

СОДЕРЖАНИЕ

Введение	4
1 Аналитический раздел	5
1.1 Постановка задачи	5
1.2 Способы перехвата функций ядра	5
1.2.1 ftrace	5
1.2.2 kprobes	7
1.2.3 Linux Security API	8
1.2.4 Модификация таблицы системных вызовов	8
1.2.5 Сплайсинг	9
1.2.6 Сравнительный анализ способов перехвата	9
1.3 Перехватываемые функции ядра	10
1.3.1 Функция getdents64	10
1.3.2 Функция unlink	11
1.3.3 Функции open	12
1.3.4 Функция rename	12
2 Конструкторский раздел	14
2.1 IDEF0	14
2.2 Алгоритм проверки необходимости сокрытия файла	15
2.3 Алгоритм проверки разрешения на удаление файла	16
2.4 Алгоритм проверки разрешения на переименование файла и каталога	17

2.5	Алгоритм проверки разрешения на чтение и запись.....	17
2.6	Структура программного обеспечения.....	19
3	Технологический раздел	20
3.1	Выбор языка и среды программирования	20
3.2	Реализация алгоритма проверки необходимости сокрытия файла..	20
3.3	Реализация алгоритма проверки разрешения на удаление файла ..	21
3.4	Реализация алгоритма проверки разрешения на чтение и запись ..	22
3.5	Реализация алгоритма проверки разрешения на переименование ..	23
3.6	Инициализация полей структуры ftrace_hook.....	23
3.7	Makefile	24
4	Исследовательский раздел	25
4.1	Пример работы разработанного программного обеспечения.....	25
	Заключение	27
	Список использованных источников.....	28
	Приложение	29

ВВЕДЕНИЕ

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

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

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. Также обеспечить запрещать удаление, переименование каталогов. Предусмотреть возможность ввода пароля для 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 [1].

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 [1].

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 ХОК(_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 [2]. 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` [3]. Подмена значений в этой таблице приводит к смене поведения всей системы. Таким образом, сохранив старые 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 [4].

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 сокрытия файла 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. Перехват вызова осуществляется при помощи создания указателя на системный вызов `getdents64`.

Листинг 1.5 — Указатель на getdents64

```
1 static asmlinkage long (*real_sys_getdents64)(const struct pt_regs *);
```

1.3.2 Функция unlink

Удаление записей из каталога производится с помощью функции `unlink`.

Листинг 1.6 — 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`. Перехват вызова осуществляется при помощи создания указателя на системный вызов.

Листинг 1.7 — Указатель на `unlink`

```
1 static asmlinkage long (*real_sys_unlink) (struct pt_regs *regs);
```

1.3.3 Функции `open`

Чтение из файла и запись в него производится с помощью функции `open`.

Листинг 1.8 — Функция `open`

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

Для того, чтобы запретить чтение из файла и запись в него, необходимо перехватить системные вызовы `open`. Перехват осуществляется при помощи создания указателей на системные вызовы.

Листинг 1.9 — Указатели на `open`, `write` и `read`.

```
1 static asmlinkage long (*real_sys_open)(struct pt_regs *regs);
```

1.3.4 Функция `rename`

Переименование файла и каталога производится с помощью функции `rename`.

Листинг 1.10 — Функция `rename`

```
1 #include <fcntl.h>
2
3 int rename(const char *oldpath, const char *newpath);
```

Для того, чтобы запретить переименовать файла и каталога, необходимо перехватить системные вызовы `rename`. Перехват осуществляется при помощи создания указателей на системные вызовы.

Листинг 1.11 — Указатели на `rename`

```
1 static asmlinkage long (*real_sys_rename)(struct pt_regs *regs);
```

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 переименования — функция `rename`, для запрета чтения из файла и записи в файл — функция `open`, 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.

