

Оглавление

| | |
|---|-----------|
| Введение | 3 |
| 1 Аналитическая часть | 4 |
| 1.1 Постановка задачи | 4 |
| 1.2 Перехват функций | 4 |
| 1.2.1 Linux Security Module | 5 |
| 1.2.2 Модификация таблицы системных вызовов | 5 |
| 1.2.3 kprobes | 6 |
| 1.2.4 Kernel tracepoints | 7 |
| 1.2.5 ftrace | 7 |
| 1.3 Информация о процессах и памяти | 9 |
| 1.3.1 Структура <code>struct task_struct</code> | 9 |
| 1.3.2 Структура <code>struct sysinfo</code> | 11 |
| 1.4 Загружаемые модули ядра | 12 |
| 1.4.1 Пространство ядра и пользователя | 12 |
| 1.5 Виртуальная файловая система <code>/proc</code> | 13 |
| 2 Конструкторская часть | 16 |
| 2.1 Архитектура приложения | 16 |
| 2.2 Структура <code>struct ftrace_hook</code> | 16 |
| 2.3 Алгоритм перехвата системного вызова | 17 |
| 2.4 Алгоритм подсчёта количества системных вызовов | 20 |
| 3 Технологическая часть | 22 |
| 3.1 Выбор языка программирования | 22 |
| 3.2 Поиск адреса перехватываемой функции | 22 |
| 3.3 Инициализация <code>ftrace</code> | 23 |
| 3.4 Функции обёртки | 25 |
| 3.5 Получение информации о количестве системных вызовов | 26 |
| 3.6 Информация о памяти в системе | 27 |
| 3.7 Получение информации о процессах | 28 |
| 3.8 Примеры работы разработанного ПО | 30 |

| | |
|---------------------|-----------|
| Заключение | 33 |
| Литература | 34 |
| Приложение А | 36 |

Введение

В настоящее время большую актуальность имеют системы, предоставляющие информацию о ресурсах операционной системы и частоте системных вызовов. Предоставив такую информацию, пользователь сможет проанализировать состояние системы и нагрузку на неё. Особое внимание уделяется операционным системам с ядром Linux [1]. Ядро Linux возможно изучать благодаря тому что оно имеет открытый исходный код.

Данная работа посвящена исследованию структур ядра, хранящим информацию о процессах в системе и памяти, способам перехвата системных вызовов ядра с их последующим логированием.

Целью данной курсовой работы является разработка загружаемого модуля ядра, предоставляющего информацию о количестве системных вызовов и выделенной памяти за выбранный промежуток времени и информацию о состоянии всех процессов в системе в текущий момент.

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с заданием необходимо разработать загружаемый модуль ядра, который позволит посмотреть количество системных вызовов, свободной и доступной оперативной памяти за выбранный промежуток времени, а так же количество процессов и их состояния на данный момент.

Для решение данной задачи необходимо:

- проанализировать различные подходы к перехвату функций;
- исследовать структуры и функции ядра, предоставляющие информацию о процессах и памяти;
- изучить методы передачи информации из пространства ядра в пространство пользователя и наоборот;
- спроектировать и реализовать загружаемый модулей ядра.

1.2 Перехват функций

Перехват функции заключается в изменении некоторого адреса в памяти процесса или кода в теле функции таким образом, чтобы при вызове этой самой функции управление передавалось не ей, а функции, которая будет её подменять. Эта функция, работая вместо системной, выполняет какие-то запланированные действия, и затем, либо вызывает оригинальную функцию, либо не вызывает ее вообще.

Далее будут рассмотрены существующие различные подходы к перехвату вызываемых функций и выбран наиболее подходящий для реализации в данной работе.

1.2.1 Linux Security Module

Linux Security Module (LSM) [2] – это специальный интерфейс, созданный для перехвата функций. В критических местах кода ядра расположены вызовы security-функций, которые вызывают коллбеки (англ. callback [3]), установленные security-модулем. Данный модуль может изучать контекст операции и принимать решение о её разрешении или запрете [2].

Особенности рассматриваемого интерфейса:

- security-модули являются частью ядра и не могут быть загружены динамически;
- в стандартной конфигурации сборки ядра флаг наличия LSM неактивен – большинство уже готовых сборок ядра не содержат внутри себя интерфейс LSM;
- в системе может быть только один security-модуль [2].

Таким образом, для использования Linux Security Module необходимо поставлять собственную сборку ядра Linux, что является трудоёмким вариантом – как минимум, придётся тратить время на сборку ядра. Кроме того, данный интерфейс обладает излишним функционалом (например решение о блокировке какой-либо операции), который не потребуется в написании разрабатываемого модуля ядра.

1.2.2 Модификация таблицы системных вызовов

Все обработчики системных вызовов расположены в таблице `sys_call_table`. Подмена значений в этой таблице приведёт к смене поведения всей системы. Сохранив старое значение обработчика и подставив в таблицу собственный обработчик, можно перехватить любой системный вызов.

Особенности данного подхода:

- минимальные накладные расходы;

- не требуется специальная конфигурация ядра;
- техническая сложность реализации – необходимо модифицировать таблицу системных вызовов;
- из-за ряда оптимизаций, реализованных в ядре, некоторые обработчики невозможно перехватить [4];
- можно перехватить только системные вызовы – нельзя перехватить обычные функции.

1.2.3 kprobes

kprobes [5] – специальный интерфейс, предназначенный для отладки и трассировки ядра. Данный интерфейс позволяет устанавливать пред- и пост-обработчики для любой инструкции в ядре, а так же обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут изменять их значение. Таким образом, **kprobes** можно использовать как и в целях мониторинга, так и для возможности повлиять на дальнейший ход работы ядра [4].

Особенности рассматриваемого интерфейса:

- перехват любой инструкции в ядре – это реализуется с помощью точек останова (инструкция `int3`), внедряемых в исполняемый код ядра. Таким образом, можно перехватить любую функцию в ядре;
- хорошо задокументированный API;
- нетривиальные накладные расходы – для расстановки и обработки точек останова необходимо большое количество процессорного времени [4];
- техническая сложность реализации. Так, например, чтобы получить аргументы функции или значения её локальных переменных нужно знать, в каких регистрах, или в каком месте на стеке они находятся, и самостоятельно их оттуда извлекать;

- при подмене адреса возврата из функции используется стек, реализованный с помощью буфера фиксированного размера. Таким образом, при большом количестве одновременных вызовов перехваченной функции, могут быть пропущены срабатывания.

1.2.4 Kernel tracepoints

`Kernel tracepoints` [6] – это фреймворк для трассировки ядра, реализованный через статическое инструментирование кода. Большинство важных функций ядра статически инструментировано – в теле функций добавлены вызовы функций фреймворка рассматриваемого фреймворка.

Особенности рассматриваемого фреймворка:

- минимальные накладные расходы – необходимо только вызвать функцию трассировки в необходимом месте;
- отсутствие задокументированного API;
- не все функции ядра статически инструментированны;
- не работает, если ядро не сконфигурировано должным образом [4].

1.2.5 ftrace

`ftrace` [7] – это фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора `-pg` [8] и `mentry` [8]. Данные функции вставляют в начало каждой функции вызов специальной трассировочной функции `mcount()` или `__fentry()`. В пользовательских программах данная возможность компилятора используется профилировщиками, с целью отслеживания всех вызываемых функций. В ядре эти функции используются исключительно для реализации рассматриваемого фреймворка.

Для большинства современных архитектур процессора доступна оптимизация: динамический `frace` [8]. Ядро знает расположение всех вызовов функций `mcount()` или `__fentry()` и на ранних этапах загрузки ядра

подменяет их машинный код на специальную машинную инструкцию NOP [9], которая ничего не делает. При включении трассировки, в нужные функции необходимые вызовы добавляются обратно. Если **ftrace** не используется, его влияние на производительность системы минимально.

Особенности рассматриваемого фреймворка:

- имеется возможность перехватить любую функцию;
- перехват совместим с трассировкой;
- фреймворк зависит от конфигурации ядра, но, в популярных конфигурациях ядра (и, соответственно, в популярных образах ядра) установлены все необходимые флаги для работы;

Сравнение методов

В таблице 1.1 приведено сравнение приведенных выше методов, позволяющих перехватывать системные вызовы.

| Название | Дин. загрузка | Перехват любых функций | Любая конфигурация ядра | Простота реализации | Наличие документации |
|---------------------------------------|---------------|------------------------|-------------------------|---------------------|----------------------|
| Linux Security Module | Нет | Да | Нет | Нет | Нет |
| Модификация таблицы системных вызовов | Да | Нет | Да | Нет | Нет |
| kprobes | Да | Да | Да | Нет | Да |
| kernel tracepoints | Да | Да | Нет | Да | Нет |
| ftrace | Да | Да | Нет | Да | Да |

Таблица 1.1: Методы перехвата системных вызовов

1.3 Информация о процессах и памяти

1.3.1 Структура `struct task_struct`

Информация о процессах в ядре хранится с помощью специальной структуры `struct task_struct` [10]. Каждому процессу в системе соответствует структура `task_struct`, которая полностью описывает процесс. Сами структуры связаны друг с другом по средствам кольцевого связанного списка.

Структура описывает текущее состояние процесса, его флаги, указатель на процессы-потомки и так далее. Стоит отметить, что для описания потоков, в ядре Linux так же используется данная структура – различие лишь в установленных флагах. В листинге 1.1 представлено объявление структуры с наиболее интересными полями.

```
1 struct task_struct {
2     #ifdef CONFIG_THREAD_INFO_IN_TASK
3     struct thread_info    thread_info;
4     #endif
5
6     unsigned int          __state;
7     ...
8     unsigned int          flags;
9     ...
10    #ifdef CONFIG_SMP
11    int                    on_cpu;
12    ...
13    int                    recent_used_cpu;
14    #endif
15    ...
16    int                    recent_used_cpu;
17    ...
18    #ifdef CONFIG_CGROUP_SCHED
19    struct task_group      *sched_task_group;
20    #endif
21    ...
22    struct sched_info      sched_info;
23    ...
24    struct list_head        tasks;
25    ...
26 }
```

Листинг 1.1: Листинг структуры `task_struct` с наиболее интересными полями

Для работы с данной структурой внутри ядра объявлен ряд макросов. Например, чтобы обойти все процессы в системе, существует макрос `for_each_process`, который итерируется по связанному списку процессов. Состояния процесса так же описываются с помощью специальных макросов. Кроме того, существует ряд макросов, позволяющих проверить текущее состояние процесса, например, узнать, выполняется ли процесс в данный момент. Список этих макросов приведён в листингах 1.2 - 1.3.

```
1 #define TASK_RUNNING      0x0000
2 #define TASK_INTERRUPTIBLE 0x0001
3 #define TASK_UNINTERRUPTIBLE 0x0002
4 #define __TASK_STOPPED    0x0004
5 #define __TASK_TRACED     0x0008
6 #define EXIT_DEAD        0x0010
7 #define EXIT_ZOMBIE      0x0020
8 #define EXIT_TRACE       (EXIT_ZOMBIE | EXIT_DEAD)
9 #define TASK_PARKED      0x0040
10 #define TASK_DEAD        0x0080
11 #define TASK_WAKEKILL    0x0100
12 #define TASK_WAKING      0x0200
13 #define TASK_NOLOAD      0x0400
14 #define TASK_NEW         0x0800
15 #define TASK_RTLOCK_WAIT 0x1000
16 #define TASK_STATE_MAX   0x2000
17 #define TASK_KILLABLE     (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
18 #define TASK_STOPPED      (TASK_WAKEKILL | __TASK_STOPPED)
19 #define TASK_TRACED       (TASK_WAKEKILL | __TASK_TRACED)
20 #define TASK_IDLE         (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
21 #define TASK_NORMAL       (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
22 #define TASK_REPORT       (TASK_RUNNING | TASK_INTERRUPTIBLE | \
23 TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
24 __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
25 TASK_PARKED)
```

