Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовой работе на тему:

Мониторинг вызовов функций ядра Linux

Студент _____ Камакин А.С.
(Подпись, дата)

Руководитель курсового проекта _____ Ковтушенко А.П.
(Подпись, дата)

Москва 2018

Содержание

Введение

Иногда, при работе с Linux-системами, требуется перехватывать вызовы важных функций внутри ядра (вроде открытия файлов и запуска процессов) для обеспечения возможности мониторинга активности в системе или превентивного блокирования деятельности подозрительных процессов.

Проект посвящен исследованию способов перехвата вызовов функций внутри ядра с их последующей отправкой на удаленный компьютер из пространства ядра. Целью проекта является разработка подхода, позволяющего удобно перехватить любую функцию в ядре по имени и выполнить свой код вокруг её вызовов с последующей отправкой данных на удаленный компьютер.

1 Аналитический раздел

В соответствии с заданием на курсовой проект необходимо разработать программное обеспечение, перехватывающее события в системе, инициирующиеся средством исполнения экспортируемых функций ядра. Так же эту информацию необходимо передавать на удаленный компьютер. Программное обеспечение должно обеспечивать перехват всех действий нужных функций и обеспечивать пользователю возможность удаленно ее анализировать.

1.1 Анализ подходов реализации

— Linux Security API
— Модификация таблицы системных вызовов
— Использование kprobes
— Сплайсинг
— Новый подход с ftrace

1.1.1 Linux Security API

Наиболее правильным было бы использование Linux Security API — специального интерфейса, созданного именно для этих целей. В критических местах ядерного кода расположены вызовы security-функций, которые в свою очередь вызывают коллбеки, установленные security-модулем. Security-модуль может изучать контекст операции и принимать решение о её разрешении или запрете.

К сожалению, у Linux Security API есть пара важных ограничений:

— security-модули не могут быть загружены динамически, являются частью ядра и требуют его пересборки

— 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

Мониторинг требовался в основном для действий, выполняемых пользовательскими приложениями, так что в принципе мог бы быть реализован на уровне системных вызовов. Как известно, Linux хранит все обработчики системных вызо-

вов в таблице sys_call_table. Подмена значений в этой таблице приводит к смене поведения всей системы. Таким образом, сохранив старое значения обработчика и подставив в таблицу собственный обработчик, мы можем перехватить любой системный вызов.

У этого подхода есть определённые преимущества:

— Полный контроль над любыми системными вызовами — единственным интерфейсом к ядру у пользовательских приложений. Используя его мы можем быть уверены, что не пропустим какое-нибудь важное действие, выполняемое пользовательским процессом.

— Минимальные накладные расходы. Есть единоразовые капитальные вложения при обновлении таблицы системных вызовов. Помимо неизбежной полезной нагрузки мониторинга, единственным расходом является лишний вызов функции (для вызова оригинального обработчика системного вызова).

— Минимальные требования к ядру. При желании этот подход не требует каких-либо дополнительных конфигурационных опций в ядре, так что в теории поддерживает максимально широкий спектр систем.

Однако, подход не лишен недостатков:

— Техническая сложность реализации. Сама по себе замена указателей в таблице не представляет трудностей. Но сопутствующие задачи требуют неочевидных решений и определённой квалификации:

поиск таблицы системных вызовов

обход защиты от модификации таблицы

атомарное и безопасное выполнение замены

— Невозможность перехвата некоторых обработчиков. В ядрах до версии 4.16 обработка системных вызовов для архитектуры x86_64 содержала целый ряд оптимизаций. Некоторые из них требовали того, что обработчик системного вызова являлся специальным переходничком, реализованным 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 в копилку технических сложностей.

— Перехватываются только системные вызовы. Этот подход позволяет заменять обработчики системных вызовов, что ограничивает точки входа только ими. Все дополнительные проверки выполняются либо в начале, либо в конце, и у нас есть лишь аргументы системного вызова и его возвращаемое значение. Иногда это приводит к необходимости дублировать проверки на адекватность аргументов и проверки доступа. Иногда вызывает лишние накладные расходы, когда требуется дважды копировать память пользовательского процесса: если аргумент передаётся через

указатель, то его сначала придётся скопировать нам самим, затем оригинальный обработчик скопирует аргумент ещё раз для себя. Кроме того, в некоторых случаях системные вызовы предоставляют слишком низкую гранулярность событий, которые приходится дополнительно фильтровать от шума.

Данный подход позволяет полностью подменить таблицу системных вызовов что является несомненным плюсом, но также ограничивает количество функций, которые можно мониторить.