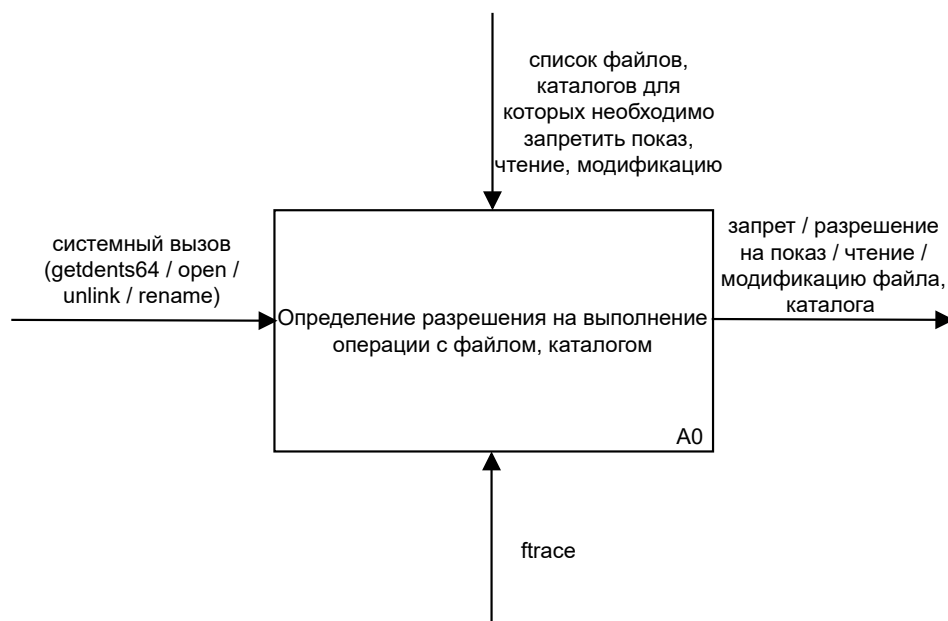


Рисунок 2.1 — Диаграмма состояний IDEF0 нулевого уровня

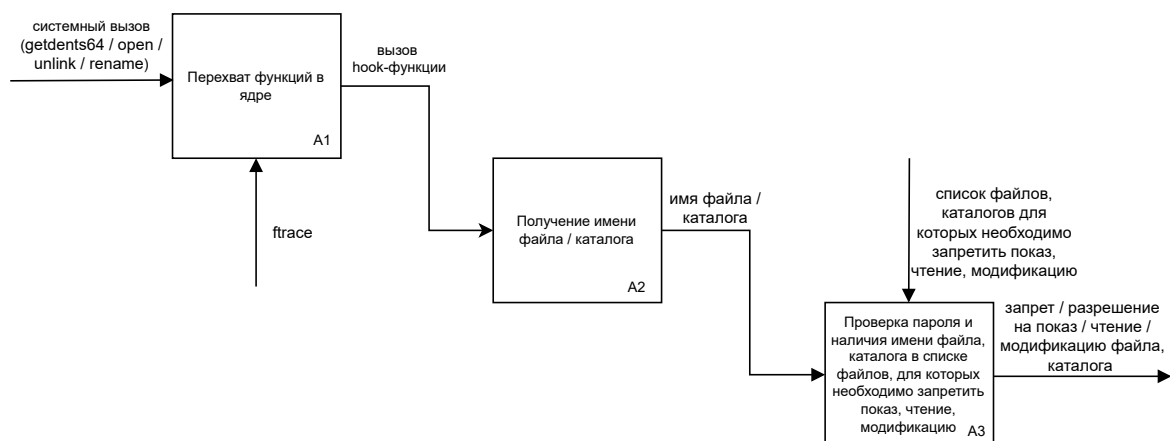


Рисунок 2.2 — Диаграмма состояний IDEF0 первого уровня

2.2 Алгоритм проверки необходимости сокрытия файла

На рисунке 2.3 приведена схема алгоритма проверки необходимости сокрытия файла.

