

Санкт-Петербургский политехнический университет Петра Великого

# СИСТЕМНЫЕ ВЫЗОВЫ

Курс: **Проектирование ОС и компонентов**

---

Студент: **Д.В. Круминьш**

Группа: **13541/3**



Преподаватель: **Е.В. Душутина**

Рассматриваемые системные вызовы: **fork, execve, exit.**

В работе рассматриваются следующие версии ядер:

- **4.13.0-38-generic - Ubuntu 16.04;**
  - glibc 2.23
- **2.6.32-21-generic - Ubuntu 10.04**
  - glibc 2.11.1

Для выполнения работы использовалась **VMware Workstation 12 pro**  
**(12.5.7 build-5813279)**

## ПЕРЕХВАТ СИСТЕМНЫХ ВЫЗОВОВ

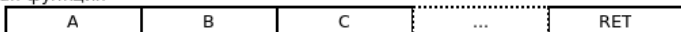
---

Перехват будет осуществляться для версии ядра 2.6.32-21, с использованием:

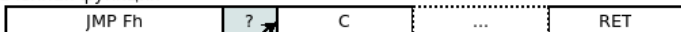
- **LKM** (Linux loadable kernel module) - динамическое подключения/отключения модулей ядра без перекомпиляции всего ядра.
- **Отображение в память** системной функции с модифицированным прологом, ссылающимся на функцию перехватчик.

# Схема перехвата

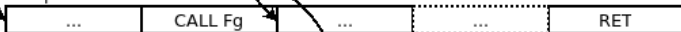
Fo : исходная функция



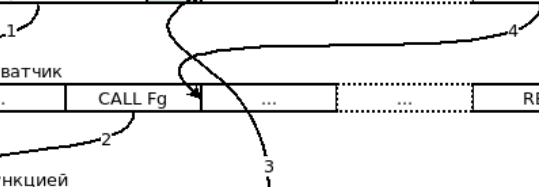
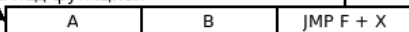
Fx : перехваченная функция



Fh : функция-перехватчик



Fs : обёртка над функцией



С ядра версии 2.6, активно началась защита ядра от внешних воздействий, многие функции были убраны из экспорта.

Возможный перехват:

1. **LSM** - фреймворк для разработки модулей безопасности ядра. Перед обращением ядра к внутреннему объекту будет обязательно вызвана функция проверки, предоставленная LSM. К сожалению предоставляет возможность перехвата не для всех системных вызовов.
2. **Полная перекомпиляция ядра** с внесением изменений, для обеспечения возможности перехвата.

## СИСТЕМНАЯ ФУНКЦИЯ FORK

---

**fork()** - системный вызов, создающий новый процесс (потомок), который является практически полной копией процесса-родителя, выполняющего этот вызов. Дочерний и родительский процессы находятся в отдельных пространствах памяти. Создавшийся процесс будет занят выполнением того же кода ровно с той же точки, что и исходный процесс.

**Расположение:** .../kernel/fork.c

**Синтаксис:** `long sys_fork(struct pt_regs *regs)`

В виде параметра выступает указатель на структуру с регистрами, возвращаемое значение - id процесса потомка.



```
8  int main()
9  {
10     pid_t pid;
11     int rv;
12     switch(pid=fork()) {
```

Листинг 1: Использование функции fork из glibc

```
1  #include <linux/kernel.h>
2  #include <sys/syscall.h>
3  #include <unistd.h>
4  int main()
5  {
6     printf("Invoking 'fork()' system call\n");
7     long resCode = syscall(57);
8     if(resCode == 0)
9         printf("I'm child process, my pid is %d\n", resCode);
10     else
11         printf("I'm parent process, my pid is %d\n", resCode);
12 }
```

Листинг 2: Прямой вызов системной функции

```
18 mmap(0x7f7ad1039000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED
   ↪ |MAP_ANONYMOUS, -1, 0) = 0x7f7ad1039000
19 close(3) = 0
20 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
   ↪ 0) = 0x7f7ad1249000
21 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
   ↪ 0) = 0x7f7ad1248000
22 arch_prctl(ARCH_SET_FS, 0x7f7ad1249700) = 0
23 mprotect(0x7f7ad1033000, 16384, PROT_READ) = 0
24 mprotect(0x600000, 4096, PROT_READ) = 0
25 mprotect(0x7f7ad1262000, 4096, PROT_READ) = 0
26 munmap(0x7f7ad124b000, 90273) = 0
27 clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|
   ↪ SIGCHLD, child_tidptr=0x7f7ad12499d0) = 58333
```

Листинг 3: Использование функции fork из glibc

По части лога видно, что сперва происходит отображение в память, а затем и создание нового процесса. Однако создание нового процесса было произведено с помощью системной функции clone(), а не fork().

```
26 munmap(0x7ff82eee1000, 90273) = 0
27 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
28 brk(NULL) = 0x12f8000
29 brk(0x1319000) = 0x1319000
30 write(1, "Invoking 'fork()' system call\n", 30Invoking 'fork()' system
    ↪ call
31 ) = 30
32 fork() = 7724
```

Листинг 4: Прямой вызов системной функции

Strace показал, что теперь, действительно был вызван систмный вызов **fork**, который вернул значение 7724.