1.1.3 Использование kprobes

Одним из вариантов, которые рассматривались, было использование kprobes: специализированного API, в первую очередь предназначенного для отладки и трассирования ядра. Этот интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут их изменять. Таким образом, можно было бы получить как мониторинг, так и возможность влиять на дальнейший ход работы.

Преимущества, которые даёт использование kprobes для перехвата:

— Зрелый API. Kprobes существуют и улучшаются с 2002 года. Они обладают хорошо задокументированным интерфейсом, большинство подводных камней уже найдено, их работа по возможности оптимизирована.

— 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).

Недостатки kprobes:

— Техническая сложность. Kprobes — это только способ установить точку останова в любом места ядра. Для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах или где на стеке они лежат, и самостоятельно их оттуда извлекать. Для блокировки вызова функции необходимо вручную модифицировать состояние процесса так, чтобы процессор подумал, что он уже вернул управление из функции.

— Jprobes объявлены устаревшими. Jprobes — это надстройка над kprobes, позволяющая удобно перехватывать вызовы функций. Она самостоятельно извлечёт аргументы функции из регистров или стека и вызовет ваш обработчик, который должен

иметь ту же сигнатуру, что и перехватываемая функция. Подвох в том, что jprobes объявлены устаревшими и вырезаны из современных ядер.

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

— 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.

— Отключенное вытеснение. Так как kprobes основывается на прерываниях и жонглирует регистрами процессора, то 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 прочие известные вещи.

1.1.4 Сплайсинг

Классический способ перехвата функций, заключающийся в замене инструкций в начале функции на безусловный переход, ведущий в обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. С помощью двух переходов вшивается (splice in) свой 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.

Преимущества сплайсинга:

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

— Минимальные накладные расходы. Два безусловных перехода — вот и все действия, которые надо выполнить перехваченному коду, чтобы передать управление обработчику и обратно. Подобные переходы отлично предсказываются процессором и являются очень дешёвыми.

Недостатки:

— Техническая сложность. Она зашкаливает. Нельзя просто так взять и переписать машинный код. Вот краткий и неполный список задач, которые придётся решить:

синхронизация установки и снятия перехвата (что если функцию вызовут прямо в процессе замены её инструкций?)

обход защиты на модификацию регионов памяти с кодом

инвалидация кешей процессора после замены инструкций

дизассемблирование заменяемых инструкций, чтобы скопировать их целыми

проверка на отсутствие переходов внутрь заменяемого куска

проверка на возможность переместить заменяемый кусок в другое место

1.1.5 Новый подход с ftrace

Ftrace — это фреймворк для трассирования ядра на уровне функций. Он разрабатывается с 2008 года и обладает удобным интерфейсом для пользовательских программ. Ftrace позволяет отслеживать частоту и длительность вызовов функций, отображать графы вызовов, фильтровать интересующие функции по шаблонам, и так далее.

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.

Вызывать ftrace из каждой функции — это, разумеется, не дёшево, поэтому для популярных архитектур доступна оптимизация: динамический ftrace. Суть в том, что ядро знает расположение всех вызовов mcount() или __fentry__() и на ранних этапах загрузки заменяет их машинный код на nop — специальную ничего не делающую инструкцию. При включении трассирования в нужные функции вызовы ftrace добавляются обратно. Таким образом, если ftrace не используется, то его влияние на систему минимально.

Преимущества:

— Зрелый API и простой код. Использование готовых интерфейсов в ядре существенно упрощает код. Вся установка перехвата требует пары вызовов функций, заполнение двух полей в структуре. Остальной код — это исключительно бизнес-логика, выполняемая вокруг перехваченной функции.

— Перехват любой функции по имени. Для указания интересующей нас функции достаточно написать её имя в обычной строке. Не требуются какие-то особые реверансы с редактором связей, разбор внутренних структур данных ядра, сканиро-

вание памяти, или что-то подобное. Можно перехватить любую функцию (даже не экспортируемую для модулей), зная лишь её имя.

— Перехват совместим с трассировкой. Очевидно, что этот способ не конфликтует с ftrace, так что с ядра всё ещё можно снимать очень полезные показатели производительности. Использование kprobes или сплайсинга может помешать механизмам ftrace.

Недостатки:

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

список символов kallsyms для поиска функций по имени
фреймворк ftrace в целом для выполнения трассировки
опции ftrace, критически важные для перехвата

Все эти возможности не являются критичными для функционирования системы и могут быть отключены в конфигурации ядра. Правда, обычно ядра, используемые популярными дистрибутивами, все эти опции в себе всё равно содержат, так как они не влияют на производительность и полезны при отладке. Однако, если вам необходимо поддерживать какие-то особенные ядра, то следует иметь в виду эти требования.

