СОДЕРЖАНИЕ

Введение

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

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра, перехватывающий события в системе и передавать информацию о них на удаленный компьютер. В рамках данной работы для выполнения перехвата был выбран системный вызов sys_clone.

Для решения поставленной цели необходимо решить следующие задачи:

1) проанализировать существующие методы перехвата системных функций;
2) реализовать загружаемый модуль ядра;
3) реализовать сервер, передающий информацию на удаленный компьютер;
4) исследовать результаты работы разработанного модуля.

1.2 Методы перехвата функций ядра Linux

1.2.1 Linux Security API

Linux Security API — специальный интерфейс, созданный именно для перехвата системных функций. В критических местах кода ядра расположены вызовы security-функций, которые в свою очередь вызывают коллбеки, установленные security-модулем. Security-модуль может изучать контекст операции и принимать решение о её разрешении или запрете.

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.

1.2.2 Модификация таблицы системных вызовов

Linux хранит все обработчики системных вызовов 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, можно перехватить любой системный вызов.

У этого подхода есть следующие преимущества:

- полный контроль над любыми системными вызовами;
- отсутствие необходимости в каких-либо дополнительных конфигурационных опций в ядре.

Однако, подход не лишен недостатков:

- техническая сложность реализации - необходимо выполнить сопутствующие задачи: поиск таблицы системных вызовов; обход защиты от модификации таблицы; атомарное и безопасное выполнение замены;
- невозможность перехвата некоторых обработчиков;
- перехватываются только системные вызовы.

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

1.2.3 Использование kprobes

Kprobes - специализированное API, в первую очередь предназначенное для отладки 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:

- хорошая документация API;
- перехват любого места в ядре - Kprobes реализуются с помощью точек останова (инструкции int3), внедряемых в исполнимый код ядра.

Недостатки kprobes:

• 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;

• накладные расходы из-за расстановки точек останова и их обработки;

• возможность переполнения буфера, хранящего адреса возврата, когда в системе выполняется слишком много одновременных вызовов перехваченной функции, в следствии чего будут пропускаться срабатывания отслеживаемых функций;

• в обработчиках нельзя ждать, выделять много памяти, спать в таймерах и семафорах и т.п. из-за отключенного вытеснения.

1.2.4 Сплайсинг

Сплайсинг – способ перехвата функций, основанный на замене инструкций в начале функции на безусловный переход, ведущий в обработчик. Исходные инструкции исполняются перед переходом обратно в перехваченную функцию. С помощью двух таких переходов выполняется дополнительный код. С помощью сплайсинга реализуются некоторые оптимизации kprobes.

Преимущества сплайсинга:

• сплайсинг не требует каких-либо особенных опций в ядре

• для перехвата нужно знать только адрес функции.

• воздействие не отлавливается антивирусами и детекторами вредоносных программ;

• обеспечение подключения любого символа ядра.

Недостатки:

• техническая сложность реализации, необходимо решить следующие задачи: синхронизация установки и снятия перехвата; обход защиты от модификации областей памяти с исходным кодом ядра; инвалидация кешей процессора после замены инструкци; дизассемблирование заменяемых инструкций, чтобы скопировать их целыми; проверка на отсутствие переходов внутрь заменяемого куска; проверка на возможность переместить заменяемый кусок в другое место.

1.2.5 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.

Для некоторых архитектур доступна оптимизация: динамический ftrace, который на ранних этапаха загрузки заменяет вызовы mcount() и __fentry__() на инструкцию 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:

- хорошо задокументированный интерфейс;
- перехват любой функции по имени;
- минимальное влияние на систему при выключенном трассировании;
- перехват совместим с трассировкой.

Недостатки:

- требования к конфигурации ядра - для успешного выполнения перехвата функций с помощью ftrace ядро должно предоставлять ряд возможностей: список символов kallsyms для поиска функций по имени; фреймворк ftrace в целом для выполнения трассировки; опции ftrace, критически важные для перехвата.
- ftrace срабатывает исключительно при входе.

1.3 Сравнительный анализ методов перехвата системных вызовов

В таблице 1.1 приведен сравнительный анализ рассмотренных методов.

Таблица 1.1 — Сравнительный анализ методов перехвата функций ядра Linux

	Документация API	Техническая простота реализации	Перехват любых функций	Перехват функции по имени
Linux Security API	-	+	+	-
Модификация таблицы системных вызовов	-	+	-	-
kprobes	+	-	+	-
Сплайсинг	-	-	+	-
ftrace	+	+	+	+

1.4 Выводы

В результате проведенного анализа подходов к реализации мониторинга за вызовами функций ядра, был выбран подход с ftrace. Первоначально был выбран способ с подменой таблицы системных вызовов. Однако, при реализации возникли следующие проблемы:

- после релиза ядра версии 3.0, таблицу больше нельзя получить простым импортом нужного модуля;
- 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.

Для реализации поставленной задачи, использовалось ядро 5.4.0, и выбран способ с использованием фреймворка ftrace. Подход поддерживает ядра версий 3.19+ для архитектуры x86_64.

2 Конструкторский раздел

На рисунке 2.1 показаны входные и выходные потоки данных, фреймворк и библиотека, необходимые для реализации поставленной задачи. IDEF0-диаграмма нулевого уровня:

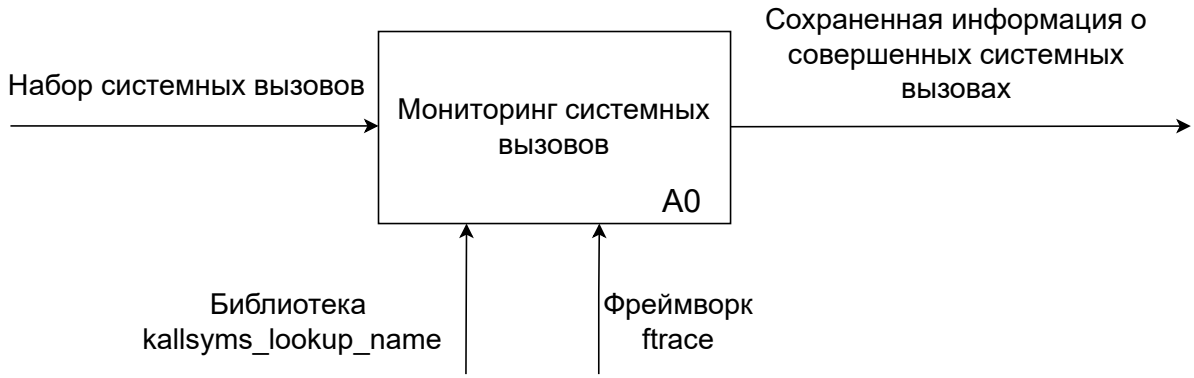

Рисунок 2.1 — IDEF0-диаграмма нулевого уровня

Загружаемый модуль ядра должен обеспечить выполнение следующей последовательности действий, изображенный на рисунке 2.2:

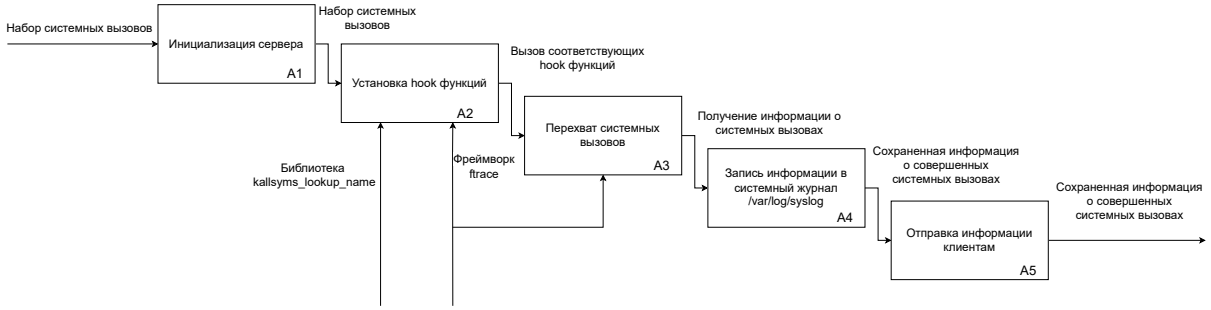

Рисунок 2.2 — IDEF0-диаграмма первого уровня

На первом этапе необходимо установить hook функцию. Далее поток системных вызовов перехватывается нашими hook функциями, и информация записывается в системный журнал /var/log/syslog.

2.1 Схемы алгоритмов

На рисунке 2.3 представлен алгоритм работы перехвата системного вызовы при использовании ftrace.

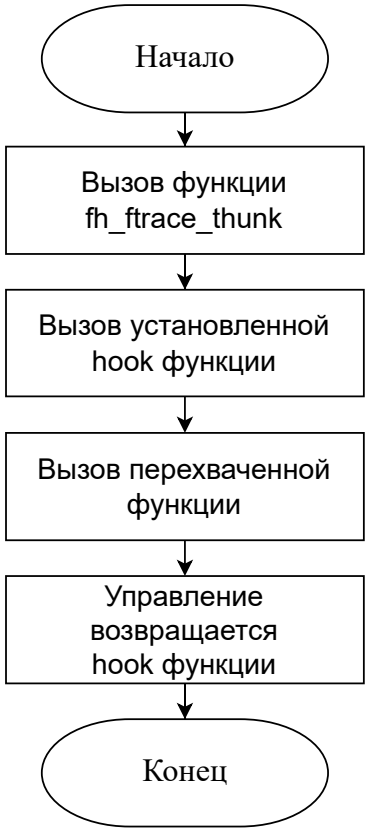

Рисунок 2.3 — Алгоритм перехвата функций

Пользовательский процесс вызывает макрос SYSCALL. Макрос переводит систему в режим ядра и управление передаётся низкоуровневому обработчику системных вызовов — entry_SYSCALL_64(). Управление переходит к конкретному обработчику. Ядро передаёт управление высокоуровневой функции do_syscall_64(). Эта функция в свою очередь обращается к таблице обработчиков системных вызовов sys_call_table и вызывает конкретный обработчик по номеру системного вызова. В начале каждой функции ядра находится вызов функции __fentry__(), которая выполняет защиту от зацикливания и реализуется фреймворком ftrace. Ftrace вызывает установленную callback-функцию, которая выполняет перехват. Далее управление с помощью безусловного перехода передаётся установленной обёрточной функции. При этом состояние процессора и памяти остаётся неизменным, поэтому данная функция получает все аргументы исходного системного вызова и при завершении вернёт управление в функцию do_syscall_64(). Обёрточная функция может проанализировать аргументы и контекст системного вызова и запретить или разрешить процессу его выполнение. В случае запрета функция просто возвращает код ошибки. Иначе происходит вызов исходной функции ядра, которая вызывается повторно, через

указатель, который был сохранён при настройке перехвата, после чего системный вызов завершается и управление передаётся пользовательскому процессу.

В результате работы обёрточной функции происходит повторный вызов функции ядра, который ftrace также обработает. Для того, чтобы избежать зацикливания при перехвате функции ядра необходимо проверять, был ли произведён вызов из текущего модуля, что выполняется функцией __fentry__(). В случае, когда адрес вызываемой стороны находится в пределах модуля обёрточная функция не будет вызвана и потому системный вызов отработает без перехвата. Алгоритм защиты от рекурсивных вызовов показан на рисунке 2.4

