



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

*«Разработка загружаемого модуля ядра для
мониторинга системных вызовов»*

Студент ИУ7-74Б
(Группа)

Д.В. Жабин
(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта

Н.Ю. Рязанова
(Подпись, дата) (И.О.Фамилия)

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 Аналитический раздел	5
1.1 Постановка задачи	5
1.2 Методы трассировки ядра	5
1.2.1 Модификация таблицы системных вызовов	6
1.2.2 Linux Security API	6
1.2.3 Kprobes	7
1.2.4 Kernel tracepoints	8
1.2.5 Фреймворк ftrace	8
1.2.6 Сравнение методов трассировки ядра	9
1.3 Средства визуализации лог-файлов	10
2 Конструкторский раздел	11
2.1 Последовательность преобразований	11
2.2 Перехват системного вызова	13
2.3 Алгоритм включения перехвата	15
2.4 Алгоритм отключения перехвата	16
2.5 Структура ПО	16
2.6 Настройка средств визуализации лог-файлов	17
3 Технологический раздел	19
3.1 Выбор языка и среды программирования	19
3.2 Функция включения перехвата	19
3.3 Хуки для перехватываемых функций	22
3.4 Сборка загружаемого модуля ядра	24
4 Исследовательский раздел	25
4.1 Примеры работы	25
ЗАКЛЮЧЕНИЕ	27
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	28
ПРИЛОЖЕНИЕ А. Исходный код загружаемого модуля ядра	29

ВВЕДЕНИЕ

В настоящее время операционная система Linux прочно занимает лидирующее положение в качестве серверной платформы, опережая многие коммерческие разработки. Тем не менее вопросы защиты информационных систем, построенных на базе этой ОС, не перестают быть актуальными. Существует большое количество технических средств, как программных, так и аппаратных, которые позволяют обеспечить безопасность системы. Это средства шифрования данных и сетевого трафика, разграничения прав доступа к информационным ресурсам, защиты электронной почты, веб-серверов, антивирусной защиты.

Один из способов защиты основан на перехвате системных вызовов операционной системы Linux. Этот способ позволяет взять под контроль работу любого приложения и тем самым предотвратить возможные деструктивные действия, которые оно может выполнить. Также перехват системных вызовов может быть использован для обеспечения возможности мониторинга активности в системе.

Данная работа посвящена исследованию способов перехвата системных вызовов с их последующим логированием и представлением собранных данных в графическом виде для наглядного анализа.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовой проект необходимо разработать загружаемый модуль ядра, позволяющий перехватить функции в ядре ОС Linux. Данная задача выполняется для перехвата системных вызовов `sys_clone()` и `sys_execve()`.

Для решения поставленной задачи необходимо:

- проанализировать существующие методы перехвата функций в ядре;
- описать алгоритм перехвата функций;
- реализовать загружаемый модуль ядра;
- обеспечить логирование информации о системных вызовах;
- представить собранные данные в графическом виде.

1.2 Методы трассировки ядра

Под трассировкой [1] понимается получение информации о том, что происходит внутри работающей системы. Для этого используются специальные программные инструменты, регистрирующие 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 логированием. 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.

1.2.1 Модификация таблицы системных вызовов

В ОС Linux все обработчики системных вызовов расположены в таблице `sys_call_table`. Подмена значений в этой таблице [2] приводит к смене поведения всей системы. Таким образом, сохранив исходный обработчик и подставив в таблицу собственный, можно перехватить любой системный вызов.

Преимуществами этого подхода являются:

- полный контроль над любыми системными вызовами;
- минимальные накладные расходы, нужно один раз изменить таблицу;
- не требуется каких-либо конфигурационных опций в ядре, а значит, поддерживается широкий спектр систем.

Недостатками являются:

- необходимость поиска таблицы системных вызовов, обхода защиты от модификации таблицы, безопасного выполнения замены;
- невозможность замены некоторых обработчиков из-за оптимизаций при обработке системных вызовов;
- перехватываются только системные вызовы.

1.2.2 Linux Security API

Linux Security API [3] – 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.

К недостаткам этого подхода можно отнести:

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

Таким образом, для использования Security API необходимо пересобирать ядро Linux.

1.2.3 Kprobes

Kprobes – это метод динамической трассировки, с помощью которого можно прервать выполнение кода ядра в любом месте, вызвать собственный обработчик и по завершении всех необходимых операций вернуться обратно. Обработчики получают доступ к регистрам и могут их изменять. Таким образом, можно получить как мониторинг, так и возможность влиять на дальнейший ход работы.

В ядре есть 3 вида kprobes:

- kprobes – «базовая» проба, которая позволяет прервать любое место ядра;
- jprobes – jump probe, вставляется только в начало функции и дает доступ к ее аргументам для обработчика, а также работает через setjmp/longjmp, то есть более легковесна;
- kretprobes — return probe, вставляется перед выходом из функции и дает доступ к ее результату.

Преимущества, которые дает использование kprobes для перехвата:

- хорошо задокументированный интерфейс, работа kprobes по возможности оптимизирована;
- kprobes реализуются с помощью точек останова (инструкции int3), что позволяет перехватить любое место в ядре, если оно известно.

К недостаткам kprobes относятся:

- для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах или где в стеке они лежат, и извлекать их оттуда;
- накладные расходы на расстановку точек останова;
- jprobes объявлены устаревшими и вырезаны из современных ядер;
- kretprobes необходимо хранить исходный адрес возврата, в случае переполнения буфера с адресами kretprobes будет пропускать срабатывания;
- kprobes основывается на прерываниях, поэтому для синхронизации все

обработчики выполняются с отключенным вытеснением, что накладывает ограничения на обработчики – в них нельзя выделять много памяти, заниматься вводом-выводом, спать в таймерах и семафорах.

1.2.4 Kernel tracepoints

Kernel tracepoints – это метод трассировки ядра, работающий через статическое инструментирование кода.

В качестве преимуществ можно выделить:

- минимальные накладные расходы – необходимо лишь вызвать функцию трассировки в нужном месте;
- возможность перехвата всех функций.

К недостаткам данного метода относится:

- отсутствие хорошо задокументированного API;
- не работает в модуле, если включен `CONFIG_MODULE_SIG`.

1.2.5 Фреймворк ftrace

Ftrace [4] – это фреймворк для трассировки 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.

Для популярных архитектур 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 `nop` – специальную ничего не делающую инструкцию. При включении трассировки в нужные функции вызовы ftrace добавляются обратно. Таким образом, если ftrace не ис-

пользуется, то его влияние на систему минимально.

В качестве преимуществ можно выделить:

- перехват любой функции;
- наличие подробной документации;
- перехват совместим с трассировкой, с ядра можно снимать полезные показатели производительности.

В качестве недостатка можно выделить требования к конфигурации ядра. Для успешного выполнения перехвата функций с помощью `ftrace` ядро должно предоставлять целый ряд возможностей:

- список символов `kallsyms` для поиска функций;
- фреймворк `ftrace` в целом для выполнения трассировки;
- опции `ftrace`, критически важные для перехвата.

