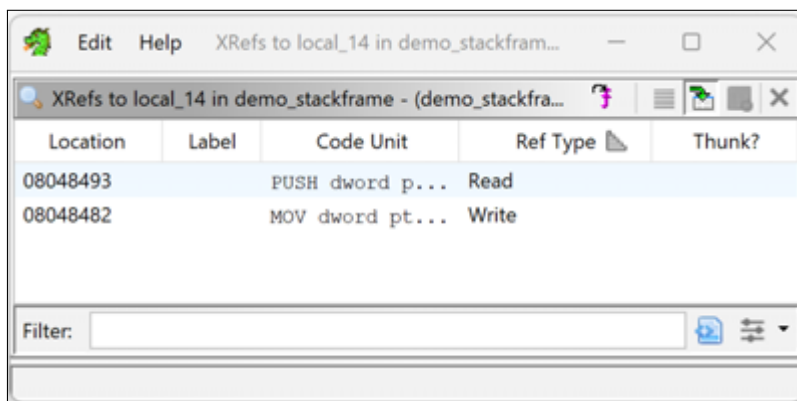


Figure 9-3: The XRefs window

By default, the window shows the location, the label (if applicable), the reference disassembly, and the reference type.

References To

Another window that can help you understand the program flow is the References To window. Right-clicking any address in the Listing window and choosing References ► Show Reference to Address brings up the window shown in Figure 9-4.

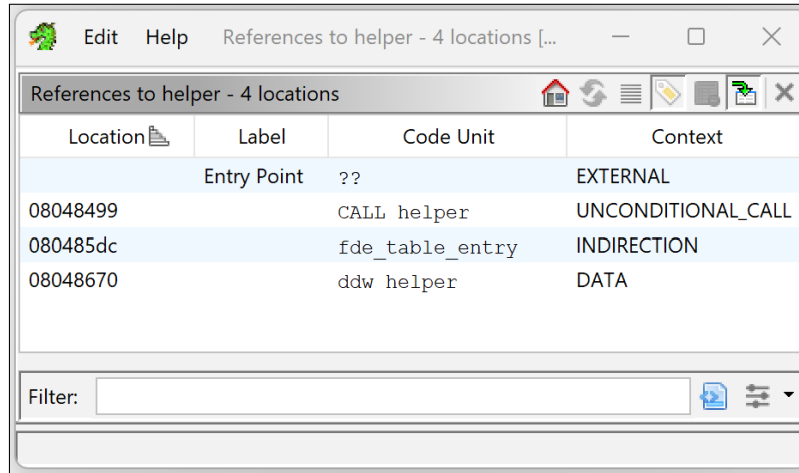


Figure 9-4: The References To window

In this example, we have selected the starting address of the helper function. Within this window, you can navigate to the associated location by clicking any entry in the window.

Symbol References

Together, the Symbol Table and Symbol Reference windows, introduced in Chapter 5, provide another reference view. By default, when you choose Window ► Symbol References, you get two related windows. One displays every symbol in the entire symbol table. The other displays the associated references to the symbols. Selecting any entry in the Symbol Table window (function, vftable, and so on) causes the associated symbol references to be displayed in the Symbol References window.

Using reference lists, you can quickly identify every location from which a particular function is called. For example, many people consider the C `strcpy` function to be dangerous, as it copies a source array of characters, up to and including the associated null termination character, to a destination array, with no checks whatsoever that the destination array is large enough to hold all of the characters from the source. Say you want to check a binary for the use of this function. Instead of taking the time to find `strcpy` used somewhere in the binary, you can open the Symbol References window and quickly locate `strcpy` and all associated references.

Advanced Reference Manipulation

At the beginning of this chapter, we briefly mentioned that Ghidra also has *forward references*, of which there are two types. *Inferred forward references* are generally added to the listing automatically and correspond one-for-one to back references, except they are traveled in the opposite direction. In other words, we traverse back references from a target address back to a source address, and we traverse inferred forward references from a source address forward to a target address.

The second type is an *explicit forward reference*. There are several types of explicit forward references, and their management is much more complex than other cross-references. The types of explicit forward references include memory references, external references, stack references, and register references. In addition to viewing references, Ghidra allows you to add and edit a variety of reference types.

You may need to add your own cross-references when Ghidra's static analysis cannot determine jump or call targets that are computed at run-time, but you know the target from other analysis. The following code calls a virtual function:

	0001072e	PUSH	EBP
	0001072f	MOV	EBP,ESP
	00010731	SUB	ESP,8
❶	00010734	MOV	EAX,dword ptr [EBP + param_1]
	00010737	MOV	EAX,dword ptr [EAX]
	00010739	ADD	EAX,8
	0001073c	MOV	EAX,dword ptr [EAX]
	0001073e	SUB	ESP,12
	00010741	PUSH	dword ptr [EBP + param_1]
❷	00010744	CALL	EAX
	00010746	ADD	ESP,16
	00010749	NOP	
	0001074a	LEAVE	
	0001074b	RET	

The value held in EAX ❷ depends on the value of the pointer passed in param_1 ❶. As a result, Ghidra does not have enough information to create a cross-reference linking 00010744 (the address of the CALL instruction) to the target of the call. Manually adding a cross-reference (to SubClass::vfunc3, for example) would, among other things, link the target functions into a call graph, thereby improving Ghidra's analysis of the program.

Right-clicking the call ❷ and selecting References ► Add Reference From opens the dialog shown in Figure 9-5. This dialog is also available through the References ► Add/Edit option.

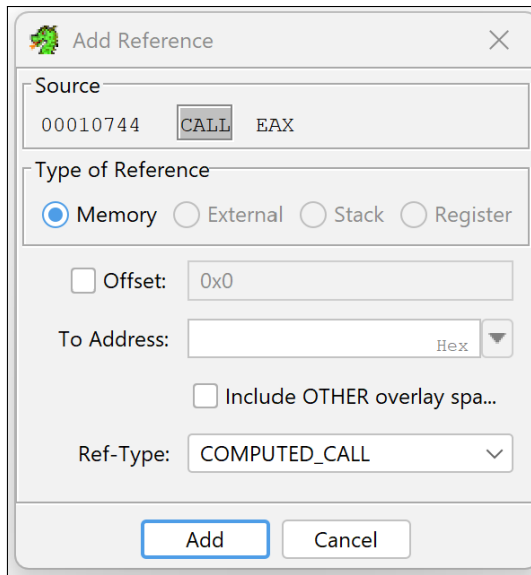


Figure 9-5: The Add Reference dialog

Specify the address of the target function as the To Address setting and select the correct setting for Ref-Type. When you close the dialog with the Add button, Ghidra creates the reference, and a new (c) cross-reference appears at the target address. You can find more information on forward references, including the remaining reference types and reference manipulation techniques, in Ghidra Help.

Summary

References are powerful tools to help you understand how artifacts within a binary are related. We discussed cross-references in detail and introduced some other capabilities associated with references that we will visit again in later chapters. In the next chapter, we look at visual representations of references and how the resulting graphs can help us better understand the control flows within functions and the relationships between functions in our binaries.

10

USING GRAPH VIEWS



Visually representing data with graphs can help us recognize patterns in the data that might otherwise be difficult to discover. In addition to its disassembly and decompiler perspectives, Ghidra offers graph views, which can give us a new perspective on the contents of a binary.

These graphs let you quickly see the control flow in a function and the relationships between functions in a file, by representing functions and other types of blocks as nodes and by representing flows and cross-references as edges (the lines that connect nodes). With enough practice, you may find that common control structures, such as switch statements and nested if-else structures, are easier to recognize in graph form than in a long text listing. In Chapter 5, we briefly introduced the Function Graph and Function Call Graph windows. In this chapter, we take a deeper dive into Ghidra's graph capabilities.

Because cross-references relate one address to another, they are a natural place to begin graphing our binaries. By restricting ourselves to sequential flows and specific types of cross-references, we can derive a number of useful graphs for analysis. While the flows and cross-references serve as the edges in our graphs, the meaning behind nodes can vary. Depending on the type of graph we wish to generate, nodes may contain one or more instructions, or entire functions. Let's start our discussion about graphs by looking

at the ways that Ghidra organizes code into *blocks* and then move on to the types of graphs available in Ghidra.

Basic Blocks

In a computer program, a *basic block* is a grouping of one or more instructions with a single entry at the beginning of the block and a single exit from the end of the block. Other than the last instruction, every instruction within a basic block transfers control to exactly one *successor* instruction within the block. Similarly, other than the first instruction, every instruction in a basic block receives control from exactly one *predecessor* instruction within the block. In “Cross-References (Back References)” on page 187, we identified this as *sequential flow*.

From time to time, you may notice a function call being made in the middle of a basic block and think to yourself, “Isn’t this precisely the type of instruction, like a jump, that should terminate a block?” For the purposes of basic block determination, the fact that function calls transfer control outside the current block is generally ignored unless it is known that the function being called does not return normally.

Once the first instruction in a basic block is executed, the remainder of the block is guaranteed to execute to completion. This can factor significantly into runtime instrumentation of a program since it is no longer necessary to set a breakpoint on every instruction in a program or even single-step the program in order to record which instructions have executed. Instead, you can set breakpoints on the first instruction of each basic block, and as each breakpoint is hit, you can assume that every instruction in the associated block will be executed. Let’s shift our focus to Ghidra’s Function Graph capabilities to provide another perspective on blocks.

Function Graphs

The Function Graph window, introduced in Chapter 5, displays a single function in a graphical format. The following program consists of a single function composed of a single basic block, and we’ll use it as a starting point to demonstrate Ghidra’s function graphs:

```
int global_array[3];
int main() {
    int idx = 2;
    global_array[0] = 10;
    global_array[1] = 20;
    global_array[2] = 30;
    global_array[idx] = 40;
}
```

When you open the Function Graph window (Window ► Function Graph) with `main` selected, you will see a function graph with only one basic block, as shown in Figure 10-1.

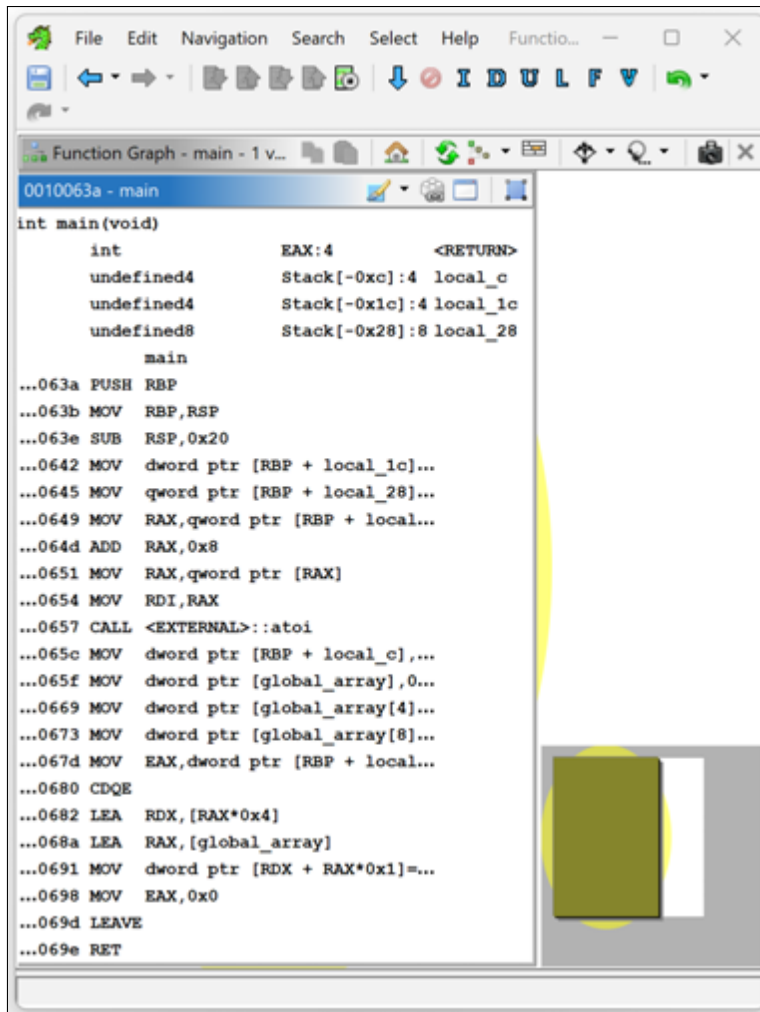


Figure 10-1: A single-block Function Graph window with a satellite view at the lower right

The Function Graph window and the Listing window have a useful bi-directional link. If you view the windows side by side, the concurrent listing and graphical representation can help you better understand the function's control flow. Changes you make in the Function Graph window (for example, renaming functions or variables) will be immediately reflected in the Listing window. Changes you make in the Listing window will also be reflected in the Function Graph window, although you may have to refresh the window to see the change.

If you click any line of text in the Function Graph window, the cursor in the Listing window will move to the corresponding location in the disassembly listing. If you double-click data in a function graph, the Listing window will navigate to the associated data in the data section of the listing, while the Function Graph window will retain its focus on the function.

WHAT IS AN INTERACTION THRESHOLD?

When interacting with the Function Graph window, particularly with a complex function, you may zoom out because you cannot see everything you want to see. When the individual nodes become too small to interact with in a meaningful way, you have passed the *interaction threshold*. Ghidra uses drop shadows on each node in the Function Graph to indicate this condition. When it happens, virtual addresses may show only the least significant values, and the sheer number of nodes in the graph display can become unwieldy. An attempt to select content within a node ends up selecting the entire block. Don't despair if the complexity of your function pushes you beyond this threshold. You can click any of the nodes to bring them into focus, or double-click a node to zoom in on it.

Using Graph Toolbars

A function graph is really nothing more than a graphical presentation of the Listing window isolated to a single function, so it should not be surprising that you can access all of the CodeBrowser's menus in the Function Graph window, with the exception of the Window menu.

ARTICULATION

As your functions become more complex, the number of blocks in each will likely increase. When you first generate a function graph, the edges connecting the blocks are articulated. This means that they bend neatly at 90-degree angles so that they are not hidden behind nodes. This results in a neat grid layout where all components of all edges are either horizontal or vertical.

If you decide to change the layout of the graph by dragging nodes around, the edges may lose their articulation and revert to straight lines that route behind other nodes in the graph. Figure 10-2 demonstrates the contrast between the articulated representation on the left and the unarticulated version on the right. You can revert to the original layout at any time by refreshing the Function Graph window.

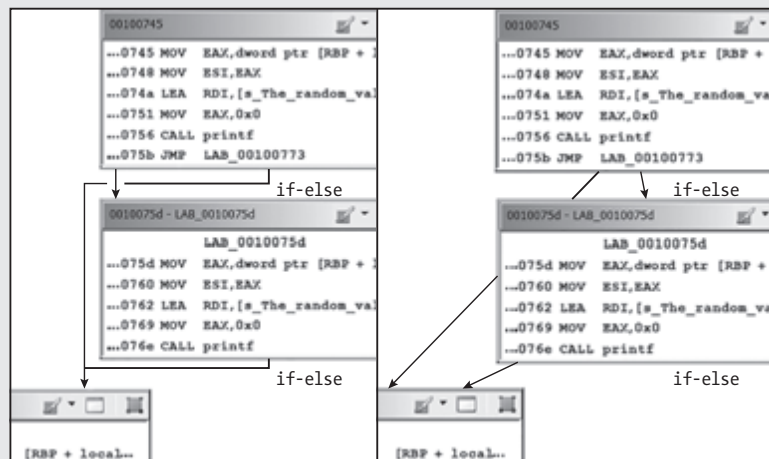


Figure 10-2: A function graph with articulated (left) and unarticulated (right) edges

The available toolbar lets you save the current state of the open file, undo and redo changes, and navigate forward and backward within the current navigation chain. It is important to note that, since the windows are linked, this may navigate you out of (and back into) the current function, which will change the contents of the Function Graph window. Figure 10-3 describes the Function Graph toolbar icons and their default behaviors.

	Copy to Ghidra Clipboard	This functionality is available in a number of Ghidra windows and varies based on the window in which they appear as the content that is selected when the operation is activated. In some cases with incompatible content, you will see an error message.
	Paste to Ghidra Clipboard	
	Go to Function Entry Point	This button takes you to the entry point block in the function graph.
	Reload Graph	When you reload a graph, all positioning and group information is lost. The operation reverts to the default view.
	Nested Code Layout	Nested code layout allows you to retain the grouping information while changing the associated layout.
	Edit Code Block Fields	This edits the code block fields in the function graph. It does not affect the code block fields in the Listing window.
	Block Hover Mode	These options help you to control the appearances of the graph as you are exploring the flow of control. The block that is currently selected is the focus block. The block that the mouse is hovering over is the hover block. This functionality allows you to closely examine relationships between blocks.
	Block Focus Mode	
	Snapshot	This button creates and opens a disconnected copy of the current Function Graph window that is not linked to the listing.

Figure 10-3: The Function Graph toolbar operations

Each basic block also has a toolbar that lets you modify the block and group it with other blocks by combining several blocks (vertices) into a single block. Figure 10-4 provides an explanation of the toolbar's icons and their default behaviors.

	Background Color	Select a background color for a block or group of blocks. This color is reflected in the Function Graph window as well as the Listing window.
	Jump to XREF	This button displays a list of cross-references to the entry point of the function.
	Fullscreen/ Graph View	This is a toggle button that allows you to view the graph block in a full window.
	Combine Vertices	This button will combine selected vertices into a single group.
	Restore Group	This option is displayed only after you have ungrouped vertices, and it provides the option to regroup them.
	Ungroup Vertices	This option is available only if vertices have been grouped and allows you to ungroup the vertices.
	Add Vertex to Group	This option is available only if vertices have been grouped and allows you to add a vertex to a group.

Figure 10-4: The Function Graph basic block toolbar

This toolbar is extremely useful for reducing the complexity of graphs that results from highly nested functions. For example, you might elect to collapse all of the blocks nested within a loop statement into a single graph node after you understand the behavior of the loop and feel less need to see the code within the loop. Depending on the number of nested blocks that you group, the grouping might significantly enhance the graph's readability.

To group nodes, select all nodes that will belong to the group by using CTRL-click, and then select the Combine Vertices tool of the node you consider to lie at the root of the group. Restore Group is a particularly helpful button that lets you quickly look inside a group and then re-collapse it.

While you will see some changes in the listing immediately reflected in the Function Graph window, in other cases the graph can become *stale*, meaning not synchronized with the Listing view. When this happens, Ghidra displays the message shown in Figure 10-5 at the bottom of the graph window.



Figure 10-5: The stale graph warning message

The recycle icon to the left of the message allows you to refresh the graph without reverting to the original layout. (Of course, you can also choose to refresh and lay out again.)

Splitting Blocks

If you have a particularly long basic block and wish to break it into smaller blocks, or wish to visually isolate a section of code for further analysis, you can split a basic block within a function graph by introducing new labels into the block. For example, say we use the hotkey L to insert a new label at line 00100651 in Figure 10-6, before the `MOV RDI,RAX`. Ghidra splits the block being split and shows the function graph.

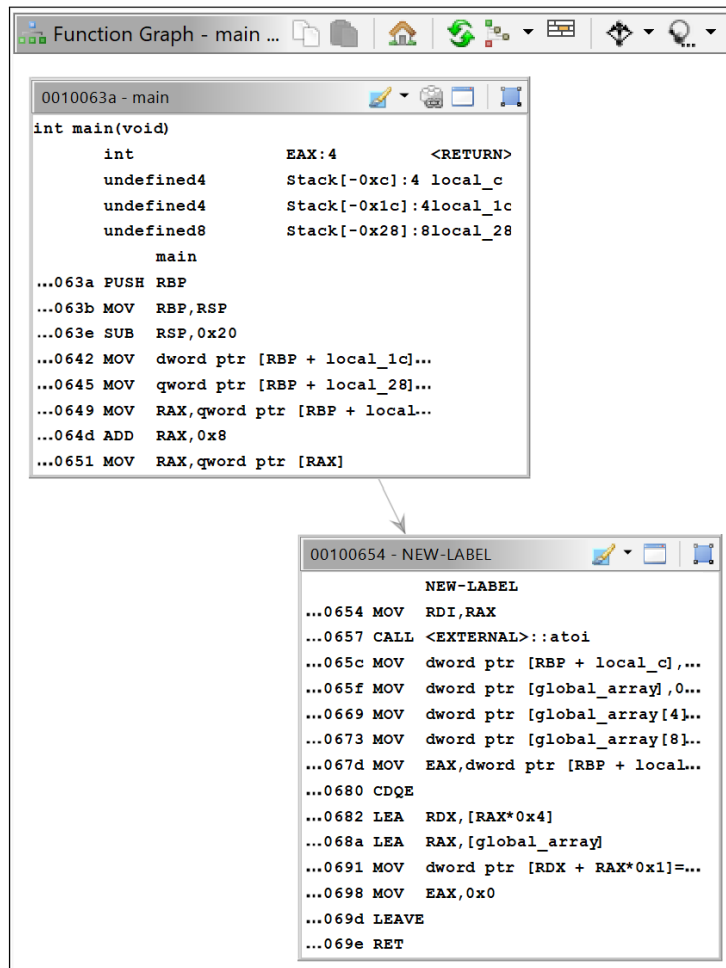


Figure 10-6: A function graph with a new label introducing a new basic block

The new edge represents flow and is not associated with a cross-reference.

Interacting with Function Graphs

While it isn't easy to show in a book, the Function Graph window displays color, animation, and informational pop-ups as you interact with the various components in the graph:

Edges Ghidra colors the edges based on the nature of the transition represented by the edge. You can control the colors through the Edit ► Tool Options ► Graph ► Function Graph window, as shown in Figure 10-7.

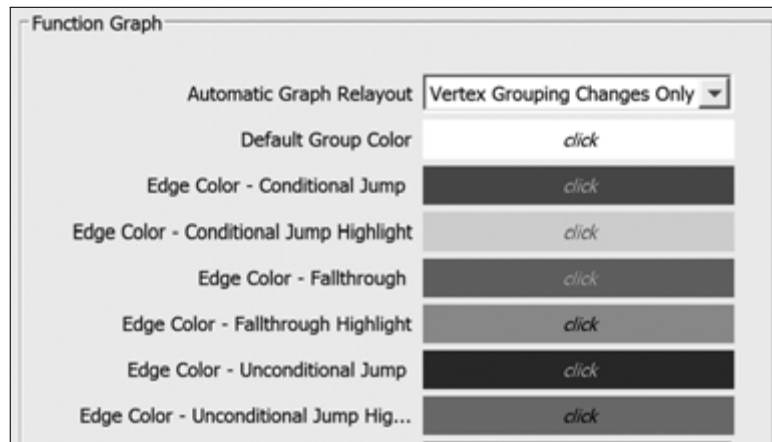


Figure 10-7: The Function Graph color customization options

By default, a green edge indicates a conditional jump when the condition is true (meaning the jump is taken), a red edge indicates a fallthrough (meaning the jump is not taken), and a blue edge indicates an unconditional jump. Clicking an individual edge or set of edges increases the thickness of the edge and changes it to a highlighted shade of the same color.

Nodes The content of each node is a disassembly listing of the corresponding basic block. The way you interact with the listing code is identical to the way you interact with code in the Listing window. For example, hovering over names opens a pop-up that displays disassembly at the named location. When you hover over a node, Ghidra utilizes a path-highlighting animation on associated edges to indicate the control flow direction consistent with the currently selected path highlight options. You can disable this functionality in Edit ► Tool Options.

Satellite The satellite (a small overview of the graph) has a yellow halo around the block currently in focus, as does the Function Graph window. For easy identification, the function's entry block (which contains the function's entry point address) is green in the satellite, and any return blocks (blocks containing a RET, or equivalent) are red. Even if you change the background color of the associated block in the graph, the entry and exit colors don't change in the satellite. All other blocks will mirror the color assigned to them in the Function Graph window.

Function Call Graphs

A function call graph can help you quickly gain an understanding of the hierarchy of function calls made within a program. Function call graphs are similar to function graphs, but each node represents an entire function body, and each edge represents a call cross-reference from one function to another.

To discuss function call graphs, we will make use of the following trivial program, which creates a simple hierarchy of function calls:

```
#include <stdio.h>
void depth_2_1() {
    printf("inside depth_2_1\n");
}
void depth_2_2() {
    fprintf(stderr, "inside depth_2_2\n");
}
void depth_1() {
    depth_2_1();
    depth_2_2();
    printf("inside depth_1\n");
}
int main() {
    depth_1();
}
```

After compiling a dynamically linked version of this program using GNU gcc and loading the binary with Ghidra, we can generate a function call graph by using Window ► Function Call Graph. By default, this creates a function call graph centered around the function currently selected. Figure 10-8 shows the function call graph generated when `main` is selected. (The satellite view is hidden in these examples for clarity. To show it, use the icon in the bottom right of the figure.)

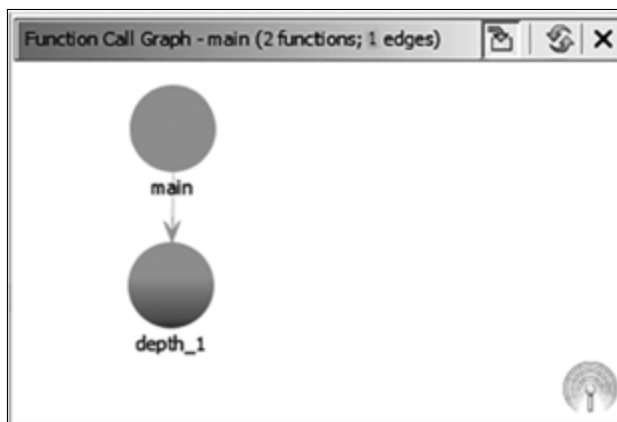


Figure 10-8: A simple function call graph with a focus on `main`

The string *main (2 functions; 1 edges)* in the title bar of the graph lets us know what function we are in, along with the number of functions and edges displayed. Hovering over a node in the graph displays + and/or – icons at the top and/or bottom of the node, as shown in Figure 10-9.

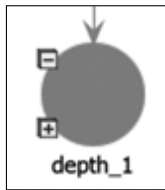


Figure 10-9: A function call graph node with expand/collapse icons

A + icon at the top or bottom means you can display additional incoming or outgoing functions. Conversely, the – icon provides the ability to contract nodes. For instance, clicking a – symbol at the bottom of the function *depth_1* when it is expanded will change the function call graph from the one shown in Figure 10-10 to the one in Figure 10-8.

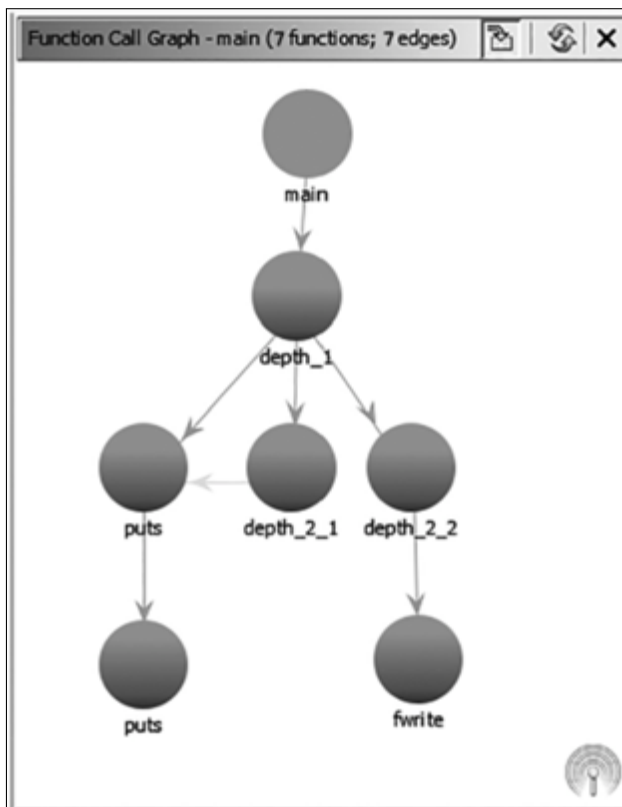


Figure 10-10: The function call graph expanded from *main*

The right-click context menu associated with each node provides you with options to expand or contract all outgoing edges for all nodes on the same horizontal level simultaneously. This is equivalent to clicking the + or - icon on all nodes in the same rank at the same time. Finally, double-clicking a node in the graph centers the graph on the selected node and fully expands all incoming and outgoing edges. (Careful readers may notice that the compiler has substituted calls to `puts` and `fwrite` for `printf` and `fprintf`, respectively, as they are more efficient when printing static strings.)

One option disabled by default, but that many find helpful, provides you with the ability to zoom out and back in. You can enable this option through Edit ► Tool Options by checking the Scroll Wheel Pans option. Ghidra maintains a short history of graphs in a cache as you shift focus to retain graph state upon return. This allows you to expand and contract nodes, navigate away, and then return to find your graph the way you left it so you can continue your analysis.

Now, let's revisit a statically linked version of the `call_tree` program. The initial graph generated from the `main` function is identical to the dynamically linked version shown in Figure 10-9, but graphs associated with statically linked binaries have the potential to become much more complex. To demonstrate this fact, let's investigate two expansions that seem relatively benign. Figure 10-11 shows the outgoing calls from the `puts` function. The title bar shows *puts (9 functions; 11 edges)*. Note that the title bar totals may be inaccurate until the program has been fully analyzed.

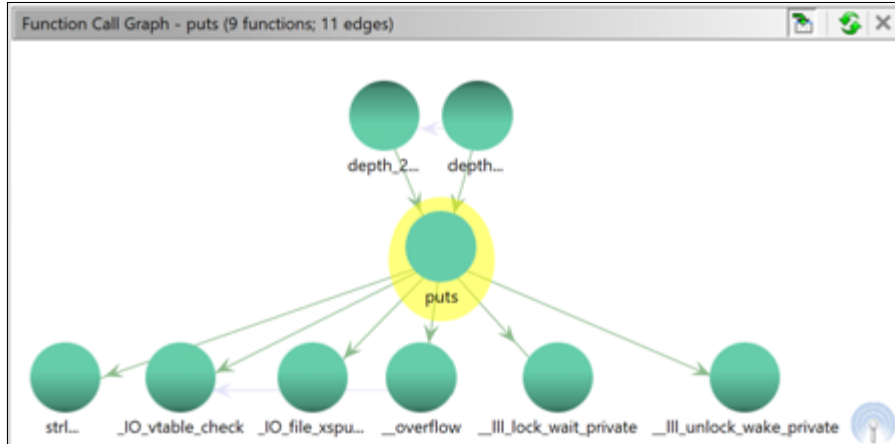


Figure 10-11: The function call graph in a statically linked binary

When we shift the focus to `__lll_lock_wait_private`, we are presented with an overwhelming graph with 70 nodes and over 200 edges, a portion of which is shown in Figure 10-12.

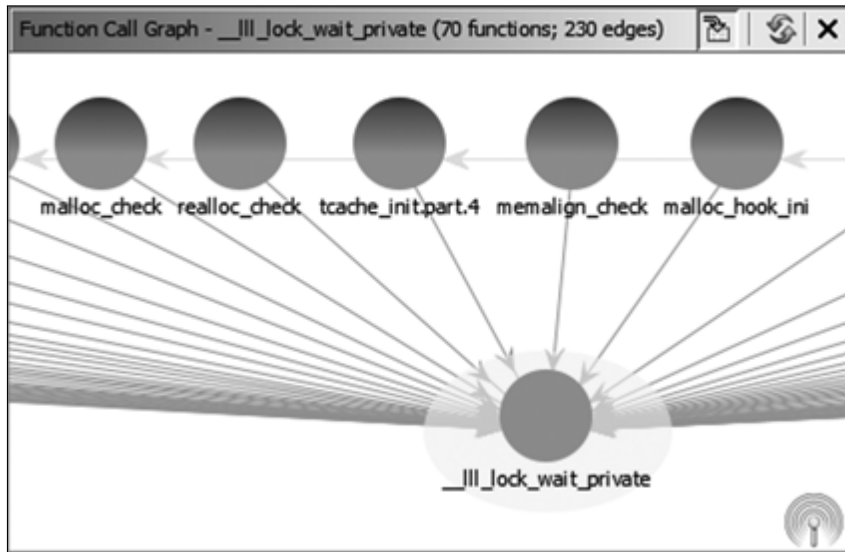


Figure 10-12: The expanded function call graph in a statically linked binary

While statically linked binaries are complex, working with the associated graphs can be challenging. Two features make these graphs easier to navigate. First, you can usually locate `main` by using the hotkey `G` or by navigating from the program’s entry symbol. Once you have located `main` in the listing, you can open and easily control what the associated function call graph displays.

Trees

Ghidra presents many hierarchical concepts associated with a particular binary as a treelike structure. While not always “trees” in a pure graph-theoretical sense, these structures provide the capability to expand and collapse nodes and to see the hierarchical relationship between nodes of varying types. When we discussed the CodeBrowser window in Chapter 5, you were introduced to Program Trees, Symbol Trees, Function Call Trees, and the Data Type Manager (which is also presented as a tree). You can use these tree views concurrently with other views to gain additional insight into the binary you are analyzing.

THUNK

You may notice that the graph in Figure 10-10 shows multiple (apparently recursive) calls to `puts`. Welcome to the magical world of thunk functions. A *thunk function* is a compiler device that facilitates calls to functions whose address is unknown at compile time (such as a dynamically linked library function). Ghidra refers to the function whose address is unknown as the *thunked* function.

The compiler replaces all calls the program makes to thunked functions with a call to a thunk function stub that the compiler inserts into the executable. The *thunk function stub* typically performs a table lookup to learn the runtime address of the thunked function before transferring control to the thunked function. The table consulted by a thunk stub is populated at runtime after the associated thunked function addresses become known. In Windows executables, this table is typically called the *import table*. In ELF binaries, it is typically called the *global offset table*, or *got*.

If we navigate to `puts` from the function `depth_1` in the Listing window, we find ourselves in the following listing:

```
*****
*                                     *
*                               THUNK FUNCTION                               *
*                                     *
*****

    thunk int puts(char * __s)
        Thunked-Function: <EXTERNAL>::puts
int          EAX:4          <RETURN>
char *       RDI:8          __s
        puts@GLIBC_2.2.5
        puts      XREF[2]: puts:00100590(T),
                                puts:00100590(c), 00300fc8(*)
00302008          ??          ??
00302009          ??          ??
0030200a          ??          ??
```

This thunk function listing appears in a program section that Ghidra names `EXTERNAL`. Ghidra thunked function listings such as this are a consequence of the way in which external libraries are dynamically loaded and linked into processes at runtime, which means the libraries are generally not available during static analysis. While the listing provides you with an indication of the function being called and the library in which it resides, the function code is not directly accessible (unless the library is also loaded into Ghidra, which is easily accomplished via the options page during the import process).

Here we also observe a new type of XREF. The (T) suffix on the first XREF indicates that this XREF is a link to the thunked function.

The Graph Menu

As reverse engineers, we often find ourselves knee-deep in disassembly listings, tracking jump instructions and call sites while trying to mentally reconstruct control flow and data relationships. While the information in the Listing window is powerful, sometimes what we really need is a visual map of what's going on, and that's exactly what the Graph menu provides.

Function Graphs and Function Call Graphs have been core features of Ghidra since its initial public release, providing essential views of control flow within individual functions and the relationships between them. In addition to these tools, CodeBrowser now includes a dedicated Graph menu that significantly extends Ghidra's visual analysis capabilities.

This menu, shown in Figure 10-13, is organized into several functional sections and offers a range of supplemental graphing tools. These tools allow you to generate custom, interactive visualizations of control flow, call relationships, and data references within specific regions of code or data. Each graph type serves a slightly different purpose, giving you multiple ways to interpret and navigate complex binaries.

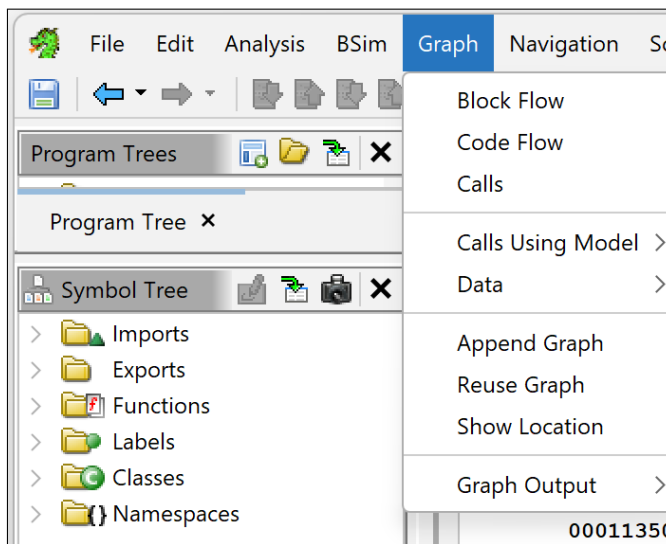


Figure 10-13: The CodeBrowser Graph menu

Let's explore these additional graph capabilities and examine how you can effectively integrate them into your reverse engineering practices.

Graph Services

The Block Flow option generates a graph where each node represents a basic block. Since a basic block has no internal jumps, this option allows you to quickly visualize the internal control logic of a function or code region. If you want a more detailed view, the Code Flow option takes it a step further

by embedding the actual instructions into each node, giving you the content as well as the structure.

If you want to understand function interactions, the Calls feature allows you to explore which functions call or are called by the currently selected function. This provides a fast method to trace dependencies or identify key points of interest in a binary. For more nuanced call graphs, the Calls Using Model option allows you to specify different graph construction models based on your analysis goals, such as handling indirect calls or treating certain blocks as isolated.

In addition to control flow, you can visualize how data elements relate to one another through references using the Data option. This approach makes it easier to track usage of important variables, structures, or memory locations, which is especially useful in data-heavy binaries or obfuscated malware.

To support flexible workflows, the menu includes options like Append Graph, which allows you to build up composite views of related elements, and Reuse Graph, which reuses the same window for new graphs to keep your workspace clean. The Show Location toggle helps keep your Graph view synchronized with the Listing window contents so you never lose your place.

Table 10-1 sums up the graph capabilities available in Ghidra and indicates what each shows, as well as when you might choose to use a particular graph option.

Table 10-1: Ghidra Graph Types

Graph type	What does it show?	When is it helpful?
Function	Control flow inside a single function, with zoomable basic block layout	Visually stepping through complex functions, especially obfuscated logic
Function Call	All function calls across the program	Seeing the big picture of function-level interactions in the binary
Block Flow	Basic blocks and their control flow	High-level understanding of logic within a code region
Code Flow	Basic blocks with embedded instructions	Detailed instruction-level flow tracing
Calls	Functions directly called by or calling the current function	Quick local dependency analysis
Calls Using Model	Customized call relationships using analysis models	Dealing with shared stubs, indirect calls, or inlined code

Let's take a moment to look at some simple examples that demonstrate a visual approach to analyzing a binary using options available in the graph menu.

CRACKME, CRACK YOURSELF

A *crackme* is a puzzle built by reverse engineers, for reverse engineers. The name derives from cracking a piece of software to bypass copy or usage restrictions—one of the more nefarious uses of reverse engineering skills. Crackmes provide a legal means to practice these skills and provide both the author of the crackme and the person analyzing the crackme a chance to show off their talent.

A common style of crackme receives a user input, transforms that input in some way, and then compares the result of the transformation to a precomputed output. When you attempt to solve a crackme, you are generally given only a compiled executable that contains both the code that performs the transformation and the final output for an unknown input. The crackme is solved when you derive the input that was used to generate the output contained in the binary, which typically requires understanding the transformation so well that you can derive the inverse transformation function.

The Graph Menu in Practice

Assume that we are about to analyze and solve a crackme file called *ch10-1*. Figure 10-14 shows a sample interaction with the file when we run it.

```
Enter the correct number: 6116
Access denied
Press any key to continue . . .
```

Figure 10-14: The first results of running the crackme, *ch10-1*

We know at least three things after this interaction:

- The crackme displays the prompt `Enter the correct number:`
- Our guess that the number is 6116 is incorrect
- The crackme displays the message `Access denied`

Can we discover the correct password? Let's load this binary into Ghidra and take a visual approach to analyzing the binary. We will ignore the traditional CodeBrowser windows and start by using the Block Flow option, as shown in Figure 10-15.

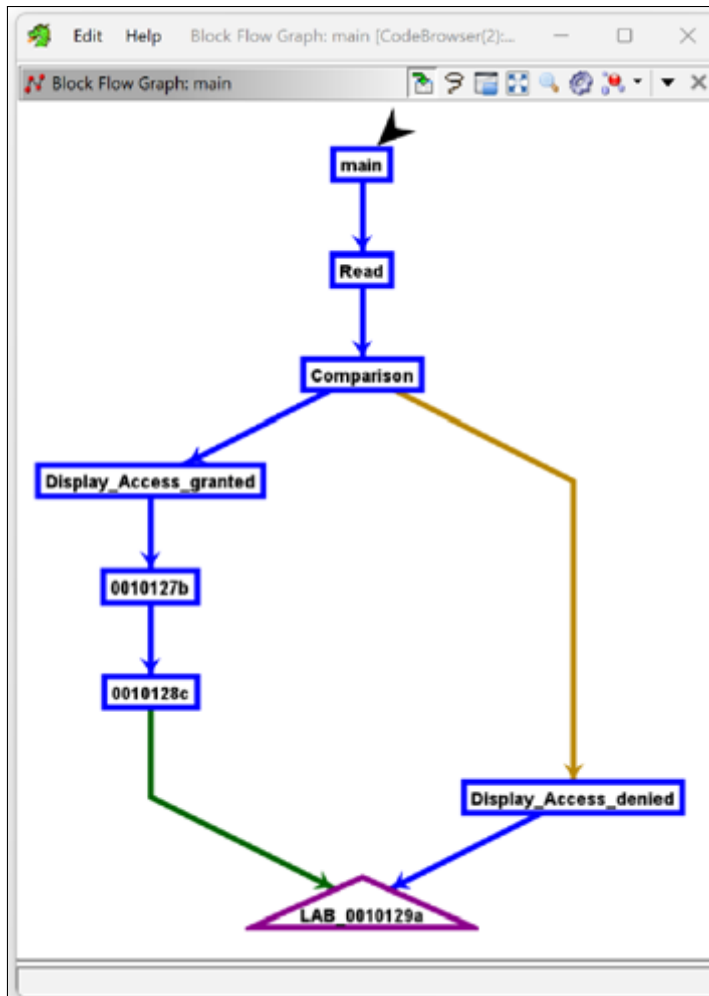


Figure 10-15: The Block Flow Graph view of ch10-1

By following the block flow visually, we can see that the program makes a comparison that determines its subsequent execution path. In Figure 10-15, the path on the left indicates a successful password choice and the path on the right indicates an unsuccessful attempt. What remains is to determine the comparison that determined which path we were destined to follow.

When we revisit the graph using the Code Flow view, we can zoom in on the node labeled Comparison and see the associated code, as shown in Figure 10-16.

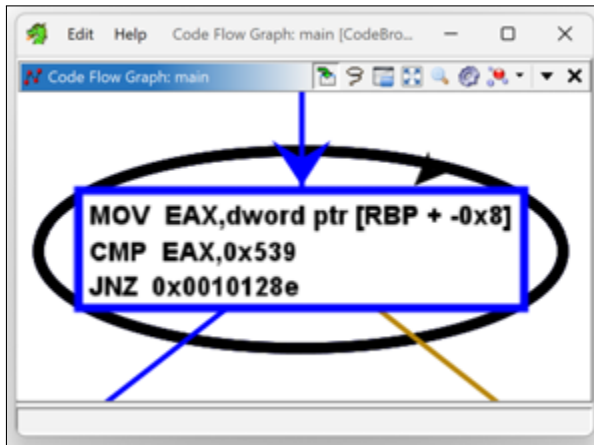


Figure 10-16: The Code Flow view of ch10-1

Let's convert the hex value (0x539) shown in the graph into a decimal (1337) and run the crackme again to check the result. As shown in Figure 10-17, we have found the solution to the crackme.

```
Enter the correct number: 1337
Access granted
Press any key to continue . . .
```

Figure 10-17: The second results of running ch10-1

We have just solved a crackme without ever looking at the Listing Window or the Decompiler window. This approach might not work for all reverse engineering challenges, but it demonstrates a few of the additional capabilities that the CodeBrowser Graph options can bring to your workflow.

While we have demonstrated the usage of some of the more commonly used graph visualization options, we haven't explored the myriad of graph types that Ghidra supports, shown in Figure 10-18.

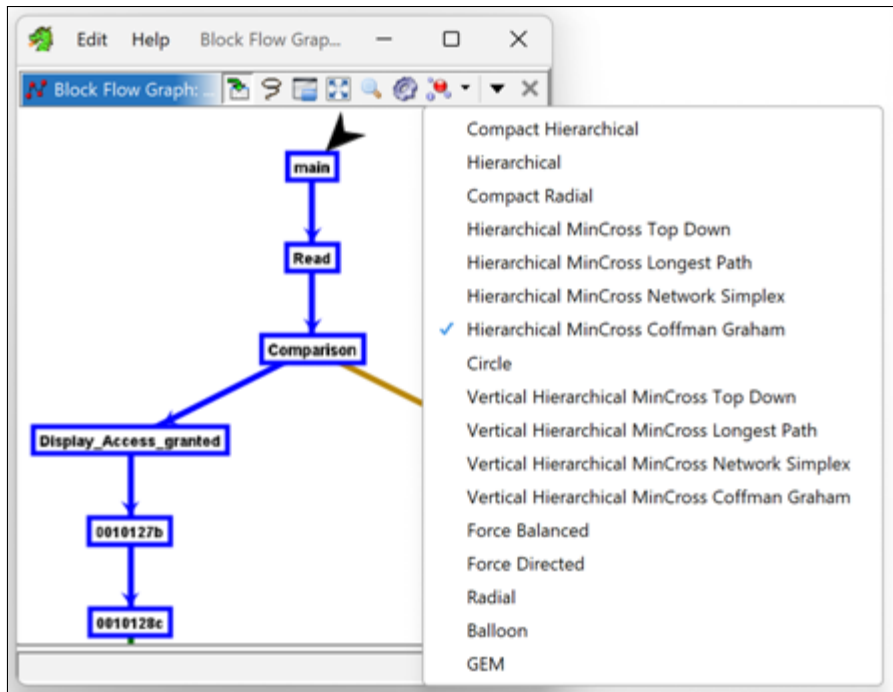


Figure 10-18: The Graph type options

These graph types allow you to tune the graph visualization to one appropriate to the problem you are solving.

Summary

Together, the graphing tools explored in this chapter turn Ghidra from a line-by-line disassembler into a visual, exploratory environment where you can uncover structure, behavior, and relationships in software more intuitively. Whether you're analyzing a complex code region or presenting findings to a colleague, the Graph menu equips you with the means to see the code, not just read it.

Now that you have seen the basic functionality available when running Ghidra as a stand-alone instance with you as the only reverse engineer, it is time to investigate options for using Ghidra as a collaborative tool. In the next chapter, we look at Ghidra Server and the environment it provides to support collaborative reverse engineering.

PART III

CUSTOMIZING AND EXTENDING GHIDRA

11

USING GHIDRA COLLABORATIVELY



At this point, you should be comfortable navigating the Ghidra project environment and the many available tools and windows.

You know how to create a project, import files, navigate, and manipulate the disassembly. You understand Ghidra data types, data structures, and cross-references. But do you understand scale? A 200MB binary is likely to generate a disassembly that is millions of lines long and consists of hundreds of thousands of functions. Even with the largest monitor you can find, you will be able to view only a few hundred lines of that disassembly at any one time.

One way to take on such a monumental task is to assign a team of people to it, but that introduces an additional problem: How will you synchronize everyone's efforts so that people aren't walking all over one another with their changes? It is time to extend our discussion of using Ghidra to cover a collaborative team working together on a shared project.

Ghidra's support for collaborative reverse engineering makes it unique among software analysis tools. SRE is a complex process, so the ability of

analysts with different skill sets to simultaneously analyze a single binary can drastically reduce the amount of time needed to obtain the desired results. For example, a rock star in navigating control flows may dread having to analyze and document the associated data structures, while an expert in malware analysis may be ill-suited for vulnerability discovery work.

In this chapter, we introduce Ghidra's collaboration server, which is included with the standard Ghidra distribution. We discuss its configuration, installation options, and use to help you get more eyes focused on your most challenging reverse engineering problems.

Starting Ghidra Server

Ghidra facilitates collaboration using a shared Ghidra Server instance. If you are the system administrator responsible for setting up the Ghidra Server, you have many choices to make, like whether to deploy it on a bare-metal server or in a virtual environment that makes it easier to migrate and repeatably install. You can configure Ghidra Server on all platforms that support Ghidra.

The deployment options used in this chapter are intended to demonstrate Ghidra's collaborative features and are suitable for development and experimentation. We will demonstrate two options to get you started, including console mode and a Docker container. If you are configuring a Ghidra Server for production use, you should carefully read the Ghidra Server documentation and determine an appropriate configuration for your environment and specific use case. (An entire book could be written to describe Ghidra Server setup and its installation options and associated approaches, but this is not that book.)

Configuration

We are going to show you how to set up a Ghidra Server on your local machine to demonstrate the associated functionality. For deployments in production environments, you should carefully read the documentation and choose configuration options that are appropriate for your environment.

To prompt users for credentials when accessing Ghidra Server, you must make a few minor modifications to the startup parameters in the Ghidra Server configuration file, *server/server.conf*. (You might want to back up the file before making changes.) Using the plaintext editor of your choice, open *server.conf* and locate the Ghidra Server startup parameters section. If you scroll past all of the comments documenting your options, you will find that there are already two parameters set for you:

```
wrapper.app.parameter.1=-a0
wrapper.app.parameter.2=${ghidra.repositories.dir}
```

You need to insert a couple of additional parameters between these. If you read the file very quickly as you scrolled to this location, you would know that *ghidra.repositories.dir* needs to be the last item in this list, so shuffle the associated numbers when adding the new parameters:

```
wrapper.app.parameter.1=-a0  
wrapper.app.parameter.2=-u  
wrapper.app.parameter.3=-ip 127.0.0.1  
wrapper.app.parameter.4=${ghidra.repositories.dir}
```

GHIDRA SERVER SPECIALS

Your Ghidra Server is happy to offer you the following installation options:

Platforms Bare metal, virtual machine, containers, and more!

Operating systems Multiple flavors of Windows, Linux, and macOS. Something to suit every taste.

Authentication methods Choose how friends and colleagues are able to access your offerings—from “open to the general public” to “PKI only” and everything in between.

Preparation Run from a container, a script, *.bat* files, or detailed instructions, or make your own recipe by banging away at the console until something exciting happens.

If you don't see an option you like, don't worry; this is just a subset of what is available. Ghidra Server options can meet the needs of even the most discriminating guests to provide them with the Ghidra deployment of their dreams. Thank you for visiting your Ghidra Server. For more information, see the extended server menu, *server/svrREADME.html*, available in a Ghidra directory near you.

Console Mode

With the new parameters added to the configuration file, perhaps the quickest way to begin using Ghidra Server is to start it in console mode. This requires the least effort and is a handy way to try out Ghidra Server. Open a command line in the *server* directory and start the server using the following:

```
.\ghidraSvr.bat console
```

From there, you can begin to administer and use the server. Before we get to the specifics of Ghidra Server administration and use, let's look at another option for starting a Ghidra Server.

Docker Container

Setting up a Ghidra Server manually on a host system can require installing dependencies, configuring paths, and managing different versions of Ghidra as they are released. Docker provides a convenient way to simplify this process by packaging everything the server needs into a lightweight, isolated container. This approach makes it easier to deploy a consistent server environment, share it across team members, and quickly restart or upgrade the server without complex reinstallation steps. The following instructions

demonstrate how to build and run a Ghidra Server using Docker, allowing you to set up a functional server environment with minimal manual configuration.

1. To begin, change into the Ghidra installation directory so you can access the Docker build scripts that come with the distribution:

```
cd path-to-ghidra-installation
```

2. If you have not already built the Docker image, you will need to do so now. This step prepares a reusable Ghidra Server image that you can run whenever needed:

```
./docker/build-docker-image.sh
```

3. Next, start the Ghidra Server container. This command sets the mode to run the server, mounts the repository and configuration files from your local machine, and opens the required network ports so that clients can connect. In this example, the container uses version *11.4_PUBLIC*:

```
docker run \  
  --env MODE=ghidra-server \  
  --rm \  
  -it \  
  --volume /mnt/d/ghidra_server_repos:/ghidra/repositories \  
  --volume /mnt/d/ghidra_11.4_PUBLIC/server/server.conf:\  
  /ghidra/server/server.conf \  
  -p 13100:13100 \  
  -p 13101:13101 \  
  -p 13102:13102 \  
  ghidra/ghidra:11.4_PUBLIC
```

4. Once the command is running, you should verify that the container has started successfully by listing all active Docker containers:

```
docker ps
```

5. To perform administrative tasks, you will need to start a shell session inside the running container. Use the container ID shown in the output of the previous command:

```
docker exec -it CONTAINER-ID bash
```

6. Inside the container, navigate to the server directory where the administrative scripts are located to use the server administrator tool:

```
cd server
```

By using Docker, we have successfully built and launched a Ghidra Server in a clean, isolated environment, avoiding many of the manual setup steps typically required. This approach makes it simple to start, stop, or update the server as needed while minimizing the host system configuration requirements. While Docker provides flexibility and portability, some environments will benefit from a more permanent setup that automatically runs the server on startup.

Another option for installing Ghidra Server would be as an installed service, allowing it to operate continuously in the background without requiring manual container management. The process for this type of installation varies greatly depending on your system and the functionality you need. If you need this type of production installation, we encourage you to explore the options available in *server/svrREADME.html*.

THE PROJECT REPOSITORY

One advantage of collaborating as a team is that multiple people can work on the same binary at the same time. One disadvantage of collaborating as a team is that multiple people can work on the same binary at the same time.

Whenever multiple users are interacting with the same content, there is the potential to introduce race conditions. In a *race condition*, the order in which operations, such as saving an updated file, are performed can affect the final outcome. Ghidra has a project repository and versioning system to control which changes are committed, when, and by whom.

The project repository checks files in and out, tracks version history, and lets you see what file is currently checked out. When you check out a file, you get a copy of the file. When you have finished working with the file and check the file back in, a new version of the file is created and becomes part of the file's legacy, and if someone else has also checked in a new version of the file, the repository helps resolve any conflicts. We demonstrate interactions with the repository later in this chapter.

Server Administration

Regardless of how you chose to start your Ghidra Server, you'll use the same server administration tool to control aspects of your instance from the command line. Let's use our administrative powers to add four users to our instance:

```
.\svrAdmin.bat -add user1
.\svrAdmin.bat -add user2
.\svrAdmin.bat -add user3
.\svrAdmin.bat -add user4
```

By default, each user must log in to a Ghidra client within 24 hours of their account being created using the default password *changeme*, and then change this password during that initial login. Otherwise, the account will be locked and must be reset. Ghidra provides the Ghidra Server System Administrator with several authentication options, ranging from simple passwords to public key infrastructure (PKI). In this section, we chose to use a local Ghidra password, the default option.

Now that we have a Ghidra Server instance running and three example user accounts ready to start working, we will move away from the command line and continue work within the Ghidra GUI environment.

Creating a Shared Project

Let's walk through the process of creating a shared project. You can make a shared project accessible to any users who are authorized to connect to your Ghidra Server, facilitating collaborative, concurrent access to the project.

When you create a new project (File ► New Project) and select Shared Project, you must specify the server information associated with your Ghidra Server, as shown on the left in Figure 11-1. The default port number (13100) is provided, but you must supply the server's hostname or IP address, and you may need to authenticate yourself, depending on the server's configuration.

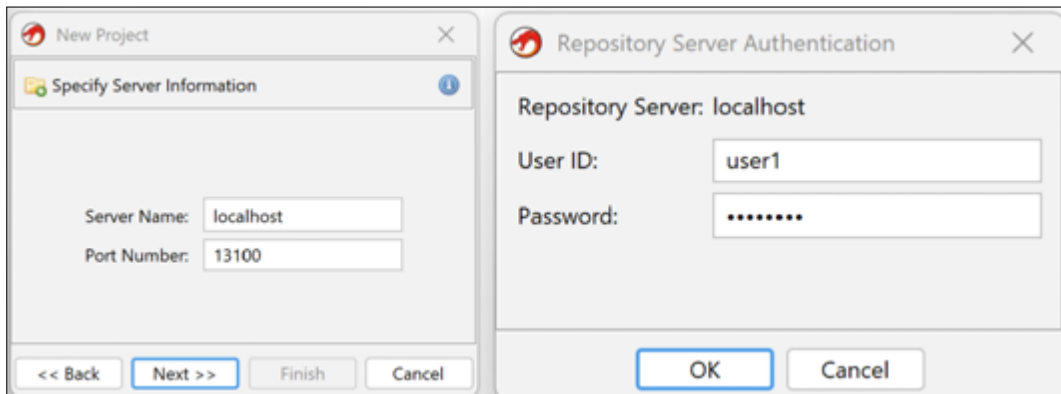


Figure 11-1: Logging in to a Ghidra Server repository

On the right side of the figure, we log in as one of the users we created with the installation script (*user1*). If this is the first time logging in as this user, you will need to change the password from *changeme*, as discussed earlier.

Next, select an existing repository or create a new one by entering a new repository name, as shown in Figure 11-2. For this example, we will create a new repository named *CH11*.

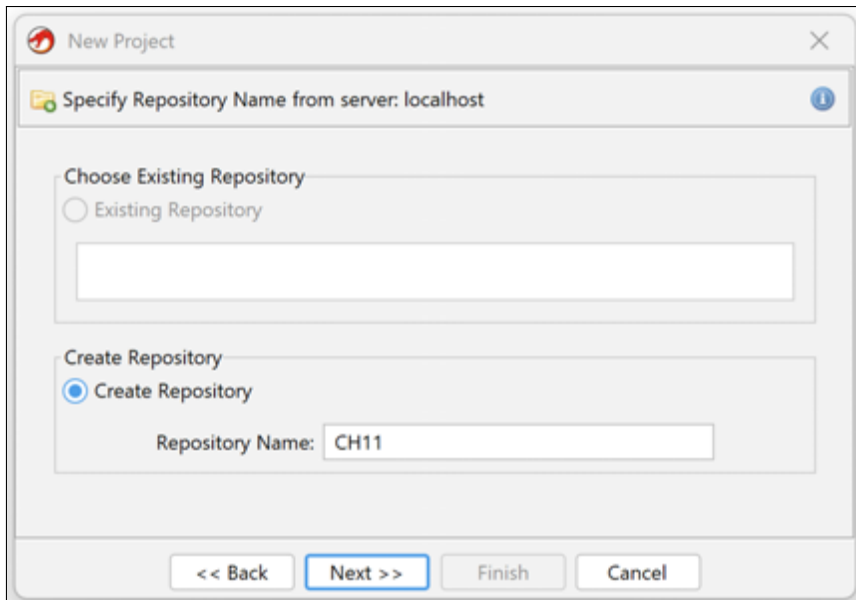


Figure 11-2: The New Project dialog

Clicking Next creates a new repository and a new project and takes you to the now-familiar Project window (Figure 11-3).

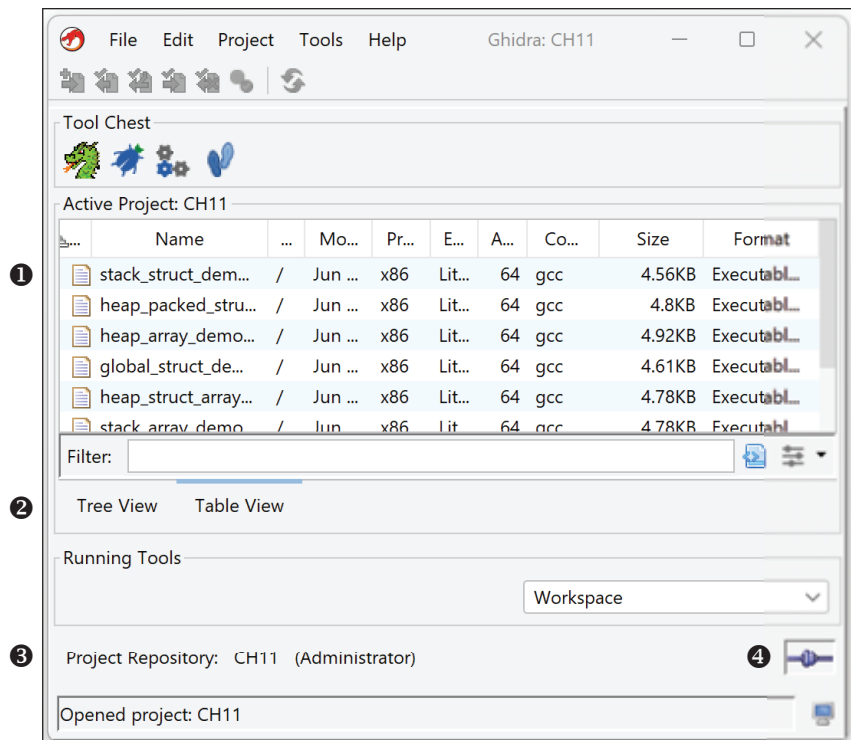


Figure 11-3: The Project window for a shared project, showing the Table View

We have imported some files ❶ and are displaying them using a table instead of the default tree structure for project files. The *Table View*, which is one of the tabbed layout choices ❷, provides much more information about each of the project's files. The Project window shows the name of the project repository (*CH11*), your role on the project (Administrator), and an icon to the right to provide information about your connection to the server ❸. In this case, hovering over the icon ❹ displays the message "Connected as user1 to localhost." If you were not connected, the icon would be a broken link rather than the connected link shown in the image.

WHO'S THE BOSS AROUND HERE?

The server administrator is responsible for creating Ghidra Server accounts and configuring an authentication protocol for connections to the server. Server administration is an inherently command line-oriented activity, and there is no requirement for the server administrator to be a Ghidra user themselves. On the client side, any authorized user may create repositories on the Ghidra Server, automatically becoming the repository administrator of each repository they create. This gives them complete control of the repository, including who can access it and the type of access each user can have. After creation, repository administrators may grant access to additional authorized users via Ghidra's Project window.

Once a project has been created and has an administrator, authorized users can log in to the server and work with the project. A successful login takes you to the Ghidra Project window, where you will have access to your authorized projects.

I DON'T WANT TO SHARE

Using a Ghidra Server installation for nonshared projects has advantages as well. Your initial introduction to Ghidra focused on installing Ghidra on a single computer and using that computer to access your projects and files (which were all stored on that computer). This means all analysis work depends on that computer. Ghidra Server facilitates multipoint access to your files from a wide variety of devices. You can require authentication before your files are accessed, and you can convert your projects from nonshared to shared if desired. One limitation is that you need to be connected to the Ghidra Server to check out or check in files.

Project Window Menus

When you connect to a Ghidra Server, the options available in the Project window become more meaningful, as some of the previously unavailable ones now have a new context. Here, as well as in Chapter 12, we discuss the

individual menu components and how you can use them to improve your analysis process.

File

Figure 11-4 shows the File menu. The first five options are pretty standard file type operations, and their behavior is what you would expect from menu-driven applications.

Ghidra: CH11	
File	Edit Project Tools Help
New Project...	Creates a new shared or nonshared project
Open Project...	Opens an existing shared or nonshared project
Reopen	Provides a list of recently opened projects to reopen
Close Project	Closes the current project
Save Project	Saves the current project
❶ Delete Project...	Deletes the current project (asks for confirmation)
❷ Archive Current Project...	Archives the current project
Restore Project...	Restores an archived project
Configure...	Displays Ghidra configuration options (see Chapter 12)
Install Extensions...	Discussed in Chapter 15
Import File...	Adds a file to the project (see Chapter 2)
❸ Batch Import...	Adds multiple files to the project
❹ Open File System...	Opens a tree view of a filesystem in a new window
Exit Ghidra	Exits the Ghidra application

Figure 11-4: The File menu

We will discuss the notable options, marked with numerals, in detail.

Deleting Projects

Deleting a project ❶ is a permanent action in Ghidra that cannot be undone. Fortunately, it takes effort and requires confirmation. First, you cannot delete your active project. This minimizes the danger of an accidental deletion. To delete a project, you must complete the following three steps:

1. Choose File ► Delete Project from the menu.
2. Browse to (or enter the name of) the project to be deleted.
3. Confirm that you want to delete the project in the resulting confirmation window.

Deleting a project deletes all of its associated files. For that reason, it may be wise to first archive the project via the Archive Current Project option ❷.

Archiving Projects

Archiving a project allows you to save a snapshot of the project, its associated files, and associated tool configurations. Reasons for archiving a project include the following:

- You are going to delete the project but want to preserve a copy “just in case.”
- You want to package a project for migration to another server.
- You want a version that you can easily transfer between Ghidra versions.
- You want to create a backup of a project.

Follow these steps to archive a project:

1. Close the CodeBrowser window and all associated tools.
2. Choose File ► Archive Current Project from the menu.
3. Choose a location and name for the archive file on your local machine.

If you choose the name of an existing file, you will have the opportunity to change the name of or overwrite the existing file. Archived files can be easily restored through the Restore Project option.

Batch Import

The Batch Import option (Ⓢ in Figure 11-4) allows you to import a collection of files into a project in a single operation. When you choose File ► Batch Import, Ghidra presents a file browser window similar to the one shown in Figure 11-5. This window allows you to navigate to the directory that contains the files you wish to import.

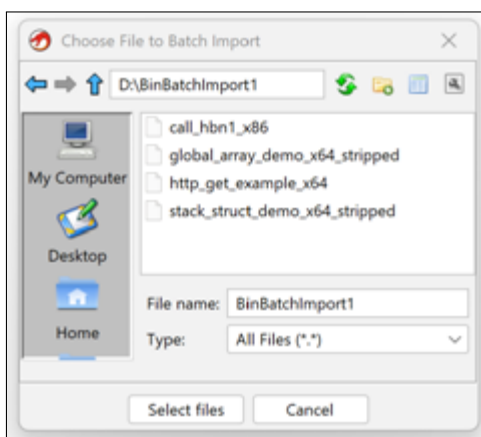


Figure 11-5: The Batch Import selection window with files selected

You can select a file (or files) from a single directory, or an entire directory, to add to the batch import list. After you highlight the files and click the Select Files button, you will be taken to the Batch Import window, which shows you the files you have already selected for import. In Figure 11-6, the files from the directory *BinBatchImport1* were loaded as individual files, and the directory *BinBatchImport2* was added as a directory with five files, as shown to the right of the directory name. You can add and remove files to refine your import list and control several options, including the depth of recursion to search for files in a directory.

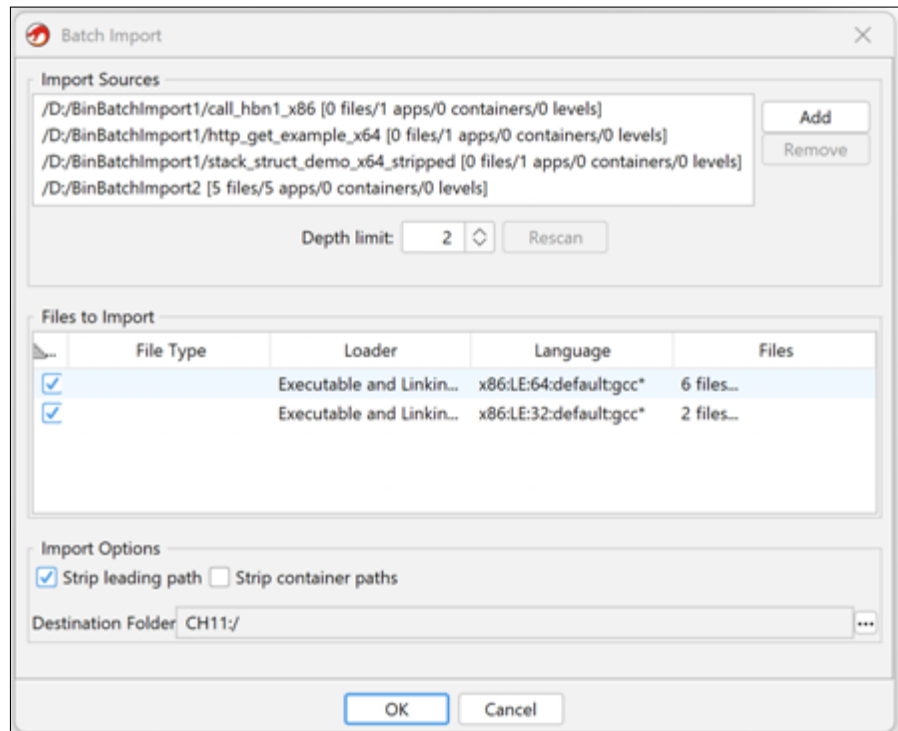


Figure 11-6: The Batch Import dialog

To determine the appropriate depth limit in the Batch Import window, or simply to explore the filesystem, use the Open File System menu option (4 in Figure 11-4).

This option opens the selected file system container (.zip file, .tar file, directory, and so on) in a separate window. (It is best to determine the depth beforehand because you would need a second Ghidra instance open to operate both windows simultaneously. Each window blocks access to the other in a single instance.)

Edit

Figure 11-7 shows the Edit menu. We discuss the Tool Options and Plugin Path options in Chapter 12, but the PKI options are related to the Ghidra Server setup and merit discussion in this chapter.

Ghidra: CH11	
File Edit Project Tools Help	
Tool Options...	Edit options associated with tools (see Chapter 12)
Plugin Path...	Plugin options (see Chapter 12)
Clear PKI Certificate...	Set and clear PKI certificates for authentication
Set PKI Certificate	

Figure 11-7: The Edit menu

At the start of this chapter, we mentioned that, when setting up a Ghidra Server, you can choose from several authentication methods. We set up a simple server that uses a username and password for authentication. PKI certificates are more complex. While PKI implementations can vary, the following example represents a reasonable PKI client authentication process for Ghidra Server:

User1 wants to be authenticated so that she can work on her Ghidra Server project. She has a client certificate that includes her username and a public cryptographic key. She also has a private cryptographic key corresponding to the public key contained in the certificate, which she keeps safely hidden away for important occasions such as this. Her certificate was cryptographically signed by a certificate authority (CA) that is trusted by the Ghidra Server.

User1 presents the server with her certificate, from which the server can extract the public key and username. The server runs checks to confirm the certificate is valid (for example, is not on a Certificate Revocation List, is within a valid date range, has a valid signature from a trusted CA, and possibly others). If all checks pass, the server confirms a valid certificate and binds *User1*'s identity to a public key. Now *User1* needs to prove that she has the corresponding private key so the Ghidra Server can verify it against the extracted public key. As long as the private key is actually held by only *User1*, the Ghidra Server correctly validates her certificate and the server verifies that *User1* actually possesses the private key, so *User1* is considered to be authenticated.

The process for managing PKI certificate authorities is described in the Ghidra Server README file (*server/svrREADME.html*). The Set PKI Certificate and Clear PKI Certificate menu options enable users to associate (or disassociate) themselves with key files (such as *.pfx, *.pks, or *.p12). When

setting a PKI certificate, the user will be provided with a file navigation window to identify the appropriate keystore. The certificate can be cleared at any time with the Clear PKI Certificate option. Should you choose to enable PKI authentication, you may use Java’s keytool utility to manage keys, certificates, and Java keystores.

Project

The Project menu, shown in Figure 11-8, provides facilities for managing project-level activities.

Ghidra: CH11	
File Edit Project Tools Help	
❶	View Project...
	Opens a read-only view of a project or repository
	View Repository...
	Provides a list of recently opened projects and repositories
❷	Close View
	Closes a selected read-only view of a project or repository
	Workspace
	Provides a menu of workspace options (see Chapter 12)
❸	Change Password...
	Allows users on shared projects to change password
❹	Edit Project Access List...
	Displays editable access control table for project administrators and a read-only version for all others
	View Project Info...
	Provides detailed information about the current project

Figure 11-8: The Project menu

These activities include viewing and copying from other projects, changing your password, and managing user access to projects that you administer.

Viewing Projects and Repositories

The first four options ❶ are related to viewing projects and repositories. View Project and View Repository open read-only versions of a (local) project or (remote server) repository in a new window adjacent to the Active Project window. In Figure 11-9, the local project *ExtraFiles* has been opened next to the active project. You can explore the read-only project or drag any file or directory from the Read-Only Project Data window to the Active Project window. In Figure 11-9, the three selected files (with the extension *NEW*) have been copied from the Project Data window to the active project: *CH11*.

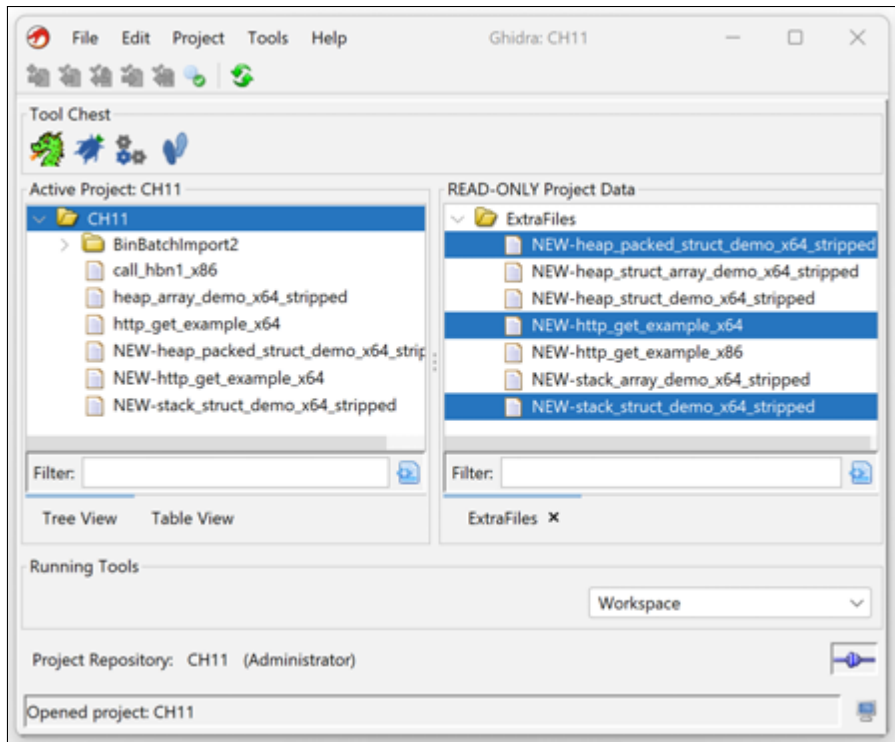


Figure 11-9: Using the Project window to view another project

The next option, View Recent, provides a list of recent projects that can speed up the process of locating a project or repository. Close View closes the read-only view (although in some versions of Ghidra, this option appears to be inactive). A simpler and more reliable alternative is to click the X at the bottom of the project tab that you wish to close, as shown in the bottom right of Figure 11-9.

Changing Passwords and Project Access

The Change Password option (🔑 in Figure 11-8) is available only to users on shared projects, provided that the Ghidra Server is configured with an authentication method that allows a password change. This is a two-step process with an initial confirmation dialog, as shown in Figure 11-10, followed by the same password change option dialog used for the initial mandatory password change.

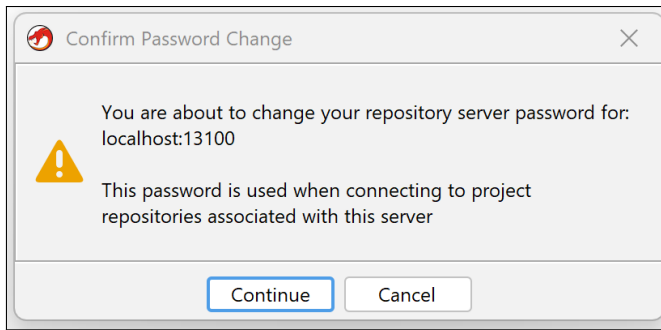


Figure 11-10: The password change initial confirmation dialog

While users can control their own passwords, shared projects also offer the capability to control who can access a project and what permissions are granted to each user. As mentioned earlier in the chapter, the Ghidra Server system administrator has some control over access. Specifically, an administrator can assign an administrator to a repository and create and delete user accounts.

On the client side, if you are an administrator, you can also control access through the Edit Project Access List option (Ⓔ in Figure 11-8) from the Project menu. When it is selected, you will see the dialog shown in Figure 11-11, which allows you to add and remove users from the project and to control their associated permissions.

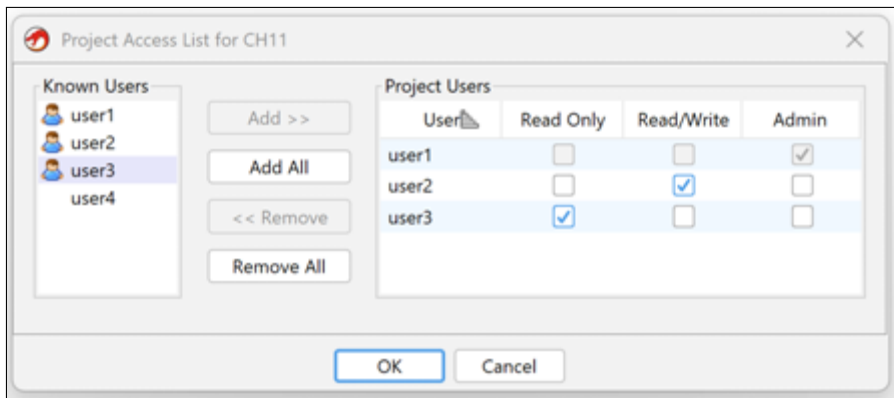


Figure 11-11: The access control window

Each user can be placed in exactly one privilege class, from least privileged (Read Only, on the left) to most privileged (Admin, on the right).

Viewing Project Information

The final menu option is View Project Info (4 in Figure 11-8). The options available in the resulting dialog depend on whether the project is hosted on a Ghidra Server. Figure 11-12 shows examples of nonserver-based (left) and server-based (right) project information dialogs. While the information displayed is pretty straightforward, buttons at the bottom of each window allow you to convert a nonshared project to a shared project (with the Convert to Shared button) or change project information.

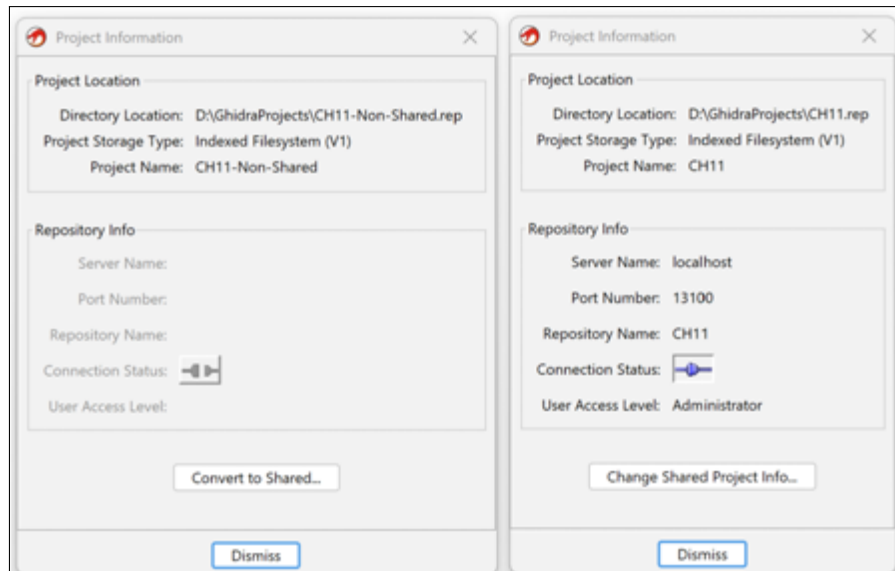


Figure 11-12: Project information windows for nonshared and shared projects

Clicking the Convert to Shared button opens a dialog that asks you to specify the server information and enter the user ID and password for the project administrator. The subsequent steps allow you to specify a repository, add users, set their permissions, and confirm that you want to convert the project. Note that this operation cannot be undone and removes all existing local version history.

Project Repositories

At this point, you may be wondering how projects can be shared while maintaining project integrity. This section covers the process that Ghidra uses to ensure that everyone's work is retained in a shared project that a team can work on concurrently. Before we delve too deeply into the process, let's investigate the file types associated with a shared Ghidra project. We start by discussing the relationship between a project and a repository.

A repository is the key facilitator of the versioning process. When you create a new nonshared project, Ghidra generates a project file (*.gpr* file) and a repository directory, with the *.rep* extension, to facilitate version control. It creates additional files to control locks, versioning, and so on, but you do not need to understand these files to use Ghidra. All files reside on your computer within the directories that you specify at project creation time (refer to Chapter 4).

When you create a shared project, you have the option of creating a new repository or selecting from existing repositories. If you create a new project and a new repository at the same time, a one-to-one relationship exists between the project and its repository, and you become the project administrator. If you choose an existing repository, you are creating a new project for which you are not the project administrator (unless you own the repository). In either case, the *.gpr* file and the *.rep* directory share the same base name. If the repository is named *RepoExample*, the project file will be named *RepoExample.gpr* and the repository folder will be named *RepoExample.rep*. (Despite having an extension, the repository is a directory rather than a file.)

To sum it up: If you create the repository, you are the project administrator and can choose who else can access your repository. If you use an existing repository, you are a user with the rights and privileges that have been assigned to you by the project administrator. So, what happens when multiple users want to make changes to the same project? That is where version control comes into play.

VERSION CONTROL VS. VERSION TRACKING

Ghidra includes two very different types of versioning systems. In this chapter, we discuss version control, a concept that will hopefully become quite clear shortly. Ghidra also has a *version tracking* capability, used to identify differences (and similarities) between two binaries. In the SRE community, this process is generally known as *binary differencing*. The goals of version tracking may include identifying updates in different versions of a binary, functions used by a malware family, signatures, behavioral similarities, and so on. This functionality can be important given that the associated source code may not be available to allow for source-based diffing. We discuss Ghidra version tracking in more detail in Chapter 23.

Version Control

Version control is an important concept in any system where multiple users can make changes or a recorded history of changes is desirable. Version control allows you to manage updates to the system, effectively controlling race conditions. The Project window has a version control toolbar (Figure 11-13). Many of the operations require that the file(s) in question be closed to complete the action.

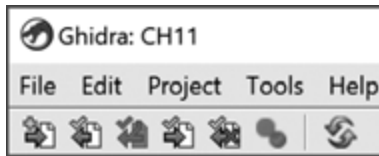


Figure 11-13: The Ghidra Project window's version control toolbar

The icons are enabled for valid version control operations based on the selected file(s). Figure 11-14 shows the basic actions that make up the version control workflow. (We have included a column that provides the rough Git equivalents for all the Git fans.)

Icon	Action	Special option(s)	Similar git commands
	Add file to version control	Keep file checked out	git add git commit
	Check out file	None	git clone (ish)
	Update checked-out file with latest version	Keep file checked out	git pull
		Create .keep File	
	Check in file	Keep file checked out	git commit git push
		Create .keep file	
	Undo checkout	Save copy with .keep extension	git checkout
	Find my checkouts recursively	None	git status

Figure 11-14: Ghidra version control toolbar actions

In addition to using the toolbar icons, you can perform the version control actions via the right-click context menu.

When a collaborative team member decides to check in changes they have made to the project, one of two conditions will be true:

No conflict In this case, no new versions of the file have been checked in since the user checked out the file. Since no potential conflict exists (there are no committed, conflicting changes that the user is not already aware of), the file that is being checked in will become the new version of the file. The old version will be retained in an archival fashion, and the version number will be incremented to ensure that a continuous chain of versions can be tracked.

Potential conflict In this case, another user has committed new changes while the user had the file checked out. The order in which the files are checked in can affect the resulting “current version.” In this case, Ghidra begins a merge process. If no conflicts are introduced by the submissions, Ghidra continues with its automatic merge process. If conflicts are detected, each must be manually resolved by the user.

As an example of a conflict, assume that *user1* and *user2* both have the same file checked out and that *user2* changes the name of `FUN_00123456` to `hash_something` and checks in their change. Meanwhile, *user1* analyzes the same function and renames it to `compute_hash`. When *user1* finally checks their changes in (after *user2*), they will be informed of a naming conflict and will be asked to choose the correct name of the function, between `hash_something` and `compute_hash`, before the check-in operation may be completed. You can find additional information about this process in Ghidra Help.

VERSION CONTROL COMMENTS

When you add or modify a file under version control, you should add a comment explaining what you have done. Each version control action displays a dialog with a comment field and special options. Figure 11-15 shows the Add File to Version Control comment dialog.

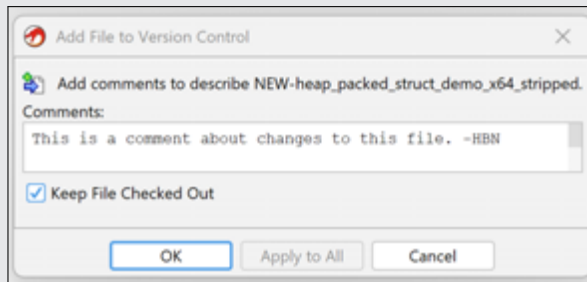


Figure 11-15: Ghidra’s Add File to Version Control comment dialog

The title bar displays the action being performed and is supplemented below by the associated icon, as well as a description of what you should be entering in the Comments text box. If you have selected more than one file, any comments will be associated with only the first file unless you clicked the Apply to All button. Below the Comments text box are specific options associated with the action being performed that the user can select or deselect. See the third column of Figure 11-14 for the special options for each of the actions.

An Example Scenario

Shared projects have many associated intricacies, options, and overloaded terminology. To clarify some of the concepts associated with Ghidra Server and shared projects, let's walk through an example that demonstrates the concepts we have been discussing, starting with the concept of a project.

A *project* is the local entity that lives on the client machine (like a local Git repo). Shared projects are also associated with a repository on a Ghidra Server (like a Git remote), and that repository is where all of the collaborative analysis effort results are stored. Files are shared after they have been imported and added to version control, and are private before that. Therefore, a user can import files into a project, at which point they are private, and then choose to add them to version control, at which point they are shared.

HELP! MY FILE HAS BEEN HIJACKED!

Ghidra has a special term (and associated Project Data Tree icon) for a situation that can frequently occur in a shared project environment. If you have a private file (imported but not yet added to version control) in your project *and* another user adds a file of the same name to the repository, your file will be *hijacked*! This is such a frequent occurrence that Ghidra provides a right-click context menu option to handle the situation. You will need to close the hijacked file and then select the Undo Hijack option from the context menu. This will provide you with the option to accept the file in the repository and keep a copy of your own file, if desired. Other options for resolving a hijack include renaming the file, moving it to another project, and deleting it.

In reality, project permissions are actually repository permissions. If you create a project using an existing repository, you are really saying, "This local project is backed remotely by that repository on the server" (like a Git clone).

Let's walk through a sequence of shared project activities and observe how they affect the shared project environment:

1. *user1* creates a new shared project (and associated new repository) called *CH11-Example*, adds *user2* and *user3*, and assigns them permissions (see Figure 11-16).

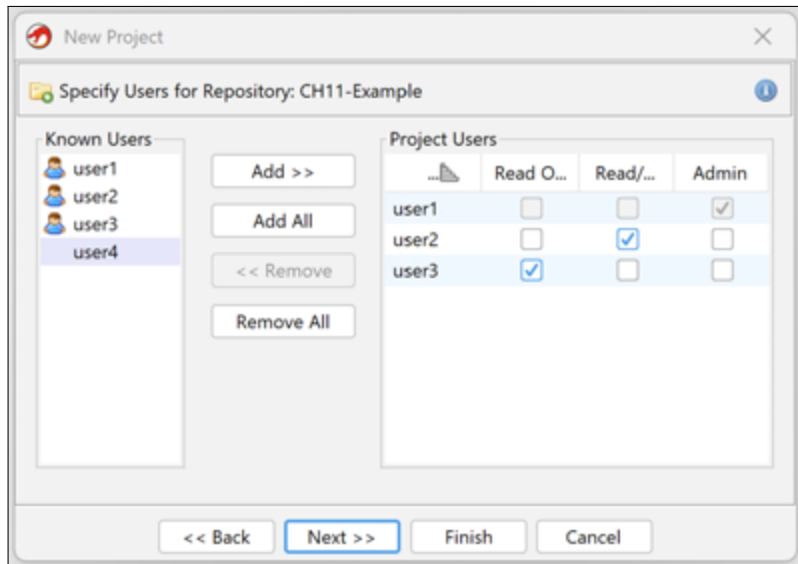


Figure 11-16: Example scenario, step 1

2. *user2* creates a new shared project associated with the existing *CH11-Example* repository (that is, *user2* clones *CH11-Example*). Note that the project is not the same name as *user1*'s project, but the remote repository is the same. In addition, *user2*'s permissions for the repository are shown at the bottom of the window (see Figure 11-17).

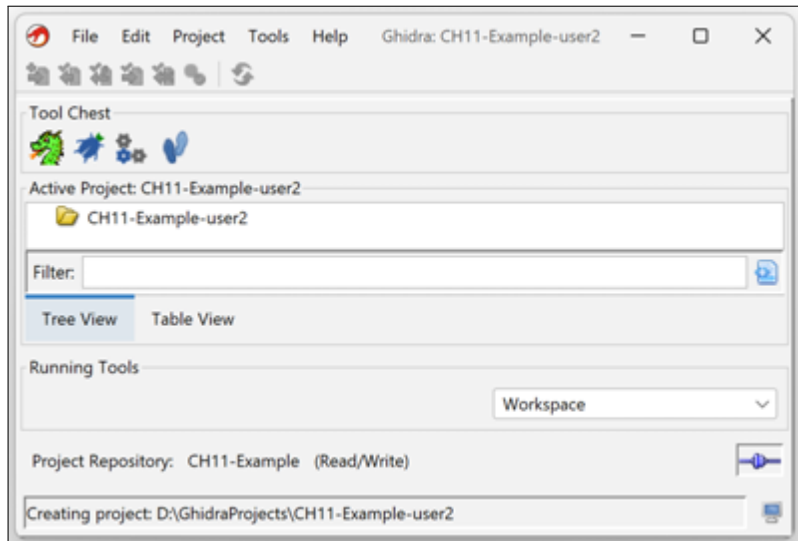


Figure 11-17: Example scenario, step 2

3. *user1* imports a file and adds it to version control, which *user2* can then also see (roughly equivalent to git add/commit/push). This is shown in Figure 11-18.

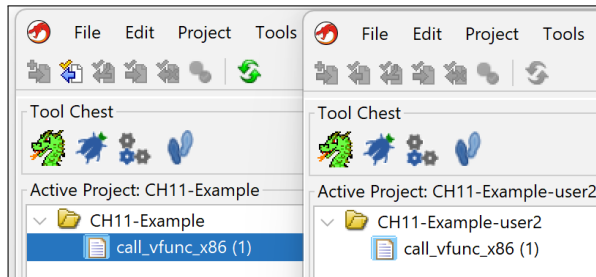


Figure 11-18: Example scenario, step 3

4. *user1* and *user2* then each import the same file to their projects but don't add them to version control. These are private files (see Figure 11-19).

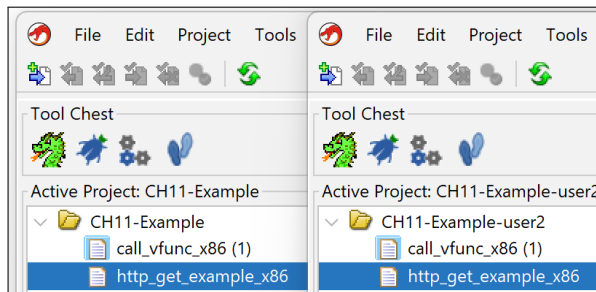


Figure 11-19: Example scenario, step 4

5. *user2* adds this second file to version control (which checks it in). As a result, the file is no longer private, and *user1* now sees this as a hijacked file (see Figure 11-20).

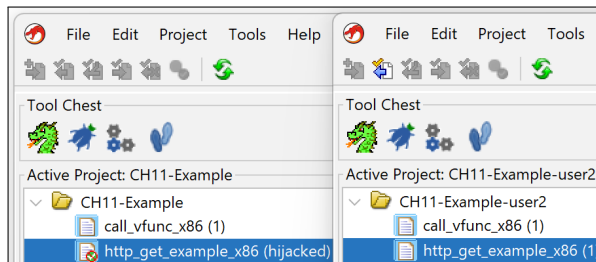


Figure 11-20: Example scenario, step 5

6. *user1* chooses Undo Hijack from the right-click context menu and has the option to replace her file with the version in the repository and keep a copy of her own file if desired. She chooses to accept the repository version and to keep a copy of her own file (which she has moved to another project and which now has the extension *.keep*). Now everything is good again. In this case, *user1* is now seeing the state of the second file as it was when *user2* added it to version control (see Figure 11-21).

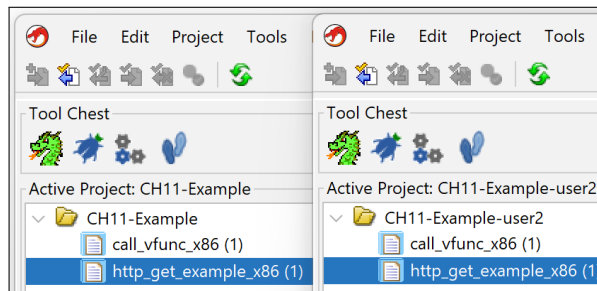


Figure 11-21: Example scenario, step 6

7. *user1* checks out the second file, analyzes it, and then checks it in. Both *user1* and *user2* now see the analyzed version of the file (version 2), as shown in Figure 11-22.

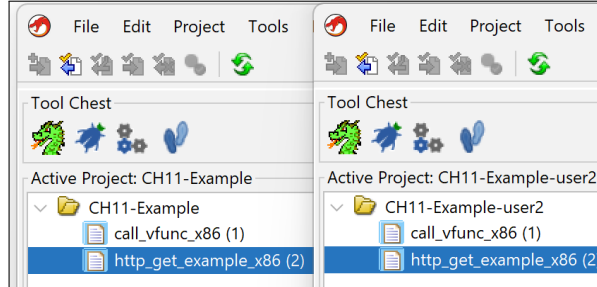


Figure 11-22: Example scenario, step 7

8. *user3* creates a project and associates it with the same repository (see Figure 11-23). *user3* can now see all of the files and can make changes locally (including adding private files), but has no option to commit to the repository because she was not given write permissions. (The project is noted as “Read Only” at the bottom of the window.)

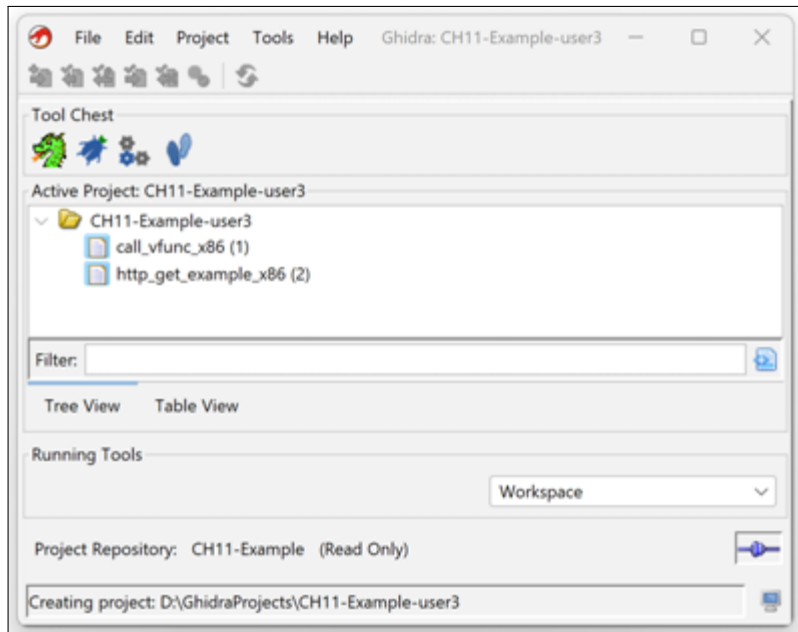


Figure 11-23: Example scenario, step 8

9. *user2* checks in all of her files before leaving work. This is important because she knows that she wants to continue working on her project by using her home computer. Because the project does not exist on her home computer, she needs to log in to the Ghidra Server and create a new project by using the existing repository. This will create the project on her home computer, where she can continue working. (Had she not checked in all of her files before leaving work, she would not have access to her latest work while at home.)
10. The remaining users go home confident that their collaborative Ghidra Server is working as intended.

Summary

Not everyone will require a Ghidra Server or shared projects to facilitate collaborative reverse engineering, but many of the associated capabilities can also be applied to nonshared projects. The remaining chapters focus on nonshared projects, with references to shared projects and Ghidra Server when appropriate. Regardless of the configuration of your Ghidra installation, there is a good chance the default configurations, tools, and views may not be perfect for your workflow. The next chapter focuses on Ghidra configurations, tools, and workspaces and how you can make them work better for you.

12

CUSTOMIZING GHIDRA



After spending time with Ghidra, you may find that you prefer certain settings over others, and you may wish to use them as defaults every time you open a new project or apply them to all files within a particular project. You may also be confused as to why some of the options you have changed carry over from session to session, while other options need resetting every time you load a new project or file. In this chapter, we examine how you can customize Ghidra's default appearance and behavior to better support your reverse engineering workflow, including tailoring it to include more advanced features such as Ghidra's debugger plugins.

To understand the scope of some customizations, you'll find it useful to understand the (admittedly fuzzy) distinction between the terms *plugin* and *tool*.

In a general sense, the following is true:

Plugin A plugin is a software component (for example, the Byte Viewer or the Listing window) that adds functionality to Ghidra. Plugins frequently present themselves as windows, but many plugins do their work behind the scenes (for example, analyzers).

Tool A tool can be a single plugin or a set of plugins that work together. They generally present as a useful graphical user interface (GUI) to help users accomplish tasks. For example, one tool we have worked with extensively, CodeBrowser, is a window that serves as a GUI framework. Function Graph is also a tool.

Don't panic if Ghidra does not strictly adhere to these definitions. In many cases, distinguishing between the two simply does not matter. For example, some menus, such as the Tool Options menu we will discuss later in this chapter, include options that you can apply to both tools and plugins, despite using the term *tool*. You should be able to successfully navigate the customization process even when the usage of the terms varies.

In addition to Ghidra customizations, this chapter will also discuss Ghidra workspaces. *Workspaces* couple a tool with a configuration and provide the capability to design and use a personalized virtual desktop.

Customizing the CodeBrowser

In Chapters 4 and 5, we introduced the CodeBrowser, many of its associated windows, and some of its basic customization options. Now we will walk through a more thorough example of customizations in the CodeBrowser before moving on to the Ghidra Project window and workspaces.

Rearranging Windows

The following six basic operations allow you to control where the individual windows appear in relation to the CodeBrowser window:

Open To open windows, you should generally use the CodeBrowser's Window menu. Each window has defaults that determine where it opens.

Close To close windows, click the X in the upper right of the window. (If you reopen a closed window, it will reappear at the same location rather than at its original default location.)

Move To move windows around, drag and drop them.

Stack To stack and unstack windows, use the drag-and-drop functionality.

Resize To resize windows, hover on a border between two windows to reveal an arrow that allows you to grow and shrink the windows adjacent to the border.

Undock To undock a tool from the CodeBrowser window, drag it outside of the CodeBrowser and drop it on your desktop.

Note that redocking is not as straightforward as you might wish, as shown in Figure 12-1.

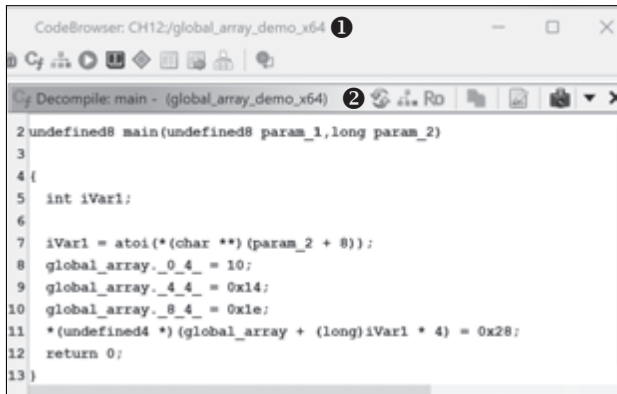


Figure 12-1: Redocking the Decompiler window

To redock a window, you cannot click the title bar ❶ as you will just drag the window around in front of the CodeBrowser. Instead, click the internal title bar ❷ to redock or stack a window. Now that we can rearrange windows, let's customize the windows themselves.

Editing Tool Options

When you choose Edit ► Tool Options, a CodeBrowser option window opens, as shown in Figure 12-2. This window allows you to control the options associated with individual CodeBrowser components. The developers of each component determine which options are available. Because describing every available tool option would take up an entire book, we will look at a few edits that affect tools we have discussed in previous chapters and some that are similar for many tools.

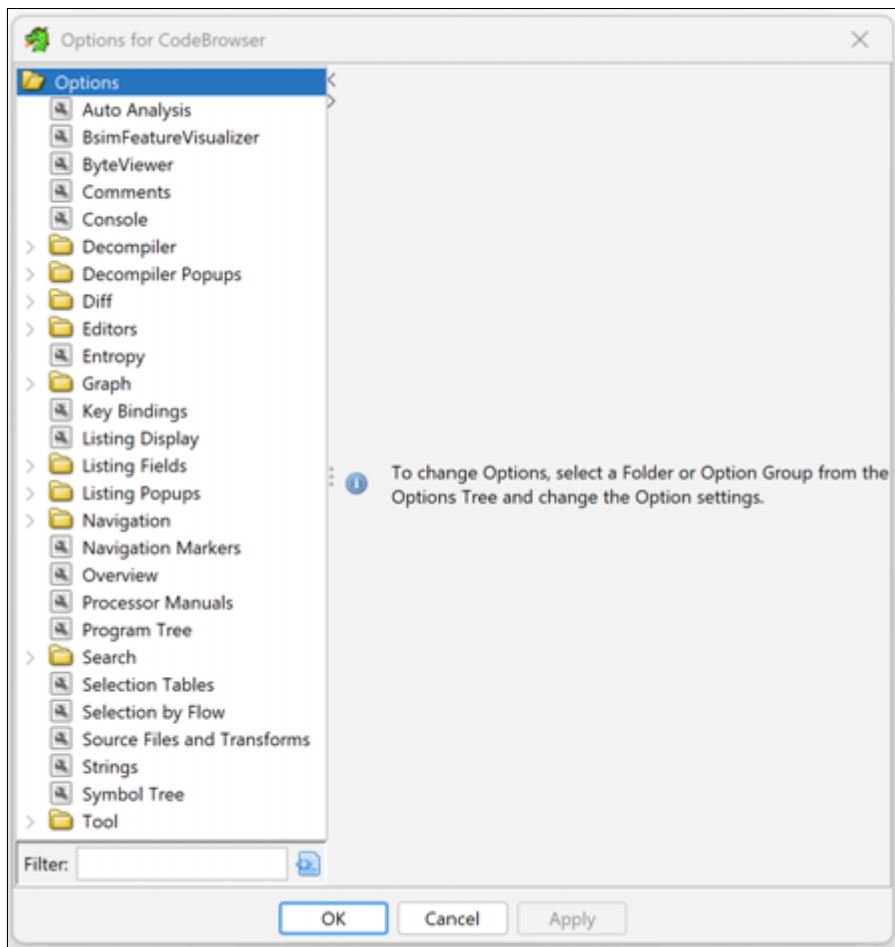


Figure 12-2: The default CodeBrowser Edit ► Tool Options window

While it may not be apparent when rendered in grayscale, many of the tools use color to identify attributes, and the associated color palette is configurable. Clicking a default color within the Options window opens a standard Color Editor dialog, as shown in the Byte Viewer options panels in Figure 12-3. This provides you with the option to control the color of a plethora of items within your CodeBrowser.

