



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 КУРСОВОЙ РАБОТЕ НА ТЕМУ:

«Мониторинг системных вызовов в ОС Linux»

Студент группы ИУ7-75Б

(Подпись, дата)

М.А. Козлов

(И.О. Фамилия)

Руководитель курсового проекта

(Подпись, дата)

Н. Ю. Рязанова

(И.О. Фамилия)

2021 г.

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)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7
(Индекс)
И.В.Рудаков
(И.О.Фамилия)
« ____ » _____ 2021 г.

ЗАДАНИЕ
на выполнение курсового проекта

по дисциплине Операционные системы

Студент группы ИУ7-75Б

Козлов Максим Антонович
(Фамилия, имя, отчество)

Тема курсового проекта Мониторинг системных вызовов в ОС Linux.

Направленность КП (учебный, исследовательский, практический, производственный)
учебный

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Техническое задание Разработать загружаемый модуль ядра, который обладает следующим функционалом – вывод информации о вызове и передаваемых параметрах функций: bdev_read_page, bdev_write_page, sys_read, sys_write, sys_open, sys_close, random_read.

Оформление курсового проекта:

Расчетно-пояснительная записка на 25-30 листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

На защиту проекта должна быть представлена презентация, состоящая из 15-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО, результаты проведенных исследований.

Дата выдачи задания « ____ » _____ 2021г.

Руководитель курсовой работы

(Подпись, дата)

Н. Ю. Рязанова

(И.О.Фамилия)

Студент

(Подпись, дата)

М. А. Козлов

(И.О.Фамилия)

СОДЕРЖАНИЕ

Введение	4
1 Аналитическая часть	5
1.1 Существующие решения для трассировки ядра	5
1.1.1 Linux Security API	5
1.1.2 Модификация таблицы системных вызовов	5
1.1.3 Kprobes	6
1.1.4 Kernel tracepoints	7
1.1.5 Сплайсинг	8
1.1.6 Ftrace	9
1.1.7 Вывод	10
1.2 Загружаемые модули ядра Linux	11
1.3 Пространство пользователя и пространство ядра	12
1.4 Вывод	12
2 Конструкторская часть	13
2.1 Общая архитектура приложения	13
2.2 Перехват функций	13
2.2.1 Перехват функций через системную таблицу	13
2.2.2 Перехват функций через ftrace	16
2.3 Вывод	19
3 Технологическая часть	20
3.1 Выбор языка программирования	20
3.2 Функции-обёртки перехватываемых системных вызовов	20
3.3 Функции-обёртки перехватываемых ftrace функций	21
3.4 Примеры работы	23
3.5 Вывод	23
Заключение	24
Список использованных источников	25

Введение

При работе с операционной системой Linux может потребоваться перехватывать вызовы важных функций внутри ядра (например, запуск процессов или запись и чтение с жесткого диска) для обеспечения возможности мониторинга активности в системе или превентивного блокирования деятельности подозрительных процессов.

Целью данной курсовой работы является разработка загружаемого модуля ядра, позволяющего логировать вызов следующих функций: `bdev_read_page`, `bdev_write_page`, `sys_read`, `sys_write`, `sys_open`, `sys_close`, `random_read`.

Для достижения поставленной цели требуется решить следующие задачи:

- 1) проанализировать существующие подходы для перехвата функций;
- 2) реализовать загружаемый модуль ядра;
- 3) исследовать поведение перехваченных функций.

1 Аналитическая часть

В данном разделе будут проанализированы различные подходы к трассировке ядра и перехвату функций, а также основные принципы загружаемых модулей ядра.

1.1 Существующие решения для трассировки ядра

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

1.1.1 Linux Security API

Linux Security API – это специальный интерфейс, позволяющий трассировать ядро Linux начиная с версии 2.6 [1]. В критических местах кода ядра расположены вызовы security-функций, которые в свою очередь вызывают коллбэки, установленные security-модулем. Security-модуль может изучать контекст операции и принимать решение о её разрешении или запрете.

Linux Security API имеет ряд ограничений:

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

Если по поводу множественности модулей позиция разработчиков ядра неоднозначная, то запрет на динамическую загрузку принципиальный: security-модуль должен быть частью ядра, чтобы обеспечивать безопасность постоянно, с момента загрузки. Таким образом, для использования Security API необходимо поставлять собственную сборку ядра, а также интегрировать дополнительный модуль с SELinux или AppArmor, которые используются популярными дистрибутивами.

1.1.2 Модификация таблицы системных вызовов

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

У этого подхода есть определённые преимущества:

- 1) Полный контроль над любыми системными вызовами. Используя его можно гарантировать перехват действия, выполняемого пользовательским процессом.
- 2) Минимальные накладные расходы. Обновление таблицы системных вызовов происходит один раз при загрузке и выгрузке модуля. Помимо полезной нагрузки мониторинга, един-

ственным дополнительным расходом является лишний вызов оригинального обработчика системного вызова.

3) Минимальные требования к версии ядра. Системные таблицы используются в любом ядре Линукса. Однако в новых версиях ядра для передачи аргументов в системные функции используются `struct pt_regs`, а в старых через стек. Но данную проблему можно решить, используя условную компиляцию и макрос `LINUX_VERSION_CODE`.

