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 ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Загружаемый модуль ядра, позволяющий скрывать файлы или запрещать их изменение, чтение и удаление»

Студент ИУ7-74Б _____ Д.А. Татаринова
(группа) (Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ Н.Ю.Рязанова
(Подпись, дата) (И.О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Обеспечение безопасного доступа к файлам в операционной системе Linux является актуальной задачей. При работе с файлами в Linux необходимо обеспечивать конфиденциальность данных и предоставлять защиту от вредоносных действий.

Данная курсовая работа посвящена разработке модуля, позволяющего ограничивать доступ к определенным файлам.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра для ОС Linux, позволяющий скрывать файлы или запрещать их изменение, чтение и удаление. Предусмотреть возможность ввода пароля для отображения файлов или разрешения операций над ними. Предоставить пользователю возможность задавать список таких файлов.

Для решения поставленной задачи необходимо:

— проанализировать возможности перехвата функций ядра Linux;

— выбрать системные вызовы, которые необходимо перехватить;

— разработать алгоритм перехвата, алгоритмы hook–функций и структуру программного обеспечения;

— реализовать программное обеспечение;

— исследовать работу ПО.

1.2 Способы перехвата функций ядра

1.2.1 ftrace

ftrace предоставляет возможности для трассировки функций. С его помощью можно отслеживать контекстные переключения, измерять время обработки прерываний, высчитывать время на активизацию заданий с высоким приоритетом и многое другое [3].

Ftrace был разработан Стивеном Ростедтом и добавлен в ядро в 2008 году, начиная с версии 2.6.27. Ftrace — фреймворк, предоставляющий отладочный

кольцевой буфер для записи данных. Собирают эти данные встроенные в ядро программы–трассировщики [3].

Работает ftrace на базе файловой системы debugfs, которая в большинстве современных дистрибутивов Linux смонтирована по умолчанию.

Каждую перехватываемую функцию можно описать следующей структурой:

Листинг 1.1 — Структура ftrace_hook

```
1     struct ftrace_hook {
2         const char *name;
3         void *function;
4         void *original;
5
6         unsigned long address;
7         struct ftrace_ops ops;
8     };
```

Поля структуры:

name — имя перехватываемой функции;

function — адрес функции–обертки, которая будет вызываться вместо перехваченной функции;

original — указатель на место, куда следует записать адрес перехватываемой функции, заполняется при установке;

address — адрес перехватываемой функции, заполняется при установке;

ops — служебная информация ftrace.

Листинг 1.2 — Пример заполнения структуры ftrace_hook

```
1     #define HOOK(_name, _function, _original)       \
2     {                                               \
3         .name = (_name),                            \
4         .function = (_function),                    \
5         .original = (_original),                    \
```

```
6       }
7
8       static struct ftrace_hook hooked_functions[] = {
9           HOOK("sys_clone",   fh_sys_clone,   &real_sys_clone),
10          HOOK("sys_execve",  fh_sys_execve,  &real_sys_execve),
11      };
```

1.2.2 kprobes

Kprobes — 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.

Kprobes реализуются с помощью точек останова (инструкции int3), внедряемых в исполнимый код ядра, что позволяет устанавливать kprobes в любом месте любой функции, если оно известно. Аналогично, kretprobes реализуются через подмену адреса возврата на стеке и позволяют перехватить возврат из любой функции.

При использовании kprobes для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах или где на стеке они лежат, и самостоятельно их оттуда извлекать. Для решения данной проблемы существует jprobes — надстройка над kprobes, самостоятельно извлекающая аргументы функции из регистров или стека и вызывающая обработчик, который должен иметь ту же сигнатуру, что и перехватываемая функция. Однако jprobes объявлен устаревшим и удален из современных ядер.

1.2.3 Linux Security API

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

Для Linux Security API характерны следующие ограничения:

— security–модули не могут быть загружены динамически, они являются частью ядра и требуют его перекомпиляции;

— в системе может быть только один security–модуль.

Таким образом, для использования Security API необходимо поставлять собственную сборку ядра, а также интегрировать дополнительный модуль с SELinux или AppArmor, которые используются популярными дистрибутивами.

1.2.4 Модификация таблицы системных вызовов

В ядре Linux все обработчики системных вызовов хранятся в таблице sys_call_table [1]. Подмена значений в этой таблице приводит к смене поведения всей системы. Таким образом, сохранив старое значения обработчика и подставив в таблицу собственный обработчик, можно перехватить системный вызов.

Однако данный подход обладает следующими недостатками:

— Техническая сложность реализации, заключающаяся в необходимости обхода защиты от модификации таблицы, атомарное и безопасное выполнение замены.

— Невозможность перехвата некоторых обработчиков. В ядрах до версии 4.16 обработка системных вызовов для архитектуры x86_64 содержала целый ряд оптимизаций. Некоторые из них требовали того, что обработчик системного вызова являлся специальным переходником, реализованным на ассемблере. Соответственно, подобные обработчики порой сложно, а иногда и вовсе невозможно заменить на собственные, написанные на Си [2].

1.2.5 Сплайсинг

Сплайсинг заключается в замене инструкций в начале функции на безусловный переход, ведущий в обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. Именно таким образом реализуется jump–оптимизация для kprobes. Используя сплайсинг, можно добиться тех же результатов, но без дополнительных расходов на kprobes и с полным контролем ситуации.

Сложность использования сплайсинга заключается в необходимости синхронизации установки и снятия перехвата, обхода защиты от модификации регионов памяти с кодом, инвалидации кешей процессора после замены инструкций, дизассемблировании заменяемых инструкций и проверки на отсутствие переходов внутрь заменяемого кода.

1.2.6 Сравнительный анализ способов перехвата

Результаты сравнения различных способов перехвата функций ядра представлены в таблице 1.1

Таблица 1.1 — Результаты сравнения

Способ перехвата	Необходимость перекомпиляции ядра	Возможность перехвата любых обработчиков	Доступ к аргументам функции через переменные	Необходимость обхода защиты от модификации регионов памяти
Linux Security API	да	нет	да	нет
Модификация таблиц системных вызовов	нет	нет	да	да
kprobes	нет	да	нет	нет
Сплайсинг	нет	да	да	да
ftrace	нет	да	да	нет

1.3 Перехватываемые функции ядра

1.3.1 Функция getdents64

Для того, чтобы скрыть файл необходимо перехватить функцию ядра getdents64, так как она возвращает записи каталога.