Обычно ядра, используемые популярными дистрибутивами, все эти опции содержат, так как они не влияют на производительность и полезны при отладке.

1.2.6 Сравнение методов трассировки ядра

Сравнение рассмотренных методов приведено в таблице 1.

Таблица 1 – Сравнение методов перехвата функций

	Linux Security API	Модиф. таблицы системных вызовов	kprobes	Kernel tracepoints	ftrace
Наличие документ. API	-	-	+	-	+
Динамическая загрузка	-	+	+	+	+
Перехват всех функций	+	-	+	+	+
Любая конфигурация ядра	-	+	+	-	-

В ходе анализа методов перехвата функций для решения поставленной задачи был выбран фреймворк `ftrace`, так как он позволяет перехватить любую функцию, может быть динамически загружен в ядро, а также обладает хорошо задокументированным API.

1.3 Средства визуализации лог-файлов

Визуализация количества вызовов конкретных функций позволяет наглядно оценить состояние системы. В данной работе для этих целей был выбран Loki [6] – набор компонентов, которые предоставляют широкие возможности по обработке и анализу поступающих данных. В отличие от других подобных систем Loki основан на идее индексировать только метаданные логов – labels, а сами логи сжимать рядом.

Loki-стек состоит из трех компонентов: Promtail, Loki, Grafana. Promtail собирает логи, обрабатывает их и отправляет в Loki для хранения.

Grafana [7] – это платформа с открытым исходным кодом для визуализации, мониторинга и анализа данных. Grafana запрашивает данные из Loki и показывает их. Инструмент позволяет создавать панели, каждая из которых отображает определенные показатели в течение установленного периода времени. Искать по логам можно в специальном интерфейсе Grafana – Explorer, для запросов используется язык LogQL.

Выводы

В результате сравнительного анализа методов перехвата функций был выбран фреймворк `ftrace`, так как он полностью отвечает требованиям реализации.

2 Конструкторский раздел

2.1 Последовательность преобразований

Последовательность выполняемых действий в ПО в виде IDEF0 представлена на рисунках 1 и 2.