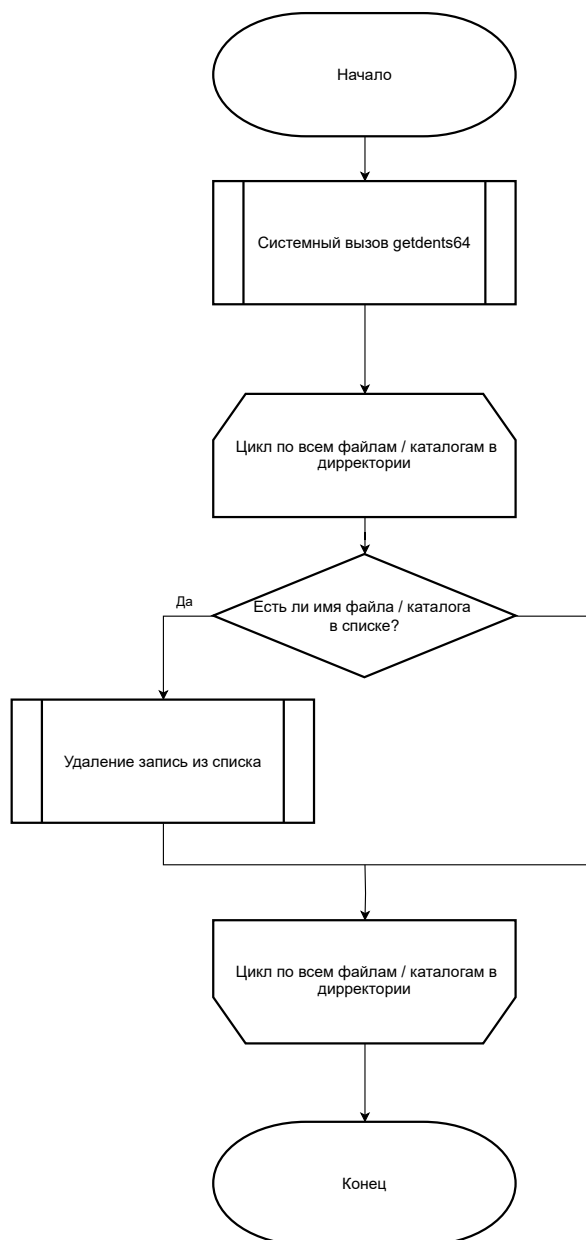


Рисунок 2.3 — Алгоритм проверки необходимости сокрытия файла

2.3 Алгоритм проверки разрешения на удаление файла

На рисунке 2.4 приведена схема алгоритма проверки разрешения на удаление файла.

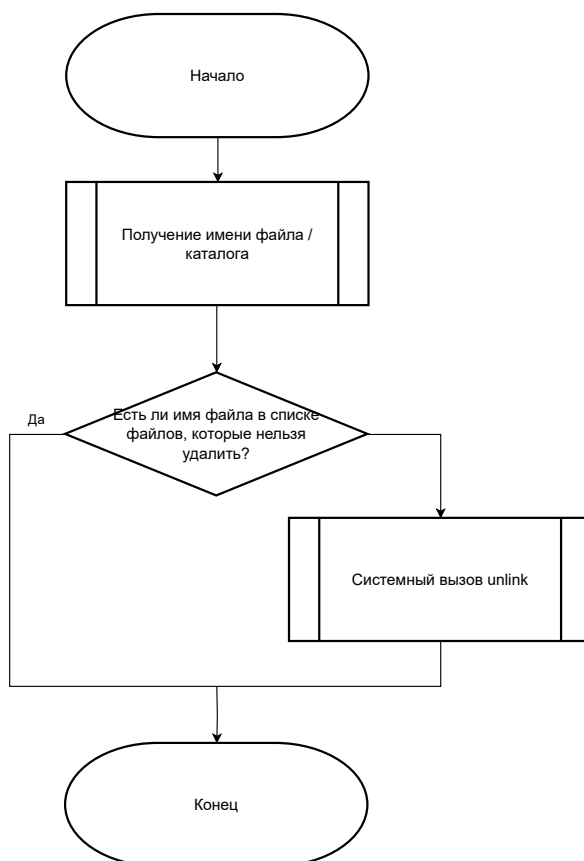


Рисунок 2.4 — Алгоритм проверки разрешения на удаления файла

2.4 Алгоритм проверки разрешения на переименование файла и каталога

На рисунке 2.5 приведена схема алгоритма проверки разрешения на чтение из файла.

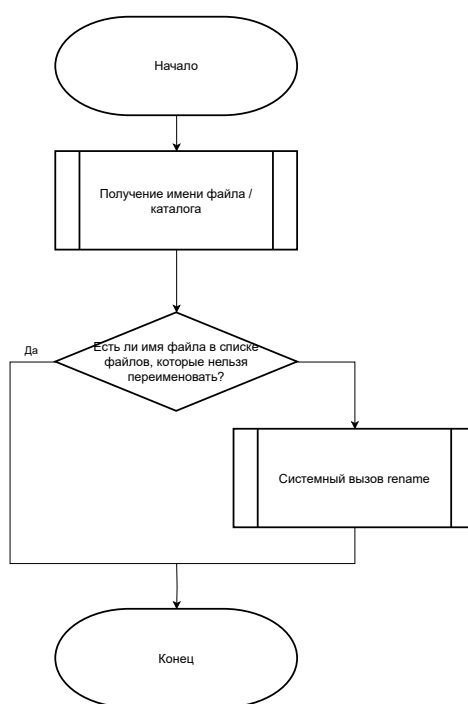


Рисунок 2.5 — Алгоритм проверки разрешения на чтение из файла

2.5 Алгоритм проверки разрешения на чтение и запись

На рисунке 2.6 приведена схема алгоритма проверки разрешения на чтение из файла и запись в него.

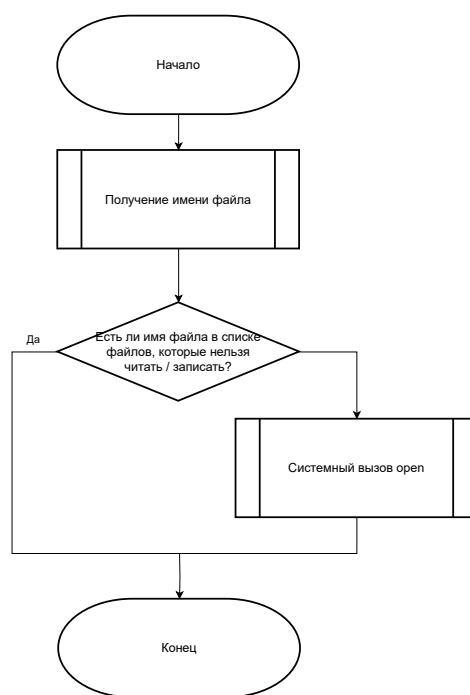


Рисунок 2.6 — Алгоритм проверки разрешения на чтение из файла и запись в него

2.6 Структура программного обеспечения

На рисунке 2.7 представлена структура программного обеспечения.

