Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

**Кафедра компьютерных систем и программных технологий**

**Отчет о лабораторной работе**

**Курс:** Проектирование ОС и компонентов

**Тема:** Системные вызовы

Выполнил студент группы 13541/3 _____ Д.В. Круминьш
(подпись)

Преподаватель _____ Е.В. Душутина
(подпись)

Санкт-Петербург
2018 г.

# Содержание

# 1 Постановка задачи

В данной работе, для заданных системных функций необходимо:

1. ознакомиться с функциональностью и параметрами;
2. изучить исходный код;
3. произвести перехват;
4. модифицировать исходный код.

Заданные системные функции: **fork, execve, exit**.
Работа будет производиться для следующих версий ядер и виртуальных систем:

- **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)**

# 2 Сведения о системе

Работа производилась на реальной системе, со следующими характеристиками:

| Элемент | Значение |
|---|---|
| Имя ОС | Майкрософт Windows 10 Pro (Registered Trademark) |
| Версия | 10.0.16299 Сборка 16299 |
| Установленная оперативная память (RAM) | 16,00 ГБ |
| Процессор | Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2496 МГц, ядер: 4, логических процессоров: 4 |

Таблица 1: Сведения о системе

# 3 Перехват системных функций

Для осуществления перехвата будет использоваться **LKM** и создание временных отображений памяти(**MMU**).

## 3.1 Linux loadable kernel module

**LKM**(Linux loadable kernel module) - динамическое подключения/отключения модулей ядра без перекомпиляции всего ядра.

Для начала необходимо загрузить заголовки текущего ядра, для этого необходимо узнать версию используемого ядра и скачать заголовки.

```
1 psaer@ubuntu:~/Desktop/syscalls/fork/hook$ uname -r
2 4.13.0-38-generic
3 psaer@ubuntu:~/Desktop/syscalls/fork/hook$ sudo apt-get install linux-headers
    ↪ -4.13.0-38-generic
```

Листинг 1: Установка заголовков

Далее необходимо создать файл **Makefile**.

```
1 obj-m+=hello.o
2
3 all:
4     make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
5 clean:
6     make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

Листинг 2: Makefile

В первой строчке указана цель сборки, в данном случае модуль **hello**. Далее указаны пути для сборки.

Далее необходимо приступить к написанию самого модуля, в данном случае он просто будет выводить текст.

```
1 #include <linux/module.h>
2
3 static int __init testModuleInit(void){
4     printk("Hello World!\n");
5     return 0;
6 }
7
8 static void __exit testModuleExit(void){
```

```
9       printk("Module unloaded. Bye!\n");
10  }
11
12  module_init(testModuleInit);
13  module_exit(testModuleExit);
```

Листинг 3: hello.c

Важными частями модуля, являются функции определенным макросами **__init** и **__exit** - точки входа и выхода[1].

Приступаем к сборке модуля.

```
1  psaer@ubuntu:~/Desktop/syscalls/fork/hook/helloWorld$ make
2  make -C /lib/modules/4.13.0-38-generic/build/ M=/home/psaer/Desktop/syscalls/
       ↪ fork/hook/helloWorld modules
3  make[1]: Entering directory '/usr/src/linux-headers-4.13.0-38-generic'
4    CC [M]  /home/psaer/Desktop/syscalls/fork/hook/helloWorld/hello.o
5    Building modules, stage 2.
6    MODPOST 1 modules
7    CC      /home/psaer/Desktop/syscalls/fork/hook/helloWorld/hello.mod.o
8    LD [M]  /home/psaer/Desktop/syscalls/fork/hook/helloWorld/hello.ko
9  make[1]: Leaving directory '/usr/src/linux-headers-4.13.0-38-generic'
```

Листинг 4: Лог сборки

По завершению сборки, будет создан файл **hello.ko** - модуль для внедрения в ядро. Для подключения модуля используется команда **insmod**, а для просмотра текущих модулей, команда **lsmod**.

```
1  psaer@ubuntu:~/Desktop/syscalls/fork/hook/helloWorld$ sudo insmod hello.ko
2  [sudo] password for psaer:
3  psaer@ubuntu:~/Desktop/syscalls/fork/hook/helloWorld$ lsmod
4  Module                  Size  Used by
5  hello                  16384  0
6  iptable_filter         16384  0
7  ip_tables              24576  1 iptable_filter
8  x_tables               40960  2 ip_tables,iptable_filter
9  vmw_vsock_vmci_transport    28672  2
10 vsock                  36864  3 vmw_vsock_vmci_transport
```

Листинг 5: Внедрение и список модулей

Как видно, модуль успешно встроен и функционирует.

Для выгрузки модуля используется команда **rmmod**.

```
1  sudo rmmod hello.ko
```

Листинг 6: Выгрузка модуля

Дополнительно посмотрим сообщения модуля, записанные в лог.

```
psaer@ubuntu:~/Desktop/syscalls/fork/hook/helloWorld$ tail -f /var/log/kern.log
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9248]    plen 24
    ↪  (255.255.255.0)
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9248]    gateway
    ↪  192.168.32.2
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9248]    server
    ↪ identifier 192.168.32.254
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9248]    lease
    ↪ time 1800
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9248]
    ↪ nameserver '192.168.32.2'
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9248]    domain
    ↪ name 'localdomain'
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9249]    wins
    ↪ '192.168.32.2'
Apr  6 02:23:52 ubuntu NetworkManager[894]: <info>  [1523006632.9249] dhcp4 (
    ↪ ens33): state changed bound -> bound
Apr  6 02:34:59 ubuntu kernel: [33889.125736] Hello World!
Apr  6 02:35:21 ubuntu kernel: [33910.750370] Module unloaded. Bye!
```

Листинг 7: Лог сообщений модуля

Как видно, в лог были добавлены новые записи. В дальнейшем, на основе данного метода будут написаны перехватчики системных функций.

## 3.2 Отображение в память

Приведенное решение перехвата, основано на соответствующей статье[2].

### 3.2.1 Методика осуществления перехвата

Суть описываемого способа осуществления перехвата будет состоять в том, чтобы модифицировать пролог (начало) целевой функции таким образом, чтобы его выполнение процессором приводило передаче управления на функцию-обработчик.

Другими словами, для каждой целевой функции осуществим модификацию пролога путём записи в её начало команды JMP . Это позволит переключить поток выполнения

с целевой функции на соответствующий обработчик.

Например, если до перехвата функция inode_permission имеет вид:

```
1 inode_permission:
2    0xffffffff811c4530 <+0>: nopl   0x0(%rax,%rax,1)
3    0xffffffff811c4535 <+5>: push   %rbp
4    0xffffffff811c4536 <+6>: test   $0x2,%sil
5    0xffffffff811c453a <+10>:    mov    0x28(%rdi),%rax
6    0xffffffff811c453e <+14>:    mov    %rsp,%rbp
7    0xffffffff811c4541 <+17>:    jne    0xffffffff811c454a <inode_permission+26>
8    0xffffffff811c4543 <+19>:    callq  0xffffffff811c4470 <__inode_permission>
```

Листинг 8: Ассемблерный пролог inode_permission

То после перехвата, пролог этой функции будет выглядеть следующим образом:

```
1 inode_permission:
2    0xffffffff811c4530 <+0>: jmpq   0xffffffffa05a60e0     => ПЕРЕДАЧА
     ↪ УПРАВЛЕНИЯ НА ПЕРЕХВАТЧИК
3    0xffffffff811c4535 <+5>: push   %rbp
4    0xffffffff811c4536 <+6>: test   $0x2,%sil
5    0xffffffff811c453a <+10>:    mov    0x28(%rdi),%rax
6    0xffffffff811c453e <+14>:    mov    %rsp,%rbp
7    0xffffffff811c4541 <+17>:    jne    0xffffffff811c454a <inode_permission+26>
8    0xffffffff811c4543 <+19>:    callq  0xffffffff811c4470 <__inode_permission>
```

Листинг 9: Модифицированный ассемблерный пролог inode_permission

Именно записанная поверх оригинальных инструкций пяти-байтовая команда JMP с кодом E9.XX.XX.XX.XX приводит к передачи управления. В этом и состоит основная суть описываемого способа осуществления перехвата. Далее будут рассмотрены некоторые особенности его реализации в ядре Linux.

### 3.2.2 Особенности осуществления перехвата функций

Как было отмечено, суть патчинга заключается в модификации кода ядра. Основной проблемой, возникающей при этом является то, что запись в страницы памяти, содержащие код невозможна т.к. в архитектуре x86 существует специальный защитный механизм, в соответствии с которым попытка записи в защищённые от записи области памяти может приводить к генерации исключения. Данный механизм носит название «страничной защиты» и является базовым для реализации многих функций ядра, таких, как например COW. Поведение процессора в этой ситуации определяется битом WP регистра CR0, а права доступа к странице описываются в соответствующей ей структуре-

описателе PTE. При установленном бите WP регистра CR0 попытка записи в защищённые от записи страницы (сброшен бит RW в PTE) ведёт к генерации процессором соответствующего исключения (#GP).

Зачастую, решением данной проблемы является временное отключение страничной защиты сбросом бита WP регистра CR0. Это решение имеет место быть, однако применять его нужно с осторожностью, ведь как было отмечено, механизм страничной защиты является основой для многих механизмов ядра. Кроме того, на SMP-системах, поток, выполняющийся на одном из процессоров и там же снимающий бит WP, может быть прерван и перемещён на другой процессор!

Более лучшим и в достаточной степени универсальным, является способ создания временных отображений. В силу особенностей работы MMU, для каждого физического фрейма памяти может быть создано несколько ссылающихся на него описателей, имеющих различные атрибуты. Это позволяет создать для целевой области памяти отображение, доступное для записи. Ниже приведена функция map_writable, которая и создаёт такое отображение:

