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 _____ М.А. Козлов
(Подпись, дата) (И.О. Фамилия)

Руководитель курсового проекта _____ Н. Ю. Рязанова
(Подпись, дата) (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 курсовой работы

по дисциплине Операционные системы

Студент группы ИУ7-75Б

Козлов Максим Антонович
(Фамилия, имя, отчество)

Тема курсовой работы Мониторинг системных вызовов в ОС Linux.

Направленность КР (учебная, исследовательская, практическая, производственная)

учебная

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 курсовой работы:
Расчетно-пояснительная записка на 25-30 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
Расчётно-пояснительная записка должна содержать введение, аналитический раздел, конструкторский раздел, технологический раздел, экспериментально-исследовательский раздел, заключение, список литературы, приложения.

Дата выдачи задания « ___ » _____ 2021г.

Руководитель курсовой работы _____ Н. Ю. Рязанова
(Подпись, дата) (И.О.Фамилия)

Студент _____ М. А. Козлов
(Подпись, дата) (И.О.Фамилия)

СОДЕРЖАНИЕ

Введение

Одна из актуальных задач мониторинга операционных систем является мониторинг системных вызовов для выявления узких мест и исследования особенностей её работы. В операционных системах семейства Linux это можно реализовать с помощью загружаемых модулей ядра, используя различные методы перехвата системных вызовов, которые будут описаны в аналитическом разделе.

Целью 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.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра, позволяющий перехватывать системные вызовы: bdev_read_page, bdev_write_page, sys_read, sys_write, sys_open, sys_close, random_read и выводить в системный журнал информацию о идентификаторе вызывающего процесса, передаваемых параметрах и возвращаемого значения.

Для достижения поставленной цели требуется решить следующие задачи:

1) проанализировать перехватываемые функции;
2) проанализировать существующие способы перехвата функций;
3) реализовать загружаемый модуль ядра;
4) исследовать поведение перехваченных функций.

1.2 Анализ перехватываемых системных вызовов

Перехватываемые системные функции позволяют отслеживать работу системы относительно основных операций по работе с файловыми системами, а также связанных с символьными и блочными устройствами.

Сигнатуры данных функций зависят от версии ядра Linux, поэтому будет реализована поддержка только версии ядра 5.0. Все дальнейшие структуры и функции ядра приведены для данной версии ядра, если явно не указана иная.

Анализ системного вызова bdev_read_page

Функция bdev_read_page имеет следующую сигнатуру [1]:

int bdev_read_page(struct block_device *bdev, sector_t sector, struct page *page),

где struct block_device *bdev – блочное устройство с которого считывается информация,

sector_t sector – смещение на устройстве для чтения страницы,

struct page *page – страница для записи считанных данных.

Данная функция начинает операцию чтения страницы с блочного устройства, блокируя страницу на время выполнения. Ошибки возвращаемые этой функцией (отрицательный errno), обычно «мягкие», т.е. существуют альтернативные способы чтения данных с устройства и не требуется распространять ошибку вверх по стеку вызовов.

Анализ системного вызова bdev_write_page

Функция bdev_write_page имеет следующую сигнатуру [2]:

int bdev_write_page(struct block_device *bdev, sector_t sector, struct page *page, struct writeback_control *wbc),

где struct block_device *bdev – блочное устройство для записи страницы,

sector_t sector – смещение на устройстве на которое будет записана страница,

struct page *page – страница с записываемыми данными,

struct writeback_control * wb – управляющая структура, которая сообщает коду обратной записи, как производиться запись.

Функция начинает операцию записи страницы на блочное устройство, блокируя страницу на время выполнения. Обработка ошибок, возвращаемых данной функцией аналогична bdev_read_page.

Анализ системного вызова sys_open

Системный вызов sys_open открывает файл по переданному символьному имени filename. Данная функция имеет следующую сигнатуру [3]:

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode).

Возвращаемое значение open() – это дескриптор файла, неотрицательное целое число, которое является индексом записи в таблица дескрипторов открытых файлов. Используя различные флаги можно изменить поведение open(). Например, если файл не был найден и указан флаг O_CREAT, то будет создан новый файл с указанным именем.

Анализ системного вызова sys_read

Системный вызов sys_read считывает из файла, на который ссылается файловый дескриптор fd, count байт и записывает их в buf – буфер из пространства пользователя. Данная функция имеет следующую сигнатуру [4]:

SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count).

Операция чтения начинается со смещения указанного в struct file, после чего смещение файла увеличивается на число прочитанных байтов. Если смещение файла находится в конце или больше конца файла, байты не читаются, а read возвращает ноль.

Анализ системного вызова sys_write

Системный вызов sys_write записывает в файл, на который ссылается файловый дескриптор fd, count байтов из буфера buf – буфера пространства пользователя. Данная функция имеет следующую сигнатуру [5]:

SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count).

Операция записи происходит в файл с указанного смещение, а смещение файла увеличивается на количество записанных байтов. Если файл был открыт с флагом O_APPEND, смещение файла сначала устанавливается на конец файла перед записью. Регулировка смещения файла и операция записи выполняется как атомарный шаг.

Анализ системного вызова sys_ close

Системный вызов sys_close закрывает файл, на который ссылается файловый дескриптор fd, освобождая ресурсы. Данная функция имеет следующую сигнатуру [6]: SYSCALL_DEFINE1(close, unsigned int, fd).

Анализ системного вызова random_ read

Функция random_read позволяет считывать массив байт из символьного устройства /dev/random и имеет следующую сигнатуру [7]: static ssize_t random_read(struct file *file , char __user *buf, size_t nbytes, loff_t *ppos).

Символьное устройство /dev/random генерирует случайные последовательности байт, используя в качестве источника энтропии CSPRNG (криптографически безопасного генератора псевдослучайных чисел). Существует похожая функция static ssize_t urandom_read(struct file *file , char __user *buf, size_t nbytes, loff_t *ppos), считывающая данные из символьного устройства /dev/urandom которое отличающается от /dev/random лишь тем, что когда энтропия останавливается, он продолжает генерировать последовательности байт.

1.3 Анализ способов перехвата системных вызовов

Linux Security API

Linux Security API – это специальный интерфейс, позволяющий трассировать ядро линукса начиная с версии 2.6 [8]. 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.

Модификация таблицы системных вызовов

В Linux все обработчики системных вызовов хранятся в таблице sys_call_table [9]. Подмена значений в этой таблице приводит к смене поведения всей системы. Таким образом,

сохранив старое 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 выгрузки модуля. Помимо полезной нагрузки мониторинга, единственным 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. 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 реализовались на ассемблере. Соответственно, подобные обработчики порой сложно, а иногда и вовсе невозможно заменить на свои, написанные на Си [10]. 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.

Kprobes

Kprobes – специализированное API, в первую очередь предназначенного для отладки и трассирования ядра [11]. 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 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. 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 (начиная с версия 3.19).

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.

Kernel tracepoints

Kernel tracepoints – это фреймворк для трассировки ядра, сделанный через статическое инструментирование кода [12]. Точка трассировки, помещенная в код, обеспечивает ловушку для вызова функции (зонда), которую можно предоставить во время выполнения. Точка трасси-

ровки 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 раместить 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.

Недостатками данного подхода являются:

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 остановки в ядровые функции необходимо перекомпилировать ядро, если для данных функций не данные точки не определенны.

3) Относительно плохо задокументированное API.

Сплайсинг

Сплайсинг – способ перехвата функций, заключающийся в замене инструкций в начале функции на безусловный переход, ведущий в обработчик [13]. 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.

Недостатками данного подхода являются:

1) Требования к надёжному метод дизассемблерования кода ядра.

2) Зависимость 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 целыми;

д) проверка на отсутствие переходов внутрь заменяемого куска;

е) проверка на возможность переместить заменяемый кусок в другое место;

Ftrace

Ftrace – это фреймворк для трассирования ядра на уровне функций [14]. 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 nop – 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.

Достоинствами ftrace являются:

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 дизассемблерования 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.

Недостатками данного подхода являются:

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) Оборачивание 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.

1.4 Сравнительный анализ методов трассировки ядра

В таблице 1.1 приведено сравнение рассмотренных методов перехвата системных вызовов.

Анализируя данную таблицу можно сделать вывод, что для решения поставленной задачи наиболее подходящими являются методы модификации таблицы системных функций и ftrace, т.к. они не требуют перекомпиляции ядра и поддерживаются большинством версий ядер, а также не вызывают большой технической сложности в реализации.

