Содержание

Введение

Иногда при работе с Linux-системами необходимо осуществлять перехват вызовов функций внутри ядра (например, открытие файлов или каталогов) для обеспечения возможности мониторинга активности в системе или превентивного блокирования деятельности подозрительных процессов. Перехват вызовов функций внутри ядра может осуществляться различными способами, такими как использование LSM, модификация таблицы системных вызовов, использование kprobes, использование сплайсинга и использование ftrace.

В настоящее время в официальное ядро Linux входят, например, такие security-модули, как AppArmor, SELinux, Smack и TOMOYO. Кроме того, с версии Linux 2.6.1 введена поддержка systrace. Systrace – это служебная программа для обеспечения компьютерной безопасности, которая ограничивает доступ приложений к системе, применяя политики доступа для системных вызовов. Systrace особенно полезен при запуске ненадежных приложений.

Данная работа посвящена исследованию способов перехвата системных вызовов. Целью проекта является разработка загружаемого модуля ядра, позволяющего перехватывать системные вызовы для отслеживания событий файловой системы.

1 Аналитический раздел

В соответствии с заданием на курсовой проект необходимо разработать загружаемый модуль ядра, перехватывающий системные вызовы, связанные с событиями в файловой системе Linux. Модуль должен осуществлять наблюдение за всеми файлами и директориями, записанными в конфигурационный файл модуля. Модуль должен отслеживать следущие события (в скобках указаны соответсвтующие системные вызовы):

- открытие файла (openat());
- создание файла (creat());
- запись данных в открытый файл (write());
- удаление записи из файла каталога (unlink(), unlinkat());
- создание каталога (mkdir(), mkdirat()).

Все произошедшие события модуль должен записывать в log-файл для того, чтобы впоследствии эту информацию можно было считать из пространства пользователя.

Для понимания алгоритма перехвата системных вызовов необходимо сначала рассмотреть, как происходит системный вызов.

1.1 Траектория системного вызова

Системный вызов - это фундаментальный интерфейс между приложением уровня пользователя и ядром Linux. Большую часть времени программы выполняются в пользовательском режиме и переключаются в режим ядра только тогда, когда им требуется служба операционной системы. Услуги операционной системы предоставляются через системные вызовы. Системные вызовы – это «ворота» в ядро, реализованные с помощью программных прерываний. Программные прерывания – это прерывания, создаваемые программой и обрабатываемые операционной системой в режиме ядра. Операционная система поддерживает «таблицу системных вызовов», в которой есть указатели на функции, реализующие системные вызовы внутри ядра.

В любой (в том числе и микроядерной) операционной системе системный вызов выполняется некоторой выделенной процессорной инструкцией, прерывающей последовательное выполнение команд и передающий управление коду режима супервизора. Это обычно некая команда программного прерывания, в зависимости от архитектуры процессора в разные времена это были команды с мнемониками вида: svc, emt, trap, int и им подобными. Если обратиться только к архитектуре Intel x86, то в ней для этого традиционно используется команда программного прерывания с различным вектором. Начиная с определенного момента (примерно с начала 2008 года или момента выхода Windows XP Service Pack 2) многие операционные системы (Windows, Linux) отказались от использования программного прерывания int, и перешли к реализации системного вызова и возврата из него через новые команды процессора sysenter (sysexit) [1], однако ничего принципиально нового не появилось.

Системные вызовы обычно вызываются не напрямую, а через функции оболочки в glibc (или, возможно, в какой-либо другой библиотеке). Все системные вызовы далее преобразуются в вызов ядра функцией syscall(), 1-м параметром которого будет идентификатор выполняемого системного вызова, например __NR_execve.

Системный вызов syscall(), попав в ядро, всегда попадает в таблицу sys_call_table, и далее переадресовывается по индексу (смещению) в этой таблице на величину 1-го параметра вызова syscall() - идентификатора требуемого системного вызова [1].

Рассмотрим теперь возможные способы перехвата системных вызовов в Linux.

1.2 Анализ подходов реализации перехвата системных вызовов

Сущетсвуют следующие способы реализации перехвата системных вызовов в Linux:

- Linux Security Modules (LSM);
- модификация таблицы системных вызовов;
- использование kprobes;
- сплайсинг;

- использование ftrace.

1.2.1 Linux Security Modules

Интерфейс LSM позволяет ядру Linux поддерживать различные модели компьютерной безопасности. LSM был создан для решения проблемы контроля доступа и является частью ядра начиная с Linux версии 2.6. LSM вставляет перехватчики (security-функций, которые в свою очередь вызывают обратные вызовы, установленные security-модулем) в каждую критическую точку ядерного кода, где системные вызовы уровня пользователя получают доступ к важным внутренним объектам ядра, таким как inode. Security-модуль может изучать контекст операции и принимать решение о её разрешении или запрете.

В частности, для файловых операций были определены три набора перехватчиков: перехватчики файловой системы, перехватчики inode и перехватчики файлов. LSM добавляет поле безопасности в каждую из связанных структур данных ядра: суперблок, индексный дескриптор и файл. Перехватчики файловой системы позволяют модулям безопасности управлять такими операциями, как, например, монтирование.

Название «модуль» несколько неверно, поскольку security-модули на самом деле не являются загружаемыми модулями ядра, а подключаются к ядру во время его сборки. Соответственно, Linux Security API имеет важное ограничение: security-модули не могут быть загружены динамически, являются частью ядра и требуют его пересборки.

Таким образом, несмотря на то, что LSM был разработан именно для мониторинга системных вызовов, для его использования необходимо поставлять собственную сборку ядра, а также интегрировать дополнительный модуль с SELinux или AppArmor, которые используются популярными дистрибутивами.

1.2.2 Модификация таблицы системных вызовов

Сохранив старое значения обработчика и подставив в таблицу системных собственный обработчик, мы можем перехватить любой системный вызов. Таблица указателей на функции ядра, которые реализуют системные вызовы, расположена в массиве sys_call_table. Такой подход известен программистам еще со времен MS-DOS.

Для перехвата системных этим способом используется механизм загружаемых модулей ядра. Для реализации модуля, перехватывающего системный вызов, необходимо определить алгоритм перехвата. Алгоритм следующий:

1. сохранить указатель на оригинальный (исходный) вызов для возможности его восстановления;
2. создать функцию, реализующую новый системный вызов;
3. в таблице системных вызовов sys_call_table произвести замену вызовов, т.е настроить соответствующий указатель на новый системный вызов;
4. по окончании работы (при выгрузке модуля) восстановить оригинальный системный вызов, используя ранее сохраненный указатель.

Данный подход имеет следующие преимущества:

- полный контроль над любыми системными вызовами;
- минимальные накладные расходы;
- минимальные требования к ядру.

Однако метод имеет и недостатки:

- техническая сложность реализации (необходимо найти таблицу системных вызовов, обойти защиту от модификации таблицы, выполнить замену атомарно и безопасно);
- невозможность перехвата некоторых обработчиков (некоторые обработчики реализованы на языке ассемблера, и их сложно или даже невозможно заменить на свои обработчики, написанные на C);
- перехватываются только системные вызовы (точки входа ограничиваются только системными вызовами, а все дополнительные проверки выполняются либо до непосредственного сиситемного вызова, либо после, поэтому необходимо дублировать проверки на адекватность аргументов).

1.2.3 Использование kprobes

Kprobes – это механизм отладки для ядра Linux, который также можно использовать для мониторинга событий внутри ядра. Этот механизм позволяет вставлять точки останова в работающее ядро. С помощью kprobes можно прервать выполнение ядерного кода в любом месте и вызвать свой обработчик. Этот интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции.

Для добавления своего собственного зонда (probe) в работающее ядро необходимо написать загружаемый модуль ядра, который реализует предварительный обработчик и пост-обработчик для зондирования.

Преимущества использования kprobes:

- зрелый API. Kprobes существуют и улучшаются с 2002 года;
- перехват любого места в ядре. Kprobes реализуются с помощью точек останова (инструкции int3), внедряемых в исполнимый код ядра. Это позволяет устанавливать kprobes в буквально любом месте любой функции, если оно известно.

Недостатки kprobes:

- техническая сложность. Kprobes — это только способ установить точку останова в любом места ядра. Для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах или где на стеке они лежат, и самостоятельно их оттуда извлекать;
- ограничения kretprobes. Kretprobes реализуются через подмену адреса возврата на 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;
- при обработке зондов (probes) приоритетное прерывание отключено. Это накладывает определённые ограничения на обработчики: в них нельзя выполнять операции ввода-вывода, спать в таймерах и семафорах;

- В текущей реализации kprobes существуют некоторые задержки в работе, причиной которых является kprobe_lock, который сериализует выполнение зондов на всех ЦП на машине SMP. Другая причина – это механизм kprobes, который использует несколько исключений для обработки одного зонда. Обработка исключений – дорогостоящая операция, которая вызывает задержки.

1.2.4 Сплайсинг

Сплайсинг – это метод перехвата функций путём изменения кода целевой функции. Инструкции в начале целевой функциии заменяются на безусловный переход, ведущий в нужный нам обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. С помощью двух переходов мы вшиваем (splice in) свой дополнительный код в функцию, поэтому такой подход называется сплайсингом. Методом сплайсинга реализована jump-оптимизация для kprobes.

Преимущества сплайсинга:

- минимальные требования к ядру. Сплайсинг не требует каких-либо особенных опций в ядре и работает в начале любой функции. Нужно только знать её адрес;

- минимальные накладные расходы. Необходимо всего лишь два безусловных перехода. Подобные переходы отлично предсказываются процессором и являются очень дешёвыми.

Однако сплайсинг имеет один серьезный недостаток: высокая техническая сложность реализации. Нельзя просто так взять и переписать машинный код. Для этого необходимо синхронизировать установку и снятие перехвата; обойти защиту на модификацию регионов памяти с кодом; дизассемблировать заменяемые инструкций, чтобы скопировать их целыми. В режиме ядра необходимо запретить прерывания для избежания переключения задач, так как при замене кода в начале функции перехватываемая функция может понадобиться другому потоку.

1.2.5 Использование ftrace

Ftrace — это внутренний трассировщик ядра, позволяющий разработчикам посмотреть, что происходит внутри ядра системы. Ftrace был включен в основную

линию ядра Linux в версии 2.6.27, выпущенной в 2008 году. С помощью ftrace можно отслеживать контекстные переключения, измерять время обработки прерываний, высчитывать время на активизацию заданий с высоким приоритетом и многое другое.

Ftrace полагается на механизм профилирования gcc для добавления машинных инструкций к скомпилированным версиям всех функций ядра, которые перенаправляют выполнение функций на плагины трассировщика ftrace, которые выполняют фактическую трассировку. В начало каждой функции добавляется вызов специальной трассировочной функции mcount() или __fentry__().

Ftrace поддерживает динамическое отслеживание вызовов функций ядра. Ядро знает расположение всех вызовов mcount() или __fentry__() и на ранних этапах загрузки заменяет их машинный код на nop — специальную инструкцию, которая предписывает ничего не делать. При включении трассирования в нужные функции вызовы ftrace добавляются обратно. Таким образом, если ftrace не используется, то его влияние на систему минимально.

Преимущества использования данного подхода:

- зрелый API и простой код. Использование готовых интерфейсов в ядре существенно упрощает код. Вся установка перехвата требует пары вызовов функций,заполнение двух полей в структуре;
- подход автоматически совместим с вытеснением, в отличие от kprobes;
- нет ограничений на функции. Подход с ftrace лишён недостатка kretprobes и из коробки поддерживает любое количество активаций перехватываемой функции;
- перехват любой функции по имени. Можно перехватить любую функцию (даже неэкспортируемую для модулей), зная лишь её имя;
- перехват совместим с трассировкой. Очевидно, что этот способ не конфликтует с ftrace, так что с ядра всё ещё можно снимать очень полезные показатели производительности.

Однако, ftrace, как и другие подходы, не лишен недостатков:

- требования к конфигурации ядра. Для поддержки 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);
- оборачиваются целиком вызовы функций. ftrace срабатывает исключительно при входе;

Несмотря на описанные недостатки следует учитывать, что обычно ядра, используемые популярными дистрибутивами, все необходимые ftrace опции в себе всё равно содержат, так как они не влияют на производительность и полезны при отладке. Иметь в виду эти требования стоит, если необходимо поддерживать какие-то особенные ядра. Оборачивание функций целиком в целом удобно, но для каких-либо специфических задач может и не подходить.

При использовании ftrace стоит учитывать, что использование kprobes или сплайсинга может помешать механизмам ftrace.

2 Конструкторский раздел

В данном разделе

2.1 Общая архитектура приложения

Для реализации подхода с ftrace необходимо реализовать загружаемый модуль ядра. В состав разрабатываемого программного обеспечения входит только загружаемый модуль ядра, следящий за необходимыми системными вызовами с последующим выводом информации в лог-файл.

2.2 Перехват системных вызовов

Перехватываемую функцию можно описать следующей структурой (листинг 3.9).

Листинг 1: ftrace_hook.

```
/**
* struct ftrace_hook — описывает перехватываемую функцию
*
* @name:        имя перехватываемой функции
*
* @function:    адрес функции—обёртки, которая будет вызываться вместо
*               перехваченной функции
*
* @original:    указатель на место, куда следует записать адрес
*               перехватываемой функции, заполняется при установке
*
* @address:     адрес перехватываемой функции, выясняется при установке
*
* @ops:         служебная информация ftrace, инициализируется нулями,
*               при установке перехвата будет доинициализирована
*/
struct ftrace_hook
{
    const char *name;
    void *function;
    void *original;

    unsigned long address;
```

```
24      struct ftrace_ops ops;
25  };
```

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 (листинг2).

