СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При работе с операционной системой Linux может потребоваться перехватывать вызовы важных функций внутри ядра (например, запуск процессов или запись и чтение с жесткого диска) для обеспечения возможности мониторинга активности в системе или превентивного блокирования деятельности подозрительных процессов. Курсовой проект посвящен исследованию способов перехвата вызовов функций внутри ядра с их последующим логированием и представлением в графической форме для наглядного мониторинга.

Цель – разработка загружаемого модуля ядра для мониторинга системных вызовов sys_clone, sys_execve и операции с дисками bdev_read_page, bdev_write_page. Собранные данные о количестве вызовов функций необходимо сохранить в лог-файле и представить в графическом виде; сохранить статистику всех вызовов функций за определенные промежутки времени и визуализировать полученные данные.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием необходимо разработать загружаемый модуля ядра для мониторинга системных вызовов sys_clone, sys_execve и операции с дисками bdev_read_page, bdev_write_page. На основе данных о количестве вызовов перечисленных функций составить статистику вызовов функций за определенные промежутки времени и визуализировать полученные данные.

Для решения данной задачи необходимо:

- проанализировать существующие подходы для трассировки ядра и перехвата функций;
- изучить средства сбора статистики и визуализации;
- реализовать загружаемый модуль ядра.

1.2 Трассировка ядра

Под трассировкой понимается получение информации о том, что происходит внутри работающей системы. Для этого используются специальные программные инструменты, регистрирующие все события в системе.

Программы-трассировщики могут одновременно отслеживать события как на уровне отдельных приложений, так и на уровне операционной системы. Полученная в ходе трассировки информация может оказаться полезной для диагностики и решения многих системных проблем.

Трассировку иногда сравнивают с логгированием. Сходство между этими двумя процедурами действительно есть, но есть и различия.

Во время трассировки записывается информация о событиях, происходящих на низком уровне. Их количество исчисляется сотнями и даже тысячами. В логи же записывается информация о высокоуровневых событиях, которые случаются гораздо реже: например, вход пользователей в систему, ошибки в работе приложений, транзакции в базах данных и другие [1].

Далее будут рассмотрены различные подходы к трассировке ядра и перехвату вызываемых функций.

1.2.1 Linux Security API

Linux Security API – специальный интерфейс, созданный именно для перехвата функций. В критических местах кода ядра расположены вызовы security-функций, которые в свою очередь вызывают коллбеки, установленные security-модулем. Security-модуль может изучать контекст операции и принимать решение о ее разрешении или запрете [2].

Недостатками данного подхода являются:

- security-модули являются частью ядра и не могут быть загружены динамически;
- в системе может быть только один security-модуль (с небольшими исключениями).

Таким образом, для использования Security API необходимо использовать собственную сборку ядра Linux, в которую входит собственный security-модуль.

1.2.2 Модификация таблицы системных вызовов

В ОС Linux адреса всех обработчиков системных вызовов расположены в таблице sys_call_table. Подмена адреса в этой таблице приводит к смене поведения системного вызова, для которого поменяли адрес обработчика. Таким образом, сохранив адрес исходного обработчика и подставив в таблицу адрес собственного обработчика, можно перехватить любой системный вызов [2].

Преимуществами данного подхода являются:

- контроль над любыми системными вызовами – единственным интерфейсом к ядру у пользовательских приложений;
- не требуется специальная конфигурация ядра, следовательно, поддерживает максимально широкий спектр систем.

Недостатками данного подхода являются:

- необходимость модифицировать таблицу системных вызовов;

- невозможность перехвата некоторых обработчиков. В ядрах до версии 4.16 обработка системных вызовов для архитектуры x86_64 содержала целый ряд оптимизаций. Некоторые из них требовали того, что обработчик системного вызова являлся специальным переходником, реализованным на ассемблере. Соответственно, подобные обработчики порой сложно, а иногда и вовсе невозможно заменить на свои, написанные на языке C. Более того, в разных версиях ядра используются разные оптимизации, что так же добавляет технических сложностей.

- перехватываются только системные вызовы. Этот подход позволяет заменять обработчики системных вызовов, что ограничивает точки входа только ими.

Данный подход позволяет подменить большинство обработчиков системных вызовов, что является несомненным плюсом, но также ограничивает количество функций, которые можно перехватить.

1.2.3 kprobes

kprobes – специализированный API, в первую очередь предназначенный для отладки и трассирования ядра. Этот интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут их изменять. Таким образом, можно получить как мониторинг, так и возможность влиять на дальнейший ход работы [2].

Преимуществами данного подхода являются:

- хорошо задокументированный интерфейс, работа kprobes по возможности оптимизирована;

- перехват любой инструкции в ядре: это реализуются с помощью точек останова (инструкции INT3), внедряемых в исполняемый код ядра. Тем самым, можно перехватить любую функцию в ядре.

Недостатками данного подхода являются:

- техническая сложность. Kprobes – это только способ установить точку останова в любой инструкции в ядре. Для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах

или где в стеке они расположены, и самостоятельно их оттуда извлекать. Для блокировки вызова функции необходимо вручную модифицировать состояние процесса так, чтобы процессор подумал, что он уже вернул управление из функции;

- Jprobes объявлены устаревшими. Jprobes – это надстройка над kprobes, позволяющая удобно перехватывать вызовы функций. Она самостоятельно извлечет аргументы функции из регистров или стека и вызовет обработчик, который должен иметь ту же сигнатуру, что и перехватываемая функция. Недостаток в том, что jprobes объявлены устаревшими и вырезаны из современных ядер;

- нетривиальные накладные расходы. Расстановка точек останова – дорогая операция, но она выполняется единоразово. Точки останова не влияют на остальные функции, однако их обработка относительно недешевая. К счастью, для архитектуры x86_64 реализована jump-оптимизация, существенно уменьшающая стоимость kprobes, но она все еще остается больше, чем, например, при модификации таблицы системных вызовов;

- kretprobes реализуются через подмену адреса возврата в стеке. Соответственно, им необходимо где-то хранить оригинальный адрес, чтобы вернуться туда после обработки kretprobe. Адреса хранятся в буфере фиксированного размера. В случае его переполнения, когда в системе выполняется слишком много одновременных вызовов перехваченной функции, kretprobes будет пропускать срабатывания.