```
1  /*
2   * map_writable creates a shadow page mapping of the range
3   * [addr, addr + len) so that we can write to code mapped read-only.
4   *
5   * It is similar to a generalized version of x86's text_poke.  But
6   * because one cannot use vmalloc/vfree() inside stop_machine, we use
7   * map_writable to map the pages before stop_machine, then use the
8   * mapping inside stop_machine, and unmap the pages afterwards.
9   *
10  * STOLEN from: https://github.com/jirislaby/ksplice
11  */
12 static void *map_writable(void *addr, size_t len)
13 {
14         void *vaddr;
15         int nr_pages = DIV_ROUND_UP(offset_in_page(addr) + len, PAGE_SIZE);
16         struct page **pages = kmalloc(nr_pages * sizeof(*pages), GFP_KERNEL);
17         void *page_addr = (void *)((unsigned long)addr & PAGE_MASK);
18         int i;
19
20         if (pages == NULL)
21                 return NULL;
22
23         for (i = 0; i < nr_pages; i++) {
24                 if (__module_address((unsigned long)page_addr) == NULL) {
25                         pages[i] = virt_to_page(page_addr);
26                         WARN_ON(!PageReserved(pages[i]));
```

```
27                  } else {
28                          pages[i] = vmalloc_to_page(page_addr);
29                  }
30                  if (pages[i] == NULL) {
31                          kfree(pages);
32                          return NULL;
33                  }
34                  page_addr += PAGE_SIZE;
35          }
36          vaddr = vmap(pages, nr_pages, VM_MAP, PAGE_KERNEL);
37          kfree(pages);
38          if (vaddr == NULL)
39                  return NULL;
40          return vaddr + offset_in_page(addr);
41  }
```

Листинг 10: Функция map_writable

Использование данной функции позволит создать доступное для записи отображение для любой области памяти. Освобождение созданного таким образом региона осуществляется с использованием функции vfree, аргументом которой должно служить выравненное на границу страницы значение адреса.

Следующим важным моментом является то, что в ходе модификации посредством патчинга так или иначе затирается часть пролога целевой функции. На это не стоит обращать внимания, если не предполагается использовать эту функцию далее. Однако, если по каким-либо причинам реализуемый целевой функцией алгоритм может быть полезен после патчинга, стоит обеспечить возможность исполнения «старого» кода учитывая «испорченность» существующего пролога.

Далее представлена иллюстрация на которой схематически представлен процесс перехвата функции с сохранением возможности обращения к исходной функциональности.

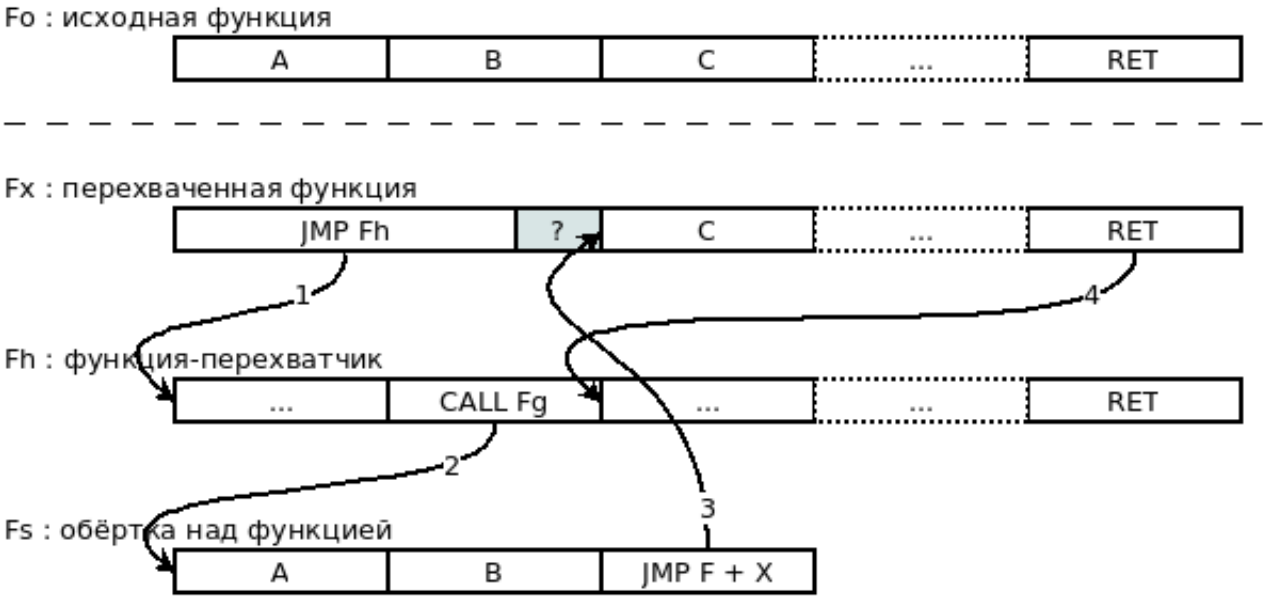


Рис. 1: Процесс перехвата функции

На рисунке 1, цифрой 1 отмечена передача управления от целевой функции к функции-перехватчику (команда JMP), цифрой 2 — вызов оригинальной функции с использованием сохранённой части пролога (команда CALL), цифрой 3 — возврат управления к части оригинальной функции, не подвергавшейся изменению (команда JMP), и, наконец, цифрой 4 — возврат управления по завершению вызова оригинальной функции из перехватчика (команда RET). Таким образом, обеспечивается возможность использования реализуемых перехватываемой функцией возможностей.

### 3.2.3 Реализация перехвата функций

Будем описывать каждую перехватываемую функцию следующей структурой:

```
1  typedef struct {
2      /* tagret's name */
3      char * name;
4
5      /* target's insn length */
6      int length;
7
8      /* target's handler address */
9      void * handler;
10
11     /* target's address and rw-mapping */
12     void * target;
13     void * target_map;
```

```
14
15      /* origin's address and rw-mapping */
16      void * origin;
17      void * origin_map;
18
19      atomic_t usage;
20  } khookstr_t;
```

Листинг 11: Структура функции для перехвата

- **name** — имя перехватываемой функции (имя символа);
- **length** — длина затираемой последовательности инструкций пролога;
- **handler** — адрес функции-перехватчика;
- **target** — адрес самой целевой функции;
- **target_map** — адрес доступной для записи проекции целевой функции;
- **origin** — адрес функции-переходника, используемой для доступа к исходной функциональности;
- **origin_map** — адрес доступной для записи проекции соответствующего переходника;
- **usage** — счётчик «залипаний», учитывающий число спящих в перехвате потоков.

Каждая перехватываемая функция должна быть представлена такой структурой. Для этого, дабы упростить регистрацию перехватчиков, используется макрос DECLARE_KHOOK(...), представляемый следующим образом:

```
1  #define __DECLARE_TARGET_ALIAS(t)    \
2      void __attribute__((alias("khook_"#t))) khook_alias_##t(void)
3
4  #define __DECLARE_TARGET_ORIGIN(t)  \
5      void notrace khook_origin_##t(void){\
6          asm volatile (          \
7              ".rept 0x20\n"      \
8              ".byte 0x90\n"      \
9              ".endr\n"           \
10         );              \
11     }
12
13 #define __DECLARE_TARGET_STRUCT(t)  \
14     khookstr_t __attribute__((unused,section(".khook"),aligned(1))) __khook_##t
15
16 #define DECLARE_KHOOK(t)        \
```

```
17      __DECLARE_TARGET_ALIAS(t);      \
18      __DECLARE_TARGET_ORIGIN(t);     \
19      __DECLARE_TARGET_STRUCT(t) = {  \
20          .name = #t,                 \
21          .handler = khook_alias_##t, \
22          .origin = khook_origin_##t, \
23          .usage = ATOMIC_INIT(0),    \
24      }
```

Листинг 12: Макрос регистрации перехватчика

Вспомогательные макросы __DECLARE_TARGET_ALIAS(...), __DECLARE_TARGET_ORIGIN(...) декларируют перехватчик и переходник (32 nop'а). Саму структуру объявляет макрос __DECLARE_TARGET_STRUCT(...), посредством атрибута section определяя её в специальную секцию (.khook).

При загрузке модуля ядра происходит перечисление всех зарегистрированных перехватов (khook_for_each), представленных структурами в секции с именем .khook. Для каждого из них осуществляется поиск адреса соответствующего символа (get_symbol_address), а также настройка вспомогательных элементов, включая создание отображений (map_witable):

```
1  static int init_hooks(void)
2  {
3      khookstr_t * s;
4
5      khook_for_each(s) {
6          s->target = get_symbol_address(s->name);
7          if (s->target) {
8              s->target_map = map_writable(s->target, 32);
9              s->origin_map = map_writable(s->origin, 32);
10
11             if (s->target_map && s->origin_map) {
12                 if (init_origin_stub(s) == 0) {
13                     atomic_inc(&s->usage);
14                     continue;
15                 }
16             }
17         }
18
19         debug("Failed to initalize \"%s\" hook\n", s->name);
20     }
21
22     /* apply patches */
23     stop_machine(do_init_hooks, NULL, NULL);
```

```
24
25      return 0;
26  }
```

Листинг 13: Загрузка модуля

Важную роль играет функция init_origin_stub, осуществляющая инициализацию и построение переходника, используемого для вызова оригинальной функции после перехвата:

```
1   static int init_origin_stub(khookstr_t * s)
2   {
3       ud_t ud;
4
5       ud_initialize(&ud, BITS_PER_LONG, \
6                 UD_VENDOR_ANY, (void *)s->target, 32);
7
8       while (ud_disassemble(&ud) && ud.mnemonic != UD_Iret) {
9           if (ud.mnemonic == UD_Ijmp || ud.mnemonic == UD_Iint3) {
10              debug("It seems that \"%s\" is not a hooking virgin\n", s->name);
11              return -EINVAL;
12          }
13
14  #define JMP_INSN_LEN    (1 + 4)
15
16          s->length += ud_insn_len(&ud);
17          if (s->length >= JMP_INSN_LEN) {
18              memcpy(s->origin_map, s->target, s->length);
19              x86_put_jmp(s->origin_map + s->length, s->origin + s->length, s->
    ↪ target + s->length);
20              break;
21          }
22      }
23
24      return 0;
25  }
```

Листинг 14: Переходник для вызова оригинальной функции

Как видно, для определения количества затираемых при патчинге пролога инструкций используется дизассемблер udis86. В принципе, для этой цели подойдёт любой дизассемблер с функцией определения длины инструкции (т.н. Length-Disassembler Engine, LDE). Для этих целей используется полноценный дизассемблер udis86, который имеет BSD-лицензию и хорошо зарекомендовал себя. Как только число инструкций определено, происходит их копирование по адресу origin_map, что соответствует RW-проекции

32-байтного переходника origin. В завершении, после сохранённых команд с использованием x86_put_jmp вставляется команда, возвращающая управление на оригинальной код целевой функции, не подвергшийся изменению.

Последним элементом, позволяющим сделать модификацию кода ядра безопасной, является механизм stop_machine:

```
1 #include <linux/stop_machine.h>
2
3 int stop_machine(int (*fn)(void *), void *data, const struct cpumask *cpus)
```

Листинг 15: Механизм безопасной модификации ядра

Суть в том, что stop_machine осуществляет выполнение функции fn с заданным набором активных в момент выполнения процессоров, что задаётся соответствующей маской cpumask. Именно это позволяет использовать данный механизм для осуществления модификации кода ядра, т.к. задание соответствующей маски автоматически исключает необходимость отслеживания тех потоков ядра, выполнение которых может затрагивать модифицируемый код.

Далее, по ходу работы, с помощью макроса **DECLARE_KHOOK** будут написаны перехватчики заданных функций.

# 4 Системная функция fork

**fork()**[3] - 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.

В ходе экспериментов выяснилось, что вместо fork() вызывается функция **clone()**.
**clone()**[4] - создаёт новый процесс подобно fork. Но в отличие от fork(), clone() позволяет процессу-потомку использовать некоторые части контекста выполнения совместно с вызывающим процессом, например: область памяти, таблица файловых дескрипторов

и таблица обработчиков сигналов.
**Расположение:** .../arch/x86/kernel/process_64.c
**Синтаксис:** long sys_clone(unsigned long clone_flags, unsigned long newsp, void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
**Аргументы конструктора**:

1. clone_flags - параметры копирования(что копировать, а что нет).
2. newsp - новый адрес функции;
3. parent_tid и child_tid - два указателя в пространстве пользователя, для хренения id родительского и дочернего процессов;
4. regs - структура регистрова процесса родителя.

## 4.1 Программы для анализа

Запуск программы произведен на версии ядра 4.13.
Представленные программы написаны на языке **C**, и показывают использование заданной функции на пользовательском уровне.

### 4.1.1 Программа с использованием fork

Программа, с помощью функции fork() создает процесс-потомок.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
8  int main()
9  {
10   pid_t pid;
11   int rv;
12   switch(pid=fork()) {
13   case -1:
14            perror("fork"); /* произошла ошибка */
15            exit(1); /*выход из родительского процесса*/
16   case 0:
17            printf(" CHILD: Это процесс потомок!\n");
18            printf(" CHILD: Мой PID %d\n", getpid());
```

```
19              printf(" CHILD: PID моего родителя %d\n",getppid());
20              printf(" CHILD: Выход!\n");
21              exit(rv);
22      default:
23              printf("PARENT: Это процесс родитель!\n");
24              printf("PARENT: Мой PID %d\n", getpid());
25              printf("PARENT: PID моего потомка %d\n",pid);
26              printf("PARENT: Я жду, пока потомок не вызовет exit()...\n");
27              wait(&rv);
28              printf("PARENT: Код возврата потомка:%d\n", WEXITSTATUS(rv));
29              printf("PARENT: Выход!\n");
30      }
31  }
```

Листинг 16: forkExample.c

Программа выводит информацию о каждом процессе.

```
1  psaer@ubuntu:~/Desktop/syscalls/fork$ ./a.out
2  PARENT: Это процесс родитель!
3  PARENT: Мой PID 55714
4  PARENT: PID моего потомка 55715
5  PARENT: Я жду, пока потомок не вызовет exit()...
6    CHILD: Это процесс потомок!
7    CHILD: Мой PID 55715
8    CHILD: PID моего родителя 55714
9    CHILD: Выход!
10 PARENT: Код возврата потомка:128
11 PARENT: Выход!
```

Листинг 17: forkExample.log

### 4.1.2 Программа с использованием clone

Программа, с помощью функции clone() создает процесс-потомок.

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sched.h>
6  #include <signal.h>
7  #define FIBER_STACK 8192
8
9  void * stack;
```

```
10 int do_something(){
11     int a = 0;
12     while (a<10){
13         printf("child pid : %d, a = %d\n", getpid(), a++);
14     }
15     exit(1);
16 }
17 int main() {
18     void * stack;
19     stack = malloc(FIBER_STACK);
20     if(!stack) {
21         printf("The stack failed\n");
22         exit(0);
23     }
24
25     int a = 0;
26     int c =0;
27     if (c == 0)
28         clone(&do_something, (char *)stack + FIBER_STACK, CLONE_VM, 0);
29     while (a<10){
30         printf("parent pid : %d, a = %d\n", getpid(), a++);
31     }
32
33     free(stack);
34     exit(1);
35 }
```

Листинг 18: cloneExample.c

Каждый процесс выводит в консоль значения от 0 до 9.

```
1  psaer@ubuntu:~/Desktop/syscalls/fork$ ./clone.o
2  child pid : 57385, a = 0
3  child pid : 57385, a = 1
4  child pid : 57385, a = 2
5  child pid : 57385, a = 3
6  child pid : 57385, a = 4
7  child pid : 57385, a = 5
8  child pid : 57385, a = 6
9  child pid : 57385, a = 7
10 child pid : 57385, a = 8
11 child pid : 57385, a = 9
12 parent pid : 57384, a = 0
13 parent pid : 57384, a = 1
14 parent pid : 57384, a = 2
15 parent pid : 57384, a = 3
```

```
parent pid : 57384, a = 4
parent pid : 57384, a = 5
parent pid : 57384, a = 6
parent pid : 57384, a = 7
parent pid : 57384, a = 8
parent pid : 57384, a = 9
```

Листинг 19: cloneExample.log

### 4.1.3 Программа с прямым вызовом fork

Программа напрямую вызывает системную функцию fork(), по её номеру(57), записанному в таблице системных функций. Таким образом игнорируется обвязка glibc.

```
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
int main()
{
    printf("Invoking 'fork()' system call\n");
    long resCode = syscall(57);
    if(resCode == 0)
        printf("I'm child process, my pid is %d\n", resCode);
    else
        printf("I'm parent process, my pid is %d\n", resCode);
}
```

Листинг 20: fork2.c

```
psaer@ubuntu:~/Desktop/syscalls/fork$ ./fork2.o
Invoking 'fork()' system call
I'm parent process, my pid is 7714
I'm child process, my pid is 0
```

Листинг 21: fork2.log

## 4.2 Ядро версии 4.13.0-38-generic

### 4.2.1 Анализ strace

С помощью команды **strace** посмотрим лог вызовов для программы fork() приведенной в листинге 16.

```
psaer@ubuntu:~/Desktop/syscalls/fork$ strace ./fork.o
execve("./fork.o", ["./fork.o"], [/* 67 vars */]) = 0
brk(NULL)                                       = 0x1c9c000
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)              = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=90273, ...}) = 0
mmap(NULL, 90273, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7ad124b000
close(3)                                        = 0
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...,
    ↪ 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f7ad124a000
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7f7ad0c73000
mprotect(0x7f7ad0e33000, 2097152, PROT_NONE) = 0
mmap(0x7f7ad1033000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_DENYWRITE, 3, 0x1c0000) = 0x7f7ad1033000
mmap(0x7f7ad1039000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_ANONYMOUS, -1, 0) = 0x7f7ad1039000
close(3)                                        = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f7ad1249000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f7ad1248000
arch_prctl(ARCH_SET_FS, 0x7f7ad1249700) = 0
mprotect(0x7f7ad1033000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)             = 0
mprotect(0x7f7ad1262000, 4096, PROT_READ) = 0
munmap(0x7f7ad124b000, 90273)                   = 0
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    ↪ child_tidptr=0x7f7ad12499d0) = 58333
 CHILD: Это процесс потомок!
```

Листинг 22: forkStrace.log

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().

Дополнительно посмотрим лог вызовов программы с clone() (листинг 19).

```
psaer@ubuntu:~/Desktop/syscalls/fork$ strace ./clone.o
execve("./clone.o", ["./clone.o"], [/* 67 vars */]) = 0
brk(NULL)                               = 0xb75000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=90273, ...}) = 0
mmap(NULL, 90273, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f3e999ab000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...,
    ↪ 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f3e999aa000
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7f3e993d3000
mprotect(0x7f3e99593000, 2097152, PROT_NONE) = 0
mmap(0x7f3e99793000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_DENYWRITE, 3, 0x1c0000) = 0x7f3e99793000
mmap(0x7f3e99799000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_ANONYMOUS, -1, 0) = 0x7f3e99799000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f3e999a9000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f3e999a8000
arch_prctl(ARCH_SET_FS, 0x7f3e999a9700) = 0
mprotect(0x7f3e99793000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7f3e999c2000, 4096, PROT_READ) = 0
munmap(0x7f3e999ab000, 90273)           = 0
brk(NULL)                               = 0xb75000
brk(0xb98000)                           = 0xb98000
```

Листинг 23: cloneStrace.log

Как видно из логов, для fork() и clone() был использован именно системный вызов clone(). Это связано с тем, как **glibc** интерпретирует эти команды.

Теперь рассмотрим историю вызовов с прямым вызовом fork() по его номеру (программа из листинга 20).