Однако модификация таблицы системных вызовов не лишена недостатков:

1) Техническая сложность реализации. Для замены указателей системных функций в таблице необходимо решить следующие задачи:

- а) поиск таблицы системных вызовов;
- б) обход защиты от модификации таблицы;
- в) атомарное и безопасное выполнение замены указателей.

2) Невозможность перехвата некоторых обработчиков. В ядрах до версии 4.16 обработка системных вызовов для архитектуры `x86_64` содержала целый ряд оптимизаций. Некоторые из них требовали того, что обработчик системного вызова реализовался на ассемблере. Соответственно, подобные обработчики порой сложно, а иногда и вовсе невозможно заменить на свои, написанные на Си. Более того, в разных версиях ядра используются разные оптимизации, что добавляет различные технические сложности.

3) Перехватываются только системные вызовы. Данный подход позволяет подменить таблицу системных вызовов, но это ограничивает количество функций, которые можно мониторить.

1.1.3 Kprobes

Kprobes – специализированное API, в первую очередь предназначенного для отладки и трассирования ядра [3]. Этот интерфейс позволяет устанавливать пред- и пост- обработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут их изменять. Таким образом, можно было бы получить как мониторинг, так и возможность влиять на дальнейший ход работы.

Преимущества, которые даёт использование kprobes для перехвата:

1) Обладает хорошо задокументированным интерфейсом, большинство подводных камней уже найдено, их работа по возможности оптимизирована.

2) 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. Аналогично, kretprobes реализуются через подмену адреса возврата на стеке и позволяют перехватить возврат из любой функции (за исключением тех, которые управление в принципе не возвращают).

Недостаткам kprobes являются:

1) Техническая сложность. Kprobes – это только способ установить точку останова в любом месте 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. Для блокировки вызова функции необходимо вручную модифицировать состояние процесса так, чтобы процессор подумал, что он уже вернул управление из функции.

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

3) Нетривиальные накладные расходы. Расстановка точек останова дорогая, но она выполняется единоразово. Точки останова не влияют на остальные функции, однако их обработка относительно недешёвая. Для архитектуры x86_64 реализована jump-оптимизация, существенно уменьшающая стоимость kprobes, но она всё ещё остаётся больше, чем, например, при модификации таблицы системных вызовов.

4) 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.

5) Отключенное вытеснение. Kprobes основывается на прерываниях и может менять значения в регистрах процессора, следовательно для синхронизации все обработчики выполняются с отключенным вытеснением (preemption). Это накладывает определённые ограничения на обработчики: в них нельзя ждать – выделять много памяти, заниматься вводом-выводом, спать в таймерах и семафорах, и прочее.

1.1.4 Kernel tracepoints

Kernel tracepoints – это фреймворк для трассировки ядра, сделанный через статическое инструментирование кода [4]. Точка трассировки, помещенная в код, обеспечивает ловушку для вызова функции (зонда), которую можно предоставить во время выполнения. Точка трассировки может быть "включена" (к ней подключен зонд) или "выключена" (зонд не подключен). Когда точка трассировки выключена, она не оказывает никакого эффекта, за исключением проверки условия для перехода и добавлением нескольких байтов для вызова функции в конце инструментированной функции и добавление данных структуру в отдельный раздел. Когда точка трассировки включена, предоставляемая функция вызывается каждый раз при выполнении точки трассировки в контексте выполнения вызывающей стороны.

Точки трассировки можно разместить в важных местах кода. Это легкие обработчики, которые могут передавать произвольное количество параметров, прототипы которых описаны

в объявлении точки трассировки, помещенном в файл заголовка. В основном они используются для отслеживания и учета производительности.

Преимуществами данного способа являются:

- 1) Малые накладные расходы на внедрение в загружаемые модули ядра. Необходимо только вызвать функцию трассировки в необходимом месте.
- 2) Маленькие затраты по памяти и процессорному времени.

К недостаткам можно отнести:

- 1) Имена точек трассировки являются глобальными для ядра. Они считаются одинаковыми независимо от того, находятся ли они в ядре или в загружаемых модулях.
- 2) Для добавления точек останова в ядровые функции необходимо перекомпилировать ядро, если для данных функций не даны точки не определены.
- 3) Относительно плохо задокументированное API.

1.1.5 Сплайсинг

Сплайсинг - способ перехвата функций, заключающийся в замене инструкций в начале функции на безусловный переход, ведущий в обработчик [5]. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. С помощью двух переходов вшивается (splice in) дополнительный код в функцию, поэтому такой подход называется сплайсингом. Именно таким образом и реализуется jump-оптимизация для kprobes. Используя сплайсинг можно добиться тех же результатов, но без дополнительных расходов на kprobes и с полным контролем ситуации.

Преимуществами сплайсинга являются:

- 1) Минимальные требования к ядру. Сплайсинг не требует каких-либо особенных опций в ядре и работает в начале любой функции, необходимо лишь знать её адрес.
- 2) Минимальные накладные расходы. Два безусловных перехода, которые надо выполнить перехваченному коду, чтобы передать управление обработчику и обратно. Подобные переходы отлично предсказываются процессором и являются очень дешёвыми.
- 3) Менее заметны для детекторов вредоносных программ.
- 4) Позволяет подключать все доступные символы в ядре.