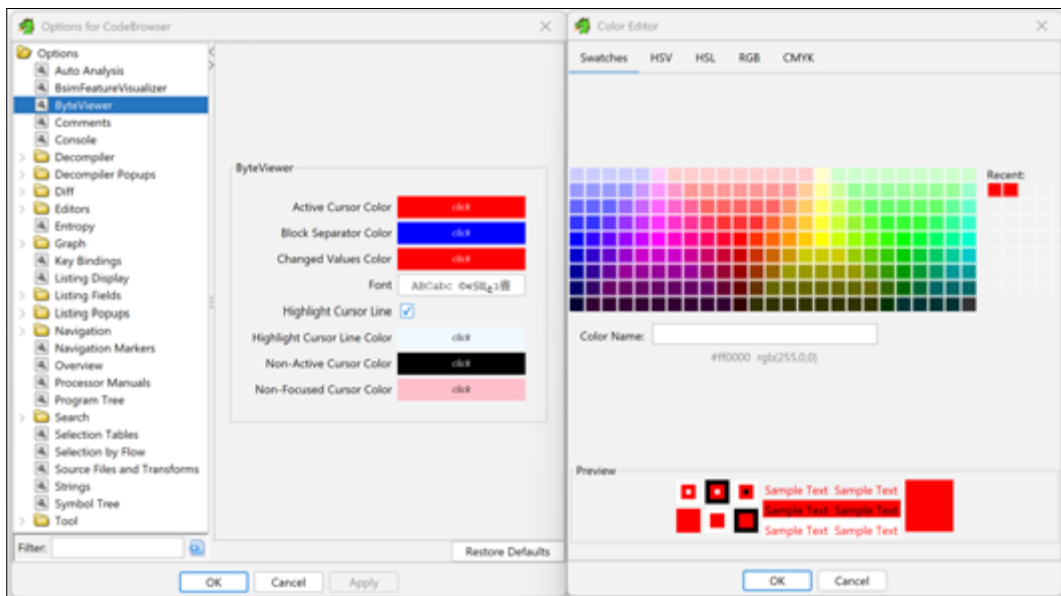


Figure 12-3: The Edit ► Tool Options Color Editor dialog

In Figure 12-3, you can select colors for six items in the Byte Viewer window: Active Cursor, Block Separator, Changed Values, Highlight Cursor Line, Non-Active Cursor, and Non-Focused Cursor. In addition to customizing colors in the Byte Viewer window, you can also select the font and choose to highlight the cursor line. Conveniently, any CodeBrowser tool's option panel includes a Restore Defaults option in the lower right. This enables you to use special color schemes during some analysis steps and then revert to the default color scheme for the tool when done.

Beyond cosmetic changes, many tools provide the ability to set parameters when editing options, such as the ability to control which analyzers to include in auto analysis. We have hinted at this potential as we introduced new functionality in previous chapters. In general, anytime something has a default, there is a way to change it to something else.

You can also access and modify the settings for some overarching tools through the Options window. For example, key bindings specify mappings between Ghidra actions and hotkey sequences, and there are over 550 actions in the default CodeBrowser window for which you can create or reassign a hotkey binding using the Options window. Hotkey reassignment is useful in many instances, including to make additional commands available via hotkeys, change default sequences to sequences that are easier to remember, and change sequences that might conflict with others in use by the operating system or your terminal application. You might even remap all hotkeys to match those of other SRE tools that you may be familiar with.

There are three fields associated with each key binding, as shown in Figure 12-4. The first is Action Name. In some cases, the action name corresponds to a menu command (for example, Analysis ► Auto Analyze). In other cases, it is a parameter associated with a menu command (for example, Aggressive Instruction Finder within Analysis Options).

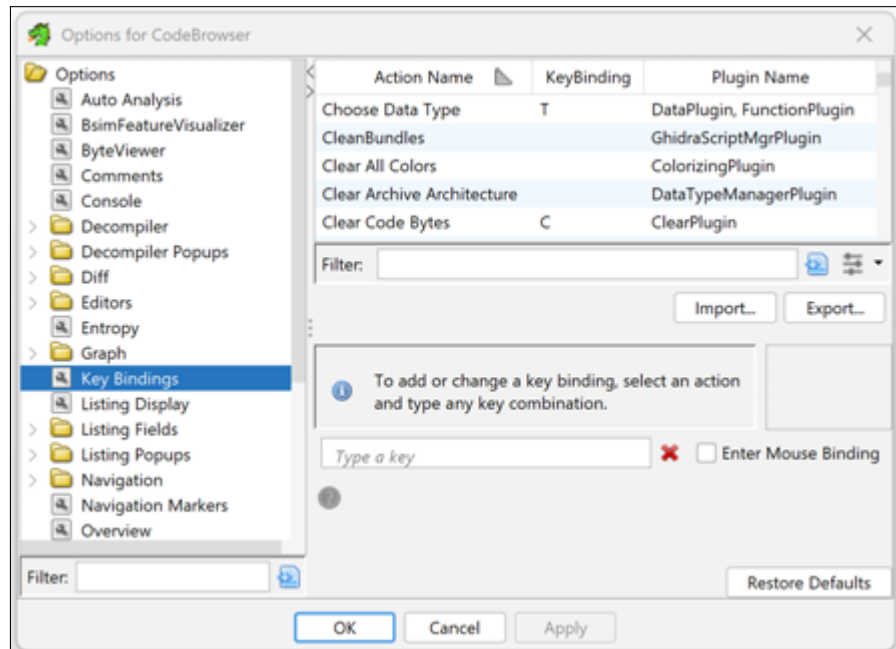


Figure 12-4: The Edit ► Tool Options Key Bindings option

The second column is the hotkey associated with the action, and the final column holds the name of the plugin in which the action is implemented. Not all actions have associated hotkeys, but you can easily assign hotkeys by selecting an action and entering the desired hotkey in the text box. If the hotkey is already associated with another action, you'll see a list of its other uses. When you use a hotkey that has multiple key bindings, Ghidra will provide you with a list of potential actions, and you will need to choose the appropriate one.

NOTE

The Plugin Name column in Figure 12-4 provides another example of Ghidra's imprecise use of the terms tool and plugin. Entries in the Plugin Name column are frequently plugins, but may also be tools, such as the Configure tool.

Editing the Tool

At the bottom of the Edit ► Tool Options window is an entry called Tool. The meaning of Tool changes depending on how you reached this Options dialog but generally refers to either the CodeBrowser or the Project window. Figure 12-5 shows the default configuration options for the CodeBrowser tool.

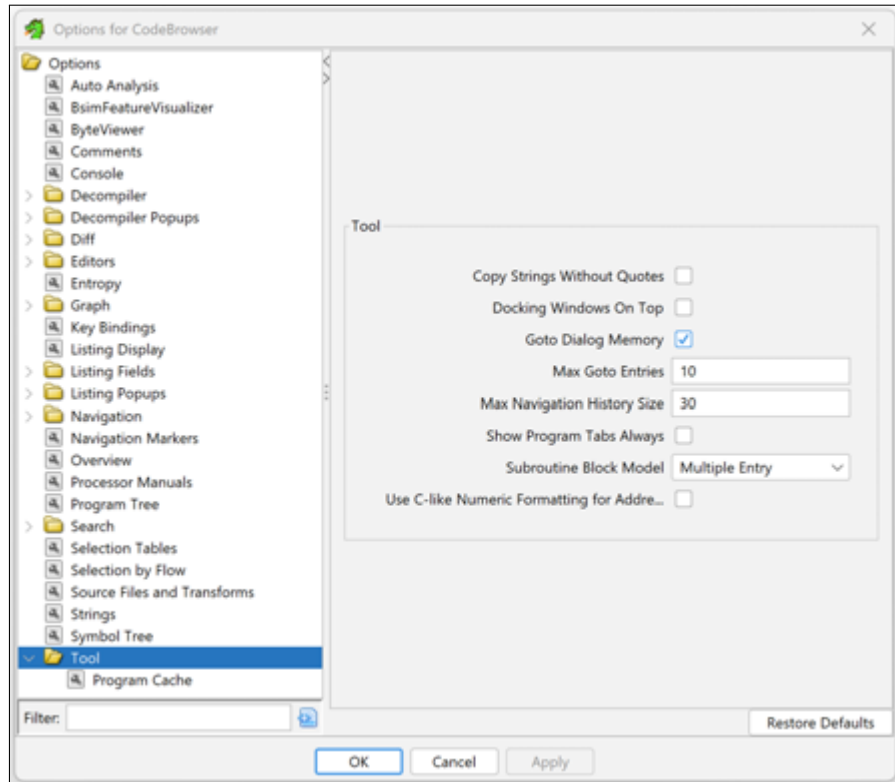


Figure 12-5: Using Edit ► Tool Options ► Tool to edit CodeBrowser options

The Options dialog’s title bar provides the most prominent clue that we are looking at the options page for the CodeBrowser.

Accessing Special Tool Editing Features

Some tools integrate editing features within their individual windows so that you can immediately see the effect of the options on the associated contents. The most extensive set of built-in editing features is available in the Listing window, which you can configure using the Browser Field Formatter introduced in “Changing Code Display Options” on page 135. Figure 12-6 shows a Listing window with the default Browser Field Formatter open.

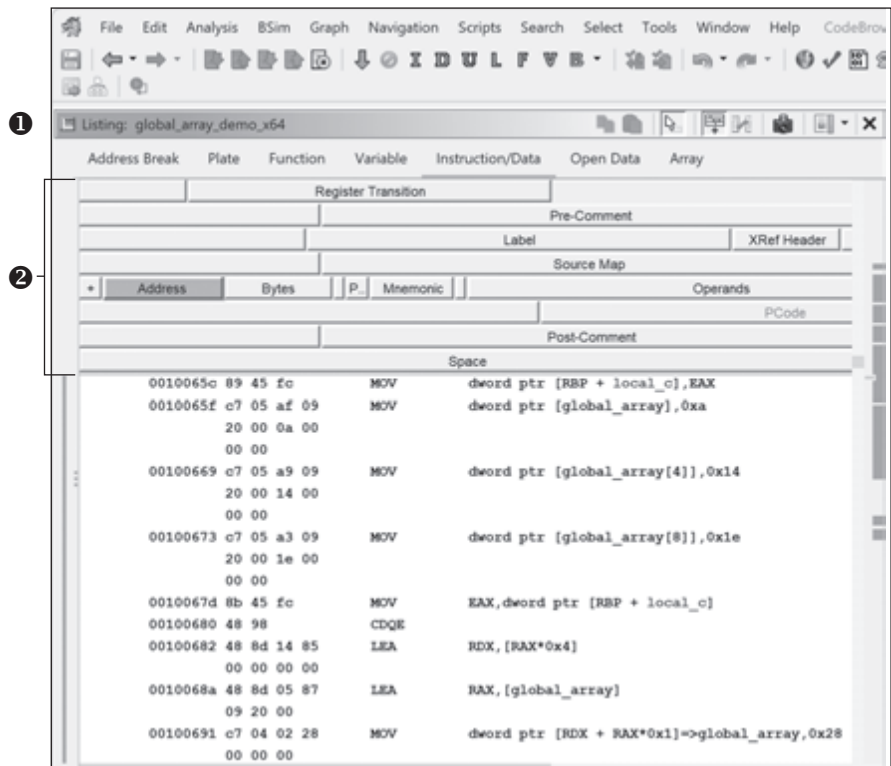


Figure 12-6: The Listing window with the Browser Field Formatter displayed

A row of tabs ❶ representing the various field types present in the disassembly appears at the top of the formatter. In this case, we are looking at instructions, so the Instruction/Data tab is selected. The remainder of the formatter ❷ displays bars for each individual field associated with an address in an Instruction/Data section. In this case, the cursor is on an address within the Listing window, so the Address field is highlighted.

You can use the Browser Field Formatter to change the appearance of the listing. Its capabilities are extensive, and each field has its own associated options. We will investigate only some of the simpler capabilities, many of which are similar to editing the appearance of windows in the CodeBrowser. You can rearrange fields by dragging them to new locations; increase or decrease the width of a field; and add, remove, enable, or disable individual fields. Figure 12-7 shows the same listing contents after removing the Bytes field.

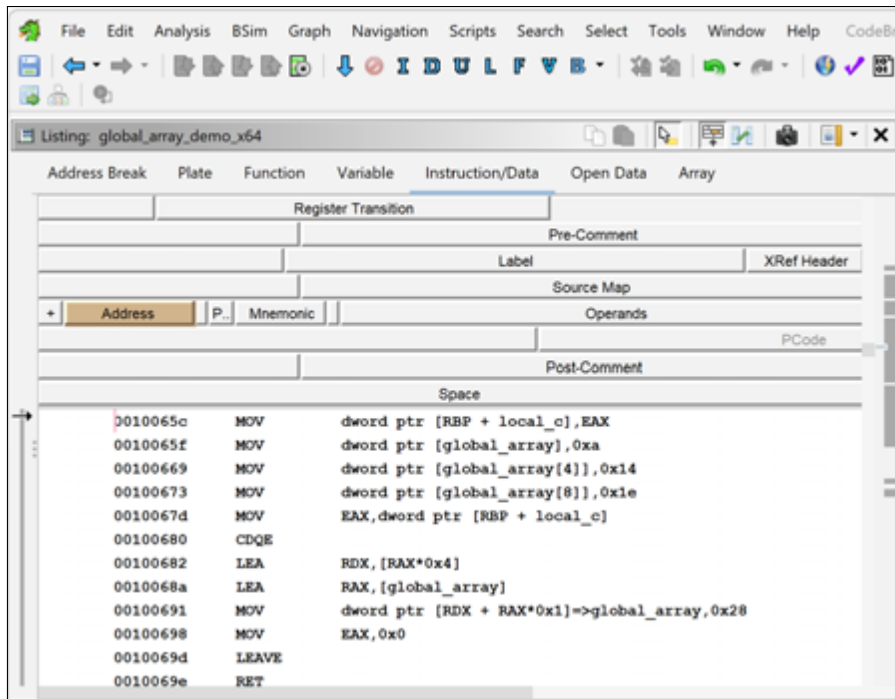


Figure 12-7: The Listing window with customized Browser Field Formatter selections

We removed the Bytes field in many of the listing images in previous chapters to condense the listing and show more content in the available space.

Saving the CodeBrowser Layout

When closing the CodeBrowser, you can choose to save any layout changes associated with a file or exit without saving, which generates a warning message. If you use the File ► Save Tool option in the CodeBrowser window, Ghidra will associate the current CodeBrowser appearance with the current file in the active project. The next time you open the file, Ghidra will use the saved CodeBrowser layout.

When you have multiple CodeBrowser instances open at the same time and have modified some (or all) of them, you may end up with conflicting tool configurations. Ghidra will then display a new Save Tool dialog, as shown in Figure 12-8.

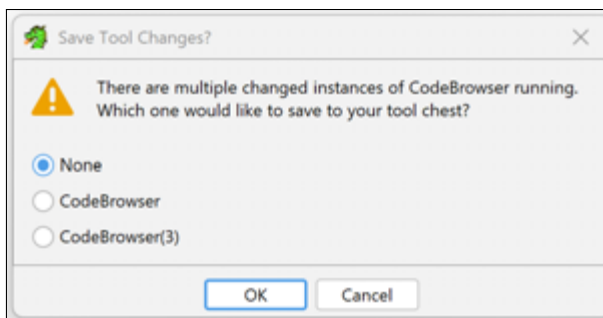


Figure 12-8: Ghidra's Save Tool Changes – Conflict dialog

Later in this chapter, we will show you how to use this and similar customization functionality to create a new powerful suite of tools tuned to your individual reverse engineering tasks and tastes.

Customizing the Project Window

Let's switch gears and venture back to the Ghidra Project window, shown in Figure 12-9. We discussed the main menu in the preceding chapter. Before we discuss Project window customizations, let's look at two areas of the window we have not yet discussed.

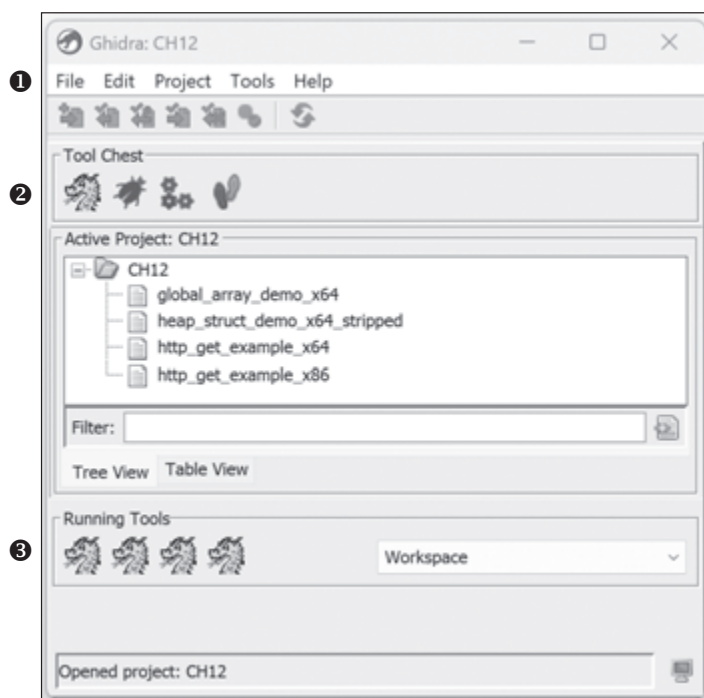


Figure 12-9: The Ghidra Project window

The Tool Chest ❷ displays icons for all of the tools capable of operating on the binaries you have imported into your projects. By default, four tools are available. The dragon icon is the default for the CodeBrowser, the bug icon launches the debugger, the gears icon is associated with Ghidra's emulator, and the footprints icon is associated with Ghidra's version control tool. We demonstrate how to supplement the Tool Chest by modifying and importing tools, as well as building our own, a little later in this chapter.

The Running Tools ❸ contains icons for each running tool instance. In this example, we have opened each of the project files in a separate CodeBrowser window. As a result, four instances of CodeBrowser are currently running. Clicking any of the Running Tools icons brings the associated tool to the foreground of your desktop.

PICK A THEME, CHANGE THE VIBE

Staring at disassembly all day can be rough on the eyes. Thankfully, Ghidra offers multiple visual themes, so you can tailor the interface to your aesthetic preferences or circadian rhythm. Prefer a bright, high-contrast look that says "I'm debugging at noon with confidence"? The Flat Light theme has you covered. It's clean, readable, and makes every hex value feel just a bit more optimistic.

On the other hand, if you do your best reversing in a dark room lit only by the glow of a dual monitor setup and an unsettling energy drink collection, the Flat Dark theme is probably more your vibe. It's easy on the eyes, mysterious, and makes every control flow anomaly feel just a bit more dramatic. As a bonus, it pairs well with existential dread and late-night CTF challenges.

Whatever your style, you can change Ghidra's theme settings from the Edit ► Theme menu. While changing the theme will not solve your reverse engineering puzzle for you, it can make the process a bit more comfortable during long analysis sessions.

Let's return to the Ghidra Project window menu ❶ and look at some of the ways we can customize the window. We will start by investigating the four Edit ► Tool Options actions for the Ghidra project shown in Figure 12-10. Two of the options are the same as in CodeBrowser: Key Bindings and Tool.

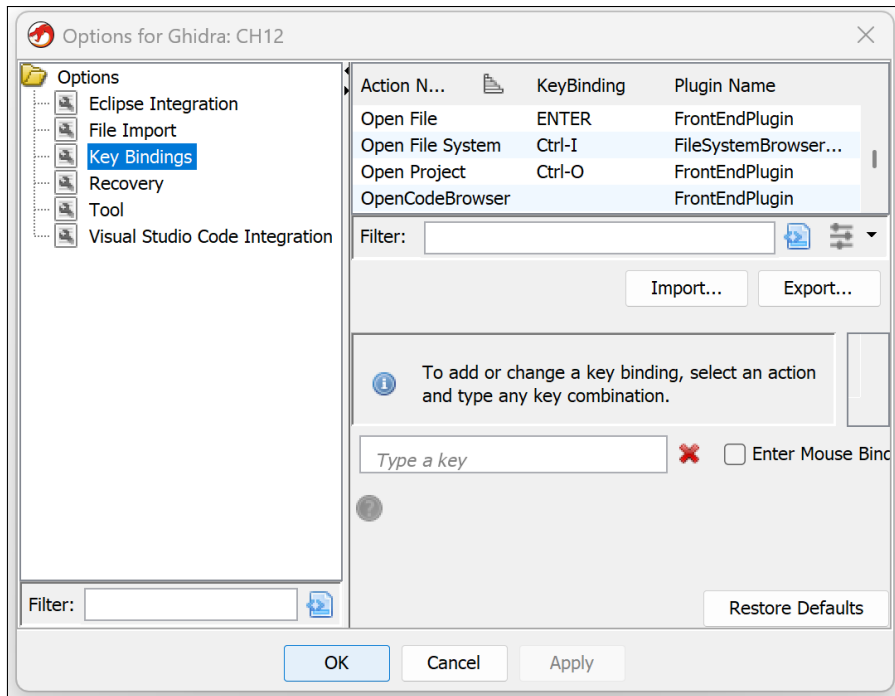


Figure 12-10: The Ghidra Project window opened via *Edit ► Tool Options*

In Figure 12-10, we have selected the Key Bindings option. The Ghidra Project tool has significantly fewer actions than the CodeBrowser tool does, and therefore fewer options for key bindings. If you're playing along at home, you may notice that most of the actions are associated with the FrontEnd-Plugin. (The Ghidra Project tool is also called the Ghidra Frontend, and these terms are used interchangeably throughout the Ghidra environment, including in Ghidra Help.)

Eclipse Integration is the focus of Chapter 15, so we will postpone discussion of the IDE options for now. File Import enables the unpacking of packed database formats during import. Recovery simply allows you to set a frequency for snapshots. The default value is five minutes. Setting this value to zero disables snapshots.

The Tool option can be quite fun to experiment with. In this case, the term refers to the Ghidra Project tool. Figure 12-11 shows the associated options.

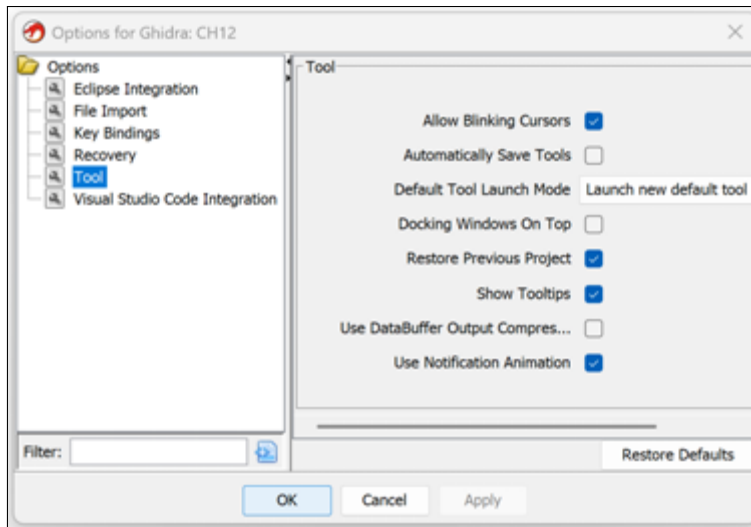


Figure 12-11: The Ghidra Project tool edit options

The File ► Configure option displays four categories of Ghidra plugin collections, as shown in Figure 12-12. Each category has a different purpose.

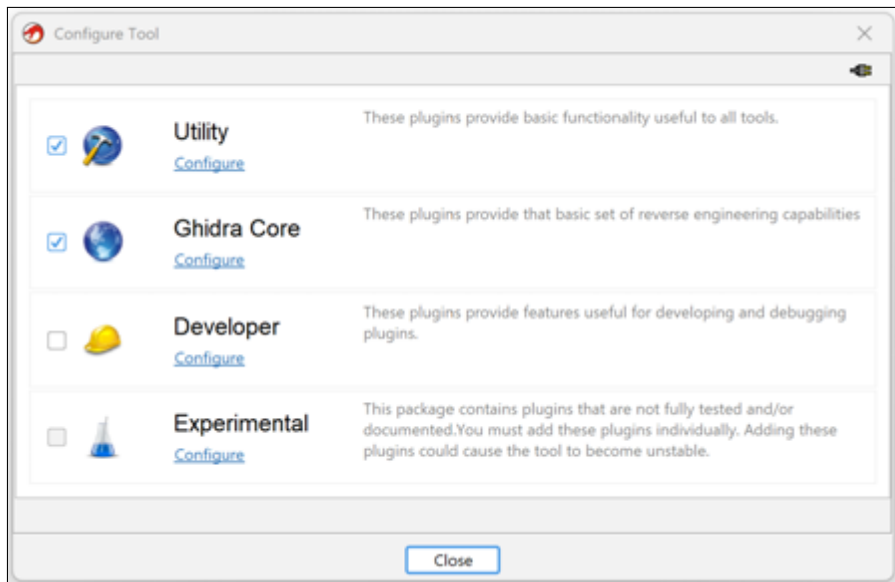


Figure 12-12: The Ghidra Project default plugin collections

Utility contains plugins that provide basic functionality for all tools. Ghidra Core contains the set of plugins that we have been using in our default Ghidra configuration. These provide the basic functionality that is essential to reverse engineering. The Developer category provides plugins that assist you in the process of developing new plugins. This is a good starting point if you want to learn more about Ghidra development. The final group of plugins is Experimental. These plugins have not been thoroughly tested and could destabilize your Ghidra instance, so use them with caution.

While only Utility and Ghidra Core are enabled as part of the default Ghidra installation, you can check the box next to the other options to enable them as well. Use the Configure option beneath a category to select (or deselect) the individual plugins that appear in the category list. Figure 12-13 displays the Ghidra Core plugins list, including a description and category for each. If you click a Ghidra plugin within this menu, a window at the bottom of the screen will provide additional information about the plugin.

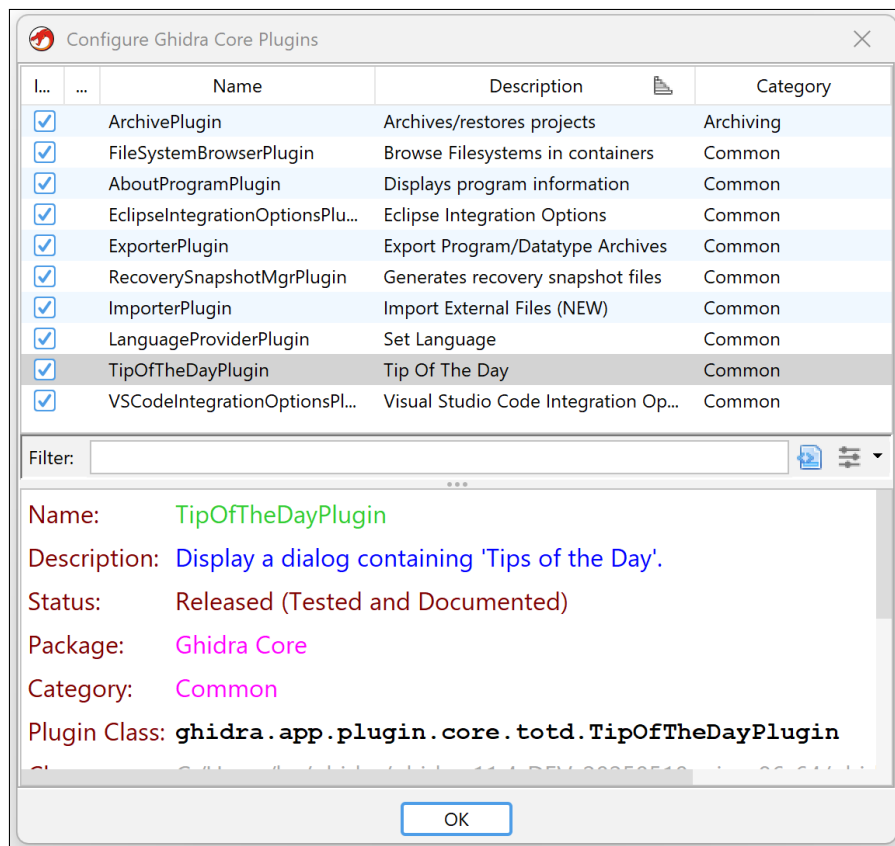


Figure 12-13: The Ghidra Core configuration window with ImporterPlugin selected

There are two additional Ghidra Project menu options you can configure. The first is File ► Install Extensions, which we discuss in Chapter 15. The other option, Edit ► Plugin Path, allows you to add, modify, and delete new user plugin paths, which tell Ghidra where to look for additional Java classes beyond its installed defaults. Through this option, you can include additional plugins and classes in your Ghidra instance. Editing the plugin path requires that you restart Ghidra in order to see the results.

Creating New Tools

The Ghidra Project window’s Tools menu, shown in Figure 12-14, allows you to create new tools if none of the existing tools exactly fit your needs.

Ghidra: CH12	
File Edit Project Tools Help	
Create Tool...	Creates an empty tool that you can populate with plugins
Run Tool	Launches a tool from your Tool Chest
Delete Tool	Removes a tool from your Tool Chest
Export Tool	Allows you to export a tool to share
Import Default Tools...	Imports default tools to your Tool Chest
Import Tool...	Imports a tool into your Tool Chest
Connect Tools...	Creates associations between tools (see Chapter 5)
Set Tool Associations...	Allows you to associate file types with specific tools

Figure 12-14: The Ghidra Tools menu options

As an example, let’s build a new tool from collections of existing plugins, rather than coding plugins from scratch.

Example 1: Building a Debugger Tool

Ghidra introduced a debugger capability in version 10. When initially released, the debugger functionality was somewhat fragile, but its reliability and associated options have improved over time. If you are relatively new to the world of debuggers, you might find the Ghidra debugger to be a bit overwhelming. Let’s address this challenge by building a specialized tool called *SimpleDebugger* to help us analyze a binary.

Choosing the Tools ► Create Tool menu option in the Ghidra Project window presents the two windows shown in Figure 12-15.

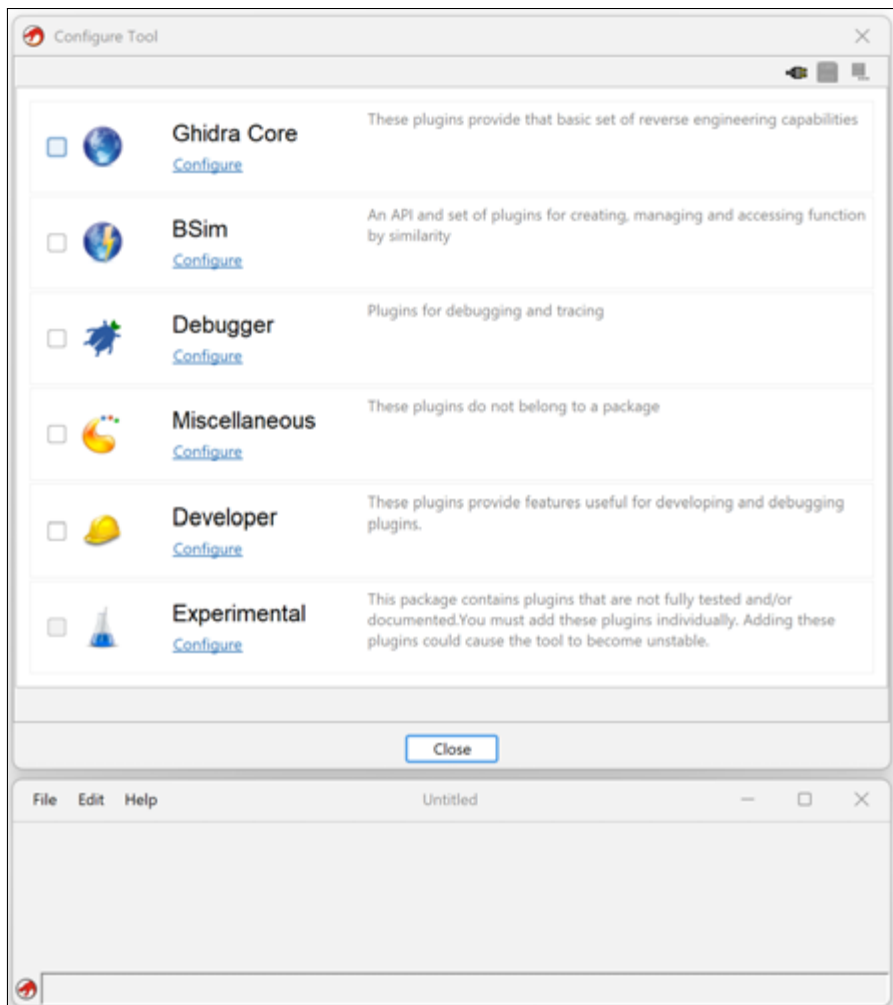


Figure 12-15: The Ghidra Create Tools windows

The upper window displays plugin options similar to those in Figure 12-12. Each option in the window represents a set of plugins useful for developing new tools containing the related functionality. Ghidra Core contains a list of plugins that make up the feature set of CodeBrowser. BSim (Behavioral Similarity) contains plugins useful for comparing binaries, which we discuss in Chapter 23. Debugger contains plugins useful in dynamic analysis contexts and is the plugin set we will use to create our new tool.

The lower window is an empty, untitled tool development window that is used to customize a tool. We are going to select two sets of plugins. First, we need the Ghidra Core plugins to ensure that we can continue to use the CodeBrowser functionality that we have been using. The new addition to our tool is the collection of Debugger plugins. Checking these two options populates the tool development window with all options from Debugger, as shown in Figure 12-16.

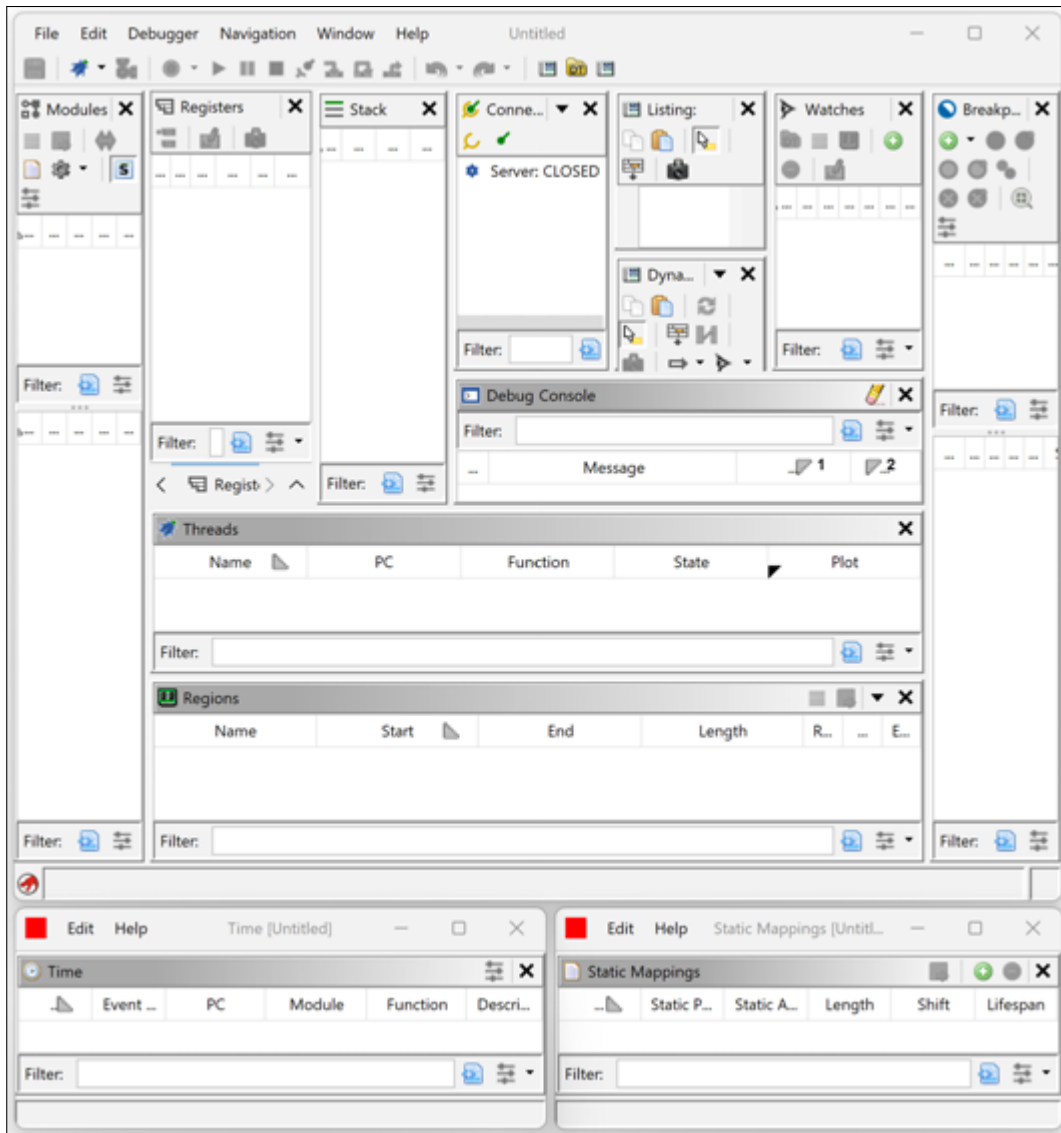


Figure 12-16: A new, untitled debugger tool before configuration

The resulting plethora of windows can be a bit confusing, and they provide far more functionality and information than we will need for Simple-Debugger. We can remove some of the unnecessary windows and plugins for our simplified debugger by clicking Configure under Debugger in the Configure options window. We will start by deleting the following:

- DebuggerStaticMappingPlugin
- DebuggerTimePlugin
- DebuggerThreadsPlugin
- DebuggerRegionsPlugin

As we begin to delete plugins, Ghidra will let us know if there are potential unanticipated side effects. Attempting to remove a plugin that is associated with another plugin will result in Ghidra displaying a warning message with the list of additional plugins that will be removed. Figure 12-17 shows an example of the warning message generated when attempting to remove the important `DebuggerPlatformServicePlugin`.

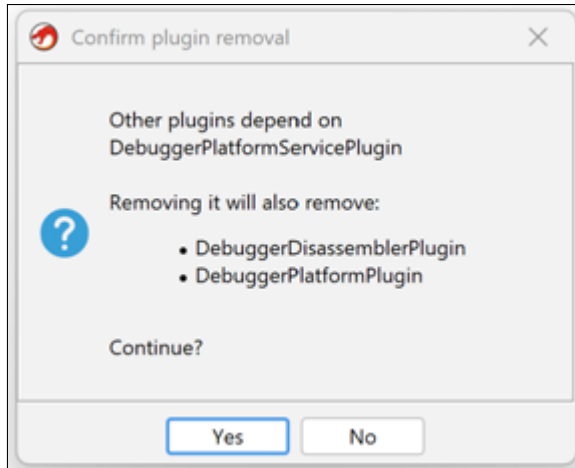


Figure 12-17: The plugin dependency warning for `DebuggerPlatformServicePlugin`

If a plugin is removed by mistake, it isn't a major concern. Plugins can be added back in by choosing **File ► Configure** from your new tool at any time and selecting the plugins. Another way to remove unneeded content such as windows, and their associated plugins, from the tool is by manually closing them. You can always add them back in using the **Windows** menu if you change your mind.

You can also control the layout of your new tool to better process what you are seeing. Using the techniques described in previous chapters, you can drag the windows into the desired locations and resize them as needed. Figure 12-18 shows a new, untitled debugger tool that is much easier to process and understand.

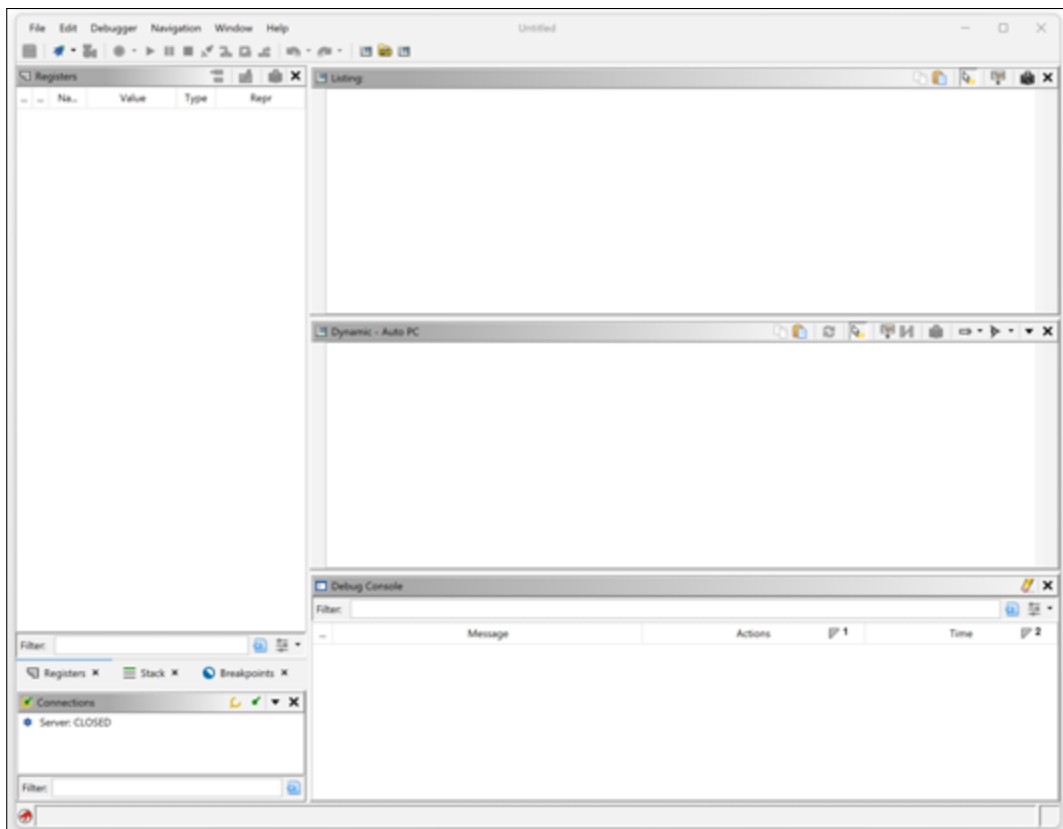


Figure 12-18: An untitled debugger tool with windows removed, resized, and rearranged

For this tool, we have chosen to include the following windows:

Stack window This window allows us to see the function call stack. The window shares space with the Breakpoints window, as we aren't going to need to see both of the windows at the same time.

Registers This window will allow us to view the registers, their associated contents, and how they change as we execute the code.

Breakpoints This window shares space with the Stack window and will be used to display Breakpoints as we set them.

Listing This is our familiar friend from the CodeBrowser tool that we use to view and navigate the disassembly.

Connections This window keeps us informed about the status of our connection to our server and will be one of the first windows we use as we launch our tool.

Dynamic This window shows memory and disassembly in the process by tracking the instruction pointer.

Debug Console This window shows the communication between Ghidra and the underlying debugger during a debugging session.

If you plan to use a tool frequently or wish to share it with your collaborators, you should save the tool by selecting File ► Save Tool As. This selection presents you with the options to name the tool and associate an icon with it (see Figure 12-19).

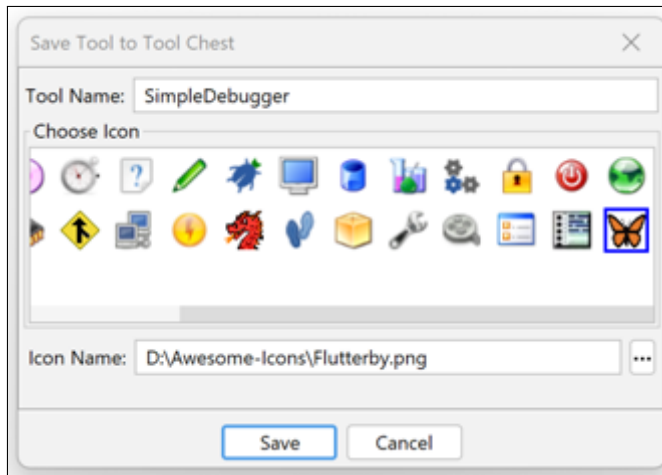


Figure 12-19: Icon options for new tools

You can choose from among the provided icons or select your own image file in a supported format (for example, *.jpg*, *.png*, or *.gif*). This new tool becomes part of your Tool Chest and will be displayed as an option in your projects, as shown in Figure 12-20.

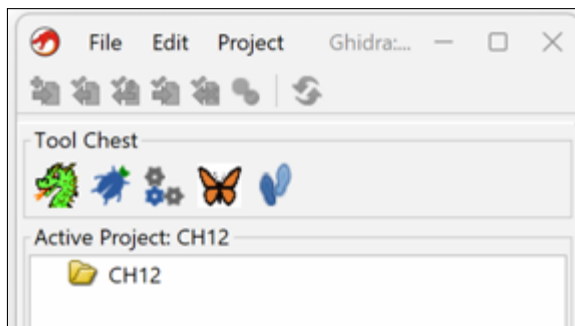


Figure 12-20: A project with the new tool options displayed in the Tool Chest

To share a tool with others, you can export it using Tools ► Export Tools. Ghidra will ask you to choose a folder in which to save the tool and then create a *.tool* file containing your tool specification. You can share this file with others who can then use the Tools ► Import Tool option to add it to their set of tools.

Now that you know how to create a tool, let's move on to actually using the tool.

Example 2: Performing Dynamic Analysis with the Debugger Tool

By default, double-clicking a file in the Ghidra Project window opens the file in the CodeBrowser, but you can choose to open a file using any tool in your Tool Chest from the right-click context menu. Alternatively, you can drag the file and drop it onto a tool.

To demonstrate the use of the SimpleDebugger tool we just created, let's revisit the stack frame we analyzed in depth in Chapter 6 (on page 110), derived from the following source code:

```
int compute_area(int width, int height) {
    int area = width * height;
    int adjustment = 2;
    area += adjustment;
    return area;
}

int main() {
    int result = compute_area(5, 10);
    printf("Computed area: %d\n", result);
    return 0;
}
```

When we first analyzed the `compute` function, we examined the stack frame that Ghidra created after completing its auto analysis. Because we were focused on static analysis, which involves studying the program's code, structure, and data without actually running it, we relied on Ghidra's initial interpretation of the stack. We then refined this view by editing the stack frame and mentally stepping through the code to understand how the values were used. (See Figures 6-5 and 6-6.)

Ghidra's decompiler, Listing view, and Edit Stack window provide powerful support for static analysis by showing a program's instructions, control flow, and variable layout. This approach is safe and effective for understanding what the binary might do. However, it has limitations, especially when dealing with runtime behavior such as encrypted data, code that is generated or modified during execution, or indirect jumps that are hard to resolve just by reading the disassembly.

This is where a debugger becomes valuable. Running the program step by step on a real system allows you to observe exactly what it does as it executes. You can watch register values change, examine memory contents,

set breakpoints at key points of interest, and even modify values to test your understanding of the code's behavior.

WARNING

When you run code in a debugger, including in the SimpleDebugger tool you just made, you are actually executing it on your system, unlike emulation mode, which only simulates execution without running the real instructions. Always be sure you trust the code or use a safe, isolated environment like a virtual machine to protect your system from potential harm.

Using a debugger can help you confirm or expand on what you discovered during static analysis. For example, during our initial static analysis, we made a prediction about how the function prologue would affect the stack frame. Now we can use our SimpleDebugger tool to check (and hopefully confirm) our understanding of the binary.

To ensure the SimpleDebugger works properly, we have done some behind-the-scenes setup and configuration on this example system. The steps needed to prepare the environment for debugging can vary depending on the system and tools being used, and you can find details about these steps in Ghidra Help, as well as in the *docs/GhidraClass* directory.

BEHIND THE DEBUGGER CURTAIN

Think of Ghidra's Debugger tool as the master controller at mission command. Instead of building a debugger from scratch, Ghidra relies on well-established tools like GDB or WinDbg to handle the actual work of interacting with your running code. It sets up the connections, launches the debugger with the correct commands and options, and then coordinates all the data so you can see what is happening inside your program.

When you run the debugger, Ghidra talks to the underlying tool to keep track of registers, memory, threads, and breakpoints. You can watch all of this via Ghidra's familiar Listing, Memory, and Console windows without needing to jump back and forth between separate tools or type every command by hand.

Behind the scenes, Ghidra acts like a skilled handler for your target program. It lets you control when to pause execution, where to inspect memory, and how to step through instructions one piece at a time. Ghidra brings everything together in one place, but you are still the one pressing the big red button to run that unknown binary, so use that power wisely.

We will use a Linux system to run the program we want to analyze. That system has `gdb` installed and an SSH server running so we can connect to it remotely. On the host side, we are running Ghidra on a Windows machine and connecting to the Linux system via SSH. We have also installed some required Python packages, including *protobuf*, *psutil*, and *ghidragdb*, so that Ghidra's debugger can communicate with the remote target.

Figure 12-21 shows the new SimpleDebugger tool view after we have opened and analyzed the *compute* binary.

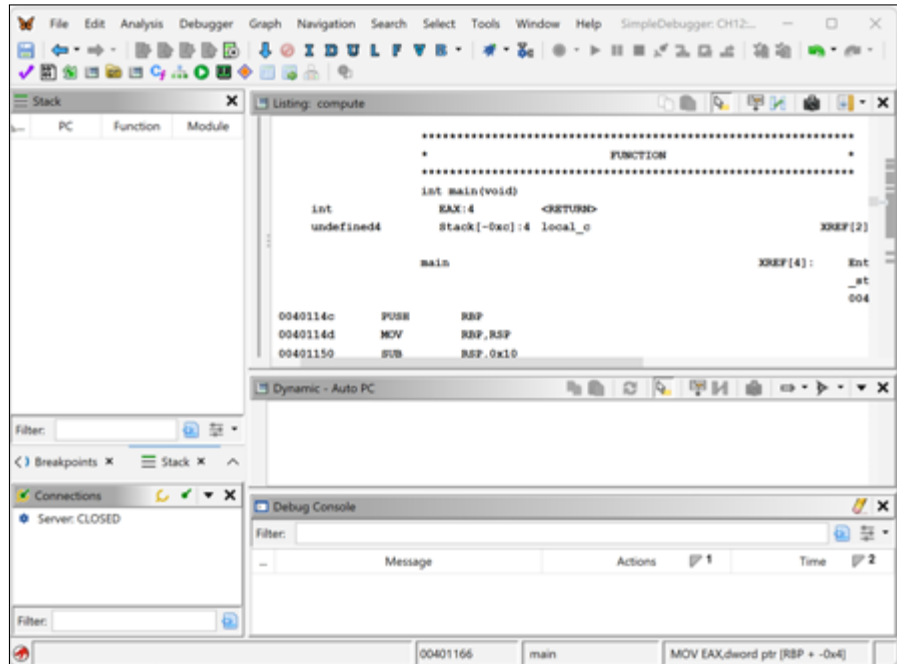


Figure 12-21: The SimpleDebugger tool with a populated Listing window

Only the Listing window is fully populated at this point because we are still looking at the program as it would appear at the start of a standard CodeBrowser static analysis task. Ghidra has already done a lot of work to analyze the code and set up important details like the stack frame, as we saw in Chapter 6. Opening the Stack Editor window would allow us to see information identical to our initial analysis, even before we configure and launch the debugger functionality.

Dynamic details like the actual values in registers, the current contents of memory, the list of active threads, and the exact execution state are not available yet, however. To see those live values and interact with the program while it runs, we still need to choose and launch an actual debugger. Once the debugger is connected, Ghidra will pull in real-time data from the running process, which will populate the rest of the windows with up-to-date information that we can monitor and control step by step.

Ghidra can work with different backend debuggers depending on the desired system and target. Figure 12-22 shows the list of debugger connectors that Ghidra currently supports, so you can choose the one that best fits your setup and workflow.

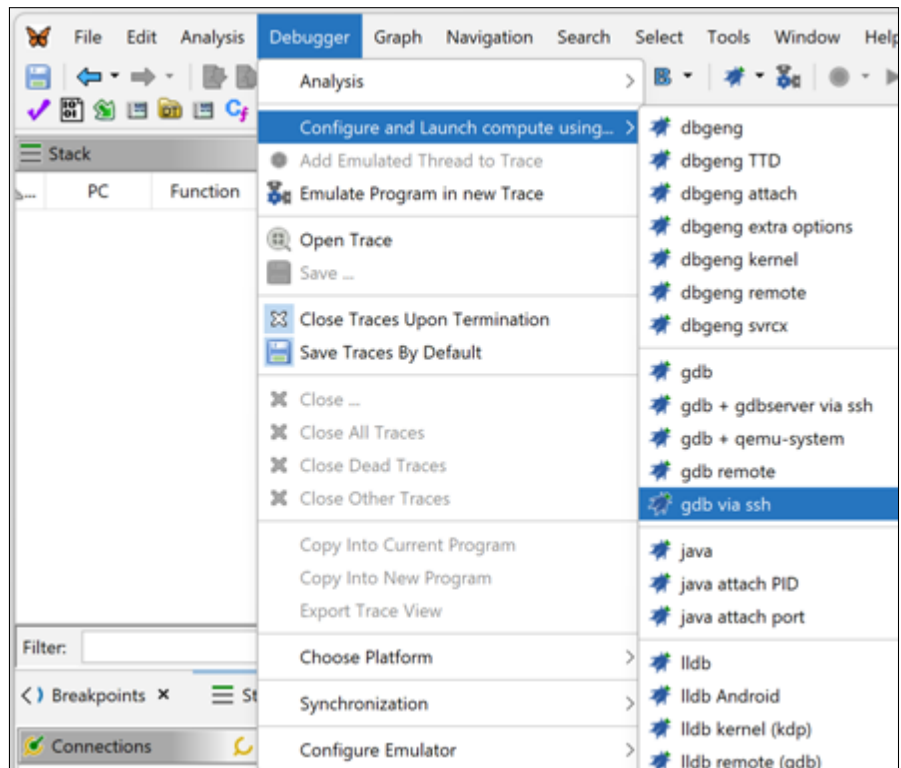


Figure 12-22: The Debugger configuration and launch options

We will choose the highlighted option to configure and launch our *compute* binary using the *gdb via ssh* option. After we complete a short dialog and provide our SSH credentials, Ghidra will connect to the target as specified. Figure 12-23 shows a successful connection, as indicated by a populated SimpleDebugger tool, including a new terminal window Ghidra adds to the bottom. This window shows the *gdb* prompt, confirming that we are on track to begin our dynamic analysis.

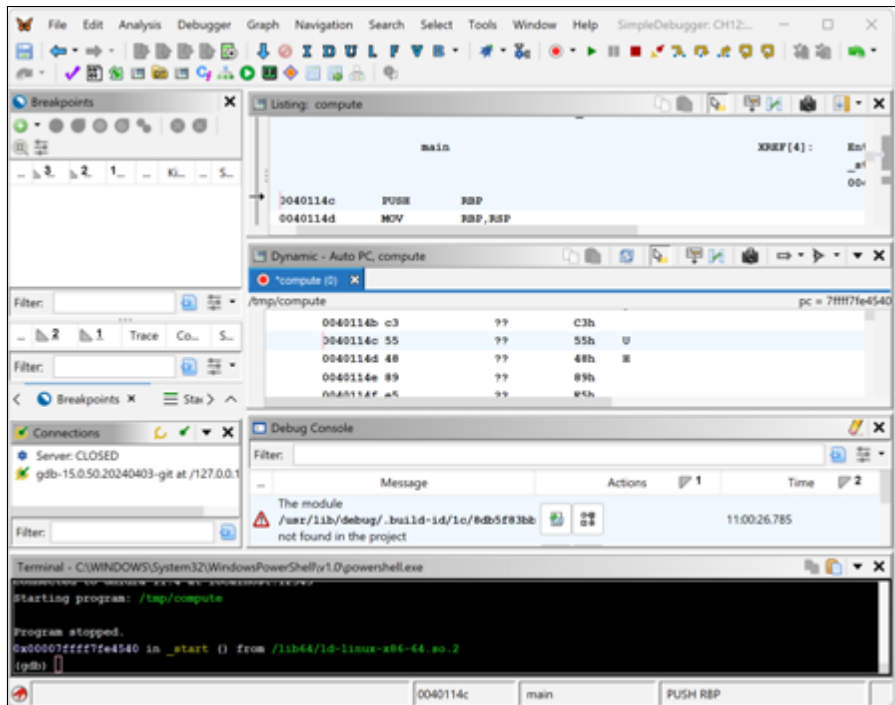


Figure 12-23: The populated SimpleDebugger tool with a new Terminal window

Let's set a breakpoint at the start of the function prologue in the compute_area function. A *breakpoint* is a marker placed in the code that tells the debugger to pause execution when the program reaches that point. This gives you a chance to inspect what is happening at a specific moment, such as by checking register values, examining the stack, or looking at memory. Breakpoints are one of the most useful options available for controlling how the program runs while during analysis. Right-clicking the first instruction in compute_area and choosing Set Breakpoint ► SW_EXECUTE brings up the dialog shown in Figure 12-24.

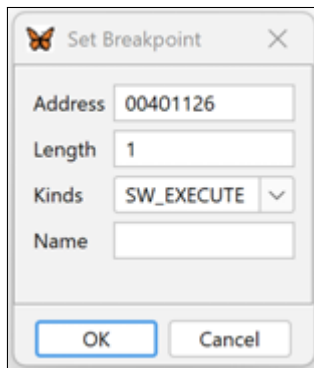


Figure 12-24: The Set Breakpoint dialog

If you are planning to create a lot of breakpoints during your debugging session, it can be helpful to name them. In this case, we will set just one breakpoint, so we will leave the Name field blank.

Let's make a few quick observations before we actually begin our dynamic analysis. First, recall that the function prologue is the part of the function that sets up the stack frame and prepares the function's environment. By stopping execution at our specified breakpoint, we can walk through the prologue instructions step by step and see exactly how the stack and registers are initialized before the rest of the function runs.

Second, we want to be able to observe changes in memory as we analyze this program. Unfortunately, the Memory window is not part of our original SimpleDebugger tool configuration. One of the nice things about Ghidra is that you can add additional windows and features to tools at any time, as needed. To inspect raw memory while debugging, we can easily add the Memory window to our current layout using Window ► Debugger ► New Memory View, which will stack the new window with the Dynamic window by default.

After making meaningful changes to a tool, such as adjusting its layout, adding useful windows, or customizing its behavior, we can save it in its new configuration so we do not have to repeat those steps later. This is especially helpful when setting up a tool for a particular type of analysis. Ghidra also allows us to return to the original version of any tool at any time, giving us the flexibility to adapt our workspace without making permanent changes. Figure 12-25 shows our new SimpleDebugger tool configuration.

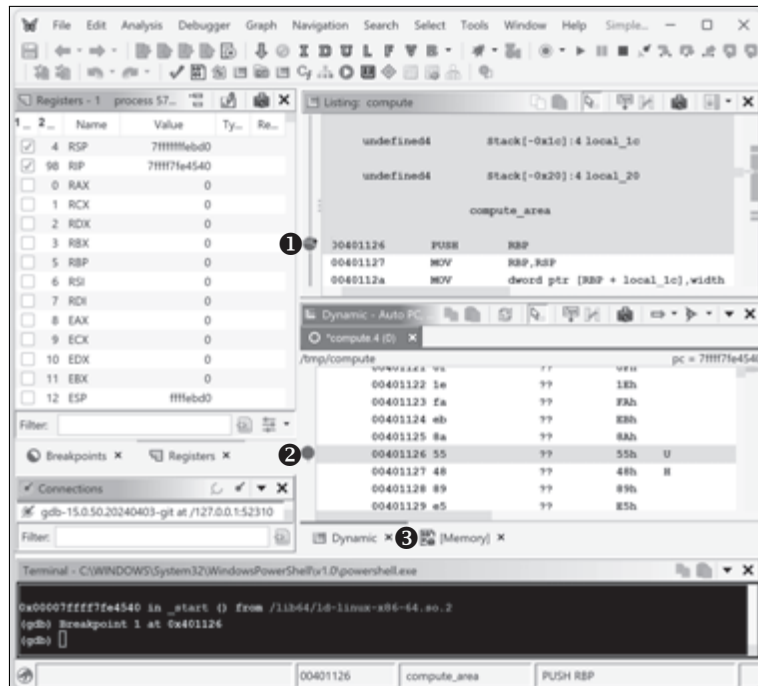


Figure 12-25: The new SimpleDebugger tool view

We can see that the breakpoint ❶ is set at the start of the `compute_area` function prologue, a fact also reflected in the Dynamic window ❷. Note that we have closed the Stack and Debug Console windows to keep the focus on the windows that are relevant to this walkthrough. Clicking the Memory window tab ❸ moves the Memory window in front of the Dynamic window, as shown in Figure 12-26.

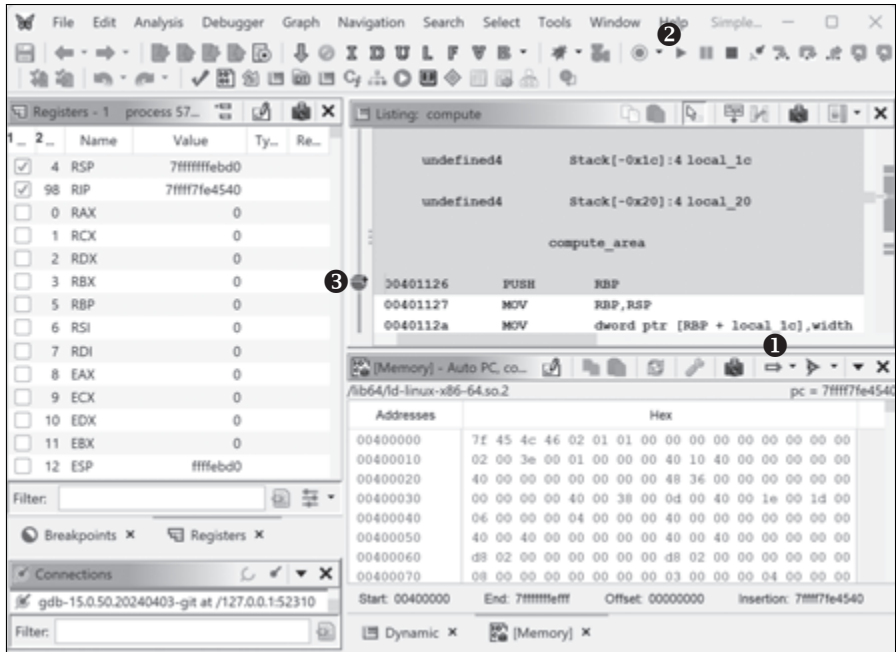


Figure 12-26: The SimpleDebugger tools with Memory window in focus

Since we are interested in analyzing the stack frame, we will set up the Memory window to track the Stack Pointer by clicking the Track Location icon ❶ and selecting Track Stack Pointer. This will allow us to view the memory around the stack pointer as we work through the example.

Now that we have set up our SimpleDebugger window to be able to validate our earlier findings, we are ready to actually start using the tool. At this point, the program is stopped before execution, essentially at the entry point. Clicking the “Continue execution from the inferior” icon ❷ will begin execution at the start of `main`, call `compute_area`, and stop at the breakpoint ❸ before the `PUSH RBP` instruction executes. Figure 12-27 shows the resulting window when the breakpoint is reached.

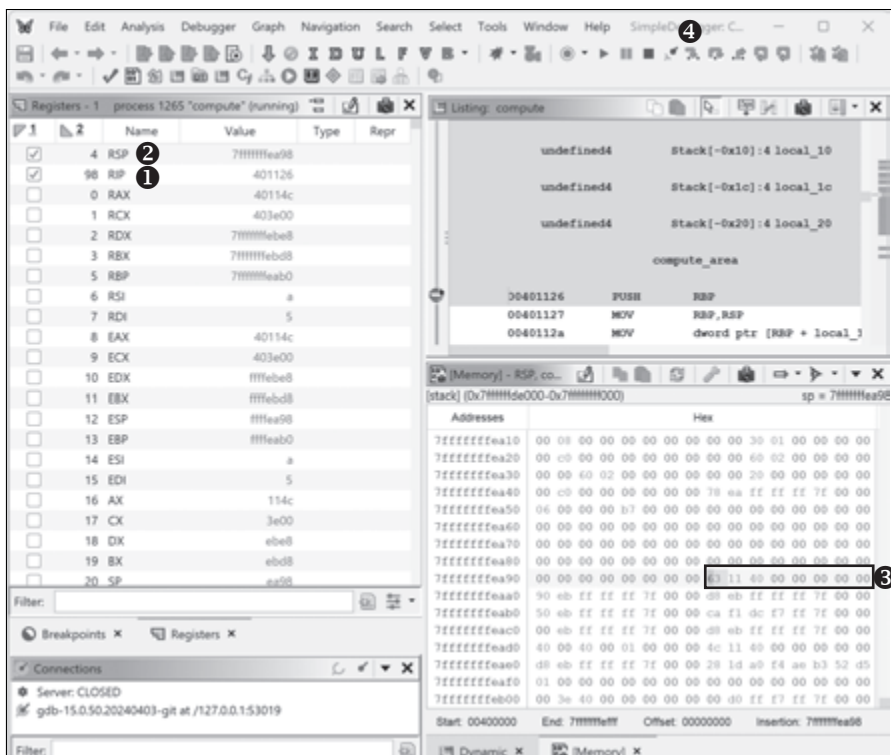


Figure 12-27: The binary status at the breakpoint

The current value stored in RIP is 401126 ❶. This indicates that the next instruction to be executed will be the first instruction in the `compute_area` function prologue, `PUSH RBP`.

In the Register window, we can now see that the stack pointer (RSP) holds the address 7fffffff98 ❷. Looking at the contents of that address in the Memory window shows that RSP is pointing at the bytes 63 11 40 00 00 00 00 00 ❸. A quick translation from little endian gives us 0x401163, which should be the return address. Looking at `main` in the disassembly, we can see that the address immediately following the call to `compute_area` is 00401163, confirming our initial analysis:

00401159	MOV	EDI,0x5
0040115e	CALL	compute_area
00401163	MOV	dword ptr [RBP + local_c],EAX

We are now ready to continue observing the stack as we execute the code. Instead of running the program up to a breakpoint, we will slow down the execution process so we can examine the effects of each instruction in the function prologue. Our Debugger menu includes options to control execution. We will choose to use the “Step one instruction exactly” icon ❹ to execute one instruction.

Figure 12-28 shows the contents of the registers and memory after executing the first instruction in the `compute_area` function prologue.

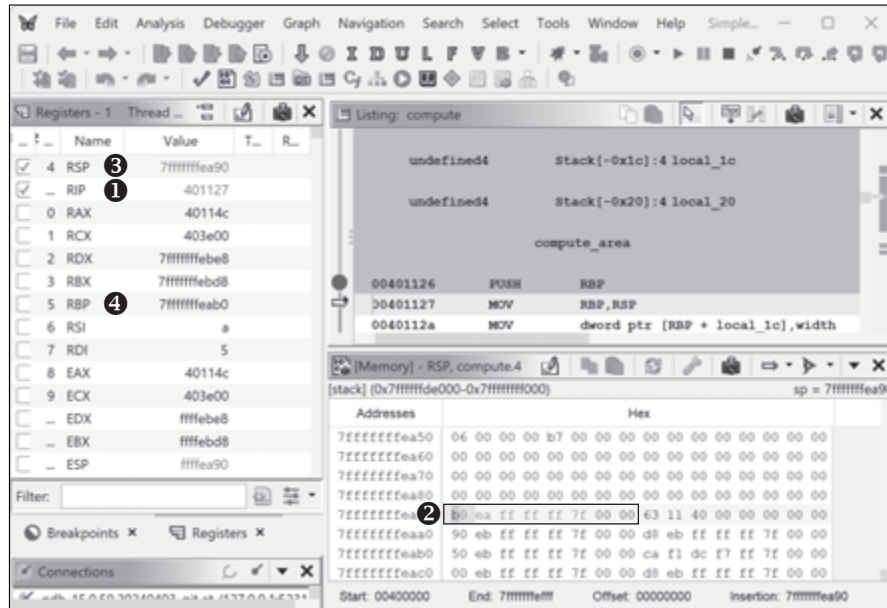


Figure 12-28: The SimpleDebugger after the first prologue instruction executes

We can see that the value in the instruction pointer, RIP, has changed to 401127 ❶. (This will be the next instruction to execute.) We can also confirm that we have executed the `PUSH RBP` at 401126 by inspecting the Memory window. In this window, we can see `f7ffffff9a90` ❷ written onto the stack (in red font by default), and that the stack pointer stored in RSP ❸, has changed to `7fffffff9a90` because we pushed 8 bytes onto the stack. RBP ❹ has not changed yet, so we can see it still contains the address `7fffffff9a90`, which is the base of `main`'s stack frame.

At this point, we have executed a single instruction in `compute_area`, `PUSH RBP`. Now we can step forward one instruction to see how the prologue finishes its setup.

The second instruction establishes a new base pointer for the current function, which creates a stable reference point for accessing parameters and local variables. Stepping through these instructions one at a time helps you see exactly how the stack frame is organized before the rest of the function begins its work.

Figure 12-29 shows the updated view after stepping forward one more instruction, thus completing the execution of the `compute_area` function prologue.

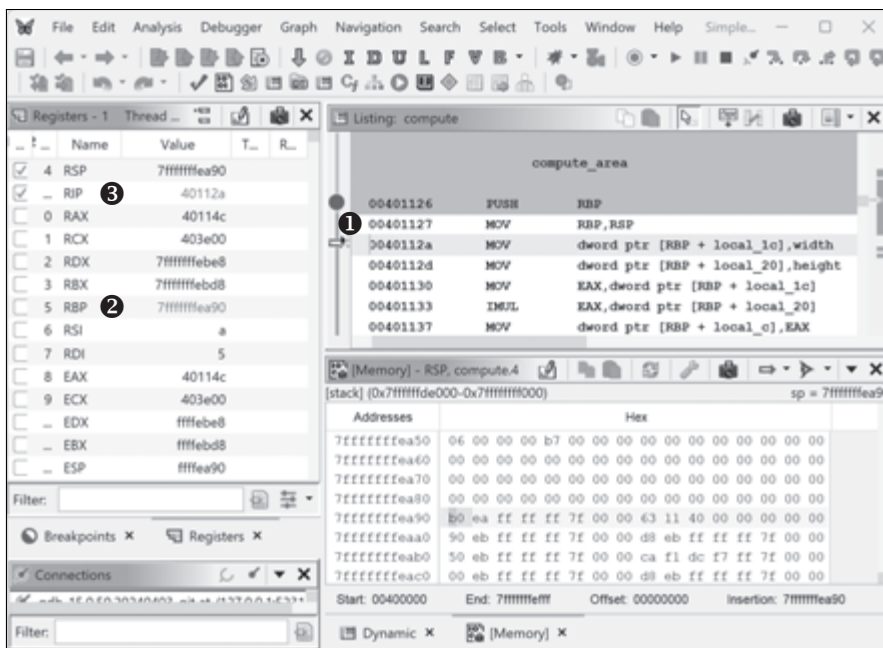


Figure 12-29: The SimpleDebugger after the first function prologue has completed execution

The second instruction in the prologue is `MOV RBP, RSP` ❶. We can confirm the results of this instruction by investigating the Register window. In that window, the content in RBP ❷ has been updated to `7fffffff90`, the base of the stack frame for `compute_area`. The rest of the stack is unchanged. We have now completed the prologue for `compute_area`, and the instruction pointer, RIP ❸, contains the address of the next instruction to execute, `40112a`:

```

00401126  PUSH  RBP
00401127  MOV   RBP, RSP
0040112a  MOV   dword ptr [RBP + local_1c], param_1
0040112d  MOV   dword ptr [RBP + local_20], param_2
00401130  MOV   EAX, dword ptr [RBP + local_1c]
00401133  IMUL  EAX, dword ptr [RBP + local_20]
00401137  MOV   dword ptr [RBP + local_c], EAX
0040113a  MOV   dword ptr [RBP + local_10], 0x2
00401141  MOV   EAX, dword ptr [RBP + local_10]
00401144  ADD   dword ptr [RBP + local_c], EAX
00401147  MOV   EAX, dword ptr [RBP + local_c]
❶ 0040114a  POP   RBP
0040114b  RET

```

We want to continue executing `compute_area` in the debugger until just before the `POP RBP` ❶ is executed. There are a few ways to do this:

- Single step several times to get to that address.
- Set another breakpoint at 40114a ❶, and use the “Continue execution of the inferior” icon we used to execute up to the first breakpoint.
- Use the “Continue execution up to the given address” button in the debugger toolbar and enter the desired address.

The “Continue execution of the inferior” icon we used to run the program up to our initial breakpoint might seem like a good option again. Another idea might be to set a breakpoint in `main` just after the call to `compute_area`. However, neither approach is ideal for what we are trying to observe. Once we return to `main` from the function, the stack frame for `compute_area` will no longer be visible. Because we want to be able to view the stack contents, and we know we want to stop before the instruction at location 40114a, we can single step through the function until we see 40114a in the instruction pointer RIP.

As we continue single stepping through the remaining instructions, watching RIP for the desired address, we can observe how the function sets up any additional values it needs. With each step forward, we can see how parameters are stored, local variables are given space on the stack, and any necessary values are initialized. This careful single stepping allows us to observe exactly how the stack frame fills out in memory, helping us connect what we see in the Listing and Stack windows to how the function runs. Figure 12-30 shows the SimpleDebugger tool before the return to `main`. (The Stack Editor window is included for reference.)

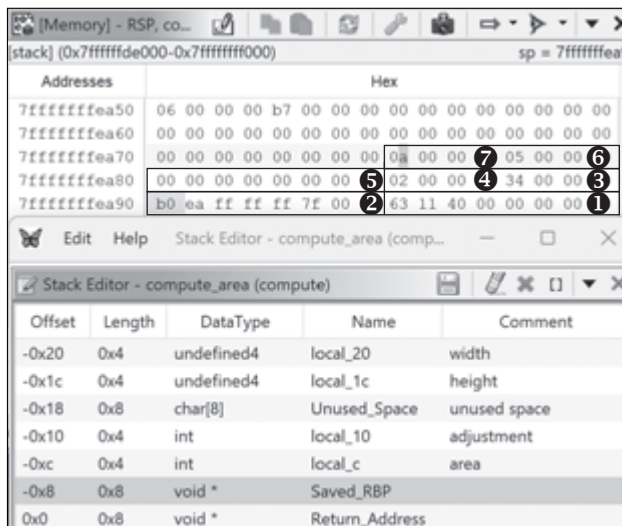


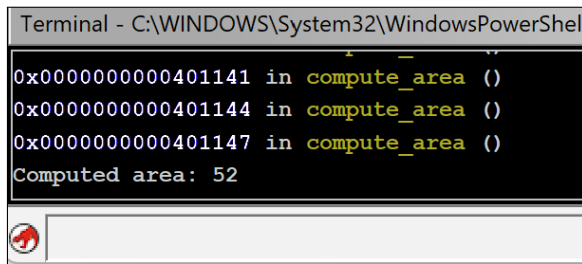
Figure 12-30: The SimpleDebugger tool before the function epilogue, with the Stack Editor window

Recall that the `compute_area` function takes the width (5) and height (10) values, multiplies them together, adds the adjustment value (2), and then returns the final result to `main`. Since we have now executed all of the `compute_area` instructions except for the `POP RBP` and the `RET`, we can see the following in the Memory window:

- The return address, 000000401163 ❶
- The saved base pointer for `main` (`RBP = 7fffffffefab0`) ❷
- The calculated result for area (`00000034 = decimal 52`) ❸
- The adjustment value, (`00000002 = decimal 2`) ❹
- The unused stack space (`000000000000000000`) ❺
- The width (`00000005 = decimal 5`) ❻
- The height (`0000000a = decimal 10`) ❼

The Stack Editor window does not display all of the names and detailed data types by default, so we have edited some of the `Data Type` and `Name` fields and added comments to show that Ghidra's initial auto analysis correctly allocated the space that the function `compute_area` needed, and also to demonstrate that we can edit the Stack Editor window to make things clearer.

Figure 12-31 shows the SimpleDebugger after we click the `Execute` button to run the program to completion.



```
Terminal - C:\WINDOWS\System32\WindowsPowerShell
0x0000000000401141 in compute_area ()
0x0000000000401144 in compute_area ()
0x0000000000401147 in compute_area ()
Computed area: 52
```

Figure 12-31: The Terminal window showing the final result

At the bottom of the tool, we can see the output of the program in the Terminal window displaying `Computed area: 52`.

In this example, Ghidra made it easy to combine static and dynamic analysis approaches. You could see how the static stack setup matches the real execution flow, step through instructions, and watch values change in real time.

While we used the SimpleDebugger tool to run real code, it is also possible to use a debugger for emulation, which lets you simulate how code would run without actually executing it on your system. Emulation can help you safely trace through instructions, test decode routines, or analyze tricky obfuscated code. In Chapter 21, you will see an example of how to use Ghidra's debugger in emulation mode to explore code behavior in a controlled way.

Just as using the debugger adds flexibility to your analysis, being able to adjust Ghidra's look and tools gives you the freedom to work in a way that

fits what you need. We have spent much of this chapter and earlier chapters discussing how you can change Ghidra's appearance and design custom tools. A final step in making these customizations useful is to save the configurations so you can easily switch to the right setup for each project. You can do this through the creation and preservation of Ghidra workspaces.

Saving Workspaces

A Ghidra *workspace* is like a virtual desktop that includes a set of configured tools and associated files. In some cases, you might find it useful to create separate workspaces for each analysis issue you encounter.

For example, imagine that you are analyzing a binary with your new debugger tool, notice some interesting characteristics, and wish you could look at the file in CodeBrowser to see an effect in the Decompiler window. One way to proceed would be to preserve your current analysis by selecting Project ► Workspace ► Add from the Ghidra Project window and giving the new workspace a name. Let's call this workspace Debugger-BN. You can then open your other tool (perhaps a different view of CodeBrowser with windows that supplement your Debugger tool) and create a second workspace called Debugger-KR using the same method.

To easily switch between the workspaces, choose between them in the pull-down menu shown in Figure 12-32.

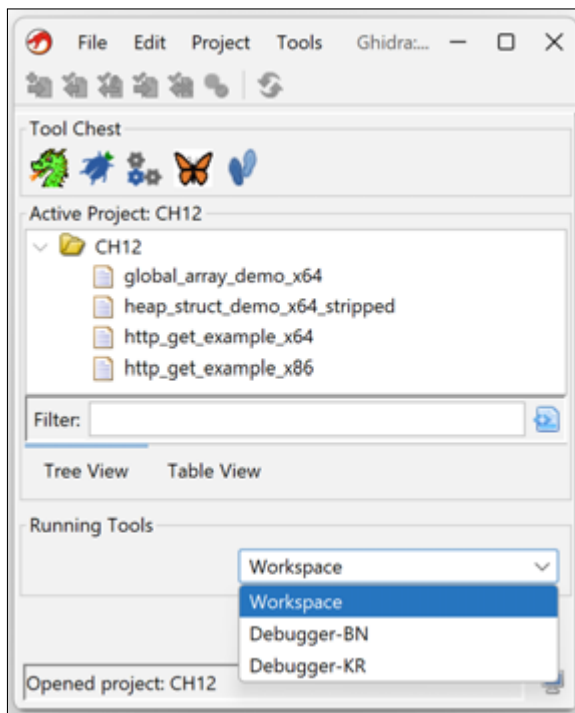


Figure 12-32: The workspace options within the Ghidra Project window

You can also use the Switch option in the Project ► Workspace menu to cycle through the available workspaces.

Summary

When you first start using Ghidra, you may find that the default behaviors and the standard CodeBrowser layout meet your needs just fine. As you become more comfortable with its features, though, you will almost certainly find opportunities to adjust how Ghidra works to better match your reverse engineering workflow.

While there is no way for a single chapter to provide complete coverage of every possible option Ghidra offers, we have introduced the customization capabilities you will most likely need at some point in your SRE experience. You have now seen how to edit existing tools like CodeBrowser and build entirely new tools, such as SimpleDebugger, with the specific features you need to address a particular type of analysis. We leave the discovery of additional useful tools and options as a matter of exploration for inquisitive readers.

13

EXTENDING GHIDRA'S WORLDVIEW



One of the features we hope a high-quality reverse engineering tool has is the ability to fully automate the identification and annotation of as much of a binary as possible. In ideal cases, the tool would identify 100 percent of instructions and group them into 100 percent of the original functions that compose the binary. It would give each of these functions a name and a full prototype, and it would identify all data manipulated by the functions to provide a full understanding of the original data types used by the programmers. This is precisely Ghidra's goal, beginning with the initial import of a binary and continuing through auto analysis. After that point, anything that Ghidra was unable to accomplish becomes an exercise for its user.

In this chapter, we look at the techniques that Ghidra uses to identify various constructs within binaries and discuss how you can enhance its ability to do so. We begin with a discussion of the initial loading and analysis processes. The choices you make during these steps help determine what resources Ghidra will bring to the table for the file you are analyzing. This is your opportunity to provide Ghidra with information that it may have failed to detect automatically so that Ghidra's analysis stages can make more informed decisions. Following that, we will look at how Ghidra utilizes word models, data types, and function identification algorithms, and how you may enhance each of these to tailor its performance to your particular reverse engineering application.

Importing Files

During import, the dialog shown in Figure 13-1 presents Ghidra's initial analysis of the file's identity, which will guide the file loading process. You can override any of the fields or proceed with the recommendations Ghidra has made. The additional options, accessed with the Options button, are specific to the type of file being loaded. Figure 13-1 shows options for a PE file.

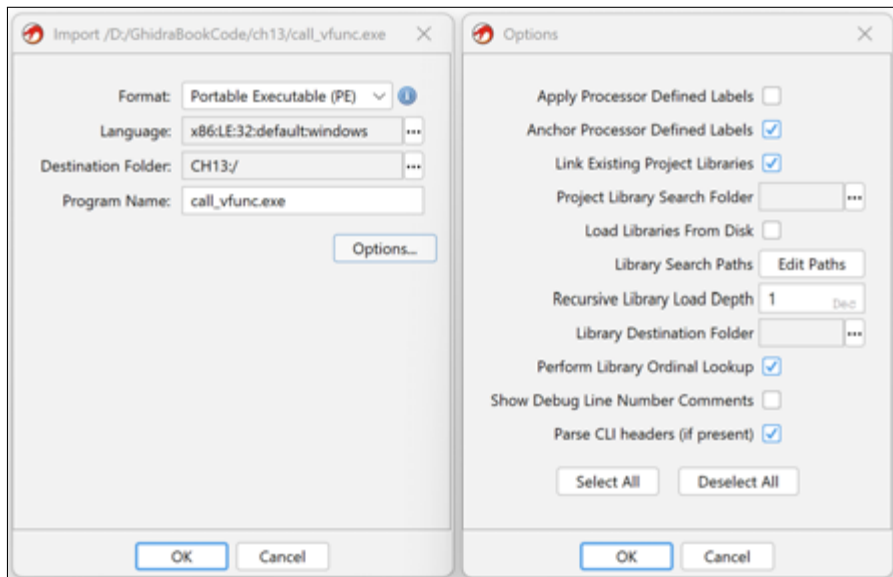


Figure 13-1: The Import dialog and options for a PE file

Figure 13-2 shows options for loading an ELF binary.

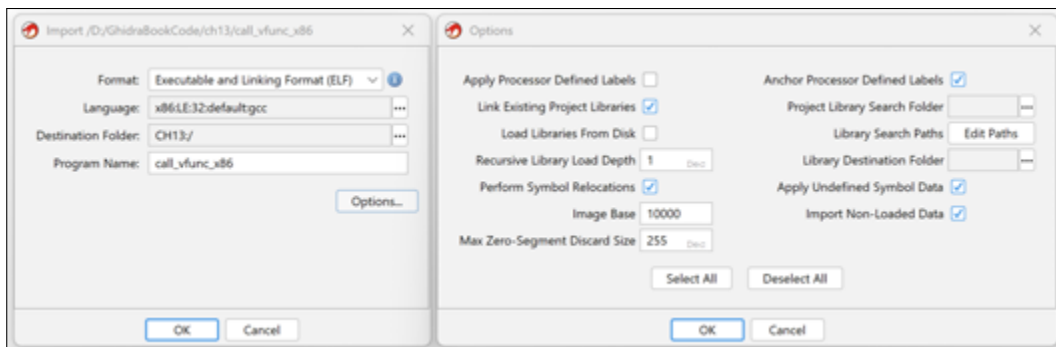


Figure 13-2: The Import dialog and options for an ELF binary

LANGUAGE/COMPILER SPECIFICATIONS

The Language field in Figures 13-1 and 13-2 dictates exactly how Ghidra will interpret any bytes recognized as machine code within the file you are loading. The language/compiler specification is composed of three to five colon-separated subfields, as described here:

- The processor name field names the processor type for which the binary was built. It directs Ghidra to a specific subdirectory under *Ghidra/Processors*.
- The endian field indicates the endianness of the binary's processor, which is either little-endian (LE) or big-endian (BE).
- The architecture size (bitness) field usually coincides with the size of a pointer for the chosen processor (16/32/64 bits).
- The processor variant/mode field is used to choose a specific model of the selected processor or identify a specific mode of operation. For example, when the x86 processor is selected, we can choose from System Management Mode, Real Mode, Protected Mode, or default. For the ARM processor, we can choose models v4, v4T, v5, v5T, v6, Cortex, v7, v8, or v8T, among others.
- When known, the compiler field names the compiler, or in some cases a calling convention, used to compile the binary. Valid names include *Visual Studio*, *gcc*, *golang*, *borlandcpp*, *Delphi*, *Swift*, *default*, and other processor-specific options.

Figure 13-3 breaks down the language identifier ARM:LE32:v7:default into its component subfields. One of a loader's most important jobs is to infer a correct language/compiler specification.

Language				Compiler
Processor	Endian	Size	Variant	
ARM	LE	32	v7	Default

Figure 13-3: An example of a language/compiler specification

The Format option specifies which loader Ghidra will use to import the file. Ghidra relies on a loader's detailed knowledge of a particular file format to identify characteristics of the file and choose the proper plugins to use for analysis. A well-written loader recognizes specific content or structural features to identify the file's type, architecture, and, hopefully, the compiler used to create the binary.

Information about the compiler can enhance function identification. To fingerprint a compiler, a loader examines the structure of a binary to look for compiler-specific characteristics (like the number, name, position, and ordering of program sections) or searches the binary for compiler-specific byte sequences (like blocks of code or strings). For example, it is not uncommon to find version strings in binaries compiled using gcc, such as *GCC: (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0*.

When Ghidra has completed the loading process, it displays an Import Results Summary window, as shown in Figure 13-4.

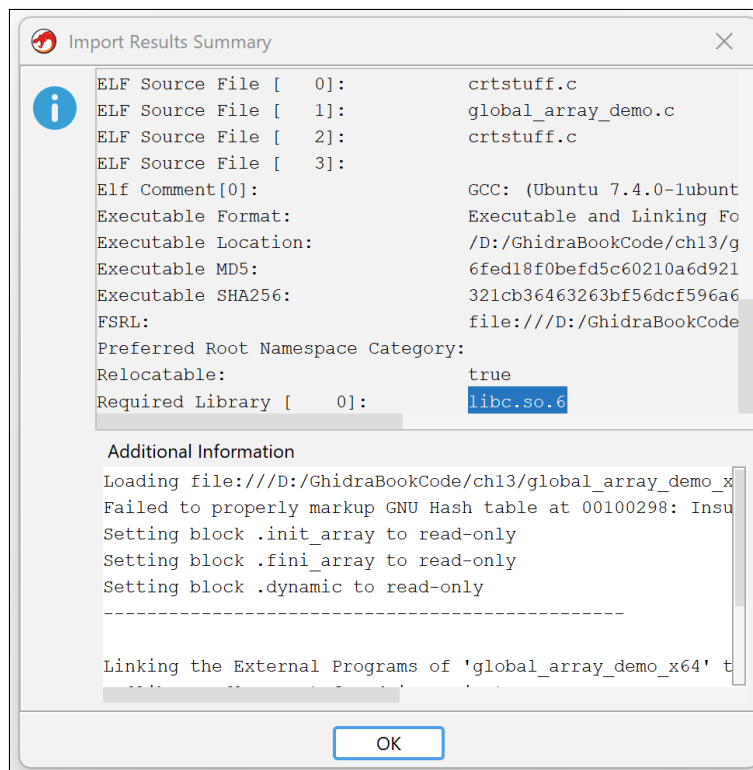


Figure 13-4: The Import Results Summary window for an ELF binary

In this summary, we highlighted an ELF Required Library, *lib.so.6*. (Note that this library would not be listed as a requirement if the file were statically linked.) More than one library file may be listed when an executable depends on multiple shared libraries. Understanding which libraries a program depends on can help direct you to resources you may need while analyzing the program. For example, if *libssl.so* or *libcrypto.so* appears in the list of required libraries, you might want to locate OpenSSL documentation and possibly source code. We discuss how Ghidra can make use of source code later in this chapter. Once Ghidra has successfully imported a file, you can auto analyze the file.

Analizers

Ghidra accomplishes auto analysis using a collection of cooperating analysis tools (analyzers) that are activated either manually (for example, when opening a new file) or automatically when a change that can affect the resulting disassembly is detected. Analyzers run sequentially in a prioritized order based on the type of analyzer because the changes an analyzer makes can affect subsequent analyzers. For example, the stack analyzers cannot look at functions until a function analyzer has looked at all calls and created the functions. We investigate this hierarchy in more detail in Chapter 15 when we build an analyzer.

When you open a new file in the CodeBrowser and choose to auto analyze it, Ghidra presents a list of analyzers it can run on that file. The list of default and optional analyzers depends on the file information provided by the loader (displayed to the user as part of the import summary, as shown in Figure 13-4). For example, the Windows x86 PE RTTI Analyzer would not be of much use in analyzing an ELF or ARM binary. You can modify the default analyzer selections as well as other parameters associated with a particular file using the Edit ► Options for menu.

Some analyzers are also available as one-shot options through the Analysis ► One Shot menu in the CodeBrowser. An analyzer appears in the list if it supports one-shot use and applies to the type of file being analyzed. One-shot analysis is useful for running analyzers not selected during the initial auto analysis, or for rerunning an analyzer after you have located new information that might benefit from additional analysis. For example, if you receive a missing PDB error message during initial analysis, you can locate the PDB file and then run one of the PDB analyzers.

The Analyze All Open option on the CodeBrowser ► Analysis menu analyzes all open files in the project at once, using the list of analyzers selected in Edit ► Options for menu. If all of the open files in the project have the same architecture (language/compiler specification), Ghidra will analyze all of the files. Otherwise, it will leave out any files that do not match the architecture of the current file. This ensures that the analyzers are consistent with the type of file being analyzed.

Many CodeBrowser tools, including analyzers, rely on various artifacts in order to identify important constructs in a file. Fortunately for us, we can extend many of these artifacts to improve Ghidra's capabilities. We will start with a discussion of word model files and how Ghidra uses them to identify special strings and types of strings within search results.

Word Models

A *word model* provides a way to identify special strings and types of strings you're interested in searching for, such as known identifiers, email addresses, directory pathnames, file extensions, and so on. When your string search is associated with a word model, the String Search results window will include a column called IsWord that specifies whether the found string is a word according to the word model. Defining strings of interest as valid words and then filtering for valid words is a good way to prioritize certain strings for further inspection.

At a high level, a word model uses training sets of valid strings to determine that "If trigram X (a sequence of three characters) appears in a sequence Y of length Z, then there is a probability, P, that Y is a word." The resulting probability is used indirectly as a threshold to determine if the string should be considered a valid word during analysis.

StringModel.sng, shown in Figure 13-5, is the default word model file for string searches in Ghidra.

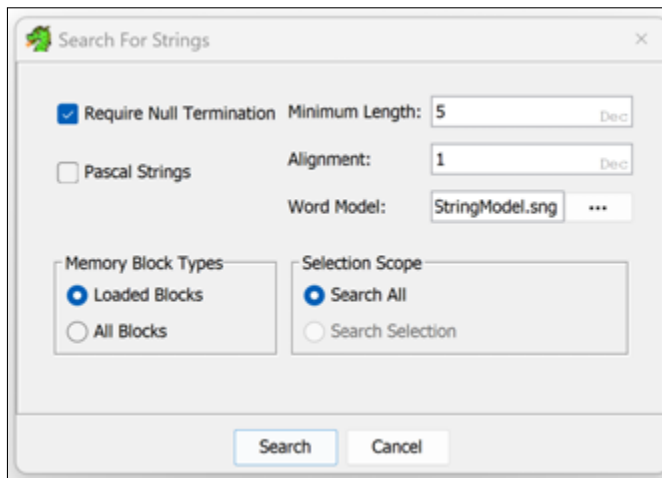


Figure 13-5: The Search For Strings dialog

The following excerpt from the *StringModel.sng* file shows the format of a valid word model file:

```

❶ # Model Type: lowercase
❷ # Training file: contractions.txt
  # Training file: uniqueStrings_012615_minLen8.edited.txt
  # Training file: connectives
  # Training file: propernames
  # Training file: web2
  # Training file: web2a
  # Training file: words
❸ # [^] denotes beginning of string
  # [$] denotes end of string
  # [SP] denotes space
  # [HT] denotes horizontal tab
  # Thresholds: -2.71, -3.26,
  -----CONTENT OMITTED HERE-----
  # Symbol Size: 128

❹ [HT]    [HT]    [HT]    17
    [HT]    [HT]    [SP]    8
    [HT]    [HT]    (      1
    [HT]    [HT]    ;      1
    [HT]    [HT]    \     25
    [HT]    [HT]    a      2
    [HT]    [HT]    b      1
    [HT]    [HT]    c      1

```

The lines that begin with # are metadata comments about the model. In this example, the model type ❶ is lowercase, which likely means the model does not distinguish between upper- and lowercase letters. The file lists the names of the training files used for this model ❷. The names generally indicate the content: *contractions.txt* is likely a file of valid contractions, like *can't*. Four lines ❸ describe the notation for some nonprinting ASCII characters used in the trigrams. The actual trigram list starts ❹ where each entry row contains the three characters in the trigram followed by a value used when determining the probability that the trigram is part of a word.

You can supplement or replace the default word model by editing *StringModel.sng* or creating new model files and storing them in *Ghidra/Features/Base/data/stringngrams* and then selecting the new file in the Word Model field in the Search for Strings dialog. There are many reasons to modify word models, like including strings specific to known malware families or detecting words in languages other than English. Ultimately, word models provide a powerful means to control the types of strings that Ghidra recognizes as higher priority by tagging them in the Strings window.

In a similar manner, we can edit and extend the data types that Ghidra recognizes.

Data Types

The Data Type Manager allows us to manage all of the data types associated with a file. Ghidra lets you reuse data type definitions by storing them in *data type archive files*. Each root node in the Data Type Manager window is a data type archive. Figure 13-6 shows a Data Type Manager window with three data type archives selected by the Ghidra loader.

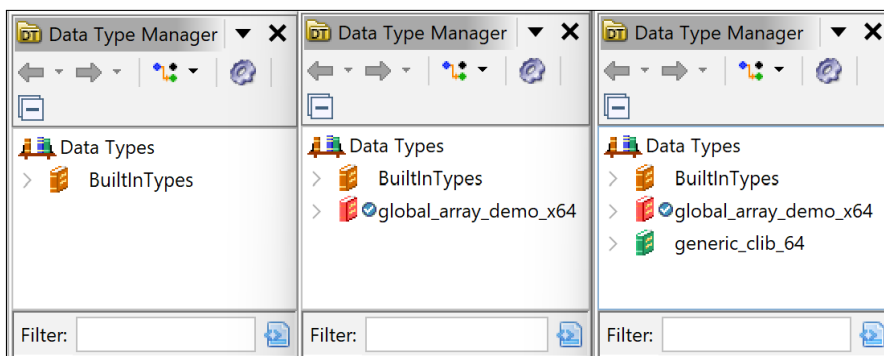


Figure 13-6: The Data Type Manager window

The `BuiltInTypes` archive is always listed. This archive includes all (and only) types that are modeled within Ghidra by Java classes that implement the `ghidra.program.model.data.BuiltInDataType` interface. Ghidra searches for every such class within its classpath in order to populate this archive.

The second archive is specific to the file that is being analyzed, and the archive shares the file's name. In this case, the archive is associated with the file *global_array_demo_x64*. The checkmark next to the archive indicates that it is associated with the active file. Initially, Ghidra populates this archive with data types specific to the file's format (for example, PE- or ELF-related data types). During auto analysis, Ghidra copies additional types, from the other archives, into this one when they are recognized to be in use in the program. In other words, this archive contains the subset of all data types known to the Data Type Manager that happen to be in use in the current program. This archive is also the home to any custom data types that you choose to create in Ghidra, as discussed in “Creating Structures with Ghidra” on page 166.

The third archive provides the 64-bit ANSI C function prototypes and C library data types. This particular archive contains information extracted from the standard C library headers of a 64-bit Linux system and is one of several platform-specific archives in a default Ghidra installation. It is present because this particular binary has a library dependency on *libc.so.6*, as indicated in Figure 13-4. A default Ghidra installation has several additional platform-specific data archives, located in the *Ghidra/Features/Base/data/typeinfo* directory under a subdirectory specific to the platform. The filenames indicate the platforms they support: *generic_clib.gdt*, *generic_clib_64.gdt*, a number of

golang related archives, *mac_osx.gdt*, *rust-common.gdt*, *windows_vs12_32.gdt*, and *windows_vs12_64.gdt*. (The *.gdt* extension is used for all Ghidra data type archives.)

Loading Additional Data Type Archives

In addition to the archives that the Ghidra loader selects automatically, you can add your own data type archives as nodes in the Data Type Manager window. For demonstration purposes, Figure 13-7 shows the Data Type Manager window after all of the default *.gdt* files have been added to the Data Types list. The right side of the figure shows the menu for manipulating archives and data types. You can load additional archives using the Open File Archive menu option, which opens a file browser from which you can select an archive of interest.

To add new built-in types to the BuiltInTypes archive, add corresponding *.class* files to Ghidra's classpath. If you add types while Ghidra is running, an option to refresh BuiltInTypes will be added to the options menu on the right side of Figure 13-7. This refresh is necessary in order for the new types to appear in the Data Type Manager window.

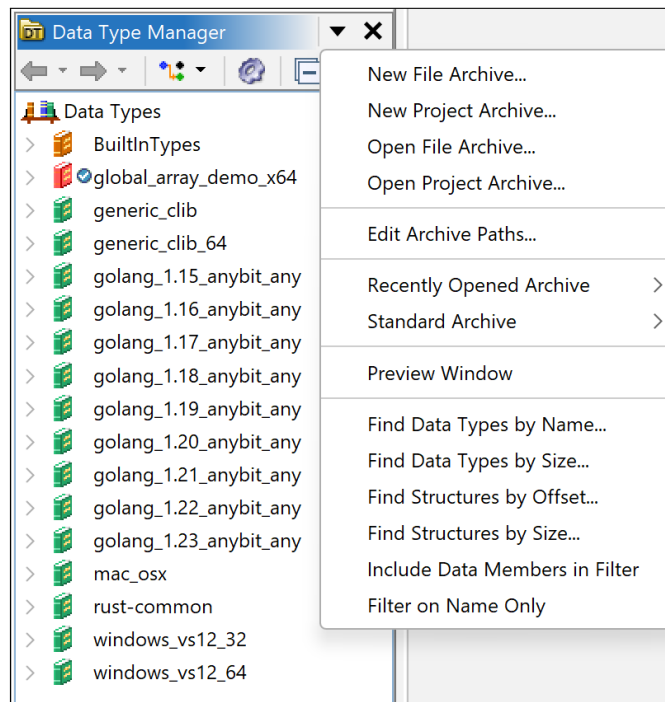


Figure 13-7: The Data Type Manager with all standard archives loaded and the options menu expanded

The refresh operation causes Ghidra to rescan its classpath to find any newly added `BuiltInDataType` classes. The inquisitive reader may find numerous examples of built-in types in their Ghidra source distribution at *Ghidra/Framework/SoftwareModeling/src/main/java/ghidra/program/model/data*.

Creating New Data Type Archives

It's impossible to anticipate every data type that you may encounter while analyzing binaries. The archives included in your Ghidra distribution consist of data types culled from frequently used libraries on common platforms. When Ghidra doesn't contain information on the data types used in a program you're analyzing, it offers you the ability to create new data type archives, populate them in a variety of ways, and share them with others. In the following sections, we discuss the three ways you are likely to create new data type archives.

Parsing C Header Files

One of the most common sources of data type information is C header files. Assuming you have the header files you need, or take the time to create them yourself, you can create your own data type archive by using the C-Parser plugin to extract the information from an existing C header file. For example, if you frequently find yourself analyzing binaries that link against the OpenSSL cryptographic library, you might download the OpenSSL source code and ask Ghidra to parse the included header files to create an archive of OpenSSL data types and function signatures.

This process is not nearly as straightforward as it might seem. Header files are often littered with macros designed to influence the behavior of a compiler based on the compiler being used and the operating system and architecture being targeted. For example, the C structure

```
struct parse_demo {  
    uint32_t int_member;  
    char    *ptr_member;  
};
```

occupies 8 bytes when compiled on a 32-bit system and 16 bytes when compiled on a 64-bit system. This variability poses a problem for Ghidra, which is attempting to act as the universal preprocessor, and it is up to you to guide Ghidra through the parsing process to create a useful archive. When the time comes to use your archive with Ghidra, you must have ensured that the archive was created in a manner compatible with the binary you are analyzing. (That is, don't load 64-bit archives to help you analyze a 32-bit file.)

To parse one or more C header files, select **File ► Parse C Source** in the CodeBrowser to open the dialog shown in Figure 13-8. The Source files to parse section provides an ordered list of header files for the plugin to parse. The order is important, as the data types and preprocessor directives from one file become available for the next file.

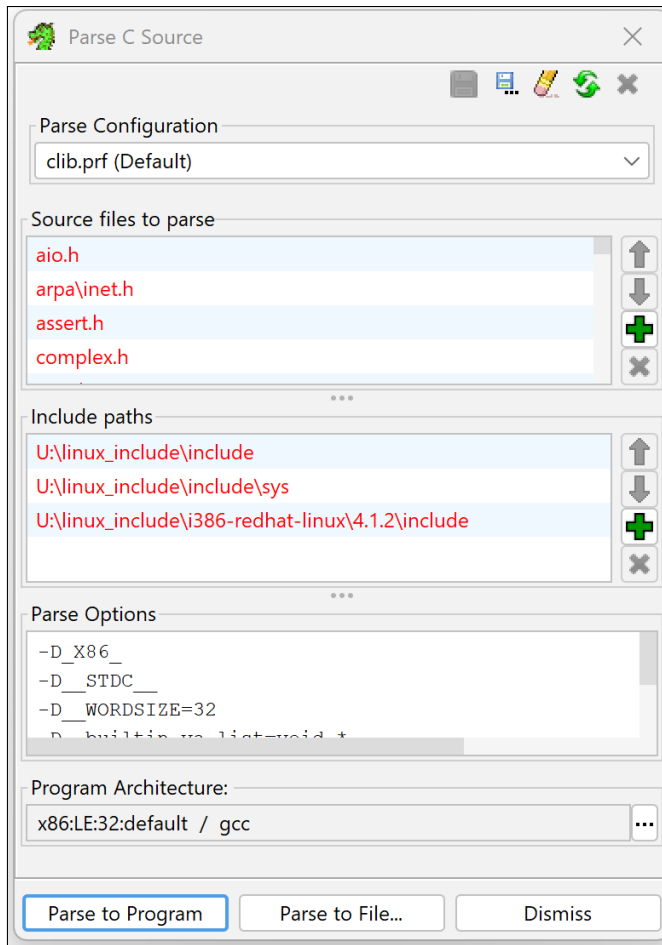


Figure 13-8: The Parse C Source dialog

The Parse Options box provides a list of options, similar to compiler command line options, that influence the behavior of the C-Parser plugin. The parser recognizes only the `-I` (include directory) and `-D` (define a macro) options understood by most compilers. Ghidra offers a number of preprocessor configurations, in the form of `.prf` files, that you can choose from to provide reasonable defaults for common operating system and compiler combinations. You can also customize any of the available configurations or create your own from scratch and save them to your own `.prf` for future use. A common change to the parser options is to correctly set the architecture that you want the C-Parser to target, as all of the supplied configurations target x86. For example, you might change `-D __X86__` in a Linux-oriented configuration to `-D __ARMEL__` if you are analyzing little-endian ARM binaries.