Данный подход обладает сложной технической реализацией, а также есть вероятность возникновения ошибок после переполнения буфера памяти.

1.2.4 Kernel tracepoints

Kernel tracepoints – это фреймворк для трассировки ядра, который позволяет встраивать точки останова в код обработчиков системных вызовов [3].

Преимуществом данного подхода является минимальное количество действий для перехвата: нужно только вызвать функцию трассировки в необходимом месте.

Недостатками данного подхода являются:

- отсутствие хорошо задокументированного API;
- не заработают в модуле, если включен CONFIG_MODULE_SIG и нет закрытого ключа для подписи.

1.2.5 ftrace

ftrace – это фреймворк для трассирования ядра на уровне функций. ftrace был разработан Стивеном Ростедтом и добавлен в ядро в 2008 году, начиная с версии 2.6.27. Работает ftrace на базе файловой системы debugfs, которая в большинстве современных дистрибутивов Linux смонтирована по умолчанию.

Реализуется ftrace на основе ключей компилятора -pg и -mfentry, которые вставляют в начало каждой функции вызов специальной трассировочной функции mcount() или __fentry__(). Обычно, в пользовательских программах эта возможность компилятора используется профилировщиками, чтобы отслеживать вызовы всех функций. Ядро же использует эти функции для реализации фреймворка ftrace.

Также доступна оптимизация: динамический ftrace. Суть в том, что ядро знает расположение всех вызовов mcount() или __fentry__() и на ранних этапах загрузки заменяет их машинный код на инструкцию NOP – специальную ничего не делающую инструкцию. Таким образом, если ftrace не используется, то его влияние на систему минимально [2].

Преимуществами данного подхода являются:

- перехват любой функции по имени. Для указания интересующей функции достаточно знать только ее имя;
- перехват совместим с трассировкой. Очевидно, что этот способ не конфликтует с ftrace, так что с ядра все еще можно снимать очень полезные показатели производительности.

Недостатками данного подхода являются:

- требования к конфигурации ядра. Для успешного выполнения перехвата функций с помощью ftrace ядро должно предоставлять целый ряд возможностей:

– список символов kallsyms для поиска функций по имени;

– фреймворк ftrace в целом для выполнения трассировки;

– опции ftrace, критически важные для перехвата.

Обычно ядра, используемые популярными дистрибутивами, все эти опции в себе все равно содержат, так как они не влияют на производительность и полезны при отладке [3].

1.2.6 Сравнение способов

В качестве критериев выбора способа трассировки ядра и перехвату функций были выбраны следующие:

- наличие задокументированного API;
- техническая простота реализации;
- динамическая загрузка;
- перехват всех функций.

В таблице 1.1 приведено сравнение способов трассировки ядра и перехвата функций.

Таблица 1.1 – Сравнение способов трассировки ядра и перехвата функций

	Наличие задокументированного API	Техническая простота реализации	Динамическая загрузка	Перехват всех функций
Linux Security API	-	+	-	+
Модификация таблицы системных вызовов	-	+	+	-
kprobes	+	-	+	+
Kernel tracepoints	-	+	+	+
ftrace	+	+	+	+

1.3 Визуализация данных

Визуализация количества вызовов функций нужна для того, чтобы можно было наглядно оценить состояние системы без необходимости разбираться с лог-файлами.

1.3.1 Loki

Loki – это набор компонентов для полноценной системы работы с логами. В отличие от других подобных систем Loki основан на идее индексировать только метаданные логов — labels (так же, как и в Prometheus), а сами логи сжимать рядом [4].

Loki-стек состоит из трех компонентов: Promtail, Loki, Grafana. Promtail собирает логи, обрабатывает их и отправляет в Loki. Loki их хранит. А Grafana умеет запрашивать данные из Loki и показывать их. Loki можно использовать не только для хранения логов и поиска по ним. Весь стек дает большие возможности по обработке и анализу поступающих данных [4].

1.3.2 Grafana

Grafana – это платформа с открытым исходным кодом для визуализации, мониторинга и анализа данных. Grafana позволяет пользователям создавать дашборды с панелями, каждая из которых отображает определенные показатели в течение установленного периода времени. Каждый дашборд универсален, поэтому его можно настроить для конкретного проекта или с учетом любых потребностей разработки [5].

Искать по логам можно в специальном интерфейсе Grafana – Explorer. Для запросов используется язык LogQL.

1.4 Выводы

В результате проведенного сравнительного анализа (таблица 1.1) был выбран способ трассировки ядра – ftrace, обеспечивающий динамическую загрузку и перехват всех функций.

Для сбора статистики вызовов был выбран набор компонентов Loki.

Для визуализации данных была выбрана платформа с открытым иходным кодом Grafana.

2 Конструкторский раздел

2.1 Последовательность преобразований

На рисунках 2.1 и 2.2 представлена IDEF0 последовательность преобразований.