Листинг 1.3 — Функции getdents64

```
1 #include <fcntl.h>
2
3 int getdents64(unsigned int fd, struct linux_dirent64 *dirp, unsigned int
      count);
```

Системный вызов getdents64 читает несколько структур linux_dirent64 из каталога, на который указывает открытый файловый дескриптор fd, в буфер, указанный в dirp. В аргументе count задается размер этого буфера.

Структура linux_dirent64 определена следующим образом:

Листинг 1.4 — Структура linux_dirent64

```
1 struct linux_dirent64 {
2         ino64_t      d_ino;
3         off64_t      d_off;
4         unsigned short  d_reclen;
5         unsigned char   d_type;
6         char     d_name[];
7 };
```

В d_ino указан номер inode. В d_off задается расстояние от начала каталога до начала следующей linux_dirent64. В d_reclen указывается размер данного linux_dirent64. В d_name задается имя файла, в d_type — тип файла.

Таким образом, перехват системного вызова getdents64 позволяет удалить запись из списка записей каталога.

1.3.2 Функция unlink

Удаление записей из каталога производится с помощью функции unlink.

Листинг 1.5 — Функция unlink

```
1 #include <unistd.h>
2
3 int unlink(const char *pathname);
```

Эта функция удаляет запись из файла каталога и уменьшает значение счетчика ссылок на файл pathname. Если на файл указывает несколько ссылок, то его содержимое будет через них по–прежнему доступно.

Таким образом, для того, чтобы запретить удаление файла, необходимо перехватить системный вызов unlink.

1.3.3 Функции open, write, read

Запись в файл осуществляется при помощи системного вызова write, чтение из файла — read.

Листинг 1.6 — Функции write и read

```
1 #include <fcntl.h>
2
3 ssize_t write(int fd, const void *buf, size_t size);
4 ssize_t read(int fd, void *buf, size_t count);
```

Функции write и read не позволяют получить имя файла, так как они работают только с файловым дескриптором. Однако для того, чтобы определить, можно ли выполнить операцию с файлом, необходимо знать его имя. Тогда следует перехватить функцию open, которая позволяет получить имя файла.

Системный вызов open осуществляет открытие файла.

Листинг 1.7 — Функции open и write

```
1 #include <fcntl.h>
2
3 int open(int dirfd, const char *pathname, int flags);
4 int open(int dirfd, const char *pathname, int flags, mode_t mode);
```

При перехвате open необходимо сохранить идентификатор процесса, открывающего файл. Затем при перехвате write и read ориентироваться не на имя файла, а на идентификатор процесса и файловый дескриптор.

Таким образом, для того, чтобы запретить чтение из файла и запись в него, необходимо перехватить системные вызовы open, write и read.

Вывод

В результате сравнительного анализа выбран способ перехвата функций ядра — ftrace, так как он позволяет перехватывать любые функции ядра и не требует его перекомпиляции. Для сокрытия файла необходимо перехватить функцию getdents64, для запрета чтения из файла и записи в файл — функции open, read и write, удаления — unlink().

2 Конструкторский раздел

2.1 IDEF0

На рисунке 2.1 приведена диаграмма состояний IDEF0 нулевого уровня, а на рисунке 2.2 — диаграмма состояний IDEF0 первого уровня.