Листинг 2: Описание перехватываемых функций.

```
1  #define HOOK(_name, _function, _original)       \
2  {                                               \
3      .name = (_name),                            \
4      .function = (_function),                    \
5      .original = (_original),                    \
6  }
7
8  static struct ftrace_hook fs_hooks[] = {
9      HOOK("sys_mkdir", fh_sys_mkdir, &real_sys_mkdir),
10     HOOK("sys_openat", fh_sys_openat, &real_sys_openat),
11     HOOK("sys_creat", fh_sys_creat, &real_sys_creat),
12     HOOK("sys_unlink", fh_sys_unlink, &real_sys_unlink),
13     HOOK("sys_write", fh_sys_write, &real_sys_write),
14     HOOK("sys_unlinkat", fh_sys_unlinkat, &real_sys_unlinkat),
15     HOOK("sys_mkdirat", fh_sys_mkdirat, &real_sys_mkdirat)
16 };
```

Обёртки над перехватываемыми функциями выглядят следующим образом (листинг 3).

Листинг 3: Обёртки над перехватываемыми функциями .

```
1  // настоящий обработчик системного вызова mkdir
2  static asmlinkage long (*real_sys_mkdir)(struct pt_regs *regs);
3
4  // наш обработчик системного вызова mkdir
5  static asmlinkage long fh_sys_mkdir(struct pt_regs *regs)
6  {
7      long ret;
8
9      ret = real_sys_mkdir(regs);
10
11     // ...
12     returnt ret;
```

```
13 }
```

Важно, чтобы сигнатуры функций в точности совпадали. Без этого, очевидно, аргументы будут переданы неправильно и всё пойдёт под откос.

2.3 Открытие, чтение из файлов и запись в файлы из пространства ядра

Открытие файлов в ядре производится с помощью функции filp_open(). Эта функция возвращает указатель на структуру struct file – структуру, описывающую открытый процессом файл (листинг 4).

Листинг 4: struct file для ядра версии 5.4.

```
1  struct file {
2      union {
3          struct llist_node   fu_llist;
4      struct rcu_head     fu_rcuhead;
5      } f_u;
6      struct path     f_path;
7      struct inode        *f_inode;   /* cached value */
8      const struct file_operations    *f_op;
9      //...
10     unsigned int        f_flags;
11     fmode_t         f_mode;
12     struct mutex        f_pos_lock;
13     loff_t          f_pos;
14     // ...
15 };
```

Структура struct file, в частности, содержит указатель на inode файла (struct inode *), которая содержит поле umode_t i_mode. Это поле можно передать макросу S_ISDIR(), определенному в «linux/stat.h». Макрос S_ISDIR() определит, является файл директорией или нет.

Чтение из файла выполняется с помощью функции kernel_read(), а запись – с помощью функции kernel_write(). Сигнатуры функций filp_open(), kernel_read() и kernel_write() приведены в листинге 5.

Листинг 5: filp_open(), kernel_read() и kernel_write() для ядра версии 5.4.

```
1 struct file *filp_open(const char *, int, umode_t);
2
3 ssize_t kernel_read(struct file *, void *, size_t, loff_t *);
4
5 ssize_t kernel_write(struct file *, const void *, size_t, loff_t *);
```

2.4 Поиск имен файлов, открытых процессом, по номеру файлового дескриптора

В разрабатываемом модуле ядра необходимо осуществлять мониторинг за файлами и директориями. В конфигурационном файле модуля записаны имена файлов и директорий, однако в системные вызовы могут передаваться не только абсолютные пути до файлов, но и относительные, а, например, в системный вызов write передается только номер дескриптора открытого файла. То есть необходимо осуществить поиск имени файла, зная номер его файлового дескриптора и PID открывшего его процесса. Помочь с этим может виртуальная файловая система /proc.

Файловая система /proc содержит каталоги (для структурирования информации) и виртуальные файлы. Виртуальный файл, как уже было сказано, может предоставлять пользователю информацию, полученную из ядра и, кроме того, служить средством передачи в ядро пользовательской информации. /proc, в частности, содержит каталоги, содержащие информацию о выполняющихся в системе процессах. Каждому запущенному процессу соответствует подкаталог с именем, соответствующим идентификатору этого процесса (его pid). (/proc/<PID>). Каждый из этих каталогов содержит, кроме всего прочего, подкаталог fd, содержащий одну запись на каждый файл, который в данный момент открыт процессом. Имя каждой такой записи соответствует номеру файлового дескриптора и является символьной ссылкой на реальный файл (/proc/<PID>/fd/N).

Чтобы получить информацию об имени файла, используя /proc/<PID>/fd/N, необходимо выполнить следующие шаги.

1. Открыть файл /proc/<PID>/fd/N для чтения с помощью функции filp_open().
2. Получить поле f_path, являющейся структурой struct path из указателя на структуру struct file, который вернула функция filp_open().

3. Вызвать функцию path_get и передать ей указатель на полученную на предыдущем шаге структуру struct path. Ее сигнатура приведена в листинге 6.

4. Вызвать функцию d_path, ее сигнатура приведена в листинге 6. Она вернет абсолютный путь до необходимого файла.

Листинг 6: d_path() и path_get() для ядра версии 5.4.

```
1 char *d_path(const struct path *, char *, int);
2
3 void path_get(const struct path *);
```

2.5 Адреса функций и процедур ядра

Виртуальная файловая система /proc содержит псевдофайл /proc/kallsyms – файл, внутри которого находится символьная таблица адресов функций и процедур, используемых ядром операционной системы Linux. В этой таблице перечислены имена переменных и функций и их адреса в памяти компьютера. В отличие от System.map, /proc/kallsyms содержит таблицу символов не только статически, но и динамически загружаемых модулей.

Псевдофайл /proc/kallsyms (таблица) - это ни что иное, как отображение во внешнее окружение некоторой внутренней структуры ядра. Ядро предоставляет пользователям (т.е. модулям) вызов kallsyms_lookup_name(), позволяющий выполнять поиск в этой внутренней структуре.

Однако kallsyms_lookup_name() присутствует в версиях ядра, начиная 2.6.32. Кроме того, начиная с версии 5.6, этот символ не экспортируется, поэтому неоюходим другой механизм поиска по /proc/kallsyms. В данной работе используется ядро версии 5.4, где этот символ является экспортируемым.

2.6 Алгоритм работы перехвата с импользованием ftrace

Рассмотрим следующий пример. При наборе в терминале команды ls, командный интерпретатор (Bash) для запуска нового процесса использует функции функций fork() + execve(). Внутри эти функции реализуются через системные вызовы clone() и execve() соответственно. Допустим, мы перехватываем системный вызов execve(), чтобы контролировать запуск новых процессов.

В графическом виде перехват функции-обработчика выглядит так, как показано на рисунке 1.

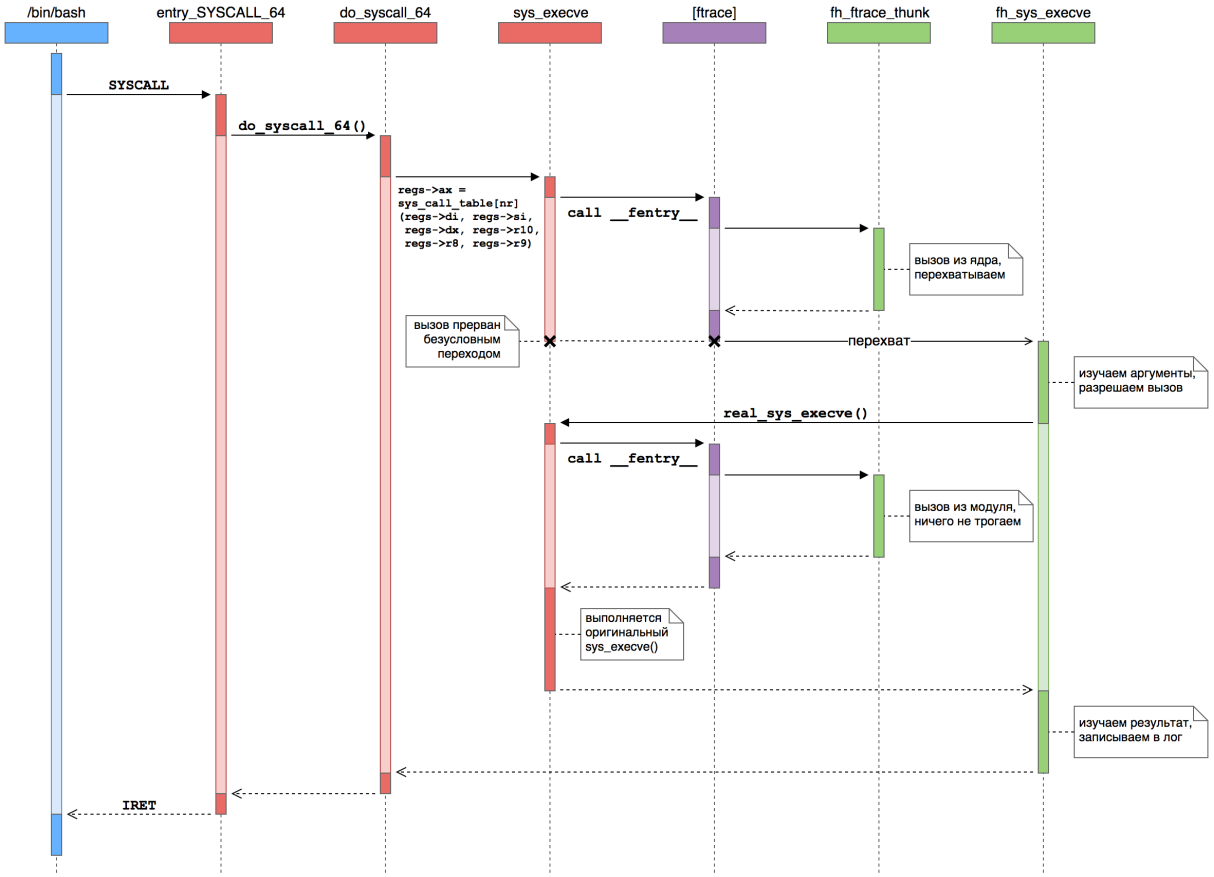

Рис. 1: Схема перехвата системного вызова execve() [7].

Пользовательский процесс (отмечен голубым) выполняет системный вызов в ядро (красный), где ftrace (фиолетовый) выполняет безусловный переход в функцию из нашего модуля (зеленый). Можно выделить следующие шаги при выполнении перехвата системного вызова на 64-битных ядрах.

1. Пользовательский процесс выполняет SYSCALL. С помощью этой инструкции выполняется переход в режим ядра и управление передаётся 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. Вызывается функция do_sysca
которая в свою очередь обращается к таблице обработчиков системных вызовов sys_call_table и вызывает оттуда конкретный обработчик по номеру системного вызова.

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

4. Ftrace вызывает наш коллбек. В процессе работы ftrace вызывает все зарегистрированные трассировочные коллбеки, включая и наш.

5. Коллбек из нашего модуля принимает решение о выполнении перехвата (если он обнаружит рекурсию, то перехват не будет выполнен, иначе произойдет зависание системы).

6. Ftrace восстанавливает регистры. 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 — что в итоге приводит к передаче управления по новому адресу.

7. Управление получает функция-обёртка из нашего модуля. При этом всё остальное состояние процессора и памяти остаётся без изменений, поэтому наша функция получает все аргументы оригинального обработчика и при завершении вернёт управление в функцию do_syscall_64().

8. Обёртка вызывает оригинальную функцию. Наша функция-обертка может проанализировать аргументы и контекст системного вызова (кто что запускает) и, например, запретить или разрешить процессу его выполнение. В случае запрета функция просто возвращает код ошибки. Иначе через указатель real_sys_execve, который был сохранён при настройке перехвата, вызывается оригинальный обработчик.

9. Управление получает коллбек. Как и при первом вызове, управление опять проходит через ftrace и передаётся в наш коллбек. Однако в этот раз он ничего не делает, потому что в этот раз вызов просиходит из нашей же функции.

10. Управление возвращается ядру.

11. Управление возвращается в пользовательский процесс.

2.7 Схемы алгоритмов

На рисунке 2 приведена схема алгоритма загрузки модуля. На рисунке 3 приведена схема алгоритма выгрузки модуля.