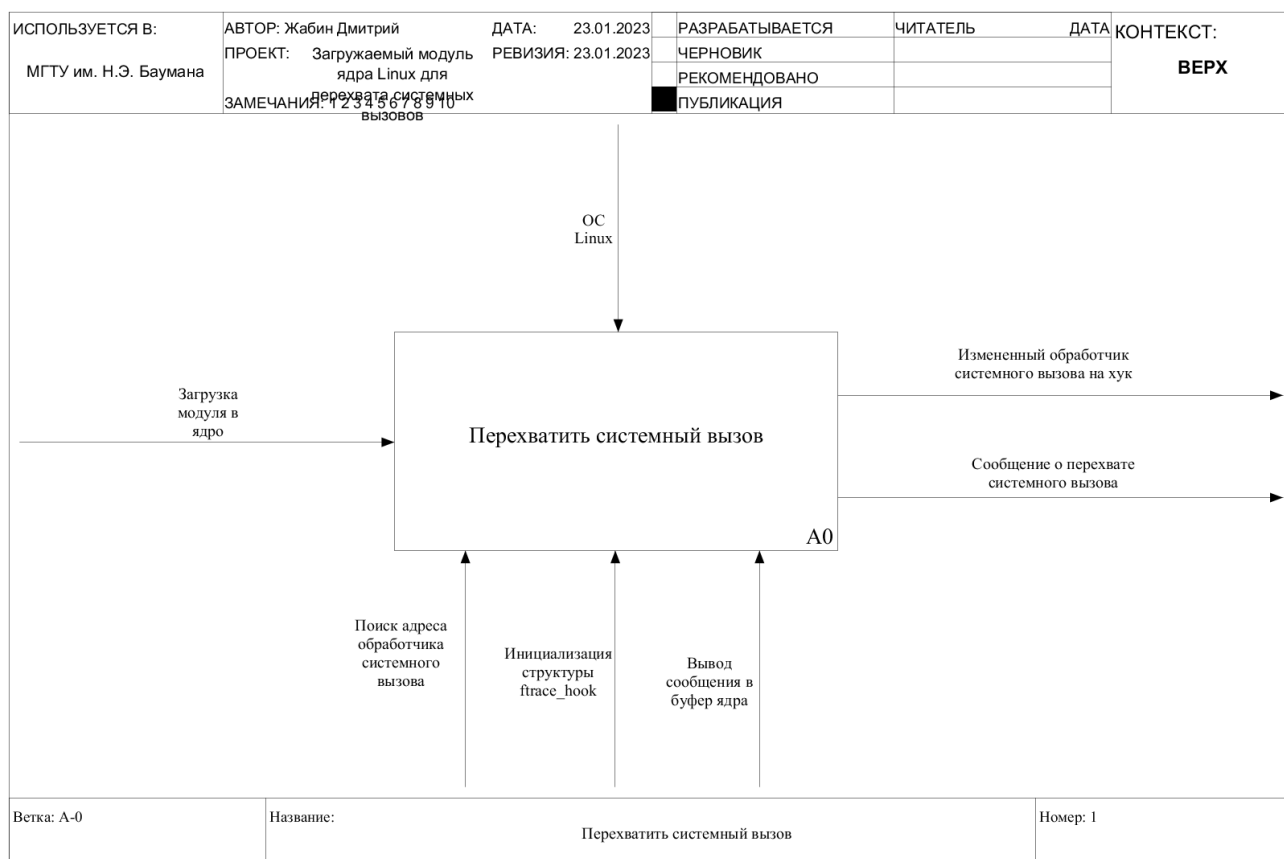


Рисунок 1 – Последовательность преобразований. Часть 1

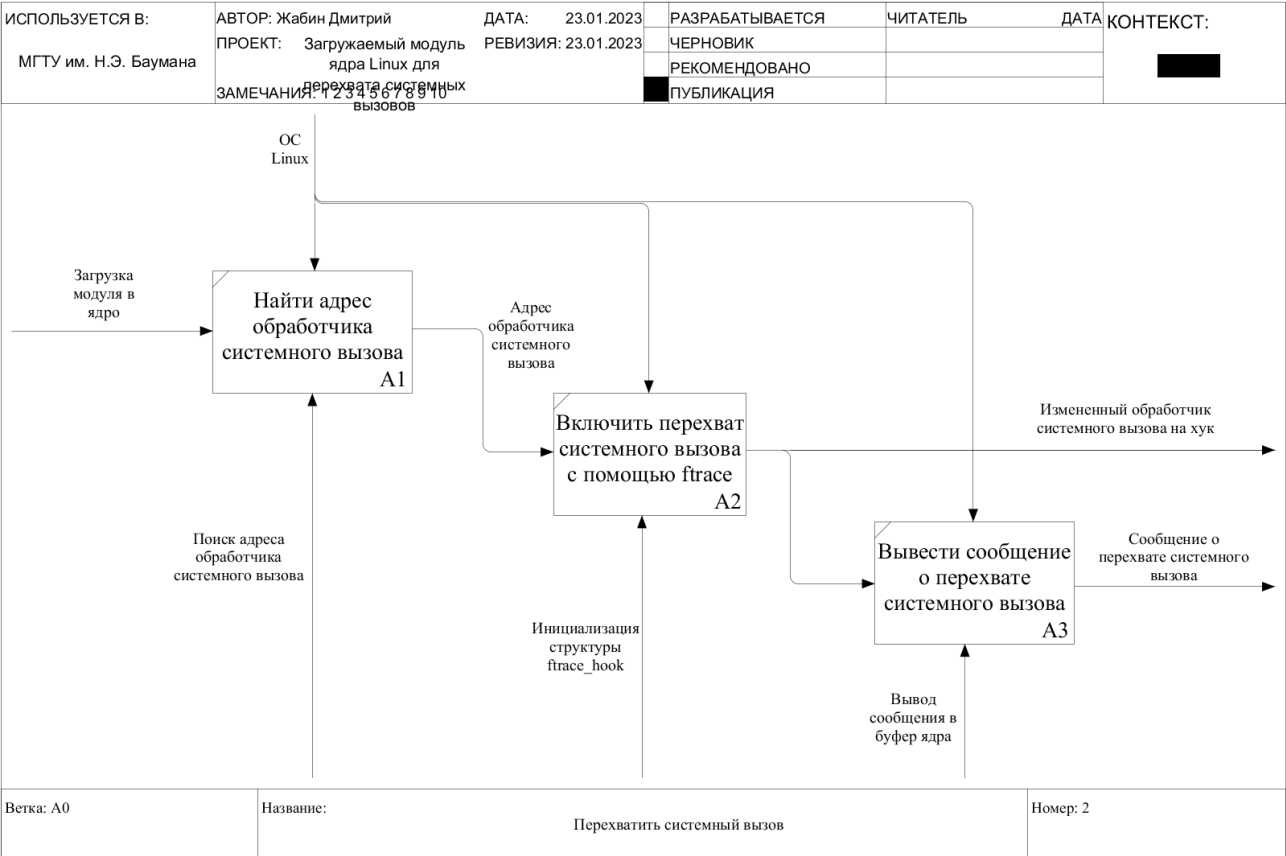


Рисунок 2 – Последовательность преобразований. Часть 2

2.2 Перехват системного вызова

Схема алгоритма перехвата системного вызова показана на рисунке 3.

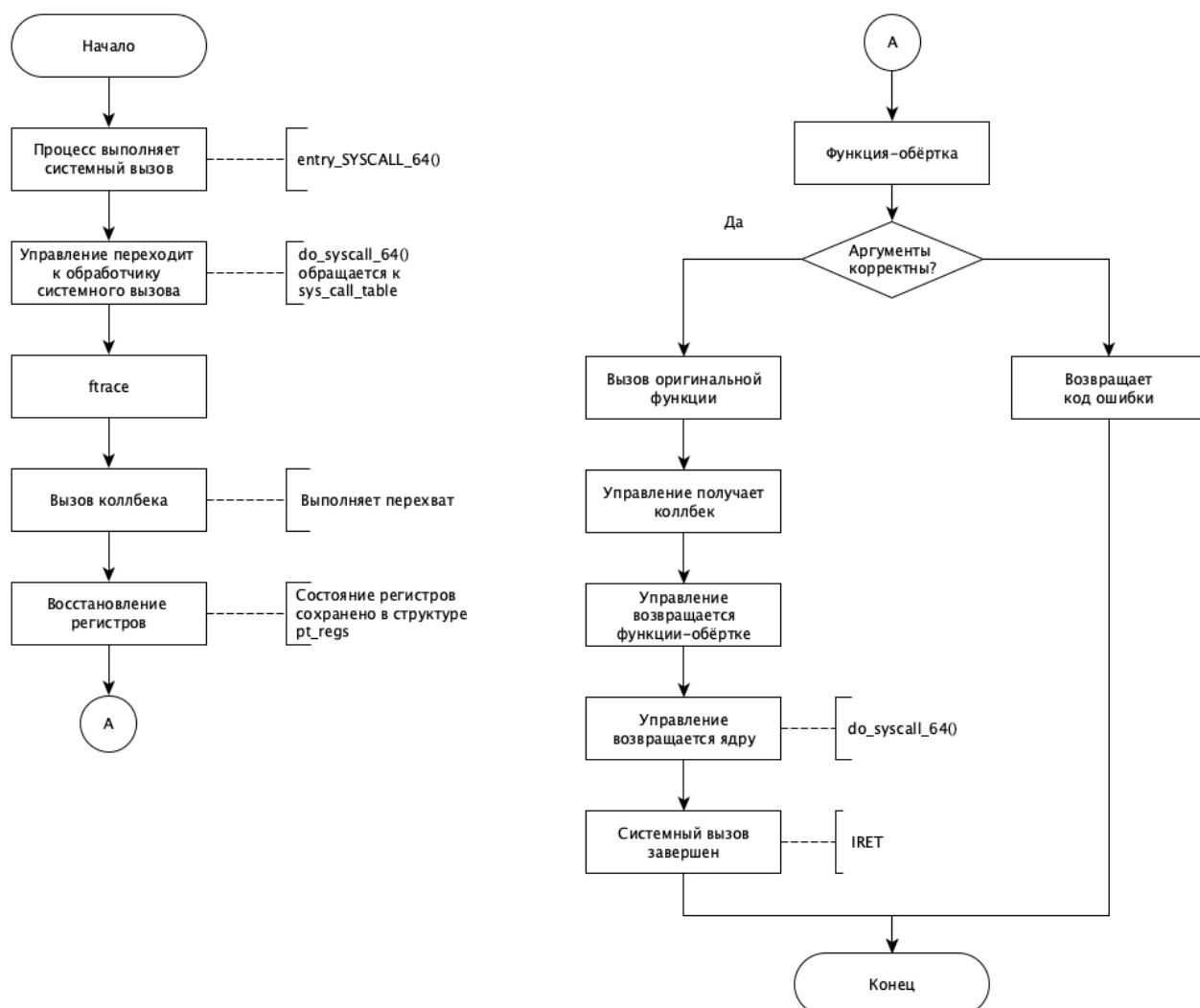


Рисунок 3 – Алгоритм перехвата системного вызова

Алгоритм перехвата системного вызова

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()`. Эта функция в свою очередь обращается к таблице обработчиков системных вызовов `sys_call_table` и вызы-

вает 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. Коллбек изменяет регистр `IP`, что в итоге приводит к передаче управления по новому адресу.