```
psaer@ubuntu:~/Desktop/syscalls/fork$ strace ./fork2.o
execve("./fork2.o", ["./fork2.o"], [/* 60 vars */]) = 0
```

```
3  brk(NULL)                                = 0x12f8000
4  access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
5  access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or directory)
6  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7  fstat(3, {st_mode=S_IFREG|0644, st_size=90273, ...}) = 0
8  mmap(NULL, 90273, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff82eee1000
9  close(3)                                 = 0
10 access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
11 open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
12 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...,
       ↪ 832) = 832
13 fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
14 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
       ↪ x7ff82eee0000
15 mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
       ↪ x7ff82e909000
16 mprotect(0x7ff82eac9000, 2097152, PROT_NONE) = 0
17 mmap(0x7ff82ecc9000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
       ↪ MAP_DENYWRITE, 3, 0x1c0000) = 0x7ff82ecc9000
18 mmap(0x7ff82eccf000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
       ↪ MAP_ANONYMOUS, -1, 0) = 0x7ff82eccf000
19 close(3)                                 = 0
20 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
       ↪ x7ff82eedf000
21 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
       ↪ x7ff82eede000
22 arch_prctl(ARCH_SET_FS, 0x7ff82eedf700) = 0
23 mprotect(0x7ff82ecc9000, 16384, PROT_READ) = 0
24 mprotect(0x600000, 4096, PROT_READ)      = 0
25 mprotect(0x7ff82eef8000, 4096, PROT_READ) = 0
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
33 I'm child process, my pid is 0
34 write(1, "I'm parent process, my pid is 77"..., 35I'm parent process, my pid is
       ↪  7724
35 ) = 35
36 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=7724, si_uid=1000,
       ↪ si_status=0, si_utime=0, si_stime=0} ---
37 exit_group(0)                             = ?
38 +++ exited with 0 +++
```

Листинг 24: fork2Strace.log

Как видно, в строчке 32 была вызвана системная функция fork(), которая вернула значение 7724 (pid процесса потомка).

### 4.2.2 Анализ glibc

Проанализируем исходный код glibc, в данном случае программы компилировались используя **glibc 2.23**. Исходный код, различных версий доступен по ссылке[5].

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 25: 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 26: glibc_2.23/sysdeps/unix/sysv/linux/x86_64/arch-fork.h

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

Дополнительно рассмотрим исходный код glibc версии 2.15.90.

По пути **glibc_2.15/nptl/sysdeps/unix/sysv/linux/** имеется файл fork.c, в котором в строках 129-134 представлен вызов функции.

```
129 #ifdef ARCH_FORK
130   pid = ARCH_FORK ();
131 #else
132 # error "ARCH_FORK must be defined so that the CLONE_SETTID flag is used"
133   pid = INLINE_SYSCALL (fork, 0);
134 #endif
```

Листинг 27: glibc_2.15/nptl/sysdeps/unix/sysv/linux/fork.c

Реализация макроса представлена по пути **glibc_2.15/nptl/sysdeps/unix/sysv/linux/x86_64/** в файле fork.c.

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

Листинг 28: glibc_2.15/nptl/sysdeps/unix/sysv/linux/x86_64/fork.c

Версия 2.15 имеет уже 6 летнюю давность, и в ней также вызывался системный вызов clone(). Единственным отличием оказалось различное расположение файлов, с исходным кодом.

Если изучить прототипы fork() и clone(), то можно прийти к выводу что использование clone() началось для общего облегчения процессов в Linux системах, так как используя clone(), имеется возможность разделения между процессами:

- контекста;
- памяти;
- файловых дескрипторов;
- обработчиков сигналов.

Все это приводит к более гибкому созданию нового процесса, что fork() не позволяет.

Конечно можно более подробно проанализировать хронологию, но не не стоит отклоняться от основной темы данной работы.

### 4.2.3 Анализ исходного кода

Рассмотрим часть файла **syscall_64.tbl**, именно в данном файле определены системные вызовы.

```
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 29: .../arch/x86/syscalls/syscall_64.tbl

В строчке 65 для вызова clone() присвоен номер 56, а для fork() 57. Именно благодарю этому уникальному номеру, ранее удалось вызвать системный вызов напрямую.

Для различных архитектур, номера системных вызовов могут несколько отличаться.

Далее рассмотрим заголовочный файл **syscalls.h**.

```
837 #ifdef CONFIG_CLONE_BACKWARDS
838 asmlinkage long sys_clone(unsigned long, unsigned long, int __user *, unsigned
    ↪ long,
839             int __user *);
840 #else
841 #ifdef CONFIG_CLONE_BACKWARDS3
842 asmlinkage long sys_clone(unsigned long, unsigned long, int, int __user *,
843                 int __user *, unsigned long);
844 #else
845 asmlinkage long sys_clone(unsigned long, unsigned long, int __user *,
846             int __user *, unsigned long);
847 #endif
848 #endif
```

Листинг 30: .../include/linux/syscalls.h

В данном файле определены прототипы объявленных в таблице системных вызовов. Как минимум каждая из платформ должна реализовывать все объявленные параметры в прототипе, но также может и расширять их, при необходимости.

Исходный код(основная часть), находится в файле **fork.c**, по пути **/kernel**. Ввиду обилия кода(2467 строк), файл приложен к отчету, а далее представлено пояснение основных моментов.

Начало реализация представлено в строке 2006.

```
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
2017     /*
2018      * Determine whether and which event to report to ptracer.  When
2019      * called from kernel_thread or CLONE_UNTRACED is explicitly
2020      * requested, no event is reported; otherwise, report if the event
2021      * for the type of forking is enabled.
2022      */
2023     if (!(clone_flags & CLONE_UNTRACED)) {
2024         if (clone_flags & CLONE_VFORK)
2025             trace = PTRACE_EVENT_VFORK;
2026         else if ((clone_flags & CSIGNAL) != SIGCHLD)
2027             trace = PTRACE_EVENT_CLONE;
2028         else
2029             trace = PTRACE_EVENT_FORK;
2030
2031         if (likely(!ptrace_event_enabled(current, trace)))
2032             trace = 0;
2033     }
2034
2035     p = copy_process(clone_flags, stack_start, stack_size,
2036              child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
```

Листинг 31: .../kernel/fork.c

Аргументы функции:

1. **clone_flags** - флаг, для определения того, что именно нужно копировать;
2. **parent_tid и child_tid** - два указателя в пространстве пользователя, для хренения id родительского и дочернего процессов;
3. **stack_start** - адрес начала стека с процессами;
4. **tls** - определение локального хранилища для нового процесса

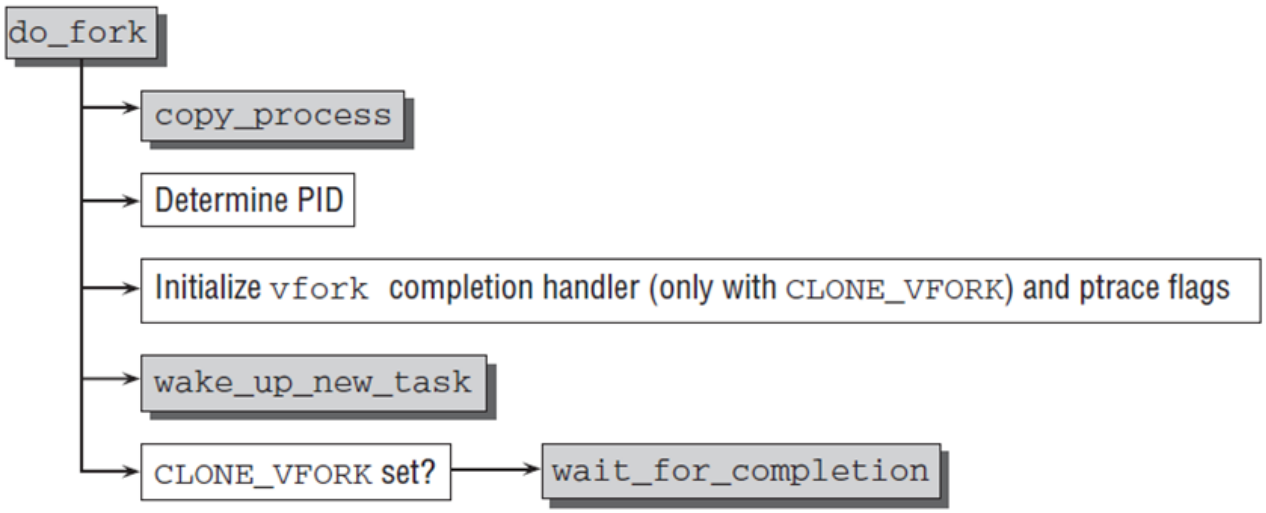


Рис. 2: Схема работы do_fork

Основным действием является вызов функции **(copy_process)**.

