Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №1

Курс: «Проектирование ОС и компонентов»

Тема: «Системные вызовы»

Выполнил студент:

Бояркин Никита Сергеевич
Группа: 13541/3

Проверил:

Душутина Елена Владимировна

Санкт-Петербург
2018 г.

Содержание

Лабораторная работа №1

1.1 Цель работы

Ознакомиться с системными вызовами, научиться разрабатывать перехватчики для системных вызывов и вносить изменения в ядро.

1.2 Программа работы

Для каждого системного вызова из индивидуального задания:

- Привести описание.
- Разработать программу с примером использования.
- Осуществить перехват вызова черех хук.
- Проанализировать исходный код вызова.

1.3 Индивидуальное задание

Системные вызовы:

- sys_getpid
- sys_setpid
- sys_chdir
- sys_sysinfo

1.4 Характеристики системы

```
1 nikita@nikita-VirtualBox:~$ who
2 nikita    tty7         2018-05-07 02:57 (:0)
3 nikita@nikita-VirtualBox:~$ cat /proc/version
4 Linux version 4.4.0-121-generic (buildd@lcy01-amd64-004) (gcc version 5.4.0 20160609 (
    Ubuntu 5.4.0-6ubuntu1~16.04.9) ) #145-Ubuntu SMP Fri Apr 13 13:47:23 UTC 2018
```

1.5 Системные вызовы

В конечном итоге, главная задача операционной системы — это обслуживание потребностей прикладных пользовательских процессов. И обеспечивается это обслуживание через механизм системных вызовов.

В любой (в том числе и микроядерной) операционной системе системный вызов выполняется некоторой выделенной процессорной инструкцией, прерывающей последовательное выполнение команд и передающий управление коду режима супервизора. Это обычно некая команда программного прерывания, в зависимости от архитектуры процессора в разные времена это были команды с мнемониками вида: svc, emt, trap, int и им подобными [1]

Команды прерывания для ОС, построенных на архитектуре Intel x86:

OS	desc
MS-DOS	21h
Windows	2Eh
Linux	80h
QNX	21h
MINIX 3	21h

Прикладной процесс вызывает требуемые ему службы посредством библиотечного вызова к множеству библиотек либо вида *.so (динамическое связывание), либо прикомпоновывая к себе фрагмент из библиотеки вида *.a (статическое связывание). Самые известные примеры - это стандартная библиотека языка C libc.so или libpthread.so — библиотека POSIX потоков [1]

Описания системных вызовов (в отличие от библиотечных) отнесены к секции 2 в руководствах man. Все системные вызовы далее преобразуются в вызов ядра функцией syscall(), 1-м параметром которого будет идентификатор выполняемого системного вызова, например __NR_execve.

Найдем описания заданных системных вызовов в таблице векторов системных вызовов syscall_32.tbl:

```
# linux/arch/x86/entry/syscalls/syscall_32.tbl
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#

< ... >


12  i386  chdir     sys_chdir      __ia32_sys_chdir

< ... >

20  i386  getpid       sys_getpid       __ia32_sys_getpid

< ... >

116 i386  sysinfo      sys_sysinfo      __ia32_compat_sys_sysinfo

< ... >
```

1.5.1 Перехват системных вызовов

Для перехвата системных вызовов воспользуемся библиотекой kmod_hooking [2] Данная библиотека содержит набор системных функций для регистрации и снятия перехватчиков с заданных функций. Принцип работы основывается на подмене функции внутри ядра. Синтаксис перехватчика иллюстрируется следующим примером перехвата системной функции inode_permission:

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

< ... >

DECLARE_KHOOK(inode_permission);
int khook_inode_permission(struct inode * inode, int mode)
{
  int result;

  KHOOK_USAGE_INC(inode_permission);

  debug("%s(%pK,%08x) [%s]\n", __func__, inode, mode, current->comm);

  result = KHOOK_ORIGIN(inode_permission, inode, mode);

  debug("%s(%pK,%08x) [%s] = %d\n", __func__, inode, mode, current->comm, result);
```