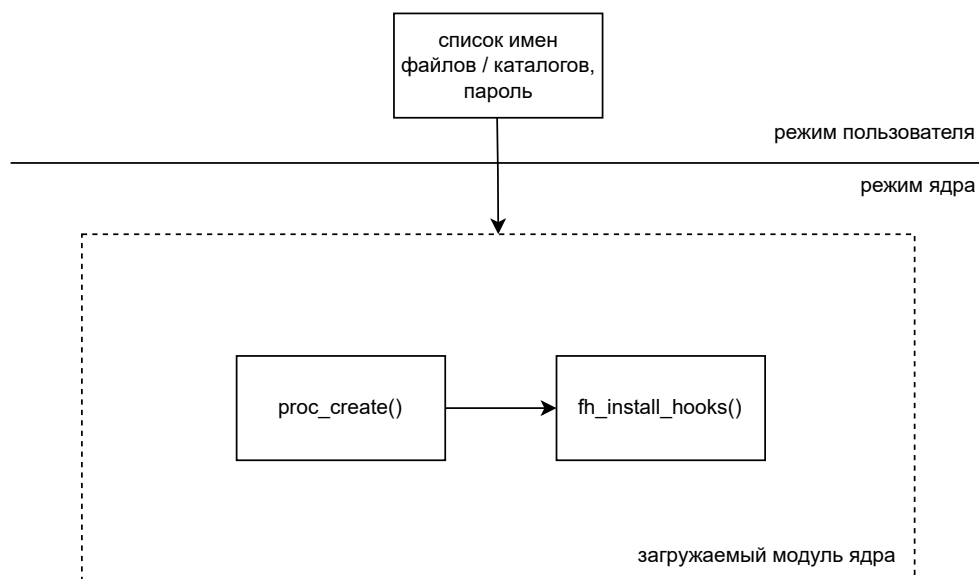


Рисунок 2.7 — Структура программного обеспечения

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 проверки необходимости сокрытия файла

В листинге 3.1 приведена реализация алгоритма проверки необходимости сокрытия файл.

Листинг 3.1 — Реализация алгоритма проверки необходимости сокрытия файл

```
1 static asmlinkage int fh_sys_getdents64(const struct pt_regs *regs)
2 {
3     struct linux_dirent64 __user *dirent = (struct linux_dirent64
4         *)regs->si;
5     struct linux_dirent64 *previous_dir, *current_dir, *dirent_ker = NULL;
6     unsigned long offset = 0;
7     int ret = real_sys_getdents64(regs);
8     dirent_ker = kzalloc(ret, GFP_KERNEL);
9
10    if ((ret <= 0) || (dirent_ker == NULL))
11    {
12        return ret;
13    }
14
15    copy_from_user(dirent_ker, dirent, ret);
16
17    while (offset < ret)
18    {
19        current_dir = (void *)dirent_ker + offset;
20
21        if (check_fs_hidelist(current_dir->d_name))
```

```

22      {
23          if (current_dir == dirent_ker)
24          {
25              ret -= current_dir->d_reclen;
26              memmove(current_dir, (void *)current_dir +
27                      current_dir->d_reclen, ret);
28              continue;
29          }
30          previous_dir->d_reclen += current_dir->d_reclen;
31      }
32      else
33      {
34          previous_dir = current_dir;
35      }
36
37      offset += current_dir->d_reclen;
38  }
39
40  copy_to_user(dirent, dirent_ker, ret);
41
42  kfree(dirent_ker);
43  return ret;
44 }

```

3.3 Реализация алгоритма проверки разрешения на удаление файла

Реализация алгоритма проверки разрешения на удаление файла представлена в листинге 3.2.

Листинг 3.2 — Реализация алгоритма проверки разрешения на удаление файла

```

1  static asmlinkage long fh_sys_unlink(struct pt_regs *regs)
2  {
3      char *kernel_filename = get_filename((void*) regs->si);
4
5      if (check_fs_blocklist(kernel_filename) ||
          check_dir_blocklist(kernel_filename))

```

```

6      {
7
8          pr_info("blocked to not remove file : %s\n", kernel_filename);
9          kfree(kernel_filename);
10         return -EPERM;
11
12     }
13
14     kfree(kernel_filename);
15     return real_sys_unlink(regs);
16 }
17
18 }
```

3.4 Реализация алгоритма проверки разрешения на чтение и запись

Реализация алгоритма проверки разрешения на чтение и запись представлена в листинге 3.3.