7. Управление получает хук. Вместо `sys_execve()` управление получает функция `hook_sys_execve()`. При этом остальное состояние процессора и памяти остается без изменений, поэтому хук получает все аргументы обработчика и при завершении вернет управление в функцию `do_syscall_64()`.

8. Функция `hook_sys_execve()` вызывает обработчик системного вызова. Она может проанализировать аргументы и контекст системного вызова и запретить или разрешить процессу его выполнение. В случае запрета функция возвращает код ошибки. Иначе же ей следует вызвать обработчик – `sys_execve()` вызывается повторно через указатель `orig_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 хуку.

12. Управление возвращается ядру. Функция `hook_sys_execve()` завершается и управление переходит в `do_syscall_64()`, которая считает, что системный

вызов был корректно завершен.

13. Управление возвращается в пользовательский процесс. Ядро выполняет инструкцию IRET, системный вызов завершен.

2.3 Алгоритм включения перехвата

Ftrace позволяет трассировать функции, но предварительно необходимо найти адрес перехватываемой функции, чтобы вызывать ее. Также нужно определить коллбек, который ftrace будет вызывать при трассировке функции. В коллбеке нужно заменить значение регистра IP на адрес нового обработчика, таким образом хук перехватит управление. Для включения перехвата необходимо сначала включить ftrace для перехватываемой функции, а затем разрешить ftrace вызывать коллбек.

Схема алгоритма включения перехвата системного вызова приведена на рисунке 4.

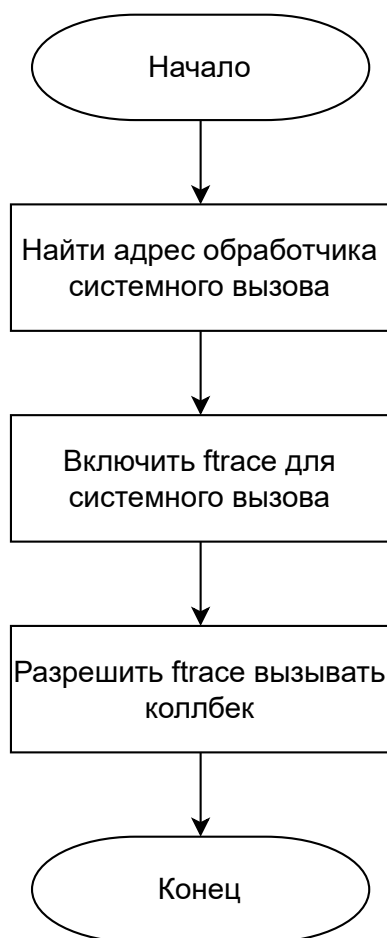


Рисунок 4 – Алгоритм включения перехвата

2.4 Алгоритм отключения перехвата

Для отключения перехвата функции необходимо выполнить обратные действия: запретить ftrace вызывать коллбек, а затем отключить ftrace для системного вызова.

Схема алгоритма отключения перехвата системного вызова приведена на рисунке 5.

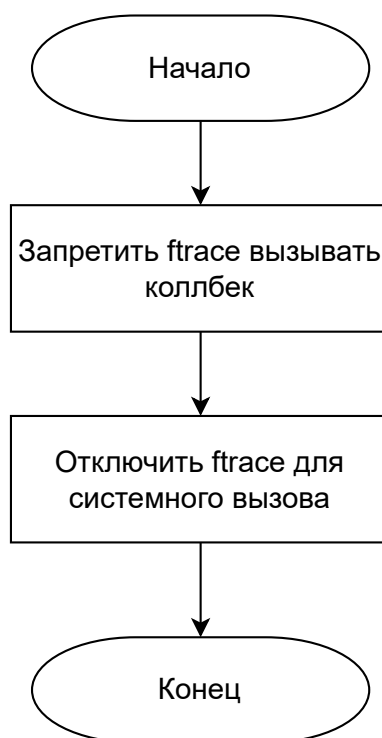


Рисунок 5 – Алгоритм отключения перехвата

2.5 Структура ПО

В состав программного обеспечения входит один загружаемый модуль ядра, который обеспечивает перехват системных вызовов, с последующим сбором информации и ее визуализацией.

Структура разрабатываемого программного обеспечения представлена на рисунке 6.

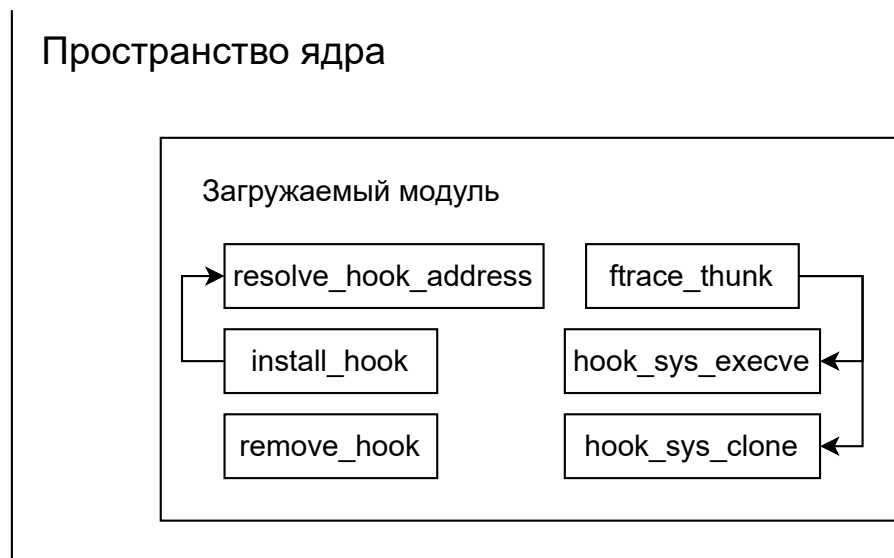


Рисунок 6 – Структура ПО

2.6 Настройка средств визуализации лог-файлов

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

Фрагмент конфигурационного файла Promtail представлен в листинге 1.

Листинг 1 – Конфигурационный файл Promtail

```
1 scrape_configs:
2 - job_name: system
3   static_configs:
4     - targets:
5       - localhost
6     labels:
7       job: syslogs
8     __path__: /var/log/syslog
```


Собранные данные Loki отправляет на порт 3100. Настройка Grafana для прослушивания Loki на этом порту представлена на рисунке 7.