The plugin's output can be merged into the current active file with the Parse to Program button or stored in a separate Ghidra data type archive file (`.gdt`) with Parse to File. You can find additional information about the C-Parser in Ghidra Help.

Creating a New File Archive

As an alternative to parsing C headers files, you might want to capture custom data types that you create while analyzing a file by gathering them into an archive that you can share with other Ghidra users or use in other Ghidra projects. The Data Type Manager's New File Archive option (refer to Figure 13-7) asks you to select a filename and save location, and then creates a new, empty archive that is listed in the Data Type Manager window. You can add new types to the archive by using the techniques described in "Creating Structures with Ghidra" on page 166. Once your archive is created, you may share it with other Ghidra users or use it in your other Ghidra projects.

Creating a New Project Archive

A project data archive exists only within the project in which it was created. This may be useful if you expect to reuse custom data types in more than one file within a project but never expect to use the data types outside your project. Within the Data Type Manager, the New Project Archive option (refer to Figure 13-7) asks you to select a folder within your project to hold your new archive, and then creates a new, empty archive listed in the Data Type Manager window. As with the other data type archives, you can add new types to the archive as needed.

Function IDs

When you set out to reverse engineer any binary, the last thing you want to do is waste time reverse engineering library functions whose behavior you could learn much more easily by simply reading a man page or some source code, or by doing a little internet research.

Unfortunately, statically linked binaries blur the distinction between application code and library code: they combine entire libraries with application code to form a single, monolithic executable file. Fortunately for us, Ghidra has tools to recognize and mark library code, regardless of whether the code was taken from a library archive or is simply the result of code reuse across multiple binaries, allowing us to focus our attention on the unique code within the application. The *Function ID analyzer* recognizes many common library functions using function signatures included with Ghidra, and you can extend the function signature databases by using the Function ID plugin.

The Function ID analyzer works with Function ID databases (FidDBs) that use a hierarchy of hash values to characterize functions. A full hash (intended to be resilient against changes that might be introduced by the linker) and a specific hash (which helps differentiate between variants of functions) are computed for each function. The major difference between the two is that the specific hash may include the specific values of any constant operands (based on a heuristic), whereas the full hash does not. The combination of the two hashes coupled with information about any associated parent and child functions forms a fingerprint for each library function, which is recorded in an FidDb.

The Function ID analyzer derives the same type of fingerprint for each function in the binary you are analyzing and compares it against all known fingerprints in relevant FidDb's. When it finds a match, Ghidra recovers the function's original name from the FidDb, applies the appropriate label to the function under analysis, adds the function to the Symbol Tree window, and updates the function's plate comment. The following is a sample plate comment for the `_malloc` function:

```
*****
* Library Function - SingleMatch                                *
* Name: _malloc                                                  *
* Library: Visual Studio 2005 Release                            *
*****
```

Information about functions in an FidDb are stored hierarchically and include a name, version, and variant. The variant field is used to encode information such as compiler settings, which affect the hashes but aren't part of the version number.

The Function ID analyzer offers several options, accessible when you select the analyzer in the Auto Analysis dialog, to control its behavior, as shown in Figure 13-9. The instruction count threshold is a tunable threshold designed to reduce false positives from random matches against small functions. False positives occur when a function is incorrectly matched to a library function. False negatives occur when a function is not matched to a library function but should be. The threshold roughly represents the minimum number of instructions that a function, its parents, and its children must contain (combined) in order to be considered for a match. Refer to *Scoring and Disambiguation* in Ghidra Help for more information on match scores.

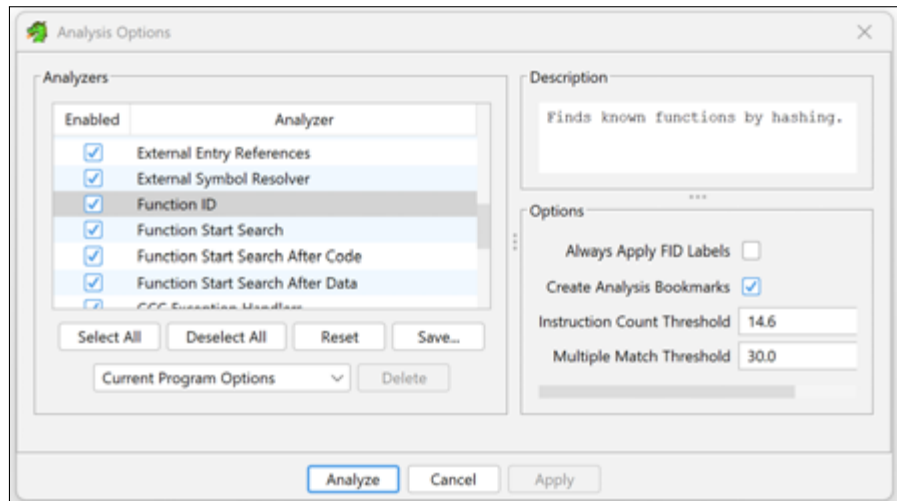


Figure 13-9: Auto analysis options

Since a binary's actual functionality is generally contained in functions, it's important to be able to extend function signatures so you can minimize the duplication of your efforts, a task facilitated by the Function ID plugin.

The Function ID Plugin

The *Function ID tool* (not to be confused with the Function ID analyzer) allows you to create, modify, and specify associations for FidDBs. You can control this tool via the CodeBrowser's Tools ► Function ID menu, as shown in Figure 13-10.

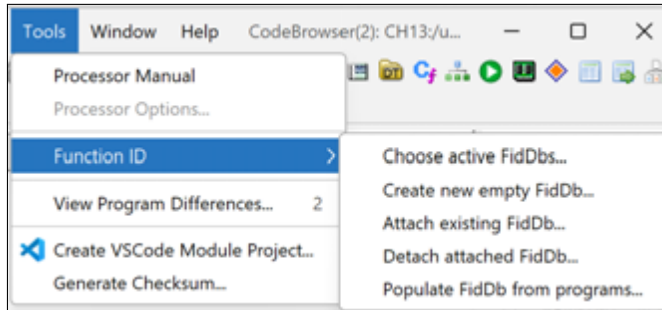


Figure 13-10: The CodeBrowser Function ID submenu

Before we walk through an example of using the Function ID plugin to extend Ghidra signatures, let's briefly discuss the five new menu options:

Choose active FidDBs Displays a list of active Function ID databases. Each may be selected or deselected using an associated checkbox.

Create new empty FidDb Allows you to create and name a new Function ID database. The new FidDb will be listed when you select Choose active FidDBs.

Attach existing FidDb Displays a file chooser dialog that lets you add an existing FidDb to the list of active FidDBs. After you add the FidDb, you can select Choose active FidDBs to see the added FidDb listed.

Detach attached FidDb Can be applied to only FidDBs that have been manually attached. The operation removes the association between the selected FidDb and the current Ghidra instance.

Populate FidDb from programs Generates new function fingerprints to add to an existing FidDb.

Example: Identifying UPX Functions

When we auto analyze binaries that contain very few functions outside of library functions that Ghidra recognizes, our reverse engineering task is somewhat simplified. We can focus on the subset of functions that Ghidra fails to recognize under the assumption that this is where the new, interesting functionality lies.

Our task is much more challenging when Ghidra can't identify any functions. When we (human analysts) recognize functions and extend Ghidra's ability to recognize those same functions in the future, we reduce our future workload. Let's walk through a demonstration of how powerful this sort of extension can be.

Let's assume we load a 64-bit Linux ELF binary into Ghidra and auto analyze the file. Figure 13-11 shows the resulting Symbol Tree entries. We use the Symbol Tree to navigate to the entry point and examine the code. Our initial analysis leads us to believe that the binary is packed using the *Ultimate Packer for eXecutables (UPX)* and that the functions we are seeing were added by the UPX packer to unpack the binary at runtime.

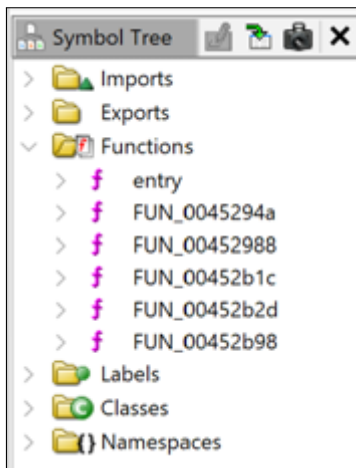


Figure 13-11: Suspected UPX packer functions for `upx_demo1_x64_static.upx`

We confirm this hypothesis by comparing the bytes we see in entry with published bytes for the UPX entry point function. (Alternatively, we could create our own UPX-packed binary for comparison.) Now, let's add this information to our FidDb so that we don't have to perform this same analysis if we encounter another UPX-packed 64-bit Linux binary in the future.

Functions you add to an FidDb should have meaningful names. Accordingly, we change the names of the functions in our example to indicate that they are part of a UPX packer, as shown in Figure 13-12, and then add these functions to a new Function ID database so that Ghidra can label the functions appropriately in the future.

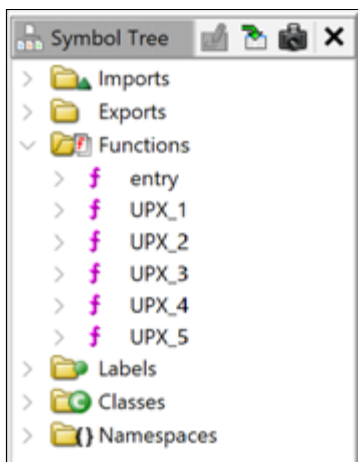


Figure 13-12: Labeled UPX packer functions for `upx_demo1_x64_static.upx`

We create a new FidDb by selecting **Tools ► Function ID ► Create new empty FidDb** and then name the new FidDb *UPX.fidbf*. Next, we populate our new database with information extracted from the updated binary by selecting **Tools ► Function ID ► Populate FidDb from programs**. Enter information about the FidDb in the resulting dialog, as shown in Figure 13-13.

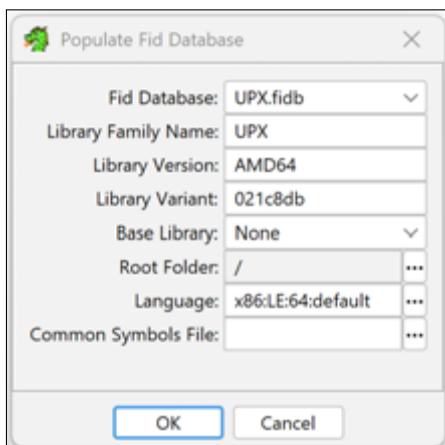


Figure 13-13: The Populate Fid Database dialog

Here is the purpose of each field and the values we have entered into them:

Fid Database *UPX.fidbf* is the name of our new FidDb. The pull-down list allows you to choose from among all of the FidDbs you have created.

Library Family Name We choose a name that describes the library from which we are extracting function data. In our case, we have input **UPX**.

Library Version This can be a version number, a platform name, or a combination of both. Since UPX is available for many platforms, we chose the library version based on the architecture of the binary.

Library Variant This field may be used for any additional information that distinguishes this library from others of the same version. In this example, we used the commit ID for this version of UPX from the UPX repository on GitHub (<https://github.com/upx/upx>).

Base Library Here, you may reference another FidDb that Ghidra will use to establish parent/child relationships. We did not use a base library, as UPX is completely self contained.

Root Folder This field names a Ghidra project folder. All files in the chosen folder will be processed during the function ingest process. In this case, we chose the project root folder.

Language This field contains the Ghidra language identifier associated with the new FidDb. To be processed from the root folder, a file's language identifier must match this value. This entry can be populated from the Imports Results Summary window for the binary, but may be modified using the button to the right of the text box.

Common Symbols File This field names a file containing a list of functions that should be excluded from the ingest process. This field is unused in this case.

When we click OK, the ingest process begins. When it's complete, we see the results of the FidDb population (Figure 13-14).

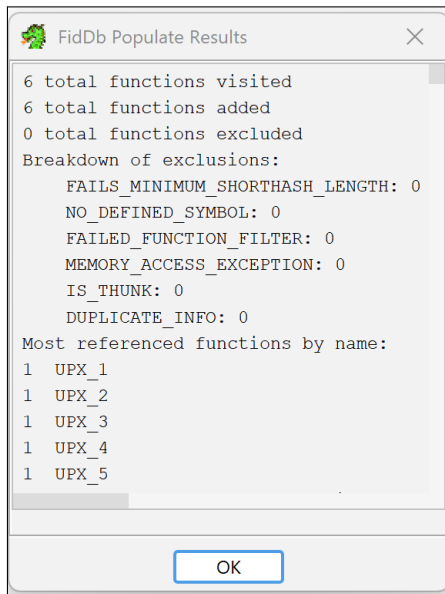


Figure 13-14: The results window following the UPX FidDb population

Once the new FidDb has been created, Ghidra can use it to identify functions in any binary you are analyzing. We demonstrate this by loading a new UPX-packed 64-bit Linux ELF binary, *upx_demo2_x64_static.upx*, and auto analyzing the file *without* the Function ID analyzer. The resulting Symbol Tree, shown in Figure 13-15, shows five unidentified functions, as we expect.

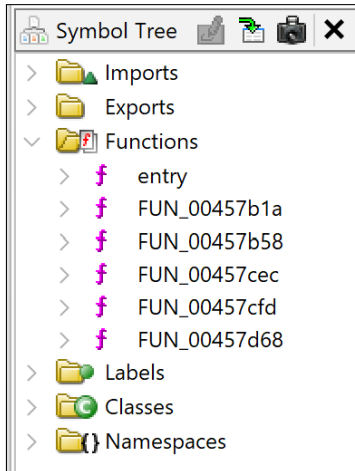


Figure 13-15: The Symbol Tree entry for *upx_demo2_x64_static.upx* before running the Function ID analyzer

Running Function ID as a one-shot analyzer (Analysis ► One Shot ► Function ID) results in the Symbol Tree shown in Figure 13-16, which includes the UPX function names.

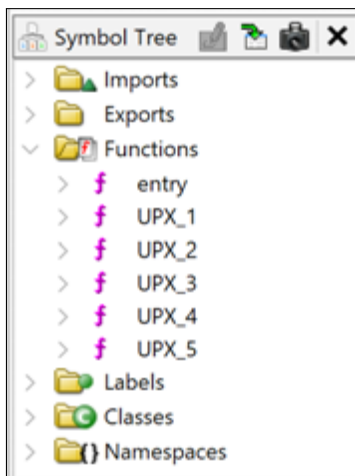


Figure 13-16: The Symbol Tree entry for *upx_demo2_x64_static.upx* after running the Function ID analyzer

The analyzer also updates the Listing window with new function names and plate comments, like the plate comment for UPX_1 shown next. This plate comment contains the information we provided when creating the FidDb:

```

*****
* Library Function - Single Match                                *
* UPX_1                                                          *
* Library: UPX AMD64 021c8db                                     *
*****

                                undefined UPX_1()
undefined      <UNASSIGNED>    <RETURN>
    UPX_1
00457b1a 48 8d 04 2f LEA      RAX,[RDI + RBP*0x1]
00457b1e 83 f9 05    CMP      ECX,0x5

```

Creating new FidDb's is only the beginning of extending Ghidra's function identification capabilities. You can also analyze parameters associated with a function and save them in a Data Type archive. Then, when Function ID correctly identifies the function, you can drag the appropriate Data Type Manager entry onto the function in the Listing window, and the function prototype will be updated with the appropriate parameters.

Example: Profiling a Static Library

When you are reverse engineering a statically linked binary, one of the first things you may wish for is an FidDb that matches the functions linked into that binary, so that Ghidra can identify the library code and save you the effort of analyzing it. The following example addresses two important questions: (1) How can you know whether you have such an FidDb, and (2) what can you do if you don't have one?

The answer to the first question is simple: Ghidra ships with 10 FidDb's (in the form of *.fidbf* files), all related to Visual Studio library code. If the binary is not a Windows binary and you have not yet created or imported any FidDb's, you'll need to make your own FidDb by using the Ghidra Function ID plugin (which addresses the second question).

The most important thing to understand when populating a new FidDb is that you need an input source that has a high probability of matching against any binaries you plan to apply the FidDb against. In the UPX example, we had a binary that contained code that our intuition told us we might see again in the future. In a common static linking case, we may simply want to match as much code in the target binary as possible.

There are a variety of ways to recognize that you're dealing with a statically linked binary. Within Ghidra, look at the *Imports* folder within the Symbol Tree. This folder will be empty for a fully statically linked binary with no need for imported functions. A partially statically linked binary may have some imports, so you can look for copyright or version strings from well-known libraries in the Defined Strings window.

On the command line, you can use simple utilities like `file` and `strings`:

```
$ file upx_demo2_x64_static_stripped
upx_demo2_x64_static_stripped: ELF 64-bit LSB executable, x86-64,
version 1 (GNU/Linux), statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=54e3569c298166521438938cc2b7a4dda7ab7f5c, stripped
$ strings upx_demo2_x64_static_stripped | grep GCC
GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
```

The output of `file` informs us that the binary is statically linked, stripped of any symbols, and from a Linux system. (A stripped binary contains no familiar names to clue us in to the behavior of any of the functions.) Filtering the output of `strings` using `grep GCC` identifies the compiler, GCC 7.4.0, as well as the Linux distribution, Ubuntu 18.04.1, used to build the binary. (You can locate the same information with CodeBrowser's Search ► Program Text functionality using `GCC` as a qualifier.)

It's likely this binary was linked with *libc.a*, an archive of C standard library functions used in statically linked binaries on Unix-style systems. In this case, a good starting point for symbol recovery is to use a copy of *libc.a* from Ubuntu 18.04.1. (Additional strings in the binary might lead us to select additional static libraries for the Function ID analysis; however, we limit this example to *libc.a*.)

To use *libc.a* to populate an `FidDb`, Ghidra must identify the instructions and functions that it contains. The archive (*.a*) file format defines a container for other files, most commonly for object files (*.o*) that a compiler might extract and link into an executable. Ghidra's process for importing container files differs from its process for importing single binaries, so when we import *libc.a* with File ► Import, as we usually do when importing a single file, Ghidra offers alternate import modes, as shown in Figure 13-17. (These other options are also available from the File menu.)

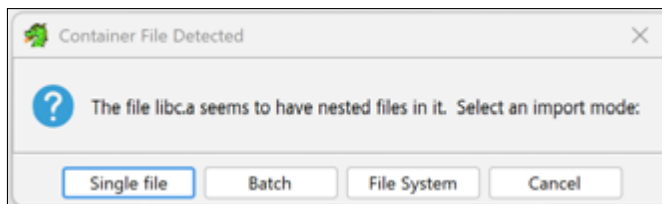


Figure 13-17: Importing a container file

Single File mode asks Ghidra to import the container as if it were a single file. Since the container is not an executable file, Ghidra is likely to suggest the Raw Binary format for your import and perform minimal automated analysis. In File System mode, Ghidra opens a file browser window (see Figure 13-18) to display the contents of the container file. In this mode,

you may choose any combination of files from the container to import into Ghidra using options from context menus.

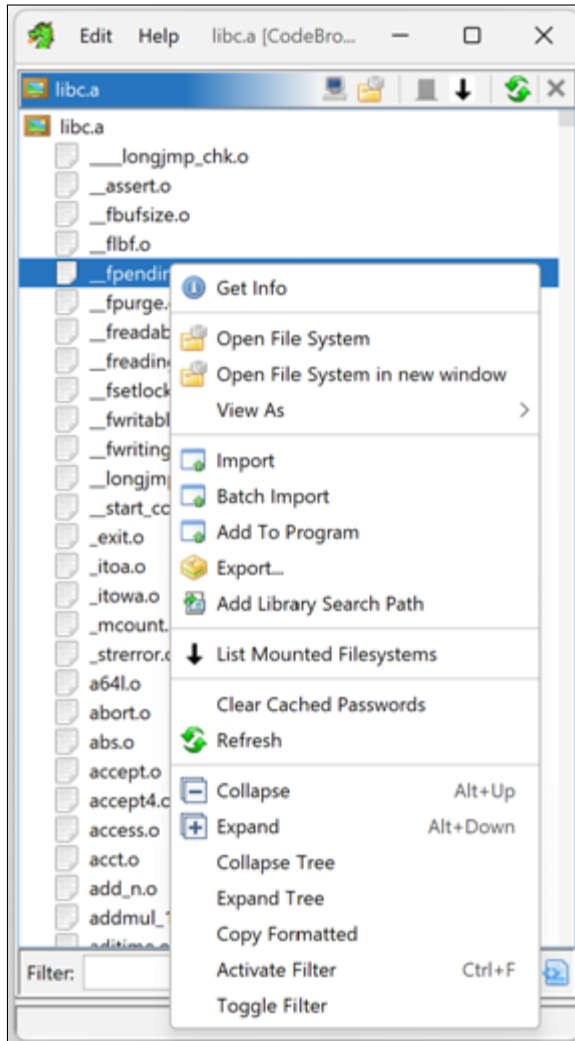


Figure 13-18: File System import mode

In Batch mode, Ghidra automatically imports all files in the container without pausing to display individual file information. After initially processing the container's contents, Ghidra displays the Batch Import dialog shown in Figure 13-19. Before clicking OK, you can view information on each file being imported, add more files to the batch import, set import options, and choose the destination folder within your Ghidra project. Figure 13-19 shows that we are about to import 1690 files from the *libc.a* archive into our CH13 project's root directory.

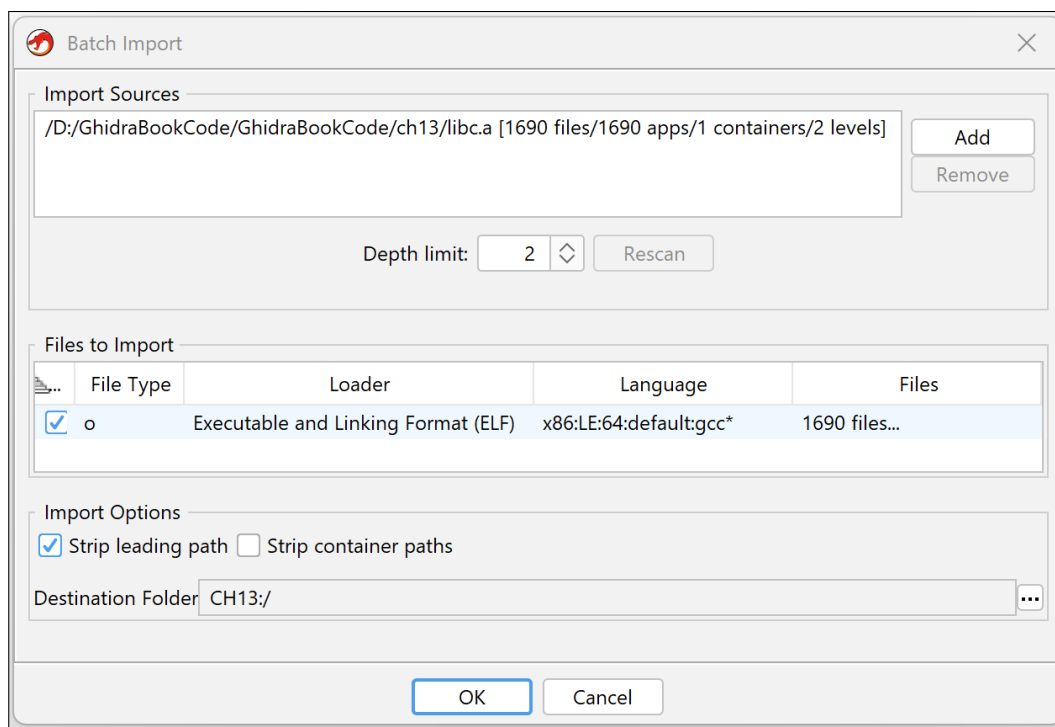


Figure 13-19: The Batch Import dialog

Click **OK** to kick off the import process (which may take some time). Once the import is complete, you will be able to browse the newly imported files in the Ghidra Project window. Because *libc.a* is a container file, it will appear as a folder in the Project window, and you can navigate its contents to open and analyze any one of the files contained in the folder.

At this point, we can finally capture fingerprints of each *libc* function into an *FidDb* and use that *FidDb* to perform Function ID analysis against our sample statically linked binary. This process parallels the UPX example, beginning with creating a new empty *FidDb* that will then be populated from programs. The programs in this case will be the entire contents of our newly imported *libc.a* folder.

Here, we run into a significant challenge. When we select the files to populate our new *FidDb*, we must ensure that Ghidra has properly analyzed every file to identify functions and their associated instructions (the input to the Function ID hashing process). Up to this point, we have seen Ghidra analyze programs only when we open them in the CodeBrowser, but with *libc.a*, we are faced with the daunting task of analyzing 1,690 individual files within the *libc.a* archive.

Opening and analyzing the files one at a time is not a good use of our time. Even selecting to open all files on import and using Ghidra's Analyze All Open option will still take us a while to work through all 1,690 files (and will likely require manual intervention to adjust our tool options and resource allocations to accommodate a task of this size within our Ghidra instance).

If this problem seems unwieldy, you are correct. This is not the sort of task that we should be solving manually using the Ghidra GUI, but a well-defined repetitive task that shouldn't require human intervention. Fortunately for us, the next three chapters introduce methods we can use to automate this and other tasks. When we get to "Example 2: Automated Function ID Database Creation" on page 383, we will revisit this specific task and demonstrate how easily batch processing can be accomplished using Ghidra's headless mode of operation.

Regardless of the method we use to process *libc.a*, once complete, it's a simple matter to return to the Function ID plugin and populate our new FidDb, yielding the following results:

FidDb Populate Results

```
2905 total functions visited
2638 total functions added
267 total functions excluded
Breakdown of exclusions:    FAILS_MINIMUM_SHORTHASH_LENGTH: 234
    DUPLICATE_INFO: 9
    FAILED_FUNCTION_FILTER: 0
    IS_THUNK: 16
    NO_DEFINED_SYMBOL: 8
    MEMORY_ACCESS_EXCEPTION: 0
Most referenced functions by name:
749  __stack_chk_fail
431  free
304  malloc
...
```

Our new FidDb is now available for use and allows the Function ID analyzer to match many of the functions contained in *upx_demo2_x64_static_stripped*, significantly reducing our reverse engineering workload for this particular binary.

Summary

This chapter demonstrated some of the ways that we can extend Ghidra, including by parsing C source files, extending word models, and extracting function fingerprints using the Function ID plugin. When a binary contains statically linked code or code reused from previously analyzed binaries, matching those functions against Ghidra FidDb's can save you the hassle of manually wading through a mountain of code. Predictably, so many static link libraries exist that it is not possible for Ghidra to include FidDb files that cover every possible use case. The ability to create your own FidDb files when necessary allows you to build up a collection of FidDb's tuned to your particular needs. In Chapters 14 and 15, we introduce Ghidra's powerful scripting capabilities to further extend Ghidra's functionality.

14

BASIC SCRIPTING WITH GHIDRA AND PYGHIDRA



No application can meet every need of every user. It is simply not possible to anticipate every potential use case that may arise.

Ghidra's open source model lets developers request and contribute features, but sometimes, you need to immediately address a problem at hand and cannot wait for someone else to implement new functionality. For those cases, Ghidra includes integrated scripting features.

Ghidra scripts can range from simple one-liners to full-featured programs that automate common tasks or perform complex analysis. In this chapter, we focus on the basic scripting capabilities available through the CodeBrowser interface. We introduce Ghidra's internal scripting environment, demonstrate how to write and run scripts using both Java and Python, and introduce PyGhidra, which provides a more seamless way to interact with Ghidra from standard Python environments. We will then move on to other integrated scripting options in Chapter 15.

The Script Manager

The Ghidra Script Manager is available through the CodeBrowser menu. Choosing Window ► Script Manager opens the window shown in Figure 14-1.

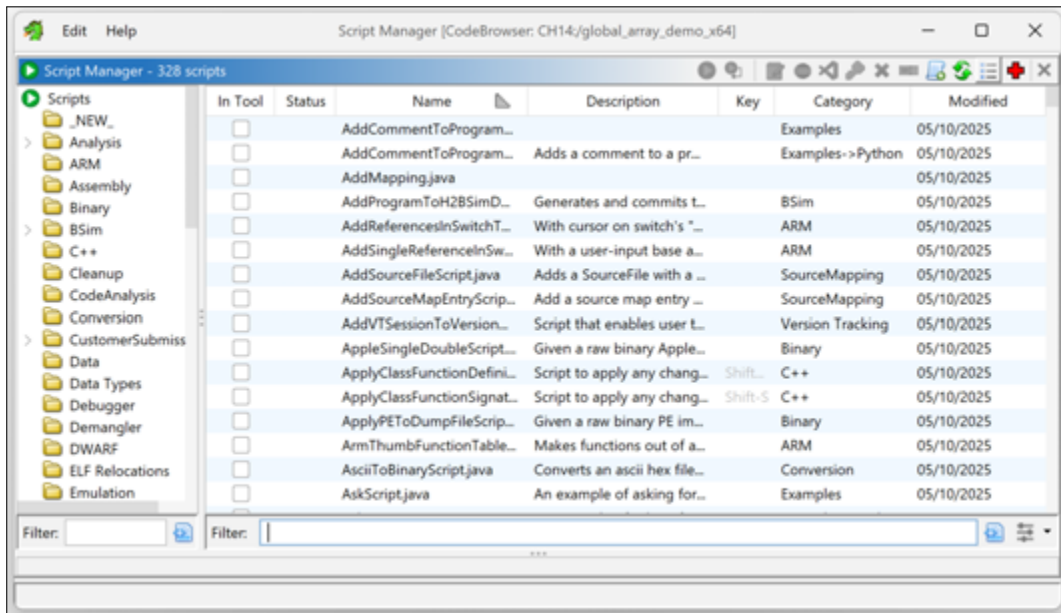


Figure 14-1: The Script Manager window

You can also open the window using the Script Manager icon in the CodeBrowser toolbar (a green circle with a right-facing arrow inside, similar to a “Play” button, also shown in the top left of the Script Manager window).

The Script Manager Window

In a new Ghidra installation, the Script Manager loads with over 325 scripts organized in a category tree, as seen on the left side of Figure 14-1. Some of the folders contain subfolders to provide an even more detailed classification of the scripts. You can expand and collapse the folders to see the organization of the scripts. Selecting an individual folder or subfolder limits the display to the scripts within the selected folder. To populate this window, Ghidra locates and indexes all scripts in subdirectories named *ghidra_scripts* within the Ghidra distribution folder. Ghidra also looks for a *ghidra_scripts* directory within your home directory and indexes any scripts it finds there.

The default set of scripts covers a wide range of functionality. Some are intended to demonstrate fundamental scripting concepts. The columns in the script list table provide additional detail about the purpose of each script. As with most Ghidra tables, you can control which columns are displayed, as well as the sort order for individual columns. By default, it displays all available fields, except Created and Path. The seven default columns provide the following insight into a script:

In Tool For scripts that advertise `@keybinding` or `@menupath` metadata, this field indicates whether Ghidra should activate the indicated key binding and/or display the indicated menu item to allow easy user activation of the script outside of the Script Manager. We discuss metadata fields in more depth in “Scripting in Java” on page 307.

Status Indicates the state of the script. The field is generally blank, but can contain a red icon to indicate an error in the script. If you have associated a toolbar icon with the script, the icon will appear in this column.

Name Contains the filename of the script, including its extension.

Description Is a description pulled from the metadata comment within the script. This field can be quite lengthy, but you can read the entire contents by hovering over the field. We discuss this field in more depth in “Script Development” on page 307.

Key Indicates if there is a key binding assigned that provides a shortcut for running the script.

Category Specifies the path at which the script will be listed in the Script Manager’s topic hierarchy. This is a logical hierarchy, *not* a filesystem directory hierarchy.

Modified The date the script was last saved. For the default scripts this is the installation date of the Ghidra instance.

The filter field on the left side of the window searches through the script categories. The filter on the right side searches script names and descriptions. Finally, at the bottom, an additional window is initially empty. This window displays metadata about a selected script in an easy-to-process format that includes the field extracted from the metadata within the script. We discuss the format and meaning of the metadata fields in “Scripting in Java” on page 307.

While the Script Manager provides a significant amount of information, the main power of this window comes from the toolbar it provides.

The Script Manager Toolbar

The Script Manager does not have menus to help you manage your scripts. Instead, all script management actions are associated with tools on the Script Manager toolbar (Figure 14-2).

	Run Script	Run a selected script. The script will be recompiled if necessary and display the error icon in the "status" field and an error message in the console if the script does not compile.
	Rerun Last Script	Runs the last script run. This option is also available in the CodeBrowser window after a script is run.
	Edit Script	Opens the selected script in a window for editing.
	Edit Script with Eclipse	Opens a script in Eclipse for editing. (See Chapter 15.)
	Edit Script with VSC	Opens a script in Visual Studio Code for editing. (See Chapter 15.)
	Assign Key Binding	Allows assignment of a shortcut key combination for the script.
	Delete Script	Permanently deletes a user-created script. System-provided scripts cannot be deleted using this method.
	Rename Script	Allows you to rename a user-created script. System-provided scripts cannot be renamed using this method.
	Create New Script	Opens an empty script template in a basic editor window.
	Refresh Script List	Rescans the script directories and regenerates the list of scripts.
	Script Directories	Provides a list of script directories that you can select/deselect. You also have the option to add new directories to the list.
	Help	Opens the Javadoc for the GhidraScript class. The first time this is selected, Ghidra automatically builds the Javadoc content.

Figure 14-2: The Script Manager toolbar

While most of the menu options should be pretty clear from the descriptions in Figure 14-2, the Edit options merit additional discussion. We cover editing scripts with Eclipse and Visual Studio Code in Chapter 15, as they facilitate more advanced scripting capabilities.

The Edit Script option opens a primitive text editor window with its own toolbar, shown in Figure 14-3. The associated actions provide the basic functionality for editing files.

	Refresh	Refreshes the script contents. This is helpful if you are using an external editor.
	Save	Saves the latest changes back to the script file. You cannot save changes to system scripts.
	Save As	Saves the script to a new file. This option can be used to edit system scripts by creating new versions of the scripts.
	Undo/Redo	Allows you to back out of modifications and reapply if desired. (Up to 50 actions are retained for this functionality.)
	Run Script	Runs the script. This option will recompile if necessary and display an error message in the console if the script does not compile.
	Select Font	Allows you to specify the font type, size, and style for the editor.

Figure 14-3: The Edit Script toolbar

With an editor in hand, we can get down to the business of writing actual scripts.

Script Development

There are several methods of developing scripts in Ghidra. In this section, we focus on scripting using Java and Python, languages used by the existing scripts in the Script Manager window. Most of the 325-plus system scripts are written in Java, so we will begin by editing and developing scripts in Java.

Scripting in Java

In Ghidra, scripts written in Java are complete class specifications designed to be seamlessly compiled, dynamically loaded into your running Ghidra instance, invoked, and finally unloaded. The class must extend the class `Ghidra.app.script.GhidraScript`, implement a `run()` method, and be annotated with comments that provide Javadoc-like metadata about the script. We will show the structure of a script file, describe the metadata requirements, look at some of the system scripts, and then move on to editing existing scripts and building our own scripts.

When you choose the Create New Script Option, you will be prompted for the type of script you wish to create. The options include Java and two Python alternatives: Jython (which only supports Python 2) and PyGhidra

(which supports Python 3). Figure 14-4 shows the script editor opened when we create a new Java script. We have named the new script *CH14_NewScript*.

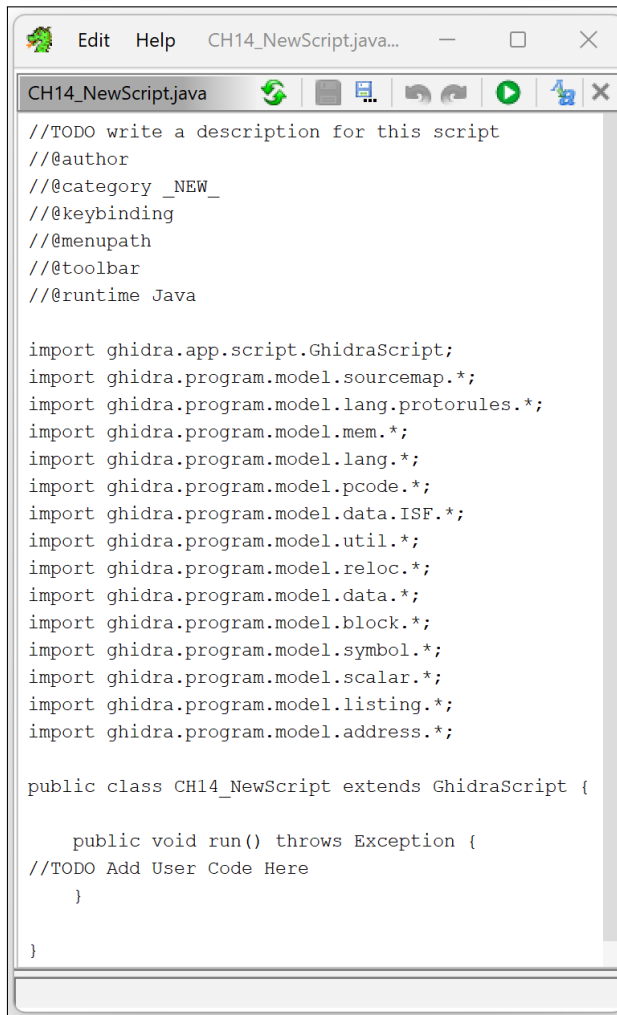


Figure 14-4: A new, empty script

At the top of the file are the metadata comments and tags used by Ghidra to describe your script within the Script Manager and integrate your script into the CodeBrowser. Other than metadata comments, any comments starting with `//` that precede the class, field, or method declarations will become part of the Description for the script. You can embed additional comments within the script, and these will not be included in the description. In addition, the following tags within the metadata comments are supported:

@author Provides information about the author of the script. The information is provided at the discretion of the author and can include any pertinent details (for example, name, contact information, and date of creation).

@category Determines where the script appears within the category tree. This is the only mandatory tag and must be present in all Ghidra scripts. The period (.) character acts as a path separator for category names (for example, @category Ghidrabook.CH14).

@keybinding Documents a shortcut to access the script from the Code-Browser window (for example, @keybinding K).

@menupath Defines a period-delimited menu path for the script that provides a means to run the script from a CodeBrowser menu (for example, @menupath File.Run.ThisScript).

@toolbar Associates an icon with the script. This icon is displayed as a toolbar button in the CodeBrowser window and may be used to run the script. If Ghidra cannot find the image in the script directory or the Ghidra installation, a default image will be used.

@runtime Indicates which runtime environment is needed for the script. This should be set to one of Java, Jython, or PyGhidra. The default is the first environment that is compatible with the script's extension.

When confronted with a new API such as Ghidra's, you may need time before you are comfortable writing scripts without constantly consulting available documentation. Java in particular is very sensitive to classpath issues and the proper inclusion of required support packages. A time- and sanity-saving option is to edit an existing program rather than creating a new program. We adopt this approach in presenting a simple example of a script.

Editing a Script: Regex Search

Assume that you are tasked with developing a script to accept a regular expression as input from the user and output matching strings within a binary to the console. Further, this script needs to appear in the Script Manager for a particular project.

Identifying an existing script with similar functionality can require some effort. For this example, you might discover some good candidate scripts by looking through the script categories for related terms. You could also filter for the term *strings*. In this case, using filters produces a more comprehensive list of string-related candidate scripts than perusing the categories.

For this example, you will edit the first script in the list that shares some functionality with what you want your script to do, *CountAndSaveStrings.java*. Open the script in the editor to confirm that it is a good starting point for our new functionality by right-clicking it and selecting Edit with basic editor; then, save this script with a new name, *FindStringsByRegex.java*, using the Save As option.

Ghidra does not allow you to edit the system scripts provided as part of your Ghidra installation within the Script Manager window (although you can do so in Eclipse and other editors). You could also edit the file prior to using Save As, since Ghidra prevents you from accidentally writing any modified content to the existing *CountAndSaveStrings.java* script.

The original *CountAndSaveStrings.java* contains the following metadata:

```
❶ /* ###
   * IP: GHIDRA
   *
   * Licensed under the Apache License, Version 2.0 (the "License");
   * you may not use this file except in compliance with the License.
   * You may obtain a copy of the License at
   * http://www.apache.org/licenses/LICENSE-2.0
   * Unless required by applicable law or agreed to in writing, software
   * distributed under the License is distributed on an "AS IS" BASIS,
   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   * See the License for the specific language governing permissions and
   * limitations under the License.
   */
❷ //Counts the number of defined strings in the current selection,
   //or current program if no selection is made,
   //and saves the results to a file.
❸ //@category CustomerSubmission.Strings
```

We can leave, modify, or delete the licensing agreement ❶ for the script without impacting its execution or the associated metadata. We will modify the script's description ❷ so that the information displayed in the Script Manager accurately describes our updated version. The script author has included only one of the five available tags ❸, so we will add placeholders for the unpopulated tags and revise the description, as follows:

```
// Counts the number of defined strings that match a regex in the current
// selection, or current program if no selection is made, and displays the
// number of matching strings to the console.
//
//@author Ghidrabook
//@category Ghidrabook.CH14
//@keybinding
//@menupath
//@toolbar
//@runtime Java
```

The category tag *Ghidrabook.CH14* will be added to the Script Manager's tree display, as shown in Figure 14-5.

The next portion of the original script contains Java *import* statements. Of the long list of imports Ghidra includes by default when you create a new script, only the following imports are necessary for string searching, so we will keep the same list as the original *CountAndSaveStrings.java*:

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.listing.*;
import ghidra.program.util.ProgramSelection;
import java.io.*;
```

Save the new script and then select it in the Script Manager to see the content shown in Figure 14-5. The script tree includes our new category (although you may have to refresh to see it), and the script’s metadata is displayed in the information window and script table. The table contains only one script, *Ghidrabook.CH14*, as it is the only script in the selected category.

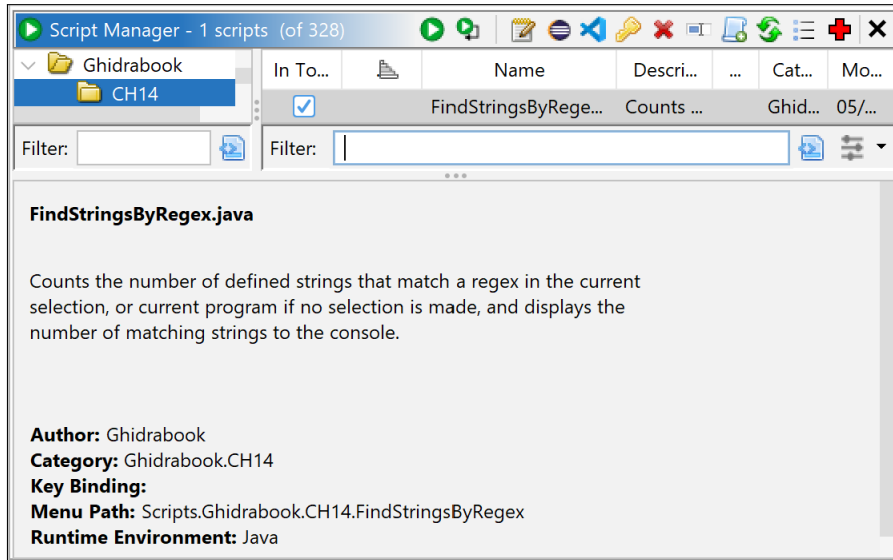


Figure 14-5: New script information displayed in the Script Manager window

As this book is not intended to be a Java tutorial, we summarize the changes we made to the script rather than explaining the Java syntax and functionality. The following list describes the behavior of the default script, *CountAndSaveStrings.java*:

1. Get the program listing content to search.
2. Get the file to save results to.
3. Open the file.
4. Iterate through the program listing, count the number of qualifying strings, and write each qualifying string to the file.
5. Close the file.
6. Write the number of qualifying strings to the console.

We want the following functionality in our modified script:

1. Get the program listing content to search.
2. Ask the user for a regular expression (regex) to search for.
3. Iterate through the program listing, count the number of qualifying strings, and write each qualifying string to the console.
4. Write the number of qualifying strings to the console.

Our new script will be significantly shorter than the original script, as there is no need to interact with the filesystem and perform associated error checking. Here is our implementation:

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.listing.*;
import ghidra.program.util.ProgramSelection;
import java.io.*;

❶ public class FindStringsByRegex extends GhidraScript {
    @Override
    public void run() throws Exception {
        String regex =
            askString("Please enter the regex",
                "Please enter the regex you're looking to match:");

        Listing listing = currentProgram.getListing();

        DataIterator dataIt;
        if (currentSelection != null) {
            dataIt = listing.getDefinedData(currentSelection, true);
        }
        else {
            dataIt = listing.getDefinedData(true);
        }

        Data data;
        String type;
        int counter = 0;
        while (dataIt.hasNext() && !monitor.isCancelled()) {
            data = dataIt.next();
            type = data.getDataType().getName().toLowerCase();
            if (type.contains("unicode") || type.contains("string")) {
                String s = data.getDefaultValueRepresentation();
                if (s.matches(regex)) {
                    counter++;
                    println(s);
                }
            }
        }
        println(counter + " matching strings were found");
    }
}
```

All Java scripts that you write for Ghidra must extend (inherit from) an existing class named `Ghidra.app.script.GhidraScript` ❶. After saving the final version of the script, select it from within the Script Manager and execute it. This script will use the active binary in CodeBrowser as the target. When the

script executes, you should see the prompt shown in Figure 14-6. This figure includes the regular expression we will search for to test our script.

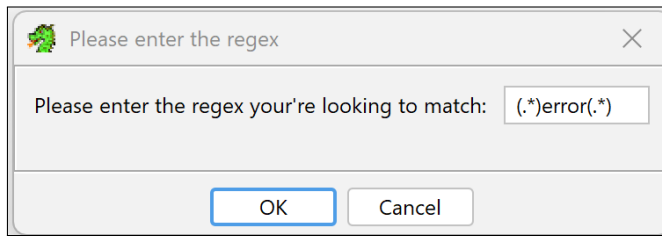


Figure 14-6: The new script's prompt to enter a regex

The CodeBrowser console displays the following when our new script has completed execution:

```
FindStringsByRegex.java> Running...
FindStringsByRegex.java> "Fatal error: glibc detected an invalid stdio handle\n"
FindStringsByRegex.java> "Unknown error "
FindStringsByRegex.java> "internal error"
FindStringsByRegex.java> "relocation error"
FindStringsByRegex.java> "symbol lookup error"
FindStringsByRegex.java> "Fatal error: length accounting in _dl_exception_create_format\n"
FindStringsByRegex.java> "Fatal error: invalid format in exception string\n"
FindStringsByRegex.java> "error while loading shared libraries"
FindStringsByRegex.java> "Unknown error"
FindStringsByRegex.java> "version lookup error"
FindStringsByRegex.java> "sdlerror.o"
FindStringsByRegex.java> "dl-error.o"
FindStringsByRegex.java> "fatal_error"
FindStringsByRegex.java> "strerror.o"
FindStringsByRegex.java> "strerror"
FindStringsByRegex.java> "__strerror_r"
FindStringsByRegex.java> "_dl_signal_error"
FindStringsByRegex.java> "__dLError"
FindStringsByRegex.java> "_dlerror_run"
FindStringsByRegex.java> "_dl_catch_error"
FindStringsByRegex.java> 20 matching strings were found
FindStringsByRegex.java> Finished!
```

This simple example demonstrates the low barrier to entry of Ghidra's extensive Java scripting capabilities. You can easily modify existing scripts and build new scripts from the ground up using the Script Manager. We present some more complex Java scripting capabilities in Chapters 15 and 16, but Java is just one of the scripting options provided by Ghidra. Ghidra also allows you to author scripts in Python through two different approaches, Jython and PyGhidra.

Scripting in Python

Python is a popular language for creating scripts because of its simplicity and numerous available libraries. While the majority of the 325-plus scripts included in the Ghidra release are written in Java, many Ghidra users prefer to use Python as their primary scripting language.

Ghidra's original release relied on Jython for Python support, which provides the advantage of allowing direct access to Ghidra's Java objects. Unfortunately, Jython is compatible with Python 2 (specifically 2.7.1) but not Python 3. Although Python 2 went end-of-life in January 2020, the available Jython scripts continue to function within Ghidra, and you can write and run Python 2 scripts using Jython if you wish.

You can easily locate the Python scripts available in Ghidra by filtering for the `.py` extension in the Script Manager. You will find the majority of the Python scripts in the Examples.Python category in the tree, and many include a disclaimer similar to the one shown in Figure 14-7.

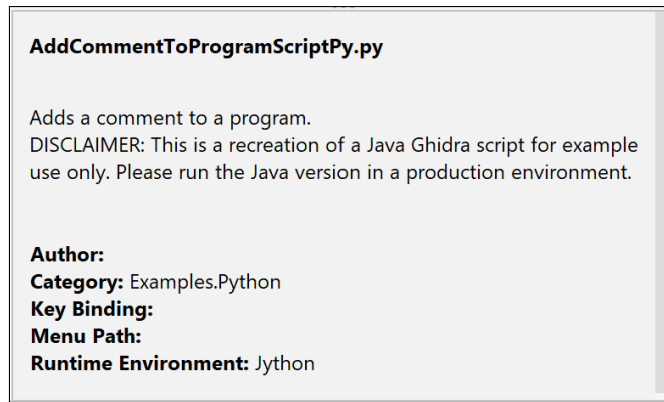


Figure 14-7: A sample Python script with a disclaimer

This warning indicates that many of the Python scripts provided in default Ghidra installations are Python equivalents of default Java scripts. This can be a good way to see how the two languages implement the same functionality (which may be helpful as you write new Python scripts).

There are some scripts that do not have this warning because they are intended as educational models rather than functional scripts. These provide starting points if you prefer to use Python 2 to extend legacy scripts:

ghidra_basics.py This script includes examples of basic Python scripting within Ghidra.

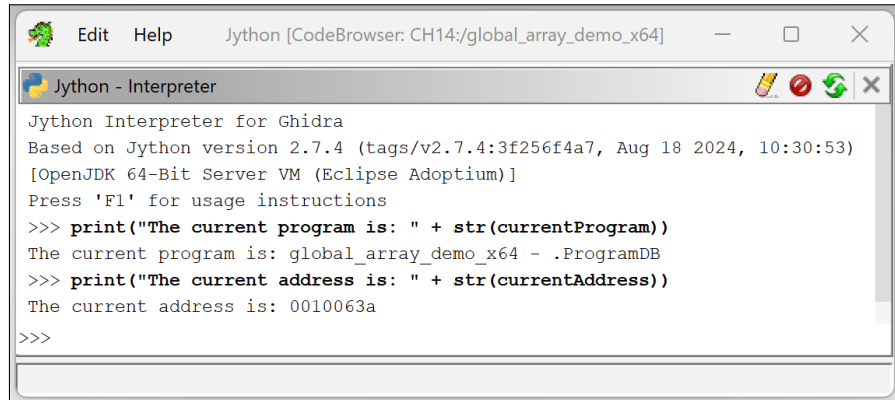
python_basics.py This is a very basic introduction to Python 2 for programmers that might be new to the language.

jython_basic.py This script demonstrates some basic techniques for accessing Java core classes from a Jython script.

The Ghidra features demonstrated in these examples barely scratch the surface of the available Ghidra APIs. You will likely still need to spend some

time reading through Ghidra's library of Java examples before you are ready to access Ghidra's full Java API from your Python scripts.

In addition to running Python scripts, Ghidra provides the Jython Interpreter to allow you to directly access the Java objects associated with Ghidra, as shown in Figure 14-8.

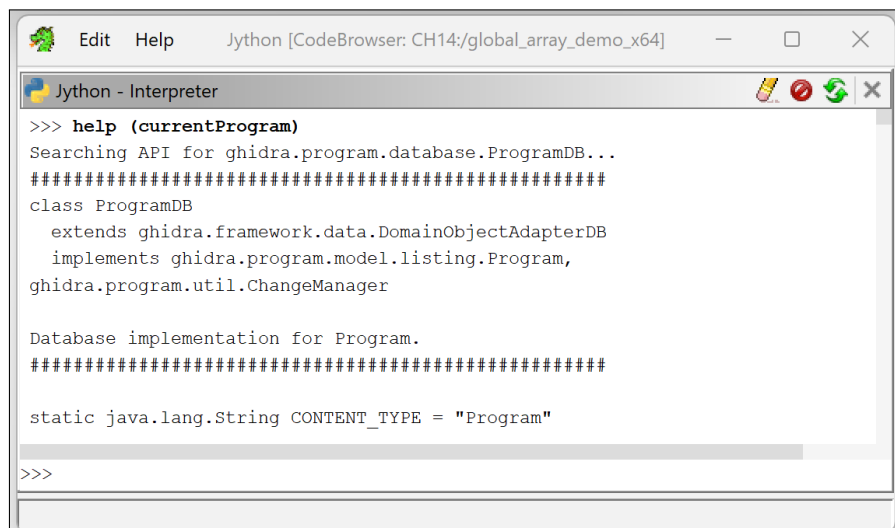


```
Jython - Interpreter
Jython Interpreter for Ghidra
Based on Jython version 2.7.4 (tags/v2.7.4:3f256f4a7, Aug 18 2024, 10:30:53)
[OpenJDK 64-Bit Server VM (Eclipse Adoptium)]
Press 'F1' for usage instructions
>>> print("The current program is: " + str(currentProgram))
The current program is: global_array_demo_x64 - .ProgramDB
>>> print("The current address is: " + str(currentAddress))
The current address is: 0010063a
>>>
```

Figure 14-8: A Jython Interpreter print example

You can access the Jython Interpreter through the CodeBrowser by selecting Window ► Jython. For more information about using the interpreter, see Ghidra Help.

To get API information when using Python and the Python Interpreter, choose Help ► Ghidra API Help at the top left of the Interpreter window shown in Figure 14-8, which opens the Javadoc content on the GhidraScript class. Alternatively, Python has a built-in function, `help()`, that has been modified in Ghidra to provide direct access to Ghidra's Javadoc. To use the function, enter `help(object)` in the interpreter, as shown in Figure 14-9.



```
Jython - Interpreter
>>> help (currentProgram)
Searching API for ghidra.program.database.ProgramDB...
#####
class ProgramDB
    extends ghidra.framework.data.DomainObjectAdapterDB
    implements ghidra.program.model.listing.Program,
ghidra.program.util.ChangeManager

Database implementation for Program.
#####

static java.lang.String CONTENT_TYPE = "Program"
>>>
```

Figure 14-9: A Jython Interpreter Help example

For example, `help(currentProgram)` displays the Ghidra Javadoc content describing the Ghidra API class `ProgramDB`.

Scripting with PyGhidra

While the available Jython functionality allows users to write scripts and access Ghidra's internal APIs, it is limited to Python 2.7 and lacks support for many widely used modern Python libraries, especially those that rely on native C extensions. This limitation became increasingly problematic after Python 2.7 officially went end-of-life, meaning it no longer receives updates, bug fixes, or security patches. As the broader software development community moved to Python 3, the inability to use up-to-date Python features and libraries within Ghidra created a growing barrier for developers and analysts.

To address this, Ghidra 11.3 introduced PyGhidra, a major upgrade that brings full CPython 3 support into the Ghidra ecosystem. Powered by JPyype, PyGhidra provides a bridge between the Java Virtual Machine and the Python interpreter, allowing users to write modern Python scripts that can also directly interact with Ghidra's Java-based APIs.

PyGhidra enables reverse engineers to integrate powerful Python tools into their workflows. For example, you can now use *networkx* to analyze call graphs, *matplotlib* to visualize function sizes or code entropy, or *requests* to pull threat intelligence data from external APIs (as we will do in the next section). By combining Ghidra's robust binary analysis engine with Python's extensive ecosystem, PyGhidra makes Ghidra scripting more powerful, flexible, and accessible.

WHEN PYTHON MET JAVA

JPyype is the behind-the-scenes enabler that lets Python and Java (two languages with famously different personalities) cooperate without argument. It embeds the JVM directly inside the Python process, which means Python code can create Java objects, call Java methods, and interact with Java libraries almost as if they were written in Python (with only a little extra ceremony). Technically, this magic is pulled off through Java's native interface (JNI), but JPyype graciously hides the complex bits so you can focus on getting work done instead of battling JNI exceptions. In the world of PyGhidra, JPyype is the crucial bridge that allows Python scripts to tap into Ghidra's powerful Java-based analysis tools without ever touching a Java IDE. It's the quiet powerhouse that lets Python do what it does best: make hard things look deceptively simple.

When you launch Ghidra using its default startup script (*ghidraRun*), you're starting the application with access to its traditional scripting environments discussed earlier in this chapter (Java and Jython). At this point, you may be wondering why Jython hasn't been replaced by PyGhidra in the Ghidra ecosystem. After all, PyGhidra offers access to modern Python 3, popular libraries, and a far more flexible scripting environment. Jython remains the default scripting engine as it is tightly integrated into Ghidra's Java-based architecture and offers full compatibility with Ghidra's built-in

scripting infrastructure, including many existing analysis scripts and extensions that were written before PyGhidra was introduced. This makes Jython stable, self-contained, and convenient for quick scripting tasks that don't rely on modern Python packages.

However, if you want to use contemporary Python libraries, take advantage of CPython's performance, or write more complex analyses and visualizations (without resorting to scripting in Java), you'll want to start Ghidra with PyGhidra enabled. You can do so by launching Ghidra via the *pyghidrarun* startup script (*pyghidrarun.bat* on Windows), found in the support directory. This alternative launcher starts Ghidra with the PyGhidra plugin activated and embeds a full JVM into a CPython process using JPyype.

The first time you start Ghidra using this script, you may be prompted for some information to successfully install and configure PyGhidra. Once this is completed, you will have this new capability to develop and run Python 3 code in Ghidra.

Example: Using PyGhidra to Add Warnings

Imagine you're analyzing a binary and you've already noticed several high-risk functions such as `strcpy`, `gets`, and `sprintf` that you know are associated with historical vulnerabilities. The real challenge isn't identifying which functions are risky, but locating where they're used in the code and determining whether those usages map to known CVEs. Manually scanning the disassembly for each function call and then cross-referencing against external vulnerability databases is time-consuming and error prone, and it disrupts the flow of analysis. What you need is a way to automatically flag these calls and overlay meaningful vulnerability information right where the reverse engineering happens, inside Ghidra.

THE NATIONAL VULNERABILITY DATABASE: CVE CONFIDENTIAL

The National Vulnerability Database (NVD) is the US government's open, searchable archive of software security flaws. It is basically a hall of fame (or shame) for bugs with names, numbers, and sometimes logos. Maintained by NIST, the NVD organizes vulnerabilities using the Common Vulnerabilities and Exposures (CVE) system, giving each entry a unique ID, severity score, affected software list, and a plain-English explanation of what's gone wrong. So, when you stumble across a vulnerability like CVE-2021-44228, you can head to the NVD to find out whether it's a minor hiccup or a digital dumpster fire.

For security analysts, reverse engineers, and curious developers, the NVD also provides a public REST API, which makes it easy to automate lookups and integrate vulnerability data into your own tools. However, the API comes with a polite request for moderation: It's rate-limited to five requests every 30 seconds per IP address (and bursts of up to 10 requests) unless you register for a free API key. So, if you're writing a script to scan every DLL in sight, it's best to space out your queries, or be prepared to get politely throttled by Uncle Sam's firewall.

In this case, using PyGhidra to query the NVD API is a natural and practical choice. Ghidra already gives you access to imported symbols, library names, and version hints during binary analysis, but querying a modern web API like NVD's requires external libraries like *requests*, which rely on native C extensions. Jython simply can't handle that, as it's limited to Python 2.7 and lacks support for those modern dependencies. PyGhidra, on the other hand, supports CPython 3, which means you can automate NVD lookups based on the artifacts found in your binary and annotate potential vulnerabilities directly in your Disassembly view. This lets you bridge the gap between static analysis and real-time threat intelligence without switching tools.

DANGEROUS FUNCTIONS

C functions like `strcpy` and `sprintf` are considered dangerous to use because they allow unbounded copying into destination buffers. While each may be safely used by programmers who check the size of source and destination buffers, such checks are all too often forgotten by programmers unaware of the dangers of these functions. The `strcpy` function, for example, is declared as follows:

```
char *strcpy(char *dest, const char *source);
```

The `strcpy` function copies all characters up to and including the first null termination character encountered in the source buffer to the given destination buffer (`dest`). The fundamental problem is that there is no way to determine, at runtime, the size of any array, and `strcpy` cannot determine whether the capacity of the destination buffer is sufficient to hold all of the data to be copied from the source. Such unchecked copy operations are a major cause of buffer overflow vulnerabilities.

This script does exactly that: It searches for calls to known dangerous functions in your binary, queries the NVD for related CVEs, and annotates each call site with a plate comment providing information about the vulnerability, allowing you to prioritize your reverse engineering efforts more efficiently and with greater situational awareness.

```
# This script searches a binary for calls to dangerous functions listed
# in the function map within the script. It then queries the
# National Vulnerability Database (NVD) and inserts a plate comment
# in the binary, explaining the vulnerability for each of the requested
# vulnerable functions in the binary.
#
# You may need to install the Python requests library on your machine.
#
#@author GhidraBook
#@category GhidraBook.CH14
#@keybinding
#@menupath
#@toolbar
```



```

#@runtime PyGhidra

import requests
from ghidra.program.model.symbol import Symbol
from ghidra.program.model.listing import Instruction
from ghidra.program.model.address import Address

# Map of sample vulnerable functions to associated library
func_map = {
    "strcpy": "libc",
    "gets": "libc",
    "sprintf": "libc"
}

# NVD API setup
NVD_URL = "https://services.nvd.nist.gov/rest/json/cves/2.0"
MAX_CVES = 3

# Query NVD for CVEs related to a given library name
def get_cves(lib):
    try:
        params = {"keywordSearch": lib, "resultsPerPage": MAX_CVES}
        r = requests.get(NVD_URL, params=params)
        r.raise_for_status()
        data = r.json()
        vulns = data.get("vulnerabilities", [])
        return [f"{v['cve']['id']}: {v['cve']['descriptions'][0]['value']}" for v in vulns]
    except Exception as e:
        print(f"Error querying CVEs for {lib}: {e}")
        return []

# Build symbol table mappings
sym_table = currentProgram.getSymbolTable()
syms = sym_table.getAllSymbols(True)
name_addr = {s.getName().lower(): s.getAddress() for s in syms}
addr_func = {addr: name for name, addr in ((n, name_addr.get(n)) for n in func_map) if addr}

# Cache CVEs per library
cves = {}
for fn in addr_func.values():
    lib = func_map[fn]
    if lib not in cves:
        cves[lib] = get_cves(lib)

# Scan instructions and annotate calls
listing = currentProgram.getListing()
instrs = listing.getInstructions(True)

```

```

for ins in instrs:
    if not ins.getMnemonicString().startswith("CALL"):
        continue
    for i in range(ins.getNumOperands()):
        for op in ins.getOpObjects(i):
            if isinstance(op, Address):
                sym = sym_table.getPrimarySymbol(op)
                if sym:
                    fn = sym.getName().lower()
                    if fn in func_map:
                        for cve in cves[func_map[fn]]:
                            addr = ins.getAddress()
                            setPlateComment(addr, f"Call to {fn}: Possible CVE - {cve}")

```

After running this script on a binary that included a call to a dangerous function, you should see the plate comment in the listing inserted by PyGhidra, as shown in Figure 14-10.

004011c6	MOV	RDX,qword ptr [RBP + local_50]
004011ca	LEA	RAX=>local_48,[RBP + -0x40]
004011ce	MOV	RSI,RDX
004011d1	MOV	RDI,RAX

* Call to strcpy: Possible CVE - CVE-1999-0789: Buffer overflow in AI... *		

004011d4	CALL	<EXTERNAL>::strcpy
004011d9	LEA	RAX=>local_48,[RBP + -0x40]
004011dd	MOV	RSI,RAX

Figure 14-10: The new plate comment in the Listing window

While the example shown here is intentionally simple, it effectively demonstrates the extended capabilities that PyGhidra brings to Ghidra scripting. By leveraging modern Python libraries like *requests*, the script performs live API queries and integrates real-time external data directly into the reverse engineering workflow. This opens the door to a vast ecosystem of Python 3 tools and libraries, enabling analysts to automate tasks that go far beyond static analysis, and to build smarter, more responsive tooling within Ghidra itself.

You may have noticed some unfamiliar imports at the start of the PyGhidra script that enabled access to artifacts within the Listing window. While PyGhidra lets us write scripts in modern Python, those scripts are still interacting with the underlying Ghidra platform via its Java classes and interfaces. Understanding how to navigate and use the Ghidra API opens the door to much more powerful automation, analysis, and customization.

An Introduction to the Ghidra API

Ghidra exposes its API in two rather different styles. The *Program* API defines an object hierarchy, many levels deep, rooted at the top by the `Program` class. This API may change from one version of Ghidra to another. The *Flat* API flattens out the *Program* API by exposing all levels of that API from a single class, `FlatProgramAPI`. The Flat API is often the most convenient way to access many Ghidra constructs. In addition, it is less likely to change from one version of Ghidra to the next.

For the remainder of the chapter, we highlight some of the more useful Flat API functionality. When necessary, we also provide detail about specific classes from the *Program* API. We use Java as the language for this discussion, as it is the native language of Ghidra.

The Ghidra API contains many packages, classes, and associated functions to interface with your Ghidra projects and associated files, all detailed in Javadoc-style documentation supplied with Ghidra that can be accessed by clicking the red plus in the Script Manager window. This documentation, in conjunction with the sample scripts supplied with Ghidra, is your primary reference for the APIs and how to use them. The most common way to figure out how to do something is to browse the Ghidra classes looking for one that, based on its name, appears to do what you need. As you gain more experience with Ghidra, your increased understanding of the naming conventions and package organization will help you identify appropriate classes more quickly.

Ghidra adheres to the Java Swing *model-delegate* architecture, in which data values and characteristics are stored in model objects and displayed by user interface delegate objects such as tree, list, and table views. Delegates handle events, such as mouse clicks, to update and refresh data and views. In the overwhelming majority of cases, your scripts will focus on the data encapsulated in the model classes used to represent various program and reverse engineering constructs.

The rest of this section focuses on commonly used model classes, their relationships to each other, and useful APIs for interacting with them. We make no attempt to cover the entire Ghidra API, and many more functions and classes are available. The authoritative documentation for the entire Ghidra API is the Javadoc that ships with Ghidra, and ultimately the Java source code from which Ghidra is built.

The Address Interface

The `Address` interface describes a model for an address within an address space. All addresses are represented by an offset up to 64 bits in size. Segmented addresses may be further qualified by a segment value. In many cases, an address's offset is equivalent to a virtual address within a program listing. The `getOffset` method retrieves the long offset value from an `Address` instance. Many Ghidra API functions require `Address` objects as arguments or return an `Address` object as a result.

The Symbol Interface

The `Symbol` interface defines properties common to all symbols. At a minimum, a symbol is composed of a name and an address. These attributes may be retrieved with the following member functions:

Address getAddress()
Returns the address of the `Symbol`

String getName()
Returns the name of the `Symbol`

The Reference Interface

A `Reference` models a cross-reference relationship (as described in Chapter 9) between a source address and a destination address and is characterized by a reference type. Useful functions associated with a `Reference` include these:

public Address getFromAddress()
Returns the source address for this reference

public Address getToAddress()
Returns the destination address for this reference

public RefType getReferenceType()
Returns a `RefType` object that describes the nature of the link between the source and destination addresses

The GhidraScript Class

Although this class does not model a specific attribute in a binary, every script that you write must be a subclass of `GhidraScript`, which, in turn, is a subclass of `FlatProgramAPI`. As a result, your scripts have instantaneous access to the entire Flat API, and your only obligation is to provide an implementation of

```
protected abstract void run() throws Exception;
```

which, hopefully, makes your script do something interesting. The remainder of the `GhidraScript` class gives you access to the most common resources for interacting with the Ghidra user and the program that is being analyzed. We summarize some of the more useful functions and data members of this class (including some inherited from `FlatProgramAPI`) in the following sections.

Useful Data Members

The `GhidraScript` class provides convenient access to a number of objects commonly referenced in scripts, including the following:

protected Program currentProgram
This is the current open program. We discuss the `Program` class later. This data member is likely your gateway to retrieving more interesting information, such as instruction and symbol lists.

protected Address currentAddress

This is the address of the current cursor location. We discuss the Address class later.

protected ProgramLocation currentLocation

A ProgramLocation object that describes the current cursor location, including its address, cursor row, column, and other information.

protected ProgramSelection currentSelection

A ProgramSelection object representing a range of addresses selected in the Ghidra GUI.

protected TaskMonitor monitor

The TaskMonitor class updates the status of long-running tasks and checks to determine whether a long-running task has been cancelled by the user (`monitor.isCancelled()`). Any long-running loops you write should incorporate a call to `monitor.isCancelled` as an additional termination condition to recognize that the user has attempted to cancel your script.

User Interface Functions

The GhidraScript class provides convenience functions for basic user interface operations, ranging from simple message output to more interactive dialog elements. We describe some of the more common user interface functions here:

public void println(String message)

Prints *message* followed by a linefeed to Ghidra's console window. This function is useful for printing status messages or results of your scripts in a nonintrusive manner.

public void printf(String message, Object... args)

Uses as a Java format string and prints the resulting string of formatted *args* to Ghidra's console window.

public void popup(final String message)

Displays *message* in a pop-up dialog that requires the user to click OK before script execution can continue. This is a more intrusive way to display status messages to a user.

public String askString(String title, String message)

One of many available ask functions. `askString` displays a text input dialog, using *message* as a prompt, and returns the text entered by the user.

public boolean askYesNo(String title, String question)

Uses a dialog to ask the user a yes-or-no *question*. Returns true for yes and false for no.

public Address askAddress(String title, String message)

Displays a dialog, using *message* as a prompt, that parses the user's input into an Address object.

public int askInt(String title, String message)

Displays a dialog, using *message* as a prompt, that parses the user's input into an int.

public File askFile(final String title, final String approveButtonText)

Displays a system file chooser dialog and returns a Java File object representing the file selected by the user.

public File askDirectory(final String title, final String approveButtonText)

Displays a system file chooser dialog and returns a Java File object representing the directory selected by the user.

public boolean goTo(Address address)

Repositions all connected Ghidra disassembly windows to *address*. Overloaded versions of this function take a Symbol or a Function argument and navigate the displays accordingly.

Address-Related Functions

For a processor, an address is typically just a number that happens to refer to a memory location. Ghidra models addresses using the Address class. GhidraScript provides a wrapper function that offers easy conversion from numbers to Ghidra Address objects:

public Address toAddr(long offset)

Convenience function to create an Address object in the default address space

Reading Program Memory

The Memory class represents contiguous ranges of byte values, such as the contents of an executable file loaded into Ghidra. Within a Memory object, every byte value is associated with an address, although addresses may be tagged as uninitialized and have no value to retrieve. If you attempt to access a location within a memory object with an invalid address, Ghidra throws a MemoryAccessException. Consult the documentation for the Memory class for a full description of available API functions. The following convenience functions expose some of the Memory class via the Flat API:

public byte getByte(Address addr)

Returns the single byte value retrieved from *addr*. Data type byte is a signed type in Java, so this value will be in the range -128..127.

public byte[] getBytes(Address addr, int length)

Returns *length* bytes from memory, beginning at *addr*.

public int getInt(Address addr)

Returns the 4-byte value, beginning at *addr*, as a Java int. This function is endianness-aware and respects the binary's underlying architecture when reconstituting the int value.

public long getLong(Address addr)

Returns the 8-byte value, beginning at *addr*, as a Java long. This function is endianness-aware and respects the binary's underlying architecture when reconstituting the long value.

Program Search Functions

Ghidra's search capabilities reside within different Program API classes according to the type of item being searched for. The `Memory` class contains raw byte search functionality. Code units (such as `Data` and `Instruction`), comment text, and associated iterators are obtained from the `Listing` class. Symbols/labels and associated iterators are accessed via the `SymbolTable` class. The following convenience functions expose some of the available search functionality via the Flat API:

public `Data` `getFirstData()`

Returns the first data item in the program.

public `Data` `getDataAfter(Data data)`

Returns the next data item after *data*, or null if no such data exists.

public `Data` `getDataAt(Address address)`

Returns the data item at *address*, or null if no such data exists.

public `Instruction` `getFirstInstruction()`

Returns the first instruction in the program.

public `Instruction` `getInstructionAfter(Instruction instruction)`

Returns the next instruction item after *instruction*, or null if no such instruction exists.

public `Instruction` `getInstructionAt(Address address)`

Returns the instruction at *address*, or null if no such instruction exists.

public `Address` `find(String text)`

Searches for a *text* string within the Listing window. Listing components are searched in the following order:

1. Plate comments
2. Pre comments
3. Labels
4. Code unit mnemonics and operands
5. EOL comments
6. Repeatable comments
7. Post comments

A successful search returns the address containing the match. Note that as a result of the search order, the returned address may *not* represent the first occurrence of text in the disassembly listing when considered in strictly increasing address order.

public `Address` `find(Address start, byte[] values)`

Searches memory, beginning at *addr*, for a specified sequence of byte values. When *addr* is null, the search begins at the lowest valid address in the binary. A successful search returns the address of the first byte in the matching sequence.

public `Address` `findBytes(Address start, String byteString)`

Searches memory, beginning at *addr*, for a specified *byteString* that may contain regular expressions. When *addr* is null, the search begins at the lowest valid address in the binary. A successful search returns the address of the first byte in the matching sequence.

Manipulating Labels and Symbols

The need to manipulate named locations arises fairly often in scripts. The following functions are available for working with named locations in a Ghidra database:

public Symbol getSymbolAt(Address address)

Returns the Symbol associated with the given *address*, or null if the location has no Symbol.

public Symbol createLabel(Address address, String name, boolean makePrimary)

Assigns the given *name* to the given *address*. Ghidra allows multiple names to be assigned to a single address. If *makePrimary* is true, the new name will become the primary name associated with *address*.

public List<Symbol> getSymbols(String name, Namespace namespace)

Returns a list of all symbols named *name* in *namespace*. When *namespace* is null, the global namespace is searched. If the result is empty, the named symbol does not exist. If the result contains only one element, the name is unique.

Working with Functions

Many scripts are designed to analyze functions within a program. The following functions can be used to access information about program functions:

public Function getFirstFunction()

Returns the first Function object in the program

public Function getGlobalFunctions(String name)

Returns a list of all Function objects matching the name

public Function getFunctionAt(Address entryPoint)

Returns the Function object for the function at *entryPoint*, or null if no such function exists

public Function getFunctionAfter(Function function)

Returns the Function object for the successor to *function*, or null if no such function exists

public Function getFunctionAfter(Address address)

Returns the Function object for the function that starts after *address*, or null if no such function exists

Working with Cross-References

We covered cross-references in Chapter 9. In the Ghidra Program API, the top-level Program object contains a ReferenceManager, which, unsurprisingly, manages the references within the program. As with many other program constructs, the Flat API offers convenience functions for accessing cross-references, some of which we detail here:

public Reference[] getReferencesFrom(Address address)

Returns an array of all Reference objects originating from *address*

public Reference[] getReferencesTo(Address address)

Returns an array of all Reference objects terminating at *address*

Program Manipulation Functions

When automating your analysis tasks, you may find yourself wanting to add new information into a program. The Flat API provides a variety of functions for modifying the contents of a program, including the following:

public final void clearListing(Address address)

Removes any instruction or data defined at *address*.

public void removeFunctionAt(Address entryPoint)

Removes the function at *entryPoint*.

public boolean disassemble(Address address)

Performs a recursive descent disassembly beginning at *address*. Returns true if the operation is successful.

public Data createByte(Address address)

Converts the item at the specified address into a data byte. There are other similar data creation functions available, including `createWord`, `createDword`, `createQword`, and others.

public boolean setEOLComment(Address address, String comment)

Adds an EOL comment at the given address. Additional comment-related functions include `setPlateComment`, `setPreComment`, and `setPostComment`.

public Function createFunction(Address entryPoint, String name)

Creates a function with the given *name* at *entryPoint*. Ghidra attempts to automatically identify the end of the function by locating the function's return instruction.

public Data createAsciiString(Address address)

Creates a null-terminated ASCII string at *address*.

public Data createAsciiString(Address address, int length)

Creates an ASCII string of the specified length at *address*. If *length* is zero or less, Ghidra attempts to automatically locate the string's null terminator.

public Data createUnicodeString(Address address)

Creates a null-terminated Unicode string at *address*.

The Program Class

The Program class represents the root of the Program API hierarchy and the outermost layer of the data model of a binary file. You will commonly use a Program object (often `currentProgram`) to access the binary model. Commonly used Program class member functions include the following:

public Listing getListing()

Retrieves the Listing object for the current program.

public FunctionManager getFunctionManager()

Retrieves the program's FunctionManager, which provides access to all of the functions that have been identified within the binary. This class provides the functionality to map an Address back to its containing Function

(Function `getFunctionContaining(Address addr)`). In addition, it provides a `FunctionIterator`, which is useful when you want to process every function in the program.

public SymbolTable getSymbolTable()

Retrieves the program's `SymbolTable` object. Using a `SymbolTable`, you can work with individual symbols or iterate over every symbol in the program.

public Memory getMemory()

Retrieves the `Memory` object associated with this program, which allows you to work with raw program byte content.

public ReferenceManager getReferenceManager()

Retrieves the program's `ReferenceManager` object. A `ReferenceManager` may be used to add and remove references as well as retrieve iterators for many types of references.

public Address getMinAddress()

Returns the lowest valid address within the program. This is most often the binary's base memory address.

public Address getMaxAddress()

Returns the highest valid address within the program.

public LanguageID getLanguageID()

Returns the object representation of the binary's language specification. The language specification itself may then be retrieved using the `getIdAsString()` function.

The Function Interface

The `Function` interface defines the required Program API behaviors of function objects. Member functions provide access to various attributes commonly associated with functions and include the following:

**public String getPrototypeString(boolean *formalSignature*,
boolean *includeCallingConvention*)**

Returns the function object's prototype as a string. The two arguments influence the format of the returned prototype string.

public AddressSetView getBody()

Returns the address set that contains the function's body of code. An *address set* is composed of one or more address ranges and allows for situations in which a function's code is distributed among several noncontiguous ranges of memory. Obtain an `AddressIterator` to visit all addresses in the set or an `AddressRangeIterator` to iterate over each range. Note that you must use a `Listing` object to retrieve the actual instructions contained in the function's body (see `getInstructions`).

public StackFrame getStackFrame()

Returns the stack frame associated with the function. The result may be used to retrieve detailed information about the layout of the function's local variables and stack-based arguments.

The Instruction Interface

The Instruction interface defines the required Program API behaviors of instruction objects. Member functions provide access to various attributes commonly associated with instructions and include the following:

public String getMnemonicString()

Returns the instruction's mnemonic.

public String getComment(int commentType)

Returns the commentType comment associated with the instruction, or null if no comment of the given type is associated with the instruction.

A *commentType* may be one of EOL_COMMENT, PRE_COMMENT, POST_COMMENT, or REPEATABLE_COMMENT.

public int getNumOperands()

Returns the number of operands associated with this instruction.

public int getOperandType(int opIndex)

Returns a bitmask of operand type flags defined in class OperandType.

public String toString()

Returns the string representation of the instruction.

Ghidra Scripting Examples

In the remainder of the chapter, we present some fairly common situations in which you could use a script to answer a question about a program. For brevity, we show only the body of each script's run function.

Example 1: Enumerating Functions

Many scripts operate on individual functions. For example, they might generate the call tree rooted at a specific function, generate the control flow graph of a function, or analyze the stack frames of every function in a program. The following script, *ch_14_1_flat.java*, iterates through every function in a program and prints basic information about each function, including its start and end addresses, the size of the function's arguments, and the size of the function's local variables. All output is sent to the console window:

```
// ch14_1_flat.java
public void run() throws Exception {
    int ptrSize = currentProgram.getDefaultPointerSize();
    ❶ Function func = getFirstFunction();
    while (func != null && !monitor.isCancelled()) {
        String name = func.getName();
        long addr = func.getBody().getMinAddress().getOffset();
        long end = func.getBody().getMaxAddress().getOffset();
        ❷ StackFrame frame = func.getStackFrame();
        ❸ int locals = frame.getLocalSize();
        ❹ int args = frame.getParameterSize();
```

```

        printf("Function: %s, starts at %x, ends at %x\n", name, addr, end);
        printf("  Local variable area is %d bytes\n", locals);
        printf("  Arguments use %d bytes (%d args)\n", args, args / ptrSize);
    ❸ func = getFunctionAfter(func);
    }
}

```

The script uses Ghidra's Flat API to iterate over all functions, starting from the first ❶ and advancing through each in succession ❷. It obtains a reference to each function's stack frame ❸ and retrieves the size of the local variables ❹ and the stack-based arguments ❺. It prints a summary for each function before continuing the iteration.

Example 2: Enumerating Instructions

Within a given function, you may want to enumerate every instruction. The *ch_14_2_flat.java* script counts the number of instructions contained in the function identified by the current cursor position:

```

// ch14_2_flat.java
public void run() throws Exception {
    Listing plist = currentProgram.getListing();
    ❶ Function func = getFunctionContaining(currentAddress);
    if (func != null) {
        ❷ InstructionIterator iter = plist.getInstructions(func.getBody(), true);
        int count = 0;
        while (iter.hasNext() && !monitor.isCancelled()) {
            count++;
            Instruction ins = iter.next();
        }
        ❸ popup(String.format("%s contains %d instructions\n",
                               func.getName(), count));
    }
    else {
        popup(String.format("No function found at location %x",
                               currentAddress.getOffset()));
    }
}

```

The function begins by obtaining a reference to the function containing the cursor ❶. If a function is found, the next step is to use the program's Listing object to obtain an InstructionIterator over the function ❷. The iteration loop counts the number of instructions retrieved and reports the total to the user with a pop-up message dialog ❸.

Example 3: Enumerating Cross-References

Iterating through cross-references can be confusing because the Ghidra API offers a number of functions for accessing cross-reference data, and because

code cross-references are bidirectional. To get the data you want, you need to access the proper type of cross-reference for your situation.

In our first cross-reference example script, *ch_14_3_flat.java*, we retrieve the list of all function calls made within a function by iterating through each instruction in the function to determine if the instruction calls another function.

One method of doing this might be to parse the results of the function `getMnemonicString` to look for call instructions. However, this would not be a very portable or efficient solution because the instruction used to call a function varies among processor types, and you would need to perform additional parsing to determine exactly which function was being called. Cross-references avoid each of these difficulties because they are independent of the processors and directly inform us about the target of the cross-reference:

```
// ch14_3_flat.java
public void run() throws Exception {
    Listing plist = currentProgram.getListing();
    ❶ Function func = getFunctionContaining(currentAddress);
    if (func != null) {
        String fname = func.getName();
        InstructionIterator iter = plist.getInstructions(func.getBody(), true);
        ❷ while (iter.hasNext() && !monitor.isCancelled()) {
            Instruction ins = iter.next();
            Address addr = ins.getMinAddress();
            Reference refs[] = ins.getReferencesFrom();
            ❸ for (int i = 0; i < refs.length; i++) {
                ❹ if (refs[i].getReferenceType().isCall()) {
                    Address tgt = refs[i].getToAddress();
                    Symbol sym = getSymbolAt(tgt);
                    String sname = sym.getName();
                    long offset = addr.getOffset();
                    printf("%s calls %s at 0x%x\n", fname, sname, offset);
                }
            }
        }
    }
}
```

We begin by obtaining a reference to the function containing the cursor ❶. Next, we iterate through each instruction in the function ❷, and for each instruction, we iterate through each cross-reference from the instruction ❸. We are interested only in cross-references that call other functions, so we must test the return value of `getReferenceType` ❹ to determine whether `isCall` is true.

Example 4: Finding Function Calls

Cross-references are also useful for identifying every instruction that references a particular location. In our *ch_14_3_flat.java* script, we iterate across

all of the cross-references *to* a particular symbol (as opposed to *from*, in the previous example):

```
// ch14_4_flat.java
❶ public void list_calls(Function tgtfunc) {
    String fname = tgtfunc.getName();
    Address addr = tgtfunc.getEntryPoint();
    Reference refs[] = getReferencesTo(addr);
    ❷ for (int i = 0; i < refs.length; i++) {
        ❸ if (refs[i].getReferenceType().isCall()) {
            Address src = refs[i].getFromAddress();
            ❹ Function func = getFunctionContaining(src);
            if (func.isThunk()) {
                continue;
            }
            String caller = func.getName();
            long offset = src.getOffset();
            ❺ printf("%s is called from 0x%x in %s\n", fname, offset, caller);
        }
    }
}

❻ public void getFunctions(String name, List<Function> list) {
    SymbolTable symtab = currentProgram.getSymbolTable();
    SymbolIterator si = symtab.getSymbolIterator();
    while (si.hasNext()) {
        Symbol s = si.next();
        if (s.getSymbolType() != SymbolType.FUNCTION || s.isExternal()) {
            continue;
        }
        if (s.getName().equals(name)) {
            list.add(getFunctionAt(s.getAddress()));
        }
    }
}

public void run() throws Exception {
    List<Function> funcs = new ArrayList<Function>();
    getFunctions("strcpy", funcs);
    getFunctions("sprintf", funcs);
    funcs.forEach((f) -> list_calls(f));
}
```

In this example, we have written the helper function `getFunctions` ❹ to collect `Function` objects associated with our functions of interest. For each function of interest, we call a second helper function, `list_calls` ❶, to process all cross-references ❷ to the function. If the cross-reference type is determined to be a call-type cross-reference ❸, the calling function is retrieved ❹ and its name is displayed to the user ❺. Among other things, this approach could be used to address the same issue as the PyGhidra script.

Example 5: Emulating Assembly Language Behavior

There are a number of reasons you might need to write a script that emulates the behavior of a program you are analyzing. For example, the program you are studying may be self modifying, as many malware programs are, or the program may contain some encoded data that gets decoded when needed at runtime. Without running the program and pulling the modified data out of the running process's memory, how can you understand the program's behavior?

If the decoding process is not very complex, you may be able to quickly write a script that performs the same actions that a program performs when it runs. Using a script to decode data in this way eliminates the need to run the program when you don't know what the program does or you don't have access to a platform on which you can run the program. For example, without a MIPS execution environment, you cannot execute a MIPS binary and observe any data decoding that it might perform. However, you could write a Ghidra script to mimic the behavior of the binary and make the required changes within your Ghidra project, all without the need for a MIPS execution environment.

The following x86 code was extracted from a DEF CON Capture the Flag (CTF) binary provided courtesy of Kenshoto, the organizers of CTF at DEF CON 15:

```
08049ede  MOV     dword ptr [EBP + local_8],0x0
                LAB_08049ee5
08049ee5  CMP     dword ptr [EBP + local_8],0x3c1
08049eec  JA      LAB_08049f0d
08049eee  MOV     EDX,dword ptr [EBP + local_8]
08049ef1  ADD     EDX,DAT_0804b880
08049ef7  MOV     EAX,dword ptr [EBP + local_8]
08049efa  ADD     EAX,DAT_0804b880
08049eff  MOV     AL,byte ptr [EAX]=>DAT_0804b880
08049f01  XOR     EAX,0x4b
08049f04  MOV     byte ptr [EDX],AL=>DAT_0804b880
08049f06  LEA     EAX=>local_8,[EBP + -0x4]
08049f09  INC     dword ptr [EAX]=>local_8
08049f0b  JMP     LAB_08049ee5
```

This code decodes a private key that has been embedded within the program binary. Using the *ch14_5_flat.java* script, we can extract the private key without running the program:

```
// ch14_5_flat.java
public void run() throws Exception {
    int local_8 = 0;
    while (local_8 <= 0x3C1) {
        long edx = local_8;
        edx = edx + 0x804B880;
        long eax = local_8;
```

```

    eax = eax + 0x804B880;
    int al = getByte(toAddr(eax));
    al = al ^ 0x4B;
    setByte(toAddr(edx), (byte)al);
    local_8++;
}
}

```

This script is a fairly literal translation of the preceding assembly language sequence generated according to the following mechanical rules:

- For each stack variable and register used in the assembly code, declare an appropriately typed script variable.
- For each assembly language statement, write a statement that mimics its behavior.
- Emulate reading and writing stack variables by reading and writing the corresponding variable declared in your script.
- Emulate reading from a nonstack location using the `getByte`, `getWord`, `getDword`, or `getQword` function, depending on the amount of data being read (1, 2, 4, or 8 bytes, respectively).
- Emulate writing to a nonstack location using the `setByte`, `setWord`, `setDword`, or `setQword` function, depending on the amount of data being written.
- If the code contains a loop with no immediately obvious termination, begin with an infinite loop such as `while(true){...}` and then insert a `break` statement when you encounter statements that cause the loop to terminate.
- When the assembly code calls functions, things become complicated. To properly simulate the behavior of the assembly code, you must mimic the behavior of the function that has been called, including providing a return value that makes sense within the context of the code being simulated.

As the complexity of the assembly code increases, it becomes more challenging to write a script that emulates all aspects of an assembly language sequence, but you don't have to fully understand how the code you are emulating works. Translate one or two instructions at a time. If each instruction has been correctly translated, the script as a whole should properly mimic the complete functionality of the original assembly code. After the script has been completed, you can use it to better understand the underlying assembly. You will see this approach, and more generic emulation functionality, used again in Chapter 21, when we discuss the analysis of obfuscated binaries.

For example, once we translate the sample algorithm and spend some time considering how it works, we can shorten the emulation script as follows:

```
public void run() throws Exception {
    for (int local_8 = 0; local_8 <= 0x3C1; local_8++) {
        Address addr = toAddr(0x804B880 + local_8);
        setByte(addr, (byte)(getByte(addr) ^ 0x4B));
    }
}
```

Once the script executes, you can see the decoded private key starting at address 0x804B880. If you don't want to modify the Ghidra database when emulating code, replace the `setByte` function call with a call to `printf`, which will output the results to the CodeBrowser console, or write the data to a disk file for binary data. Don't forget that in addition to Ghidra's Java API, you have access to all of the standard Java API classes as well as any other Java packages that you choose to install on your system.

Summary

Scripting provides a powerful means of automating repetitive tasks and extending Ghidra's capabilities. This chapter has introduced Ghidra's functionality for editing and building new scripts using both Java and Python as well as the integration of PyGhidra for seamless Python 3 integration. The integrated ability to build, compile, and run Java-based scripts within the CodeBrowser environment lets you extend Ghidra's capabilities without requiring an in-depth understanding of the underlying intricacies of the Ghidra development environment. Chapters 15 and 16 introduce more advanced scripting, IDE integration, and the ability to run Ghidra in headless mode.

15

INTEGRATED SCRIPTING WITH ECLIPSE AND GHIDRADEV



The basic script editor provided by Ghidra's Script Manager is fine for quick-and-dirty work, but it lacks the sophistication needed to manage complex projects. For more substantial development, Ghidra has long provided a plugin for the Eclipse integrated development environment (IDE), which we will demonstrate in this section. Eclipse allows us to build more advanced Ghidra scripts and create new Ghidra modules. We will return to this topic in later chapters as we expand Ghidra's inventory of loaders and examine the inner workings of Ghidra processor modules.

More recently, Ghidra has also added support for Visual Studio Code. We will use Eclipse for our examples, but those comfortable with VS Code will be able to follow along easily using its similar feature set.

Eclipse

Eclipse is an IDE that is used by many Java developers, making it a natural fit for Ghidra development. While it is possible to run both Eclipse and Ghidra on the same machine without any interaction between them, the integration of the two provides you with a rich IDE that includes Ghidra-specific functionality, resources, and templates to facilitate your Ghidra development process.

Integrating Eclipse and Ghidra does not require significant effort; you just need to provide each with some information about the other so that they can be used together.

Integrating Ghidra and Eclipse

In order for Ghidra to work with Eclipse, Eclipse must have the GhidraDev plugin installed. You can integrate the two applications from within either Ghidra or Eclipse. Detailed instructions for each of the GhidraDev integration approaches are included in the *README.html* document found in the *Extensions/Eclipse/GhidraDev* directory of your Ghidra installation.

An easy way to start is to select a Ghidra action that requires Eclipse, such as Edit Script with Eclipse. If you select this option and have not previously integrated Eclipse and Ghidra, you will be prompted for the directory information required to make the connection. Depending on your configuration, you may need to provide the path to your Eclipse installation directory, your Eclipse workspace directory, your Ghidra installation directory, your Eclipse drop-in directory, and possibly the port number used to communicate with Eclipse for script editing.

Ghidra's documentation will help you overcome any obstacles you encounter during the integration process. The truly adventurous can explore the integration plugins in the *Ghidra/Features/Base/src/main/java/ghidra/app/plugin/core/eclipse* directory in Ghidra's source repository.

Starting Eclipse

Once Ghidra and Eclipse are successfully integrated, you can use them to write Ghidra scripts and plugins. The first time you launch Eclipse after integrating it with Ghidra, you are likely to see the dialog shown in Figure 15-1, requesting to establish a communication path between your Ghidra instance and your Eclipse GhidraDev instance.

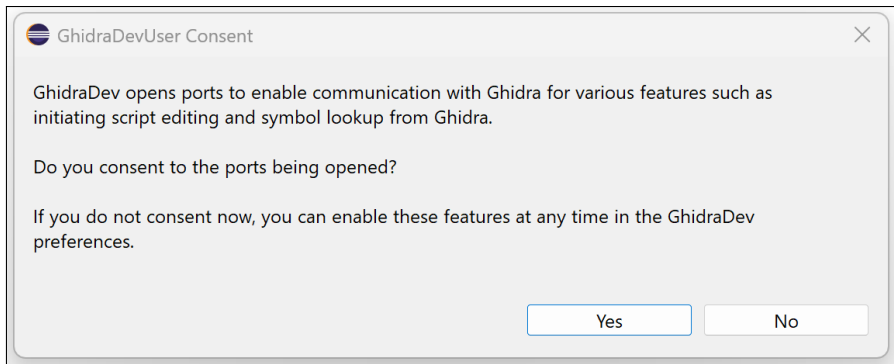


Figure 15-1: The GhidraDevUser Consent dialog

Venturing onward, you will see the Eclipse IDE welcome screen, as shown in Figure 15-2. This instance of Eclipse has a new addition to the menu bar: GhidraDev. This is the menu we will use to create more complex scripts and Ghidra tools.

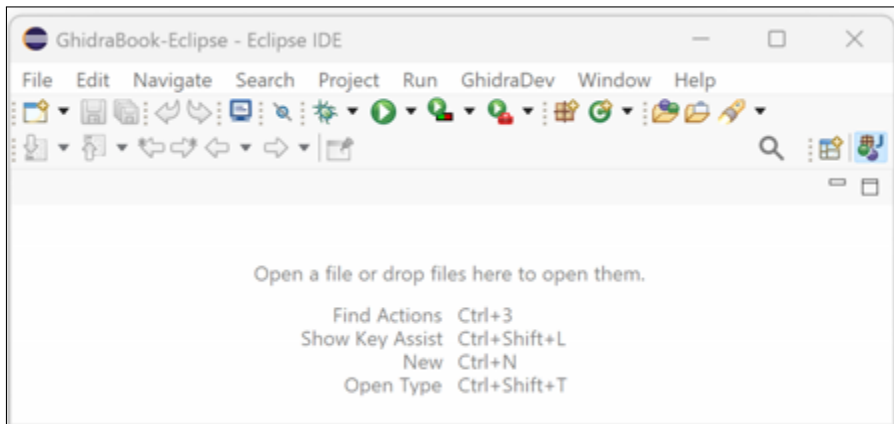


Figure 15-2: The Eclipse IDE welcome screen

The landing page for Eclipse, the Welcome to the Eclipse IDE for Java Developers workbench, includes links to numerous tutorials, documentation, and information about the Eclipse IDE and Java that should provide the necessary background support to users new to Eclipse as well as an optional refresher for experienced users.

To move ahead with Ghidra, we will focus our discussion on the use of the GhidraDev menu to augment Ghidra's existing capabilities, build new capabilities, and customize Ghidra to improve our reverse engineering workflow.

Editing Scripts with Eclipse

Once you have installed the GhidraDev plugin in Eclipse, you are ready to create new scripts or edit existing ones using the Eclipse IDE. As we migrate from using Ghidra's Script Manager to Eclipse, remember that while it is possible to launch Eclipse from Script Manager, it is possible to do so only to edit an existing script (see Figure 14-2 on page 306). If you want to edit a new script using Eclipse, you will need to first launch Eclipse and then use the GhidraDev menu to create the new script.

Let's use Eclipse to modify the first script we created in "Editing a Script: Regex Search" on page 309. Select File ► Open File from the Eclipse menu and navigate to the script *FindStringByRegex.java*. This opens the script in the Eclipse IDE, and you can begin using Eclipse's rich set of editing options. Figure 15-3 shows the first few lines of the script with the comments and imports collapsed.

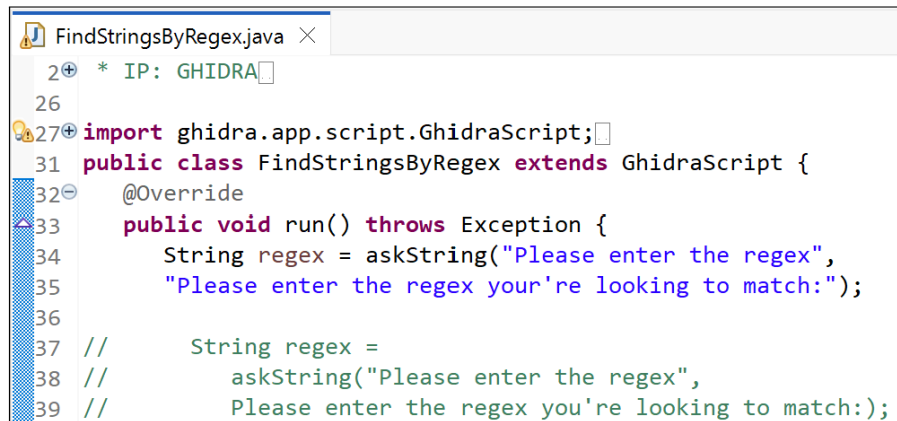


Figure 15-3: The Eclipse editor presentation of *FindStringsByRegex*

Collapsing lines is a default feature of the Eclipse IDE that could cause some confusion if you are switching between the basic editor provided by Ghidra and Eclipse. By default, only one line of comments is displayed. You can click the + icon to expand the content and display all of the comments, and then collapse them with the - icon. The same is true on line 26, with the import statements. Hovering over the icon for any collapsed section displays the hidden content in a pop-up window.

Before you can start building examples that expand Ghidra's capabilities, you need to understand the GhidraDev menu and the Eclipse IDE. Let's shift our focus back to the GhidraDev menu and investigate the various options and how they can be used in context.

The GhidraDev Menu

Figure 15-4 shows the expanded GhidraDev menu, which includes five options that you can use to control your development environment and work with files.

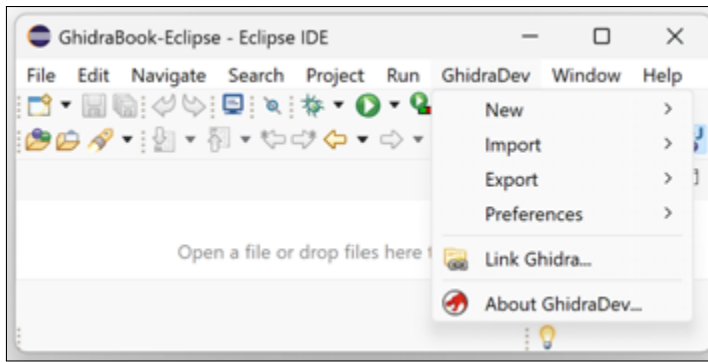


Figure 15-4: The GhidraDev menu options

In this chapter, we focus on developing in Java, although Python is an option in several of the windows.

The GhidraDev New Menu

The GhidraDev ► New menu provides you with three submenu options, as shown in Figure 15-5.

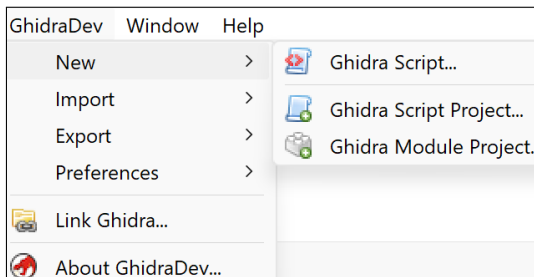


Figure 15-5: The GhidraDev ► New submenu

All three of the options launch wizards that guide you through an associated creation process. We start with the simplest option, which is to create a new Ghidra script. This is an alternative path to creating scripts from the one discussed in Chapter 14.

Creating a Script

Creating a new script using GhidraDev ► New ► Ghidra Script results in a dialog that allows you to enter information about your new script. Figure 15-6 shows an example of the dialog populated with content. In addition to the directory and file information, the dialog collects the same metadata that we manually entered into our script files in the Script Manager’s basic editor.