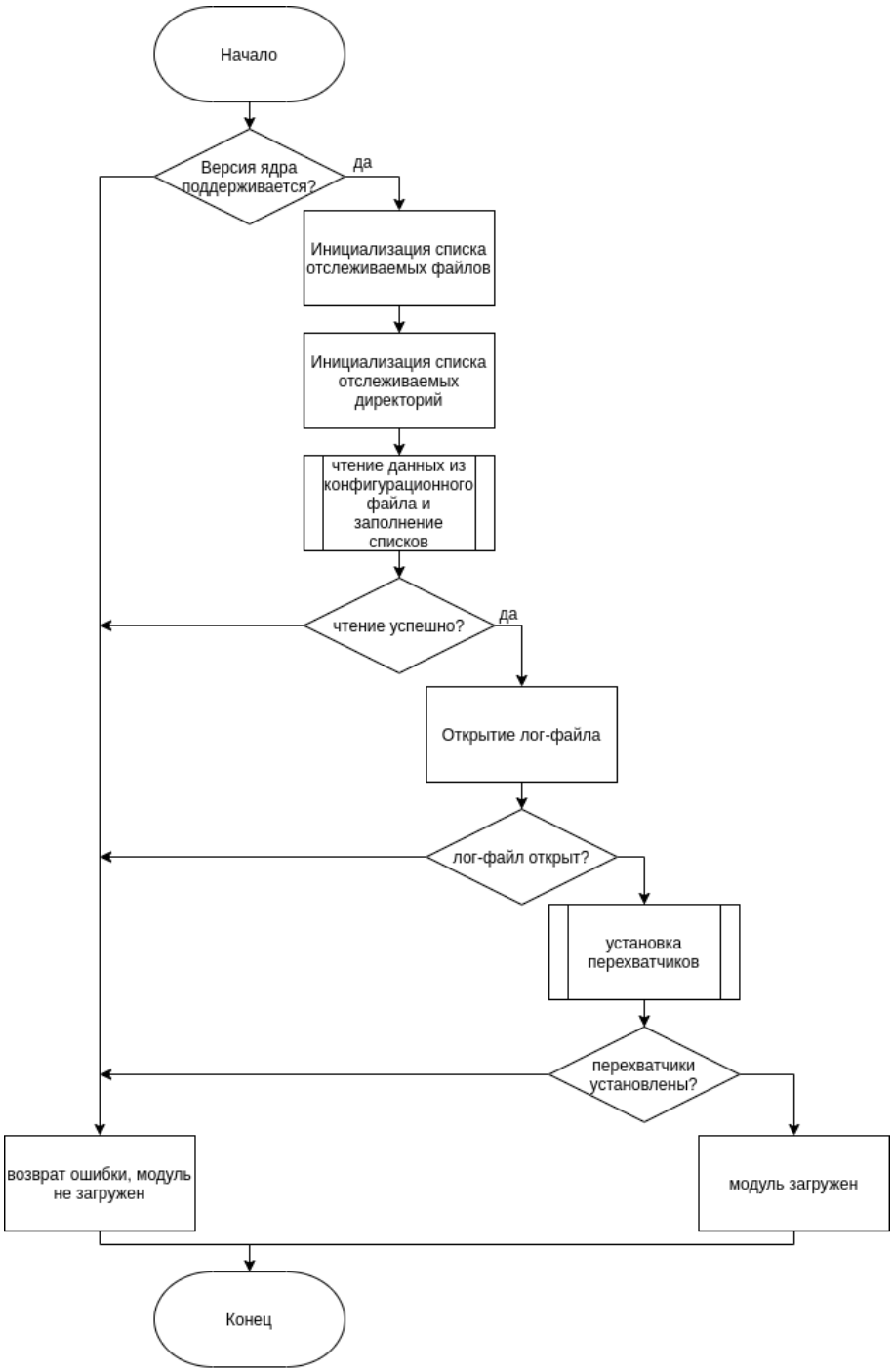

Рис. 2: Схема алгоритма загрузки модуля.

Рис. 3: Схема алгоритма выгрузки модуля.

На рисунках 4 приведены схемы наших обработчиков системных вызовов openat(), creat(), write(), unlink(), unlinkat(), mkdir(), mkdirat().

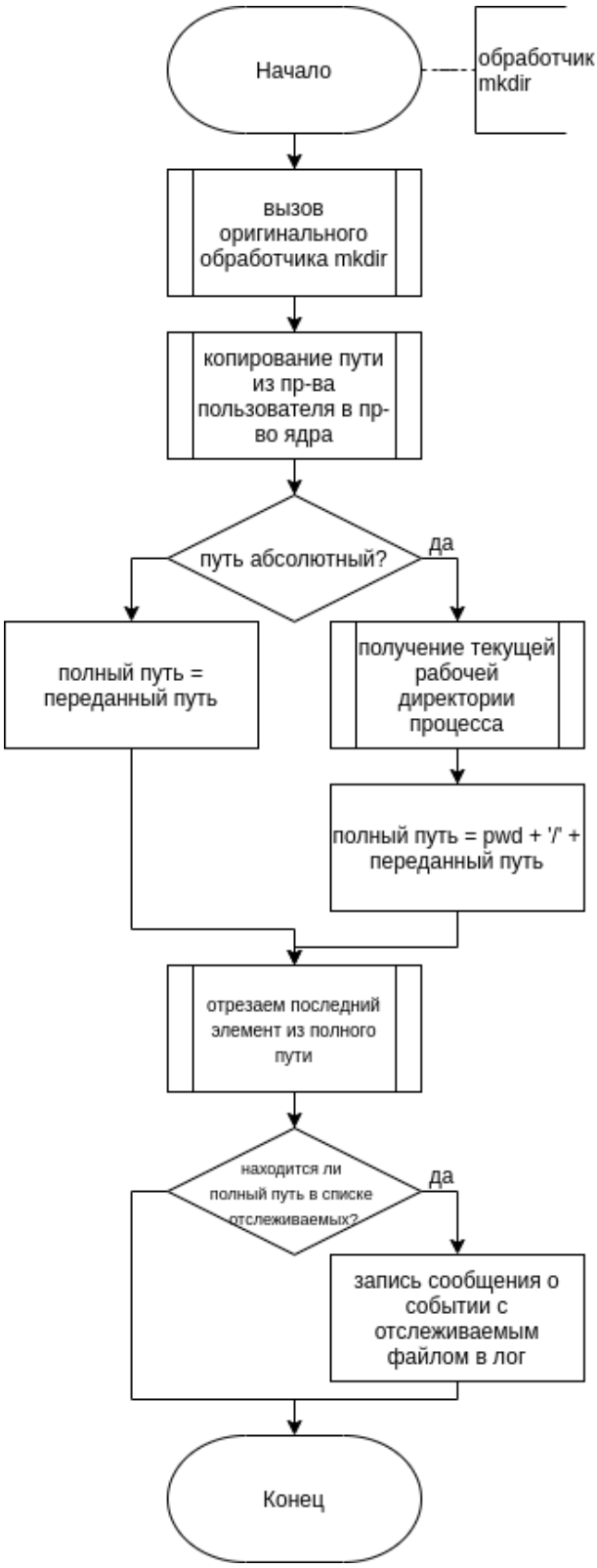

Рис. 4: Наш обработчик mkdir().

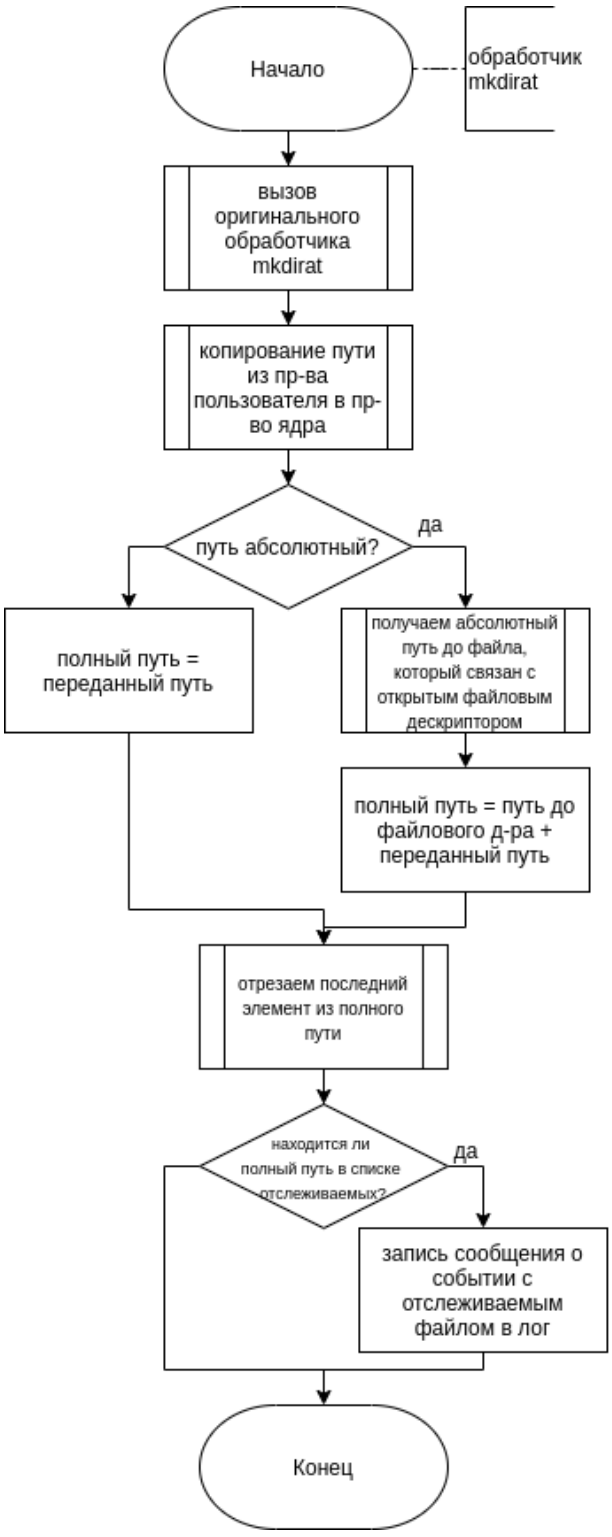

Рис. 5: Наш обработчик mkdirat().

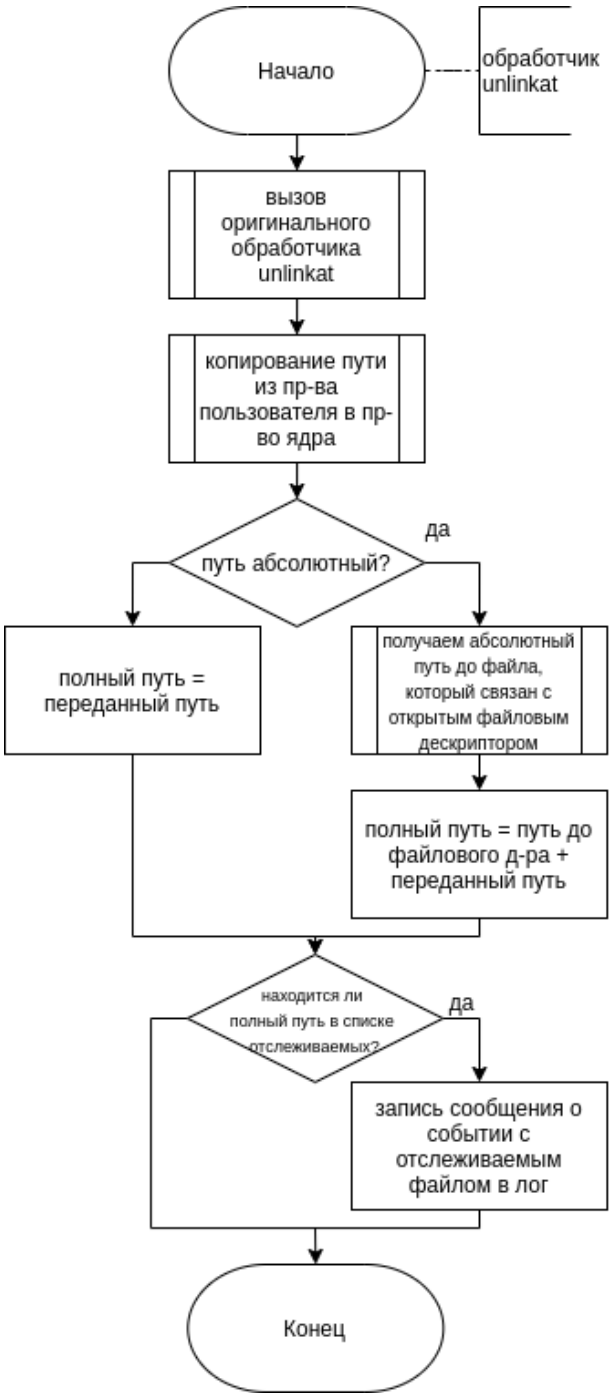

Рис. 6: Наш обработчик unlinkat().

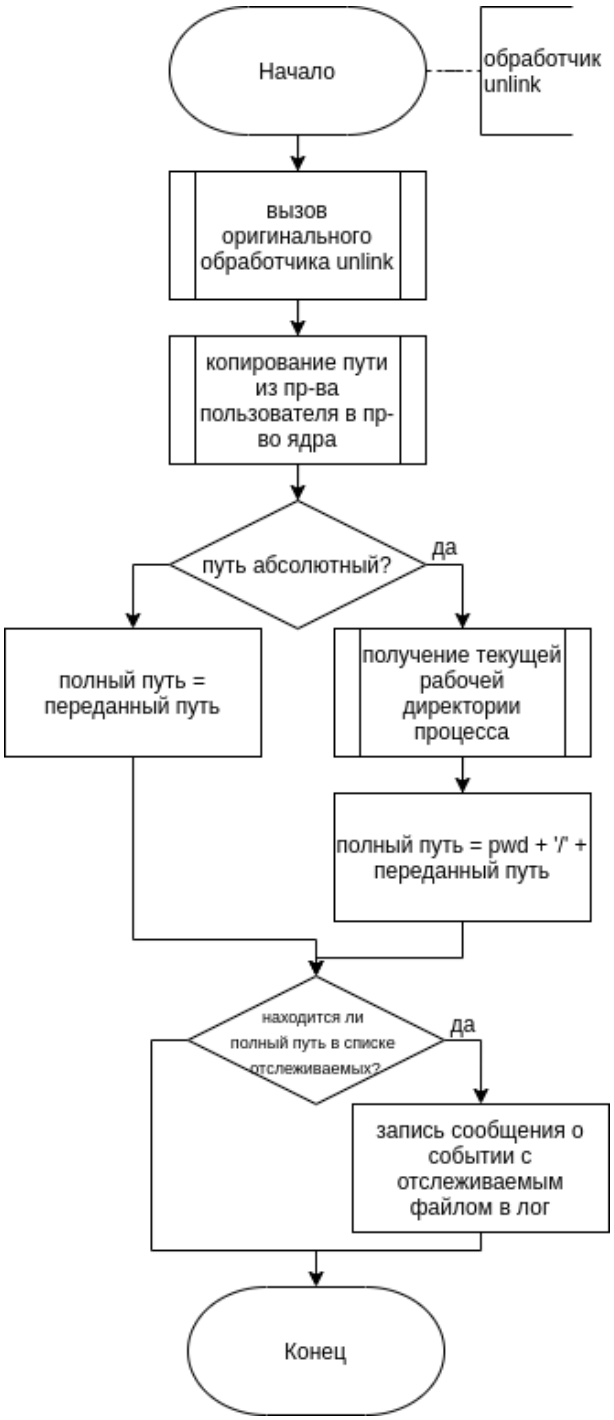

Рис. 7: Наш обработчик unlink().