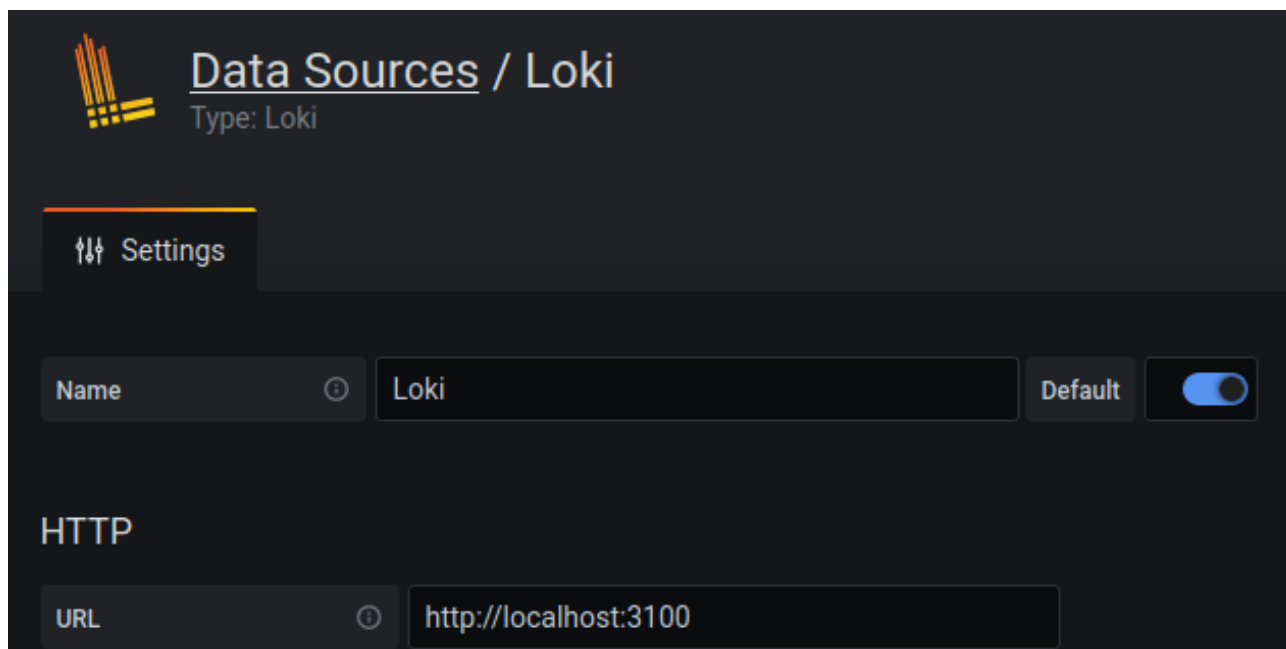


Рисунок 7 – Настройка Grafana

3 Технологический раздел

3.1 Выбор языка и среды программирования

Исходный код операционной системы Linux написан на языке C, поэтому для реализации загружаемого модуля выбран язык C.

Visual Studio Code [8] предлагает такой инструмент для разработчиков, как IntelliSense. Это множество функций редактирования кода, таких как, например, code completion, parameter info, quick info, member lists. Поэтому в качестве среды программирования была выбрана VS Code.

3.2 Функция включения перехвата

Для описания перехватываемых функций используется структура `struct ftrace_hook`, которая приведена в листинге 2.

Листинг 2 – Структура для описания перехватываемой функции

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

Поля приведенной структуры имеют следующее значение: `name` – имя перехватываемой функции, `function` – адрес хука, вызываемого вместо перехваченной функции, `original` – указатель на место, куда будет записан адрес перехватываемой функции, `address` – адрес перехватываемой функции.

Для обеспечения перехвата [9] необходимо заполнить только поля `name`, `function`, `original`. Для удобства описания можно использовать макрос, а все перехватываемые функции собрать в массив, что показано в листинге 3.

Листинг 3 – Массив перехватываемых функций

```
1 #define HOOK(_name, _function, _original) \  
2 { \  
3     .name = (_name), \  
4     .function = (_function), \  
5     .original = (_original), \  
6 } \  
7 static struct ftrace_hook my_hooks[] = { \  
8     HOOK("__x64_sys_clone", hook_sys_clone, &orig_sys_clone), \  
9     HOOK("__x64_sys_execve", hook_sys_execve, &orig_sys_execve), \  
10 };
```

Найти адрес перехватываемой функции можно с использованием kprobes, его получение показано в листинге 4.

Листинг 4 – Получение адреса перехватываемой функции

```
1 static unsigned long lookup_name(const char *name) \  
2 { \  
3     struct kprobe kp = { \  
4         .symbol_name = name \  
5     }; \  
6     unsigned long retval; \  
7     \  
8     if (register_kprobe(&kp) < 0) return 0; \  
9     retval = (unsigned long) kp.addr; \  
10    unregister_kprobe(&kp); \  
11    return retval; \  
12 } \  
13 \  
14 static int resolve_hook_address(struct ftrace_hook *hook) \  
15 { \  
16     hook->address = lookup_name(hook->name); \  
17     \  
18     if (!hook->address) { \  
19         pr_debug("unresolved symbol: %s\n", hook->name); \  
20         return -ENOENT; \  
21     } \  
22     *((unsigned long*) hook->original) = hook->address; \  
23     return 0; \  
24 }
```

Для включения перехвата необходимо проинициализировать структуру `ftrace_ops`. В ней обязательным полем является лишь `func`, указывающая на коллбек, но также необходимо установить некоторые важные флаги. Они предписывают `ftrace` сохранить и восстановить регистры процессора, содержимое которых может измениться в коллбеке.

Функция включения перехвата и коллбек представлены в листингах 5 и 6 соответственно.

Листинг 5 – Функция включения перехвата

```
1 int install_hook(struct ftrace_hook *hook)
2 {
3     int err = resolve_hook_address(hook);
4
5     if (err)
6         return err;
7
8     hook->ops.func = ftrace_thunk;
9     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
10                     | FTRACE_OPS_FL_RECURSION
11                     | FTRACE_OPS_FL_IPMODIFY;
12
13     err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
14     if (err)
15     {
16         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
17         return err;
18     }
19
20     err = register_ftrace_function(&hook->ops);
21     if (err)
22     {
23         pr_debug("register_ftrace_function() failed: %d\n", err);
24         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
25         return err;
26     }
27
28     return 0;
29 }
```

Листинг 6 – Коллбек, выполняющий перехват

```
1 static void notrace ftrace_thunk(unsigned long ip, unsigned long parent_ip,
2     struct ftrace_ops *ops, struct ftrace_regs *fregs)
3 {
4     struct pt_regs *regs = ftrace_get_regs(fregs);
5     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
6
7     if (!within_module(parent_ip, THIS_MODULE))
8         regs->ip = (unsigned long)hook->function;
9 }
```

Функция отключения перехвата представлена в листинге 7.

Листинг 7 – Функция отключения перехвата

```
1 void remove_hook(struct ftrace_hook *hook)
2 {
3     int err = unregister_ftrace_function(&hook->ops);
4     if (err)
5         pr_debug("unregister_ftrace_function() failed: %d\n", err);
6
7     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
8     if (err)
9         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
10 }
```

3.3 Хуки для перехватываемых функций

Порядок и типы аргументов и возвращаемого значения хуков должны соответствовать прототипу системного вызова. В хуках происходит вызов обработчика и его логирование.

Реализации хуков для системных вызовов `sys_clone()` и `sys_execve()` приведены в листингах 8 и 9 соответственно.