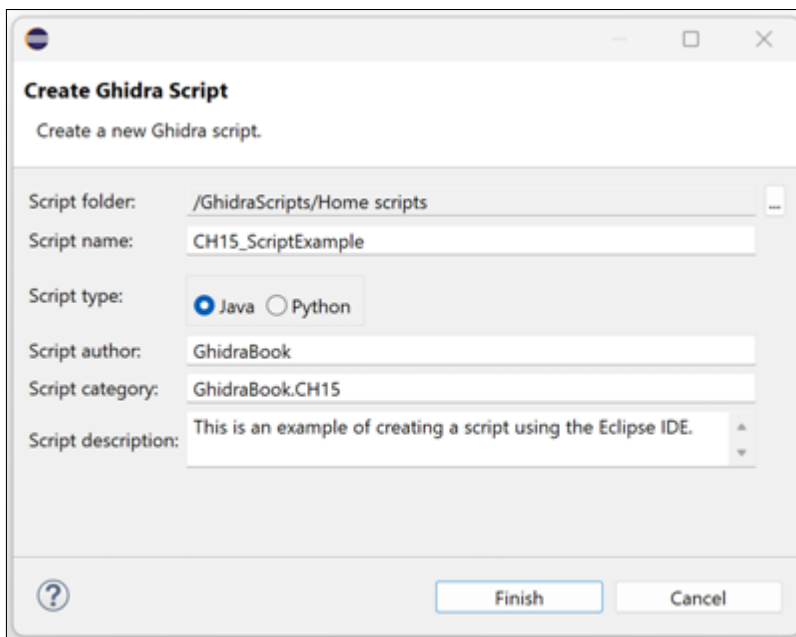


Figure 15-6: The Create Ghidra Script dialog

The Finish button at the bottom of the dialog produces the script template shown in Figure 15-7. The metadata entered in Figure 15-6 is included in the comment section at the top of the script. This content is in the same format as the metadata we saw in Chapter 14 (see the top of Figure 14-4 on page 308). When you edit this script in Eclipse, the task tag (the clipboard icon, seen on the left side of line 14 in Figure 15-7) associated with each TODO item in the script identifies locations where work needs to be done. You can delete and insert task tags at will.

```

1 //This is an example of creating a script using the Eclipse IDE.
2 //@author GhidraBook
3 //@category GhidraBook.CH15
4 //@keybinding
5 //@menupath
6 //@toolbar
7
8 import ghidra.app.script.GhidraScript;
9
10 public class CH15_ScriptExample extends GhidraScript {
11
12     @Override
13     protected void run() throws Exception {
14         //TODO: Add script code here
15     }
16 }

```

Figure 15-7: GhidraDev ► NewScript script shell

Eclipse does not preload your script with the list of import statements like the Ghidra basic editor does. Not to worry. Eclipse helps you manage your import statements by letting you know when you use something that requires an associated import statement. For example, if we replace the TODO comment in Figure 15-7 with the declaration of a Java ArrayList, Eclipse adds an error tag to the line and underlines ArrayList in red. Hovering over the error tag or ArrayList displays a pop-up window suggesting quick fixes to solve the issue, as shown in Figure 15-8.



Figure 15-8: The Eclipse Quick Fix options

Choosing the first option in the suggestion list instructs Eclipse to add the selected import statement to the script, as shown in Figure 15-9.

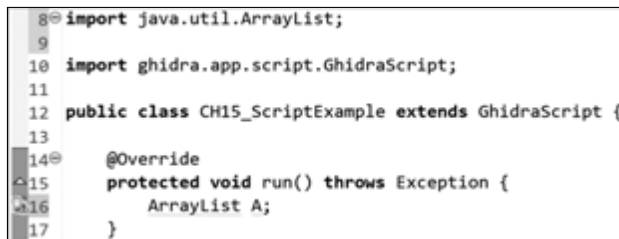


Figure 15-9: Eclipse after the Quick Fix import is applied

While it was helpful to have the list of potential import statements loaded when creating a new script in the CodeBrowser Script Manager, it is not essential in Eclipse.

Creating a Script Project

The second option in the GhidraDev ► New menu creates a new script project, as shown in Figure 15-10. We name our first script project *CH15_ProjectExample_linked* and place it in the default directory we have set up for Eclipse. The Create run configuration checkbox allows you to create a *run configuration*, which provides Eclipse with the necessary information (command line arguments, directory paths, and so on) to launch Ghidra and allows us to use Eclipse to run and debug the script in Ghidra. Leave this checkbox in its default state, selected. Click Finish to complete creation of the script using the default format, which links the script project to your home directory.

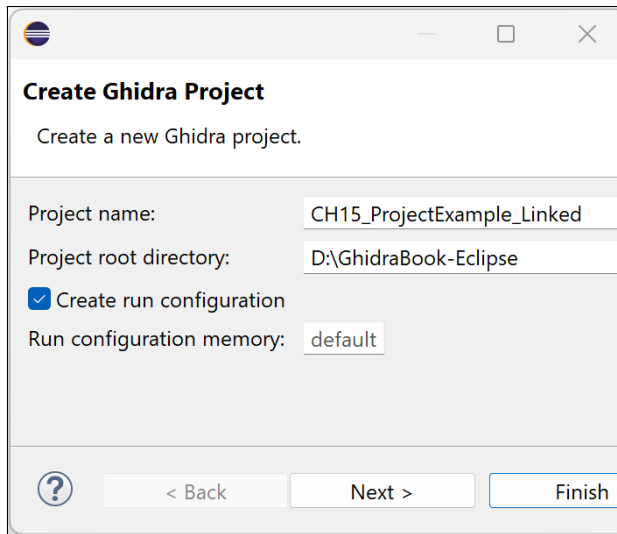


Figure 15-10: The Eclipse Ghidra Script Project dialog

We will create a second script project, *CH15_ProjectExample*, and this time choose the Next button. Choosing Next yields the dialog with two Link options that are set by default (hence the *_linked* extension on our first project name). The first option creates a link to your home script directory. The second allows you to link to the Ghidra installation script directories. *Link*, in this case, is a way of saying that folders representing your home script directory and/or Ghidra's own script directories will be added to your new project, making any script in those directories easily accessible to you while working on your project.

The results of selecting or deselecting these options and then clicking the Finish button will become clear later in the chapter when we discuss the Eclipse Package Explorer. For this second script project, deselect both checkboxes, as shown in Figure 15-11.

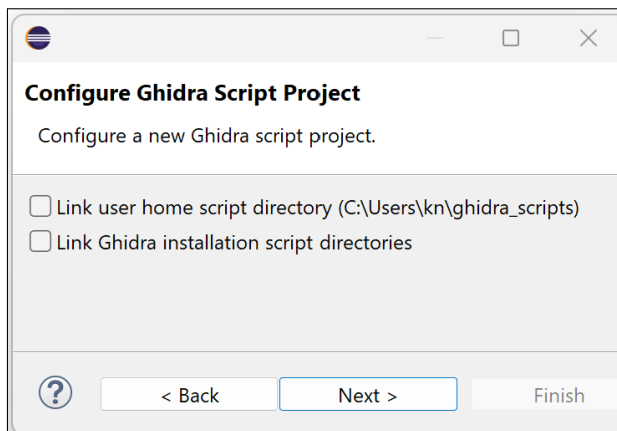


Figure 15-11: The Eclipse configuration options for script projects

Creating a Module Project

The final option in the GhidraDev ► New menu creates a Ghidra module project. Not to be confused with a Ghidra module (for example, the analyzer or loader), a *Ghidra module project* aggregates code for a new Ghidra module with associated help files, documentation, and other resources, such as icons. In addition, it allows you some control over how your new module interacts with the other modules within Ghidra. We demonstrate Ghidra modules in context in this and future chapters.

NOTE

Both types of Ghidra modules are distinct from Java modules, which were introduced in Java 9 as a means to encapsulate packages and other resources and provide the capability to keep packages private or choose to share individual packages with select other modules, effectively allowing modules to control the sharing of services.

Choosing New ► Ghidra Module Project displays the dialog shown in Figure 15-12. This dialog should be familiar because it is exactly the same as the Script Project dialog. We name our new project *CH15_ModuleExample* to make it easy to identify in the Package Explorer.

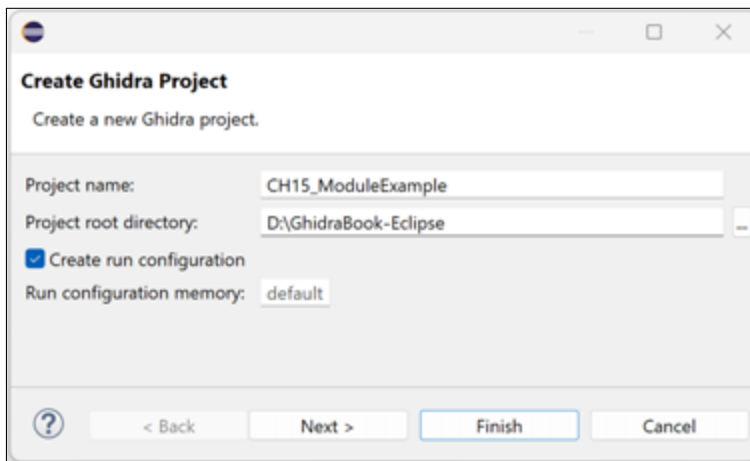


Figure 15-12: The Eclipse Module Project dialog

Clicking Next at this point allows you to base your module on existing Ghidra templates, as shown in Figure 15-13. By default, all of the options are selected. You can change this to include none, some, or all of the templates, depending on your development goals. Any of the options you choose will be grouped together in a project within the Package Explorer. In our case, we have deselected all of the options.

While most of the selections will produce an associated source code template with task tags, there are two exceptions. First, if you do not select any of the module templates, you will not have a template file. In addition, the processor module does not produce a template file but does generate other supporting content. (Processor modules are discussed in Chapter 18.)

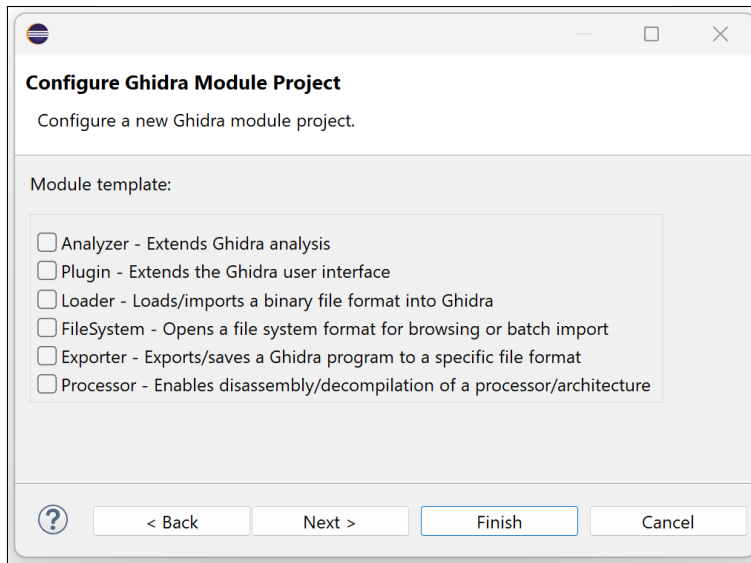


Figure 15-13: The template options for Ghidra module projects

Now that you know how to create Ghidra scripts, script projects, and module projects, let's shift our focus to the Eclipse Package Explorer to better understand how we can work with our new creations.

NOTE

Package Explorer has been around for a while, whereas modules are a more recent addition to Java. In the default configuration, Package Explorer can be thought of as a project explorer for the Java project you have created or imported.

The Package Explorer

Eclipse's Package Explorer is the gateway to the Ghidra files you need to complete your Ghidra extension. Here, we present the hierarchical organization and then drill down into examples of Ghidra projects and modules created through the GhidraDev menu. Figure 15-14 displays a sample Eclipse Package Explorer window containing the items we created earlier in this chapter, as well as a few others we created to demonstrate the effect of various options on the resulting Package Explorer contents.

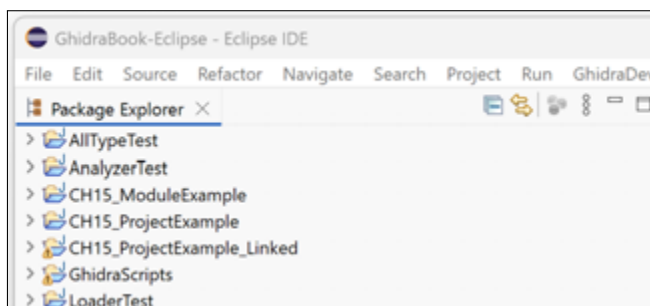


Figure 15-14: The Package Explorer populated with example modules and projects

We start by looking at the two script projects. *CH15_ProjectExample_linked* is the script project that we created with both link options checked. Immediately above it, we see a similar project, *CH15_ProjectExample*, but in this case, neither link option was checked (refer to Figure 15-11).

Figure 15-15 shows a partially expanded Package Explorer entry for *CH15_ProjectExample*. You can see the following four components included in this script project:

JUnit4 This is an open source unit-testing framework for Java. More information about it is available at <https://junit.org/junit4/index.html>.

JRE System Library This is the Java Runtime Environment System Library.

Referenced Libraries These are referenced libraries that are not part of the JRE System Library, but are part of our Ghidra installation.

Ghidra This is the directory for your current Ghidra installation. We have expanded this directory so that you can see the familiar file structure introduced in Chapter 3 (see Figure 3-1) and used throughout this book.

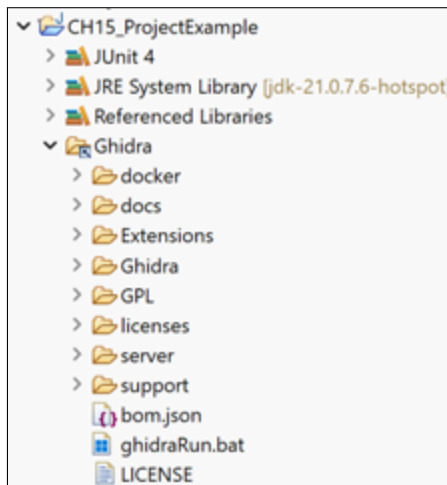


Figure 15-15: The Package Explorer script project entries without links

Compare the contents of Figure 15-15 with the expanded contents from *CH15_ProjectExample_linked* shown in Figure 15-16. For this script project, we selected both link options. Linking the user home script directory results in the *Home scripts* entry in the project hierarchy and provides us with easy access to the scripts we have previously written to use as examples or modify.

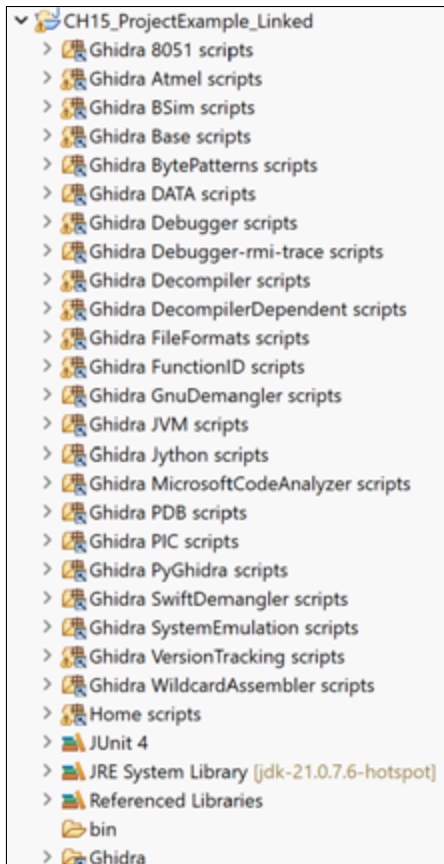


Figure 15-16: The Package Explorer script project entries with links

Linking Ghidra installation script directories results in all of the folders in Figure 15-16 that start with *Ghidra* and end with *scripts*. Each one of these corresponds to a script directory within the *Ghidra/Features* directory in your Ghidra installation. Expanding any of these folders provides access to the source code for each of the scripts included in your Ghidra installation.

Like the home scripts, these can serve as examples to modify or use as a base for creating new scripts. While you are not permitted to overwrite these scripts from within the Ghidra Script Manager basic editor, you can edit them in Eclipse and other editors outside of the Ghidra Project environment. When you have finished creating or editing a new script, you can save it in the appropriate script directory within your script project, and it will be available to use the next time you open the Ghidra Script Manager.

NOTE

The location of a script within Eclipse Package Explorer (and Ghidra/Features directory) does not necessarily coincide with the organization of the scripts within the Script Manager. This is to be expected, as the scripts within the Ghidra/Features directory are organized into folders that share a common functionality. The organization of scripts within the Ghidra Script Manager is based on the category metadata within each script.

Now that we have looked at scripts within the Eclipse Package Explorer, let's see how the Ghidra module project we built is represented. Figure 15-17 shows the partially expanded content of our project in the Package Explorer.

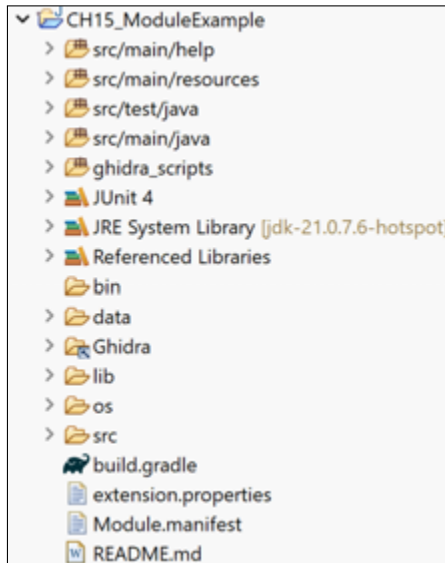


Figure 15-17: The Package Explorer hierarchy for CH15_ModuleExample

Module projects include the following new elements:

src/main/java This is the location of the source code. If you create a module type that has a template available, the associated *.java* files are placed in this directory.

src/main/help When you create or extend content, you have the opportunity to add useful information to Ghidra Help using the files and information in this directory.

src/main/resources As with many of the other entries in the *src/main* directory, expanding this content will lead you to a *README.txt* file that provides additional information about the purpose of the directory and how it should be used. For example, the *src/main/resources/images/README.txt* file lets you know that the directory is the location in which any image or icon files associated with the module should be stored.

ghidra_scripts This is where the Ghidra scripts that are specific to this module are stored.

data This folder contains any independent data files that are used with this module. (While not prohibited from use with other module types, this folder is primarily used with processor modules and is discussed in Chapter 18.)

lib Any *.jar* files required by the module should be stored in this folder.

os There are *linux_x86_64*, *mac_x86_64*, and *win_x86_64* subdirectories to hold any native binaries that the module may depend upon.

src This directory is used to store unit test cases.

build.gradle Gradle is an open source build system. This file is used to build your Ghidra extension.

extension.properties This file stores metadata about the extension.

Module.manifest You can enter information about the module such as configuration information in this file.

ARE WE BUILDING THAT SCRIPT AGAIN?

In Chapter 14, we presented a toy example within the Ghidra Script Manager environment where we modified the existing script *CountAndSaveStrings* and used it to build a new script called *FindStringsByRegex*. The following steps do the same task within the Eclipse IDE:

1. Search for *CountAndSaveStrings.java* in Eclipse (CTRL-SHIFT-R in Windows).
2. Double-click to open the file in the Eclipse editor.
3. Replace the existing class and comments with the new class and comments.
4. Save the file (*EclipseFindStringByRegex.java*) in the recommended *ghidra_scripts* directory.
5. Run the new script from the Script Manager window in Ghidra.

You can launch Ghidra manually to get access to the Script Manager window. Alternatively, you can select the Run As option in the Eclipse IDE, which will show the dialog in Figure 15-18. The first option launches Ghidra for you. The second option launches a non-GUI version of Ghidra, which is the topic of Chapter 16.

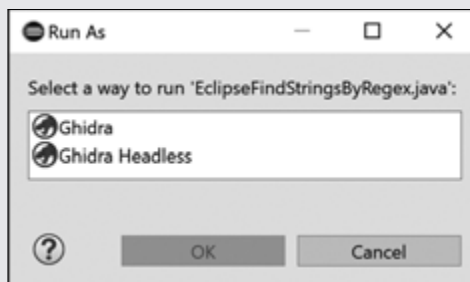


Figure 15-18: The Eclipse Run As options

Once Ghidra has been launched, you can run your script from the Script Manager and edit it using Eclipse.

You may have noticed in Figure 15-14 that we created additional *Test* modules (*AnalyzerTest*, *AllTypeTest*, and *LoaderTest*). Each was created using a different combination of Module Template options (see Figure 15-13), which results in a different set of files being instantiated for each project. When you use these templates as a starting point for your projects, it's useful to know just how much work Eclipse and Ghidra have done for you—and how much work is left for you to complete.

Let's begin by looking in the *AnalyzerTest* directory we created to explore an analyzer template. Expand the *src/main/java* directory to find a file called *AnalyzerTestAnalyzer.java*. The name is a concatenation of the module name (*AnalyzerTest*) and the template type (*Analyzer*). Double-click this file to open it in the editor and see the code shown in Figure 15-19.

```

2* * IP: GHIDRA
16 package analyzertest;
17
18*import ghidra.app.services.AbstractAnalyzer;
26
27*/**
28 * Provide class-level documentation that describes what this analyzer does.
29 */
30 public class AnalyzerTestAnalyzer extends AbstractAnalyzer {
31
32* public AnalyzerTestAnalyzer() {}
37
38
40* public boolean getDefaultEnablement(Program program) {}
45
46
48* public boolean canAnalyze(Program program) {}
54
55
57* public void registerOptions(Options options, Program program) {}
63
64
66* public boolean added(Program program, AddressSetView set, TaskMonitor monitor, MessageLog log) {}
73
74 }

```

Figure 15-19: The default analyzer template for a module (with imports and methods collapsed)

As in the script templates shown earlier in the chapter, the Eclipse IDE provides task tags with associated comments to guide us through building our analyzer as well as the options to expand and collapse content. The *LoaderTest* module contains the template for building a loader, a topic we discuss further in Chapter 17. The remaining module, *AllTypeTest*, is the default module that results when you bypass the module template options. This populates the *src/main/java* directory with each type of template, as shown in Figure 15-20.

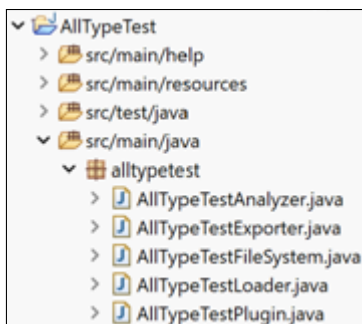


Figure 15-20: The sample default module source code contents

Now that we have seen how helpful Ghidra and Eclipse can be when we create new modules, let's use this information to build a new analyzer.

Example: Building a Ghidra Analyzer Module

With the Eclipse integration basics behind us, let's walk through building a simple Ghidra analyzer to identify potential return-oriented programming (ROP) gadgets in our listing. We will use a simplified software development process for demonstration purposes. Our process includes the following steps:

1. Define the problem.
2. Create the Eclipse module.
3. Build the analyzer.
4. Add the analyzer to our Ghidra installation.
5. Test the analyzer from our Ghidra installation.

WHAT'S A ROP GADGET, AND WHY DO WE CARE?

For those unfamiliar with exploit development, ROP stands for *return-oriented programming*. One software security mitigation that aims to defeat raw shellcode injection is to ensure that no memory region that is writable is, at the same time, also executable. Such mitigations are often referred to as *NoneExecutable (NX)* or *Data Execution Prevention (DEP)* because they make it impossible to simultaneously inject shellcode into memory (which must be writable) and then transfer control to that shellcode (in which case it must be executable).

ROP techniques aim to hijack a program's stack (often through a stack-based buffer overflow) to place a carefully crafted sequence of return addresses and data into the stack. At some point after the overflow, the program begins using the attacker-supplied return addresses rather than return addresses placed on the stack by normal program execution. The return addresses the attacker places on the stack point to program memory locations that already contain code as a result of normal program and library loading operations. Because the original

author of the exploited program did not design the program to do the attacker's work for them, the attacker often needs to pick and choose small portions of this existing code to sequence together.

A *ROP gadget* is a single one of these code fragments, and the sequencing mechanism often relies on the gadget terminating in a return (hence return-oriented) instruction, which retrieves an address from the now attacker-controlled stack to transfer control to the next gadget. A gadget often performs a very simple task, such as loading a register from the stack. The following simple gadget could be used to initialize RAX on an x86-64 system:

```
POP RAX ; pop the next item on the attacker-controlled stack into RAX
RET     ; transfer control to the address contained in the next stack item
```

Because every exploitable program is different, attackers can't depend on a specific set of gadgets being present in any given binary. Automated gadget finders are tools that search a binary for instruction sequences that may be used as gadgets and present these gadgets to the attacker, who must decide which ones are useful in crafting their attack. The most sophisticated gadget finders infer the semantics of a gadget and automatically sequence gadgets to perform a specified action, saving the attacker the trouble of doing it themselves.

Step 1: Defining the Problem

Our task is to design and develop an instruction analyzer that will identify simple ROP gadgets within a binary. The analyzer needs to be added to Ghidra and be available as a documented, selectable analyzer in the Ghidra Analyzer menu.

Step 2: Creating the Eclipse Module

We use GhidraDev ► New ► Ghidra Module Project to create a module called *SimpleROP* using the analyzer module template. This creates a file called *SimpleROPAnalyzer.java* in the *src/main/java* folder within the *SimpleROP* module. Figure 15-21 shows the resulting Package Explorer view.

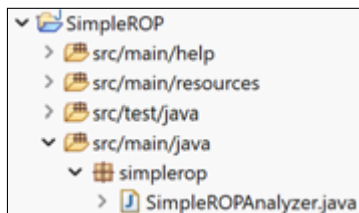


Figure 15-21: Package Explorer
src/main entries for SimpleROP

Step 3: Building the Analyzer

Figure 15-22 shows a portion of the generated *SimpleROPAnalyzer.java* code. We have collapsed the functions so we can see all of the analyzer methods

provided. Eclipse will recommend imports if we need them as we develop our code, so we can jump right into coding the tasks we need to perform and add the recommended import statements when Eclipse detects that we need them.

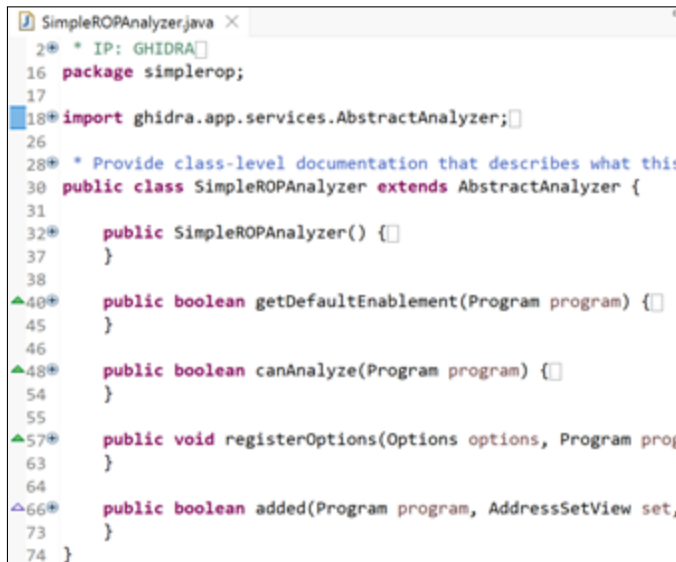


Figure 15-22: The SimpleROPAnalyzer template

The empty methods in Figure 15-22 indicate where we should start our development. We will expand the associated sections as we address each task and include the before-and-after content associated with each task. (Note that we will wrap or reformat some content for readability and minimize comments to conserve space.)

For functionality, we will rely on the following class-level declarations:

```

private int gadgetCount = 0;          // Counts the number of gadgets
private BufferedWriter outFile;       // Output file
// List of "interesting" instructions
private List<String> usefulInstructions = Arrays.asList(
    "NOP", "POP", "PUSH", "MOV", "ADD", "SUB", "MUL", "DIV", "XOR");
// List of "interesting" instructions that do not have operands
private List<String> require0Operands = Arrays.asList("NOP");
// List of "interesting" instructions that have one operand
private List<String> require1RegOperand = Arrays.asList("POP", "PUSH");
// List of "interesting" instructions for which we want the first
// parameter to be a register
private List<String> requireFirstRegOperand = Arrays.asList(
    "MOV", "ADD", "SUB", "MUL", "DIV", "XOR");
// List of "start" instructions that have ZERO operands
private List<String> startInstr0Params = Arrays.asList("RET");
// List of "start" instructions that have ONE register operand
private List<String> startInstr1RegParam = Arrays.asList("JMP", "CALL");

```

Comments associated with each declaration describe the purpose of each variable. The various `List` variables contain the instructions from which our gadgets will be composed, and they classify those instructions based on the number and type of operands they require and whether the instruction is a legal start instruction for one of our gadgets. Because our gadget construction algorithm works its way backward in memory, *start* here actually means a starting point for our algorithm. At runtime, these same start instructions would actually be the last instructions executed in a given gadget.

Step 3-1: Documenting the Class

When we expand the first task tag, we see the following task description:

```
/**
 * Provide class-level documentation that describes what this
 * analyzer does.
 */
```

We replace the existing comments with comments that describe what the analyzer does:

```
/**
 * This analyzer searches through a binary for ROP gadgets.
 * The address and contents of each gadget are written to a
 * file called inputfilename_gadgets.txt in the user's home directory.
 */
```

Step 3-2: Naming and Describing Our Analyzer

Expanding the next task tag provides us with a `TODO` comment and the line of code that we need to edit. Within the Eclipse IDE, the code to modify appears in purple font and has a name indicative of the associated task. The second task contains the following:

```
// Name the analyzer and give it a description.
public SimpleROPAnalyzer() {
    super("My Analyzer",
          "Analyzer description goes here",
          AnalyzerType.BYTE_ANALYZER);
}
```

We must replace these two strings with meaningful content and specify the analyzer's type:

```
public SimpleROPAnalyzer() {
    super("SimpleROP",
          "Search a binary for ROP gadgets",
          AnalyzerType.INSTRUCTION_ANALYZER);
}
```

To facilitate dependency resolution across analyzers, Ghidra groups analyzers into the following categories: byte, data, function, function modifiers, function signatures, and instruction. In this case, we are building an instruction analyzer.

Step 3-3: Choosing Whether to Make Our Analyzer the Default

The third task asks us to return true if the analyzer should be enabled by default:

```
public boolean getDefaultEnablement(Program program) {  
    // Return true if analyzer should be enabled by default.  
    return false;  
}
```

We do not want this analyzer enabled by default; therefore, we make no code modifications.

Step 3-4: Determining the Appropriate Input

Now we must determine whether our analyzer is compatible with the program content:

```
public boolean canAnalyze(Program program) {  
    // Examine 'program' to determine if this analyzer should analyze it.  
    // Return true if it can.  
    return false;  
}
```

Because we are building this analyzer for demonstration purposes only, we assume that the input file is compatible with our analysis and simply return true:

```
public boolean canAnalyze(Program program) {  
    return true;  
}
```

In a real implementation, we would add code to verify the compatibility of the analysis file prior to using our analyzer. For example, we might return true only after we have determined that the file is an x86 binary. You can find worked examples of this verification in most analyzers included in your Ghidra installation (at *Ghidra/Features/Base/lib/Base-src/ghidra/app/analyzers*), accessible through your module directory within Eclipse.

Step 3-5: Registering Analyzer Options

Now we can specify any special options we wish to present to users of our analyzer:

```
public void registerOptions(Options options, Program program) {  
    // If this analyzer has custom options, register them here.
```

```

        options.registerOption("Option name goes here", false, null,
                                "Option description goes here");
    }

```

In this example, we will not add any options:

```

public void registerOptions(Options options, Program program) {
}

```

Options might include user-controlled choices such as the selection of the output file, the option to annotate the listing, and so on. These options will be displayed in the Analyzer window when an individual analyzer is selected.

Step 3-6: Performing the Analysis

Now we can turn our attention to the function that gets called when our analyzer is invoked:

```

public boolean added(Program program, AddressSetView set, TaskMonitor
                    monitor, MessageLog log) throws CanceledException {
    // Perform analysis when things get added to the 'program'.
    // Return true if the analysis succeeded.
    return false;
}

```

This is the part of the module that does the actual work. The module uses four methods, detailed next:

```

//*****
// This method is called when the analyzer is invoked.
//*****
❶ public boolean added(Program program, AddressSetView set, TaskMonitor
                    monitor, MessageLog log) throws CanceledException {
    gadgetCount = 0;
    String outFileName = System.getProperty("user.home") + "/" +
                        program.getName() + "_gadgets.txt";
    monitor.setMessage("Searching for ROP Gadgets");
    try {
        outFile = new BufferedWriter(new FileWriter(outFileName));
    } catch (IOException e) { /* pass */ }
    // Iterate through each instruction in the binary.
    Listing code = program.getListing();
    InstructionIterator instructions = code.getInstructions(set, true);
    ❷ while (instructions.hasNext() && !monitor.isCancelled()) {
        Instruction inst = instructions.next();
        ❸ if (isStartInstruction(inst)) {
            // We found a "start" instruction. This will be the last
            // instruction in the potential ROP gadget, so we will try to
            // build the gadget from here.

```

```

        ArrayList<Instruction> gadgetInstructions =
            new ArrayList<Instruction>();
        gadgetInstructions.add(inst);
        Instruction prevInstr = inst.getPrevious();
        ④ buildGadget(program, monitor, prevInstr, gadgetInstructions);
    }
}
try {
    outFile.close();
} catch (IOException e) { /* pass */}
return true;
}
//*****
// This method is called recursively until it finds an instruction that
// we don't want in the ROP gadget.
//*****
private void buildGadget(Program program, TaskMonitor monitor,
                        Instruction inst,
                        ArrayList<Instruction> gadgetInstructions) {
    ⑤ if (inst == null || !isUsefulInstruction(inst) ||
        monitor.isCancelled()) {
        return;
    }
    gadgetInstructions.add(inst);
    ⑥ buildGadget(program, monitor, ⑦ inst.getPrevious(), gadgetInstructions);
    gadgetCount += 1;
    ⑧ for (int ii = gadgetInstructions.size() - 1; ii >= 0; ii--) {
        try {
            Instruction insn = gadgetInstructions.get(ii);
            if (ii == gadgetInstructions.size() - 1) {
                outFile.write(insn.getMinAddress() + ";");
            }
            outFile.write(insn.toString() + ";");
        } catch (IOException e) { /* pass */}
    }
    try {
        outFile.write("\n");
    } catch (IOException e) { /* pass */}
    // Report count to monitor every 100th gadget.
    if (gadgetCount % 100 == 0) {
        monitor.setMessage("Found " + gadgetCount + " ROP Gadgets");
    }
    gadgetInstructions.remove(gadgetInstructions.size() - 1);
}
//*****
// This method determines if an instruction is useful in the context of
// a ROP gadget.
//*****

```



```

private boolean isUsefulInstruction(Instruction inst) {
    if (!usefulInstructions.contains(inst.getMnemonicString())) {
        return false;
    }
    if (require0Operands.contains(inst.getMnemonicString())) {
        return true;
    }
    if (require1RegOperand.contains(inst.getMnemonicString()) &&
        inst.getNumOperands() == 1) {
        Object[] opObjects0 = inst.getOpObjects(0);
        for (int ii = 0; ii < opObjects0.length; ii++) {
            if (opObjects0[ii] instanceof Register) {
                return true;
            }
        }
    }
    if (requireFirstRegOperand.contains(inst.getMnemonicString()) &&
        inst.getNumOperands() >= 1) {
        Object[] opObjects0 = inst.getOpObjects(0);
        for (int ii = 0; ii < opObjects0.length; ii++) {
            if (opObjects0[ii] instanceof Register) {
                return true;
            }
        }
    }
    return false;
}

//*****
// This method determines if an instruction is the "start" of a
// potential ROP gadget.
//*****
private boolean isStartInstruction(Instruction inst) {
    if (startInstr0Params.contains(inst.getMnemonicString())) {
        return true;
    }
    if (startInstr1RegParam.contains(inst.getMnemonicString()) &&
        inst.getNumOperands() >= 1) {
        Object[] opObjects0 = inst.getOpObjects(0);
        for (int ii = 0; ii < opObjects0.length; ii++) {
            if (opObjects0[ii] instanceof Register) {
                return true;
            }
        }
    }
    return false;
}

```

Ghidra invokes an analyzer's added method ❶ to initiate analysis. Our algorithm tests every instruction ❷ in the binary to determine whether the instruction is a valid "start" point ❸ for our gadget builder. Each time it finds a valid start instruction it invokes our gadget creation function, `buildGadget` ❹. Gadget creation is a recursive ❺ walk backward ❻ through the instruction list that continues for as long as an instruction is considered useful ❼ to us. Finally, we print each gadget by iterating over its instructions ❽ as it is completed.

Step 4: Testing the Analyzer in Eclipse

During the development process, it is common to test and modify code frequently. As you are building your analyzer, you can test its functionality within Eclipse by using the Run As option and choosing Ghidra. This opens Ghidra with the current version of the module temporarily installed. If you receive unexpected results, you can edit the file within Eclipse and retest until you are satisfied with your result. Using this method to test your code within Eclipse can be a great time-saver during the development process.

Step 5: Adding the Analyzer to Ghidra

To add this analyzer to our Ghidra installation, we need to export our module from Eclipse and then install the extension in Ghidra. Exporting is accomplished by selecting GhidraDev ► Export ► Ghidra Module Extension, choosing the module, and clicking Next. In the next window, select the Gradle Wrapper option shown in Figure 15-23 if you do not have a local Gradle installation. (Note that you will need an internet connection for the wrapper to reach out to *gradle.org*.) Click Finish to complete the export process.

If this is your first time exporting the module, Eclipse will add a *dist* directory to your module and save a *.zip* file of the exported content to the folder.

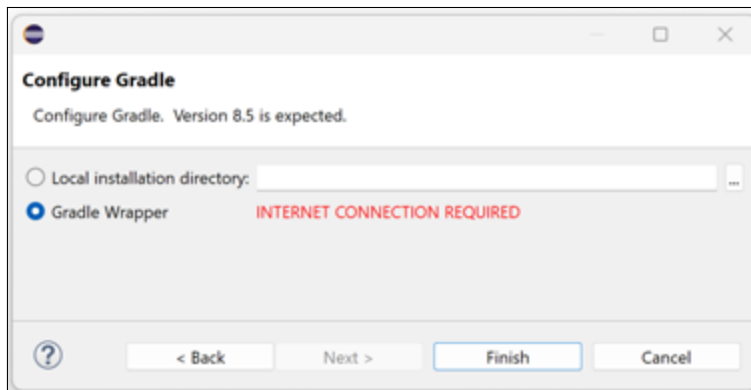


Figure 15-23: The Configure Gradle dialog

In the Ghidra Project window, add the new analyzer by selecting File ► Install Extensions. You will see a window similar to that shown in Figure 15-24, listing all of the existing extensions that have not been installed.

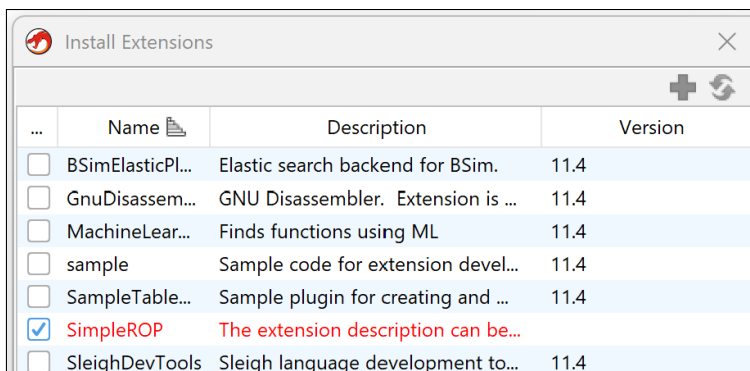


Figure 15-24: The Install Extensions window

Add the new analyzer, *SimpleROP*, by selecting the + icon at the top right and navigating to the newly created *.zip* file in the associated *dist* directory. Once the analyzer appears in the list, we can select it and click OK (not shown). Restart Ghidra to use the new functionality from the Analysis menu.

Step 6: Testing the Analyzer in Ghidra

We performed limited testing to demonstrate the analyzer's functionality. *SimpleROP* passed acceptance testing, as it met the following criteria:

1. (Pass) *SimpleROP* appears in the Analysis Options in the Code Browser ► Analysis menu.
2. (Pass) The description of *SimpleROP* appears in the Analysis Options description window when selected.

You can see that test cases 1 and 2 passed in Figure 15-25. Had we chosen to register and program associated options, they would appear in the Options panel on the right side of the window.

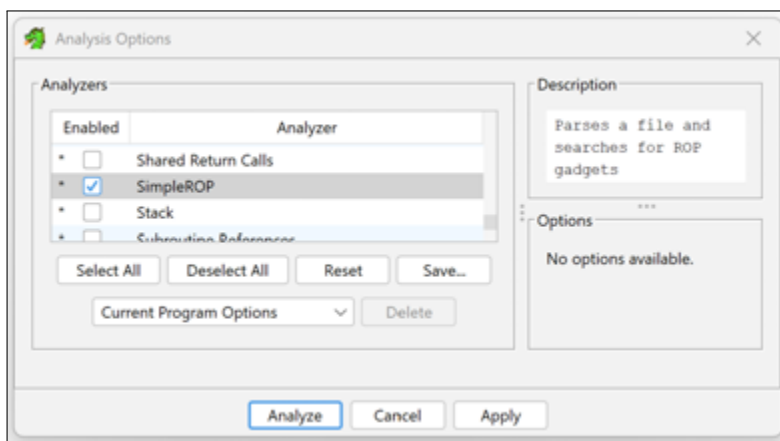
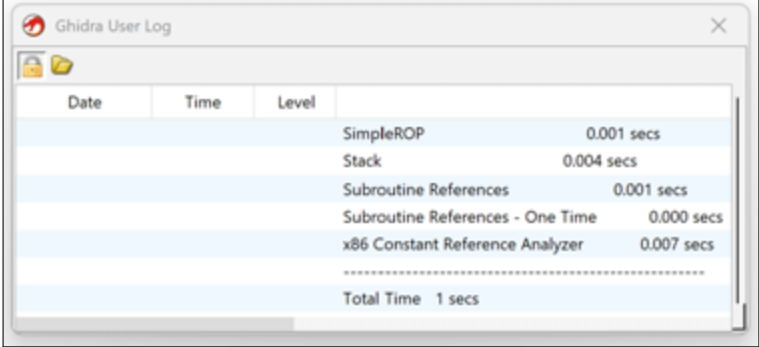


Figure 15-25: The Analysis Options window

3. (Pass) *SimpleROP* executes when selected as a single shot analyzer and as part of auto analysis. Running *SimpleROP* on an unanalyzed file would not yield any results, as INSTRUCTION_ANALYZER extensions require instructions to have been previously identified (a default part of auto analysis). When *SimpleROP* is run as part of the auto analysis, it is prioritized appropriately because of the analyzer type we assigned. Figure 15-26 shows the Ghidra Log confirmation that the *SimpleROP* analyzer ran.



Date	Time	Level	
			SimpleROP 0.001 secs
			Stack 0.004 secs
			Subroutine References 0.001 secs
			Subroutine References - One Time 0.000 secs
			x86 Constant Reference Analyzer 0.007 secs

			Total Time 1 secs

Figure 15-26: The Ghidra User Log window confirming analysis

4. (Pass) *SimpleROP* writes each gadget to a file called *fileZZZ_gadgets.txt* when analyzing *fileZZZ*. Take a look at the following excerpt from the file *call_tree_x64_static_gadgets.txt*:

```

00400412;ADD RSP,0x8;RET;
004004ce;NOP;RET;
00400679;ADD RSP,0x8;POP RBX;POP RBP;POP R12;POP R13;POP R14;POP R15;RET;
0040067d;POP RBX;POP RBP;POP R12;POP R13;POP R14;POP R15;RET;
0040067e;POP RBP;POP R12;POP R13;POP R14;POP R15;RET;
0040067f;POP R12;POP R13;POP R14;POP R15;RET;
00400681;POP R13;POP R14;POP R15;RET;
00400683;POP R14;POP R15;RET;
00400685;POP R15;RET;
00400a8b;POP RBP;MOV EDI,0x6babd0;JMP RAX;
00400a8c;MOV EDI,0x6babd0;JMP RAX;
00400a98;POP RBP;RET;

```

This excerpt shows that many of the gadgets are taken from the portion of the *call_tree_x64_static* listing shown in Figure 15-27.

LAB_00400672		
00400672	MOV	qword ptr
00400679	ADD	RSP,0x8
0040067d	POP	RBX
0040067e	POP	RBP
0040067f	POP	R12
00400681	POP	R13
00400683	POP	R14
00400685	POP	R15
00400687	RET	

Figure 15-27: A CodeBrowser listing of *call_tree_x64_static*

This example shows how straightforward it can be to extend Ghidra’s capabilities with your own analyzer when you need functionality beyond what the default analyzers provide. With this approach, you can tailor Ghidra to handle unique tasks that match your specific analysis goals.

Summary

In Chapter 14, we introduced scripting as a means of extending Ghidra’s capabilities. In this chapter, we introduced Ghidra extension modules along with Ghidra’s Eclipse integration capabilities. While Eclipse is not your only option for editing Ghidra extensions, the integration of Ghidra and the Eclipse IDE provides an incredibly powerful environment for developers extending Ghidra’s capabilities. The development wizards and templates lower the bar for authoring extensions as they present coders with a guided approach to modifying existing content and building new extensions. In Chapter 16, we take a look at headless Ghidra, an option that appeared in Figure 15-18. Subsequent chapters build on the integration of Ghidra and the Eclipse IDE to further extend Ghidra’s capabilities and provide a solid foundation for making Ghidra into the optimal tool for your reverse engineering workflow.

16

RUNNING GHIDRA IN HEADLESS MODE



In earlier chapters, we focused on exploring a single file within a single project, facilitated by the Ghidra GUI. In addition to the GUI, Ghidra has a command line interface called the *Ghidra headless analyzer*. The headless analyzer provides some of the same capabilities as the GUI, including the ability to work with projects and files, but it's better suited for batch processing and scripted control. In this chapter, we discuss Ghidra's headless mode and how it can help you perform repetitive tasks across a larger number of files. We start with a familiar example and then expand our discussion to more complex options.

Getting Started with the Headless Analyzer

To gain experience using Ghidra's headless analyzer's command line interface, let's start by recalling our first use of Ghidra in Chapter 4. We successfully accomplished the following steps:

1. Launch Ghidra.
2. Create a new Ghidra project.
3. Identify a location for the project.
4. Import a file to the project.
5. Auto analyze the file.
6. Save and exit.

Let's replicate these tasks using the Ghidra headless analyzer's command line interface. You can find the headless analyzer (*analyzeHeadless* or *analyzeHeadless.bat*) as well as a helpful file called *analyzeHeadlessREADME.html* in the *support* directory of your Ghidra installation. To simplify filepaths, we have temporarily placed the file *global_array_demo_x64* in the same directory.

We will first identify the commands and parameters needed for each of the individual tasks and then put them all together to accomplish our goal. While it hasn't made a significant difference in previous chapters, there are more distinctions between the three Ghidra platforms when we are operating from the command line. In our examples, we use the Windows installation and make note of significant differences on other platforms.

TO SLASH OR BACKSLASH?

A major difference among the operating system platforms that support Ghidra is the manner in which they identify filesystem paths. While the syntax is consistent, different platforms use different directory separators. Windows uses a backward slash, whereas Linux and macOS use a forward slash. A path looks like this in Windows:

```
D:\GhidraProjects\ch16\demo_stackframe_32
```

And it looks like this in Linux and macOS:

```
/GhidraProjects/ch16/demo_stackframe_32
```

This syntax can be even more confusing for Windows users, as forward slashes are used in URLs and command line switches (and Ghidra documentation). Operating systems recognize this issue and try to accept either, but not always in a predictable manner. For the examples in this chapter, we use the Windows convention so readers can enjoy being backward compatible with DOS.

Step 1: Launching Ghidra

To launch Ghidra, we use the `analyzeHeadless` command. We will accomplish all additional steps using select parameters and options associated with this command. Running `analyzeHeadless` without any parameters displays a usage message with the command's syntax and options, as shown in Figure 16-1.

```
Headless Analyzer Usage: analyzeHeadless
  <project_location> <project_name>[/<folder_path>]
    | ghidra://<server>[:<port>]/<repository_name>[/<folder_path>]
  [[-import [<directory>|<file>]+] | [-process <project_file>]]]
  [-prescript <ScriptName>]
  [-postscrip <ScriptName>]
  [-scriptPath "<path1>[;<path2>...]"
  [-propertiesPath "<path1>[;<path2>...]"
  [-scriptlog <path to script log file>]
  [-log <path to log file>]
  [-overwrite]
  [-recursive]
  [-readOnly]
  [-deleteproject]
  [-noanalysis]
  [-processor <languageID>]
  [-cspec <compilerSpecID>]
  [-analysisTimeoutPerFile <timeout in seconds>]
  [-keystore <KeystorePath>]
  [-connect [<userID>]]
  [-p]
  [-commit ["<comment>"]]
  [-okToDelete]
  [-max-cpu <max cpu cores to use>]
  [-librarySearchPaths <path1>[;<path2>...]]
  [-loader <desired loader name>]
  [-loader-<loader argument name> <loader argument value>]

Please refer to 'analyzeHeadlessREADME.html' for detailed usage examples and notes.
```

Figure 16-1: Headless analyzer syntax

To launch Ghidra, we need to add some of these parameters to the command.

Steps 2, 3, and 4: Creating a New Ghidra Project and Importing a File

In headless mode, Ghidra creates a project for you if the project does not already exist. If the project already exists in the specified location, Ghidra opens the existing project. As a result, we must use three parameters: the project location, the project name, and the name of the file to import. We will import `global_array_demo_x64`, which we have used in the past:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64
```

As mentioned, for simplicity in this initial example, we have placed the file in the *support* directory. Alternatively, we could specify the full path to the file on the command line.

Steps 5 and 6: Auto Analyzing the File, Saving, and Exiting

In headless mode, auto analysis and saving happen by default, so the command in step 4 accomplishes everything we want. We must use an option to *not* analyze the file (-noanalysis), and there are options available to control how the project and associated files are saved.

This completed command achieves our six objectives:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64
```

As with many console commands, you may be asking yourself, “How can I be sure that something has happened?” Your first sign of success (or failure) is the messages displayed on the console. Informational messages starting with the prefix INFO provide progress reports as the headless analyzer starts its work. Error messages start with the prefix ERROR. Listing 16-1 includes a subset of the messages, including an example error message.

```
❶ INFO HEADLESS Script Paths:
...
C:\Users\Ghidrabook\ghidra_scripts
❷ C:\ghidra_PUBLIC\Ghidra\Extensions\SimpleROP\ghidra_scripts
C:\ghidra_PUBLIC\Ghidra\Features\BytePatterns\ghidra_scripts
C:\ghidra_PUBLIC\Ghidra\Features\Base\ghidra_scripts
C:\ghidra_PUBLIC\Ghidra\Features\Decompiler\ghidra_scripts
...
INFO HEADLESS: execution starts (HeadlessAnalyzer)
INFO Opening existing project: D:\GhidraProjects\CH16 (HeadlessAnalyzer)
...
❸ ERROR Abort due to Headless analyzer error:
ghidra.framework.store.LockException:
Unable to lock project! D:\GhidraProjects\CH16 (HeadlessAnalyzer)
java.io.IOException: ghidra.framework.store.LockException:
Unable to lock project! D:\GhidraProjects\CH16
...

```

Listing 16-1: Headless analyzer with error condition

Ghidra lists the script paths used in headless mode ❶. Later in the chapter, we show how to use additional scripts with our headless commands. The extension we created in the preceding chapter, SimpleROP, is included in the script path ❷ because every extension adds a new path to the script path. The LockException ❸ is perhaps the most common error associated with the headless analyzer. The headless analyzer fails if you attempt to run it on a project that you already have open in another Ghidra instance. When this occurs, the headless analyzer is unable to lock the project for its own, exclusive use, so the command fails.

To fix the error, close any running Ghidra instance that has the *CH16* project open and run the command again. Figure 16-2 shows the tail end of the output for a successful execution of our command, which is similar to the pop-up windows that we see when analyzing files in the Ghidra GUI.

INFO -----	
ASCII Strings	0.179 secs
Apply Data Archives	0.113 secs
Call Convention ID	0.235 secs
Call-Fixup Installer	0.000 secs
Create Address Tables	0.001 secs
Create Function	0.002 secs
DWARF	0.000 secs
Data Reference	0.001 secs
Decompiler Switch Analysis	0.741 secs
Demangler GNU	0.040 secs
Disassemble Entry Points	0.024 secs
ELF Scalar Operand References	0.002 secs
Embedded Media	0.002 secs
External Entry References	0.001 secs
Function ID	0.017 secs
Function Start Search	0.014 secs
Function Start Search After Code	0.001 secs
Function Start Search After Data	0.000 secs
GCC Exception Handlers	0.037 secs
Non-Returning Functions - Discovered	0.000 secs
Non-Returning Functions - Known	0.000 secs
Reference	0.004 secs
Shared Return Calls	0.002 secs
Stack	0.008 secs
Subroutine References	0.001 secs
Subroutine References - One Time	0.000 secs
x86 Constant Reference Analyzer	0.020 secs

Total Time	1 secs

Figure 16-2: Headless analyzer results displayed to the console

To verify the results in the Ghidra GUI, open the project and confirm that the file has been loaded, as shown in Figure 16-3, and then open the file in the CodeBrowser to confirm the analysis.

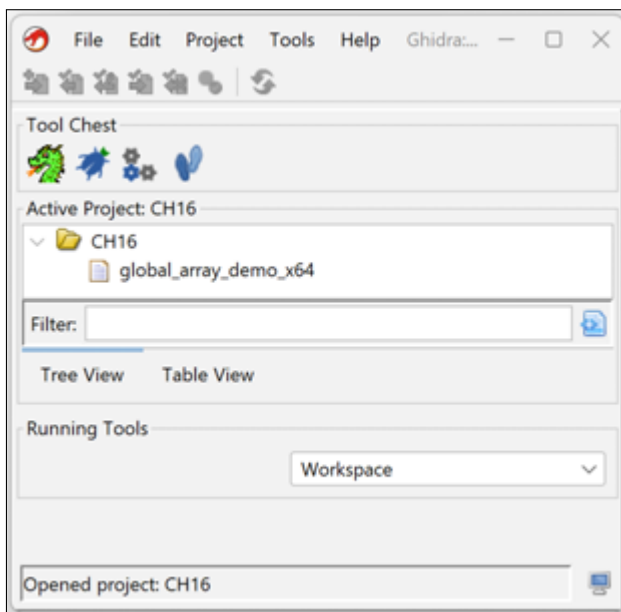


Figure 16-3: Ghidra GUI's confirmation that the project has been created and the file loaded

Now that we have replicated our earlier analysis using Ghidra in headless mode, let's investigate some situations where headless mode has an advantage over the GUI. To create a project and load and analyze all the files shown in Figure 16-4 using the Ghidra GUI, we could first create the project and then load each file individually, or select files to include in a batch import operation, as discussed in “Batch Import” on page 230. Headless Ghidra allows us to name a directory and analyze all contained files.

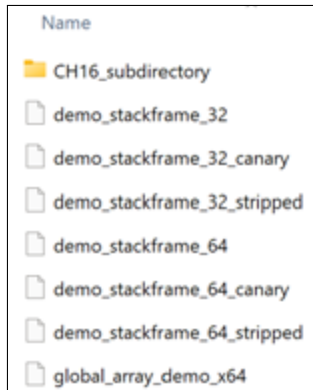


Figure 16-4: The input directory for headless Ghidra examples

The following command tells the headless analyzer to open or create a project named *CH16* in the *D:\GhidraProjects* directory and import and analyze all the files in the *D:\ch16* directory:

```
analyzeHeadless D:\GhidraProjects CH16 -import D:\ch16
```

After the command is executed, we can load the new project into the Ghidra GUI and see its associated files, as shown in Figure 16-5.

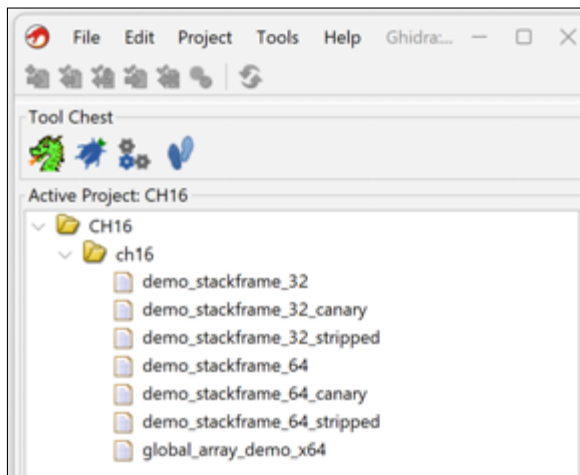


Figure 16-5: The project resulting from pointing headless Ghidra at a directory

The subdirectory *D:\ch16\CH16_subdirectory* does not appear in the project, nor do any of the files within the subdirectory. We will come back to this when we discuss additional options and parameters we can use with headless Ghidra in the following section.

Options and Parameters

We covered simple examples of using headless Ghidra to create a project, load and analyze a single file, and use batch processing to import an entire directory, but these are just the beginning of what is possible. While we will not be able to discuss all capabilities of headless Ghidra, we will provide a brief introduction to each of the options currently available.

General Options

The following are brief descriptions of additional options, with related examples, that we could use to further control what is happening in our simple examples. (The wrapped lines are indented.) We discuss common error conditions when encountered. Specialized error conditions are left as an exercise for the reader to investigate with the Ghidra Help file.

-log logfilepath

Many things can go wrong (and right) when executing Ghidra from the command line. Fortunately, Ghidra plugins provide continuous feedback as to what is happening while Ghidra is running. While this feedback is less vital in the Ghidra GUI (because you have visual cues as to what is happening), it is important in headless Ghidra.

By default, Ghidra writes a logfile to *AppData/Roaming/ghidra/ghidra_<VER>/application.log* in the user's home directory. You may select a new location by adding the **-log** option to your command line. To create a directory, *CH16-logs*, and write a logfile to *CH16-Logfile*, use the following command:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64  
-log D:\GhidraProjects\CH16-logs\CH16-Logfile
```

-noanalysis

This option instructs Ghidra not to analyze any files that you import from the command line. Opening the file *global_array_demo_x64* in the Ghidra GUI after the following statement is executed would present you with a loaded, but not analyzed, version of the file within the *CH16* project:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64  
-noanalysis
```

-overwrite

In Listing 16-1, we saw an error condition when Ghidra tried to open a project that was already open. A second common error occurs when Ghidra tries to import a file into a project and the file has already been imported. To import a new version of the file, or overwrite the existing file regardless of contents, use the `-overwrite` option:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64  
-overwrite
```

Without this option, running the previous headless command twice would result in an error during the second execution. With this option, we can rerun the command as many times as we wish.

-readOnly

To import a file without saving the file in the project, use the `-readOnly` option:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64  
-readOnly
```

If you use this option, the `-overwrite` option will be ignored (if present). This option also has meaning when used with the `-process` option rather than the `-import` command. We cover the `-process` option later in the chapter.

-deleteProject

This option instructs Ghidra not to save any project being created with the `-import` option:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64  
-deleteProject
```

This option can be used with any of the other options, but is assumed (even if omitted) when using `-readOnly`. The newly created project is deleted after analysis is complete. This option will not delete an existing project.

-recursive

By default, Ghidra does not recurse into subdirectories when asked to process an entire directory. Use this option when you want Ghidra to perform recursive directory processing (that is, to process any subdirectories it finds along the way). To demonstrate this functionality, we will point Ghidra at the same *ch16* directory we processed earlier, this time using the `-recursive` option:

```
analyzeHeadless D:\GhidraProjects CH16 -import D:\ch16 -recursive
```

Opening the project, *CH16*, after running this command results in the project structure shown in Figure 16-6.

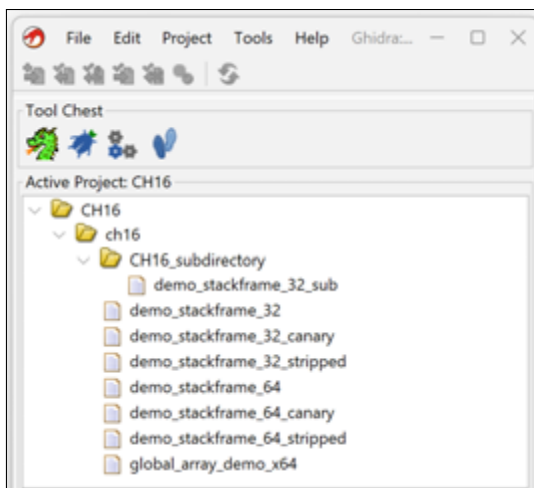


Figure 16-6: A headless Ghidra project resulting from the `-recursive` option

In contrast to Figure 16-5, the subdirectory `CH16_subdirectory` is included in the project as well as its associated file, and the directory hierarchy is retained within the project hierarchy.

WILDCARDS!

Wildcards provide an easy method of selecting multiple files for headless Ghidra without listing each one separately. In short, an asterisk (*) matches any sequence of characters, and a question mark (?) matches a single character. To load and analyze only the 32-bit files from Figure 16-7, use a wildcard as follows:

```
analyzeHeadless D:\GhidraProjects CH16 -import D:\ch16\demo_stackframe_32*
```

This creates the CH16 project and loads and analyzes all of the 32-bit files in the `ch16` directory. The resulting project is shown in Figure 16-7.

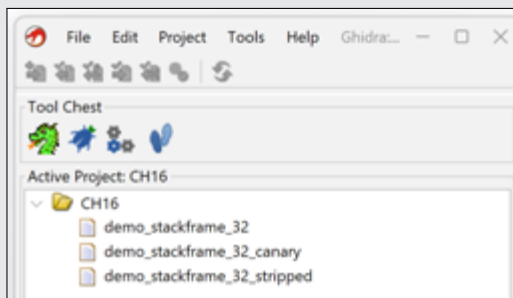


Figure 16-7: The project files resulting from the wildcard `demo_stackframe_32*`

(continued)

See *analyzeHeadlessREADME.html* for detailed information about using wildcards to specify files for import and processing. You will also see wildcards in future headless Ghidra scripting examples.

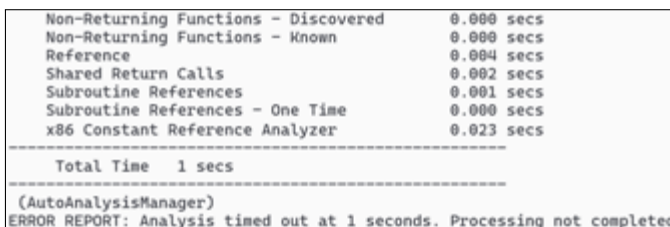
-analysisTimeoutPerFile seconds

As you have analyzed (or sat and watched Ghidra analyze) files, you may have noticed several factors that impact the analysis time, like the size of the file, whether it's statically linked, and the decompiler analysis options. Regardless of the file contents and options, you can't know in advance exactly how long it may take to analyze a file.

In headless Ghidra, particularly when you are processing a large number of files, you can use the `-analysisTimeoutPerFile` option to ensure that your task ends in a reasonable amount of time. With this option, you specify a timeout in seconds, and analysis will be interrupted should time expire. For example, our existing headless Ghidra command takes a little over one second to analyze on our system (refer to Figure 16-2). If we had *really* limited time to execute this script, the following headless command would stop analysis after one second:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64
-analysisTimeoutPerFile 1
```

This would result in the console display shown in Figure 16-8.



Non-Returning Functions - Discovered	0.000 secs
Non-Returning Functions - Known	0.000 secs
Reference	0.004 secs
Shared Return Calls	0.002 secs
Subroutine References	0.001 secs
Subroutine References - One Time	0.000 secs
x86 Constant Reference Analyzer	0.023 secs

Total Time	1 secs

(AutoAnalysisManager)	
ERROR REPORT: Analysis timed out at 1 seconds. Processing not completed	

Figure 16-8: A console warning that analysis has timed out

-processor languageID and -cspec compilerSpecID

As shown in previous examples, Ghidra is generally quite good at identifying information about a file and making import recommendations. Figure 16-9 shows a sample window containing the recommendations for a particular file. Ghidra displays this window every time you use the GUI to import a file into a project.

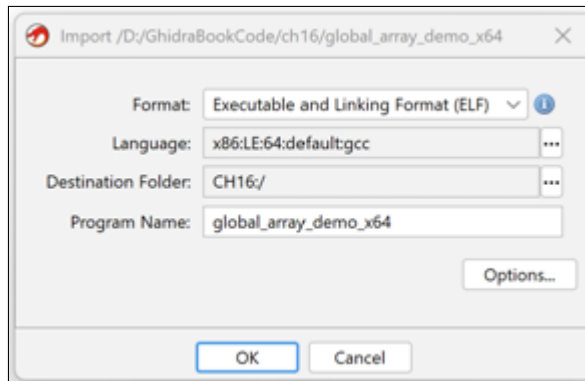


Figure 16-9: The Ghidra GUI import confirmation dialog

If you feel that you have additional insight regarding the appropriate language or compiler, you can expand the box to the right of the Language specification. This presents you with the window shown in Figure 16-10, which gives you the opportunity to select a language and compiler specification.

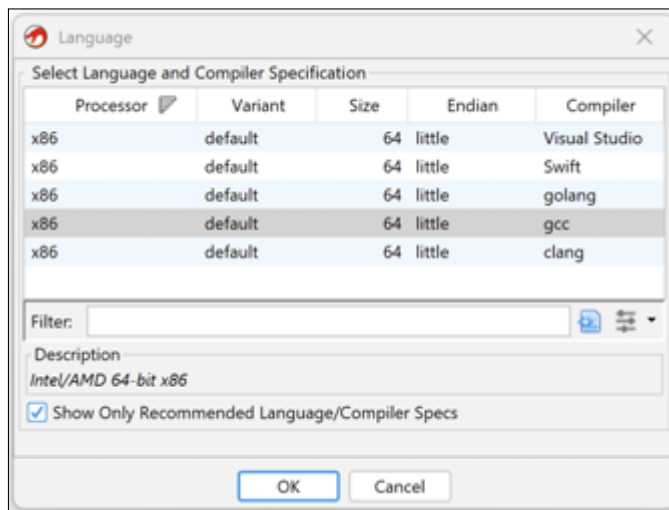


Figure 16-10: The Ghidra language/compiler specification selection window

To do the same in headless Ghidra, use the `-cspec` and/or `-processor` options, as shown next:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64
-processor "x86:LE:64:default" -cspec "gcc"
```

You cannot use the `-cspec` option without using the `-processor` option, but you can use the `-processor` option without the `-cspec` option, in which case Ghidra will choose the default compiler associated with the processor.

-loader *loadername*

The `-loader` option can be the most complex of the headless Ghidra options. The *loadername* argument specifies the particular Ghidra loader module (discussed in Chapter 17) to use to import a new file into the named project. Sample loader names include `PeLoader`, `ElfLoader`, and `MachOloader`. Each loader module may recognize additional command line arguments of its own. These additional arguments are discussed in <support/analyzeHeadlessREADME.html>.

-max-cpu *number*

This option allows you to put an upper limit on the number of processor (CPU) cores used to process the headless Ghidra command:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64
-max-cpu 5
```

The option requires an integer value as an argument. If the value is less than 1, the maximum number of cores is set to 1.

Server Options

Some commands are relevant only when interacting with a Ghidra Server. As this is not the focus of this book, we will mention these commands only briefly. If you need additional information about these commands, we encourage you to explore the <analyzeheadlessREADME.html> file.

ghidra://server[:*port*]/repository_name[/*folder_path*]

The previous examples have all specified a project location or project name. This alternative allows you to specify a Ghidra Server repository and optional folder path.

-p

With Ghidra Server, this option forces a password prompt via the console.

-connect [*userID*]

This option provides a *userID* to override the default *userID* when connecting to a Ghidra Server.

-keystore *path*

This option allows you to specify a private keystore file when using PKI or SSH authentication.

-commit ["*comment*"]

While **commit** is enabled by default, this option allows you to associate a comment with a commit.

Script Options

Perhaps the most powerful applications for headless Ghidra are associated with Ghidra's scripting abilities. Chapters 14 and 15 both demonstrated how to create scripts with the Ghidra GUI. After we present script options, we will demonstrate how powerful headless Ghidra can be in a scripting context.

-process [*project_file*]

This option processes select files (as opposed to importing them). If you do not specify a file, Ghidra will process all files in the project folder. It will also analyze all specified files unless you use the **-noanalysis** option. Ghidra accepts two wildcard characters (* and ?) for the **-process** option in order to simplify the selection of multiple files. For this option, unlike with the **-import** option, you must name Ghidra imported project files, *not* local filesystem files, so you need to quote any filenames that contain these wildcards in order to prevent your shell from expanding them prematurely.

-scriptPath "*path1*;*path2*..."

By default, headless Ghidra includes many default script paths as well as script paths for imported extensions, as seen in Listing 16-1. To extend the list of paths that Ghidra searches for available scripts, use the **-scriptPath** option, which requires a quoted path list argument. Within the quotes, separate multiple paths using a semicolon. Ghidra recognizes two special prefix designators in path components: `$GHIDRA_HOME` and `$USER_HOME`. `$GHIDRA_HOME` refers to the Ghidra installation directory, and `$USER_HOME` refers to the user's home directory. Note that these are not environment variables, and that your command shell may require you to escape the leading `$` character in order for it to be passed to Ghidra. The following example adds the *D:\GhidraScripts* directory to the script path:

```
analyzeHeadless D:\GhidraProjects CH16 -import global_array_demo_x64
      -scriptPath "D:\GhidraScripts"
```

After you run the command, the new script directory, *D:\GhidraScripts*, is included in the script path:

```
INFO  HEADLESS Script Paths:
      D:\GhidraScripts
      C:\Users\Ghidrabook\ghidra_scripts
      D:\ghidra_PUBLIC\Ghidra\Extensions\SimpleROP\ghidra_scripts
      D:\ghidra_PUBLIC\Ghidra\Features\Base\ghidra_scripts
      ...
INFO  HEADLESS: execution starts (HeadlessAnalyzer)
```

-preScript

This option names a script to be run before analysis. The script may contain an optional list of arguments.

-postScript

This option names a script to be run after analysis. The script may contain an optional list of arguments.

-propertiesPath

This option specifies the path to any property files associated with a script. Property files provide input to scripts that are run in headless mode. Examples of scripts and their associated property files are included in the headless analyzer documentation.

-okToDelete

As scripts can do whatever their creators intend, it is possible for a script to delete (or try to delete) files within a Ghidra project. To prevent this occurrence as an undesired side effect, headless Ghidra will not allow deletion of files by a script unless the `-okToDelete` option is included when the script is invoked. Note: This parameter is not required when running in `-import` mode.

Writing Scripts

Now that you understand the basic components of a headless Ghidra command, let's build some scripts to run from the command line.

Example 1: Headless ROP Gadget Identification

Recall the SimpleROP analyzer we created in Chapter 15. We wrote the module using the Eclipse IDE and then imported the extension into Ghidra so that we could run it on any file we imported. Now say we want to point SimpleROP at a directory and have it identify ROP gadgets in every file (or select files) in the directory. In addition to the SimpleROP output file with ROP gadgets for each existing binary, we also want a summary file that lists each file and the number of identified ROP gadgets in each.

Running SimpleROP through the Ghidra GUI would introduce a time penalty for actions like opening and closing the CodeBrowser to display each file in the listing window, and so on. Luckily, we do not need to see any of the files in the CodeBrowser window to accomplish our new goal, so we can write a script to find the gadgets independently of the GUI. This is exactly the kind of use case appropriate for headless Ghidra.

While we could modify SimpleROP's functionality to accomplish our goal, we do not want to lose the utility of an existing Ghidra extension that other users may find useful. (We realize that we just introduced it in

the preceding chapter... but it might have gone viral.) Instead, we will use some of the code from SimpleROP as a base to create our new script, *HeadlessSimpleROP*, which finds all ROP gadgets in *<filename>*, creates and writes them to *<filename>_gadgets.txt*, and then appends *<path>/<filename>* and records the count of ROP gadgets in a *HeadlessSimpleROP* summary file called *gadget_summary.txt*. Ghidra will provide all other required functionality (parsing directories, files, and so on) using the options we discussed earlier in this chapter.

To simplify development, we will create a new script using the Eclipse ► GhidraDev approach presented in Chapter 15, and then copy the *SimpleROP Analyzer.java* source code into the new script template and edit the code as needed. Finally, we will run the script, using the *-postScript* option so that it is invoked following the analysis phase for each opened file.

Creating a HeadlessSimpleROP Script

Begin by creating a template. From the GhidraDev menu, choose New ► GhidraScript and fill in the information shown in the dialog in Figure 16-11. While we could place the script in any folder, we will place it in the *ghidra_scripts* folder within our existing SimpleROP module in Eclipse.

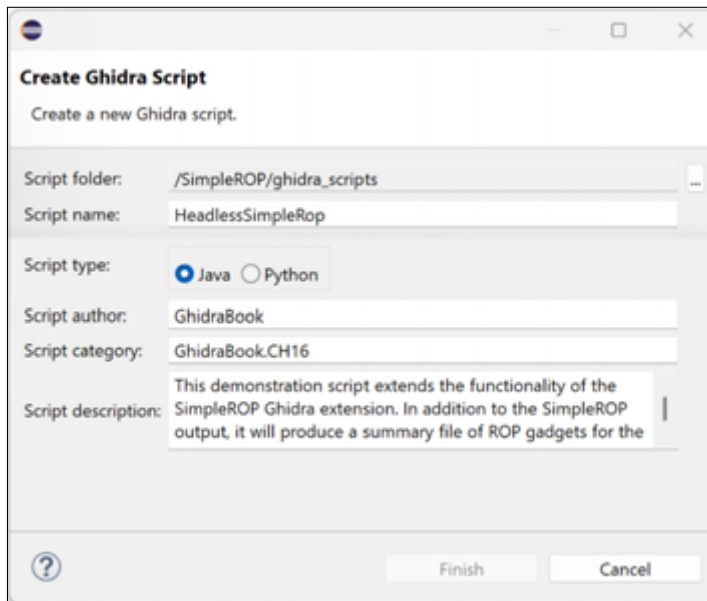
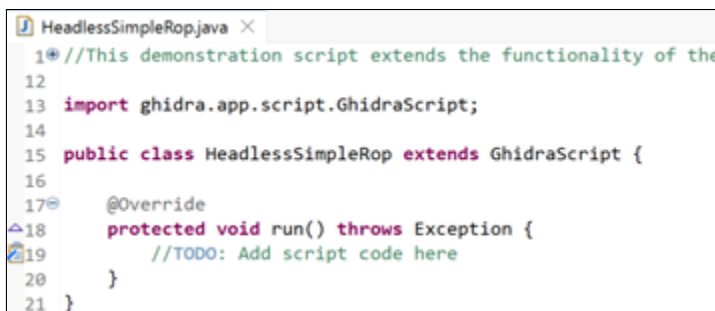


Figure 16-11: The Create Ghidra Script dialog

Click Finish to see the new script template, complete with metadata, as shown in Figure 16-12. The task tag on line 19 shows you where to get started.



```

1 //This demonstration script extends the functionality of the
12
13 import ghidra.app.script.GhidraScript;
14
15 public class HeadlessSimpleROP extends GhidraScript {
16
17     @Override
18     protected void run() throws Exception {
19         //TODO: Add script code here
20     }
21 }

```

Figure 16-12: The new *HeadlessSimpleROP* script template

To convert the SimpleROP analyzer into the *HeadlessSimpleROP* script, we need to do the following:

1. Remove the unneeded import statements.
2. Remove the analyzer public methods.
3. Duplicate the functionality of the added method that is called when the SimpleROPAnalyzer is invoked with the run method, which is called when the *HeadlessSimpleROP* script is invoked.
4. Add the functionality to append the filename and number of gadgets found to the summary file, *gadget_summary.txt*.

In the next sections, we will run a series of tests invoking the *HeadlessSimpleROP* script using items in the directory structure shown in Figure 16-6. These tests also demonstrate some of the options associated with headless Ghidra.

Test Scenario 1: Loading, Analyzing, and Processing a Single File

In the following listing, we use headless Ghidra to import, analyze, and invoke the script, which generates a ROP gadget report for a single file:

```

analyzeHeadless D:\GhidraProjects CH16_ROM ^
    -import D:\ch16\demo_stackframe_32 ^
    -postScript HeadlessSimpleROP.java

```

Note that ^ is the line-continuation character in a Windows command shell. When executed, the Ghidra headless analyzer creates a project called *CH16_ROM* in the *GhidraProjects* directory and then imports the file *demo_stackframe_32* to load and analyze. After analysis, we run our script on the imported and analyzed file.

Once the command has completed, we check the contents of the *gadget_summary.txt* and *demo_stackframe_32_gadgets.txt* files to determine whether our script worked correctly. The *demo_stackframe_32_gadgets.txt* contains 16 potential ROP gadgets:

```

080482c6;ADD ESP,0x8;POP EBX;RET;
080482c9;POP EBX;RET;
08048343;MOV EBX,dword ptr [ESP];RET;
08048360;MOV EBX,dword ptr [ESP];RET;
08048518;SUB ESP,0x4;PUSH EBP;
        PUSH dword ptr [ESP + 0x2c];
        PUSH dword ptr [ESP + 0x2c];
        CALL dword ptr [EBX + EDI*0x4 + 0xffffffff0c];
0804851b;PUSH EBP;PUSH dword ptr [ESP + 0x2c];
        PUSH dword ptr [ESP + 0x2c];
        CALL dword ptr [EBX + EDI*0x4 + 0xffffffff0c];
0804851c;PUSH dword ptr [ESP + 0x2c];PUSH dword ptr [ESP + 0x2c];
        CALL dword ptr [EBX + EDI*0x4 + 0xffffffff0c];
08048520;PUSH dword ptr [ESP + 0x2c];
        CALL dword ptr [EBX + EDI*0x4 + 0xffffffff0c];
08048535;ADD ESP,0xc;POP EBX;POP ESI;POP EDI;POP EBP;RET;
08048538;POP EBX;POP ESI;POP EDI;POP EBP;RET;
08048539;POP ESI;POP EDI;POP EBP;RET;
0804853a;POP EDI;POP EBP;RET;
0804853b;POP EBP;RET;
0804854d;ADD EBX,0x1ab3;ADD ESP,0x8;POP EBX;RET;
08048553;ADD ESP,0x8;POP EBX;RET;
08048556;POP EBX;RET;

```

Here is the associated entry in *gadget_summary.txt*:

```
demo_stackframe_32: Found 16 potential gadgets
```

Test Scenario 2: Loading, Analyzing, and Processing All Files in a Directory

In this test, we import an entire directory, rather than a file, with the import statement:

```

analyzeHeadless D:\GhidraProjects CH16_ROP ^
    -import D:\ch16 ^
    -postScript HeadlessSimpleROP.java

```

When the headless analyzer is complete, we find the following contents in *gadget_summary.txt*:

```

demo_stackframe_32: Found 16 potential gadgets
demo_stackframe_32_canary: Found 16 potential gadgets
demo_stackframe_32_stripped: Found 16 potential gadgets
demo_stackframe_64: Found 24 potential gadgets
demo_stackframe_64_canary: Found 24 potential gadgets
demo_stackframe_64_stripped: Found 24 potential gadgets

```

These are the six files in the root directory shown in Figure 16-6. In addition to the gadget summary file, we also produced individual gadget files listing the potential ROP gadgets associated with each file. In the remaining examples, we will concern ourselves only with the gadget summary file.

Test Scenario 3: Recursively Loading, Analyzing, and Processing All Files in a Directory

In this test, we add the `-recursive` option, which extends the import operation to recursively visit all files in all subdirectories within the `ch16` directory:

```
analyzeHeadless D:\GhidraProjects CH16_ROP ^
  -import D:\ch16 ^
  -postScript HeadlessSimpleROP.java ^
  -recursive
```

When the headless analyzer is complete, we find the following contents in *gadget_summary.txt*:

```
demo_stackframe_32_sub: Found 16 potential gadgets
demo_stackframe_32: Found 16 potential gadgets
demo_stackframe_32_canary: Found 16 potential gadgets
demo_stackframe_32_stripped: Found 16 potential gadgets
demo_stackframe_64: Found 24 potential gadgets
demo_stackframe_64_canary: Found 24 potential gadgets
demo_stackframe_64_stripped: Found 24 potential gadgets
```

The subdirectory file appears at the top of the list, and it has the `_sub` appended to the subdirectory name.

Test Scenario 4: Loading, Analyzing, and Processing All 32-bit Files in a Directory

In this test, we use `*` as a shell wildcard to restrict the import contents to the files with the 32-bit designator:

```
analyzeHeadless D:\GhidraProjects CH16ROP ^
  -import D:\ch16\demo_stackframe_32* ^
  -recursive ^
  -postScript HeadlessSimpleROP.java ^
  -scriptPath D:\GhidraScripts
```

The resulting *gadget_summary* file contains the following:

```
demo_stackframe_32: Found 16 potential gadgets
demo_stackframe_32_canary: Found 16 potential gadgets
demo_stackframe_32_stripped: Found 16 potential gadgets
```

If you know in advance that you are interested in only the generated gadget files, use the `-readOnly` option. This option instructs Ghidra not to save imported files into the project named in the command and is useful for avoiding project clutter from batch-processing many files.

Example 2: Automated Function ID Database Creation

In Chapter 13, we started creating a Function ID database (FidDb) populated with fingerprints of functions taken from a static version of *libc*. Using the GUI and Ghidra's batch file import mode, we imported 1,690 object files from a *libc.a* archive. However, when it came to analyzing the files, we ran into a roadblock because the GUI has limited support for batch analysis. Now that you are familiar with headless Ghidra, we can use it to complete our new FidDb.

The preceding examples have shown us everything we need to know to make short work of this task. We consider two cases here and provide command line examples for each.

In the first case, let's assume that *libc.a* has not yet been imported into a Ghidra project. We can extract the contents of *libc.a* into a directory and then use Ghidra in headless mode to process all of the files in that directory at once:

```
$ mkdir libc.a && cd libc.a
$ ar x __path\to\archive__ && cd ..
$ analyzeHeadless D:\GhidraProjects CH16 -import libc.a ^
    -processor x86:LE:64:default -cspec gcc -loader ElfLoader ^
    -recursive
```

The command results in thousands of lines of output as Ghidra reports its progress on the 1,690 files it processes, but once the command has completed, you will have a new *libc.a* folder in your project that contains 1,690 analyzed files.

In the second case, suppose we have already used the GUI to batch import *libc.a* but have not processed any of the 1,690 imported files. The following command line would take care of the analysis:

```
$ analyzeHeadless D:\GhidraProjects CH16\libc.a -process *
```

With the entire static archive efficiently imported and analyzed, we can now use the features of the Function ID plugin to create and populate an FidDb, as detailed in Chapter 13.

Summary

While GUI Ghidra remains the most straightforward and fully featured version of the software, running Ghidra in headless mode offers tremendous flexibility when creating complex tools built around Ghidra's automated analysis. At this point, we have covered all of Ghidra's most commonly used features and examined ways that you can make Ghidra work for you. It is time to move on to more advanced features.

Over the course of the next few chapters, we will look at approaches for some of the more challenging problems that arise while reverse engineering binaries, including dealing with unknown file formats and unknown processor architectures by building sophisticated Ghidra extensions. We'll also spend some time investigating Ghidra's decompiler and discuss some of the ways that compilers can vary in their generation of code to improve your fluency in reading disassembly listings.

PART IV

A DEEPER DIVE

17

LOADERS



Except for in a brief example used to demonstrate the Raw Binary loader in Chapter 4, Ghidra has identified the file type of all of the files we have thrown at it, and then happily loaded and analyzed them. This will not always be the case. At some point, you are likely to be confronted with a dialog like the one shown in Figure 17-1.

This particular file is shellcode, which Ghidra is unable to recognize, as it has no defined structure, meaningful file extension, or magic number.

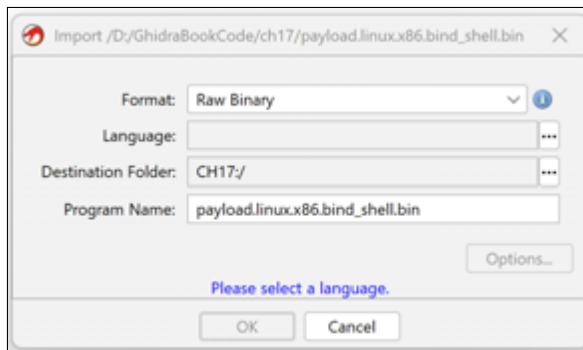


Figure 17-1: A Raw Binary loader example

What happened when we tried to import this file? Let's start with a high-level view of Ghidra's process for loading a file:

1. In the Ghidra Project window, the user specifies a file to load into a project.
2. The Ghidra Importer polls all of the Ghidra loaders, and each loader tries to identify the file. Each then responds with a list of load specifications to populate the Import dialog if it can load the file. (An empty list means "I can't load this file.")
3. The Importer collects responses from all of the loaders, builds a list of loaders that recognize the file, and presents a populated Import dialog to the user.
4. The user chooses a loader and associated information for loading the file.
5. The Importer invokes the user-selected loader that then loads the file.

For the file in Figure 17-1, none of the format-specific loaders responded with a "yes." As a result, Ghidra passed the task to the only loader willing to take any file at any time—the Raw Binary loader. This loader performs almost no work, shifting the analysis burden to the reverse engineer.

If you ever find yourself analyzing similar files that all appear to have the "raw" format, it may be time to build a specialized loader to help you with some or all of the loading process. You must undertake several tasks to create a new loader that Ghidra can use to load a file in a new format.

In this chapter, we first walk you through the analysis of a file whose format Ghidra does not recognize. This will help you understand the process of analyzing an unknown file and also make a strong case for building a loader, which is how we will spend the second half of the chapter.

Unknown File Analysis

Ghidra includes loader modules to recognize many of the most common executable and archive file formats, but there is no way that Ghidra can accommodate the ever-increasing number of file formats for storing executable code. Binary images may contain executable files formatted for use with specific operating systems, ROM images extracted from embedded systems, firmware images extracted from flash updates, or simply raw blocks of machine language, perhaps extracted from network packet captures. The format of these images may be dictated by the operating system (in the case of executable files), the target processor and system architecture (in the case of ROM images), or nothing at all (in the case of exploit shellcode embedded in application layer data).

Assuming that a processor module exists for disassembling the code contained in the unknown binary, it will be your job to properly arrange the file image within Ghidra before informing Ghidra of which portions of the binary represent code and which portions of the binary represent data. For most processor types, the result of loading a file using the raw format is

simply a list of the contents of the file piled into a single segment, beginning at address zero, as shown in Listing 17-1 for an example PE file.

00000000	4d	??	4Dh	M
00000001	5a	??	5Ah	Z
00000002	90	??	90h	
00000003	00	??	00h	
00000004	03	??	03h	
00000005	00	??	00h	
00000006	00	??	00h	
00000007	00	??	00h	

Listing 17-1: Initial lines of an unanalyzed PE file loaded using the Raw Binary loader

In some cases, depending on the sophistication of the selected processor module, some disassembly takes place. For example, a selected processor for an embedded microcontroller can make specific assumptions about the memory layout of ROM images, or an analyzer with knowledge of common code sequences associated with a specific processor can optimistically format any matches as code.

When faced with an unrecognized file, arm yourself with as much information about the file as you can get your hands on. Useful resources might include notes on how and where the file was obtained, processor references, operating system references, system design documentation, and any memory layout information obtained through debugging or hardware-assisted analysis (such as via logic analyzers).

In the following section, for the sake of an example, we assume that Ghidra does not recognize the Windows PE file format. In reality, PE is a well-known file format that Ghidra (and many readers) are very familiar with. More importantly, documents detailing the structure of PE files are widely available, making dissecting an arbitrary PE file a relatively simple task.

Listing 17-1 consists of the first few lines of an unanalyzed PE file loaded into Ghidra using the Raw Binary loader and `x86:LE:32:default:VisualStudio` as its language/compiler specification. (Choosing *Visual Studio* as your compiler results in *windows* being inserted in your language/compiler specification. For most other compilers, the selection name more closely matches the display name.) The PE specification states that a valid PE file begins with an MS-DOS header structure, beginning with the 2-byte signature, 4Dh 5Ah (MZ), which we see in the first two lines of Listing 17-1. The 4-byte value located at offset 0x3C in the file contains the offset to the next header we need to find: the PE header.

Using the Data Type Manager

Two strategies for breaking down the fields of the MS-DOS header are (1) to define appropriately sized data values for each field in the MS-DOS header and (2) to use Ghidra's Data Type Manager functionality to define and apply an `IMAGE_DOS_HEADER` structure in accordance with the PE file specification. We will look at the challenges associated with option 1 in an example later in the chapter. In this case, option 2 requires significantly less effort.

When using the Raw Binary loader, Ghidra does not load the Data Type Manager with the Windows data types, so we can load the archive containing MS-DOS types, *windows_vs12_32.gdt*, ourselves. Locate the `IMAGE_DOS_HEADER` by either navigating to it within the archive or choosing CTRL-F to find it in the Data Type Manager window; then, drag and drop the header onto the start of the file. You can also place the cursor on the first address in the listing and choose Data ► Choose Data Type (or press the hotkey T) from the right-click context menu and enter, or navigate to, the data type in the resulting Data Type Chooser dialog.

Any of these options yields the following listing, with descriptive end-of-line comments describing each field:

```

00000000 4d 5a      WORD      5A4Dh    e_magic
00000002 90 00      WORD      90h      e_cblp
00000004 03 00      WORD      3h       e_cp
00000006 00 00      WORD      0h       e_crlc
00000008 04 00      WORD      4h       e_cparhdr
0000000a 00 00      WORD      0h       e_minalloc
0000000c ff ff      WORD      FFFFh    e_maxalloc
0000000e 00 00      WORD      0h       e_ss
00000010 b8 00      WORD      B8h      e_sp
00000012 00 00      WORD      0h       e_csum
00000014 00 00      WORD      0h       e_ip
00000016 00 00      WORD      0h       e_cs
00000018 40 00      WORD      40h      e_lfarlc
0000001a 00 00      WORD      0h       e_ovno
0000001c 00 00 00   WORD[4]          e_res
           00 00 00
           00 00
00000024 00 00      WORD      0h       e_oemid
00000026 00 00      WORD      0h       e_oeminfo
00000028 00 00 00   WORD[10]         e_res2
           00 00 00
           00 00 00
0000003c d8 00 00   LONG      D8h      e_lfanew

```

The `e_lfanew` field in the final line of the previous listing has a value of `D8h`, indicating that a PE header should be found at offset `D8h` (216 bytes) into the binary. Examining the bytes at offset `D8h` should reveal the magic number for a PE header, `50h 45h` (PE), which indicates that we should apply an `IMAGE_NT_HEADERS` structure at offset `D8h` into the binary. Here is a portion of the resulting expanded Ghidra listing:

```

000000d8      IMAGE_NT_HEADERS
000000d8      DWORD      4550h    Signature
000000dc      IMAGE_FILE_HEADER
           000000dc      WORD      14Ch    ❶ Machine
           000000de      WORD      5h      ❷ NumberOfSections
           000000e0      DWORD      40FDFD    TimeDateStamp

```

000000e4	DWORD	0h	PointerToSymbolTable
000000e8	DWORD	0h	NumberOfSymbols
000000ec	WORD	E0h	SizeOfOptionalHeader
000000ee	WORD	10Fh	Characteristics
000000f0	IMAGE_OPTIONAL_HEADER32		OptionalHeader
000000f0	WORD	10Bh	Magic
000000f2	BYTE	'\u0006'	MajorLinkerVersion
000000f3	BYTE	'\0'	MinorLinkerVersion
000000f4	DWORD	21000h	SizeOfCode
000000f8	DWORD	A000h	SizeOfInitializedData
000000fc	DWORD	0h	SizeOfUninitializedData
00000100	DWORD	14E0h ❸	AddressOfEntryPoint
00000104	DWORD	1000h	BaseOfCode
00000108	DWORD	1000h	BaseOfData
0000010c	DWORD	400000h ❹	ImageBase
00000110	DWORD	1000h ❺	SectionAlignment
00000114	DWORD	1000h ❻	FileAlignment

At this point, we have revealed several interesting pieces of information that will help us to further refine the layout of the binary. First, the Machine field ❶ in a PE header indicates the target processor type for which the file was built. The value 14Ch indicates that the file is for use with x86 processor types. Had the machine type been something else, such as 1C0h (ARM), we would have needed to close the CodeBrowser, right-click our file in the Project window to select the Set Language option, and choose the correct language setting.

The ImageBase field ❹ indicates the base virtual address for the loaded file image. Using this information, we can incorporate some virtual address information into the CodeBrowser. Using the Window ► Memory Map menu option, we are shown the list of memory blocks (Figure 17-2) that make up the current program. In this case, a single memory block contains all of the program's content. The Raw Binary loader has no means of determining appropriate memory addresses for any of our program's content, so it places all of the content in a single memory block starting at address zero.

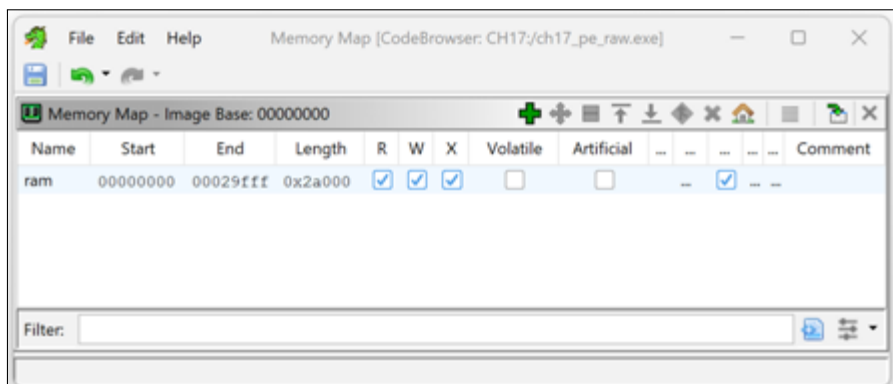


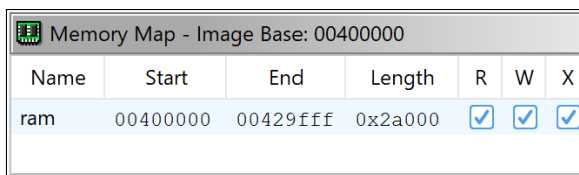
Figure 17-2: The Memory Map window

The Memory Map window's tool buttons, shown in Figure 17-3, are used to manipulate memory blocks. In order to properly map our image into memory, the first thing we need to do is set the base address specified in the PE header.

		
	Add Block	This button displays the Add Memory Block Dialog to allow you to add information needed to create a new memory block.
	Move Block	When a memory block is selected, this option allows you to modify the start or end address of the block, effectively moving it.
	Split Block	When a memory block is selected, this option allows you to split the memory block into two memory blocks.
	Expand Up	When a memory block is selected, this option allows you to append additional bytes before the memory block.
	Expand Down	When a memory block is selected, this option allows you to append additional bytes after the memory block.
	Merge Blocks	When two (or more) memory blocks are selected, this option merges them into one.
	Delete Block	Deletes all selected memory blocks.
	Set Image Base	This option allows you to choose (or modify) the base address of a program.
	Make Selection	This option uses the selected row(s) in the Memory Map window to make a program selection.
	Auto Update	This toggle determines whether or not the memory block is updated when you change locations in the global program location.

Figure 17-3: The Memory Map window tools

The ImageBase field tells us that the correct base address for this binary is 00400000. We can use the Set Image Base option to adjust the image base from the default to this value. Once we click OK, all Ghidra windows will be updated to reflect the new memory layout of the program, as shown in Figure 17-4. (Be careful using this option after you already have multiple memory blocks defined; it will shift every memory block the same distance as the base memory block.)



Memory Map - Image Base: 00400000						
Name	Start	End	Length	R	W	X
ram	00400000	00429fff	0x2a000	☒	☒	☒

Figure 17-4: The Memory Map after setting the image base

The `AddressOfEntryPoint` field ❸ specifies the relative virtual address (RVA) of the program entry point. In the PE file specification, an RVA is a relative offset from the program's base virtual address, while the program entry point is the address of the first instruction within the program file that will be executed. In this case, an entry point RVA of 14E0h indicates that the program will begin execution at virtual address 4014E0h (400000h + 14E0h). This is our first indication of where we should begin looking for code within the program. However, before we can do that, we need to properly map the remainder of the program to appropriate virtual addresses.

The PE format uses sections to describe the mapping of file content to memory ranges. By parsing the section headers for each section in the file, we can complete the basic virtual memory layout of the program. The `NumberOfSections` field ❷ indicates the number of sections contained in a PE file (in this case, five). According to the PE specification, an array of section header structures immediately follows the `IMAGE_NT_HEADERS` structure. Individual elements in the array are `IMAGE_SECTION_HEADER` structures, which we define in the Ghidra structures editor and apply (five times, in this case) to the bytes following the `IMAGE_NT_HEADERS` structure. Alternatively, you can select the first byte of the first section header and set its type to `IMAGE_SECTION_HEADER[n]`, where n is 5 in this example, to collapse the entire array into a single Ghidra display line.

The `FileAlignment` field ❹ and the `SectionAlignment` field ❺ indicate how the data for each section is aligned within the file and how that same data will be aligned when mapped into memory. In our example, both fields are set to align on 1000h byte offsets. In the PE format, there is no requirement that these two numbers be the same. However, the fact that they are the same does make our lives easier, as it means that offsets to content within the disk file are identical to offsets to the corresponding bytes in the loaded memory image of the file. Understanding how sections are aligned is important in helping us avoid errors when we manually create sections for our program.

NOTE

Alignment describes the starting address or offset of a block of data. The address or offset must be an even multiple of the alignment value. For example, when data is aligned to a 200h (512) byte boundary, it must begin at an address (or offset) that is divisible by 200h.

Creating Segments

After structuring each of the section headers, we have enough information to create additional segments within the program. Applying an `IMAGE_SECTION_HEADER` template to the bytes immediately following the `IMAGE_NT_HEADERS` structure yields the first section header in our Ghidra listing:

004001d0	IMAGE_SECTION_HEADER		
004001d0	BYTE[8]	".text"	❶ Name
004001d8	_union_226		Misc
004001d8	DWORD	20A80h	PhysicalAddress
004001d8	DWORD	20A80h	VirtualSize
004001dc	DWORD	1000h	❷ VirtualAddress
004001e0	DWORD	21000h	❸ SizeOfRawData
004001e4	DWORD	1000h	❹ PointerToRawData
004001e8	DWORD	0h	PointerToRelocations
004001ec	DWORD	0h	PointerToLinenumbers
004001f0	WORD	0h	NumberOfRelocations
004001f2	WORD	0h	NumberOfLinenumbers

The Name field ❶ informs us that this header describes the `.text` section. All of the remaining fields are potentially useful in formatting the listing, but we will focus on the three that describe the layout of the section. The `PointerToRawData` field ❹ (1000h) indicates the file offset at which the content of the section can be found. Note that this value is a multiple of the file alignment value, 1000h. Sections within a PE file are arranged in increasing file offset (and virtual address) order. Since this section begins at file offset 1000h, the first 1000h bytes of the file contain file header data and padding; if there are fewer than 1000h bytes of header data, the section must be padded to a 1000h byte boundary. Therefore, even though the header bytes do not, strictly speaking, constitute a section, we can highlight the fact that they are logically related by grouping them into a memory block in the Ghidra listing.

Ghidra offers two ways to create new memory blocks, both accessed through the Memory Map window from Figure 17-2. The Add Block tool (refer to Figure 17-3) opens the dialog shown in Figure 17-5, which is used to add new memory blocks that do not overlap with any existing memory block. The dialog asks for the name of the new memory block, its start address, and its length. The block may be initialized with a constant value (zero-filled, for example), initialized with content from the current file (you indicate the file offset from which the content is taken), or left uninitialized.

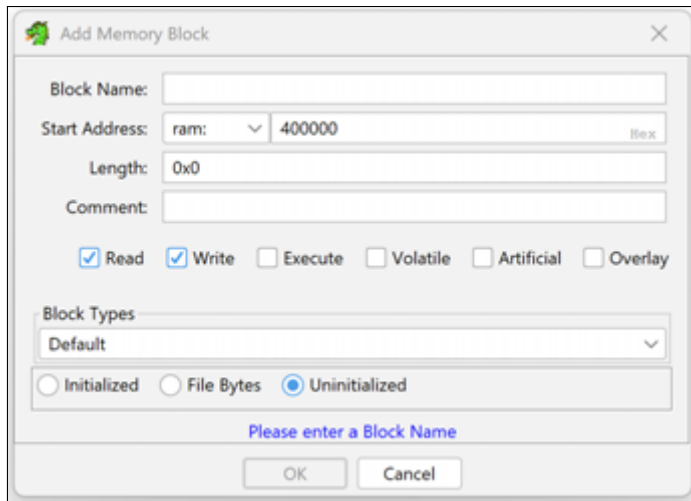


Figure 17-5: The Add Memory Block dialog

The second way to create a new block is to split an existing block. To split a block in Ghidra, you must first select the block to split in the Memory Map window and then use the Split Block tool (refer to Figure 17-3) to open the dialog shown in Figure 17-6. We are just starting out, so we have only one block to split. We start by splitting the file at the beginning of the `.text` section to carve the program headers off of the beginning of the existing block. When we enter the length (1000h) of our block to split (the header section), Ghidra automatically computes the remaining address and length fields. All that is left is to provide a name for the new block being created at the split point. Here, we use the name contained in the first section header: `.text`.

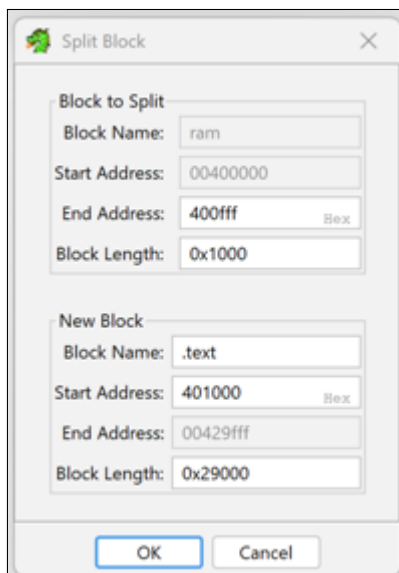


Figure 17-6: The Split Block dialog

We now have two blocks in our memory map. The first block contains the correctly sized program headers. The second block contains the correctly named, but not correctly sized, .text section. This situation is reflected in Figure 17-7, where we can see that the size of the .text section is 0x29000 bytes.

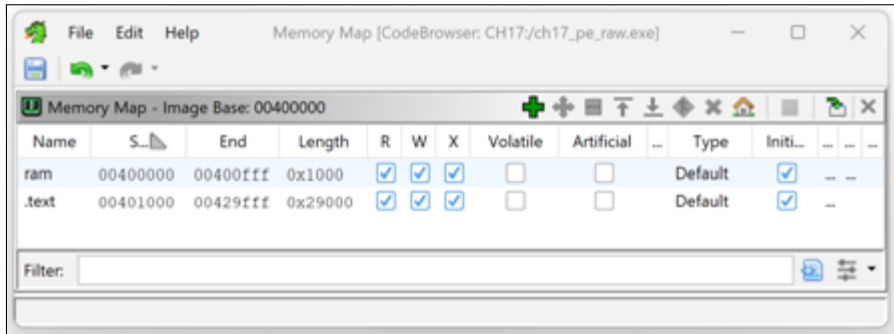


Figure 17-7: The Memory Map window after splitting a block

Returning to the header for the .text section, we see that the Virtual Address field ❷ (1000h) is an RVA that specifies the memory offset (from ImageBase) at which the section content begins and that the SizeOfRawData field ❸ (21000h) indicates how many bytes of data are present in the file. In other words, this particular section header tells us that the .text section is created by mapping the 21000h bytes from file offsets 1000h–21FFFh to virtual addresses 401000h–421FFFh.