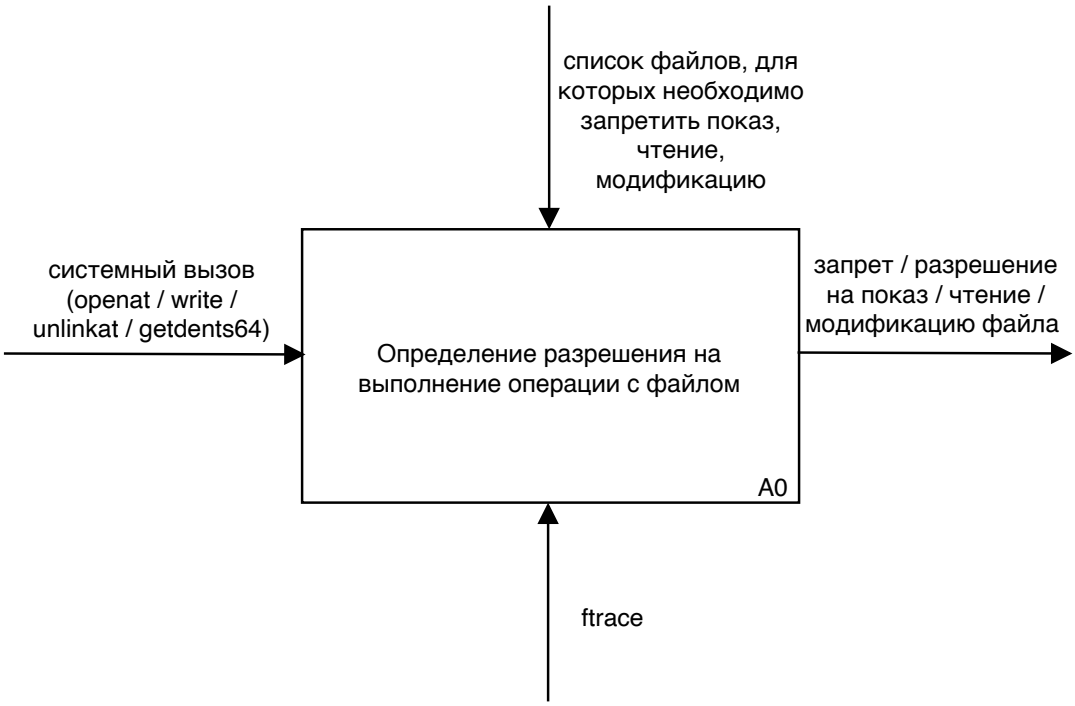

Рисунок 2.1 — Диаграмма состояний IDEF0 нулевого уровня

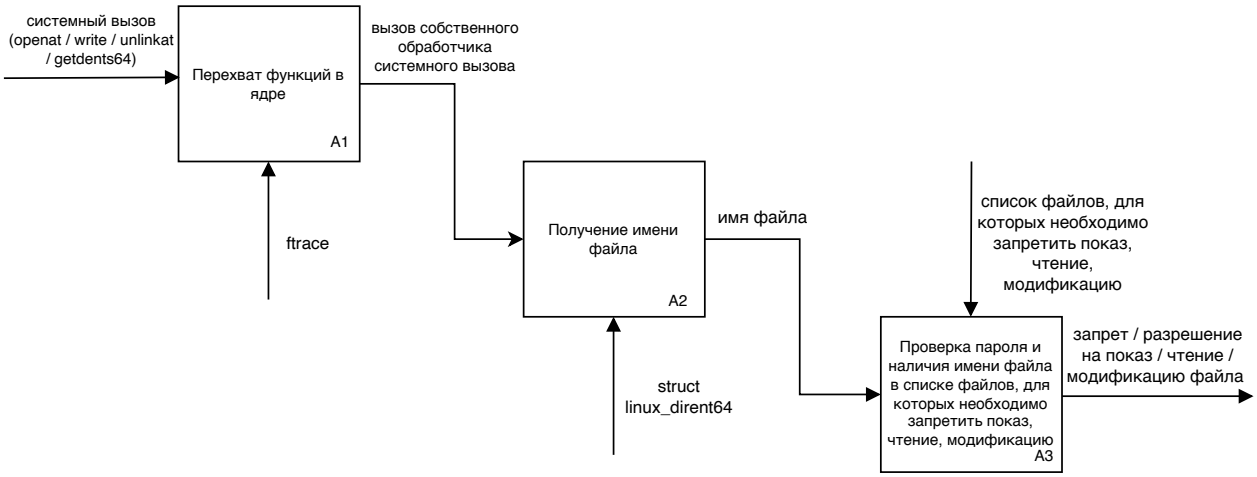

Рисунок 2.2 — Диаграмма состояний IDEF0 первого уровня

2.2 Алгоритм создания символьного устройства

На рисунке 2.3 приведена схема создания символьного устройства.

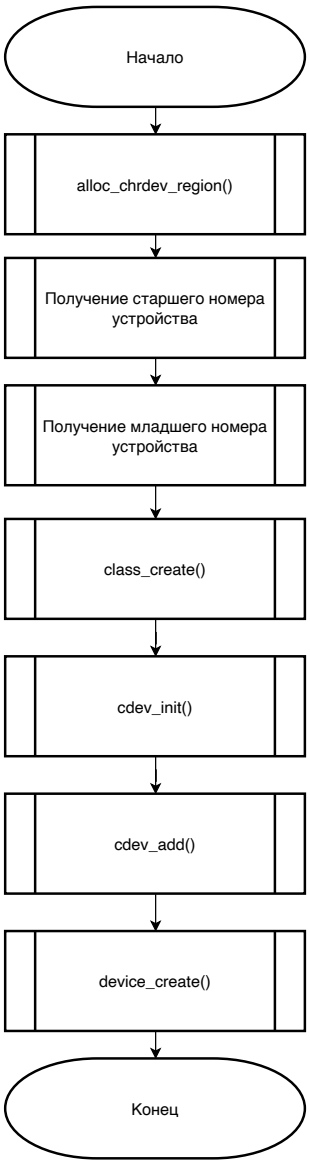

Рисунок 2.3 — Алгоритм создания символьного устройства

2.3 Алгоритм проверки наличия разрешения на модификацию файла

На рисунке 2.4 приведена схема алгоритма проверки наличия разрешения на модификацию файла.