Таблица представлена в виде:

**<number> <abi> <name> <entry point>**

**number** - уникальный номер системного вызова

**abi** - интерфейс для использования

**name** - название системного вызова

**entry point** - входная точка

63	54	64	setsockopt	sys_setsockopt
64	55	64	getsockopt	sys_getsockopt
65	56	common	clone	sys_clone / ptregs
66	57	common	fork	sys_fork / ptregs
67	58	common	vfork	sys_vfork / ptregs
68	59	64	execve	sys_execve / ptregs
69	60	common	exit	sys_exit

Листинг 5: .../arch/x86/syscalls/syscall\_64.tbl

## АНАЛИЗ GLIBC

---

Проанализируем исходный код glibc, в данном случае программы компилировались используя **glibc 2.23**.

По пути **glibc\_2.23/sysdeps/nptl/fork.c** имеется файл fork.c, в котором в строках 124-129 и представлен вызов функции.

```
124  #ifdef ARCH_FORK
125      pid = ARCH_FORK ();
126  #else
127  # error "ARCH_FORK must be defined so that the CLONE_SETTID flag is
      ↪ used"
128      pid = INLINE_SYSCALL (fork , 0);
129  #endif
```

Листинг 6: glibc\_2.23/sysdeps/nptl/fork.c

Реализация макроса представлена по пути `glibc_2.23/sysdeps/unix/sysv/linux/x86_64/arch-fork.h` в файле `arch-fork.h`.

```
24 #define ARCH_FORK() \  
25     INLINE_SYSCALL (clone, 4,  
    ↪          \  
26     CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID | SIGCHLD ,  
    ↪ 0,          \  
27     NULL, &THREAD_SELF->tid)
```

Листинг 7: `glibc_2.23/sysdeps/unix/sysv/linux/x86_64/arch-fork.h`

Как видно из реализации макроса, вызывается системный вызов `clone()`, а не `fork()`.

Исходный код(основная часть), находится в файле **fork.c**, по пути **/kernel**.

```
2006 long _do_fork(unsigned long clone_flags ,
2007             unsigned long stack_start ,
2008             unsigned long stack_size ,
2009             int __user *parent_tidptr ,
2010             int __user *child_tidptr ,
2011             unsigned long tls)
2012 {
```

Листинг 8: .../kernel/fork.c

- **clone\_flags** - флаг, для определения того, что именно нужно копировать;
- **parent\_tid** и **child\_tid** - два указателя в пространстве пользователя, для хранения id родительского и дочернего процессов;
- **stack\_start** - адрес начала стека с процессами;
- **tls** - определение локального хранилища для нового процесса



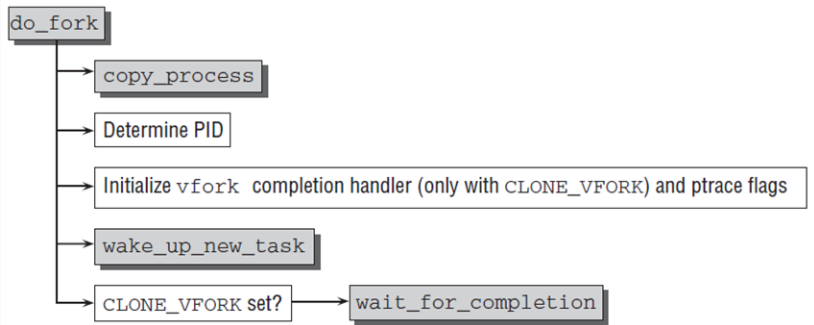


Схема работы do\_fork

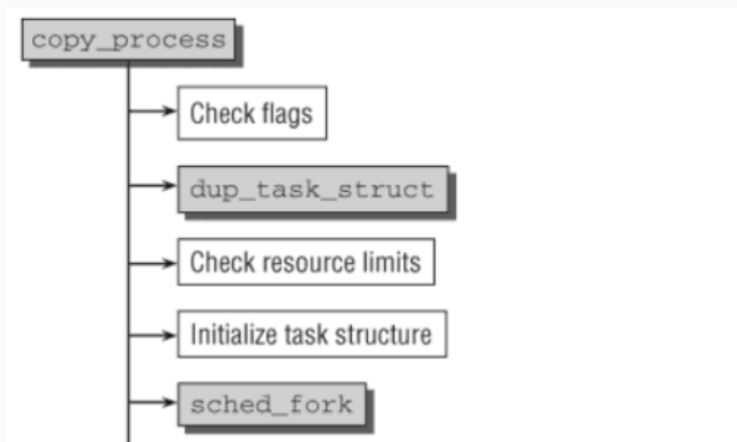


Схема работы copy\_process

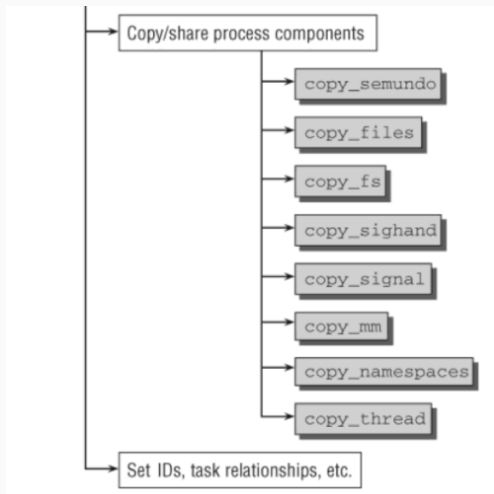


Схема работы copy\_process

Реализация также представлена в файле **fork.c**, по пути **/kernel/fork.c**.