1.5 Выводы

В результате анализа работы операционной системы Linux с файловыми системами были определены необходимые для мониторинга функции, которые позволят исследовать особенности работы ядра с файлами.

В результате сравнительного анализа методов перехвата было выбрано два метода. Первый метод – метод модификации таблицы системных вызовов sys_call_table позволит перехватывать вызовы системных функции sys_read, sys_write, sys_open и sys_close без больших накладных расходов. Второй – используя библиотеку ftrace, которая позволит перехватывать

Таблица 1.1 — Сравнение существующих методов трассировки ядра.

Критерий	Linux Security	Модификация syscall table	Kprobes	Kernel tracepoints	Сплайсинг	Ftrace
Накладные расходы	Средние	Мини-мальные	Большие	Малые	Мини-мальные	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
Докумен-тация	Средняя	Средняя	Средняя	Малая	Малая	Большая
Требования к ядру	Версия старше 2.6	Нет	Версия старше 2.0	-	Нет	Наличие ftrace

функции bdev_read_page, bdev_write_page и random_read, т.к. они не определены в таблице sys_call_table.

Выбранные методы перехвата требуют, чтобы сигнатуры перехватываемой функций и функции-перехватчика 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.

2 Конструкторский раздел

2.1 IDEF0

На рисунках 2.1 - 2.2 показаны нулевой и первый уровень диаграммы IDEF0, показывающие процесс мониторинга системных вызовов.

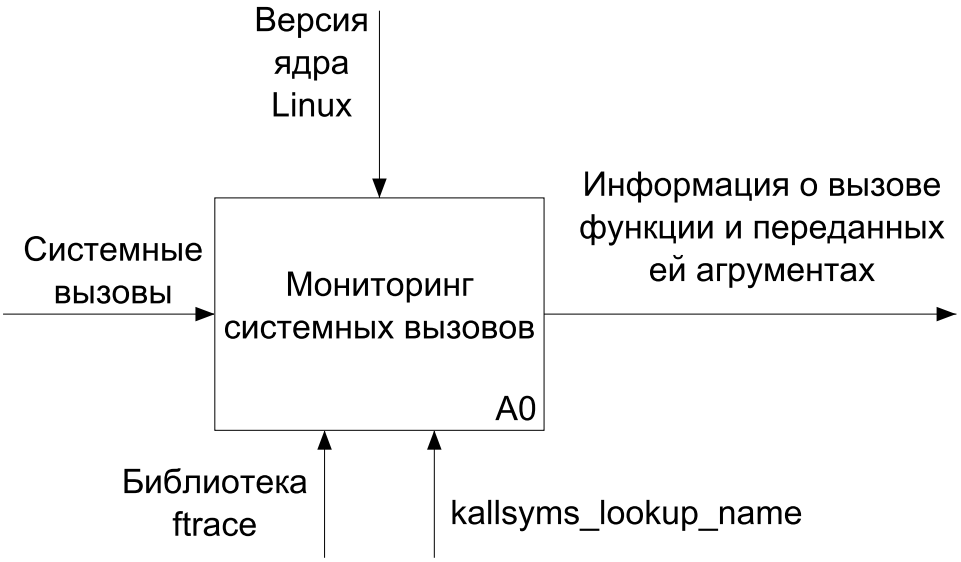

Рисунок 2.1 — IDEF0 нулевого уровня.

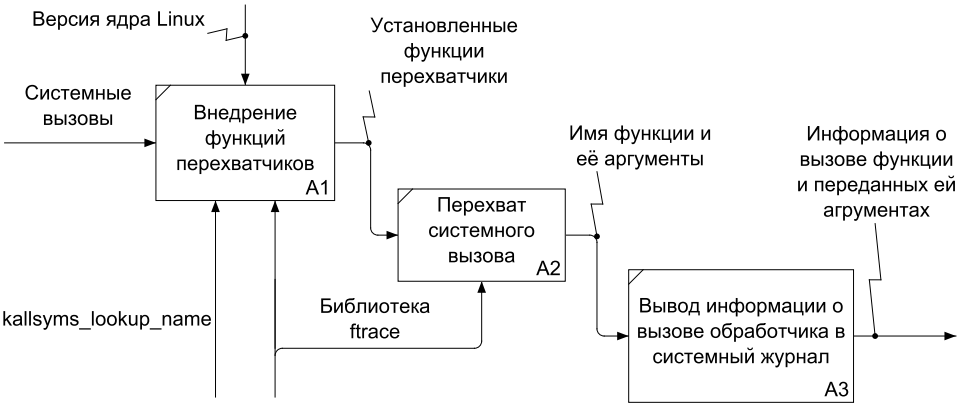

Рисунок 2.2 — IDEF0 первого уровня.

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 в системный журнал (/var/log/syslog).

2.2 Используемые структуры данных

В новых версиях ядра прототип обработчика системного вызова описывается следующим образом (листинг 2.2):

Прототип обработчиков системных вызовов

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

где struct pt_regs – структура, описывающая регистры процессора, которая может отличаться для разных версий ядра и процессоров. В документации ядра Linux описано назначения регистров для каждого конкретного системного вызова [9]. Одно из определений struct pt_regs представлено в листинге 2.2 [15].

Структура регистров

```
struct pt_regs {
    /*
         * NB: 32-bit x86 CPUs are inconsistent as what happens in the
         * following cases (where %seg represents a segment register):
         *
         * - pushl %seg: some do a 16-bit write and leave the high
         *   bits alone
         * - movl %seg, [mem]: some do a 16-bit write despite the movl
         * - IDT entry: some (e.g. 486) will leave the high bits of CS
         *   and (if applicable) SS undefined.
         *
         * Fortunately, x86-32 doesn't read the high bits on POP or IRET,
         * so we can just treat all of the segment registers as 16-bit
         * values.
         */
    unsigned long bx;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    unsigned long bp;
    unsigned long ax;
    unsigned short ds;
    unsigned short __dsh;
    unsigned short es;
    unsigned short __esh;
    unsigned short fs;
    unsigned short __fsh;
    /* On interrupt, gs and __gsh store the vector number. */
    unsigned short gs;
    unsigned short __gsh;
    /* On interrupt, this is the error code. */
    unsigned long orig_ax;
    unsigned long ip;
    unsigned short cs;
    unsigned short __csh;
    unsigned long flags;
    unsigned long sp;
    unsigned short ss;
    unsigned short __ssh;
```

```
41 };
```

Ключевой структурой, используемой ftrace для установки перехвата функции, является struct ftrace_ops, которая описывает каждую перехватываемую функцию (листинг 2.2) [16]. Обязательной для заполнения является поле func – адрес функции обратного вызова, которая будет вызываться в самом начале перехватываемой функции. Для решения проблемы рекурсии (перехватываемая функция вызывает коллбэк-функцию, которая опять вызывает перехватываемую) используют вспомогательную структуру struct ftrace_hook (листинг 2.2), в которой описывается имя перехватываемой функции, адрес функции-перехватчика, адрес перехватываемой функции и struct ftrace_ops.

struct ftrace_ops

```
1  typedef void ( *ftrace_func_t)(unsigned long ip, unsigned long parent_ip,
2          struct ftrace_ops *op, struct pt_regs *regs);
3
4  struct ftrace_ops {
5      ftrace_func_t           func;
6      struct ftrace_ops __rcu     *next;
7      unsigned long           flags;
8      void                *private;
9      ftrace_func_t           saved_func;
10 #ifdef CONFIG_DYNAMIC_FTRACE
11     struct ftrace_ops_hash      local_hash;
12     struct ftrace_ops_hash      *func_hash;
13     struct ftrace_ops_hash      old_hash;
14     unsigned long           trampoline;
15     unsigned long           trampoline_size;
16     struct list_head        list;
17 #endif
18 };
```

struct ftrace_hook

```
1  struct ftrace_hook {
2      const char *name;
3      void *function;
4      void *original;
5
6      unsigned long address;
7      struct ftrace_ops ops;
8  };
```

2.3 Алгоритм внедрения функции-перехватчика в таблицу системных вызовов

Алгоритм встраивания функций-перехватчиков в таблицу системных вызовов представлен на рисунке 2.3.