— Накладные расходы на ftrace меньше, чем у kprobes (так как ftrace не использует точки останова), но они выше, чем у сплайсинга, сделанного вручную. Действительно, динамический ftrace, является сплайсингом, только вдобавок выполняющий код ftrace и другие коллбеки.

— Оборачиваются функции целиком. Как и традиционный 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.

1.2 Загружаемые модули ядра Linux

Ядро Linux относится к категории так называемых монолитных – это означает, что большая часть функциональности операционной системы называется ядром и запускается в привилегированном режиме. Этот подход отличен от подхода микроядра, когда в режиме ядра выполняется только основная функциональность (взаимодействие между процессами [inter-process communication, IPC], диспетчеризация, базовый ввод-вывод [I/O], управление памятью), а остальная функциональность вытесняется за пределы привилегированной зоны (драйверы, сетевой стек, файловые системы). Можно было бы подумать, что ядро Linux очень статично, но на самом де-

ле все как раз наоборот. Ядро Linux динамически изменяемое – это означает, что вы можете загружать в ядро дополнительную функциональность, выгружать функции из ядра и даже добавлять новые модули, использующие другие модули ядра. Преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули (это может оказаться важным для встроенных систем) [**?**]

Linux – не единственное (и не первое) динамически изменяемое монолитное ядро. Загружаемые модули поддерживаются в BSD-системах, Sun Solaris, в ядрах более старых операционных систем, таких как OpenVMS, а также в других популярных ОС, таких как Microsoft Windows и Apple Mac OS X.

1.2.1 Устройство модуля ядра

Загружаемые модули ядра имеют ряд фундаментальных отличий от элементов, интегрированных непосредственно в ядро, а также от обычных программ. Обычная программа содержит главную процедуру (main)в отличие от загружаемого модуля, содержащего функции входа и выхода (в версии 2.6 эти функции можно именовать как угодно). Функция входа вызывается, когда модуль загружается в ядро, а функция выхода – соответственно при выгрузке из ядра. Поскольку функции входа и выхода являются пользовательскими, для указания назначения этих функций используются макросы module_init и module_exit . Загружаемый модуль содержит также набор обязательных и дополнительных макросов. Они определяют тип лицензии, автора и описание модуля, а также другие параметры. Пример очень простого загружаемого модуля приведен на рисунке 1.1 .

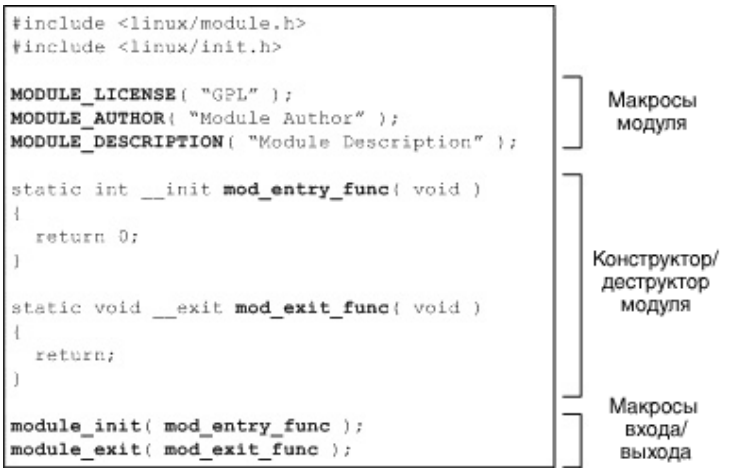

Рисунок 1.1 — Пример загружаемого модуля с разделами ELF

1.2.2 Объектный код модуля ядра

Загружаемый модуль представляет собой просто специальный объектный файл в формате ELF (Executable and Linkable Format). Обычно объектные файлы обрабатываются компоновщиком, который разрешает символы и формирует исполняемый файл. Однако в связи с тем, что загружаемый модуль не может разрешить символы до загрузки в ядро, он остается ELF-объектом. Для работы с загружаемыми модулями можно использовать стандартные средства работы с объектными файлами (которые в версии 2.6 имеют суффикс .ko, от kernel object). Например, если вывести информацию о модуле утилитой objdump, вы обнаружите несколько привычных разделов, в том числе .text (инструкции), .data (инициализированные данные) и .bss (Block Started Symbol или неинициализированные данные)[?]

В модуле также обнаружатся дополнительные разделы, ответственные за поддержку его динамического поведения. Раздел .init.text содержит код module_init, а раздел .exit.text – код module_exit code (рисунок 1.2). Раздел .modinfo содержит тексты макросов, указывающие тип лицензии, автора, описание и т.д.