```
17
18    KHOOK_USAGE_DEC(inode_permission);
19
20    return result;
21 }
22
23 < ... >
```

DECLARE_KHOOK указывает на то, какие функции являются перехватчиками. KHOOK_USAGE_INC и KHOOK_USAGE_DEC определяют область, в которой выполняется хук. KHOOK_ORIGIN вызывает непосредственно системный вызов с заданными аргументами.

Добавление собственного кода в ядро производится с помощью модулей. Модули располагаются в директории /lib/modules/$(uname -r). Сборка библиотеки производится при помощи следующего makefile:

```
1  NAME    := hooking
2
3  obj-m   := $(NAME).o
4  obj-y   := libudis86/
5
6  ldflags-y := -T$(src)/layout.lds
7
8  $(NAME)-y := module-init.o libudis86/built-in.o
9
10 all:
11   make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)
12
13 clean:
14   make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Добавление нового модуля в систему производится через утилиту insmod:

```
1 nikita@nikita-VirtualBox:~/files/student/temp/kmod_hooking$ sudo insmod hooking.ko
```

Удаление модуля из систему производится через утилиту rmmod:

```
1 nikita@nikita-VirtualBox:~/files/student/temp/kmod_hooking$ sudo rmmod hooking
```

1.5.2 Системный вызов sys_getpid

Системный вызов sys_getpid() возвращает идентификатор текущего процесса. Пользовательская функция getpid() предоставляет доступ к данному системному вызову и объявляется в заголовочном файле unistd.h.

Пример использования

Функция не принимает никаких аргументов и возвращает целочисленный идентификатор процесса:

```
1 #include <iostream>
2 #include <unistd.h>
3
4 int main() {
5     std::cout << "Process PID: " << getpid() << std::endl;
6     return 0x0;
7 }
```

Результат выполнения программы:

```
1 nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ ./executable &
2 [1] 9225
3 Process PID: 9225
4 [1]+  Done                    ./executable
```

Отслеживание системного вызова

Попробуем отследить системный вызов в предыдущей программе через утилиту strace:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ strace ./executable
execve("./executable", ["./executable"], [/* 74 vars */]) = 0

< ... >

brk(NULL)                               = 0x1418000
brk(0x144a000)                          = 0x144a000
getpid()                                = 9225
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 19), ...}) = 0
write(1, "Process PID: 9225\n", 18Process PID: 9225
)      = 18
exit_group(0)                           = ?
+++ exited with 0 +++
```

После создания процесса вызывается функция getpid и выводится на экран функцией write, после чего программа завершается.

Перехват системного вызова

Реализуем перехват системного вызова sys_getpid с помощью kmod_hooking:

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

< ... >

DECLARE_KHOOK(sys_getpid);
long khook_sys_getpid()
{
  KHOOK_USAGE_INC(sys_getpid);

  long result = KHOOK_ORIGIN(sys_getpid);
  printk("Syscall 'sys_getpid' hooked. Result: %d.\n", result) ;

  KHOOK_USAGE_DEC(sys_getpid);

  return result;
}

< ... >
```

Данный код добавляется в linux как модуль ядра, а результат работы журналируется в файл kern.log. Убедимся в правильности работы хука:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ ./executable &
[1] 9431
Process PID: 9431
[1]+  Done                    ./executable
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ tail /var/log/kern.log
May  3 06:10:55 nikita-VirtualBox kernel: [11609.557046] raid6: sse2x2   xor()  8688 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.625101] raid6: sse2x4   gen() 15389 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692700] raid6: sse2x4   xor() 10589 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692702] raid6: using algorithm sse2x4
    gen() 15389 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692703] raid6: .... xor() 10589 MB/s,
    rmw enabled
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692704] raid6: using ssse3x2 recovery
    algorithm
May  3 06:10:56 nikita-VirtualBox kernel: [11609.693802] xor: automatically using best
    checksumming function:
May  3 06:10:56 nikita-VirtualBox kernel: [11609.732719]    avx       : 20010.000 MB/sec
May  3 06:10:56 nikita-VirtualBox kernel: [11609.749930] Btrfs loaded
May  3 08:20:33 nikita-VirtualBox kernel: [19386.834543] Syscall 'sys_getpid' hooked.
    Result: 9431.
```

После запуска программы в kern.log появилась запись о попытке доступа к системному вызову sys_getpid.

Анализ исходного кода

Прототип системного вызова sys_getpid находится в include/linux/syscalls.h в исходных файлах ядра linux [3]:

