### Angr应用示例

Angr是一个二进制代码分析工具，能够自动化完成二进制文件的分析，并找出漏洞。在二进制代码中寻找并且利用漏洞是一项非常具有挑战性的工作，它的挑战性主要在于人工很难直观的看出二进制代码中的数据结构、控制流信息等。

Angr是一个基于python的二进制漏洞分析框架，它将以前多种分析技术集成进来，它能够进行动态的符号执行分析（如KLEE和Mayhem），也能够进行多种静态分析。

**1. Angr安装**

**Windows下安装Angr**。首先安装Python3，如果安装了就忽略。可以到python官方网站下载安装版本，选择将python增加到path中。然后，打开命令控制台，使用PIP命令安装angr：pip install angr。

**测试安装。**输入命令python，进入python界面，然后输入import angr，如果成功，则说明安装没有问题。如下图8-20：

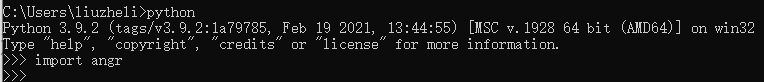


图8-20 测试成功安装angr

**Angr官方手册。**网址<https://docs.angr.io/introductory-errata/install>提供了各类操作系统的安装方法。此外，GitHub上有angr的开源项目https://github.com/angr以及相关的文档信息，建议将<https://github.com/angr/angr-doc>里的所有文档以zip方式下载到本地。

**2. Angr示例**

在angr-doc里有各类Example，展示了Angr的用法，比如cmu\_binary\_bomb、simple\_heap\_overflow等二进制爆破、堆溢出等漏洞挖掘、软件分析的典型案例。下面，我们以sym-write为例子，来说明angr的用法。

源码issue.c（详见angr-doc-master\examples\sym-write\issue.c）如下：

|  |
| --- |
| #include <stdio.h>  char u=0;  int main(void)  {  int i, bits[2]={0,0};  for (i=0; i<8; i++) {  bits[(u&(1<<i))!=0]++;  }  if (bits[0]==bits[1]) {  printf("you win!");  }  else {  printf("you lose!");  }  return 0;  } |

源码solve.py（详见angr-doc-master\examples\sym-write\solve.py）如下：

|  |
| --- |
| 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  # add\_options获取一个独立的选项来添加到某个state中，更多选项说明见https://docs.angr.io/appendix/options  # 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())) |

上述代码定义了一个main函数。

整个python程序将执行print(repr(main()))语句，进而，将main函数的返回值打印出来，repr()函数将object对象转化为string类型。

在上述Angr示例中，几个关键步骤如下：

（1）新建一个Angr工程，并且载入二进制文件。auto\_load\_libs设置为false，将不会自动载入依赖的库，默认情况下设置为false。如果设置为true，转入库函数执行，有可能给符号执行带来不必要的麻烦。

（2）初始化一个模拟程序状态的SimState对象state（使用函数entry\_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。此外，也可以使用函数blank\_state()初始化模拟程序状态的对象state，在该函数里可通过给定参数addr的值指定程序起始运行地址。

（3）将要求解的变量符号化，注意这里符号化后的变量存在二进制文件的存储区。

（4）创建模拟管理器（Simulation Managers）进行程序执行管理。初始化的state可以经过模拟执行得到一系列的states，模拟管理器sm的作用就是对这些states进行管理。

（5）进行符号执行得到想要的状态，得到想要的状态。上述程序所表达的状态就是，符号执行后，源程序里打印出的字符串里包含win字符串，而没有包含lose字符串。在这里，状态被定义为两个函数，通过符号执行得到的输出state.posix.dumps(1)中是否包含win或者lose的字符串来完成定义。

注意：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。使用IDA反汇编可知0x80484e3是printf("you win!")对应的汇编语句；0x80484f5则是printf("you lose!")对应的汇编语句。

（6）获得到state之后，通过solver求解器，求解u的值。

这里有多个函数可以使用，eval\_upto(e, n, cast\_to=None, \*\*kwargs) 求解一个表达式多个可能的求解方案，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；eval\_one(e, \*\*kwargs)求解表达式以获得唯一可能的解决方案。

**实验验证。**在windows 10环境下，选择填写的solve.py，点右键选择Edit with IDLE🡪Edit with IDLE 3.9 (64 bit)，将弹出界面，选择Run🡪run model，界面如下图8-21：

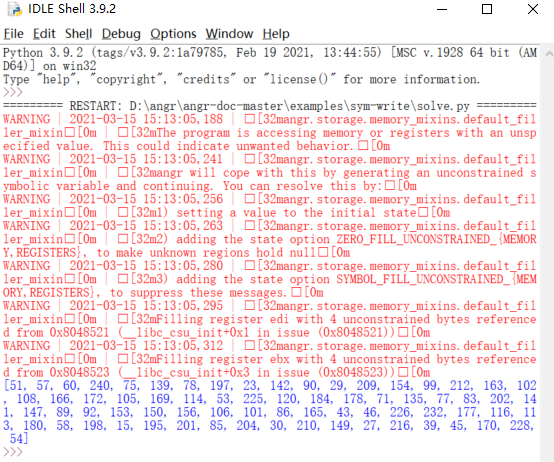


图8-21 程序运行结果验证

蓝色部分，就是输出的u的求解的结果，因为我们采用了eval\_upto函数，给出了多个解，对每个解我们都可以带入源程序进行验证。

**其他解法。**对于上述程序，我们也可以采用下面的代码来进行求解：

|  |
| --- |
| #!/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))) |

上述代码与前面的解法有三处区别：

（1）采用了hook函数，将0x08048485处的长度为2的指令通过自定义的hook\_demo进行替代，功能是一致的，原始xor eax, eax和state.regs.eax = 0是相同的作用，这里只是演示，可以将一些复杂的系统函数调用，比如printf等，可以进行hook，提升符号执行的性能。

（2）进行符号执行得到想要的状态，有变化，变更为find=0x080484DB。因为源程序win和lose是互斥的，所以，只需要给定一个find条件即可。

（3）最后，eval(u)替代了原来的eval\_upto，将打印一个结果出来。

注意：Angr的功能非常强大，远比上面举得例子复杂，还需要同学们自己去查阅angr的用户手册，对照Examples自行学习。