Начало реализация представлено в строке 1166.

```
1166 long do_fork(unsigned long clone_flags ,
1167             unsigned long stack_start ,
1168             struct pt_regs *regs ,
1169             unsigned long stack_size ,
1170             int __user *parent_tidptr ,
1171             int __user *child_tidptr)
1172 {
```

Листинг 9: .../kernel/fork.c

Вся реализация уже описана для версии ядра 4.13, имеются следующие отличия:

1. у конструктора функции убран аргумент **tls** (определение локального хранилища);
2. убраны различные проверки флагов, которые ранее информировали **ptrace** о вызванном событии.

В остальном, за исключением меньшего количество проверок входных данных, все идентично.

В файле по пути `/include/asm-generic/` имеется файл `syscalls.h`, в котором определен прототип `fork()`.

```
17  #ifndef sys_fork
18  asmlinkage long sys_fork(struct pt_regs *regs);
19  #endif
```

Листинг 10: `.../kernel/asm-generic/syscalls.h`

Для того, чтобы перехватить данную функцию, напомним метод `khook_sys_fork`, который будет перехватывать системный вызов и перенаправлять управление нам:

```
203 DECLARE_KHOOK(sys_fork);  
204 int khook_sys_fork(struct pt_regs *regs)  
205 {  
206     int result;  
207  
208     KHOOK_USAGE_INC(sys_fork);  
209  
210     printk("System call for fork hooked\n");  
211  
212     result = KHOOK_ORIGIN(sys_fork, regs);  
213  
214     KHOOK_USAGE_DEC(sys_fork);  
215  
216     return result;  
217 }
```

Листинг 11: Функция перехвата `fork()`

Выполним Makefile для компиляции модуля ядра:

```
1  psaer@ubuntu: ~/Desktop/forkHook$ make
2  make -C /lib/modules/2.6.32-21-generic/build M=/home/psaer/Desktop/
   ↪ forkHook
3  make[1]: Entering directory `/usr/src/linux-headers-2.6.32-21-generic '
4    CC      /home/psaer/Desktop/forkHook/libudis86/decode.o
5    CC      /home/psaer/Desktop/forkHook/libudis86/itab.o
6    CC      /home/psaer/Desktop/forkHook/libudis86/udis86.o
7    LD      /home/psaer/Desktop/forkHook/libudis86/built-in.o
8    LD      /home/psaer/Desktop/forkHook/built-in.o
9    CC [M]   /home/psaer/Desktop/forkHook/module-init.o
10   LD [M]   /home/psaer/Desktop/forkHook/hooks.o
11   Building modules, stage 2.
12   MODPOST 1 modules
13   CC      /home/psaer/Desktop/forkHook/hooks.mod.o
14   LD [M]   /home/psaer/Desktop/forkHook/hooks.ko
15  make[1]: Leaving directory `/usr/src/linux-headers-2.6.32-21-generic '
```

Листинг 12: Лог сборки



Необходимые операции для работы:

1. **insmod** - вставка модуля в ядро (без полной перекомпиляции);
2. **lsmod** - просмотр списка текущих, работающих модулей;
3. **rmmod** - удаления заданного модуля.

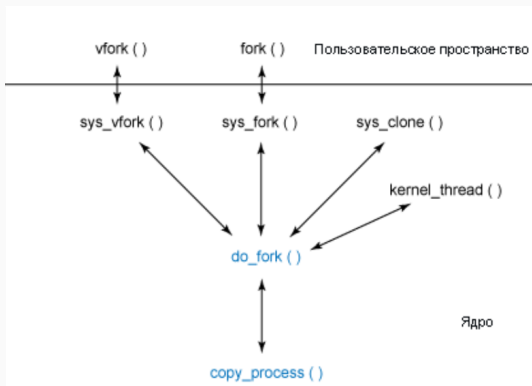
Операции должны быть произведены от пользователя с правами администратора.

## [Ядро 2.6.32][FORK] Проверка перехвата

```
1  psaer@ubuntu:~/Desktop$ ./fork2.o
2  Invoking 'fork()' system call
3  I'm parent process, my pid is 4380
4  psaer@ubuntu:~/Desktop$ I'm child process, my pid is 0
5  psaer@ubuntu:~/Desktop$ tail /var/log/kern.log
6  Apr  7 11:58:34 ubuntu kernel: [27282.741397] [hooking] khook_inode_permission(
   ↳ ffff88003e732fc0,00000024) [rmmod] = 0
7  Apr  7 11:58:34 ubuntu kernel: [27282.741413] [hooking] khook_inode_permission(
   ↳ ffff88003e735b40,00000024) [rmmod]
8  Apr  7 11:58:34 ubuntu kernel: [27282.741414] [hooking] khook_inode_permission(
   ↳ ffff88003e735b40,00000024) [rmmod] = 0
9  Apr  7 11:58:34 ubuntu kernel: [27282.741633] [hooking] khook_inode_permission(
   ↳ ffff88003e7b11b0,00000024) [rmmod]
10 Apr  7 11:58:34 ubuntu kernel: [27282.741635] [hooking] khook_inode_permission(
   ↳ ffff88003e7b11b0,00000024) [rmmod] = 0
11 Apr  7 11:59:04 ubuntu kernel: [27312.501337] [hooking] Symbol "module_free" found
   ↳ @ ffffffff810358c0
12 Apr  7 11:59:04 ubuntu kernel: [27312.501520] [hooking] Symbol "module_alloc" found
   ↳ @ ffffffff810358e0
13 Apr  7 11:59:04 ubuntu kernel: [27312.503292] [hooking] Symbol "sort_extable" found
   ↳ @ ffffffff812b1b50
14 Apr  7 11:59:04 ubuntu kernel: [27312.503559] [hooking] Symbol "sys_fork" found @
   ↳ ffffffff8101afa0
15 Apr  7 11:59:37 ubuntu kernel: [27346.294393] System call for fork hooked
```

Листинг 13: Лог с сообщением о перехвате

Дополнительно, если привести иерархию вызовов, то можно заметить что не только `fork()` и `clone()` используют функцию `do_fork()`.