К недостаткам данного подхода можно отнести:

- 1) Требуется надёжный метод дизассемблирования кода ядра.
- 2) Зависит от архитектуры, поскольку каждая архитектура имеет свои собственные инструкции перехода.
- 3) Большая техническая сложность реализации. Ниже приведён краткий и неполный список задач, которые необходимо решить:

- а) синхронизация установки и снятия перехвата;

- б) обход защиты от модификации областей памяти с исходным кодом ядра;
- в) инвалидация кешей процессора после замены инструкций;
- г) дизассемблирование заменяемых инструкций, чтобы скопировать их целиком;
- д) проверка на отсутствие переходов внутрь заменяемого куска;
- е) проверка на возможность переместить заменяемый кусок в другое место;

1.1.6 Ftrace

Ftrace – это фреймворк для трассирования ядра на уровне функций [6]. Его можно использовать для отладки или анализа задержек и проблем с производительностью, возникающих за пределами пользовательского пространства.

Хотя ftrace обычно считается трассировщиком функций, на самом деле это структура из нескольких различных утилит трассировки. Имеется трассировка задержки для изучения того, что происходит между отключенными и включенными прерываниями, а также для вытеснения и с момента пробуждения задачи до фактического запланированного выполнения задачи.

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.

Вызов ftrace не является дешёвой операцией, поэтому для популярных архитектур доступна оптимизация: динамический ftrace. Суть заключается в том, что ядро знает расположение всех вызовов mcount() или __fentry__() и на ранних этапах загрузки заменяет их машинный код на пор – специальную ничего не делающую инструкцию. При включении трассирования в нужные функции вызовы ftrace добавляются обратно. Таким образом, если ftrace не используется, то его влияние на систему минимально.

Ниже описаны преимущества данного подхода:

- 1) Использование готовых интерфейсов в ядре существенно упрощает код. Вся установка перехвата требует пары вызовов функций и заполнение двух полей в структуре.
- 2) Перехват любой функции по имени. Для указания интересующей нас функции достаточно написать её имя в обычной строке. Не требуются большой разбор внутренних структур данных ядра, сканирование памяти, или дизассемблирование кода ядра. Можно перехватить любую функцию (даже не экспортируемую для модулей), зная лишь её имя.
- 3) Перехват совместим с трассировкой. Данный способ не конфликтует с ftrace, так что с ядра можно снимать полезные показатели производительности. Однако использование kprobes или сплайсинга может помешать механизмам ftrace.
- 4) Средние накладные расходы. Накладные расходы на ftrace меньше, чем у kprobes (так как ftrace не использует точки останова), но они выше, чем у сплайсинга, сделанного вручную.

В действительности динамический ftrace является сплайсингом, только дополнительно выполняющий код ftrace и другие коллбеки.

К недостаткам данного подхода можно отнести:

1) Требования к конфигурации ядра. Для успешного выполнения перехвата функций с помощью ftrace ядро должно предоставлять целый ряд возможностей:

- а) список символов kallsyms для поиска функций по имени;
- б) фреймворк ftrace для выполнения трассировки;
- в) различные опции ftrace важные для перехвата.

2) Оборачиваются функции целиком. Как и сплайсинг, данный подход полностью оборачивает вызовы функций. Однако, если сплайсинг технически возможно выполнить в любом месте функции, то ftrace срабатывает исключительно при входе. Естественно, обычно это не вызывает сложностей и даже наоборот удобно, но подобное ограничение иногда может быть недостатком.

1.1.7 Вывод

В таблице 1.1 приведено сравнение рассмотренных методов.

Таблица 1.1 — Сравнение существующих методов трассировки ядра.

Критерий	Linux Security	Модификация syscall table	Kprobes	Kernel tracepoints	Сплайсинг	Ftrace
Накладные расходы	Средние	Минимальные	Большие	Малые	Минимальные	Средние
Сложность реализации	Средняя	Средняя	Большая	Средняя	Очень большая	Малая
Требуется компиляция ядра	Да	Нет	Иногда	Иногда	Нет	Нет
Возможности мониторинга	Ряд функции	Системные функции	Любое место в ядре	Любое место в ядре	Любое место в ядре	Большинство функции по имени
Документация	Средняя	Средняя	Средняя	Малая	Малая	Большая
Требования к ядру	Версия старше 2.6	Нет	Версия младше 2.5	-	Версия младше 2.5	Наличие ftrace

В ходе анализа приведенных подходов к перехвату функций, были выбраны метод модификации таблицы системных вызовов и фреймворк ftrace, так как первый позволит перехватить

вызов системных функции `sys_read`, `sys_write`, `sys_open` и `sys_close` без больших накладных расходов. Второй же метод будет перехватывать функции `bdev_read_page`, `bdev_write_page` и `random_read`, т.к. они не являются системными. Основными критериями выбора были необходимость перекомпилировать ядро, требования к ядру, а также сложность программной реализации.

Выбранные методы перехвата требуют, чтобы сигнатуры перехватываемой функций и функции-обёртки должны в точности совпадать. Иначе, очевидно, аргументы будут переданы неправильно и дальнейшее поведение ядра не определено. Из-за этого возникают сложности с поддержкой разных версий ядер Линукса, т.к. разработчики ядра не поддерживают обратную совместимость. В рамках курсовой работы будет реализована поддержка лишь одной версии ядра – 5.0.