Finalizing the Memory Map

Because we split the original memory block at the beginning of the .text section, the newly created .text section temporarily contains all remaining sections, since its current size of 0x29000 is greater than the correct size of 0x21000. By consulting the remaining section headers and repeatedly splitting the last memory block, we make progress toward a correct final memory map for the program. However, a problem arises when we reach the following pair of section headers:

00400220	IMAGE_SECTION_HEADER		
00400220	BYTE[8]	".data"	Name
00400228	_union_226		Misc
00400228	DWORD	5624h	PhysicalAddress
00400228	DWORD	5624h	❶ VirtualSize
0040022c	DWORD	24000h	❷ VirtualAddress
00400230	DWORD	4000h	❸ SizeOfRawData
00400234	DWORD	24000h	PointerToRawData
00400238	DWORD	0h	PointerToRelocations
0040023c	DWORD	0h	PointerToLinenumbers
00400240	WORD	0h	NumberOfRelocations
00400242	WORD	0h	NumberOfLinenumbers
00400244	DWORD	C0000040h	Characteristics

00400248	IMAGE_SECTION_HEADER		
00400248	BYTE[8]	".idata"	Name
00400250	_union_226	Misc	
00400250	DWORD	75Ch	PhysicalAddress
00400250	DWORD	75Ch	VirtualSize
00400254	DWORD	2A000h ❹	VirtualAddress
00400258	DWORD	1000h	SizeOfRawData
0040025c	DWORD	28000h ❺	PointerToRawData
00400260	DWORD	0h	PointerToRelocations
00400264	DWORD	0h	PointerToLinenumbers
00400268	WORD	0h	NumberOfRelocations
0040026a	WORD	0h	NumberOfLinenumbers
0040026c	DWORD	C0000040h	Characteristics

The .data section's virtual size ❶ is larger than its file size ❸. What does this mean, and how does it impact our memory map?

The compiler has concluded that the program requires 5624h bytes of runtime static data, but supplies only 4000h bytes to initialize that data. The remaining 1624h bytes of runtime data will not be initialized with content from the executable file, as they are allocated for uninitialized global variables. (It is not uncommon to see such variables allocated within a dedicated program section named .bss.)

To finalize our memory map, we must choose an appropriate size for the .data section and ensure that subsequent sections are also correctly mapped. The .data section maps 4000h bytes of file data from file offset 24000h to memory address 424000h ❷ (ImageBase + VirtualAddress). The next section (.idata) maps 1000h bytes from file offset 28000h ❺ to memory address 42A000h ❹.

If you're paying close attention, you may have noticed that the .data section appears to occupy 6000h bytes in memory (42A000h–424000h), and in fact it does. The reasoning behind this size is that the .data section requires 5624h bytes, but this is not an even multiple of 1000h, so the section will be padded up to 6000h bytes so that the .idata section properly adheres to the section alignment requirement specified in the PE header. In order to finish our memory map, we must perform the following actions:

1. Split the .data section using a length of 4000h. The resulting .idata section will, for the moment, start at 428000h.
2. Move the .idata section to address 42A000h by clicking the Move Block icon (Figure 17-3) and setting the start address to 42A000h.
3. Split off, and, if necessary, move any remaining sections to achieve the final program layout.
4. Optionally, expand any sections whose virtual size aligns to a higher boundary than their file size. In our example, the .data section's virtual size, 5624h, aligns to 6000h, while its file size, 4000h, aligns to 4000h. Once we have created room by moving the .idata section to its proper location, we will expand the .data section from 4000h to 6000h bytes.

To expand the .data section, highlight the .data section in the Memory Map window and then select the Expand Down tool (refer to Figure 17-3) to modify the end address (or length) of the section. Figure 17-8 shows the Expand Block Down dialog. (This operation will add the .exp extension to the section name.)

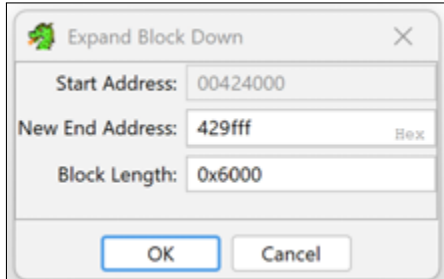


Figure 17-8: The Expand Block Down dialog

Our final memory map, obtained after the series of block moves, splits, and expansions, appears in Figure 17-9.

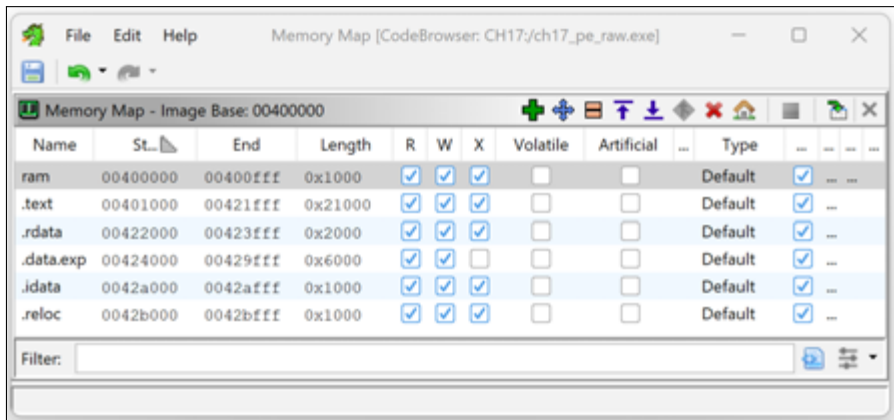


Figure 17-9: The final Memory Map window after creating all sections

In addition to the section name, start and end addresses, and length columns, the Memory Map window shows read (R), write (W), and execute (X) permissions for each section in the form of checkboxes. For PE files, these values are specified via bits in the Characteristics field of each section header. Consult the PE specification for information on parsing the Characteristics field to properly set permissions for each section.

Identifying Code

With all program sections properly mapped, we need to locate bytes that have a high likelihood of being code. The AddressOfEntryPoint (RVA 14E0h, or virtual address 4014E0h) leads us to the program's entry point, which is

known to be code. Navigating to this location, we see the following raw byte listing:

004014e0	??	55h	U
004014e1	??	8Bh	
004014e2	??	ECh	
...			

Using the context menu to disassemble (hotkey D) from address 004014e0 starts the recursive descent process, whose progress you may track in the lower-right corner of the CodeBrowser, and causes the bytes to be reformatted as the code seen here:

FUN_004014e0			
004014e0	PUSH	EBP	
004014e1	MOV	EBP,ESP	
004014e3	PUSH	-0x1	
004014e5	PUSH	DAT_004221b8	
004014ea	PUSH	LAB_004065f0	
004014ef	MOV	EAX,FS:[0x0]	
004014f5	PUSH	EAX	

At this point, we would hope that we had enough code to perform a comprehensive analysis of the binary. If we had fewer clues regarding the memory layout of the binary, or the separation between code and data within the file, we would need to rely on other sources of information to guide our analysis. Some potential approaches to determining correct memory layout and locating code include the following:

- Use processor reference manuals to understand where reset vectors may be found.
- Search for strings in the binary that might suggest the architecture, operating system, or compiler used to build the binary.
- Search for common code sequences, such as function prologues associated with the processor for which the binary was built.
- Perform statistical analysis over portions of the binary to find regions that look statistically similar to known binaries.
- Look for repetitive data sequences that might be tables of addresses (for example, many nontrivial 32-bit integers that all share the same upper 12 bits). These may be pointers and may provide clues regarding the memory layout of the binary.

NOTE

Trivial numbers typically have very few significant digits, and include -1, 0, and other small integers. Interesting numbers tend to have many significant digits, usually on the order of an architecture's bit size, and are more likely to yield more relevant search results.

In rounding out our discussion of loading raw binaries, consider that you would need to repeat each step covered in this section every time you open a binary with the same format that remains unknown to Ghidra. Along the way, you might automate some of your actions by writing scripts that perform some of the header parsing and segment creation for you. This is exactly the purpose of a Ghidra loader module! In the next section, we will write a simple loader module to introduce Ghidra's loader module architecture before moving on to more sophisticated loader modules that perform some common tasks associated with loading files that adhere to a structured format.

WHAT IS SHELLCODE, AND WHY DO WE CARE?

To be pedantic, *shellcode* is raw machine code whose sole purpose is to spawn a user-space shell process (for example, `/bin/sh`), most often by communicating directly with the operating system kernel using system calls. The use of system calls eliminates any dependencies on user-space libraries such as *libc*. The term *raw* in this case should not be confused with a Ghidra Raw Binary loader. Raw machine code is code that has no packaging in the form of file headers and is quite compact when compared to a compiled executable that carries out the same actions. Compact shellcode for x86-64 on Linux may be as small as 30 bytes, but a compiled version of the following C program, which also spawns a shell, is still over 6,000 bytes, even after it has been stripped:

```
#include <stdlib.h>
int main(int argc, char **argv, char **envp) {
    execve("/bin/sh", NULL, NULL);
}
```

The drawback to shellcode is that it can't be run directly from the command line. Instead, it is typically injected into an existing process, and action is taken to transfer control to the shellcode. Attackers may attempt to place shellcode into a process's memory space, in conjunction with other input consumed by the process, and then trigger a control flow hijack vulnerability that allows the attacker to redirect the process's execution to their injected shellcode. Because shellcode is often embedded within other input intended for a process, you may observe it in network traffic intended for a vulnerable server process or within a file meant to be opened by a vulnerable viewing application.

Over time, the term shellcode has come to refer generically to any raw machine code incorporated into an exploit, regardless of whether the execution of that machine code spawns a user-space shell on the target system.

Example 1: Simple Shellcode Loader Module

At the beginning of this chapter, we tried to load a shellcode file into Ghidra and were referred to the Raw Binary loader. In Chapter 15, we used Eclipse and GhidraDev to create an analyzer module and then added it as an extension to Ghidra. Recall that one of the module options provided by Ghidra was to create a loader module. In this section, we will extend Ghidra by building a simple loader module as an extension to Ghidra to load shellcode. As in our

Chapter 15 example, we will use a simplified software development process as this is just a simple demonstration project. Our process will include the following steps:

1. Define the problem.
2. Create the Eclipse module.
3. Build the loader.
4. Add the loader to our Ghidra installation.
5. Test the loader from our Ghidra installation.

Step 0: Taking a Step Back

Before we can even start to define the problem, we need to understand (a) what Ghidra currently does with a shellcode file and (b) what we would like Ghidra to do with a shellcode file. To that end, we will load and analyze a shellcode file as a raw binary and then use the information we discover to inform the development of our shellcode loader (and potentially an analyzer). Fortunately for us, most shellcode is not nearly as complicated as a PE file.

Let's start by analyzing the shellcode file we tried to load at the beginning of the chapter. Ghidra referred us to the Raw Binary loader as our only option because none of the other loaders wanted it. Let's select a relatively common language/compiler specification, x86:LE:32:default:gcc, as shown in Figure 17-10.

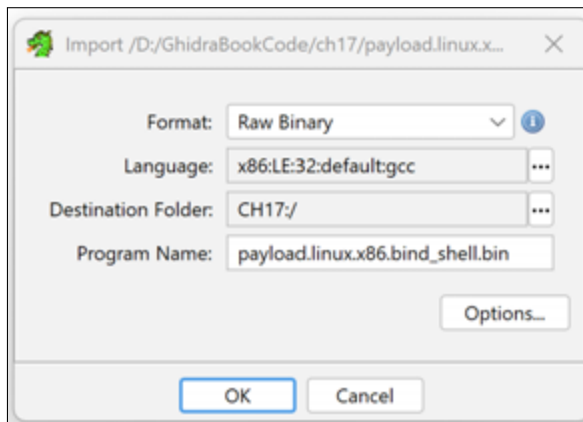


Figure 17-10: The Import dialog with language/compiler specification

We click OK and get an Import Results Summary window that includes the content shown in Figure 17-11.

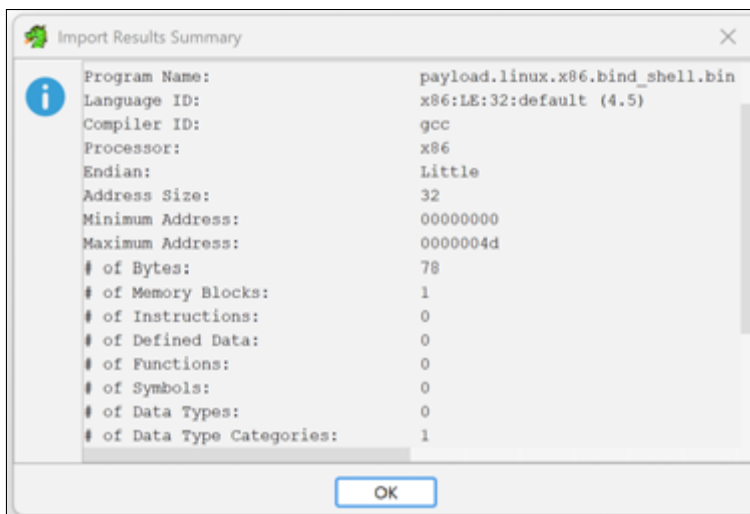


Figure 17-11: The import results summary for a shellcode file

Based on the contents of the import summary, we know that there are only 78 bytes in the file in one memory/data block, and that is about all the help we get from the Raw Binary loader.

If we open the file in the CodeBrowser, Ghidra will offer to auto analyze the file. Regardless of whether or not Ghidra does so, the Listing window in the CodeBrowser displays the content shown in Figure 17-12.

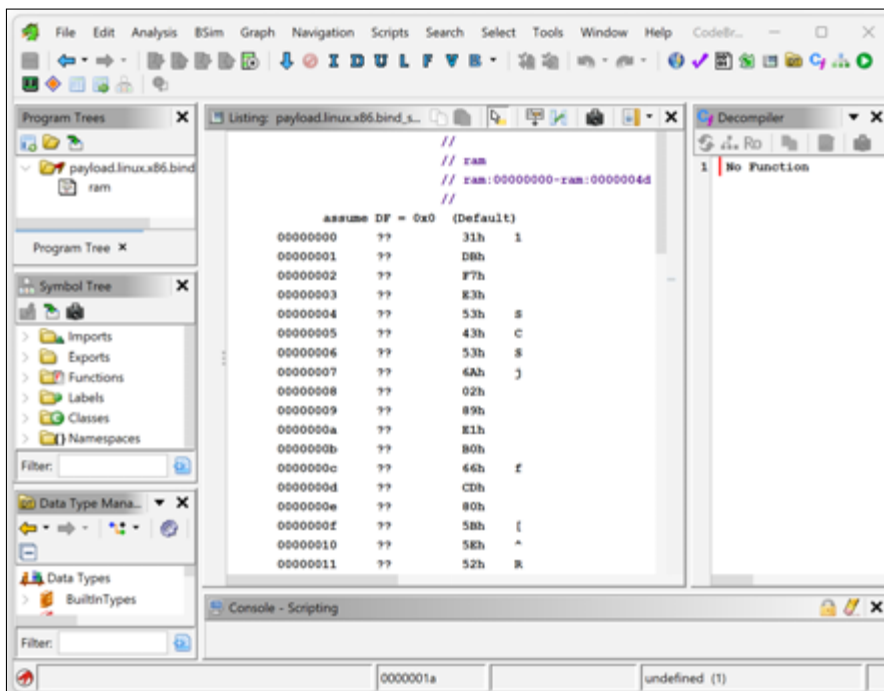


Figure 17-12: The CodeBrowser window after loading (or analyzing) the shellcode file

Note that there is only one section in Program Trees, the Symbol Tree is empty, and the Data Type Manager has no entries in the folder specific to the file. In addition, the Decompiler window remains empty, as no functions have been identified in the file.

Right-click the first address in the file and choose Disassemble (hotkey D) from the context menu. In the Listing window, we now see something we can work with—a list of instructions! Listing 17-2 shows the instructions after disassembly, and after we have done some analysis on the file. The end-of-line comments document some of the analysis of this short file.

```

0000002b  INC    EBX
0000002c  MOV    AL,0x66      ; 0x66 is Linux sys_socketcall
0000002e  INT    0x80          ; transfers flow to kernel to
                        ; execute system call

00000030  XCHG   EAX,EBX
00000031  POP    ECX
                        LAB_00000032      XREF[1]: 00000038(j)
00000032  PUSH   0x3f          ; 0x3f is Linux sys_dup2
00000034  POP    EAX
00000035  INT    0x80          ; transfers flow to kernel to
                        ; execute system call

00000037  DEC    ECX
00000038  JNS    LAB_00000
0000003a  PUSH   0x68732f2f    ; 0x68732f2f converts to "//sh"
0000003f  PUSH   0x6e69622f    ; 0x6e69622f converts to "/bin"
00000044  MOV    EBX,ESP
00000046  PUSH   EAX
00000047  PUSH   EBX
00000048  MOV    ECX,ESP
0000004a  MOV    AL,0xb        ; 0xb is Linux sys_execve which
                        ; executes a specified program
0000004c  INT    0x80          ; transfers flow to kernel to
                        ; execute system call

```

Listing 17-2: The Disassembled 32-bit Linux shellcode

Based on our analysis, we see that the shellcode invokes the Linux *execve* system call (at 0000004c) to launch */bin/sh* (which was pushed onto the stack at 0000003a and 0000003f). The fact that these instructions have meaning to us indicates that we likely chose an appropriate language and disassembly starting point.

We now know enough about the loading process to define our loader. (We also have enough information to build a simple shellcode analyzer, but that is a task for another day.)

Step 1: Defining the Problem

Our task is to design and develop a simple loader that will load shellcode into our Listing window and set the entry point, which will facilitate auto analysis. The loader needs to be added to Ghidra and be available as a Ghidra Loader option. It also needs to be able to respond to the Ghidra Importer poll in an appropriate manner the same way as the Raw Binary loader does. This will make our new loader a second catchall loader option. As a side note, all the examples will utilize FlatProgramAPI. While the FlatProgramAPI is not generally used for building extensions, its use will reinforce the scripting concepts presented in Chapter 14 that you are likely to use when developing Ghidra scripts in Java.

Step 2: Creating the Eclipse Module

As discussed in Chapter 15, use GhidraDev ► New ► Ghidra Module Project to create a module called SimpleShellcode that uses the Loader Module template. Figure 17-13 shows part of the folder hierarchy.

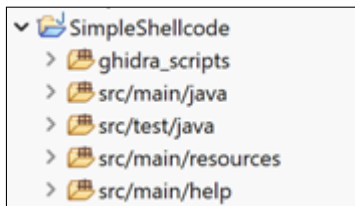
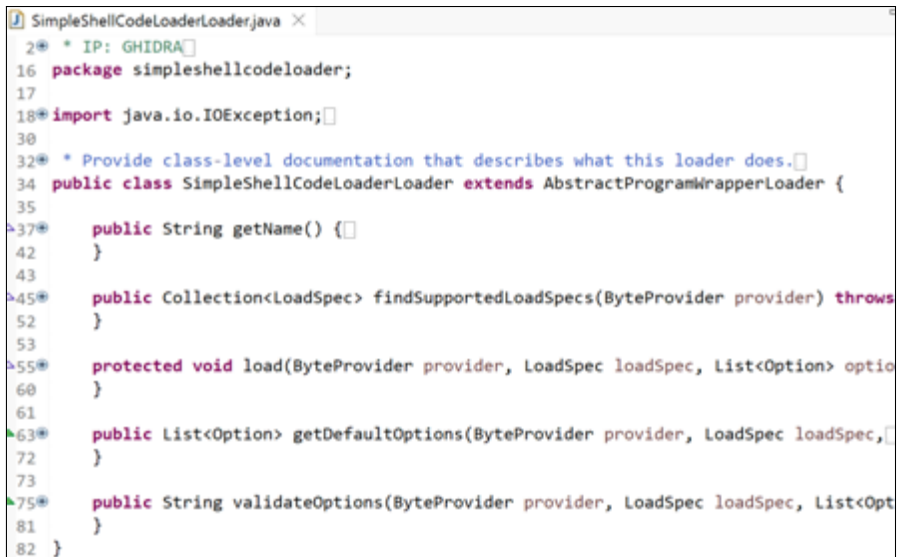


Figure 17-13: The SimpleShellcode hierarchy

We have created a file called *SimpleShellcodeLoader.java* in the *src/main/java* folder within the SimpleShellcode module.

Step 3: Building the Loader

Figure 17-14 shows a partial image of the loader template *SimpleShellcodeLoader.java*. The functions have been collapsed so that you can see all of the loader methods provided in the loader template. Recall that Eclipse will recommend imports if you need them as you develop your code, so you can jump right into coding and accept the recommended `import` statements when Eclipse detects that you need them.



```

SimpleShellCodeLoaderLoader.java
20 * IP: GHIDRA
16 package simpleshellcodeloader;
17
18 import java.io.IOException;
30
32 * Provide class-level documentation that describes what this loader does.
34 public class SimpleShellCodeLoaderLoader extends AbstractProgramWrapperLoader {
35
37 public String getName() {}
42 }
43
45 public Collection<LoadSpec> findSupportedLoadSpecs(ByteProvider provider) throws
52 }
53
55 protected void load(ByteProvider provider, LoadSpec loadSpec, List<Option> optio
60 }
61
63 public List<Option> getDefaultOptions(ByteProvider provider, LoadSpec loadSpec,
72 }
73
75 public String validateOptions(ByteProvider provider, LoadSpec loadSpec, List<Opt
81 }
82 }

```

Figure 17-14: The SimpleShellcodeLoader template

Within the loader template in Figure 17-14 are several method stubs that indicate where you should start your development. We will expand each section as we address specific tasks and include the before and after content associated with each task so that you understand how you need to modify the template. (We wrapped or reformatted some content for readability and minimized comments to conserve space.) Unlike the analyzer module you wrote in Chapter 15, this module does not require any obvious class member variables, so you can jump right into the tasks at hand.

Step 3-1: Documenting the Class

When you expand the method, you see the following task description:

```

/**
 * Provide class-level documentation that describes what this
 * loader does.
 */

```

This task involves replacing the existing comments with comments that describe what the loader does:

```

/*
 * This loader loads shellcode binaries into Ghidra,
 * including setting an entry point.
 */

```

Step 3-2: Naming and Describing the Loader

Expanding the next task tag reveals a comment and the string you need to edit. This makes it easy to identify where you should start working. The second task contains the following:

```
public String getName() {  
    // Name the loader. This name must match the name  
    // of the loader in the .opinion files.  
    ❶ return "My loader";  
}
```

Change the string ❶ to something meaningful. You don't need to worry about matching the name in the *.opinion* files, as these files are not applicable to loaders that will accept any files. Ignoring the *.opinion* file comment in the template results in the following code:

```
public String getName() {  
    return "Simple Shellcode Loader";  
}
```

You will see *.opinion* files when you get to the third example.

Step 3-3: Determining Whether the Loader Can Load This File

The second step in the loading process we described at the beginning of the chapter involved the Importer loader poll. This task requires you to determine whether your loader can load the file and provide a response to the Importer through your method's return value:

```
public Collection<LoadSpec> findSupportedLoadSpecs(ByteProvider provider)  
    throws IOException {  
    List<LoadSpec> loadSpecs = new ArrayList<>();  
  
    // Examine the bytes in 'provider' to determine if this loader  
    // can load it. If it can load it, return the appropriate load  
    // specifications.  
    return loadSpecs;  
}
```

Most loaders do this by examining the content of the file to find a magic number or header structure. The `ByteProvider` input parameter is a Ghidra-provided read-only wrapper around an input file stream.

We'll simplify our task and adopt the `LoadSpec` list that the Raw Binary loader uses, which ignores file content and simply lists all possible `LoadSpecs`. We will then give our loader a lower priority than the Raw Binary loader so that if a more specific loader exists, it will automatically have a higher priority in the Ghidra Import dialog:

```

public Collection<LoadSpec> findSupportedLoadSpecs(ByteProvider provider)
    throws IOException {

    // The list of load specs supported by this loader
    List<LoadSpec> loadSpecs = new ArrayList<>();
    List<LanguageDescription> languageDescriptions =
        getLanguageService().getLanguageDescriptions(false);
    for (LanguageDescription languageDescription : languageDescriptions) {
        Collection<CompilerSpecDescription> compilerSpecDescriptions =
            languageDescription.getCompatibleCompilerSpecDescriptions();
        for (CompilerSpecDescription compilerSpecDescription :
            compilerSpecDescriptions) {
            LanguageCompilerSpecPair lcs =
                new LanguageCompilerSpecPair(languageDescription.getLanguageID(),
                    compilerSpecDescription.getCompilerSpecID());
            loadSpecs.add(new LoadSpec(this, 0, lcs, false));
        }
    }
    return loadSpecs;
}

```

Every loader has an associated tier and a tier priority. Ghidra defines four tiers of loaders, ranging from highly specialized (tier 0) to format agnostic (tier 3). When multiple loaders are willing to accept a file, Ghidra sorts the loader list displayed to the user in increasing tier order. Loaders within the same tier are further sorted in increasing tier priority order (that is, tier priority 10 is listed before tier priority 20).

For example, the PE loader and the Raw Binary loader are both willing to load PE files, but the PE loader is a better choice for loading this format (its tier is 1), so it will appear before the Raw Binary loader (which has the tier 3 and tier priority 100) in the list.

We set the Simple Shellcode Loader's tier to 3 (`LoaderTier.UNTARGETED_LOADER`) and its priority to 101, so that the Importer will give it the lowest priority when populating the Import window with candidate loaders. To accomplish this, add the following two methods to your loader:

```

@Override
public LoaderTier getTier() {
    return LoaderTier.UNTARGETED_LOADER;
}

@Override
public int getTierPriority() {
    return 101;
}

```

Step 3-4: Loading the Bytes

The following method contains the appropriate parameters, but we need to edit the content to perform the heavy lifting of loading content from the file being imported into our Ghidra project:

```
protected void load(ByteProvider provider, LoadSpec loadSpec,
    List<Option> options, Program program, TaskMonitor monitor,
    MessageLog log) throws CancellationException, IOException {

    // Load the bytes from 'provider' into the 'program'.
}
```

When we replace the comment with the required content to load the shellcode, we end up with the following:

```
protected void load(ByteProvider provider, LoadSpec loadSpec,
    List<Option> options, Program program, TaskMonitor monitor,
    MessageLog log) throws CancellationException, IOException {
❶ FlatProgramAPI flatAPI = new FlatProgramAPI(program);
    try {
        monitor.setMessage("Simple Shellcode: Starting loading");
        // Create the memory block we're going to load the shellcode into.
        Address start_addr = flatAPI.toAddr(0x0);
        ❷ MemoryBlock block = flatAPI.createMemoryBlock("SHELLCODE",
            start_addr, provider.readBytes(0, provider.length()), false);
        // Make this memory block read/execute but not writeable.
        ❸ block.setRead(true);
        block.setWrite(false);
        block.setExecute(true);
        // Set the entry point for the shellcode to the start address.
        ❹ flatAPI.addEntryPoint(start_addr);
        monitor.setMessage("Simple Shellcode: Completed loading");
    } catch (Exception e) {
        e.printStackTrace();
        throw new IOException("Failed to load shellcode");
    }
}
```

Note that, unlike the scripts in Chapters 14 and 15, which inherit from `GhidraScript` (and ultimately `FlatProgramAPI`), our loader class has no direct access to the Flat API. Therefore, to simplify our access to some commonly used API classes, we instantiate our own `FlatProgramAPI` object ❶. Next, we create a `MemoryBlock` named `SHELLCODE` at address zero ❷ and populate it with the entire contents of the input file. We take the time to set some reasonable permissions ❸ on the new memory region before adding an entry point ❹ that informs Ghidra where it should begin its disassembly.

Adding an entry point is a very important step for a loader. The presence of entry points is the primary means by which Ghidra locates addresses known to contain code (as opposed to data). As it parses the input file, the loader is ideally suited to discover any entry points and identify them to Ghidra.

Step 3-5: Registering Custom Loader Options

Some loaders offer users the option to modify various parameters associated with the loading process. You may override the `getDefaultOptions` function to provide Ghidra with a list of custom options available for your loader:

```
public List<Option> getDefaultOptions(ByteProvider provider, LoadSpec
    loadSpec, DomainObject domainObject, boolean isLoadIntoProgram) {
    List<Option> list = super.getDefaultOptions(provider, loadSpec,

        domainObject, isLoadIntoProgram);
    // If this loader has custom options, add them to 'list'.
    list.add(new Option("Option name goes here",
        Default option value goes here));
    return list;
}
```

As this loader is for demonstration purposes only, we won't add options:

```
public List<Option> getDefaultOptions(ByteProvider provider, LoadSpec
    loadSpec, DomainObject domainObject, boolean isLoadIntoProgram) {
    // No options
    List<Option> list = new ArrayList<Option>();
    return list;
}
```

Options for a loader might include setting an offset into the file at which to start reading or setting the base address at which to load the binary. To view the options associated with any loader, click the Options button on the bottom right of the Import dialog (refer to Figure 17-1).

Step 3-6: Validating Options

The next task is to validate the options:

```
public String validateOptions(ByteProvider provider, LoadSpec loadSpec,
    List<Option> options, Program program) {

    // If this loader has custom options, validate them here.
    // Not all options require validation.
    return super.validateOptions(provider, loadSpec, options, program);
}
```

As we do not have any options, we just return null:

```
public String validateOptions(ByteProvider provider, LoadSpec loadSpec,
    List<Option> options, Program program) {

    // No options, so no need to validate
    return null;
}
```

TESTING MODULES FROM ECLIPSE

If you are one of those programmers who does not always get the code exactly right on the first try, you can avoid the multiple “export, start Ghidra, import extension, add extension to import list, choose extension, restart Ghidra, test extension” cycles by running the new code from Eclipse. If you choose Run ► Run As from the Eclipse menu, you will be given the option to run as Ghidra (or as Ghidra Headless). This will launch Ghidra, and you can import a file to the current project. Your loader will be included in the import options, and all console feedback will be provided in the Eclipse console. You can interact with the file in Ghidra, just like any other file. You can then exit your Ghidra project without saving and either (1) adjust the code, or (2) “export, start Ghidra, import extension, add extension to import list, choose extension, restart Ghidra, and test extension” just one time.

Step 4: Adding the Loader to Ghidra

After confirming that this module functions correctly, export the Ghidra module extension from Eclipse and then install the extension in Ghidra, just as we did with the SimpleROPAnalyzer module in Chapter 15. Select GhidraDev ► Export ► Ghidra Module Extension, choosing the Simple-Shellcode module, and follow the same click-through process that you did in Chapter 15.

To import the extension into Ghidra, choose File ► Install Extensions from the Ghidra Project window. Add the new loader to the list and select it. Once you restart Ghidra, the new loader should be available as an option, but you should test it to be sure.

Step 5: Testing the Loader in Ghidra

We will follow a simplified test plan designed to demonstrate the module’s functionality. SimpleShellcode passed an acceptance test consisting of the following criteria:

1. (Pass) SimpleShellcode appears as a loader option with lower priority than Raw Binary.
2. (Pass) SimpleShellcode loads a file and sets the entry point.

You can see that test case 1 passed in Figure 17-15. Figure 17-16, where the PE file analyzed earlier in the chapter is being loaded, shows a second confirmation. In both cases, we see that the Simple Shellcode Loader option has the lowest priority in the Format list.

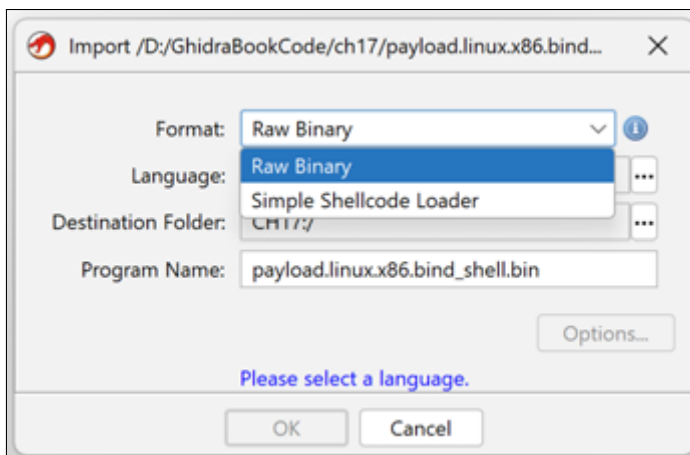


Figure 17-15: The Import window with our new loader listed as an option

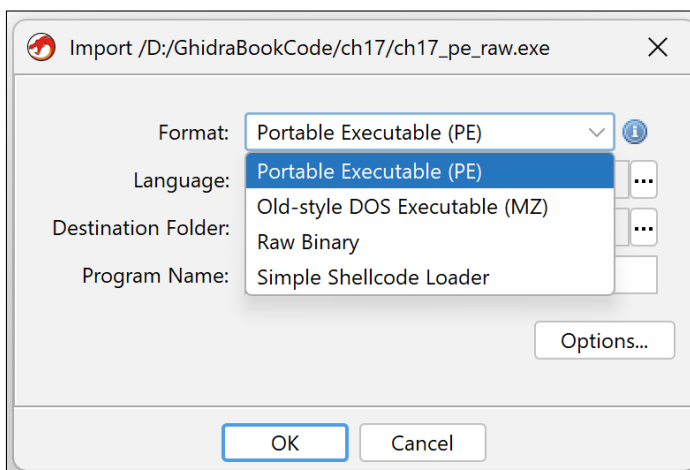


Figure 17-16: The Import window with our new loader listed as an option for a PE file

Choose the language specification based on the information available about the binary and how it was obtained. Let's assume that the shellcode was captured from packets headed for an x86 box. In that case, selecting x86:LE:32:default:gcc for our language/compiler specification is probably a good starting point.

After we select a language and click OK for the file shown in Figure 17-15, the binary will be imported into our Ghidra project. We can then open the program in the CodeBrowser, and Ghidra will provide us with an option to analyze the file. If we accept the analysis, we will see the following listing:

```

undefined FUN_00000000()
    undefined    <UNASSIGNED> <RETURN>

FUN_00000000
00000000 31 db      XOR    EBX,EBX
00000002 f7 e3      MUL    EBX
00000004 53         PUSH   EBX
00000005 43         INC    EBX
00000006 53         PUSH   EBX
00000007 6a 02      PUSH   0x2
00000009 89 e1      MOV    ECX,ESP
0000000b b0 66      MOV    AL,0x66
0000000d cd 80      INT    0x80
0000000f 5b         POP     EBX
00000010 5e         POP     ESI
00000011 52         PUSH   EDX
00000012 68 02 00 11 5c PUSH   0x5c110002

```

❶ XREF[1]: Entry Point(*)

An entry point ❶ is identified, so Ghidra is able to provide us with a disassembly to begin our analysis.

Example 2: Simple Shellcode Source Loader

SimpleShellcodeLoader was a trivial example, as shellcode is generally found embedded within some other data. Let's use our loader module as the basis for another loader module that extracts shellcode from C source files and loads the shellcode for analysis. This may, for example, allow us to build shellcode signatures that Ghidra can recognize in other binaries. We will not go into great depth for each step, as we are just augmenting the capabilities of our existing shellcode loader.

Rather than creating a new module, choose File ► New ► File from the Eclipse menu and add a new file (*SimpleShellcodeSourceLoader.java*) to your SimpleShellcode *src/main/java* folder. By doing this, you will include all of your new loaders in your new Ghidra extension.

Paste the contents of your existing *SimpleShellcodeLoader.java* into this new file and update the comments about what the loader does. The following steps highlight the parts of the existing loader that you need to change to make the new loader work as expected. For the most part, you will be adding onto the existing code.

Step 1: Modifying the Response to the Importer Poll

The simple source loader will make its decision strictly based on the file extension. If the file does not end in `.c`, the loader will return an empty `loadSpecs` list. If the file does end with `.c`, it will return the same `loadSpecs` list as for the previous loader. To make this work, you need to add the following test to the `findSupportLoadSpecs` method:

```
// The List of load specs supported by this loader
List<LoadSpec> loadSpecs = new ArrayList<>();
// Activate loader if the filename ends in a .c extension.
if (!provider.getName().endsWith(".c")) {
    return loadSpecs;
}
```

We have also decided that our loader deserves a higher priority than the Raw Binary loader because ours identifies a particular type of file to accept and is better suited for that type of file. So, we have the `getTierPriority` method return a higher priority (a lower value):

```
public int getTierPriority() {
    // Priority of this loader
    return 99;
}
```

Step 2: Finding the Shellcode in the Source Code

Recall that shellcode is just raw machine code that does something useful for us. The individual bytes in the shellcode will be in the range 0..255, and many of these values fall outside the range of ASCII printable characters. Therefore, when shellcode is embedded in a source file, much of it must be represented using hex escape sequences such as `\xFF`.

Strings of this sort are rather unique, and we can build a regular expression to help our loader identify them. The following instance variable declaration describes the regular expression that all of the functions in our loader may use to find shellcode bytes with the selected C file:

```
private String pattern = "\\x[0-9a-fA-F]{1,2}";
```

Within the `load` method, the loader parses the file looking for patterns that match the regular expression to help calculate the amount of memory needed when loading the file into Ghidra. As shellcode is frequently not contiguous, the loader should parse the entire file looking for shellcode regions to load from the file:

```
// Set up the regex matcher
CharSequence provider_char_seq =
    ❶ new String(provider.readBytes(0, provider.length()), "UTF-8");
Pattern p = Pattern.compile(pattern);
```

```

❷ Matcher m = p.matcher(provider_char_seq);
// Determine how many matches (shellcode bytes) were found so that we can
// correctly size the memory region, then reset the matcher.
int match_count = 0;
while (m.find()) {
    ❸ match_count++;
}
m.reset();

```

After loading the entire contents of the input file ❶, we count all of the matches ❸ against our regular expression ❷.

Step 3: Converting Shellcode to Byte Values

The `load()` method next needs to convert the hex escape sequences into byte values and put them in a byte array:

```

byte[] shellcode = new byte[match_count];
// Convert the hex representation of bytes in the source code to actual
// byte values in the binary we're creating in Ghidra.
int ii = 0;
while (m.find()) {
    // Strip out the \x.
    ❶ String hex_digits = m.group().replaceAll("[^0-9a-fA-F]+", "");
    // Parse what's left into an integer and cast it to a byte, then
    // set current byte in byte array to that value.
    ❷ shellcode[ii++] = ❸ (byte)Integer.parseInt(hex_digits, 16);
}

```

We extract the hex digits from each matching string ❶ and convert them into byte values ❸ that get appended to our shellcode array ❷.

Step 4: Loading the Converted Byte Array

Finally, because the shellcode is in a byte array, the `load()` method needs to copy it from the byte array into the program's memory:

```

// Create the memory block and populate it with the shellcode.
Address start_addr = flatAPI.toAddr(0x0);
MemoryBlock block =
    flatAPI.createMemoryBlock("SHELLCODE", start_addr, shellcode, false);

```

This is the actual loading step and the last required step for your loader to accomplish the goal.

Steps 5: Testing the Loader

To test our new loader, we create a C source file that contains the following escaped representation of x86 shellcode:

```

unsigned char buf[] =
    "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd\x80"
    "\x5b\x5e\x52\x68\x02\x00\x11\x5c\x6a\x10\x51\x50\x89\xe1\x6a"
    "\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd\x80\x43\xb0"
    "\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49\x79\xf8\x68\x2f"
    "\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0"
    "\x0b\xcd\x80";

```

Because our source file's name ends in `.c`, our loader appears in the list as the top selection, with a higher priority than the Raw Binary and Simple Shellcode loaders, as shown in Figure 17-17.

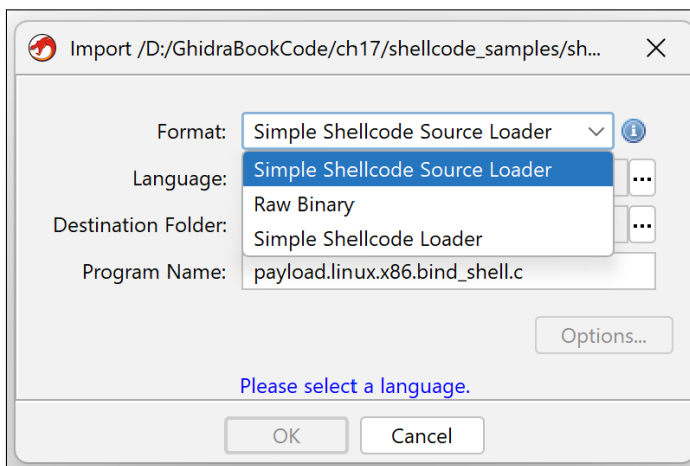


Figure 17-17: The Import dialog for a shellcode source file

Selecting this loader, using the same default compiler/language specification as the previous example (`x86:LE:32:default:gcc`), and letting Ghidra auto analyze the file yields the following function in the disassembly listing:

```

*****
*                               *
*                               FUNCTION                               *
*****
undefined FUN_00000000()
    undefined <UNASSIGNED> <RETURN>

FUN_00000000                                XREF[1]: Entry Point(*)
00000000  XOR     EBX,EBX
00000002  MUL     EBX
00000004  PUSH    EBX
00000005  INC     EBX
00000006  PUSH    EBX

```

Scrolling down through the listing leads us to the familiar content shown here (with comments added for clarity):

```
LAB_00000032
00000032  PUSH    0x3f
00000034  POP     EAX
00000035  INT     0x80
00000037  DEC     ECX
00000038  JNS     LAB_00000032
0000003a  PUSH    0x68732f2f      ; 0x68732f2f converts to "//sh"
0000003f  PUSH    0x6e69622f      ; 0x6e69622f converts to "/bin"
```

Most reverse engineering efforts focus on binaries. In this case, we have stepped outside that box and used Ghidra to load shellcode for analysis as well as to extract shellcode from C source files. Our goal was to demonstrate the flexibility and simplicity of creating loaders for Ghidra. Now, let's step back into that box and create a loader for a structured file format.

Assume that our target shellcode is contained within an ELF binary and that, for the sake of this example, Ghidra does not recognize ELF binaries. Further, none of us have ever heard of an ELF binary. Let the adventure begin.

Example 3: Simple ELF Shellcode Loader

Congratulations! You are now the resident shellcode reverse engineering expert, and colleagues are reporting what they suspect is shellcode contained in binaries, referred by Ghidra to the Raw Binary loader. Since this does not appear to be a one-off problem, and you think there is a good chance that you will see more binaries with similar characteristics, you decide to build a loader that will handle this new type of file.

As discussed in Chapter 13, you can use tools internal or external to Ghidra to capture information about the file. If you once again turn to the command line, `file` provides helpful information to start building your loader:

```
$ file elf_shellcode_min
elf_shellcode_min: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, corrupted section header size
```

The `file` command provides information about a format you have never heard of before, ELF. Your first step is to do some research to see if you can locate information about this type of binary. Many online resources can point you to references about the ELF format, and you can use these to locate the information you need to build your loader. Any resource that provides enough accurate information to solve the problem is a viable candidate.

As this project is a bigger challenge than the previous two loader examples, we will break it into sections associated with the individual files in your Eclipse SimpleShellcode module, each of which you will need to create,

modify, or delete to complete your new *SimpleELFShellcodeLoader*. We will start with some simple housekeeping.

Step 1: Performing Housekeeping

The first step is to create a *SimpleELFShellcodeLoader.java* file within the SimpleShellcode module in Eclipse. As you don't want to start from nothing, you should use Save As with *SimpleShellcodeLoader.java* to create this new file. Once you have done this, there are a few minor modifications to make to the new file before you can start focusing on the new challenge:

- Change the name of the class to *SimpleELFShellcodeLoader*.
- Modify the *getTier* method return value from *UNTARGETED_LOADER* to *GENERIC_TARGET_LOADER*.
- Delete the *getTierPriority* method.
- Modify the *getName* method to return "Simple ELF Shellcode Loader".

Now let's apply the information you learned from your research about the new header format.

Step 2: Researching the ELF Header Format

While researching this new format, you discover that the ELF format contains three types of headers: the file header (or ELF header), the program header(s), and the section header(s). You can start by focusing on the ELF header. Associated with each field in the ELF header is an offset as well as other information about the field. Since you need to access only a few of these fields and you won't be modifying the offsets, declare the following constants as instance variables within your loader class to help your loader correctly parse this new header format:

```
private final byte[] ELF_MAGIC           = {0x7f, 0x45, 0x4c, 0x46};
private final long  EH_MAGIC_OFFSET      = 0x00;
private final long  EH_MAGIC_LEN        = 4;

private final long  EH_CLASS_OFFSET      = 0x04;
private final byte  EH_CLASS_32BIT      = 0x01;

private final long  EH_DATA_OFFSET       = 0x05;
private final byte  EH_DATA_LITTLE_ENDIAN = 0x01;

private final long  EH_ETYPE_OFFSET      = 0x10;
private final long  EH_ETYPE_LEN        = 0x02;
private final short EH_ETYPE_EXEC       = 0x02;

private final long  EH_EMACHINE_OFFSET   = 0x12;
private final long  EH_EMACHINE_LEN     = 0x02;
private final short EH_EMACHINE_X86     = 0x03;
```

```

private final long EH_EFLAGS_OFFSET      = 0x24;
private final long EN_EFLAGS_LEN        = 4;

private final long EH_EEHSIZE_OFFSET     = 0x28;
private final long EH_PHENTSIZE_OFFSET   = 0x2A;
private final long EH_PHNUM_OFFSET       = 0x2C;

private short e_ehsize;
private short e_phentsize;
private short e_phnum;

```

With a description of the ELF header in hand, the next step is to determine how to respond to the Importer poll to ensure that the new ELF loader is capable of loading only files that adhere to the ELF format. In the previous two examples, the shellcode loaders did not look at file contents to determine if they could load a file. This significantly simplified coding of these examples. Now things are a bit more complicated. Fortunately, the ELF documentation provides important clues to help determine the appropriate loader specifications.

Step 3: Finding Supported Load Specifications

The loader cannot load anything that is not in the right format. To reject a file, it must return an empty loadSpecs list. Within the findSupportedLoadSpecs() method, immediately eliminate all binaries that do not have the expected magic number by using the following code:

```

List<LoadSpec> loadSpecs = new ArrayList<>();
byte[] magic = provider.readBytes(EH_MAGIC_OFFSET, EH_MAGIC_LEN);
if (!Arrays.equals(magic, ELF_MAGIC)) {
    // Confirm the binary is not an ELF.
    return loadSpecs;
}

```

Once you have eliminated the undesirables, the loader can check the bit width and endianness to see if the architecture is reasonable for an ELF binary. For this demonstration, let's further limit the types of binaries the loader will accept to 32-bit little-endian:

```

byte ei_class = provider.readByte(EH_CLASS_OFFSET);
byte ei_data = provider.readByte(EH_DATA_OFFSET);
if ((ei_class != EH_CLASS_32BIT) || (ei_data != EH_DATA_LITTLE_ENDIAN)) {
    // In these cases, it is NOT an ELF we want to accept.
    return loadSpecs;
}

```

To round out the verification process, the following code checks if the file is an ELF executable (as opposed to a shared library) for the x86 architecture:

```

byte[] etyp = provider.readBytes(EH_ETYPE_OFFSET, EH_ETYPE_LEN);
short e_type =
    ByteBuffer.wrap(etyp).order(ByteOrder.LITTLE_ENDIAN).getShort();
byte[] emach = provider.readBytes(EH_EMACHINE_OFFSET, EH_EMACHINE_LEN);
short e_machine =
    ByteBuffer.wrap(emach).order(ByteOrder.LITTLE_ENDIAN).getShort();
if ((e_type != EH_ETYPE_EXEC) || (e_machine != EH_EMACHINE_X86)) {
    // In these cases, it is NOT an ELF we want to accept.
    return loadSpecs;
}

```

Now that you have limited your file types, you can query the opinion service for matching language and compiler specifications. Conceptually, you query the opinion services with values extracted from the file you are loading (for example, the ELF header `e_machine` field), and in response receive a list of language/compiler specifications that your loader is willing to accept. (We describe the “behind the scenes” actions that take place when you query the opinion service in more detail in the following sections.)

```

byte[] eflag = provider.readBytes(EH_EFLAGS_OFFSET, EN_EFLAGS_LEN);
int e_flags = ByteBuffer.wrap(eflag).order(ByteOrder.LITTLE_ENDIAN).getInt();
List<QueryResult> results =
    QueryOpinionService.query(getName(), Short.toString(e_machine),
                              Integer.toString(e_flags));

```

Let’s assume that the opinion service is likely to yield more results than you want to handle with this loader. You can further pare down the list by excluding results based on the attributes specified in the associated language/compiler specifications. The following code filters out a compiler and a processor variant:

```

for (QueryResult result : results) {
    CompilerSpecID cspec = result.pair.getCompilerSpec().getCompilerSpecID();
    ❶ if (cspec.toString().equals("borlanddelphi")) {
        // Ignore anything created by Delphi.
        continue;
    }
    String variant = result.pair.getLanguageDescription().getVariant();
    ❷ if (variant.equals("System Management Mode")) {
        // Ignore anything where the variant is "System Management Mode".
        continue;
    }
    // Valid load spec, so add it to the list.
    ❸ loadSpecs.add(new LoadSpec(this, 0, result));
}
return loadSpecs;

```

These examples (which you are free to include in your loader) specifically exclude the Delphi compiler ❶ and x86 system management mode ❷. You can exclude others if you wish. All results that you have not excluded need to be added to your loadSpecs list ❸.

Step 4: Loading File Content into Ghidra

The load() method of your simplified loader assumes that the file consists of a minimal ELF header and a short program header, followed by the shell-code in a text section. You need to determine the total length of the header to allocate the correct amount of space for it. The following code determines the required size by using the EH_EEHSIZE_OFFSET, EH_PHENTSIZE_OFFSET, and EH_PHNUM_OFFSET fields from the ELF header:

```
// Get some values from the header needed for the load process.
//
// How big is the ELF header?
byte[] ehsz = provider.readBytes(EH_EEHSIZE_OFFSET, 2);
e_ehsize = ByteBuffer.wrap(ehsz).order(ByteOrder.LITTLE_ENDIAN).getShort();

// How big is a single program header?
byte[] phsz = provider.readBytes(EH_PHENTSIZE_OFFSET, 2);
e_phentsize =
    ByteBuffer.wrap(phsz).order(ByteOrder.LITTLE_ENDIAN).getShort();

// How many program headers are there?
byte[] phnum = provider.readBytes(EH_PHNUM_OFFSET, 2);
e_phnum = ByteBuffer.wrap(phnum).order(ByteOrder.LITTLE_ENDIAN).getShort();

// Determine the total header size for our simplified ELF format.
// (This includes the ELF Header plus program headers.)
long hdr_size = e_ehsize + e_phentsize * e_phnum;
```

Now that you know the size, create and populate the memory blocks for the ELF header section and the text section as follows:

```
// Create the memory block for the ELF header.
long LOAD_BASE = 0x10000000;
Address hdr_start_addr = flatAPI.toAddr(LOAD_BASE);
MemoryBlock hdr_block =
    flatAPI.createMemoryBlock(".elf_header", hdr_start_addr,
        provider.readBytes(0, hdr_size), false);
// Make this memory block read-only.
hdr_block.setRead(true);
hdr_block.setWrite(false);
hdr_block.setExecute(false);

// Create the memory block for the text from the simplified ELF binary.
```

```

Address txt_start_addr = flatAPI.toAddr(LOAD_BASE + hdr_size);
long txt_size = provider.length() - hdr_size;
MemoryBlock txt_block =
    flatAPI.createMemoryBlock(".text", txt_start_addr,
        provider.readBytes(hdr_size, txt_size),
        false);

// Make this memory block read and execute.
txt_block.setRead(true);
txt_block.setWrite(false);
txt_block.setExecute(true);

```

A few more steps, and you will be done.

Step 5: Formatting Data Bytes and Adding an Entry Point

Loaders often apply data types and create cross-references for information derived from file headers. It is also the loader's job to identify any entry points in the binary. Creating a list of entry points at load time provides the disassembler with a list of locations it should consider code. Our loader follows these practices:

```

// Add structure to the ELF header.
❶ flatAPI.createData(hdr_start_addr, new ElfDataType());
// Add label and entry point at start of shellcode.
❷ flatAPI.createLabel(txt_start_addr, "shellcode", true);
❸ flatAPI.addEntryPoint(txt_start_addr);

// Add a cross-reference from the ELF header to the entry point.
Data d = flatAPI.getDataAt(hdr_start_addr).getComponent(0).getComponent(11);
❹ flatAPI.createMemoryReference(d, txt_start_addr, RefType.DATA);

```

First, we apply the Ghidra ELF header data type at the start of the ELF headers ❶. (If this were truly a new format, you would probably have to create this structure within Ghidra based on your research. For this example, use the one in the Ghidra Data Type Manager window.) Second, we create a label ❷ and an entry point ❸ for the shellcode. Finally, we create a cross-reference between the entry point field in the ELF header and the start of the shellcode ❹.

Congratulations! You are done writing the Java code for your loader, but we need to address a couple of issues to ensure that you understand all of the dependencies between your new loader and some important related files in order for your loader to operate as expected.

This example leverages an existing processor architecture (x86), and some work happened behind the scenes to help this loader work correctly. Recall that the Importer polled the loaders and magically produced acceptable language/compiler specifications. Two files provided information critical to the loader. The first of these files is the x86 language definition file *x86.ldefs*, a component of the x86 processor module.

Step 6: Understanding Language Definition Files

Every processor has an associated language definition file. This XML-formatted file includes all the information required to generate language/compiler specifications for the processor. The following listing shows language definitions from the *x86.ldefs* file that meet the requirements for a 32-bit ELF binary:

```
<language processor="x86"
    endian="little"
    size="32"
    variant="default"
    version="4.6"
    slafile="x86.sla"
    processorspec="x86.pspec"
    manualindexfile="../manuals/x86.idx"
    id="x86:LE:32:default">
<description>Intel/AMD 32-bit x86</description>
<compiler name="Visual Studio" spec="x86win.cspec" id="windows"/>
<compiler name="clang" spec="x86win.cspec" id="clangwindows"/>
<compiler name="gcc" spec="x86gcc.cspec" id="gcc"/>
<compiler name="Borland C++" spec="x86borland.cspec" id="borlandcpp"/>
<compiler name="Delphi" spec="x86delphi.cspec" id="borlanddelphi"/> ❶
<compiler name="golang" spec="x86-32-golang.cspec" id="golang"/>
<external_name tool="gnu" name="i386:intel"/>
<external_name tool="IDA-PRO" name="8086"/>
<external_name tool="IDA-PRO" name="80486p"/>
<external_name tool="IDA-PRO" name="80586p"/>
<external_name tool="IDA-PRO" name="80686p"/>
<external_name tool="IDA-PRO" name="k62"/>
<external_name tool="IDA-PRO" name="p2"/>
<external_name tool="IDA-PRO" name="p3"/>
<external_name tool="IDA-PRO" name="athlon"/>
<external_name tool="IDA-PRO" name="p4"/>
<external_name tool="IDA-PRO" name="metapc"/>
<external_name tool="DWARF.register.mapping.file" name="x86.dwarf"/>
<external_name tool="Golang.register.info.file" name="x86-32-golang.register.info"/>
<external_name tool="qemu" name="qemu-i386"/>
<external_name tool="qemu_system" name="qemu-system-i386"/>
</language>
<language processor="x86"
    endian="little"
    size="32"
    variant="System Management Mode" ❷
    version="4.6"
    slafile="x86.sla"
    processorspec="x86-16.pspec"
```



```

manualindexfile="../manuals/x86.idx"
id="x86:LE:32:System Management Mode">
<description>Intel/AMD 32-bit x86 System Management Mode</description>
<compiler name="default" spec="x86-16.cspect" id="default"/>
<external_name tool="DWARF.register.mapping.file" name="x86.dwarf"/>
</language>

```

This file is used to populate the recommended language/compiler specs presented as import options. In this case, there are several recommended specifications (each starting with the compiler tag), which will be returned based on information associated with the ELF binary, but our loader eliminates two from consideration based on the compiler ❶ and the variant ❷.

Step 7: Creating the Opinion Files

Another type of support file is the *.opinion* file. This is an XML-formatted file that contains constraints associated with your loader. To be recognized by the opinion query service, each loader must have an entry in an opinion file. The following listing shows a suitable opinion file entry for the loader you just built:

```

<opinions>
  <constraint loader="Simple ELF Shellcode Loader" compilerSpecID="gcc">
    <constraint primary="3" processor="x86" endian="little" size="32" />
    <constraint primary="62" processor="x86" endian="little" size="64" />
  </constraint>
</opinions>

```

Everything in the entry should seem familiar, except possibly the *primary* field. This field is the primary key for a search that identifies the machine as defined in the ELF header. Within the ELF header, the value 0x03 in the *e_machine* field means x86, and 0x3E in the *e_machine* field means amd64. A *<constraint>* tag defines an association between a primary key ("3"/x86) and the remaining attributes of the *<constraint>* tag. The query service uses this information to locate the appropriate entries in the language definition files.

Our only remaining task is to place our opinion data somewhere Ghidra will find it. The only opinion files that ship with Ghidra reside in the *data/languages* subdirectory of a Ghidra processor module. Although you could insert your opinion data into an existing opinion file, it is a good idea to avoid modifying any processor opinion files, as you will need to reapply your modifications whenever you upgrade your Ghidra installation.

Instead, create a new opinion file that contains the opinion data. You can name the file anything you wish, but *SimpleShellcode.opinion* seems reasonable. Our Eclipse Loader Module template contains its own *data* subdirectory. Save your opinion file in this location to associate it with your loader module. Ghidra will locate it when looking for opinion files, and any upgrades to the Ghidra installation should not affect your module.

Now that you understand what is going on behind the scenes, it is time to test your loader and see if it behaves as anticipated.

Step 8: Observing the Results

To demonstrate the success of the new, simplified ELF loader (with one program header and no sections), let's walk through the loading process and observe how the loader performs at each step of the process.

From the Ghidra Project window, import a file. The importer will poll all of Ghidra's loaders, including yours, to see which are willing to load this file. Recall that your loader is expecting a file that fits the following profile:

- Has the ELF magic number at the start of the file
- Is 32-bit little endian
- Is an ELF executable for the x86 architecture
- Cannot have been compiled by Delphi
- Cannot have the variant "System Management Mode"

If you load a file that fits that profile, you should see an Import dialog similar to the one in Figure 17-18 that displays a prioritized list of the loaders willing to process this file.

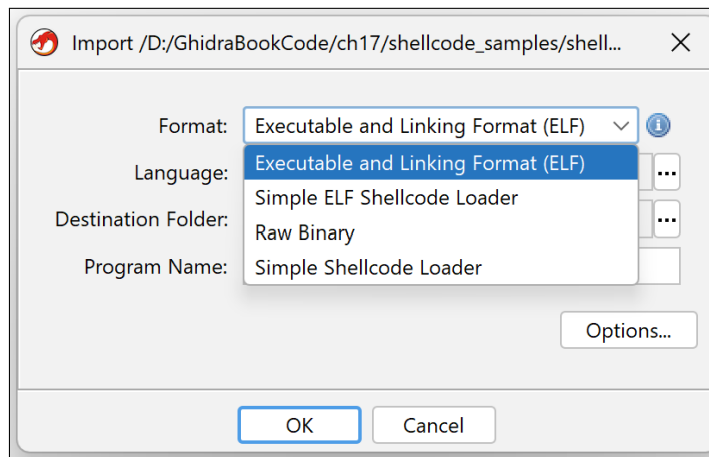


Figure 17-18: The import options for elf_shellcode_min

The loader with the highest priority is Ghidra's ELF loader. Let's compare the language/compiler specifications that it will accept (at the top of Figure 17-19) with the ones that your new loader will accept (at the bottom).

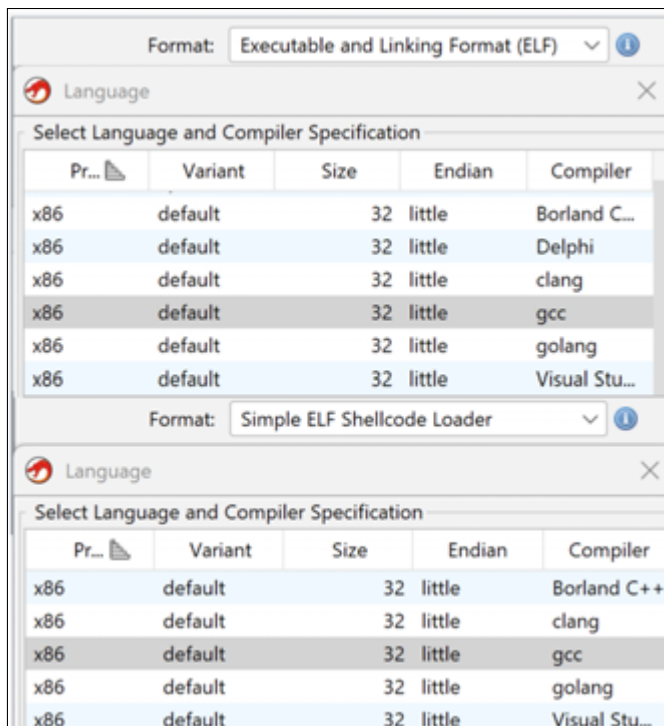


Figure 17-19: The acceptable language/compiler specifications for two different loaders

The Delphi compiler and the System Management Mode variant are accepted by the stock ELF loader but not by your loader, as they have been filtered out. When you select your loader for the file *elf_shellcode_min*, you should see a summary similar to Figure 17-20.

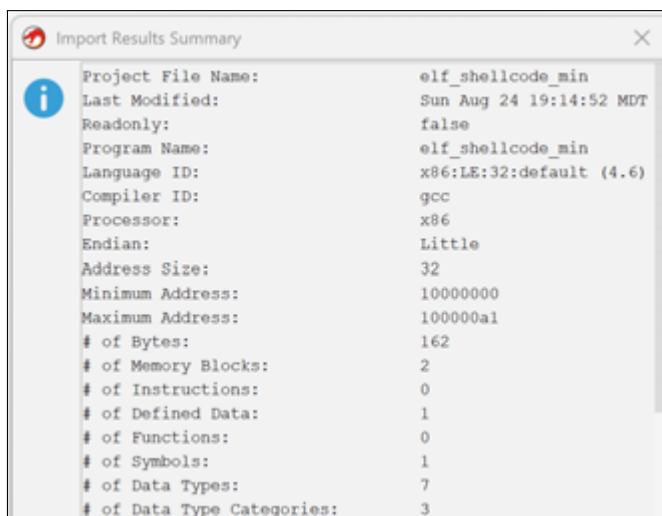


Figure 17-20: The Import Results Summary window for the new ELF Shellcode Loader

If you open the file in the CodeBrowser and allow Ghidra to auto analyze the file, you should see the following ELF header definition at the top of the file:

10000000	7f	db	7Fh	e_ident_magic_num
10000001	45 4c 46	ds	"ELF"	e_ident_magic_str
10000004	01	db	1h	e_ident_class
10000005	01	db	1h	e_ident_data
10000006	01	db	1h	e_ident_version
10000007	00 00 00 00 00 00 00 00	db[9]		e_ident_pad
	00 00 00 00			
10000010	02 00	dw	2h	e_type
10000012	03 00	dw	3h	e_machine
10000014	01 00 00 00	ddw	1h	e_version
10000018	54 00 00 10	ddw	❶ shellcode	e_entry
1000001c	34 00 00 00	ddw	34h	e_phoff
10000020	00 00 00 00	ddw	0h	e_shoff
10000024	00 00 00 00	ddw	0h	e_flags
10000028	34 00	dw	34h	e_ehsize

Within the listing, the shellcode label ❶ is clearly associated with the entry point. Double-clicking the shellcode label takes you to a function, named shellcode, that contains the same shellcode contents we've seen in our previous two examples, including the following:

1000008c	JNS	LAB_10000086	
1000008e	PUSH	"hs//"	; "//sh"
10000093	PUSH	"nib/"	; "/bin"
10000098	MOV	EBX,ESP	
1000009a	PUSH	EAX	

Now that you have confirmed that your new loader works, you can add it as an extension to your Ghidra installation and share it with your colleagues, who have been anxiously awaiting this functionality.

Summary

In this chapter, we focused on the challenges associated with dealing with unrecognized binary files. We walked through examples of the loading and analysis processes that we can use within Ghidra to help us with these challenging reverse engineering scenarios. Finally, we extended our module creation capabilities to the world of Ghidra loaders.

While the examples that we built were trivial, they provided the foundation and introduced all of the components required to write more complex loader modules in Ghidra. In the next chapter, we round out our discussion of Ghidra modules with an introduction to processor modules—the components most responsible for the overall formatting of a disassembled binary.

18

PROCESSORS



Perhaps the most complex of Ghidra's module types, processor modules are responsible for all of the disassembly operations that take place in Ghidra. Beyond the obvious conversion of machine language opcodes into their assembly language equivalents, processor modules also enable the creation of functions, cross-references, and stack frames.

While Ghidra supports an impressive number of processors and adds new ones with every major release, you may sometimes need to develop a processor module yourself. The obvious reason to do so is to reverse engineer a binary for which no processor module exists in Ghidra. Among other things, such a binary might represent a firmware image for an embedded microcontroller or an executable image pulled from handheld or IoT devices.

A less-obvious reason to develop a new processor module is to disassemble the instructions of a custom virtual machine embedded within an obfuscated x86 executable. In such cases, the existing Ghidra x86 processor module would help you understand only the virtual machine itself, not the virtual machine's underlying byte code.

Developing a processor module is an arduous task compared to the analyzer and loader module examples in previous chapters, which required modifying a single Java file. If you created these modules within the Eclipse GhidraDev environment, you were given a module template and task comments to help you complete your task. Processor modules are more complex, and you must maintain relationships between several related files for the processor module to work correctly.

While we will not build a processor module from scratch in this chapter, we will provide you with a solid foundation to help you understand Ghidra processor modules and demonstrate creating and modifying components within those modules.

Understanding Ghidra Processor Modules

Creating a processor module for a real-world architecture is a highly specialized, time-consuming effort beyond the scope of this book. However, you will find it helpful to have some understanding of how processors and their associated instruction sets are represented in Ghidra so that you can identify where to look for resources when you need information about a Ghidra processor module.

Eclipse Processor Modules

We will start in somewhat familiar territory, using Eclipse ► GhidraDev to create a processor module. The resulting folder structure is similar to that for every other module type, except that a processor module does not provide a Java source file, complete with comments about the steps needed to complete achieve your objectives. Instead, you see the contents shown in Figure 18-1.

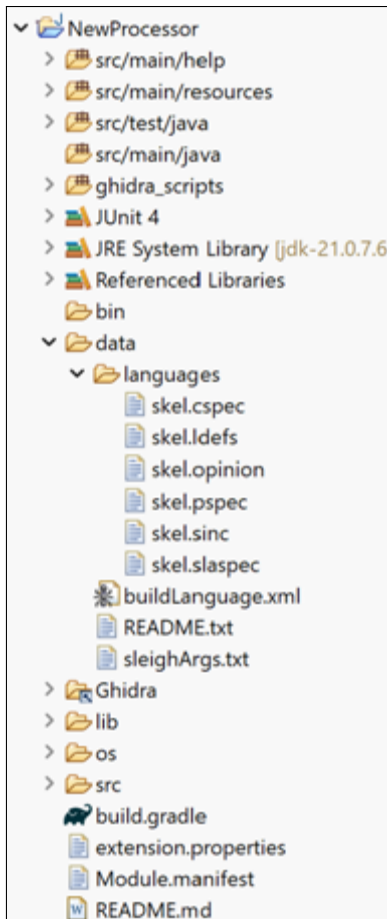


Figure 18-1: The processor module contents

The processor module *data* folder (expanded in the figure) contains a lot more than the brief *README.txt* provided in the *data* folder for other module types. Let's briefly meet its nine files, with a focus on their file extensions. (The *skel* prefix lets us know we are working with a skeleton.)

skel.cspec An XML-formatted, initially overwhelming compiler specification file.

skel.ldefs An XML-formatted language definition file. The skeleton has a commented-out template for defining a language.

skel.opinion An XML-formatted importer opinion file. The skeleton has a commented-out template for defining a language/compiler specification.

skel.pspec An XML-formatted processor specification file.

skel.sinc Generally a SLEIGH file for language instructions. For large instruction sets, such as the x86 instruction set, it may be broken into multiple *.sinc* files, some of which may be used as header files with definitions and include statements.

skel.slaspec This is a SLEIGH specification file.

buildLanguage.xml This XML file describes the build process for the files in the *data/languages* directory.

README.txt This file is the same in all of the modules, but within this module it finally makes sense, as it focuses on the contents of the *data/* directory.

sleighArgs.txt This file holds SLEIGH compiler options.

You encountered the *.ldefs* and *.opinion* files in Chapter 17 when building your ELF shellcode loader and will see the other file extensions in context as you work through examples. Before we work with these files to modify a processor module, let's discuss a new term specific to processor modules, SLEIGH.

The SLEIGH Language

SLEIGH is a language specific to Ghidra that describes microprocessor instruction sets, and it is used to support the Ghidra disassembly and compilation processes. You can find detailed information about the SLEIGH language in your Ghidra installation in *docs/languages/html/sleigh.html*. Because files within the *languages* directory are either written in SLEIGH or presented in XML format, you will definitely need to learn a little about SLEIGH to create or modify a processor module.

A *.slaspec* file specifies how instructions are encoded and how a processor should interpret them, somewhat analogous to the role of a *.c* file. When a processor family has a number of distinct variants, each variant may have its own *.slaspec* file, while separate *.sinc* files factor out common behaviors across variants (similar to the role of *.h* files), which many *.slaspec* files may include. Ghidra's ARM processor is an excellent example of this design, with over a dozen *.slaspec* files, each referencing one or more of six *.sinc* files. These files constitute the SLEIGH source code for a processor module, and it is the SLEIGH compiler's job to compile them into a *.sla* file suitable for use by Ghidra.

Rather than taking a deep dive into SLEIGH from a theoretical perspective, we will introduce various components of the SLEIGH language as we encounter and require them in our examples. First let's look at the sort of information that a SLEIGH file contains about instructions.

To see additional information associated with an instruction in a Code-Browser listing, right-click the instruction and select Instruction Info from the context menu. The displayed information is derived from SLEIGH file specifications for the selected instruction. Figure 18-2 shows the Instruction Info window for an x86-64 PUSH instruction.

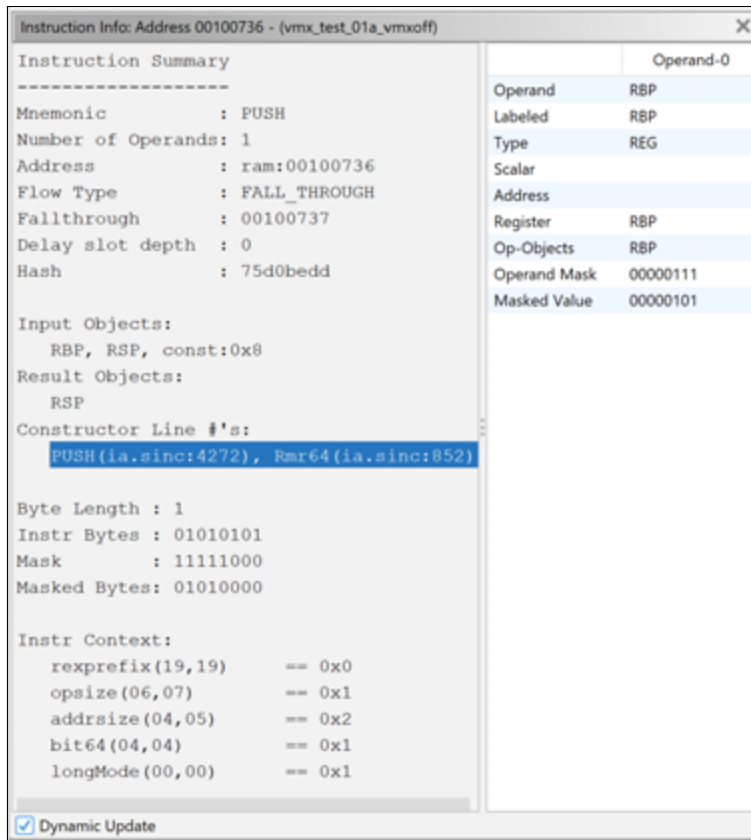


Figure 18-2: The Instruction Info window

The Instruction Info window combines information about the PUSH instruction from the SLEIGH file and highlights details about the specific use of PUSH at address 00100736. Later in the chapter, we will work with instruction definitions within a SLEIGH file and will revisit this window in the context of the instructions we are working with.

Processor Manuals

The documentation provided by a processor's manufacturer is an important resource for obtaining information about the instruction set. While Ghidra cannot bundle these copyrighted manuals, it provides a mechanism to connect them so you can jump directly from an instruction in the Listing window to the relevant page in the official documentation.

If you right-click any instruction and select Processor Manual, you are likely to see a message similar to that shown in Figure 18-3, informing you that the manual for the current processor is not available in the expected location.

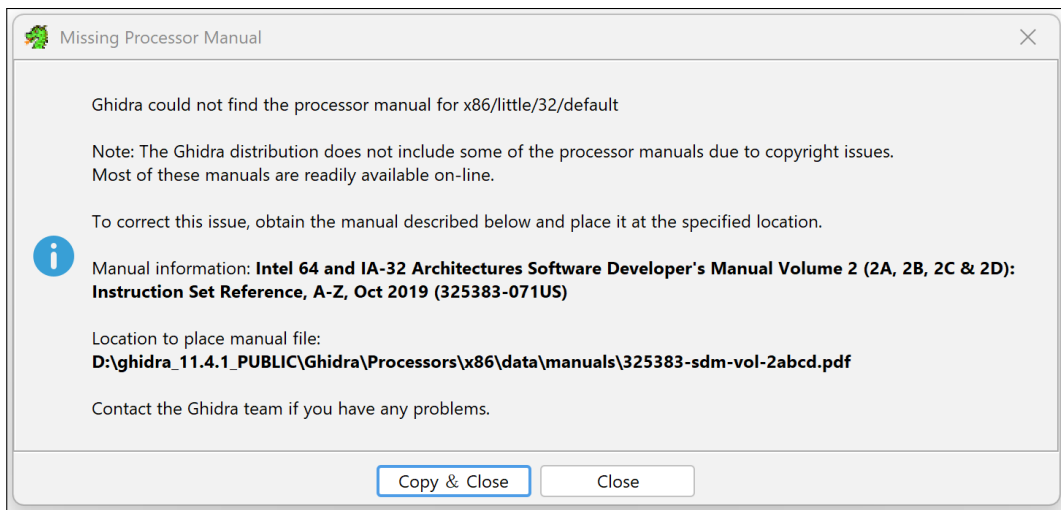


Figure 18-3: The Missing Processor Manual dialog

Here, Ghidra provides you the information needed to resolve the missing manual situation. In this particular example, you first need to locate the x86 manual online and then save it with the specified name and location.

NOTE

There are many processor manuals associated with the x86. Find the correct manual online by searching for the identifier provided at the end of the manual information provided in the Missing Process Manual dialog, in this case 326019-074US.

Once you have properly installed a manual, selecting Processor Manual will display it. Since processor manuals tend to be large (the complete manual for the x86 processor weighs in at almost 5,200 pages), Ghidra helpfully includes the capability to process index files that map an instruction to a specific page in a manual. Fortunately, the index for this specific x86 manual has already been created for you.

You should place processor manuals in the directory appropriate for your processor (*Ghidra/Processors/<targetprocessor>/data/manuals*). Index files should reside in the same directory as their associated manual. The format of an index file is relatively straightforward. The following listing shows the first few lines of Ghidra's *x86.idx* file:

```
@325383-sdm-vol-2abcd.pdf [Intel 64 and IA-32 Architectures
    Software Developer's Manual Volume 2 (2A, 2B, 2C \& 2D):
    Instruction Set Reference, A-Z, 2019 (325383-071US)]
AAA, 120
AAD, 122
BLENDPS, 123
AAM, 124
AAS, 126
```

The first line in the file (which we have wrapped across three lines in this listing) pairs the manual's local filename with descriptive text displayed to the user when the manual is not present on the system. The format of the line is as follows:

```
@FilenameInGhidraManualDirectory [Description of manual file]
```

Each additional line is of the form *INSTRUCTION*, *page*. The instruction must be uppercase, and the page number is counted from the first page of the .pdf file. (This is not necessarily the page number that appears on the given page of the document.) Keeping the index file aligned with the correct manual ensures a smooth workflow.

A single .idx file can reference several manuals. Simply use additional @ directives to delineate each additional manual's instruction map. You can find more information about processor manual index files in *docs/languages/manual_index.txt* in your Ghidra installation directory.

Once you have a manual saved and indexed, selecting Processor Manual for any instruction in the Listing window should take you to its corresponding page within the manual. If the manual does not appear, you may need to choose **Edit ► Tool Options ► Processor Manuals** to configure an appropriate viewer application for your manual. Figure 18-4 shows a sample viewer setting for opening the manual using the default system viewer.

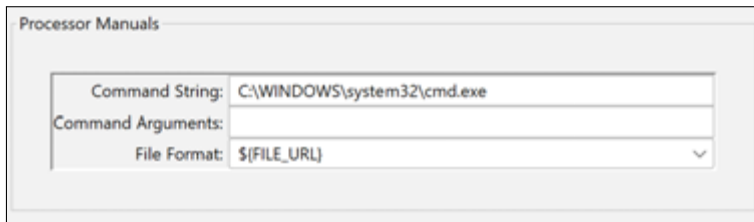


Figure 18-4: The Processor Manuals tool options

Now that you have some basic processor module terminology under your belt, it's time to dive into the internals of a processor module implementation.

Modifying Processor Modules

Building a processor module from scratch is a significant undertaking. Rather than jumping in headfirst, let's modify an existing module. To demonstrate concepts related to real-world problems, we will start by identifying a hypothetical issue regarding Ghidra's x86 processor module and then walk through some examples that address the issue. Finally, we will explore how the various components we discussed work together to form a complete Ghidra processor module.

GHIDRA'S SLEIGH EDITOR

To assist you in modifying and building processor modules, Ghidra includes a SLEIGH editor that easily integrates into the Eclipse environment. The installation instructions for the editor are part of the SLEIGH README file in *Extensions/Eclipse/GhidraSleighEditor* and take only a few steps to complete. Here is some of the editor's special functionality:

Syntax highlighting Colorizes content that has special meaning (for example, comments, tokens, strings, and variables).

Validation Marks many syntax errors and generates warnings for errors that would otherwise remain undetected until compilation.

QuickFix Provides recommendations for resolving issues detected by the editor. (This is similar to the QuickFix options for `import` statements we saw in Chapter 15.)

Hover Provides additional information for many constructs when you hover over the construct.

Navigation Provides navigation functionality specific to SLEIGH (for example, subconstructors, tokens, registers, and pcodeops).

Find references Quickly finds all uses of a variable.

Renaming Rather than traditional string-based search and replace, this renames an actual variable in the file and other related *.sinc* and *.slaspec* files.

Code formatting Reformats files specific to the structure of the SLEIGH language. (For example, it lines up constructors based on keywords and lines up entries within attach blocks). You can apply this functionality to an entire file or a selected section.

While we recommend using this editor, especially for the helpful early syntax checking, the development of our examples in this chapter is not specific to this editor.

Problem Statement

Let's begin with the problem we will attempt to solve. A quick search of the *Ghidra/Processors* directory in your local installation shows that the x86 processor module includes many instructions but appears to be missing a hypothetical virtual machine extension (VMX) management instruction for the IA32 and IA64 architectures.

This instruction (which we just invented for this example) is called `VMXPLODE`. Its behavior is similar to the `VMXOFF` instruction, which Ghidra does support. While the existing `VMXOFF` instruction causes the processor to leave VMX operation, `VMXPLODE` causes it to leave with a flourish! We will walk you through adding this very important hypothetical instruction to the existing Ghidra x86 processor module in order to introduce some of the concepts associated with building and modifying a processor module.

Example 1: Adding an Instruction to a Processor Module

Our first goal is to locate the files we need to modify to support the VMXPLODE instruction. The *Ghidra/Processors* directory contains subdirectories for all processors supported by Ghidra, one of which is for x86. You can open the x86 processor module (or any other processor module) directly in Eclipse using File ► Open Projects from File System or Archive and providing the path to the processor folder, *Ghidra/Processors/x86*. This will link your Eclipse instance to Ghidra's x86 processor module, meaning any changes you make in Eclipse will be directly modified in your actual Ghidra processor module.

Figure 18-5 shows a partially expanded version of the x86 module in Eclipse. It exactly reflects the associated Ghidra directory structure and contains the processor manual you downloaded, along with the x86 index file.

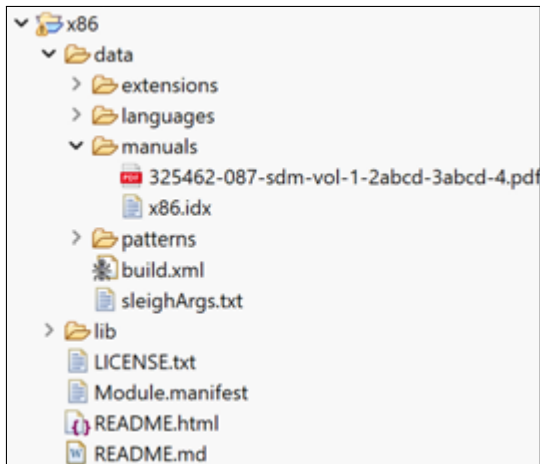


Figure 18-5: The x86 processor module in Eclipse Package Explorer

The *x86* folder contains a *data* folder, like the one you saw in the processor module you created using Eclipse ► GhidraDev. Within this folder is the *languages* folder, which contains over 45 files, including 22 *.sinc* files that define language instructions.

Because the x86 instruction set is rather large, the instruction set is broken up into files that group similar instructions. We'll add our instruction to an existing x86 *.sinc* file, but if we were adding a collection of new instructions (for example, the x86 SGX instruction set), we might instead create a new *.sinc* file to group them all together. (In fact, the SGX instructions all live in a common file called *sgx.sinc*. That accounts for one of the many *.sinc* files!)

By searching the *.sinc* files, we find that *ia.sinc* contains the definitions of the existing VMX instruction set. We'll use this definition as the model for defining VMXPLODE.

Two different sections in *ia.sinc* reference VMXOFF. The first section contains the definitions for the Intel IA hardware-assisted virtualization instructions:

```
# MFL: definitions for Intel IA hardware assisted virtualization instructions
define pcodeop invept;    # Invalidate Translations Derived from extended page
                        # tables (EPT); opcode 66 0f 38 80

# -----CONTENT OMITTED HERE-----

define pcodeop vmread;    # Read field from virtual-machine control structure;
                        # opcode 0f 78

define pcodeop vmwrite;   # Write field to virtual-machine control structure;
                        # opcode 0f 79

define pcodeop vmxoff;    # Leave VMX operation; opcode 0f 01 c4
define pcodeop vmxon;     # Enter VMX operation; opcode f3 0f C7 /6
```

Each entry in the definitions section defines a *pcodeop*, which is a new microcode operation for the x86 architecture.

The definition includes a name and, in this case, a comment that includes a description and an opcode. We will need to populate the comment for our new command. A quick alt-reality web search (with a side of testing) confirms that the opcode 0f 01 c5 has long been reserved for VMXPLDDE. We now have the information necessary to add our new instruction to the file. The following shows our new definition in context:

```
define pcodeop vmxoff;    # Leave VMX operation; opcode 0f 01 c4
define pcodeop vmxplode; # Explode (Fake) VMX operation; opcode 0f 01 c5
define pcodeop vmxon;     # Enter VMX operation; opcode f3 0f C7 /6
```

The second location in *ia.sinc* at which we encounter VMXOFF (and where we will insert our new instruction) is the opcode definition section. (We omitted part of this content for clarity and wrapped some instruction definition lines for readability.) While we won't completely dissect the 10,000-plus lines of code in the *ia.sinc* file, there are several interesting points to make regarding the following listing:

```
# Intel hardware assisted virtualization opcodes
# -----CONTENT OMITTED HERE-----
# TODO: invokes a VM function specified in EAX ❶
:VMFUNC EAX      is vexMode=0 & byte=0x0f; byte=0x01; byte=0xd4 & EAX      { vmfunc(EAX); }
# TODO: this launches the VM managed by the current VMCS. How is the
#       VMCS expressed for the emulator? For Ghidra analysis?
:VMLAUNCH        is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc2          { vmlaunch(); }
# TODO: this resumes the VM managed by the current VMCS. How is the
#       VMCS expressed for the emulator? For Ghidra analysis?
:VMRESUME        is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc3          { vmresume(); }
# -----CONTENT OMITTED HERE-----
:VMWRITE Reg32, rm32 is vexMode=0 & opsize=1 & byte=0x0f; byte=0x79; ❷
                  rm32 & Reg32 ... & check_Reg32_dest ... { vmwrite(rm32,Reg32);
                  build check_Reg32_dest; }
```

```

@ifdef IA64 ❸
:VMWRITE Reg64, rm64  \$(LONGMODE\_ON) \& is vexMode=0 & opsize=2 & byte=0x0f; byte=0x79;
    rm64 & Reg64 ...    { vmwrite(rm64,Reg64); }
@endif
:VMXOFF          is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc4          { vmxoff(); } ❹
:VMXPLODE        is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5          { vmxplode(); } ❺
# -----CONTENT OMITTED HERE-----
#END of changes for VMX opcodes

```

TODO comments ❶, found in many Ghidra files, identify tasks that have yet to be done. Searching for TODO tasks in Ghidra files is a great way to identify opportunities to contribute to this open source project.

Next, we see the VMWRITE instruction for 32-bit ❷ and 64-bit architectures. The 64-bit instruction is surrounded by a test ❸ to ensure it is included in only the 64-bit *.sla* file. While 32-bit instructions are valid in a 64-bit world (for example, EAX is the 32 least-significant bits of RAX), the converse is not true. The conditional statement ensures that instructions that operate on 64-bit registers are included for only 64-bit builds.

The VMXOFF instruction ❹ doesn't directly involve registers, so there is no need to distinguish between 32- and 64-bit versions of the instruction. The constructor for our new instruction, VMXPLODE ❺, complete with its new opcode, is very similar to the constructor for VMXOFF. Let's break this into the components that make up the line:

:VMXPLODE

This is the instruction being defined and will be displayed in the disassembly listing.

is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5

These are the bit patterns associated with the instruction and provide a constraint for the instruction. The & represents a logical AND operation. The semicolons serve a dual purpose of concatenation and logical AND. Put together, this part says, "If we are not in VEX mode and the opcode consists of these 3 bytes in this order, then this constraint is met."

{ vmxplode(); }

Curly brackets enclose the semantic actions section of an instruction. The SLEIGH compiler translates these actions into an internal Ghidra form known as p-code, discussed later in this chapter. Defining an instruction requires understanding SLEIGH operators and syntax. This portion of the constructor, which performs the real work associated with most instructions, can quickly become a complex sequence of multiple statements separated by semicolons. In this case, since we have defined VMXPLODE as a new p-code operation (define pcodeop vmxplode;), we can invoke the instruction here. In future examples, we will add additional SLEIGH semantic actions to this section.

The largest x86 *.sinc* file, *ia.sinc*, defines many instructions (including our new VMXPLODE instruction), as well as key attributes of the processor (for example, endianness, registers, contexts, tokens, and variables). Much of

this foundational x86-specific content is not replicated in the other *.sinc* files in this directory, as all the *.sinc* files are in turn included in a SLEIGH specification (*.slaspec*) file.

The current SLEIGH setup for x86 uses two *.slaspec* files. The first file, *x86.slaspec*, is the 32-bit specification. The second, *x86-64.slaspec*, is the 64-bit specification and builds on the *x86.slaspec*. Both rely on *.sinc* files for modularity. For smaller processors with fewer instructions, it might make sense to write all the content directly in a single *.slaspec* file, but for x86, the modular approach organizes the content into manageable chunks.

Here are the contents of the *x86.slaspec* file:

```
❶ @include "ia.sinc"
❷ @include "lockable.sinc"
❸ with : lockprefix=0 {
    @include "avx.sinc"
    @include "avx_manual.sinc"
    @include "avx2.sinc"
    @include "avx2_manual.sinc"
    @include "adx.sinc"
    @include "clwb.sinc"
    @include "pclmulqdq.sinc"
    @include "mpx.sinc"
    @include "lzcnt.sinc"
    @include "bmi1.sinc"
    @include "bmi2.sinc"
    @include "sha.sinc"
    @include "smx.sinc"
    @include "cet.sinc"
    @include "rdrand.sinc"
}
```

This structure shows how *ia.sinc* ❶ provides the foundation of the file, while *lockable.sinc* ❷ and the `with : lockprefix=0 { ... }` ❸ block bring in instruction set extensions, such as AVX, AVX-512, BMI, and SHA.

Here are the contents of the *x86-64.slaspec* file:

```
@define IA64 "IA64"
@include "x86.slaspec"
with : lockprefix=0 {
    @include "sgx.sinc"
    @include "fma.sinc"
}
```

The *x86-64.slaspec* file directly includes *x86.slaspec*, meaning the 64-bit specification inherits everything from the 32-bit definition and simply extends it with SGX and FMA instruction set extensions. The list of included files already contains *ia.sinc* ❶, in which we defined `VMXPLD`, so we don't need to add anything here. If you created a new *.sinc* file, however, you would

need to add an include statement in *x86.slaspec* in order for the instruction to be recognized in both 32- and 64-bit binaries.

To test whether Ghidra can recognize the new instruction when used in a binary, we construct a test file. The file will first verify that Ghidra continues to recognize the VMXOFF instruction and then verify that we have successfully added VMXPLODE. The C source file for testing VMXOFF contains the following:

```
#include <stdio.h>

// The following function declares an assembly block and tells the
// compiler that it should execute the code without moving or changing it.

void do_vmx(int v) {
    asm volatile (
        "vmxon %0;"          // Enable hypervisor operation.
        "vmxoff;"           // Disable hypervisor operation.
        "nop;"              // Tiny nop slide to accommodate examples
        "nop;"
        "nop;"
        "nop;"
        "nop;"
        "nop;"
        "vmxoff;"           // Disable hypervisor operation.
        :
        : "m"(v)            // Holds the input variable
        :
    );
}

int main() {
    int x;
    printf("Enter an int: ");
    scanf("%d", &x);
    printf("After input, x=%d\n", x);
    do_vmx(x);
    printf("After do_vmx, x=%d\n", x);
    return 0;
}
```

When we load the compiled binary into Ghidra, we see the following body of the `do_vmx` function in the Listing window:

0010071a	55	PUSH	RBP
0010071b	48 89 e5	MOV	RBP, RSP
0010071e	89 7d fc	MOV	dword ptr [RBP + local_c], EDI
00100721	f3 0f c7	VMXON	qword ptr [RBP + local_c]
	75 fc		
❶ 00100726	0f 01 c4	VMXOFF	

```

00100729 90      NOP
0010072a 90      NOP
0010072b 90      NOP
0010072c 90      NOP
0010072d 90      NOP
0010072e 90      NOP
0010072f 90      NOP
❷ 00100730 0f 01 c4  VMXOFF
00100733 90      NOP
00100734 5d      POP RBP
00100735 c3      RET

```

The bytes displayed for the opcode (0f 01 c4) in the two calls to VMXOFF ❶ ❷ match the opcode we observed in *ia.sinc* for this command. The following listing from the Decompiler window is consistent with what we know about the source code and the associated disassembly:

```

void do_vmx(undefined4 param_1)
{
    undefined4 unaff_EBP;

    vmxon(CONCAT44(unaff_EBP,param_1));
    vmxoff();
    vmxoff();
    return;
}

```

To test that Ghidra detects the VMXPLDE instruction, we replace the first occurrence of VMXOFF in the `do_vmx` test function with VMXPLDE. However, the VMXPLDE instruction is missing not only from Ghidra's processor definition but also from our compiler's knowledge base. In order for the assembler to accept our code, we hand-assembled the instruction using a data declaration, instead of using the instruction mnemonic directly, so that the assembler can process the new instruction:

```

// "vmxoff;" // Replace this line
".byte 0x0f, 0x01, 0xc5;" // with this hand-assembled one.

```

When we load the updated binary into Ghidra, we see the following in the Listing window:

```

0010071a 55 PUSH RBP
0010071b 48 89 e5 MOV RBP,RSP
0010071e 89 7d fc MOV dword ptr [RBP + local_c],EDI
00100721 f3 0f c7 VMXON qword ptr [RBP + local_c]
          75 fc
❶ 00100726 0f 01 c5 VMXPLDE

```

```

00100729 90 NOP
0010072a 90 NOP
0010072b 90 NOP
0010072c 90 NOP
0010072d 90 NOP
0010072e 90 NOP
0010072f 90 NOP
00100730 0f 01 c4 VMXOFF
00100733 90 NOP
00100734 5d POP RBP
00100735 c3 RET

```

Your new instruction ❶ appears along with the opcode (0f 01 c5) we have assigned to it. The Decompiler window also shows the new instruction:

```

void do_vmx(undefined4 param_1)
{
    undefined4 unaff_EBP;

    vmxon(CONCAT44(unaff_EBP,param_1));
    vmxplode();
    vmxoff();
    return;
}

```

So, what work has Ghidra undertaken in the background to add our new instruction to its x86 processor instruction set? When we restart Ghidra to make the changes take effect, Ghidra detects that the underlying *.sinc* file changed and generates a new *.sla* file when one is needed. In this case, it displays the window shown in Figure 18-6 while it recompiles the *ia.sinc* file.

Note that Ghidra recompiles files only when needed, not automatically on restart. Because we loaded a 64-bit file, only *x86-64.sla* was updated and not *x86.sla*. Later, when we loaded the updated file, complete with the VMXPLODE command, Ghidra did not recompile the file, as we made no changes to any underlying SLEIGH source files since the previous load.



Figure 18-6: The Ghidra window displayed while recompiling a language file

Here is a summary of the steps used to add a new instruction to a processor module:

1. Locate the *languages* directory for the target processor (for example, *Ghidra/Processor/<targetprocessor>/data/languages*).
2. Add the instruction to a selected processor *.sinc* file, or create a new *.sinc* file (for example, *Ghidra/Processor/<targetprocessor>/data/languages/<targetprocessor>.sinc*).
3. If you created a new *.sinc* file, make sure it is included in the *.slaspec* file (for example, *Ghidra/Processor/<targetprocessor>/data/languages/<targetprocessor>.slaspec*).

Example 2: Modifying an Instruction in a Processor Module

We have now successfully added an instruction to the Ghidra x86 processor module, but we have not yet accomplished our goal of making VMXPLODE leave with a flourish. Currently, it just exits without any excitement whatsoever. While it is challenging to make an assembly language instruction do anything that would qualify as a flourish, we can attempt to make our instruction dab when it exits.

NOTE

A dab (or dabbing) was a celebratory dance move used by international sports figures starting in 2012. We chose it for this example as it is one of the few dance moves that can be spelled correctly using only hexadecimal digits.

In this example, we will step through three options for making VMXPLODE dab for us.

Option 1: Setting EAX to a Constant Value

Our first option is exiting after setting EAX to a hardcoded value: 0xDAB. Having the VMXPLODE instruction set the value of EAX to 0xDAB prior to exiting requires only a minor modification to one instruction in the same *ia.sinc* file we worked with in Example 1. The following listing shows the VMXOFF and VMXPLODE instructions as we left them after Example 1:

:VMXOFF	is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc4	{ vmxoff(); }
:VMXPLODE	is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5	{ vmxplode(); }

Within the instruction contents, we add the assignment to EAX immediately before the vmxplode action, as shown in the following listing:

:VMXOFF	is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc4	{ vmxoff(); }
:VMXPLODE	is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5	{ EAX=0xDAB; vmxplode(); }

When we restart Ghidra and load our test file, Ghidra once again displays a window letting us know that it has detected a change in an associated language file and is regenerating *x86-64.sla*. The Listing window doesn't show any changes after Ghidra auto analyzes the file, but the difference is apparent in the Decompiler window:

```
undefined4 do_vmx(undefined4 param_1)
{
    undefined4 unaff_EBP;

    vmxon(CONCAT44(unaff_EBP,param_1));
    vmxplode();
    vmxoff();
    return 0xdab;
}
```

In the Decompiler window, the return statement now returns the contents of EAX (0xDAB). This is interesting because we know this is a void function and doesn't have a return value. The Listing window entry for the new instruction doesn't show that the VMXPLODE command has changed in any way:

00100726 0f 01 c5	VMXPLODE

An important distinction between decompilers and disassemblers is that decompilers understand and incorporate the full semantic behavior of each instruction as part of their analysis, while disassemblers are focused largely on the proper syntactic representation of each instruction. In this example, VMXPLODE takes no operands, and the disassembler displays it correctly, providing no visual cue that EAX has changed. When reading a disassembly, it is entirely your responsibility to understand the semantic behavior of each instruction.

This example also demonstrates the value of the decompiler, which understands the full semantics of VMXPLODE and so is able to recognize that EAX has changed as a side effect of the instruction. In addition, the decompiler recognizes that the remainder of the function does not use EAX and assumes that the value is intended to be returned to the calling function.

Ghidra offers you the opportunity to dive a little deeper into how instructions work and allows you to detect and test subtle differences in instructions like this one. First, let's look at some of the instruction information associated with VMXPLODE, shown in Figure 18-7.

Instruction Info: Address 00100726 - (vmx_test_01b_v	Instruction Info: Address 00100726 - (vmx_test_01b_v
Instruction Summary ----- Mnemonic : VMXPLODE Number of Operands: 0 Address : ram:00100726 Flow Type : FALL_THROUGH Fallthrough : 00100729 Delay slot depth : 0 Hash : fc3d0a7d Input Objects: const:0x27 Result Objects: Constructor Line #'s: VMXPLODE(ia.sinc:4906)	Instruction Summary ----- Mnemonic : VMXPLODE Number of Operands: 0 Address : ram:00100726 Flow Type : FALL_THROUGH Fallthrough : 00100729 Delay slot depth : 0 Hash : fc3d0a7d Input Objects: const:0x27, const:0xdab ❶ Result Objects: EAX ❷ Constructor Line #'s: VMXPLODE(ia.sinc:4906)

Figure 18-7: The VMXPLODE Instruction Info windows

On the left is our original VMXPLODE instruction, and on the right is the modified version, with 0xdab listed in the Input Objects ❶ section and EAX under Result Objects ❷. We can obtain additional insight about any instruction by looking at underlying information, called p-code, that we haven't looked at previously. The p-code associated with an instruction can be very informative about what exactly an instruction does.

NOTE

P-code appears without the hyphen (pcode) in Ghidra Help, and with the hyphen in most other locations, including the p-code documentation. If you are having trouble finding information about p-code within Ghidra, try searching for pcode without the hyphen.

P-CODE: HOW LOW CAN YOU GO?

The Ghidra documentation describes p-code as a “register transfer language designed for reverse engineering applications.” A *register transfer language (RTL)* is an architecture-independent, assembly-language-like language often used as an intermediate representation (IR), or intermediate language between a high-level language such as C and a target assembly language such as x86 or ARM.

Compilers are often composed of a language-specific frontend that translates source code into an IR, and an architecture-specific backend that translates IR into a specific assembly language. This modularity allows you to combine a C frontend with an x86 backend to create a C compiler that produces x86 code, or to replace the backend with an ARM module to instantly have a C compiler that generates ARM code. Similarly, if you swap out the C frontend for a FORTRAN frontend, you now have a FORTRAN compiler for ARM.

Working at the IR level allows us to build tools that operate on our IR rather than maintaining a set of C-specific or ARM-specific tools that are useless to us with other languages or architectures. For example, once we have an optimizer that operates on IR, we can reuse that optimizer with any of our frontend/backend combinations without rewriting the optimizer in each case.

A reverse engineering toolchain, not surprisingly, runs in the opposite direction of a traditional software build chain. A reverse engineering frontend needs to translate machine code to IR (a process often called *lifting*), while a reverse engineering backend translates IR to a high-level language such as C. A pure disassembler doesn't qualify as a frontend under this definition, as it gets us only from machine code to assembly language. Ghidra's decompiler is an IR-to-C backend, and Ghidra processor modules are machine-code-to-IR frontends.

When you build or modify a Ghidra processor module in SLEIGH, one of the first things you do is let the SLEIGH compiler know about any new p-code operations that you need to introduce in order to describe the semantic actions of any new or modified instructions. For example, the operation definition

```
define pcodeop vmxplode
```

we added to the *ia.sinc* file tells the SLEIGH compiler that *vmxplode* is a valid semantic action available for describing the behavior of any instruction in our architecture. One of the most difficult challenges you will face is describing each new or changed instruction using a sequence of syntactically correct SLEIGH statements that correctly describe the actions associated with the instruction. All of this information is captured in the *.laspec* and included *.sinc* files that make up your processor. If you do a good enough job, Ghidra will hand you the decompiler backend for free.

To view the p-code within the Listing window, open the Browser Field Formatter and choose the Instruction/Data tab, right-click the p-code bar, and enable the field. Once the Listing window displays the p-code associated with each instruction, we can compare the previous two listings to observe any differences. With p-code enabled, our first implementation of *VMXPLODE* appears as follows, with the p-code displayed below each instruction:

0010071a 55	PUSH	RBP	\$U27d00:8 = COPY RBP RSP = INT_SUB RSP, 8:8 STORE ram(RSP), \$U27d00:8
0010071b 48 89 e5	MOV	RBP,RSP	RBP = COPY RSP
0010071e 89 7d fc	MOV	dword ptr [RBP + local_c],EDI	\$U4700:8 = INT_ADD RBP, -4:8 \$U6a80:4 = COPY EDI STORE ram(\$U4700:8), \$U6a80:4
00100721 f3 0f c7 75 fc VMXON		qword ptr [RBP + local_c]	\$U4700:8 = INT_ADD RBP, -4:8 \$U6b00:8 = LOAD ram(\$U4700:8) CALLOTHER "vmxon", \$U6b00:8
00100726 0f 01 c5	VMXPLODE		CALLOTHER "vmxplode"
00100729 90	NOP		

And here is the modified VMXPLODE:

```
00100726 0f 01 c5      VMXPLODE
                                ❶ EAX = COPY 0xdab:4
                                CALLOTHER "vmxplode"
```

The associated p-code now shows the constant value (0xdab) being moved into EAX ❶.

Option 2: Setting a Register Determined by an Operand to a Constant Value

Instruction sets typically include a mix of individual instructions that operate on zero or more operands. As the number and types of operands associated with an instruction increase, so too does the level of difficulty in describing the instruction's semantics. In this example, we'll extend the behavior of VMXPLODE to require a single register operand, and then make that register dab.

This option will require us to visit sections of the *ia.sinc* file we have not previously encountered. Let's start with a modified version of the instruction and then work backward. The following listing shows the modifications we need to make to our instruction definition to accept an operand that will identify the register that ultimately will hold 0xDAB:

```
:VMXPLODE ❶ Reg32 is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5; ❷ Reg32
{ ❸ Reg32=0xDAB; vmxplode(); }
```

We declare Reg32 ❶ as a local identifier and then concatenate it with the opcode ❷ to make it part of the constraints associated with the instruction. Rather than assigning 0xDAB directly into EAX as we did previously, the instruction now assigns the value into Reg32 ❸.

To accomplish our goal, we will need to determine a way to associate the value in Reg32 with the x86 register of our choosing. Let's investigate other components within *ia.sinc* to help us understand how to correctly map an operand to a specific x86 general-purpose register.