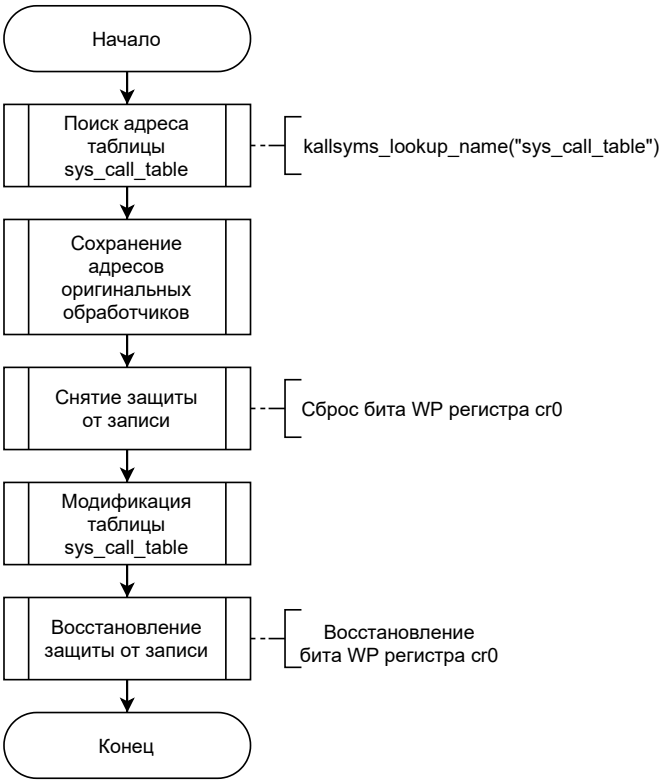

Рисунок 2.3 — Схема алгоритма встраивания функций-перехватчиков в таблицу системных вызовов.

Для поиска адреса системной таблицы используется функция kallsyms_lookup_name, которая позволяет найти абсолютный адрес любого экспортируемого символа ядра. Номера системных вызовов описаны в исходном коде линукса [17]. Зная их и начальный адрес таблицы, можно получить и запомнить абсолютные адреса оригинальных системных вызовов. Однако таблица системных вызовов находиться в области памяти доступной только на чтение, поэтому на время изменения адрес обработчика системного вызова требуется отключить глобальную защиту страниц от записи, изменением флага WP (Write Protection) в регистре CR0.

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

2.4 Алгоритм перехвата функций с использованием библиотеки ftrace

Алгоритм встраивания функций-перехватчиков в функции ядра, используя ftrace, представлен на рисунке 2.4.

Для перехвата функции ядра с помощью ftrace необходимо сначала найти и сохранить её адрес. Аналогично поиску адреса системной таблицы для этого можно использовать функ-

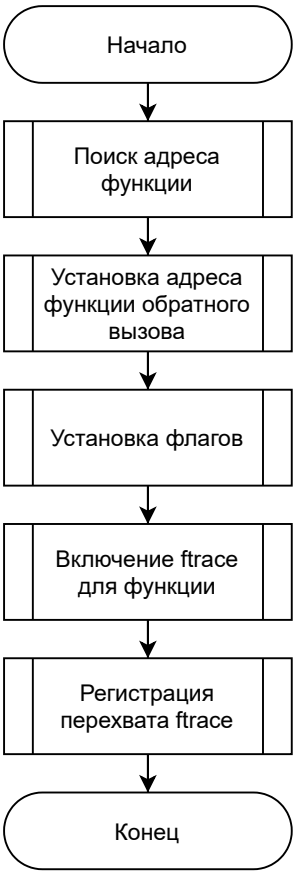

Рисунок 2.4 — Схема алгоритма встраивания функций-перехватчиков, используя ftrace.

цию kallsyms_lookup_name. После чего устанавливается функция обратного вызова и необходимые флаги ftrace, включается обработка ftrace при вызове перехватываемой функции и регистрируется перехватчик.

Отключение перехвата происходит в обратном порядке: дерегистрация перехвата ftrace, потом отключение ftrace для функции.

Для решения проблемы рекурсии используется алгоритм представленный на рисунке 2.5. В начале каждой функции ядра, для которой включён ftrace, находиться вызов функции __fentry__(), который вызывает функцию защиты от рекурсии, анализирующая значение parent_ip – адрес вызывающей стороны, на основании которого принимается решение о необходимости вызова функции перехватчика. Однако для корректного изменения регистра ip необходимо установить соответствующие флаги при регистрации обработчика:

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.

Алгоритм работы всех функций-перехватчиков данных системных вызовов одинаков: логируется вызовы конкретного обработчика и передаваемые ему аргументы, после чего вызы-

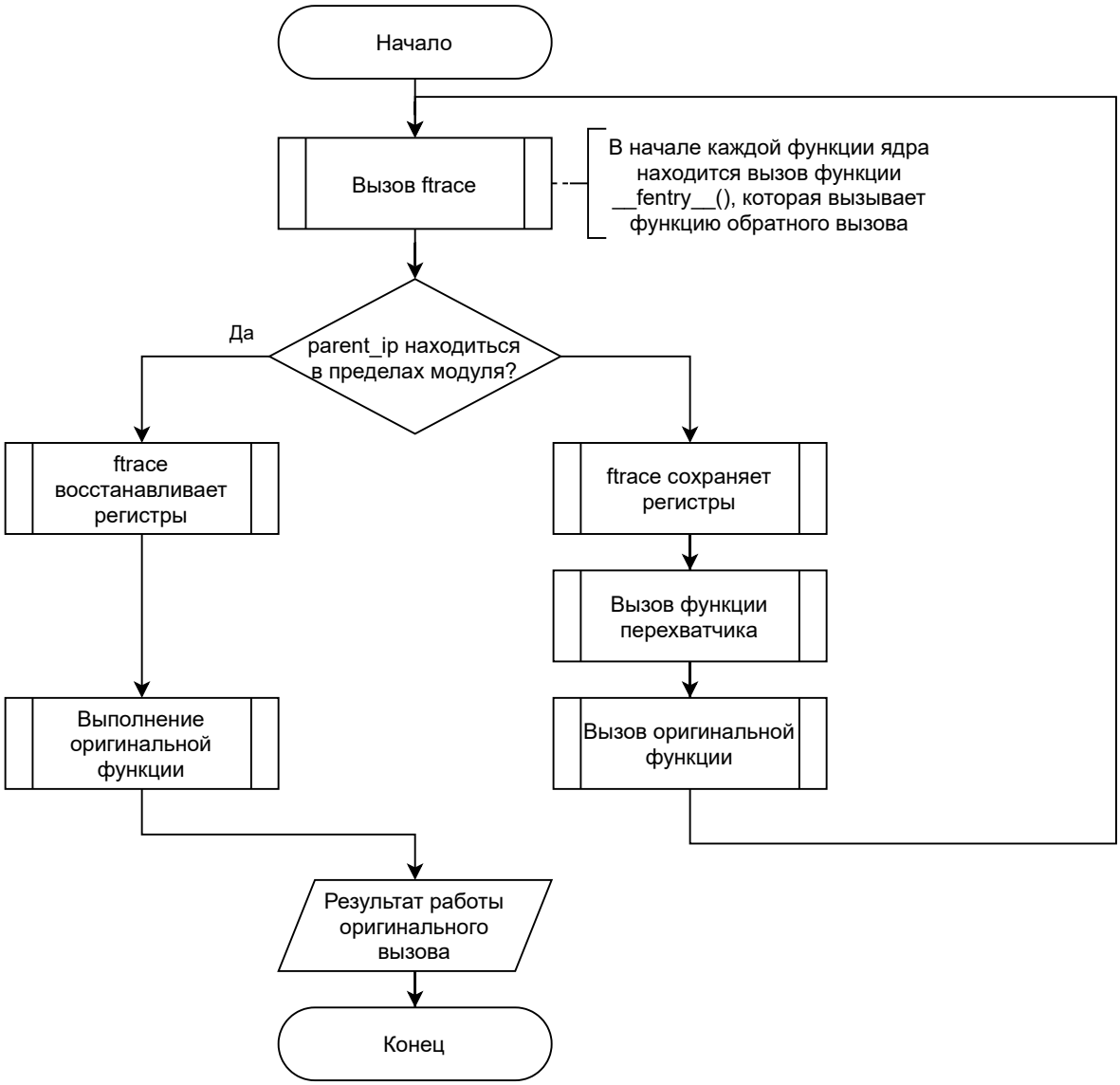

Рисунок 2.5 — Схема алгоритма защиты от рекурсии.