1.2 Загружаемые модули ядра Linux

Linux является динамическим ядром, поддерживающим добавление и удаление программных компонентов на лету [7]. Это означает, есть возможность добавить функциональность в ядро (и убрать её), когда система запущена и работает. Они называются динамически загружаемыми модулями ядра. Ядро Linux предлагает поддержку довольно большого числа типов (или классов) модулей. Каждый модуль является подготовленным объектным кодом (не слинкованным для самостоятельной работы), который может быть динамически подключен в работающее ядро, а позднее может быть выгружен из ядра. Каждый модуль ядра регистрирует себя для того, чтобы обслуживать в будущем запросы, и его функция инициализации немедленно прекращается. Иными словами, задача функции инициализации модуля заключается в подготовке функций модуля для последующего вызова. Функция выхода модуля вызывается только непосредственно перед выгрузкой модуля. Функция выхода модуля должна тщательно отменить все изменения, сделанные функцией инициализации, или функции модуля сохранятся до перезагрузки системы. Возможность выгрузить модуль помогает сократить время разработки; можно тестировать последовательные версии новых драйверов, не прибегая каждый раз к длительному циклу выключения/перезагрузки. Модуль связан только с ядром и может вызывать только те функции, которые экспортированы ядром, нет библиотек для установления связи. Например, функция `printk`, является версией `printf`, определённой в ядре и экспортированной для модулей.

В основном загружаемые модули ядра используются в следующих целях:

- 1) драйверы устройств;
- 2) драйверы файловой системы;
- 3) добавление новых системных вызовов и расширение функционала существующих.

1.3 Пространство пользователя и пространство ядра

Систему памяти в Linux можно разделить на две различные области: пространства ядра и пользовательского пространства. Пространство ядра – адресное пространство в абсолютных адресах, где выполняется ядро операционной системы и предоставляет свои услуги [8]. Пользовательское пространство (user space) – адресное пространство виртуальной памяти операционной системы, в которой выполняются процессы пользователя [9].

Доступ к пространству ядра для пользовательских процессов возможен только с помощью системных вызовов. Системные вызовы – это запросы в Unix-подобной операционной системе активного процесса для службы, выполняемой ядром, такой как ввод/вывод (I/O) или создание процесса и т.д.

1.4 Вывод

В данном разделе были проанализированы подходы к трассировке ядра и перехвату функций и выбраны наиболее оптимальные методы для реализации поставленных задач – модификации таблицы системных вызовов и фреймворк ftrace, а также рассмотрены основные принципы загружаемых модулей ядра и понятия пространства ядра и пространства пользователя.

2 Конструкторская часть

В данном разделе будет рассмотрена общая архитектура приложения и методы перехвата функций с помощью системной таблицы и ftrace.

2.1 Общая архитектура приложения

В состав программного обеспечения входит один загружаемый модуль ядра, который следит за вызовом определенных функций, с последующим логированием информации об аргументах и имени вызываемой функции.

2.2 Перехват функций

2.2.1 Перехват функций через системную таблицу

Общий алгоритм установки перехвата функции через системную таблицу состоит из следующих этапов:

- 1) поиск адреса системной таблицы;
- 2) сохранение адресов оригинальных обработчиков системных вызовов;
- 3) снятие защиты от модификации таблицы;
- 4) модификация таблицы;
- 5) восстановление защиты от записи.

Рассмотрим каждый из этапов.

Адрес системной таблицы можно найти с помощью функции `kallsyms_lookup_name`. Данная функция позволяет найти абсолютный адрес любого экспортируемого символа ядра. На листинге 2.1 приведён код поиска начального адреса таблицы системных вызовов.

Листинг 2.1 — Поиск начального адреса таблицы системных вызовов

```
1 /* Адрес таблицы системных вызовов */
2 static unsigned long * __sys_call_table;
3
4 /* Поиск начального адреса таблицы системных вызовов */
5 __sys_call_table = kallsyms_lookup_name("sys_call_table");
```

Таблица системных вызовов находится в области памяти доступной только на чтение, поэтому на время изменения требуется отключить глобальную защиту страниц от записи, изменением флага WP (Write Protection) в регистре CR0. Данные функции представлены в листинге 2.2. Однако встроенная в Linux функция `write_cr0()` не позволяет изменять бит WP, поэтому требуется своя функция, представленная в листинге 2.3.

Листинг 2.2 — Функции включение и отключение защиты от записи страницы

```
1 #define CR0_WP 0x00010000
2 static inline void protect_memory(void)
3 {
```

```

4     unsigned long cr0 = read_cr0();
5     cr0_write(cr0 | CR0_WP);
6 }
7
8 static inline void unprotect_memory(void)
9 {
10     unsigned long cr0 = read_cr0();
11     cr0_write(cr0 & ~CR0_WP);
12 }

```

Листинг 2.3 — Функция изменения значения регистра cr0

```

1 extern unsigned long __force_order;
2 inline void cr0_write(unsigned long cr0)
3 {
4     // mov cr0, rax
5     asm volatile("mov %0, %%cr0" : "+r"(cr0), "+m"(__force_order));
6 }

```

В новых версиях ядра сигнатура обработчика системного вызова описывается следующим образом (листинг 2.4):

Листинг 2.4 — Сигнатура обработчиков системных вызовов

```

1 typedef asmlinkage long ( *syscall_t)(const struct pt_regs *);

```

где struct pt_regs может отличаться для разных версий ядра и процессоров. Одно из определений представлено в листинге 2.5 [10].