Листинг 8 – Реализация hook `sys_clone()`

```
1 static asmlinkage long (*orig_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     long ret;
10
11     ret = orig_sys_clone(clone_flags, newsp, parent_tidptr,
12         child_tidptr, tls);
13
14     pr_info("clone(): %ld\n", ret);
15
16     return ret;
17 }

```

Листинг 9 – Реализация hook sys execve()

```

1 static asmlinkage long (*orig_sys_execve)(const char __user *filename,
2     const char __user *const __user *argv,
3     const char __user *const __user *envp);
4
5 static asmlinkage long hook_sys_execve(const char __user *filename,
6     const char __user *const __user *argv,
7     const char __user *const __user *envp)
8 {
9     long ret;
10
11     ret = orig_sys_execve(filename, argv, envp);
12
13     pr_info("execve(): %ld\n", ret);
14
15     return ret;
16 }

```

3.4 Сборка загружаемого модуля ядра

Функции инициализации и выхода для загружаемого модуля приведены в листинге 10.

Листинг 10 – Функции инициализации и выхода

```
1 static int __init hook_module_init(void)
2 {
3     int err = install_hooks(my_hooks, ARRAY_SIZE(my_hooks));
4     if (err)
5         return err;
6
7     pr_info("my_hook module loaded\n");
8     return 0;
9 }
10
11 static void __exit hook_module_exit(void)
12 {
13     remove_hooks(my_hooks, ARRAY_SIZE(my_hooks));
14     pr_info("my_hook module unloaded\n");
15 }
16
17 module_init(hook_module_init);
18 module_exit(hook_module_exit);
```

Сборка загружаемого модуля ядра осуществляется с помощью make-файла, текст которого приведен в листинге 11.

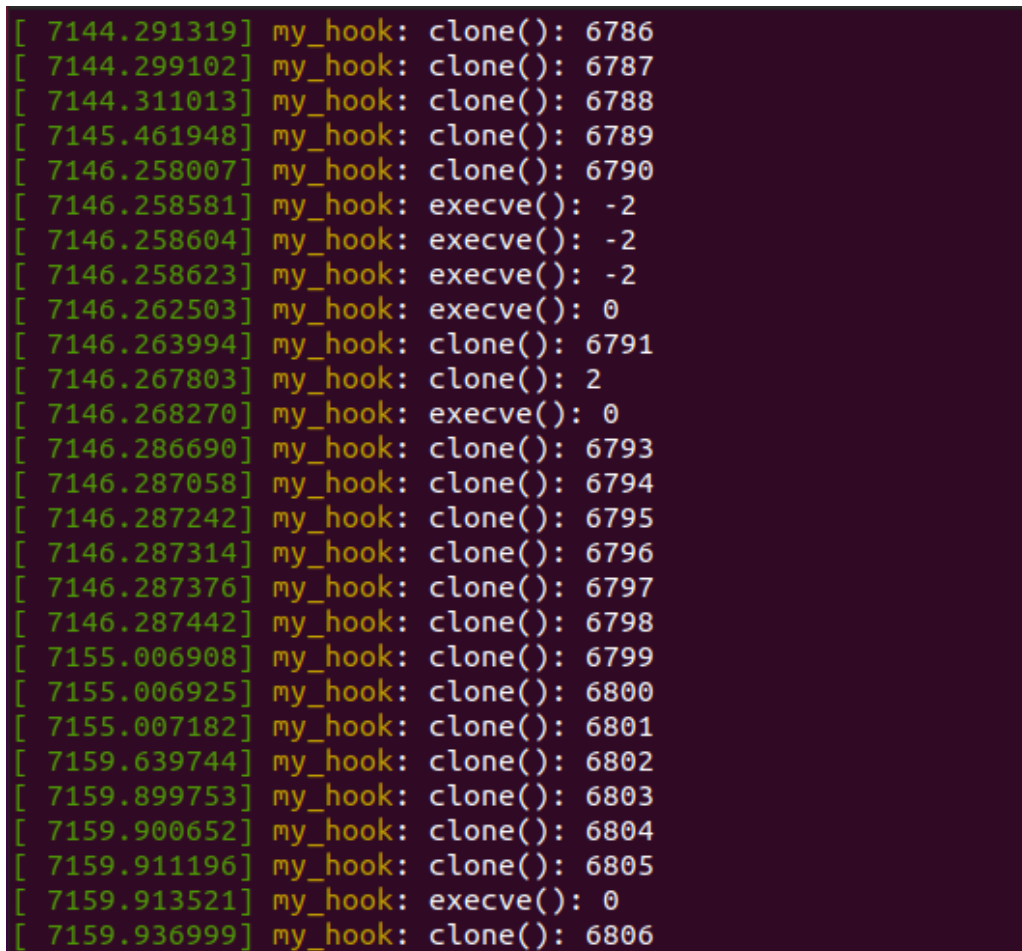
Листинг 11 – Make-файл для сборки модуля

```
1 KERNEL_PATH ?= /lib/modules/$(shell uname -r)/build
2
3 obj-m += my_hook.o
4
5 all:
6     make -C $(KERNEL_PATH) M=$(PWD) modules
7
8 clean:
9     make -C $(KERNEL_PATH) M=$(PWD) clean
```

4 Исследовательский раздел

4.1 Примеры работы

Фрагмент логов, собранных в /var/log/syslog, приведен на рисунке 8.



```
[ 7144.291319] my_hook: clone(): 6786
[ 7144.299102] my_hook: clone(): 6787
[ 7144.311013] my_hook: clone(): 6788
[ 7145.461948] my_hook: clone(): 6789
[ 7146.258007] my_hook: clone(): 6790
[ 7146.258581] my_hook: execve(): -2
[ 7146.258604] my_hook: execve(): -2
[ 7146.258623] my_hook: execve(): -2
[ 7146.262503] my_hook: execve(): 0
[ 7146.263994] my_hook: clone(): 6791
[ 7146.267803] my_hook: clone(): 2
[ 7146.268270] my_hook: execve(): 0
[ 7146.286690] my_hook: clone(): 6793
[ 7146.287058] my_hook: clone(): 6794
[ 7146.287242] my_hook: clone(): 6795
[ 7146.287314] my_hook: clone(): 6796
[ 7146.287376] my_hook: clone(): 6797
[ 7146.287442] my_hook: clone(): 6798
[ 7155.006908] my_hook: clone(): 6799
[ 7155.006925] my_hook: clone(): 6800
[ 7155.007182] my_hook: clone(): 6801
[ 7159.639744] my_hook: clone(): 6802
[ 7159.899753] my_hook: clone(): 6803
[ 7159.900652] my_hook: clone(): 6804
[ 7159.911196] my_hook: clone(): 6805
[ 7159.913521] my_hook: execve(): 0
[ 7159.936999] my_hook: clone(): 6806
```

Рисунок 8 – Фрагмент /var/log/syslog

Результат визуализации собранных данных о системных вызовах с использованием Grafana представлен на рисунках 9 и 10.