```
// include/linux/syscalls.h

asmlinkage long sys_getpid(void);
```

Непосредственно реализация sys_getpid находится в kernel/sys.c в исходных файлах ядра linux [4]:

```
// kernel/sys.c

SYSCALL_DEFINE0(getpid)
{
    return task_tgid_vnr(current);
}
```

Можно заметить, что вызывается функция ядра task_tgid_vnr, которая в свою очередь вызывает __task_pid_nr в kernel/pid.c в исходных файлах ядра linux [5]:

```
// kernel/pid.c

pid_t __task_pid_nr_ns(struct task_struct *task, enum pid_type type,
        struct pid_namespace *ns)
{
  pid_t nr = 0;

  rcu_read_lock();
  if (!ns)
    ns = task_active_pid_ns(current);
  if (likely(pid_alive(task))) {
    if (type != PIDTYPE_PID) {
      if (type == __PIDTYPE_TGID)
        type = PIDTYPE_PID;

      task = task->group_leader;
    }
    nr = pid_nr_ns(rcu_dereference(task->pids[type].pid), ns);
  }
  rcu_read_unlock();

  return nr;
}
```

На время работы функции блокируется чтение, проверяется наличие текущего процесса и наличие gpid, после чего вызывается функция pid_nr_ns:

```
// kernel/pid.c

pid_t pid_nr_ns(struct pid *pid, struct pid_namespace *ns)
{
  struct upid *upid;
  pid_t nr = 0;

  if (pid && ns->level <= pid->level) {
    upid = &pid->numbers[ns->level];
    if (upid->ns == ns)
      nr = upid->nr;
  }
  return nr;
}
```

Если оба аргумента (структуры pid и pid_namespace) корректны, то возвращается pid текущего процесса, в противном случае возвращается 0.

1.5.3 Системный вызов sys_setpid

Такого системного вызова, очевидно, не существует, потому что идентификатор процесса задается единожды при запуске процесса.

Для выделения нового pid используется функция alloc_pid:

```
// kernel/pid.c

struct pid *alloc_pid(struct pid_namespace *ns)
{
    < ... >

    pid = kmem_cache_alloc(ns->pid_cachep, GFP_KERNEL);
    if (!pid)
        return ERR_PTR(retval);

    tmp = ns;
  pid->level = ns->level;

  for (i = ns->level; i >= 0; i--) {
    int pid_min = 1;

    idr_preload(GFP_KERNEL);
    spin_lock_irq(&pidmap_lock);

    /*
     * init really needs pid 1, but after reaching the maximum
     * wrap back to RESERVED_PIDS
     */
    if (idr_get_cursor(&tmp->idr) > RESERVED_PIDS)
      pid_min = RESERVED_PIDS;

    /*
     * Store a null pointer so find_pid_ns does not find
     * a partially initialized PID (see below).
     */
    nr = idr_alloc_cyclic(&tmp->idr, NULL, pid_min,
              pid_max, GFP_ATOMIC);
    spin_unlock_irq(&pidmap_lock);
    idr_preload_end();

    if (nr < 0) {
      retval = nr;
      goto out_free;
    }

    pid->numbers[i].nr = nr;
    pid->numbers[i].ns = tmp;
    tmp = tmp->parent;
  }

    < ... >
}
```

Функция вызывает kmem_cache_alloc с аргументом в виде набора кэшированных структур для выделения памяти под структуру нового pid [6].

Далее происходит инициализация полей структуры в зависимости от уровня аргумента pid_namespace. Функция idr_alloc_cyclic пытается выделить новый идентификатор процесса. Если получилось, то идентификатор записывается в структуру.

1.5.4 Системный вызов sys_chdir

Системный вызов sys_chdir() задает текущий путь выполнения процесса. Пользовательская функция chdir() предоставляет доступ к данному системному вызову и объявляется в заголовочном файле unistd.h.

Пример использования

Для примера выведем текущий путь выполнения процесса, поменяем его функцией chdir() и выведем путь еще раз. Путь должен измениться на аргумент функции chdir():

