Rootkit 工具集周报

2014-01-26

组员：柴宇龙，林坤，霍南风，王春萍

1. 柴宇龙

1) Linux 内核模块

内核模块至少必须有两个函数： init_module 和 cleanup_module。第一个函数是在把模块插入内核时调用的；第二个函数则在删除该模块时调用。一般来说，init_module 可以为内核的某些东西注册一个处理程序，或者也可以用自身的代码来取代某个内核函数(通常是先干点别的什么事，然后再调用原来的函数)。函数 cleanup_module 的任务是清除掉 init_module 所做的一切，这样，这个模块就可以安全地卸载了。

内核模块并不是一个独立的可执行文件，而是一个对象文件，在运行时内核模块被链接到内核中。因此，应该使用-c 命令参数来编译它们。还有一点需要注意，在编译所有内核模块时，都将需要定义好某些特定的符号。

• _ _KERNEL_ _—这个符号告诉头文件：这个程序代码将在内核模式下运行，而不要作为用户进程的一部分来执行。

• MODULE—这个符号告诉头文件向内核模块提供正确的定义。

内核模块主要通过两种方法与进程打交道。一种方法是通过设备文件(例如在目录/ d e v 中的文件)，另一种方法是使用 proc 文件系统。

Kernel 里有一个变量叫 module_list，每当 user 将一个 module 载到 kernel 里的时候，这个 module 就会被记录在 module_list 里面。当 kernel 要使用到这个 module 提供的 function 时，他就会去 search 这个 list，找到 module，然后再使用其提供的 function 或 variable。每一个 module 都能 export 一些 function 或变量来让别人使用。除此之外，module 也能使用已载到 kernel 里的 module 提供的 function。这种情形叫做 module stack。比方说，module A 用到 module B 的东西，那在加载 module A 之前必须要先加载 module B。否则 module A 会无法加载。除了 module 会 export 东西之外，kernel 本身也会 export 一些 function 或 variable。同样的，module 也能使用 kernel 所 export 出来的东西。Linux 模块就是一种可在系统启动后的所有时候动态连入核心的代码块。当我们不再需要它时又能将它从核心中卸载并删除。Linux 模块多指设备驱动、伪设备驱动, 如网络设备和文件系统。

Linux 为我们提供了两个命令：使用 insmod 来显式加载核心模块，使用 rmmod 来卸载模块。同时核心自身也能请求核心后台进程 kerneld 来加载和卸载模块。一旦 Linux 模块被加载则它和普通核心代码相同都是核心的一部分。它们具有和其他核心代码相同的权限和职责；换句话说 Linux 核心模块能象所有核心代码和设备驱动相同使核心崩溃。

核心模块的加载方式有两种。首先一种是使用 insmod 命令手工加载模块。另外一种则是在需要时加载模块，可以称它为请求加载。当核心发现有必要加载某个模块时，如用户安装了核心中不存在的文件系统时，核心将请求核心后台进程(kerneld)准备加载适当的模块。这个核心后台进程仅仅是个带有终极用户权限的普通用户进程。当系统启动时它也被启动并为核心打开了一个进程间通讯（IPC）通道。核心需要执行各种任务时用它来向 kerneld 发送消息。

kerneld 的主要功能是加载和卸载核心模块, 不过它还能执行其他任务, 如通过串行线路建立 PPP 连接并在适当时候关闭它。kerneld 自身并不执行这些任务，它通过某些程序如 insmod 来做此工作。它只是核心的代理，为核心进行调度。

insmod 必须找到需求加载的核心模块。请求加载核心模块一般被保存在/lib/modules/kernel-version 中。这些核心模块和系统中其他程序相同是已连接的目标文件，不过它们被连接成可重定位映象。即映象没有被连接到在特定地址上运行。这些核心模块能是 a.out 或 ELF 文件格式。insmod 将执行一个特权级系统调用来找到核心的输出符号。这些都以符号名及数值形式，如地址值成对保存。核心输出符号表被保存在核心维护的模块链表的第一个 module 结构中，同时 module_list 指针指向此结构。只有特别符号被添加到此表中，它们在核心编译和连接时确定，不是核心每个符号都被输出到其模块中。例如设备驱动为了控制某个特定系统中断而由核心例程调用的"request_irq"符号。我们能通过使用 ksyms 工具或查看/proc/ksyms 来观看当前核心输出符号。ksyms 工具既能显示所有核心输出符号也能只显示那些已加载模块的符号。insmod 将模块读入虚拟内存并通过使用来自核心输出符号来修改其未解析的核心例程和资源的引用地址。这些修改工作采取由 insmod 程序直接将符号的地址写入模块中相应地址来修改内存中的模块映象。