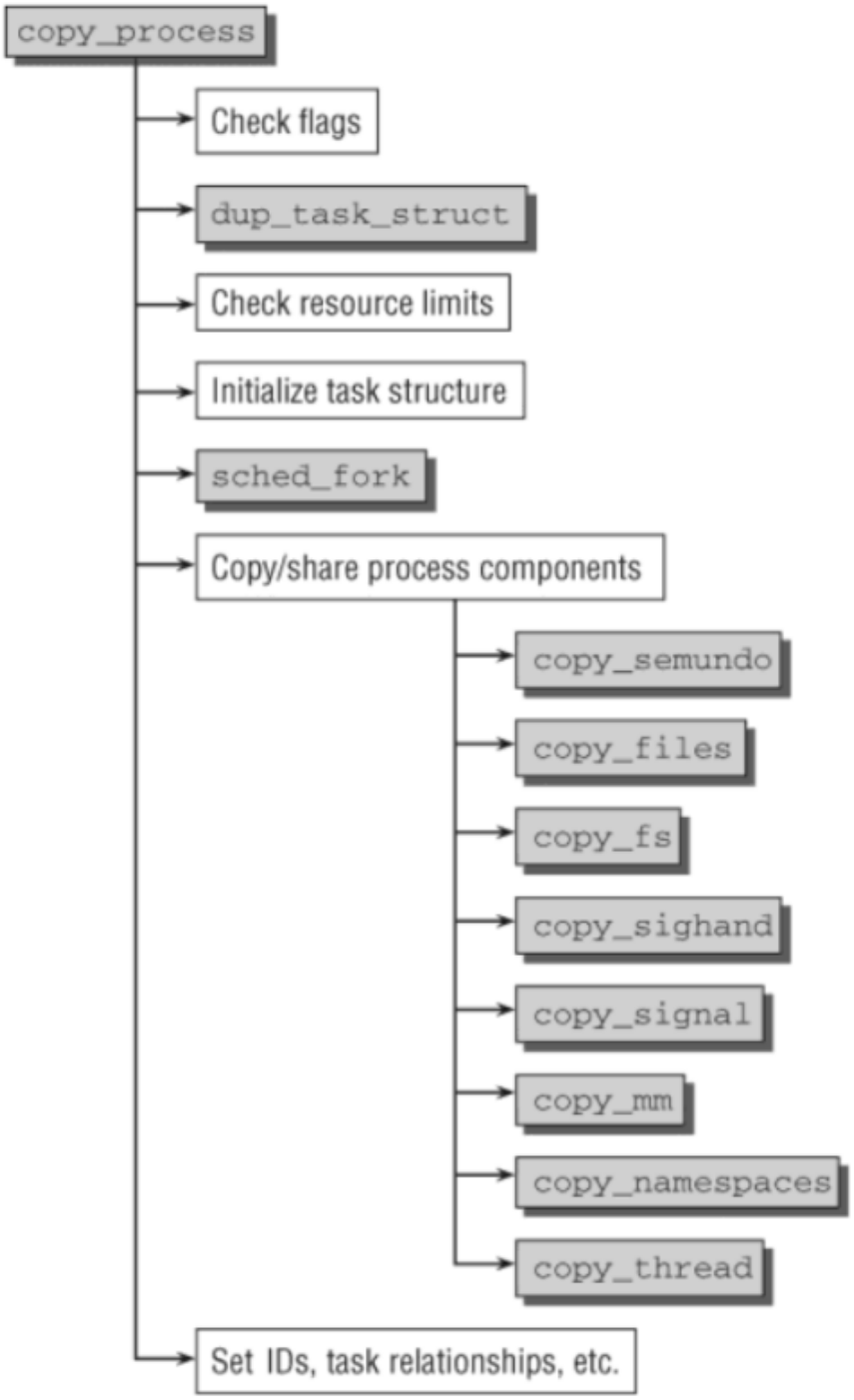


Рис. 3: Схема работы copy_process

**copy_process()**, также как и do_fork(), начинает свою работу спроверки на сочетание установленных флагов. Опишем некоторые проверки:

- если установлен флаг CLONE_THREAD, но не установлен флаг CLONE_SIGHAND, то возвращается ошибка, так как потоки группыдолжны разделять сигналы;
- если установлен флаг CLONE_SIGHAND, но не установлен флаг CLONE_VM, то воз-

вращается ошибка, так как общие обработчикисигналов, подразумевают общее адресное пространство.

Далее вызывается функция dup_task_struct() (определена в файле kernel/fork.c), которая создает новый экземпляр структуры task_struct и копирует в него различные дескрипторы, относящиеся к текущему процессу. Затем в copy_process производится проверка лимитов и полномочий, производится проверка на ”fork bomb”[6].

Выполняются вспомогательные действия, включающие инициализацию различных полейструктуры task_struct. Потом происходит вызов функций, выполняющих копирование составляющих процесса:

- таблицы дескрипторов открытых файлов (copy_files);
- таблицы сигналов и обработчиков сигналов(copy_signal и copy_sighand);
- адресного пространства процесса(copy_mm) и структуры thread_info (copy_thread).

Затем только что созданная задача назначается процессору из числа разрешенных для ее выполнения (cpus_allowed). После того как новый процесс унаследует приоритет родительского процесса, выполняется еще небольшое число вспомогательных действий и происходит возврат в do_fork().

По завершению **copy_process**, инициализируется обработчики для **vfork**(если флаг был передан). Далее идет вызов **wake_up_new_task**(помещает новый процесс в очередь выполнениями инициирует его выполнение), что является стандартным для всех новых процессов. И наконец, есть выставлен флаг на vfork, необходимо дождаться его завершения, прежде снова передать управление родительскому процессу.

## 4.3 Ядро версии 2.6.32-21-generic

### 4.3.1 Анализ strace

Ввиду полного совпадения(анализируемой функции) логов, по сравнению с логами ядра версии 4.13, их рассмотрение не требуется. Но они также приложены к данной работе.

### 4.3.2 Анализ исходного кода

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 {
1173     struct task_struct *p;
1174     int trace = 0;
1175     long pid;
1176 
1177     if (unlikely(current->ptrace)) {
1178         trace = fork_traceflag (clone_flags);
1179         if (trace)
1180             clone_flags |= CLONE_PTRACE;
1181     }
1182 
1183     p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr,
    ↪  child_tidptr);
```

Листинг 32: .../kernel/fork.c

Вся реализация уже описана для версии ядра 4.13, рассмотрим 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.

## 4.4 Перехват вызова

Для ядра версии 4.13 компиляция не удалась(что вполне ожидаемо), так как используемые функции и символы были убраны из экпорта ядра, то есть получить к ним доступа более не предоставляется возможным.

Как вариант, на новых версиях ядер, для перехвата можно использовать **LSM**(Linux Security Modules), но в моем случаем не все системные вызовы им поддерживаются. Поэтому, перехват вызовов будет предстален для версии ядра **2.6.32-21**.

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 33: .../kernel/asm-generic/syscalls.h

Для того, чтобы перехватить данную функцию, напишем 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 34: Функция перехвата fork()

Полный код программы приведен в приложении.

Для компиляции модуля ядра, необходим специальный 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
```

Листинг 35: Makefile

Выполним Makefile для компиляции модуля ядра:

```
psaer@ubuntu:~/Desktop/forkHook$ make
make -C /lib/modules/2.6.32-21-generic/build M=/home/psaer/Desktop/forkHook
make[1]: Entering directory '/usr/src/linux-headers-2.6.32-21-generic'
  CC      /home/psaer/Desktop/forkHook/libudis86/decode.o
  CC      /home/psaer/Desktop/forkHook/libudis86/itab.o
  CC      /home/psaer/Desktop/forkHook/libudis86/udis86.o
  LD      /home/psaer/Desktop/forkHook/libudis86/built-in.o
  LD      /home/psaer/Desktop/forkHook/built-in.o
  CC [M]  /home/psaer/Desktop/forkHook/module-init.o
  LD [M]  /home/psaer/Desktop/forkHook/hooking.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/psaer/Desktop/forkHook/hooking.mod.o
  LD [M]  /home/psaer/Desktop/forkHook/hooking.ko
make[1]: Leaving directory '/usr/src/linux-headers-2.6.32-21-generic'
```

Листинг 36: Лог сборки

Далее встроим модуль в ядро:

```
psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ sudo insmod hooking.ko
```

Листинг 37: Встраивание модуля в ядро

Далее, в другом окне терминала выполним программу с прямым вызовом fork, и посмотрим системный лог.

```
psaer@ubuntu:~/Desktop$ ./fork2.o
Invoking 'fork()' system call
I'm parent process, my pid is 4380
psaer@ubuntu:~/Desktop$ I'm child process, my pid is 0
psaer@ubuntu:~/Desktop$ tail /var/log/kern.log
Apr  7 11:58:34 ubuntu kernel: [27282.741397] [hooking] khook_inode_permission(
    ↪ ffff88003e732fc0,00000024) [rmmod] = 0
Apr  7 11:58:34 ubuntu kernel: [27282.741413] [hooking] khook_inode_permission(
    ↪ ffff88003e735b40,00000024) [rmmod]
Apr  7 11:58:34 ubuntu kernel: [27282.741414] [hooking] khook_inode_permission(
    ↪ ffff88003e735b40,00000024) [rmmod] = 0
Apr  7 11:58:34 ubuntu kernel: [27282.741633] [hooking] khook_inode_permission(
    ↪ ffff88003e7b11b0,00000024) [rmmod]
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 38: Лог с сообщением о перехвате

Как видно из лога, перехват не повлиял на работоспособность функции, а в системный лог, как и ожидлось было добавлено сообщение о перехвате.

Теперь выгружаем модуль из ядра.

```
1 psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ sudo rmmod hooking
```

Листинг 39: Выгрузка модуля из ядра

После выгрузки, перехватчик более не активен.

## 4.5 Общая иерархия вызовов

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

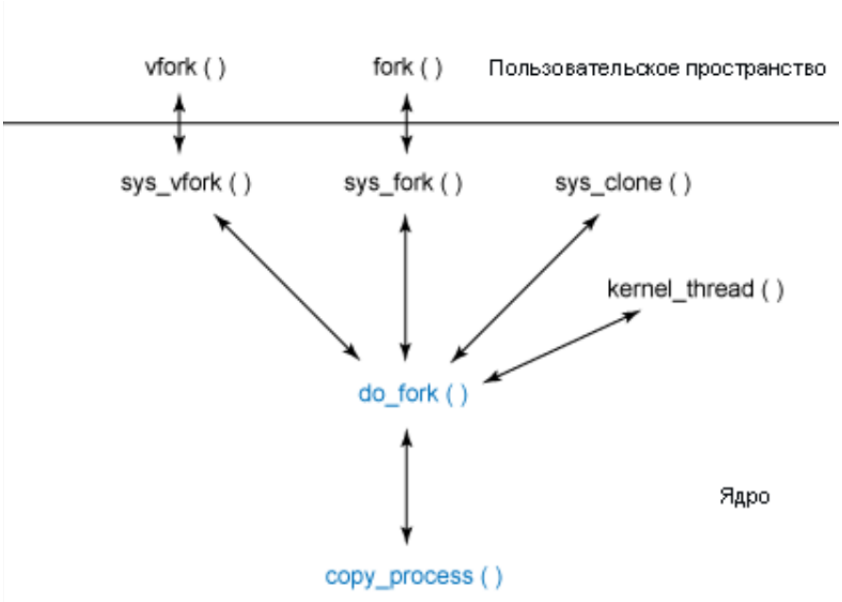


Рис. 4: Иерархия вызовов при создании процесса

# 5 Системная функция execve

**execve()**[7] - 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)
Аргементы:

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

## 5.1 Программы для анализа

Запуск программы произведен на версии ядра 4.13.

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 40: sys_execve.c

```
1  psaer@ubuntu:~/Desktop/execve$ ./sys_execve.o World!
2  Hello World!
3  0: XDG_VTNR=7
```