```
#include <iostream>
#include <unistd.h>

static const int DIRECTORY_PATH_SIZE = 1024;
static const char* DIRECTORY_PATH = "/home";

static char currentDirectory[DIRECTORY_PATH_SIZE];

const char* getCurrentDirectory();

int main() {
    std::cout << "Current directory: " << getCurrentDirectory() << std::endl;
    chdir(DIRECTORY_PATH);
    std::cout << "Current directory after chdir: " << getCurrentDirectory() << std::endl;
    return 0x0;
}

const char* getCurrentDirectory() {
    char* result = getcwd(currentDirectory, DIRECTORY_PATH_SIZE);

    if (result == NULL) {
        return NULL;
    }

    return currentDirectory;
}
```

Результат выполнения программы:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ ./executable
Current directory: /home/nikita/files/student/ezsem4/sys/1/listings
Current directory after chdir: /home
```

Отслеживание системного вызова

Попробуем отследить системный вызов в предыдущей программе через утилиту strace:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ strace ./executable
execve("./executable", ["./executable"], [/* 74 vars */]) = 0

< ... >

getcwd("/home/nikita/files/student/ezsem4/sys/1/listings", 1024) = 49
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 19), ...}) = 0
write(1, "Current directory: /home/nikita/"..., 68Current directory: /home/nikita/files/
    student/ezsem4/sys/1/listings
) = 68
chdir("/home")                          = 0
getcwd("/home", 1024)                   = 6
write(1, "Current directory after chdir: /"..., 37Current directory after chdir: /home
) = 37
exit_group(0)                           = ?
+++ exited with 0 +++
```

Как и ожидалось, была вызвана цепочка системных вызовов: getcwd -> write -> chdir -> getcwd -> write, в результате которой изменилось значение путь выполнения процесса.

Перехват системного вызова

Реализуем перехват системного вызова sys_chdir с помощью kmod_hooking:

```c
#include <linux/syscalls.h>

< ... >

DECLARE_KHOOK(sys_chdir);
long khook_sys_chdir(const char __user *filename)
{
  KHOOK_USAGE_INC(sys_chdir);

  long result = KHOOK_ORIGIN(sys_chdir, filename);
  printk("Syscall 'sys_chdir' hooked. Arguments: %s. Result: %d.\n", filename, result);

  KHOOK_USAGE_DEC(sys_chdir);

  return result;
}

< ... >
```

Данный код добавляется в linux как модуль ядра, а результат работы журналируется в файл kern.log. Убедимся в правильности работы хука:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ ./executable
Current directory: /home/nikita/files/student/ezsem4/sys/1/listings
Current directory after chdir: /home
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ tail /var/log/kern.log
May  3 06:10:56 nikita-VirtualBox kernel: [11609.625101] raid6: sse2x4   gen() 15389 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692700] raid6: sse2x4   xor() 10589 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692702] raid6: using algorithm sse2x4
    gen() 15389 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692703] raid6: .... xor() 10589 MB/s,
    rmw enabled
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692704] raid6: using ssse3x2 recovery
    algorithm
May  3 06:10:56 nikita-VirtualBox kernel: [11609.693802] xor: automatically using best
    checksumming function:
May  3 06:10:56 nikita-VirtualBox kernel: [11609.732719]    avx       : 20010.000 MB/sec
May  3 06:10:56 nikita-VirtualBox kernel: [11609.749930] Btrfs loaded
May  3 08:20:33 nikita-VirtualBox kernel: [19386.834543] Syscall 'sys_getpid' hooked.
    Result: 9431.
May  3 08:21:32 nikita-VirtualBox kernel: [19445.923432] Syscall 'sys_chdir' hooked.
    Arguments: /home. Result: 0.
```

После запуска программы в kern.log появилась запись о попытке доступа к системному вызову sys_chdir.