Иерархия вызовов для `do_fork`

## СИСТЕМНАЯ ФУНКЦИЯ EXECVE

---

**execve()** - выполняет программу, задаваемую аргументом filename.

**Расположение:** .../fs/exec.c

**Синтаксис:** long sys\_execve(char \_\_user \*filename, char \_\_user \* \_\_user \*argv, char \_\_user \* \_\_user \*envp, struct pt\_regs \*regs)

Аргументы:

- **filename** - имя файла для выполнения;
- **argv** и **envp** - вектор аргументов и среда выполнения;
- **regs** - указатель на структуру регистров, на момент вызова данной функции.

Программа печатает в консоль сообщение, часть которого передается в виде одно из аргументов запуска. Также, в цикле выводятся все переменные окружения. Для определения размера массива с переменными, согласно документации, последний элемент будет NULL.

```
1  #include <unistd.h>
2
3  int main(int argc, char* argv[], char* envp[])
4  {
5      printf("Hello %s\n", argv[1]);
6
7      int i=0;
8      char* item = envp[i];
9      while(item != NULL){
10         printf("%d: %s\n", i, item);
11         i++;
12         item = envp[i];
13     }
14 }
```

Листинг 14: sys\_execve.c

```
25: DESKTOP_SESSION=ubuntu
26: QT_IM_MODULE=ibus
27: QT_QPA_PLATFORMTHEME=appmenu-qt5
28: XDG_SESSION_TYPE=x11
29: JOB=dbus
30: PWD=/home/psaer/Desktop/execve
31: XMODIFIERS=@im=ibus
32: GNOME_KEYRING_PID=
33: LANG=en_US.UTF-8
34: GDM_LANG=en_US
35: MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
36: IM_CONFIG_PHASE=1
37: COMPIZ_CONFIG_PROFILE=ubuntu
38: GDMSESSION=ubuntu
39: SESSIONTYPE=gnome-session
40: GTK2_MODULES=overlay-scrollbar
41: XDG_SEAT=seat0
42: HOME=/home/psaer
43: SHLVL=1
```

Листинг 15: sys\_execve.log

Для сокращения лога, приведены первые 2 строчки лога strace.

```
1  psaer@ubuntu:~/Desktop/execve$ strace ./sys_execve.o World!  
2  execve("./sys_execve.o", ["/sys_execve.o", "World!"], [/* 60 vars */])  
   ↪   = 0
```

Листинг 16: sys\_execve\_strace.log

Аргументы соответствуют ожиданиям, первый аргумент соответствует программе для запуска. Далее расположен массив аргументов, передаваемых в запускаемую программу. И наконец передаются переменные окружения, правда в данном случае, из-за их обилия они скрыты, и показано лишь их количество.



Основная часть, архитектурно независимого кода находится в файле `exec.c` по пути `/fs/`.

```
1828 int do_execve(struct filename *filename,
1829             const char __user *const __argv,
1830             const char __user *const __envp)
1831 {
1832     struct user_arg_ptr argv = { .ptr.native = __argv };
1833     struct user_arg_ptr envp = { .ptr.native = __envp };
1834     return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
1835 }
```

Листинг 17: `../fs/exec.c`

Если сравнивать с ядром керсии 2.6.32, то в данном случае, оригинальная функция **do\_execve** превратилась в некоторую обертку, а основная функциональность была перенесена в функцию **do\_execveat\_common**.

```
1679  /*
1680   * sys_execve() executes a new program.
1681   */
1682  static int do_execveat_common(int fd, struct filename *filename,
1683                               struct user_arg_ptr argv,
1684                               struct user_arg_ptr envp,
1685                               int flags)
1686  {
```

Листинг 18: ../fs/exec.c

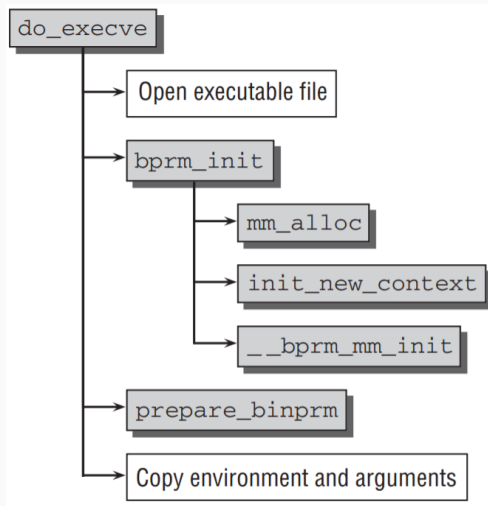


Схема работы `do_execve`

Основная часть, архитектурно независимого кода находится в файле **exec.c** по пути **/fs/**.

```
1069  /*  
1070   * sys_execve() executes a new program.  
1071   */  
1072  int do_execve(char * filename ,  
1073               char __user *__user *argv ,  
1074               char __user *__user *envp ,  
1075               struct pt_regs * regs)  
1076  {
```

Листинг 19: ../fs/exec.c

В отличие от ядра 4.13, в данном случае никакой обертки над функцией нет. По коду, основным отличием является то, что отдельные функции из **bprm\_init** были перенесены непосредственно в тело функции **do\_execve**.