Рисунок 2.1 – IDEF0 нулевого уровня

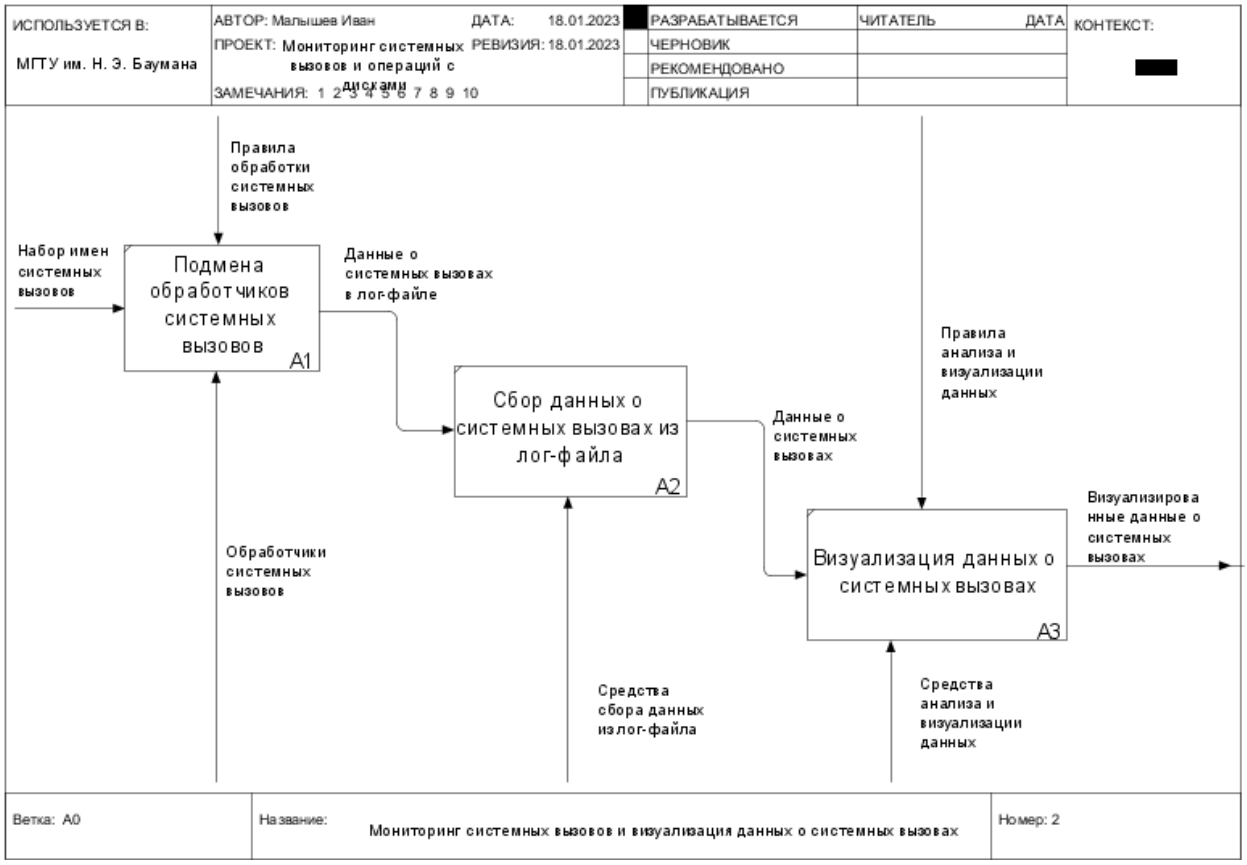

Рисунок 2.2 – IDEF0 первого уровня

2.2 Алгоритм перехвата функции

На рисунке 2.3 представлен алгоритм перехвата функции.