Анализ исходного кода

Прототип системного вызова sys_chdir находится в include/linux/syscalls.h в исходных файлах ядра linux [3]:

```c
// include/linux/syscalls.h

asmlinkage long sys_chdir(const char __user *filename);
```

Непосредственно реализация sys_chdir находится в kernel/sys.c в исходных файлах ядра linux [4]:

```c
// kernel/sys.c

SYSCALL_DEFINE1(chdir, const char __user *, filename)
{
    return ksys_chdir(filename);
}
```

Можно заметить, что вызывается функция ядра ksys_chdir в fs/open.c в исходных файлах ядра linux [7]:

```c
// fs/open.c

```

```
int ksys_chdir(const char __user *filename)
{
  struct path path;
  int error;
  unsigned int lookup_flags = LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
retry:
  error = user_path_at(AT_FDCWD, filename, lookup_flags, &path);
  if (error)
    goto out;

  error = inode_permission(path.dentry->d_inode, MAY_EXEC | MAY_CHDIR);
  if (error)
    goto dput_and_out;

  set_fs_pwd(current->fs, &path);

dput_and_out:
  path_put(&path);
  if (retry_estale(error, lookup_flags)) {
    lookup_flags |= LOOKUP_REVAL;
    goto retry;
  }
out:
  return error;
}
```

Функция проверяет наличие задаваемого пути и наличие прав на его изменение. После этого вызывается функция set_fs_pwd, которая непосредственно задает путь:

```
// fs/open.c

void set_fs_pwd(struct fs_struct *fs, const struct path *path)
{
  struct path old_pwd;

  path_get(path);
  spin_lock(&fs->lock);
  write_seqcount_begin(&fs->seq);
  old_pwd = fs->pwd;
  fs->pwd = *path;
  write_seqcount_end(&fs->seq);
  spin_unlock(&fs->lock);

  if (old_pwd.dentry)
    path_put(&old_pwd);
}
```

1.5.5 Системный вызов sys_sysinfo

Системный вызов sys_sysinfo() выводит системную информацию, такую как: время с момента запуска системы в секундах, количество памяти и количество свободно памяти в байтах, количество активных процессов, размер свфпа и др. Пользовательская функция sysinfo() предоставляет доступ к данному системному вызову и объявляется в заголовочном файле sys/sysinfo.h.

Пример использования

Для примера выведем системную информацию в удобном для чтения формате:

```
#include <iostream>
#include <sys/sysinfo.h>

static const int MEGABYTE_IN_BYTES = 1024 * 1024;
static const int MINUTE_IN_SECONDS = 60;
static const int HOUR_IN_SECONDS = MINUTE_IN_SECONDS * 60;
static const int DAY_IN_SECONDS = HOUR_IN_SECONDS * 24;

```

```
int main() {
    struct sysinfo information;
    sysinfo(&information);

    std::cout << "Time since boot: days " << information.uptime / DAY_IN_SECONDS << ", "
                                          << (information.uptime % DAY_IN_SECONDS) /
    HOUR_IN_SECONDS << ":"
                                          << (information.uptime % HOUR_IN_SECONDS) /
    MINUTE_IN_SECONDS << ":"
                                          << information.uptime % MINUTE_IN_SECONDS <<
    std::endl;

    std::cout << "Total RAM: " << information.totalram / MEGABYTE_IN_BYTES << " Mb" <<
    std::endl;
    std::cout << "Free RAM: " << information.freeram / MEGABYTE_IN_BYTES << " Mb" << std
    ::endl;
    std::cout << "Process count: " << information.procs << std::endl;
    return 0x0;
}
```

Результат выполнения программы:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ ./executable
Time since boot: days 0, 14:42:23
Total RAM: 3951 Mb
Free RAM: 605 Mb
Process count: 631
```

Отслеживание системного вызова

Попробуем отследить системный вызов в предыдущей программе через утилиту strace:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ strace ./executable
execve("./executable", ["./executable"], [/* 74 vars */]) = 0

< ... >

sysinfo({uptime=53030, loads=[31328, 29824, 20160], totalram=4142927872, freeram
    =628195328, sharedram=67964928, bufferram=390832128, totalswap=4292866048, freeswap
    =4292866048, procs=633, totalhigh=0, freehigh=0, mem_unit=1}) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 19), ...}) = 0
write(1, "Time since boot: days 0, 14:43:5"..., 34Time since boot: days 0, 14:43:50
) = 34
write(1, "Total RAM: 3951 Mb\n", 19Total RAM: 3951 Mb
)    = 19
write(1, "Free RAM: 599 Mb\n", 17Free RAM: 599 Mb
)      = 17
write(1, "Process count: 633\n", 19Process count: 633
)    = 19
exit_group(0)                           = ?
+++ exited with 0 +++
```

Как и ожидалось была получена структура sysinfo через соответствующий системный вызов.