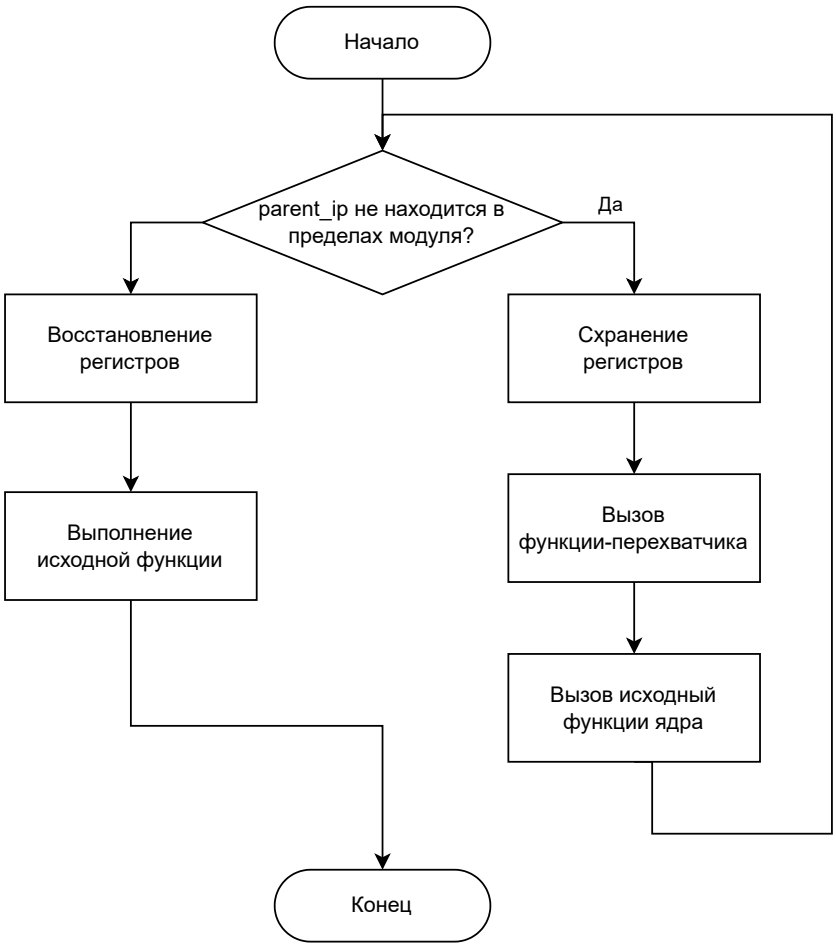

Рисунок 2.4 — Алгоритм защиты от зацикливания

2.2 Алгоритм запуска сервера

При загрузке модуля, выделяется память под структуры tcp_server_service и tcp_conn_handler и создается поток, слушающий, все приходящие соединения (листинг 2.5).

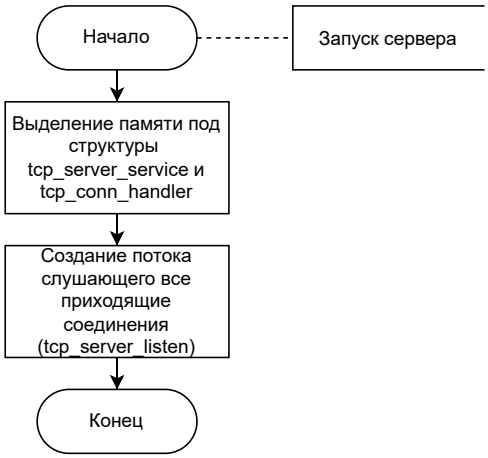

Рисунок 2.5 — Алгоритм запуска сервера

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 (листинг 2.6).

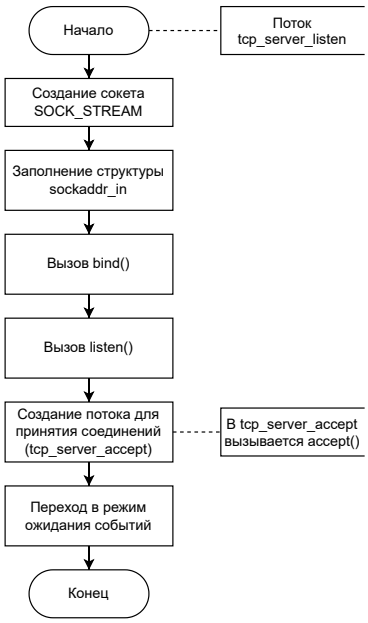

Рисунок 2.6 — Алгоритм потока слушающего все приходящие соединения

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, 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

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

2.3 Структура программного обеспечения

При срабатывании системного вызова из списка отслеживаемых, информация о нем записыфвается в системный журнал и отправляется подключенным клиентам с помощью сервера (рисунок 2.7). Таким образом, можно следить за вызовами функций ядра Linux удаленно.

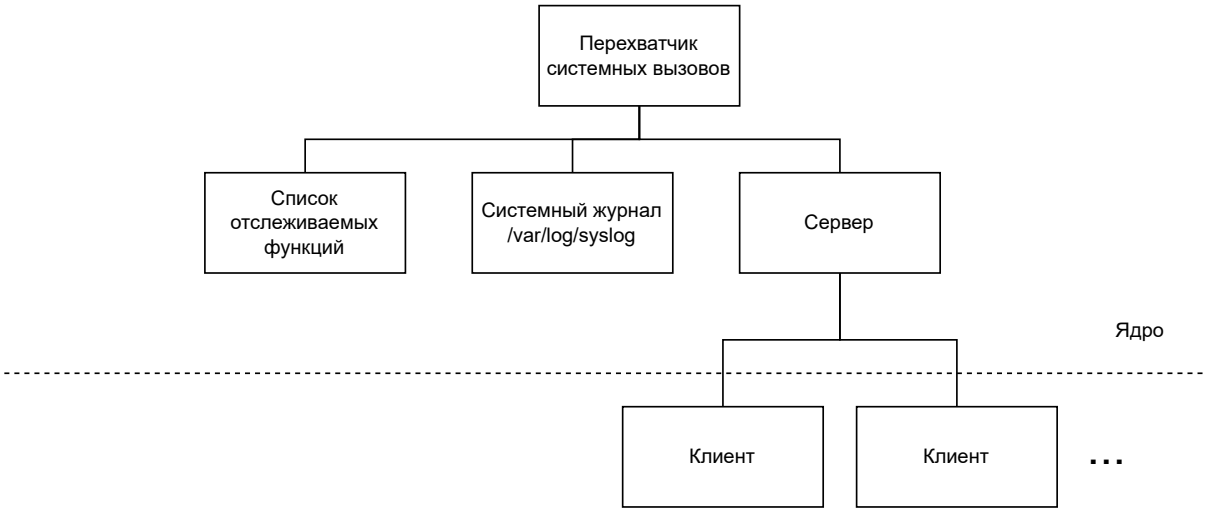

Рисунок 2.7 — Структура ПО

3 Технологический раздел

3.1 Выбор языка программирования и среды разработки

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, который имеется в любой современной ОС. В качестве среды разработки выбран VS Code.

3.2 Перехват функций

В листинге 3.1 представлена структура, которая описывает любую перехватываемую функцию.

Листинг 3.1 — ftrace_hook

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

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, как показано в листинге 3.2.

Листинг 3.2 — ftrace_hook define

```
1  #define HOOK(_name, _function, _original)   \
2  {                                            \
3      .name = (_name),                         \
4      .function = (_function),                 \
5      .original = (_original),                 \
6  }
7  static struct ftrace_hook hooked_functions[] = {
8      HOOK("sys_clone",  fh_sys_clone,  &real_sys_clone),
9  };
```

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.