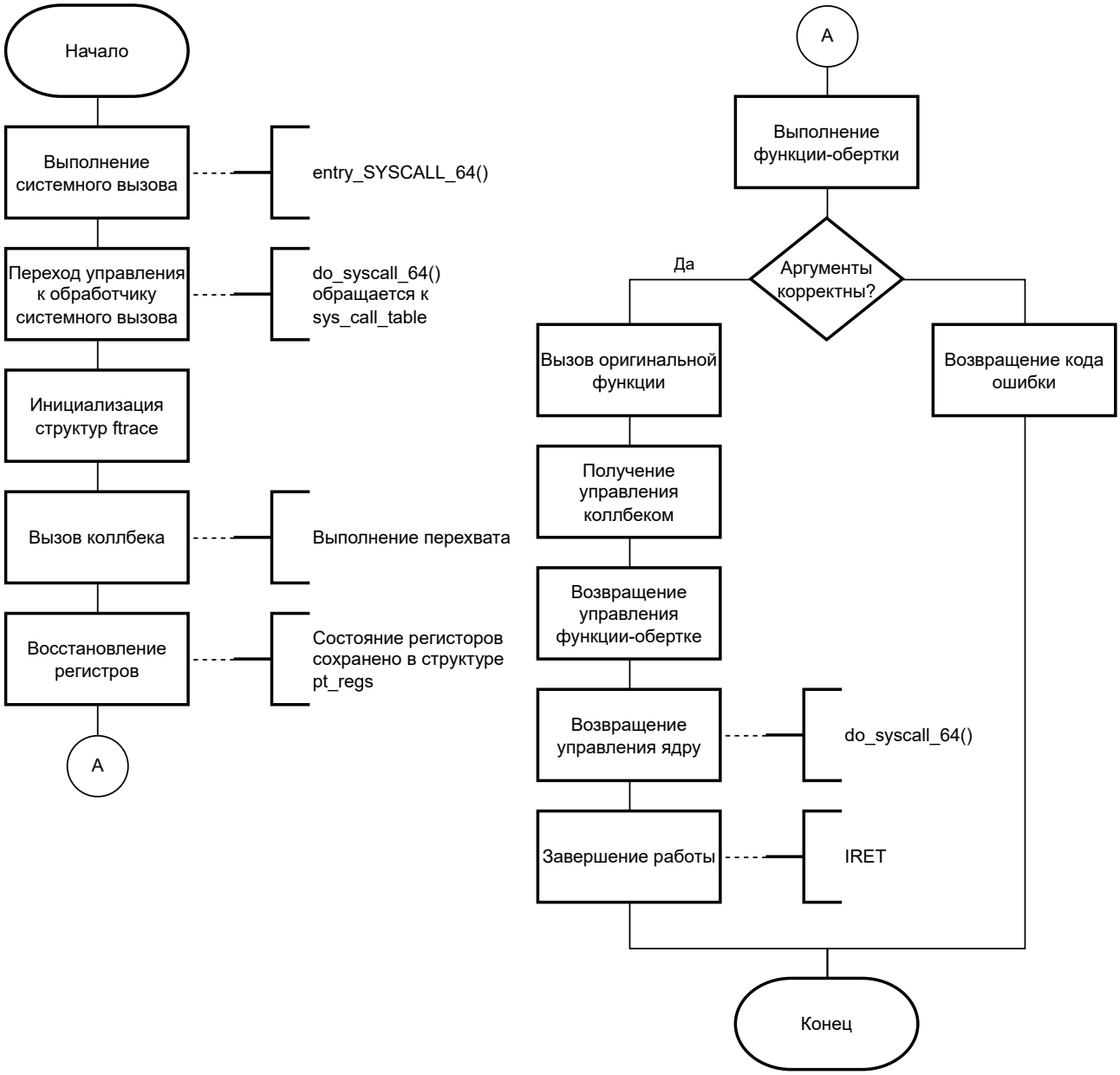

Рисунок 2.3 – Алгоритм перехвата функций

Далее представлено более подробное описание перехвата функций на примере функции sys_execve.

1. Пользовательский процесс выполняет SYSCALL. С помощью этой инструкции выполняется переход в режим ядра и управление передается низкоуровневому обработчику системных вызовов – entry_SYSCALL_64(). Он отвечает за все системные вызовы 64-битных программ на 64-битных ядрах.

2. Управление переходит к конкретному обработчику. Ядро передает управление высокоуровневой функции do_syscall_64(), написанной на C. Эта

функция в свою очередь обращается к таблице обработчиков системных вызовов sys_call_table и вызывает оттуда конкретный обработчик по номеру системного вызова – sys_execve().

3. Вызывается ftrace. В начале каждой функции ядра находится вызов функции __fentry__(), которая реализуется фреймворком ftrace.

4. ftrace вызывает разработанный коллбек.

5. Коллбек выполняет перехват.

6. ftrace восстанавливает регистры. Следуя флагу FTRACE_SAVE_REGS, ftrace сохраняет состояние регистров в структуре pt_regs перед вызовом обработчиков. При завершении обработки ftrace восстанавливает регистры из этой структуры. Наш обработчик изменяет регистр %rip – указатель на следующую исполняемую инструкцию – что в итоге приводит к передаче управления по новому адресу.

7. Управление получает функция-обертка. Из-за безусловного перехода активация функции sys_execve() прерывается. Вместо нее управление получает функция hook_sys_execve(). При этом все остальное состояние процессора и памяти остается без изменений, поэтому данная функция получает все аргументы оригинального обработчика и при завершении вернет управление в функцию do_syscall_64().

8. Обертка вызывает оригинальную функцию. Функция hook_sys_execve() может проанализировать аргументы и контекст системного вызова (кто что запускает) и запретить или разрешить процессу его выполнение. В случае запрета функция просто возвращает код ошибки. Иначе же ей следует вызвать оригинальный обработчик – sys_execve() вызывается повторно, через указатель real_sys_execve, который был сохранен при настройке перехвата.

9. Управление получает коллбек. Как и при первом вызове sys_execve(), управление опять проходит через ftrace и передается в коллбек.

10. Коллбек ничего не делает. Потому что в этот раз функция sys_execve() вызывается функцией hook_sys_execve(), а не ядром из do_syscall_64().

Поэтому коллбек не модифицирует регистры и выполнение функции sys_execve() продолжается как обычно.

11. Управление возвращается обертке.

12. Управление возвращается ядру. Функция hook_sys_execve() завершается и управление переходит в do_syscall_64(), которая считает, что системный вызов был завершен как обычно.

13. Управление возвращается в пользовательский процесс. Наконец ядро выполняет инструкцию IRET (или SYSRET, но для execve() – всегда IRET), устанавливая регистры для нового пользовательского процесса и переводя центральный процессор в режим исполнения пользовательского кода. Системный вызов (и запуск нового процесса) завершен.

2.3 Алгоритм сбора и визуализации данных

На рисунке 2.4 представлен алгоритм сбора и визуализации данных о системных вызовах.

Рисунок 2.4 – Алгоритм сбора и визуализации данных о системных вызовах

2.4 Структура программного обеспечения

На рисунке 2.5 представлена структура программного обеспечения.

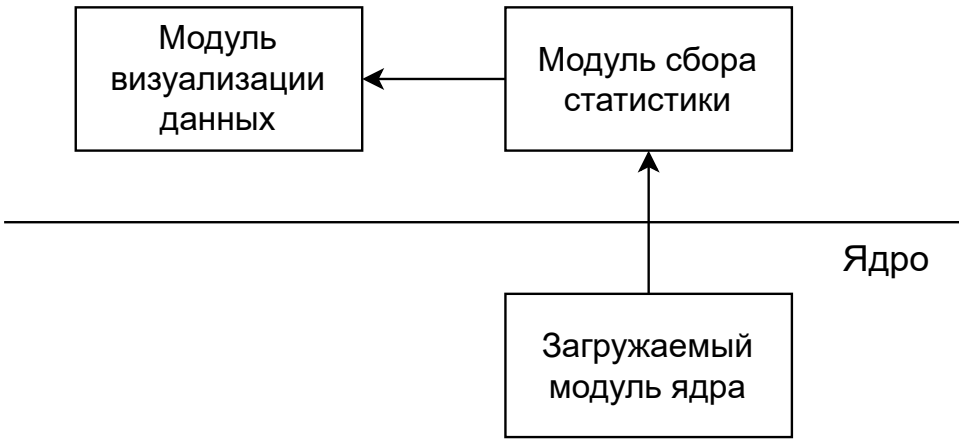

Рисунок 2.5 – Структура программного обеспечения

3 Технологический раздел

3.1 Выбор языка и среды программирования

Модуль ядра написан на языке программирования C [6]. Выбор языка основан на том, что исходный код системы, все модули ядра и драйверы операционной системы Linux написаны на языке C.