Листинг 3.3 — Реализация алгоритма проверки разрешения на чтение из файла

```

1  static asmlinkage long fh_sys_open(struct pt_regs *regs)
2  {
3      char *kernel_filename;
4      kernel_filename = get_filename((void*) regs->si);
5
6      if (check_fs_blocklist(kernel_filename))
7      {
8          DMSG("block open file : %s", kernel_filename);
9          kfree(kernel_filename);
10         return -EPERM;
11     }
12
13     kfree(kernel_filename);
14
15     return real_sys_open(regs);
16 }
17
```

3.5 Реализация алгоритма проверки разрешения на переименование

Реализация алгоритма проверки разрешения на переименование представлена в листинге 3.4.

Листинг 3.4 — Реализация алгоритма проверки разрешения на запись в файл

```

1 static asmlinkage long fh_sys_rename(struct pt_regs *regs)
2 {
3     long ret=0;
4     char *kernel_filename = get_filename((void*) regs->si);
5
6     if (check_fs_blocklist(kernel_filename) ||
7         check_dir_blocklist(kernel_filename))
8     {
9         pr_info("blocked to not rename file : %s\n", kernel_filename);
10        kfree(kernel_filename);
11        return -EPERM;
12    }
13
14    kfree(kernel_filename);
15    ret = real_sys_rename(regs);
16
17    return ret;
18 }
19 }
```

3.6 Инициализация полей структуры ftrace_hook

Инициализация полей структуры ftrace_hook представлена в листинге 3.5.

Листинг 3.5 — Инициализация полей структуры ftrace_hook

```

1 static struct ftrace_hook demo_hooks[] = {
2     HOOK("sys_open", fh_sys_open, &real_sys_open),
3     HOOK("sys_unlink", fh_sys_unlink, &real_sys_unlink),
4     HOOK("sys_rename", fh_sys_rename, &real_sys_rename),
5     HOOK("sys_getdents64", fh_sys_getdents64, &real_sys_getdents64)
6 };

```

3.7 Makefile

В листинге 3.6 представлен Makefile.

Листинг 3.6 — Makefile

```

1 CONFIG_MODULE_SIG=n
2 PWD := $(shell pwd)
3 CC := gcc
4 KERNEL_PATH ?= /lib/modules/$(shell uname -r)/build
5 ccflags-y += -Wall -Wdeclaration-after-statement
6
7 obj-m += my_module.o
8 casperfs-objs := main.o hooked.o
9
10 all:
11 make -C $(KERNEL_PATH) M=$(PWD) modules
12
13 clean:
14 make -C $(KERNEL_PATH) M=$(PWD) clean

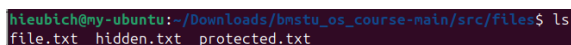
```

4 Исследовательский раздел

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

4.1 Пример работы разработанного программного обеспечения

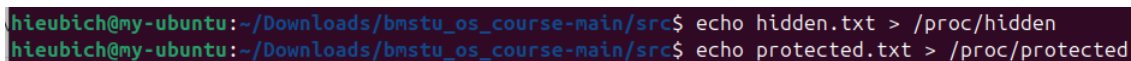
Пусть содержимое рассматриваемой директории имеет вид, изображенный на рисунке 4.1.



```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ ls
file.txt  hidden.txt  protected.txt
```

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.

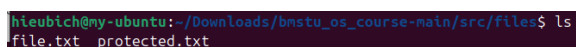


```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src$ echo hidden.txt > /proc/hidden
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src$ echo protected.txt > /proc/protected
```

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).



```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ ls
file.txt  protected.txt
```

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).


```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ sudo chmod 666 /dev/usb15
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ echo 1234 > /dev/usb15
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$
```

Рисунок 4.4 — Ввод пароля

```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ ls
file.txt hidden.txt protected.txt
```

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.

```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ rm protected.txt
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ ls
file.txt hidden.txt protected.txt
```

Рисунок 4.6 — Удаление файла protected.txt

```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ cat protected.txt
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$
```

Рисунок 4.7 — Вывод содержимого файла protected.txt

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

```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ echo 5678 > /dev/usb15
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$
```

Рисунок 4.8 — Ввод пароля

```
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$ cat protected.txt
qqq
hieubich@my-ubuntu:~/Downloads/bmstu_os_course-main/src/files$
```

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`, так как он позволяет перехватывать любые функции ядра и не требует его перекомпиляции.