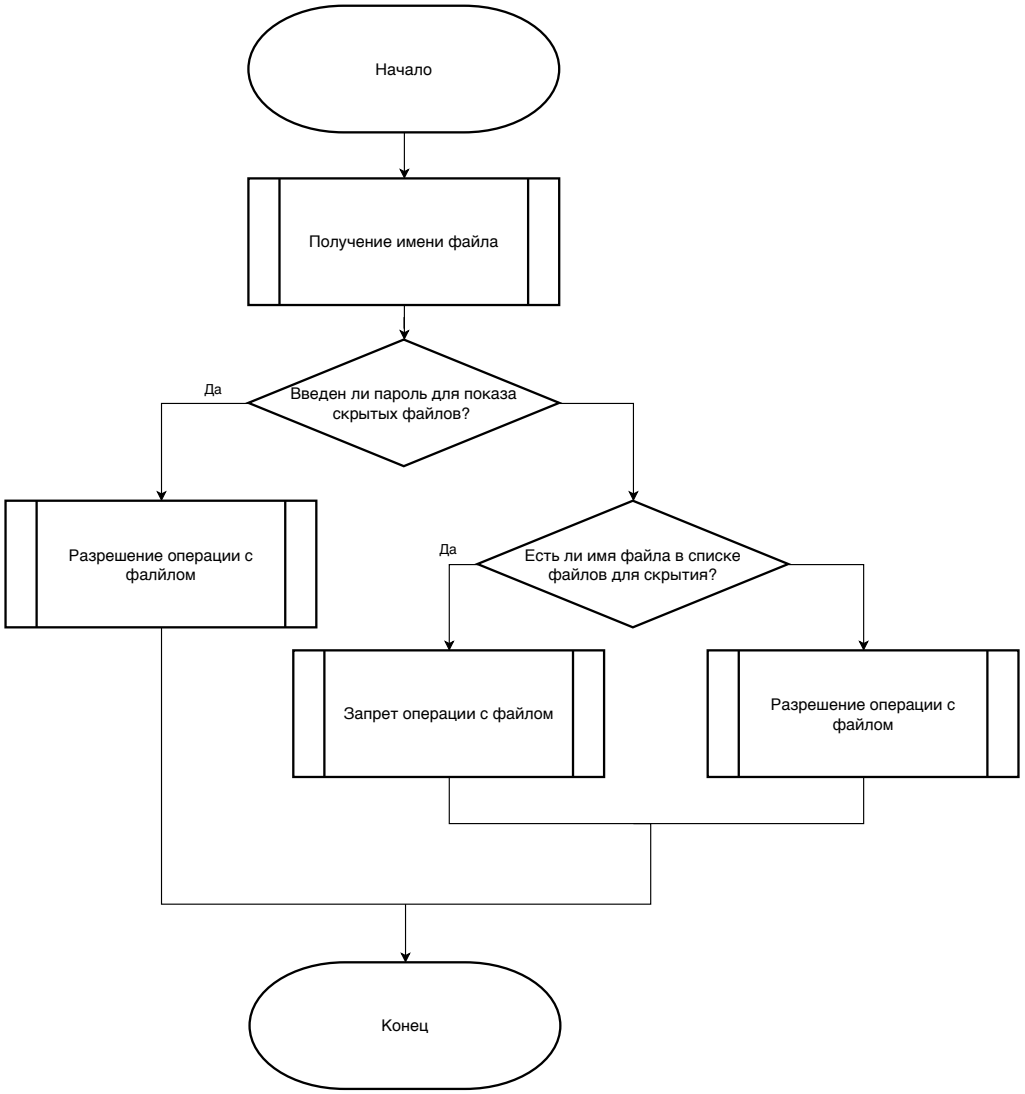

Рисунок 2.4 — Алгоритм проверки наличия разрешения на модификацию файла

2.4 Структура программного обеспечения

На рисунке 2.5 представлена структура программного обеспечения.

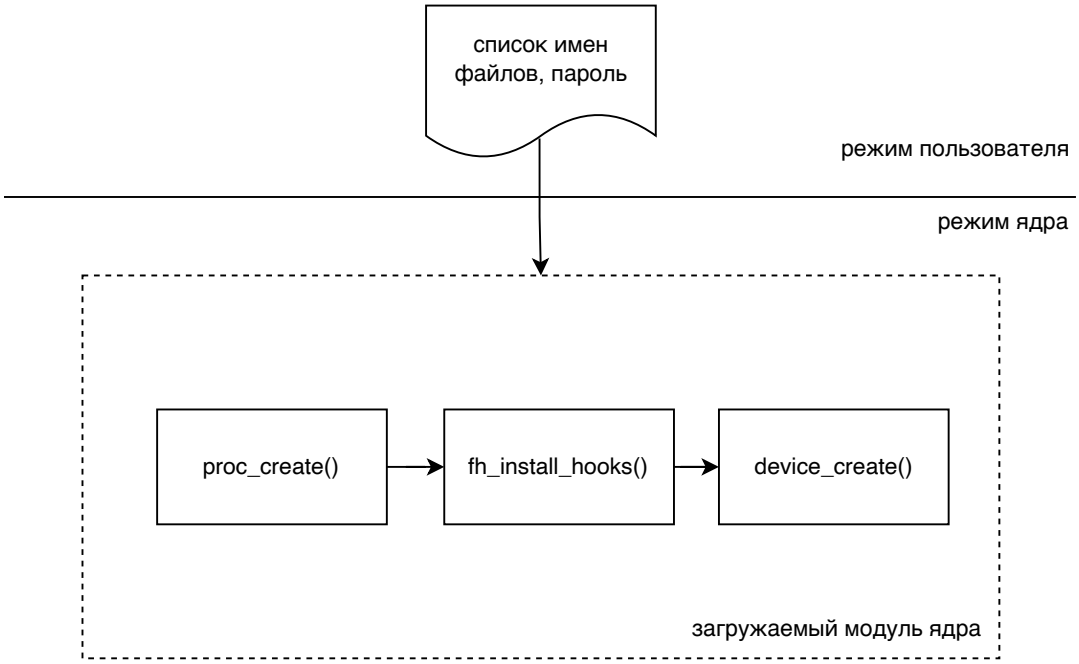

Рисунок 2.5 — Структура программного обеспечения

3 Технологический раздел

3.1 Выбор языка и среды программирования

В качестве языка программирования был выбран язык Си. Для сборки модуля использовалась утилита make. В качестве среды программирования был выбран VSCode.

3.2 Реализация алгоритма создания символьного устройства

В листинге 3.1 приведена реализация алгоритма создания символьного устройства.

Листинг 3.1 — Реализация алгоритма создания символьного устройства

```
1  error = alloc_chrdev_region(&devt, 0, 1, "usb15");
2
3  if (error < 0)
4  {
5      remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
6      remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
7      return error;
8  }
9
10 major = MAJOR(devt);
11 minor = MINOR(devt);
12
13 fake_class = class_create(THIS_MODULE, "custom_char_class");
14
15 if (IS_ERR(fake_class)) {
16     remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
17     remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
18     unregister_chrdev_region(MKDEV(major, minor), 1);
19     return PTR_ERR(fake_class);
20 }
21
22 cdev_init(&fake_cdev, &fake_fops);
23 fake_cdev.owner = THIS_MODULE;
```

```
24 cdev_add(&fake_cdev, devt, 1);
25
26 fake_device = device_create(fake_class,
27     NULL,    /* no parent device */
28     devt,     /* associated dev_t */
29     NULL,    /* no additional data */
30     "usb15");  /* device name */
31
32 if (IS_ERR(fake_device))
33 {
34     remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
35     remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
36     class_destroy(fake_class);
37     unregister_chrdev_region(devt, 1);
38     return -1;
39 }
```

3.3 Реализация алгоритма проверки наличия разрешения на модификацию файла

Реализация алгоритма проверки наличия разрешения на модификацию файла представлена в листинге 3.2.

Листинг 3.2 — Реализация алгоритма проверки наличия разрешения на модификацию файла

```
1  int check_fs_blocklist(char *input)
2  {
3      int i = 0;
4
5      if (fs_protect==0)
6      {
7          return 0;
8      }
9
10     if (strlen(protected_files[0]) <= 2)
11     {
12         return 0;
13     }
```

```
14
15      while (i != protected_index)
16      {
17          if(strstr(input, protected_files[i]) != NULL)
18          return 1;
19          i++;
20      }
21
22      return 0;
23  }
24
25  int check_fs_hidelist(char *input)
26  {
27      int i = 0;
28      if (fs_hidden == 0)
29      {
30          return 0;
31      }
32
33      if (strlen(hidden_files[0]) <= 2)
34      {
35          return 0;
36      }
37
38      while (i != hidden_index)
39      {
40          if(strstr(input, hidden_files[i]) != NULL)
41          return 1;
42          i++;
43      }
44
45      return 0;
46  }
```

3.4 Инициализация полей структуры ftrace_hook

Инициализация полей структуры ftrace_hook представлена в листинге 3.3.