В файле по пути `/include/asm-generic/` имеется файл `syscalls.h`, в котором определен прототип `execve()`.

```
25 #ifndef sys_execve
26 asmlinkage long sys_execve(char __user *filename, char __user * __user
    ↪ *argv,
27 char __user * __user *envp, struct pt_regs *regs);
28 #endif
```

Листинг 20: `.../kernel/asm-generic/syscalls.h`

Для того, чтобы перехватить данную функцию, напомним метод `khook_sys_execve`, который будет перехватывать системный вызов и перенаправлять управление нам:

```
203 DECLARE_KHOOK(sys_execve);
204 int khook_sys_execve(
205     char __user *filename, char __user * __user *argv,
206     char __user * __user *envp, struct pt_regs *regs){
207
208     int result;
209     KHOOK_USAGE_INC(sys_execve);
210
211     printk("System call for execve hooked\n");
212     printk("Executed file: %s\n", filename);
213
214     result = KHOOK_ORIGIN(sys_execve, filename, argv, envp, regs);
215     KHOOK_USAGE_DEC(sys_execve);
216
217     return result;
218 }
```

Листинг 21: Функция перехвата `execve()`

```
1  psaer@ubuntu:~/Desktop/execve$ tail /var/log/kern.log
2  Apr  8 04:31:54 ubuntu kernel: [45114.746314] [hooking] Symbol "sys_execve" found @
   ↳ ffffffff81011570
3  Apr  8 04:33:05 ubuntu kernel: [45117.704677] System call for execve hookedSystem
   ↳ call for execve hookedSystem call for execve hookedSystem call for execve
   ↳ hookedSystem call for execve hookedSystem call for execve hookedSystem
   ↳ call for execve hookedSystem call for execve hookedSystem call for execve
   ↳ hooked
4  Apr  8 04:33:05 ubuntu kernel: [45185.028182] [hooking] Symbol "module_free" found
   ↳ @ ffffffff810358c0
5  Apr  8 04:33:05 ubuntu kernel: [45185.028483] [hooking] Symbol "module_alloc" found
   ↳ @ ffffffff810358e0
6  Apr  8 04:33:05 ubuntu kernel: [45185.030155] [hooking] Symbol "sort_extable" found
   ↳ @ ffffffff812b1b50
7  Apr  8 04:33:05 ubuntu kernel: [45185.030222] [hooking] Symbol "sys_execve" found @
   ↳ ffffffff81011570
8  Apr  8 04:33:09 ubuntu kernel: [45189.628750] System call for execve hooked
9  Apr  8 04:33:09 ubuntu kernel: [45189.628753] Executed file: /usr/bin/tail
10 Apr  8 04:33:17 ubuntu kernel: [45196.874239] System call for execve hooked
11 Apr  8 04:33:17 ubuntu kernel: [45196.874244] Executed file: ./sys_execve.o
```

Листинг 22: Системный лог

## СИСТЕМНАЯ ФУНКЦИЯ EXIT

---



**exit()** - завершает работу программы. Все дескрипторы файлов, принадлежащие процессу, закрываются; все его дочерние процессы начинают управляться процессом 1 (init), а родительскому процессу посылается сигнал SIGCHLD.

**Расположение:** .../kernel/exit.c

**Синтаксис:** long sys\_exit(int error\_code)

Аргументы:

- **error\_code** - код выхода.

Программа выводит в консоль два сообщения, одно до, а другое после системного вызова `exit`, по коду 60 (системный номер функции).

```
1  #include <linux/kernel.h>
2  #include <sys/syscall.h>
3  #include <unistd.h>
4
5  int main()
6  {
7      printf("Invoking 'exit()' system call\n");
8      syscall(60);
9      printf("Message after exit call\n");
10 }
```

Листинг 23: `sys_exit.c`

```
25 mprotect(0x7f4ba90a6000, 4096, PROT_READ) = 0
26 munmap(0x7f4ba908f000, 90273) = 0
27 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
28 brk(NULL) = 0x92e000
29 brk(0x94f000) = 0x94f000
30 write(1, "Invoking 'exit()' system call\n", 30Invoking 'exit()' system
    ↪ call
31 ) = 30
32 exit(9625616) = ?
33 +++ exited with 16 +++
```

Листинг 24: sys\_exit\_strace.log

Внимания стоит уделить строчкам 30 и 32. В строчке 30 происходит вывод текста в консоль, а далее, в строке 32 происходит вызов системного вызова `exit`, после которого, никаких других системных вызовов не последовало.

Основная часть, архитектурно независимого кода находится в файле `exit.c` по пути `/kernel/`. Далее приведено лишь начало функции `do_exit`.

```
763 void __noreturn do_exit(long code)
764 {
765     struct task_struct *tsk = current;
766     int group_dead;
767     TASKS_RCU(int tasks_rcu_i);
768
769     profile_task_exit(tsk);
770     kcov_task_exit(tsk);
771
772     WARN_ON(blk_needs_flush_plug(tsk));
773
774     if (unlikely(in_interrupt()))
775         panic("Aiee, killing interrupt handler!");
776     if (unlikely(!tsk->pid))
777         panic("Attempted to kill the idle task!");
```

Листинг 25: ../kernel/exit.c

1. В вызвавшем процессе закрываются все дескрипторы открытых файлов;
2. Если родительский процесс находится в состоянии вызова wait, то системный вызов wait завершается, выдавая родительскому процессу в качестве результата идентификатор терминировавшегося процесса;
3. Если родительский процесс не находится в состоянии вызова wait, то процесс, вызвавший exit, переходит в состояние зомби. Элемент таблицы процессов, занятый зомби-процессом, содержит информацию о времени, затраченном процессом.