Для сокрытия файла была перехвачена функция `getdents64`, для запрета чтения из файла и записи в файл — функции `open`, `read` и `write`, удаления — `unlink`.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Трассировка ядра с ftrace [Электронный ресурс]. — Режим доступа: <https://habr.com/ru/companies/selectel/articles/280322/> (дата обращения: 12.12.2024).
2. Стивенс Раго. UNIX. Профессиональное программирование. — Питер, 2018. — 944 с.
3. Код ядра Linux [Электронный ресурс]. — Режим доступа: <https://elixir.bootlin.com/linux/latest/source> (дата обращения: 10.12.2024).
4. Встраивание в ядро Linux: перехват функций [Электронный ресурс]. — Режим доступа: <https://habr.com/ru/companies/securitycode/articles/237089/> (дата обращения: 12.12.2024).

ПРИЛОЖЕНИЕ

Листинг 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 2024");
14 MODULE_AUTHOR("Pham Minh Hieu");
15 MODULE_LICENSE("GPL");
16
17 #define PROC_FILE_NAME_HIDDEN "hidden"
18 #define PROC_FILE_NAME_PROTECTED "protected"
19 #define PROC_DIR_NAME_PROTECTED "dir"
20
21 static char *buffer[MAX_BUF_SIZE];
22 char tmp_buffer[MAX_BUF_SIZE];
23 char hidden_files[100][50];
24 int hidden_index = 0;
25 char protected_files[100][50];
26 int protected_index = 0;
27 char protected_dirs[100][50];
28 int dir_index = 0;
29
30 static int read_index = 0;
31 static int write_index = 0;
32
33 static struct proc_dir_entry *proc_file_hidden;
34 static struct proc_dir_entry *proc_file_protected;
35 static struct proc_dir_entry *proc_dir_protected;
36
```

```

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

```

```

71         dir_index++;
72         DMSG("file written to dir %s", protected_dirs[dir_index - 1]);
73     }
74     else
75     {
76         DMSG("Unknown file in proc %s", file->f_path.dentry->d_iname);
77     }
78     return len;
79 }
80
81 static ssize_t my_proc_read(struct file *file, char __user *buf, size_t
    len, loff_t *f_pos)
82 {
83     DMSG("my_proc_read called.\n");
84
85     if (*f_pos > 0 || write_index == 0)
86         return 0;
87
88     if (read_index >= write_index)
89         read_index = 0;
90
91     int read_len = snprintf(tmp_buffer, MAX_BUF_SIZE, "%s\n",
        &buffer[read_index]);
92     if (copy_to_user(buf, tmp_buffer, read_len) != 0)
93     {
94         DMSG("copy_to_user error.\n");
95         return -EFAULT;
96     }
97
98     read_index += read_len;
99     *f_pos += read_len;
100
101     return read_len;
102 }
103
104 static const struct proc_ops fops =
105 {
106     proc_read: my_proc_read,
107     proc_write: my_proc_write
108 };
109

```

```

110
111 static int fh_init(void)
112 {
113     DMSG(" call init ");
114
115     proc_file_hidden = proc_create(PROC_FILE_NAME_HIDDEN, S_IRUGO |
        S_IWUGO, NULL, &fops);
116     if (!proc_file_hidden)
117         return -ENOMEM;
118
119     proc_file_protected = proc_create(PROC_FILE_NAME_PROTECTED, S_IRUGO |
        S_IWUGO, NULL, &fops);
120     if (!proc_file_protected)
121     {
122         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
123         return -ENOMEM;
124     }
125
126     proc_dir_protected = proc_create(PROC_DIR_NAME_PROTECTED, S_IRUGO |
        S_IWUGO, NULL, &fops);
127     if (!proc_dir_protected)
128     {
129         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
130         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
131         return -ENOMEM;
132     }
133     DMSG("proc file created");
134
135     if (start_hook_resources() != 0)
136     {
137         remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
138         remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
139         remove_proc_entry(PROC_DIR_NAME_PROTECTED, NULL);
140         DMSG("Problem in hook functions");
141         return -1;
142     }
143
144     return 0;
145 }
146
147

```

```

148 static void fh_exit(void)
149 {
150     remove_proc_entry(PROC_FILE_NAME_HIDDEN, NULL);
151     remove_proc_entry(PROC_FILE_NAME_PROTECTED, NULL);
152     remove_proc_entry(PROC_DIR_NAME_PROTECTED, NULL);
153     fh_remove_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
154     DMSG("called exit module");
155 }
156
157 module_init(fh_init);
158 module_exit(fh_exit);

```

Листинг 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 #define FILE_NAME (strchr(__FILE__, '/') ? strchr(__FILE__, '/') + 1 :
    __FILE__)
23 #define DMSG(msg_fmt, msg_args...) \
24     printk(KERN_INFO "OS: %s(%04u): " msg_fmt "\n", FILE_NAME, __LINE__,
    ##msg_args)
25