На первом графике отображены данные за последние 30 минут, сбор метрик проводился каждые 5 минут, а на втором – за последние 5 минут, сбор метрик проводился каждую минуту. При наведении на график можно увидеть, сколько раз была вызвана каждая функция за последние 10 минут.

Зеленый график отображает количество вызовов `execve()`, желтый – `clone()`.

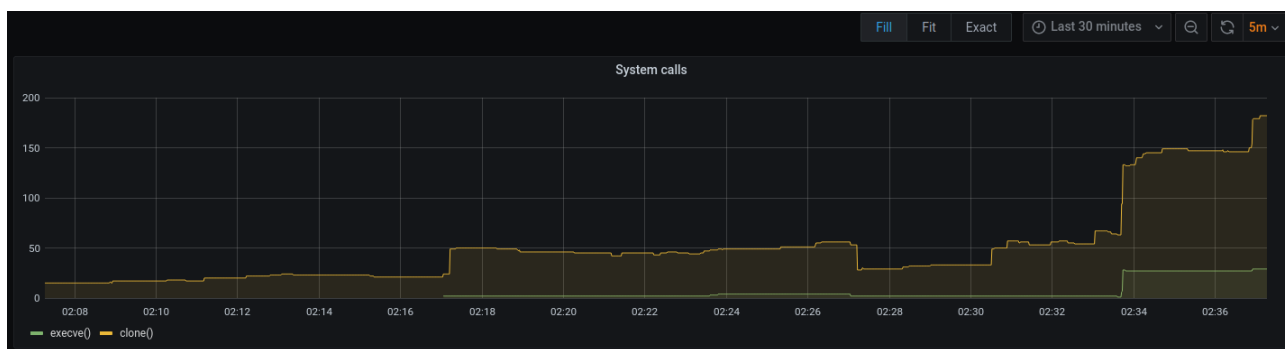


Рисунок 9 – Вызовы `clone()` и `execve()`. Пример 1

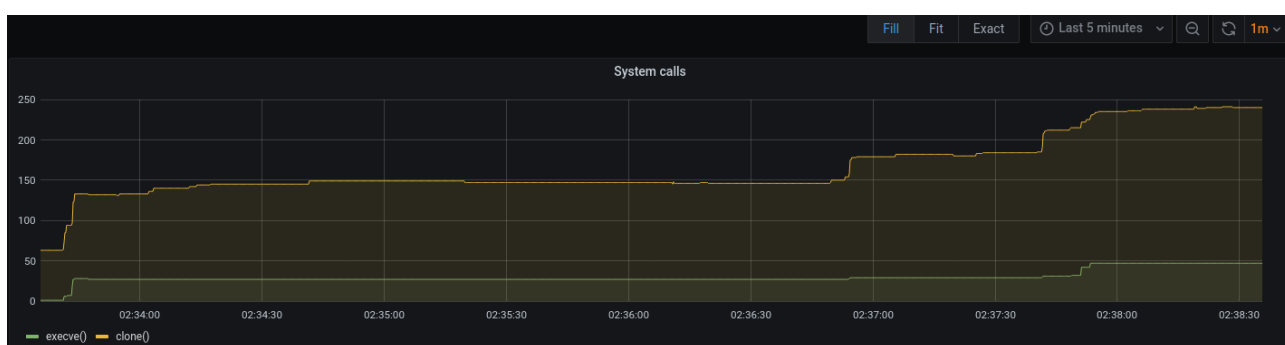


Рисунок 10 – Вызовы `clone()` и `execve()`. Пример 2

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта разработан загружаемый модуль ядра, позволяющий перехватить системные вызовы `sys_clone()` и `sys_execve()`.

В процессе работы:

- проанализированы существующие методы перехвата функций ядра;
- описан алгоритм перехвата функций;
- реализован загружаемый модуль ядра;
- обеспечено логирование информации о системных вызовах;
- собранные данные представлены в графическом виде.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Трассировка ядра с ftrace [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/company/selectel/blog/280322/> (дата обращения: 20.11.2022).
2. Linux Rootkits — Multiple ways to hook syscall [Электронный ресурс]. – Режим доступа: <https://foxtrot-sq.medium.com/linux-rootkits-multiple-ways-to-hook-syscall-s-7001cc02a1e6> (дата обращения: 12.12.2022).
3. Перехват функций в ядре Linux с помощью ftrace [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/413241/> (дата обращения: 20.12.2022).
4. Документация ftrace [Электронный ресурс]. – Режим доступа: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt> (дата обращения: 20.12.2022).
5. Цилюрик О.И. Модули ядра Linux. Внутренние механизмы ядра [Электронный ресурс]. – Режим доступа: <http://rus-linux.net/MyLDP/BOOKS/Modulyadra-Linux/kern-mod-index.html> (дата обращения: 15.12.2022).
6. Документация к Loki [Электронный ресурс]. – Режим доступа: <https://grafana.com/docs/loki/latest/> (дата обращения: 27.12.2022).
7. Документация к Grafana [Электронный ресурс]. – Режим доступа: <https://grafana.com/docs/grafana/latest/> (дата обращения: 27.12.2022).
8. Документация к Visual Studio Code [Электронный ресурс]. – Режим доступа: <https://code.visualstudio.com/docs> (дата обращения: 16.12.2022).
9. Hooking or Monitoring System calls in linux using ftrace [Электронный ресурс]. – Режим доступа: <https://nixhacker.com/hooking-syscalls-in-linux-using-ftrace/> (дата обращения: 20.12.2022).

ПРИЛОЖЕНИЕ А. Исходный код загружаемого модуля ядра

Листинг 12 – Исходный код загружаемого модуля ядра

```
1  #define pr_fmt(fmt) "my_hook: " fmt
2
3  #include <linux/init.h>
4  #include <linux/ftrace.h>
5  #include <linux/kallsyms.h>
6  #include <linux/kernel.h>
7  #include <linux/linkage.h>
8  #include <linux/module.h>
9  #include <linux/slab.h>
10 #include <linux/uaccess.h>
11 #include <linux/version.h>
12 #include <linux/syscalls.h>
13 #include <linux/kprobes.h>
14
15 MODULE_DESCRIPTION("Syscalls hook");
16 MODULE_AUTHOR("Dmitriy Zhabin");
17 MODULE_LICENSE("GPL");
18
19 /* Two ways of preventing recursive loops when hooking:
20  * - detect recursion using function return address (USE_FENTRY_OFFSET = 0)
21  * - avoid recursion by jumping over the ftrace call (USE_FENTRY_OFFSET = 1) */
22 #define USE_FENTRY_OFFSET 0
23
24 struct ftrace_hook {
25     const char *name;
26     void *function;
27     void *original;
28
29     unsigned long address;
30     struct ftrace_ops ops;
31 };
32
33 static unsigned long lookup_name(const char *name)
34 {
35     struct kprobe kp = {
36         .symbol_name = name
```

```

37     };
38     unsigned long retval;
39
40     if (register_kprobe(&kp) < 0) return 0;
41     retval = (unsigned long) kp.addr;
42     unregister_kprobe(&kp);
43     return retval;
44 }
45
46 static int resolve_hook_address(struct ftrace_hook *hook)
47 {
48     hook->address = lookup_name(hook->name);
49
50     if (!hook->address) {
51         pr_debug("unresolved symbol: %s\n", hook->name);
52         return -ENOENT;
53     }
54
55     #if USE_FENTRY_OFFSET
56         *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_SIZE;
57     #else
58         *((unsigned long*) hook->original) = hook->address;
59     #endif
60
61     return 0;
62 }
63
64 static void notrace ftrace_thunk(unsigned long ip, unsigned long parent_ip,
65     struct ftrace_ops *ops, struct ftrace_regs *fregs)
66 {
67     struct pt_regs *regs = ftrace_get_regs(fregs);
68     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
69
70     #if USE_FENTRY_OFFSET
71         regs->ip = (unsigned long) hook->function;
72     #else
73         if (!within_module(parent_ip, THIS_MODULE))
74             regs->ip = (unsigned long) hook->function;
75     #endif
76 }