Листинг 1.2: Описание состояний процесса с помощью макросов

```
1 #define task_is_running(task)  (READ_ONCE((task)->__state) == TASK_RUNNING)
2 #define task_is_traced(task)  ((READ_ONCE(task->__state) & __TASK_TRACED) !=
3                                0)
4 #define task_is_stopped(task) ((READ_ONCE(task->__state) & __TASK_STOPPED)
5                                != 0)
```

```

4 #define task_is_stopped_or_traced(task) ((READ_ONCE(task->__state) & (
    __TASK_STOPPED | __TASK_TRACED)) != 0)

```

Листинг 1.3: Макросы с помощью которых можно узнать текущее состояние процесса

1.3.2 Структура `struct sysinfo`

Структура `struct sysinfo` [11] хранит информацию статистику о всей системе: информацию о времени, прошедшем с начала запуска системы, количество занятой памяти и так далее. В листинге 1.4 приведёно объявление рассматриваемой структуры.

```

1 struct sysinfo {
2     __kernel_long_t uptime;    /* Seconds since boot */
3     __kernel_ulong_t loads[3]; /* 1, 5, and 15 minute load averages */
4     __kernel_ulong_t totalram; /* Total usable main memory size */
5     __kernel_ulong_t sharedram; /* Amount of shared memory */
6     __kernel_ulong_t bufferram; /* Memory used by buffers */
7     __kernel_ulong_t totalswap; /* Total swap space size */
8     __kernel_ulong_t freeswap; /* swap space still available */
9     __u16 procs;               /* Number of current processes */
10    __u16 pad;                  /* Explicit padding for m68k */
11    __kernel_ulong_t totalhigh; /* Total high memory size */
12    __kernel_ulong_t freehigh;  /* Available high memory size */
13    __u32 mem_unit;             /* Memory unit size in bytes */
14    char _f[20-2*sizeof(__kernel_ulong_t)-sizeof(__u32)];
15 };

```

Листинг 1.4: Листинг структуры `struct sysinfo`

Для инициализации этой структуры используется функция `si_meminfo()`. Стоит отметить, что рассматриваемая структура не содержит информации о свободной памяти в системе. Для того чтобы получить эту информацию, необходимо воспользоваться функцией `si_mem_available()`.

1.4 Загружаемые модули ядра

Одной из особенностей ядра Linux является способность расширения функциональности во время работы, без необходимости компиляции ядра заново. Таким образом, существует возможность добавить (или убрать) функциональность в ядро можно когда система запущена и работает. Часть кода, которая может быть добавлена в ядро во время работы, называется модулем ядра. Ядро Linux предлагает поддержку большого числа классов модулей. Каждый модуль – это подготовленный объектный код, который может быть динамически подключен в работающее ядро, а позднее может быть выгружен из ядра.

Каждый модуль ядра сам регистрирует себя для того, чтобы обслуживать в будущем запросы, и его функция инициализации немедленно прекращается. Задача инициализации модуля заключается в подготовке функций модуля для последующего вызова. Функция выхода модуля вызывается перед выгрузкой модуля из ядра. Функция выхода должна отменить все изменения, сделанные функцией инициализации, освободить захваченные в процессе работы модуля ресурсы.

Возможность выгрузить модуль помогает сократить время разработки – нет необходимости перезагрузки компьютера при последовательном тестировании новых версий разрабатываемого модуля ядра.

Модуль связан только с ядром и может вызывать только те функции, которые экспортированы ядром.

1.4.1 Пространство ядра и пользователя

Приложения работают в пользовательском пространстве, а ядро и его модули – в пространстве ядра. Такое разделение пространств – базовая концепция теории операционных систем.

Ролью операционной системы является обеспечение программ надёжным доступом к аппаратной части компьютера. Операционная система должна обеспечивать независимую работу программ и защиту от несанкционированного доступа к ресурсам. Решение этих задач становится возмож-

ным только в том случае, если процессор обеспечивает защиту системного программного обеспечения от прикладных программ.

Выбранный подход заключается в обеспечении разных режимов работы (или уровней) в самом центральном процессоре. Уровни играют разные роли и некоторые операции на более низких уровнях не допускаются; программный код может переключить один уровень на другой только ограниченным числом способов. Все современные процессоры имеют не менее двух уровней защиты, а некоторые, например семейство процессоров x86, имеют больше уровней; когда существует несколько уровней, используется самый высокий и самый низкий уровень защиты.

Ядро Linux выполняется на самом высоком уровне, где разрешено выполнение любых инструкций и доступ к произвольным участкам памяти, а приложения выполняются на самом низком уровне, в котором процессор регулирует прямой доступ оборудованию и несанкционированный доступ к памяти. Ядро выполняет переход из пользовательского пространства в пространство ядра, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, выполняя системный вызов, работает в контексте процесса – он действует от имени вызывающего процесса и в состоянии получить данные в адресном пространстве процесса. Код, который обрабатывает прерывания является асинхронным по отношению к процессам и не связан с каким-либо определенным процессом [4].

Ролью модуля ядра является расширение функциональности ядра без его перекомпиляции. Код модулей выполняется в пространстве ядра.

1.5 Виртуальная файловая система /proc

Для организации доступа к разнообразным файловым системам в Unix используется промежуточный слой абстракции – виртуальная файловая система. С точки зрения программиста, виртуальная файловая система организована как специальный интерфейс. Виртуальная файловая система объявляет API доступа к ней, а реализацию этого API отдаёт на откуп к драйверам конкретных файловых систем.

Виртуальная файловая система `/proc` – специальный интерфейс, с помощью которого можно мгновенно получить некоторую информацию о ядре в пространство пользователя. `/proc` отображает в виде дерева каталогов внутренние структуры ядра.

В каталоге `/proc` в Linux присутствуют несколько деревьев файловой системы. В основном дереве, каждый каталог имеет числовое имя и соответствует процессу, с соответствующим PID. Файлы в этих каталогах соответствуют структуре `task_struct`. Так, например, с помощью команды `cat /proc/1/cmdline`, можно узнать аргументы запуска процесса с идентификатором равным единице. В дереве `/proc/sys` отображаются внутренние переменные ядра.

Ядро предоставляет возможность добавить своё дерево в каталог `/proc`. Внутри ядра объявлена специальная структура `struct proc_ops` [12]. Эта структура содержит внутри себя указатели на функции чтения файла, записи в файл и прочие, определенные пользователем. В листинге [?] представлено объявление данной структуры в ядре.

```
1 struct proc_ops {
2     unsigned int proc_flags;
3     int (*proc_open)(struct inode *, struct file *);
4     ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
5     ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
6     ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t *);
7     loff_t (*proc_lseek)(struct file *, loff_t, int);
8     int (*proc_release)(struct inode *, struct file *);
9     __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
10    long (*proc_ioctl)(struct file *, unsigned int, unsigned long);
11    #ifdef CONFIG_COMPAT
12    long (*proc_compat_ioctl)(struct file *, unsigned int, unsigned long);
13    #endif
14    int (*proc_mmap)(struct file *, struct vm_area_struct *);
15    unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long,
16    unsigned long, unsigned long, unsigned long);
16 } __randomize_layout;
```

Листинг 1.5: Листинг структуры `struct sysinfo`

С помощью вызова функций `proc_mkdir()` и `proc_create()` в модуле ядра можно зарегистрировать свои каталоги и файлы в `/proc` соответственно. Функции `copy_to_user()` и `copy_from_user()` реализуют передачу данных (набора байтов) из пространства ядра в пространство пользователя

и наоборот.

Таким образом, с помощью виртуальной файловой системы `/proc` можно получать (или передавать) какую-либо информацию из пространства ядра в пространство пользователя (из пространства пользователя в пространство ядра).

Вывод

В данном разделе были проанализированы различные подходы к перехвату функций. В ходе анализа, был выбран фреймворк `ftrace`, так как он позволяет перехватить любую функцию зная лишь её имя, может быть загружен в ядро динамически и не требует специальной сборки ядра и имеет хорошо задокументированный API. Были рассмотрены структуры и функции ядра, предоставляющие информацию о процессах и памяти; рассмотрены особенности загружаемых модулей ядра и понятия пространств ядра и пространства пользователя, а так же рассмотрен способ взаимодействия этих двух пространств с целью передачи данных из одного в другого.

2 Конструкторская часть

2.1 Архитектура приложения

В состав разработанного программного обеспечения входит один загружаемый модуль ядра, который перехватывает все вызовы системных вызовов, подсчитывая их количество за определенный промежуток времени, предоставляет пользователю информацию о процессах и их состояниях, а так же информацию и состояние о загруженности оперативной памяти – её общее количество, свободной и доступной в данный момент.

2.2 Структура `struct ftrace_hook`

В листинге 2.1 представлено объявление структуры `struct ftrace_hook`, которая описывает каждую перехватываемую функцию.

```
1 struct ftrace_hook {  
2     const char *name;  
3     void *function;  
4     void *original;  
5  
6     unsigned long address;  
7     struct ftrace_ops ops;  
8 };
```

Листинг 2.1: Листинг структуры `ftrace_hook`

Необходимо заполнить только первые три поля:

- `name` – имя перехватываемой функции;
- `function` – адрес функции обёртки, вызываемой вместо перехваченной функции;
- `original` – указатель на перехватываемую функцию.

Остальные поля считаются деталью реализации. Описание всех перехватываемых были собраны в массив, а для инициализации был написан

специальный макрос (см. листинг 2.2).

```
1 #define ADD_HOOK(_name, _function, _original) \
2 { \
3     .name = SYSCALL_NAME(_name), \
4     .function = (_function), \
5     .original = (_original), \
6 }
7
8 static struct ftrace_hook hooked_functions[] = {
9     ADD_HOOK("sys_execve", hook_sys_execve, &real_sys_execve),
10    ADD_HOOK("sys_write", hook_sys_write, &real_sys_write),
11    ADD_HOOK("sys_open", hook_sys_open, &real_sys_open),
12    ADD_HOOK("sys_close", hook_sys_close, &real_sys_close),
13    ADD_HOOK("sys_mmap", hook_sys_mmap, &real_sys_mmap),
14    ADD_HOOK("sys_sched_yield", hook_sys_sched_yield, &real_sys_sched_yield),
15    ADD_HOOK("sys_socket", hook_sys_socket, &real_sys_socket),
16    ADD_HOOK("sys_connect", hook_sys_connect, &real_sys_connect),
17    ADD_HOOK("sys_accept", hook_sys_accept, &real_sys_accept),
18    ADD_HOOK("sys_sendto", hook_sys_sendto, &real_sys_sendto),
19    ADD_HOOK("sys_recvfrom", hook_sys_recvfrom, &real_sys_recvfrom),
20    ADD_HOOK("sys_sendmsg", hook_sys_sendmsg, &real_sys_sendmsg),
21    ADD_HOOK("sys_recvmsg", hook_sys_recvmsg, &real_sys_recvmsg),
22    ADD_HOOK("sys_shutdown", hook_sys_shutdown, &real_sys_shutdown),
23    ADD_HOOK("sys_read", hook_sys_read, &real_sys_read),
24    ADD_HOOK("sys_clone", hook_sys_clone, &real_sys_clone),
25    ADD_HOOK("sys_mkdir", hook_sys_mkdir, &real_sys_mkdir),
26    ADD_HOOK("sys_rmdir", hook_sys_rmdir, &real_sys_rmdir),
27 };
```

Листинг 2.2: Объявление массива перехватываемых функций и специальный макрос для его инициализации

2.3 Алгоритм перехвата системного вызова

На рисунке 2.1 представлена схема алгоритма перехвата системных вызовов на примере `sys_clone`.