вается оригинальная функции, результат которой возвращается вызывающей стороне. Схема алгоритма функции-перехватчика представлена на рисунке 2.6.

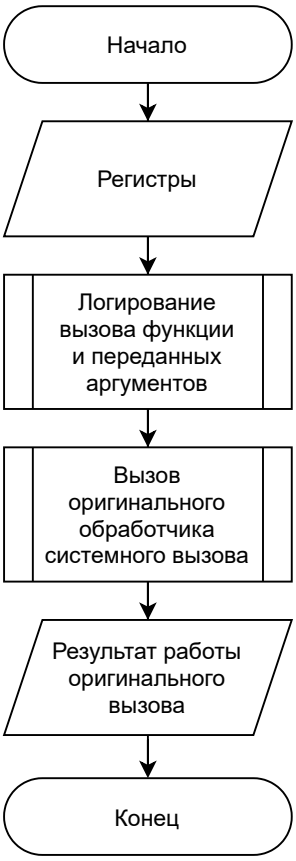

Рисунок 2.6 — Схема алгоритма работы «перехватчика».

3 Технологический раздел

3.1 Выбор языка программирования и среды 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 Linux и загружаемых моделей написана на языке C, а также у меня есть опыт разработки модулей на данном языке программирования. Для сборки загружаемого модуля была выбрана утилита make.

В качестве среды разработки была выбрана кроссплатформенная программа Visual Studio Code, разрабатываемая компанией Microsoft. Visual Studio Code содержит редактор кода, отладчик, средства для статического анализа кода и средства для сборки проекта.

3.2 Реализация загружаемого модуля ядра

На листинге А.1 представлен Makefile загружаемого модуля ядра. Кроме непосредственной сборки модуля Makefile содержит цели clean и test, которые позволяют очистить директорию от файлов сборки и протестировать работу модуля программой указанной в переменной program.

Как было показано в разделе 2.4 для корректного внедрения функций-перехватчиков с использованием ftrace, был реализован алгоритм защиты от рекурсии, представленный на листинге 3.1.

Листинг 3.1 — Защита ftrace от рекурсии.

```
1 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
    struct ftrace_ops *ops, struct pt_regs *regs)
2 {
3     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
4     /* проверка адреса возврата функции */
5     if(!within_module(parent_ip, THIS_MODULE))
6         regs->ip = (unsigned long) hook->function; // вызов функции-перехватчика
7 }
```

На листингах 3.2 и 3.3 представлены функции установки и отключения функций перехватчиков с использованием библиотеки ftrace.

Листинг 3.2 — 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  }
```

Листинг 3.3 — 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  }
```

Встроенная в Linux функция write_cr0() не позволяет изменять бит WP, поэтому была реализована своя функция, представленная на листинге 3.4.

Листинг 3.4 — Функция изменения значения регистра cr0

```
1   extern unsigned long __force_order;
2   inline void cr0_write(unsigned long cr0)
```

```
{
// mov cr0, rax
asm volatile("mov %0, %%cr0" : "+r"(cr0), "+m"(__force_order));
}
```

На листинге 3.5 представлена часть кода, внедряющая функции перехватчики в таблицу системных вызовов.

Листинг 3.5 — Внедрение функций перехватчиков в таблицу системных вызовов

```
#define CR0_WP 0x00010000

/* Поиск начального адреса таблицы системных вызовов */
__sys_call_table = kallsyms_lookup_name("sys_call_table");
if (!__sys_call_table)
    return -1;

/* Получение адресов оригинальных системных вызовов */
orig_open   = (syscall_t)__sys_call_table[__NR_open];
orig_close  = (syscall_t)__sys_call_table[__NR_close];
orig_read   = (syscall_t)__sys_call_table[__NR_read];
orig_write  = (syscall_t)__sys_call_table[__NR_write];

/* Снятие защиту от записи */
unsigned long cr0 = read_cr0();
cr0_write(cr0 & ~CR0_WP);

/* Замена системных функций hooks*/
// __sys_call_table[__NR_mkdir]    = (unsigned long)hook_mkdir;

__sys_call_table[__NR_open]     = (unsigned long)hook_open;
__sys_call_table[__NR_close]    = (unsigned long)hook_close;
__sys_call_table[__NR_read]     = (unsigned long)hook_read;
__sys_call_table[__NR_write]    = (unsigned long)hook_write;

/* Восстановление защиты от записи */
cr0 = read_cr0();
cr0_write(cr0 | CR0_WP);
```

На листингах 3.6-3.13 представлены реализации алгоритма из рисунка 2.6 для каждой функции перехватчика системных вызовов: open, close, read, write через таблицу системных вызовов, a random_read, do_filp_open, bdev_read_page, bdev_write_page через ftrace соответственно.

Листинг 3.6 — Функция-обёртка системного вызова open

```
syscall_t orig_open;
asmlinkage int hook_open(const struct pt_regs *regs)
{
```

```
    const char __user *filename = (char *)regs->di;
    int flags = (int)regs->si;
    umode_t mode = (umode_t)regs->dx;

    char kernel_filename[NAME_MAX] = {0};

    long error = strncpy_from_user(kernel_filename, filename, NAME_MAX);

    int fd = orig_open(regs);

    if (!error && current->real_parent->pid > 3)
        printk(KERN_INFO KERNEL_MONITOR "Process %d; open: %s, flags: %x; mode: %x;
            fd: %d\n", current->pid, kernel_filename, flags, mode, fd);

    return fd;
}
```

Листинг 3.7 — Функция-обёртка системного вызова close

```
syscall_t orig_close;
asmlinkage int hook_close(const struct pt_regs *regs)
{
    unsigned int fd = (unsigned int)regs->di;

    /* Не логировать стандартный ввод/вывод, а так же системные процессы */
    if (fd > 2 && current->real_parent->pid > 3)
    {
        printk(KERN_INFO KERNEL_MONITOR "Process %d; close fd: %d; filename: %s\n",
            current->pid, fd,
             current->files->fdt->fd[fd]->f_path.dentry->d_iname);
    }
    return orig_close(regs);
}
```

Листинг 3.8 — Функция-обёртка системного вызова read

```
syscall_t orig_read;
asmlinkage int hook_read(const struct pt_regs *regs)
{
    unsigned int fd = (unsigned int)regs->di;
    char __user *buf = (char*)regs->si;
    size_t count = (size_t)regs->dx;

    /* Не логировать стандартный ввод/вывод, а так же системные процессы */
    if (fd > 2 && current->real_parent->pid > 3)
        printk(KERN_INFO KERNEL_MONITOR "Process %d; read fd: %d; buf: %p; count:
            %ld; filename: %s\n", current->pid, fd, buf, count,
             current->files->fdt->fd[fd]->f_path.dentry->d_iname);
```

```
12      return orig_read(regs);
13  }
```

Листинг 3.9 — 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  }
```

Листинг 3.10 — 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 3.11 — 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 3.12 — 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 3.13 — 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  }
```

4 Исследовательский раздел

4.1 Технические характеристики системы

Исследование поведения системных функций проводилось на ноутбуке с процессором Intel(R) Core(TM) i5-7200U CPU 2.50 GHz в виртуальной машине VmWare с 8 гб оперативной памяти под управлением операционной системы Linux (дистрибутив Ubuntu 18.04 x86-64, ядро версии 5.0).

4.2 Результаты мониторинга процессов

1) Программа никаких действий не выполняет (листинг 4.1)

Листинг 4.1 — Программа не выполняющая никаких действий

```
1 int main()
2 {
3     return 0;
4 }
```

Система запускает процесс, который ничего не выполняет. На рисунке 4.1 представлен результат мониторинга системных вызовов выполняемых в ходе работы программы.

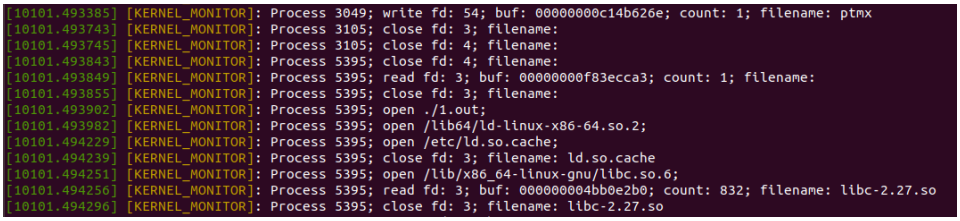

Рисунок 4.1 — Результат мониторинга программы не выполняющей никаких действий.