У всех существующих потомков терминировавшихся процессов, а также у зомби-процессов идентификатор родительского процесса устанавливается равным 1. Таким образом, все эти процессы наследуются инициализационным процессом.

- **exit\_sem()** - если процесс находится в очереди в ожидании семафора IPC, то он выгружается.
- **\_\_exit\_files(), \_\_exit\_fs(), exit\_namespace(), и exit\_sighand()** - уменьшения показателя использования объектов связанных с файловыми дескрипторами, данными файловой системы, пространством имен процессов и обработчиками сигналов. Если какой-либо показатель использования достигает нуля, объект больше не используется каким-либо процессом и удаляется.
- **exit\_notify()** - отправка сигнала о завершении процесса родительскому и всем(если имеются) дочерним процессам. Перевод процесса в состояние зомби.

Как и у ядра 4.13, основная часть кода расположена в файле **exit.c**, а далее приведено начало функции **do\_exit**.

```
796 asmlinkage NORET_TYPE void do_exit(long code)
797 {
798     struct task_struct *tsk = current;
799
800     if (unlikely(in_interrupt()))
801         panic("Aiee, killing interrupt handler!");
802     if (unlikely(!tsk->pid))
```

Листинг 26: ../kernel/exit.c

Вся реализация, подобна реализации в ядре 4.13, за исключением того, что в данном случае, порядок действий в несколько ином порядке, а также уменьшено количество действий по обеспечению откладочной информации, например отсутствует нотификация ptrace.

В файле по пути `/include/linux/` имеется файл `syscalls.h`, в котором определен прототип `exit()`.

```
418  asmlinkage long sys_exit(int error_code);
```

Листинг 27: .../kernel/linux/syscalls.h



Для того, чтобы перехватить данную функцию, напомним метод `khook_sys_exit`, который будет перехватывать системный вызов и перенаправлять управление нам:

```
203 DECLARE_KHOOK(sys_exit);
204 int khook_sys_exit(
205     int error_code)
206 {
207     int result;
208
209     KHOOK_USAGE_INC(sys_exit);
210
211     printk("System call for exit hooked\n");
212
213     result = KHOOK_ORIGIN(sys_exit, error_code);
214
215     KHOOK_USAGE_DEC(sys_exit);
216
217     return result;
218 }
```

Листинг 28: Функция перехвата `exit()`

```
1  psaer@ubuntu:~/Desktop/exit$ tail /var/log/kern.log
2  Apr  8 04:33:30 ubuntu kernel: [45210.466832] Executed file: /usr/bin/
   ↪ sudo
3  Apr  8 04:33:30 ubuntu kernel: [45210.470229] System call for execve
   ↪ hooked
4  Apr  8 04:33:30 ubuntu kernel: [45210.470232] Executed file: /sbin/
   ↪ rmmod
5  Apr  8 04:55:43 ubuntu kernel: [46542.395861] [0]: VMCI: Updating
   ↪ context from (ID=0xfe596972) to (ID=0xfe596972) on event (type
   ↪ =0).
6  Apr  8 06:24:51 ubuntu kernel: [51525.368101] [hooking] Symbol "
   ↪ module_free" found @ ffffffff810358c0
7  Apr  8 06:24:51 ubuntu kernel: [51525.368376] [hooking] Symbol "
   ↪ module_alloc" found @ ffffffff810358e0
8  Apr  8 06:24:51 ubuntu kernel: [51525.370529] [hooking] Symbol "
   ↪ sort_extable" found @ ffffffff812b1b50
9  Apr  8 06:24:51 ubuntu kernel: [51525.371212] [hooking] Symbol "
   ↪ sys_exit" found @ ffffffff8106b6d0
10 Apr  8 06:24:54 ubuntu kernel: [51528.082697] System call for exit
   ↪ hooked
```

Листинг 29: Системный лог

## МОДИФИКАЦИЯ СИСТЕМНЫХ ВЫЗОВОВ

---

В начале основной функции **do\_fork**(файл `/kernel/fork.c`) было добавлено информационное сообщение(строка 2016) для записи в системный лог.

```
2006 long _do_fork(unsigned long clone_flags ,
2007             unsigned long stack_start ,
2008             unsigned long stack_size ,
2009             int __user *parent_tidptr ,
2010             int __user *child_tidptr ,
2011             unsigned long tls)
2012 {
2013     struct task_struct *p;
2014     int trace = 0;
2015     long nr;
2016     printk("Modified fork system call");
```

Листинг 30: Модифицированный fork

```
1682 static int do_execveat_common(int fd, struct filename *filename,
1683                               struct user_arg_ptr argv,
1684                               struct user_arg_ptr envp,
1685                               int flags)
1686 {
1687     char *pathbuf = NULL;
1688     struct linux_binprm *bprm;
1689     struct file *file;
1690     struct files_struct *displaced;
1691     int retval;
1692
1693     if (IS_ERR(filename))
1694         return PTR_ERR(filename);
1695
1696     printk("Modified system call from exec. File: %s", filename->name);
```

Листинг 31: Модифицированный exec

Сразу после успешной проверки на валидность файла(строка 1693), происходит вывод информационного сообщения(строка 1696).