1. Пользовательский процесс выполняет инструкцию `SYSCALL`. С помощью этой инструкции выполняется переход в режим ядра и управление передаётся низкоуровневому обработчику системных вызовов

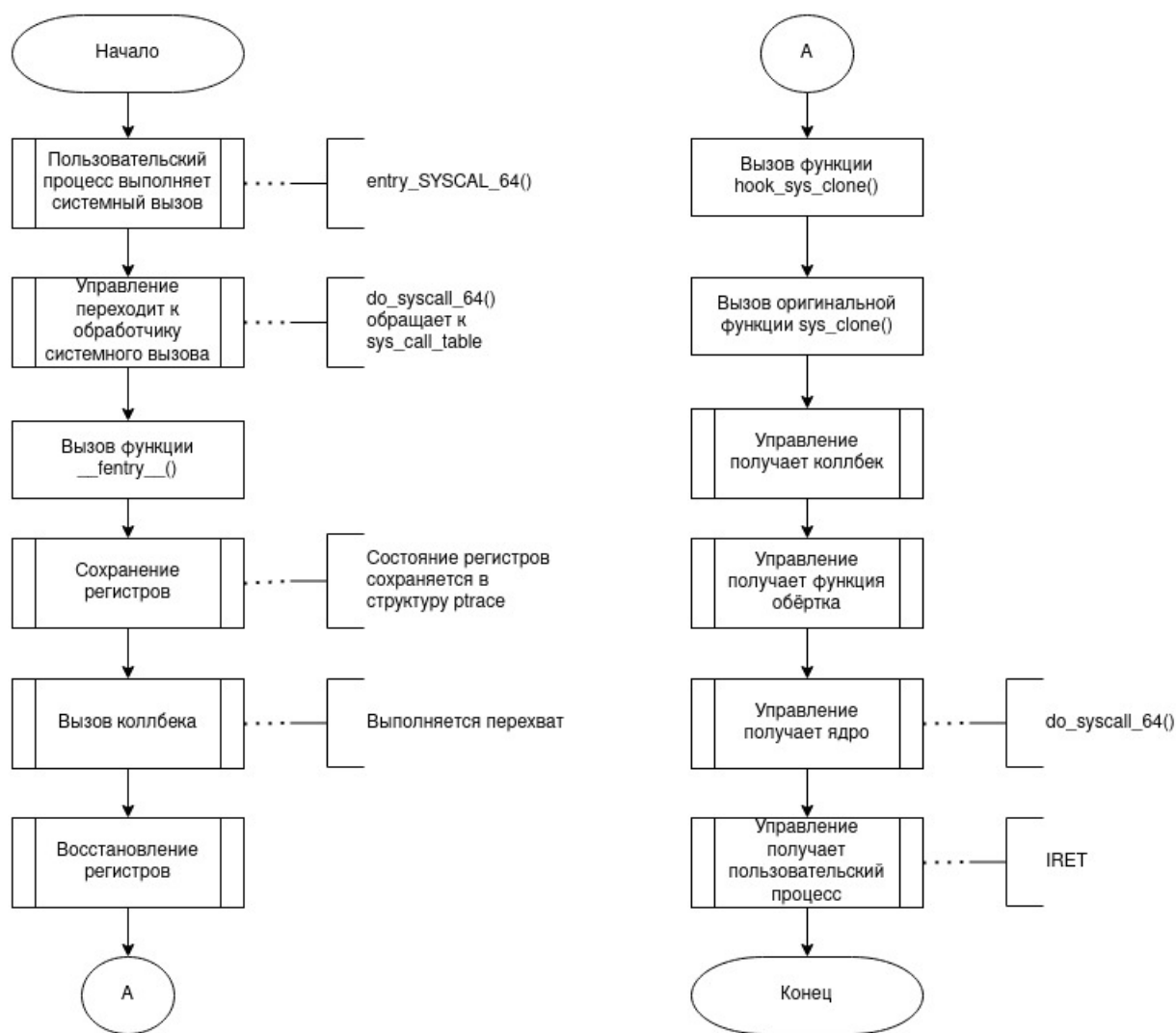


Рис. 2.1: Алгоритм перехвата системного вызова

`entry_SYSCALL_64()`. Этот обработчик отвечает за все системные вызовы 64-битных программ на 64-битных машинах.

2. Управление переходит к обработчику системного вызова. Ядро передаёт управление функции `do_syscall_64()`. Эта функция обращается к таблице обработчиков системных вызовов `sys_call_table` и с помощью неё вызывает конкретный обработчик системного вызова – `sys_clone()`.
3. Вызывается `ftrace`. В начале каждой функции ядра находится вызов функции `__fentry__()`, реализованная фреймворком `ftrace`. Перед этим состояние регистров сохраняется в специальную структуру `pt_regs`.

4. **ftrace** вызывает разработанный коллбек.
5. Коллбек выполняет перехват. Коллбек анализирует значение **parent_ip** и выполняет перехват, обновляя значение регистра **rip** (указатель на следующую исполняемую инструкцию) в структуре **pt_regs**.
6. **ftrace** восстанавливает значение регистров с помощью структуры **pt_regs**. Так как обработчик изменяет значение регистра **rip** – это приведёт к передаче управления по новому адресу.
7. Управление получает функция обёртка. Благодаря безусловному переходу, управление получает наша функция **hook_sys_clone()**, а не оригинальная функция **sys_clone()**. При этом всё остальное состояние процессора и памяти остаётся без изменений – функция получает все аргументы оригинального обработчика и при завершении вернёт управление в функцию **do_syscall_64()**.
8. Функция обёртка вызывает оригинальную функцию.
Функция **hook_sys_clone()** может проанализировать аргументы и контекст системного вызова и запретить или разрешить процессу его выполнение. В случае его запрета, функция просто возвращает код ошибки. Иначе – вызывает оригинальный обработчик **sys_clone()** повторно, с помощью указателя **real_sys_clone**, который был сохранён при настройке перехвата.
9. Управление получает коллбек. Как и при первом вызове **sys_clone()**, управление проходит через **ftrace** и передается в коллбек.
10. Коллбек ничего не делает. В этот раз функция **sys_clone()** вызывается разработанной функцией **hook_sys_clone()**, а не ядром из функции **do_syscall_64()**. Коллбек не модифицирует регистры и выполнение функции **sys_clone()** продолжается как обычно.
11. Управление передаётся функции обёртке.
12. Управление передаётся ядру. Функция **hook_sys_clone()** завершается и управление переходит к **do_syscall_64()**.

13. Управление возвращает в пользовательский процесс. Ядро выполняет инструкцию `IRET`, устанавливая регистры для нового пользовательского процесса и переводя центральный процессор в режим исполнения пользовательского кода.

2.4 Алгоритм подсчёта количества системных вызовов

На рисунке 2.2 представлена схема алгоритма подсчёта системных вызовов.



Рис. 2.2: Алгоритм подсчёта количества системных вызовов

- Агрегирующий массив – это массив на 86400 элементов, состоящий из структур, имеющих два поля в виде 64-битных без знаковых целых чисел. Это позволяет фиксировать до 128 системных вызовов в секунду на протяжении 24 часов. Такой массив занимает всего лишь 1350 килобайт оперативной памяти;

- спин-блокировка необходима с той целью, что несколько системных вызовов могут быть вызваны в один и тот же момент времени – в таком случае, без блокировки, агрегирующий массив потеряет часть данных;

Вывод

В данном разделе была рассмотрена общая архитектура приложения, алгоритм перехвата системных вызовов и подсчёта их количества за выбранный промежуток.

3 Технологическая часть

3.1 Выбор языка программирования

Разработанный модуль ядра написан на языке программирования C [13]. Выбор языка программирования C основан на том, что исходный код ядра Linux, все его модули и драйверы написаны на данном языке.

В качестве компилятора выбран gcc [14].

3.2 Поиск адреса перехватываемой функции

Для корректной работы `ftrace` необходимо найти и сохранить адрес функции, которую будет перехватывать разрабатываемый модуль ядра.

В старых версиях ядра (в версии ядра 5.7.0 данная функция перестала быть экспортируемой [15]) найти адрес функции можно было с помощью функции `kallsyms_lookup_name()` – списка всех символов в ядре, в том числе не экспортируемых для модулей. Так как модуль ядра разрабатывался на системе с версией ядра 5.14.9, воспользоваться данным способом было нельзя. В конечном итоге проблемы была решена с помощью интерфейса `kprobes` (который был описан в 1.1.3).

Из-за того что данный способ имеет больше накладных расходов, чем поиск с помощью `kallsyms_lookup_name()` (требуется регистрация и удаление `kprobes` в системе), для версий ядра ниже 5.7.0 поиск адреса производится с помощью `kallsyms_lookup_name()`. Такая реализация стала возможной благодаря директивам условной компиляции [16] и специальным макросам `LINUX_VERSION_CODE` и `KERNEL_VERSION()`.

Реализация функции `lookup_name()`, возвращающей адрес функции перехватываемой функции по её названию, представлена в листинге 3.1.

```
1 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0)
2 static unsigned long lookup_name(const char *name)
3 {
4     struct kprobe kp = {
5         .symbol_name = name
6     };
```

```

7   unsigned long  retval;
8
9   ENTER_LOG();
10
11  if (register_kprobe(&kp) < 0) {
12      EXIT_LOG();
13      return 0;
14  }
15
16  retval = (unsigned long) kp.addr;
17  unregister_kprobe(&kp);
18
19  EXIT_LOG();
20
21  return retval;
22  }
23  #else
24  static unsigned long lookup_name(const char *name)
25  {
26      unsigned long  retval;
27
28      ENTER_LOG();
29      retval = kallsyms_lookup_name(name);
30      EXIT_LOG();
31
32      return retval;
33  }
34  #endif

```

Листинг 3.1: Реализация функции `lookup_name()`

3.3 Инициализация `ftrace`

В листинге 3.2 представлена реализация функции, которая инициализирует структуру `ftrace_ops`.

```

1  static int install_hook(struct ftrace_hook *hook) {
2      int rc;
3
4      ENTER_LOG();
5
6      if ((rc = resolve_hook_address(hook))) {
7          EXIT_LOG();
8          return rc;

```

```

9   }
10
11   hook->ops.func = ftrace_thunk;
12   hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
13   | FTRACE_OPS_FL_RECURSION
14   | FTRACE_OPS_FL_IPMODIFY;
15
16   if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0))) {
17       pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
18       return rc;
19   }
20
21   if ((rc = register_ftrace_function(&hook->ops))) {
22       pr_debug("register_ftrace_function() failed: %d\n", rc);
23       ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
24   }
25
26   EXIT_LOG();
27
28   return rc;
29 }

```

Листинг 3.2: Реализация функции `install_hook()`

В листинге 3.3 представлена реализация отключения перехвата функции.

```

1  static void remove_hook(struct ftrace_hook *hook) {
2      int rc;
3
4      ENTER_LOG();
5
6      if (hook->address == 0x00) {
7          EXIT_LOG();
8          return;
9      }
10
11     if ((rc = unregister_ftrace_function(&hook->ops))) {
12         pr_debug("unregister_ftrace_function() failed: %d\n", rc);
13     }
14
15     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0))) {
16         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
17     }
18
19     hook->address = 0x00;
20
21     EXIT_LOG();

```


Листинг 3.3: Реализация функции `remove_hook()`

3.4 Функции обёртки

При объявлении функций обёрток, которые будут запущены вместо перехватываемой функции, необходимо в точности соблюдать сигнатуру. Так, должны совпадать порядок, типы аргументов и возвращаемого значения. Оригинальные описания функций были из исходных кодов ядра Linux.