Рисунок 1.2 — Пример загружаемого модуля с разделами ELF

1.2.3 Жизненный цикл загружаемого модуля ядра

Процесс загрузки модуля начинается в пользовательском пространстве с команды insmod (вставить модуль). Команда insmod определяет модуль для загрузки и выполняет системный вызов уровня пользователя init_module для начала процесса загрузки. Команда insmod для ядра версии 2.6 стала чрезвычайно простой (70 строк кода) за счет переноса части работы в ядро. Команда insmod не выполняет никаких действий по разрешению символов (вместе с командой kerneld), а просто копирует

двоичный код модуля в ядро при помощи функции init_module; остальное делает само ядро.

Функция init_module работает на уровне системных вызовов и вызывает функцию ядра sys_init_module (рисунок 1.3). Это основная функция для загрузки модуля, обращающаяся к нескольким другим функциям для решения специальных задач. Аналогичным образом команда rmmod выполняет системный вызов функции delete_module, которая обращается в ядро с вызовом sys_delete_module для удаления модуля из ядра.

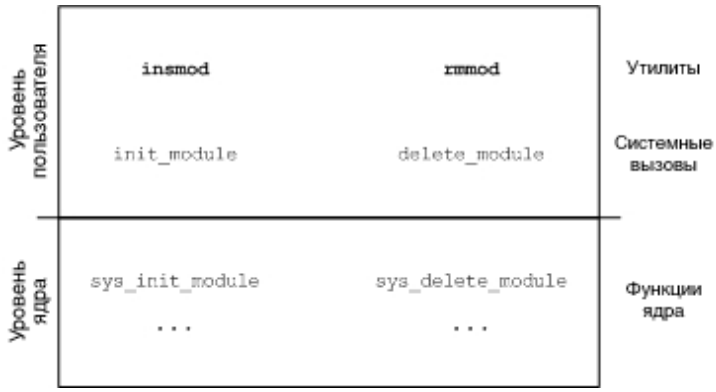

Рисунок 1.3 — Основные команды и функции, участвующие в загрузке и выгрузке модуля

1.2.4 Подробности загрузки модуля

Теперь давайте взглянем на внутренние функции для загрузки модуля (рисунок 1.4). При вызове функции ядра sys_init_module сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения (при помощи функции capable). Затем вызывается функция load_module, которая выполняет механическую работу по размещению модуля в ядре и производит необходимые операции (я вскоре расскажу об этом). Функция load_module возвращает ссылку, которая указывает на только что загруженный модуль. Затем он вносится в двусвязный список всех модулей в системе, и все потоки, ожидающие изменения состояния модуля, уведомляются при помощи специального списка. В конце вызывается функция init() и статус модуля обновляется, чтобы указать, что он загружен и доступен. Внутренние процессы загрузки модуля представляют собой анализ и управление модулями ELF. Функция load_module (которая находится в ./linux/kernel/module.c) начинает с выделения блока временной памяти для хранения всего модуля ELF. Затем модуль ELF считывается из пользовательского пространства во временную память при помощи copy_from_user. Являясь объектом ELF, этот файл имеет очень специфичную структуру, которая легко поддается анализу и проверке.

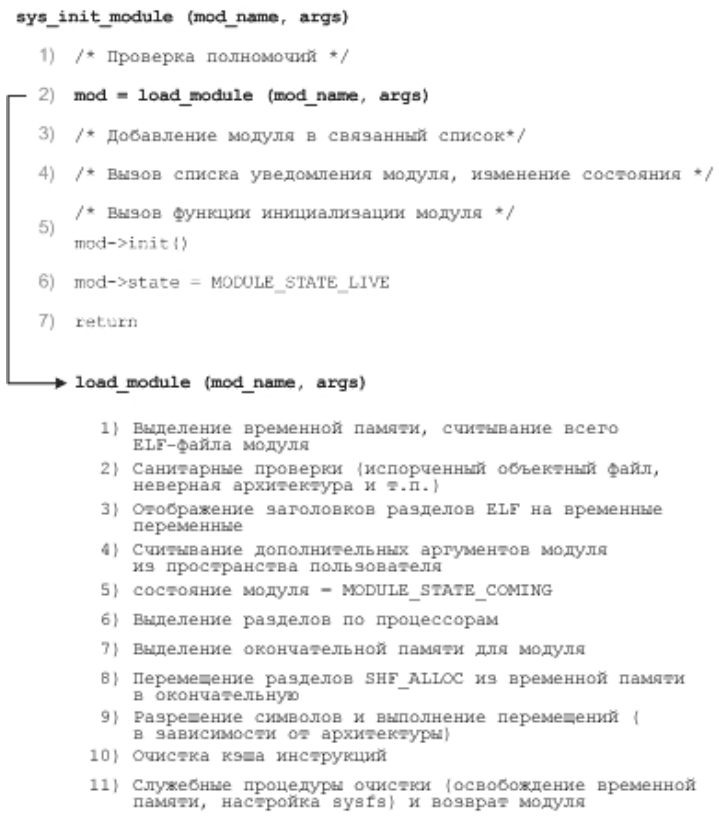

Рисунок 1.4 — Внутренний процесс загрузки модуля(в упрощенном виде)