Листинг 2.5 — Структура регистров

```

1 struct pt_regs {
2     /*
3      * NB: 32-bit x86 CPUs are inconsistent as what happens in the
4      * following cases (where %seg represents a segment register):
5      *
6      * — pushl %seg: some do a 16-bit write and leave the high
7      *   bits alone
8      * — movl %seg, [mem]: some do a 16-bit write despite the movl
9      * — IDT entry: some (e.g. 486) will leave the high bits of CS
10     *   and (if applicable) SS undefined.
11     *
12     * Fortunately, x86-32 doesn't read the high bits on POP or IRET,
13     * so we can just treat all of the segment registers as 16-bit
14     * values.
15     */
16     unsigned long bx;
17     unsigned long cx;
18     unsigned long dx;

```

```

19     unsigned long si;
20     unsigned long di;
21     unsigned long bp;
22     unsigned long ax;
23     unsigned short ds;
24     unsigned short __dsh;
25     unsigned short es;
26     unsigned short __esh;
27     unsigned short fs;
28     unsigned short __fsh;
29     /* On interrupt, gs and __gsh store the vector number. */
30     unsigned short gs;
31     unsigned short __gsh;
32     /* On interrupt, this is the error code. */
33     unsigned long orig_ax;
34     unsigned long ip;
35     unsigned short cs;
36     unsigned short __csh;
37     unsigned long flags;
38     unsigned long sp;
39     unsigned short ss;
40     unsigned short __ssh;
41 };

```

Номера системных вызовов описаны в исходном коде линукса [11]. Зная их и начальный адрес таблицы можно получить и запомнить абсолютные адреса оригинальных системных вызовов. После чего изменить их на функции обёртки (листинг 2.6).

Листинг 2.6 — Модификация таблицы системных вызовов

```

1  /* Получение адресов оригинальных системных вызовов */
2  orig_open   = (syscall_t) __sys_call_table[__NR_open];
3  orig_close  = (syscall_t) __sys_call_table[__NR_close];
4  orig_read   = (syscall_t) __sys_call_table[__NR_read];
5  orig_write  = (syscall_t) __sys_call_table[__NR_write];
6
7  /* Для модификации системной таблицы необходимо снять со страницы защиту от записи */
8  unprotect_memory();
9
10 /* Замена системных функций hooks */
11 __sys_call_table[__NR_open]    = (unsigned long)hook_open;
12 __sys_call_table[__NR_close]   = (unsigned long)hook_close;
13 __sys_call_table[__NR_read]    = (unsigned long)hook_read;
14 __sys_call_table[__NR_write]   = (unsigned long)hook_write;
15
16 /* Восстановить защиту от записи */
17 protect_memory();

```

Восстановление системных вызовов происходит аналогично перехвату, только в таблицу записываются изначальные адреса обработчиков (2.7).

Листинг 2.7 — Восстановление таблицы системных вызовов

```
1 /* Восстановление системной таблицы */
2 unprotect_memory();
3 __sys_call_table[__NR_open]      = (unsigned long)orig_open;
4 __sys_call_table[__NR_close]     = (unsigned long)orig_close;
5 __sys_call_table[__NR_read]      = (unsigned long)orig_read;
6 __sys_call_table[__NR_write]     = (unsigned long)orig_write;
7 protect_memory();
```

2.2.2 Перехват функций через ftrace

В листинге 2.8 представлена структура `struct ftrace_hook`, которая описывает каждую перехватываемую функцию. Необходимо заполнить только первые три поля: имя, адрес функции-обертки и оригинальной функции. Остальные поля считаются деталью реализации. Для повышения компактности кода рекомендуется использовать макросы представленные в листинге 2.9.

Листинг 2.8 — Структура перехватываемой функции

```
1 struct ftrace_hook {
2     const char *name; // имя перехватываемой функции
3     void *function;   // адрес функции-обертки
4     void *original;   // адрес оригинальной функции
5
6     unsigned long address;
7     struct ftrace_ops ops;
8 };
```

Листинг 2.9 — Макрос для заполнения структуры перехватываемой функции

```
1 #define HOOK(_name, _hook, _orig) \
2 { \
3     .name = (_name), \
4     .function = (_hook), \
5     .original = (_orig), \
6 }
7
8 /* массив перехватываемых функций */
9 static struct ftrace_hook hooks[] = {
10     HOOK(<func name>, <hook func>, <original func>)
11 };
```


Для трассировки функции ядра с помощью ftrace необходимо сначала найти и сохранить её адрес. Аналогично поиску адреса системной таблицы для поиска адреса функции можно использовать функцию kallsyms (листинг 2.10).