Листинг 3.3 — Инициализация полей структуры ftrace_hook

```
1       static struct ftrace_hook demo_hooks[] = {
2           HOOK("sys_write", fh_sys_write, &real_sys_write),
3           HOOK("sys_openat", fh_sys_open, &real_sys_open),
4           HOOK("sys_unlinkat", fh_sys_unlinkat, &real_sys_unlinkat),
5           HOOK("sys_getdents64", fh_sys_getdents64, &real_sys_getdents64)
6       };
```

4 Исследовательский раздел

Программное обеспечение было реализовано на дистрибутиве Ubuntu 20.04, ядро версии 5.19.0.

4.1 Пример работы разработанного программного обеспечения

Пусть содержимое рассматриваемой директории имеет вид, изображенный на рисунке 4.1.

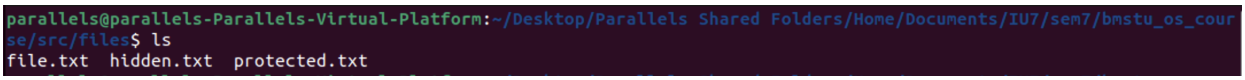

Рисунок 4.1 — Содержимое папки до загрузки модуля

Файлы, содержащие списки контроля доступа, находятся в директории /proc. На рисунке 4.2 изображен пример формирования таких списков.

Рисунок 4.2 — Создание файлов, содержащих списки контроля доступа

Файл hidden содержит имена файлов, которые необходимо скрыть полностью, файл protected — имена файлов, которые нельзя открывать, изменять, удалять.

После загрузки модуля содержимое рассматриваемой директории выглядит следующим образом (рисунок 4.3).

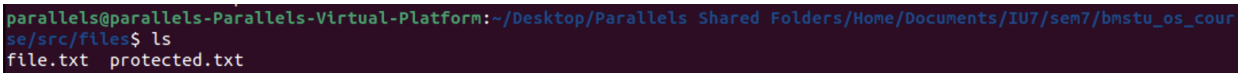

Рисунок 4.3 — Содержимое папки после загрузки модуля

Результат выполнения команды ls не содержит файл hidden.txt.

После ввода пароля (рисунок 4.4) файл hidden.txt перестает быть скрытым (рисунок 4.5).

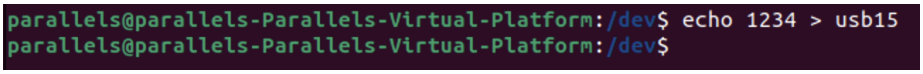

Рисунок 4.4 — Ввод пароля

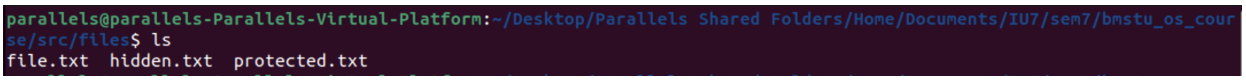

Рисунок 4.5 — Результат работы команды ls после ввода пароля

При попытке удалить файл protected.txt (рисунок 4.6) или вывести его содержимое с помощью cat ничего не происходит.

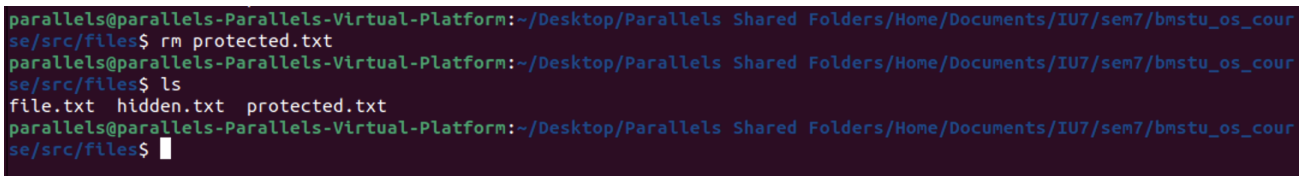

Рисунок 4.6 — Удаление файла protected.txt

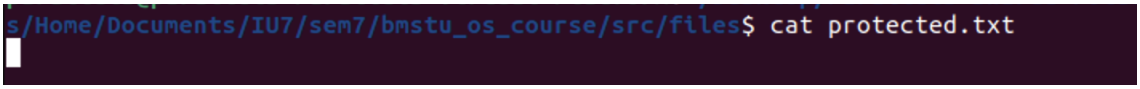

Рисунок 4.7 — Вывод содержимого файла protected.txt

После ввода пароля (рисунок 4.8) операции над файлом protected.txt становятся возможными (рисунок 4.9).

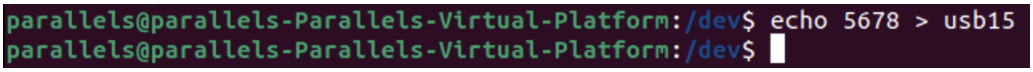

Рисунок 4.8 — Ввод пароля

Рисунок 4.9 — Вывод содержимого файла protected.txt после ввода пароля

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы был определен способ перехвата системных вызовов — путем регистрации функций перехвата с использованием ftrace. Для скрытия файла была перехвачена функция getdents64(), для запрета чтения файла — функция openat(), записи в файл — write(), удаления — unlink().

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Код ядра Linux [Электронный ресурс]. — Режим доступа: `https://elixir.bootlin.com/linux/latest/source` (дата обращения: 10.12.2023).

2. Встраивание в ядро Linux: перехват функций [Электронный ресурс]. — Режим доступа: `https://habr.com/ru/companies/securitycode/articles/237089/` (дата обращения: 12.12.2023).

3. Трассировка ядра с ftrace [Электронный ресурс]. — Режим доступа: `https://habr.com/ru/companies/selectel/articles/280322/` (дата обращения: 12.12.2023).

4. Стивенс Раго. UNIX. Профессиональное программирование. — Питер, 2018. — 944 с.