Из рисунка 4.1 видно, что процессы с идентификаторами 3105 (терминал из которого производился запуск программы) и 5395 (новый процесс созданный под выполнение программы) производят некоторые действия перед открытием исполняемого файла 1.out, одно из которых вызовов системного вызова fork для создания процесса программы. К сожалению, у некоторых файлов не указывается имя, поэтому сложно сказать, какие именно действия производятся После открытия исполняемого файла происходит загрузка библиотек необходимых для работы программы и выполнение кода программы.

Можно заметить, что не была вызвана функция-обёртка системного вызова sys_open, а только do_filp_open, что может быть связано с ассемблерной оптимизацией данного обработчика системного вызова, описанной в аналитическом разделе.

2) Запускается та же программа (листинг 4.1), но в загружаемый модуль ядра для дополнительного логирования, было добавлено логирование системного вызова get_unused_fd_flags, используя ftrace.

Из рисунка 4.2 видно, что файл с исходным кодом открывается с помощью функции do_filp_open, а не sys_open, что объясняет отсутствие в лог файле записи о его чтении и закрытии.

```
maxim@ubuntu:~/Desktop/cp_os$ dmesg | grep 5159\;
[ 2232.189289] [KERNEL_MONITOR]: Process 5159; close fd: 4; filename:
[ 2232.189297] [KERNEL_MONITOR]: Process 5159; close fd: 3; filename:
[ 2232.189339] [KERNEL_MONITOR]: Process 5159; open ./1.out;
[ 2232.189408] [KERNEL_MONITOR]: Process 5159; open /lib64/ld-linux-x86-64.so.2;
[ 2232.189598] [KERNEL_MONITOR]: Process 5159; get_unused_fd 3;
[ 2232.189599] [KERNEL_MONITOR]: Process 5159; open /etc/ld.so.cache;
[ 2232.189606] [KERNEL_MONITOR]: Process 5159; close fd: 3; filename: ld.so.cache
[ 2232.189616] [KERNEL_MONITOR]: Process 5159; get_unused_fd 3;
[ 2232.189616] [KERNEL_MONITOR]: Process 5159; open /lib/x86_64-linux-gnu/libc.so.6;
[ 2232.189651] [KERNEL_MONITOR]: Process 5159; close fd: 3; filename: libc-2.27.so
```

Рисунок 4.2 — Результат мониторинга программы не выполняющей никаких действий с дополнительной информацией об открываемом файловом дескрипторе.

3) Программа открывающая и закрывающая файлы (листинг 4.2).

Листинг 4.2 — Программа открывающая и закрывающая файлы

```
#include <fcntl.h>

int main()
{
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("test.txt", O_RDONLY);

    close(fd1);
    close(fd2);
    return 0;
}
```

Система запускает процесс, который открывает существующие файлы alphabet.txt и test.txt, после чего закрывает файлы в порядке их открытия.

На рисунке 4.3 представлен результат мониторинга системных вызовов, выполняемых в ходе работы данной программы.

Первые 12 строк инициализации процесса были рассмотрены выше. Из последних четырёх можно сделать вывод, что open возвращает наименьший свободный файловый дескриптор и первые три файловых дескрипторов изначально заняты под stdin, stdout, stderr, а также корректность определения имени файла по его файловому дескриптору.

4) Программа с небуферизованным вводом данных (листинг 4.3).

Листинг 4.3 — Программа с небуфферизованным вводом данных из файла

```
#include <fcntl.h>
int main()
{
    int fd = open("alphabet.txt", O_RDONLY);
    char buf[128];
```

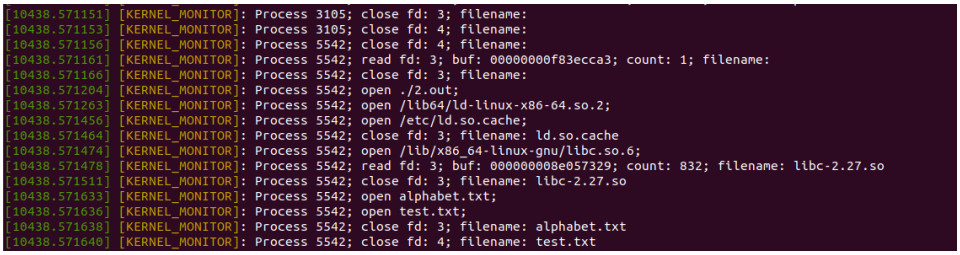

Рисунок 4.3 — Результат мониторинга программы открывающей и закрывающей файлы.

```
6
7       int len = read(fd, buf, 128);
8       buf[len] = 0;
9       write(1, buf, len);
10
11      close(fd);
12      return 0;
13  }
```

Система запускает процесс, который открывает существующий файл alphabet.txt читает информацию из него, используя системный вызов read, после чего файл закрывается.

На рисунке 4.4 представлен результат мониторинга системных вызовов выполняемых в ходе работы данной программы. Из рисунка 4.4 видно, что небуферизованный ввод использует ровно один вызов функция sys_read, который пытается прочитать из файла 128 байт из открытого файла.

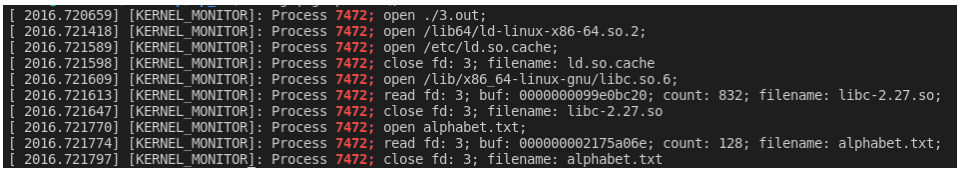

Рисунок 4.4 — Результат мониторинга программы с небуферизованным вводом данных.

5) Программа с буферизованным вводом данных (листинг 4.4).

Листинг 4.4 — Программа с буфферизованным вводом данных из файла

```
1  #include <stdio.h>
2
3  int main()
4  {
5      FILE *f = fopen("alphabet.txt", "r");
6      char buf[128];
7
8      fscanf(f, "%s", buf);
9      printf("%s", buf);
10
11     fclose(f);
```

```
12      return 0;
13  }
```

Система запускает процесс, который открывает существующий файл alphabet.txt и читает информацию из него, используя для этого функцию fscanf библиотеки буферизованного ввода/вывода stdio, после чего файл закрывается.

На рисунке 4.5 представлены результаты мониторинга системных вызовов выполняемых в ходе работы данной программы.

Из рисунка 4.5 видно, что программа с буферизованным вводом дважды вызывает системный вызов sys_read для заполнения буфера размером 4096 байт.

```
[11016.476689] [KERNEL_MONITOR]: Process 3105; close fd: 3; filename:
[11016.476691] [KERNEL_MONITOR]: Process 3105; close fd: 4; filename:
[11016.476769] [KERNEL_MONITOR]: Process 5683; close fd: 4; filename:
[11016.476775] [KERNEL_MONITOR]: Process 5683; read fd: 3; buf: 00000000f83ecca3; count: 1; filename:
[11016.476780] [KERNEL_MONITOR]: Process 5683; close fd: 3; filename:
[11016.476828] [KERNEL_MONITOR]: Process 5683; open ./3.out;
[11016.476929] [KERNEL_MONITOR]: Process 5683; open /lib64/ld-linux-x86-64.so.2;
[11016.477145] [KERNEL_MONITOR]: Process 5683; open /etc/ld.so.cache;
[11016.477154] [KERNEL_MONITOR]: Process 5683; close fd: 3; filename: ld.so.cache
[11016.477163] [KERNEL_MONITOR]: Process 5683; open /lib/x86_64-linux-gnu/libc.so.6;
[11016.477167] [KERNEL_MONITOR]: Process 5683; read fd: 3; buf: 00000000d4169008; count: 832; filename: libc-2.27.so
[11016.477202] [KERNEL_MONITOR]: Process 5683; close fd: 3; filename: libc-2.27.so
[11016.477220] [KERNEL_MONITOR]: Process 3049; write fd: 3; buf: 0000000031557b2b; count: 35; filename: UNIX
[11016.477255] [KERNEL_MONITOR]: Process 3049; read fd: 54; buf: 0000000086bce53f; count: 65536; filename: ptmx
[11016.477306] [KERNEL_MONITOR]: Process 3049; write fd: 3; buf: 00000000381eec8a; count: 67; filename: UNIX
[11016.477367] [KERNEL_MONITOR]: Process 5683; open alphabet.txt;
[11016.477379] [KERNEL_MONITOR]: Process 5683; read fd: 3; buf: 000000000331c5f6; count: 4096; filename: alphabet.txt
[11016.477385] [KERNEL_MONITOR]: Process 5683; read fd: 3; buf: 000000000331c5f6; count: 4096; filename: alphabet.txt
[11016.477390] [KERNEL_MONITOR]: Process 3010; read fd: 12; buf: 0000000053c51ff2; count: 1024; filename: [eventfd]
[11016.477392] [KERNEL_MONITOR]: Process 5683; close fd: 3; filename: alphabet.txt
```

Рисунок 4.5 — Результат мониторинга программы с буферизованным вводом данных.