```

```

77
78 int install_hook(struct ftrace_hook *hook)
79 {
80     int err = resolve_hook_address(hook);
81
82     if (err)
83         return err;
84
85     hook->ops.func = ftrace_thunk;
86     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
87                     | FTRACE_OPS_FL_RECURSION
88                     | FTRACE_OPS_FL_IPMODIFY;
89
90     /* enable ftrace for function */
91     err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
92     if (err)
93     {
94         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
95         return err;
96     }
97
98     /* let ftrace invoke callback */
99     err = register_ftrace_function(&hook->ops);
100    if (err)
101    {
102        pr_debug("register_ftrace_function() failed: %d\n", err);
103        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
104        return err;
105    }
106
107    return 0;
108 }
109
110 void remove_hook(struct ftrace_hook *hook)
111 {
112     int err = unregister_ftrace_function(&hook->ops);
113     if (err)
114         pr_debug("unregister_ftrace_function() failed: %d\n", err);
115
116     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);

```

```

117     if (err)
118         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
119 }
120
121 int install_hooks(struct ftrace_hook *hooks, size_t count)
122 {
123     int err;
124     size_t i;
125
126     for (i = 0; i < count; i++)
127     {
128         err = install_hook(&hooks[i]);
129         if (err)
130             goto error;
131     }
132     return 0;
133
134 error:
135     while (i > 0)
136         remove_hook(&hooks[--i]);
137
138     return err;
139 }
140
141 void remove_hooks(struct ftrace_hook *hooks, size_t count)
142 {
143     size_t i;
144     for (i = 0; i < count; i++)
145         remove_hook(&hooks[i]);
146 }
147
148 #if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >= KERNEL_VERSION(4,17,0))
149 #define PTREGS_SYSCALL_STUBS 1
150 #endif
151
152 #if !USE_FENTRY_OFFSET
153 #pragma GCC optimize("-fno-optimize-sibling-calls")
154 #endif
155
156 #ifndef PTREGS_SYSCALL_STUBS

```

```

157 static asmlinkage long (*orig_sys_clone)(struct pt_regs *regs);
158
159 static asmlinkage long hook_sys_clone(struct pt_regs *regs)
160 {
161     long ret;
162
163     //pr_info("clone() before\n");
164
165     ret = orig_sys_clone(regs);
166
167     pr_info("clone(): %ld\n", ret);
168
169     return ret;
170 }
171 #else
172 static asmlinkage long (*orig_sys_clone)(unsigned long clone_flags,
173     unsigned long newsp, int __user *parent_tidptr,
174     int __user *child_tidptr, unsigned long tls);
175
176 static asmlinkage long hook_sys_clone(unsigned long clone_flags,
177     unsigned long newsp, int __user *parent_tidptr,
178     int __user *child_tidptr, unsigned long tls)
179 {
180     long ret;
181     //pr_info("clone() before\n");
182
183     ret = orig_sys_clone(clone_flags, newsp, parent_tidptr,
184         child_tidptr, tls);
185
186     pr_info("clone(): %ld\n", ret);
187
188     return ret;
189 }
190 #endif
191
192 static char *duplicate_filename(const char __user *filename)
193 {
194     char *kernel_filename = kmalloc(4096, GFP_KERNEL);
195     if (!kernel_filename)
196         return NULL;

```



```

197
198     if (strncpy_from_user(kernel_filename, filename, 4096) < 0)
199     {
200         kfree(kernel_filename);
201         return NULL;
202     }
203     return kernel_filename;
204 }
205
206 #ifdef PTREGS_SYSCALL_STUBS
207 static asmlinkage long (*orig_sys_execve)(struct pt_regs *regs);
208
209 static asmlinkage long hook_sys_execve(struct pt_regs *regs)
210 {
211     long ret;
212     char *kernel_filename;
213
214     kernel_filename = duplicate_filename((void*) regs->di);
215     pr_info("execve(): %s\n", kernel_filename);
216
217     kfree(kernel_filename);
218
219     ret = orig_sys_execve(regs);
220     //pr_info("execve(): %ld\n", ret);
221
222     return ret;
223 }
224 #else
225 static asmlinkage long (*orig_sys_execve)(const char __user *filename,
226     const char __user *const __user *argv,
227     const char __user *const __user *envp);
228
229 static asmlinkage long hook_sys_execve(const char __user *filename,
230     const char __user *const __user *argv,
231     const char __user *const __user *envp)
232 {
233     long ret;
234     char *kernel_filename;
235
236     kernel_filename = duplicate_filename(filename);

```

```

237     pr_info("execve(): %s\n", kernel_filename);
238
239     kfree(kernel_filename);
240
241     ret = orig_sys_execve(filename, argv, envp);
242     //pr_info("execve(): %ld\n", ret);
243
244     return ret;
245 }
246 #endif
247
248 #define HOOK(_name, _function, _original) \
249 { \
250     .name = (_name), \
251     .function = (_function), \
252     .original = (_original), \
253 }
254 static struct ftrace_hook my_hooks[] = {
255     HOOK("__x64_sys_clone", hook_sys_clone, &orig_sys_clone),
256     HOOK("__x64_sys_execve", hook_sys_execve, &orig_sys_execve),
257 };
258
259 static int __init hook_module_init(void)
260 {
261     int err = install_hooks(my_hooks, ARRAY_SIZE(my_hooks));
262     if (err)
263         return err;
264
265     pr_info("my_hook module loaded\n");
266     return 0;
267 }
268
269 static void __exit hook_module_exit(void)
270 {
271     remove_hooks(my_hooks, ARRAY_SIZE(my_hooks));
272     pr_info("my_hook module unloaded\n");
273 }
274
275 module_init(hook_module_init);
276 module_exit(hook_module_exit);

```