В листинге 3.4 представлена реализация функции обёртки на примере `sys_clone()`.

```

1 static asmlinkage long (*real_sys_clone)(unsigned long clone_flags ,
2 unsigned long newsp, int __user *parent_tidptr ,
3 int __user *child_tidptr , unsigned long tls);
4
5 static asmlinkage long hook_sys_clone(unsigned long clone_flags ,
6 unsigned long newsp, int __user *parent_tidptr ,
7 int __user *child_tidptr , unsigned long tls)
8 {
9     update_syscall_array(SYS_CLONE_NUM);
10    return real_sys_clone(clone_flags , newsp, parent_tidptr , child_tidptr , tls);
11 }
```

Листинг 3.4: Реализация функции обёртки

В листинге 3.5 представлена реализация функции которая обновляет массив, хранящий количество системных вызовов за последние 24 часа.

```

1 static DEFINE_SPINLOCK(my_lock);
2
3 static void inline update_syscall_array(int syscall_num) {
4     ktime_t time;
5
6     time = ktime_get_boottime_seconds() - start_time;
7
8     spin_lock(&my_lock);
9
10    if (syscall_num < 64) {
11        syscalls_time_array[time % TIME_ARRAY_SIZE].p1 |= 1UL << syscall_num;
```

```

12 } else {
13     syscalls_time_array[time % TIME_ARRAY_SIZE].p2 |= 1UL << (syscall_num %
14         64);
15 }
16 spin_unlock(&my_lock);
17 }

```

Листинг 3.5: Реализация функции `update_syscall_array()`

3.5 Получение информации о количестве системных вызовов

В листинге 3.6 представлена реализация функций, которые агрегируют информацию о системных вызовах (данные массива `update_syscall_array`) и предоставляют ее в читаемом для пользователя виде.

```

1 static inline void walk_bits_and_find_syscalls(struct seq_file *m, uint64_t
   num, int syscalls_arr_cnt[]) {
2     int i;
3
4     for (i = 0; i < 64; i++) {
5         if (num & (1UL << i)) {
6             syscalls_arr_cnt[i]++;
7         }
8     }
9 }
10
11 void print_syscall_statistics(struct seq_file *m, const ktime_t mstart,
   ktime_t range) {
12     int syscalls_arr_cnt[128];
13     uint64_t tmp;
14     size_t i;
15     ktime_t uptime;
16
17     memset((void*)syscalls_arr_cnt, 0, 128 * sizeof(int));
18     uptime = ktime_get_boottime_seconds() - mstart;
19
20     if (uptime < range) {
21         range = uptime;
22     }
23
24     for (i = 0; i < range; i++) {

```

```

25     if ((tmp = syscalls_time_array[uptime - i].p1) != 0) {
26         walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt);
27     }
28
29     if ((tmp = syscalls_time_array[uptime - i].p2) != 0) {
30         walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt + 64);
31     }
32 }
33
34 show_int_message(m, "Syscall statistics for the last %d seconds.\n\n", range
35 );
36
37 for (i = 0; i < 128; i++) {
38     if (syscalls_arr_cnt[i] != 0) {
39         show_str_message(m, "%s called ", syscalls_names[i]);
40         show_int_message(m, "%d times.\n", syscalls_arr_cnt[i]);
41     }
42 }

```

Листинг 3.6: Реализация функций агрегации данных о системных вызовах

3.6 Информация о памяти в системе

Для сбора информации о доступной и свободной памяти в системе запускается отдельный поток ядра, который находится в состоянии сна, и просыпаясь каждые 10 секунд, фиксирует эту информацию в результирующий массив. В листинге 3.7 представлена реализация этого потока, а в листинге 3.8 его инициализация.

```

1 mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
2 int mem_info_calls_cnt;
3
4 int memory_cnt_task_handler_fn(void *args) {
5     struct sysinfo i;
6     struct timespec64 t;
7
8     ENTER_LOG();
9
10    allow_signal(SIGKILL);
11
12    while (!kthread_should_stop()) {
13        si_meminfo(&i);

```

```

14
15     ktime_get_real_ts64(&t);
16
17     mem_info_array[mem_info_calls_cnt].free = i.freeram;
18     mem_info_array[mem_info_calls_cnt].available = si_mem_available();
19     mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
20
21     ssleep(10);
22
23     if (signal_pending(worker_task)) {
24         break;
25     }
26 }
27
28 EXIT_LOG();
29 do_exit(0);
30 return 0;
31 }

```

Листинг 3.7: Реализация функции сохраняющей информацию о доступной в системе памяти

```

1 int my_thread_init() {
2     cpu = get_cpu();
3     worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
4         counter thread");
5     kthread_bind(worker_task, cpu);
6
7     if (worker_task == NULL) {
8         cleanup();
9         return -1;
10    }
11
12    wake_up_process(worker_task);
13    return 0;
14 }

```

Листинг 3.8: Функция инициализации потока ядра

3.7 Получение информации о процессах

В листинге 3.9 представлена реализация функции, которая выводит информацию о состоянии всех текущих процессах на данный момент.

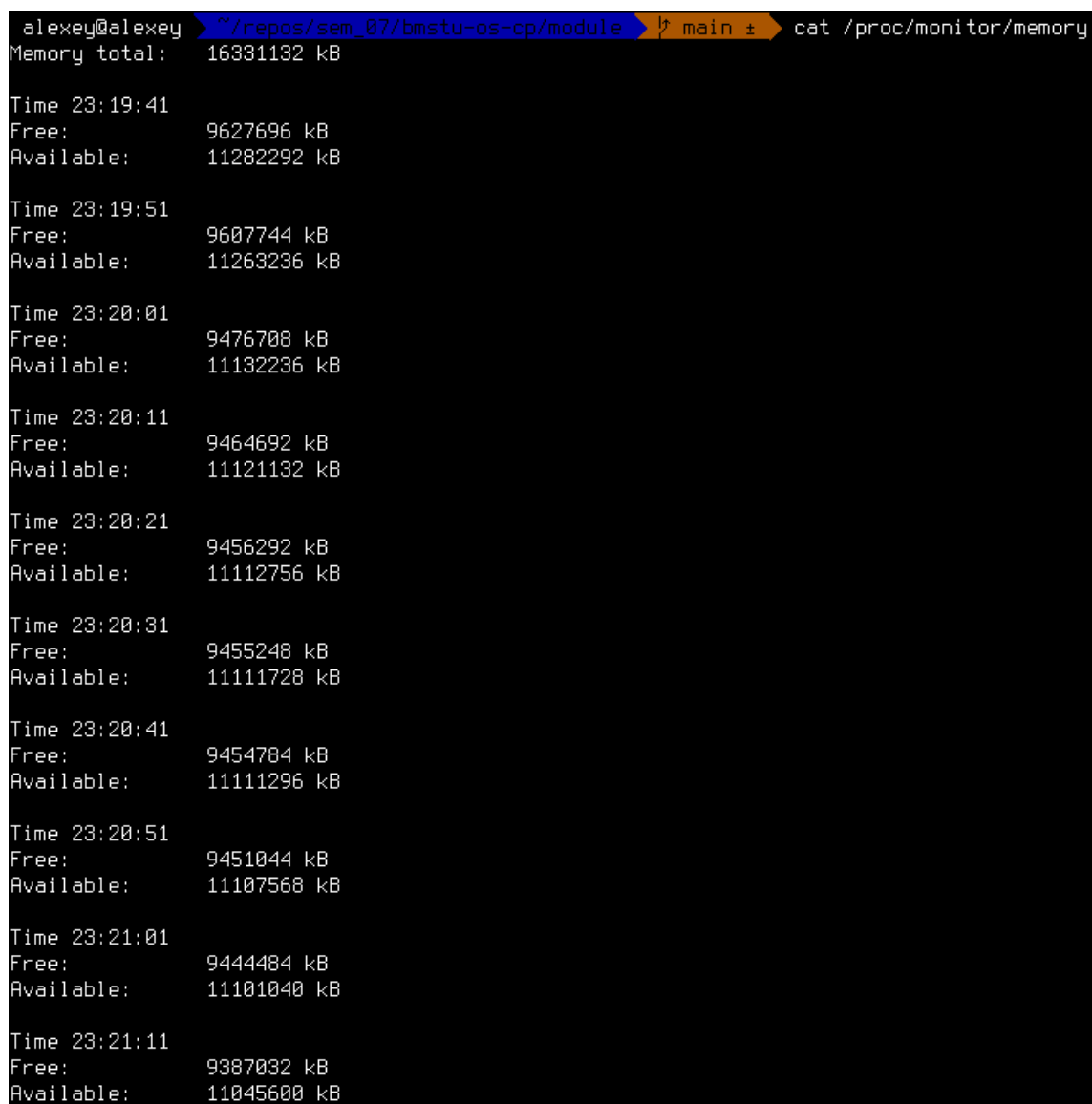
```

1 void print_task_statistics(struct seq_file *m) {
2     struct task_struct *task;
3     int total = 0, running = 0, stopped = 0, zombie = 0, interruptible = 0,
        uninterruptible;
4
5     ENTER_LOG();
6
7     for_each_process(task) {
8         switch (task->TASK_STATE_FIELD) {
9             case TASK_RUNNING:
10                running++;
11                break;
12             case TASK_INTERRUPTIBLE:
13                interruptible++;
14                break;
15             case TASK_IDLE: /* (TASK_UNINTERRUPTIBLE | TASK_NOLOAD) */
16                uninterruptible++;
17                break;
18             case TASK_STOPPED:
19                stopped++;
20                break;
21             case TASK_TRACED: /* (TASK_WAKEKILL | __TASK_TRACED) */
22                stopped++;
23                break;
24             default:
25                printk(KERN_INFO "%x %s %d\n", task->TASK_STATE_FIELD, task->comm, task
->pid);
26            }
27
28            if (task->exit_state == EXIT_ZOMBIE)
29            {
30                zombie++;
31            }
32
33            total++;
34        }
35
36        show_int_message(m, "Total processes: %d\n", total);
37        show_int_message(m, "Running: %d\n", running);
38        show_int_message(m, "Sleeping: %d ", total - running - stopped - zombie);
39        show_int_message(m, "[Interruptible: %d | ", interruptible);
40        show_int_message(m, "Uninterruptible: %d]\n", uninterruptible);
41        show_int_message(m, "Stopped: %d\n", stopped);
42        show_int_message(m, "Zombie: %d\n", zombie);
43
44        EXIT_LOG();
45    }