6) Программа, которая записывает информацию в файл, используя библиотеку буферизованного ввода/вывода (листинг 4.5).

Листинг 4.5 — Программа с буферизованным выводом данных в файл

```
1  #include <stdio.h>
2
3  int main()
4  {
5      FILE *f = fopen("test.txt", "w");
6      char buf[128] = "1234567890";
7      fprintf(f, "%s", buf);
8      fprintf(f, "%s", buf);
9      fclose(f);
10     return 0;
11 }
```

Система запускает процесс, который открывает файл test.txt и два раза записывает в него по 10 байт, используя для этого функцию fprintf библиотеки буферизованного ввода/вывода stdio, после чего файл закрывается.

На рисунке 4.6 представлен результат мониторинга системных вызовов выполняемых в ходе работы данной программы, из которого видно, что системный вызов sys_write был вызван единожды на запись 20 байт, т.к. использовался буфер на 4096 байт, который записывается в файл, либо если буфер полностью заполнен, либо файл закрывается.

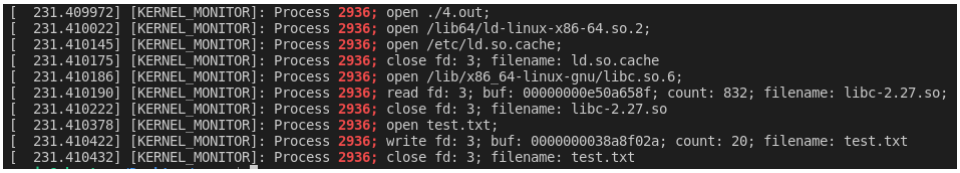

Рисунок 4.6 — Результат мониторинга программы с буферизованным вывод данных.

7) Многопоточная программа, которая записывает в файл с потерей данных (листинг 4.6).

Листинг 4.6 — Многопоточная программа записывающая данные в один файл

```c
#include <stdio.h>
#include <sys/stat.h>
#include <pthread.h>

#define THREADS 4
void *write_file(void *arg)
{
    int num = (int)arg;
    struct stat statbuf;
    FILE *f = fopen("write_thread.txt", "w");

    for (char c = 'a' + num; c <= 'z'; c += THREADS)
        fprintf(f, "%c", c);

    fclose(f);
    return 0;
}

int main()
{
    pthread_t threads[THREADS];

    for (int i = 0; i < THREADS; i++)
    {
        int code = pthread_create(threads + i, NULL, write_file, i);
        if (code != 0)
        {
            printf("can't create thread, code = %d\n", code);
            return code;
        }
    }

    for (int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
    return 0;
}
```

Проанализируем поведение перехватываемых функций, в случае многопоточной обработки файла. Для этого была реализована пятая программа , которая в несколько потоков записывает данные в один файл с потерей данных.

Система запускает процесс, который в четыре потока, открывает файл write_thread.txt и записывает в него информацию с потерей данных.

На рисунке 4.7 представлен отфильтрованный результат мониторинга системных вызовов выполняемых в ходе работы данной программы. Из него можно сделать вывод, что потоки в линуксе на самом деле реализованы в виде процессов.

```
[11608.703072] [KERNEL_MONITOR]: Process 6147; close fd: 4; filename:
[11608.703073] [KERNEL_MONITOR]: Process 3105; close fd: 3; filename:
[11608.703075] [KERNEL_MONITOR]: Process 3105; close fd: 4; filename:
[11608.703076] [KERNEL_MONITOR]: Process 6147; read fd: 3; buf: 00000000f83ecca3; count: 1; filename:
[11608.703225] [KERNEL_MONITOR]: Process 6147; close fd: 3; filename:
[11608.703291] [KERNEL_MONITOR]: Process 6147; open ./5.out;
[11608.703352] [KERNEL_MONITOR]: Process 6147; open /lib64/ld-linux-x86-64.so.2;
[11608.703571] [KERNEL_MONITOR]: Process 6147; open /etc/ld.so.cache;
[11608.703579] [KERNEL_MONITOR]: Process 6147; close fd: 3; filename: ld.so.cache
[11608.703588] [KERNEL_MONITOR]: Process 6147; open /lib/x86_64-linux-gnu/libpthread.so.0;
[11608.703593] [KERNEL_MONITOR]: Process 6147; read fd: 3; buf: 000000006d13414a; count: 832; filename: libpthread-2.27.so
[11608.703625] [KERNEL_MONITOR]: Process 6147; close fd: 3; filename: libpthread-2.27.so
[11608.703637] [KERNEL_MONITOR]: Process 6147; open /lib/x86_64-linux-gnu/libc.so.6;
[11608.703641] [KERNEL_MONITOR]: Process 6147; read fd: 3; buf: 0000000091377c55; count: 832; filename: libc-2.27.so
[11608.703668] [KERNEL_MONITOR]: Process 6147; close fd: 3; filename: libc-2.27.so
[11608.706322] [KERNEL_MONITOR]: Process 6148; open write_thread.txt;
[11608.706399] [KERNEL_MONITOR]: Process 6149; open write_thread.txt;
[11608.708121] [KERNEL_MONITOR]: Process 6150; open write_thread.txt;
[11608.708182] [KERNEL_MONITOR]: Process 6150; write fd: 5; buf: 00000000c38792fe; count: 6; filename: write_thread.txt
[11608.708192] [KERNEL_MONITOR]: Process 6150; close fd: 5; filename: write_thread.txt
[11608.708410] [KERNEL_MONITOR]: Process 3022; read fd: 72; buf: 00000000566f8430; count: 4096; filename: inotify
[11608.708618] [KERNEL_MONITOR]: Process 6148; write fd: 3; buf: 00000000ce1b5a1c; count: 7; filename: write_thread.txt
[11608.708633] [KERNEL_MONITOR]: Process 6148; close fd: 3; filename: write_thread.txt
[11608.708675] [KERNEL_MONITOR]: Process 6149; write fd: 4; buf: 000000007f37cdf6; count: 7; filename: write_thread.txt
[11608.708679] [KERNEL_MONITOR]: Process 6149; close fd: 4; filename: write_thread.txt
[11608.708701] [KERNEL_MONITOR]: Process 6151; open write_thread.txt;
[11608.708757] [KERNEL_MONITOR]: Process 6151; write fd: 3; buf: 000000007f37cdf6; count: 6; filename: write_thread.txt
[11608.708767] [KERNEL_MONITOR]: Process 6151; close fd: 3; filename: write_thread.txt
```

Рисунок 4.7 — Результат мониторинга многопоточной программы записывающая данные в файл с потерей данных.

Заключение

В соответствии с заданием на курсовую работу по операционным системам был реализован загружаемый модуль ядра операционной системы Linux. В процессе разработки был реализовать метод, 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.

Исследованы особенности системных вызовов осуществляющих взаимодеиствие с файловыми системами и различными функциями пространства пользователя для буферизованного и небуферизованного ввода/вывода.

Обнаружены файлы, которые не имеют символического имени и выявлена проблема с перехватом sys_open через таблицу системных вызов sys_call_table, что может быть связано с ассемблерной оптимизацией данного обработчика системного вызова.

Показано, что библиотека для работы с потоками в операционной системе Linux, на самом деле создаёт процессы, т.к. потоки в Linux «дорогие».