Листинг 2.10 — Поиск адреса функции по символьному имени

```
1  /* ftrace.h */
2  #define MCOUNT_INSN_SIZE    4 /* sizeof mcount call */
3
4  #define USE_FENTRY_OFFSET 0
5  #if !USE_FENTRY_OFFSET
6  #pragma GCC optimize("-fno-optimize-sibling-calls")
7  #endif
8
9  static int fh_resolve_hook_address(struct ftrace_hook *hook)
10 {
11     hook->address = kallsyms_lookup_name(hook->name);
12
13     if (!hook->address)
14     {
15         printk(KERN_DEBUG "unresolved symbol: %s\n", hook->name);
16         return -ENOENT;
17     }
18
19 #if USE_FENTRY_OFFSET
20     *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_SIZE;
21 #else
22     *((unsigned long*) hook->original) = hook->address;
23 #endif
24     return 0;
25 }
```

Недостатком ftrace является возможность бесконечной рекурсии при перехвате функции, в результате чего может произойти паника системы. Существуют два способа избежать этого:

- 1) обнаружить рекурсию, посмотрев на адрес возврата функции;
- 2) перепрыгнуть через вызов ftrace (+ MCOUNT_INSN_SIZE).

Для переключения между этими методами существует флаг USE_FENTRY_OFFSET. Если установлено значение 0, используется первый вариант, в противном случае – второй.

Если используется первый вариант, то необходимо отключить защиту, которую предоставляет ftrace. Она работает на сохранение регистров возврата rip, но он будет изменён нами, поэтому следует реализовать собственные средства защиты. Все сводится к тому, что в .original поле ftrace_hook структуры устанавливается адрес памяти системного вызова, указанного в .name. Для корректной работы необходимо указать следующие флаги:

- 1) FTRACE_OPS_FL_IP_MODIFY информирует ftrace, что регистр rip может быть изменён;
- 2) FTRACE_OPS_FL_SAVE_REGS передавать struct pt_regs исходного системного вызова хуку (необходим для установки FTRACE_OPS_FL_IP_MODIFY);
- 3) FTRACE_OPS_FL_RECURSION_SAFE отключает встроенную защиту от рекурсий.

Листинг 2.11 — Защита от рекурсии.

```

1 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
2     struct ftrace_ops *ops, struct pt_regs *regs)
3 {
4     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
5
6     #if USE_FENTRY_OFFSET
7         /* рекурсия не возникнет, т.к. с помощью смещения в оригинальной функции был проп
8            ущен вызов ftrace */
9         regs->ip = (unsigned long) hook->function;
10    #else
11        /* проверка адреса возврата функции */
12        if(!within_module(parent_ip, THIS_MODULE))
13            regs->ip = (unsigned long) hook->function;
14    #endif
15 }

```

Листинг 2.12 — Установка перехвата функции

```

1 int fh_install_hook(struct ftrace_hook *hook)
2 {
3     int err;
4     err = fh_resolve_hook_address(hook);
5     if(err)
6         return err;
7
8     hook->ops.func = fh_ftrace_thunk;
9     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
10        | FTRACE_OPS_FL_RECURSION_SAFE
11        | FTRACE_OPS_FL_IPMODIFY;
12
13     /* вызывать fh_ftrace_thunk только тогда когда rip == hook->address */
14     err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
15     if(err)
16     {
17         printk(KERN_DEBUG "ftrace_set_filter_ip() failed: %d\n", err);
18         return err;
19     }
20
21     /* регистрация перехвата */

```

```

22     err = register_ftrace_function(&hook->ops);
23     if(err)
24     {
25         printk(KERN_DEBUG "register_ftrace_function() failed: %d\n", err);
26         return err;
27     }
28
29     return 0;
30 }

```

При выгрузке модуля отключение перехватов происходит в обратном порядке (Листинг 2.13).

Листинг 2.13 — Отключение перехвата функции

```

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

```

2.3 Вывод

В данном разделе была рассмотрена общая архитектура приложения и методы перехвата функций с помощью таблицы системных вызовов и ftrace.

3 Технологическая часть

В данном разделе рассматривается выбор языка программирования и реализация программного обеспечения.

3.1 Выбор языка программирования

Операционная система Linux позволяет писать загружаемые модули ядра на Rust и на C. Для реализации загружаемого модуля был выбран последний, так как большая часть ядра и загружаемых моделей написана на языке C, а также у меня есть опыт разработки модулей на данном языке программирования.

3.2 Функции-обёртки перехватываемых системных вызовов

На листингах 3.1-3.4 представлены реализации функций-обёрток системных вызовов open, close, read, write соответственно.

Листинг 3.1 — Функция-обёртка системного вызова open

```
1  syscall_t orig_open;
2  asmlinkage int hook_open(const struct pt_regs *regs)
3  {
4      const char __user *filename = (char *)regs->di;
5      int flags = (int)regs->si;
6      umode_t mode = (umode_t)regs->dx;
7
8      char kernel_filename[NAME_MAX] = {0};
9
10     long error = strncpy_from_user(kernel_filename, filename, NAME_MAX);
11
12     int fd = orig_open(regs);
13
14     if (!error && current->real_parent->pid > 3)
15         printk(KERN_INFO KERNEL_MONITOR "Process %d; open: %s, flags: %x; mode: %x;
16             fd: %d\n", current->pid, kernel_filename, flags, mode, fd);
17
18     return fd;
19 }
```