Следующим шагом является ряд "санитарных проверок"загруженного образа (является ли ELF-файл допустимым? соответствует ли он текущей архитектуре? и так далее). После того как проверка пройдена, образ ELF анализируется и создается набор вспомогательных переменных для заголовка каждого раздела, чтобы облегчить дальнейший доступ к ним. Поскольку базовый адрес объектного файла ELF равен 0 (до перемещения), эти переменные включают соответствующие смещения в блок временной памяти. Во время создания вспомогательных переменных также проверяются заголовки разделов ELF, чтобы убедиться, что загружаемый модуль корректен.

Дополнительные параметры модуля, если они есть, загружаются из пользовательского пространства в другой выделенный блок памяти ядра (шаг 4), и статус модуля обновляется, чтобы обозначить, что он загружен (MODULE_STATE_COMING). Если необходимы данные для процессоров (согласно результатам проверки заголовков разделов), для них выделяется отдельный блок.

В предыдущих шагах разделы модуля загружались в память ядра (временную), и было известно, какие из них используются постоянно, а какие могут быть удалены. На следующем шаге (7) для модуля в памяти выделяется окончательное расположение, и в него перемещаются необходимые разделы (обозначенные в заголовках SHF_ALLOC или расположенные в памяти во время выполнения). Затем

производится дополнительное выделение памяти размера, необходимого для требуемых разделов модуля. Производится проход по всем разделам во временном блоке ELF„ и те из них, которые необходимы для выполнения, копируются в новый блок. Затем следуют некоторые служебные процедуры. Также происходит разрешение символов, как расположенных в ядре (включенных в образ ядра при компиляции), так и временных (экспортированных из других модулей).

Затем производится проход по оставшимся разделам и выполняются перемещения. Этот шаг зависит от архитектуры и соответственно основывается на вспомогательных функциях, определенных для данной архитектуры (./linux/arch/<arch>/kernel/module.c). В конце очищается кэш инструкций (поскольку использовались временные разделы .text), выполняется еще несколько служебных процедур (очистка памяти временного модуля, настройка sysfs) и, в итоге, модуль возвращает load_module

1.2.5 Подробности выгрузки модуля

Выгрузка модуля фактически представляет собой зеркальное отражение процесса загрузки за исключением того, что для безопасного удаления модуля необходимо выполнить несколько "санитарных проверок". Выгрузка модуля начинается в пользовательском пространстве с выполнения команды rmmod (удалить модуль). Внутри команды rmmod выполняется системный вызов delete_module, который в конечном счете инициирует sys_delete_module внутри ядра (см рисунок 1.3). Основные операции удаления модуля показаны на рисунке 1.5. При вызове функции ядра sys_delete_module (с именем удаляемого модуля в качестве параметра) сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения. Затем по списку проверяются зависимости других модулей от данного модуля. При этом используется список modules_which_use_me, содержащий по элементу для каждого зависимого модуля. Если список пуст, т.е. зависимостей не обнаружено, то модуль становится кандидатом на удаление (иначе возвращается ошибка). Затем проверяется, загружен ли модуль. Ничто не запрещает пользователю запустить команду rmmod для модуля, который в данный момент устанавливается, поэтому данная процедура проверяет, активен ли модуль. После нескольких дополнительных служебных проверок предпоследним шагом вызывается функция выхода данного модуля (предоставляемая самим модулем). В заключение вызывается функция free_module.

К моменту вызова free_module уже известно, что модуль может быть безопасно удален. Зависимостей не обнаружено, и для данного модуля можно начать процесс очистки ядра. Этот процесс начинается с удаления модуля из различных списков, в которые он был помещен во время установки (sysfs, список модулей и т.д.). Потом инициируется команда очистки, зависящая от архитектуры (она рас-