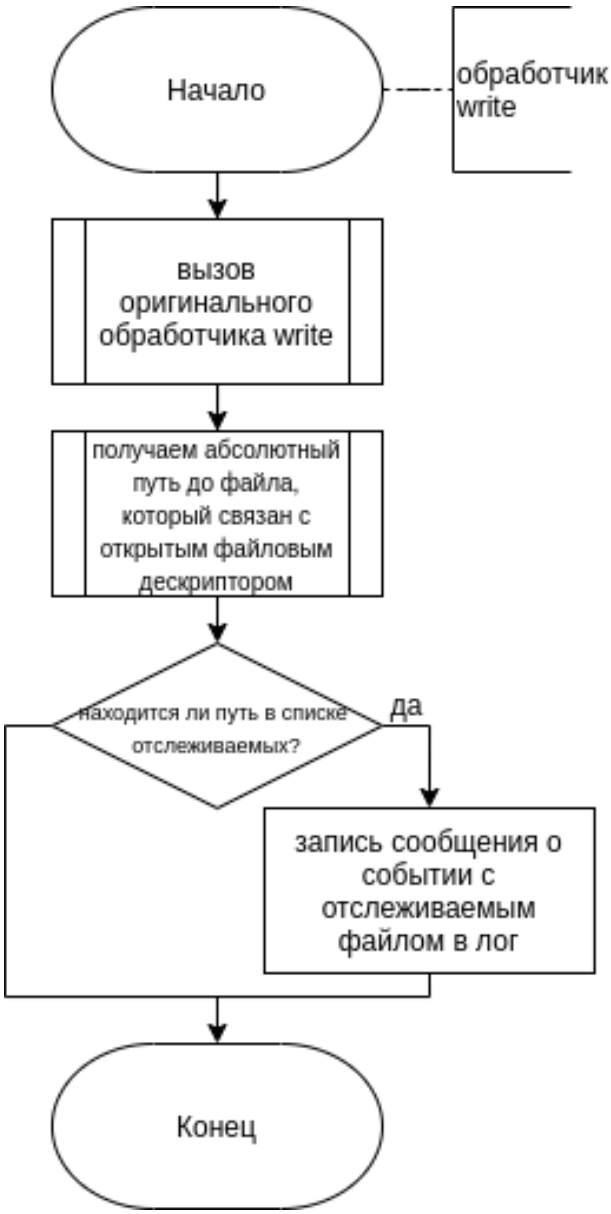

Рис. 8: Наш обработчик write().

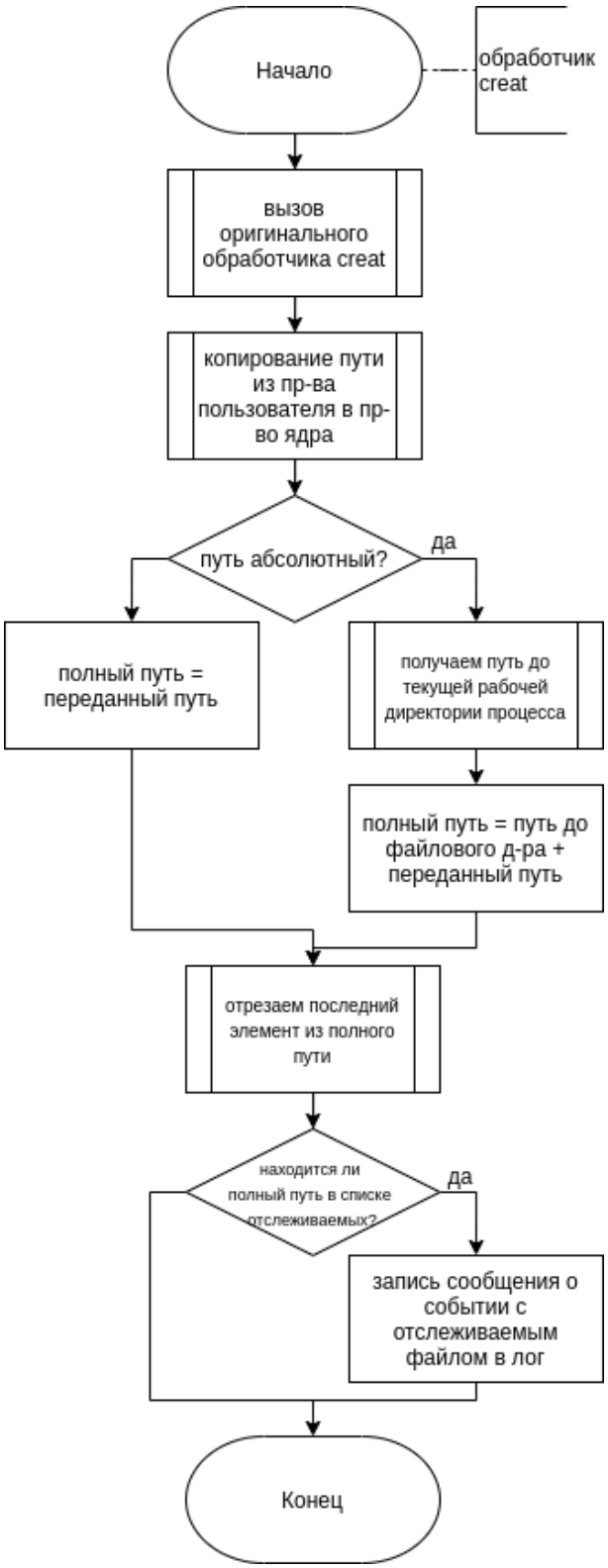

Рис. 9: Наш обработчик creat().

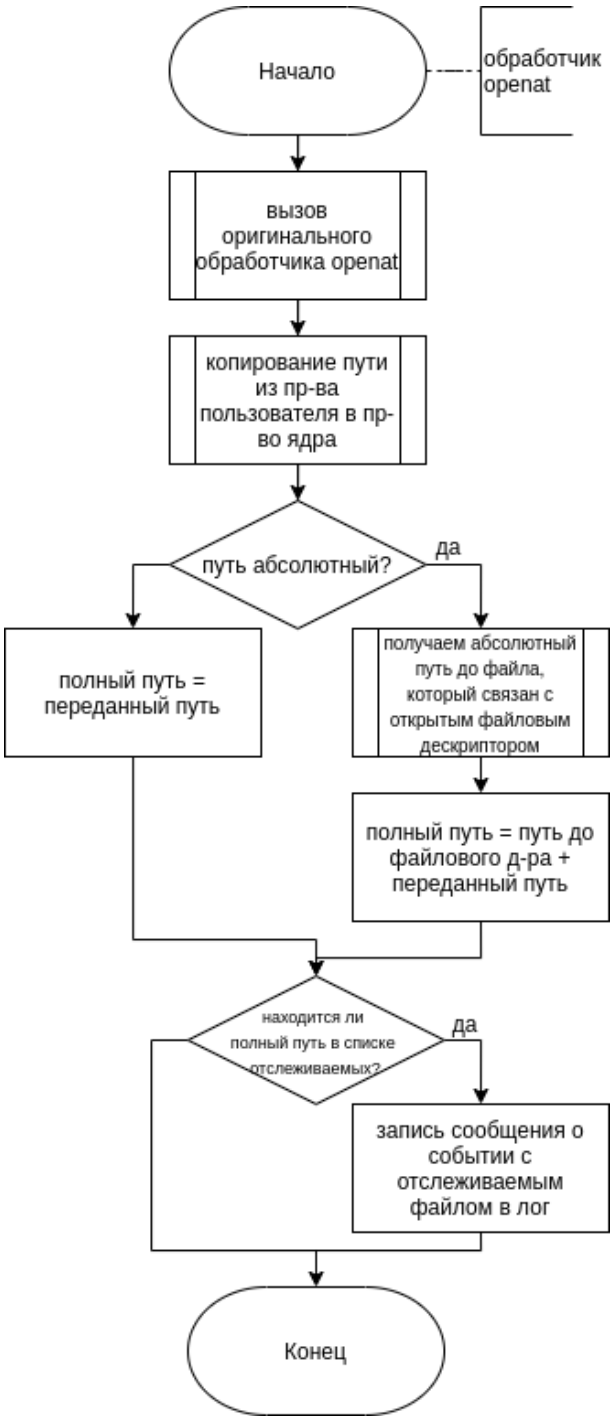

Рис. 10: Наш обработчик openat().

3 Технологический раздел раздел

В данном разделе будет выбран язык и среда программирования, будет приведена реализация наиболее важных функций, будут описаны действия по установке разработанного модуля ядра, а также описано его тестирование.

3.1 Выбор языка программирования

В настоящее время операционная система Linux позволяет создавать загружаемые модули ядра на языках C и Rust.

На виртуальной конференции «2020 Linux Plumbers Conference», где ведущие разработчики ядра Linux обсуждают будущее Linux, говорилось о введении Rust в качестве второго языка ядра. Rust – это системный язык программирования высокого уровня, спонсируемый Mozilla, являющейся материнской компанией Firefox.

Однако Rust еще только начал развиваться и не так популярен, как C, поэтому не обладает достаточным количеством документации и примеров. По этой причине для реализации загружаемого модуля ядра был выбран язык C.

3.2 Выбор среды разработки

Для написания модуля был выбран редактор Visual Studio Code. Он имеет следующие преимущества:

- является «лёгким» редактором кода;
- включает в себя отладчик, инструменты для работы с Git, подсветку синтаксиса, IntelliSense, средства для рефакторинга и навигацию по коду;
- имеет широкие возможности для кастомизации: пользовательские темы, сочетания клавиш и файлы конфигурации;
- распространяется бесплатно, разрабатывается как программное обеспечение с открытым исходным кодом;
- посредством встроенного в продукт пользовательского интерфейса можно загрузить и установить несколько тысяч полезных расширений.

3.3 Взаимодействие с пользователем

Модуль выводит информацию в системный журнал и в свой лог-файл.

В системный журнал выводятся отладочные сообщения, сообщающие о том, что модуль был загружен или выгружен, либо о том, что произошла ошибка при чтении конфигурационного файла.

В лог-файл выводятся сообщения о событиях, произошедших с отслеживаемыми файлами. Лог-файл создается в каталоге /var/log и имеет имя fsmonitor.log. Таким образом, абсолютный путь до лог-файла – /var/log/fsmonitor.log.

Информацию о файлах, действия с которыми необходимо отслеживать, модуль считывает из своего конфигурационного файла, который необходимо создать до загрузки модуля. Конфигурационный файл необходимо создать в каталоге /etc с именем fsmonitor.conf. Таким образом, абсолютный путь до конфигурационного файла – /etc/fsmonitor.conf.

Формат и создание конфигурационного файла описано далее в этой главе.

3.4 Ограничения

Загружаемый модуль был разработан для версий ядра 4.17 - 5.6 и архитектуры x86_64. Это связано с ограничениями на символ ядра kallsyms_lookup_name, который в более поздних версиях ядра перестал быть экспортируемым.

В случае, если в системный вызов передается путь, содержащий «точки» (каталоги . и ..), то такие пути модулем не обрабатываются. Даже если такой конечный путь ведет к тому же файлу, что и один из путей в конфигурационном файле, то действия с таким файлом не будут отслежены модулем.

В модуле не реализован механизм очистки или закольцовывания лог-файла, поэтому при больших количествах перехватов его размер может быть значительным. В случае чрезмерного разрастания лог-файла следует выгрузить модуль и загрузить его заново, так как при загрузке модуль очищает лог-файл.

3.5 Формат конфигурационного файла

Все пути в конфигурационном файле должны быть абсолютными и начинаться с символа «/». Это разделитель пути в операционной системе Linux.

На каждой строке должен располагаться только один путь до одного файла.

Путь не должен заканчиваться символом «/» (в этом случае возможно некорректное определение, является ли файл отслеживаемым).

При необходимости следить за всеми файлами, можно указать в конфигурационном файле маркер «ALL». Если при чтении конфигурационного файла модуль встречает этот маркер, то, независимо от того, указаны ли там другие пути, он будет производить мониторинг действий с любыми файлами. Следует обратить внимание, что в этом случае лог-файл будет очень быстро увеличиваться в размерах (см. ограничения) из-за того, что перехватываемые системные вызовы вызываются постоянно. Поэтому не рекомендуется использовать маркер ALL.

3.6 Реализация загружаемого модуля

Нам потребуется найти и сохранить адрес функции, которую мы будем перехватывать. Ftrace позволяет трассировать функции по имени, но нам всё равно надо знать адрес оригинальной функции, чтобы вызывать её. Добыть адрес можно с помощью kallsyms — списка всех символов в ядре. Поиск адеса функции по ее имени приведен в листинге 7.

Листинг 7: Поиск адреса функции по ее имени.

```
/**
* fh_resolve_hook_address() — поиск адреса функции,
*                             которую будем перехватывать
* @hook: хук, в котором заполнено поле name
*
* @returns 0 в случае успеха, иначе отрицательный код ошибки.
*/
static int fh_resolve_hook_address(struct ftrace_hook *hook)
{
    hook->address = kallsyms_lookup_name(hook->name);

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

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

```
20 #else
21     *((unsigned long *)hook->original) = hook->address;
22 #endif
23
24     return 0;
25 }
```

Дальше необходимо инициализировать структуру ftrace_ops. В ней обязательным полем является лишь func, указывающая на коллбек, но нам также необходимо установить некоторые важные флаги (листинг 8). fh_ftrace_thunk() — это наш коллбек, который ftrace будет вызывать при трассировании функции. Флаги, которые мы устанавливаем, будут необходимы для выполнения перехвата. Они предписывают ftrace сохранить и восстановить регистры процессора, содержимое которых мы сможем изменить в коллбеке.

Для включения перехвата необходимо сначала включить ftrace для интересующей нас функции с помощью ftrace_set_filter_ip(), а затем разрешить ftrace вызывать наш коллбек с помощью register_ftrace_function(). Защита ftrace от рекурсии бесполезна, если изменять %rip, поэтому вычлючаем ее с помощью RECURSION_SAFE. Проверки для защиты от рекурсии будут выполнены на входе в трассируемую функцию.

Листинг 8: Инициализация структуры ftrace_ops.