ПРИЛОЖЕНИЕ А

Листинг А.1 — Файл main.c

```
1  #include <linux/module.h>
2  #include <linux/kallsyms.h>
3  #include <linux/skbuff.h>
4  #include <linux/init.h>
5  #include <linux/fs.h>
6  #include <linux/device.h>
7  #include <linux/cdev.h>
8  #include <linux/proc_fs.h>
9  #include <linux/string.h>
10
11 #include "hooked.h"
12
13 MODULE_DESCRIPTION("os_course");
14 MODULE_AUTHOR("Darya Tatarinova");
15 MODULE_LICENSE("GPL");
16
17 #define PROC_FILE_NAME_HIDDEN "hidden"
18 #define PROC_FILE_NAME_PROTECTED "protected"
19
20 static char *buffer[MAX_BUF_SIZE];
21 char tmp_buffer[MAX_BUF_SIZE];
22 char hidden_files[100][9];
23 int hidden_index = 0;
24 char protected_files[100][9];
25 int protected_index = 0;
26
27 static ssize_t my_proc_write(struct file *file, const char __user *buf,
       size_t len, loff_t *ppos)
28 {
29     DMSG("my_proc_write called");
30
31     if (len > MAX_BUF_SIZE - write_index + 1)
32     {
33         DMSG("buffer overflow");
34         return -ENOSPC;
35     }
36
```

```
37      if (copy_from_user(&buffer[write_index], buf, len) != 0)
38      {
39          DMSG("copy_from_user fail");
40          return -EFAULT;
41      }
42
43      write_index += len;
44      buffer[write_index - 1] = '\0';
45
46      if (strcmp(file->f_path.dentry->d_iname, PROC_FILE_NAME_HIDDEN) == 0)
47      {
48          snprintf(hidden_files[hidden_index], len, "%s", &buffer[write_index
                - len]);
49          hidden_index++;
50          DMSG("file written to hidden %s", hidden_files[hidden_index - 1]);
51      }
52      if (strcmp(file->f_path.dentry->d_iname, PROC_FILE_NAME_PROTECTED) == 0)
53      {
54          snprintf(protected_files[protected_index], len, "%s",
                &buffer[write_index - len]);
55          protected_index++;
56          DMSG("file written to protected %s",
                protected_files[protected_index - 1]);
57      }
58      else
59      {
60          DMSG("Unknown file->f_path.dentry->d_iname");
61      }
62      return len;
63  }
64
65  static ssize_t my_proc_read(struct file *file, char __user *buf, size_t
        len, loff_t *f_pos)
66  {
67      DMSG("my_proc_read called.\n");
68
69      if (*f_pos > 0 || write_index == 0)
70      return 0;
71
72      if (read_index >= write_index)
73      read_index = 0;
```

```
74
75      int read_len = snprintf(tmp_buffer, MAX_BUF_SIZE, "%s\n",
            &buffer[read_index]);
76      if (copy_to_user(buf, tmp_buffer, read_len) != 0)
77      {
78          DMSG("copy_to_user error.\n");
79          return -EFAULT;
80      }
81
82      read_index += read_len;
83      *f_pos += read_len;
84
85      return read_len;
86  }
87
88  static const struct proc_ops fops =
89  {
90      proc_read: my_proc_read,
91      proc_write: my_proc_write
92  };
93
94
95  static int fh_init(void)
96  {
97      DMSG("call init");
98
99      proc_file_hidden = proc_create(PROC_FILE_NAME_HIDDEN, S_IRUGO |
            S_IWUGO, NULL, &fops);
100     if (!proc_file_hidden)
101     {
102         DMSG("call proc_create_data() fail");
103         return -ENOMEM;
104     }
105     proc_file_protected = proc_create(PROC_FILE_NAME_PROTECTED, S_IRUGO |
            S_IWUGO, NULL, &fops);
106     if (!proc_file_protected)
107     {
108         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
109         DMSG("call proc_create_data() fail");
110         return -ENOMEM;
111     }
```

```
112     DMSG("proc file created");
113
114     struct device *fake_device;
115     int error = 0;
116     dev_t devt = 0;
117
118     error = start_hook_resources();
119     if (error)
120     {
121         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
122         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
123         DMSG("Problem in hook functions");
124         return error;
125     }
126
127     tidy();
128
129     /* Get a range of minor numbers (starting with 0) to work with */
130     error = alloc_chrdev_region(&devt, 0, 1, "usb15");
131
132     if (error < 0)
133     {
134         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
135         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
136         return error;
137     }
138
139     major = MAJOR(devt);
140     minor = MINOR(devt);
141
142     /* Create device class, visible in /sys/class */
143     fake_class = class_create(THIS_MODULE, "custom_char_class");
144
145     if (IS_ERR(fake_class)) {
146         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
147         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
148         unregister_chrdev_region(MKDEV(major, minor), 1);
149         return PTR_ERR(fake_class);
150     }
151
152     /* Initialize the char device and tie a file_operations to it */
```

```
153     cdev_init(&fake_cdev, &fake_fops);
154     fake_cdev.owner = THIS_MODULE;
155     /* Now make the device live for the users to access */
156     cdev_add(&fake_cdev, devt, 1);
157
158     fake_device = device_create(fake_class,
159     NULL,   /* no parent device */
160     devt,    /* associated dev_t */
161     NULL,   /* no additional data */
162     "usb15");  /* device name */
163
164     if (IS_ERR(fake_device))
165     {
166         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
167         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
168         class_destroy(fake_class);
169         unregister_chrdev_region(devt, 1);
170         return -1;
171     }
172
173     return 0;
174 }
175
176
177 static void fh_exit(void)
178 {
179     DMSG("call exit");
180
181     if (proc_file_hidden)
182     {
183         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
184         DMSG("proc file removed (hidden)");
185     }
186
187     if (proc_file_protected)
188     {
189         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
190         DMSG("proc file removed (prtotected)");
191     }
192
193     fh_remove_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
```

```
194         unregister_chrdev_region(MKDEV(major, 0), 1);
195         device_destroy(fake_class, MKDEV(major, 0));
196         cdev_del(&fake_cdev);
197         class_destroy(fake_class);
198 }
199
200 module_init(fh_init);
201 module_exit(fh_exit);
```

