《软件安全》实验报告

姓名：李雅帆　　学号：2213041　　班级：信息安全

一、实验名称

Angr 应用实例实验

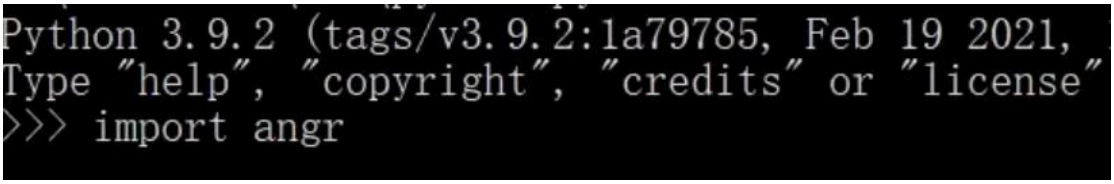
二、实验要求

根据课本 8.4.3 章节，复现 sym-write 实例的两种 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。

三、实验过程

1.安装 python3 并使用 pip 命令在控制台安装 angr。

测试 Angr 是否安装成功，能够正常导入说明 angr 包安装成功。



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。

2.以 sym-write 为例子，说明 angr 的用法

源码 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;
}
```

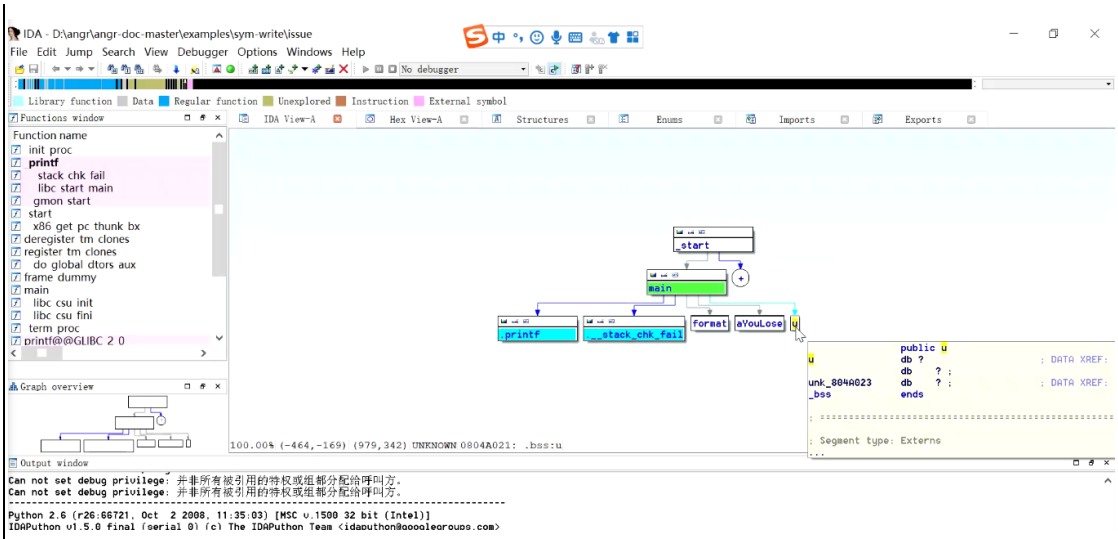
在代码中，我们想用符号执行工具找到哪个 u 的值满足我们的条件。

此时我们已经写好了 linux 下的可执行文件，要对进行符号执行，在源码脚本中，定义了一个主函数，在入口处会执行主函数，在 print 函数里，将主函数返回的对象类型打印成字符串类型。

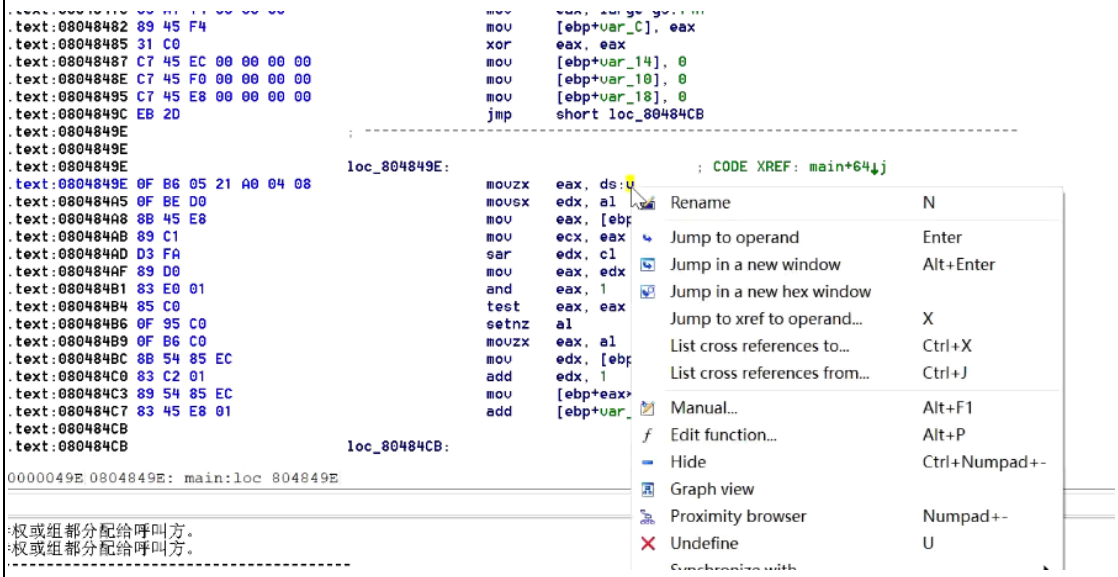
在定义的主函数中,创建了一个工程,在工程共定义了二进制文件和我们需要的启动项，False 会自动加载依赖项。然后我们创建了一个模拟程序状态的对象 state，这个对象包含了程序在运行过程中的数据、符号信息等，创建了 sym-state 的对象，默认从程序入口点执行。

源代码脚本：

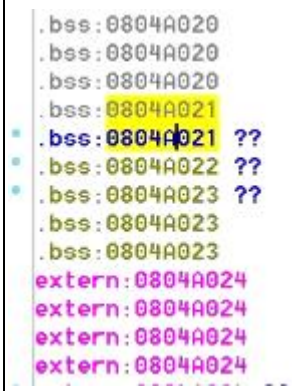
```
def main():
    # 1. 新建一个工程，导入二进制文件，后面的选项是选择不自动加载依赖项，不会自动载入依赖的库
    p = angr.Project('./issue', load_options={"auto_load_libs": False})//创建工程 指定启动项
auto_load_libs 设置为 false，将不会自动载入依赖的库，默认情况下设置为 false。如果设置为 true，转入库函数执行，有可能给符号执行带来不必要的麻烦。