```



```

26 #define MAX_BUF_SIZE 1000
27
28 extern char hidden_files[100][50];
29 extern int hidden_index;
30 extern char protected_files[100][50];
31 extern int protected_index;
32 extern char protected_dirs[100][50];
33 extern int dir_index;
34
35 int check_fs_blocklist(char *input);
36 int check_fs_hidelist(char *input);
37 int check_dir_blocklist(char *input);
38
39
40 static unsigned long lookup_name(const char *name)
41 {
42     struct kprobe kp = {
43         .symbol_name = name
44     };
45     unsigned long retval;
46
47     if (register_kprobe(&kp) < 0)
48     {
49         DMSG("register_kprobe failed for %s", name);
50         return 0;
51     }
52     retval = (unsigned long) kp.addr;
53     unregister_kprobe(&kp);
54     return retval;
55 }
56
57
58 #define USE_FENTRY_OFFSET 0
59
60 struct ftrace_hook {
61     const char *name;
62     void *function;
63     void *original;
64
65     unsigned long address;
66     struct ftrace_ops ops;

```

```

67  };
68
69  static int fh_resolve_hook_address(struct ftrace_hook *hook)
70  {
71      hook->address = lookup_name(hook->name);
72
73      if (!hook->address) {
74          pr_debug("unresolved symbol: %s\n", hook->name);
75          return -ENOENT;
76      }
77
78      *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_SIZE;
79
80      return 0;
81  }
82
83  static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long
      parent_ip,
84
85                                     struct ftrace_ops *ops, struct
      ftrace_regs *fregs)
86  {
87      struct pt_regs *regs = ftrace_get_regs(fregs);
88      struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
89
90      regs->ip = (unsigned long)hook->function;
91  }
92
93  int fh_install_hook(struct ftrace_hook *hook);
94  void fh_remove_hook(struct ftrace_hook *hook);
95  int fh_install_hooks(struct ftrace_hook *hooks, size_t count);
96  void fh_remove_hooks(struct ftrace_hook *hooks, size_t count);
97
98  #define PTREGS_SYSCALL_STUBS 1
99
100 static char *get_filename(const char __user *filename)
101 {
102     char *kernel_filename=NULL;
103
104     kernel_filename = kmalloc(4096, GFP_KERNEL);
105     if (!kernel_filename)

```

```

106         return NULL;
107
108     if (strncpy_from_user(kernel_filename, filename, 4096) < 0) {
109         kfree(kernel_filename);
110         return NULL;
111     }
112
113     return kernel_filename;
114 }
115
116
117 static asmlinkage long (*real_sys_getdents64)(const struct pt_regs *);
118
119 static asmlinkage int fh_sys_getdents64(const struct pt_regs *regs)
120 {
121     struct linux_dirent64 __user *dirent = (struct linux_dirent64
122         *)regs->si;
123     struct linux_dirent64 *previous_dir, *current_dir, *dirent_ker = NULL;
124     unsigned long offset = 0;
125     int ret = real_sys_getdents64(regs);
126
127     dirent_ker = kzalloc(ret, GFP_KERNEL);
128
129     if ((ret <= 0) || (dirent_ker == NULL))
130     {
131         return ret;
132     }
133
134     long error;
135     error = copy_from_user(dirent_ker, dirent, ret);
136
137     if (error)
138     {
139         kfree(dirent_ker);
140         return ret;
141     }
142
143     while (offset < ret)
144     {
145         current_dir = (void *)dirent_ker + offset;

```

```

146         if (check_fs_hidelist(current_dir->d_name))
147         {
148             if (current_dir == dirent_ker)
149             {
150                 ret -= current_dir->d_reclen;
151                 memmove(current_dir, (void *)current_dir +
152                     current_dir->d_reclen, ret);
153                 continue;
154             }
155             previous_dir->d_reclen += current_dir->d_reclen;
156         }
157         else
158         {
159             previous_dir = current_dir;
160         }
161
162         offset += current_dir->d_reclen;
163     }
164
165     error = copy_to_user(dirent, dirent_ker, ret);
166     if (error)
167     {
168         DMSG("copy_to_user error");
169     }
170
171     kfree(dirent_ker);
172     return ret;
173 }
174
175 static asmlinkage long (*real_sys_rename)(struct pt_regs *regs);
176
177 static asmlinkage long fh_sys_rename(struct pt_regs *regs)
178 {
179     long ret=0;
180     char *kernel_filename = get_filename((void*) regs->si);
181
182     if (check_fs_blocklist(kernel_filename) ||
183         check_dir_blocklist(kernel_filename))
184     {