```
1  /**
2  * fh_install_hook() - регистрация и активация хука
3  * @hook: хук для установки
4  *
5  * @returns 0 в случае успеха, иначе отрицательный код ошибки.
6  */
7  int fh_install_hook(struct ftrace_hook *hook)
8  {
9      int err;
10
11     err = fh_resolve_hook_address(hook);
12     if (err)
13         return err;
14
15      hook->ops.func = fh_ftrace_thunk;
16      hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS |
           FTRACE_OPS_FL_RECURSION_SAFE | FTRACE_OPS_FL_IPMODIFY;
17
```

```
18      err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
19      if (err)
20      {
21          printk(KERN_INFO "ftrace_set_filter_ip() failed: %d\n", err);
22          return err;
23      }
24
25      err = register_ftrace_function(&hook->ops);
26      if (err)
27      {
28          printk(KERN_INFO "register_ftrace_function() failed: %d\n",err);
29          ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
30          return err;
31      }
32      return 0;
33  }
```

Выключается перехват аналогично, только в обратном порядке (листинг 9). После завершения вызова unregister_ftrace_function() гарантируется отсутствие активаций установленного коллбека в системе (а вместе с ним — и наших обёрток). Поэтому мы можем спокойно выгрузить модуль-перехватчик, не опасаясь, что где-то в системе ещё выполняются наши функции.

Листинг 9: Выключение перехвата.

```
1   /**
2   * fh_remove_hook() - выключить хук
3   * @hook: хук для выключения
4   */
5   void fh_remove_hook(struct ftrace_hook *hook)
6   {
7       int err;
8
9       err = unregister_ftrace_function(&hook->ops);
10      if (err)
11      {
12          printk(KERN_INFO "unregister_ftrace_function() failed: %d\n",
                err);
13      }
14
15      err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
16      if (err)
```

```
17      {
18          printk(KERN_INFO "ftrace_set_filter_ip() failed: %d\n", err);
19      }
20  }
```

Ftrace позволяет изменять состояние регистров после выхода из коллбека. Изменяя регистр %rip — указатель на следующую исполняемую инструкцию,— мы изменяем инструкции, которые исполняет процессор — то есть можем заставить его выполнить безусловный переход из текущей функции в нашу. Таким образом мы перехватываем управление на себя.

Коллбек для ftrace выглядит следующим образом (листинг 10). С помощью макроса container_of() мы получаем адрес нашей struct ftrace_hook по адресу внедрённой в неё struct ftrace_ops, после чего заменяем значение регистра %rip в структуре struct pt_regs на адрес нашего обработчика. Спецификатором notrace необходимо помечать функции, запрещённые для трассировки с помощью ftrace. Это помогает предотвратить зависание системы в бесконечном цикле при трассировании всех функций в ядре.

Когда наша обёртка вызовет оригинальную функцию, та опять попадёт в ftrace, который опять вызовет наш коллбек, который опять передаст управление обёртке. Эту бесконечную рекурсию необходимо оборвать. Для этого используется parent_ip – один из аргументов ftrace-коллбека, который содержит адрес возврата в функцию, которая вызвала трассируемую функцию. Мы же можем воспользоваться им для того, чтобы отличить первый вызов перехваченной функции от повторного. При повторном вызове parent_ip должен указывать внутрь нашей обёртки, тогда как при первом — куда-то в другое место ядра. Проверку на вхождение можно очень эффективно выполнить, сравнивая адрес с границами текущего модуля (который содержит все наши функции).

Листинг 10: Обратный вызов для ftrace.

```
1  static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long
       arent_ip, struct ftrace_ops *ops, struct pt_regs *regs)
2  {
3      struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops
           );
4
5  #if USE_FENTRY_OFFSET
6      regs->ip = (unsigned long)hook->function;
7  #else
```

```
8       if (!within_module(parent_ip, THIS_MODULE))
9       regs->ip = (unsigned long)hook->function;
10  #endif
11  }
```

Функция-обёртка, которая вызывается позже, будет выполняться в том же контексте, что и оригинальная функция. Поэтому там можно делать то же, что позволено делать в перехватываемой функции.

Пути к отслеживаемым файлам хранятся в двух списувх – списке файлов и списке директорий. Структуры, описывающие такие списки, приведены в листинге 11.

Листинг 11: Реализация списка.

```
1  struct list_node
2  {
3      struct list_node *next_node;
4      void *value;
5      size_t type_size;
6  };
7
8  struct list
9  {
10     struct list_node *head;
11     struct list_node *tail;
12 };
```

Функции загрузки и выгрузки модуля, а также список всех перехватываемых системных вызовов, представлены в листинге 12

Листинг 12: Загрузка и выгрузка модуля.

```
1  static struct ftrace_hook fs_hooks[] = {
2      HOOK("sys_mkdir", fh_sys_mkdir, &real_sys_mkdir),
3      HOOK("sys_openat", fh_sys_openat, &real_sys_openat),
4      HOOK("sys_creat", fh_sys_creat, &real_sys_creat),
5      HOOK("sys_unlink", fh_sys_unlink, &real_sys_unlink),
6      HOOK("sys_write", fh_sys_write, &real_sys_write),
7      HOOK("sys_unlinkat", fh_sys_unlinkat, &real_sys_unlinkat),
8      HOOK("sys_mkdirat", fh_sys_mkdirat, &real_sys_mkdirat)
9  };
10
11 static int fh_init(void)
```

```
12 {
13     int err;
14     pr_info("==========");
15 #ifndef PTREGS_SYSCALL_STUBS
16     pr_info("Kernel version is not supported\n");
17     return -1;
18 #else
19 
20     init(&monitor_dirs);
21     init(&monitor_files);
22     if ((err = read_config()) != 0)
23     {
24         if (err == -1)
25             pr_info("Unable to read config file\n");
26         if (err == -2)
27             pr_info("Invalid config file format\n");
28         if (err == -3)
29             pr_info("Files writen in config do not exist\n");
30         return err;
31     }
32 
33     f = filp_open(LOG_FILE, O_CREAT | O_TRUNC | O_WRONLY | O_LARGEFILE,
        0);
34     if (IS_ERR(f))
35     {
36         pr_info("Unable to open log file\n");
37         return -1;
38     }
39     pr_info("Log file opened\n");
40 
41     err = fh_install_hooks(fs_hooks, ARRAY_SIZE(fs_hooks));
42     if (err)
43     {
44         free_list(&monitor_dirs);
45         free_list(&monitor_files);
46         pr_info("Unable to install hooks\n");
47         return err;
48     }
49 
50     pr_info("Module loaded\n");
51 
```

```
52      return 0;
53  #endif
54  }
55  module_init(fh_init);
56
57  static void fh_exit(void)
58  {
59      filp_close(f, NULL);
60      pr_info("Log file closed\n");
61      fh_remove_hooks(fs_hooks, ARRAY_SIZE(fs_hooks));
62      pr_info("Hooks removed\n");
63      free_list(&monitor_dirs);
64      free_list(&monitor_files);
65      pr_info("Lists cleared\n");
66      pr_info("Module unloaded\n");
67  }
68  module_exit(fh_exit);
```

Нам также необходимо отключить оптимизацию хвостовых вызовов (tail call optimization) с помощью директивы pragma (листинг 13). Она позволяет компилятору заменить честный вызов функции на прямой переход к её телу, если одна функция вызывает другую и сразу же возвращает её значение.

Оптимизация хвостовых вызовов позволяет сэкономить немного времени на формировании стекового фрейма, в который входит и адрес возврата, сохраняемый в стеке инструкцией CALL. Однако, для нас корректность адреса возврата играет критичную роль — мы используем parent_ip для принятия решения о перехвате. После оптимизации функция наши функции-обработчики больше не сохраняют новый адрес возврата на стеке, там остаётся старый — указывающий в ядро. Поэтому parent_ip продолжает указывать внутрь ядра, что и приводит в конечном итоге к образованию бесконечного цикла.

Листинг 13: Отключение оптимизации хвостовых вызовов.

```
1  #pragma GCC optimize("-fno-optimize-sibling-calls")
```

Полная реализация модуля представлена в приложении А.

3.7 Сборка и установка модуля

Сборка модуля производится с помощью утилиты make. Makefile приведен в листинге 14.

Листинг 14: Makefile.

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
TARGET = fsmonitor

obj-m := $(TARGET).o

default:
	$(MAKE) -I /usr/include/x86_64-linux-gnu -C $(KDIR) M=$(PWD) modules
	make clean
clean:
	@rm -f *.o .*.cmd .*.flags *.mod.c *.order *.mod
	@rm -fR .tmp*
	@rm -rf .tmp_versions
disclean: clean
	@rm *.ko *.symvers
```

Перед установкой модуля необходимо создать конфигурационный файл модуля командой *sudo touch /etc/fsmonitor.conf*. Для записи необходимых путей в конфигурационный файл необходимо запустить текстовый редактор с привелегиями суперпользователя и открыть конфигурационный файл.

После сборки и создания конфигурационного файла, необходимо загрузить модуль в ядро командой *sudo insmod fsmonitor.ko*. В случае, если конфигурационный файл был успешно прочитан, модуль будет загружен в ядро. Для проверки, что модуль загружен, можно использовать команду *lsmod | grep fsmonitor*. Если же во время чтения конфигурационного файла произошла ошибка, модуль не будет загружен.

Проверить сообщения от модуля в системном журнале можно с помощью команды *sudo dmesg | grep ftrace_hook*. Проверить сообщения от модуля в лог-файле можно командой *sudo cat /var/log/fsmonitor.log*.

Для выгрузки модуля из ядра необходимо выполнить команду *sudo rmmod fsmonitor*.

При изменении конфигурационного файла необходимо выгрузить модуль, если

он загружен, и загрузить его в ядно заново.

В случае успешной загрузки модуля в системный журнал будет выведено сообщение «Module loaded». В случае успешной выгрузки модуля в системный журнал будет выведено сообщение «Module unloaded».

3.8 Возможные ошибки при загрузке модуля

При загрузке модуля могут возникнуть следующие ошибки:

- версия ядра не поддерживается;
- ошибка чтения конфигурационного файла;
- ошибка открытия лог-файла;
- ошибка при установке перехватчиков.

При чтении конфигурационного файла могут произойти следующие ошибки:

- конфигурационного файла не существует (он не был создан до загрузки модуля или был удален). В этом случае в системном журнале появится сообщение «Unable to read config file»;
- неверный формат записей в конфигурационном файле. В этом случае в системном журнале появится сообщение «Invalid config file format»;
- файлов, указанных в конфигурационном файле, не существует. В этом случае в системном журнале появится сообщение «Files writen in config do not exist».

При ошибке чтения конфигурационного файла необходимо проверить наличие файла и корректность указанных там путей, после чего снова загрузить модуль.

При ошибке открытия лог-файла в системный журнал будет выведено сообщени «Unable to open log file».

При ошибке установки перехватчиков в системный журнал будет выведено сообщени «Unable to install hooks».

3.9 Тестирование разработанного модуля

Отладка и тестирование в ядре Linux – довольно сложная задача, так как единственным способом взаимодействия ядра с пользователем являются различные лог-файлы или системный журнал. В связи с этим было проведено ручное тестирование разработанного модуля ядра. На каждом шаге был установлен отладочный вывод сообщений в системный журнал и контролировалась их корректность.

С учетом изложенных выше ограничений на работу разработанного модуля, тестирование было пройдено.

Заключение

В данной работе были выполнены следующие задачи.

1. Были проанализированы возможные способы перехвата системных вызовов и выбран наиболее подходящий – ftrace.
2. Был изучен алгоритм перехвата системных вызовов с использованием ftrace.
3. Был реализован загружаемый модуль ядра, перехватывающий системные вызовы openat(), creat(), write(), unlink(), unlinkat(), mkdir(), mkdirat().
4. Было проведено ручное тестирование разработанного модуля. Тесты были пройдены.

Таким образом, поставленная цель была достигнута.

Список литературы

[1] Цилюрик О.И. Модули ядра Linux. Модификация системных вызовов. [Электронный ресурс]. – Режим доступа: http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/08/kern-mod-08-04.html, свободный – (27.11.2020).

[2] Linux Security Modules: General Security Hooks for Linux [Электронный ресурс]. – Режим доступа: https://www.kernel.org/doc/html/latest/security/lsm.html, свободный – (27.11.2020).

[3] Linux Security Modules: General Security Support for the Linux Kernel [Электронный ресурс]. – Режим доступа: https://www.usenix.org/legacy/events/sec02/full_papers/wright/wright_html/index.html, свободный – (27.11.2020).

[4] syscalls(2) — Linux manual page. – Режим доступа: https://man7.org/linux/man-pages/man2/syscalls.2.html, свободный – (27.11.2020).

[5] Kernel Probes (Kprobes). – Режим доступа: https://www.kernel.org/doc/Documentation/kprobes.txt, свободный – (28.11.2020).

[6] Kernel debugging with Kprobes. – Режим доступа: https://www.ibm.com/developerworks/library/l-kprobes/index.html, свободный – (28.11.2020).

[7] Перехват функций в ядре Linux с помощью ftrace: https://habr.com/ru/post/413241/, свободный – (28.11.2020).

[8] Loadable Kernel Module Programming and System Call Interception: https://www.linuxjournal.com/article/4378, свободный – (28.11.2020).

[9] ftrace - Function Tracer: https://www.kernel.org/doc/html/latest/trace/ftrace.html, свободный – (28.11.2020).

[10] Bootlin: https://elixir.bootlin.com, свободный – (18.12.2020).

[11] proc: https://www.opennet.ru/man.shtml?topic=proc&category=5&russian=0, свободный – (18.12.2020).

[12] Что нового в ядре Linux: https://habr.com/ru/post/520950/, свободный – (21.12.2020).