```

3.8 Примеры работы разработанного ПО

На рисунках 3.1 - 3.4 представлены примеры работы разработанного модуля ядра. Для наглядности перехватываются только 18 системных вызовов.



```
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main > cat /proc/monitor/memory
Memory total: 16331132 kB

Time 23:19:41
Free: 9627696 kB
Available: 11282292 kB

Time 23:19:51
Free: 9607744 kB
Available: 11263236 kB

Time 23:20:01
Free: 9476708 kB
Available: 11132236 kB

Time 23:20:11
Free: 9464692 kB
Available: 11121132 kB

Time 23:20:21
Free: 9456292 kB
Available: 11112756 kB

Time 23:20:31
Free: 9455248 kB
Available: 11111728 kB

Time 23:20:41
Free: 9454784 kB
Available: 11111296 kB

Time 23:20:51
Free: 9451044 kB
Available: 11107568 kB

Time 23:21:01
Free: 9444484 kB
Available: 11101040 kB

Time 23:21:11
Free: 9387032 kB
Available: 11045600 kB
```

Рис. 3.1: Информация о оперативной памяти в системе

```

alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 306
Running: 1
Sleeping: 303 [Interruptible: 203 | Uninterruptible: 100]
Stopped: 1
Zombie: 1
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 304
Running: 1
Sleeping: 302 [Interruptible: 202 | Uninterruptible: 100]
Stopped: 1
Zombie: 0
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 304
Running: 1
Sleeping: 302 [Interruptible: 202 | Uninterruptible: 100]
Stopped: 1
Zombie: 0
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 302
Running: 1
Sleeping: 301 [Interruptible: 201 | Uninterruptible: 100]
Stopped: 0
Zombie: 0
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ±

```

Рис. 3.2: Информация о процессах и их состояниях на текущий момент в системе

```

alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/syscalls
Syscall statistics for the last 122 seconds.

sys_read called 122 times.
sys_write called 121 times.
sys_open called 2 times.
sys_close called 87 times.
sys_mmap called 69 times.
sys_sched_yield called 53 times.
sys_socket called 17 times.
sys_connect called 15 times.
sys_accept called 4 times.
sys_sendto called 86 times.
sys_recvfrom called 24 times.
sys_sendmsg called 111 times.
sys_recvmsg called 122 times.
sys_shutdown called 4 times.
sys_clone called 30 times.
sys_execve called 24 times.
sys_mkdir called 4 times.
sys_rmdir called 2 times.
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ±

```

Рис. 3.3: Информация о количестве системных вызовов за последние 122 секунды

На рисунке 3.5 представлена визуализация данных о свободной и доступной памяти в системе, полученных из разработанного модуля ядра.

```
alexey@alexey ~/repos/sem_07/bmstu-os-cp/module main ± echo 00h00m15s > /proc/monitor/syscalls
alexey@alexey ~/repos/sem_07/bmstu-os-cp/module main ± cat /proc/monitor/syscalls
Syscall statistics for the last 15 seconds.
sys_read called 15 times.
sys_write called 15 times.
sys_close called 14 times.
sys_mmap called 14 times.
sys_sched_yield called 1 times.
sys_socket called 1 times.
sys_connect called 1 times.
sys_sendto called 6 times.
sys_recvfrom called 4 times.
sys_sendmsg called 15 times.
sys_recvmsg called 15 times.
sys_clone called 3 times.
sys_execve called 2 times.
alexey@alexey ~/repos/sem_07/bmstu-os-cp/module main ±
```

Рис. 3.4: Конфигурирование модуля для отображение информации о системных вызовах за последние 15 секунд



Рис. 3.5: Визуализация данных о свободной и доступной памяти в системе за 25 минут

Вывод

В данном разделе был обоснован выбор языка программирования, рассмотрены листинги реализованного программного обеспечения и приведены результаты работы ПО.

Заключение

В ходе проделанной работы был разработан загружаемый модуль ядра, предоставляющий информацию о загрузке системы: количество системных вызовов за выбранный промежуток времени, количество свободной и доступной оперативной памяти, статистика по процессам и в каких состояниях они находятся.

Изучены структуры и функции ядра, которые предоставляют информацию о процессах и памяти. Проанализированы существующие подходы к перехвату системных вызовов.

На основе полученных знаний и проанализированных технологий реализован загружаемый модуль ядра.

Литература

- [1] Linux - Operating System [Электронный ресурс]. Режим доступа: <https://www.linux.org/> (дата обращения: 08.11.2021).
- [2] Linux Security Module Usage [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html> (дата обращения: 08.11.2021).
- [3] Колбэк-функция – Глоссарий – MDN Web Docs [Электронный ресурс]. Режим доступа: https://developer.mozilla.org/ru/docs/Glossary/Callback_function (дата обращения: 08.11.2021).
- [4] Механизмы профилирования Linux – Habr [Электронный ресурс]. Режим доступа: <https://habr.com/ru/company/metrotek/blog/261003/> (дата обращения: 08.11.2021).
- [5] Kernel Probes (Kprobes) [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/kprobes.html> (дата обращения: 08.11.2021).
- [6] Using the Linux Kernel Tracepoints [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/tracepoints.html> (дата обращения: 08.11.2021).
- [7] Using ftrace | Android Open Source Project [Электронный ресурс]. Режим доступа: <https://source.android.com/devices/tech/debug/ftrace> (дата обращения: 08.11.2021).
- [8] Трассировка ядра с ftrace – Habr [Электронный ресурс]. Режим доступа: <https://habr.com/ru/company/selectel/blog/280322/> (дата обращения: 08.11.2021).
- [9] NOP: No Operation (x86 Instruction Set Reference) [Электронный ресурс]. Режим доступа: https://c9x.me/x86/html/file_module_x86_id_217.html (дата обращения: 08.11.2021).

- [10] `include/linux/sched.h` - Linux source code (v5.15.3) [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h> (дата обращения: 08.11.2021).
- [11] `include/uapi/linux/sysinfo.h` - Linux source code (v5.15.3) [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/sysinfo.h#L8> (дата обращения: 08.11.2021).
- [12] `include/linux/proc_fs.h` - Linux source code (v5.15.3) [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/linux/proc_fs.h#L29 (дата обращения: 08.11.2021).
- [13] C99 standard note [Электронный ресурс]. Режим доступа: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf> (дата обращения: 10.11.2021).
- [14] GCC, the GNU Compiler Collection [Электронный ресурс]. Режим доступа: <https://gcc.gnu.org/> (дата обращения: 10.11.2021).
- [15] Unexporting `kallsyms_lookup_name()` [Электронный ресурс]. Режим доступа: <https://lwn.net/Articles/813350/> (дата обращения: 10.11.2021).
- [16] Директивы препроцессора C [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/preprocessor-directives> (дата обращения: 10.11.2021).

ПРИЛОЖЕНИЕ А

```
1 #include <linux/module.h>
2 #include <linux/proc_fs.h>
3 #include <linux/time.h>
4 #include <linux/kthread.h>
5
6 #include "hooks.h"
7 #include "memory.h"
8 #include "stat.h"
9
10 MODULE_LICENSE("GPL");
11 MODULE_AUTHOR("Romanov Alexey");
12 MODULE_DESCRIPTION("A utility for monitoring the state of the system and
    kernel load");
13
14 static struct proc_dir_entry *proc_root = NULL;
15 static struct proc_dir_entry *proc_mem_file = NULL, *proc_task_file = NULL, *
    proc_syscall_file = NULL;
16 static struct task_struct *worker_task = NULL;
17
18 extern ktime_t start_time;
19 /* default syscall range value is 10 min */
20 static ktime_t syscalls_range_in_seconds = 600;
21
22 static int show_memory(struct seq_file *m, void *v) {
23     print_memory_statistics(m);
24     return 0;
25 }
26
27 static int proc_memory_open(struct inode *sp_inode, struct file *sp_file) {
28     return single_open(sp_file, show_memory, NULL);
29 }
30
31 static int show_tasks(struct seq_file *m, void *v) {
32     print_task_statistics(m);
33     return 0;
34 }
35
36 static int proc_tasks_open(struct inode *sp_inode, struct file *sp_file) {
37     return single_open(sp_file, show_tasks, NULL);
38 }
39
40 static int show_syscalls(struct seq_file *m, void *v) {
41     print_syscall_statistics(m, start_time, syscalls_range_in_seconds);
42     return 0;
43 }
44
```

```

45 static int proc_syscalls_open(struct inode *sp_inode, struct file *sp_file) {
46     return single_open(sp_file, show_syscalls, NULL);
47 }
48
49 static int proc_release(struct inode *sp_node, struct file *sp_file) {
50     return 0;
51 }
52
53 mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
54 int mem_info_calls_cnt;
55
56 int memory_cnt_task_handler_fn(void *args) {
57     struct sysinfo i;
58     struct timespec64 t;
59
60     ENTER_LOG();
61
62     allow_signal(SIGKILL);
63
64     while (!kthread_should_stop()) {
65         si_meminfo(&i);
66
67         ktime_get_real_ts64(&t);
68
69         mem_info_array[mem_info_calls_cnt].free = i.freeram;
70         mem_info_array[mem_info_calls_cnt].available = si_mem_available();
71         mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
72
73         ssleep(10);
74
75         if (signal_pending(worker_task)) {
76             break;
77         }
78     }
79
80     EXIT_LOG();
81     do_exit(0);
82     return 0;
83 }
84
85 #define CHAR_TO_INT(ch) (ch - '0')
86
87 static ktime_t convert_strf_to_seconds(char buf[]) {
88     /* time format: xxhyyymmzs. For example: 01h23m45s */
89     ktime_t hours, min, secs;
90
91     hours = CHAR_TO_INT(buf[0]) * 10 + CHAR_TO_INT(buf[1]);
92     min = CHAR_TO_INT(buf[3]) * 10 + CHAR_TO_INT(buf[4]);

```

```

93     secs = CHAR_TO_INT(buf[6]) * 10 + CHAR_TO_INT(buf[7]);
94
95     return hours * 60 * 60 + min * 60 + secs;
96 }
97
98 static ssize_t proc_syscall_write(struct file *file, const char __user *buf,
99     size_t len, loff_t *ppos) {
100     char syscalls_time_range[10];
101     ENTER_LOG();
102
103     if (copy_from_user(&syscalls_time_range, buf, len) != 0) {
104         EXIT_LOG()
105         return -EFAULT;
106     }
107
108     syscalls_range_in_seconds = convert_strf_to_seconds(syscalls_time_range);
109
110     EXIT_LOG();
111     return len;
112 }
113
114 static const struct proc_ops mem_ops = {
115     proc_read: seq_read,
116     proc_open: proc_memory_open,
117     proc_release: proc_release,
118 };
119
120 static const struct proc_ops tasks_ops = {
121     proc_read: seq_read,
122     proc_open: proc_tasks_open,
123     proc_release: proc_release,
124 };
125
126 static const struct proc_ops syscalls_ops = {
127     proc_read: seq_read,
128     proc_open: proc_syscalls_open,
129     proc_release: proc_release,
130     proc_write: proc_syscall_write,
131 };
132
133 static void cleanup(void) {
134     ENTER_LOG();
135
136     if (worker_task) {
137         kthread_stop(worker_task);
138     }
139

