

Оглавление

| | |
|---|-----------|
| Введение | 2 |
| 1 Аналитическая часть | 3 |
| 1.1 Трассировка ядра | 3 |
| 1.1.1 Linux Security Module | 3 |
| 1.1.2 Модификация таблицы системных вызовов | 4 |
| 1.1.3 kprobes | 5 |
| 1.1.4 Kernel tracepoints | 6 |
| 1.1.5 ftrace | 6 |
| 1.2 Информация о процессах и памяти | 8 |
| 1.2.1 Структура <code>struct task_struct</code> | 8 |
| 1.2.2 Структура <code>struct sysinfo</code> | 10 |
| 1.3 Загружаемые модули ядра | 11 |
| 1.3.1 Пространство ядра и пользователя | 11 |
| 1.4 Виртуальная файловая система <code>/proc</code> | 12 |
| 2 Конструкторская часть | 15 |
| 2.1 Архитектура приложения | 15 |
| 2.2 Алгоритм перехвата системного вызова | 15 |
| 2.3 Алгоритм подсчёта количества системных вызовов | 17 |
| 3 Технологическая часть | 19 |
| Заключение | 20 |
| Литература | 21 |

Введение

Работая с операционной системой Linux [1], пользователю может потребоваться отслеживать её загрузенность. Для обнаружения и предотвращения сбоев необходимо иметь хорошую систему мониторинга, которая будет анализировать работу операционной системы. Данный курсовой проект посвящен исследованию структур ядра, хранящим информацию о процессах в системе и памяти, и способам перехвата системных вызовов ядра с их последующим логированием.

Целью данной курсовой работы является разработка загружаемого модуля ядра, предоставляющего информацию о загрузенности системы: количество системных вызовов за выбранный промежуток времени, количество выделенной памяти в текущий момент, статистика по процессам и в каких состояниях они находятся.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить структуры и функции ядра, которые предоставляют информацию о процессах и памяти;
- проанализировать существующие подходы к перехвату системных вызовов и выбрать наиболее подходящий;
- реализовать загружаемый модуль ядра.

1 Аналитическая часть

В данном разделе будут рассмотрены и проанализированы:

- различные подходы к трассировке ядра и перехвату функций;
- структуры и функции ядра, предоставляющие информацию о процессах и памяти;
- основные принципы загружаемых модулей ядра;
- способы получения пользователем информации из ядра.

1.1 Трассировка ядра

Трассировка ядра – получение информации о том, что происходит внутри работающей системы. Для этого используются специальные программные инструменты, регистрирующие все происходящие события в системе.

Такие программы могут одновременно отслеживать события как на уровне отдельных приложений, так и на уровне операционной системы. Полученная в ходе трассировки информации может оказаться полезной для диагностики и решения системных проблем.

Во время трассировки записывается информация о событиях, происходящих на низком уровне. Их количество исчисляется сотнями и даже тысячами.

Далее будут рассмотрены существующие различные подходы к трассировке ядра и перехвату вызываемых функций, и выбран наиболее подходящий для реализации в курсовой работе.

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.1.4 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].

1.1.5 ftrace

`ftrace` [7] – это фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора `-pg` [8] и `mfentry` [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;

Вывод

В таблице 1.1 приведено сравнение приведенных выше технологий трассировки ядра.

| 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: Сравнение технологий, позволяющих трассировать ядро

В ходе анализа подходов к перехвату функций, 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.

1.2 Информация о процессах и памяти

1.2.1 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)
3 #define task_is_stopped(task)    ((READ_ONCE(task->__state) & __TASK_STOPPED)
!= 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

1.2.2 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()`.

1.3 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.

1.3.1 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.

1.4 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 трассировке ядра и перехвату функций и был выбран наиболее оптимальный метод для реализации поставленной задачи. Были рассмотрены структуры и функции ядра, предоставляющие информацию о процессах и памяти; основные принципы 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

В данном разделе будет рассмотрена общая архитектура приложения, алгоритм перехвата системных вызовов и подсчёт количества этих вызовов за выбранный промежуток времени.

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 о состоянии о загруженности оперативной памяти – её общее количество, свободной и доступной в данный момент.