В начале основной функции `do_exit`(файл `/kernel/exit.c`) было добавлено информационное сообщение(строка 765) для записи в системный лог.

```
763 void __noreturn do_exit(long code)
764 {
765     printk("Modified exit system call. Code: %ld", code);
```

Листинг 32: Модифицированный exit

## ПЕРЕКОМПИЛЯЦИЯ ЯДРА

---

1. Скачать интересующее ядро по ссылке, в данном случае 4.13.0

```
https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/
```

2. Далее необходимо установить некоторые пакеты следующей командой:

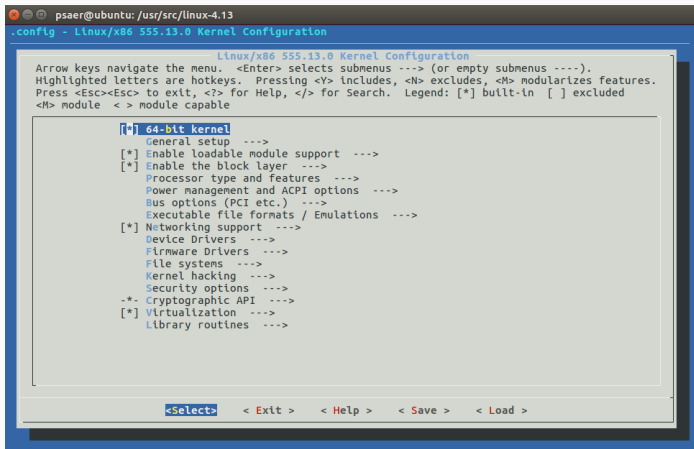
```
sudo apt-get install build-essential gcc libncurses5-dev libssl-dev
```

3. Распаковать архив с исходным кодом по пути **/usr/src/**.



# Конфигурация ядра

Выполним команду: `sudo make menuconfig`



```
psaer@ubuntu: /usr/src/linux-4.13
.config - Linux/x86 555.13.0 Kernel Configuration

Linux/x86 555.13.0 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu ----).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features.
Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded
<M> module < > module capable

[*] 64-bit kernel
  General setup --->
  [*] Enable loadable module support --->
  [*] Enable the block layer --->
  Processor type and features --->
  Power management and ACPI options --->
  Bus options (PCI etc.) --->
  Executable file formats / Emulations --->
[*] Networking support --->
  Device Drivers --->
  Firmware Drivers --->
  File systems --->
  Kernel hacking --->
  Security options --->
  *- Cryptographic API --->
  [*] Virtualization --->
  Library routines --->

<Select> < Exit > < Help > < Save > < Load >
```

Конфигурация ядра

Перед компиляцией ядра, внесем изменения в файл **Makefile**, который находится в корне разархивированного ядра.

```
1 VERSION = 555
2 PATCHLEVEL = 13
3 SUBLEVEL = 0
4 EXTRAVERSION =
5 NAME = Fearless Coyote
```

Листинг 33: Файл Makefile

В представленных первых 5 строках представлена основная информация о версии ядра. В моем случае, вместо версии 4 была поставлена версия 555.

Теперь приступаем к компиляции, для этого выполняем следующую команду:

```
1 sudo make -j 3 && sudo make modules_install -j 3 && sudo make install -  
   ↪ j 3
```

Листинг 34: Компиляция ядра

Ключ `j` означает количество задействованных ядер системы. В моем случае, в настройках VMware, виртуальной машине было выделено 3 ядра процессора.

Первые две команды, из листинга выше, выполняют компиляцию ядра, а последняя компилирует въедино в образ ядра системы.

Процесс, в моем случае занимает около 20 минут.

Далее необходимо включить показ меню **GRUB**. Для этого редактируем файл **grub** по пути **/etc/default/**.

```
1 # If you change this file , run 'update-grub' afterwards to update
2 # /boot/grub/grub.cfg.
3 # For full documentation of the options in this file , see:
4 #   info -f grub -n 'Simple configuration'
5
6 GRUB_DEFAULT=0
7 #GRUB_HIDDEN_TIMEOUT=0
8 #GRUB_HIDDEN_TIMEOUT_QUIET=true
9 GRUB_TIMEOUT=10
10 GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
```

Листинг 35: Файл grub

В данном файле необходимо закомментировать(поставить знак # в начале строки) строки 7 и 8.

GNU GRUB version 2.02~beta2-36ubuntu3.17

```
Ubuntu
*Advanced options for Ubuntu
Memory test (memtest86+)
Memory test (memtest86+, serial console 115200)
```

Use the ↑ and ↓ keys to select which entry is highlighted.  
Press enter to boot the selected OS, `e' to edit the commands  
before booting or `c' for a command-line.

# Выбор ядра в GRUB

GNU GRUB version 2.02~beta2-36ubuntu3.17

```
*Ubuntu, with Linux 555.13.0
Ubuntu, with Linux 555.13.0 (upstart)
Ubuntu, with Linux 555.13.0 (recovery mode)
Ubuntu, with Linux 555.13.0.old
Ubuntu, with Linux 555.13.0.old (upstart)
Ubuntu, with Linux 555.13.0.old (recovery mode)
Ubuntu, with Linux 4.13.0-38-generic
Ubuntu, with Linux 4.13.0-38-generic (upstart)
Ubuntu, with Linux 4.13.0-38-generic (recovery mode)
Ubuntu, with Linux 4.13.0-36-generic
Ubuntu, with Linux 4.13.0-36-generic (upstart)
Ubuntu, with Linux 4.13.0-36-generic (recovery mode)
Ubuntu, with Linux 4.13.0
Ubuntu, with Linux 4.13.0 (upstart)
Ubuntu, with Linux 4.13.0 (recovery mode)
```

Use the ↑ and ↓ keys to select which entry is highlighted.  
Press enter to boot the selected OS, 'e' to edit the commands  
before booting or 'c' for a command-line. ESC to return previous  
menu.