В качестве компилятора выбран gcc [7].

В качестве среды программирования выбрана среда Visual Studio Code [8].

3.2 Перехват функций

В листинге 3.1 представлена структура struct ftrace_hook, которая описывает каждую перехватываемую функцию.

Листинг 3.1 – Структура перехватываемой функции

```
struct ftrace_hook {
    const char *name;       // имя перехватываемой функции
    void *function;         // адрес функции-обертки, вызываемой
       вместо перехваченной функции
    void *original;         // указатель на мест, куда будет
       записан адрес перехватываемой функции

    unsigned long address;  // адрес перехватываемой функции
    struct ftrace_ops ops;
};
```

Пользователю необходимо заполнить только первые три поля: name, function, original. Остальные поля считаются деталью реализации. Описание всех перехватываемых функций можно собрать в массив и использовать макросы, чтобы повысить компактность кода. В листинге 3.2 представлен массив перехватываемых функций.

Листинг 3.2 – Массив перехватываемых функций

```
#define HOOK(_name, _function, _original)  \
    {                                      \
        .name = (_name),                   \
        .function = (_function),           \
        .original = (_original),           \
    }

/* массив перехватываемых функций */
static struct ftrace_hook demo_hooks[] = {
    HOOK("__x64_sys_clone",  hook_sys_clone,  &orig_sys_clone),
    HOOK("__x64_sys_execve", hook_sys_execve, &orig_sys_execve),
    HOOK("bdev_read_page", hook_bdev_read_page,
       &orig_bdev_read_page),
    HOOK("bdev_write_page", hook_bdev_write_page,
       &orig_bdev_write_page),
};
```

3.3 Инициализация ftrace

Для инициализации необходимо найти и сохранить адрес функции, которую будет перехватывать разрабатываемый модуль ядра. ftrace позволяет трассировать функции по имени, но при этом все равно надо знать адрес оригинальной функции, чтобы вызывать ее.

Найти адрес можно с помощью kallsyms – списка всех символов в ядре. В этот список входят все символы, не только экспортируемые для модулей. Получение адреса перехватываемой функции представлено в листинге 3.3.

Листинг 3.3 – Получение адреса перехватываемой функции

```
static int resolve_hook_address(struct ftrace_hook *hook)
{
    hook->address = kallsyms_lookup_name(hook->name);
    if (!hook->address) {
        pr_debug("unresolved symbol: %s\n", hook->name);
        return -ENOENT;
    }
    return 0;
}
```

В листинге 3.4 инициализируется структура ftrace_ops. В ней обязательным полем является лишь func, указывающая на коллбек, но также необходимо установить некоторые важные флаги.

Листинг 3.4 – Регистрация перехвата

```
int install_hook(struct ftrace_hook *hook)
{
    int err;

    err = resolve_hook_address(hook);
    if (err)
        return err;

    hook->ops.func = ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION_SAFE
                    | FTRACE_OPS_FL_IPMODIFY;

    /* включить ftrace для функции */
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err) {
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
        return err;
    }

    /* разрешить ftrace вызывать коллбек */
    err = register_ftrace_function(&hook->ops);
    if (err) {
        pr_debug("register_ftrace_function() failed: %d\n", err);
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }

    return 0;
}
```

В листинге 3.5 представлена функция дерегистрации перехвата.

Листинг 3.5 – Дерегистрация перехвата

```
void remove_hook(struct ftrace_hook *hook)
{
    int err;

    err = unregister_ftrace_function(&hook->ops);
    if (err) {
        pr_debug("unregister_ftrace_function() failed: %d\n",
           err);
    }

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err) {
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
    }
}
```

3.4 Функции-обертки для перехватываемых функций

Для перехвата необходимо определить функции-обертки. Очень важно в точности соблюдать сигнатуру функции: порядок и типы аргументов и возвращаемого значения. Оригинальные функции были взяты из исходных кодов ядра Linux [9].

На листингах 3.6-3.9 представлены реализации функций-оберток для системных вызовов и считывания информации с блочного устройства.

Листинг 3.6 – Функция-обертка для sys_clone

```
static asmlinkage long (*orig_sys_clone)(unsigned long
   clone_flags,
        unsigned long newsp, int __user *parent_tidptr,
        int __user *child_tidptr, unsigned long tls);

static asmlinkage long hook_sys_clone(unsigned long clone_flags,
        unsigned long newsp, int __user *parent_tidptr,
        int __user *child_tidptr, unsigned long tls)
{
    long ret = orig_sys_clone(clone_flags, newsp, parent_tidptr,
        child_tidptr, tls);
    pr_info("clone(): %ld\n", ret);

    return ret;
}
```

Листинг 3.7 – Функция-обертка для sys_execve

```
static asmlinkage long (*orig_sys_execve)(const char __user
   *filename,
        const char __user *const __user *argv,
        const char __user *const __user *envp);

static asmlinkage long hook_sys_execve(const char __user
   *filename,
        const char __user *const __user *argv,
        const char __user *const __user *envp)
{
    long ret;
    char *kernel_filename;

    kernel_filename = duplicate_filename(filename);
    kfree(kernel_filename);
    ret = orig_sys_execve(filename, argv, envp);
    pr_info("execve(): %ld\n", ret);

    return ret;
}
```

Листинг 3.8 – Функция-обертка для bdev_read_page

```
static asmlinkage int (*orig_bdev_read_page)(struct block_device
   *bdev, sector_t sector,
struct page *page);

static asmlinkage int hook_bdev_read_page(struct block_device
   *bdev, sector_t sector,
    struct page *page)
{
    int res;

    res = orig_bdev_read_page(bdev, sector, page);
    pr_info("read page: from /dev/sda");

    return res;
}
```