```

```

140  if (proc_mem_file != NULL) {
141      remove_proc_entry("memory", proc_root);
142  }
143
144  if (proc_syscall_file != NULL) {
145      remove_proc_entry("syscalls", proc_root);
146  }
147
148  if (proc_task_file != NULL) {
149      remove_proc_entry("tasks", proc_root);
150  }
151
152  if (proc_root != NULL) {
153      remove_proc_entry(MODULE_NAME, NULL);
154  }
155
156  remove_hooks();
157
158  EXIT_LOG();
159 }
160
161 static int proc_init(void) {
162     ENTER_LOG();
163
164     if ((proc_root = proc_mkdir(MODULE_NAME, NULL)) == NULL) {
165         goto err;
166     }
167
168     if ((proc_mem_file = proc_create("memory", 066, proc_root, &mem_ops)) ==
169         NULL) {
170         goto err;
171     }
172
173     if ((proc_task_file = proc_create("tasks", 066, proc_root, &tasks_ops)) ==
174         NULL)
175     {
176         goto err;
177     }
178
179     if ((proc_syscall_file = proc_create("syscalls", 066, proc_root, &
180         syscalls_ops)) == NULL)
181     {
182         goto err;
183     }
184
185     EXIT_LOG();
186     return 0;

```

```

185     err :
186     cleanup();
187     EXIT_LOG();
188     return -ENOMEM;
189 }
190
191 static int __init md_init(void) {
192     int rc;
193     int cpu;
194
195     ENTER_LOG();
196
197     if ((rc = proc_init())) {
198         return rc;
199     }
200
201     if ((rc = install_hooks())) {
202         cleanup();
203         return rc;
204     }
205
206     start_time = ktime_get_boottime_seconds();
207
208     cpu = get_cpu();
209     worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
        counter thread");
210     kthread_bind(worker_task, cpu);
211
212     if (worker_task == NULL) {
213         cleanup();
214         return -1;
215     }
216
217     wake_up_process(worker_task);
218
219     printk("%s: module loaded\n", MODULE_NAME);
220     EXIT_LOG();
221
222     return 0;
223 }
224
225 static void __exit md_exit(void) {
226     cleanup();
227
228     printk("%s: module unloaded\n", MODULE_NAME);
229 }
230
231 module_init(md_init);

```



```
232 module_exit(md_exit);
```

Листинг 3.10: Листинг файла monitor_main.c

```
1 #include "stat.h"
2
3 #define TASK_STATE_FIELD state
4 #define TASK_STATE_SPEC "%ld"
5
6 #if LINUX_VERSION_CODE > KERNEL_VERSION(5,13,0)
7 #define TASK_STATE_FIELD __state
8 #define TASK_STATE_SPEC "%d"
9 #endif
10
11 static inline long convert_to_kb(const long n) {
12     return n << (PAGE_SHIFT - 10);
13 }
14
15 void print_memory_statistics(struct seq_file *m) {
16     struct sysinfo info;
17     long long secs;
18     int i;
19
20     ENTER_LOG();
21
22     si_meminfo(&info);
23     show_int_message(m, "Memory total: \t%ld kB\n", convert_to_kb(info.totalram)
24 );
25
26     for (i = 0; i < mem_info_calls_cnt; i++) {
27         secs = mem_info_array[i].time_secs;
28         show_int3_message(m, "\nTime %.2llu:%.2llu:%.2llu\n", (secs / 3600 + 3) %
29 24, secs / 60 % 60, secs % 60);
30         show_int_message(m, "Free: \t%ld kB\n", convert_to_kb(mem_info_array[
31 i].free));
32         show_int_message(m, "Available: \t%ld kB\n", convert_to_kb(mem_info_array[
33 i].available));
34     }
35
36     EXIT_LOG();
37 }
38
39 void print_task_statistics(struct seq_file *m) {
40     struct task_struct *task;
41     int total = 0, running = 0, stopped = 0, zombie = 0, interruptible = 0,
42         uninterruptible;
```

```

41 for_each_process(task) {
42     switch (task->TASK_STATE_FIELD) {
43         case TASK_RUNNING:
44             running++;
45             break;
46         case TASK_INTERRUPTIBLE:
47             interruptible++;
48             break;
49         case TASK_IDLE: /* (TASK_UNINTERRUPTIBLE | TASK_NOLOAD) */
50             uninterruptible++;
51             break;
52         case TASK_STOPPED:
53             stopped++;
54             break;
55         case TASK_TRACED: /* (TASK_WAKEKILL | __TASK_TRACED) */
56             stopped++;
57             break;
58         default:
59             printk(KERN_INFO "%x %s %d\n", task->TASK_STATE_FIELD, task->comm, task
->pid);
60     }
61
62     if (task->exit_state == EXIT_ZOMBIE)
63     {
64         zombie++;
65     }
66
67     total++;
68 }
69
70 show_int_message(m, "Total processes: %d\n", total);
71 show_int_message(m, "Running: %d\n", running);
72 show_int_message(m, "Sleeping: %d ", total - running - stopped - zombie);
73 show_int_message(m, "[Interruptible: %d | ", interruptible);
74 show_int_message(m, "Uninterruptible: %d]\n", uninterruptible);
75 show_int_message(m, "Stopped: %d\n", stopped);
76 show_int_message(m, "Zombie: %d\n", zombie);
77
78 EXIT_LOG();
79 }
80
81 syscalls_info_t syscalls_time_array[TIME_ARRAY_SIZE];
82
83 static const char *syscalls_names[] = {
84     "sys_read", /* 0 */
85     "sys_write",
86     "sys_open",
87     "sys_close",

```

```

88  "sys_newstat", /* 4 */
89  "sys_newfstat",
90  "sys_newlstat",
91  "sys_poll",
92  "sys_lseek",
93  "sys_mmap", /* 9 */
94  "sys_mprotect",
95  "sys_munmap",
96  "sys_brk",
97  "sys_rt_sigaction",
98  "sys_rt_sigprocmask", /* 14 */
99  "stub_rt_sigreturn",
100 "sys_ioctl",
101 "sys_pread64",
102 "sys_pwrite64",
103 "sys_readv", /* 19 */
104 "sys_writev",
105 "sys_access",
106 "sys_pipe",
107 "sys_select",
108 "sys_sched_yield", /* 24 */
109 "sys_mremap",
110 "sys_msync",
111 "sys_mincore",
112 "sys_madvise",
113 "sys_shmget", /* 29 */
114 "sys_shmat",
115 "sys_shmctl",
116 "sys_dup",
117 "sys_dup2",
118 "sys_pause", /* 34 */
119 "sys_nanosleep",
120 "sys_gettimer",
121 "sys_alarm",
122 "sys_settimer",
123 "sys_getpid", /* 39 */
124 "sys_sendfile64",
125 "sys_socket",
126 "sys_connect",
127 "sys_accept",
128 "sys_sendto", /* 44 */
129 "sys_recvfrom",
130 "sys_sendmsg",
131 "sys_recvmsg",
132 "sys_shutdown",
133 "sys_bind", /* 49 */
134 "sys_listen",
135 "sys_getsockname",

```

```

136 "sys_getpeername",
137 "sys_socketpair",
138 "sys_setsockopt", /* 54 */
139 "sys_getsockopt",
140 "sys_clone",
141 "sys_fork",
142 "sys_vfork",
143 "sys_execve", /* 59 */
144 "sys_exit",
145 "sys_wait4",
146 "sys_kill",
147 "sys_newuname",
148 "sys_semget", /* 64 */
149 "sys_semop",
150 "sys_semctl",
151 "sys_shmdt",
152 "sys_msgget",
153 "sys_msgsnd", /* 69 */
154 "sys_msgrcv",
155 "sys_msgctl",
156 "sys_fcntl",
157 "sys_flock",
158 "sys_fsync", /* 74 */
159 "sys_fdatasync",
160 "sys_truncate",
161 "sys_ftruncate",
162 "sys_getdents",
163 "sys_getcwd", /* 79 */
164 "sys_chdir",
165 "sys_fchdir",
166 "sys_rename",
167 "sys_mkdir",
168 "sys_rmdir", /* 84 */
169 "sys_creat",
170 "sys_link",
171 "sys_unlink",
172 "sys_symlink",
173 "sys_readlink", /* 89 */
174 "sys_chmod",
175 "sys_fchmod",
176 "sys_chown",
177 "sys_fchown",
178 "sys_lchown", /* 94 */
179 "sys_umask",
180 "sysgettimeofday",
181 "sys_getrlimit",
182 "sys_getrusage",
183 "sys_sysinfo", /* 99 */

```

```

184     "sys_times",
185     "sys_ptrace",
186     "sys_getuid",
187     "sys_syslog",
188     "sys_getgid", /* 104 */
189     "sys_setuid",
190     "sys_setgid",
191     "sys_geteuid",
192     "sys_getegid",
193     "sys_getpgid", /* 109 */
194     "sys_getppid",
195     "sys_getpgrp",
196     "sys_setsid",
197     "sys_setreuid",
198     "sys_setregid", /* 114 */
199     "sys_getgroups",
200     "sys_setgroups",
201     "sys_setresuid",
202     "sys_getresuid",
203     "sys_setresgid", /* 119 */
204     "sys_getresgid",
205     "sys_getpgid",
206     "sys_setfsuid",
207     "sys_setfsgid",
208     "sys_getsid", /* 124 */
209     "sys_capget",
210     "sys_capset",
211     "sys_rt_sigpending", /* 127 */
212 };
213
214 static inline void walk_bits_and_find_syscalls(struct seq_file *m, uint64_t
        num, int syscalls_arr_cnt[]) {
215     int i;
216
217     for (i = 0; i < 64; i++) {
218         if (num & (1UL << i)) {
219             syscalls_arr_cnt[i]++;
220         }
221     }
222 }
223
224 void print_syscall_statistics(struct seq_file *m, const ktime_t mstart,
        ktime_t range) {
225     int syscalls_arr_cnt[128];
226     uint64_t tmp;
227     size_t i;
228     ktime_t uptime;
229

```

```

230  memset((void*)syscalls_arr_cnt, 0, 128 * sizeof(int));
231  uptime = ktime_get_boottime_seconds() - mstart;
232
233  if (uptime < range) {
234      range = uptime;
235  }
236
237  for (i = 0; i < range; i++) {
238      if ((tmp = syscalls_time_array[uptime - i].p1) != 0) {
239          walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt);
240      }
241
242      if ((tmp = syscalls_time_array[uptime - i].p2) != 0) {
243          walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt + 64);
244      }
245  }
246
247  show_int_message(m, "Syscall statistics for the last %d seconds.\n\n", range
248  );
249
250  for (i = 0; i < 128; i++) {
251      if (syscalls_arr_cnt[i] != 0) {
252          show_str_message(m, "%s called ", syscalls_names[i]);
253          show_int_message(m, "%d times.\n", syscalls_arr_cnt[i]);
254      }
255  }