当 insmod 修改完模块对核心输出符号的引用后，它将再次使用特权级系统调用来申请足够的空间来容纳新核心。核心将为其分配一个新的 module 结构及足够的核心内存来保存新模块，并将它放到核心模块链表的尾部。 然后将其新模块标志为 UNINITIALIZED。

模块能通过使用 rmmod 命令来删除，不过请求加载模块将被 kerneld 在其使用记数为 0 时自动从系统中删除。 kerneld 在其每次 idle 定时器到期时都执行一个系统调用以将系统中所有不再使用的请求加载模块从系统中删除。如果核心中的其他部分还在使用某个模块，则此模块不能被卸载。执行 lsmod 将看到每个模块的引用记数。

2) proc 文件系统

通过创建 proc 文件，保存需要隐藏的文件、进程号 pid、目录、网络信息和需要重定位的系统调用函数。

/proc 文件系统是一种内核和内核模块用来向进程 (process) 发送信息的机制 (所以叫做 /proc)。这个伪文件系统让你可以和内核内部数据结构进行交互，获取 有关进程的有用信息，在运行中改变设置 (通过改变内核参数)。 与其他文件系统不同，/proc 存在于内存之中而不是硬盘上。

要在 /proc 文件系统中创建一个虚拟文件，可以使用 create_proc_entry 函数。这个函数接收一个文件名、一组权限和这个文件在 /proc 文件系统中出现的位置。create_proc_entry 的返回值是一个 proc_dir_entry 指针（或者为 NULL，说明在 create 时发生了错误）。然后就可以使用这个返回的指针来配置这个虚拟文件的其他参数，例如在对该文件执行读操作时应该调用的函数。create_proc_entry 的原型和 proc_dir_entry 结构中的一部分如下所示：

```
struct proc_dir_entry *create_proc_entry( const char *name, mode_t mode,
                                          struct proc_dir_entry *parent );
struct proc_dir_entry {
        const char *name;                       // virtual file name
        mode_t mode;                                    // mode permissions
        uid_t uid;                              // File's user id
        gid_t gid;                              // File's group id
        struct inode_operations *proc_iops;     // Inode operations functions
        struct file_operations *proc_fops;      // File operations functions
        struct proc_dir_entry *parent;          // Parent directory
        ...
        read_proc_t *read_proc;                         // /proc read function
```

```
        write_proc_t *write_proc;          // /proc write function
        void *data;                                // Pointer to private data
        atomic_t count;                            // use count
        ...
};
void remove_proc_entry( const char *name, struct proc_dir_entry *parent );
```

可以使用 read_proc 和 write_proc 命令来插入对这个虚拟文件进行读写。

要从 /proc 中删除一个文件，可以使用 remove_proc_entry 函数。要使用这个函数，我们需要提供文件名字符串，以及这个文件在 /proc 文件系统中的位置（parent）。这个函数原型如上所示。

parent 参数可以为 NULL（表示 /proc 根目录），也可以是很多其他值，这取决于我们希望将这个文件放到什么地方。下表列出了可以使用的其他一些父 proc_dir_entry，以及它们在这个文件系统中的位置。

proc_dir_entry	在文件系统中的位置
proc_root_fs	/proc
proc_net	/proc/net
proc_bus	/proc/bus
proc_root_driver	/proc/driver

可以使用 write_proc 函数向 /proc 中写入一项。这个函数的原型如下：

```
int mod_write( struct file *filp, const char __user *buff,
               unsigned long len, void *data );
```

filp 参数实际上是一个打开文件结构（可以忽略这个参数）。buff 参数是传递的字符串数据。缓冲区地址实际上是一个用户空间的缓冲区，因此不能直接读取它。len 参数定义了在 buff 中有多少数据要被写入。data 参数是一个指向私有数据的指针(proc_dir_entry 类型)。

Linux 提供了一组 API 来在用户空间和内核空间之间移动数据。对于 write_proc 的情况来说，使用 copy_from_user 函数来维护用户空间的数据。

可以使用 read_proc 函数从一个 /proc 项中读取数据（从内核空间到用户空间）。这个函数的原型如下：

```
int mod_read( char *page, char **start, off_t off,
              int count, int *eof, void *data );
```

page 参数是这些数据写入到的位置，其中 count 定义了可以写入的最大字符数。在返回多页数据（通常一页是 4KB）时，需要使用 start 和 off 参数。当所有数据全部写入之后，就需要设置 eof（文件结束参数）。与 write 类似，data 表示的也是私有数据。此处提供的 page 缓冲区在内核空间中。因此，可以直接写入，而不用调用 copy_to_user。