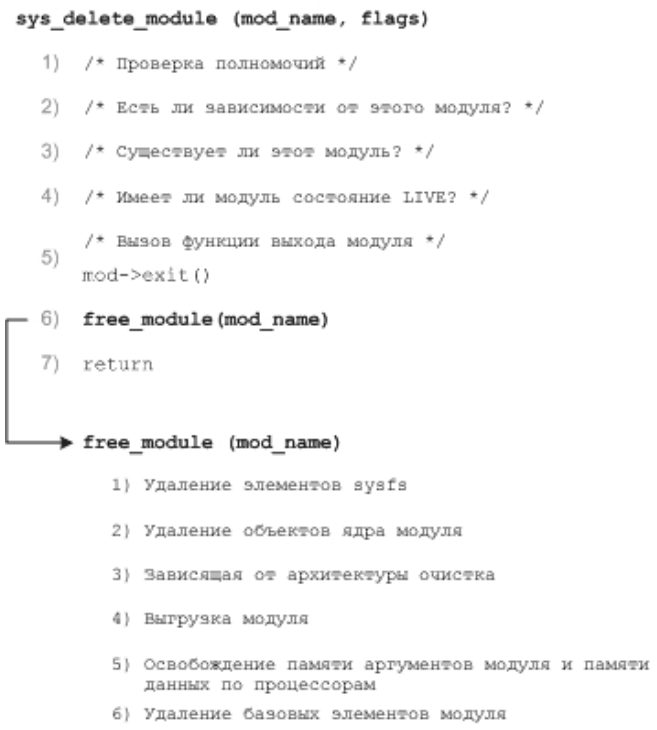

Рисунок 1.5 — Внутренний процесс выгрузки модуля(в упрощенном виде)

положена в ./linux/arch/<arch>/kernel/module.c). Затем обрабатываются зависимые модули, и данный модуль удаляется из их списков. В конце, когда с точки зрения ядра очистка завершена, освобождаются различные области памяти, выделенные для модуля, в том числе память для параметров, память для данных по процессорам и память модуля ELF (core и init).

1.3 Вывод

Проанализировав все изученные подходы к реализации мониторинга за вызовами функций ядра, был выбран подход с ftrace. Изначально наиболее подходящим вариантом была подмена таблицу системных вызовов, но при реализации возникли некоторые проблемы:

— после релиза ядра версии 3.0, таблицу больше нельзя получить простым импортом нужного модуля, доступ к ней осуществляется посредством поиска по памяти

— не работает на ядрах выше версии 4.16, так как участок памяти, в котором она находится, заблокирован, а его разблокировка ведет к ошибке установки модуля

— не позволяет следить за чем-то кроме функций, находящихся в таблице системных вызовов.

На момент написания программы, использовалось ядро 4.19.12, поэтому лучше всего подходил вариант с ftrace. Подход поддерживает ядра версий 3.19+ для архитектуры x86_64.

2 Конструкторский раздел

2.1 Общая архитектура приложения

В состав программного обеспечения входит только загружаемый модуль ядра, следящий за вызовом нужных функций, с поледующией отправкой информации о них клиенту напрямую из пространства ядра.

2.2 Перехват функций

Каждую перехватываемую функцию можно описать следующей структурой:

Листинг 2.1 — ftrace_hook

```
1  /**
2  * struct ftrace_hook - описывает перехватываемую функцию
3  *
4  * @name:     имя перехватываемой функции
5  *
6  * @function: адрес функции-обёртки, которая будет вызываться вместо
7  *            перехваченной функции
8  *
9  * @original: указатель на место, куда следует записать адрес
10 *            перехватываемой функции, заполняется при установке
11 *
12 * @address:  адрес перехватываемой функции, выясняется при установке
13 *
14 * @ops:      служебная информация ftrace, инициализируется нулями,
15 *            при установке перехвата будет доинициализирована
16 */
17 struct ftrace_hook {
18     const char *name;
19     void *function;
20     void *original;
21
22     unsigned long address;
23     struct ftrace_ops ops;
24 };
```

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

Листинг 2.2 — ftrace_hook define

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

static struct ftrace_hook hooked_functions[] = {
    HOOK("sys_clone",    fh_sys_clone,    &real_sys_clone),
};
```

Сигнатуры функций должны совпадать один к одному. Без этого, очевидно, аргументы будут переданы неправильно и всё пойдёт под откос. Для перехвата системных вызовов это важно в меньшей степени, так как их обработчики очень стабильные и для эффективности аргументы принимают в том же порядке, что и сами системные вызовы.

2.2.1 Инициализация ftrace

Для начала нам потребуется найти и сохранить адрес функции, которую мы будем перехватывать. Ftrace позволяет трассировать функции по имени, но нам всё равно надо знать адрес оригинальной функции, чтобы вызывать её.

Добыть адрес можно с помощью kallsyms — списка всех символов в ядре. В этот список входят все символы, не только экспортируемые для модулей. Получение адреса перехватываемой функции выглядит примерно так:

Листинг 2.3 — resolve_hook_address

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

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

    *((unsigned long*) hook->original) = hook->address;

    return 0;
}
```