```
1: XDG_SESSION_ID=c2
2: CLUTTER_IM_MODULE=xim
3: XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/psaer
4: SESSION=ubuntu
5: GPG_AGENT_INFO=/home/psaer/.gnupg/S.gpg-agent:0:1
6: TERM=xterm-256color
7: SHELL=/bin/bash
8: XDG_MENU_PREFIX=gnome-
9: VTE_VERSION=4205
10: QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
11: WINDOWID=10485770
12: UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1703
13: GNOME_KEYRING_CONTROL=
14: GTK_MODULES=gail:atk-bridge:unity-gtk-module
15: USER=psaer
16: LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd
    ↪ =40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw
    ↪ =30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.
    ↪ arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.
    ↪ tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z
    ↪ =01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz
    ↪ =01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb
    ↪ =01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar
    ↪ =01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz
    ↪ =01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm
    ↪ =01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif
    ↪ =01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.
    ↪ pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.
    ↪ webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob
    ↪ =01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb
    ↪ =01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl
    ↪ =01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv
    ↪ =01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid
    ↪ =00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra
    ↪ =00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
17: QT_ACCESSIBILITY=1
18: XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
19: XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
20: SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
21: SESSION_MANAGER=local/ubuntu:@/tmp/.ICE-unix/1957,unix/ubuntu:/tmp/.ICE-
    ↪ unix/1957
22: DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
23: XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
24: PATH=/home/psaer/bin:/home/psaer/.local/bin:/usr/local/sbin:/usr/local/bin
    ↪ :/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

```
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
44: LANGUAGE=en_US
45: GNOME_DESKTOP_SESSION_ID=this-is-deprecated
46: XDG_SESSION_DESKTOP=ubuntu
47: LOGNAME=psaer
48: QT4_IM_MODULE=xim
49: XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share:/usr/
    ↪ share:/var/lib/snapd/desktop
50: DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-HBGGlqfjgR
51: LESSOPEN=| /usr/bin/lesspipe %s
52: INSTANCE=
53: XDG_RUNTIME_DIR=/run/user/1000
54: DISPLAY=:0
55: XDG_CURRENT_DESKTOP=Unity
56: GTK_IM_MODULE=ibus
57: LESSCLOSE=/usr/bin/lesspipe %s %s
58: XAUTHORITY=/home/psaer/.Xauthority
59: _=./sys_execve.o
```

Листинг 41: sys_execve.log

Из лога можно видеть множество системных констант, можно определить в каком каталоге был запуск, каким пользователем, на какой системе и многое другое.

## 5.2 Ядро версии 4.13.0-38-generic

### 5.2.1 Анализ strace

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 42: 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.

### 5.2.2 Анализ исходного кода

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 __user *__argv,
1830     const char __user *const __user *__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 43: ../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 44: ../fs/exec.c

Принцип работы представлен на схеме далее.

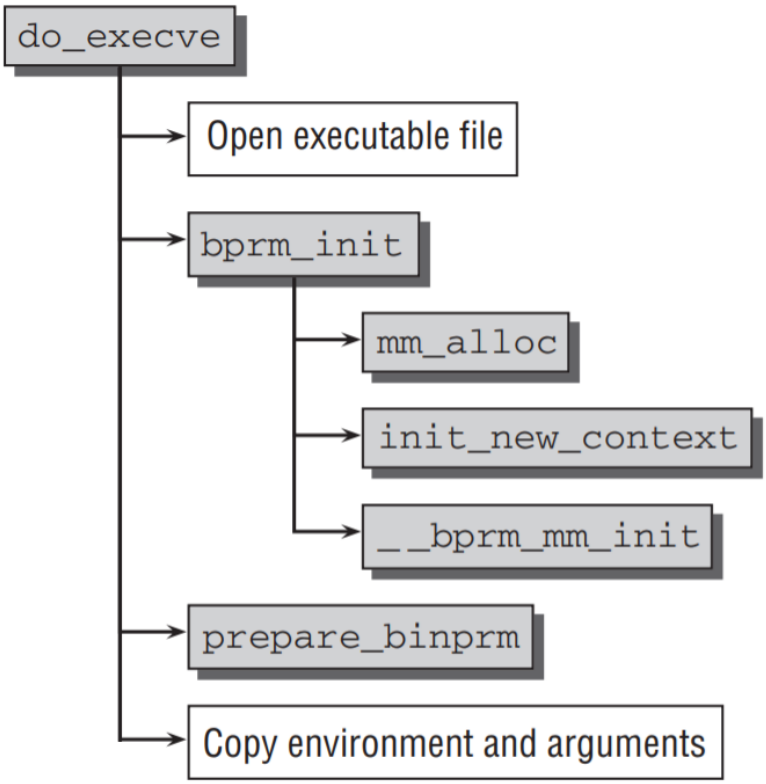


Рис. 5: Схема работы do_execve

1. Различные проверки корректности аргументов, открытие перадаваемого файла, получение его дескриптора;
2. Вызов функции **bprm_init**
   2.1. **mm_alloc** - генерация структуры **mm_struct**, для организации адресного пространства процесса;
   2.2. **init_new_context** - создание нового контекста для процесса;
   2.3. **__bprm_mm_init** - установка нового процесса в стек;
3. **prepare_binprm** - предоставления доступа к различным значениям(euid, egid, имя файла, окружение ...) родительского процесса(ов).

## 5.3 Ядро версии 2.6.32-21-generic

### 5.3.1 Анализ strace

Лог strace неотличается от версии ядра 4.13, за исключением того что, в массиве переменных окружения было передано 38 переменных вместо 60. Более подробно можно посмотреть в приложенных логах к данной работе.

### 5.3.2 Анализ исходного кода

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 45: ../fs/exec.c

В отличии 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**.

## 5.4 Перехват вызова

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 46: .../kernel/asm-generic/syscalls.h

Для того, чтобы перехватить данную функцию, напишем 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,
206     char __user * __user *argv,
207     char __user * __user *envp,
208     struct pt_regs *regs)
209 {
210     int result;
211 
212     KHOOK_USAGE_INC(sys_execve);
213 
214     printk("System call for execve hooked\n");
215     printk("Executed file: %s\n", filename);
216 
217     result = KHOOK_ORIGIN(sys_execve, filename, argv, envp, regs);
218 
219     KHOOK_USAGE_DEC(sys_execve);
220 
221     return result;
222 }
```

Листинг 47: Функция перехвата execve()

Помимо сообщения о перехвате, выводим название программы, запуск которой был произведен. Полный код программы приведен в приложении.

Выполним Makefile для компиляции модуля ядра:

```
1 make -C /lib/modules/2.6.32-21-generic/build M=/home/psaer/Desktop/new/
    ↪ kmod_hooking-master
2 make[1]: Entering directory '/usr/src/linux-headers-2.6.32-21-generic'
3   CC [M]  /home/psaer/Desktop/new/kmod_hooking-master/module-init.o
4   LD [M]  /home/psaer/Desktop/new/kmod_hooking-master/hooking.o
5   Building modules, stage 2.
6   MODPOST 1 modules
7   CC      /home/psaer/Desktop/new/kmod_hooking-master/hooking.mod.o
8   LD [M]  /home/psaer/Desktop/new/kmod_hooking-master/hooking.ko
9 make[1]: Leaving directory '/usr/src/linux-headers-2.6.32-21-generic'
```

Листинг 48: Лог сборки

Далее встроим модуль в ядро:

```
1 psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ sudo insmod hooking.ko
```

Листинг 49: Лог сборки

Далее, в другом окне терминала выполним программу, представленную в листинге 40, и посмотрим системный лог.

```
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 50: Системный лог

Как видно из лога, функция была успешно перехвачена, было выведено соответствующее сообщение и имя файла для запуска, в данном случае **./sys_execve.o**.

Теперь выгружаем модуль из ядра.

```
1  psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ sudo rmmod hooking
```

Листинг 51: Выгрузка модуля из ядра

После выгрузки, перехватчик более не активен.

# 6 Системная функция exit

**exit()**[8] - 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)
Аргементы:

- **error_code** - код выхода.

## 6.1 Программы для анализа

Запуск программы произведен на версии ядра 4.13.

Программа выводит в консоль два сообщения, одно до, а другое после системного вызова exit, по коду 60 (системный номер функции, взят из листинга 29).

```
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 52: sys_exit.c

```
1  psaer@ubuntu:~/Desktop/exit$ ./sys_exit.o
2  Invoking 'exit()' system call
```

Листинг 53: sys_exit.log

Как и ожидалось, после системного вызова exit, никаких сообщений выведено не было.

## 6.2 Ядро версии 4.13.0-38-generic

### 6.2.1 Анализ strace

```
1  psaer@ubuntu:~/Desktop/exit$ strace ./sys_exit.o
2  execve("./sys_exit.o", ["./sys_exit.o"], [/* 60 vars */]) = 0
3  brk(NULL)                               = 0x92e000
4  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
5  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
6  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7  fstat(3, {st_mode=S_IFREG|0644, st_size=90273, ...}) = 0
```

```
 8 mmap(NULL, 90273, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4ba908f000
 9 close(3)                                = 0
10 access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
11 open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
12 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...,
    ↪ 832) = 832
13 fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
14 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f4ba908e000
15 mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7f4ba8ab7000
16 mprotect(0x7f4ba8c77000, 2097152, PROT_NONE) = 0
17 mmap(0x7f4ba8e77000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_DENYWRITE, 3, 0x1c0000) = 0x7f4ba8e77000
18 mmap(0x7f4ba8e7d000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_ANONYMOUS, -1, 0) = 0x7f4ba8e7d000
19 close(3)                                = 0
20 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f4ba908d000
21 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7f4ba908c000
22 arch_prctl(ARCH_SET_FS, 0x7f4ba908d700) = 0
23 mprotect(0x7f4ba8e77000, 16384, PROT_READ) = 0
24 mprotect(0x600000, 4096, PROT_READ)     = 0
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 54: 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 систеного вызова exit, после которого, никаких других сисмных вызовов не последовало.

### 6.2.2 Анализ исходного кода

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 55: ../kernel/exit.c

По ходу выполнения производятся следующие действия:

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. Это такое состояние, когда процесс занимает только элемент в таблице процессов и не занимает памяти ни в адресном пространстве пользователя, ни в адресном пространстве ядра. 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.

Все присоединенные разделяемые сегменты памяти отсоединяются и в связанных с ними структурах данных значения полей shm_nattach уменьшаются на 1.

Родительскому процессу посылается сигнал SIGCLD (завершение порожденного процесса).