```

```

185         pr_info("blocked to not rename file : %s\n", kernel_filename);
186         kfree(kernel_filename);
187         return -EPERM;
188
189     }
190
191     kfree(kernel_filename);
192     ret = real_sys_rename(regs);
193
194     return ret;
195 }
196
197
198
199 static asmlinkage long (*real_sys_open)(struct pt_regs *regs);
200
201 static asmlinkage long fh_sys_open(struct pt_regs *regs)
202 {
203     char *kernel_filename;
204     kernel_filename = get_filename((void*) regs->si);
205
206     if (check_fs_blocklist(kernel_filename))
207     {
208         DMSG("block open file : %s", kernel_filename);
209         kfree(kernel_filename);
210         return -EPERM;
211     }
212
213     kfree(kernel_filename);
214
215     return real_sys_open(regs);
216 }
217
218
219 static asmlinkage long (*real_sys_unlink) (struct pt_regs *regs);
220
221 static asmlinkage long fh_sys_unlink(struct pt_regs *regs)
222 {
223     char *kernel_filename = get_filename((void*) regs->si);
224

```

```

225     if (check_fs_blocklist(kernel_filename) ||
        check_dir_blocklist(kernel_filename))
226     {
227
228         pr_info("blocked to not remove file : %s\n", kernel_filename);
229         kfree(kernel_filename);
230         return -EPERM;
231
232     }
233
234     kfree(kernel_filename);
235     return real_sys_unlink(regs);
236 }
237
238 #define SYSCALL_NAME(name) ("__x64_" name)
239
240 #define HOOK(_name, _function, _original) \
241 { \
242     .name = SYSCALL_NAME(_name), \
243     .function = (_function), \
244     .original = (_original), \
245 }
246
247 static struct ftrace_hook demo_hooks[] = {
248     HOOK("sys_open", fh_sys_open, &real_sys_open),
249     HOOK("sys_unlink", fh_sys_unlink, &real_sys_unlink),
250     HOOK("sys_rename", fh_sys_rename, &real_sys_rename),
251     HOOK("sys_getdents64", fh_sys_getdents64, &real_sys_getdents64)
252 };
253
254
255 static int start_hook_resources(void)
256 {
257     int err;
258     err = fh_install_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
259     if (err)
260     {
261         return err;
262     }
263     return 0;
264 }

```

Листинг 3 — Файл hook.c

```
1  #include "hooked.h"
2
3  int check_dir_blocklist(char *input)
4  {
5      int i = 0;
6
7      while (i != dir_index)
8      {
9          if(strstr(input, protected_dirs[i]) != NULL)
10             return 1;
11         i++;
12     }
13
14     return 0;
15 }
16
17 int check_fs_blocklist(char *input)
18 {
19     int i = 0;
20
21     while (i != protected_index)
22     {
23         if(strstr(input, protected_files[i]) != NULL)
24             return 1;
25         i++;
26     }
27
28     return 0;
29 }
30
31 int check_fs_hidelist(char *input)
32 {
33     int i = 0;
34
35     while (i != hidden_index)
36     {
37         if(strstr(input, hidden_files[i]) != NULL)
```

```

38         return 1;
39         i++;
40     }
41
42     return 0;
43 }
44
45 int fh_install_hook(struct ftrace_hook *hook)
46 {
47     int error;
48
49     error = fh_resolve_hook_address(hook);
50     if (error)
51     {
52         return error;
53     }
54
55     hook->ops.func = fh_ftrace_thunk;
56     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
57         | FTRACE_OPS_FL_RECURSION
58         | FTRACE_OPS_FL_IPMODIFY;
59
60     error = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
61     if (error)
62     {
63         DMSG("ftrace_set_filter_ip() failed: %d\n", error);
64         return error;
65     }
66
67     error = register_ftrace_function(&hook->ops);
68     if (error)
69     {
70         DMSG("register_ftrace_function() failed: %d\n", error);
71         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
72         return error;
73     }
74
75     return 0;
76 }
77
78

```



```

79 void fh_remove_hook(struct ftrace_hook *hook)
80 {
81     int err;
82
83     err = unregister_ftrace_function(&hook->ops);
84     if (err)
85     {
86         DMSG("unregister_ftrace_function() failed: %d\n", err);
87     }
88
89     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
90     if (err)
91     {
92         DMSG("ftrace_set_filter_ip() failed: %d\n", err);
93     }
94 }
95
96
97 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
98 {
99     int err;
100     size_t i;
101
102     for (i = 0; i < count; i++)
103     {
104         err = fh_install_hook(&hooks[i]);
105         if (err)
106         {
107             while (i != 0)
108             {
109                 fh_remove_hook(&hooks[--i]);
110             }
111             return err;
112         }
113     }
114
115     return 0;
116 }
117
118
119 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)

```

```
120 {  
121     size_t i;  
122  
123     for (i = 0; i < count; i++)  
124     {  
125         fh_remove_hook(&hooks[i]);  
126     }  
127 }
```