Выбор ядра в GRUB

Выполним программу, напрямую вызывающую `fork`.

```
1  psaer@ubuntu:~/Desktop/sysUtils$ ./fork.o
2  Invoking 'fork()' system call
3  I'm parent process, my pid is 2470
4  I'm child process, my pid is 0
```

Листинг 36: Выполнением программы с прямым вызовом `fork`

Для поиска сообщений, которые записываются в системный лог, будет использована следующая команда:

```
1  psaer@ubuntu:~/Desktop/sysUtils$ sudo grep -rnw '/var/log/' -e 'fork'
```

Листинг 37: Команда для поиска некоторого сообщения

Поиск происходит рекурсивно, в каталоге `/var/log/` на предмет наличия в тексте `fork`.

```
122 /var/log/syslog:157:Apr 10 09:55:15 ubuntu kernel: [ 516.073716]  
    ↪ Modified fork system call  
123 /var/log/syslog:159:Apr 10 09:55:15 ubuntu kernel: [ 516.079235]  
    ↪ Modified fork system call  
124 /var/log/syslog:164:Apr 10 09:55:34 ubuntu kernel: [ 534.641490]  
    ↪ Modified fork system call  
125 /var/log/syslog:165:Apr 10 09:55:34 ubuntu kernel: [ 534.641698]  
    ↪ Modified fork system call  
126 /var/log/syslog:168:Apr 10 09:55:37 ubuntu kernel: [ 538.214193]  
    ↪ Modified fork system call  
127 /var/log/syslog:169:Apr 10 09:55:38 ubuntu kernel: [ 538.214392]  
    ↪ Modified system call from exec. File: ./fork.o  
128 /var/log/syslog:170:Apr 10 09:55:38 ubuntu kernel: [ 538.214810]  
    ↪ Modified fork system call  
129 /var/log/syslog:173:Apr 10 09:55:40 ubuntu kernel: [ 540.848736]  
    ↪ Modified fork system call  
130 /var/log/auth.log:2:Apr 10 09:55:15 ubuntu sudo:    psaer : TTY=pts/4 ;  
    ↪ PWD=/home/psaer/Desktop/sysUtils ; USER=root ; COMMAND=/bin/  
    ↪ grep -rnw /var/log/ -e fork
```

Листинг 38: Результаты поиска fork



```
1  psaer@ubuntu:~/Desktop/sysUtils$ sudo grep -rnw '/var/log/' -e 'exit' | tail
2  [sudo] password for psaer:
3  /var/log/syslog:603:Apr 10 10:20:30 ubuntu kernel: [ 2030.094680] Modified exit
   ↳ system call. Code: 0
4  /var/log/syslog:604:Apr 10 10:20:32 ubuntu kernel: [ 2030.207164] Modified exit
   ↳ system call. Code: 0
5  /var/log/syslog:605:Apr 10 10:20:32 ubuntu kernel: [ 2032.802104] Modified exit
   ↳ system call. Code: 0
6  /var/log/syslog:607:Apr 10 10:20:32 ubuntu kernel: [ 2032.809867] Modified system
   ↳ call from exec. File: ./exit.o
7  /var/log/syslog:608:Apr 10 10:20:35 ubuntu kernel: [ 2032.810497] Modified exit
   ↳ system call. Code: 4096
8  /var/log/syslog:613:Apr 10 10:20:35 ubuntu kernel: [ 2035.897177] Modified exit
   ↳ system call. Code: 0
9  /var/log/syslog:614:Apr 10 10:20:36 ubuntu kernel: [ 2035.897348] Modified exit
   ↳ system call. Code: 0
10 /var/log/syslog:617:Apr 10 10:20:36 ubuntu kernel: [ 2036.465294] Modified exit
   ↳ system call. Code: 0
11 /var/log/syslog:618:Apr 10 10:20:43 ubuntu kernel: [ 2036.465335] Modified exit
   ↳ system call. Code: 0
12 /var/log/auth.log:17:Apr 10 10:20:46 ubuntu sudo:      psaer : TTY=pts/4 ; PWD=/home/
   ↳ psaer/Desktop/sysUtils ; USER=root ; COMMAND=/bin/grep -rnw /var/log/ -e
   ↳ exit
```

Листинг 39: Результаты поиска exit

- Writing a Linux Kernel Module. -- URL: <http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/> (дата обращения: 2018-04-06).
- Встраивание в ядро Linux: перехват функций. -- URL: <https://habrahabr.ru/company/securitycode/blog/237089/> (дата обращения: 2018-04-07).
- fork(2) - Linux man page. -- URL: <https://linux.die.net/man/2/fork> (дата обращения: 2018-04-01).
- clone(2) - Linux man page. -- URL: <https://linux.die.net/man/2/clone> (дата обращения: 2018-04-01).
- glibc, archive of versions. -- URL: <https://ftp.gnu.org/gnu/glibc/> (дата обращения: 2018-04-04).

- Fork bomb. — URL: [https://en.wikipedia.org/wiki/Fork\\_bomb](https://en.wikipedia.org/wiki/Fork_bomb) (дата обращения: 2018-04-07).
- `execve` - execute program. — URL: [http://man7.org/linux/man-pages/man2\\_execve.2.html](http://man7.org/linux/man-pages/man2_execve.2.html) (дата обращения: 2018-04-08).
- `exit` – terminate process. — URL: <http://man.cat-v.org/unix-1st/2/sys-exit> (дата обращения: 2018-04-08).
- Анатомия управления процессами в Linux. — URL: <https://www.ibm.com/developerworks/ru/library/l-linux-process-management/index.html> (дата обращения: 2018-04-07).