Листинг 3.9 – Функция-обертка для bdev_write_page

```
static asmlinkage int (*orig_bdev_write_page)(struct
   block_device *bdev, sector_t sector,
struct page *page, struct writeback_control **wc);

static asmlinkage int hook_bdev_write_page(struct block_device
   *bdev, sector_t sector,
struct page *page, struct writeback_control **wc)
{
    int res;

    res = orig_bdev_write_page(bdev, sector, page, wc);
    pr_info("write page: to /dev/sda");

    return res;
}
```

3.5 Сбор данных для визуализации

Собранные данные о вызове функций хранятся в лог-файле /var/log/syslog. Для того, чтобы передать собранные метрики в платформу Grafana для визуализации, необходимо настроить Promtail для считывания данных из лог-файла и их дальнейшую передачу в Loki для

хранения. В листинге 3.10 представлен фрагмент конфигурационного файла Promtail.

Листинг 3.10 – Сбор метрик /var/log/syslog

```
scrape_configs:
- job_name: system
  static_configs:
  - targets:
      - localhost
    labels:
      job: syslogs
      __path__: /var/log/syslog
```

Собранные данные Loki отправляет на порт 3100. На рисунке 3.1 представлена настройка Grafana для прослушивания Loki на порту 3100.

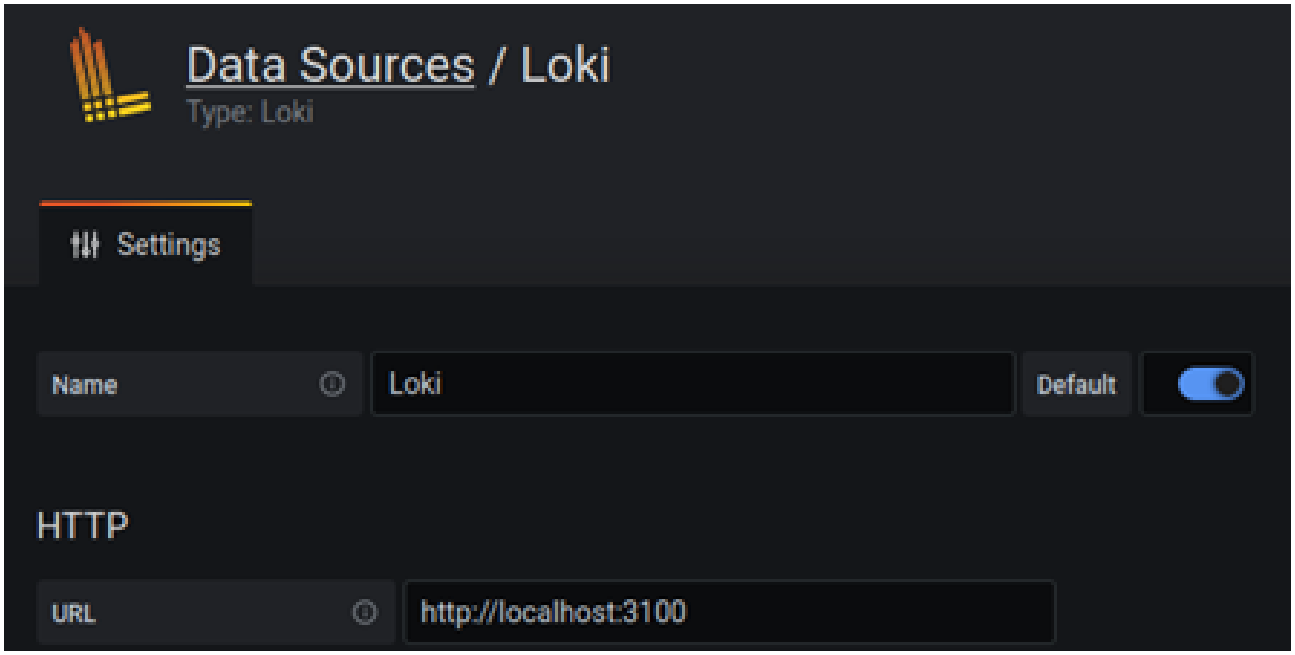

Рисунок 3.1 – Настройка Grafana

3.6 Выводы

В данном разделе был обоснован выбор языка программирования и средств программирования, представлены листинги реализованных функций.

4 Исследовательский раздел

4.1 Примеры работы

На рисунке 4.1 представлен пример собранных логов в /var/log/syslog.

Рисунок 4.1 – /var/log/syslog

На рисунке 4.2 представлен результат визуализации собранных данных о системных вызовах на платформе Grafana. На графике отображены данные за последний час, сбор метрик проводился каждые 5 минут. При наведении на график можно посмотреть, сколько раз была вызвана каждая функция в конкретный момент времени.

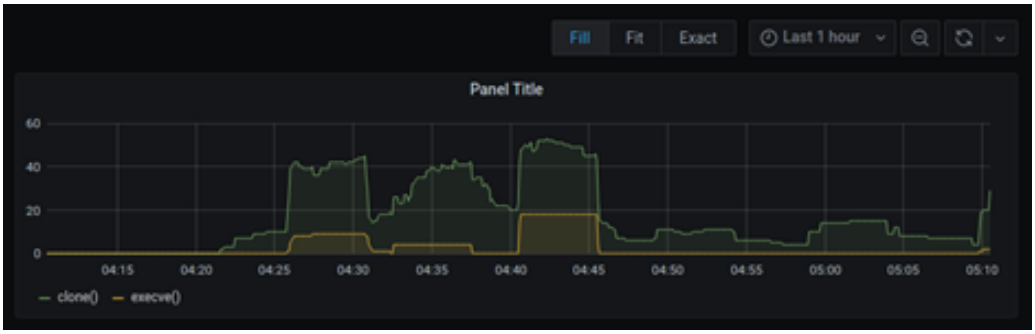

Рисунок 4.2 – Результат работы за 1 час

На рисунке 4.3 отображены данные об операциях с дисками за последние 15 минут, сбор метрик проводился каждые 5 минут.