Листинг 3.2 — Функция-обёртка системного вызова close

```
1  syscall_t orig_close;
2  asmlinkage int hook_close(const struct pt_regs *regs)
3  {
4      unsigned int fd = (unsigned int)regs->di;
5
6      /* Не логировать стандартный ввод/вывод, а так же системные процессы */
7      if (fd > 2 && current->real_parent->pid > 3)
```

```

8      {
9          printk(KERN_INFO KERNEL_MONITOR "Process %d; close fd: %d; filename: %s\n",
              current->pid, fd,
10             current->files->fdt->fd[fd]->f_path.dentry->d_iname);
11      }
12      return orig_close(regs);
13  }

```

Листинг 3.3 — Функция-обёртка системного вызова read

```

1  syscall_t orig_read;
2  asmlinkage int hook_read(const struct pt_regs *regs)
3  {
4      unsigned int fd = (unsigned int)regs->di;
5      char __user *buf = (char*)regs->si;
6      size_t count = (size_t)regs->dx;
7
8      /* Не логировать стандартный ввод/вывод, а так же системные процессы */
9      if (fd > 2 && current->real_parent->pid > 3)
10         printk(KERN_INFO KERNEL_MONITOR "Process %d; read fd: %d; buf: %p; count:
              %ld; filename: %s\n", current->pid, fd, buf, count,
11             current->files->fdt->fd[fd]->f_path.dentry->d_iname);
12     return orig_read(regs);
13 }

```

Листинг 3.4 — Функция-обёртка системного вызова write

```

1  syscall_t orig_write;
2  asmlinkage int hook_write(const struct pt_regs *regs)
3  {
4      unsigned int fd = (unsigned int)regs->di;
5      const char __user *buf = (const char*)regs->si;
6      size_t count = (size_t)regs->dx;
7
8      /* Не логировать стандартный ввод/вывод, а так же системные процессы */
9      if (fd > 2 && current->real_parent->pid > 3)
10         printk(KERN_INFO KERNEL_MONITOR "Process %d; write fd: %d; buf: %p; count:
              %ld; filename: %s\n", current->pid, fd, buf, count,
11             current->files->fdt->fd[fd]->f_path.dentry->d_iname);
12     return orig_write(regs);
13 }

```

3.3 Функции-обёртки перехватываемых ftrace функций

На листингах 3.5-3.8 представлены реализации функций-обёрток функций `random_read`, `do_filp_open`, `bdev_read_page`, `bdev_write_page` соответственно.

Листинг 3.5 — Функция-обёртка функции random_read

```

1 static asmlinkage ssize_t ( *orig_random_read)(struct file *file , char __user *buf,
    size_t nbytes , loff_t *ppos);
2 static asmlinkage ssize_t hook_random_read(struct file *file , char __user *buf,
    size_t nbytes , loff_t *ppos)
3 {
4     /* Вызов оригинального random_read() */
5     int bytes_read;
6     bytes_read = orig_random_read(file , buf, nbytes , ppos);
7     printk(KERN_INFO KERNEL_MONITOR "Process %d read %d bytes from /dev/random\n",
        current->pid , bytes_read);
8     return bytes_read;
9 }

```

Листинг 3.6 — Функция-обёртка функции do_filp_open

```

1 static asmlinkage struct file* ( *orig_do_filp_open)(int dfd , struct filename
    *pathname , const struct open_flags *op);
2 static asmlinkage struct file* hook_do_filp_open(int dfd , struct filename *pathname ,
    const struct open_flags *op)
3 {
4     if (current->real_parent->pid > 3)
5         printk(KERN_INFO KERNEL_MONITOR "Process %d; open %s;\n", current->pid ,
            pathname->name);
6
7     struct file* file;
8     file = orig_do_filp_open(dfd , pathname , op);
9
10    return file;
11 }

```

Листинг 3.7 — Функция-обёртка функции bdev_read_page

```

1 static asmlinkage int ( *orig_bdev_read_page)(struct block_device *bdev , sector_t
    sector , struct page *page);
2 static asmlinkage int hook_bdev_read_page(struct block_device *bdev , sector_t
    sector , struct page *page)
3 {
4     int err;
5     err = orig_bdev_read_page(bdev , sector , page);
6     printk(KERN_INFO KERNEL_MONITOR "Process %d bdev_read_page; dev: %d\n",
        current->pid , bdev->bd_dev);
7     return err;
8 }

```

Листинг 3.8 — Функция-обёртка функции bdev_write_page

```

1 static asmlinkage int ( *orig_bdev_write_page)(struct block_device *bdev, sector_t
    sector, struct page *page, struct writeback_control *wbc);
2 static asmlinkage int hook_bdev_write_page(struct block_device *bdev, sector_t
    sector, struct page *page, struct writeback_control *wbc)
3 {
4     int err;
5     err = orig_bdev_write_page(bdev, sector, page, wbc);
6     printk(KERN_INFO KERNEL_MONITOR "Process %d bdev_write_page; dev: %d\n",
        current->pid, bdev->bd_dev);
7     return err;
8 }

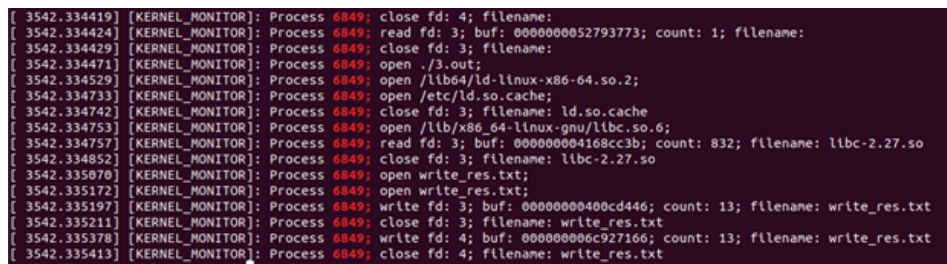
```

3.4 Примеры работы

TODO обновить скрин.