Приложение А

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

#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/fs.h>
#include <linux/fs_struct.h>
#include <linux/timekeeping.h>
#include <linux/stat.h>
#include <linux/types.h>
#include <linux/dcache.h>
#include <asm/segment.h>
#include <linux/buffer_head.h>
#include <linux/fdtable.h>

MODULE_DESCRIPTION("File system monitor");
MODULE_AUTHOR("Ovchinnikova Anastasia");
MODULE_LICENSE("GPL");

/*
 * There are two ways of preventing vicious recursive loops when hooking:
 * - detect recusion using function return address (USE_FENTRY_OFFSET = 0)
 * - avoid recusion by jumping over the ftrace call (USE_FENTRY_OFFSET = 1)
 */
#define USE_FENTRY_OFFSET 0

/**
 * struct ftrace_hook - описывает перехватываемую функцию
 *
 * @name:       имя перехватываемой функции
 *
```

```
37  * @function:    адрес функции-обёртки, которая будет вызываться вместо
38  *               перехваченной функции
39  *
40  * @original:    указатель на место, куда следует записать адрес
41  *               перехватываемой функции, заполняется при установке
42  *
43  * @address:     адрес перехватываемой функции, выясняется при установке
44  *
45  * @ops:         служебная информация ftrace, инициализируется нулями,
46  *               при установке перехвата будет доинициализирована
47  */
48 struct ftrace_hook
49 {
50     const char *name;
51     void *function;
52     void *original;
53
54     unsigned long address;
55     struct ftrace_ops ops;
56 };
57
58 #define LOG_FILE "/var/log/fsmonitor.log"
59 #define CONFIG_PATH "/etc/fsmonitor.conf"
60 #define BUFF_SIZE 1024 //PATH_MAX
61 #define MONITOR_ALL_MARKER "ALL"
62
63 struct list_node
64 {
65     struct list_node *next_node;
66     void *value;
67     size_t type_size;
68 };
69
70 struct list
71 {
72     struct list_node *head;
73     struct list_node *tail;
74 };
75
76 short Monitor_All = 0;
77 struct file *f;
```

```
78 struct list monitor_files, monitor_dirs;
79 loff_t File_Pos = 0;
80
81 /**
82  * инициализация списка
83  */
84 void init(struct list *lst)
85 {
86     lst->head = NULL;
87     lst->tail = NULL;
88 }
89
90 /**
91  * добавление элемента в список
92  */
93 struct list_node *push(struct list *node, void *value, size_t size)
94 {
95     void *next_node = kmalloc(sizeof(struct list_node) + size,
          GFP_KERNEL);
96     struct list_node *__next_node = next_node;
97     __next_node->value = next_node + sizeof(struct list_node);
98     __next_node->type_size = size;
99     __next_node->next_node = NULL;
100    memcpy(__next_node->value, value, size);
101
102    if (node->head == NULL)
103    {
104        node->head = __next_node;
105        node->tail = __next_node;
106    }
107    else
108    {
109        node->tail->next_node = __next_node;
110        node->tail = __next_node;
111    }
112
113    return next_node;
114 }
115
116 /*
117  * удаление элемента из списка
```

```
118  */
119 struct list_node *pop(struct list *node)
120 {
121     struct list_node *value = node->head == NULL
122                                     ? NULL
123                                     : node->head->next_node;
124
125     if (value != NULL)
126     {
127         kfree(node->head);
128         node->head = value;
129     }
130     return value;
131 }
132
133 /**
134  * очищение списка
135  */
136 void free_list(struct list *list)
137 {
138     if (list->head != NULL)
139     {
140         do
141         {
142             //pr_info("Deleting %s\n", *(char **)list->head->value);
143             kfree(*(char **)list->head->value);
144         } while (pop(list) != NULL);
145     }
146 }
147
148 /**
149  * удаляет последний элемент пути (path/to/file -> path/to)
150  */
151 char *cut_last_filename(char *filename)
152 {
153     size_t n;
154     int i = 0, go = 1;
155     n = strlen(filename);
156     for (i = n - 1; i >= 0 && go == 1; --i)
157     {
158         if (filename[i] == '/')
```

```
159         {
160             go = 0;
161         }
162         filename[i] = '\0';
163     }
164     return filename;
165 }
166 
167 int write_log(const char *log)
168 {
169     time64_t cur_seconds;
170     unsigned long local_time;
171     char *new_sl;
172     int ret;
173 
174     new_sl = kmalloc(BUFF_SIZE, GFP_KERNEL);
175     if (new_sl == NULL)
176     {
177         pr_info("Unable to allocate memory\n");
178         return -1;
179     }
180 
181     if (log == NULL)
182     {
183         pr_info("Empty log message.\n");
184         return -1;
185     }
186 
187     if (IS_ERR(f))
188     {
189         pr_info("Failed to write log");
190         return -1;
191     }
192 
193     cur_seconds = ktime_get_real_seconds();
194     local_time = (u32)(cur_seconds - (sys_tz.tz_minuteswest * 60));
195 
196     snprintf(new_sl, BUFF_SIZE, "%.2lu:%.2lu:%.6lu \t %s",
197             (local_time / 3600) % (24),
198             (local_time / 60) % (60),
199             local_time % 60,
```

```
200                 log);
201
202     ret = kernel_write(f, new_sl, strlen(new_sl), &File_Pos);
203     File_Pos += strlen(new_sl);
204     kfree(new_sl);
205
206     return 0;
207 }
208
209 /**
210  * проверяет, содержит ли список имя name
211  * @returns 1 если да, иначе 0
212  */
213 int list_find(struct list *list, const char *name)
214 {
215     struct list_node *_node = list->head;
216     int ret = 0;
217     while (_node != NULL && ret == 0)
218     {
219         if (strcmp(name, *(char **)_node->value) == 0)
220         {
221             ret = 1;
222         }
223         _node = _node->next_node;
224     }
225     return ret;
226 }
227
228 /**
229  * проверяет, находится ли данный файл в списке отслеживаемых
230  * @returns 1 если да, иначе 0
231  */
232 int check_filename(const char *filename, int search_file, int search_dir
        )
233 {
234     int ret = 0;
235
236     if (search_file == 1 && search_dir == 0)
237     {
238         ret = list_find(&monitor_files, filename);
239         return ret;
```

```
240     }
241     if (search_dir == 1 && search_file == 0)
242     {
243         ret = list_find(&monitor_dirs, filename);
244         return ret;
245     }
246     if (search_file == 1 && search_dir == 1)
247     {
248         ret = list_find(&monitor_files, filename);
249         ret += list_find(&monitor_dirs, filename);
250         if (ret == 2)
251         {
252             ret--;
253         }
254         return ret;
255     }
256     return ret;
257 }
258
259 /**
260  * fh_resolve_hook_address() - поиск адреса функции,
261  *                             которую будем перехватывать
262  * @hook: хук, в котором заполнено поле name
263  *
264  * @returns 0 в случае успеха, иначе отрицательный код ошибки.
265  */
266 static int fh_resolve_hook_address(struct ftrace_hook *hook)
267 {
268     hook->address = kallsyms_lookup_name(hook->name);
269
270     if (!hook->address)
271     {
272         printk(KERN_INFO "unresolved symbol: %s\n", hook->name);
273         return -ENOENT;
274     }
275
276 #if USE_FENTRY_OFFSET
277     *((unsigned long *)hook->original) = hook->address +
        MCOUNT_INSN_SIZE;
278 #else
279     *((unsigned long *)hook->original) = hook->address;
```

```
280 #endif
281
282     return 0;
283 }
284
285 /**
286  *  fh_ftrace_thunk() — обратный вызов, который будет вызываться
287                         при трассировании функции
288  * Изменяя регистр %rip — указатель на следующую исполняемую
289  * инструкцию,-- мы изменяем инструкции, которые исполняет процессор
290  * — то есть можем заставить его выполнить безусловный переход из
291  * текущей функции в нашу. Таким образом мы перехватываем
292  * управление на себя.
293  * notrace помогает предотвратить зависание системы в бесконечном цикле
294  */
295 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long
       parent_ip,
296                                     struct ftrace_ops *ops, struct
                                           pt_regs *regs)
297 {
298     // получаем адрес нашей struct ftrace_hook
299     // по адресу внедрённой в неё struct ftrace_ops
300     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops
          );
301
302     // заменяем значение регистра %rip в структуре
303     // struct pt_regs на адрес нашего обработчика
304 #if USE_FENTRY_OFFSET
305     regs->ip = (unsigned long)hook->function;
306 #else
307     // parent_ip содержит адрес возврата в функцию,
308     // которая вызвала трассируемую функцию
309     // можно воспользоваться им для того,
310     // чтобы отличить первый вызов перехваченной функции от повторного
311     if (!within_module(parent_ip, THIS_MODULE))
312         regs->ip = (unsigned long)hook->function;
313 #endif
314 }
315
316 /**
317  * fh_install_hook() - регистрация и активация хука
```

```
318  * @hook: хук для установки
319  *
320  * @returns 0 в случае успеха, иначе отрицательный код ошибки.
321  */
322 int fh_install_hook(struct ftrace_hook *hook)
323 {
324     int err;
325 
326     err = fh_resolve_hook_address(hook);
327     if (err)
328         return err;
329 
330     /*
331      * Мы будем модифицировать регистр %rip поэтому необходим флаг
           IPMODIFY
332      * и SAVE_REGS. Флаги предписывают ftrace сохранить и восстановить
333      * регистры процессора, содержимое которых мы сможем изменить в
334      * коллбеке. Защита ftrace от рекурсии бесполезна, если
335      * изменять %rip, поэтому вычлючаем ее с помощью RECURSION_SAFE.
336      * Проверки для защиты от рекурсии будут выполнены на входе в
337      * трассируемую функцию.
338      */
339     hook->ops.func = fh_ftrace_thunk;
340     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS |
           FTRACE_OPS_FL_RECURSION_SAFE | FTRACE_OPS_FL_IPMODIFY;
341 
342     // включить ftrace для интересующей нас функции
343     err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
344     if (err)
345     {
346         printk(KERN_INFO "ftrace_set_filter_ip() failed: %d\n", err);
347         return err;
348     }
349 
350     // разрешить ftrace вызывать наш коллбек
351     err = register_ftrace_function(&hook->ops);
352     if (err)
353     {
354         printk(KERN_INFO "register_ftrace_function() failed: %d\n", err)
               ;
355         // выключаем ftrace в случае ошибки
```

```
356         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
357         return err;
358     }
359 
360     return 0;
361 }
362 
363 /**
364  * fh_remove_hook() - выключить хук
365  * @hook: хук для выключения
366  */
367 void fh_remove_hook(struct ftrace_hook *hook)
368 {
369     int err;
370 
371     // отключаем наш коллбек
372     err = unregister_ftrace_function(&hook->ops);
373     if (err)
374     {
375         printk(KERN_INFO "unregister_ftrace_function() failed: %d\n",
               err);
376     }
377 
378     // отключаем ftrace
379     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
380     if (err)
381     {
382         printk(KERN_INFO "ftrace_set_filter_ip() failed: %d\n", err);
383     }
384 }
385 
386 /**
387  * fh_install_hooks() - регистрация хуков
388  * @hooks: массив хуков для регистрации
389  * @count: количество хуков для регистрации
390  *
391  * Если один из хуков не удалось зарегистрировать,
392  * то все остальные (которые удалось установить), удаляются.
393  *
394  * @returns 0 в случае успеха, иначе отрицательный код ошибки.
395  */
```

```
396 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
397 {
398     int err = 0;
399     size_t i;
400 
401     for (i = 0; i < count && err == 0; i++)
402     {
403         err = fh_install_hook(&hooks[i]);
404     }
405     if (err == 0)
406     {
407         return 0;
408     }
409     while (i != 0)
410     {
411         fh_remove_hook(&hooks[--i]);
412     }
413 
414     return err;
415 }
416 
417 /**
418  * fh_remove_hooks() — выключить хуки
419  * @hooks: массив хуков для выключения
420  * @count: количество хуков для выключения
421  */
422 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
423 {
424     size_t i;
425 
426     for (i = 0; i < count; i++)
427         fh_remove_hook(&hooks[i]);
428 }
429 
430 #ifndef CONFIG_X86_64
431 #error Currently only x86_64 architecture is supported
432 #endif
433 
434 #if defined(CONFIG_X86_64) &&                                   \
435     (LINUX_VERSION_CODE >= KERNEL_VERSION(4, 17, 0)) && \
436     (LINUX_VERSION_CODE <= KERNEL_VERSION(5, 6, 0))
```

```
437 #define PTREGS_SYSCALL_STUBS 1
438 #endif
439
440 /*
441  * Оптимизация хвостового вызова может помешать обнаружению рекурсии
442  * на основе обратного адреса в стеке.
443  * Отключаем ее, чтобы предотвратить зависание.
444  */
445 #if !USE_FENTRY_OFFSET
446 #pragma GCC optimize("-fno-optimize-sibling-calls")
447 #endif
448
449 /**
450  * копирование имени файла из пользовательского пространства в пространс
        тво ядра
451  */
452 static char *duplicate_filename(const char __user *filename)
453 {
454     char *kernel_filename;
455     int res;
456
457     if (filename == NULL)
458     {
459         pr_info("Filename is null\n");
460         return NULL;
461     }
462
463     kernel_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
464     if (!kernel_filename)
465     {
466         pr_info("kmalloc() failed\n");
467         return NULL;
468     }
469
470     if ((res = strncpy_from_user(kernel_filename, filename, BUFF_SIZE))
        < 0)
471     {
472         pr_info("strncpy_from_user() failed: %d \n", res);
473         kfree(kernel_filename);
474         return NULL;
475     }
```