3.2.1 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 показано в листинге 3.3:

Листинг 3.3 — resolve_hook_address

```
1  static int resolve_hook_address(struct ftrace_hook *hook)
2  {
3      hook->address = kallsyms_lookup_name(hook->name);
4      if (!hook->address) {
5          pr_debug("unresolved symbol: %s\n", hook->name);
6          return -ENOENT;
7      }
8      *((unsigned long*) hook->original) = hook->address;
9      return 0;
10 }
```

Дальше необходимо инициализировать структуру ftrace_ops. В ней обязательным полем является только func, указывающая на коллбек, но, кроме этого необходимо установить некоторые важные флаги (листинг 3.4).

Листинг 3.4 — fh_install_hook

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

Выключается перехват аналогично, только в обратном порядке (листинг 3.5).

Листинг 3.5 — fh_remove_hook

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

После завершения вызова unregister_ftrace_function() гарантируется отсутствие активаций установленного коллбека в системе. Поэтому модно выгрузить модуль-перехватчик, не опасаясь, что в системе ещё выполняются функции.

3.2.2 Выполнение перехвата функций

Ftrace позволяет изменять состояние регистров после выхода из коллбека. Изменяя регистр RIP (указатель на следующую исполняемую инструкцию) мы изменяются инструкции, которые исполняет процессор — то есть можно заставить его выполнить безусловный переход из текущей функции в нашу. Таким образом перехватывается управление. Коллбек для ftrace показан в листинге 3.6:

Листинг 3.6 — 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.

3.2.3 Схема работы перехвата

Рассмотрим пример: в 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:

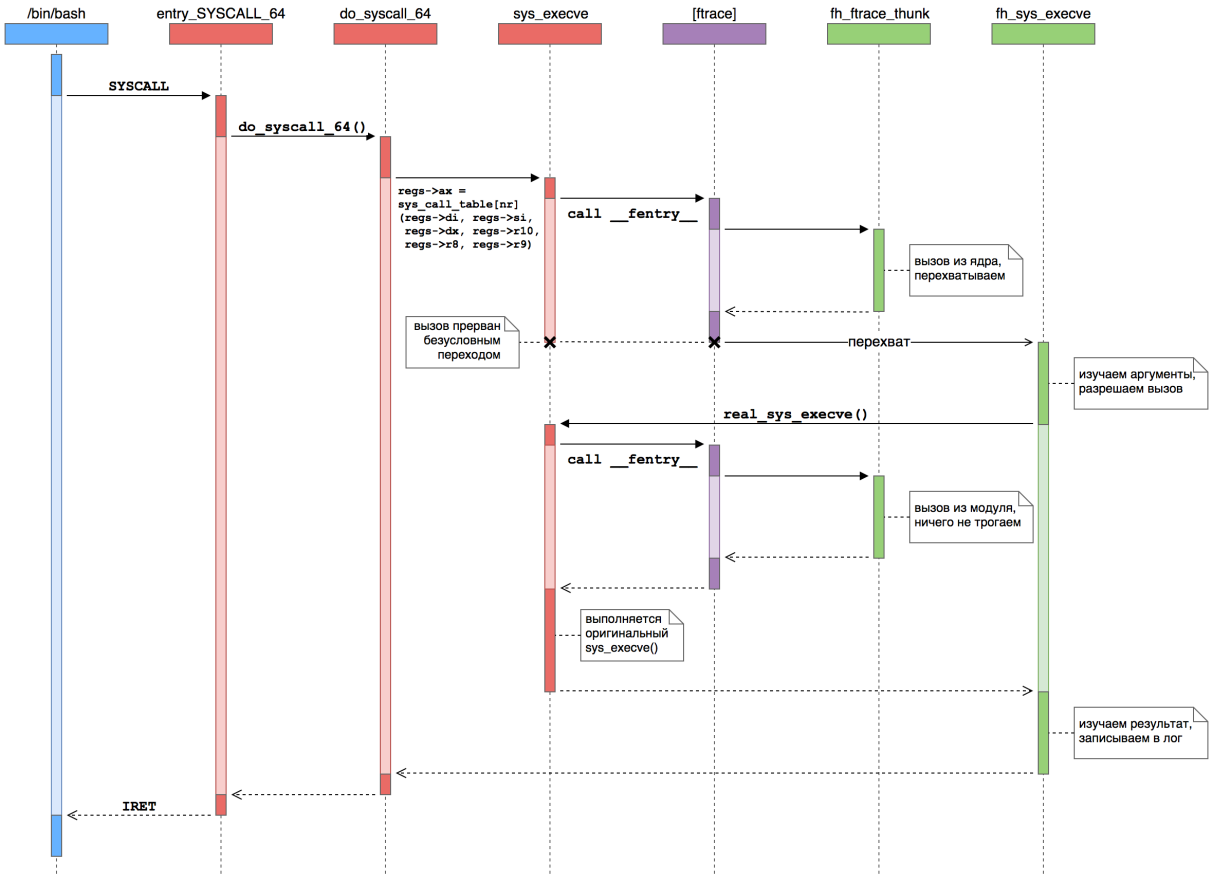

Рисунок 3.1 — Алгоритм работы функции-перехватчика

3.3 Сервер

Сервер описывается следующими структурами:

Листинг 3.7 — tcp_server_service

```
1   // Структура, описывающая и хранящие данные о текущем соединении для ка
        ждого 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 в
        сех 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  };
```

3.4 Установка модуля

Для компиляции реализованного модуля ядра необходимо запустить make-файл командой *make*. Make файл показан в листинге 3.8

Листинг 3.8 — Makefile

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

4 Исследовательский раздел

Технические характеристики устройства, на котором было проведено испытания разработанного модуля следующие: операционная система: Windows 10 (64-разрядная); оперативная память: 32 GB; процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz.