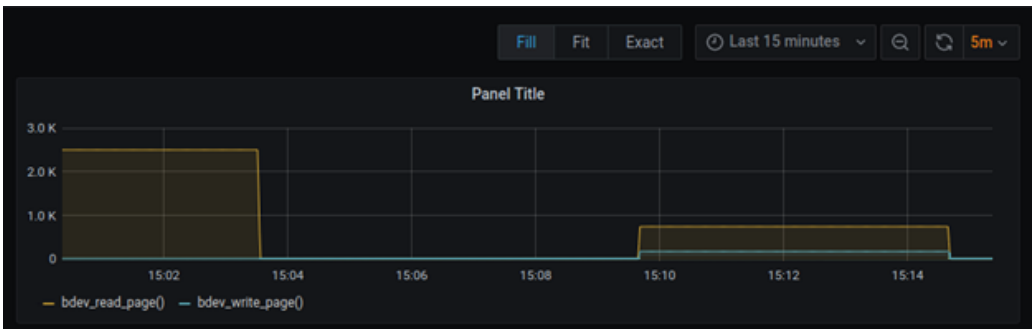

Рисунок 4.3 – Результат работы за 15 минут

4.2 Выводы

В данном разделе были приведены примеры работы реализованного программного обеспечения. Также были приведены статистики системных вызовов sys_clone, sys_execve за 1 час и операции с дисками bdev_read_page, bdev_write_page за 15 минут.

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы был разработан загружаемый модуль ядра, позволяющий перехватить указанные функции по их имени. В данной работе это системные вызовы sys_clone, sys_execve и операции с дисками bdev_read_page, bdev_write_page. Проанализированы существующие подходы для перехвата функций и средства для сбора и визуализации необходимых метрик. С помощью выбранных подходов и технологий был реализован загружаемый модуль ядра.

Цель была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Руководство по трассировке системных событий в Linux [Электронный ресурс]. — Режим доступа: https://www.kernel.org/doc/Documentation/trace/events.txt (дата обращения: 1.12.2022).
2. Перехват функций в ядре Linux с помощью ftrace [Электронный ресурс]. — Режим доступа: https://habr.com/ru/post/413241/ (дата обращения: 2.12.2022).
3. Трассировка ядра с ftrace [Электронный ресурс]. — Режим доступа: https://habr.com/ru/company/selectel/blog/280322/ (дата обращения: 4.12.2022).
4. Loki – сбор логов, используя подход Prometheus [Электронный ресурс]. — Режим доступа: https://habr.com/ru/company/otus/blog/487118/ (дата обращения: 11.12.2022).
5. Документация к Grafana [Электронный ресурс]. — Режим доступа: https://grafana.com/docs/grafana/latest/ (дата обращения: 10.12.2022).
6. C99 standard note [Электронный ресурс]. — Режим доступа: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf (дата обращения: 13.12.2022).
7. GCC, the GNU Compiler Collection [Электронный ресурс]. — Режим доступа: https://gcc.gnu.org/ (дата обращения: 13.12.2022).
8. Visual Studio Code - Code Editing. Redefined [Электронный ресурс]. — Режим доступа: https://code.visualstudio.com (дата обращения: 13.12.2022).
9. Исходные коды ядра Linux [Электронный ресурс]. — Режим доступа: https://github.com/torvalds/linux (дата обращения: 15.12.2022).

