《漏洞利用及渗透测试基础》实验报告

**实验名称：**

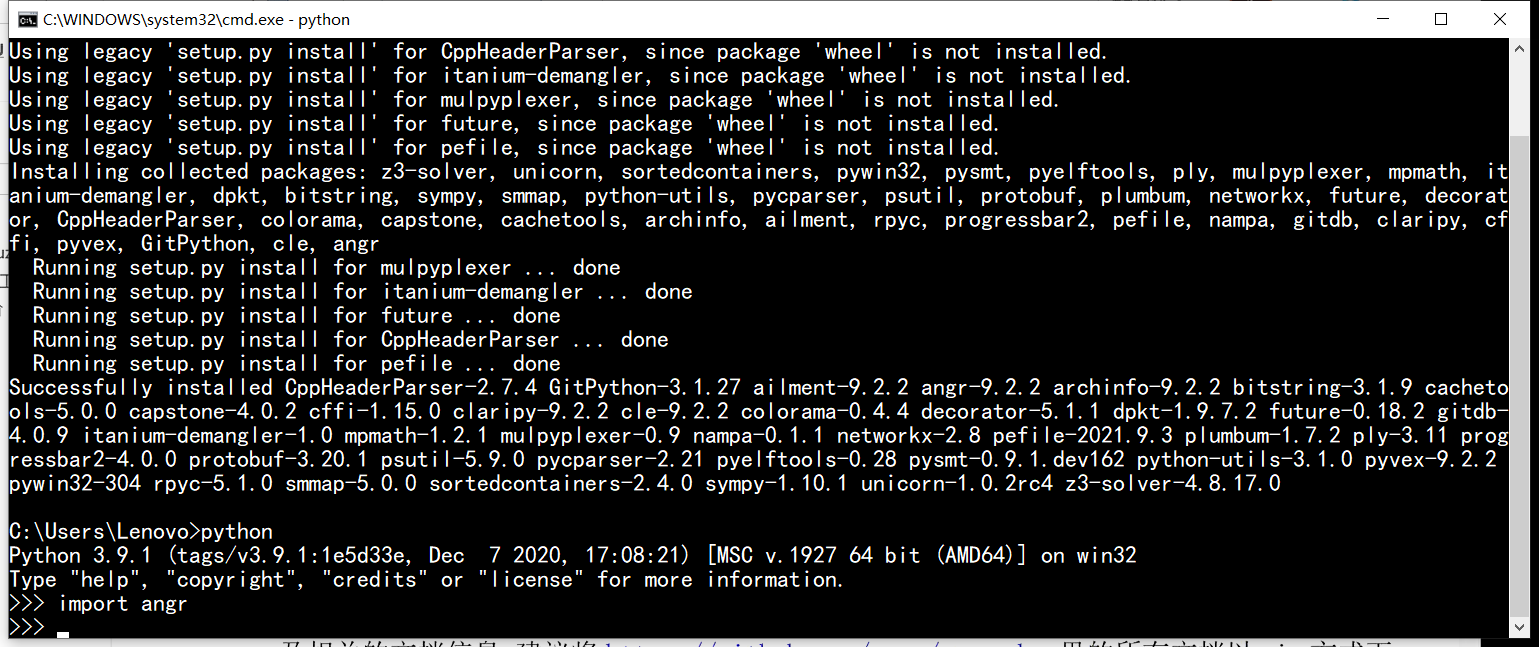
Angr应用示例

**实验要求：**

根据课本8.4.3章节，复现sym-write示例的两种angr求解方法，并就如何使用angr以及怎么解决一些实际问题做一些探讨。

**实验过程：**

**1.安装**angr



angr是一个多架构的二进制分析平台，具备对二进制文件的动态符号执行能力和多种静态分析能力。其基本的使用方法也比较简单，直接对二进制文件进行符号执行，获取相应结果。

**2.下载官方文档**

在相应github开源仓库中，我们可以直接下载所有的官方帮助文档，其中包含许多例子可以很好地帮助我们去学习。本次使用的例子就是sys\_write。



**3.分析测试文件的C源代码**

|  |
| --- |
| #include <stdio.h>  char u=0;  int main(void)  {  int i, bits[2]={0,0};  for (i=0; i<8; i++) { //该循环统计字符u中零和一的个数  bits[(u&(1<<i))!=0]++;  }  if (bits[0]==bits[1]) { //如果u中零和一的个数相等，则打印“you win!”  printf("you win!");  }  else {  printf("you lose!");  }  return 0;  } |

在样例issue.c中，首先通过循环结合位运算将一个单字节char类型数据u的0位的个数和1位的个数分别存储到bits[0]和bits[1]中。然后通过if分支判断u的字节中的0和1的个数是否相等，若相等打印“you win!”。将u作为符号执行的输入，再通过到达相应的输出得到u的约束条件。而通过约束求解器就可以解出符合我们要求的u的解集。