TODO добавить прога включилась/выключилась прога открыла 1 файл считала 10 байт. прога открыла 1 файл, открыла 2ой, закрыла 1ый, закрыла 2ой. 1.out 2.out 3.out

На рисунке 3.1 представлен пример собранных логов в /var/log/syslog.



```

[3542.334419] [KERNEL_MONITOR]: Process 6849; close fd: 4; filename:
[3542.334424] [KERNEL_MONITOR]: Process 6849; read fd: 3; buf: 0000000052793773; count: 1; filename:
[3542.334429] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename:
[3542.334471] [KERNEL_MONITOR]: Process 6849; open ./3.out;
[3542.334529] [KERNEL_MONITOR]: Process 6849; open /lib64/ld-linux-x86-64.so.2;
[3542.334733] [KERNEL_MONITOR]: Process 6849; open /etc/ld.so.cache;
[3542.334742] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename: ld.so.cache
[3542.334753] [KERNEL_MONITOR]: Process 6849; open /lib/x86_64-linux-gnu/libc.so.6;
[3542.334757] [KERNEL_MONITOR]: Process 6849; read fd: 3; buf: 000000004168cc3b; count: 832; filename: libc-2.27.so
[3542.334852] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename: libc-2.27.so
[3542.335070] [KERNEL_MONITOR]: Process 6849; open write_res.txt;
[3542.335172] [KERNEL_MONITOR]: Process 6849; open write_res.txt;
[3542.335197] [KERNEL_MONITOR]: Process 6849; write fd: 3; buf: 00000000400cd446; count: 13; filename: write_res.txt
[3542.335211] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename: write_res.txt
[3542.335378] [KERNEL_MONITOR]: Process 6849; write fd: 4; buf: 000000006c927166; count: 13; filename: write_res.txt
[3542.335413] [KERNEL_MONITOR]: Process 6849; close fd: 4; filename: write_res.txt

```

Рисунок 3.1 — Пример работы загружаемого модуля ядра.

3.5 Вывод

В данном разделе был обоснован выбор языка программирования, рассмотрены листинги реализованных функций. Приведены результаты работы ПО.

Заключение

В данной работе был реализован загружаемый модуль ядра операционной системы Linux. В процессе разработки был реализован подход, позволяющий перехватить необходимые функции и системные вызовы, и логировать необходимую информацию без перехода в режим пользователя. Реализуемый модуль поддерживает ядра версий 5.0 для архитектуры x86_64

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Chris Wright Crispin Cowan, James Morris. Linux Security Module Framework / James Morris Chris Wright, Crispin Cowan. — 2002.
2. Linux Syscall Reference (64 bit). // [Электронный ресурс]. Режим доступа: <https://syscalls64.paolostivanin.com/>, (дата обращения: 26.10.2021).
3. Jim Keniston Prasanna S Panchamukhi, Masami Hiramatsu. Linux Tracing Technologies. Kernel Probes (Kprobes). // [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/kprobes.html>, (дата обращения: 26.10.2021).
4. Desnoyers, Mathieu. Linux Tracing Technologies. Using the Linux Kernel Tracepoints. // [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/tracepoints.html>, (дата обращения: 26.10.2021).
5. Turpitka, Dennis. Splice Hooking for Unix-Like Systems. — 2018. // [Электронный ресурс]. Режим доступа: <https://www.linux.com/training-tutorials/splice-hooking-unix-systems/>, (дата обращения: 26.10.2021).
6. Rostedt, Steven. Linux Tracing Technologies. Ftrace – Function Tracer. — 2008. // [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/ftrace.html>, (дата обращения: 26.10.2021).
7. Jones, M. Anatomy of the Linux kernel. — 2007. // [Электронный ресурс]. Режим доступа: <https://developer.ibm.com/articles/l-linux-kernel/>, (дата обращения: 26.10.2021).
8. DevelopersWorks, IBM. Kernel Space Definition. — 2005. // [Электронный ресурс]. Режим доступа: http://www.linfo.org/kernel_space.html, (дата обращения: 26.10.2021).
9. DevelopersWorks, IBM. User Space Definition. — 2005. // [Электронный ресурс]. Режим доступа: http://www.linfo.org/user_space.html, (дата обращения: 26.10.2021).
10. Elixir.bootlin. struct pt_regs. // [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/v5.0/source/arch/x86/include/asm/ptrace.h#L12>, (дата обращения: 26.11.2021).
11. AArch32 (compat) system call definitions. // [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/v5.0/source/arch/arm64/include/asm/unistd32.h#L34>, (дата обращения: 26.11.2021).