На рисунке 4.1 показан результат работы разработанного модуля после загрузки, выполнения команды ls и выгрузки:

```
arseny@arseny-VirtualBox:~/shared$ sudo insmod kern_monitor.ko
arseny@arseny-VirtualBox:~/shared$ ls
kern_monitor.c  kern_monitor.ko  kern_monitor.mod  kern_monitor.mod.c  kern_monitor.mod.o  kern_monitor.o  Makefile  mod
arseny@arseny-VirtualBox:~/shared$ sudo rmmod kern_monitor

arseny@arseny-VirtualBox:~/shared$ dmesg | tail -22
[ 1120.115868]  *** network_server initiated | network_server_init ***
[ 1120.117463]  *** creating the accept socket | tcp_server_accept ***
[ 1120.117469] accept_socket: 0000000015f564ce
[ 1120.167277] ftrace_hook: sys_clone() called: clone_flags=-82601882140840 newsp=0
[ 1120.167503] ftrace_hook: sys_clone() returns 5582
[ 1120.169209] ftrace_hook: sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696
[ 1120.169294] ftrace_hook: sys_clone() returns 5583
[ 1120.169441] ftrace_hook: sys_execve() called: filename=0000000072f84b72 argv=0000000072f84b72 envp=0000000000000000
[ 1120.169562] ftrace_hook: sys_execve() returns: 0
[ 1120.173583] ftrace_hook: sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696
[ 1120.173691] ftrace_hook: sys_clone() returns 5584
[ 1120.173792] ftrace_hook: sys_execve() called: filename=0000000072f84b72 argv=0000000072f84b72 envp=0000000000000000
[ 1120.173899] ftrace_hook: sys_execve() returns: 0
[ 1120.176890] ftrace_hook: sys_clone() called: clone_flags=-82601812918440 newsp=-82601812918440
[ 1120.177610] ftrace_hook: sys_clone() returns 5585
[ 1120.177700] ftrace_hook: sys_execve() called: filename=00000000bd64d9b3 argv=00000000bd64d9b3 envp=0000000000000000
[ 1120.177842] ftrace_hook: sys_execve() returns: 0
[ 1120.178962]  *** tcp server acceptor thread stopped | tcp_server_accept ***
[ 1120.179121]  *** tcp server acceptor thread stopped | network_server_exit ***
[ 1123.131588]  *** tcp server listening thread stopped | tcp_server_listen ***
[ 1123.132068]  *** tcp server listening thread stopped | network_server_exit ***
[ 1123.132086]  *** mtp | network server module unloaded | network_server_exit ***
arseny@arseny-VirtualBox:~/shared$
```

Рисунок 4.1 — Результат работы модуля

На рисунках 4.2 и 4.3 можно увидеть как клиент получает информацию о системных вызовах через сервер.

```
arseny@arseny-VirtualBox:~/shared$ sudo insmod kern_monitor.ko
arseny@arseny-VirtualBox:~/shared$ dmesg | tail -16
[ 1248.727159] connection from: 127.0.0.0 60836
[ 1248.727159] handle connection
[ 1248.727160] gave free id: 0
[ 1248.727410] accept_socket: 00000000a42fa5b3
[ 1256.028532] receiving message
[ 1256.028533] recv queue empty ? no
[ 1256.028538] client-> 127.0.0.0:60836, says: hello from client

[ 1261.195890] ftrace_hook: sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696
[ 1261.196052] ftrace_hook: sys_clone() returns 5602
[ 1261.196123] ftrace_hook: sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696
[ 1261.196243] ftrace_hook: sys_clone() returns 5603
[ 1261.196802] ftrace_hook: sys_execve() called: filename=00000000b9a524be argv=00000000b9a524be envp
=0000000000000000
[ 1261.196914] ftrace_hook: sys_execve() called: filename=0000000087357a46 argv=0000000087357a46 envp
=0000000000000000
[ 1261.197012] ftrace_hook: sys_execve() returns: 0
[ 1261.197057] ftrace_hook: sys_execve() returns: 0
arseny@arseny-VirtualBox:~/shared$ sudo rmmod kern_monitor
```

Рисунок 4.2 — Взаимодействия между сервером модуля и клиентом

```
arseny@arseny-VirtualBox:~$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello from client
sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696sys_clone() returns 5602
sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696sys_clone() returns 5603
sys_execve() called: filename=00000000b9a524be argv=00000000b9a524be envp=0000000000000000
sys_execve() called: filename=0000000087357a46 argv=0000000087357a46 envp=0000000000000000
sys_execve() returns: 0
sys_execve() returns: 0
sys_clone() called: clone_flags=-82601845014696 newsp=-82601845014696sys_clone() returns 5604
sys_execve() called: filename=00000000bf16c5e4 argv=00000000bf16c5e4 envp=0000000000000000
sys_execve() returns: 0
sys_clone() called: clone_flags=-82601801302184 newsp=-82601801302184sys_clone() returns 5605
sys_execve() called: filename=00000000a3cda9b3 argv=00000000a3cda9b3 envp=0000000000000000
sys_execve() returns: 0
Connection closed by foreign host.
arseny@arseny-VirtualBox:~$
```

Рисунок 4.3 — Взаимодействия между сервером модуля и клиентом

Заключение

В данной работе был реализован загружаемый модуль ядра операционной системы Linux, В процессе разработки 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

ПРИЛОЖЕНИЕ А

В Листинге А.1 приведен код разработанного модуля

Листинг А.1 — Код модуля

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/kthread.h>
#include <linux/ftrace.h>
#include <linux/kallsyms.h>
#include <linux/linkage.h>
#include <linux/uaccess.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/netdevice.h>
#include <linux/ip.h>
#include <linux/in.h>
#include <linux/unistd.h>
#include <linux/wait.h>
#include <net/tcp.h>
#include <net/inet_connection_sock.h>
#include <net/request_sock.h>
#define DEFAULT_PORT 8080
#define MAX_CONNS 16
#define MSG_LEN 64
#define MODULE_NAME "kern_monitor"
int tcp_listener_stopped = 0;
int tcp_acceptor_stopped = 0;
struct tcp_conn_handler_data {
    struct sockaddr_in *address;
    struct socket *accept_socket;
    int thread_id;
};
struct tcp_conn_handler {
    struct tcp_conn_handler_data *data[MAX_CONNS];
    struct task_struct *thread[MAX_CONNS];
    int tcp_conn_handler_stopped[MAX_CONNS];
};
struct tcp_conn_handler *tcp_conn_handler;
struct tcp_server_service {
    int running;
    struct socket *listen_socket;
    struct task_struct *thread;
    struct task_struct *accept_thread;
};
```