Дальше необходимо инициализировать структуру ftrace_ops. В ней обязательным полем является лишь func, указывающая на коллбек, но нам также необходимо установить некоторые важные флаги:

Листинг 2.4 — fh_install_hook

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

Выключается перехват аналогично, только в обратном порядке:

Листинг 2.5 — fh_remove_hook

```
1  void fh_remove_hook(struct ftrace_hook *hook)
2  {
3      int err;
4
5      err = unregister_ftrace_function(&hook->ops);
6      if (err) {
7          pr_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         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
13     }
14 }
```

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

2.2.2 Выполнение перехвата функций

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

Коллбек для ftrace выглядит следующим образом:

Листинг 2.6 — fh_remove_hook

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

Функция-обёртка, которая вызывается позже, будет выполняться в том же контексте, что и оригинальная функция. Поэтому там можно делать то же, что позволено делать в перехватываемой функции. Например, если перехватывается обработчик прерывания, то спать в обёртке нельзя.

2.2.3 Схема работы перехвата

Рассмотрим пример: терминале набирается команда ls, чтобы увидеть список файлов в текущей директории. Командный интерпретатор для запуска нового процесса использует пару функций fork() + execve() из стандартной библиотеки языка Си. Внутри эти функции реализуются через системные вызовы clone() и execve() соответственно. Допустим, мы перехватываем системный вызов execve(), чтобы контролировать запуск новых процессов.

В графическом виде перехват функции-обработчика выглядит так:

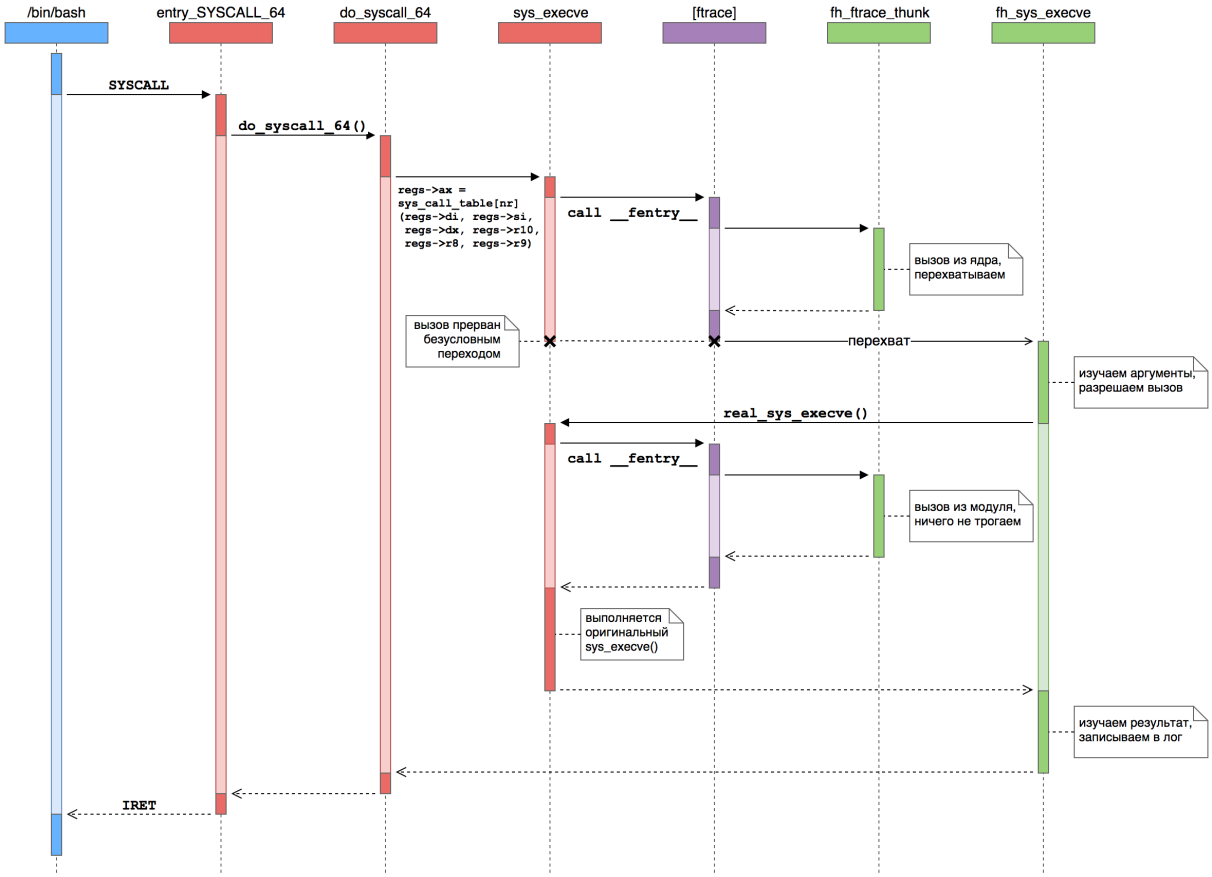

Рисунок 2.1 — Алгоритм работы функции-перехватчика

2.3 Сервер

Сервер описывается следующими структурами:

Листинг 2.7 — tcp_server_service

```
1  // Структура, описывающая и хранящие данные о текущем соединении для каждог
       о клиента
2  struct tcp_conn_handler_data {
3      struct sockaddr_in *address;
4      struct socket *accept_socket;
5      int thread_id;
6  };
7
8  // Структура, описывающая и хранящие данные о текущих соединениях для всех
       клиентов
9  struct tcp_conn_handler {
10     struct tcp_conn_handler_data *data[MAX_CONNS];
11     struct task_struct *thread[MAX_CONNS];
12     int tcp_conn_handler_stopped[MAX_CONNS];
13 };
14
15 // Структура, описывающая весь сервис
16 struct tcp_server_service {
17     int running;
18     struct socket *listen_socket;
19     struct task_struct *thread;
20     struct task_struct *accept_thread;
21 };
```

2.3.1 Алгоритм работы сервера

При загрузке модуля, выделяется память под структуры tcp_server_service и tcp_conn_handler, running устанавливается в 1 и создается поток, слушающий, все приходящие соединения. Этому потоку передается на вход функция tcp_server_listen, которая делает стандартную последовательность действий для любого сервера, а именно: заполняет структуру sockaddr_in, далее вызывает функции bind и listen. После создается новый поток для принятия соединений, ему на вход передается функция tcp_server_accept. Ранее созданный поток переходит в режим ожидания событий.

Соединения на сервере работают по принципу keep-alive, это значит, что не рвется соединение с клиентом, а постоянно что-то записывается в сокет. Поэтому необходимо, чтобы клиенты не блокировали друг-друга. После того, как соединение принято функцией accept и заполнился сокет для каждого клиента, обработка соединения передается новому потоку, чтобы можно было принимать клиентов дальше.

Преимущества подхода:

— Клиенты не блокируют и не ждут друг-друга
— Клиенты никак не взаимодействуют между собой
— Падение клиента не влияет на работу системы

Недостатки:

— На каждого клиента создается новый поток, а значит будет создано N потоков в пространстве ядра на N клиентов. Это несомненно плохо, потому что больше задач будет бороться за получение кванта процессорного времени.

— Отправка данных клиенту происходит из функции перехватчика, поэтому требуется каким-то образом передать ей данные о соединении. Самый простой способ - глобальный массив клиентов. В этом подходе есть существенная проблема: если описать несколько функций перехватчиков и передавать данные одним и тем же клиентам, то будет конкурентный доступ к этому массиву. В функциях перехватчиках нельзя вызывать блокировки, в связи с тем, что они могут повесить систему.

3 Технологический раздел

3.1 Выбор языка программирования

Для реализации загружаемого модуля был выбран язык C. Операционная система Linux позволяет писать загружаемые модули ядра на Rust и на C. Rust непопулярен и ещё только развивается и не обладает достаточным количеством документации. Для реализации клиента был выбран протокол telnet.

— Имеется в любой современной ОС.
— Простота использования.
— Скорость и эффективность.

3.2 Выбор среды разработки

Для написания программы, был выбран текстовый редактор vim.

— Огромное количество плагинов, которые позволяют делать работу быстрее
— Возможность гибкой настройки под себя
— Много встроенных команд и комбинаций

Заключение

В данной работе был реализован загружаемый модуль ядра операционной системы Linux, В процессе разработки нам удалось реализовать подход, позволяющий удобно перехватить любую функцию в ядре по имени и выполнить свой код вокруг её вызовов и сразу отправить нужную информацию клиенту без перехода в режим пользователя. Перехватчик можно устанавливать из загружаемого GPL-модуля, без пересборки ядра. Подход поддерживает ядра версий 3.19+ для архитектуры x86_64.

Список использованных источников

1. Перехват функций в ядре Linux с помощью ftrace
https://m.habr.com/post/413241/

2. Haifa Linux Club - Networking Lectures
http://haifux.org/network.html

3. Loadable Kernel Module Programming and System Call Interception
https://www.linuxjournal.com/article/4378

4. М. Джонс *Анатомия загружаемых модулей ядра Linux.*
https://www.ibm.com/developerworks/ru/library/l-lkm/index.html

5. Исходные коды ядра Linux
http://elixir.free-electrons.com