Перехват системного вызова

Реализуем перехват системного вызова sys_sysinfo с помощью kmod_hooking:

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

< ... >

DECLARE_KHOOK(sys_sysinfo);
long khook_sys_sysinfo(struct sysinfo __user *info)
{
  KHOOK_USAGE_INC(sys_sysinfo);

```

```
  long result = KHOOK_ORIGIN(sys_sysinfo, info);
  printk("Syscall 'sys_sysinfo' hooked. Arguments: %ld, %ld, %ld, %d. Result: %d.\n",
    info->uptime, info->totalram, info->freeram, info->procs, result);

  KHOOK_USAGE_DEC(sys_sysinfo);

  return result;
}

< ... >
```

Данный код добавляется в linux как модуль ядра, а результат работы журналируется в файл kern.log. Убедимся в правильности работы хука:

```
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ ./executable
Time since boot: days 0, 5:24:33
Total RAM: 3951 Mb
Free RAM: 208 Mb
Process count: 644
nikita@nikita-VirtualBox:~/files/student/ezsem4/sys/1/listings$ tail /var/log/kern.log
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692700] raid6: sse2x4   xor() 10589 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692702] raid6: using algorithm sse2x4
    gen() 15389 MB/s
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692703] raid6: .... xor() 10589 MB/s,
    rmw enabled
May  3 06:10:56 nikita-VirtualBox kernel: [11609.692704] raid6: using ssse3x2 recovery
    algorithm
May  3 06:10:56 nikita-VirtualBox kernel: [11609.693802] xor: automatically using best
    checksumming function:
May  3 06:10:56 nikita-VirtualBox kernel: [11609.732719]    avx       : 20010.000 MB/sec
May  3 06:10:56 nikita-VirtualBox kernel: [11609.749930] Btrfs loaded
May  3 08:20:33 nikita-VirtualBox kernel: [19386.834543] Syscall 'sys_getpid' hooked.
    Result: 9431.
May  3 08:21:32 nikita-VirtualBox kernel: [19445.923432] Syscall 'sys_chdir' hooked.
    Arguments: /home. Result: 0.
May  3 08:22:01 nikita-VirtualBox kernel: [19473.324429] Syscall 'sys_sysinfo' hooked.
    Arguments: 19473, 4142923776, 218103808, 644. Result: 0.
```

После запуска программы в kern.log появилась запись о попытке доступа к системному вызову sys_sysinfo.

Анализ исходного кода

Прототип системного вызова sys_sysinfo находится в include/linux/syscalls.h в исходных файлах ядра linux [3]:

```
// include/linux/syscalls.h

asmlinkage long sys_sysinfo(struct sysinfo __user *info);
```

Непосредственно реализация sys_sysinfo находится в kernel/sys.c в исходных файлах ядра linux [4]:

```
// kernel/sys.c

SYSCALL_DEFINE1(sysinfo, struct sysinfo __user *, info)
{
    struct sysinfo val;

    do_sysinfo(&val);

    if (copy_to_user(info, &val, sizeof(struct sysinfo)))
        return -EFAULT;

    return 0;
}
```

Можно заметить, что вызывается функция ядра do_sysinfo в kernel/sys.c в исходных файлах ядра linux [4]:

```c
// kernel/sys.c

static int do_sysinfo(struct sysinfo *info)
{
  unsigned long mem_total, sav_total;
  unsigned int mem_unit, bitcount;
  struct timespec tp;

  memset(info, 0, sizeof(struct sysinfo));

  get_monotonic_boottime(&tp);
  info->uptime = tp.tv_sec + (tp.tv_nsec ? 1 : 0);

  get_avenrun(info->loads, 0, SI_LOAD_SHIFT - FSHIFT);

  info->procs = nr_threads;

  si_meminfo(info);
  si_swapinfo(info);