```
476
477     return kernel_filename;
478 }
479
480 //asmlinkage long sys_open(const char __user *filename,
481 //                int flags, umode_t mode);
482
483 // настоящий обработчик системного вызова openat
484 static asmlinkage long (*real_sys_openat)(struct pt_regs *regs);
485
486 // обработчик системного вызова openat
487 static asmlinkage long fh_sys_openat(struct pt_regs *regs)
488 {
489     int ret;
490     char *kernel_filename;
491     char *proc_filename;
492     char *buffer;
493     int fd;
494     char *full_filename;
495
496     ret = real_sys_openat(regs);
497     fd = (long)(void *)regs->di;
498
499     // копируем имя директории из пространства пользователя в пространст
        во ядра
500     kernel_filename = duplicate_filename((void *)regs->si);
501     if (kernel_filename == NULL)
502     {
503         pr_info("Unable to duplicate filename\n");
504         return ret;
505     }
506
507     proc_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
508     buffer = kmalloc(BUFF_SIZE, GFP_KERNEL);
509     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
510     if (proc_filename == NULL || buffer == NULL || full_filename == NULL
        )
511     {
512         pr_info("Unable to allocate memory\n");
513         kfree(kernel_filename);
514         if (proc_filename != NULL) kfree(proc_filename);
```

```
515         if (buffer != NULL) kfree(buffer);
516         if (full_filename != NULL) kfree(full_filename);
517         return ret;
518     }
519 
520     // если путь не является абсолютным, получаем абсолютный путь до фай
        ла, который связан с открытым файловым дескриптором
521     if (fd != AT_FDCWD && kernel_filename[0] != '/')
522     {
523         char *path;
524         struct path pwd;
525         char *pwd_buff;
526         struct file *_file;
527 
528         snprintf(proc_filename, BUFF_SIZE, "/proc/%d/fd/%d", current->
            pid, fd);
529         _file = filp_open(proc_filename, 0, 0);
530 
531         pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
532         if (pwd_buff == NULL)
533         {
534             pr_info("Unable to allocate memory\n");
535             kfree(kernel_filename);
536             kfree(proc_filename);
537             kfree(full_filename);
538             kfree(buffer);
539             return ret;
540         }
541         pwd = _file->f_path;
542         path_get(&pwd);
543         path = d_path(&pwd, pwd_buff, BUFF_SIZE);
544         kfree(pwd_buff);
545 
546         full_filename = strcat(full_filename, path);
547         full_filename = strcat(full_filename, "/");
548         full_filename = strcat(full_filename, kernel_filename);
549     }
550     else // путь абсолютный, ничего делать не надо
551     {
552         full_filename = strcpy(full_filename, kernel_filename);
553     }
```

```
554
555     // проверяем, находится ли файл или директория в списке отслеживаемы
            x
556     if (check_filename(full_filename, 1, 1) == 1)
557     {
558         char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
559         if (buff == NULL)
560         {
561             pr_info("Unable to allocate memory\n");
562             kfree(kernel_filename);
563             kfree(proc_filename);
564             kfree(full_filename);
565             kfree(buffer);
566             return ret;
567         }
568         snprintf(buff, BUFF_SIZE * 2, "Process %d OPENAT '%s'. Syscall
                returned %d\n",
569                 current->pid, full_filename, ret);
570         write_log(buff);
571         kfree(buff);
572     }
573
574     kfree(kernel_filename);
575     kfree(proc_filename);
576     kfree(full_filename);
577     kfree(buffer);
578
579     return ret;
580 }
581
582 //static asmlinkage long (*real_sys_creat)(const char __user *pathname,
        umode_t mode);
583 // настоящий обработчик системного вызова creat
584 static asmlinkage long (*real_sys_creat)(struct pt_regs *regs);
585
586 // обработчик системного вызова creat
587 static asmlinkage long fh_sys_creat(struct pt_regs *regs)
588 {
589     int ret;
590     char *kernel_filename;
591     char *full_filename;
```

```
592     char *path;
593     struct path pwd;
594     char *pwd_buff;
595 
596     ret = real_sys_creat(regs);
597 
598     // копируем имя директории из пространства пользователя в пространст
        во ядра
599     kernel_filename = duplicate_filename((void *)regs->di);
600     if (kernel_filename == NULL)
601     {
602         pr_info("Unable to duplicate filename\n");
603         return ret;
604     }
605 
606     // получаем путь до текущей рабочей директории процесса
607     pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
608     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
609     if (pwd_buff == NULL || full_filename == NULL)
610     {
611         pr_info("Unable to allocate memory\n");
612         kfree(kernel_filename);
613         if (pwd_buff != NULL) kfree(pwd_buff);
614         if (full_filename != NULL) kfree(full_filename);
615         return ret;
616     }
617 
618     pwd = current->fs->pwd;
619     path_get(&pwd);
620     path = d_path(&pwd, pwd_buff, BUFF_SIZE);
621 
622     if (kernel_filename[0] != '/')
623     {
624         full_filename = strcat(full_filename, path);
625         full_filename = strcat(full_filename, "/");
626         full_filename = strcat(full_filename, kernel_filename);
627     }
628     else
629     {
630         full_filename = strcpy(full_filename, kernel_filename);
631     }
```

```
632     full_filename = cut_last_filename(full_filename);
633
634     // проверяем, находится ли файл или директория в списке отслеживаемы
        x
635     if (check_filename(full_filename, 0, 1) == 1)
636     {
637         char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
638         if (buff == NULL)
639         {
640             pr_info("Unable to allocate memory\n");
641             kfree(kernel_filename);
642             kfree(full_filename);
643             kfree(pwd_buff);
644             return ret;
645         }
646         snprintf(buff, BUFF_SIZE * 2, "Process %d CREAT '%s' at '%s'.
               Syscall returned %d\n",
647                 current->pid, kernel_filename, full_filename, ret);
648         write_log(buff);
649         kfree(buff);
650     }
651
652     kfree(kernel_filename);
653     kfree(full_filename);
654     kfree(pwd_buff);
655
656     return ret;
657 }
658
659 //static asmlinkage long (*real_sys_write)(unsigned int fd, const char
       __user *buf,
660 //                                         size_t count);
661 // настоящий обработчик системного вызова write
662 static asmlinkage long (*real_sys_write)(struct pt_regs *regs);
663
664 // обработчик системного вызова write
665 static asmlinkage long fh_sys_write(struct pt_regs *regs)
666 {
667     int ret;
668     char *proc_filename;
669     char *buffer;
```

```
670     int fd;
671     char *full_filename;
672     char *path;
673     struct path pwd;
674     char *pwd_buff;
675     struct file *_file;
676 
677     ret = real_sys_write(regs);
678     fd = (long)(void *)regs->di;
679 
680     proc_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
681     buffer = kmalloc(BUFF_SIZE, GFP_KERNEL);
682     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
683     if (proc_filename == NULL || buffer == NULL || full_filename == NULL
        )
684     {
685         pr_info("Unable to allocate memory\n");
686         if (proc_filename != NULL) kfree(proc_filename);
687         if (buffer != NULL) kfree(buffer);
688         if (full_filename != NULL) kfree(full_filename);
689         return ret;
690     }
691 
692     snprintf(proc_filename, BUFF_SIZE, "/proc/%d/fd/%d", current->pid,
        fd);
693     _file = filp_open(proc_filename, 0, 0);
694     if (IS_ERR(_file))
695     {
696         //pr_info("Unable to open proc file\n");
697         return ret;
698     }
699 
700     pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
701     if (pwd_buff == NULL)
702     {
703         pr_info("Unable to allocate memory\n");
704         kfree(proc_filename);
705         kfree(buffer);
706         kfree(full_filename);
707         return ret;
708     }
```

```
709
710     // получаем путь до файла, в который производится запись
711     pwd = _file->f_path;
712     path_get(&pwd);
713     path = d_path(&pwd, pwd_buff, BUFF_SIZE);
714     kfree(pwd_buff);
715
716     full_filename = strcat(full_filename, path);
717
718     // проверяем, находится ли файл или директория в списке отслеживаемы
        x
719     if (check_filename(full_filename, 1, 1) == 1)
720     {
721         char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
722         if (buff == NULL)
723         {
724             pr_info("Unable to allocate memory\n");
725             kfree(proc_filename);
726             kfree(buffer);
727             kfree(full_filename);
728             return ret;
729         }
730         snprintf(buff, BUFF_SIZE * 2, "Process %d WRITE AT '%s'. Syscall
                returned %d\n",
731                 current->pid, full_filename, ret);
732         write_log(buff);
733         kfree(buff);
734     }
735
736     kfree(proc_filename);
737     kfree(full_filename);
738     kfree(buffer);
739
740     return ret;
741 }
742
743 // настоящий обработчик системного вызова unlink
744 static asmlinkage long (*real_sys_unlink)(struct pt_regs *regs);
745
746 // обработчик системного вызова unlink
747 static asmlinkage long fh_sys_unlink(struct pt_regs *regs)
```

```
748 {
749     int ret;
750     char *kernel_filename;
751     char *full_filename;
752     char *path;
753     struct path pwd;
754     char *pwd_buff;
755 
756     ret = real_sys_unlink(regs);
757 
758     // копируем имя директории из пространства пользователя в пространст
        во ядра
759     kernel_filename = duplicate_filename((void *)regs->di);
760     if (kernel_filename == NULL)
761     {
762         pr_info("Unable to duplicate filename\n");
763         return ret;
764     }
765 
766     pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
767     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
768     if (pwd_buff == NULL || full_filename == NULL)
769     {
770         pr_info("Unable to allocate memory\n");
771         kfree(kernel_filename);
772         if (pwd_buff != NULL) kfree(pwd_buff);
773         if (full_filename != NULL) kfree(full_filename);
774         return ret;
775     }
776 
777     // получаем путь до текущей рабочей директории процесса
778     pwd = current->fs->pwd;
779     path_get(&pwd);
780     path = d_path(&pwd, pwd_buff, BUFF_SIZE);
781 
782     if (kernel_filename[0] != '/')
783     {
784         full_filename = strcat(full_filename, path);
785         full_filename = strcat(full_filename, "/");
786         full_filename = strcat(full_filename, kernel_filename);
787     }
```

```
788     else
789     {
790         full_filename = strcpy(full_filename, kernel_filename);
791     }
792 
793     // проверяем, находится ли файл или директория в списке отслеживаемы
        x
794     if (check_filename(full_filename, 1, 1) == 1)
795     {
796         char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
797         if (buff == NULL)
798         {
799             pr_info("Unable to allocate memory\n");
800             kfree(kernel_filename);
801             kfree(full_filename);
802             kfree(pwd_buff);
803             return ret;
804         }
805         snprintf(buff, BUFF_SIZE * 2, "Process %d UNLINK '%s'. Syscall
               returned %d\n", current->pid, full_filename, ret);
806         write_log(buff);
807         kfree(buff);
808     }
809 
810     kfree(kernel_filename);
811     kfree(full_filename);
812     kfree(pwd_buff);
813 
814     return ret;
815 }
816 
817 // static asmlinkage long sys_unlinkat(int dfd, const char __user *
       pathname, int flag);
818 
819 // настоящий обработчик системного вызова unlinkat
820 static asmlinkage long (*real_sys_unlinkat)(struct pt_regs *regs);
821 
822 // обработчик системного вызова unlinkat
823 static asmlinkage long fh_sys_unlinkat(struct pt_regs *regs)
824 {
825     int ret;
```

```
826     char *kernel_filename;
827     char *proc_filename;
828     char *buffer;
829     int fd;
830     char *full_filename;
831 
832     ret = real_sys_unlinkat(regs);
833     fd = (long)(void *)regs->di;
834 
835     // копируем имя файла из пространства пользователя в пространство ядра
836     kernel_filename = duplicate_filename((void *)regs->si);
837 
838     if (kernel_filename == NULL)
839     {
840         pr_info("Unable to duplicate filename\n");
841         return ret;
842     }
843 
844     proc_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
845     buffer = kmalloc(BUFF_SIZE, GFP_KERNEL);
846     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
847     if (proc_filename == NULL || buffer == NULL || full_filename == NULL)
848     {
849         pr_info("Unable to allocate memory\n");
850         kfree(kernel_filename);
851         if (proc_filename != NULL) kfree(proc_filename);
852         if (buffer != NULL) kfree(buffer);
853         if (full_filename != NULL) kfree(full_filename);
854         return ret;
855     }
856     // если путь не является абсолютным, получаем абсолютный путь до файла, который связан с открытым файловым дескриптором
857     if (fd != AT_FDCWD && kernel_filename[0] != '/')
858     {
859         char *path;
860         struct path pwd;
861         char *pwd_buff;
862         struct file *_file;
863 
```

```
864             snprintf(proc_filename, BUFF_SIZE, "/proc/%d/fd/%d", current->
                    pid, fd);
865             _file = filp_open(proc_filename, 0, 0);
866 
867             pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
868             if (pwd_buff == NULL)
869             {
870                 pr_info("Unable to allocate memory\n");
871                 kfree(kernel_filename);
872                 kfree(proc_filename);
873                 kfree(full_filename);
874                 return ret;
875             }
876             pwd = _file->f_path;
877             path_get(&pwd);
878             path = d_path(&pwd, pwd_buff, BUFF_SIZE);
879             kfree(pwd_buff);
880 
881             full_filename = strcat(full_filename, path);
882             full_filename = strcat(full_filename, "/");
883             full_filename = strcat(full_filename, kernel_filename);
884         }
885         else // путь абсолютный, ничего делать не надо
886         {
887             full_filename = strcpy(full_filename, kernel_filename);
888         }
889 
890         // проверяем, находится ли файл или директория в списке отслеживаемы
                x
891         if (check_filename(full_filename, 1, 1) == 1)
892         {
893             char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
894             if (buff == NULL)
895             {
896                 pr_info("Unable to allocate memory\n");
897                 kfree(kernel_filename);
898                 kfree(proc_filename);
899                 kfree(full_filename);
900                 return ret;
901             }
```