```
43 struct tcp_server_service *tcp_server;
44 char *inet_ntoa(struct in_addr *in)
45 {
46     char *str_ip = NULL;
47     u_int32_t int_ip = 0;
48     if (!(str_ip = kmalloc(16 * sizeof(char), GFP_KERNEL)))
49         return NULL;
50     else
51         memset(str_ip, 0, 16);
52     int_ip = in->s_addr;
53     sprintf(str_ip, "%d.%d.%d.%d", (int_ip) & 0xFF, (int_ip >> 8) & 0xFF,
           (int_ip >> 16) & 0xFF, (int_ip >> 16) & 0xFF);
54     return str_ip;
55 }
56 int tcp_server_send(struct socket *sock, int id, const char *buf, const
       size_t length, unsigned long flags)
57 {
58     struct msghdr msg;
59     struct kvec vec;
60     int len, written = 0, left = length;
61     mm_segment_t oldmm;
62     msg.msg_name    = 0;
63     msg.msg_namelen = 0;
64     msg.msg_control = NULL;
65     msg.msg_controllen = 0;
66     msg.msg_flags = flags;
67     msg.msg_flags   = 0;
68     oldmm = get_fs();
69     set_fs(KERNEL_DS);
70     while (1) {
71         vec.iov_len = left;
72         vec.iov_base = (char *)buf + written;
73         len = kernel_sendmsg(sock, &msg, &vec, left, left);
74         if ((len == -ERESTARTSYS) || (!(flags & MSG_DONTWAIT) && (len ==
               -EAGAIN)))
75             continue;
76         if (len > 0) {
77             written += len;
78             left -= len;
79             if (!left)
80                 break;
81         }
82     }
83     set_fs(oldmm);
84     return written ? written : len;
85 }
```

```
86  int tcp_server_receive(struct socket *sock, int id, struct sockaddr_in
        *address, unsigned char *buf, int size, unsigned long flags)
87  {
88      struct msghdr msg;
89      struct kvec vec;
90      int len;
91      char *tmp = NULL;
92      if (sock == NULL) {
93          pr_info(" *** tcp server receive socket is NULL |
                tcp_server_receive *** \n");
94          return -1;
95      }
96      msg.msg_name = 0;
97      msg.msg_namelen = 0;
98      msg.msg_control = NULL;
99      msg.msg_controllen = 0;
100     msg.msg_flags = flags;
101     vec.iov_len = size;
102     vec.iov_base = buf;
103     while (1) {
104         if (!skb_queue_empty(&sock->sk->sk_receive_queue))
105             pr_info("recv queue empty ? %s \n",
                    skb_queue_empty(&sock->sk->sk_receive_queue) ? "yes" :
                    "no");
106         len = kernel_recvmsg(sock, &msg, &vec, size, size, flags);
107         if (len != -EAGAIN && len != -ERESTARTSYS)
108             break;
109     }
110     tmp = inet_ntoa(&(address->sin_addr));
111     pr_info("client-> %s:%d, says: %s\n", tmp, ntohs(address->sin_port),
            buf);
112     kfree(tmp);
113     return len;
114 }
115 int connection_handler(void *data)
116 {
117     struct tcp_conn_handler_data *conn_data = (struct
            tcp_conn_handler_data *)data;
118     struct sockaddr_in *address = conn_data->address;
119     struct socket *accept_socket = conn_data->accept_socket;
120     int id = conn_data->thread_id;
121     int ret;
122     unsigned char in_buf[MSG_LEN];
123     DECLARE_WAITQUEUE(recv_wait, current);
124     allow_signal(SIGKILL | SIGSTOP);
125     while (1) {
126         add_wait_queue(&accept_socket->sk->sk_wq->wait, &recv_wait);
```

```
127         while (skb_queue_empty(&accept_socket->sk->sk_receive_queue)) {
128             __set_current_state(TASK_INTERRUPTIBLE);
129             schedule_timeout(HZ);
130             if (kthread_should_stop()) {
131                 pr_info(" *** tcp server handle connection thread stopped
                        | connection_handler *** \n");
132                 tcp_conn_handler->tcp_conn_handler_stopped[id] = 1;
133                 __set_current_state(TASK_RUNNING);
134                 remove_wait_queue(&accept_socket->sk->sk_wq->wait,
                        &recv_wait);
135                 kfree(tcp_conn_handler->data[id]->address);
136                 kfree(tcp_conn_handler->data[id]);
137                 sock_release(tcp_conn_handler->data[id]->accept_socket);
138                 return 0;
139             }
140             if (signal_pending(current)) {
141                 __set_current_state(TASK_RUNNING);
142                 remove_wait_queue(&accept_socket->sk->sk_wq->wait,
                        &recv_wait);
143                 goto release;
144             }
145         }
146         __set_current_state(TASK_RUNNING);
147         remove_wait_queue(&accept_socket->sk->sk_wq->wait, &recv_wait);
148         pr_info("receiving message\n");
149         memset(in_buf, 0, MSG_LEN);
150         ret = tcp_server_receive(accept_socket, id, address, in_buf,
                MSG_LEN, MSG_DONTWAIT);
151         if (ret > 0)
152             if (memcmp(in_buf, "done", 4) == 0) {
153                     memset(in_buf, 0, MSG_LEN);
154                     strcat(in_buf, "Connection closed... See you again");
155                     tcp_server_send(accept_socket, id, in_buf,
                            strlen(in_buf), MSG_DONTWAIT);
156                     break;
157             }
158     }
159 release:
160     tcp_conn_handler->tcp_conn_handler_stopped[id] = 1;
161     kfree(tcp_conn_handler->data[id]->address);
162     kfree(tcp_conn_handler->data[id]);
163     tcp_conn_handler->data[id] = NULL;
164     sock_release(tcp_conn_handler->data[id]->accept_socket);
165     tcp_conn_handler->thread[id] = NULL;
166     do_exit(0);
167 }
168 int tcp_server_accept(void)
```

```
169 {
170     struct socket *socket;
171     struct socket *accept_socket = NULL;
172     struct inet_connection_sock *isock;
173     int accept_err = 0, id = 0;
174     DECLARE_WAITQUEUE(accept_wait, current);
175     allow_signal(SIGKILL | SIGSTOP);
176     socket = tcp_server->listen_socket;
177     pr_info(" *** creating the accept socket | tcp_server_accept *** \n");
178     while (1) {
179         struct tcp_conn_handler_data *data = NULL;
180         struct sockaddr_in *client = NULL;
181         char *tmp;
182         int addr_len;
183         accept_err = sock_create(socket->sk->sk_family, socket->type,
                socket->sk->sk_protocol, &accept_socket);
184         pr_info("accept_socket: %p\n", accept_socket);
185         if (accept_err < 0 || !accept_socket) {
186             pr_info(" *** accept_error: %d while creating tcp server
                    accept socket | tcp_server_accept *** \n", accept_err);
187             goto err;
188         }
189         accept_socket->type = socket->type;
190         accept_socket->ops  = socket->ops;
191         isock = inet_csk(socket->sk);
192         add_wait_queue(&socket->sk->sk_wq->wait, &accept_wait);
193         while (reqsk_queue_empty(&isock->icsk_accept_queue)) {
194             __set_current_state(TASK_INTERRUPTIBLE);
195             schedule_timeout(HZ);
196             if (kthread_should_stop()) {
197                 pr_info(" *** tcp server acceptor thread stopped |
                        tcp_server_accept *** \n");
198                 tcp_acceptor_stopped = 1;
199                 __set_current_state(TASK_RUNNING);
200                 remove_wait_queue(&socket->sk->sk_wq->wait, &accept_wait);
201                 sock_release(accept_socket);
202                 return 0;
203             }
204             if (signal_pending(current)) {
205                 __set_current_state(TASK_RUNNING);
206                 remove_wait_queue(&socket->sk->sk_wq->wait, &accept_wait);
207                 goto release;
208             }
209         }
210         __set_current_state(TASK_RUNNING);
211         remove_wait_queue(&socket->sk->sk_wq->wait, &accept_wait);
212         pr_info("accept connection\n");
```