Near the start of *ia.sinc*, we see all of the definitions that the entire specification will need, shown in Listing 18-1.

```
# SLA specification file for Intel x86
#ifdef IA64 ❶
#define SIZE      "8"
#define STACKPTR "RSP"
#else
#define SIZE      "4"
#define STACKPTR "ESP"
#endif
define endian=little; ❷
define space ram type=ram_space size=$(SIZE) default;
define space register type=register_space size=4;
# General purpose registers ❸
#ifdef IA64
```



```

define register offset=0 size=8 [ RAX   RCX   RDX   RBX   RSP   RBP   RSI   RDI ]; ❹
define register offset=0 size=4 [ EAX _  ECX _  EDX _  EBX _  ESP _  EBP _  ESI _  EDI ];
define register offset=0 size=2 [ AX   _  _  CX   _  _  DX   _  _  BX ]; # Truncated
define register offset=0 size=1 [ AL AH _  _  _  _  CL CH _  _  _  _ ]; # Truncated
define register offset=0x80 size=8 [ R8   R9   R10  R11  R12  R13  R14  R15 ]; ❺
define register offset=0x80 size=4 [ R8D  R9D  R10D R11D R12D R13D R14D R15D ];
define register offset=0x80 size=2 [ R8W  _  _  R9W  _  _  R10W _  _  R11W ]; # Truncated
define register offset=0x80 size=1 [ R8B  _  _  _  _  _  _  R9B  _  _  _  _ ]; # Truncated
@else
define register offset=0 size=4 [ EAX ECX EDX EBX ESP EBP ESI EDI ];
define register offset=0 size=2 [ AX _ CX _ DX _ BX _ SP _ BP _ SI _ DI ];
define register offset=0 size=1 [ AL AH _ _ CL CH _ _ DL DH _ _ BL BH ];
@endif

```

Listing 18-1: A partial SLEIGH specification for x86 registers (adapted from ia.sinc)

At the top of the file, we see the name and size of the stack pointer for 32- and 64-bit builds ❶, as well as the endianness ❷ for the x86. A comment ❸ introduces the start of the definitions of the general-purpose registers.

As with all its other components, SLEIGH has a special convention for naming and defining registers: registers reside in a special address space named register, and every register (which may span 1 or more bytes) is assigned an offset within the address space. A SLEIGH register definition indicates the offset at which a list of registers begins within the register address space. All registers in a register list are contiguous unless an underscore creates space between them. Figure 18-8 shows the address space layout of the 64-bit RAX and RCX registers ❹.

Size	Offset															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	RAX								RCX							
4	EAX				—				ECX				—			
2	AX		—		—		—		CX		—		—		—	
1	AL	AH	—	—	—	—	—	—	CL	CH	—	—	—	—	—	—

Figure 18-8: The register layout for x86-64 RAX and RCX registers

The register named AL occupies exactly the same location as the least significant byte of RAX, EAX, and AX (since x86 is a little-endian). Similarly, EAX occupies the low 4 bytes of RAX. An underscore indicates that no name is associated with a given range of bytes for the given size. In this case, there is no name for the 4-byte block at offsets four to seven, although these bytes are synonymous with the upper half of the RAX register.

Listing 18-1 describes a separate block of registers beginning with R8 at offset 0x80 ❺. The 1-byte register at offset 0x80 is known as R8B, and the 1-byte register at offset 0x88 is known as R9B. Hopefully, the similarity between the textual register definition in Listing 18-1 and the tabular representation in

Figure 18-8 are obvious since the register definitions in a SLEIGH file are nothing more than the textual representation of an architecture's register address space.

If you are writing a SLEIGH description of an architecture that is entirely unsupported by Ghidra, it will be your job to lay out the register address space for that architecture, ensuring that there is no overlap between registers unless the architecture requires it (as in the case of RAX, EAX, AX, AH, and AL in the x86-64 architecture).

Now that you understand how registers are represented in SLEIGH, let's return to our objective of choosing a register to dab. In order for our instruction to function properly, it needs to map the identifier `Reg32` to a general-purpose register. To accomplish this task, we can use an existing definition in *ia.sinc*, found within the following lines of code:

```
❶ define token modrm (8)
    mod          = (6,7)
    reg_opcode    = (3,5)
    reg_opcode_hb = (5,5)
    r_m          = (0,2)
    row          = (4,7)
    col          = (0,2)
    page         = (3,3)
    cond         = (0,3)
    reg8         = (3,5)
    reg16        = (3,5)
    ❷ reg32       = (3,5)
    reg64        = (3,5)
    reg8_x0      = (3,5)
```

The define statement ❶ is declaring an 8-bit token called `modrm`. A SLEIGH token is a syntactic element used to represent byte-sized components that make up the instructions being modeled. You can find this concept described in detail in the Tokens and Fields (6) section of the SLEIGH documentation.

SLEIGH allows the definition of any number of bitfields (a range of one or more contiguous bits) within a token. When you're defining instructions in SLEIGH, these bitfields provide a convenient, symbolic means of specifying the associated operands. In this listing, a bitfield named `reg32` ❷ spans bits 3 through 5 of `modrm`. This 3-bit field can take on the values 0 to 7 and can be used to choose one of the eight 32-bit x86 registers.

If we move to the next reference of `reg32` in the file, we see the following interesting line of code with comments added for clarity:

```
# attach variables fieldlist registerlist;
  attach variables [ r32  reg32  base  index ] [ EAX ECX EDX EBX ESP EBP ESI EDI ];
#                                           0   1   2   3   4   5   6   7
```

The first and last lines of the listing contain comments that show the SLEIGH syntax for this statement and the ordinal values for each register. The `attach variables` statement associates the field with a list (in this case,

a list of the x86 general-purpose registers). Here is a rough interpretation of the line of code, taking the preceding `modrm` definition into account: It determines the value of `reg32` by looking at bits 3 to 5 of the token `modrm` and then uses the resulting value (0 to 7) as an index to select a register from the list.

We now have a way to identify the general-purpose registers to target for `0xDAB`. Our next encounter with `Reg32` within the file finds the following non-adjacent lines of code, with comments added. These contain the constructor for `Reg32` for both 32- and 64-bit registers, letting us see the association between `reg32` and `Reg32`:

```
Reg32:    reg32  is rexRprefix=0 & reg32  { export reg32; } #64-bit Reg32
Reg32:    reg32  is reg32                  { export reg32; } #32-bit Reg32
```

Let's return to the command that started this little adventure:

```
:VMXPLODE Reg32 is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5; Reg32
                                { Reg32=0xDAB; vmxplode(); }
```

We will include an operand with our call to `VMXPLODE` that will determine which register gets the value `0xDAB`. We will update our test binary further by removing the first `NOP` and appending the value `0x08` to our hand-assembled instruction. The first 3 bytes are the opcode (`0f 01 c5`), and the following byte (`08`) will be the operand that specifies the register to use:

```
".byte 0x0f, 0x01, 0xc5, 0x08;" // Hand-assembled with operand
```

Figure 18-9 demonstrates the step-by-step translation from the operand through to the determination of the register based on the information in the *ia.sinc* file.

❶	Operand	08							
❷	Value	0	0	0	0	1	0	0	0
❸	modrm bits	7	6	5	4	3	2	1	0
❹	Reg32	001							
❺	Ordinals	0	1	2	3	4	5	6	7
❻	Registers	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

Figure 18-9: The translation path from operand to register

The original operand value, shown in the first row, is `08` ❶. The value is decoded into its binary ❷ form and overlaid with the fields of the `modrm` token ❸. Bits 3 to 5 are extracted, yielding the `Reg32` value `001` ❹. This value is used to index the ordinal map ❺ to select the `ECX` register ❻. Therefore, the operand `0x08` specifies that `ECX` will get the value `0xDAB`.

When we save the updated *ia.sinc* file, restart Ghidra, and then load and analyze the file, Ghidra generates the following listing showing the use of our new instruction. As expected, ECX is the register selected to hold 0xDAB:

```

00100721 f3 0f c7 75 fc VMXON      qword ptr [RBP + local_c]
                                   $U4700:8 = INT_ADD RBP, -4:8
                                   $U6b00:8 = LOAD ram($U4700:8)
                                   CALLOTHER "vmxon", $U6b00:8

00100726 0f 01 c5 08   VMXPLODE   ECX
                                   ECX = COPY 0xdab:4
                                   CALLOTHER "vmxplore"

0010072a 90              NOP

```

The value 0xDAB no longer appears in the Decompiler window because the decompiler assumes that the return value is in EAX. In this case, we are using ECX, so the decompiler does not identify a return value.

Now that we can make a selected register dab, let's add a 32-bit immediate value as a second operand. This will double our celebratory potential.

Option 3: Including Register and Value Operands

To extend the syntax of our instruction to take two operands, a destination register and a source constant, we need to update the definition of VMXPLODE, as shown here:

```

:VMXPLODE Reg32,imm32 is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5;
                Reg32; imm32                { Reg32=imm32; vmxplore(); }

```

The addition of an immediate 32-bit constant to the instruction requires 4 additional bytes to encode. Accordingly, we replace the next four NOPs with values that correctly encode our *imm32* in little-endian order, as seen here:

```

".byte 0x0f, 0x01, 0xc5, 0x08, 0xb8, 0xdb, 0xee, 0x0f;"
"nop;"
"nop;"

```

When we reload the file, VMXPLODE exits with another flourish, as shown in the following listing, with p-code displayed:

```

00100726 0f 01 c5      VMXPLODE ECX,0xfeedbb8
                08 b8 db
                ee 0f
                                   ECX = COPY 0xfeedbb8:4
                                   CALLOTHER "vmxplore"

```

The ECX register now has the value 0xFEEDBB8 (which might be a more appealing exit flourish for science fiction fans).

Example 3: Adding a Register to a Processor Module

We close out our processor module examples by extending an architecture with two entirely new registers, a concept described in detail starting in the Naming Registers (4.4) section of the SLEIGH documentation. Recall the definition of the 32-bit general-purpose registers from earlier in the chapter:

```
define register offset=0 size=4 [ EAX ECX EDX EBX ESP EBP ESI EDI ];
```

This definition requires an offset, a size, and the list of registers. We chose a starting offset into the registry memory address space after reviewing the currently allocated offsets and finding the space we need for two 4-byte registers. We can use this information to define two new 32-bit registers in the *ia.sinc* file, called VMID and VMVER, as shown in the following listing:

```
# Define VMID and VMVER.  
define register offset=0x2500 size=4 [ VMID VMVER ];
```

Our instructions need a way to identify which new register (VMID or VMVER) they are operating on. In the previous example, we used a 3-bit field to select one of eight registers, but selecting between the two new registers requires only a single bit. The following statement defines a 1-bit field within the *modrm* token and associates the field with *vmreg*:

```
# Associate vmreg with a single bit in the modrm token.  
vmreg = (3, 3)
```

The following statement attaches *vmreg* to the ordinal set containing the two registers, with 0 representing VMID and 1 representing VMVER:

```
attach variables [ vmreg ] [ VMID VMVER ];
```

Instruction definitions may refer to *vmreg* when any of the attached registers are valid within the instruction, while assembly language programmers may refer to VMID and VMER as operands in any instruction that allows a *vmreg* operand. Let's compare the following two definitions of *VMXPLODE*. The first is from our previous example, where we chose the register from among the general-purpose registers, and the second selects one of our two registers rather than any of the general-purpose registers:

```
:VMXPLODE Reg32,imm32 is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5;  
                Reg32, imm32                { Reg32=imm32; vmxplode(); }  
:VMXPLODE vmreg,imm32 is vexMode=0 & byte=0x0f; byte=0x01; byte=0xc5;  
                vmreg, imm32                { vmreg=imm32; vmxplode(); }
```

The second listing replaces Reg32 with vmreg. If we use the same input file with the test instruction `vmxplode 0x08,0xFEEDBB8`, the immediate operand `0xFEEDBB8` will be loaded into VMVER since the input value `0x08` maps to an ordinal value of 1 (because bit 3 is set) and VMVER is register 1 in vmreg. After saving *ia.sinc*, restarting Ghidra, and loading the test file, we see that the p-code in the Listing window shows the immediate operand loaded into VMVER:

```
00100726 0f 01 c5      VMXPLODE  VMVER,0xfeedbb8
          08 b8 db
          ee 0f

VMVER = COPY 0xfeedbb8:4
CALLOTHER "vmxplode"
```

You could also confirm the change by viewing the Instruction Info window.

Summary

While we introduced only a small fraction of the x86 processor file contents in this chapter, we looked at the major components of a processor module, including instruction definitions, register definitions, and tokens, as well as how the Ghidra-specific language, SLEIGH, can be used to build, modify, and augment Ghidra processor modules. If you have a desire (or need) to add a new processor to Ghidra, we highly recommend looking at some of the more recent processors added to Ghidra. The *SuperH4.sinc* file is particularly well documented, and the processor is significantly less complex than the x86 processor.

We cannot emphasize enough the role that patience and experimentation play in any processor-development situation. This hard work will more than pay off when you are able to reuse your processor module with each new binary you collect and potentially contribute the module to the Ghidra project for the benefit of other reverse engineers.

In the next chapter, we take a deep dive into the functionality associated with the Ghidra Decompiler.

19

THE DECOMPILER



Until now, we have focused our reverse engineering analysis primarily on the Listing window and presented Ghidra's features through the disassembly listing lens. In this chapter, we shift our focus to the Decompiler window and investigate how we can accomplish familiar analysis tasks (and some new ones) with the decompiler and its associated functionality.

We start with a brief overview of the decompilation process before moving on to the functionality available in the Decompiler window. We then walk through some examples to help you identify ways that the Decompiler window can improve your reverse engineering process.

Decompiler Analysis

It would be logical to assume that the Decompiler window derives its content from the Listing window, but, in fact, Ghidra produces the contents of the Listing and Decompiler windows independently, which is why they sometimes disagree and why you should evaluate both in context when you're trying to determine ground truth. The main function of Ghidra's decompiler is to convert machine language instructions into p-code (discussed in Chapter 18), and then to convert the p-code to C and present it in the Decompiler window.

In a simplified view, the decompilation process includes three distinct phases. First, the decompiler uses the SLEIGH specification file to create a draft of the p-code and derive associated basic blocks and flows. The second phase focuses on simplification: It eliminates unneeded content, such as unreachable code, and adjusts control flows in response to the changes. The wrap-up phase adds finishing touches, makes some final checks, and sends the final results through a pretty-printing algorithm before presenting them in the Decompiler window.

Of course, this explanation greatly simplifies a very complex process, but the main takeaways are the following:

- The Decompiler is an analyzer.
- It starts its work with the binary and produces p-code.
- It converts the p-code to C.
- The C code and any associated messages are displayed in the Decompiler window.

We discuss some of these steps in more detail as we navigate through Ghidra's decompilation functionality. Let's start our investigation with the analysis process and the primary capabilities it unleashes.

Analysis Options

During the auto analysis process, several analyzers pertain to the Decompiler window. You can manage these through the Edit ► Tool Options menu, shown in Figure 19-1 with defaults selected.

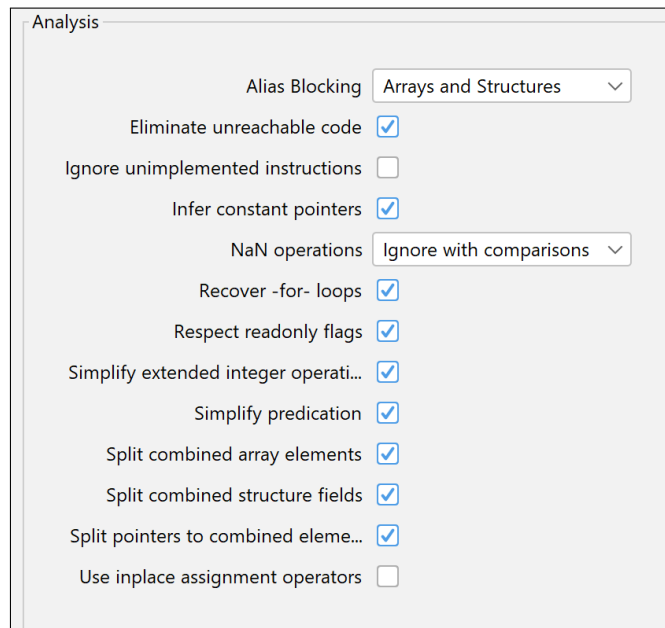


Figure 19-1: The Decompiler analysis options with defaults selected

We discuss two of these options, “Eliminate unreachable code” and “Simplify predication,” next. To understand the remaining options, experiment with their results or refer to Ghidra Help.

Eliminating Unreachable Code

The “Eliminate unreachable code” option excludes unreachable code from the Decompiler listing. For example, the following C function has two conditions that can never be met, which makes the corresponding conditional blocks unreachable:

```
int demo_unreachable(volatile int a) {
    volatile int b = a ^ a;
    ❶ if (b) {
        printf("This is unreachable\n");
        a += 1;
    }
    ❷ if (a - a > 0) {
        printf("This should be unreachable too\n");
        a += 1;
    } else {
        printf("We should always see this\n");
        a += 2;
    }
    printf("End of demo_unreachable()\n");
    return a;
}
```

This code initializes the variable `b` to zero in a perhaps less-than-obvious manner. When `b` is tested ❶, its value can never be nonzero, so the body of the corresponding if statement will never execute. Similarly, `a - a` can never be greater than zero, so the condition in the second if statement ❷ can never evaluate to true.

When we select the “Eliminate unreachable code” option, the Decompiler window displays warning messages:

```
/* WARNING: Removing unreachable block (ram,0x00100777) */
/* WARNING: Removing unreachable block (ram,0x0010079a) */
int demo_unreachable(int param_1)
{
    puts("We should always see this");
    puts("End of demo_unreachable()");
    return (param_1 + 2);
}
```

These messages let us know it has removed unreachable code.

Simplifying Predication

The “Simplify predication” option optimizes if-else blocks by merging blocks that share the same condition. In the following listing, the first two if statements share the same condition:

```
int demo_simppred(int a) {
    if (a > 0) {
        printf("A is > 0\n");
    }
    if (a > 0) {
        printf("Yes, A is definitely > 0!\n");
    }
    if (a > 2) {
        printf("A > 2\n");
    }
    return a * 10;
}
```

Here is what the Decompiler listing shows when we enable the “Simplify predication” option:

```
int demo_simppred(int param_1)
{
    if (0 < param_1) {
        puts("A is > 0");
        puts("Yes, A is definitely > 0!");
    }
    if (2 < param_1) {
        puts("A > 2");
    }
    return param_1 * 10;
}
```

In this listing, the decompiler has combined the blocks.

The Decompiler Window

Now that you understand how the Decompiler Analysis Engine populates the Decompiler window, let’s see how you can use the window to facilitate your analysis. Navigating the Decompiler window is relatively easy, as it displays only one function at a time. To move between functions or see the function in context, it is helpful to correlate it with the Listing window. Because the Decompiler window and the Listing window are linked by default, you can navigate both by using the available options in the CodeBrowser toolbar.

The function displayed in the Decompiler window helps with analysis, but it may not be so easy to read at first. Any lack of information about the data types used by the functions that it decompiles requires Ghidra to infer

those data types itself. As a result, the decompiler may overuse type casts, as you can see in the following sample statements:

```
printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d\n", (ulong)param_1,
      (ulong)param_2, (ulong)uVar1, (ulong)uVar2, (ulong)(uVar1 + param_1),
      (ulong)(uVar2 * 100), (ulong)uVar4);

uStack44 = *(undefined4 *)** (undefined4 **)(iStack24 + 0x10);
```

As you provide more accurate type information using the decompiler editing options, you will notice that the decompiler relies less and less on type casts, and the generated C code becomes easier to read. In the examples that follow, we'll discuss some of the Decompiler window's most useful features for cleaning up the generated source code. The ultimate goal is to produce readable source code that makes it easier to understand the code's behavior.

Example 1: Editing in the Decompiler Window

Consider a program that accepts two integer values from the user and then calls the following function:

```
int do_math(int a, int b) {

    int c, d, e, f, g;
    srand(time(0));

    c = rand();
    printf("c=%d\n", c);

    d = a + b + c;
    printf("d=%d\n", d);

    e = a + c;
    printf("e=%d\n", e);

    f = d * 100;
    printf("f=%d\n", f);

    g = rand() - e;
    printf("g=%d\n", g);

    printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d\n", a, b, c, d, e, f, g);

    return g;
}
```

The function uses two integer parameters with five local variables to generate its output. We can sum up the interdependencies as follows:

- Variable `c` depends on the `rand()` return value, influences `d` and `e` directly, and influences `f` and `g` indirectly.
- Variable `d` depends on `a`, `b`, and `c`, and influences `f` directly.
- Variable `e` depends on `a` and `c`, and influences `g` directly.
- Variable `f` depends on `d` directly and on `a`, `b`, and `c` indirectly, and influences nothing.
- Variable `g` depends on `e` directly and on `a` and `c` indirectly, and influences nothing.

When the associated binary is loaded into Ghidra and the function is analyzed, you see the following representation of the `do_math` function in the Decompiler window:

```
uint do_math(uint param_1,uint param_2)

{
    uint uVar1;
    uint uVar2;
    uint uVar3;
    int iVar4;
    uint uVar5;
    time_t tVar6;

    tVar6 = time((time_t *)0x0);
    srand((uint)tVar6);
    uVar1 = rand();
    printf("c=%d\n", (ulong)uVar1);
    uVar2 = uVar1 + param_1 + param_2;
    printf("d=%d\n", (ulong)uVar2);
    uVar3 = uVar1 + param_1;
    printf("e=%d\n", (ulong)uVar3);
    printf("f=%d\n", (ulong)(uVar2 * 100));
    iVar4 = rand();
    uVar5 = iVar4 - uVar3;
    printf("g=%d\n", (ulong)uVar5);
    printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d\n", (ulong)param_1,
        (ulong)param_2, (ulong)uVar1, (ulong)uVar2, (ulong)uVar3,
        (ulong)(uVar2 * 100), (ulong)uVar5);
    return uVar5;
}
```

If you want to do your analysis using the decompiler, you'll want to make sure the code the decompiler is generating is as accurate as possible. Usually, you can accomplish this by providing as much information as possible about data types and function prototypes.

Overriding Function Signatures

Functions that accept a variable number of arguments, such as `printf`, can be tricky for the decompiler since the decompiler would need to fully understand the semantics of the required arguments in order to estimate the number of supplied optional arguments.

Since `printf` takes a variable number of arguments, you can override the function signature at each calling location and (based on the format string) indicate that the `printf` statement should take one integer argument. To make this change, right-click a `printf` statement and choose **Override Signature** from the context menu to open the dialog shown in Figure 19-2. Of course, for functions that do not have a variable number of arguments, you should change the function signature rather than overriding it from the call.

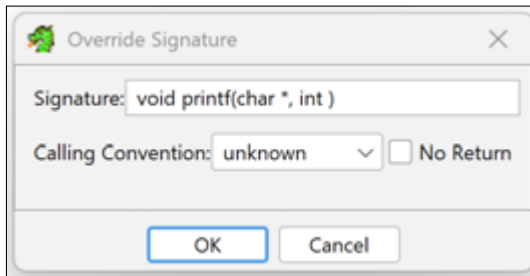


Figure 19-2: The Override Signature dialog

Correcting the second parameter type, `int`, in the signature (as shown in Figure 19-2) for each of the `printf` statements results in the following listing:

```
ulong do_math(uint param_1,uint param_2)
{
❶ uint uVar1;
  uint uVar2;
  uint uVar3;
  int iVar4;
  uint uVar5;
  time_t tVar6;

  tVar6 = time((time_t *)0x0);
  srand((uint)tVar6);
  uVar1 = rand();
  printf("c=%d\n",uVar1);
  uVar2 = uVar1 + param_1 + param_2;
  printf("d=%d\n",uVar2);
❷ uVar3 = uVar1 + param_1;
  printf("e=%d\n",uVar3);
  printf("f=%d\n",uVar2 * 100);
  iVar4 = rand();
❸ uVar5 = iVar4 - uVar3;
  printf("g=%d\n",uVar5);
```

```

❷ printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d\n", (ulong)param_1,
        (ulong)param_2, (ulong)uVar1, (ulong)uVar2, (ulong)(uVar1 + param_1),
        (ulong)(uVar2 * 100), (ulong)uVar4);
    return (ulong)uVar4;
}

```

In addition to the updated calls to `printf` with the correct arguments, two new lines have been added to the Decompiler listing as a result of overriding the `printf` function ❷ ❸. These statements weren't included previously because Ghidra believed the results were not used. Once the Decompiler understands that the results are used in each `printf`, the statements become meaningful and are displayed in the Decompiler window.

Editing Variable Types and Names

After correcting the function calls, you can continue cleaning up the listing by renaming (hotkey L) and retyping (hotkey CTRL-L) the parameters and the variables ❶ based on the names found in the `printf` format strings. As an aside, format strings are an extremely valuable source of information regarding the type and purpose of variables in any program.

Even after making these changes, the final `printf` statement ❹ is still a bit cumbersome:

```

printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d\n", (ulong)a,
        (ulong)(uint)b, (ulong)(uint)c, (ulong)(uint)d, (ulong)(uint)e,
        (ulong)(uint)(d * 100), (ulong)(uint)g);

```

Right-clicking this statement allows you to override the function signature. The first argument in this `printf` statement is the format string, and it doesn't need to be modified. Changing the rest of the arguments to type `int` results in the following cleaner code in the Decompiler window:

```

int do_math(int a, int b)

{
    int c;
    int d;
    int e;
    int g;
    time_t tVar1;

    tVar1 = time((time_t *)0x0);
    srand((uint)tVar1);
    c = rand();
    printf("c=%d\n", c);
    d = c + a + b;
    printf("d=%d\n", d);
    e = c + a;
    printf("e=%d\n", e);
    printf("f=%d\n", d * 100);
    g = rand();
}

```

```

g = g - e;
printf("g=%d\n",f);
printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d\n",a,b,c,d,e,d * 100,g);
return g;
}

```

This function is very similar to our original source code and much easier to read than the original Decompiler listing, as Ghidra has propagated the modifications we made to the function arguments throughout the listing. One difference between the Decompiler listing and our original source code is that the variable `f` has been replaced by an equivalent expression.

Highlighting Slices

Now that you have a more understandable Decompiler window, you can begin further analysis. Suppose that you want to know how individual variables affect and are affected by other variables. A *program slice* is a collection of statements that contribute to the value of a variable (in the case of a *backward slice*) or are affected by a variable (in the case of a *forward slice*). In vulnerability analysis scenarios, an investigation of program slices might begin with a question like the following: “I have control of this variable; where does its value get used?”

Ghidra’s right-click context menu provides five options for highlighting relationships between variables and instructions in a function. If you right-click a variable in the Decompiler window, you can choose from the following:

Highlight Def-use This option highlights all uses of the variable within the function. (You can use a middle mouse click to get the same effect.)

Highlight Forward Slice This option highlights everything that is impacted by the value in the selected variable. For example, if you select variable `b` in our updated decompilation and choose this option, all occurrences of `b` and `d` will be highlighted in the listing because a change in the value of `b` could also result in a change in the value of `d`.

Highlight Backward Slice This is the inverse of the previous option as it highlights all of the variables that contribute to a particular value. If you right-click variable `e` in the final `printf` statement of our updated decompilation and choose this option, all of the variables that affect the value of `e` (in this case `e`, `a`, and `c`) will be highlighted. Changing `a` or `c` could also change the value of `e`.

Highlight Forward Operator Slice This option highlights the entire statement associated with the Highlight Forward Slice option. In our updated decompilation, if you use this option while variable `b` is selected, all statements in which `b` or `d` appear will be highlighted.

Highlight Backward Operator Slice This option highlights the entire statement associated with the Highlight Backward Slice option. In our updated decompilation, selecting this option while highlighting variable `e` in the final `printf` statement will cause all statements in which `a`, `c`, or `e` appear to be highlighted.

Now that you have a general understanding of some approaches for working with the Decompiler window and using it for analysis, let's look at a more specific example.

Example 2: Non-Returning Functions

In general, Ghidra can safely assume that function calls return and therefore it can treat function calls as if they exhibit sequential flow within basic blocks. However, some functions, such as those marked with the `noreturn` keyword in source code or ended with an obfuscated jump instruction in malware, do not return, and Ghidra may generate inaccurate disassembled or decompiled code. Ghidra offers three approaches for dealing with non-returning functions: two non-returning function analyzers and the capability to edit function signatures manually.

Ghidra can identify non-returning functions based on a list of known `noreturn` functions such as `exit` and `abort` using the Non-Returning Functions – Known analyzer. Ghidra selects this analyzer by default as part of auto analysis, and its job is straightforward: If a function name appears in its list, it marks the function as non-returning and does its best to correct any associated issues (for example, setting associated calls to non-returning, finding flows that might need repairing).

The Non-Returning Functions – Discovered analyzer looks for clues that might indicate that a function doesn't return (for example, data or bad instructions right after the call). What it does with the information largely depends on the three options associated with the analyzer, shown in Figure 19-3.

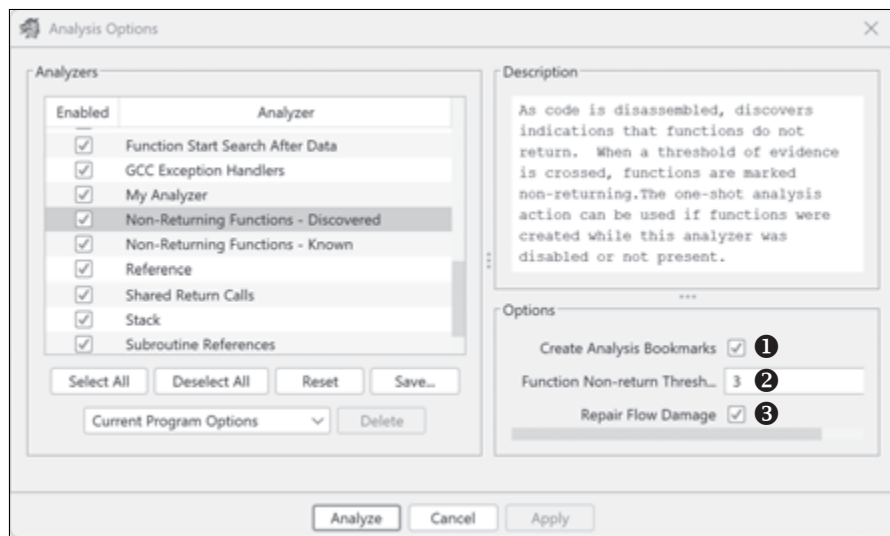


Figure 19-3: The Analysis options for Non-Returning Functions – Discovered

The first option ❶ allows the automatic creation of analysis bookmarks, which appear on the Listing window's bookmark bar. The second option ❷

allows you to specify a threshold that determines whether to designate a function as non-returning based on a series of checks for characteristics that are likely to indicate a non-returning function. Finally, a checkbox  determines whether to repair the associated flow damage.

When Ghidra is unable to identify a non-returning function, you can choose to edit the function signature yourself. If you complete the analysis and see error bookmarks flagging bad instructions, then that is a good indication that something is not quite right with Ghidra's own analysis. If the bad instruction follows a CALL, as in

00100839	CALL	noReturnA
0010083e	??	FFh

then you are likely to see an associated post-comment warning you about the situation in the Decompiler window, like this:

```
noReturnA(1);  
/* WARNING: Bad instruction - Truncating control flow here */  
halt_baddata();
```

Clicking the function name (noReturnA in this case) in the Decompiler window and then choosing Edit Function Signature will give you the option to modify attributes associated with the function, as shown in Figure 19-4.

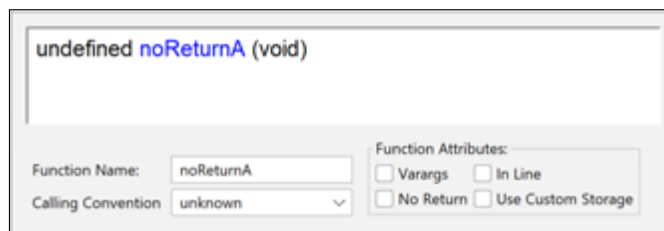


Figure 19-4: The dialog for editing function attributes

Check the No Return box to mark the function as non-returning. Ghidra will then insert a pre comment, shown next, in the Decompiler window, as well as a post comment in the Listing window:

```
/* WARNING: Subroutine does not return */  
noReturnA(1);
```

With this error corrected, you can move on to other issues.

Example 3: Automated Structure Creation

When analyzing decompiled C source code, you're likely to come across statements that appear to contain structure field references. Ghidra helps you create a structure and populate it based on the associated references that the decompiler has detected. Let's walk through an example starting with the source code and Ghidra's initial decompilation of the code.

Suppose you have source code that defines two struct types and then creates a global instance of each:

```
❶ struct s1 {
    int a;
    int b;
    int c;
};

❷ typedef struct s2 {
    int x;
    char y;
    float z;
} s2_type;

struct s1 GLOBAL_S1;
s2_type GLOBAL_S2;
```

One structure ❶ contains homogeneous elements, and the other ❷ contains a heterogeneous collection of types. The source code also contains three functions, one of which (`do_struct_demo`) declares a local instance of each structure type:

```
void display_s1(struct s1* s) {
    printf("The fields in s1 = %d, %d, and %d\n", s->a, s->b, s->c);
}

void update_s2(s2_type* s, int v) {
    s->x = v;
    s->y = (char)('A' + v);
    s->z = v * 2.0;
}

void do_struct_demo() {
    s2_type local_s2;
    struct s1 local_s1;

    printf("Enter six ints: ");
    scanf("%d %d %d %d %d %d", (int *)&local_s1, &local_s1.b, &local_s1.c,
        &GLOBAL_S1.a, &GLOBAL_S1.b, &GLOBAL_S1.c);

    printf("You entered: %d and %d\n", local_s1.a, GLOBAL_S1.a);
    display_s1(&local_s1);
    display_s1(&GLOBAL_S1);

    update_s2(&local_s2, local_s1.a);
}
```

The decompiled version of `do_struct_demo` appears in Listing 19-1.

```

void do_struct_demo(void)
{
    uint local_20;
    undefined1 local_1c [4];
    undefined1 local_18 [4];
    undefined1 local_14 [12];

    printf("Enter six ints: ");
    __isoc99_scanf("%d %d %d %d %d %d",&local_20,local_1c,local_18,
        &GLOBAL_S1,&DAT_0030101c,&DAT_00301020);
    printf("You entered: %d and %d\n",(ulong)local_20,(ulong)GLOBAL_S1);
    ❶ display_s1(&local_20);
    ❷ display_s1(&GLOBAL_S1);
    update_s2(local_14,local_20);
    return;
}

```

Listing 19-1: Initial decompilation of do_struct_demo

Navigating to the `display_s1` function from either function call ❶ ❷ by double-clicking it in the Decompiler window yields the following:

```

void display_s1(uint *param_1)
{
    printf("The fields in s1 = %d, %d, and %d\n", (ulong)*param_1,
        (ulong)param_1[1],(ulong)param_1[2]);
    return;
}

```

Because you suspect the argument to `display_s1` might be a structure pointer, you can ask Ghidra to automate the process of creating a struct for you by right-clicking `param_1` in the function's argument list and selecting Auto Create Structure from the context menu. In response, Ghidra tracks all uses of `param_1`, treats all arithmetic performed on the pointer as referencing a member of a struct, and automatically creates a new struct type containing fields at each referenced offset. This changes a few things in the Decompiler listing:

```

void display_s1(astruct *param_1)
{
    printf("The fields in s1 = %d, %d, and %d\n", (ulong)param_1->field_0x0,
        (ulong)param_1->field_0x4, (ulong)param_1->field_0x8);
    return;
}

```

The type of the parameter has changed and is now `astruct*`, and the call to `printf` now contains field references. The new type has also been added to the Data Type Manager, and hovering over the structure name displays the field definitions, as shown in Figure 19-5.

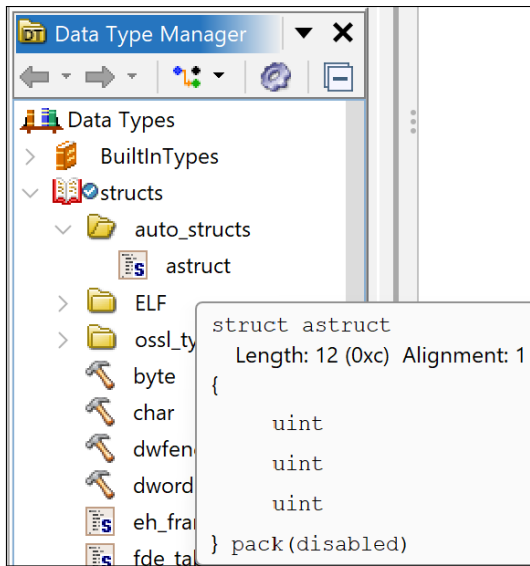


Figure 19-5: The Data Type Manager window showing the new struct

You can update the type for GLOBAL_S1 to astruct by using the Retype Variable option from the right-click context menu. The following listing shows the results:

```

void do_struct_demo(void)

{
❶ astruct local_20;
   undefined1 local_14 [12];

   printf("Enter six ints: ");
❷ __isoc99_scanf("%d %d %d %d %d %d",&local_20,&local_20.field1_0x4,
❸   &local_20.field2_0x8,&GLOBAL_S1,0x30101c,0x301020);

   printf("You entered: %d and %d\n",(ulong)local_20.field0_0x0,
❹   (ulong)GLOBAL_S1.field0_0x0);

   display_s1(&local_20);
   display_s1(&GLOBAL_S1);
   update_s2(local_14,local_20.field0_0x0);
   return;
}
  
```

Comparing this listing with the initial decompilation shows the modification of the type for local_20 ❶ and the addition of field references for both local_20 ❷ ❸ and GLOBAL_S1 ❹.

Let's shift focus to the decompilation of the third function, `update_s2` (Listing 19-2).

```
void update_s2(int *param_1,int param_2)

{
    *param_1 = param_2;
    *(char *)(param_1 + 1) = (char)param_2 + 'A';
    param_1[2] = (int)((float)param_2 + (float)param_2);
    return;
}
```

Listing 19-2: Initial decompilation of `update_s2`

You can use the previous approach to automatically create a structure for `param_1`. Simply right-click `param_1` in the function and choose **Auto Create Structure** from the context menu:

```
void update_s2(astruct_1 *param_1,int param_2)

{
    param_1->field0_0x0 = param_2;
    param_1->field1_0x4 = (char)param_2 + 'A';
    param_1->field5_0x8 = (float)param_2 + (float)param_2;
    return;
}
```

The Data Type Manager now has a second struct definition associated with this file, as shown in Figure 19-6.

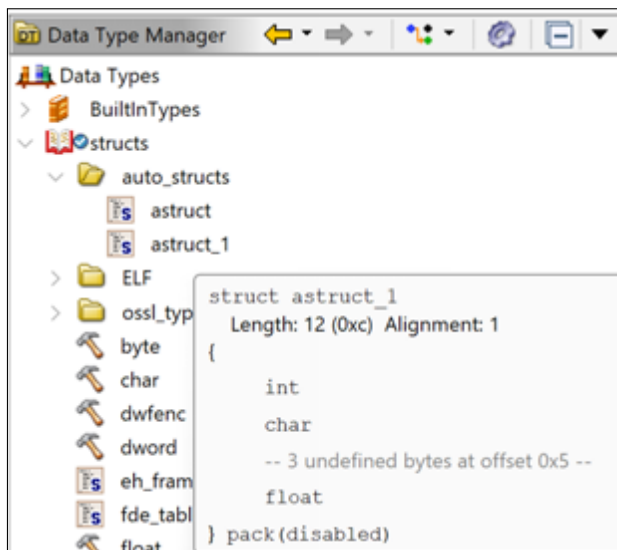


Figure 19-6: The Data Type Manager window showing the additional new struct

This structure has an int, a char, three undefined bytes (likely padding inserted by the compiler), and a float. To edit the structure, right-click `astruct_1` and choose Edit from the context menu, which opens the Structure Editor window. If we choose to name the int field `x`, the char field `y`, and the float field `z`, and then save the changes, the new field names will be reflected in the Decompiler listing:

```
void update_s2(astruct_1 *param_1,int param_2)
{
    param_1->x = param_2;
    param_1->y = (char)param_2 + 'A';
    param_1->z = (float)param_2 + (float)param_2;
    return;
}
```

This listing is much easier to read and understand than the original decompilation in Listing 19-2.

Summary

The Decompiler window, like the Listing window, provides you with a view into a binary, and each has its associated strengths and weaknesses. The decompiler provides a higher-level view that can help you understand the general structure and functionality of a single function more quickly than looking at the disassembly (particularly for those who do not have years of experience reading disassembly listings). The Listing window provides a lower-level view of the entire binary, with all of the available detail, but this can make it more difficult to gain insight into the big picture.

Ghidra's decompiler can be used effectively with the Listing window and all of the other tools we have introduced throughout the book to aid you in your reverse engineering process. In the end, it is the reverse engineer's role to determine the best approach to solving the problem at hand.

This chapter focused on the Decompiler window and issues associated with decompilation. Many of these challenges can be traced to the wide variety of compilers and associated compiler options that directly influence the resulting binary. In the next chapter, we take a look at some compiler-specific behaviors and compiler build options to better understand the resulting binaries.

20

COMPILER VARIATIONS



If we have done our job properly, you now possess the essential skills needed to use Ghidra effectively and, more importantly, to bend it to your will. The next step is to learn how to adapt to the challenges that binaries (as opposed to Ghidra itself) will throw at you.

Your ability to make sense of a disassembly depends on your context. If you spend all of your time examining code that was compiled using gcc on a Linux platform, you'll become quite familiar with the style of code it generates, but you may be baffled by a debug version of a program compiled using the Microsoft C/C++ compiler. If you are a malware analyst, you may regularly see code created using gcc, clang, Microsoft's C++ compiler, and others, all in the same afternoon.

Like you, Ghidra is more familiar with the output of some compilers than other compilers, and a familiarity with code generated by one compiler in no way guarantees that you will recognize high-level constructs compiled using an entirely different one (or even different versions of the same compiler family). Rather than relying entirely on Ghidra's analysis capabilities

to recognize commonly used code and data constructs, you should always be prepared to use your own skills to properly interpret a disassembly: your familiarity with a given assembly language, your knowledge of compilers, and your research abilities.

In this chapter, we cover some of the ways that compiler differences manifest themselves in disassembly listings. In some cases, the differences between compilers are merely cosmetic, but in other cases, they are much more significant. In this section, we look at high-level language constructs and demonstrate how different compilers and compiler options may significantly impact the resulting disassembly listing. We begin with switch statements and the two mechanisms most commonly employed to resolve switch case selection. Following that, we look at the way that compiler options affect code generation for common expressions before moving on to discuss how different compilers implement some security-related behaviors, apply C++-specific constructs, and handle program startup.

Our examples consist primarily of compiled C code, as the variability of C compilers and target platforms demonstrates foundational concepts that can be extended to other compiled languages.

switch Statements

The C switch statement is a frequent target for compiler optimizations. The goal of these optimizations is to match the switch variable to a valid case label in the most efficient manner possible, and the distribution of the switch statement's case labels constrains the type of search the compiler can use.

We measure the efficiency of a search by the number of comparisons required to find the correct case. Constant time algorithms, such as a table lookup, are the most efficient methods. At the other end of the continuum is linear search, which, in the worst case, requires comparing the switch variable against every case label before finding a match or resolving to the default, and thus is the least efficient. The efficiency of a binary search is much better, on average, than linear search but introduces additional constraints as it requires a sorted list.

NOTE

For you algorithm analysis fans, the use of a table lookup allows the target case to be found in a constant number of operations regardless of the size of the search space, which, as you may recall from your algorithms class, is also called constant time, or $O(1)$. Linear time algorithms are $O(n)$ and, fortunately, not used in switch statements. Binary search is $O(\log n)$. This will not be on your final exam.

To identify the most efficient implementation for a particular switch statement, it helps to understand how the case label distribution affects the compiler's decision-making process.

Jump Tables

When case labels are closely clustered as they are in the following listing, compilers generally resolve the switch variable by performing a table lookup to match the switch variable to the address of its associated case, specifically by using a jump table (Listing 20-1).

```
switch (a) {  
    /** NOTE: case bodies omitted for brevity **/  
    case 1: /*...*/ break;  
    case 2: /*...*/ break;  
    case 3: /*...*/ break;  
    case 4: /*...*/ break;  
    case 5: /*...*/ break;  
    case 6: /*...*/ break;  
    case 7: /*...*/ break;  
    case 8: /*...*/ break;  
    case 9: /*...*/ break;  
    case 10: /*...*/ break;  
    case 11: /*...*/ break;  
    case 12: /*...*/ break;  
}
```

Listing 20-1: A statement with consecutive case labels

A *jump table* is an array of pointers (or relative offsets), each of which references a possible jump target. At runtime, a dynamic index into the table chooses one of the many potential jumps each time the jump table is referenced.

Jump tables work well when switch case labels are closely spaced, with most of the cases falling into a consecutive number sequence. Compilers take this into account when deciding whether to use a jump table. For any switch statement, we can compute the minimum number of entries an associated jump table will contain as follows:

$$\text{num\_entries} = \text{max\_case\_value} - \text{min\_case\_value} + 1$$

We can then compute the *density*, or utilization rate, of the jump tables:

$$\text{density} = \text{num\_cases} / \text{num\_entries}$$

A completely contiguous list with every value used as a case label would have a density value of 100 percent (1.0). Finally, here is how we calculate the total amount of space required to store the jump table:

$$\text{table\_size} = \text{num\_entries} * \text{sizeof}(\text{void}^*)$$

While it makes sense to implement a switch statement with 100 percent density using a jump table, compilers might not use this approach to implement a set of cases with a density of 30 percent, since they would still need to allocate jump table entries for the absent cases, which would take up 70 percent of the jump table. If `num_entries` were 30, the jump table would contain entries for 21 unreferenced case labels. On a 64-bit system, these would comprise 168 of the 240 bytes allocated to the table, which is not a lot of overhead, but if `num_entries` jumped to 300, then the overhead would become 1,680 bytes, which may not be worth the trade-off for 90 possible cases.

In these cases, a compiler that is optimizing for speed may favor jump table implementations, while a compiler that is optimizing for size may choose an alternative implementation with lower memory overhead: binary search.

Binary Search

Binary search is efficient in situations when the case labels are widely spread (or low density), as in Listing 20-2.

```
switch (a) {
/** NOTE: case bodies omitted for brevity */
    case 1:      /*...*/ break;
    case 211:    /*...*/ break;
    case 295:    /*...*/ break;
    case 462:    /*...*/ break;
    case 528:    /*...*/ break;
    case 719:    /*...*/ break;
    case 995:    /*...*/ break;
    case 1024:   /*...*/ break;
    case 8000:   /*...*/ break;
    case 13531:  /*...*/ break;
    case 13532:  /*...*/ break;
    case 15027:  /*...*/ break;
}
```

Listing 20-2: A statement with no consecutive case labels

Here, the density is only 0.0008. For those analyzing algorithms at home, this means that the switch variable is matched after at most $\log_2 N$ comparisons, where N is the number of cases contained in the switch statement. This is $O(\log n)$.

Because binary search works only on sorted lists, the compiler must ensure that the case labels are ordered before it begins the search with the median value. This may result in the reordering of case blocks when viewed in a disassembly, as compared to the order they appear in the corresponding source.

NOTE

While the complexity of sorting is very high compared to the complexity of searching, it is important to note the sorting would occur only once, at compilation time, whereas the search would take place every time the switch statement is encountered during execution.

Listing 20-3 shows an outline for a non-iterative binary search through a fixed number of constant values. This is the rough framework that the compiler uses to implement the switch from Listing 20-2.

```
if (value < median) {
    // Value is in 0-50 percentile.
    if (value < lower_half_median) {
        // Value is in 0-25 percentile.
        // Continue successive halving until value is resolved.
    } else {
        // Value is in 25-50 percentile.
        // Continue successive halving until value is resolved.
    }
} else {
    // Value is in 50-100 percentile.
    if (value < upper_half_median) {
        // Value is in 50-75 percentile.
        // Continue successive halving until value is resolved.
    } else {
        // Value is in 75-100 percentile.
        // Continue successive halving until value is resolved.
    }
}
}
```

Listing 20-3: Non-iterative binary search through a fixed number of constant values

Compilers are also capable of performing more fine-grained optimizations across a range of case labels. For example, when confronted with the following case labels

```
label_set = [1, 2, 3, 4, 5, 6, 7, 8, 50, 80, 200, 500, 1000, 5000, 10000]
```

a less aggressive compiler might see a density of 0.0015 here and generate a binary search through all 15 cases. A more aggressive compiler might emit a jump table to resolve cases 1 to 8, and then use a binary search for the remaining cases, achieving optimal performance for over half of the cases.

Before we look at the disassembled versions of Listings 20-1 and 20-2, let's look at the Ghidra Function Graph windows corresponding to the listings, shown side by side in Figure 20-1.

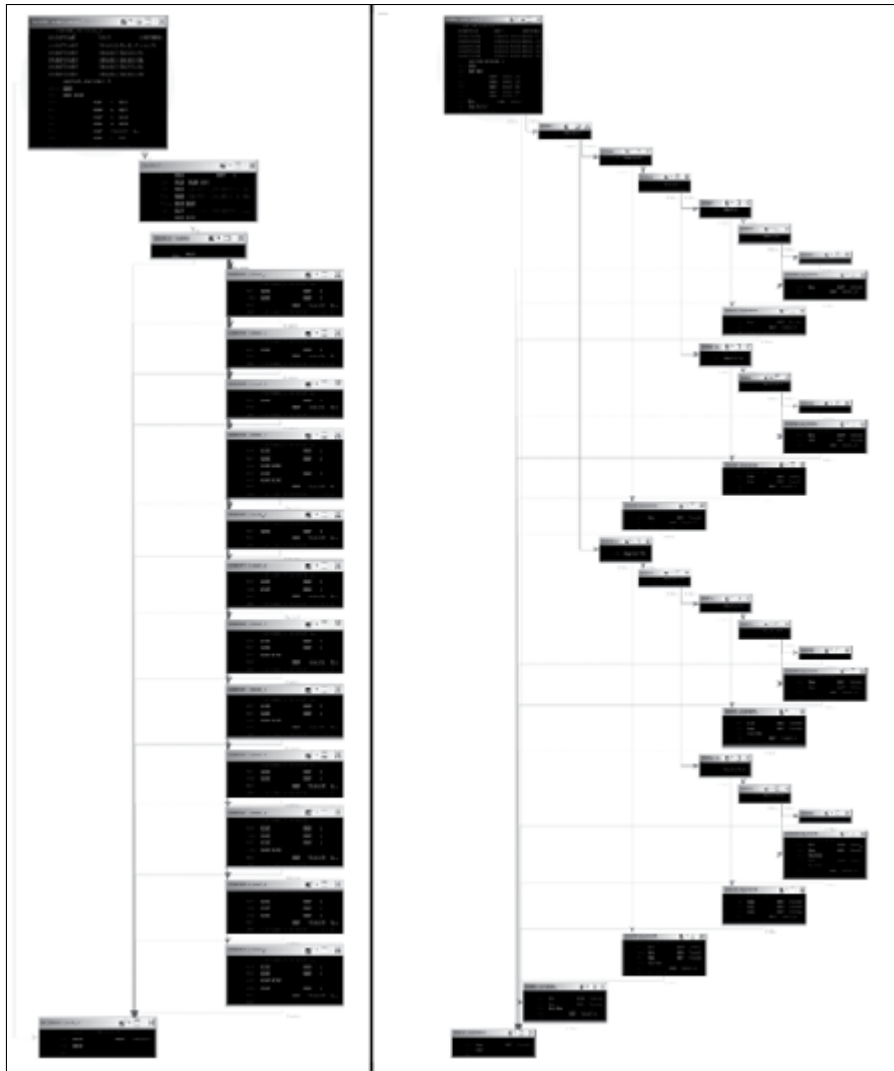


Figure 20-1: The Ghidra Function Graph switch statement examples

On the left, the graph for Listing 20-1 shows a nice horizontal spread of cases with minimal expansion. Each code block resides at the same nesting depth (three), as is true for cases in a switch statement. The spread suggests that we can use an index to quickly select one block from the many blocks at the same level. This is precisely how jump table resolution works, and the left-hand graph provides us with a visual hint that this is the case even before we have looked at a single line of the disassembly.

The right-hand graph is Ghidra's result based solely on its understanding of the disassembly of Listing 20-2. The lack of a jump table makes it much more challenging to identify as a switch statement. What you are seeing is a visual representation of the switch statement using Ghidra's Nested Code Layout. This is the default layout for function graphs in Ghidra and is intended

to represent the flow structures in a program. The horizontal branching (splitting, in the binary sense) in this graph suggests conditional execution (if-else) branching to mutually exclusive alternatives. The vertical symmetry (the depth of the binary search) suggests that the alternative execution paths have been very carefully balanced to place equal numbers of blocks in each horizontal (left/right) half of the graph. The depth of the search is dictated by the total number of case labels present in the switch. For a binary search, this depth will always be on the order of $\log_2(\text{num\_cases})$.

Turning our attention to the Decompiler window, Figure 20-2 shows the partial decompilation of the functions displayed in Figure 20-1. On the left is the decompiled version of Listing 20-1. As with the graph, the presence of a jump table in the binary helps Ghidra identify the code as a switch statement.

Decompile: switch1 - (switch_demo_1_x86_gcc)	Decompile: switch4 - (switch_demo_1_x86_gcc)
<pre> 1 int switch1(int a,int b,int c,int d) 2 3 4 { 5 int result; 6 7 result = 0; 8 switch(a) { 9 case 1: 10 result = b + a; 11 break; 12 case 2: 13 result = c + a; 14 break; 15 case 3: 16 result = d + a; 17 break; 18 case 4: 19 result = c + b; 20 break; 21 case 5: 22 result = d + b; 23 break; 24 case 6: </pre>	<pre> 1 int switch4(int a,int b,int c,int d) 2 3 4 { 5 int result; 6 7 result = 0; 8 if (a == 719) { 9 result = d + c; 10 } 11 else { 12 if (a < 720) { 13 if (a == 295) { 14 result = d + 0x127; 15 } 16 else { 17 if (a < 296) { 18 if (a == 1) { 19 result = b + 1; 20 } 21 else { 22 if (a == 211) { 23 result = c + 0xd3; 24 } </pre>

Figure 20-2: Ghidra-decompiled switch statement examples

On the right is the decompiled version of Listing 20-2. The decompiler has presented the switch statement as a nested if-else structure consistent with a binary search and similar in structure to Listing 20-3. You can see that the first comparison is against 719, the median value in the list, and that subsequent comparisons continue to divide the search space in half. Referring to Figure 20-1 (as well as Listing 20-3), we can again observe that the graphical representations of each function closely correspond to the indentation patterns observed in the Decompiler window.

Now that you have an idea of what is happening from a high level, let's look inside the binaries and investigate what is happening at a low level. Since our objective in this chapter is to observe differences between compilers, we

present this example as a series of comparisons between two compilers, gcc and Microsoft C/C++.

NOTE

<gcc> accepts a large number of command line arguments, and each may affect the resulting generated code. We used the following gcc command to compile our starting example: <gcc switch_demo_1.c -m32 -fno-pie -fno-pic -fno-stack-protector -o switch_demo_1_x86>. The Microsoft C/C++ example is an unmodified x86 debug build. We will introduce additional options in subsequent examples.

Example: Comparing gcc to the Microsoft C/C++ Compiler

In this example, we compare two 32-bit x86 binaries generated for Listing 20-1 by two distinct compilers. We will attempt to identify components of a switch statement in each binary, locate the associated jump table in each binary, and point out significant differences between the two binaries. Let's start by looking at the switch-related components for Listing 20-1 in the binary built with gcc:

```
0001075a  CMP    dword ptr [EBP + value],12 ❶
0001075e  JA     switchD_00010771::caseD_0 ❷
00010764  MOV    EAX,dword ptr [EBP + a]
00010767  SHL    EAX,0x2
0001076a  ADD    EAX,switchD_00010771::switchdataD_00010ee0      = 00010805
0001076f  MOV    EAX,dword ptr [EAX]=>->switchD_00010771::caseD_0 = 00010805
                                switchD_00010771::switchD
00010771  JMP    EAX
                                switchD_00010771::caseD_1      XREF[2]: 00010771(j), 00010ee4(*) ❸
00010773  MOV    EDX,dword ptr [EBP + a]
00010776  MOV    EAX,dword ptr [EBP + b]
00010779  ADD    EAX,EDX
0001077b  MOV    dword ptr [EBP + result],EAX
0001077e  JMP    switchD_00010771::caseD_0
...
                                switchD_00010771::switchdataD_00010ee0  XREF[2]: switch_version_1:0001076a(*), ❹
                                                                switch_version_1:0001076f(R)
00010ee0  addr   switchD_00010771::caseD_0 ❺
00010ee4  addr   switchD_00010771::caseD_1
00010ee8  addr   switchD_00010771::caseD_2
00010eec  addr   switchD_00010771::caseD_3
00010ef0  addr   switchD_00010771::caseD_4
00010ef4  addr   switchD_00010771::caseD_5
00010ef8  addr   switchD_00010771::caseD_6
00010efc  addr   switchD_00010771::caseD_7
00010f00  addr   switchD_00010771::caseD_8
00010f04  addr   switchD_00010771::caseD_9
00010f08  addr   switchD_00010771::caseD_a
00010f0c  addr   switchD_00010771::caseD_b
00010f10  addr   switchD_00010771::caseD_c
```

Ghidra recognizes the switch bounds test ❶, the jump table ❷, and individual case blocks by value, such as the one at `switchD_00010771::caseD_1` ❸. The compiler generated a jump table with 13 entries, although Listing 20-1 contained only 12 cases. The additional case, case 0 (the first entry ❹ in the jump table), shares a target address with every value outside the range 1 to 12. In other words, case 0 is part of the default case.

While it may seem that negative numbers are being excluded from the default, the `CMP, JA` sequence works as a comparison on unsigned values; thus, -1 (0xFFFFFFFF) would be seen as 4294967295, which is much larger than 12 and therefore excluded from the valid range for indexing the jump table. The `JA` instruction effectively directs all such cases to the default location: `switchD_00010771::caseD_0` ❺.

Now that we understand the basic components of the code generated by the `gcc` compiler, let's shift our focus to the same components in code generated by the Microsoft C/C++ compiler in debug mode:

```

00411e88 MOV    ECX,dword ptr [EBP + local_d4]
00411e8e SUB    ECX,0x1 ❶
00411e91 MOV    dword ptr [EBP + local_d4],ECX
00411e97 CMP    dword ptr [EBP + local_d4],11 ❷
00411e9e JA     switchD_00411eaa::caseD_c
00411ea4 MOV    EDX,dword ptr [EBP + local_d4]
               switchD_00411eaa::switchD
00411eaa JMP    dword ptr [EDX*0x4 + ->switchD_00411eaa::caseD      = 00411eb1
               switchD_00411eaa::caseD_1      XREF[2]: 00411eaa(j), 00411f4c(*)
00411eb1 MOV    EAX,dword ptr [EBP + param_1]
00411eb4 ADD    EAX,dword ptr [EBP + param_2]
00411eb7 MOV    dword ptr [EBP + local_c],EAX
00411eba JMP    switchD_00411eaa::caseD_c
...
               switchD_00411eaa::switchdataD_00411f4c  XREF[1]: switch_version_1:00411eaa(R)
00411f4c addr   switchD_00411eaa::caseD_1 ❸
00411f50 addr   switchD_00411eaa::caseD_2
00411f54 addr   switchD_00411eaa::caseD_3
00411f58 addr   switchD_00411eaa::caseD_4
00411f5c addr   switchD_00411eaa::caseD_5
00411f60 addr   switchD_00411eaa::caseD_6
00411f64 addr   switchD_00411eaa::caseD_7
00411f68 addr   switchD_00411eaa::caseD_8
00411f6c addr   switchD_00411eaa::caseD_9
00411f70 addr   switchD_00411eaa::caseD_a
00411f74 addr   switchD_00411eaa::caseD_b
00411f78 addr   switchD_00411eaa::caseD_c

```

Here, the switch variable (`local_d4` in this case) is decremented ❶ to shift the range of valid values from 0 to 11 ❷, eliminating the need for a dummy table entry for the value 0. As a result, the first entry (or 0 index entry) in the jump table ❸ actually refers to the code for switch case 1.

Another, perhaps more subtle, difference between the two listings is the location of the jump table within the file. The gcc compiler places switch jump tables in the read-only data (.rodata) section of the binary, providing a logical separation between the code associated with the switch statement and the data required to implement the jump table. The Microsoft C/C++ compiler, on the other hand, inserts jump tables into the .text section, immediately following the function containing the associated switch statement. Positioning the jump table in this manner has little effect on the behavior of the program. In this example, Ghidra is able to recognize the switch statements for both compilers and uses the term switch within the associated labels.

One of the key points here is that there is no single correct way to compile source to assembly. As a result, you cannot assume that something is not a switch statement simply because Ghidra fails to label it as such. Understanding the switch statement characteristics that factor into the compiler implementation can help you make a more accurate inference about the original source code.

Compiler Build Options

Compilers convert high-level code that solves a particular problem into low-level code that solves the same problem. Multiple compilers may solve the same problem in rather different ways. Further, a single compiler may solve a problem very differently based on the associated compiler options. In this section, we look at the assembly language code that results from using different compilers and different command line options. Some differences will have a clear explanation; others will not.

Microsoft's Visual Studio can build either debug or release versions of program binaries. Other compilers, such as gcc, also offer the ability to insert debugging symbols during the compilation process. To see how the two versions are different, you should compare the build options specified for each. Release versions are generally optimized, while debug versions are not. Debug versions are linked with additional symbol information and may utilize debugging versions of required runtime libraries, while release versions do not. Debugging-related symbols allow debuggers to map assembly language statements back to their source code counterparts during debugging sessions and to determine the names and types of local variables (such information is otherwise lost during the compilation process). The debugging versions of Microsoft's runtime libraries have also been compiled with debugging symbols included, optimizations disabled, and additional safety checks enabled to verify that some function parameters are valid.

NOTE

Optimization generally involves the elimination of redundancy in code or the selection of faster but potentially larger sequences of code in order to satisfy a developer's desire to create either faster or smaller executable files. Optimized code may not be as straightforward to analyze or debug as unoptimized code and may therefore be considered a bad choice for use during a program's development and debugging phases.

When disassembled using Ghidra, debug builds of Visual Studio projects look significantly different from release builds. This is a result of compiler and linker options specified only in debug builds, such as basic runtime checks (/RTCx), which introduce extra code into the resulting binary. Let's jump right in and look at some of these differences in disassemblies.

Example 1: Modulo Operator

We begin with an example of a simple mathematical operation, modulo. The following listing contains the source code for a program whose only goal is to accept an integer value from the user and demonstrate integer division and the modulo operator:

```
int main(int argc, char **argv) {
    int x;
    printf("Enter an integer: ");
    scanf("%d", &x);
    printf("%d %% 10 = %d\n", x, x % 10);
}
```

Let's investigate how the disassembly varies across compilers for the modulo operator in this example.

Microsoft C/C++ Win x64 Debug

The following listing shows the code that Visual Studio generates when configured to build a debug version of the binary:

1400119c6	MOV	EAX,dword ptr [RBP + local_f4]
1400119c9	CDQ	
1400119ca	MOV	ECX,0xa
❶ 1400119cf	IDIV	ECX
1400119d1	MOV	EAX,EDX
❷ 1400119d3	MOV	R8D,EAX
1400119d6	MOV	EDX,dword ptr [RBP + local_f4]
1400119d9	LEA	RCX,[s_d_%%_10_=_d_]
1400119e0	CALL	printf

A straightforward x86 IDIV instruction ❶ leaves the quotient in EAX and the remainder of the division in EDX. The result is then moved to the lower 32 bits of R8 (R8D) ❷, which is the third argument in the call to printf.

Microsoft C/C++ Win x64 Release

Release builds optimize software for speed and size in order to enhance performance and minimize storage requirements. When optimizing for speed, compiler writers may resort to non-obvious implementations of common

operations. The following listing shows us how Visual Studio generates the same modulo operation in a release binary:

```

140001136 MOV     ECX,dword ptr [RSP + local_18]
14000113a MOV     EAX,0x66666667
❶ 14000113f IMUL   ECX
140001141 MOV     R8D,ECX
140001144 SAR     EDX,0x2
140001147 MOV     EAX,EDX
140001149 SHR     EAX,0x1f
14000114c ADD     EDX,EAX
14000114e LEA     EAX,[RDX + RDX*0x4]
140001151 MOV     EDX,ECX
140001153 ADD     EAX,EAX
140001155 LEA     RCX,[s_d_10_=_d_ ]
❷ 14000115c SUB     R8D,EAX
❸ 14000115f CALL    printf

```

In this case, the compiler uses multiplication ❶ rather than division, and after a long sequence of arithmetic operations, what must be the result of the modulo operation ends up in R8D ❷ (again, the third argument in the call to printf ❸). Intuitive, right? An explanation of this code follows our next example.

gcc for Linux x64

We've seen how differently one compiler can behave simply by changing the compile-time options used to generate a binary. We might expect that a completely unrelated compiler would generate entirely different code yet again. As it turns out, the disassembly showing the gcc version of the same modulus operation looks somewhat familiar:

```

00100708 MOV     ECX,dword ptr [RBP + local_c ]
0010070b MOV     EDX,0x66666667
00100710 MOV     EAX,ECX
❶ 00100712 IMUL   EDX
00100714 SAR     EDX,0x2
00100717 MOV     EAX,ECX
00100719 SAR     EAX,0x1f
0010071c SUB     EDX,EAX
0010071e MOV     EAX,EDX
00100720 SHL     EAX,0x2
00100723 ADD     EAX,EDX
00100725 ADD     EAX,EAX
00100727 SUB     ECX,EAX
❷ 00100729 MOV     EDX,ECX
0010072b MOV     EAX ,dword ptr [RBP + local_c ]
0010072e MOV     ESI ,EAX
00100730 LEA     RDI ,[s_d_10_=_d_001008a9 ]

```

```
00100737  MOV     EAX ,0x0
0010073c  CALL    <EXTERNAL>::printf
```

The code is very similar to the assembly produced by the Visual Studio release version. We again see multiplication ❶ rather than division, followed by a sequence of arithmetic operations that eventually leaves the result in EDX ❷, where it is eventually used as the third argument to printf.

The code is using a multiplicative inverse to perform division by multiplying because hardware multiplication is faster than hardware division. You may also see multiplication implemented using a series of additions and arithmetic shifts, as each of these operations is significantly faster in hardware than multiplication.

Your ability to recognize this code as modulo 10 depends on your experience, patience, and creativity. If you've seen similar code sequences in the past, you are probably more apt to recognize what's taking place here. Lacking that experience, you might instead work through the code manually with sample values, hoping to recognize a pattern in the results. You might even take the time to extract the assembly language, wrap it in a C test harness, and do some high-speed data generation to assist you. Ghidra's decompiler can be another useful resource for reducing complex or unusual code sequences to their more recognizable C equivalents. Ghidra's decompiler correcting identified this as `local_c % 10`.

As a last resort, or first resort (don't be ashamed), you might turn to the internet for answers. But what should you be searching for? Usually, unique, specific searches yield the most relevant results, and the most unique feature in the sequence of code is the integer constant 0x66666667. Cryptographic algorithms also often use unique constants, and a quick internet search of these values may be all it takes to identify exactly what crypto routine you are staring at, as is the case here.

Example 2: The Ternary Operator

The ternary operator in C/C++ evaluates an expression and then yields one of two possible results depending on the boolean value of that expression. Conceptually, the ternary operator can be thought of as an if-else statement (and can even be replaced with one). The following intentionally unoptimized source code demonstrates the use of this operator:

```
int main() {
    volatile int x = 3;
    volatile int y = x * 13;
    ❶ volatile int z = y == 30 ? 0 : -1;
}
```

The `volatile` keyword asks the compiler not to optimize code involving the associated variables. Without its use here, some compilers will optimize away the entire body of this function since none of the statements contribute to the function's result. This is one of the challenges you might face when coding examples for yourself or for others.

As for the behavior of the unoptimized code, the assignment into the variable `z` ❶ could be replaced with the following if-else statement without changing the semantics of the program:

```

if (y == 30) {
    z = 0;
} else {
    z = -1;
}

```

Let's see how different compilers and compiler options handle the ternary operator code differently.

gcc on Linux x64 (Unoptimized)

With no options, gcc generated the following assembly for the initialization of `z`:

```

00100616  MOV     EAX,dword ptr [RBP + y]
❶ 00100619  CMP     EAX,0x1e
0010061c  JNZ     LAB_00100625
0010061e  MOV     EAX,0x0
00100623  JMP     LAB_0010062a
LAB_00100625
00100625  MOV     EAX,0xffffffff
LAB_0010062a
❷ 0010062a  MOV     dword ptr [RBP + z],EAX

```

This code uses the if-else implementation. It compares local variable `y` to 30 ❶ to decide whether to set `EAX` to 0 or 0xffffffff in opposing branches of the if-else before assigning the result into `z` ❷.

Microsoft C/C++ Win x64 Release

Visual Studio yields a very different implementation of the statement containing the ternary operator. Here, the compiler recognizes that it can use a single instruction to conditionally generate either 0 or -1 (and no other possible value), and it uses this instruction in lieu of the if-else construct we saw earlier:

```

140001013  MOV     EAX,dword ptr [RSP + local_res8]
❶ 140001017  SUB     EAX,0x1e
❷ 14000101a  NEG     EAX
❸ 14000101c  SBB     EAX,EAX
14000101e  MOV     dword ptr [RSP + local_res8],EAX

```

The `SBB` (*subtract with borrow*) instruction ❸ subtracts the second operand from the first operand and then subtracts the carry flag, `CF`, which can be only 0 or 1. The equivalent arithmetic expression to `SBB EAX,EAX` is `EAX - EAX - CF`, which reduces to `0 - CF`. This, in turn, can result only in 0 (when `CF == 0`) or -1 (when `CF == 1`).

For this trick to work, the compiler must set the carry flag properly prior to executing the SBB instruction. It accomplishes this by comparing EAX to the constant 0x1e (30) ❶ using a subtraction that leaves EAX equal to 0 only when EAX was initially 0x1e. The NEG instruction ❷ then sets CF for the SBB instruction that follows. The NEG instruction clears CF when its operand is zero and sets the carry flag in all other cases.

gcc on Linux x64 (Optimized)

When we ask gcc to try a little harder by optimizing its code (-O2), the result is not unlike the Visual Studio code in the previous example:

```
00100506  MOV    EAX,dword ptr [RSP + y]
0010050a  CMP    EAX,0x1e
❶ 0010050d  SETNZ  AL
00100510  MOVZX  EAX,AL
❷ 00100513  NEG    EAX
❸ 00100515  MOV    dword ptr [RSP + z],EAX
```

In this case, gcc uses SETNZ ❶ to conditionally set the AL register to either 0 or 1 based on the state of the zero flag resulting from the preceding comparison. It then negates the result ❷ to become either 0 or -1 before assigning the value into variable z ❸.

You might ask yourself how either of the latter two examples are improvements over the first (if-else) example. The answer lies in the fact that both eliminate two jumps (one conditional and one unconditional) from the generated code. Because jumps represent nonlinear flow through a program's address space, they can negatively impact processor performance when the processor is attempting to predict what instruction may execute next to facilitate early fetching of the next instruction before the current instruction is finished executing.

Example 3: Function Inlining

When a programmer marks a function inline, they are suggesting to the compiler that any calls to the function should be replaced with a copy of the entire function body. The intent of function inlining is to speed up the function call by eliminating parameter and stack frame setup and teardown, but many copies of an inlined function make the binary larger. Inlined functions can also be very difficult to recognize in binaries because the distinctive call instruction is eliminated.

Even when the inline keyword has not been used, compilers may elect to inline a function on their own initiative. In our third example, we are making a call to the following function:

```
int maybe_inline() {
    return 0x12abcdef;
}
```

```
int main() {
    int v = maybe_inline();
    printf("after maybe_inline: v = %08x\n", v);return 0;
}
```

Let's see how this call looks in the disassembly when inlined and not inlined.

gcc on Linux x86 (Unoptimized)

After building a Linux x86 binary using gcc with no optimizations, we disassemble it to see the following listing:

00010775	PUSH	EBP
00010776	MOV	EBP,ESP
00010778	PUSH	ECX
00010779	SUB	ESP,0x14
❶ 0001077c	CALL	maybe_inline
00010781	MOV	dword ptr [EBP + local_14],EAX
00010784	SUB	ESP,0x8
00010787	PUSH	dword ptr [EBP + local_14]
0001078a	PUSH	s_after_maybe_inline:_v=_%08x_000108e2
0001078f	CALL	printf

We can clearly see the call to the `maybe_inline` function ❶ in this disassembly, even though it is just a single line of code returning a constant value.

gcc on Linux x86 (Optimized)

Next, we look at an optimized (-O2) version of the same source code:

0001058a	PUSH	EBP
0001058b	MOV	EBP,ESP
0001058d	PUSH	ECX
0001058e	SUB	ESP,0x8
❶ 00010591	PUSH	0x12abcdef
00010596	PUSH	s_after_maybe_inline:_v=_%08x_000108c2
0001059b	PUSH	0x1
0001059d	CALL	__printf_chk

Unlike in the unoptimized code, we see here that the call to `maybe_inline` has been eliminated, and the constant value ❶ returned by `maybe_inline` is pushed directly onto the stack to be used as an argument for the call to `printf`. This optimized version of the function call is identical to what you would see if the function had been designated inline.

Having examined some of the ways that optimizations can influence the code generated by compilers, let's turn our attention to the different ways that compiler designers choose to implement language-specific features when language designers leave implementation details to the compiler writers.

Compiler-Specific C++ Implementations

When a language permits a programmer to do A, B, and C, it's up to the compiler writers to find a way to make these things possible. C++ gives us three excellent examples of behaviors required by the language, but whose implementation details were left to the compiler writer to sort out:

- Within a nonstatic member function of a class, programmers may refer to a variable named `this` that they never explicitly declare.
- C++ permits function overloading, which means programmers are free to reuse function names as often as they like, subject to restrictions on their parameter lists.
- C++ supports type introspection through the use of the `dynamic_cast` and `typeid` operators.

See Chapter 8 for the compiler's treatment of this. In this section, we discuss how the compiler handles the other two behaviors, function overloading and type introspection.

Function Overloading

Function overloading in C++ allows programmers to name functions identically, with the caveat that any two functions that share a name must have different parameter sequences. Name mangling, introduced in Chapter 8, is the under-the-hood mechanism that allows overloading to work by ensuring that no two symbols share the same name by the time the linker is asked to do its job.

Often, one of the earliest signs that you are working with a C++ binary is the presence of mangled names. The two most popular name mangling schemes are Microsoft's and the Intel Itanium ABI. The Intel standard has been widely adopted by other Unix compilers, such as g++ and clang. The following shows a C++ function name and the mangled version of that name under both the Microsoft and Intel schemes:

Function `void SubClass::vfunc1()`

Microsoft scheme `?vfunc1@SubClass@@UAEXXZ`

Intel scheme `_ZN8SubClass6vfunc1Ev`

Most languages that permit overloading, including Objective-C, Swift, and Rust, incorporate some form of name mangling at the implementation level. A passing familiarity with name-mangling styles can provide you with clues about a program's original source language as well as the compiler used to build the program.

RTTI Implementations

In Chapter 8, we discussed C++ runtime type identification (RTTI) and the lack of a standard for implementing RTTI by a compiler. In fact, the C++

standard does not mention RTTI at all, so it should come as no surprise that implementations differ.

To support the `dynamic_cast` operator, RTTI data structures record not only a class's name but its entire inheritance hierarchy, including any multiple inheritance relationships. Locating RTTI-related data structures can be extremely useful in recovering the object model of a program, but Ghidra's ability to automatically recognize RTTI-related constructs within a binary varies across compilers.

Microsoft C++ programs contain no embedded symbol information, but Microsoft's RTTI data structures are well understood. Ghidra will locate them when present and then summarize any RTTI-related information in the Symbol Tree's *Classes* folder, which will contain an entry for each class that Ghidra locates using its RTTI analyzer.

Programs built with g++ include symbol table information unless they have been stripped. For unstripped g++ binaries, Ghidra relies exclusively on the mangled names it finds in the binary, and it uses those names to identify RTTI-related data structures and the classes they are associated with. As with Microsoft binaries, it will include any RTTI-related information in the Symbol Tree's *Classes* folder.

One strategy for understanding how a specific compiler embeds type information for C++ classes is to write a simple program that uses classes containing virtual functions. After compiling the program, you can load the resulting executable into Ghidra and search for instances of strings that contain the names of classes used in the program. Regardless of the compiler used to build a binary, one thing that RTTI data structures have in common is that they all reference, in some manner, a string containing the mangled name of the class that they represent. Using extracted strings and data cross-references, it should be possible to locate candidate RTTI-related data structures within the binary. The last step is to link a candidate RTTI structure back to the associated class's vtable, which is best accomplished by following data cross-references backward from the candidate RTTI structure until you reach a table of function pointers (the vtable). Let's walk through an example that uses this method.

Example: Locating RTTI Information in a Linux x86-64 g++ Binary

To demonstrate these concepts, we created a small program with a `BaseClass`, a `SubClass`, a `SubSubClass`, and a collection of virtual functions unique to each. The following listing shows part of the main program we used to reference our classes and functions:

```
BaseClass *bc_ptr_2;
srand(time(0));
if (rand() % 2) {
    bc_ptr_2 = new SubClass();
}
else {
    bc_ptr_2 = new SubSubClass();
}
```

We compiled the program using g++ to build a 64-bit Linux binary with symbols. After we analyze the program, the Symbol Tree provides the information shown in Figure 20-3.

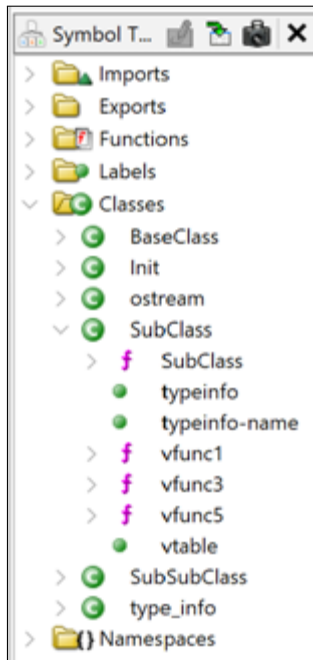


Figure 20-3: The Symbol Tree classes for an unstripped binary

The *Classes* folder contains entries for all three of our classes. The expanded *SubClass* entry reveals additional information that Ghidra has uncovered about it. The stripped version of the same binary contains a lot less information, as shown in Figure 20-4.

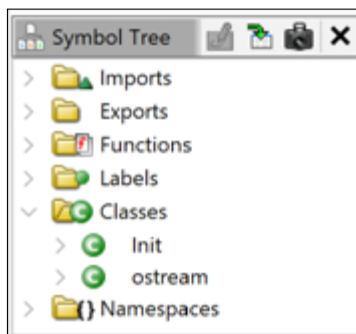


Figure 20-4: The Symbol Tree classes for a stripped binary

In this case, we might incorrectly assume that the binary contains no C++ classes of interest, although it is likely a C++ binary based on the reference to

a core C++ class (ostream). Since stripping removes only symbol information, we may still be able to find RTTI information by searching for class names in the program's strings and walking our way back to any RTTI data structure. A string search yields the results shown in Figure 20-5.

...	...	Label	Code Unit	String View	Strin...	Le...	Is ...
A	00101767	s_typdeid(bc_ptr)=_00101767	ds "typdeid(bc_ptr)..."	"typdeid(bc_...	string	20	true
A	0010177b	s_typdeid(*bc_ptr)=_0010177b	ds "typdeid(*bc_ptr)..."	"typdeid(*bc_...	string	20	true
A	0010178f	s_typdeid(sc_ptr)=_0010178f	ds "typdeid(sc_ptr)..."	"typdeid(sc_...	string	20	true
A	001017a3	s_typdeid(*sc_ptr)=_001017a3	ds "typdeid(*sc_ptr)..."	"typdeid(*sc_...	string	20	true
A	001017b7	s_typdeid(bc_ptr_2)=_001017b7	ds "typdeid(bc_ptr_2)..."	"typdeid(bc_...	string	22	true
A	001017cd	s_typdeid(*bc_ptr_2)=_001017cd	ds "typdeid(*bc_ptr_2)..."	"typdeid(*bc_...	string	22	true
A	001017e8	s_P8SubClass_001017e8	ds "P8SubClass"	"P8SubClass"	string	11	true
A	001017f8	s_P9BaseClass_001017f8	ds "P9BaseClass"	"P9BaseClass"	string	12	true
A	00101808	s_11SubSubClass_00101808	ds "11SubSubClass"	"11SubSubC..."	string	14	true
A	00101818	s_8SubClass_00101818	ds "8SubClass"	"8SubClass"	string	10	true

Figure 20-5: The string search results revealing class names

If we click the "8SubClass" string, we are taken to this portion of the Listing window:

```

s_8SubClass_00101818      XREF[1]:  00301d20(*)
00101818 ds "8SubClass"

```

In g++ binaries, RTTI-related structures contain references to the corresponding class name string. If we follow the cross-reference on the first line to its source, we arrive at the following section of the disassembly listing:

```

PTR_vtable_00301d18      XREF[2]:  ❶ FUN_00101241:00101316 (*),
                                ❷ 00301d10 (*)
❸ 00301d18 pointer __cxxabiv1::__si_class_type_info::vtable+0x10 = ??
❹ 00301d20 addr      s_8SubClass_00101818                      = "8SubClass"
00301d28 addr      PTR_vtable_00301d30                        = 00303028

```

The source of the cross-reference ❹ is the second field within SubClass's typeinfo structure, which starts at address 00301d18 ❸. Unfortunately, unless you are willing to dive into the source code for g++, structure layouts like this are just something you need to learn by experience.

Our last remaining task is to locate SubClass's vtable. In this example, if we follow the lone cross-reference to the typeinfo structure that originates from a data region ❷ (as the other cross-reference ❶ originates from a function and can't possibly be the vtable), we hit a dead end. A little math tells us that the cross-reference originates from the location immediately preceding the typeinfo struct (00301d18 - 8 = 00301d10). Under normal circumstances, a cross-reference would exist from the vtable to the typeinfo structure; however, lacking symbols, Ghidra fails to create that reference.

Since we know that another pointer to our typeinfo structure must exist somewhere, we can ask Ghidra for help. With the cursor positioned at the start of the structure ❸, we can use the menu option Search ► For Direct References, which asks Ghidra to find the current address in memory for us. Figure 20-6 shows the results.

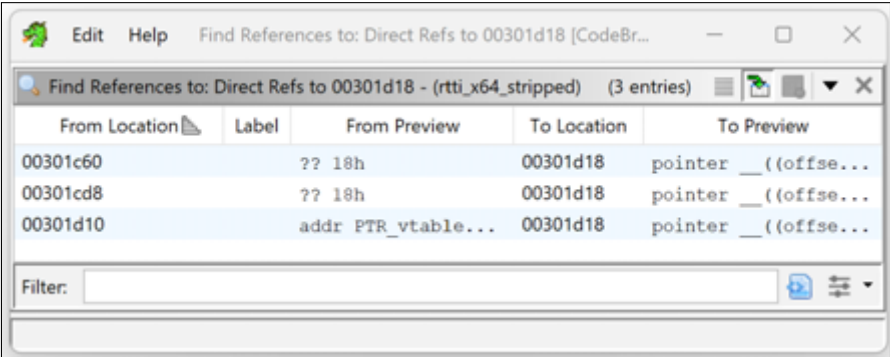


Figure 20-6: The results of a direct reference search

Ghidra has found two additional references to this typeinfo structure. Investigating each of them finally leads us to a vtable:

❶	00301c60	??	18h	?	❷ -> 00301d18
	00301c61	??	1Dh		
	00301c62	??	30h	0	
	00301c63	??	00h		
	00301c64	??	00h		
	00301c65	??	00h		
	00301c66	??	00h		
	00301c67	??	00h		
	PTR_FUN_00301c68			XREF[2]: FUN_00101098:001010b0(*), FUN_00101098:001010bb(*)	
❸	00301c68	addr	FUN_001010ea		
	00301c70	addr	FUN_00100ff0		
	00301c78	addr	FUN_00101122		
	00301c80	addr	FUN_00101060		
	00301c88	addr	FUN_0010115a		

Ghidra has not formatted the source ❶ of the typeinfo cross-reference as a pointer (which explains the lack of a cross-reference), but it does provide an EOL comment that hints at it being a pointer ❷. The vtable itself begins 8 bytes later ❸ and contains five pointers to virtual functions belonging to SubClass. The table contains no mangled names because the binary has been stripped.

In the next section, we apply this “follow the breadcrumbs” analysis technique to help identify the main function in C binaries generated by several compilers.

Locating the main Function

From a programmer's perspective, program execution typically begins with the `main` function, so it's not a bad strategy to start analyzing a binary from the `main` function. However, compilers and linkers (and the use of libraries) add code that executes before `main` is reached. Thus, it's often inaccurate to assume that the entry point of a binary corresponds to the `main` function written by the program's author.

In fact, the notion that all programs have a `main` function is a C/C++ compiler convention rather than a hard-and-fast rule for writing programs. If you have ever written a Windows GUI application, you may be familiar with the `WinMain` variation on `main`. Once you step away from C/C++, you may find that other languages use other names for their primary entry-point function. We refer to this function generically as the `main` function.

All executables must designate an address within the binary as the first instruction to execute after the binary file has been mapped into memory. Ghidra refers to this address as `entry` or `_start`, depending on the file type and the availability of symbols. Most executable file formats specify this address within the file's header region, and Ghidra loaders know exactly how to find it. In an ELF file, the entry point address is specified in a field named `e_entry`, while PE files contain a field named `AddressOfEntryPoint`. A compiled C program, regardless of the platform the executable is running on, has code at the entry point, inserted by the compiler, to make the transition from a brand-new process to a running C program. Part of this transition involves ensuring that arguments and environment variables provided to the kernel at process creation are gathered and provided to `main` utilizing the C calling convention.

When you open a binary with CodeBrowser for the first time, Ghidra must decide where to position the Listing window. If Ghidra finds a symbol for `main`, it positions the Listing view at the address associated with `main`. In the absence of a `main` symbol, Ghidra positions the initial Listing view at the entry point address defined in the file headers. In this case, you will need to do a bit more work to find `main` on your own. With a little understanding of how executables operate, and a little experience, this shouldn't prove too daunting a task.

NOTE

Your operating system kernel neither knows nor cares what language any executable was written in. Your kernel knows exactly one way to pass parameters to a new process, and that way may not be compatible with your program's entry function. It is the compiler's job to bridge this gap by providing code that executes prior to reaching the entry function.

Now that we know execution begins at a published entry point and eventually reaches the `main` function, we can look at some compiler-specific approaches for this transition.

Example 1: *_start to main with gcc on Linux x86-64*

By examining the start code in an unstripped executable, we can learn exactly how execution reaches `main` for a given compiler on a given operating system. Linux `gcc` offers one of the simpler approaches:

```
    _start
001004f0 XOR     EBP,EBP
001004f2 MOV     R9,param_1
001004f5 POP     RSI
001004f6 MOV     param_1,RSP
001004f9 AND     RSP,-0x10
001004fd PUSH    RAX
001004fe PUSH    RSP=>local_10
001004ff LEA     R8,[__libc_csu_fini ]
00100506 LEA     RCX,[__libc_csu_init ]
0010050d LEA     ❶ RDI,[main ]
00100514 CALL    ❷ qword ptr [-><EXTERNAL>::__libc_start_main ]
```

The address of `main` is loaded into RDI ❶ immediately before a call ❷ is made to a library function named `__libc_start_main`, which means that the address of `main` is passed as the first argument to `__libc_start_main`. Armed with this knowledge, we can easily locate `main` in a stripped binary. The following listing shows the lead-up to the call to `__libc_start_main` in a stripped binary:

```
001004ff LEA     R8,[FUN_001006b0 ]
00100506 LEA     RCX,[FUN_00100640 ]
0010050d LEA     ❶ RDI,[FUN_001005fa ]
00100514 CALL    qword ptr [-><EXTERNAL>::__libc_start_main ]
```

Though the code contains references to three generically named functions, we conclude that `FUN_001005fa` must be `main` because it is being passed as the first argument to `__libc_start_main` ❶.

Example 2: *_start to main with Microsoft's C/C++ compiler*

The Microsoft C/C++ compiler's startup stub is a bit more complicated because the primary interface to the Windows kernel is via *kernel32.dll* (rather than *libc* on most Unix systems), and it provides no C library functions. As a result, the compiler often statically links many C library functions directly into executables. The startup stub uses these and other functions to interface with the kernel to set up your C program's runtime environment.

However, in the end, the startup stub still needs to call `main` and `exit` after it returns. Tracking down `main` among all of the startup code is usually a matter of identifying a three-argument function (`main`) closely followed by a call to `exit`. The following excerpt from this type of binary contains calls to the two functions we are looking for:

```
140001232 CALL API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL::__p__argv
140001237 MOV RDI ,qword ptr [RAX ]
14000123a CALL API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL::__p__argc
14000123f MOV RBX ,RAX
140001242 CALL API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL::_get_initial_narrow_environment ❶
140001247 MOV R8,RAX
14000124a MOV RDX ,RDI
14000124d MOV ECX ,dword ptr [RBX ]
14000124f CALL FUN_140001000 ❷
140001254 MOV EBX ,EAX
140001256 CALL __scrt_is_managed_app
14000125b TEST AL,AL
14000125d JZ LAB_1400012b4
14000125f TEST SIL ,SIL
140001262 JNZ LAB_140001269
140001264 CALL API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL::_cexit ❸
```

Here, `FUN_140001000` ❷ is the three-argument function that turns out to be `main`. The dynamically linked functions `_get_initial_narrow_environment` ❶ and `_cexit` ❸ are particularly useful in helping to identify the setup for the call to `main` and tracing the behavior once `main` returns.

Summary

The sheer volume of compiler-specific behaviors is too large to cover in a single chapter (or even a single book, for that matter). Among other behaviors, compilers differ in the algorithms they select to implement various high-level constructs and the manner in which they optimize generated code. Because a compiler's behavior is heavily influenced by the arguments supplied to the compiler during the build process, it is possible for one compiler to generate radically different binaries when fed the same source with different build options selected.

Mastering all of these variations takes practice, but that is part of what makes reverse engineering such a rewarding skill to develop. It can sometimes be tricky to find help by searching for specific assembly constructs, since the right keywords are not always obvious and each case can look a little different. When you hit that wall, one of the best resources is the community itself. Forums dedicated to reverse engineering give you a place to share code, ask questions, and learn from others who have faced similar challenges. In the process, you will build up the experience that makes the next puzzle easier to solve.

PART V

REAL-WORLD APPLICATIONS

21

OBFUSCATION AND EMULATION



Even under ideal circumstances, disassembly listings can be challenging to understand. Ghidra's many features have made it easier to produce the high-quality disassemblies necessary for understanding the inner workings of a binary, and its capabilities have likely contributed to recent advances in the state of binary reverse engineering.

These advances certainly have not been lost on those who do not want their software to be analyzed. Over the last several years, an arms race of sorts has taken place between reverse engineers and programmers who wish to keep their code secret.

In this chapter, we explore both sides of this contest. First, we examine the techniques used to resist reverse engineering, with a focus on obfuscation. Next, we turn to the reverse engineer's perspective, highlighting how Ghidra supports static analysis and deobfuscation. We then demonstrate how emulation can provide an additional edge in this ongoing struggle, before concluding with ways to integrate static analysis and emulation.

Anti-Reverse Engineering

Reverse engineering is rarely a straightforward process, especially when the binary you're analyzing is actively trying to resist you. Obfuscation techniques are widely used to make binaries harder to understand, whether to protect intellectual property, frustrate malware analysts, or slow down would-be crackers. These techniques don't prevent analysis entirely, but they do change the game, requiring you to think more like a detective and less like a tourist. Before we can talk about how to analyze obfuscated code in Ghidra, we need to understand what kinds of tricks binaries use to throw us off. That brings us to anti-reverse engineering.

Anti-reverse engineering is an umbrella topic that covers all techniques software developers might employ to make reverse engineering their products more challenging. Many tools and techniques exist to assist developers with this goal, and more appear every day. The reverse engineering/anti-reverse engineering ecosystem is similar to the escalating dynamic that plays out between malware authors and antivirus vendors.

As a reverse engineer, you are likely to encounter techniques ranging from trivial to nearly impossible to defeat. The approaches you will be required to use will also vary depending on the nature of the anti-reversing techniques you encounter, and may require some level of comfort with both static and dynamic analysis techniques. In the sections that follow, we discuss some of the more common anti-reversing techniques, why they are employed, and approaches for defeating them.

Obfuscation

Various dictionary definitions will inform you that obfuscation is the act of making something obscure, perplexing, confusing, or bewildering in order to prevent others from understanding the obfuscated item. In the context of this book and the use of Ghidra, the items being obfuscated are binary executable files (as opposed to source files or silicon chips, for example). Obfuscation, by itself, is too broad to be considered an anti-reverse engineering technique. It also fails to cover all known anti-reverse engineering techniques. Specific, individual techniques can often be described as obfuscating or non-obfuscating techniques and, where applicable, we point these out in the sections that follow. It is important to note that there is no one correct way to categorize techniques, as the general categories often overlap in their descriptions. In addition, new anti-reverse engineering techniques are under continuous development, and it is not possible to provide a single all-inclusive list.

Anti-Static Analysis Techniques

Modern binaries, especially those designed to resist analysis, often employ a variety of anti-static analysis techniques. These don't necessarily prevent analysis, but they introduce noise, complexity, and ambiguity, all of which force the analyst to spend more effort. Because these approaches specifically

target disassemblers, they are of particular concern whenever you are using a tool like Ghidra to reverse engineer binaries. Common obfuscation tactics include the following:

Packing Packers conceal a program's true contents by compressing or encrypting the binary on disk and including a small stub that unpacks it at runtime. To a disassembler, much of the file appears as data rather than executable code, leaving little to analyze. Only once the program executes is the original code revealed in memory. For reverse engineers, this adds the extra step of unpacking or emulating the stub to recover the actual binary before meaningful analysis can begin.

Custom virtual machines Some obfuscators go even further by translating critical routines into instructions for a custom virtual machine. These are not full operating system simulators like VMware but lightweight interpreters embedded within the binary itself. Each instruction from the original code is replaced with "virtual opcodes" that only the custom virtual machine understands. To analyze such a binary, the reverse engineer must first understand the design of the virtual machine and then interpret the translated logic, a process that can be far more time-consuming than working with native instructions.

Control-flow obfuscation One of the most common ways to resist analysis is by disrupting a program's normal execution path. Obfuscators insert opaque predicates within control structures, which are conditions that always evaluate to the same result, such as `if ((x * x) % 4 == 2)`. This condition always evaluates to false, but not obviously so. They may also flatten control flow into a dispatcher loop, where a single "state" variable governs which block of code to execute next. Computed jumps (JMP instructions) are often used to scatter logic unpredictably, breaking up what would otherwise be clear function boundaries and basic blocks. The result is a tangled and confusing flow of execution that makes both the disassembler and the human analyst work much harder to discern the program's true structure.

Instruction substitution Instead of inserting junk, some obfuscators replace straightforward instructions with less obvious equivalents. For example, `MOV EAX, 0` may be replaced with `XOR EAX, EAX`, or even with a longer sequence of arithmetic instructions that ultimately zero the register. The end result is identical at runtime, but the transformation makes the intent harder for both humans and automated tools to recognize quickly. This slows down analysis by hiding simple operations behind more complex or unusual encodings.

Junk code insertion Another technique is to fill the binary with instructions that serve no purpose other than creating visual clutter. These can include long sequences of NOPs, meaningless arithmetic such as `ADD EAX, 0`, or blocks of code that appear significant but are never actually executed. At runtime, these instructions have no functional effect, but in a disassembly, they add noise, making it harder to spot the logic that truly matters. Automated tools like Ghidra's decompiler may also be misled

by this extraneous material, requiring the reverse engineer to spend additional time sorting real code from junk.

String obfuscation Strings are often the most revealing clues in a binary, containing error messages, filenames, registry keys, or embedded commands. To keep them hidden, many binaries encode or encrypt strings so that they appear in plaintext only at runtime. Techniques range from simple XOR encoding to Base64 or custom ciphers. Without first reversing the decode logic, a reverse engineer will find only scrambled data where helpful semantic clues should be. This lack of context makes it far harder to infer the binary's purpose through static inspection alone.

API hashing and indirection Obfuscators also target library calls since imported functions often give away a program's behavior. Instead of calling an API such as `LoadLibrary` directly, the binary may compute a hash of the function name and resolve it through a custom lookup table. This indirection makes it difficult for disassemblers to automatically label imported functions, and it forces the analyst to spend extra time reconstructing what the program is really calling. Without clear API references, understanding intent requires a deeper dive into the binary's internal logic.

Symbol stripping Finally, many binaries simply strip away helpful metadata. By removing or mangling debug symbols, function names, and variable identifiers, obfuscators deprive analysts of high-level context that would normally guide their interpretation of the disassembly. Instead of meaningful identifiers, the reverse engineer is left with raw addresses and anonymous instructions. While the underlying logic is still present, the loss of semantic support makes the task of understanding it much more labor-intensive.

Together, these techniques form a toolkit developers (and attackers) use to frustrate static analysis. Yet while they succeed in adding noise and complexity, they do not make analysis impossible. Ghidra provides powerful capabilities to combat these obstacles, but using them effectively requires careful attention and a mix of strategies. From the reverse engineer's perspective, the challenge is to recognize these obfuscation patterns, strip away the clutter, and recover the underlying logic. Let's shift our focus to static deobfuscation in Ghidra to show how its features support the reconstruction of control flow and data relationships hidden beneath layers of obfuscation.

Static Deobfuscation

The techniques described in the previous section show how adversaries attempt to frustrate static analysis by disrupting control flow, cluttering code with meaningless instructions, concealing data, or hiding the true intent of a program. Fortunately, from the reverse engineer's perspective, these obstacles are challenges to be overcome rather than absolute barriers. Ghidra provides a rich set of features that map directly to these challenges, allowing

analysts to restore structure, recover meaning, and ultimately understand a binary without executing it.

Unpacking Binaries

Before tackling code-level obfuscation, reverse engineers must often deal with binaries that have been packed. Packers compress or encrypt the original binary and include a small stub that unpacks it at runtime. To a disassembler, much of the packed file appears as data, leaving little to analyze until the unpacking step is performed. In practice, this means the reverse engineer must first emulate or otherwise recover the unpacked code in memory before meaningful static analysis can begin. Ghidra provides tools for inspecting packed binaries, and unpacking often serves as the essential first step in the deobfuscation workflow.

Navigating Obfuscation

Control-flow obfuscation targets the analyst's ability to follow a program's logic. Flattened code, opaque predicates, and dispatcher loops disguise straightforward branches as sprawling, confusing graphs. Obfuscators also rely heavily on indirect jumps and computed calls, where execution targets are calculated at runtime through tables or arithmetic, breaking the visible link between call sites and their true destinations.

In advanced cases, obfuscators replace native code altogether with custom virtual machines. Instead of executing normal x86 or ARM instructions, the binary embeds an interpreter and translates routines into a custom set of "virtual opcodes." Reverse engineering then requires first identifying the virtual machine interpreter, reconstructing the virtual instruction set, and finally mapping it back to the program's intended logic. This is one of the most time-consuming forms of obfuscation, but with careful use of Ghidra's disassembly, annotation, and scripting capabilities, analysts can gradually reverse engineer the virtual machine's design.

For the reverse engineer, the goal is to untangle these patterns and restore clarity. In Ghidra, this often means:

- Restructuring flattened or overly linearized code by identifying state variables and dispatcher loops
- Renaming and labeling functions, jump tables, or dispatcher blocks to clarify structure
- Using tools such as Edit Flow Override, Set Equate, and instruction overrides to clarify opaque logic
- Recognizing disassembly desynchronization and resynchronizing views using byte-level inspection and instruction overrides

Even a small step such as renaming a dispatcher variable can make a flattened routine more understandable. For more complex cases, analysts often use p-code stepping or an emulator to simulate how variables and indirect

jumps influence program flow. By incrementally annotating and restructuring, a confusing control graph can be transformed into something far more readable.

Resolving Indirect Jumps and Obfuscated Calls

Instruction substitution and indirect calls are intended to obscure how a program actually transfers control. Instead of a clear `CALL` to a function, the binary may compute an address through arithmetic, lookups, or pointer chains. Ghidra can help reverse engineers resolve this indirection and reconnect calls to their true targets. A practical workflow in Ghidra might involve these steps:

- Identifying the indirect call by locating instructions such as `CALL ECX`
- Tracing how the target register was populated using p-code analysis or register tracking
- Inspecting memory tables or data regions to reinterpret them as arrays of pointers
- Using emulation, debugging, or scripting when calculations are too complex to resolve by inspection

Once the analyst identifies table entries or reconstructs function pointers, they can rename and annotate indirect calls to reflect their true purpose. This not only restores clarity for the current function but also makes later analysis much easier.

Recovering Data and Strings

String obfuscation, API hashing, and symbol stripping all aim to remove or disguise semantic clues that would normally help an analyst understand a program's behavior. Plaintext strings may be XOR-encoded, hashed function calls may replace direct imports, and symbols may be stripped from binaries entirely. Ghidra provides assistance to help you reverse these techniques statically:

- You can often recover obfuscated strings by recognizing common encoding schemes such as XOR, ROT, or base64-like patterns and then replicating the decode logic with a script or through p-code stepping, emulation, or debugging.
- To address API hashing and pointer-based resolution, you can analyze tables and interpret memory contents as function pointers and then annotate and rename them.
- Symbol stripping removes helpful identifiers, but reverse engineers can counter this fact by applying consistent naming conventions, labeling data structures, and adding comments as they uncover functionality.

These efforts return essential context to the analyst, making it easier to connect code fragments to their actual roles within the binary. For example,

an XOR-encoded string may not appear in Ghidra's Strings window, but once the decode loop is identified, the plaintext can be recovered and annotated for later reference. These steps reintroduce the semantic anchors that obfuscation was designed to hide.

Luckily, Ghidra gives us a variety of tools for recovering and reanalyzing these strings statically, even when they're not immediately visible in the .rodata or .data sections. Whether strings are encoded in place or dynamically constructed at runtime, the reverse engineering process often involves recognizing how the data is hidden, analyzing the decode logic, and then either scripting or emulating the process to retrieve the original content.

SANDBOX ENVIRONMENTS

The purpose of a *sandbox environment* for reverse engineering is to allow programs to be executed in a way that makes their behavior observable without putting critical components of the analyst's platform, or any connected systems, at risk. Sandboxes are commonly constructed using virtualization software, though they can also be built on dedicated systems that can be restored to a known-good state after executing malware.

In addition to providing isolation, sandbox systems are typically heavily instrumented so that analysts can observe and collect detailed information about program behavior. This may include filesystem activity, registry modifications on Windows, or network traffic generated by the program. Because of this instrumentation, sandboxes are often good environments in which to debug binaries that may be malicious since debugging them directly on a host system could be dangerous.

Another advantage of sandboxes is that the environment can be defined more precisely than would be practical on a normal system. Analysts can adjust the system clock, set up controlled network responses such as DNS queries, or create specific user accounts, registry keys, language packs, or filesystem paths. This flexibility makes it possible to test how a binary behaves under targeted conditions, which can reveal hidden functionality or environment-specific triggers that would otherwise go unnoticed.

Emulating Code

While static analysis provides a strong foundation for reverse engineering, there are limits to what you can uncover without running any code. Obfuscated binaries often depend on runtime behaviors such as dynamic string decoding, computed jumps, or conditional logic that only become clear when the program is running. Unfortunately, it's not always safe to execute a suspicious binary.

This is where Ghidra's emulator becomes valuable. By simulating execution in a controlled environment, you can observe how values change, follow control flow, and decode hidden content without ever launching the program. Let's begin our discussion of emulation by comparing it to traditional live debugging.

Emulation vs. Live Debugging

At first glance, emulation and debugging may seem like the same thing. Both allow you to step through code, inspect register values, and observe memory changes. But in practice, especially in Ghidra, they serve different purposes and are used in different contexts.