```

ЛИСТИНГ 3.11: ЛИСТИНГ файла stat.c

```

1  #include "log.h"
2
3  void show_int_message(struct seq_file *m, const char *const f, const long num)
4  {
5      char tmp[256];
6      int len;
7
8      len = snprintf(tmp, 256, f, num);
9      seq_write(m, tmp, len);
10 }
11
12 void show_int3_message(struct seq_file *m, const char *const f, const long n1,
13     const long n2, const long n3) {
14     char tmp[256];
15     int len;
16
17     len = snprintf(tmp, 256, f, n1, n2, n3);
18     seq_write(m, tmp, len);
19 }

```

```

18
19 void show_str_message(struct seq_file *m, const char *const f, const char *
    const s) {
20     char tmp[256];
21     int len;
22
23     len = snprintf(tmp, 256, f, s);
24     seq_write(m, tmp, len);
25 }

```

Листинг 3.12: Листинг файла log.c

```

1 #include "hooks.h"
2
3 #pragma GCC optimize("-fno-optimize-sibling-calls")
4
5 #if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >= KERNEL_VERSION(4,17,0))
6 #define PTREGS_SYSCALL_STUBS 1
7 #endif
8
9 #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
10 #define FTRACE_OPS_FL_RECURSION FTRACE_OPS_FL_RECURSION_SAFE
11 #endif
12
13 #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
14 #define ftrace_regs pt_regs
15
16 static __always_inline struct pt_regs *ftrace_get_regs(struct ftrace_regs *
    fregs)
17 {
18     return fregs;
19 }
20 #endif
21
22 ktime_t start_time;
23 static DEFINE_SPINLOCK(my_lock);
24
25 static void inline update_syscall_array(int syscall_num) {
26     ktime_t time;
27
28     time = ktime_get_boottime_seconds() - start_time;
29
30     spin_lock(&my_lock);
31
32     if (syscall_num < 64) {
33         syscalls_time_array[time % TIME_ARRAY_SIZE].p1 |= 1UL << syscall_num;
34     } else {
35         syscalls_time_array[time % TIME_ARRAY_SIZE].p2 |= 1UL << (syscall_num %
    64);

```

```

36     }
37
38     spin_unlock(&my_lock);
39 }
40
41 /* 0 - sys_read */
42 #ifdef PTREGS_SYSCALL_STUBS
43 static asmlinkage long (*real_sys_read)(struct pt_regs *regs);
44
45 static asmlinkage long hook_sys_read(struct pt_regs *regs)
46 {
47     update_syscall_array(SYS_READ_NUM);
48     return real_sys_read(regs);
49 }
50 #else
51 static asmlinkage long (*real_sys_read)(unsigned int fd, char __user *buf,
52     size_t count);
53
54 static asmlinkage long hook_sys_read(unsigned int fd, char __user *buf, size_t
55     count)
56 {
57     update_syscall_array(SYS_READ_NUM);
58     return real_sys_read(fd, buf, count);
59 }
60 #endif
61
62 /* 1 - sys_write */
63 #ifdef PTREGS_SYSCALL_STUBS
64 static asmlinkage long (*real_sys_write)(struct pt_regs *regs);
65
66 static asmlinkage long hook_sys_write(struct pt_regs *regs)
67 {
68     update_syscall_array(SYS_WRITE_NUM);
69     return real_sys_write(regs);
70 }
71 #else
72 static asmlinkage long (*real_sys_write)(unsigned int fd, const char __user *
73     buf, size_t count);
74
75 static asmlinkage long hook_sys_write(unsigned int fd, const char __user *buf,
76     size_t count)
77 {
78     update_syscall_array(SYS_WRITE_NUM);
79     return real_sys_write(fd, buf, count);
80 }
81 #endif
82
83 /* 2 - sys_open */

```



```

80 #ifdef PTREGS_SYSCALL_STUBShttps://github.com/oljakon/information-security/
    blob/master/lab4_RSA/main.py
81 static asmlinkage long (*real_sys_open)(struct pt_regs *regs);
82
83 static asmlinkage long hook_sys_open(struct pt_regs *regs)
84 {
85     update_syscall_array(SYS_OPEN_NUM);
86     return real_sys_open(regs);
87 }
88 #else
89 static asmlinkage long (*real_sys_open)(const char __user *filename, int flags
    , umode_t mode);
90
91 static asmlinkage long hook_sys_open(const char __user *filename, int flags,
    umode_t mode);
92 {
93     update_syscall_array(SYS_OPEN_NUM);
94     return real_sys_open(filename, flags, mode);
95 }
96 #endif
97
98 /* 3 - sys_close */
99 #ifdef PTREGS_SYSCALL_STUBS
100 static asmlinkage long (*real_sys_close)(struct pt_regs *regs);
101
102 static asmlinkage long hook_sys_close(struct pt_regs *regs)
103 {
104     update_syscall_array(SYS_CLOSE_NUM);
105     return real_sys_close(regs);
106 }
107 #else
108 static asmlinkage long (*real_sys_close)(unsigned int fd);
109
110 static asmlinkage long hook_sys_close(unsigned int fd);
111 {
112     update_syscall_array(SYS_CLOSE_NUM);
113     return real_sys_close(fd);
114 }
115 #endif
116
117 /* 9 - sys_mmap */
118 #ifdef PTREGS_SYSCALL_STUBS
119 static asmlinkage long (*real_sys_mmap)(struct pt_regs *regs);
120
121 static asmlinkage long hook_sys_mmap(struct pt_regs *regs)
122 {
123     update_syscall_array(SYS_MMAP_NUM);
124     return real_sys_mmap(regs);

```

```

125 }
126 #else
127 static asmlinkage long (*real_sys_mmap)(unsigned int fd);
128
129 static asmlinkage long hook_sys_mmap(unsigned long addr, unsigned long len,
130 int prot, int flags,
131 int fd, long off)
132 {
133     update_syscall_array(SYS_CLOSE_NUM);
134     return real_sys_mmap(addr, len, prot, flags, fd, off);
135 }
136 #endif
137
138 /* 24 - sys_sched_yield */
139 #ifdef PTREGS_SYSCALL_STUBS
140 static asmlinkage long (*real_sys_sched_yield)(struct pt_regs *regs);
141
142 static asmlinkage long hook_sys_sched_yield(struct pt_regs *regs)
143 {
144     update_syscall_array(SYS_SCHED_YIELD_NUM);
145     return real_sys_sched_yield(regs);
146 }
147 #else
148 static asmlinkage long (*real_sys_sched_yield)(void);
149
150 static asmlinkage long hook_sys_sched_yield(void)
151 {
152     update_syscall_array(SYS_SCHED_YIELD_NUM);
153     return real_sys_sched_yield();
154 }
155 #endif
156
157 /* 41 - sys_socket */
158 #ifdef PTREGS_SYSCALL_STUBS
159 static asmlinkage long (*real_sys_socket)(struct pt_regs *regs);
160
161 static asmlinkage long hook_sys_socket(struct pt_regs *regs)
162 {
163     update_syscall_array(SYS_SOCKET_NUM);
164     return real_sys_socket(regs);
165 }
166 #else
167 static asmlinkage long (*real_sys_socket)(int, int, int);
168
169 static asmlinkage long hook_sys_socket(int a, int b, int c)
170 {
171     update_syscall_array(SYS_SOCKET_NUM);
172     return real_sys_socket(a, b, c);

```

```

173 }
174 #endif
175
176 /* 42 - sys_connect */
177 #ifdef PTREGS_SYSCALL_STUBS
178 static asmlinkage long (*real_sys_connect)(struct pt_regs *regs);
179
180 static asmlinkage long hook_sys_connect(struct pt_regs *regs)
181 {
182     update_syscall_array(SYS_CONNECT_NUM);
183     return real_sys_connect(regs);
184 }
185 #else
186 static asmlinkage long (*real_sys_connect)(int, struct sockaddr __user *, int)
187     ;
188
189 static asmlinkage long hook_sys_connect(int a, struct sockaddr __user * b, int
190     c);
191 {
192     update_syscall_array(SYS_CONNECT_NUM);
193     return real_sys_connect(a, b, c);
194 }
195 #endif
196
197 /* 43 - sys_accept */
198 #ifdef PTREGS_SYSCALL_STUBS
199 static asmlinkage long (*real_sys_accept)(struct pt_regs *regs);
200
201 static asmlinkage long hook_sys_accept(struct pt_regs *regs)
202 {
203     update_syscall_array(SYS_ACCEPT_NUM);
204     return real_sys_accept(regs);
205 }
206 #else
207 static asmlinkage long (*real_sys_accept)(int, struct sockaddr __user *, int
208     __user *)
209     ;
210
211 static asmlinkage long hook_sys_accept(int a, struct sockaddr __user * b, int
212     __user *c)
213 {
214     update_syscall_array(SYS_ACCEPT_NUM);
215     return real_sys_accept(a, b, c);
216 }
217 #endif
218
219 /* 44 - sys_sendto */
220 #ifdef PTREGS_SYSCALL_STUBS
221 static asmlinkage long (*real_sys_sendto)(struct pt_regs *regs);

```

```

217
218 static asmlinkage long hook_sys_sendto(struct pt_regs *regs)
219 {
220     update_syscall_array(SYS_SENDTO_NUM);
221     return real_sys_sendto(regs);
222 }
223 #else
224 static asmlinkage long (*real_sys_sendto)(int, void __user *, size_t, unsigned
225 ,
226 struct sockaddr __user *, int);
227 static asmlinkage long hook_sys_sendto(int a, void __user * b, size_t c,
228     unsigned d,
229 struct sockaddr __user *e, int f);
230 {
231     update_syscall_array(SYS_SENDTO_NUM);
232     return real_sys_sendto(a, b, c, d, e, f);
233 }
234 #endif
235 /* 45 - sys_recvfrom */
236 #ifdef PTREGS_SYSCALL_STUBS
237 static asmlinkage long (*real_sys_recvfrom)(struct pt_regs *regs);
238
239 static asmlinkage long hook_sys_recvfrom(struct pt_regs *regs)
240 {
241     update_syscall_array(SYS_RECVFROM_NUM);
242     return real_sys_recvfrom(regs);
243 }
244 #else
245 static asmlinkage long (*real_sys_recvfrom)(int, void __user *, size_t,
246     unsigned,
247 struct sockaddr __user *, int __user *)
248
249 static asmlinkage long hook_sys_recvfrom(int a, void __user *b, size_t c,
250     unsigned d,
251 struct sockaddr __user * e, int __user *f)
252 {
253     update_syscall_array(SYS_RECVFROM_NUM);
254     return real_sys_recvfrom(a, b, c, d, e, f);
255 }
256 #endif
257 /* 46 - sys_sendmsg */
258 #ifdef PTREGS_SYSCALL_STUBS
259 static asmlinkage long (*real_sys_sendmsg)(struct pt_regs *regs);
260
261 static asmlinkage long hook_sys_sendmsg(struct pt_regs *regs)