```
213         accept_err = socket->ops->accept(socket, accept_socket,
                O_NONBLOCK, false);
214         if (accept_err < 0) {
215             pr_info(" *** accept_error: %d while accepting tcp server |
                    tcp_server_accept *** \n", accept_err);
216             goto release;
217         }
218         client = kmalloc(sizeof(struct sockaddr_in), GFP_KERNEL);
219         memset(client, 0, sizeof(struct sockaddr_in));
220         addr_len = sizeof(struct sockaddr_in);
221         accept_err = accept_socket->ops->getname(accept_socket, (struct
                sockaddr *)client, addr_len);
222         if (accept_err < 0) {
223             pr_info(" *** accept_error: %d in getname tcp server |
                    tcp_server_accept *** \n", accept_err);
224             goto release;
225         }
226         tmp = inet_ntoa(&(client->sin_addr));
227         pr_info("connection from: %s %d \n", tmp, ntohs(client->sin_port));
228         kfree(tmp);
229         pr_info("handle connection\n");
230         for (id = 0; id < MAX_CONNS; id++)
231             if (tcp_conn_handler->thread[id] == NULL)
232                 break;
233         pr_info("gave free id: %d\n", id);
234         if (id == MAX_CONNS) {
235             goto release;
236         }
237         data = kmalloc(sizeof(struct tcp_conn_handler_data), GFP_KERNEL);
238         memset(data, 0, sizeof(struct tcp_conn_handler_data));
239         data->address = client;
240         data->accept_socket = accept_socket;
241         data->thread_id = id;
242         tcp_conn_handler->tcp_conn_handler_stopped[id] = 0;
243         tcp_conn_handler->data[id] = data;
244         tcp_conn_handler->thread[id] = kthread_run((void
                *)connection_handler, (void *)data, MODULE_NAME);
245         if (kthread_should_stop()) {
246             pr_info(" *** tcp server acceptor thread stopped |
                    tcp_server_accept *** \n");
247             tcp_acceptor_stopped = 1;
248             return 0;
249         }
250         if (signal_pending(current)) {
251             break;
252         }
253     }
```

```
254 release:
255     sock_release(accept_socket);
256 err:
257     tcp_acceptor_stopped = 1;
258     do_exit(0);
259 }
260 int tcp_server_listen(void)
261 {
262     int server_err;
263     struct socket *conn_socket;
264     struct sockaddr_in server;
265     DECLARE_WAIT_QUEUE_HEAD(wq);
266     allow_signal(SIGKILL | SIGTERM);
267     if ((server_err = sock_create(PF_INET, SOCK_STREAM, IPPROTO_TCP,
        &tcp_server->listen_socket)) < 0) {
268         pr_info(" *** Error: %d while creating tcp server listen socket |
            tcp_server_listen *** \n", server_err);
269         goto err;
270     }
271     conn_socket = tcp_server->listen_socket;
272     tcp_server->listen_socket->sk->sk_reuse = 1;
273     server.sin_addr.s_addr = INADDR_ANY;
274     server.sin_family = AF_INET;
275     server.sin_port = htons(DEFAULT_PORT);
276     if ((server_err = conn_socket->ops->bind(conn_socket, (struct
        sockaddr*)&server, sizeof(server))) < 0) {
277         pr_info(" *** Error: %d while binding tcp server listen socket |
            tcp_server_listen *** \n", server_err);
278         goto release;
279     }
280     if ((server_err = conn_socket->ops->listen(conn_socket, MAX_CONNS)) <
        0) {
281         pr_info(" *** Error: %d while listening in tcp server listen
            socket | tcp_server_listen *** \n", server_err);
282         goto release;
283     }
284     tcp_server->accept_thread = kthread_run((void*)tcp_server_accept,
        NULL, MODULE_NAME);
285     while (1) {
286         wait_event_timeout(wq, 0, 3 * HZ);
287         if (kthread_should_stop()) {
288             pr_info(" *** tcp server listening thread stopped |
                tcp_server_listen *** \n");
289             return 0;
290         }
291         if (signal_pending(current)) {
292             break;
```

```
293         }
294     }
295 release:
296     sock_release(conn_socket);
297 err:
298     tcp_listener_stopped = 1;
299     do_exit(0);
300 }
301 int tcp_server_start(void)
302 {
303     tcp_server->running = 1;
304     tcp_server->thread = kthread_run((void *)tcp_server_listen, NULL,
        MODULE_NAME);
305     return 0;
306 }
307 static int network_server_init(void)
308 {
309     pr_info(" *** network_server initiated | network_server_init ***\n");
310     tcp_server = kmalloc(sizeof(struct tcp_server_service), GFP_KERNEL);
311     memset(tcp_server, 0, sizeof(struct tcp_server_service));
312     tcp_conn_handler = kmalloc(sizeof(struct tcp_conn_handler),
        GFP_KERNEL);
313     memset(tcp_conn_handler, 0, sizeof(struct tcp_conn_handler));
314     tcp_server_start();
315     return 0;
316 }
317 static void network_server_exit(void)
318 {
319     int ret;
320     int id;
321     if (tcp_server->thread == NULL)
322         pr_info(" *** No kernel thread to kill | network_server_exit ***
            \n");
323     else {
324         for (id = 0; id < MAX_CONNS; id++) {
325             if (tcp_conn_handler->thread[id] != NULL) {
326                 if (!tcp_conn_handler->tcp_conn_handler_stopped[id]) {
327                     if (!(ret =
                        kthread_stop(tcp_conn_handler->thread[id]))) {
328                         pr_info(" tcp server connection handler thread: %d
                            stopped | network_server_exit *** \n", id);
329                     }
330                 }
331             }
332         }
333         if (!tcp_acceptor_stopped) {
334             if (!(ret = kthread_stop(tcp_server->accept_thread)))
```