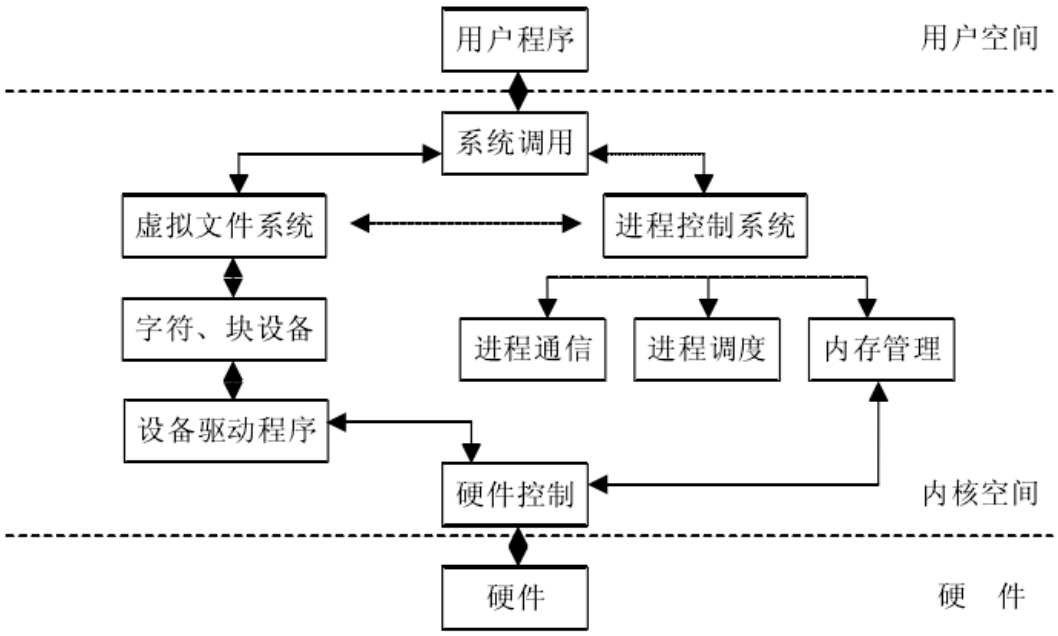
还可以使用 proc_mkdir、symlinks 以及 proc_symlink 在 /proc 文件系统中创建目录。对于只需要一个 read 函数的简单 /proc 项来说，可以使用 create_proc_read_entry，这会创建一个 /proc 项，并在一个调用中对 read_proc 函数进行初始化。这些函数的原型如下所示。

```
/* Create a directory in the proc filesystem */
struct proc_dir_entry *proc_mkdir( const char *name,
                                   struct proc_dir_entry *parent );
```

```
/* Create a symlink in the proc filesystem */
struct proc_dir_entry *proc_symlink( const char *name,
                                     struct proc_dir_entry *parent,
                                     const char *dest );
/* Create a proc_dir_entry with a read_proc_t in one call */
struct proc_dir_entry *create_proc_read_entry( const char *name,
                                               mode_t mode,
                                               struct proc_dir_entry *base,
                                               read_proc_t *read_proc,
                                               void *data );
/* Copy buffer to user-space from kernel-space */
unsigned long copy_to_user( void __user *to,
                            const void *from,
                            unsigned long n );
/* Copy buffer to kernel-space from user-space */
unsigned long copy_from_user( void *to,
                              const void __user *from,
                              unsigned long n );
/* Allocate a 'virtually' contiguous block of memory */
void *vmalloc( unsigned long size );
/* Free a vmalloc'd block of memory */
void vfree( void *addr );
/* Export a symbol to the kernel (make it visible to the kernel) */
EXPORT_SYMBOL( symbol );
/* Export all symbols in a file to the kernel (declare before module.h) */
EXPORT_SYMTAB
```

2. **王春萍** 隐藏文件模块。分析 linux 的 ls，find 等系统命令中显示文件信息的过程，得到相应的系统函数。

1) 了解了 linux 系统调用的过程及其各部分的依赖关系。



用户程序调用系统调用，通过软中断 0x80 进入内核空间，通过中断服务程序找到系统调用表，然后找到系统调用地址进行系统调用。系统调用大致分为虚拟文件系统和进程控制系统的操作。其中虚拟文件系统包括字符块设备，然后通过设备驱动程序实现硬件控制，而进程控制系统包括进程通信、进程调度和内存管理。

2) 重点了解学习了文件隐藏的大致的方法（具体实现有待学习）

linux 使用 sys_getdents64()系统调用来获取目录下的文件和子目录信息，然后将这些信息返回给 ls 等命令程序进行显示，其原型为：int sys_getdents(unsigned int fd, struct dirent *dirp,unsigned int count)，其中 fd 为指向目录文件的文件描述符，该函数根据 fd 所指向的目录文件读取相应 dirent 结构，并放入 dirp 中，其中 count 为 dirp 中返回的数据量，正确时该函数返回值为填充到 dirp 的字节数。可以通过 strace 来观察，例如 strace ls 将列出命令 ls 用到的系统调用，从中可以发现 ls 的流程。当查询文件或者目录的相关信息时，Linux 系统用 sys_getedents 来执行相应的查询操作，并把得到的信息传递给用户空间运行的程序，修改该系统调用，去掉结果中与某些特定文件的相关信息，这样所有利用该系统调用的程序将不能显示该文件，从而达到隐藏的目的。