Debugging typically refers to interacting with a live process. This might involve launching the binary within a debugger or attaching to an already running instance. In this case, you are executing real code on real hardware or in a virtual machine, which introduces real risks. Malware could perform harmful tasks, anti-debugging checks could activate, and the system state might change in unpredictable ways. While this approach can be useful for analyzing malware or heavily obfuscated software, it can also be risky and noisy.

Ghidra can emulate the behavior of the CPU within the static analysis environment. In this case, you are not actually running the program; instead, you are stepping through instructions one at a time and observing how they affect registers and memory in a controlled space. Emulation is like asking Ghidra to show what would happen if a particular instruction were executed, without running anything for real. This makes emulation especially useful for reverse engineers who need to evaluate logic, decrypt data, or trace control flow without executing the full binary.

For example, imagine you have identified a small loop that decodes a string using arithmetic operations. Rather than manually working through the instructions or running the full program to see the result, you can emulate just that loop, observe the changes in memory, and recover the decoded output. This approach is safer, repeatable, and often easier to isolate.

In Ghidra, the Debugger plugin, which we added to Ghidra to create SimpleDebugger in Chapter 10, includes emulator functionality that operates on static binaries, giving you the benefits of dynamic inspection without requiring a running process.

In short, debugging runs the actual code, which can be useful but may also be risky. Emulation, which appears to provide similar functionality, allows you to observe the changes that an instruction (or series of instructions) would make to the state of a system in a safe and static environment. It is especially helpful for investigating obfuscation, decoding routines, and tricky control flow without allowing the binary to execute freely.

The Ghidra Emulator

Before you can emulate code in Ghidra, you need to set up your environment. The Debugger plugin, which we added to Ghidra in 12, includes a built-in CPU emulator that works directly with loaded binaries. You're not attaching to a live process here; instead, you're creating a virtualized execution context that lets you see how the code would run, instruction by instruction.

The setup process has a few steps:

1. Open the binary in the emulator tool (the gear icon on the Project Manager window).
2. Start an emulation trace (Debugger ► Emulate Program in New Trace).
3. Initialize registers and memory (if necessary).
4. Start the emulation.

Setting up the emulator may feel like extra overhead at first, especially compared to clicking *run* in a traditional debugger. The benefit is precision. Using the emulator, you can test just the parts of the program you care about, under conditions you fully control, without the risks or noise of real execution. Once you get comfortable creating traces and initializing state, Ghidra's emulator becomes one of your most reliable allies in reverse engineering, particularly when dealing with obfuscated or suspicious code.

Use Cases

When does it make sense to use the Ghidra emulator? Emulation is especially useful in situations where static analysis becomes difficult. These include cases where control flow is obscured, strings are generated or decoded during execution, or values are computed dynamically in ways that are hard to follow by simply reading the disassembly. Unlike full debugging, which requires launching or attaching to a live process, emulation lets you selectively execute just the instructions you care about. This makes it ideal for examining suspicious functions, decoding routines, or resolving indirect control flow without running the entire binary or exposing your system to risk.

Here are a few typical use case examples based on the static deobfuscation techniques described earlier in this chapter. (We have reorganized the emulator tool windows for clarity.)

Example 1: Identifying Opaque Predicates

We mentioned that obfuscated binaries often use logic that always evaluates to true or false in ways that are not obvious. Consider the example presented earlier in this chapter, where a control structure like `if ((x * x) % 4 == 2)` looks legitimate but is always false for integers. You can emulate the surrounding code to see how variables like `x` are initialized and verify that the branch never gets taken to help you prune away paths.

Example 2: Deobfuscating Function Control Flow

At times, you might observe interesting control flow, like the following:

```
MOV EAX, [EDX+ECX*4]
CALL EAX
```

Without context, it's not clear where the `CALL` goes. You could spend time statically tracing back through table definitions and register setup, or

you could set up the emulator, initialize registers manually, and step forward to observe which address is loaded into EAX. This approach saves time and can make you more confident in your analysis, especially when you've combined it with memory inspection and the effective use of comments and annotations.

Example 3: Emulating a String Decode Routine

Suppose a binary you are analyzing includes this encoded array where the array content has been XOR'd with 0x55

```
char encoded[] = { 0x3d, 0x30, 0x39, 0x39, 0x3a, 0x74, 0x55 };
```

and has the following decode logic:

```
for (int i = 0; i < sizeof(encoded); i++) {
    encoded[i] = encoded[i] ^ 0x55;
}
```

Using emulation, you can retrieve a hidden string without ever running the binary, and without needing to implement the logic in a script or needing to guess the XOR key. Here is one approach to decoding this content:

1. Add the binary to a Ghidra project and then open it in the emulator (gear icon) and auto analyze it.
2. Create a new trace (Debugger ► Emulate Program in a New Trace). Navigating to the encoded string in Figure 21-1 confirms the bytes are still encoded.

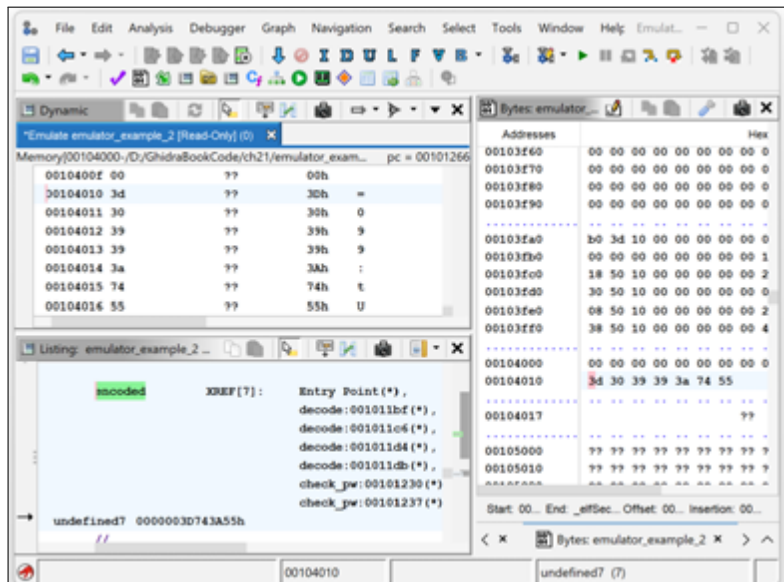


Figure 21-1: The encoded string in the Ghidra emulator

3. In the Registers window, set RIP to 1011a9, (the first instruction in the decode function) and set a breakpoint at 1011ec in the Listing window using the right-click context menu.
4. Clicking the green “play” button in the toolbar runs the emulator to the breakpoint. While the Listing window retains the encoded content, Figure 21-2 shows that the emulator’s dynamic window contains the decoded string “hello!”.

Address	Offset	Hex	ASCII
0010400c	00	??	00h
0010400d	00	??	00h
0010400e	00	??	00h
0010400f	00	??	00h
00104010	68	??	68h h
00104011	65	??	65h e
00104012	6c	??	6Ch l
00104013	6c	??	6Ch l
00104014	6f	??	6Fh o
00104015	21	??	21h !
00104016	00	??	00h

Figure 21-2: The decoded string in the Ghidra emulator

5. You can then run the program from the command to test and confirm your findings.

As you have seen, emulation is best used as a precise and focused tool. It allows you to verify control flow, decode data that is processed at runtime, and understand dynamic behavior, all within Ghidra’s static analysis environment. Once you feel comfortable with setting up the emulator, you will likely turn to it frequently when static analysis does not provide the full picture.

Building an Emulator

Emulators vary in capabilities. At a minimum, an emulator requires a stream of instruction bytes and sufficient memory to dedicate to stack operations and processor registers. More sophisticated emulators may provide access to emulated hardware devices and operating system services.

Fortunately, Ghidra provides a rich `Emulator` class and an `EmulatorHelper` class, which provides a higher-level abstraction of common emulator functionality and facilitates the quick-and-easy creation of emulation scripts. Ghidra can emulate p-code instructions as well. Recall from Chapter 18 that p-code is an intermediate representation of the underlying assembly that allows the decompiler to work against a variety of target architectures. Ghidra’s `ghidra.pcode.emulate.Emulate` class provides the ability to emulate p-code instructions.

We can use Ghidra's emulator-related classes to build emulators that mimic a wide variety of processors. As with other Ghidra packages and classes, the Javadoc supplied with Ghidra documents this functionality, and you can pull it up as a reference by clicking the red plus tool in the Script Manager window. If you're interested in writing emulators, we encourage you to check out the Javadoc associated with the emulator methods used in the following example.

Let's walk through the creation of an emulation script using a crackme challenge as an example. In this case, the binary contains encoded content that eventually serves as the body of a function. Even with access to the source code, solving the challenge requires effort. Working through this example will demonstrate how to create a simple emulator that can not only tackle the crackme but also serve as a versatile tool for addressing a wide range of analysis challenges.

```
❶ unsigned char check_access[] = {
    0xf0, 0xed, 0x2c, 0x40, 0x2c, 0xd8, 0x59, 0x26, 0xd8,
    0x59, 0xc1, 0xaa, 0x31, 0x65, 0xaa, 0x13, 0x65, 0xf8, 0x66
};
unsigned char key = 0xa5;
void unpack() {
    for (int ii = 0; ii < sizeof(check_access); ii++) {
        ❷ check_access[ii] ^= key;
    }
}
void do_challenge() {
    int guess;
    int access_allowed;
    int (*check_access_func)(int);
    ❸ unpack();
    printf("Enter the correct integer: ");
    scanf("%d", &guess);
    check_access_func = (int (*)(int))check_access;
    ❹ access_allowed = check_access_func(guess);
    if (access_allowed) {
        printf("Access granted!\n");
    } else {
        printf("Access denied!\n");
    }
}
int main() {
    do_challenge();
    return 0;
}
```

From the source code, we can see the encoded content ❶. Ghidra's decompiler is frequently an awesome partner for solving crackme challenges, but this one has interesting characteristics that complicate the process. Ghidra sees only the encoded function body, but we need to know the function's actual purpose before we can solve the challenge. At runtime, the `unpack` ❷ function call results in the decoding of the `check_access` ❸ function before `check_access` is called ❹. Let's build an emulator script in Ghidra to help us attack this challenge.

REALITY CHECK

If you are thinking, "Wait a second, in a modern system you can't just start executing code that lives outside the `.text` section," you are absolutely right. A real binary would need to allocate memory, mark it executable, and then jump into it. Without those steps, the program would crash spectacularly the moment it tried to run the decoded function. To keep the focus on emulation rather than operating system page tables, we are pretending here that the decoded function is directly executable. Think of it as a classroom demo where we sweep a little complexity under the rug so we can concentrate on the part that matters: the use of Ghidra's emulator to tackle obfuscation.

If you do want to try this in a more realistic setting, you can experiment in Ghidra by writing the decoded bytes into a new memory block, adjusting its permissions, and then pointing execution at that block. This is essentially what a loader or unpacker would do before transferring control.

Step 1: Defining the Problem

Our task is to design and develop a simple emulator that will allow us to choose a region of a disassembly and will emulate the instructions in that region. The emulator needs to be added to Ghidra and be available as a script. For example, if we select the `unpack` function for the crackme challenge and run the script, our emulator should use the key to unpack the `check_access` array and let us know the solution to the crackme challenge. The script should write the unpacked code bytes into the program's memory in Ghidra.

Step 2: Creating the Eclipse Script Project

Create a project called *SimpleEmulator* using GhidraDev ► New ► Ghidra Script Project. This gives us a *SimpleEmulator* folder in Eclipse with a folder called *Home scripts* waiting for our new script. We still need to create the script itself and enter metadata used to document and catalog the script. The metadata collected in the script creation dialog is included in the file (Figure 21-3).

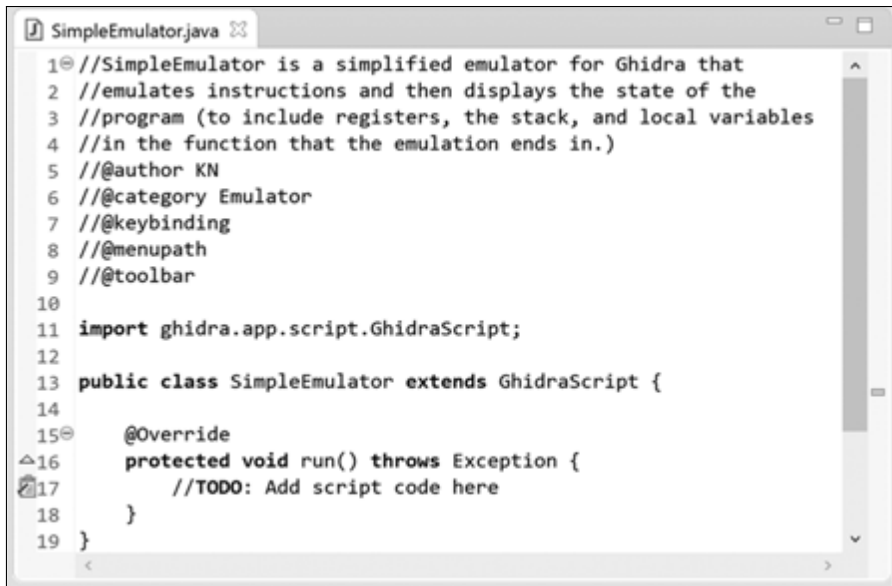


Figure 21-3: The script template for SimpleEmulator

We will add the code where the template says Add script code here.

Step 3: Setting Up the Emulator

We know that Eclipse will recommend imports if we need them as we develop our code, so we can jump right into the coding tasks we need to perform and add the recommended import statements when Eclipse detects that we need them.

For functionality, we will rely on the following instance variable declarations throughout the SimpleEmulator class:

```

private EmulatorHelper emuHelper;    // EmulatorHelper member variable object
private Address executionAddress;    // Initially the start of the selection
private Address endAddress;          // End of the selected region

```

Comments associated with each declaration describe the purpose of each variable. We initially set the executionAddress to the start of the selected range, but we will also use it to advance through the selection.

The first thing we will do in our script's run method is instantiate our emulator helper object and activate the tracking of any memory written in the emulator so we can write updated values back into the current program:

```

emuHelper = new EmulatorHelper(currentProgram);
emuHelper.enableMemoryWriteTracking(true);

```

The instantiation acts as a lock, similar to the lock that the CodeBrowser places on an open binary.

Step 4: Selecting the Address Range to Emulate

Because we want the user to choose the section of code to be emulated, we need to ensure they have selected something in the Listing window:

```
if (currentSelection != null) {
    executionAddress = currentSelection.getMinAddress();
    endAddress = currentSelection.getMaxAddress().next();
} else {
    println("Nothing selected");
    return;
}
```

Otherwise, we generate an error message.

Step 5: Checking for an Instruction

Within the selection, we want to ensure we are looking at an instruction in order to establish the initial processor context, initialize the stack pointer, and set up a breakpoint at the end of the selected region:

```
Instruction executionInstr = getInstructionAt(executionAddress);
if (executionInstr == null) {
    printerr("Instruction not found at: " + executionAddress);
    return;
}
long stackOffset = (executionInstr.getAddress().getAddressSpace().
    getMaxAddress().getOffset() >>> 1) - 0x7fff;
emuHelper.writeRegister(emuHelper.getStackPointerRegister(), stackOffset);
// Set up breakpoint at the end address.
emuHelper.setBreakpoint(endAddress);
// Set continuing to false as we are just starting the emulation.
boolean continuing = false;
```

The continuing flag indicates whether we are just starting the emulation or continuing the emulation, and determines which version of `emuHelper.run` is called in the next step.

Step 6: Performing Emulation

To perform the emulation, we will use some of the Ghidra API functions introduced in Chapter 14, such as `monitor.isCancelled`. We use a loop to drive the emulation until the script reaches a defined termination condition:

```
❶ while (!monitor.isCancelled() &&
    !emuHelper.getExecutionAddress().equals(endAddress)) {
    if (continuing) {
        emuHelper.run(monitor);
```

```

    } else {
        emuHelper.run(executionAddress, executionInstr, monitor);
    }
    ❷ executionAddress = emuHelper.getExecutionAddress();
    // Determine why the emulator stopped, and handle each possible reason.
    ❸ if (emuHelper.getEmulateExecutionState() ==
        EmulateExecutionState.BREAKPOINT) {
        continuing = true;
    } else if (monitor.isCancelled()) {
        println("Emulation cancelled at 0x" + executionAddress);
        continuing = false;
    } else {
        println("Emulation Error at 0x" + executionAddress +
            ": " + emuHelper.getLastErrorMessage());
        continuing = false;
    }
    ❹ writeBackMemory();
    if (!continuing) {
        break;
    }
}

```

Emulation continues as long as the monitor hasn't detected a user cancellation, we haven't reached the end of the selected range of instructions, or an error condition hasn't been triggered ❶. When the emulator stops, we update the current execution address ❷ and then handle the stop condition appropriately ❸. The final step is to call the `writeBackMemory()` method ❹.

Step 7: Writing Memory Back to the Program

Testing this emulator on an unpack routine will change the bytes of the binary, but changes will exist only in the emulator's working memory. We need to write the content back to the binary so Ghidra's user interfaces can accurately reflect these changes. We implement this task in the `writeBackMemory()` method called in the previous step:

```

private void writeBackMemory() {
    ❶ AddressSetView memWrites = emuHelper.getTrackedMemoryWriteSet();
    AddressIterator aIter = memWrites.getAddresses(true);
    Memory mem = currentProgram.getMemory();
    ❷ while (aIter.hasNext()) {
        Address a = aIter.next();
        MemoryBlock mb = getMemoryBlock(a);
        if (mb == null) {
            continue;
        }
        if (!mb.isInitialized()) {
            // Initialize memory.

```



```

        try {
            mem.convertToInitialized(mb, (byte)0x00);
        } catch (Exception e) {
            println(e.toString());
        }
    }
    try {
        ❸ mem.setByte(a, emuHelper.readMemoryByte(a));
    } catch (Exception e) {
        println(e.toString());
    }
}
}
}

```

Ghidra provides functionality within its `emulatorHelper` to facilitate the writing process. Using `emulatorHelper`, we retrieve ❶ the memory that the emulator wrote to its own working memory, iterate ❷ over each of those written bytes, and then write ❸ the bytes back to memory.

Step 8: Cleaning Up Resources

In this step (which is the final statement in the script's `Run` method), we need to clean up resources and release the lock placed on the current program. We can accomplish both tasks in one easy statement:

```
emuHelper.dispose();
```

Because we created this emulator for demonstration purposes only, we omitted many of the comments, functionality, error checking, and error handling normally included in a production script. All that remains is to confirm that the emulator script is able to accomplish our goal.

Step 9: Testing the Script

If you set up the script project as a linked project, Ghidra already knows where to find it. If you did not link your script project (or if you created your emulator script in another editor), you need to save it in one of Ghidra's script directories, as discussed in Chapter 14.

To test the script, we will load the binary associated with the crackme challenge source code. When we navigate to the `unpack` function, we note that it contains references to the `check_access` label:

```
0010077d 48 8d 05 8c 08 20 00 LEA    RAX,[check_access]
```

The code in the Decompiler window contains the following, which does not get us any closer to solving our crackme:

```
check_access[(int)local_c] = check_access[(int)local_c] ^ key;
```

Double-clicking `check_access` within the Listing window leads us to address `00301010`, which does not look like instructions within a function:

```
00301010 f0 ed 2c 40 2c d8 59 undefined1[19]
          26 d8 59 c1 aa 31 65
          aa 13 65 f8 66
```

If we chose to disassemble this content, we would receive a bad data error in Ghidra. The Decompiler window also provides no help for this location.

Let's use our script to see if we can emulate the `unpack` function. When we select the instructions that make up the function, open the Script Manager, and run our script, we see no observable change in the `unpack` function or in the Decompiler window, but if we navigate to `check_access` (`00301010`), the content has changed:

```
00301010 55 48 89 e5 89 7d undefined1[19]
          fc 83 7d fc 64 0f
          94 c0 0f b6 c0 5d c3
```

We can clear these code bytes (hotkey C) and then disassemble (hotkey D) and obtain the following results:

check_access	
00301010 55	PUSH RBP
00301011 48 89 e5	MOV RBP,RSP
00301014 89 7d fc	MOV dword ptr [RBP + -0x4],EDI
00301017 83 7d fc 64	CMP dword ptr [RBP + -0x4],100
0030101b 0f 94 c0	SETZ AL
0030101e 0f b6 c0	MOVZX EAX,AL
00301021 5d	POP RBP
00301022 c3	RET

Here is the corresponding code in the Decompiler window:

```
ulong UndefinedFunction_00301010(int param_1)
{
    return (ulong)(param_1 == 100);
}
```

This proof-of-concept script demonstrates how you can build a general-purpose emulator using Ghidra's emulator support classes and then use your emulator to help you deobfuscate code without ever actually executing potentially malicious code.

Summary

As reverse engineers, we're often handed binaries designed to confuse, mislead, or outright deceive us, where the obvious answers are hidden beneath layers of obfuscation. Whether it's flattened control flow, indirect jumps, or encrypted strings, these techniques aim to frustrate static analysis and complicate reverse engineering.

Emulation in Ghidra gives you the ability to simulate code execution safely, selectively, and with surgical precision. Instead of running a suspicious or hostile binary, you can trace through specific functions, watch how registers and memory evolve, and recover hidden values without leaving the safety of your static workspace.

When you encounter an obfuscated binary, don't panic. Take it apart statically to understand its layout, then emulate the tricky parts to bring clarity where decompilation falls short. Treat emulation not as a magic wand, but as your magnifying glass: perfect for zooming in on the details that matter.

With both approaches in your Ghidra toolkit, you're not just reading binaries, you're reasoning through them. The techniques presented in this chapter are intended to demonstrate that Ghidra is capable of far more than simply generating disassembly listings. We build on this theme in the next chapter as we look at how we can use Ghidra to patch our disassembly listings.

22

PATCHING BINARIES



Occasionally when reverse engineering a binary, you may decide that you want to modify its behavior. Behavioral modification is usually accomplished by inserting, removing, or altering existing instructions. There are many reasons to make such modifications, some more controversial than others:

- Eliminating anti-debug techniques from a malware sample that prevent it from being studied
- Patching vulnerabilities in software for which you have no source code
- Customizing an application's splash screen or string content
- Modifying game logic for the purposes of cheating
- Unlocking hidden features
- Bypassing licensing checks or other anti-piracy protections

In this chapter, we have no intention of teaching you how to do anything unethical, but we discuss the high-level challenges of modifying a binary

to reflect any changes that you have made within Ghidra. In Chapters 14 and 21, we covered the use of emulation scripts to modify the content of a program loaded into Ghidra. These techniques have no effect whatsoever on the original binary file that Ghidra processed during the import process, however. To patch the binary, you'll have to get Ghidra to write changes back to a file on disk. This chapter covers this task and discusses the challenges that different types of patches might pose.

Planning Your Patch

The patching process typically involves the following steps:

1. Determine the type of patch you intend to make. This often depends on your reason for patching.
2. Identify the exact locations of the program contents that need to be patched. This typically involves some amount of research and analysis of the program to be patched.
3. Plan the content of your patch. Content changes may require new data, new machine code, or both. In any case, your changes must be well thought out to prevent the program from exhibiting any unintended behavior.
4. Use Ghidra to replace existing program content (data or code) with your replacement content.
5. Verify that your changes appear to be correctly implemented.
6. Use Ghidra to export your changes into a new binary file.
7. Verify that the new binary file behaves as intended, repeating from step 2 as necessary.

In some patching scenarios, many of these steps are trivial; in others, they pose a much bigger challenge. In the sections that follow, we review the steps that Ghidra can help you with and discuss situations that may push you or Ghidra to your limits. We'll start by reviewing some of the ways that Ghidra helps you locate items of interest in a patching context.

Finding Program Content to Change

The exact nature of your patch will dictate what you need to change. For example, customizing splash screens or strings requires that you locate the original data that needs changing, while changing the logic of a program requires modifying specific parts of the code to change the program's behavior. In either case, you may need to perform a significant amount of reverse engineering to find the content you need to modify. We've covered many Ghidra capabilities that can facilitate these activities. Let's review some of the ones useful for patching.

Memory

When your patch involves modifying program data, your primary means of identifying where to apply your patches will be some form of memory search. The most general memory search is the CodeBrowser Search ► Memory menu option (hotkey S), shown in Figure 22-1 with the advanced options expanded. We discussed the Search Memory dialog in Chapter 6.

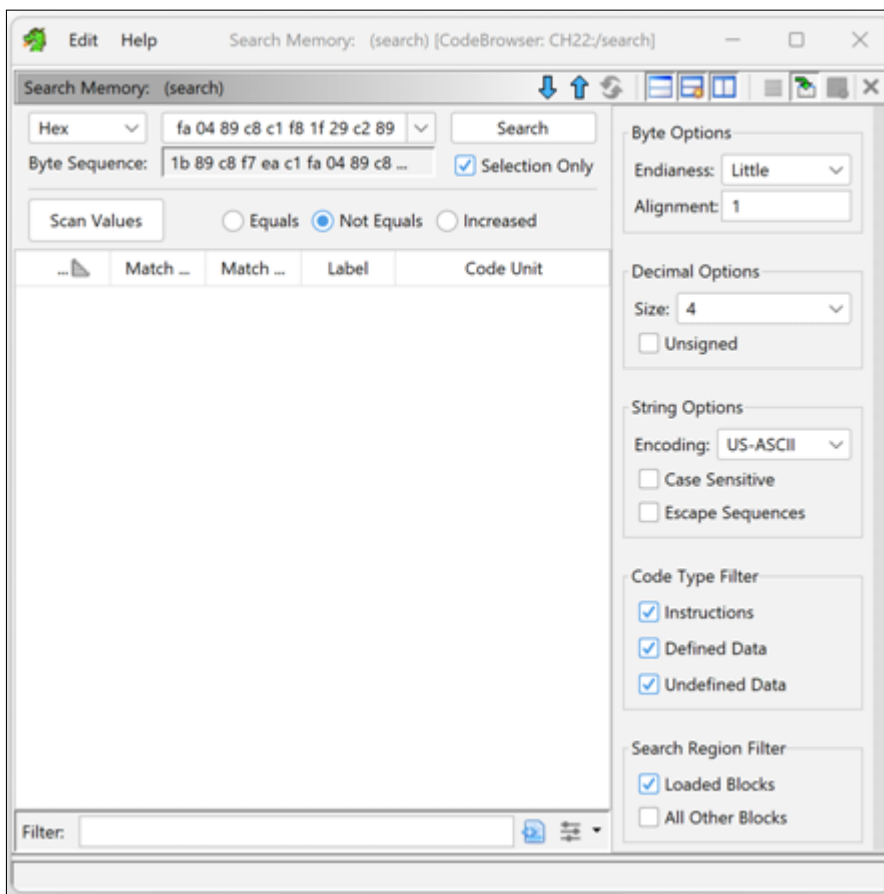


Figure 22-1: The Search Memory dialog

In a patching context, the Search Memory dialog is most useful for finding specific, known data within the binary, such as known strings or hex sequences. A successful search will reposition all linked displays to the location of the matching bytes or, in the case of Search All, open a new dialog containing a list of all addresses at which the matched content may be found.

For very large binaries, it may be useful to limit the scope of your search to specific regions likely to contain a match (such as Instructions, Defined Data, Undefined Data, and so on) by deselecting any uninteresting code unit types.

Keep in mind that Search ► Memory searches the raw byte content of the database, which may not be suitable for finding the type of data you are looking for. For example, Search ► Memory is the wrong choice if you want to search within the body of comments entered into the program. Refer to “Searching Program Text” on page 115 for more information on searching within the disassembly listing itself.

Direct References

In Chapter 20, we used Search ► For Direct References to scan the program’s binary content for all occurrences of a specific address. The most common use for this search type is to locate pointers to interesting data when Ghidra has failed to create a cross-reference to the data. In a patching context, we can use this strategy to update all references to a data or code location and maintain proper relationships between the code and data in the patched binary.

Instruction Patterns

Ghidra’s Search ► For Instruction Patterns feature finds a specific sequence of instructions by matching a pattern. When defining an instruction pattern, you need to strike a delicate balance between patterns that are too specific and patterns that are too general. Let’s look at an example to illustrate this idea. Assume the program includes a `cleanup_and_exit` function that exits the program:

```
int test_even(int v) {
    return (v % 2 == 0);
}
int test_multiple_10(int v) {
    return (v % 10 == 0);
}
int test_lt_100(int v) {
    return v < 100;
}
int test_gte_20(int v) {
    return v >= 20;
}
❶ void cleanup_and_exit(int rv, char* s) {
    printf("Result: %s\n", s);
    exit(rv);
}
```



```

void do_testing() {
    int v;
    srand(time(0));
    v = rand() % 150;
    printf("Testing %d\n", v);
    ❷ if (!test_even(v)) {
        cleanup_and_exit(-1, "failed even test");
    }
    if (test_multiple_10(v)) {
        cleanup_and_exit(-2, "failed not multiple of 10 test");
    }
    if (!test_lt_100(v)) {
        cleanup_and_exit(-3, "failed <100 test");
    }
    if (!test_gte_20(v)) {
        cleanup_and_exit(-4, "failed > 20 test");
    }
    // All tests passed, so do interesting work here.
    ❸ system("/bin/sh");
    cleanup_and_exit(0, "success!");
}
int main() {
    do_testing();
    return 0;
}

```

The `do_testing` function conducts a series of tests ❷. If any of the tests fail, it calls the `cleanup_and_exit` function ❶ and execution ends. If all tests succeed, some very interesting code ❸ will execute. Our patching challenge is to determine where we need to patch to ensure that all of the tests pass so we can reach the interesting code.

If we load the binary into Ghidra, we can search for all calls to `cleanup_and_exit` to search for candidate locations. We have several options to consider:

- We could patch the `cleanup_and_exit` function to return so that a failed test doesn't exit the program but rather continues. However, this isn't an optimal solution because the function is also used for a legitimate exit at the end of the program after it completes the interesting work.
- We could use search functionality to find cross-references to `cleanup_and_exit`. This would give us all of this function's calls, but we wish to patch only some of them.
- We could identify an instruction pattern that the calls have in common and use Search ► For Instruction Patterns to find the correct calls to patch.

To use the instruction pattern search functionality, we need to identify a useful pattern. Each test we are trying to pass takes the following form in the Listing window:

```

001008af  CALL    test_even
001008b4  TEST    EAX,EAX
001008b6  JNZ     LAB_001008c9
001008b8  LEA     RSI,[s_failed_even_test_00100a00]
001008bf  MOV     EDI,0xffffffff
001008c4  CALL    cleanup_and_exit

```

Let's try searching for that instruction sequence by selecting the instruction sequence and choosing Search ► For Instruction Patterns. Doing so should automatically populate the Instruction Pattern Search dialog, as shown in Figure 22-2.

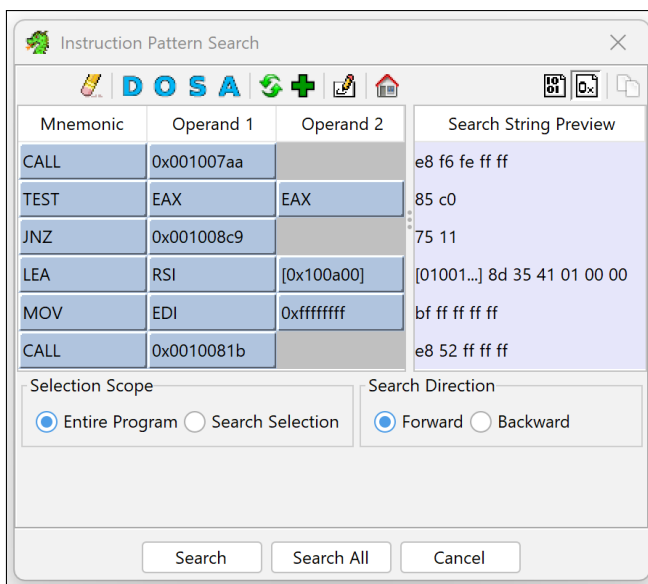


Figure 22-2: The Instruction Pattern Search dialog with all fields selected

If we click Search All, we see only one result: the specific location we selected when we started the search, as shown in Figure 22-3.

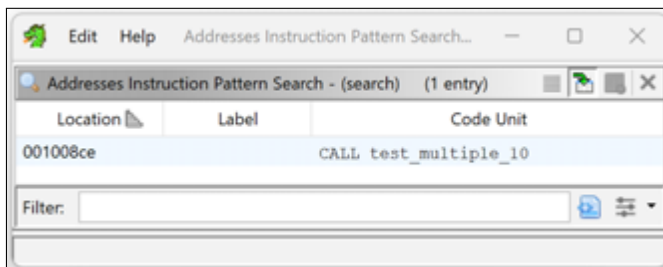


Figure 22-3: The Instruction Pattern Search results from all fields selected

Our issue is that the search included operands that do not remain constant between the test cases. For example, the operand to the first call is the address of a specific test function. To make the search more general, we can deselect the individual mnemonics and operands of any individual instruction in the pattern, as shown in Figure 22-4. Anything that has been deselected becomes a wildcard in subsequent searches.

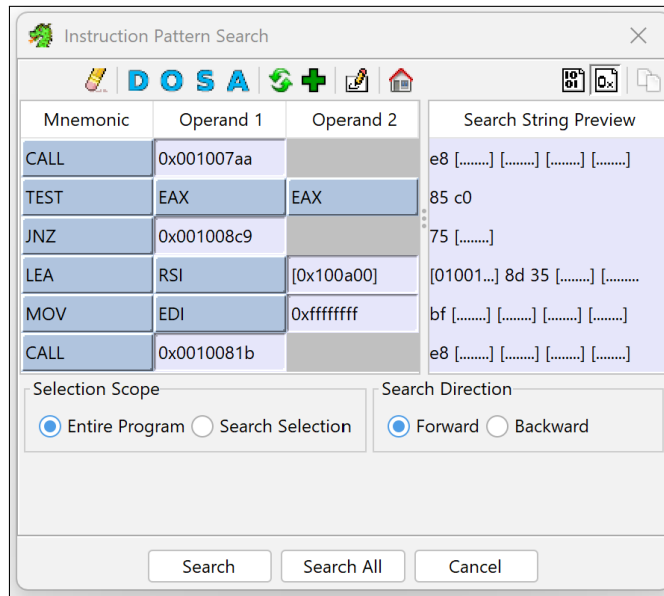


Figure 22-4: The Instruction Pattern Search dialog with some operands deselected

If we click Search All with operand fields disabled, we see the three results shown in Figure 22-5.

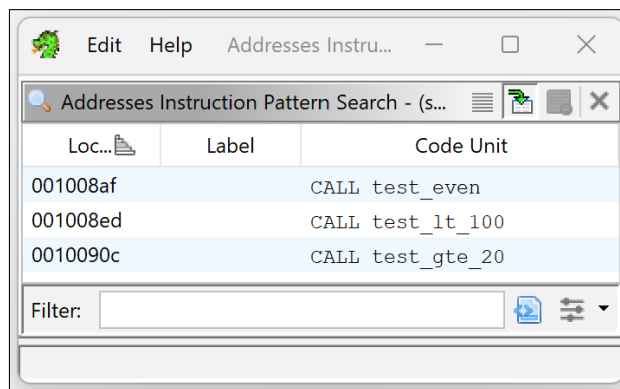


Figure 22-5: The result of deselecting operands

The search still fails to identify the call to `test_multiple_10`, which uses a `JZ` rather than a `JNZ` instruction. Deselecting the mnemonic field for the `JNZ`

instruction and rerunning the search yields the results shown in Figure 22-6, which includes the four calls we wish to patch and does not include the final call to `cleanup_and_exit` that we do not want to patch.

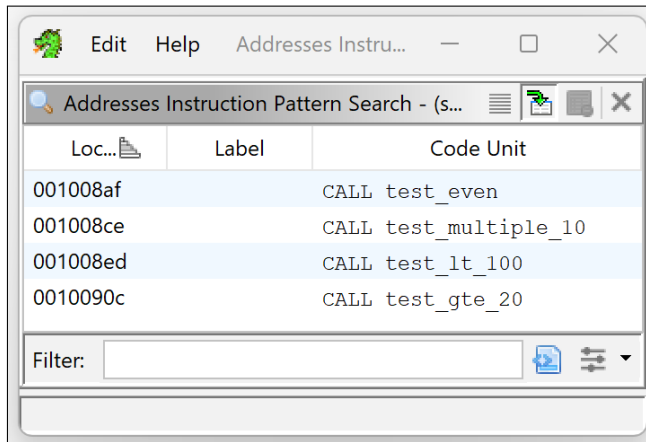


Figure 22-6: Useful search results

This search functionality has a number of uses other than locating candidate instruction patterns for patching. You might apply it to analyzing vulnerabilities or finding specific functionality, for example.

THE SECRET ART OF PROFITABLE PURCHASES

Imagine you are playing a role-playing game that lets you purchase items from shopkeepers (perhaps a shiny new sword you have been saving up for). Normally, when you make a purchase, the game subtracts the cost of the item from your stash of loot. A single machine instruction controls this behavior: a simple SUB operation that says to take this much gold away.

Now imagine you open the game's binary in Ghidra and patch that instruction, changing it from SUB to ADD. The next time you buy an item, not only do you receive the gear you wanted, but instead of losing gold, the game happily adds the cost of the item to your loot pile. Every purchase makes you richer, and the shopkeepers never seem to notice.

Binary patching can be a powerful tool for legitimate reverse engineering work, but this playful example shows how altering just one instruction can completely flip the rules of a program's logic. Using Ghidra, you can see exactly where in the code some behavior takes place and, if you so choose, can make every shopping trip a very profitable experience.

Specific Behaviors

A program's behavior is defined by the instructions it executes combined with the data on which it operates. When your patching task involves modifying a program's behavior, locating the exact behavior you want to modify is usually much more difficult than locating data you wish to change.

Because we can never predict the exact instruction sequence that a compiler might generate for any source code, it can be hard to use Ghidra's automated search features to pinpoint an exact location at which to apply a code patch.

The process of locating specific behaviors boils down to analyzing the program's functions using techniques covered throughout this book. For example, you might carefully review all functions in the binary or traverse the call tree, beginning with a well-known function such as `main`. Other common techniques involve searching for a function's name (assuming the binary has symbols) and using cross-references from interesting data to backtrack to potentially interesting functions.

For instance, say we are interested in locating the authentication-related functions within a binary. We might search for common strings associated with authentication, such as "Please enter your password:" and "Authentication failed". Strings similar to these often bookend an authentication process, and locating functions that reference these strings may significantly reduce our search space for other authentication-related functions.

Regardless of the approach you use, always verify that the function does in fact implement the behavior you wish to modify. In particular, be wary of the names that programmers assign to functions, as there is no guarantee that a function's behavior matches its name.

Applying Your Patch

At long last, your hard work and perseverance have paid off, and you have located the code or data you wish to modify. What now? Once you have developed the replacement content, you need to use Ghidra features to modify the program.

The first thing you need to consider is the size of your new content relative to the content you are replacing. If the new content's size is less than or equal to the original content's size, you are in good shape because your patch will fit within the memory footprint of the original content. Things get a bit tricky when your patch is larger than the original content; we will dedicate some time to this case in "Making Oversized Patches" on page 529.

Making Basic Changes

Whether you have a pile of bytes in hand or need some help from an assembler, you'll eventually need to get your content into Ghidra. For short runs of bytes, you may find it easiest to use Ghidra's built-in byte editor or assembler. For longer runs, you'll probably want to automate the replacement process. The next few sections describe some of Ghidra's byte-level editing features.

Using the Byte Viewer

The Ghidra Byte Viewer (Window ► Bytes) provides a standard hex dump view of the raw byte content at the current listing location, synchronized with every other linked window, as shown in Figure 22-7.

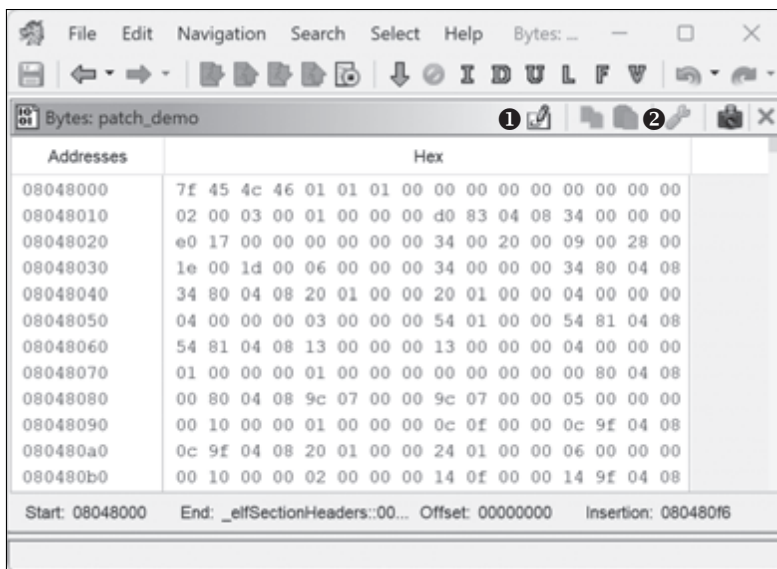


Figure 22-7: The Ghidra Byte Viewer

The Byte Viewer can also double as a hex editor if you toggle the Edit Mode tool ❶, a convenient option when you need to change a few bytes at a time.

Inconveniently, Ghidra will not allow you to edit any bytes that are part of an existing instruction. The workaround for this limitation is to clear the associated instruction in the Listing window (by right-clicking Clear Code Bytes or pressing hotkey C). The Byte Viewer Options tool ❷ opens the dialog shown in Figure 22-8, which allows you to customize your Byte Viewer display.

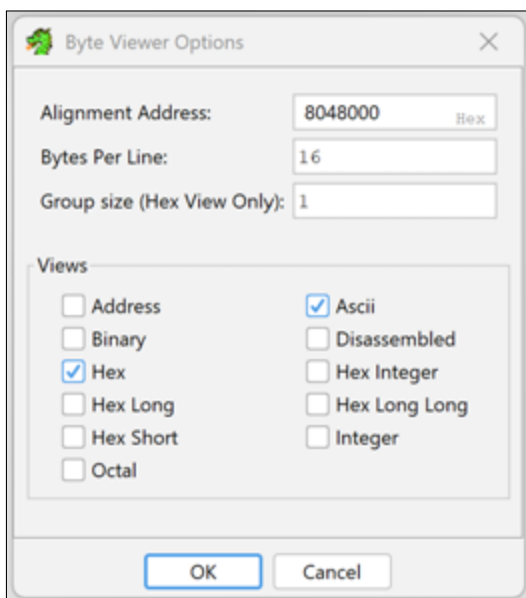


Figure 22-8: The Byte Viewer Options dialog

Selecting the Ascii option adds an ASCII dump to the Byte Viewer (Figure 22-9), which doubles as an ASCII editor while in Edit Mode.

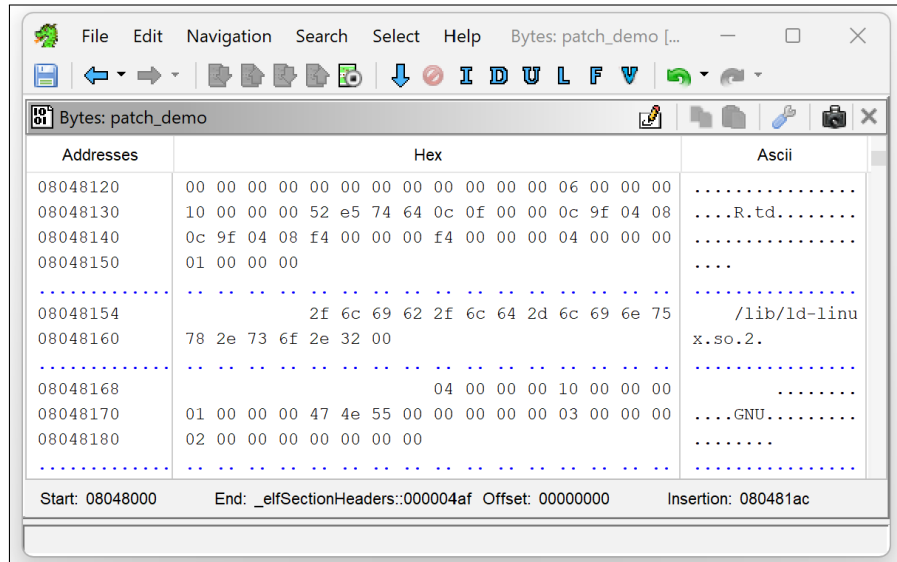


Figure 22-9: The Byte Viewer with ASCII dump enabled

Once you have finished entering your new values, you should toggle out of Edit Mode and return to the Listing window to verify that your changes are correct.

Scripting Your Changes

Unless your patch is very short, the most efficient means of modifying the original bytes in Ghidra is to have a script do it for you. The following function accepts a patch in the form of a byte array, as well as the start address of the patch, and then applies the patch in Ghidra:

```
public void patchBytes(Address start, byte[] patch) throws Exception {
    Address end = start.add(patch.length);
    ❶ clearListing(start, end);
    setBytes(start, patch);
}
```

You could include this function in a script that creates the array of patch bytes from a source of your choosing (for example, by declaring an initialized array or by loading the contents of a file). The `clearListing` call ❶ is necessary, as Ghidra will not allow you to modify bytes that are part of an existing instruction or data item. Once the script completes, you will need to manually format the patched bytes as either code or data and verify the correctness of your patch.

Using the Assembler

You may find yourself thinking about code patches in terms of replacing one assembly language instruction with another—for example, replacing `CALL _exit` with `NOP`. This manner of thinking is not necessarily incorrect, but tends to gloss over some of the complexities associated with patching code.

When the time comes to actually apply your patch to the program, you can't paste in your replacement assembly language statements; instead, you must paste in the corresponding machine code bytes, which means you'll probably want to use an assembler to generate machine code versions of all your replacement instructions.

One approach to this problem is to use an external editor to write your replacement assembly statements, assemble them with an external assembler (for example, `nasm` or `as`), extract the raw machine code, and finally patch them into the program, perhaps using a script. An alternative approach is to use Ghidra's built-in assembler capability, which you can access by right-clicking any instruction and selecting the Patch Instruction menu option.

NOTE

For `nasm`, the `-f bin` option emits raw machine code with no file headers. When using `as`, you will need a second utility, such as `objcopy`, to extract the raw bytes from the resulting object file.

Just as SLEIGH specifications tell Ghidra how to translate machine code into assembly language, they also enable Ghidra to perform assembly-to-machine-code translations—that is, to act like an assembler. The first time you choose the Patch Instruction option for a given architecture, Ghidra will build an assembler based on that architecture's SLEIGH specification. You will see a message similar to the one shown in Figure 22-10.



Figure 22-10: The Assembler Rating dialog

The Ghidra developers have run tests on the accuracy of Ghidra-generated assemblers. If a processor's assembler has been tested, it is assigned one of the following ratings, in decreasing order of accuracy: platinum, gold, silver, bronze, and poor. Any untested assemblers are marked *unrated*. You can find more information about Ghidra assembler ratings, along with the current rating for all available assemblers, in Ghidra Help.

Once you dismiss the Assembler Rating dialog, Ghidra will build the required assembler capability from the current processor's SLEIGH specification while displaying a dialog like the one shown in Figure 22-11.



Figure 22-11: The Assemble wait dialog

Once it has built your assembler, Ghidra replaces the selected instruction in the Listing window with two text input boxes (Figure 22-12) that allow you to edit the instruction's mnemonic and operands. The ESC key discards your changes before they are assembled, while the ENTER key assembles your new instruction and replaces the old instruction's machine code bytes with those of the new instruction.

004006ae	MOV	RAX,qword ptr [RBP + local_10]
004006b2	MOV	RDI,RAX
004006b5	CALL	0x004004d0
004006ba	MOV	EAX,0x0
004006bf	LEAVE	
004006c0	RET	

Figure 22-12: Assembling a new instruction

Because they derive from the same specification as the corresponding disassembler, Ghidra's assemblers recognize the same assembly syntax used in the Ghidra Listing window. Also, the assemblers are case sensitive and provide auto-completion options as you enter your new instructions. After you enter an instruction, Ghidra returns you to the normal Listing window view, and you can reselect Patch Instruction if there are additional instructions you want to modify. For short patches, Ghidra's assembler offers a convenient way to simultaneously assemble your instructions and modify the program.

Navigating Instruction Replacement Pitfalls

What happens if your new replacement instruction is shorter or longer than the old instruction? (These problems can arise on architectures without a fixed instruction size, such as x86.)

Consider the first case, in which your replacement instruction is shorter than the original instruction, as reflected in the following listing:

```

;BEFORE:
❶ 0804851b 83 45 f4 01  ADD    dword ptr [EBP + local_10],0x1
    0804851f 83 45 f0 01  ADD    dword ptr [EBP + local_14],0x1
;AFTER
❷ 0804851b 66 90          ❸ NOP
    0804851d f4          ❹ ??      F4h
    0804851e 01          ??      01h
    0804851f 83 45 f0 01  ADD    dword ptr [EBP + local_14],0x1
;FIXED:
    0804851b 66 90          NOP
❺ 0804851d 90          NOP
    0804851e 90          NOP
    0804851f 83 45 f0 01  ADD    dword ptr [EBP + local_14],0x1

```

In this case, we have replaced a 4-byte ADD instruction ❶ with a 2-byte NOP ❸. Ghidra’s assembler has done its best to fill the available space by inserting an x86 prefix byte (66) ❷ ahead of the x86 opcode for NOP (90).

Unfortunately, the replacement instruction is still too short to account for the two remaining bytes ❹ of the original instruction, one of which translates to a HLT (meaning you can use the hotkey D to disassemble it) and the other that Ghidra can’t disassemble, indicating that it does not represent a valid instruction. If you were to patch the original binary in this way and run it, it would almost certainly crash upon reaching this location.

Other than the ?? characters that appear in the listing, Ghidra provides no clues that a problem may exist, as your motivation for making this change and the “correct” solution depends on your particular use case.

If you are modifying instructions within the listing without the intention to export, you could use Ghidra’s fallthrough override option from the right-click context menu to bypass the unneeded bytes. (Fallthrough also has an additional option that allows you to bypass portions of a disassembly by identifying start and end addresses to bypass.) Alternatively, you can ask Ghidra to disassemble the undefined bytes, but it’s highly unlikely that they will disassemble into an instruction that you will find useful.

The most common solution in this situation is to replace all excess bytes from the original instruction with NOPs ❺ to pad to the start of the next instruction.

In situations where your replacement instruction is longer than the original instruction, you face a new set of challenges, as shown here:

```

;BEFORE:
❶ 08048502 6a 01          PUSH    0x1
❷ 08048504 ff 75 f0      PUSH    dword ptr [EBP + local_14]
    08048507 ff 75 08      PUSH    dword ptr [EBP + param_1]
    0804850a e8 51 fe ff ff CALL    read
;AFTER:
❸ 08048502 68 00 01 00 00 PUSH    0x100

```

```
08048507 ff 75 08      PUSH    dword ptr [EBP + param_1]
0804850a e8 51 fe ff ff CALL     read
```

In this case, the goal of the patch is to read 256 (0x100) bytes rather than 1 byte. We replace the original, 2-byte PUSH instruction ❶, which places the third argument to read (the length argument) onto the stack, with a 5-byte PUSH ❷ to push a larger constant. The additional bytes in the replacement instruction completely overwrite the instruction that was responsible for the second argument to read (the read buffer) ❸.

The resulting code not only fails to provide read with enough arguments, it also passes an integer where a pointer is expected. As with the previous example, this will almost certainly cause the patched program to crash. We discuss potential solutions to this particular patching problem in the next section.

Making Oversized Patches

The moment the size of your patch grows larger than the instructions or data that you are replacing, your life gets more complicated. In most cases this doesn't mean that your patch will be impossible to implement, but rather that your patch will require considerably more effort to implement. In this section, we discuss several approaches for handling this "patch is too large" problem, based on whether the patch contains code or data.

Oversized Code Patches

When your patch is too big to fit on top of the instructions you want to modify, you have no choice but to locate or create an unused region of sufficient size, patch your code into this empty region, and then insert a jump (known as a *hook*) at the original patch location to transfer control to your actual patch. In most cases, you will also need to append a jump to your replacement code to transfer control back to an appropriate location in the hooked function. Figure 22-13 shows the notional flow of a patched function with a jump hook installed.

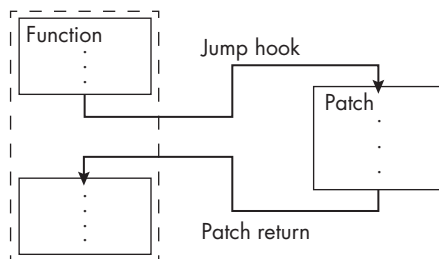


Figure 22-13: A function with an installed patch

The available unused space for your oversized code patch must be at least as large as your patch, reside at an address that will be executable at runtime, and be initialized from file content; otherwise, your patch won't get loaded at runtime.

The easiest place to begin your search for large unused executable blocks of bytes is by looking for any *code caves*: padding in executable sections, such as the .text section, used to adhere to section alignment requirements mandated by the executable file's format. Code caves are very common in Windows PE binaries, which frequently require every section of the binary to be a multiple of 512 bytes in size.

To find a code cave, start by looking at the end of the .text section. Navigate there by double-clicking the section name in the CodeBrowser's Program Trees window and then scrolling to the end of the Listing window. In our sample PE binary, the Listing window shows the following at the end of the .text section:

140012df8 ??	00h
140012df9 ??	00h
140012dfa ??	00h
140012dfb ??	00h
140012dfc ??	00h
140012dfd ??	00h
140012dfe ??	00h
140012dff ??	00h

The listing tells us that Ghidra has not classified the bytes (??) and that the bytes are initialized to 00h. Also, we see that the .text section ends at address 140012dff, which satisfies the file alignment requirement that the section be a multiple of 512 bytes in size.

Navigate to the previous instruction by scrolling up (or choosing the I tool in the CodeBrowser). You will arrive at the following:

140012cbd POP	RBP
140012cbe RET	
140012cbf ??	CCh
140012cc0 ??	00h

RET is the last meaningful instruction in this particular binary. So, we can compute the size of this code cave as $0x140012e00 - 0x140012cbf = 0x141$, or 321 bytes. This means we can easily patch as many as 321 bytes of new code into this binary. Assuming that we patched our new code in at address $0x140012cbf$, we would need to patch a jump to $0x140012cbf$ somewhere in the binary's existing code to ensure that the execution flow eventually reaches our patch.

When you cannot find a code cave, or the code cave is not large enough to hold your patch, you will need to get a little creative. Depending on the compiler options used to build the binary, you may be able to spread your patch across space gathered from inter-function alignment gaps.

Inter-function alignment gaps exist when compilers choose to align the start of every function to an address that is a multiple of 2 (often 16). When function alignment is being forced, there will be an average of $\text{align} / 2$ bytes and as many as $\text{align} - 1$ bytes of padding inserted between each function in

the binary. The following listing shows an optimal alignment gap for patching (align = 16) between two adjacent functions:

```
1400010a0 RET
❶ 1400010a1 ??    CCh
1400010a2 ??    CCh
1400010a3 ??    CCh
1400010a4 ??    CCh
1400010a5 ??    CCh
1400010a6 ??    CCh
1400010a7 ??    CCh
1400010a8 ??    CCh
1400010a9 ??    CCh
1400010aa ??    CCh
1400010ab ??    CCh
1400010ac ??    CCh
1400010ad ??    CCh
1400010ae ??    CCh
❷ 1400010af ??    CCh
*****
*                               *
*                               *
*****
```

You may safely overwrite all bytes from 1400010a1 ❶ through 1400010af ❷ with patch code.

Additional methods for squeezing patch code into a binary involve expanding existing program sections or injecting entirely new ones. Any technique that manipulates sections in such a manner also requires you to update the binary's section headers to make sure they remain consistent with the modifications you have made. Accordingly, these techniques are very file-format specific and require a detailed understanding of file header data structures.

Oversized Data Patches

In some respects, patching data is easier than patching code; in others, it is more difficult. For structured data types, you must make sure you use the correct size and byte ordering for each member of the structure, but because the size of a structure is determined at compile time, you don't need to worry about oversized replacement structures.

When patching string data, on the other hand, we recommend that you fit any replacement data entirely within the footprint of the original string. If your new string is larger than the original string, you may be fortunate to find a few bytes of padding between the end of the string and the next data item, but you must be careful not to corrupt any data that the program depends upon. If your data simply does not fit into the memory footprint of the original data, you will be forced to find a new location for it, but moving data properly can be difficult.

All global data items are referred to by their offsets from the program's code or data sections. To relocate a data item, you need to find sufficient unused space and locate every reference to the original data item, and then patch these references to refer to the new data item. Ghidra cross-references go a long way toward identifying every reference to a global, but will fail to identify derived pointers (those resulting from pointer arithmetic), which might require significant effort to correctly identify.

Once you have entered all of your patches in Ghidra and are happy with the resulting program listing, you'll want to push your changes into the original binary to verify that your patches behave as expected.

Exporting Patched Files

In this section, we discuss some of Ghidra's export features as they relate to patching. The File ► Export Program menu option offers the capability to export a program's information in any of several formats, as shown in Figure 22-14.

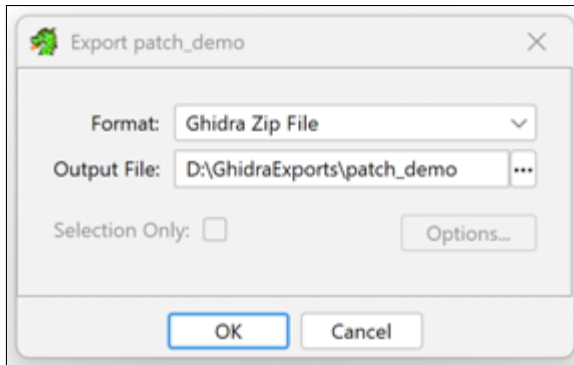


Figure 22-14: The Ghidra Export dialog

You can also access the Export dialog from the Project Manager by right-clicking the file you wish to export and selecting Export from the context menu. In the dialog, you are asked to specify the export format and the output file location, and to indicate whether you wish to limit the scope of your export to a range you have selected in the CodeBrowser. Some export formats offer additional options for more fine-grained control over the export process.

Ghidra Export Formats

Ghidra's exporter supports the following formats:

ASCII Can be used to save a textual representation of the program, similar to what is displayed in the Listing window, with options to choose which fields to include in the output file.

C/C++ Used to save the decompiler-generated source representation of the program, along with declarations for all types known to the Data Type Manager. This option is also available from the Decompiler window.

Ghidra Zip File A serialized Java object representation of your program suitable for import into other Ghidra instances.

HTML Generates an HTML representation of the program listing. Options similar to those available in the ASCII exporter format allow you to choose which fields to include in the output file. Labels and cross-references are represented as hyperlinks to provide a basic navigational capability throughout the generated output.

Intel Hex Defines an ASCII representation for binary data, commonly used for programming EPROM devices.

Original File Writes a file back to its original layout and is the most useful option for many patching applications.

Raw Bytes Creates a binary file with the raw bytes from a memory block concatenated.

SARIF SARIF (Static Analysis Results Interchange Format) is a standard for static analysis results.

XML Outputs the contents of the program in a structured XML format, with options to choose which program constructs to include in the generated file. This functionality is also available for individual functions in the Decompiler window to facilitate debugging function compilation. Although Ghidra includes a corresponding XML loader, this exporter includes the following warning: “Warning: XML is lossy and intended only for transferring data to external tools. GZF is the recommended format for saving and sharing program data.”

The Original Format Export

Ghidra’s Original File format exporter writes a program’s underlying binary content to a file. It concatenates all of the program’s initialized memory blocks (see Window ► Memory Map) to form the output file.

Whether the output file is identical to the original, imported file depends on the loader module used to import the file. If you used the Raw Binary loader, the Original File export option is guaranteed to re-create the original input file because it loads every byte of the original file into a single memory block. Other loaders may or may not load every file byte.

If you encounter a situation where the Original File export doesn’t meet your needs, it is always possible to script up a solution that works with any loader.

To wrap up this section, let’s look at an example in which we patch a binary and confirm that the patch runs as we expect it to.

Example: Evading Anti-Debugger Checks

Let's assume you are analyzing a piece of malware that checks for a debugger and exits if one is present, without allowing you to examine its behavior. The following source code outlines this functionality in a trivial program:

```
int is_debugger_present() {
    return ptrace(PTRACE_TRACEME, 0, 0, 0) == -1;
}

void do_work() {
    ❶ if (is_debugger_present()) {
        printf("No debugging allowed - exiting!\n\n");
        exit(-1);
    }
    // Do interesting things here.
    printf("Confirmed that there is no debugger, so do\n"
        "interesting things here that we don't want\n"
        "analysts to see!\n\n");
}

int main() {
    do_work();
    return 0;
}
```

The code checks for a debugger ❶ and exits if it finds one. Otherwise, it goes about its nefarious business. The following shows the output of the program running without a debugger:

```
# ./debug_check_x64
Confirmed that there is no debugger, so do
interesting things here that we don't want
analysts to see!
```

When the program runs under a debugger, we see a different response:

```
# gdb ./debug_check_x64
Reading symbols from ./debug_check_x64...(no debugging symbols found)...done.
(gdb) run
Starting program: /ghidrabook/CH22/debug_check_x64
No debugging allowed - exiting!
[Inferior 1 (process 434) exited with code 0377]
(gdb)
```

If we load the binary into Ghidra, we see the following debugger check in the Listing window:

```

        undefined do_work()
            undefined <UNASSIGNED> <RETURN>
001006f8  PUSH    RBP
001006f9  MOV     RBP,RSP
001006fc  MOV     EAX,0x0
00100701  CALL    is_debugger_present
00100706  TEST    EAX,EAX
00100708  JZ      LAB_00100720
0010070a  LEA     RDI,[s_No_debugging_allowed_-_exiting!_001007d8]
00100711  CALL    puts
00100716  MOV     EDI,0xffffffff
0010071b  CALL    exit
        -- Flow Override: CALL_RETURN (CALL_TERMINATOR)
        LAB_00100720
00100720  LEA     RDI,[s_Confirmed_that_there_is_no_debug_001008]
00100727  CALL    puts
0010072c  NOP
0010072d  POP     RBP
0010072e  RET

```

The Decompiler window provides the following corresponding code:

```

void do_work(void)
{
    int iVar1;

    iVar1 = is_debugger_present();
    if (iVar1 != 0) {
        puts("No debugging allowed - exiting!\n");
        /* WARNING: Subroutine does not return */
        exit(-1);
    }
    puts("Confirmed that there is no debugger, so do\n"
        "interesting things here that we don't want\n"
        "analysts to see!\n"
        );
    return;
}

```

To bypass this check with a patch, you could NOP the call to the `is_debugger_present` function, change the test condition, or change the contents of the `is_debugger_present` function. For example, if you use the Patch Instruction option from the right-click context menu, it is easy to replace the `JZ` with a `JNZ`, effectively flipping the condition to run only if it is being debugged, as shown in Figure 22-15.

001006f8	PUSH	RBP
001006f9	MOV	RBP, RSP
001006fc	MOV	EAX, 0x0
00100701	CALL	is_debugger_present
00100706	TEST	EAX, EAX
00100708	JNZ	0x00100720
0010070a	75 16	
00100711	0f 85 12 00 00 00	
00100716	48 0f 85 11 00 00 00	
0010071b	JNZ	

Figure 22-15: The Patch Instruction option after replacing JZ with JNZ

As a result, you should see the following code in the Decompiler window:

```
void do_work(void)
{
    int iVar1;

    iVar1 = is_debugger_present();
    if (iVar1 == 0) {
        puts("No debugging allowed - exiting!\n");
        /* WARNING: Subroutine does not return */
        exit(-1);
    }
    puts("Confirmed that there is no debugger, so do\n"
        "interesting things here that we don't want\n"
        "analysts to see!\n"
        );
    return;
}
```

If we then export the file as a binary using the export script and run it again, we see the following two listings, which demonstrate the behavior we were hoping to accomplish with our patch:

```
# ./debug_check_x64.patched
No debugging allowed - exiting!
# gdb ./debug_check_x64.patched
Reading symbols from ./debug_check_x64.patched...(no debugging symbols found)...done.
(gdb) run
Starting program: /ghidrabook/CH22/debug_check_x64.patched
Confirmed that there is no debugger, so do
interesting things here that we don't want
analysts to see!
[Inferior 1 (process 445) exited normally]
(gdb)
```

While there are many external tools (for example, `VBinDiff`) available to confirm that only 1 byte was changed within the file for this example, you can also use Ghidra's internal tools to reach the same conclusion. The next chapter focuses on methods to accomplish this goal.

Summary

Regardless of your particular motivation for patching a binary, your patch will require careful planning and deployment. Ghidra provides the tools you need to approach patching methodically: You can design the change; implement it through hex editing, the built-in assembler, or scripting; and immediately see how the modification affects the program. The next chapter demonstrates how you can use Ghidra to compare the unpatched and patched versions of your binaries and explores Ghidra's capabilities for more advanced binary differencing, including version tracking and the use of BSim for behavioral similarity analysis.

23

BSIM AND OTHER COMPARISON TOOLS



We have spent the previous chapters introducing you to the many ways Ghidra can assist your reverse engineering analysis efforts. Throughout this journey, we have discussed many ways of transforming and annotating your work to document and facilitate understanding of the binary.

In this chapter, we introduce Ghidra's suite of comparison tools, moving from basic structural differencing to advanced behavioral analysis. We begin with traditional binary differencing, which highlights similarities and differences across entire programs. From there, we narrow the scope to function-level comparisons, where you can directly contrast specific routines such as cryptographic implementations.

Building on these approaches, we explore Ghidra's BSim tool, which identifies behavioral similarities between functions even when the code is not identical. Finally, we look at version tracking, which integrates these comparison methods into a workflow for reusing and correlating analysis results across multiple program versions.

Comparing Files with the Program Diff Tool

In the preceding chapter, we patched a binary to modify the flow of a function, bypassing a call to exit by changing a single byte in a single instruction: JZ (74) to JNZ (75). To confirm the change and document exactly what was changed, we could use an external tool such as VBinDiff to compare the two files at the byte level. However, to compare files at the instruction level, we need a much more sophisticated tool: the Program Diff tool available in Ghidra's Listing window. Once this tool has computed the differences, we can view them using custom displays designed to emphasize the discrepancies, facilitate our understanding of each change, and provide opportunities to take action based on the type of difference.

Let's investigate how we can use Ghidra's Program Diff to analyze our patched binary and pinpoint the modified instruction in context.

Example: Spotting a Patched Instruction

To compare two files that have been imported to a project and are in the same state (for example, both analyzed or neither analyzed), open one of the files in the CodeBrowser and then choose Tools ► View Program Differences and select another file within the current project for comparison. Alternatively, you can use the Listing window tool icon shown in Figure 23-1. This icon serves as a toggle to open or close the Program Diff tool.

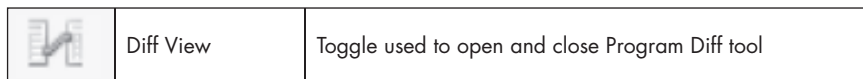


Figure 23-1: The CodeBrowser Diff View toggle icon

For example purposes, let's start by opening the unpatched version of the file from the preceding chapter, selecting the Diff View icon, and choosing the patched version of the file. This opens the Determine Program Differences dialog, shown in Figure 23-2.

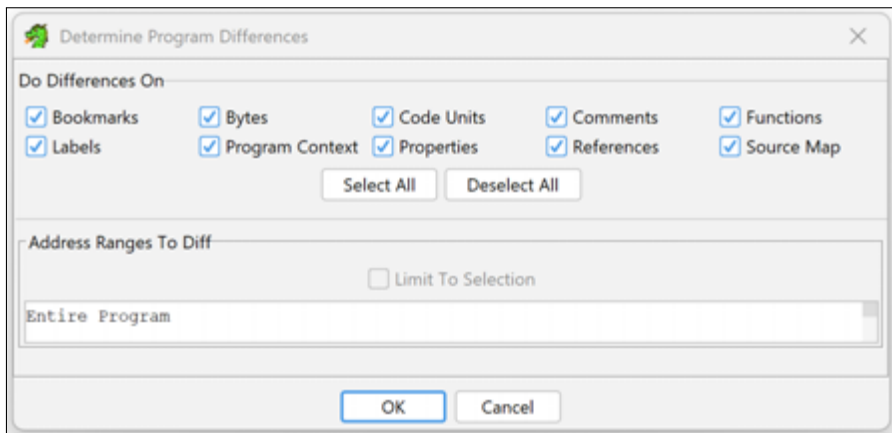


Figure 23-2: The Program Diff tool's Determine Program Differences dialog

While all available differences fields will be selected by default, in this case we need only Bytes to confirm that the patch worked correctly, so it is the only difference option we will select. When you click OK, you can see the two binaries in the Listing window, which has been split into side-by-side listings, with one file displayed in each. By default, the listings are synchronized, so navigating in one also navigates in the other. There are several ways to navigate the detected differences within this tool, which we present later in the chapter.

When you open two files for diffing, Ghidra initially positions the Listing view at the beginning of each file. You may use the down arrow tool on the Listing window toolbar (or CTRL-ALT-N) to navigate to the first difference between the two files. To call your attention to the differing code, changes appear as color-highlighted disassembly lines in each file's Listing window, as well as in the Decompiler window if the difference is within a function. (The Decompiler window is synchronized with the first of the two files.) Navigating to the first detected difference reveals the single byte that is 32 (74) in the original listing and JNZ (75) in the second listing.

To view more details, choose Window ► Diff ► Diff Details. You should see the following report in the Diff Details window at the bottom of the CodeBrowser window:

```
Diff address range 1 of 1.
Difference details for address range: [ 00100708 - 00100709 ]
Byte Diffs :
    Address  Program1  Program2
    00100708    0x74      0x75
Code Unit Diffs :
    Program1 CH23:/DiffDemo/debug_check_x64 :
            00100708 - 00100709    JZ 0x00100720
                                Instruction Prototype hash = 16af243b
    Program2 CH23:/DiffDemo/debug_check_x64.patched :
            00100708 - 00100709    JNZ 0x00100720
                                Instruction Prototype hash = 176d4e0c
```

The first line indicates the number of address ranges that contain differences. In this case, only one range in the file contains differences, so you can be confident that the two programs differ by only a single byte. This simple example barely scratches the surface of the capabilities associated with Ghidra's Program Diff tool, so let's take some time to investigate other capabilities that this tool provides.

While this simple case illustrates how Program Diff can highlight an instruction-level change, the tool offers much more than a basic side-by-side comparison. Beyond flagging differences, it provides a range of options to control what is compared, how results are displayed, and how changes can be applied or merged into an analysis. Exploring these features reveals the full power of Program Diff and shows how it can support more complex workflows.

Program Diff Options

The 10 options at the top of Figure 23-2 form the basis of your comparison, and you can select any or all of them. By default, the Program Diff tool operates on the entire program for each file. If you wish to limit your comparison to a specific address range, you must highlight the range in the first file before opening the tool. Once you have made your selections and clicked OK, the split Listing window you see is called a Program Diff.

Program Diff View

Program Diff View lets you view both files simultaneously. Basically, the Listing window now has two listings, one on the left side and one on the right. When you open the Diff Details window, it opens at the bottom of the CodeBrowser window. Within the Diff Details window, the left file is

considered Program1 (the file you opened originally), while the right file is considered Program2 (the file you chose to compare to Program1). The Decompiler window reflects the contents of Program1. When you are comparing two files, Ghidra can compute the differences in either direction. It is up to you to remember which file is which as you use the Program Diff tool.

A common workflow is to begin analyzing a file and then realize that some or all of the code looks familiar, which may prompt you to open a file you have previously analyzed in order to begin diffing. Fortunately, Program Diff maintains alignment between the two files by inserting blank lines when necessary. Differences are highlighted, and the Program Diff toolbar (Figure 23-3) provides you with a navigational capability and the means to determine how you wish to handle the differences.

	 		 	 
	Apply Differences	Applies the selected settings and remains in the same location		
	Apply Differences and Move	Applies the selected settings and moves you to the next highlighted difference		
	Ignore Differences and Move	Ignores the selected settings and moves you to the next highlighted difference		
	Show Details	Opens the Diff Details window and provides information about the selected difference		
	Go to Next	Moves you to the next highlighted difference		
	Go to Previous	Moves you to the previous highlighted difference		
	Display Diff Apply Settings	Opens the Diff Apply Settings window and allows you to modify the settings		
	Determine Program Differences	Reopens the Determine Program Differences dialog to allow you to change selection fields and the range		

Figure 23-3: The Program Diff toolbar options

These additional tools extend the Listing window toolbar options.

The Diff Apply Settings

The Diff Apply Settings define the actions that you would like to take when there is a difference between the two files. Choosing the Display Diff Apply Settings option displays the window shown in Figure 23-4.

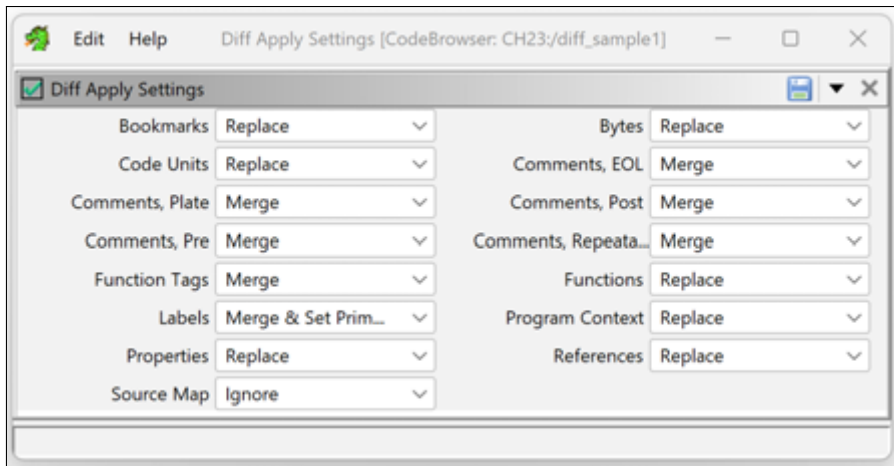


Figure 23-4: The Diff Apply Settings window

Each setting specifies a default action you would like to apply from the second program to the first program you opened and how you would like that option applied. The following four options are available from each drop-down:

Ignore Do not change the first program (available for all cases).

Replace Change the first program's content to match the second program's content (available for all cases).

Merge Add the difference from the second program to the first program. If applied to a label, this will not change which label is set to primary (available for only Comments, Labels, and Function Tags).

Merge & Set Primary The same as Merge, but the primary label is set to the second program label, if possible (available for Labels only).

At the top of Figure 23-4 are two toolbar icons. The Save as Default icon saves the current Diff Apply Settings. The arrow opens a menu that allows you to make changes to all of the settings at one time by choosing one of the options shown in Figure 23-5.

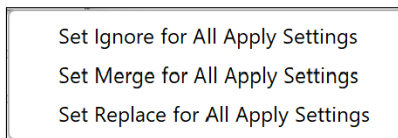


Figure 23-5: The Diff Apply Settings pull-down menu

If you select the Set Merge option and Merge is not a valid choice for a particular setting, it will be changed to Set Replace. For labels, it will be changed to Merge & Set Primary.

Choose Apply Differences from the toolbar if you wish to apply all of the default changes. When you have finished with the Program Diff tool, toggling the Listing window Diff View icon in the Listing window (see Figure 23-1) or closing the window will display the dialog displayed in Figure 23-6.

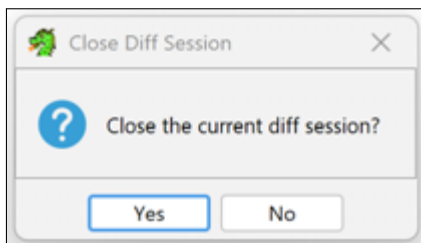


Figure 23-6: The Close Diff Session confirmation dialog

Confirming that you wish to close the current diff session will close the display of the second file and return you to a normal Listing window with the first file (and all of the changes that you have selected from the diff analysis) displayed.

The Program Diff tool is especially useful in situations where analysts do not share a Ghidra Server instance and need to compare their independently analyzed files. It is also valuable when examining different versions of the same code base, such as an unpatched and a patched release of a shared library. In the following example, we will walk through the process of using Program Diff to reconcile two copies of the same binary, each of which was analyzed independently.

Example: Merging Two Analyzed Files

Assume that you are analyzing a binary that contains a crypto routine. A colleague mentions that she is midway through analyzing a binary that also appears to have a crypto routine and is likely from the same malware family. She agrees to provide you with her project so that you can compare the two files. When you look at the files in Diff View, you immediately notice that the two of you appear to be analyzing the same binary.

The challenge is that you have each made progress and have modified the contents of the file based on your individual analysis. You need to merge the two analyzed files so that you can each benefit from the other's analysis. Having agreed to take on this responsibility, you have opened your binary in the CodeBrowser and initiated a Program Diff session, and then added your colleague's binary for comparison.

Choosing the down arrow on the Program Diff toolbar takes you to the first difference in this file. At this point, you can open the Diff Details window by choosing the option from the Program Diff toolbar (or hotkey F5).

This window provides you with the following listing (broken into two sections to facilitate discussion). In the first section, at the top of the Diff Details, you see the following:

```

❶ Diff address range 1 of 4.
❷ Difference details for address: 0010075a
❸ Function Diffs :
    Program1 CH23:/diff_KN :
    ❹ Signature: void encrypt_rot13(char * inbuffer, char * outbuffer)
        Thunk? : no
    ❺ Return Value :
        DataType      Storage      FirstUse  Name      Size Source
        /void          <VOID>        0x0      <RETURN>   0 USER_DEFINED
    Stack Frame:
    ❻ Parameters:
        DataType      Storage      FirstUse  Name      Size Source
        /char *        RDI:8        0x0      inbuffer   8 USER_DEFINED
        /char *        RSI:8        0x0      outbuffer  8 USER_DEFINED
    ❼ Local Variables:
        DataType      Storage      FirstUse  Name      Size Source
        /int           EAX:4        0xc0     length     4 USER_DEFINED
        /int           Stack[-0x1c]:4  0x0     loop_ctr   4 USER_DEFINED
        /char          Stack[-0x1d]:1  0x0     curr_char  1 USER_DEFINED

❽ Program2 CH23:/diff_CS :
    Signature: undefined encrypt(char * param_1, long param_2)
    Thunk? : no
    Return Value :
        DataType      Storage      FirstUse  Name      Size Source
        /undefined     <UNASSIGNED>    0x0      <RETURN>   1 DEFAULT
    Stack Frame:
    Parameters:
        DataType      Storage      FirstUse  Name      Size Source
        /char *        RDI:8        0x0      param_1    8 DEFAULT
        /long          RSI:8        0x0      param_2    8 DEFAULT
    Local Variables:
        DataType      Storage      FirstUse  Name      Size Source
        /undefined4     Stack[-0x1c]:4  0x0     local_1c   4 DEFAULT
        /undefined1     Stack[-0x1d]:1  0x0     local_1d   1 DEFAULT

```

The first of four identified difference address ranges ❶ in this file is being displayed and is associated with the current address, 0010075a ❷. The listing begins by detailing a difference in the function headers of the two binaries ❸. In your version of the binary, you have provided a meaningful name for the function and the parameters in the function signature ❹ as well as the function return ❺. Further, each parameter has an appropriately defined type ❻, and you have given the local variables meaningful names and

types ⑦. In the second program ⑧, the analyst did not make any changes to the default Ghidra header for the corresponding function.

You want to retain your version of the changes for the function definition and local variables. You could use the toolbar icon to reject the change, but this would reject all of the differences associated with this address. Because you have not yet reviewed all of the differences, scroll down to the next difference in the Diff Details window.

The next section of the differences associated with the first address range contains the label and comment differences:

① Label Diffs :

Program1 CH23:/diff_KN at 0010075a :
0010075a is an External Entry Point.

Name	Type	Primary	Source	Namespace
② encrypt_rot13	Function	yes	IMPORTED	Global

Program2 CH23:/diff_CS at 0010075a :
0010075a is an External Entry Point.

Name	Type	Primary	Source	Namespace
③ encrypt	Function	yes	USER_DEFINED	Global

④ Plate-Comment Diffs :

⑤ Program1 CH23:/diff_KN at 0010075a :

```
*****
*                               Function                               *
* This is a crypto function originally named cryptor. Renamed *
* to use out standard format encrypt_rot13. Changed the      *
* function parameters to char *. Added meaningful variable   *
* names. Function first seen in fileC13d by Ken H.           *
*****
```

Program2 CH23:/diff_CS at 0010075a :
No Plate-Comment.

⑥ EOL-Comment Diffs :

Program1 CH23:/diff_KN at 0010075a :
No EOL-Comment.

⑦ Program2 CH23:/diff_CS at 0010075a :

This looks like a crypto routine. TODO Analyze to get more information.

Within the label differences ①, the only difference is the name of the function ② ③, which we have already discussed. In the Plate-Comment section ④, your file has a detailed comment ⑤, but the other file has no plate comments. In the EOL-Comment section ⑥, there is a brief comment by the other analyst ⑦, but none in your file. When you examine the comment, you see that it is a TODO action item that you have already done in your file.

After evaluating all of the differences at this address, you determine that you have marked up this function appropriately and the other analyst has not provided any additional information. Choosing the Ignore Differences and Move icon retains your content and doesn't accept any new content from the other binary. This takes you to the next difference. Since you already have

the Diff Details window open, its contents will update as soon as you navigate, displaying the following:

-
- ❶ Diff address range 1 of 3.
Difference details for address range: [0010081a - 0010081e]
Reference Diffs :
 Program1 CH23:/diff_KN at 0010081a :
 Reference Type: WRITE From: 0010081a Mnemonic To: register:
 RAX USER_DEFINED Primary
 Program2 CH23:/diff_CS at 0010081a :
 No unmatched references.
-

You have decreased the number of ranges containing differences by rejecting the previous difference ❶. Once again, your file has more information than the second file. This time, you will navigate to the next difference by clicking the down arrow, taking you to the following:

-
- ❶ Diff address range 2 of 3.
Difference details for address: 00100830
Function Diffs :
 Program1 CH23:diff_KN :
 Signature: undefined display_message()
 Thunk? : no
 Calling Convention: unknown
 Return Value :

DataType	Storage	FirstUse Name	Size	Source
/undefined	<UNASSIGNED>	0x0 <RETURN>	1	DEFAULT

 Parameters:
 No parameters.
 Program2 CH23:diff_CS :
❷ Signature: void display_message(char * message)
 Thunk? : no
 Calling Convention: __stdcall
 Return Value :

DataType	Storage	FirstUse Name	Size	Source
/void	<VOID>	0x0 <RETURN>	0	USER_DEFINED

 Parameters:

DataType	Storage	FirstUse Name	Size	Source
/char *	RDI:8	0x0 message	8	USER_DEFINED
-

Notice that the number of ranges has not changed ❶; you have been moved to the next difference range without impacting the total number of difference ranges. Evaluating this new difference shows that the second file contains information provided by the other analyst that is not available in your file: She gave the function signature a return type and added a parameter to the function signature ❷. You can include this information in your binary by right-clicking the difference in the right-hand Listing window and choosing Apply Selection (hotkey F3), or by clicking the Apply Differences icon on the toolbar.

Navigating to the next difference, you see the following details:

- ❶ Diff address range 2 of 2.
Difference details for address range: [00100848 - 0010084c]
Pre-Comment Diffs :
 Program1 CH23:/diff_KN at 00100848 :
 No Pre-Comment.
 Program2 CH23:/diff_CS at 00100848 :
❷ This is a potential vulnerability. The parameter is being passed
 in to printf as the first/only parameter which may result in a format
 string vulnerability.
-

The number of difference ranges ❶ has decreased because you applied the differences in the previous range. In this final difference, you see an interesting entry in the Pre-Comment section ❷ in the other file. The analyst has detected a potential vulnerability. To be sure this information is included in your file, you choose Apply Differences.

Now that you have completed the comparison of the two files, you can close the Diff View window and confirm that you want to end the current Program Diff session. Your Listing view now reflects the combined analysis from both binaries, and you can save and close your file.

The Ghidra Program Diff tool provides the ability to investigate the differences between two versions of the same file. While it will attempt to diff two unrelated files, any results are likely to reflect only coincidental similarities. Let's shift our focus to a different tool that facilitates comparisons between selected functions within the same or different programs.

THE OTHER FUNCTION IS INSIDE THE FILE!

Program Diff is built to compare one file against another, making it the right choice when you need to analyze how two files differ. However, Function Comparison Provider (FCP) offers a different kind of revelation. FCP allows you to compare functions that exist not only across separate binaries but also within the same binary.

This capability can feel a little unsettling at first, like realizing the source of a problem is closer than you expected. Imagine tracing an if statement that branches into two nearly identical functions in the same program. On the surface, they seem interchangeable, but FCP can place them side by side, and reveal subtle differences in logic, flow, or data handling. What looks identical at a glance often hides key distinctions, and FCP ensures that those hidden differences do not stay concealed for long.

Comparing Functions with the Function Comparison Provider

When we examined the binaries using Program Diff, we identified a patched instruction by comparing two versions of a binary at the program level. That approach worked well, but sometimes you already know the specific function

where a change has occurred and want to narrow your focus. Other times, you may come across a function that looks familiar because it resembles one you have already analyzed in the past. In either case, it can be useful to compare functions directly so that the results of an earlier analysis can be applied to the current case when appropriate.

Ghidra provides this capability through its Function Comparison Provider (FCP), which allows you to view two functions side by side. By highlighting differences in instructions, control flow, and references, FCP makes it easier to confirm what has changed and where those changes matter.

Let's investigate how we can use FCP to revisit the patched binary and confirm the modified instruction within the affected function.

Example: Revisiting Patched Functions

To compare the patched and unpatched versions of our function, we need to make sure both binaries are open in the CodeBrowser. We will load the unpatched version into the FCP by highlighting the function in the active CodeBrowser tab, and then right-click and select **Function ► Compare Function(s)**. The Function Comparison window opens with the selected function displayed in both panels.

To choose a function to compare with the current function, you need to make the binary containing the new function active in the CodeBrowser. At that point, add the patched function to the comparison by choosing the **Add Functions** icon and selecting the patched version from the list of functions in the active program. This will load the function into the right panel of the comparison window, allowing the two functions to be viewed side by side as shown in Figure 23-7.

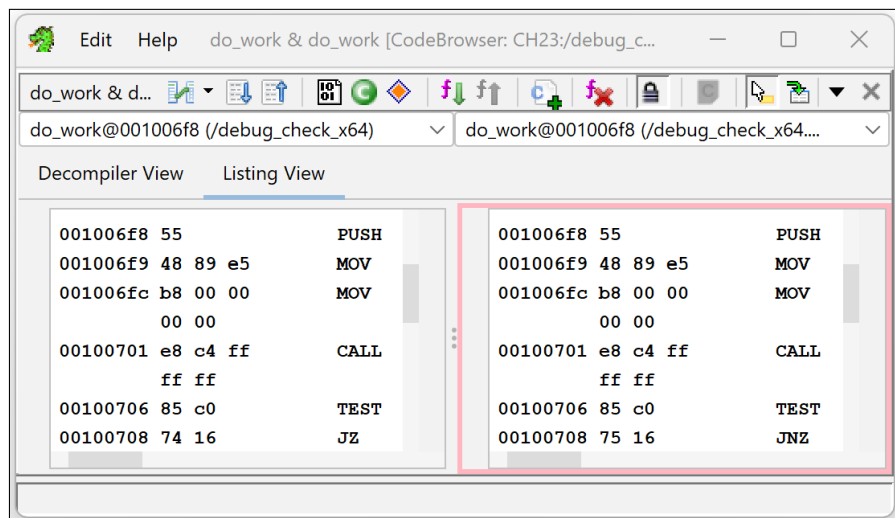


Figure 23-7: The Listing view in the Function Comparison window

In this example, the patched instruction appears on the right, while the unpatched version is visible on the left. Figure 23-7 shows that the two functions are identical except for the final instruction, confirming that the patch was successfully saved.

The Function Comparison window allows more than two functions to be loaded, so you could add additional candidates from either binary if needed. A pull-down menu above each panel makes it easy to switch the function displayed in that window. You can also toggle between Listing view and Decompiler view, as shown in Figure 23-8, to see the patched instruction highlighted in decompiled form.

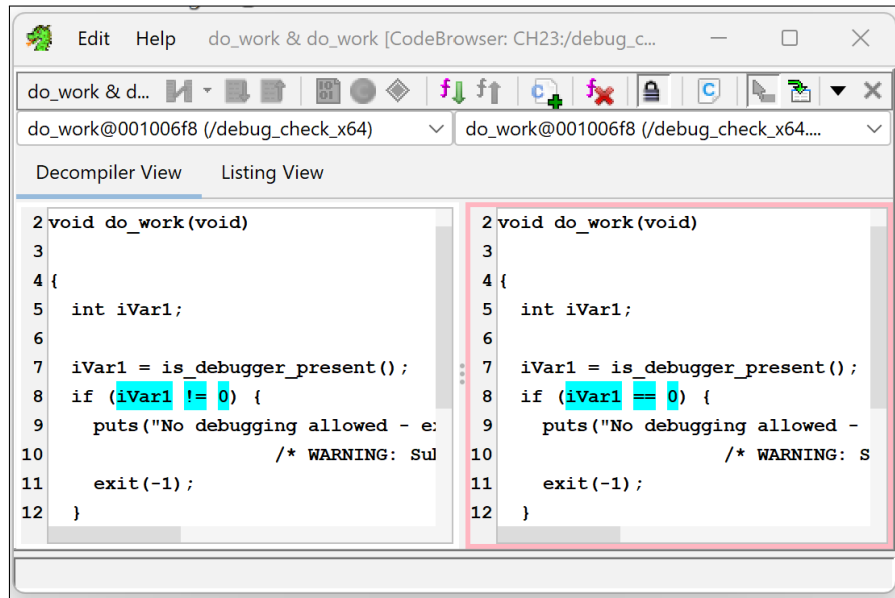


Figure 23-8: The Decompiler view in the Function Comparison window

While this simple case shows how the FCP can confirm a change within a single function, the tool includes a variety of other features that extend its usefulness far beyond basic comparisons. These options control how functions are aligned, how differences are displayed, and how results can be interpreted. Understanding these capabilities is important because they allow you to apply FCP effectively in more complex situations where functions are larger, more varied, or only loosely related.

The Function Comparison Toolbar

The exploratory capabilities available in this window overlap significantly with the Program Diff tool, except you are comparing only two functions at a time and you can easily switch between the decompiled code and the listing. Figure 23-9 shows the toolbar menu for this window.

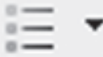
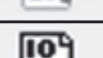
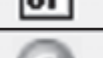
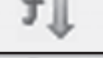
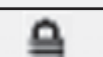
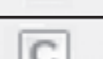
		
	Marker Selection	Toggles between All Area Marker, Unmatched Area Marker, and Area Markers
	Diff View	Toggle used to open and close Program Diff View
	Go to Next	Go to next unmatched area
	Go to Previous	Go to previous unmatched area
	Byte Differences	If toggled on, do not highlight byte differences
	Constant Differences	If toggled on, do not highlight operand constants
	Register Differences	If toggled on, do not highlight operand registers
	Next Function	Go to the next function for the side that is in focus
	Previous Function	Go to the previous function for the side that is in focus
	Add Functions	Add new function to the comparison
	Remove Function	Close the current function for the side that is in focus
	Scroll Lock	Toggles between lock and unlock for synchronized scrolling
	Constants	Toggle to determine if constants must match exactly in Decompiler Diff View
	Mouse Hover	If toggled on, show information when hovering over an item
	Synchronized Navigation	When toggled on, navigate the other panel to the same function when a new function is selected
	Listing Options	Allows you to set listing options such as listing headings

Figure 23-9: The Function Comparison toolbar options

Let's walk through an example that demonstrates some additional FCP capabilities.

Example: Comparing Crypto Routines

Congratulations on your promotion! Based on your successful analysis and use of the Program Diff tool for crypto routines, you have now been labeled the crypto expert in your shop. Every time one of your colleagues suspects that they have a crypto routine, they send you the binary to see if it is a crypto routine you recognize.

You now have a new file from a colleague and wish to determine whether the crypto routine used in this file is something new, or whether it is a routine you have identified in the past. Rather than loading and comparing each crypto routine against the new function, you have set up a special Ghidra project that contains all of your previously analyzed and documented crypto routines. Your goal is to load your crypto routines on one side of the Function Comparison window and then import the new file on the other side to compare against the existing crypto routines. (To simplify this example, you currently have only one analyzed crypto routine in your collection: the ROT13 routine that you merged in the previous example.)

After you load your complete collection of analyzed crypto files into the CodeBrowser and load your function, `encrypt_rot13`, in the Function Comparison window, you need to load the new file into the same CodeBrowser instance (File ► Open) and make it the active file. At this point, you can explore the file, but it isn't necessary to do so; by choosing the Add Functions option from the Function Comparison toolbar, you can see the complete list of functions in the new binary, and you can always switch back to the CodeBrowser window if you can't find the function you need. In this case, halfway down the list is a function with a name that intrigues you, `encrypt`, as shown in Figure 23-10.

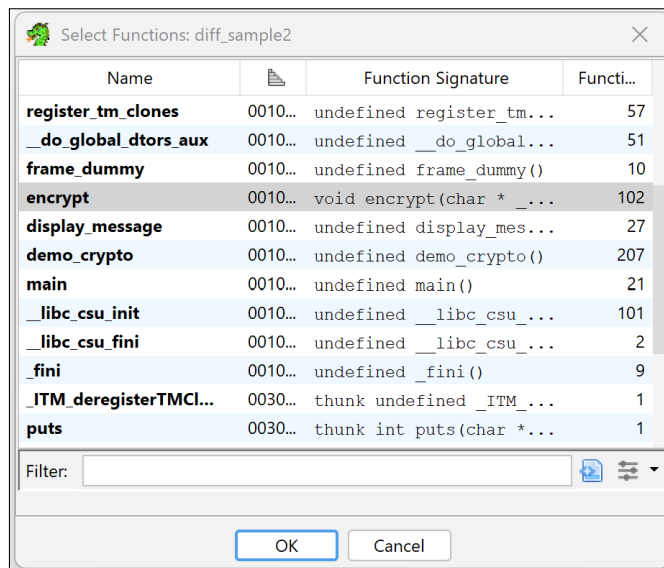


Figure 23-10: The Select Functions window with the `encrypt` function selected

A cursory glance at the loaded files in the Decompiler view of the functions, shown in Figure 23-11, suggests that these two functions are quite different.

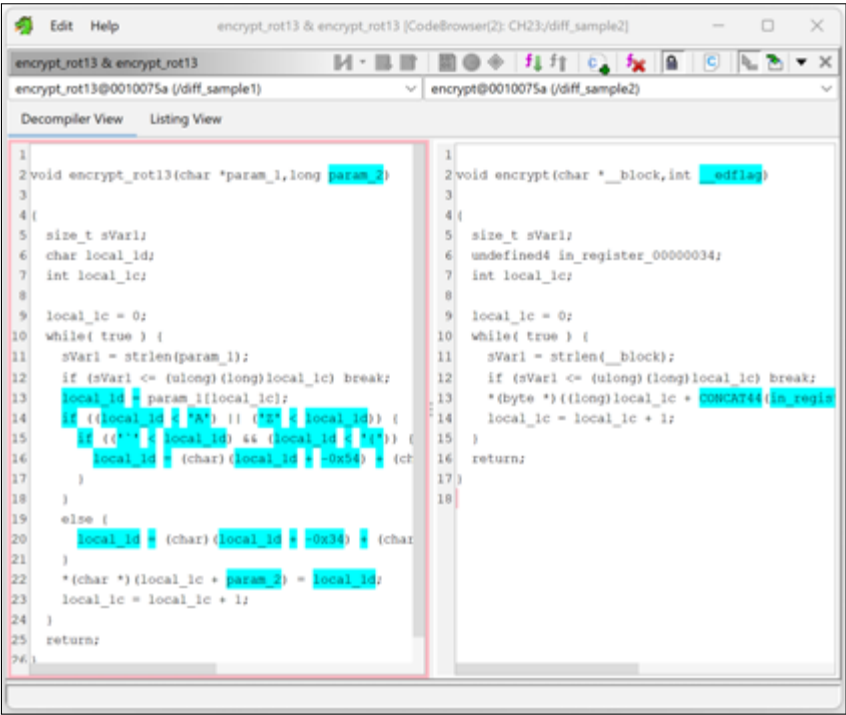


Figure 23-11: The Decompiler view of the Function Comparison window for two crypto routines

The Listing view, shown in Figure 23-12, confirms that these two functions have significant differences.

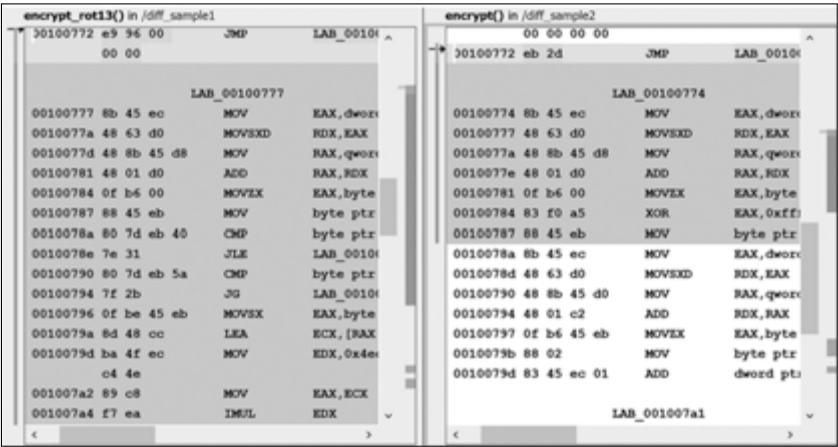


Figure 23-12: The Listing view of the Function Comparison window for two crypto routines, with differences highlighted

On further analysis, you discover that the new routine XORs each byte with the constant value 0xa5. This is definitely different from your current crypto routine, so you name this new function and add it to your collection (which now has two members!). Returning to the CodeBrowser, you update the function signature and add comments to document the new crypto routine. The changes you make will be reflected in the Function Comparison window as well.

As you are documenting, you notice the new binary has a function called `display_message`, as does the binary you are comparing it against. You recall that this function was identified as having a vulnerability in your current binary, so you decide to compare these two functions. You load them into the Function Comparison window to see if they have similarities beyond the common name. They seem different, as shown in the Decompiler view in Figure 23-13.

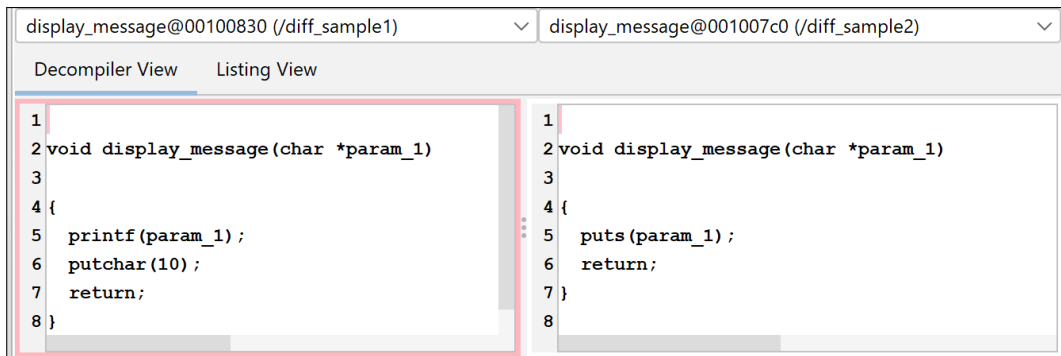


Figure 23-13: The Decompiler and Listing views for `display_message` functions

In the second example, `param_1` is being passed to `puts` for output, which fixes the vulnerability.

Now that you have documented this crypto routine, you see that you have received yet another binary from your colleagues. To reset to the start of your crypto comparison process, you can use the Function Comparison toolbar icon to remove the `display_message` functions from the window, leaving you with your crypto routine collection, which now has two distinct members: `encrypt_rot13` and `encrypt_XOR_a5`.

An initial exploration of this new file indicates that three functions seem to involve encryption: `encrypt`, `encrypt_strong`, and `encrypt_super_strong`. You load these into the Function Comparison window so you can compare them to your existing crypto routines.

After comparing `encrypt_rot13` against each of the new functions, you notice the following: The instructions in `encrypt_rot13` and `encrypt_strong` are identical. The differences primarily consist of address labels, as shown in Figure 23-14.

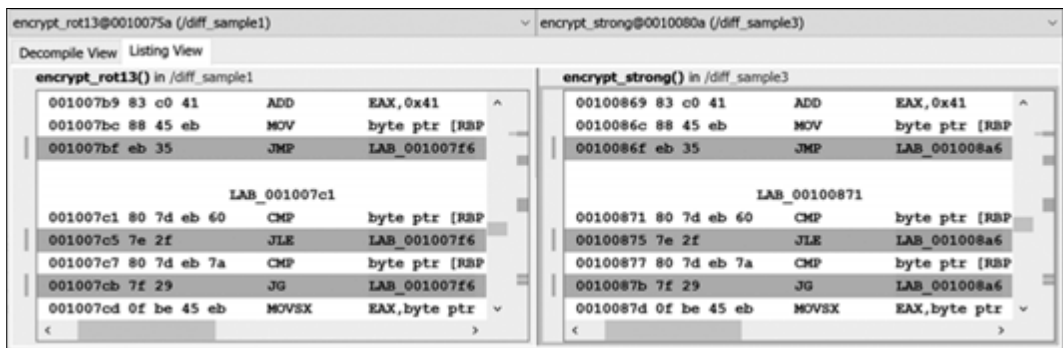


Figure 23-14: The Function Comparison window with address label differences

You would not expect address labels to match perfectly in this case, as the locations of the functions within the binaries are different. The locations are consistent relative to the current address, so we are likely dealing with the same function. The only other difference is 1 byte associated with the call to `strlen`, as shown in Figure 23-15. This is a similar issue and can be explained by the difference in the relative positions of the encryption function and `strlen` in each binary.

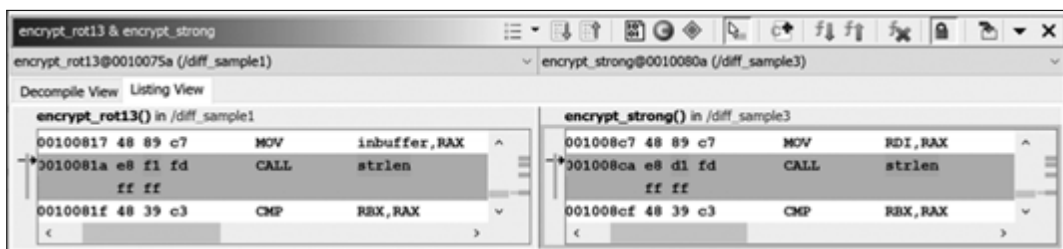


Figure 23-15: The Function Comparison window with a byte difference in the call to `strlen`

After determining that these are the same function, you can right-click the new function and select **Apply from Other ► Function Signature and Data Types**. This will update the function signature in all needed locations, including the Listing window and Symbol Tree. Note that the Function Comparison window does not provide all of the capabilities available in the Diff View. To copy additional information (such as the detailed comments associated with the function), use the Program Diff tool.

Having completed your comparative analysis of `encrypt_rot13`, you turn your attention to `encrypt_XOR_a5` and observe the following relationships with each of the new functions:

encrypt_XOR_a5 vs. encrypt Almost entirely different.

encrypt_XOR_a5 vs. encrypt_strong Very different. A closer look at the differences between these two functions also leads you to believe that they are not the same function.

encrypt_XOR_a5 vs. encrypt-super_strong Almost entirely the same.