**4.解法一**

|  |
| --- |
| import angr  import claripy  def main():  # 1. 新建一个工程，导入二进制文件，后面的选项是选择不自动加载依赖项，不会自动载入依赖的库  p = angr.Project('./issue', load\_options={"auto\_load\_libs": False})  # 2. 初始化一个模拟程序状态的SimState对象state，该对象包含了程序的内存、寄存器、文件系统数据、符号信息等等模拟运行时动态变化的数据  # blank\_state():可通过给定参数addr的值指定程序起始运行地址  # entry\_state():指明程序在初始运行时的状态，默认从入口点执行  # SYMBOLIC\_WRITE\_ADDRESSES：允许通过具体化策略处理符号地址的写操作  state = p.factory.entry\_state(add\_options={angr.options.SYMBOLIC\_WRITE\_ADDRESSES})  # 3. 创建一个符号变量，这个符号变量以8位bitvector形式存在，名称为u  u = claripy.BVS("u", 8)  # 把符号变量保存到指定的地址中，这个地址是就是二进制文件中.bss段u的地址  state.memory.store(0x804a021, u)  # 4. 创建一个Simulation Manager对象，这个对象和我们的状态有关系  sm = p.factory.simulation\_manager(state)  # 5. 使用explore函数进行状态搜寻，检查输出字符串是win还是lose  # state.posix.dumps(1)获得所有标准输出  # state.posix.dumps(0)获得所有标准输入  def correct(state):  try:  return b'win' in state.posix.dumps(1)  except:  return False  def wrong(state):  try:  return b'lose' in state.posix.dumps(1)  except:  return False  # 进行符号执行得到想要的状态，即得到满足correct条件且不满足wrong条件的state  sm.explore(find=correct, avoid=wrong)  # 也可以写成下面的形式，直接通过地址进行定位  # sm.explore(find=0x80484e3, avoid=0x80484f5)  # 获得到state之后，通过solver求解器，求解u的值  # eval\_upto(e, n, cast\_to=None, \*\*kwargs) 求解一个表达式指定个数个可能的求解方案 e - 表达式 n - 所需解决方案的数量  # eval(e, \*\*kwargs) 评估一个表达式以获得任何可能的解决方案。 e - 表达式  # eval\_one(e, \*\*kwargs) 求解表达式以获得唯一可能的解决方案。 e - 表达式  return sm.found[0].solver.eval\_upto(u, 256)  if \_\_name\_\_ == '\_\_main\_\_':  # repr()函数将object对象转化为string类型  print(repr(main())) |

解法1大致流程如下：

(1) 在主函数中首先创建一个新项目，传入可执行二进制文件的路径。为了提高分析效率，auto\_load\_libs设置为false，将不会自动载入依赖的库。

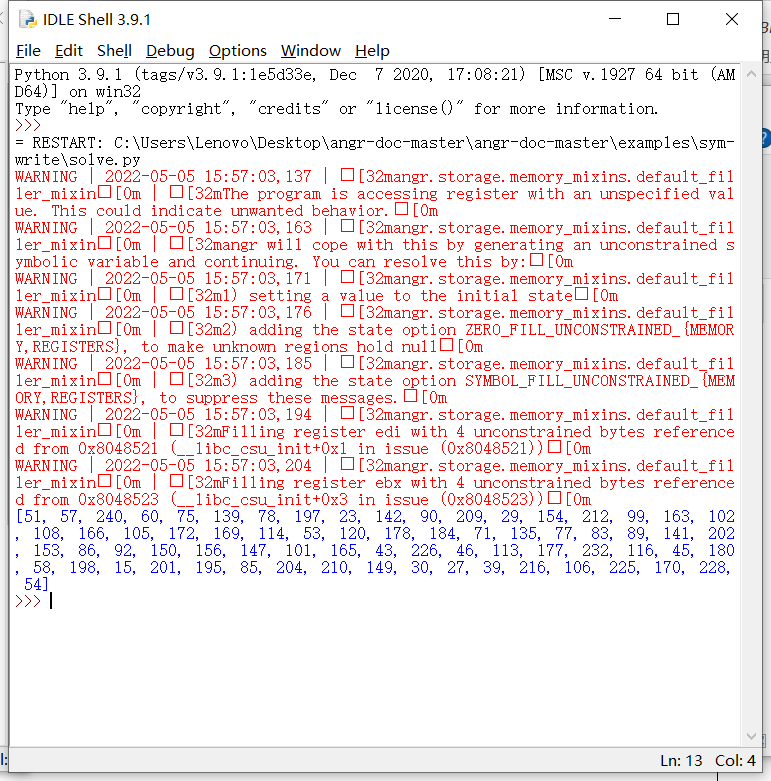
(2) 接下来设置程序入口状态。SimState能够模拟程序运行时的内存，寄存器，符号变量，文件系统等等运行时的信息，是符号执行的载体。