```
335                 pr_info(" *** tcp server acceptor thread stopped |
                        network_server_exit *** \n");
336         }
337         if (!tcp_listener_stopped) {
338             if (!(ret = kthread_stop(tcp_server->thread)))
339                 pr_info(" *** tcp server listening thread stopped |
                        network_server_exit *** \n");
340         }
341         if (tcp_server->listen_socket != NULL && !tcp_listener_stopped) {
342             sock_release(tcp_server->listen_socket);
343             tcp_server->listen_socket = NULL;
344         }
345         kfree(tcp_conn_handler);
346         kfree(tcp_server);
347         tcp_server = NULL;
348     }
349     pr_info(" *** mtp | network server module unloaded |
            network_server_exit *** \n");
350 }
351 #define HOOK(_name, _function, _original)   \
352     {                       \
353             .name = (_name),        \
354         .fake_func = (_function),   \
355         .orig_func = (_original),   \
356     }
357 struct ftrace_hook {
358     const char *name;
359     void *fake_func;
360     void *orig_func;
361     unsigned long address;
362     struct ftrace_ops ops;
363 };
364 #define USE_FENTRY_OFFSET 0
365 #define pr_fmt(fmt) "ftrace_hook: " fmt
366 static int fh_resolve_hook_address(struct ftrace_hook *hook)
367 {
368     if (!(hook->address = kallsyms_lookup_name(hook->name))) {
369         pr_debug("unresolved symbol: %s\n", hook->name);
370         return -ENOENT;
371     }
372 #if USE_FENTRY_OFFSET
373     *((unsigned long*) hook->orig_func) = hook->address + MCOUNT_INSN_SIZE;
374 #else
375     *((unsigned long*) hook->orig_func) = hook->address;
376 #endif
377     return 0;
378 }
```

```
379 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long
        parent_ip,
380         struct ftrace_ops *ops, struct pt_regs *regs)
381 {
382     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
383 #if USE_FENTRY_OFFSET
384     regs->ip = (unsigned long) hook->fake_func;
385 #else
386     if (!within_module(parent_ip, THIS_MODULE)) {
387         regs->ip = (unsigned long) hook->fake_func;
388         }
389 #endif
390 }
391 int fh_install_hook(struct ftrace_hook *hook)
392 {
393     int err;
394     if ((err = fh_resolve_hook_address(hook))) {
395         return err;
396         }
397     hook->ops.func = fh_ftrace_thunk;
398     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
399                     | FTRACE_OPS_FL_RECURSION_SAFE
400                     | FTRACE_OPS_FL_IPMODIFY;
401     if ((err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0))) {
402         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
403         return err;
404     }
405     if ((err = register_ftrace_function(&hook->ops))) {
406         pr_debug("register_ftrace_fake_func() failed: %d\n", err);
407         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
408         return err;
409     }
410     return 0;
411 }
412 void fh_remove_hook(struct ftrace_hook *hook)
413 {
414     int err;
415     if ((err = unregister_ftrace_function(&hook->ops))) {
416         pr_debug("unregister_ftrace_fake_func() failed: %d\n", err);
417     }
418     if ((err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0))) {
419         pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
420     }
421 }
422 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
423 {
424     int err;
```

```
425     size_t i;
426     for (i = 0; i < count; i++) {
427         err = fh_install_hook(&hooks[i]);
428         if (err) {
429                         while (i != 0) {
430                         fh_remove_hook(&hooks[--i]);
431                     }
432             break;
433                 }
434     }
435     return err;
436 }
437 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
438 {
439     size_t i;
440     for (i = 0; i < count; i++) {
441         fh_remove_hook(&hooks[i]);
442         }
443 }
444 #ifndef CONFIG_X86_64
445 #error Currently only x86_64 architecture is supported
446 #endif
447 #if !USE_FENTRY_OFFSET
448 #pragma GCC optimize("-fno-optimize-sibling-calls")
449 #endif
450 static void generic_send(unsigned char *msg)
451 {
452         int id;
453         for (id = 0; id < MAX_CONNS; ++id) {
454                 struct tcp_conn_handler_data *data =
                        tcp_conn_handler->data[id];
455                 if (data != NULL) {
456                         tcp_server_send(data->accept_socket, id, msg,
                                strlen(msg), MSG_DONTWAIT);
457                 }
458         }
459 }
460 static asmlinkage long *(real_sys_execve)(const char __user *filename,
    const char __user *const __user *argv, const char __user *const __user
    *envp);
461 static asmlinkage long fh_sys_clone(unsigned long clone_flags, unsigned
    long newsp, int __user *parent_tidptr, int __user *child_tidptr,
    unsigned long tls)
462 {
463     long ret;
464         unsigned char out_buf[MSG_LEN];
465         memset(out_buf, 0, MSG_LEN);
```

```
466         sprintf(out_buf, "clone() before\n");
467     pr_info("clone() before\n");
468         generic_send(out_buf);
469     ret = real_sys_clone(clone_flags, newsp, parent_tidptr, child_tidptr,
        tls);
470
471         memset(out_buf, 0, MSG_LEN);
472         sprintf(out_buf, "clone() after: %ld\n", ret);
473     pr_info("clone() after: %ld\n", ret);
474         generic_send(out_buf);
475     return ret;
476 }
477 static struct ftrace_hook demo_hooks[] = {
478     HOOK("__x64_sys_clone", fh_sys_clone, &real_sys_clone),
479 };
480 static int fh_init(void)
481 {
482         return fh_install_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
483 }
484 static void fh_exit(void)
485 {
486     fh_remove_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
487 }
488 static int __init kern_monitor_init(void)
489 {
490         int err;
491         if ((err = network_server_init())) {
492                 return err;
493         }
494         if ((err = fh_init())) {
495                 network_server_exit();
496                 return err;
497         }
498         return 0;
499 }
500 static void __exit kern_monitor_exit(void)
501 {
502         network_server_exit();
503         fh_exit();
504 }
505 module_init(kern_monitor_init)
506 module_exit(kern_monitor_exit)
507 MODULE_LICENSE("GPL");
508 MODULE_AUTHOR("Pronin Arseny");
```