## 6.3 Ядро версии 2.6.32-21-generic

### 6.3.1 Анализ strace

Лог strace неотличается от версии ядра 4.13. Лог приложен к работе.

### 6.3.2 Анализ исходного кода

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))
803         panic("Attempted to kill the idle task!");
804     if (unlikely(tsk->pid == 1))
805         panic("Attempted to kill init!");
806     if (tsk->io_context)
807         exit_io_context();
808     tsk->flags |= PF_EXITING;
809     del_timer_sync(&tsk->real_timer);
```

Листинг 56: ../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.

## 6.4 Перехват вызова

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

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

Листинг 57: .../kernel/linux/syscalls.h

Для того, чтобы перехватить данную функцию, напишем 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 58: Функция перехвата exit()

При успешном перехвате, в системный лог добавляется запись о успешном перехвате.

Выполним Makefile для компиляции модуля ядра:

```
1  psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ make
2  make -C /lib/modules/2.6.32-21-generic/build M=/home/psaer/Desktop/new/
       ↪ kmod_hooking-master
3  make[1]: Entering directory '/usr/src/linux-headers-2.6.32-21-generic'
4    CC [M]  /home/psaer/Desktop/new/kmod_hooking-master/module-init.o
5    LD [M]  /home/psaer/Desktop/new/kmod_hooking-master/hooking.o
6    Building modules, stage 2.
7    MODPOST 1 modules
8    CC      /home/psaer/Desktop/new/kmod_hooking-master/hooking.mod.o
9    LD [M]  /home/psaer/Desktop/new/kmod_hooking-master/hooking.ko
10 make[1]: Leaving directory '/usr/src/linux-headers-2.6.32-21-generic'
```

Листинг 59: Лог сборки

Далее встроим модуль в ядро:

```
1  psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ sudo insmod hooking.ko
```

Листинг 60: Лог сборки

Далее, в другом окне терминала выполним программу, представленную в листинге 52, и посмотрим системный лог.

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

Листинг 61: Системный лог

Как видно из лога, функция была успешно перехвачена.

Теперь выгружаем модуль из ядра.

```
psaer@ubuntu:~/Desktop/new/kmod_hooking-master$ sudo rmmod hooking
```

Листинг 62: Выгрузка модуля из ядра

После выгрузки, перехватчик более не активен.

**Примечание:** после попытки выгрузки модуля, консоль в которой производилась работа перестает отвечать на команды, а через некоторое время и сама система зависает.

# 7 Модификация системных вызовов

Модификации будут применены относительно исходного кода ядра версии 4.13.0.

## 7.1 Вносимые модификации

В виде примера модификации, будет производится запись сообщения о вызове функции в системный лог. А для **execve** также вывод имени запускаемого файла.

### 7.1.1 Системная функция fork

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 63: Модифицированный fork

### 7.1.2 Системная функция execve

Модификации подверглась функция **do_execveat_common** из файла **/fs/exec.c**.

```
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 64: Модифицированный exec

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

### 7.1.3 Системная функция exit

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 65: Модифицированный exit

## 7.2 Перекомпиляция ядра

Для начала необходимо скачать исходный код интересующей версии ядра, в данном случае это также ядро 4.13.0.

Далее необходимо установить некоторые пакеты следующей командой:

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

Листинг 66: Установка недостающий пакетов

Архив с исходным кодом был распакован по пути **/usr/src/**. Перейдем в эту папку и выполним команду:

```
1 sudo make menuconfig
```

Листинг 67: Конфигурация ядра

После выполнения данной команды, в консоли откроется меню для конфигруации ядра.

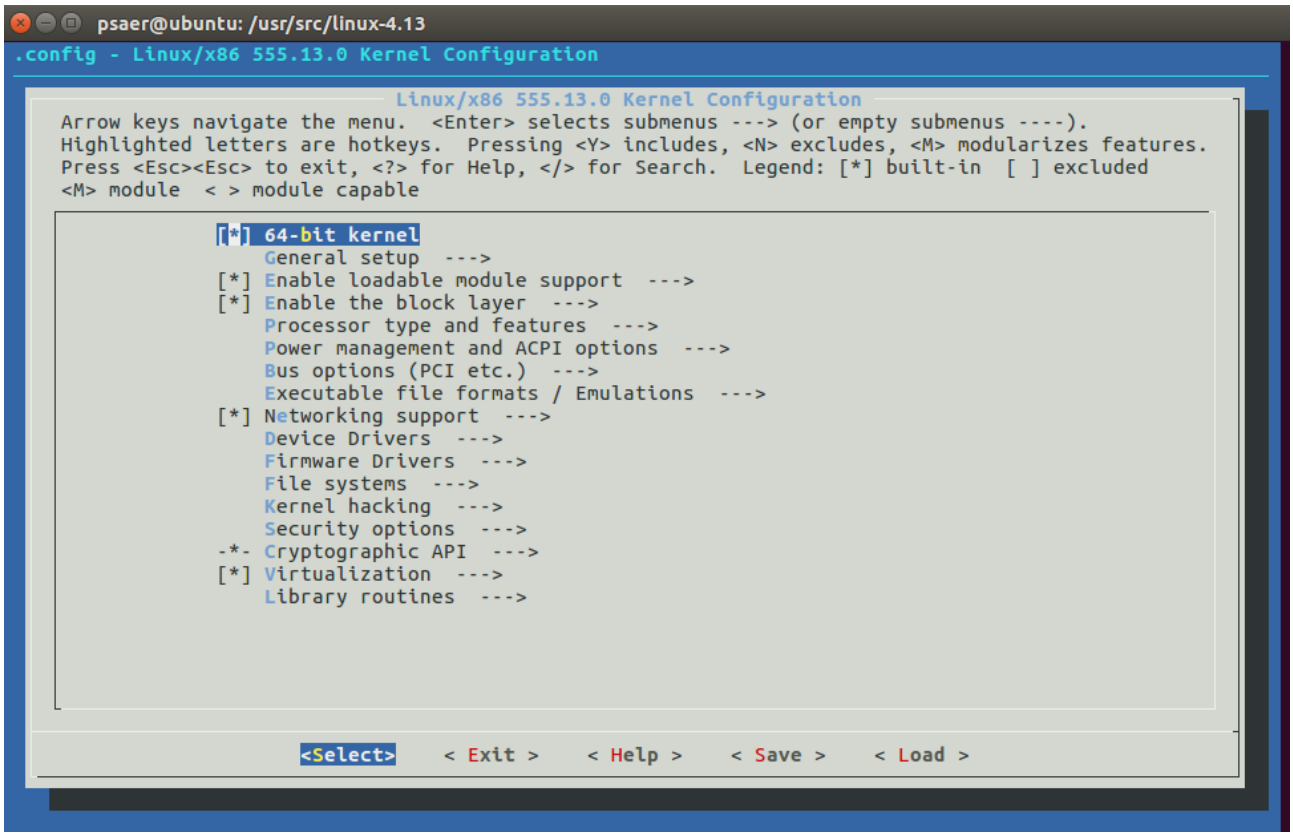


Рис. 6: Конфигурация ядра

В данном случае, в нем нет необходимости, закрываем его.

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 68: 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 69: 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 занимиает 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`
11 GRUB_CMDLINE_LINUX_DEFAULT="quiet"
12 GRUB_CMDLINE_LINUX="find_preseed=/preseed.cfg auto noprompt priority=critical
       ↪ locale=en_US"
13
14 # Uncomment to enable BadRAM filtering, modify to suit your needs
15 # This works with Linux (no patch required) and with any kernel that obtains
16 # the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)
17 #GRUB_BADRAM="0x01234567,0xfefefefe,0x89abcdef,0xefefefef"
18
19 # Uncomment to disable graphical terminal (grub-pc only)
20 #GRUB_TERMINAL=console
21
22 # The resolution used on graphical terminal
23 # note that you can use only modes which your graphic card supports via VBE
24 # you can see them in real GRUB with the command `vbeinfo'
25 #GRUB_GFXMODE=640x480
26
27 # Uncomment if you don't want GRUB to pass "root=UUID=xxx" parameter to Linux
28 #GRUB_DISABLE_LINUX_UUID=true
29
30 # Uncomment to disable generation of recovery mode menu entries
31 #GRUB_DISABLE_RECOVERY="true"
32
33 # Uncomment to get a beep at grub start
```