3. 霍南风

擦除模块的知识预备：基址重定位

当链接器创建可执行文件时，它假定这个文件会被映射到内存的某一个地址上。基于此，链接器把代码和数据项的真实地址放在可执行文件中。如果由于某些原因，这个可执行文件不能被加载到这个事先设定的地址上，那么链接器放在这个映像中的地址就变成错误的了。存储在.reloc节中的信息允许PE加载器修正已加载映像中的这些地址以便使它们重新成为正确的地址。

如果加载器可以把这个文件加载到事先由链接器设定的基地址上，那么.reloc节就变成多余的了，因此可以被忽略。.reloc节中的每个元素之所以被称为基址重定位信息是因为它们的使用依赖于映像的基地址。

注意：链接器链接时会按ImageBase的值填入一个地址（例如：设默认加载地址为：0x04000000）间接调用外部函数时，汇编代码：FF 15 C0 83 41 00 即（0x004183C0），**（以上是链接器做的事）**。

但是当印象加载不是0x00400000,该函数需要重定位。这个情况下，这个 0x004183C0 地址就会产生错误，加载器需要将它重新定位在：0x012C83C0 地址上。（设其加载地址为0x012b0000）

这个 0x012C83C0 是等于：0x004183C0 - 0x00400000 + 0x012b0000 = 0x012C83C0

注意：

位域	位	长度	描述
Type	[15:12]	4 Bits	高 4 位用来表示 Base Relocation Table 的类型
Offset	[11:0]	12 Bits	低 12 位是 RVA 值，指出需要重定位的位置

这个 12 位的 Offset 值是基于 IMAGE_BASE_RELOCATION 结构的 VirtualAddress， 而 VirtualAddress 是基于 ImageBase 的 RVA 值

因此最终的 Offset 值应该是 ImageBase + VirtualAddress（reloc.virtualaddress） + Offset(RVA)

这是加载器做的事。

```
00428000  00 10 01 00          // VirtualAddress = 0x00011000
00428004  DC 00 00 00          // SizeOfBlock = 0x000000DC

// Entries

00428008  93 34                // type = 3, offset = 493
0042800A  9F 34                // type = 3, offset = 49F
0042800C  AF 34                // type = 3, offset = 4AF
0042800E  BB 34                // type = 3, offset = 4BB
00428010  F4 34                // type = 3, offset = 4F4
00428012  10 35                // type = 3, offset = 510
00428014  2F 35                // type = 3, offset = 52F
00428016  46 35                // type = 3, offset = 546
00428018  59 35                // type = 3, offset = 559
0042801A  6F 35                // type = 3, offset = 56F
0042801C  94 35                // type = 3, offset = 594
0042801E  A0 35                // type = 3, offset = 5A0
```

该表中两个字节就可以存储入口信息。

```
 11000 RVA,       DC SizeOfBlock
   493  HIGHLOW          00417200  ?szTitle@@3PA_WA (wchar_t * szTitle)
   49F  HIGHLOW          004183C0  __imp__LoadStringW@16
   4AF  HIGHLOW          00417138  ?szWindowClass@@3PA_WA (wchar_t * szWindowClass)
   4BB  HIGHLOW          004183C0  __imp__LoadStringW@16
   4F4  HIGHLOW          004183C4  __imp__LoadAcceleratorsW@8
   510  HIGHLOW          004183C8  __imp__GetMessageW@16
   52F  HIGHLOW          004183CC  __imp__TranslateAcceleratorW@12
   546  HIGHLOW          004183D0  __imp__TranslateMessage@4
   559  HIGHLOW          004183D4  __imp__DispatchMessageW@4
   56F  HIGHLOW          00411590
   594  HIGHLOW          00411598
   5A0  HIGHLOW          004115A4
```

疑问：

1. 文献中没给出这些字符串是怎么和这地址（如RVA:493）一一对应的！

   字符串是存在资源段中（.rsrc）的？

域	.reloc 节
VirtualSize	0x00000564
VirtualAddress	0x00028000
SizeOfRawData	0x00000600
PointerToRawData	0x00015400
PointerToRelocations	0
PointerToLinenumbers	0
NumberOfRelocations	0
NumberOfLinenumbers	0
Characteristics	0x42000040

文献没给出每个段的具体大小，上面问题下一阶段工作中解决.

总结：

重定位表的作用：一旦模块没有价加载到预期地址，加载器就会根据基址重定位表去修正哪些需要修正的指令，这样程序就可以正常执行了。

说白了，该表是为加载器服务。

4. 林坤

家里有事，下周补上，请见谅。