The identified differences between `encrypt_XOR_a5` and `encrypt_super_strong` are also just address labels and some bytes in the call to `strlen`. You can handle this situation the same way you did the previous matching functions.

While this is a trivial example (and probably not consistent with the actual crypto routines you might see in the wild), it does demonstrate how to use the FCP to minimize the duplication of analysis effort when you encounter familiar routines in new binaries.

Traditional function comparison works well when routines are similar or even identical, but functions often change in ways that make exact matching difficult. These changes may be intentional, such as efforts to conceal behavior, or unintentional, resulting from compiler differences, optimization choices, or code updates. In such cases, relying only on side-by-side comparison is not enough. Instead, it is more effective to look for similarities in how functions behave. Ghidra's BSim tool is designed for this task. By analyzing function behavior rather than just the instructions, BSim helps you uncover related routines and continue your analysis even when the code has shifted.

FOLLOWING THE CLUES: BEHAVIOR REVEALS THE HIDDEN MATCH

When two crimes are committed by the same person, investigators often look for a modus operandi, or MO, rather than focusing on surface details. The tools used or the exact sequence of steps may vary, but the underlying pattern of behavior remains consistent.

BSim works the same way for code. Traditional function comparison depends on structural similarity, which can be broken by compiler differences, optimizations, or deliberate obfuscation. However, BSim looks at the behavioral "signature" of a function. It compares how routines operate, not just how they are written.

By focusing on the MO of a function, BSim exposes similarities that might otherwise remain hidden. Even when the code has been changed to mask its appearance, the underlying behavior often gives it away.

Comparing Behaviors with BSim

Imagine that you are reverse engineering a piece of malware you have seen before, but this time, its creator has updated it to avoid detection. You have already spent hours analyzing a function in the original version that decrypted strings before they were used. In the new version, the author has added junk instructions, reordered some logic, or used a different compiler so the function looks different in a standard diff tool.

A simple byte-for-byte or instruction-for-instruction comparison might not find this modified decrypt routine because the code layout changed just enough to break the match. However, the function's behavior, decrypting strings in memory, is still there.

This is exactly the kind of situation where BSim helps. Short for *behavioral similarity*, BSim analyzes the structure and flow of a function to estimate how it behaves instead of relying on the raw bytes. This behavior makes it especially useful for detecting code that has been changed on purpose or by a different compiler. It can uncover related routines that traditional diffing or function comparison might miss and spot familiar functionality hidden under small edits, saving you from having to redo the same analysis every time you run into a slightly modified version of code you have seen before.

How BSim Works

In Chapter 13, we demonstrated how Function ID Databases (FidDb) work in Ghidra. FidDb's help you identify known functions by comparing them to a database of trusted signatures. It does this by creating a fingerprint of each function's bytes and structure. If a function in your binary matches one of these stored fingerprints, Ghidra can label it automatically. This is very effective when the code has not changed much, such as with common library routines.

BSim goes further by analyzing how a function actually operates. It creates a signature based on factors like control flow, how inputs and outputs are used, and how the function interacts with memory or other parts of the program.

Once Ghidra generates these behavioral signatures, BSim compares them to other signatures in a database. It does not only identify perfect matches. Instead, it scores how closely two functions resemble each other in what they do. A higher similarity score means they likely perform a similar task, even if the code was changed intentionally, recompiled with different options, or even across architectures and instruction sets. You can run BSim within a single project or connect it to a larger collection of signatures from other binaries you have studied.

In practice, FidDb's works best when you expect the code to stay the same, while BSim helps when you want to find connections even when the code has been tweaked to avoid detection. Together, they give you a stronger set of tools to identify what you already know and discover what has been disguised.

The BSim Workflow

Now that you know what BSim does, let's see how you can use it in a typical reverse engineering project. BSim's workflow in Ghidra is designed to be flexible, so you can run it on one binary or compare many functions across different projects.

While the steps will vary depending on your objectives, the basic procedure usually looks something like the following:

1. Begin by generating function signatures. After you open your program in Ghidra and run your usual analysis, run BSim's signature generator. This generator scans each function and creates a behavioral signature that describes how that function works.
2. Next, connect to a BSim database. Ghidra can keep these signatures in a local database just for your project, or you can connect to a shared BSim database that stores signatures from other binaries you or your team have analyzed before. Using a larger database can help you find more matches over time.
3. Once the signatures are in the database, search for functions that have similar behavior. Ghidra shows you a list of results with similarity scores that help you decide which matches are worth investigating.
4. Review and confirm matches. When you see a potential match, you can use Ghidra's usual tools, like the Listing view, Decompiler, or Function Graph, to review the details and confirm whether two functions really do the same thing. If they do, you can copy over your earlier analysis or comments to save time.
5. Apply and reuse your findings. After you confirm a good match, you can annotate it in your project, carry over labels, or add notes to help you and others recognize the function next time. In this way, BSim becomes a living resource that grows more useful with every binary you analyze.

BSim's workflow is simple but powerful. It gives you a new way to connect what you have already learned to new files so that you spend less time repeating work and more time focusing on what is actually new or unexpected in your reverse engineering projects.

Example: Investigating Ransomware Evolution

Your growing reputation as the office crypto expert has kept you busy, but now a new challenge is emerging. Ransomware attacks are becoming more common, and with them comes a flood of binaries packed with cryptographic routines. Manually comparing each one, even with the help of Program Diff and FCP, is quickly becoming unsustainable. You need a way to recognize when functions behave like the crypto you have already analyzed, even if they have been compiled differently, optimized, or deliberately altered to evade detection. To stay ahead of the curve, you decide it is time to bring Ghidra's BSim into your workflow, giving you a tool that can identify behavioral similarities and help you scale your crypto analysis efforts.

Activating BSim

The first step in putting BSim to work is making sure your CodeBrowser installation includes the BSim plugins. These are not always enabled by default, so you should confirm that they are available in your environment by selecting File ► Configure from the CodeBrowser menu and confirming that BSim is selected. Once the plugins are present, you will see BSim in your CodeBrowser menu and will have access to the BSim tools needed to build a database and run similarity queries.

Creating a BSim Database

Once you have confirmed that BSim is active, the next task is to create and populate a BSim database. This database holds the function fingerprints that BSim uses to measure similarity. There are several ways to get started with BSim, but one of the easiest is to use the BSim scripts shown in Figure 23-16.

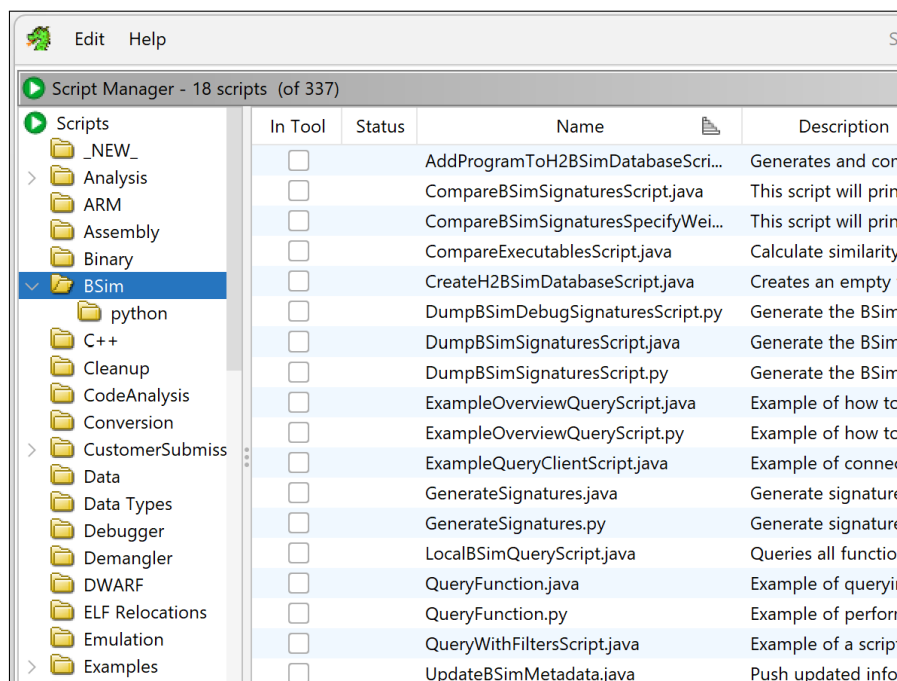


Figure 23-16: The default BSim scripts available in the Script Manager window

These scripts, which are included in the default Ghidra distributions, greatly simplify the process of mastering BSim and facilitate the use of headless Ghidra to use BSim. Running the *CreateH2BSimDatabaseScript.java* script from the CodeBrowser Script Manager window brings up the dialog shown in Figure 23-17.

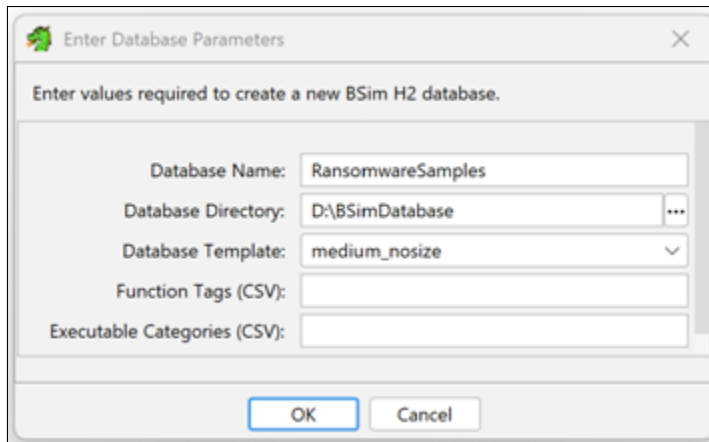


Figure 23-17: The dialog for entering BSim database parameters

After you have entered a name and location for your BSim database, you will manage the BSim server, which in this case is just a local file, using the dialog shown in Figure 23-18.

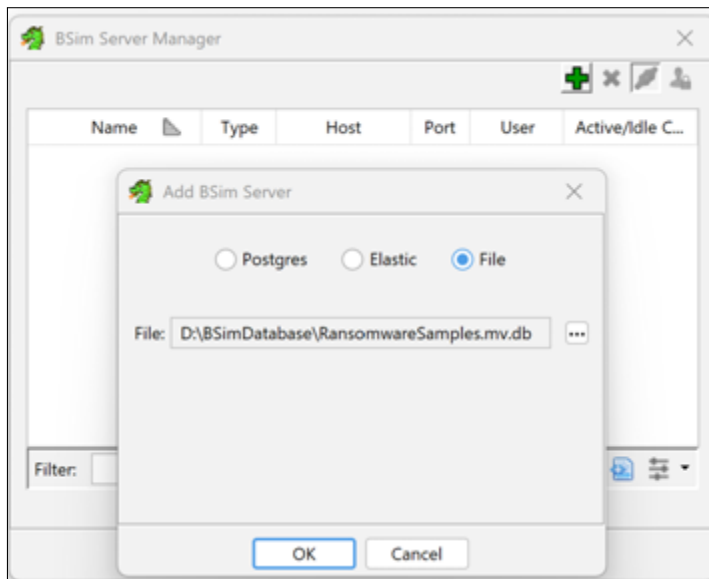


Figure 23-18: The dialog for connecting to a BSim server

Because you are the only crypto expert on your team, you choose to create a local database by selecting the BSim ► Manage Servers from the Code-Browser window and adding your BSim server.

Performing Analysis with BSim

Over time, as you analyzed ransomware samples, you've continued expanding your dedicated ransomware database within BSim. For example, after analyzing *ransomware_v236*, you saved the results and added the program to your BSim database using the default Ghidra script, *AddProgramToH2BSimDatabaseScript.java*. You evaluated each sample you processed in the same way and added it to your BSim database if appropriate, steadily expanding your library of ransomware functions.

Eventually, a new file arrives on your desk for analysis, called *ransomware_new*. As usual, you begin by running auto analysis to let Ghidra process the binary. Once the analysis completes, you save the program and then use BSim ► Search Functions to compare its functions against the steadily expanding library of ransomware crypto routines you have built. Figure 23-19 shows the associated BSim Search dialog.

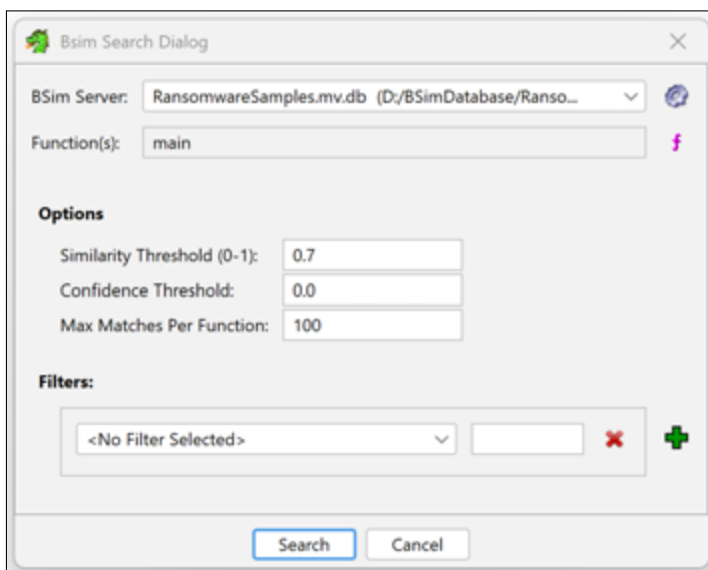


Figure 23-19: The BSim Search dialog with default settings

Figure 23-20 shows the result of the search indicating that there is a potential match with content in your BSim database.

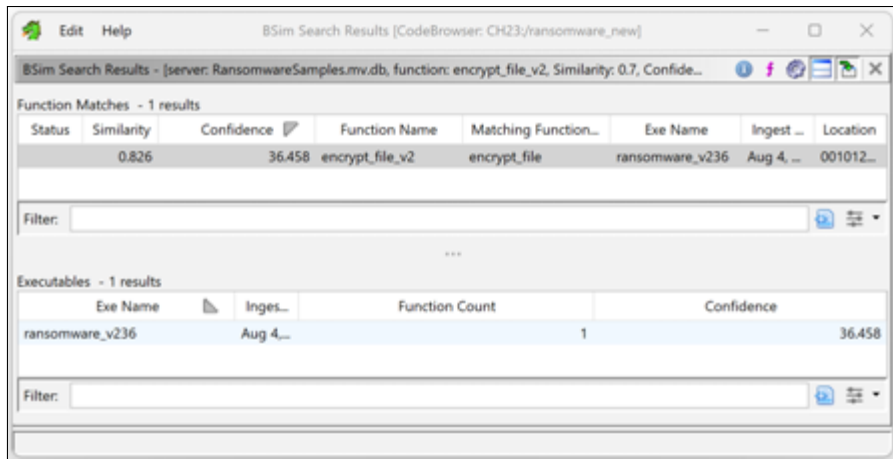


Figure 23-20: The BSim search dialog indicating a potential match

Clicking one of the function matches in the top window, you can use the right-click context menu and bring up the FCP window to visually inspect the results. FCP provides you with the familiar side-by-side Decompiler/Listing view options shown in Figure 23-21.

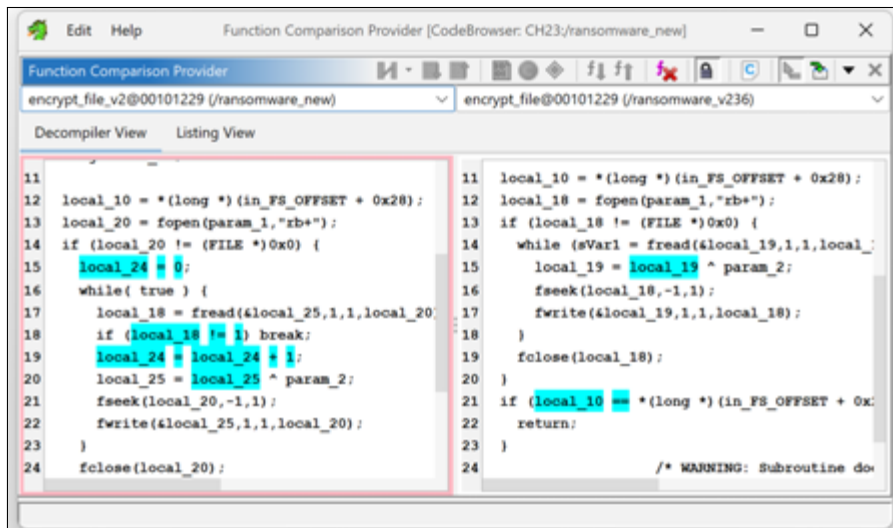


Figure 23-21: The FCP window Listing view

At first glance, the files do not look the same, and without context, it would be easy to dismiss them as unrelated. To see why BSim identified them as similar, let's examine the source code for each. We will start with the source code for the original file, *ransomware_v236*, located in the BSim database (with comments added for clarity):

```
#include <stdio.h>
#include <stdlib.h>

void encrypt_file(const char *filename, unsigned char key) {
    FILE *fp = fopen(filename, "rb+");
    if (!fp) return;

    unsigned char buffer;
    while (fread(&buffer, 1, 1, fp) == 1) {
        buffer ^= key;                // XOR encryption
        fseek(fp, -1, SEEK_CUR);
        fwrite(&buffer, 1, 1, fp);
    }

    fclose(fp);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <file>\n", argv[0]);
        return 1;
    }
    encrypt_file(argv[1], 0xAA); // Hardcoded key for demo
    printf("File encrypted.\n");
    return 0;
}
```

This code demonstrates an early version of the routine. It implements a very basic XOR encryption scheme with a hardcoded key. This design suggests that the author of this ransomware was relatively inexperienced, as the algorithm is trivial to reverse once identified.

In the new file, *ransomware_new*, the code is a bit different (and we've once again added comments for clarity):

```
#include <stdio.h>
#include <stdlib.h>

void encrypt_file_v2(const char *filename, unsigned char key) {
    FILE *fp = fopen(filename, "rb+");
    if (!fp) return;
```

```

unsigned char buffer;
size_t bytes;
// Added a useless variable and extra steps
int counter = 0;

while ((bytes = fread(&buffer, 1, 1, fp)) == 1) {
    counter++;
    __asm__("nop");
    buffer ^= key;
    fseek(fp, -1, SEEK_CUR);
    fwrite(&buffer, 1, 1, fp);
}

fclose(fp);
printf("Processed %d bytes\n", counter); // Extra log
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <file>\n", argv[0]);
        return 1;
    }
    encrypt_file_v2(argv[1], 0xAA);
    printf("File encrypted with modified routine.\n");
    return 0;
}

```

We can see how the author attempted to obscure the function's real purpose. They have added several distractions: a counter variable that does not affect the encryption, an assembly language no-operation instruction (nop) inserted for padding, and some extra logging. Despite these changes, the essential behavior of the function remains the same. It still performs XOR encryption with the same hard-coded key, but the structural differences make the two versions look less alike. This is exactly the kind of case where BSim excels. Even when surface details are altered to disguise intent, BSim's behavioral fingerprints allow you to identify that both functions accomplish the same underlying task.

An experienced analyst with a solid background in programming, reverse engineering, and even anti-reverse engineering could sit down with Function Comparison Provider or Program Diff and eventually determine that the two routines were equivalent. The problem is that doing so would require time and careful, line-by-line analysis, especially once the author has started adding meaningless operations to throw you off.

With BSim in the workflow, you do not have to exhaustively compare every suspicious function by hand. The tool automatically performs the behavioral analysis and highlights potential matches, giving you feedback that says, in effect, “These two functions likely perform the same or similar tasks.” That allows you to focus your attention on confirming the result rather than discovering it from scratch. By offloading this repetitive task to BSim, your workload is greatly reduced, and you can move more quickly through each new sample without sacrificing confidence in your findings.

AMPLIFYING EXPERTISE: FOCUSING TALENT WHERE IT COUNTS

It is important to recognize what BSim does and does not do. BSim is not a replacement for skilled reverse engineers. Anyone with experience in programming, reverse engineering, and anti-reverse engineering techniques (and a lot of time) could use tools like FCP or Program Diff to conclude that these two functions are essentially the same. The expertise is still necessary, and in complex cases, it is irreplaceable.

What BSim does is eliminate the need to spend that expertise on repetitive, low-value work. Instead of forcing you to dig through every function exhaustively, it performs the initial behavioral analysis and surfaces likely matches. This allows you to direct your energy where it matters most: confirming results, spotting meaningful differences, and tackling the genuinely challenging problems that require human judgment and creativity.

Another important benefit is that BSim helps you take advantage of both your own previous work and the work of others. Analysis you completed months or even years earlier can still be useful when those results are stored in a BSim database. Because these databases can be shared, your entire team can contribute to and benefit from a growing collection of fingerprints that captures collective experience.

By incorporating BSim into your workflow, you maximize the value of your expertise. The tool handles the routine comparisons that would otherwise consume your time, leaving you free to focus on confirming results, identifying meaningful differences, and solving the challenging problems that demand human skill and judgment.

This section covered a relatively simple example, but we could have just as easily demonstrated more complicated cases in which BSim was able to detect behavioral similarities. For example, functions compiled for different architectures, built with varying compiler options, or even stripped of symbols can still be matched when their underlying behavior is consistent.

To gain experience with BSim, experiment with using it in your own workflow. By exploring how it performs across different compilers, build configurations, and binary formats, you will discover both its strengths and its boundaries and learn how to best integrate it into your analysis process.

Behavioral similarity can reveal important connections across different binaries, but analysts often need to track how a single program changes over time. Successive versions may introduce new features, fix vulnerabilities, or

modify existing functions, and manually repeating the same analysis for each release can be tedious. To address this challenge, Ghidra provides Version Tracking, a framework that links results from one analysis to another so you can reuse work, focus on new code, and build continuity across program versions.

Comparing Versions with Version Tracking

Imagine that you have spent months analyzing a very large binary. The binary has hundreds or thousands of functions and no symbols. As part of your effort, you have provided meaningful names to the majority of the functions; renamed data, local variables, and function parameters; and added a mountain of comments that would take days or longer to re-create.

Now imagine that a new version of the binary is released, and the world stops using the version you know so much about. You could continue to analyze the old version to learn more about it under the assumption that the new version behaves similarly, but you would fail to learn about any new or modified behaviors in the updated binary. Instead, you decide to begin working on the newer version of the binary, and quickly realize that you are spending significant amounts of time reading the markup in the older binary to help guide you through the new binary.

Alternating back and forth between two CodeBrowser windows is not an optimal use of your time. It's time to switch from the CodeBrowser to another default tool that Ghidra provides in the Project Tool Chest: the Version Tracking tool, shown as footprints in Figure 23-22.

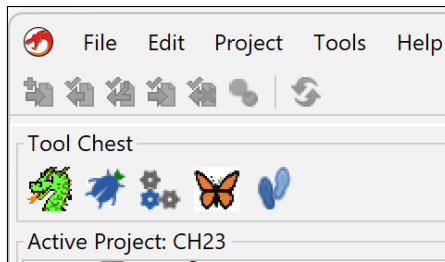


Figure 23-22: The Version Tracking tool in the Project Tool Chest

Ghidra's Version Tracking tool is designed to help you with precisely this situation. Through the use of various correlators, Ghidra attempts to match items, such as functions or data, in a source binary with their corresponding versions in a destination binary. Once functions have been matched between the two binaries, Ghidra can automatically migrate information, including your labels and comments, from the source binary to the destination binary. In addition to rapidly migrating your existing analysis, the Version Tracking tool makes it easy to identify which things haven't changed, which things have only minor changes (detected by diffing), and which things are entirely new.

The Version Tracking tool is one of the most configurable tools in Ghidra, which makes it easy to adapt to a particular line of inquiry. It is also a challenging tool to present in its entirety. In the following sections, we walk you through the version tracking process at a very high level and point you to resources that can help you discover the correct settings and components to use when using the Version Tracking tool to assist you in discovering the relationship between two files.

While Function Comparison and Program Diff tools answered specific questions about the atomic differences between two files or functions, the Version Tracking tool provides you with functionality to answer a more holistic question: How similar are these two binaries, and can you highlight and provide insight about the similarities between them? It further has the potential to harness the power of BSim when investigating these similarities. The foundational work unit in the Version Tracking tool is called a *session*, and each session is configured to identify and handle *correlations* between two files.

Correlators

At a high level, the Version Tracking tool is looking for correlations between two files. There are seven types of correlators that generate matches between two binaries:

- Data Match correlators
- Function Match correlators
- Legacy Import correlators
- Implied correlators
- Manual Match correlators
- Symbol Name Match correlators
- Reference correlators

Rather than just counting and compiling a list of specific differences in each of the categories, the Version Tracking tool extends correlations between the two files to identify matches with varying levels of exactness:

Exact matches These are one-to-one matches between the two files and can match data, function bytes, function instructions, or function mnemonics (for example, when two binaries contain the exact same function).

Duplicate data match These are exact matches that are not one-to-one matches (for example, when a string appears once in one file and seven times in the other file).

Similar matches These are matches that pass a user-controlled similarity threshold. The matching is similar to the approach used for word models described in Chapter 13, but uses 4-grams as well as trigrams.

With the ability to introduce thresholds and accept and reject matches, this tool offers a powerful capability to migrate your previous analysis to new

versions of a binary. Further, the information associated with each session provides an effective analysis audit trail that can help capture the incremental changes of a binary or the evolution of a malware family.

Sessions

While an in-depth walk-through of a complete session would take a significant amount of time, a basic version tracking session might include the following steps:

1. Open Ghidra's Version Tracking tool.
2. Create a new session by choosing a source file and a destination file.
3. For all appropriate correlators, add to the existing session, choose the correlator, select all resulting matches, and accept all matches and apply their markup items.
4. Save the session.
5. Close the session.

This workflow provides a very general overview, and the combinatorial potential associated with the correlator step is extensive. The potential of and the nuances associated with this tool cannot be covered in a single chapter. The Ghidra team has provided sample workflows (as well as significant documentation about the Version Tracking tool) in Ghidra Help. It is up to you to determine how best to apply the capabilities of this tool in your reverse engineering workflow.

Summary

In this chapter, we stepped away from a single binary to begin looking at ways to identify differences and similarities between binaries by using the Program Diff, Function Comparison Provider (FCP), BSim, and Version Tracking tools. These tools are valuable time-savers for porting existing work to new binaries, merging annotations from your colleagues, and rapidly identifying exactly what has changed between two versions of the same program.

As we wrap up our tour of Ghidra's vast landscape of features, know that we have only scratched the surface of Ghidra's capabilities. You should now have a deeper understanding of Ghidra and how it can be applied to the reverse engineering challenges you face. When you have questions, the Ghidra community is there to help through resources like GitHub, Stack Exchange, Reddit, YouTube, and many other forums.

More importantly, you should now be in a position to contribute by answering questions and providing help to others. Ghidra is community-supported software and continually evolving. We hope you participate by posting tutorials, writing and publishing Ghidra scripts and modules, identifying and addressing issues, or perhaps even developing new functionality for Ghidra itself. The future of Ghidra will be determined by the community, and that now includes you. Welcome, and happy reversing!

APPENDIX: GHIDRA FOR IDA USERS



If you are an experienced IDA Pro user interested in giving Ghidra a test run, either as a curiosity or as a more permanent transition, you may be familiar with many of the concepts presented in this book. This appendix is intended to map IDA terminology and usage to similar functionality in Ghidra. For specific usage information about any Ghidra feature mentioned here, please refer to the relevant chapters in this book, which discuss the features in far more detail.

We make no attempt to compare the performance of the two tools, nor do we argue for the superiority of one over the other. Your choice of which to use might be motivated by price or a specific feature offered by one and not the other. What follows is a whirlwind tour through the topics of the book from the perspective of an IDA user.

As you begin your journey, you may want a guide to help you learn an entirely new set of hotkeys. The *Ghidra Cheat Sheet*, which can be found in the *docs* directory, is a useful trifold that lists common user actions and their

associated hotkeys and/or tool buttons. Shortly, we'll cover how to remap hotkeys in the event that you miss your trusted IDA favorites.

Database Creation

Whereas IDA typically focuses on one binary at a time, Ghidra was designed using a project-oriented approach. So, it can contain multiple files per project, and it supports collaborative reversing on shared projects. The concept of an IDA database most closely maps to a single *program* within a Ghidra project, although there are now add-ons available to facilitate collaborative reverse engineering.

Your first interaction with Ghidra will involve creating projects (whether shared or non-shared) and importing “programs” (binaries) into those projects through the Project window. When you use IDA to open a new binary, and ultimately create a new database, you and IDA perform the following actions:

1. (IDA) Query every available loader to learn which loaders recognize the newly selected file.
2. (IDA) Display the load file dialog, presenting a list of acceptable loaders, processor modules, and analysis options.
3. (User) Choose the loader module that should be used to load file content into the new database, or accept IDA's default choice.
4. (User) Choose the processor module that should be used when disassembling database content, or accept IDA's default choice (which may be dictated by a loader module).
5. (User) Choose any analysis options that should be used when creating the initial database, or accept IDA's default choices. You may also elect to disable analysis altogether at this point.
6. (User) Confirm your choices by clicking OK.
7. (IDA) The selected loader module populates the database with byte content taken from the original file. IDA loaders generally do not load the entire file into the database, and it is generally not possible to re-create the original file from content available in the new database.
8. (IDA) If analysis is enabled, the selected processor module is used to disassemble code identified by the loader and any selected analyzers. (IDA calls analyzers *kernel options*.)
9. (IDA) The resulting database is displayed in IDA's user interface.

Ghidra has analogues for each of the listed steps; however, Ghidra's user interface is split into two main components, the Project window and the CodeBrowser window, and the process of loading binaries is broken into two distinct phases: import and analysis.

The Ghidra import process is generally initiated from the Project window and includes the following steps:

1. (Ghidra) Query every available loader to learn which loaders recognize the newly selected file.
2. (Ghidra) Display the import dialog, presenting a list of acceptable formats (roughly equivalent to loaders) and languages (roughly equivalent to processor modules).
3. (User) Choose the format for importing the file into the current project, or accept Ghidra's default choice.
4. (User) Choose the language for disassembling program content, or accept Ghidra's default choice.
5. (User) Confirm your choices by clicking OK.
6. (Ghidra) Use the loader associated with the selected format to load byte content taken from the original file into a new "program" in the current project. The loader creates program sections and processes the binary's symbol, import, and export tables, but it performs no analysis involving disassembly. Ghidra loaders generally load the entire file into your Ghidra project, though the CodeBrowser may not display some portions of the file.

Though this process is similar to IDA database creation, some steps are missing. With Ghidra, analysis takes place in the CodeBrowser, and once you have successfully imported a file, double-clicking that file in the Project view opens the file in the CodeBrowser. When you open a program for the first time, Ghidra performs the following steps:

1. (Ghidra) Open the CodeBrowser and display the results of the import process, asking whether you would like to analyze the file.
2. (User) Decide whether to analyze the file. If you elect not to analyze the file, you are dropped into the CodeBrowser, where you can scroll through byte content but will have no disassembly. In this case, you may choose Analysis ► Auto Analyze to analyze the file at any time. In either case, when you decide to analyze the file, Ghidra will display a list of "analyzers" compatible with the current file format and language setting. You may choose which analyzers to run and then modify any options the analyzer utilizes before allowing Ghidra to perform its initial analysis.
3. (Ghidra) Execute all selected analyzers and drop the user into the CodeBrowser to begin working with the fully analyzed program.

For more information about the import and analysis stages, refer to the appropriate chapters in this book. IDA has neither an analogy for Project view nor any collaborative reversing capabilities other than the shared Lumina database. We introduce Project view in Chapter 4 and discuss shared projects and support for collaborative reverse engineering in Chapter 11.

Basic Windows and Navigation

The CodeBrowser is Ghidra’s primary interface for analyzing programs, and as such, it is the Ghidra component most similar to IDA’s user interface. So, let’s spend some time relating IDA user-interface elements to their CodeBrowser equivalents.

In its default configuration, Ghidra’s CodeBrowser is a container for multiple specialty windows that display information about features of a program. We introduce the CodeBrowser in Chapter 4 and discuss it in more depth beginning in Chapter 5 and continuing through the remainder of the book.

The Listing View

At the center of the CodeBrowser is the Ghidra Listing window, which provides a classic disassembly, similar to your IDA View in text mode. To customize the format of your listings, use the Browser Field Formatter to modify, rearrange, and delete individual listing elements. As in IDA, your primary means of navigation within the Listing windows is double-clicking *labels* (the equivalent of IDA names) to move to the address associated with a label. Right-click, context-sensitive menus provide access to common operations associated with labels, including renaming and retyping.

Similar to IDA, each function in the listing has a header comment that lists the function’s prototype, provides a summary of the function’s local variables, and displays cross-references that target the function. You can access the Ghidra equivalent of IDA’s Stack view by right-clicking in a function’s header and selecting Function ► Edit Stack Frame.

If you enjoy the fact that IDA highlights all occurrences of a string that you click (such as a register name or instruction mnemonic), you may be disappointed to learn that this is not a default behavior in Ghidra. To enable it, visit Edit ► Tool Options ► Listing Fields ► Cursor Text Highlight and change Mouse Button to Activate from MIDDLE to LEFT.

Another feature you may either love or hate is Markup Register Variable References, which causes Ghidra to automatically rename the registers used to hold a function’s incoming parameters. To disable this behavior and have Ghidra use register name instruction operands, navigate to Edit ► Tool Options ► Listing Fields ► Operands Fields and uncheck Markup Register Variable References.

Finally, if you are longing for Ghidra to “do the right thing” when muscle memory causes you to use your favorite IDA hotkey sequences, you’ll want to spend some time in Edit ► Tool Options ► Key Bindings to reassign default Ghidra hotkeys to match those that you use in IDA. This is such a common task for IDA users that third-party key binding files have been published to automate the reassignment of all your favorite hotkey sequences.

The Graph View

Ghidra's Listing window is a text-only view. If you prefer working in IDA's graph view, you'll need to open a separate Function Graph or Graph window in Ghidra. Like IDA's graph view, Ghidra's Function Graph window can display a single function at any one time, and you can manipulate the items in the Function Graph window just as you would in the Listing window.

By default, Ghidra's graph layout algorithm may route edges behind basic block nodes, which may make tracing the edge more difficult. You can disable this behavior by visiting **Edit ► Tool Options ► Function Graph ► Nested Code Layout** and checking **Route Edges Around Vertices**.

The Decompiler

Ghidra includes a decompilation capability for all supported processors. By default, the Decompiler window appears to the right of the Listing window and will display decompiled C source code whenever your cursor is positioned within a function in the Listing view. If you like to add and view end-of-line comments in the generated C source, you'll need to enable them at **Edit ► Tool Options ► Decompiler ► Display** by checking **Display EOL comments**. On the same options tab, you'll also find an option to disable the printing of type casts, which can improve readability in some cases by dramatically decluttering the resulting code.

The decompiler has a tendency to aggressively optimize the code it generates. If you find yourself reading the disassembled version of a function and feel like behaviors are missing in the decompiled version, this may be because the decompiler has eliminated what it believes to be unreachable code within the function. To display that code in the Decompiler window, navigate to **Edit ► Tool Options ► Decompiler ► Analysis** and deselect "Eliminate unreachable code." We discuss the decompiler further in Chapter 19.

The Symbol Tree

The CodeBrowser's Symbol Tree window provides a hierarchical view of all symbols contained in a program. It contains six top-level folders representing six classes of symbols that may exist within a program. Clicking a name in any Symbol Tree folder will navigate the Listing window to the corresponding address:

Imports Relevant to dynamically linked binaries; provides a listing of external functions and libraries referenced by the program. Most closely correlates to IDA's Imports tab.

Exports Lists any symbols in the program that are publicly visible outside of the program. The symbols in this folder are often similar to those output by the `nm` utility.

Functions Contains an entry for each function in the program listing.

Labels Contains entries for any additional nonlocal labels within the program.

Classes Contains the names of any C++ classes for which Ghidra has located Runtime Type Identification (RTTI).

Namespaces Contains an entry for each namespace created by Ghidra during program analysis. Refer to Ghidra Help for more information on Ghidra namespaces.

The Data Type Manager

The Data Type Manager maintains all of Ghidra's knowledge about data structures and function prototypes. Each folder in the Data Type Manager is the rough equivalent of an IDA type library (*.til*). The Data Type Manager fills the role of IDA's Structures, Enums, Local Types, and Type Libraries windows, and we discuss it in detail in Chapter 8.

Scripting

Ghidra is implemented in Java, and while Java is its natural scripting language, it also provides full support for Python. In addition to routine scripts, the primary Java extensions to Ghidra include analyzers, plugins, and loader modules.

Together, Ghidra's analyzers and plugins take on the role that IDA's plugins fill, while its loaders perform essentially the same role as IDA loaders. Ghidra supports the concept of Ghidra processor modules, defined using a specification language known as SLEIGH.

Ghidra includes a basic script editor for routine scripting tasks as well as Eclipse and Visual Code Studio plugins to facilitate the creation of more complex Ghidra scripts and extensions. The use of Python, previously supported via Jython, is now also facilitated by PyGhidra, which provides support for Python 3. We discuss PyGhidra in Chapter 14.

The Ghidra API is implemented as a class hierarchy that represents the features of a binary as Java objects, and convenience classes provide easy access to some of the most commonly used API classes. We discuss Ghidra scripts in Chapters 14 and 15 and extensions in Chapters 15, 17, and 18.

Summary

Ghidra's capabilities are similar to IDA's. In some cases, the displays are close enough that the only things that will slow you down are new hotkeys, tool buttons, and menus. In other cases, Ghidra presents information in a different manner than in IDA, and your learning curve might be steeper. Whether you take advantage of Ghidra's customization capabilities to make it drive like IDA or take the time to learn a new way of doing things, you're likely to find that Ghidra meets most of your reverse engineering needs, and in some cases opens up entirely new ways of getting things done.

INDEX

A

- AAPCS (ARM Procedure Call Standard)
 - calling conventions, 99
- ABI (application binary interface), 99, 101, 102, 485
- About Ghidra, 32
- abstract base class (C++), 174
- abstract function, 174
- access control (Ghidra Server), 233, 235
- activation records, 53, 96, 97, 100, 102–112
- Add Block (Memory Map toolbar), 392, 394
- addEntryPoint method, 408, 421
- Add File to Version Control, 239
- Add Functions, 552, 553
- adding an analyzer module (Eclipse), 352, 361
- Add Reference dialog, 198
- Address interface, 321
 - getOffset method, 329–332
- AddressOfEntryPoint field, 393
- address ranges, 542, 546
- address table, 10
- addrinfo data type, 148, 149, 172
- Add Vertex to Group (Function Graph), 204
- ai_sockettype, 150
- alignment gap, 530
- analysis
 - dynamic, 6
 - static, 6, 11
- analysis engine (Decompiler), 456
- Analysis menu (CodeBrowser), 59, 67
- analysis options, 454
 - analyzers, 49, 50, 462
- analysisTimeoutPerFile (headless analyzer), 374
- Analyze All Open, 283
- analyzeHeadless, 366–377, 380–383
 - analyzeHeadlessREADME.html*, 32, 366, 374, 376
- analyzer modules
 - creating with Eclipse, 352
 - template (Eclipse), 351, 353
- analyzers, 49, 50, 284, 462
 - Decompiler, 454, 456
 - Decompiler Parameter ID, 50, 54
 - PE files, 45, 192, 280, 389, 393, 394, 401
 - Decompiler Switch Analysis, 50, 54
 - Function ID, 290–292, 296
 - headless, 363, 366–370, 378, 380–382
 - Non-Returning Functions –
 - Discovered, 462
 - Non-Returning Functions –
 - Known, 462
 - one shot, 283, 296
 - RTTI (runtime type identification), 181
 - Stack, 96
- annotation, 133, 153
- anti-debug techniques, 515
- anti-piracy protections, 515
- anti-static analysis, 496
- API (application programming interface), 53, 498, 500, 509
- application binary interface (ABI), 99, 101, 102, 485
- Apply Differences, 543, 545, 548, 549
- applying structure layouts, 171
- Apply Selection (hotkey F3), 548
- architecture size, 281
- archives
 - creating data type archives, 287, 288
 - creating new file archives, 290
 - creating new project archives, 290
- arguments. *See* parameters
- ARM, 430, 444, 499
 - instructions, 98, 99

- ARM Procedure Call Standard (AAPCS)
 - calling conventions, 99
- arrays, 142, 145, 146
 - Array type option (Ghidra), 156
 - base address, 151–153, 156–161, 170
 - bounds, 151
 - constant indices, 153, 154, 161
 - create, 141
 - elements, 150–153, 156, 159
 - heap-allocated, 157, 159
 - index value, 150, 161, 165
 - member access, 150
 - reference, 150
 - stack-allocated, 154–157, 162
 - static assignments, 157
 - of structures, 165
 - variable indices, 153
- Array type option (Ghidra), 156
- articulation, 202
- ASCII, 16, 28, 185
- ASCII export (Ghidra), 525
- askAddress method, 323
- askDirectory method, 324
- askFile method, 324
- askInt method, 323
- askString method, 312, 323
- askYesNo method, 323
- assemblers, 4, 7, 9
- assembler (Ghidra), 523, 526–528, 537
- Assembler Rating, 526, 527
- assembly language directives, 4
- Attach existing FidDb, 292
- authentication
 - functions, 523
 - Ghidra Server, 223, 226, 232–235
- auto analysis, 92, 109, 283, 286, 291, 296
 - Analysis Options, 49, 50
 - options, 49, 50, 454, 462
 - summary dialog, 51
- Auto Create Structure, 465
- automated structure creation, 463
- automatic storage class (C++), 178

B

- back references, 186, 187, 197
- backward navigation, 95
- backward slice, 461

- base address, 48
 - array 151–153, 156–161, 170
- Base Library (FidDb), 295
- base virtual address (PE files), 391, 393
- basic block, 69, 192, 200, 203–206, 212, 213
- basic data transformations, 142
- basic disassembly algorithm, 8
- Batch mode, 230, 231
 - headless analyzer, 370
 - Import, 299, 300, 370
- behavioral similarity. *See* BSim
- big-endian architecture, 10
- binaries, 4
 - ELF, 94, 286
 - Mach-O, 23, 24, 28
 - stripped, 18, 152, 488
- binary differencing, 539
- binary format export (Ghidra), 522, 533
- binary search, 470, 472, 473, 475
- bitness, 281
- Browser Field Formatter, 67, 135, 251–253, 449
- BSim, 540, 557
 - activating, 560
 - analysis, 562
 - database, 560
 - ransomware example, 559
- bss section, 73, 151, 153, 397
- buffer overflow, 186, 352
- buildLanguage.xml*, 430
- build options (compiler), 152, 470, 478, 482
- BuiltInTypes archive, 286, 287
- byte code, 4
- Byte Viewer, 523–525
 - editing, 523
 - options, 81, 248, 524
 - window, 81, 82, 523–525

C

- C++
 - abstract base class, 174
 - compilers, 160, 476–484, 491
 - compiler variations
 - function overloading, 485
 - RTTI, 485
 - constructor, 177–179, 182

- delete operator, 179
- destructors, 177–179
- dynamic_cast, 180, 181, 485, 486
- inheritance, 172, 173,
 - 179–182, 486
- name mangling, 25, 26, 180,
 - 181, 485
- new operator, 175, 178
- object life cycle, 177
- polymorphism, 172, 180
- pure virtual function, 174, 175
- reversing, 172
- RTTI, 180, 181, 485–488
- storage class, 178
- this pointer, 173, 176, 181
- typeid, 180, 181, 485
- vftables, 174–182, 194–196, 486–489
 - pointer, 174–179
- virtual functions, 173–176, 194, 197
- call flow, 191
- calling conventions, 99–101, 107,
 - 114, 141
 - AAPCS, 99
 - cdecl, 98
 - C language, 490
 - fastcall, 98
 - Microsoft x64, 99
 - standard calling convention, 97
 - stdcall, 98
 - System V AMD64 ABI, 99, 101
- call mechanics, 96, 103
- C/C++ format export (Ghidra), 533
- cdecl calling convention, 98
- c++filt utility, 25, 26
- Characteristics field (PE files), 398
- CheatSheet.html*, 33
- Choose active FiddBs, 292
- Choose Data Type (hotkey T), 390
- C language
 - calling conventions, 490
 - compilers, 160, 173, 176, 178, 181,
 - 476–479, 482, 486, 491
 - format export (Ghidra), 533
 - malloc function, 157
 - strcpy function, 196
- Classes folder (Symbol Tree), 77, 195,
 - 486, 487
- classification tools, 16, 23
- Clear Code Bytes (hotkey C), 141, 512
- clearListing method, 327, 525
- clipboard icon (Eclipse), 340
- Close View (Ghidra Project), 234
- CodeBrowser, 48–50, 53–55
- CodeBrowser menu
 - Analysis menu, 59, 67
 - Analyze All Open, 283, 300
 - Auto Analyze, 49–54, 283, 286, 291
 - One Shot, 283, 296
 - Edit menu, 59, 61, 64
 - Tool Options, 55, 248, 251
 - File menu, 58
 - Export Program, 532
 - Parse C Source, 288, 289
 - Help menu, 60
 - Search menu, 60, 72
 - For Direct References, 518
 - For Instruction Patterns, 518–521
 - Memory, 117, 517–518
 - Program Text, 116
 - Tools menu, 60, 67, 69
 - Function ID menu, 292
 - View Program Differences, 540
 - Window menu. *See* CodeBrowser windows
- CodeBrowser toolbar
 - Navigation toolbar, 72, 95
 - Redo, 120
 - Undo, 120
- CodeBrowser windows
 - Bytes, 81, 82, 248
 - Console, 78
 - Data Type Manager, 51, 60, 77, 149,
 - 169, 170, 210, 287
 - Decompiler, 61, 78–80, 454, 456–458,
 - 460–463, 465, 468
 - Defined Data, 82–84
 - Defined Strings, 84, 85
 - Function Call Graph, 89
 - Function Graph, 68–71, 200–206
 - Listing, 53, 509, 512, 524–527,
 - 530–534
 - Memory Map, 48, 88, 391–398
 - Program Trees, 73, 210, 530
 - rearranging, 246
 - Script Manager, 304–314, 321,
 - 340–343, 348–350

- CodeBrowser windows (*continued*)
 - Symbol References, 85–88
 - Symbol Table, 85–88
 - Symbol Tree, 53, 60, 74–76, 92–94, 148–150, 181, 291–293, 296, 487
- code caves, 530
- code cross-reference, 187, 192
- code display options, 134, 135
- code optimization, 100, 110, 479, 481, 492
- collaborative SRE, 36, 222–228, 232–236
- collapsing blocks (Function Graph), 71
- color customization (Function Graph), 206
- Color Editor, 248, 249
- comments (Eclipse), 355
- comments (Ghidra), 126, 129–135, 139, 146
 - annotation, 133
 - Symbol option, 153
 - EOL (end-of-line) comments, 130, 133, 135, 325
 - plate comments, 131, 297, 325
 - post comments, 131, 325
 - pre comments, 131, 132, 325
 - repeatable comments, 129, 132, 325
 - Set Comment, 129, 130, 133
- commit (headless analyzer), 377
- Common Symbols File, 295
- Common Vulnerabilities and Exposures (CVE), 317, 320
- Compare Selected Functions (hotkey `SHIFT-C`), 550, 551, 554–556
- comparing functions, 549–551, 554–556
- compilers
 - build options, 470, 478, 480, 482
 - compiler field, 281
 - debug versions, 476–479
 - GNU gcc/g++, 160, 476–484, 491
 - O2 option, 483, 484
 - identification, 52
 - linker options, 480
 - Microsoft C/C++, 160, 173, 176, 178, 181, 476–479, 482, 486, 491
 - release versions, 478
- compiler validation, 7
- compiler variations
 - build options, 470, 478, 482
- C++ Function Overloading, 485
- C++ RTTI, 485
- inline functions, 483, 484
- main(), 490
- modulo operator, 479–481
- switch statement, 470–478
- ternary operator, 481, 482
- conditional branching instructions, 11
- configurations (Ghidra)
 - Developer, 32, 258
 - Experimental, 256, 258
 - Function ID, 32, 76
 - Ghidra Core, 32, 258–260
- Configure Gradle (Eclipse), 360
- connect (headless analyzer), 376
- Console window (CodeBrowser), 78
- constant indices, 153, 154, 161
- constructors
 - C++, 174, 177–179, 182
 - inline, 182
 - SLEIGH, 434
- control flow, 201, 212, 213
- converting data and code
 - Clear Code Bytes (hotkey `C`), 141, 512
 - Disassemble (hotkey `D`), 142, 399, 403, 512
- Convert to Shared (Ghidra Server), 236
- Copy (Function Graph toolbar), 203
- correlators, 567–569
- C-Parser plugin, 288, 289
- crackme, 506, 507, 511
- Create Array (hotkey `[]`), 146
- createAsciiString method, 327
- createByte method, 327
- createData method, 421
- Create Function (hotkey `F`), 138
- createFunction method, 327
- Create Ghidra Script, 379
- createLabel method, 326, 421
- createMemoryBlock method, 408, 414, 420, 421
- createMemoryReference method, 421
- Create new empty FidDb, 292
- Create Structure (hotkey `SHIFT-[]`), 167
- Create Tool, debugger example, 259, 265–278

- createUnicodeString method, 327
- creating
 - analyzer modules (Eclipse), 352
 - data type archives, 288
 - FidDBs, 292
 - file archives, 290
 - loader modules, 401
 - module projects (Eclipse), 345
 - project archives, 290
 - projects (Ghidra), 43
 - script projects (Eclipse), 343
 - scripts (Eclipse), 340, 341, 343
 - shared projects (Ghidra Server), 226
 - structures, 161, 166–168
 - tools (Ghidra), 259
- cross-references, 65, 187–189, 192–198, 489, 518
 - code, 187, 192
 - data, 193
 - enumerating, 330
 - pointer, 193, 194
 - read, 193
 - suffix (*), 211
 - suffix (T), 211
 - write, 193
- cspec (headless analyzer), 374, 376
- C-style structs. *See* structures
- currentAddress, 323, 330, 331
- current file location, 65
- currentLocation, 323
- currentProgram, 312, 316, 319, 322, 327–332
- currentSelection, 312, 323
- customizing Ghidra, 245
- CVE (Common Vulnerabilities and Exposures), 317, 320
- cycle groups, 143

D

- dangerous functions, 318
- data cross-reference, 193
- Data Execution Prevention (DEP), 352
- data/languages* directory, 406, 423
- .data* section, 73, 151, 396–398
- data structures, 150
- data type archives, 286–288, 290
 - BuiltInTypes, 286, 287
 - opening, 289
- Data Type Manager window, 51, 60, 77, 149, 169, 170, 210
 - BuiltIn Types, 286, 287
 - data type archives, 287
 - creating, 288
 - New File Archive, 290
 - New Project Archive, 290
 - opening, 289
- data types, 456–458
 - addrinfo, 148, 149, 172
- DAT_ prefix, 93, 120, 127
- dead listings, 92
- debuggers, 7, 502–504, 534–537
 - plugins, 260
 - WinDbg, 10
- debugging displays, 7
- debug mode, 476–479
- decode, 498, 500, 501, 504, 505
- Decompiler analyzer, 454, 456
- Decompiler Parameter ID, 50, 54
 - PE files, 45, 192, 280, 389, 393, 394, 401, 410, 411
- decompilers, 4–5
- Decompiler Switch Analysis, 50, 54
- Decompiler window, 61, 78–80, 460–463, 468
 - analysis engine, 456
 - analysis options, 454, 462
 - Eliminate unreachable code, 455
 - Simplify predication, 455, 456
 - Auto Create Structure, 465
 - automated structure creation, 463
 - backward slice, 461
 - data types, 456–458
 - Edit Function Signature, 463
 - editing in the Decompiler window, 457–458
 - editing variable types and names, 460
 - forward slice, 461
 - function prototypes, 458
 - highlighting slices, 461
 - non-returning functions, 462–463
 - overriding function signatures, 459
 - program slice, 461
 - retyping variables, 460
 - Structure Editor window, 465
- deep inspection tools, 26
- defined data, 518

- Defined Data window, 82–84
- Defined Strings window, 84, 85, 145
- delete operator (C++), 179
- Delete Project (Ghidra), 229
- deleteProject (headless analyzer), 372
- deleting functions (hotkey DEL), 138
- density, 471–473
- deobfuscation, 498, 499
- DEP (Data Execution Prevention), 352
- Destination Folder field (Import File), 44
- destructors (C++), 177–179
- desynchronization, 499
- Detect-It-Easy (DiE), 18
- Determine Program Differences dialog, 541, 543
- Developer configuration (Ghidra), 32, 258
- Diff Details window, 542, 543, 545, 548
 - EOL-Comment section, 547
 - Pre-Comment section, 549
- Diff View, 540–542, 545, 549, 552, 556
- directives, 4
- direct references search, 492, 518
- Disassemble (hotkey D), 142, 399, 403, 512
- disassemble method, 327
- disassemblers, 4, 7, 11–13
 - diStorm, 28
 - NASM, 28
 - ndisasm, 28
- disassembly
 - basic algorithm, 8
 - conditional branching instructions, 11
 - desynchronization, 499
 - function call instructions, 11
 - introduction to, 4–14
 - linear sweep, 9
 - example, 9–11, 13
 - navigation, 92
 - recursive descent, 10, 13, 142
 - example, 13
 - return instructions, 12
 - sequential flow instructions, 11
 - theory, 4
 - unconditional branching instructions, 11
- dist* directory, 360, 361

- diStorm, 28
- Docker, 35, 224
- docs* directory, 32, 33, 35, 37
- downloading Ghidra, 33
- dumpbin utility, 23–25, 28
- dynamic analysis, 6
- dynamic_cast, 180, 181, 485, 486
- dynamic linking, 21, 22, 207, 211
- dynamic memory allocation, 157, 178

E

- Eclipse, 338–357
 - analyzer module template, 351, 353
 - clipboard icon, 340
 - comments, 355
 - Create Ghidra Script dialog, 341, 343
 - creating module projects, 345
 - creating modules, 401
 - creating script projects, 343
 - creating scripts, 340, 341, 343
 - Edit Script, 338, 340
 - error tag, 343
 - expand lines, 340, 351
 - Exporter, 360
 - export module, 360
 - GhidraDev, 338–347, 349, 351, 353, 355, 357, 359–361, 363, 428, 435
 - home scripts, 347, 507
 - integration, 352, 363
 - Link options, 347
 - loader module template, 351
 - module directories, 349
 - module templates, 423
 - Package Explorer, 344–349, 353
 - Quick Fix options, 343
 - Run options, 350, 360
 - task tag, 342, 355
 - testing modules, 410
 - TODD comments, 342, 343, 355, 436, 437
 - tutorials, 339
- edges, 186
- Edit function (hotkey F), 139, 141
- Edit Function Signature, 463
- editing
 - in the Decompiler window, 457–458
 - labels, 125, 127
 - scripts (Eclipse), 338, 340

- structure members, 169
- themes, 255
- tool (Tool Options), 251
- variable types and names, 460
- Edit menu (CodeBrowser), 59, 61, 64
- Edit menu (Ghidra Project), 232
- Edit Plugin Path (Ghidra Project), 231
- Edit Tool Options (Ghidra Project), 232
- ELF binaries
 - data types, 286
 - file format, 8, 17, 18, 22–24, 27, 28
 - loader example, 416–426
 - locating `main()`, 490
 - navigating, 94
 - UPX example, 293, 296
 - utilities, 27
- Eliminate unreachable code, 455
- emulating assembly language
 - behavior, 333
- emulation, 497, 499–507, 509–511, 513
- emulator, 499, 501–508, 510–512
 - SimpleEmulator* example, 507–508
- Emulator class, 505
- EmulatorHelper class, 505
 - dispose method, 511
 - enableMemoryWriteTracking
 - method, 508
 - getEmulateExecutionState
 - method, 510
 - getTrackedMemoryWriteSet
 - method, 510
 - readMemoryByte method, 511
 - run method, 509, 510
 - setBreakpoint method, 509
- encoding, 498
- encryption, 498, 499
- endianness, 281
- end-of-line (EOL) comment, 129, 130, 133–135, 325, 489
- entry point, 69, 75, 88, 490
- enumerating cross-references, 330
- enumerating functions, 329
- enumerating instructions, 330
- EOL (end-of-line) comment, 129, 130, 133–135, 325, 489
- EOL-Comment section, 547
- epilogue, 97, 101, 105, 107
- EPROM, 533

- error messages (headless analyzer), 368
- error tag (Eclipse), 343
- exception, 511
- Executable and Linkable Format (ELF).
 - See* ELF binaries
- Expand Down (Memory Map toolbar), 392, 398
- expand lines (Eclipse), 340, 351
- Expand Up (Memory Map toolbar), 392
- Experimental configuration (Ghidra), 256, 258
- explicit forward reference, 197
- exploit, 186, 352
- Export dialog, 532
- Exporter (Eclipse), 360
- exporting files (Ghidra), 516
 - ASCII format, 525
 - binary format, 522, 533
 - C/C++ format, 533
 - HTML format, 533
 - Intel Hex format, 533
 - XML format, 533
 - Zip format, 533
- export module (Eclipse), 360
- Export Program menu
 - (CodeBrowser), 532
- Exports* folder (Symbol Tree), 75
- Extensions* directory, 32, 35
- EXT_ prefix, 93, 120, 127

F

- fallthrough, 528
- fastcall calling convention, 98
- FCP. *See* Function Comparison Provider (FCP)
- .fidbf*, 76, 297
- FidDBs (Function ID databases), 290–301
 - attaching, 292
 - creating, 292
 - populating from programs, 292, 294
- file extensions, 15, 387, 410
 - .a*, 298–301, 383, 464
 - .class*, 16, 287
 - .cspec*, 422, 423, 429
 - .dll*, 21, 24, 25, 491
 - .fidbf*, 76, 297
 - .gdt*, 286, 287, 289, 390

- file extensions (*continued*)
 - .gif*, 264
 - .gpr*, 237
 - .h*, 207, 400, 430, 439, 564
 - .idx*, 422, 423, 432, 433
 - .jpg*, 264
 - .keep*, 238, 243
 - .ldefs*, 421, 422, 429, 430
 - .o*, 19, 298, 313
 - .opinion*, 406, 423, 429, 430
 - .pdf*, 432, 433
 - .png*, 264
 - .prf*, 289
 - .pspec*, 422, 429
 - .py*, 314
 - .rep*, 237
 - .sinc*, 430, 434–442, 445–452
 - .sla*, 422, 430, 437, 441–443
 - .slaspec*, 430, 434, 438, 439, 442, 445
 - .sng*, 284, 285
 - .tar*, 231
 - .tool*, 265
 - .txt*, 285, 349, 355, 357, 362, 379–382, 429, 430, 433
 - .xml*, 430
 - .zip*, 34, 231, 360, 361
- File menu (CodeBrowser), 58, 532
- File menu (Ghidra Project), 229
- file offset, 48
- files
 - hijacked (Ghidra Server), 240, 242, 243
 - private (Ghidra Server), 242, 243
- File System mode (Import), 298
- filesystem paths, 366
- file utility, 16, 17, 298
- findBytes method, 325
- find method, 325
- findSupportedLoadSpecs method, 406, 407, 418
- Flat API, 321–327, 330, 404, 408
- FlatProgramAPI class
 - addEntryPoint method, 408, 421
 - clearListing method, 327
 - createAsciiString method, 327
 - createByte method, 327
 - createData method, 421
 - createFunction method, 327
 - createLabel method, 326, 421
 - createMemoryBlock method, 408, 414, 420, 421
 - createMemoryReference method, 421
 - createUnicodeString method, 327
 - disassemble method, 327
 - findBytes method, 325
 - find method, 325
 - getByte method, 324, 334, 335
 - getBytes method, 324
 - getDataAfter method, 325
 - getDataAt method, 325, 421
 - getFirstData method, 325
 - getFirstFunction method, 326, 329
 - getFirstInstruction method, 325
 - getFunctionAfter method, 326, 330
 - getFunctionAt method, 326, 332
 - getInstructionAfter method, 325
 - getInstructionAt method, 325
 - getInt method, 324
 - getLong method, 324
 - getReferencesFrom method, 326, 331
 - getReferencesTo method, 326, 332
 - getSymbolAt method, 326, 331
 - getSymbols method, 326
 - removeFunctionAt method, 327
 - setEOLComment method, 327
- flow, 188, 191, 192, 196
- flow arrow, 66
- flow types
 - call, 191
 - jump, 191
 - sequential, 188, 191, 200
- footprints icon (Version Tracking), 567
- forbidden labels and names, 124
- For Direct References
 - (Search menu), 492
- For Instruction Patterns (Search menu), 518–520, 522
- Format option, 282
- formatting instruction operands, 134, 136
- formatting XREFs, 188
- forward navigation, 95
- forward references, 197
- forward slice, 461
- fragment, 73
- frame pointer, 105, 110
- FrontEndPlugin. *See* Ghidra Project window

- full hash, 290
- fullscreen/graph view (Function Graph), 204
- Function Call Graph window, 89, 207–210
- function call instructions, 11
- Function Comparison Provider (FCP), 549
 - Add Functions, 552, 553
 - Listing view, 574
 - toolbar, 551, 554–556
- Function Graph window, 68–71, 199–204
 - articulation, 202
 - basic block toolbar
 - Add Vertex to Group, 204
 - Background Color, 204, 206
 - Combine Vertices, 204
 - Jump to XREF, 204
 - Restore Group, 204
 - Ungroup Vertices, 204
 - collapsing blocks, 71
 - color customization, 206
 - customizing, 71
 - Function Graph View zooming, 60, 62, 70
 - grouping blocks, 71, 203
 - interaction threshold, 202
 - nodes, 199, 206–210
 - rearranging, 62, 70
 - satellite view, 69, 70, 201, 206
 - toolbar
 - Copy, 203
 - Nested Code Layout, 203
 - Paste, 203
 - Snapshot, 203
- Function ID analyzer, 290–292, 296
- Function ID configuration (Ghidra), 32, 76
- Function ID database (FidDb), 290–298, 300, 301
- Function ID menu
 - Attach existing FidDb, 292
 - Choose active FidDb, 292
 - Create new empty FidDb, 292
 - Populate FidDb from programs, 292, 294
- Function ID Plugin, 290, 292, 301

- Function Interface
 - getBody method, 328–331
 - getPrototypeString method, 328
 - getStackFrame method, 328, 329
- functions
 - arguments, identifying, 53
 - attributes, 138–140
 - call mechanics, 96, 103
 - comparing, 549–551, 554–556
 - Create (hotkey F), 138
 - Delete (hotkey DEL), 138
 - Edit (hotkey F), 139, 141
 - epilogue, 97, 101, 105, 107
 - inline, 182, 483, 484
 - library, 148, 178, 211
 - locating main(), 490
 - manipulating, 134, 138
 - modifying signatures, 459, 463
 - namespace, 124
 - non-returning, 462, 463
 - overloading, 25, 179, 180, 485
 - parameters, 148–150, 159, 179, 180, 483–485
 - prologue, 97, 101, 105, 108, 112, 399
 - prototypes, 142, 148, 179, 458
 - signature, 463, 546–548, 555
 - modifying, 459, 462
 - thunk, 211
 - variable number of arguments, 98, 459
- Functions* folder (Symbol Tree), 76, 77
- FUN_ prefix, 93, 120, 127
- fuzzing, 6

G

- gcc, 476–484, 491
- gcc/ld (-s option), 152
- getAddress method, 319, 320, 322, 332
- getaddrinfo, 148, 149
- getBody method, 328–331
- getByte method, 324, 334, 335
- getBytes method, 324
- getComment method, 329
- getDataAfter method, 325
- getDataAt method, 325, 421
- getDefaultOptions method, 409
- getFirstData method, 325
- getFirstFunction method, 326, 329

- getFirstInstruction method, 325
- getFromAddress method, 322, 332
- getFunctionAfter method, 326, 330
- getFunctionAt method, 326, 332
- getFunctionManager method, 327
- getInstructionAfter method, 325
- getInstructionAt method, 325
- getInt method, 324
- getLanguageID method, 328
- getListing method, 312, 319, 327, 330, 331
- getLong method, 324
- getMaxAddress method, 328, 329
- getMemory method, 328
- getMinAddress method, 328, 329, 331
- getMnemonicString method, 320, 329
- getName method, 312, 319–322, 329–332, 406, 413, 417–419
- getNumOperands method, 320, 329
- getOffset method, 329–332
- getOperandType method, 329
- getPrototypeString method, 328
- getReferenceManager method, 328
- getReferencesFrom method, 326, 331
- getReferencesTo method, 326, 332
- getReferenceType method, 322, 331, 332
- getStackFrame method, 328, 329
- getSymbolAt method, 326, 331
- getSymbols method, 326
- getSymbolTable method, 319, 328, 332
- getTier method, 407, 417
- getTierPriority method, 407, 413
- getToAddress method, 322, 331
- Ghidra
 - CheatSheet.html*, 33
 - downloading, 33
 - educational content, 35
 - icon, 36
 - licenses, 32, 36
 - source code, 33, 34
 - startup script, 37
 - support directory, 37
 - support documentation, 32, 33, 35, 37
 - tutorials, 33
- Ghidra.app.script.GhidraScript, 307, 312
- ghidra_basics.py*, 314
- GhidraClass*, 35
- Ghidra Core configuration, 32, 258, 260
 - Plugin Path, 259
- GhidraDev, 338–343, 345–347, 355–363, 428
 - New menu
 - Ghidra Module Project, 341, 345, 353
 - Ghidra Script, 340–344, 348–350
 - Ghidra Script Project, 341, 343, 344
 - README.html*, 338
- GhidraDevUser Consent, 339
- Ghidra directory structure, 34
 - docker*, 35
 - docs*, 32, 33, 35, 37
 - Extensions*, 32, 35
 - Ghidra*, 36
 - GPL*, 36
 - licenses*, 32, 36
 - server*, 36
 - support*, 36
- Ghidra extensions
 - Gradle, 355, 360
 - install, 361
- Ghidra/Features* directory, 338, 348, 356
- Ghidra GUI, 368–371, 375–378
- Ghidra Help (F1 hotkey), 32
- \$GHIDRA_HOME, 377
- ghidra.ico*, 36
- Ghidra installation directory, 377
- Ghidra module directories
 - data/languages*, 423
 - dist*, 360, 361
 - Ghidra/Features*, 338, 348, 356
- Ghidra Module Extension Export, 360
- Ghidra Module Project templates, 345, 346, 349, 354
 - Analyzer, 351, 353
 - Loader, 351
- Ghidra modules, 345
- Ghidra Project menu
 - Edit menu
 - Plugin Path, 231
 - Tool Options, 232
 - File menu
 - Delete Project, 229
 - Import File, 44
 - New Project, 43, 226, 227

- Project menu, 235
 - View Project, 236
 - View Recent, 234
 - View Repository, 233
 - Tools menu
 - Create Tool, 259
 - Import Tool, 265
 - Ghidra Project window, 42, 43, 54, 254–256
 - Running Tools, 255
 - Table View, 227, 228
 - Tool Chest, 255, 259, 567
 - ghidraRun*, 37
 - Ghidra Script, 341–344, 348–350
 - GhidraScript class, 306, 307, 310, 315, 322–324
 - askAddress method, 323
 - askDirectory method, 324
 - askFile method, 324
 - askInt method, 323
 - askString method, 312, 323
 - askYesNo method, 323
 - currentAddress instance variable, 323, 330, 331
 - currentLocation instance variable, 323
 - currentProgram instance variable, 312, 316, 319, 322, 327–332
 - currentSelection instance variable, 312, 323
 - goTo method, 324
 - popup method, 323, 330
 - printf method, 312, 323, 330–332
 - toAddr method, 324, 334, 335
 - Ghidra Script Project, 344
 - ghidra_scripts*, 304
 - Ghidra Server, 228, 234–236
 - access control, 233, 235
 - authentication methods, 223, 226, 232, 233
 - configuration file, 222, 224
 - Convert to Shared, 236
 - directory, 33
 - headless analyzer options, 376, 377
 - hostname, 226
 - installation, 222–226
 - IP address, 226
 - platforms, 223
 - race condition, 225
 - server administrator, 225
 - Ghidra Server project, 240
 - Ghidra source code, 33, 34
 - Ghidra themes, 255
 - Ghidra Tools
 - CodeBrowser, 48–50, 53–55
 - Version Tracking, 540, 567–569
 - Ghidra User Agreement, 32
 - Ghidra website, 32
 - Ghidra workspace, 277
 - GitHub, 32–34
 - global namespace, 124
 - global structures, 161, 168
 - global variable, 152
 - GNU gcc/g++, 160, 476–484, 491
 - packed attribute, 160
 - pack pragma, 160
 - Go To (hotkey G), 94, 95, 210
 - goTo method, 324
 - GPL (GNU General Public License), 32, 36
 - Gradle Wrapper option, 355, 360
 - grep, 298
- ## H
- hash function, 290
 - hashing
 - full hash, 290
 - specific hash, 290
 - headless analyzer, 363, 366–370, 378–382
 - batch import example, 370
 - error messages, 368
 - launching, 366–377, 380–383
 - README file, 366, 374, 376
 - syntax, 367
 - headless analyzer options
 - general
 - analysisTimeoutPerFile, 374
 - cspec, 374, 376
 - deleteProject, 372
 - import, 367–382
 - loader, 376
 - log, 371
 - max-cpu, 376
 - noanalysis, 368, 371, 377
 - overwrite, 372

headless analyzer options (*continued*)

general (*continued*)

- processor, 374, 376, 383
- readOnly, 372, 382
- recursive, 372, 373, 382, 383

script

- okToDelete, 378
- postScript, 378–382
- preScript, 378
- process, 372, 377
- propertiesPath, 378
- scriptPath, 377, 382

server

- commit, 377
- connect, 376
- keystore, 376
- p, 376

headless Ghidra. *See* headless analyzer

heap-allocated array, 157, 159

heap-allocated structures, 163

Help menu (CodeBrowser), 60

highlighting slices, 461

hijacked file (Ghidra Server), 240,
242, 243

Home scripts (Eclipse), 347, 507

hostname (Ghidra Server), 226

hotkeys

- [(Create Array), 146
- ; (Set Comment), 129–133, 297
- ALT-left arrow (backward
navigation), 95
- ALT-right arrow (forward
navigation), 95
- C (Clear Code Bytes), 141, 512
- CTRL-L (Retype), 460
- CTRL-SHIFT-Z (Redo), 120
- CTRL-Z (Undo), 120
- D (Disassemble), 142, 399, 403, 512
- E (Set Equate), 137
- editing, 61
- F (Create/Edit Function), 138,
139, 141
- F1 (Ghidra Help), 32
- F3 (Apply Selection), 548
- F5 (Program Diff toolbar), 543, 545
- G (Go To Address/Label), 94, 95, 210
- H (Label History), 125, 128
- keybinding, 61

L (Label), 120, 125–129

S (Search), 60, 72, 116, 117, 517–521

SHIFT-[(Create Structure), 167

SHIFT-C (Compare Selected
Functions), 550, 551, 554–556

T (Choose Data Type), 390

HTML format export (Ghidra), 533

I

ia.sinc, 435–442, 445–452

.idata section, 397

IDE (integrated development
environment), 338–340, 350, 351,
355, 363

IL (intermediate language), 444

ImageBase field, 391, 392

IMAGE_DOS_HEADER, 389, 390

IMAGE_NT_HEADERS, 390, 393, 394

-import (headless analyzer), 367–382

Import dialog, 280, 281, 299, 300

imported libraries (Symbol Tree), 75

importer loader poll, 388, 406

ImporterPlugin, 258

Import File (Ghidra), 44

importing

- Batch Import dialog, 299, 300
- batch mode, 230, 231, 299
- ELF files, 293, 296
- File System mode, 298
- PE files, 280

Import Results Summary, 45, 46, 282

Imports folder (Symbol Tree), 75, 297

Import Tool, 265

index (array), 150, 161, 165

inheritance (C++), 172, 173,
179–182, 486

inline constructors, 182

inline functions, 182, 483, 484

inlining (compiler variations), 483

installing Ghidra

- Installation Guide, 33, 34, 36, 37
- on Linux, 32, 37
- on macOS, 32, 37
- on Windows, 32, 34

Instruction Info window, 431

Instruction Interface

- getComment method, 329
- getMnemonicString method, 320, 329

- getNumOperands method, 320, 329
- getOperandType method, 329
- toString method, 329
- instruction patching, 518, 522, 526–529
- instruction patterns, 518–520, 522
 - Search dialog, 520, 521
- integrated development environment (IDE), 338–340, 350, 351, 355, 363
- Intel Hex format export (Ghidra), 533
- Intel x86. *See* x86
- interaction threshold, 202
- inter-function alignment gap, 530
- intermediate language (IL), 444
- intermediate representation (IR), 444
- IP address, 148
 - Ghidra Server, 226
- IR (intermediate representation), 444

J

- Java modules vs. Ghidra modules, 345
- JPyPe, 316, 317
- jump flow, 191
- jump hook, 529
- jump table, 10, 471–478
- jump to XREF (Function Graph), 204, 206
- Jython, 307, 309, 313–318

K

- kernel32.dll*, 491
 - LoadLibrary function, 498
- key bindings, 249, 250, 255, 256
 - scripts, 305, 309, 310, 318
- keystore (headless analyzer), 376

L

- Label History, 125, 128
- labels, 92, 93, 95
 - adding, 126, 127, 205
 - manipulating, 120
 - navigating, 129
 - pinned, 128
 - prefixes, 93, 120, 127
 - removing, 128
 - rules for, 124
- Labels* folder (Symbol Tree), 77
- LAB_ prefix, 93, 120, 127
- language/compiler specification, 44,

- 389, 411, 419–425
- fields, 281
- language definition file (*ldef*), 421, 422
- Language field, 281
- language generations, 4
- languages* directory, 423, 430, 433–435, 442–444
- launching headless, 366
- layouts, 171
- ldd (list dynamic dependencies), 21–24
- ldef* (language definition file), 421, 422
- libraries
 - functions, 148, 178
 - dynamically linked, 211
 - imported, 75
 - libc*, 300, 491
 - libc.a*, 298–301
 - libcrypto.so*, 283
 - lib.so.6*, 283
 - libssl.so*, 283
 - loading external, 45
 - type libraries, 149
- Library Family Name (FidDBs), 294
- Library Variant (FidDBs), 295
- Library Version (FidDBs), 295
- licenses* directory, 32, 36
- lifting, 445
- linear sweep disassembly, 9
 - example, 9–11, 13
- linker, 152
- Link options (Eclipse), 347
- Linux, 32, 37
- list dynamic dependencies (ldd), 21–24
- Listing View (Function Comparison), 574
- Listing window, 53, 456, 462, 468, 505, 509, 512, 524–527, 530–534
 - Browser Field Formatter, 67, 135, 251–253, 449
 - editing, 251
 - rearranging fields, 252
- little-endian architecture, 10
- loader (headless analyzer), 376
- loader module
 - creating modules, 401
 - example, 406–409, 417–420
 - importer poll, 388, 406
 - module template, 423
 - Raw Binary, 388–391, 400–407

- loader module (*continued*)
 - template, 351, 423
 - unknown file types, 388
- loaders, 282
 - option fields, 48
 - PE files, 45
- load external libraries, 45
- LoadLibrary function, 498
- load method, 408, 413
- loadSpecs list, 413, 418, 420
- local_ prefix, 120, 124
- local variables, 96–97, 105–108, 110, 114, 152–154, 156, 546, 567
 - identifying, 53
 - layout, 102, 103
- locating main(), 75, 490–492
- log (headless analyzer), 371
- lossy, 533

M

- Mach-O binaries, 23, 24, 28
- macOS, 32, 37
- magic files, 17
- magic number, 16, 17, 418, 424
- main(), locating, 75, 490–492
- Make Char Array, 146
- Make String, 145
- malloc (C), 157
- malware, 6, 496, 501, 502
- manipulating functions, 134, 138
- max-cpu (headless analyzer), 376
- meaningful names, 120, 125
- member functions
 - nonstatic, 173, 485
 - static, 92, 98, 102–105
- memory (search), 517
- memory allocation, 175
 - dynamic, 157, 178
- memory blocks, 88, 391–396, 408, 414
- Memory class, 324, 325
- memory footprint, 523
- memory layout
 - base address, 48
 - file offset, 48
- memory leaks, 178
- Memory Map toolbar
 - Add Block, 392, 394
 - Expand Down, 392, 398

- Expand Up, 392
- Merge Blocks, 392
- Move Block, 392, 397
- Set Image Base, 392
- Split Block tool, 395
- Memory Map window, 48, 88, 391–398
- memory search, 517–518
- Merge Blocks (Memory Map toolbar), 392
- merging analyzed files, 545
- Metasploit, 28
- Microsoft C/C++, 173, 176–178, 181, 476–479, 482, 486, 491
 - pack pragma, 160
- Microsoft x64 calling convention, 99
- MIPS instruction, 9
- mnemonics, 4, 8, 325, 521, 527
- modules
 - analyzer, 351–353
 - loader, 400, 404, 412, 423
 - processor, 428–430, 433–435, 442, 445, 451, 452
- modulo operator (compiler variations), 479–481
- monitor, 312, 323, 329–331
- Move Block (Memory Map toolbar), 392, 397
- MS-DOS header, 389

N

- name mangling (C++), 25, 26, 180, 181, 485
- names, 94, 95, 105, 108–110, 113–117, 121–128, 134–136, 146
 - manipulating, 120
 - meaningful, 120, 125
 - prefixes, 120, 124
 - rules for, 124
 - symbolic, 92
- namespace, 123–128
 - function, 124
 - global, 124
- Namespaces folder (Symbol Tree), 77
- naming conventions, Ghidra
 - decompiler, 93
- naming registers (SLEIGH), 451
- NASM (Netwide Assembler), 28
- National Security Agency (NSA), 31

- National Vulnerability Database (NVD), 317–319
- navigable objects, 187
- navigating labels, 129
- navigational target, 93
- Navigation bar, 63, 64
- navigation history, 95
- navigation marker, 63, 64
- Navigation toolbar (CodeBrowser), 72
- ndisasm, 28
- nested code layout (Function Graph toolbar), 203
- Netwide Assembler (NASM), 28
- New File Archive, 290
- new operator (C++), 175, 178
- New Project (Ghidra), 43, 226, 227
- New Project Archive, 290
- nm utility, 19, 20, 23–26
- noanalysis (headless analyzer), 368, 371, 377
- nodes (Function Graph), 199, 206–210
- NoneXecutable (NX), 352
- nonlinear flow, 66
- non-returning functions, 463
 - Non-Returning Functions – Discovered, 462
 - Non-Returning Functions – Known, 462
- nonshared projects, 32, 228, 244
- nonstatic functions, 173, 485
- NSA (National Security Agency), 31
- NX (NoneXecutable), 352

O

- O2 option (gcc), 483, 484
- obfuscation, 18, 19, 496–505, 507, 513
 - UPX packer utility, 293–297, 300
- objdump utility, 10, 24, 28, 92
- object file, 19, 526
- object life cycle (C++), 177
- object-oriented concepts, 172, 174
- OFF_ prefix, 93, 120, 127
- offsets, 150, 158–167
- okToDelete (headless analyzer), 378
- one-shot analyzers, 283, 296
- opcode, 8, 440, 449
- Open File Archive, 289

- open source, 31
- OpenSSL, 283, 288
- operating systems
 - Linux, 32, 37
 - macOS, 32, 37
 - Windows, 32, 34, 45, 501
- opinion file, 406, 423
- opinion service, 419
- optimized code, 100, 110, 479–481, 492
- Options frame (analyzers), 49
- otool utility, 23, 24, 28
- overloaded functions, 25, 179, 180
- overriding function signatures, 459
- oversized code patches, 529
- overview bar, 64
- overwrite (headless analyzer), 372

P

- p (headless analyzer), 376
- Package Explorer, 344–349, 353
- packed attribute, 160
- packet captures, 47
- pack pragma, 160
- padding bytes, 160, 171
- parameters, 148–150, 159, 179, 180, 483–485
 - renaming, 121, 460
- param_ prefix, 120, 124
- parsing C source files, 288, 289
- passwords, 234, 235
- Paste (Function Graph toolbar), 203
- patching, 516–523, 525–527, 531–533, 537
 - basic patches
 - assembler, 526
 - byte viewer, 523
 - scripting, 525
 - instructions, 518, 522, 526–529
 - oversized code patches, 529
 - patched file, 540–542, 545, 549–551
 - Patch Instruction, 526, 527, 535, 536
- p-code, 437, 444–446, 450–454
- pcodeop*, 436, 437, 445
- PDB (Program Database), 51, 54
- PE-bear, 18

- PE files, 8, 23, 530
 - base virtual address, 391, 393
 - Characteristics field, 398
 - example, 389, 393, 394, 401, 410, 411
 - format, 389
 - headers, 18
 - IMAGE_DOS_HEADER, 389, 390
 - IMAGE_NT_HEADERS, 390, 393, 394
 - importing, 280
 - loading, 45, 389, 393, 394, 401, 410, 411
 - PDB (Program Database), 51, 54
 - PE headers, 192
 - specification, 389, 393, 394, 401, 410, 411
- pinned labels, 128
- PKI certificates, 223, 226, 232, 233
- plate comments, 131, 297, 325
- Platforms (Ghidra Server), 223
- Plugin Path, 259
- plugins, 48, 256–260, 338
 - collections, 256–262
 - C-Parser, 288, 289
 - debugger, 246, 250, 256–260, 262, 502
 - dependencies, 262
 - FrontEndPlugin, 254
 - Function ID, 290, 292, 301
 - Plugin Path, 231
 - PyGhidra, 305, 307, 309, 311, 313, 315–321, 323, 325, 327, 329, 331–335
- pointer arithmetic, 172
- pointer cross-reference, 193, 194
- PointerToRawData, 394
- polymorphism (C++), 172, 180
- populating FidDb, 291–295
- popup method, 323, 330
- Portable Executable (PE) format.
 - See* PE files
- post comments, 131, 325
- postScript (headless analyzer), 378–382
- pre comments, 131, 132, 325
- Pre-Comment section, 549
- prefixes, 120, 126, 127
- preScript (headless analyzer), 378
- printf method, 323, 330–332
- println method, 312, 323
- private files (Ghidra Server), 242, 243
- private headers, 23
- process (headless analyzer), 372, 377
- processor (headless analyzer), 374, 376, 383
- processor manual, 431–433, 435
- processor modules, 428–430, 433–435, 445, 452
 - adding instructions, 435
 - adding registers, 451
- files
 - buildLanguage.xml*, 430
 - README.txt*, 429, 430, 434
 - sleighArgs.txt*, 430
- modifying instructions, 442
- processor name field, 281, 309
- processors
 - ARM, 98, 430, 444, 499
 - MIPS, 9
 - SuperH, 452
 - x86, 8, 499
- processor specification language, 35
- processor type, 281
- processor variant/mode field, 281
- Program API, 321, 325–329
- Program class, 322
 - getFunctionManager method, 327
 - getLanguageID method, 328
 - getListing method, 312, 319, 327, 330, 331
 - getMaxAddress method, 328, 329
 - getMemory method, 328
 - getMinAddress method, 328, 329, 331
 - getReferenceManager method, 328
 - getSymbolTable method, 319, 328, 332
- Program Database (PDB), 51, 54
- Program Diff
 - Diff View, 542, 552
 - tool, 540, 542, 543, 545, 549–553, 565–568
 - toolbar (hotkey F5), 543, 545
 - Apply Differences, 543, 545, 548, 549
- program entry point, 69, 75, 88, 490
- program section, 73
- program slice, 461
- program stack pointer, 97–105, 110, 509
- program text search, 116, 518

- Program Trees window, 73, 210, 530
- Project menu (Ghidra), 233, 235
- project repository, 225, 240
- projects
 - Ghidra Server, 240
 - nonshared, 32, 228, 244
 - shared, 226, 227, 236, 237, 240, 241
- prologue, 97, 101, 105, 108, 112, 399
- propertiesPath (headless analyzer), 378
- pure virtual function, 174, 175
- PyGhidra, 305, 307, 309, 311, 313, 315–321, 323, 325, 327, 329, 331–333, 335
 - example, 317–320
- python_basics.py*, 314

Q

- Quick Fix options (Eclipse), 343

R

- race condition (Ghidra Server), 225
- Raw Binary loader, 47, 388–391, 400–407, 413, 416
- read cross-reference, 193
- readelf, 24
- README files
 - analyzeHeadlessREADME.html*, 32, 366, 374, 376
 - GettingStarted.html*, 32
 - GhidraDev/README.html*, 338
 - README.txt* (processor), 429, 430, 434
 - server/svrREADME.html*, 223, 225, 232
- readOnly (headless analyzer), 372, 382
- rearranging windows, 62, 70, 246
 - Redock, 247
 - Resize, 247
 - Stack, 247
 - Undock, 247
- recognizing data structure use, 150
- recursion, 360
- recursive (headless analyzer), 372, 373, 382, 383
- recursive descent disassembly, 10–13, 142
 - example, 13
- Redo (CTRL-SHIFT-Z hotkey), 120
- Redock (display windows), 247

- Reference interface
 - getFromAddress method, 322, 332
 - getReferenceType method, 322, 331, 332
 - getToAddress method, 322, 331
- references
 - Add Reference dialog, 198
 - back, 186, 187, 197
 - cross-references, 65, 187–189, 192–198, 488, 489, 518
 - explicit forward, 197
 - formatting XREFs, 188
 - forward, 197
 - stack, 197
 - to symbols, 153
 - XREF, 66, 71, 187–195
- References To window, 196
- register-to-memory transfer instructions, 10, 13
- register transfer language (RTL), 444
- registry keys, 498, 501
- RegOpenKey, 148
- regparm, 100
- relative virtual address (RVA), 393, 396, 398
- release versions, 478
- removeFunctionAt method, 327
- removing a label, 128
- renaming parameters and variables, 121–125, 153, 460
- renaming search windows, 121, 460
- repeatable comments, 129, 132, 325
- Resize (display windows), 247
- Restore Defaults, 249
- Restore Group (Function Graph), 204
- return address, 97, 105, 107
- return instructions, 12
- retyping variables, 460
- reversing, C++, 172
- ROM images, 29
- Root Folder, 295
- ROP gadget (analyzer module example), 352, 357–359
- RTL (register transfer language), 444
- RTTI (Runtime Type Identification), 180, 485–486, 488
 - analyzer, 181
 - inheritance, 486

- Running Tools, 255
- Run options (Eclipse), 350, 360
- runtime stack, 96
- RVA (relative virtual address), 393, 396, 398

S

- s (gcc/ld), 152
- sandbox, 501
- Satellite View (Graphs), 69, 70, 201, 206, 207
- Save Tool As, 264
- saving layout changes, 253
- scripting
 - Jython, 307–309, 313–318
 - PyGhidra, 305, 307, 309, 311, 313, 315–321, 323, 325, 327, 329, 331–333, 335
- Script Manager, 340–343, 348–350
 - basic editor, 306, 309
 - Eclipse, 306, 309
 - window, 304–314, 321
- scriptPath (headless analyzer), 377, 382
- Search Memory, 117
- Search menu, 115
- Search menu (CodeBrowser), 60, 72
 - For Direct References, 492, 518
 - For Instruction Patterns, 518–520, 522
 - Search All, 520, 521
 - For Strings, 144, 284, 285
 - Memory, 117, 517
 - Program Text, 518
 - Next, 116
 - Previous, 116
 - Search All, 116
- search windows, renaming, 121, 122, 124, 125, 460
- section headers, 23
- sections
 - .bss, 73, 151, 153, 397
 - .data, 73, 151, 396–398
 - .idata, 397
 - .text, 64, 73, 394–396, 421
- sequential flow, 188, 191, 200
 - instructions, 11
- server administrator (Ghidra Server), 225

- server directory, 33, 226
- server/server.conf (Ghidra Server), 222, 224
- server/svrREADME.html, 223, 225, 232
- sessions, 569
- Set Comment, 129, 130, 133
- Set Data Type submenu, 143
 - Array option, 156
- setEOLComment method, 327
- Set Equate, 137
- Set Image Base (Memory Map toolbar), 392
- Set Language, 391
- shared projects, 227, 237, 240, 241
 - access, 234, 235
 - archiving, 230
 - authentication, 223, 226, 232–235
 - Close View, 234
 - creating, 226
 - example, 226
 - passwords, 234, 235
 - PKI, 223, 226, 232, 233
 - project information, 236
 - projects and repositories, 225, 233, 240
 - version control, 237–240, 242
 - viewing project information, 236
- shellcode, 388, 400–408, 410–418, 421–426
- Simplify predication, 455, 456
- .sinc file, 430, 435–438, 441, 442, 445, 446, 449–452
- .slaspec file, 430, 438, 454, 526
- SLEIGH, 35, 430, 431, 437, 438, 441, 445, 447, 452, 527
 - attaching variables, 448, 451
 - constructors, 434
 - Editor (Eclipse), 434
 - naming registers, 451
 - specification (.slaspec), 454, 526
 - tokens and fields, 448, 449, 451
- sleighArgs.txt, 430
- snapshot (Function Graph toolbar), 203
- Snapshot icon, 68, 82
- software interoperability, 6
- source code recovery, 5
- specific hash, 290
- specifying data types, 142
 - Create Array (hotkey []), 146
 - cycle groups, 143

- Split Block tool (Memory Map toolbar), 395
- stack, 505, 509
- Stack (display windows), 247
- stack-allocated array, 157, 162
 - example, 154–155
- stack-allocated parameters, 172
- stack-allocated structures, 162, 163
- Stack analyzer, 96
- stack-based parameters, 172
- stack frame, 53, 96, 97, 100–112
- stack frame analysis, 103
 - Decompiler Parameter ID, 50, 54
 - frame pointer, 105, 110
 - PE files, 45, 192, 280, 389, 393, 401
- stack-manipulation operations, 11
- stack pointer, 97–105, 110, 509
- stack references, 197
- stack variables, 120–123, 133
- Stack view, 104
- stale graph, 204
- standard calling convention, 97
- _start, 75, 490–492
- static analysis, 6, 11, 92, 103–109
- static array assignments, 157
- static linking, 21, 209, 210
- static storage class (C++), 179
- stdcall calling convention, 98
- storage class, 178
- strcpy function (C), 196
- stream disassemblers, 28, 29
- stream socket, 150
- String Search window, 284
- strings utility, 27, 28, 298
- stripped binary, 18, 152, 488
- structs. *See* structures
- Structure Editor window, 170, 465
- structure pointers, 171
- structures
 - applying structure layouts, 171
 - arrays of, 165
 - Byte Offset, 171
 - Component Bits, 171
 - creating, 161, 166–168
 - decompiled, 161
 - disassembled, 158, 161, 163, 165
 - editing, 169–170
 - field access, 163, 164
 - field alignment, 160, 171
 - globally allocated, 161, 168
 - heap-allocated, 163–165
 - layouts, 164, 171
 - member access, 160, 172
 - offsets, 150, 153, 158–167, 171, 172
 - recognizing use, 150
 - size, 164
 - stack-allocated, 162, 163
 - starting address, 153, 158, 163
 - Structure Editor window, 170
 - symbolic references, 150, 153, 158–167, 171, 172
 - Union Editor window, 167
 - within structures, 165
- SUB_ prefix, 127
- superclasses, 176
- SuperH4.sinc*, 452
- support* directory, 36, 37
- support documentation, 32, 33, 35, 37
- svrAdmin, 225
- switch statement, compiler variations, 470–478
- symbolic names, 92
- Symbol Interface
 - getAddress method, 319, 320, 322, 332
 - getName method, 312, 319–322, 329–332
- Symbol option (annotations), 153
- Symbol References window, 85–88, 196
- symbols, renaming, 121, 122, 124, 125, 153
- symbol table, 23, 53, 152
- Symbol Table window, 85–88, 196
- Symbol Tree window, 53, 60, 74–76, 92, 94, 148–150, 181, 291–293
 - Classes* folder, 77, 195, 486, 487
 - Exports* folder, 75
 - Functions* folder, 76, 77
 - imported libraries, 75
 - Imports* folder, 75, 297
 - Labels* folder, 77
 - Namespaces* folder, 77
- syntax (headless analyzer), 367
- system call, 100, 101
- System V AMD64 ABI, 99, 101

T

- table lookup, 470, 471
- Table View (Ghidra Project window), 227, 228
- targets (navigational), 93
- TaskMonitor, 323
- task tag (Eclipse), 342, 355
- ternary operator (compiler variations), 481, 482
- testing modules (Eclipse), 410
- .text section, 64, 73, 394–396, 421, 530
- themes, 255
- third-party components, 32, 36
- this pointer, 173, 176, 181
- thunk function, 211
- tip of the day, 42
- toAddr method, 324, 334, 335
- TODD comments, 342, 343, 355, 436
- Toggle Overview Margin, 64
- token (SLEIGH), 448, 449, 451
- Tool Chest, 255, 259, 264, 265, 567
- Tool Options
 - Color Editor, 248, 249
 - Key Bindings, 255–256
 - Restore Defaults, 249
 - Tool, 255–256
 - Window, 55, 248, 251
- tools (external)
 - c++filt utility, 25, 26
 - Detect-It-Easy, 18, 23
 - dumpbin utility, 23–25, 28
 - file utility, 16, 17
 - ldd utility, 21–24
 - nm utility, 19, 20, 23–26
 - objdump utility, 10, 24, 28, 92
 - otool utility, 23, 24, 28
 - PE-bear, 18
 - strings utility, 27, 28
 - WinDbg, 10
- tools (Ghidra), 246
 - connecting Ghidra tools, 69
 - Program Diff, 540, 542, 543, 545, 549–553, 556, 559, 565–568
 - Running Tools, 255
 - Tool Chest, 255, 259, 264, 265, 567
 - VBinDiff, 537, 540
- Tools menu (CodeBrowser), 60, 67, 69

Tools menu (Ghidra Project)

- Create, 259
- Save Tool As, 264

toString method, 329

tutorials

- Eclipse, 339
- Ghidra, 33
- Python scripting, 314

typeid, 180, 181, 485

type libraries, 149

U

Ultimate Packer for eXecutables.

- See* UPX

unconditional branching

- instructions, 11

undefined data, 518

Undo (CTRL-Z hotkey), 120

Undock (display windows), 247

Ungroup Vertices (Function Graph), 204

union construct, 167

Union Editor window, 167

unknown file analysis, 388

unknown file formats, 384, 400

unknown processor architectures, 384

UNK_ prefix, 93, 120, 127

unpack, 506, 507, 510–512

unpatched file, 541, 545, 550, 551

UPX

- example, 293
- packer, 293–297

user agreement, 32

\$USER_HOME directory, 377

utilization rate, 471

V

validateOptions method, 409

variable indices, 153

variable number of arguments, 98, 459

variables

- global, 152
- layout, 102, 103
- local, 96–97, 102, 103, 105–108, 110, 114, 152–154, 156
- renaming, 121, 122, 124, 125, 153

VBinDiff, 537, 540

version control, 237–240, 242

- version tracking, 237
- Version Tracking tool, 540
 - correlators, 567–569
 - footprints icon, 567
 - sessions, 569
- vertices, 186
- vftables (C++), 174–182, 194–196, 486–489
 - indexing, 176
 - pointer, 174–179
- View Project (Ghidra Project), 236
- View Recent (Ghidra Project), 234
- View Repository (Ghidra Project), 233
- virtual functions (C++), 173–176, 194, 197
- virtualization, 501
- virtual machine (VM), 497, 499, 502
- virtual machine extension (VMX), 434–437
- Visual Studio (VS) Code, 337
- VMware, 497
- vulnerabilities, 5, 515, 522
- vulnerability analysis, 6

W

- wildcards, 521
- WinDbg, 10
- Windows
 - API, 169
 - registry, 148
 - registry keys, 148, 498, 501
 - WSL (Windows Subsystem for Linux), 16, 24

- word models, 144, 284, 285
- write cross-reference, 193
- WSL (Windows Subsystem for Linux), 16, 24

X

- x86, 98, 101, 102, 499
 - instruction set, 8, 28
 - processor files
 - ia.sinc*, 435–442, 445–452
 - x86-64.sla*, 441, 443
 - x86-64.slaspec*, 438
 - x86.idx*, 432
 - x86.slaspec*, 438, 439
- XML format export (Ghidra), 533
- XREF, 66, 71, 187–195
- XREFs Field edit window, 188
- XRefs Window, 195

Z

- Zip format export (Ghidra), 533
- zooming (Function Graph View), 60, 62, 70

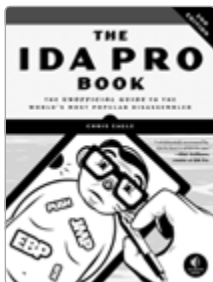
RESOURCES

Visit <https://nostarch.com/ghidra-book-2e> for errata and more information.

More no-nonsense books from

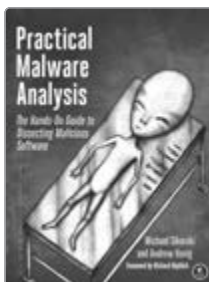


NO STARCH PRESS



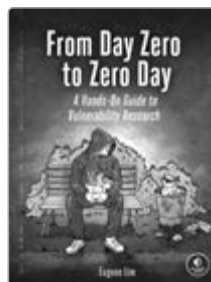
THE IDA PRO BOOK, 2ND EDITION **The Unofficial Guide to the World's Most Popular Disassembler**

BY CHRIS EAGLE
672 PP., \$69.95
ISBN 978-1-59327-289-0



PRACTICAL MALWARE ANALYSIS **The Hands-On Guide to Dissecting Malicious Software**

BY MICHAEL SIKORSKI AND
ANDREW HONIG
800 PP., \$59.99
ISBN 978-1-59327-290-6



FROM DAY ZERO TO ZERO DAY **A Hands-On Guide to Vulnerability Research**

BY EUGENE LIM
344 PP., \$59.99
ISBN 978-1-7185-0394-6



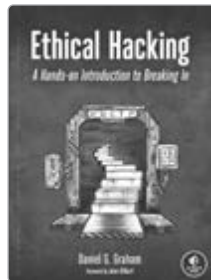
EVASIVE MALWARE **A Field Guide to Detecting, Analyzing, and Defeating Advanced Threats**

BY KYLE CUCCI
488 PP., \$69.99
ISBN 978-1-7185-0326-7



EVADING EDR **The Definitive Guide to Defeating Endpoint Detection Systems**

BY MATT HAND
312 PP., \$59.99
ISBN 978-1-7185-0334-2

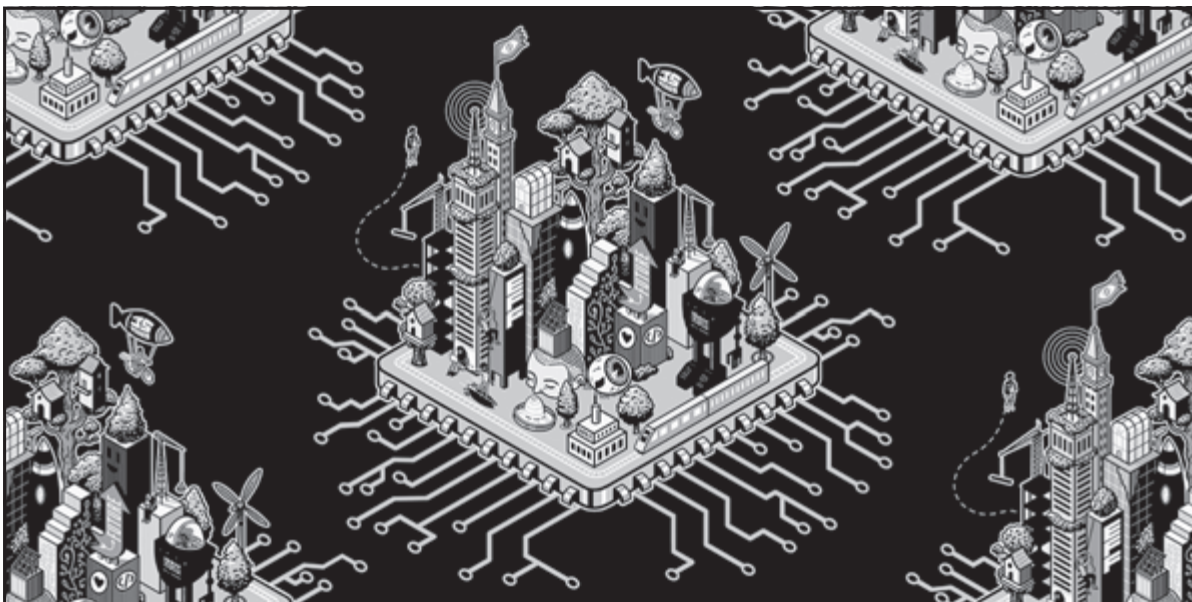


ETHICAL HACKING **A Hands-on Introduction to Breaking In**

BY DANIEL G. GRAHAM
376 PP., \$49.99
ISBN 978-1-7185-0187-4

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM
WEB:
WWW.NOSTARCH.COM



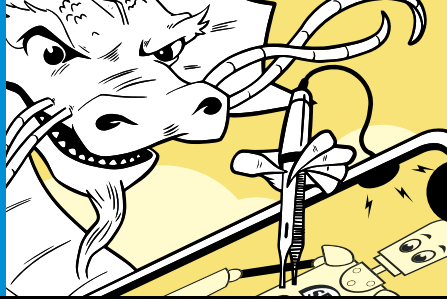
Never before has the world relied so heavily on the internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people—more urgent than ever.

For over 35 years, EFF has fought for your rights through activism, in the courts, and by developing software because we believe in a better future—one where your device is truly yours, you can speak without being surveilled, and technology helps you connect with the people you care about. With your help, we can realize that vision for a brighter world together.



LEARN MORE AND JOIN EFF AT EFF.ORG/NOSTARCH

DANCE WITH THE DRAGON



**"A must-read
for any reverse
engineer."**

—MAX "LIBRA" KERSTEN

Developed through more than a decade of research within the NSA, Ghidra was created to address some of the most challenging reverse engineering problems faced by the US government. With its open source release, this formerly restricted tool suite is now freely available to cybersecurity practitioners, researchers, and students worldwide. *The Ghidra Book* provides a clear, practical guide to understanding and using this powerful platform.

In addition to introducing core reverse engineering techniques for software and malware analysis, the book explains Ghidra's key components, features, and support for extensibility and collaborative analysis. Beginning with the fundamentals and progressing to more advanced workflows, you'll learn how to use Ghidra effectively and adapt it to new challenges.

You'll learn how to:

- Navigate and interpret a disassembly
- Use Ghidra's built-in decompiler to expedite analysis
- Analyze unfamiliar and obfuscated binaries
- Extend Ghidra to recognize new data types
- Build custom analyzers, loaders, and processor support modules

- Script Ghidra tasks to automate analysis tasks
- Set up a collaborative reverse engineering environment

This fully revised second edition reflects the modern Ghidra platform as it is used in practice, with updated tooling, improved workflows, and hands-on real-world examples. Designed for beginners and experienced users alike, *The Ghidra Book* prepares you to tackle real reverse engineering problems with confidence.

New to this edition: Behavioral analysis with BSim • Full Python 3 support via PyGhidra
• Enhanced debugging and graphing tools • Modern container-based deployment

ABOUT THE AUTHORS

Dr. Kara Nance is a private security consultant and has been a professor of computer science for many years. She regularly speaks at conferences around the world and enjoys building Ghidra extensions as well as providing Ghidra training.

Chris Eagle has been reverse engineering software for 45 years. He is the author of *The IDA Pro Book* (No Starch Press) and is a highly sought-after provider of reverse engineering training.



THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com