Список использованных источников

1. Elixir.bootlin. bdev_read_page. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/fs/block_dev.c#L694, (дата обращения: 26.11.2021).

2. Elixir.bootlin. bdev_write_page. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/fs/block_dev.c#L732, (дата обращения: 26.11.2021).

3. Elixir.bootlin. sys_open. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/fs/open.c#L1076, (дата обращения: 26.11.2021).

4. Elixir.bootlin. sys_read. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/fs/read_write.c#L586, (дата обращения: 26.11.2021).

5. Elixir.bootlin. sys_write. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/fs/read_write.c#L591, (дата обращения: 26.11.2021).

6. Elixir.bootlin. sys_close. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/fs/open.c#L1157, (дата обращения: 26.11.2021).

7. Elixir.bootlin. random_read. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/v5.0/source/drivers/char/random.c#L1876, (дата обращения: 26.11.2021).

8. Chris Wright Crispin Cowan, James Morris. Linux Security Module Framework / James Morris Chris Wright, Crispin Cowan. — 2002.

9. Linux Syscall Reference (64 bit). // [Электронный ресурс]. Режим доступа: https://syscalls64.paolostivanin.com/, (дата обращения: 26.10.2021).

10. Elixir.bootlin. random_read. // [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/413241/, (дата обращения: 26.11.2021).

11. 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).

12. 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).

13. 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).

14. 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).

15. 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).

16. Elixir.bootlin. ftrace_ops. // [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/linux/ftrace.h#L225, (дата обращения: 26.11.2021).

17. 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).

Приложение А Исходный код программы

Листинг А.1 — Makefile для сбоки загружаемого модуля ядра.

```
obj-m += kernel_monitor.o
moduleko-objs := kernel_monitor.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

test:
	sudo dmesg -C
	sudo insmod kernel_monitor.ko
	-cd test-program; ./$(program).out
	-sudo rmmod kernel_monitor.ko
	dmesg | grep $(program).out
```

Листинг А.2 — syscall_hooks.h.

```
#ifndef HOOKS_H
#define HOOKS_H
#include <linux/syscalls.h>

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

#define CR0_WP 0x00010000

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

static inline void protect_memory(void)
{
    unsigned long cr0 = read_cr0();
    cr0_write(cr0 | CR0_WP);
}

static inline void unprotect_memory(void)
{
    unsigned long cr0 = read_cr0();
    cr0_write(cr0 & ~CR0_WP);
}
```

```
27
28 #endif // HOOKS_H
```

Листинг А.3 — ftrace_helper.h.

```
1  #include <linux/ftrace.h>
2  #include <linux/linkage.h>
3  #include <linux/slab.h>
4  #include <linux/uaccess.h>
5  #include <linux/version.h>
6
7  #define HOOK(_name, _hook, _orig)   \
8  {                   \
9      .name = (_name),        \
10     .function = (_hook),        \
11     .original = (_orig),        \
12 }
13
14 #define USE_FENTRY_OFFSET 0
15 #if !USE_FENTRY_OFFSET
16 #pragma GCC optimize("-fno-optimize-sibling-calls")
17 #endif
18
19 struct ftrace_hook {
20     const char *name;
21     void *function;
22     void *original;
23
24     unsigned long address;
25     struct ftrace_ops ops;
26 };
27
28 static int fh_resolve_hook_address(struct ftrace_hook *hook)
29 {
30     hook->address = kallsyms_lookup_name(hook->name);
31
32     if (!hook->address)
33     {
34         printk(KERN_DEBUG "unresolved symbol: %s\n", hook->name);
35         return -ENOENT;
36     }
37
38 #if USE_FENTRY_OFFSET
39     *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_SIZE;
40 #else
41     *((unsigned long*) hook->original) = hook->address;
42 #endif
```

```
43
44      return 0;
45  }
46
47  static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
        struct ftrace_ops *ops, struct pt_regs *regs)
48  {
49      struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
50
51  #if USE_FENTRY_OFFSET
52      regs->ip = (unsigned long) hook->function;
53  #else
54      if(!within_module(parent_ip, THIS_MODULE))
55          regs->ip = (unsigned long) hook->function;
56  #endif
57  }
58
59  int fh_install_hook(struct ftrace_hook *hook)
60  {
61      int err;
62      err = fh_resolve_hook_address(hook);
63      if(err)
64          return err;
65
66      hook->ops.func = fh_ftrace_thunk;
67      hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
68              | FTRACE_OPS_FL_RECURSION_SAFE
69              | FTRACE_OPS_FL_IPMODIFY;
70
71      err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
72      if(err)
73      {
74          printk(KERN_DEBUG "ftrace_set_filter_ip() failed: %d\n", err);
75          return err;
76      }
77
78      err = register_ftrace_function(&hook->ops);
79      if(err)
80      {
81          printk(KERN_DEBUG "register_ftrace_function() failed: %d\n", err);
82          return err;
83      }
84
85      return 0;
86  }
87
88  void fh_remove_hook(struct ftrace_hook *hook)
```

```
89  {
90      int err;
91      err = unregister_ftrace_function(&hook->ops);
92      if(err)
93      {
94          printk(KERN_DEBUG "rootkit: unregister_ftrace_function() failed: %d\n", err);
95      }
96
97      err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
98      if(err)
99      {
100         printk(KERN_DEBUG "rootkit: ftrace_set_filter_ip() failed: %d\n", err);
101     }
102 }
103
104 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
105 {
106     int err;
107     size_t i;
108
109     for (i = 0 ; i < count ; i++)
110     {
111         err = fh_install_hook(&hooks[i]);
112         if(err)
113         {
114             fh_remove_hooks(hooks, i);
115             return err;
116         }
117     }
118     return 0;
119 }
120
121 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
122 {
123     size_t i;
124     for (i = 0 ; i < count ; i++)
125         fh_remove_hook(&hooks[i]);
126 }
```

Листинг А.4 — kernel_monitor.c.

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  #include <linux/syscalls.h>
5  #include <linux/kallsyms.h>
6  #include <linux/namei.h>
```

```
7
8  #include <linux/module.h>
9  #include <linux/kernel.h>
10 #include <linux/syscalls.h>
11 #include <linux/kallsyms.h>
12
13 #include <linux/sched.h>
14 #include <linux/fdtable.h>
15 #include <linux/path.h>
16 #include <linux/dcache.h>
17
18 /* open.c */
19 struct open_flags {
20     int open_flag;
21     umode_t mode;
22     int acc_mode;
23     int intent;
24     int lookup_flags;
25 };
26
27 #include "syscall_hooks.h"
28 #include "ftrace_helper.h"
29
30 MODULE_LICENSE("GPL");
31 MODULE_AUTHOR("Kozlov M.A.");
32 MODULE_DESCRIPTION("Kernel monitoring");
33 MODULE_VERSION("0.1");
34
35 /* Адрес таблицы системных вызовов */
36 static unsigned long * __sys_call_table;
37
38 /* Префикс лога */
39 #define KERNEL_MONITOR "[KERNEL_MONITOR]: "
40
41 syscall_t orig_open;
42 asmlinkage int hook_open(const struct pt_regs *regs)
43 {
44     printk(KERN_INFO KERNEL_MONITOR "Process %d; open\n", current->pid);
45     const char __user *filename = (char *)regs->di;
46     int flags = (int)regs->si;
47     umode_t mode = (umode_t)regs->dx;
48
49     char kernel_filename[NAME_MAX] = {0};
50
51     /* копировать имя файла из пр-ва пользователя в пр-во ядра */
52     long error = strncpy_from_user(kernel_filename, filename, NAME_MAX);
53
```