```
34 #GRUB_INIT_TUNE="480 440 1"
```

Листинг 70: Файл grub

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

После этого перезагружаем систему.

После старта виртуальной машины, будет показано меню GRUB.

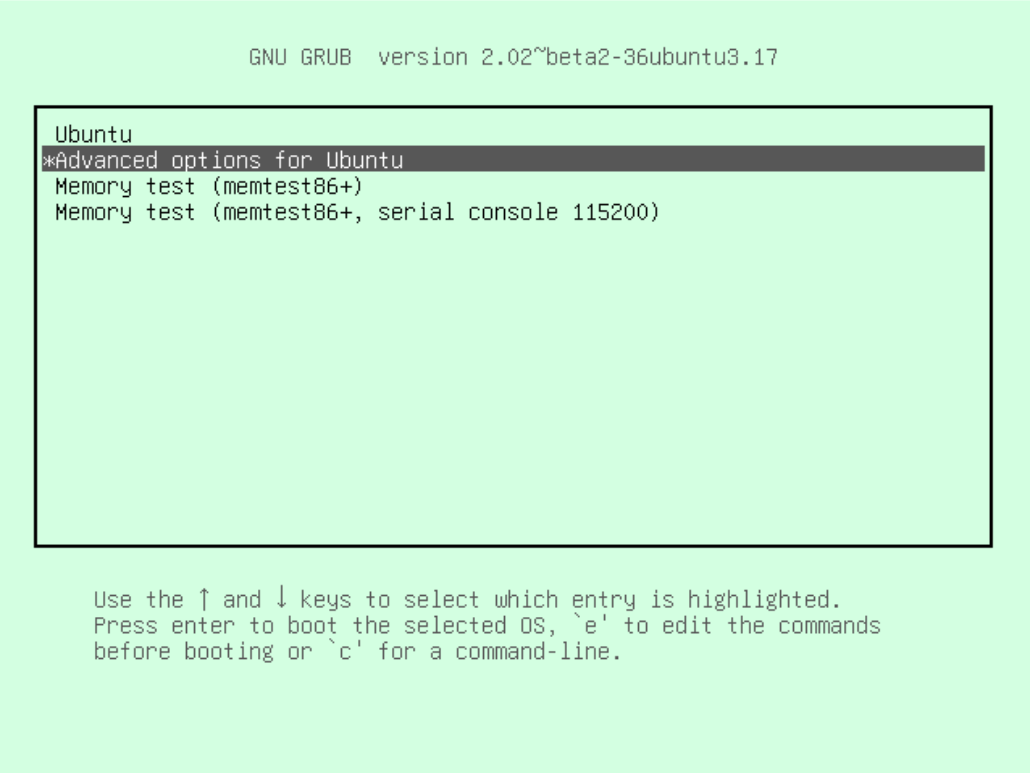


Рис. 7: Меню GRUB

Выбираем пункт **Advance options for Ubuntu** и нажимаем enter. Будет выведен список с воможными ядрами для загрузки.

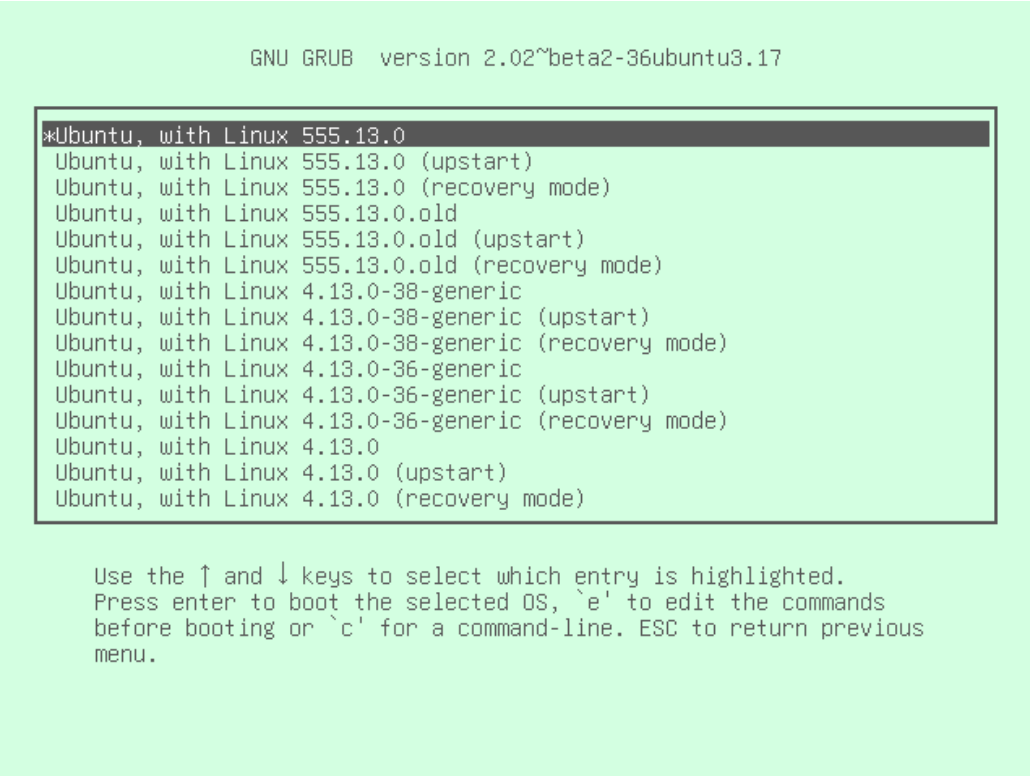


Рис. 8: Выбор ядра в GRUB

Выбираем скомпилированное ядра версии 555.13.0. В случае корректно скомпилированного ядра, система успешно загрузится.

Дополнительно, после загрузки ОС, можно проверить версию ядра.

```
1 psaer@ubuntu:~$ uname -r
2 555.13.0
```

Листинг 71: Версия ядра

Теперь можно приступить к тестированию.

## 7.3 Проверка модификации

Выполним программу из листинга 20.

```
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 72: Выполнением программы с прямым вызовом fork

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

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

Листинг 73: Команда для поиска некоторого сообщения

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

```
/var/log/syslog:157:Apr 10 09:55:15 ubuntu kernel: [  516.073716] Modified fork
    ↪ system call
/var/log/syslog:159:Apr 10 09:55:15 ubuntu kernel: [  516.079235] Modified fork
    ↪ system call
/var/log/syslog:164:Apr 10 09:55:34 ubuntu kernel: [  534.641490] Modified fork
    ↪ system call
/var/log/syslog:165:Apr 10 09:55:34 ubuntu kernel: [  534.641698] Modified fork
    ↪ system call
/var/log/syslog:168:Apr 10 09:55:37 ubuntu kernel: [  538.214193] Modified fork
    ↪ system call
/var/log/syslog:169:Apr 10 09:55:38 ubuntu kernel: [  538.214392] Modified
    ↪ system call from exec. File: ./fork.o
/var/log/syslog:170:Apr 10 09:55:38 ubuntu kernel: [  538.214810] Modified fork
    ↪ system call
/var/log/syslog:173:Apr 10 09:55:40 ubuntu kernel: [  540.848736] Modified fork
    ↪ system call
/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 74: Результаты поиска fork

Ввиду обилия вывода, приведена лишь часть, из которой видно:

1. Системный вызов **fork** успешно произовит записи в системный лог, причем, судя по логу, он был вызван системой при загрузки множество раз;
2. В строчке 127, была обнаружена запись от модифицированного вызова **exec**;
3. В строчке 130, была обнаружена запись о ранее вводимой команде, то есть в систему встроен некоторый логгер пользовательских команд.

Теперь проверим модификацию вызова **exit**, с помощью утилиты, приведенной в листинге 52

```
psaer@ubuntu:~/Desktop/exit$ ./sys_exit.o
Invoking 'exit()' system call
```

Листинг 75: sys_exit.log

```
psaer@ubuntu:~/Desktop/sysUtils$ sudo grep -rnw '/var/log/' -e 'exit' | tail
[sudo] password for psaer:
/var/log/syslog:603:Apr 10 10:20:30 ubuntu kernel: [ 2030.094680] Modified exit
    ↪ system call. Code: 0
/var/log/syslog:604:Apr 10 10:20:32 ubuntu kernel: [ 2030.207164] Modified exit
    ↪ system call. Code: 0
/var/log/syslog:605:Apr 10 10:20:32 ubuntu kernel: [ 2032.802104] Modified exit
    ↪ system call. Code: 0
/var/log/syslog:607:Apr 10 10:20:32 ubuntu kernel: [ 2032.809867] Modified
    ↪ system call from exec. File: ./exit.o
/var/log/syslog:608:Apr 10 10:20:35 ubuntu kernel: [ 2032.810497] Modified exit
    ↪ system call. Code: 4096
/var/log/syslog:613:Apr 10 10:20:35 ubuntu kernel: [ 2035.897177] Modified exit
    ↪ system call. Code: 0
/var/log/syslog:614:Apr 10 10:20:36 ubuntu kernel: [ 2035.897348] Modified exit
    ↪ system call. Code: 0
/var/log/syslog:617:Apr 10 10:20:36 ubuntu kernel: [ 2036.465294] Modified exit
    ↪ system call. Code: 0
/var/log/syslog:618:Apr 10 10:20:43 ubuntu kernel: [ 2036.465335] Modified exit
    ↪ system call. Code: 0
/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 76: Результаты поиска exit

Как и ожидалось, в системный лог были добавлены соответствующие записи.

# Вывод

В данной работе был рассмотрен перехват системных функций для разных весрий ядер.

По ходу работу, для ядра версии 4.13 (достаточно свежее ядро), предпринималось множество попыток по перехвату системных функций, но не одна из них не увенчалась успехом. Это впринципе и ожидаемом, так как с версии ядра 2.6 началась активная защита ядра, от подобных действий в том числе.

Основные проблемы, при попытке перехвата на новой версии ядра:

1. невозможность экспорта многих системных функций и символов;
2. области, например таблица системных вызовов защищена от записи.

В данной работе это не рассматривалось, но подобные проблемы можно обойти слеющими способами:

1. Использовать LSM - некий встроенный в ядро перехватчик.
    - Может перехватить далеко не все функции.
2. Полностью перекомпилировать ядро, с изменением критические важных для перехва участков кода.
    - Данный метод требует крайне много времени, хорошего понимания структуры ядра и понимания что конкретно нужно изменить.

Однако на версии ядра 2.6.32 не так все сурово по степени защиты, и удалось произвести перехват всех заданных системных функций.

Модификация исходного показала простоту всей перекомпиляцию ядра, буквально в несколько команд возможно загрузить новое ядро, скомпилировать его и загрузиться с ним.

Однако, могут возникнуть проблемы, если внесенные изменения, в исходный код чего-либо помещает корректной загрузки ОС.

# Список литературы


[1] Writing a Linux Kernel Module. — URL: http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/ (дата обращения: 2018-04-06).

[2] Встраивание в ядро Linux: перехват функций. — URL: https://habrahabr.ru/company/securitycode/blog/237089/ (дата обращения: 2018-04-07).

[3] fork(2) - Linux man page. — URL: https://linux.die.net/man/2/fork (дата обращения: 2018-04-01).

[4] clone(2) - Linux man page. — URL: https://linux.die.net/man/2/clone (дата обращения: 2018-04-01).

[5] glibc, archive of versions. — URL: https://ftp.gnu.org/gnu/glibc/ (дата обращения: 2018-04-04).

[6] Fork bomb. — URL: https://en.wikipedia.org/wiki/Fork_bomb (дата обращения: 2018-04-07).

[7] execve - execute program. — URL: http://man7.org/linux/man-pages/man2/execve.2.html (дата обращения: 2018-04-08).

[8] exit – terminate process. — URL: http://man.cat-v.org/unix-1st/2/sys-exit (дата обращения: 2018-04-08).

[9] Анатомия управления процессами в Linux. — URL: https://www.ibm.com/developerworks/ru/library/l-linux-process-management/index.html (дата обращения: 2018-04-07).