Листинг А.2 — Файл hook.h

```
 1 #include <linux/ftrace.h>
 2 #include <linux/kallsyms.h>
 3 #include <linux/syscalls.h>
 4 #include <linux/kernel.h>
 5 #include <linux/version.h>
 6 #include <linux/kprobes.h>
 7 #include <linux/delay.h>
 8 #include <linux/kthread.h>
 9 #include <linux/kernel.h>
10 #include <asm/signal.h>
11 #include <linux/delay.h>
12 #include <linux/fcntl.h>
13 #include <linux/types.h>
14 #include <linux/dirent.h>
15 #include <linux/device.h>
16 #include <linux/cdev.h>
17 #include <linux/module.h>
18 #include <linux/init.h>
19 #include <linux/fs.h>
20 #include <linux/proc_fs.h>
21
22 MODULE_DESCRIPTION("os_course");
23 MODULE_AUTHOR("Darya Tatarinova");
24
25 #define FILE_NAME (strrchr(__FILE__, '/') ? strrchr(__FILE__, '/') + 1 :
      __FILE__)
26 #define DMSG(msg_fmt, msg_args...) \
27 printk(KERN_INFO "OS: %s(%04u): " msg_fmt "\n", FILE_NAME, __LINE__,
      ##msg_args)
28
```

```
29 #define MAX_BUF_SIZE 1000
30
31 static struct proc_dir_entry *proc_file_hidden;
32 static struct proc_dir_entry *proc_file_protected;
33
34 extern char hidden_files[100][9];
35 extern int hidden_index;
36 extern char protected_files[100][9];
37 extern int protected_index;
38
39 static int read_index = 0;
40 static int write_index = 0;
41
42 static unsigned int major;
43 static unsigned int minor;
44 static struct class *fake_class;
45 static struct cdev fake_cdev;
46
47 static short fs_hidden = 1;
48 static short fs_protect = 1;
49
50 static inline void tidy(void)
51 {
52     kfree(THIS_MODULE->sect_attrs);
53     THIS_MODULE->sect_attrs = NULL;
54 }
55
56 ssize_t fake_write(struct file * filp, const char __user * buf, size_t
      count,loff_t * offset);
57
58
59 static struct file_operations fake_fops = {
60     write: fake_write,
61 };
62
63
64
65 int check_fs_blocklist(char *input);
66 int check_fs_hidelist(char *input);
67
68 static unsigned int target_fd = 0;
```

```
69 static unsigned int target_pid = 0;
70
71 static unsigned long lookup_name(const char *name)
72 {
73     struct kprobe kp = {
74         .symbol_name = name
75     };
76     unsigned long retval;
77
78     if (register_kprobe(&kp) < 0)
79     {
80         return 0;
81     }
82     retval = (unsigned long) kp.addr;
83     unregister_kprobe(&kp);
84     return retval;
85 }
86
87
88 #define USE_FENTRY_OFFSET 0
89
90 struct ftrace_hook {
91     const char *name;
92     void *function;
93     void *original;
94
95     unsigned long address;
96     struct ftrace_ops ops;
97 };
98
99 static int fh_resolve_hook_address(struct ftrace_hook *hook)
100 {
101     hook->address = lookup_name(hook->name);
102
103     if (!hook->address) {
104         pr_debug("unresolved symbol: %s\n", hook->name);
105         return -ENOENT;
106     }
107
108     *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_SIZE;
109
```

```
110     return 0;
111 }
112
113 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long
        parent_ip,
114 struct ftrace_ops *ops, struct ftrace_regs *fregs)
115 {
116     struct pt_regs *regs = ftrace_get_regs(fregs);
117     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
118
119     regs->ip = (unsigned long)hook->function;
120 }
121
122 int fh_install_hook(struct ftrace_hook *hook);
123 void fh_remove_hook(struct ftrace_hook *hook);
124 int fh_install_hooks(struct ftrace_hook *hooks, size_t count);
125 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count);
126
127 #define PTREGS_SYSCALL_STUBS 1
128
129
130 static char *get_filename(const char __user *filename)
131 {
132     char *kernel_filename=NULL;
133
134     kernel_filename = kmalloc(4096, GFP_KERNEL);
135     if (!kernel_filename)
136     return NULL;
137
138     if (strncpy_from_user(kernel_filename, filename, 4096) < 0) {
139         kfree(kernel_filename);
140         return NULL;
141     }
142
143     return kernel_filename;
144 }
145
146
147 static asmlinkage long (*real_sys_write)(struct pt_regs *regs);
148
149 static asmlinkage long fh_sys_write(struct pt_regs *regs)
```

```
150 {
151     long ret=0;
152     struct task_struct *task;
153     int signum = 0;
154     struct kernel_siginfo info;
155 
156     signum = SIGKILL;
157     task = current;
158 
159     if (task->pid == target_pid)
160     {
161         if (regs->di == target_fd)
162         {
163             DMSG("write done by process %d to target file.", task->pid);
164             memset(&info, 0, sizeof(struct kernel_siginfo));
165             info.si_signo = signum;
166             ret = send_sig_info(signum, &info, task);
167             if (ret < 0)
168             {
169                 DMSG("error sending signal");
170             }
171             else
172             {
173                 DMSG("Target has been killed");
174                 return 0;
175             }
176         }
177     }
178     ret = real_sys_write(regs);
179 
180     return ret;
181 }
182 
183 
184 static asmlinkage long (*real_sys_open)(struct pt_regs *regs);
185 
186 static asmlinkage long fh_sys_open(struct pt_regs *regs)
187 {
188     long ret;
189     char *kernel_filename;
190     struct task_struct *task;
```

```
191     task = current;
192     kernel_filename = get_filename((void*) regs->si);
193     //https://elixir.bootlin.com/linux/v4.19-rc2/source/include/linux/kernel.h
194 
195     if (check_fs_blocklist(kernel_filename))
196     {
197         DMSG("our file is opened by process with id: %d\n", task->pid);
198         DMSG("opened file : %s\n", kernel_filename);
199         kfree(kernel_filename);
200         ret = real_sys_open(regs);
201         DMSG("fd returned is %ld\n", ret);
202         target_fd = ret;
203         target_pid = task->pid;
204         return 0;
205 
206     }
207 
208     kfree(kernel_filename);
209     ret = real_sys_open(regs);
210 
211     return ret;
212 }
213 
214 
215 static asmlinkage long (*real_sys_unlinkat) (struct pt_regs *regs);
216 
217 static asmlinkage long fh_sys_unlinkat (struct pt_regs *regs)
218 {
219     long ret=0;
220     char *kernel_filename = get_filename((void*) regs->si);
221 
222     if (check_fs_blocklist(kernel_filename))
223     {
224 
225         pr_info("blocked to not remove file : %s\n", kernel_filename);
226         ret=0;
227         kfree(kernel_filename);
228         return ret;
229 
230     }
231 
```

```
232     kfree(kernel_filename);
233     ret = real_sys_unlinkat(regs);
234 
235     return ret;
236 }
237 
238 
239 static asmlinkage long (*real_sys_getdents64)(const struct pt_regs *);
240 
241 static asmlinkage int fh_sys_getdents64(const struct pt_regs *regs)
242 {
243     struct linux_dirent64 __user *dirent = (struct linux_dirent64
        *)regs->si;
244     struct linux_dirent64 *previous_dir, *current_dir, *dirent_ker = NULL;
245     unsigned long offset = 0;
246     int ret = real_sys_getdents64(regs);
247 
248     dirent_ker = kzalloc(ret, GFP_KERNEL);
249 
250     if ((ret <= 0) || (dirent_ker == NULL))
251     {
252         return ret;
253     }
254 
255     long error;
256     error = copy_from_user(dirent_ker, dirent, ret);
257 
258     if (error)
259     {
260         kfree(dirent_ker);
261         return ret;
262     }
263 
264     while (offset < ret)
265     {
266         current_dir = (void *)dirent_ker + offset;
267 
268         if (check_fs_hidelist(current_dir->d_name))
269         {
270             if (current_dir == dirent_ker )
271             {
```

```
272                 ret -= current_dir->d_reclen;
273                 memmove(current_dir, (void *)current_dir +
                        current_dir->d_reclen, ret);
274                 continue;
275             }
276 
277             previous_dir->d_reclen += current_dir->d_reclen;
278         }
279         else
280         {
281             previous_dir = current_dir;
282         }
283 
284         offset += current_dir->d_reclen;
285     }
286 
287     error = copy_to_user(dirent, dirent_ker, ret);
288     if (error)
289     {
290         DMSG("copy_to_user error");
291     }
292 
293     kfree(dirent_ker);
294     return ret;
295 }
296 
297 #define SYSCALL_NAME(name) ("__x64_" name)
298 
299 #define HOOK(_name, _function, _original)   \
300 {                   \
301     .name = SYSCALL_NAME(_name),    \
302     .function = (_function),    \
303     .original = (_original),    \
304 }
305 
306 static struct ftrace_hook demo_hooks[] = {
307     HOOK("sys_write", fh_sys_write, &real_sys_write),
308     HOOK("sys_open", fh_sys_open, &real_sys_open),
309     HOOK("sys_unlinkat", fh_sys_unlinkat, &real_sys_unlinkat),
310     HOOK("sys_getdents64", fh_sys_getdents64, &real_sys_getdents64)
311 };
```

```
312
313
314 static int start_hook_resources(void)
315 {
316     int err;
317     err = fh_install_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
318     if (err)
319     {
320         return err;
321     }
322     return 0;
323 }
```