```
54      int fd = orig_open(regs);
55
56      if (!error && current->real_parent->pid > 3)
57              printk(KERN_INFO KERNEL_MONITOR "Process %d; open: %s, flags: %x; mode:
                %x; fd: %d\n", current->pid, kernel_filename, flags, mode, fd);
58
59      return fd;
60  }
61
62  syscall_t orig_close;
63  asmlinkage int hook_close(const struct pt_regs *regs)
64  {
65      unsigned int fd = (unsigned int)regs->di;
66
67      /* Не логировать закрытие stdin, stdout, stderr */
68      if (fd > 2 && current->real_parent->pid > 3)
69      {
70          /* Open file information: */
71          // current->files
72
73          printk(KERN_INFO KERNEL_MONITOR "Process %d; close fd: %d; filename: %s\n",
                current->pid, fd,
74              current->files->fdt->fd[fd]->f_path.dentry->d_iname);
75      }
76      return orig_close(regs);
77  }
78
79  syscall_t orig_read;
80  asmlinkage int hook_read(const struct pt_regs *regs)
81  {
82      unsigned int fd = (unsigned int)regs->di;
83      char __user *buf = (char*)regs->si;
84      size_t count = (size_t)regs->dx;
85
86      /* Не логировать стандартный ввод/вывод, а так же системные процессы */
87      if (fd > 2 && current->real_parent->pid > 3)
88          printk(KERN_INFO KERNEL_MONITOR "Process %d; read fd: %d; buf: %p; count:
                %ld; filename: %s\n", current->pid, fd, buf, count,
89              current->files->fdt->fd[fd]->f_path.dentry->d_iname);
90      return orig_read(regs);
91  }
92
93  syscall_t orig_write;
94  asmlinkage int hook_write(const struct pt_regs *regs)
95  {
96      unsigned int fd = (unsigned int)regs->di;
97      const char __user *buf = (const char*)regs->si;
```

```
98      size_t count = (size_t)regs->dx;
99
100     /* Не логировать стандартный ввод/вывод, а так же системные процессы */
101     if (fd > 2 && current->real_parent->pid > 3)
102         printk(KERN_INFO KERNEL_MONITOR "Process %d; write fd: %d; buf: %p; count:
            %ld; filename: %s\n", current->pid, fd, buf, count,
103             current->files->fdt->fd[fd]->f_path.dentry->d_iname);
104     return orig_write(regs);
105 }
106
107 static asmlinkage struct file* ( *orig_do_filp_open)(int dfd, struct filename
    *pathname, const struct open_flags *op);
108 static asmlinkage struct file* hook_do_filp_open(int dfd, struct filename *pathname,
    const struct open_flags *op)
109 {
110     if (current->real_parent->pid > 3)
111         printk(KERN_INFO KERNEL_MONITOR "Process %d; open %s;\n", current->pid,
            pathname->name);
112
113     struct file* file;
114     file = orig_do_filp_open(dfd, pathname, op);
115     return file;
116 }
117
118 static asmlinkage int ( *orig_bdev_read_page)(struct block_device *bdev, sector_t
    sector, struct page *page);
119 static asmlinkage int hook_bdev_read_page(struct block_device *bdev, sector_t
    sector, struct page *page)
120 {
121     int err;
122     err = orig_bdev_read_page(bdev, sector, page);
123     printk(KERN_INFO KERNEL_MONITOR "Process %d bdev_read_page; dev: %d\n",
        current->pid, bdev->bd_dev);
124     return err;
125 }
126
127 static asmlinkage int ( *orig_bdev_write_page)(struct block_device *bdev, sector_t
    sector, struct page *page, struct writeback_control *wbc);
128 static asmlinkage int hook_bdev_write_page(struct block_device *bdev, sector_t
    sector, struct page *page, struct writeback_control *wbc)
129 {
130     int err;
131     err = orig_bdev_write_page(bdev, sector, page, wbc);
132     printk(KERN_INFO KERNEL_MONITOR "Process %d bdev_write_page; dev: %d\n",
        current->pid, bdev->bd_dev);
133     return err;
134 }
```

```
135
136 static asmlinkage ssize_t ( *orig_random_read)(struct file *file, char __user *buf,
    size_t nbytes, loff_t *ppos);
137 static asmlinkage ssize_t hook_random_read(struct file *file, char __user *buf,
    size_t nbytes, loff_t *ppos)
138 {
139     /* Вызов оригинального random_read() */
140     int bytes_read;
141     bytes_read = orig_random_read(file, buf, nbytes, ppos);
142     printk(KERN_INFO KERNEL_MONITOR "Process %d read %d bytes from /dev/random\n",
        current->pid, bytes_read);
143     return bytes_read;
144 }
145
146 static asmlinkage ssize_t ( *orig_urandom_read)(struct file *file, char __user *buf,
    size_t nbytes, loff_t *ppos);
147 static asmlinkage ssize_t hook_urandom_read(struct file *file, char __user *buf,
    size_t nbytes, loff_t *ppos)
148 {
149     /* Вызов оригинального urandom_read() */
150     int bytes_read;
151     bytes_read = orig_urandom_read(file, buf, nbytes, ppos);
152     printk(KERN_INFO KERNEL_MONITOR "Process %d read %d bytes from /dev/urandom\n",
        current->pid, bytes_read);
153     return bytes_read;
154 }
155
156 static struct ftrace_hook hooks[] =
157 {
158     HOOK("random_read", hook_random_read, &orig_random_read),
159     HOOK("urandom_read", hook_urandom_read, &orig_urandom_read),
160     HOOK("bdev_read_page", hook_bdev_read_page, &orig_bdev_read_page),
161     HOOK("bdev_write_page", hook_bdev_write_page, &orig_bdev_write_page),
162     HOOK("do_filp_open", hook_do_filp_open, &orig_do_filp_open)
163 };
164
165 /* Функция инициализации модуля */
166 static int __init kernel_monitor_init(void)
167 {
168     /* Поиск начального адреса таблицы системных вызовов */
169     __sys_call_table = kallsyms_lookup_name("sys_call_table");
170     if (!__sys_call_table)
171         return -1;
172
173     int err;
174     /* Установка ftrace hooks */
175     err = fh_install_hooks(hooks, ARRAY_SIZE(hooks));
```

```
176     if (err)
177         return err;
178 
179     /* Получение адресов оригинальных системных вызовов */
180     orig_mkdir  = (syscall_t)__sys_call_table[__NR_mkdir];
181 
182     orig_open   = (syscall_t)__sys_call_table[__NR_open];
183     orig_close  = (syscall_t)__sys_call_table[__NR_close];
184     orig_read   = (syscall_t)__sys_call_table[__NR_read];
185     orig_write  = (syscall_t)__sys_call_table[__NR_write];
186 
187     /* Логгирование адресов системных вызовов */
188     printk(KERN_INFO KERNEL_MONITOR "Loading ...");
189     printk(KERN_DEBUG KERNEL_MONITOR "Found the syscall table at 0x%lx\n",
        __sys_call_table);
190     printk(KERN_DEBUG KERNEL_MONITOR "mkdir: 0x%lx\n", orig_mkdir);
191     printk(KERN_DEBUG KERNEL_MONITOR "open:  0x%lx\n", orig_open);
192     printk(KERN_DEBUG KERNEL_MONITOR "close: 0x%lx\n", orig_close);
193     printk(KERN_DEBUG KERNEL_MONITOR "read:  0x%lx\n", orig_read);
194     printk(KERN_DEBUG KERNEL_MONITOR "write: 0x%lx\n", orig_write);
195 
196     printk(KERN_INFO KERNEL_MONITOR "Hooking syscalls\n");
197 
198     /* Для модификации системной таблицы необходимо снять со страницы защиту от запис
        и */
199     unprotect_memory();
200     __sys_call_table[__NR_open]     = (unsigned long)hook_open;
201     __sys_call_table[__NR_close]    = (unsigned long)hook_close;
202     __sys_call_table[__NR_read]     = (unsigned long)hook_read;
203     __sys_call_table[__NR_write]    = (unsigned long)hook_write;
204     /* Восстановить защиту от записи */
205     protect_memory();
206 
207     return 0;
208 }
209 
210 static void __exit kernel_monitor_exit(void)
211 {
212     printk(KERN_INFO KERNEL_MONITOR "Restoring syscalls\n");
213 
214     /* Удаление hooks ftrace */
215     fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
216 
217     /* Восстановление системной таблицы */
218     unprotect_memory();
219     __sys_call_table[__NR_open]     = (unsigned long)orig_open;
220     __sys_call_table[__NR_close]    = (unsigned long)orig_close;
```

```
221     __sys_call_table[__NR_read]      = (unsigned long)orig_read;
222     __sys_call_table[__NR_write]     = (unsigned long)orig_write;
223     protect_memory();
224 
225     printk(KERN_INFO KERNEL_MONITOR "Unloaded.\n");
226 }
227 
228 module_init(kernel_monitor_init);
229 module_exit(kernel_monitor_exit);
```