ПРИЛОЖЕНИЕ А

Листинг 4.1 – Исходный код загружаемого модуля ядра для мониторинга системных вызовов sys_clone, sys_execve и операций с дисками bdev_read_page, bdev_write_page

```
#define pr_fmt(fmt) "hook_functions: " fmt

#include <linux/init.h>
#include <linux/ftrace.h>
#include <linux/kallsyms.h>
#include <linux/kernel.h>
#include <linux/linkage.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/version.h>
#include <linux/syscalls.h>

MODULE_DESCRIPTION("System calls and block devices operations
   monitoring module");
MODULE_AUTHOR("Malyshev Ivan");
MODULE_LICENSE("GPL");

#define USE_FENTRY_OFFSET 0

/* struct ftrace_hook описывает перехватываемую функцию
name - имя перехватываемой функции
function - адрес функции-обертки, вызываемой вместо
   перехваченной функции
original - указатель на мест, куда будет записан адрес
   перехватываемой функции
address - адрес перехватываемой функции
*/
struct ftrace_hook {
    const char *name;
    void *function;
    void *original;

    unsigned long address;
    struct ftrace_ops ops;
};
```

```
/* адрес перехватываемой функции */
static int resolve_hook_address(struct ftrace_hook *hook)
{
    hook->address = kallsyms_lookup_name(hook->name);

    if (!hook->address) {
        pr_debug("unresolved symbol: %s\n", hook->name);
        return -ENOENT;
    }

#if USE_FENTRY_OFFSET
    *((unsigned long*) hook->original) = hook->address +
       MCOUNT_INSN_SIZE;
#else
    *((unsigned long*) hook->original) = hook->address;
#endif

    return 0;
}

/* выполнение перехвата функций */
static void notrace fh_ftrace_thunk(unsigned long ip, unsigned
   long parent_ip,
        struct ftrace_ops *ops, struct ftrace_regs *fregs)
{
    struct pt_regs *regs = ftrace_get_regs(fregs);

    /* получаем адрес struct ftrace_hook по адресу внедренной в
       неё struct ftrace_ops */
    struct ftrace_hook *hook = container_of(ops, struct
       ftrace_hook, ops);

#if USE_FENTRY_OFFSET
    regs->ip = (unsigned long) hook->function;
#else
    /* пропускаем вызовы функции из текущего модуля */
    if (!within_module(parent_ip, THIS_MODULE))
        regs->ip = (unsigned long) hook->function;
#endif
}
```

```
/* инициализация структуры ftrace_ops */
int install_hook(struct ftrace_hook *hook)
{
    int err;

    err = resolve_hook_address(hook);
    if (err)
        return err;

    hook->ops.func = fh_ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION
                    | FTRACE_OPS_FL_IPMODIFY;

    /* включить ftrace для функции */
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err) {
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
        return err;
    }

    /* разрешить ftrace вызывать коллбек */
    err = register_ftrace_function(&hook->ops);
    if (err) {
        pr_debug("register_ftrace_function() failed: %d\n", err);
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }

    return 0;
}

/* выключить перехват */
void remove_hook(struct ftrace_hook *hook)
{
    int err;

    err = unregister_ftrace_function(&hook->ops);
    if (err) {
```

```
        pr_debug("unregister_ftrace_function() failed: %d\n",
           err);
    }

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err) {
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
    }
}

int install_hooks(struct ftrace_hook *hooks, size_t count)
{
    int err;
    size_t i;

    for (i = 0; i < count; i++) {
        err = install_hook(&hooks[i]);
        if (err)
            goto error;
    }

    return 0;

error:
    while (i != 0) {
        remove_hook(&hooks[--i]);
    }

    return err;
}

void remove_hooks(struct ftrace_hook *hooks, size_t count)
{
    size_t i;

    for (i = 0; i < count; i++)
        remove_hook(&hooks[i]);
}

#ifndef CONFIG_X86_64
#error Currently only x86_64 architecture is supported
```

```
#endif

#if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >=
   KERNEL_VERSION(4,17,0))
#define PTREGS_SYSCALL_STUBS 1
#endif

#if !USE_FENTRY_OFFSET
#pragma GCC optimize("-fno-optimize-sibling-calls")
#endif

static asmlinkage long (*orig_sys_clone)(unsigned long
   clone_flags,
        unsigned long newsp, int __user *parent_tidptr,
        int __user *child_tidptr, unsigned long tls);

static asmlinkage long hook_sys_clone(unsigned long clone_flags,
        unsigned long newsp, int __user *parent_tidptr,
        int __user *child_tidptr, unsigned long tls)
{
    long ret;

    //pr_info("clone() before\n");

    ret = orig_sys_clone(clone_flags, newsp, parent_tidptr,
        child_tidptr, tls);

    pr_info("clone(): %ld\n", ret);

    return ret;
}

static char *duplicate_filename(const char __user *filename)
{
    char *kernel_filename;

    kernel_filename = kmalloc(4096, GFP_KERNEL);
    if (!kernel_filename)
        return NULL;

    if (strncpy_from_user(kernel_filename, filename, 4096) < 0) {
```

```
        kfree(kernel_filename);
        return NULL;
    }

    return kernel_filename;
}


/* Указатель на оригинальный обработчки системного вызова execve
Можно вызывать из обертки
*/
static asmlinkage long (*orig_sys_execve)(const char __user
   *filename,
        const char __user *const __user *argv,
        const char __user *const __user *envp);

/* Функция, которая будет вызываться вместо перехваченной
 Возвращаемое значение будет передано вызывающей функции
*/
static asmlinkage long hook_sys_execve(const char __user
   *filename,
        const char __user *const __user *argv,
        const char __user *const __user *envp)
{
    long ret;
    char *kernel_filename;

    kernel_filename = duplicate_filename(filename);

    //pr_info("execve() before: %s\n", kernel_filename);

    kfree(kernel_filename);

    ret = orig_sys_execve(filename, argv, envp);

    pr_info("execve(): %ld\n", ret);

    return ret;
}
```

```
static asmlinkage int (*orig_bdev_read_page)(struct block_device
   *bdev, sector_t sector,
struct page *page);

static asmlinkage int hook_bdev_read_page(struct block_device
   *bdev, sector_t sector,
    struct page *page)
{
    int res;

    res = orig_bdev_read_page(bdev, sector, page);
    pr_info("read page: from /dev/sda");

    return res;
}

static asmlinkage int (*orig_bdev_write_page)(struct
   block_device *bdev, sector_t sector,
struct page *page, struct writeback_control **wc);


static asmlinkage int hook_bdev_write_page(struct block_device
   *bdev, sector_t sector,
struct page *page, struct writeback_control **wc)
{
    int res;

    res = orig_bdev_write_page(bdev, sector, page, wc);
    pr_info("write page: to /dev/sda");

    return res;
}

#ifdef PTREGS_SYSCALL_STUBS
#define SYSCALL_NAME(name) ("__x64_" name)
#else
#define SYSCALL_NAME(name) (name)
#endif

#define HOOK(_name, _function, _original)    \
    {                    \
```

```
        .name = (_name),    \
        .function = (_function),    \
        .original = (_original),    \
    }

/* массив перехватываемых функций */
static struct ftrace_hook demo_hooks[] = {
    HOOK("__x64_sys_clone",  hook_sys_clone,  &orig_sys_clone),
    HOOK("__x64_sys_execve", hook_sys_execve, &orig_sys_execve),
    HOOK("bdev_read_page", hook_bdev_read_page,
       &orig_bdev_read_page),
    HOOK("bdev_write_page", hook_bdev_write_page,
       &orig_bdev_write_page),
};


static int __init hook_module_init(void)
{
    int err;

    err = install_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
    if (err)
        return err;

    pr_info("module loaded\n");

    return 0;
}

static void __exit hook_module_exit(void)
{
    remove_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
    pr_info("module unloaded\n");
}

module_init(hook_module_init);
module_exit(hook_module_exit);
```