(3) 将要求解的变量符号化，这里符号化后的变量存在二进制文件的存储区。(4) 新建一个SimManager用来对状态进行管理求解。接下来的correct函数和wrong函数作为符号运行时结果的判断。如果输出中含有“win”则正确，输出中含有“loss”则避免。

(5) 利用explore进行符号执行，这里也可以用find=0x80484e3, avoid=0x80484f5来代替，即通过符号执行是否到达特定代码区的地址。

(6) 通过solver求解器，求解u的值。

运行python程序结果如下，可以看到输出了一系列符合条件的u值。

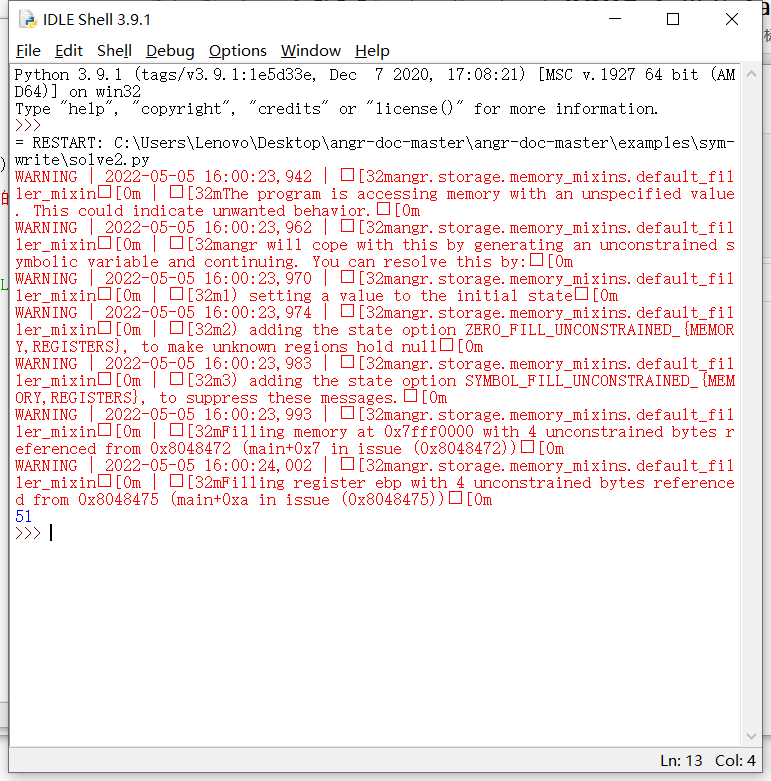


**5.解法二**

|  |
| --- |
| #!/usr/bin/env python  # coding=utf-8  import angr  import claripy  def hook\_demo(state):  state.regs.eax = 0    p = angr.Project("./issue", load\_options={"auto\_load\_libs": False})  # hook函数：addr为待hook的地址  # hook为hook的处理函数，在执行到addr时，会执行这个函数，同时把当前的state对象作为参数传递过去  # length 为待hook指令的长度，在执行完 hook 函数以后，angr 需要根据 length 来跳过这条指令，执行下一条指令  # hook 0x08048485处的指令（xor eax,eax），等价于将eax设置为0  # hook并不会改变函数逻辑，只是更换实现方式，提升符号执行速度  p.hook(addr=0x08048485, hook=hook\_demo, length=2)  state = p.factory.blank\_state(addr=0x0804846B, add\_options={"SYMBOLIC\_WRITE\_ADDRESSES"})  u = claripy.BVS("u", 8)  state.memory.store(0x0804A021, u)  sm = p.factory.simulation\_manager(state)  sm.explore(find=0x080484DB)  st = sm.found[0]  print(repr(st.solver.eval(u))) |

解法2使用hook插入的方式运行。根据样例，编写hook\_demo。hook\_demo中的语句等效为重置寄存器eax(xor eax, eax)。此处替换的hook代码和原来是等效的，此处只是作为一个例子。在以后的使用中我们可以将一些复杂的系统函数调用，比如printf等，进行hook，以提升符号执行的性能，相应的可以提高分析速度或者起到其它效果。此外，解法2的explore函数中只传入了一个参数，为find=0x080484DB，显然，这样做也是合理的，因位我们的目的只是找到win的位置。

运行python程序结果如下，可以看到输出了符合条件的u的一个值。



**心得体会：**

通过本次实验，我了解了更多的符号执行的相关知识，学习了利用angr工具进行符号执行对程序进行分析。并进一步实现了angr的两种求解方法。我以后会在该领域进行更多的尝试和学习，提高软件漏洞攻防能力。