    # 2. 初始化一个模拟程序状态的 SimState 对象 state，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})// 默认从入口点开始执行

    # 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)

对于地址，我们可以进行分析：
```



在 IDA 中打开文件，会询问我们是否需要优化，此时我们可以转到文本视图，此时可以发现我 i 们已经访问到 u。



u 也可以转到对应的位置，此时我们可以看到 u 的地址，就是 804a021，与我们分析的二进制文件中 u 的地址对应。




```
# 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)
```

这句代码通过函数调用执行，Sm 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. 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)//搜索满足状态的特定结果（包含 win ）
# 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)//地址可达
```

```
.text:080484B9 0F B6 C0
.text:080484BC 8B 54 85 EC
.text:080484C0 83 C2 01
.text:080484C3 89 54 85 EC
.text:080484C7 83 45 E8 01
.text:080484CB
.text:080484CB
.text:080484CB 83 7D E8 07
.text:080484CF 7E CD
.text:080484D1 8B 55 EC
.text:080484D4 8B 45 F0
.text:080484D7 39 C2
.text:080484D9 75 12
.text:080484DB 83 EC 0C
.text:080484DE 68 A0 85 04 08
.text:080484E3 E8 48 FE FF FF
.text:080484E8 83 C4 10
.text:080484EB EB 10
.text:080484ED
.text:080484ED
.text:080484ED
.text:080484ED 83 EC 0C
.text:080484F0 68 A9 85 04 08
.text:080484F5 E8 36 FE FF FF
.text:080484FA 83 C4 10
```

```
# 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)//求解结果最大数量


if __name__ == '__main__':
    # repr()函数将 object 对象转化为 string 类型
    print(repr(main()))
```

此时运行这个程序，结果如下：

```
[51, 57, 240, 60, 75, 139, 78, 197, 23, 142, 90, 29, 209, 154, 99, 212, 163, 102,
 108, 166, 172, 105, 169, 114, 120, 53, 178, 184, 71, 135, 77, 83, 202, 89, 147,
 86, 153, 92, 150, 156, 106, 101, 141, 165, 43, 113, 232, 226, 177, 116, 46,
180, 45, 58, 198, 15, 201, 195, 85, 204, 30, 149, 210, 27, 216, 39, 225, 170,
228, 54]

进程已结束,退出代码0
```

对于目标程序，也有第二种方式，如果把 u 看作一种符号时，也可以求出 u 的结果。

对于上述程序，也可以使用下面的代码来求解：

```
#!/usr/bin/env python
# coding=utf-8
import angr
import claripy

def hook_demo(state):
    state.regs.eax = 0//在这里使用了一个简单的 Hook 函数

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)/对指令地址进行 Hook，按照我指定的方式进行执行。指令长度是 2，提高了符号执行速度和可替代的功能。
这样便于进行复杂处理
state = p.factory.blank_state(addr=0x0804846B, //整个程序的入口点

add_options={"SYMBOLIC_WRITE_ADDRESSES"})
u = claripy.BVS("u", 8)//定义符号变量
```

```
state.memory.store(0x0804A021, u)
sm = p.factory.simulation_manager(state)
sm.explore(find=0x080484DB)//因为求解的目标路径里有 if else 分支，不用 avoid

st = sm.found[0]

    print(repr(st.solver.eval(u)))
```

```
.text:08048485 31 C0                                    xor     eax, eax
.text:08048487 C7 45 FC 00 00 00 00                     mov     [ebp+var_4], 0
```

```
.text:0804846B
.text:0804846B
.text:0804846B
.text:0804846B
.text:0804846B
```

```
.text:080484D9 75 12                                    jnz     short loc_80484ED
.text:080484DB 83 EC 0C                                 sub     esp, 0Ch
.text:080484DE 68 A0 85 04 08                           push    offset format   ; "you win!"
.text:080484E3 E8 48 FE FF FF                           call    _printf
```

四、心得体会

通过这次 Angr 应用实例实验，我进一步了解了 Angr 的安装和基本使用方法，成功复现了 sym-write 实例并探索了其两种求解方法。实验过程中，我学会了如何利用 Angr 进行二进制文件分析和漏洞挖掘，通过符号执行自动化找到满足特定条件的输入值。这种自动化分析大大提高了效率，避免了人工分析的繁琐和错误。

此外，我还体会到 Angr 在动态符号执行和静态分析方面的强大能力，能够处理复杂的二进制文件结构和控制流信息。通过实验，我不仅加深了对二进制代码分析的理解，也感受到 Angr 在实际漏洞挖掘和软件分析中的广泛应用前景。这次实验不仅提升了我的技术能力，也为我今后在二进制分析领域的学习和研究提供了宝贵经验。