```

```

261 {
262     update_syscall_array(SYS_SENDMSG_NUM);
263     return real_sys_sendmsg(regs);
264 }
265 #else
266 static asmlinkage long (*real_sys_sendmsg)(int fd, struct user_msghdr __user *
      msg, unsigned flags);
267
268 static asmlinkage long hook_sys_sendmsg(int fd, struct user_msghdr __user *msg
      , unsigned flags)
269 {
270     update_syscall_array(SYS_SENDMSG_NUM);
271     return real_sys_sendmsg(fd, msg, flags);
272 }
273 #endif
274
275 /* 47 - sys_recvmsg */
276 #ifdef PTREGS_SYSCALL_STUBS
277 static asmlinkage long (*real_sys_recvmsg)(struct pt_regs *regs);
278
279 static asmlinkage long hook_sys_recvmsg(struct pt_regs *regs)
280 {
281     update_syscall_array(SYS_RECVMSG_NUM);
282     return real_sys_recvmsg(regs);
283 }
284 #else
285 static asmlinkage long (*real_sys_recvmsg)(int fd, struct user_msghdr __user *
      msg, unsigned flags);
286
287 static asmlinkage long hook_sys_recvmsg(int fd, struct user_msghdr __user *msg
      , unsigned flags)
288 {
289     update_syscall_array(SYS_RECVMSG_NUM);
290     return real_sys_recvmsg(fd, msg, flags);
291 }
292 #endif
293
294 /* 48 - sys_shutdown */
295 #ifdef PTREGS_SYSCALL_STUBS
296 static asmlinkage long (*real_sys_shutdown)(struct pt_regs *regs);
297
298 static asmlinkage long hook_sys_shutdown(struct pt_regs *regs)
299 {
300     update_syscall_array(SYS_SHUTDOWN_NUM);
301     return real_sys_shutdown(regs);
302 }
303 #else
304 static asmlinkage long (*real_sys_shutdown)(int, int);

```

```

305
306 static asmlinkage long hook_sys_shutdown(int t, int m)
307 {
308     update_syscall_array(SYS_SHUTDOWN_NUM);
309     return real_sys_shutdown(t, m);
310 }
311 #endif
312
313 /* 56 - sys_clone */
314 #ifdef PTREGS_SYSCALL_STUBS
315 static asmlinkage long (*real_sys_clone)(struct pt_regs *regs);
316
317 static asmlinkage long hook_sys_clone(struct pt_regs *regs)
318 {
319     update_syscall_array(SYS_CLONE_NUM);
320     return real_sys_clone(regs);
321 }
322 #else
323 static asmlinkage long (*real_sys_clone)(unsigned long clone_flags,
324 unsigned long newsp, int __user *parent_tidptr,
325 int __user *child_tidptr, unsigned long tls);
326
327 static asmlinkage long hook_sys_clone(unsigned long clone_flags,
328 unsigned long newsp, int __user *parent_tidptr,
329 int __user *child_tidptr, unsigned long tls)
330 {
331     update_syscall_array(SYS_CLONE_NUM);
332     return real_sys_clone(clone_flags, newsp, parent_tidptr, child_tidptr, tls);
333 }
334 #endif
335
336 /* 59 - sys_execve */
337 #ifdef PTREGS_SYSCALL_STUBS
338 static asmlinkage long (*real_sys_execve)(struct pt_regs *regs);
339
340 static asmlinkage long hook_sys_execve(struct pt_regs *regs)
341 {
342     update_syscall_array(SYS_EXECVE_NUM);
343     return real_sys_execve(regs);
344 }
345 #else
346 static asmlinkage long (*real_sys_execve)(const char __user *filename,
347 const char __user *const __user *argv,
348 const char __user *const __user *envp);
349
350 static asmlinkage long hook_sys_execve(const char __user *filename,
351 const char __user *const __user *argv,
352 const char __user *const __user *envp)

```

```

353 {
354     update_syscall_array(SYS_EXECVE_NUM);
355     return real_sys_execve(filename, argv, envp);
356 }
357 #endif
358
359 /* 83 - sys_mkdir */
360 #ifdef PTREGS_SYSCALL_STUBS
361 static asmlinkage long (*real_sys_mkdir)(struct pt_regs *regs);
362
363 static asmlinkage long hook_sys_mkdir(struct pt_regs *regs)
364 {
365     update_syscall_array(SYS_MKDIR_NUM);
366     return real_sys_mkdir(regs);
367 }
368 #else
369 static asmlinkage long (*real_sys_mkdir)(const char __user *pathname, umode_t
    mode);
370
371 static asmlinkage long hook_sys_mkdir(const char __user *pathname, umode_t
    mode);
372 {
373     update_syscall_array(SYS_MKDIR_NUM);
374     return real_sys_mkdir(pathname, mode);
375 }
376 #endif
377
378 /* 84 - sys_rmdir */
379 #ifdef PTREGS_SYSCALL_STUBS
380 static asmlinkage long (*real_sys_rmdir)(struct pt_regs *regs);
381
382 static asmlinkage long hook_sys_rmdir(struct pt_regs *regs)
383 {
384     update_syscall_array(SYS_RMDIR_NUM);
385     return real_sys_rmdir(regs);
386 }
387 #else
388 static asmlinkage long (*real_sys_rmdir)(const char __user *pathname);
389
390 static asmlinkage long hook_sys_rmdir(const char __user *pathname);
391 {
392     update_syscall_array(SYS_RMDIR_NUM);
393     return real_sys_rmdir(pathname);
394 }
395 #endif
396
397
398 /*

```

```

399 * x86_64 kernels have a special naming convention for syscall entry points in
      newer kernels.
400 * That's what you end up with if an architecture has 3 (three) ABIs for system
      calls.
401 */
402 #ifdef PTREGS_SYSCALL_STUBS
403 #define SYSCALL_NAME(name) ("__x64_" name)
404 #else
405 #define SYSCALL_NAME(name) (name)
406 #endif
407
408 #define ADD_HOOK(_name, _function, _original) \
409 { \
410     .name = SYSCALL_NAME(_name), \
411     .function = (_function), \
412     .original = (_original), \
413 }
414
415 static struct ftrace_hook hooked_functions[] = {
416     ADD_HOOK("sys_execve", hook_sys_execve, &real_sys_execve),
417     ADD_HOOK("sys_write", hook_sys_write, &real_sys_write),
418     ADD_HOOK("sys_open", hook_sys_open, &real_sys_open),
419     ADD_HOOK("sys_close", hook_sys_close, &real_sys_close),
420     ADD_HOOK("sys_mmap", hook_sys_mmap, &real_sys_mmap),
421     ADD_HOOK("sys_sched_yield", hook_sys_sched_yield, &real_sys_sched_yield),
422     ADD_HOOK("sys_socket", hook_sys_socket, &real_sys_socket),
423     ADD_HOOK("sys_connect", hook_sys_connect, &real_sys_connect),
424     ADD_HOOK("sys_accept", hook_sys_accept, &real_sys_accept),
425     ADD_HOOK("sys_sendto", hook_sys_sendto, &real_sys_sendto),
426     ADD_HOOK("sys_recvfrom", hook_sys_recvfrom, &real_sys_recvfrom),
427     ADD_HOOK("sys_sendmsg", hook_sys_sendmsg, &real_sys_sendmsg),
428     ADD_HOOK("sys_recvmsg", hook_sys_recvmsg, &real_sys_recvmsg),
429     ADD_HOOK("sys_shutdown", hook_sys_shutdown, &real_sys_shutdown),
430     ADD_HOOK("sys_read", hook_sys_read, &real_sys_read),
431     ADD_HOOK("sys_clone", hook_sys_clone, &real_sys_clone),
432     ADD_HOOK("sys_mkdir", hook_sys_mkdir, &real_sys_mkdir),
433     ADD_HOOK("sys_rmdir", hook_sys_rmdir, &real_sys_rmdir),
434 };
435
436 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0)
437 static unsigned long lookup_name(const char *name)
438 {
439     struct kprobe kp = {
440         .symbol_name = name
441     };
442     unsigned long retval;
443
444     ENTER_LOG();

```



```

445
446     if (register_kprobe(&kp) < 0) {
447         EXIT_LOG();
448         return 0;
449     }
450
451     retval = (unsigned long) kp.addr;
452     unregister_kprobe(&kp);
453
454     EXIT_LOG();
455
456     return retval;
457 }
458 #else
459 static unsigned long lookup_name(const char *name)
460 {
461     unsigned long retval;
462
463     ENTER_LOG();
464     retval = kallsyms_lookup_name(name);
465     EXIT_LOG();
466
467     return retval;
468 }
469 #endif
470
471 static int resolve_hook_address(struct ftrace_hook *hook)
472 {
473     ENTER_LOG();
474
475     if (!(hook->address = lookup_name(hook->name))) {
476         pr_debug("unresolved symbol: %s\n", hook->name);
477         EXIT_LOG();
478         return -ENOENT;
479     }
480
481     *((unsigned long*) hook->original) = hook->address;
482
483     EXIT_LOG();
484
485     return 0;
486 }
487
488 static void notrace ftrace_thunk(unsigned long ip, unsigned long parent_ip,
489 struct ftrace_ops *ops, struct ftrace_regs *fregs)
490 {
491     struct pt_regs *regs = ftrace_get_regs(fregs);
492     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);

```

```

493
494     if (!within_module(parent_ip, THIS_MODULE)) {
495         regs->ip = (unsigned long)hook->function;
496     }
497 }
498
499 static int install_hook(struct ftrace_hook *hook) {
500     int rc;
501
502     ENTER_LOG();
503
504     if ((rc = resolve_hook_address(hook))) {
505         EXIT_LOG();
506         return rc;
507     }
508
509     /* Callback function. */
510     hook->ops.func = ftrace_thunk;
511     /* Save processor registers. */
512     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
513         | FTRACE_OPS_FL_RECURSION
514         | FTRACE_OPS_FL_IPMODIFY;
515
516     /* Turn of ftrace for our function. */
517     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0))) {
518         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
519         return rc;
520     }
521
522     /* Allow ftrace call our callback. */
523     if ((rc = register_ftrace_function(&hook->ops))) {
524         pr_debug("register_ftrace_function() failed: %d\n", rc);
525         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
526     }
527
528     EXIT_LOG();
529
530     return rc;
531 }
532
533 static void remove_hook(struct ftrace_hook *hook) {
534     int rc;
535
536     ENTER_LOG();
537
538     if (hook->address == 0x00) {
539         EXIT_LOG();
540         return;

```

```

541     }
542
543     if ((rc = unregister_ftrace_function(&hook->ops))) {
544         pr_debug("unregister_ftrace_function() failed: %d\n", rc);
545     }
546
547     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0))) {
548         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
549     }
550
551     hook->address = 0x00;
552
553     EXIT_LOG();
554 }
555
556 int install_hooks(void) {
557     size_t i;
558     int rc;
559
560     ENTER_LOG();
561
562     for (i = 0; i < ARRAY_SIZE(hooked_functions); i++) {
563         if ((rc = install_hook(&hooked_functions[i]))) {
564             pr_debug("instal_hooks failed: %d\n", rc);
565             goto err;
566         }
567     }
568
569     EXIT_LOG();
570
571     return 0;
572
573     err:
574     while (i != 0) {
575         remove_hook(&hooked_functions[--i]);
576     }
577
578     EXIT_LOG();
579
580     return rc;
581 }
582
583 void remove_hooks(void) {
584     size_t i;
585
586     ENTER_LOG();
587
588     for (i = 0; i < ARRAY_SIZE(hooked_functions); i++) {

```

```
589     remove_hook(&hooked_functions[i]);  
590 }  
591  
592 EXIT_LOG();  
593 }
```

Листинг 3.13: Листинг файла hooks.c