2.2 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 `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`. Пе-

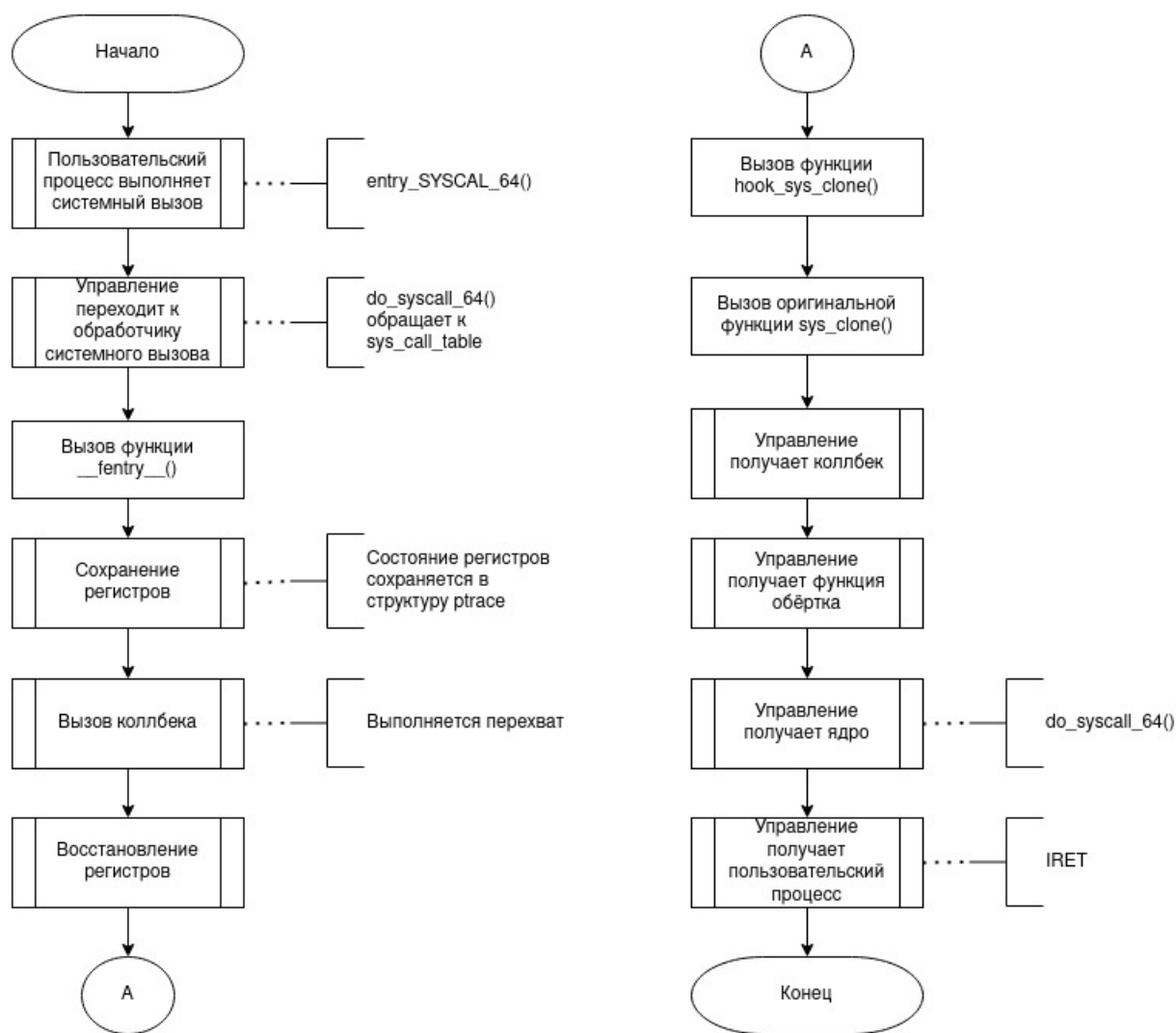


Рис. 2.1: Алгоритм перехвата системного вызова

ред 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 регистр `rip` – это приведёт к передаче управления по новому адресу.
7. Управление получает функция обёртка. Благодаря безусловному пе-

реходу, 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.

2.3 Алгоритм подсчёта количества системных вызовов

На рисунке 2.2 представлена схема алгоритма подсчёта системных вызовов.



Рис. 2.2: Алгоритм подсчёта количества системных вызовов

- Аггрегирующий массив – это массив на 86400 элементов, состоящий из структур, имеющих два поля в виде 64-битных беззнаковых 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 количества этих вызовов за выбранный промежуток.

3 Технологическая часть

В данном разделе рассматривается выбор языка программирования для реализации поставленной задачи, листинги реализации разработанного программного обеспечения и приведены результаты работы ПО.

Вывод

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

Заключение

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).