  /*
   * If the sum of all the available memory (i.e. ram + swap)
   * is less than can be stored in a 32 bit unsigned long then
   * we can be binary compatible with 2.2.x kernels.  If not,
   * well, in that case 2.2.x was broken anyways...
   *
   *  -Erik Andersen <andersee@debian.org>
   */

  mem_total = info->totalram + info->totalswap;
  if (mem_total < info->totalram || mem_total < info->totalswap)
    goto out;
  bitcount = 0;
  mem_unit = info->mem_unit;
  while (mem_unit > 1) {
    bitcount++;
    mem_unit >>= 1;
    sav_total = mem_total;
    mem_total <<= 1;
    if (mem_total < sav_total)
      goto out;
  }

  /*
   * If mem_total did not overflow, multiply all memory values by
   * info->mem_unit and set it to 1.  This leaves things compatible
   * with 2.2.x, and also retains compatibility with earlier 2.4.x
   * kernels...
   */

  info->mem_unit = 1;
  info->totalram <<= bitcount;
  info->freeram <<= bitcount;
  info->sharedram <<= bitcount;
  info->bufferram <<= bitcount;
  info->totalswap <<= bitcount;
  info->freeswap <<= bitcount;
  info->totalhigh <<= bitcount;
  info->freehigh <<= bitcount;

out:
  return 0;
}
```

В первую очередь структура обнуляется, после чего собирается информация о времени, памяти, свопе и количестве процессов через соответствующие функции get_monolitic_boottime, get_averrun, si_meminfo,

si_swapinfo. В завершении делается поправка на версии ядра 2.2.x с побитовым сдвигом полей результирующей структуры.

1.6 Вывод

В данной работе были изучены системные вызовы, принципы перехвата системных вызовов, а также добавление модулей в ядро ОС Linux.

В ОС Linux(версия ядра v4.17-rc4) определено около 384 системных вызовов для архитектуры i386. Такое количество системных вызовов позволяет программисту получить удобный и полный доступ ко всем компонентам ОС в рамках прав доступа.

Каждый системный вызов содержит элементы синхронизации там где это необходимо для корректной работы в условиях многопоточности.

Каждый разработчик сообщества может улучшать исходный код системных вызовов и других частей ядра путем создания pull request в GitHub репозитории ядра Linux [8]. На сегодняшний день ядро Linux разрослось до такой степени, что его изучение путем анализа кода малоэффективно.

Перехват системных вызовов достаточно несложно реализуется подменой функции внутри ядра, однако, для этого необходимы права суперпользователя.

Литература

[1] Часть 5. Системные вызовы [Электронный ресурс], IBM. — URL: https://www.ibm.com/developerworks/ru/library/l-linux_kernel_05/index.html (дата обращения: 03.05.2018).

[2] Kmod Hooking library,Electronic resource, GitHub. — URL: https://github.com/milabs/kmod_hooking/ (online; accessed: 03.05.2018).

[3] Linux syscalls.h,Electronic resource, GitHub. — URL: https://github.com/torvalds/linux/blob/master/include/linux/syscalls.h (online; accessed: 03.05.2018).

[4] Linux sys.c,Electronic resource, GitHub. — URL: https://github.com/torvalds/linux/blob/master/kernel/sys.c (online; accessed: 03.05.2018).

[5] Linux pid.c,Electronic resource, GitHub. — URL: https://github.com/torvalds/linux/blob/master/kernel/pid.c (online; accessed: 03.05.2018).

[6] PID Allocation in Linux KernelElectronic resource, Medium Corporation. — URL: https://medium.com/@gargi_sharma/pid-allocation-in-linux-kernel-dc0c78d14e77 (online; accessed: 03.05.2018).

[7] Linux open.c,Electronic resource, GitHub. — URL: https://github.com/torvalds/linux/blob/master/fs/open.c (online; accessed: 03.05.2018).

[8] Linux,Electronic resource, GitHub. — URL: https://github.com/torvalds/linux (online; accessed: 03.05.2018).