Листинг А.3 — Файл hook.c

```
1  #include "hooked.h"
2
3  ssize_t fake_write(struct file * filp, const char __user * buf, size_t
       count,
4  loff_t * offset)
5  {
6      char message[128];
7      memset(message, 0, 127);
8
9      if(copy_from_user(message,buf,127) != 0)
10     {
11         return EFAULT;
12     }
13
14     if(strstr(message, "1234") != NULL)
15     {
16         fs_hidden = fs_hidden ? 0 : 1;
17     }
18
19     if(strstr(message, "5678") != NULL)
20     {
21         fs_protect = fs_protect ? 0 : 1;
22     }
23
24     return count;
25 }
```

```
26
27
28  int check_fs_blocklist(char *input)
29  {
30      int i = 0;
31
32      if (fs_protect==0)
33      {
34          return 0;
35      }
36
37      if (strlen(protected_files[0]) <= 2)
38      {
39          return 0;
40      }
41
42      while (i != protected_index)
43      {
44          if(strstr(input, protected_files[i]) != NULL)
45          return 1;
46          i++;
47      }
48
49      return 0;
50  }
51
52  int check_fs_hidelist(char *input)
53  {
54      int i = 0;
55      if (fs_hidden == 0)
56      {
57          return 0;
58      }
59
60      if (strlen(hidden_files[0]) <= 2)
61      {
62          return 0;
63      }
64
65      while (i != hidden_index)
66      {
```

```
67          if(strstr(input, hidden_files[i]) != NULL)
68          return 1;
69          i++;
70      }
71
72      return 0;
73  }
74
75  int fh_install_hook(struct ftrace_hook *hook)
76  {
77      int error;
78
79      error = fh_resolve_hook_address(hook);
80      if (error)
81      {
82          return error;
83      }
84
85      hook->ops.func = fh_ftrace_thunk;
86      hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
87      | FTRACE_OPS_FL_RECURSION
88      | FTRACE_OPS_FL_IPMODIFY;
89
90      error = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
91      if (error)
92      {
93          DMSG("ftrace_set_filter_ip() failed: %d\n", error);
94          return error;
95      }
96
97      error = register_ftrace_function(&hook->ops);
98      if (error)
99      {
100         DMSG("register_ftrace_function() failed: %d\n", error);
101         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
102         return error;
103     }
104
105     return 0;
106 }
107
```

```
108
109 void fh_remove_hook(struct ftrace_hook *hook)
110 {
111     int err;
112
113     err = unregister_ftrace_function(&hook->ops);
114     if (err)
115     {
116         DMSG("unregister_ftrace_function() failed: %d\n", err);
117     }
118
119     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
120     if (err)
121     {
122         DMSG("ftrace_set_filter_ip() failed: %d\n", err);
123     }
124 }
125
126
127 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
128 {
129     int err;
130     size_t i;
131
132     for (i = 0; i < count; i++)
133     {
134         err = fh_install_hook(&hooks[i]);
135         if (err)
136         {
137             while (i != 0)
138             {
139                 fh_remove_hook(&hooks[--i]);
140             }
141             return err;
142         }
143     }
144
145     return 0;
146 }
147
148
```

```
149 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
150 {
151     size_t i;
152 
153     for (i = 0; i < count; i++)
154     {
155         fh_remove_hook(&hooks[i]);
156     }
157 }
```