```
902         snprintf(buff, BUFF_SIZE * 2, "Process %d UNLINKAT '%s'. Syscall
                returned %d\n",
903                 current->pid, full_filename, ret);
904         write_log(buff);
905         kfree(buff);
906     }
907 
908     kfree(kernel_filename);
909     kfree(proc_filename);
910     kfree(full_filename);
911 
912     return ret;
913 }
914 
915 //static asmlinkage long sys_mkdirat(int dfd, const char __user *
       pathname, umode_t mode);
916 
917 // настоящий обработчик системного вызова mkdirat
918 static asmlinkage long (*real_sys_mkdirat)(struct pt_regs *regs);
919 
920 // обработчик системного вызова mkdirat
921 static asmlinkage long fh_sys_mkdirat(struct pt_regs *regs)
922 {
923     int ret;
924     char *kernel_filename;
925     char *proc_filename;
926     char *buffer;
927     int fd;
928     char *full_filename;
929 
930     ret = real_sys_mkdirat(regs);
931     fd = (long)(void *)regs->di;
932 
933     // копируем имя файла из пространства пользователя в пространство яд
           ра
934     kernel_filename = duplicate_filename((void *)regs->si);
935     if (kernel_filename == NULL)
936     {
937         pr_info("Unable to duplicate filename\n");
938         return ret;
939     }
```

```
940
941     proc_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
942     buffer = kmalloc(BUFF_SIZE, GFP_KERNEL);
943     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
944     if (proc_filename == NULL || buffer == NULL || full_filename == NULL
            )
945     {
946         pr_info("Unable to allocate memory\n");
947         kfree(kernel_filename);
948         if (proc_filename != NULL) kfree(proc_filename);
949         if (buffer != NULL) kfree(buffer);
950         if (full_filename != NULL) kfree(full_filename);
951         return ret;
952     }
953     // если путь не является абсолютным, получаем абсолютный путь до фай
            ла, который связан с открытым файловым дескриптором
954     if (fd != AT_FDCWD && kernel_filename[0] != '/')
955     {
956         char *path;
957         struct path pwd;
958         char *pwd_buff;
959         struct file *_file;
960
961         snprintf(proc_filename, BUFF_SIZE, "/proc/%d/fd/%d", current->
                pid, fd);
962         _file = filp_open(proc_filename, 0, 0);
963
964         pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
965         if (pwd_buff == NULL)
966         {
967             pr_info("Unable to allocate memory\n");
968             kfree(kernel_filename);
969             kfree(proc_filename);
970             kfree(full_filename);
971             kfree(buffer);
972             return ret;
973         }
974         pwd = _file->f_path;
975         path_get(&pwd);
976         path = d_path(&pwd, pwd_buff, BUFF_SIZE);
977         kfree(pwd_buff);
```

```
978
979         full_filename = strcat(full_filename, path);
980     }
981     else // путь абсолютный, ничего делать не надо
982     {
983         full_filename = strcpy(full_filename, kernel_filename);
984     }
985
986     // проверяем, находится ли файл или директория в списке отслеживаемы
        x
987     if (check_filename(full_filename, 0, 1) == 1)
988     {
989         char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
990         if (buff == NULL)
991         {
992             pr_info("Unable to allocate memory\n");
993             kfree(kernel_filename);
994             kfree(proc_filename);
995             kfree(full_filename);
996             kfree(buffer);
997             return ret;
998         }
999         snprintf(buff, BUFF_SIZE * 2, "Process %d MKDIR '%s' AT '%s'.
            Syscall returned %d\n",
1000                 current->pid, kernel_filename, full_filename, ret);
1001         write_log(buff);
1002         kfree(buff);
1003     }
1004
1005     kfree(kernel_filename);
1006     kfree(proc_filename);
1007     kfree(full_filename);
1008     kfree(buffer);
1009
1010     return ret;
1011 }
1012
1013 // настоящий обработчик системного вызова mkdir
1014 static asmlinkage long (*real_sys_mkdir)(struct pt_regs *regs);
1015
1016 // обработчик системного вызова mkdir
```

```
1017 static asmlinkage long fh_sys_mkdir(struct pt_regs *regs)
1018 {
1019     long ret;
1020     char *kernel_filename;
1021     char *full_filename;
1022     char *path;
1023     struct path pwd;
1024     char *pwd_buff;
1025 
1026     ret = real_sys_mkdir(regs);
1027 
1028     // копируем имя директории из пространства пользователя в пространст
        во ядра
1029     kernel_filename = duplicate_filename((void *)regs->di);
1030     if (kernel_filename == NULL)
1031     {
1032         pr_info("Unable to duplicate filename\n");
1033         return ret;
1034     }
1035 
1036     pwd_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
1037     full_filename = kmalloc(BUFF_SIZE, GFP_KERNEL);
1038     if (pwd_buff == NULL || full_filename == NULL)
1039     {
1040         pr_info("Unable to allocate memory\n");
1041         kfree(kernel_filename);
1042         if (pwd_buff != NULL) kfree(pwd_buff);
1043         if (full_filename != NULL) kfree(full_filename);
1044         return ret;
1045     }
1046 
1047     // получаем путь до текущей рабочей директории процесса
1048     pwd = current->fs->pwd;
1049     path_get(&pwd);
1050     path = d_path(&pwd, pwd_buff, BUFF_SIZE);
1051 
1052     if (kernel_filename[0] != '/')
1053     {
1054         full_filename = strcat(full_filename, path);
1055         full_filename = strcat(full_filename, "/");
1056         full_filename = strcat(full_filename, kernel_filename);
```

```
1057        }
1058        else
1059        {
1060            full_filename = strcpy(full_filename, kernel_filename);
1061        }
1062        full_filename = cut_last_filename(full_filename);
1063
1064        // проверяем, находится ли файл или директория в списке отслеживаемы
               х
1065        if (check_filename(full_filename, 0, 1) == 1)
1066        {
1067            char *buff = kmalloc(BUFF_SIZE * 2, GFP_KERNEL);
1068            if (buff == NULL)
1069            {
1070                pr_info("Unable to allocate memory\n");
1071                kfree(kernel_filename);
1072                kfree(pwd_buff);
1073                kfree(full_filename);
1074                return ret;
1075            }
1076            snprintf(buff, BUFF_SIZE * 2, "Process %d MKDIR '%s' AT %s'.
                   Syscall returned %ld\n", current->pid, kernel_filename,
                   full_filename, ret);
1077            write_log(buff);
1078            kfree(buff);
1079        }
1080
1081        kfree(kernel_filename);
1082        kfree(full_filename);
1083        kfree(pwd_buff);
1084
1085        return ret;
1086    }
1087
1088    /*
1089     * ядра x86_64 имеют особое соглашение о названиях входных точек системн
           ых вызовов.
1090     */
1091    #ifdef PTREGS_SYSCALL_STUBS
1092    #define SYSCALL_NAME(name) ("__x64_" name)
1093    #else
```

```
1094 #define SYSCALL_NAME(name) (name)
1095 #endif
1096
1097 #define HOOK(_name, _function, _original) \
1098     {                                     \
1099         .name = SYSCALL_NAME(_name),      \
1100         .function = (_function),          \
1101         .original = (_original),          \
1102     }
1103
1104 void my_str_replace(char *str, size_t len, char what, char with)
1105 {
1106     size_t i;
1107     for (i = 0; i < len; ++i)
1108     {
1109         if (str[i] == what)
1110         {
1111             str[i] = with;
1112         }
1113     }
1114 }
1115
1116 /**
1117  * Проверяет, является ли указанный путь абсолютным и до существующего ф
        айла.
1118  * @returns -2 — путь некорректный в принципе,
1119  *          -3 — путь до несуществующего файла,
1120  *          0 — файл существует и является директорией,
1121  *          1 — файл существует и не является директорией,
1122  *          2 — передана пустая строка
1123  */
1124 int is_valid(const char *filename)
1125 {
1126     struct file *_f;
1127
1128     if (strlen(filename) == 0)
1129     {
1130         return 2;
1131     }
1132     if (filename[0] != '/')
1133     {
```

```
1134            return -2;
1135        }
1136
1137        _f = filp_open(filename, 0, 0);
1138        if (IS_ERR(_f))
1139        {
1140            pr_info("Unable to open file\n");
1141            return -3;
1142        }
1143        else
1144        {
1145            int is_dir = S_ISDIR(_f->f_inode->i_mode);
1146            filp_close(_f, NULL);
1147            return is_dir;
1148        }
1149    }
1150
1151    /**
1152     * Проверяет имя файла
1153     * @returns -2 — путь некорректный в принципе,
1154     *          -3 — путь до несуществующего файла,
1155     *          0 — файл существует и является директорией,
1156     *          1 — файл существует и не является директорией,
1157     *          2 — передана пустая строка,
1158     *          3 — указаное имя файла == MONITOR_ALL_MARKER
1159     */
1160    int process_filename(const char *filename)
1161    {
1162        if (strcmp(filename, MONITOR_ALL_MARKER) == 0)
1163        {
1164            Monitor_All = 1;
1165            return 3;
1166        }
1167        if (strlen(filename) == 0)
1168        {
1169            return 2;
1170        }
1171        return is_valid(filename);
1172    }
1173
1174    /**
```

```
1175  * чтение данных из конфигурационного файла
1176  * @returns -1 в случае ошибки
1177  *          -2 в случае, если данные в конфигурационном файле записаны в
          неверном формате
1178  *          -3 в случае, если файлы, записанные в конфигурационный файл,
          не существуют
1179  *           0 в случае успеха
1180  */
1181 int read_config(void)
1182 {
1183     struct file *config_file;
1184     int res = 1;
1185     loff_t offset = 0;
1186     loff_t inner_offset = 0;
1187     int return_val = 0;
1188     size_t data_len;
1189 
1190     config_file = filp_open(CONFIG_PATH, O_RDONLY, 0);
1191     if (IS_ERR(config_file))
1192     {
1193         return -1;
1194     }
1195 
1196     pr_info("Reading config from %s\n", CONFIG_PATH);
1197 
1198     while (res > 0 && return_val == 0)
1199     {
1200         char *data_buff = kmalloc(BUFF_SIZE, GFP_KERNEL);
1201         if (IS_ERR(data_buff))
1202         {
1203             pr_info("Unable to allocate memory\n");
1204             return_val = -1;
1205         }
1206         else
1207         {
1208             offset = inner_offset;
1209             res = kernel_read(config_file, data_buff, BUFF_SIZE, &offset
                  );
1210             if (res > 0)
1211             {
1212                 my_str_replace(data_buff, res, '\n', '\0');
```

```
1213                data_len = strlen(data_buff) - 1;
1214                if (data_buff[data_len] == '/')
1215                {
1216                    data_buff[data_len] = '\0';
1217                }
1218                inner_offset += strlen(data_buff) + 1;
1219                return_val = process_filename(data_buff);
1220                if (return_val == 3) // считали маркер, будем следить за
                        всеми файлами
1221                {
1222                    kfree((void *)data_buff);
1223                }
1224                else if (return_val == 2) // считали пустую строку, чита
                       ем дальше
1225                {
1226                    kfree((void *)data_buff);
1227                    return_val = 0;
1228                }
1229                else if (return_val == 0) // файл существует и не являет
                       ся директорией, следим за ним, читаем дальше
1230                {
1231                    push(&monitor_files, &data_buff, sizeof(char *));
1232                    return_val = 0;
1233                }
1234                else if (return_val == 1) // файл существует и является
                       директорией, следим за ним, читаем дальше
1235                {
1236                    push(&monitor_dirs, &data_buff, sizeof(char *));
1237                    return_val = 0;
1238                }
1239            }
1240        }
1241    }
1242    filp_close(config_file, NULL);
1243    return return_val;
1244 }
1245
1246 static struct ftrace_hook fs_hooks[] = {
1247    HOOK("sys_mkdir", fh_sys_mkdir, &real_sys_mkdir),
1248    HOOK("sys_openat", fh_sys_openat, &real_sys_openat),
1249    HOOK("sys_creat", fh_sys_creat, &real_sys_creat),
```

```
1250     HOOK("sys_unlink", fh_sys_unlink, &real_sys_unlink),
1251     HOOK("sys_write", fh_sys_write, &real_sys_write),
1252     HOOK("sys_unlinkat", fh_sys_unlinkat, &real_sys_unlinkat),
1253     HOOK("sys_mkdirat", fh_sys_mkdirat, &real_sys_mkdirat)
1254 };
1255
1256 static int fh_init(void)
1257 {
1258     int err;
1259     pr_info("==========");
1260 #ifndef PTREGS_SYSCALL_STUBS
1261     pr_info("Kernel version is not supported\n");
1262     return -1;
1263 #else
1264
1265     init(&monitor_dirs);
1266     init(&monitor_files);
1267     if ((err = read_config()) != 0)
1268     {
1269         if (err == -1)
1270             pr_info("Unable to read config file\n");
1271         if (err == -2)
1272             pr_info("Invalid config file format\n");
1273         if (err == -3)
1274             pr_info("Files writen in config do not exist\n");
1275         return err;
1276     }
1277
1278     f = filp_open(LOG_FILE, O_CREAT | O_TRUNC | O_WRONLY | O_LARGEFILE,
        0);
1279     if (IS_ERR(f))
1280     {
1281         pr_info("Unable to open log file\n");
1282         return -1;
1283     }
1284     pr_info("Log file opened\n");
1285
1286     err = fh_install_hooks(fs_hooks, ARRAY_SIZE(fs_hooks));
1287     if (err)
1288     {
1289         free_list(&monitor_dirs);
```

```
1290            free_list(&monitor_files);
1291            pr_info("Unable to install hooks\n");
1292            return err;
1293        }
1294
1295        pr_info("Module loaded\n");
1296
1297        return 0;
1298    #endif
1299    }
1300    module_init(fh_init);
1301
1302    static void fh_exit(void)
1303    {
1304        filp_close(f, NULL);
1305        pr_info("Log file closed\n");
1306        fh_remove_hooks(fs_hooks, ARRAY_SIZE(fs_hooks));
1307        pr_info("Hooks removed\n");
1308        free_list(&monitor_dirs);
1309        free_list(&monitor_files);
1310        pr_info("Lists cleared\n");
1311        pr_info("Module unloaded\n");
1312    }
1313    module_exit(fh_exit);
```