

《软件安全》实验报告

姓名：蒋柄言 学号：2313546 班级：信息安全班

一、实验名称

Angr 应用进阶

二、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。

三、实验过程

(一) 安装 Angr: 我已经安装了 Python 3，只需要打开命令控制台，执行命令 `pip install angr`。安装完成后进入 Python 环境，输入 `import angr`，发现不报错，说明安装成功。

```
C:\Users\Lenovo>python
Python 3.13.0 (tags/v3.13.0:60403a5, Oct  7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
```

访问 <https://github.com/angr/angr-doc> 下载 Angr 的官方文档，下载得到 `angr-doc-master` 压缩包，解压后里面有一个 `examples` 文件夹，通常包含了一些示例代码，这些代码展示了如何使用 Angr 进行不同的二进制分析任务。

下面以 sym-write 为例子，来说明 Angr 的基本用法。

(二) Angr 第一种求解方法： `sym-write` 与符号执行相关，专门用于展示如何在符号执行过程中模拟“写操作”的分析。现在有以下 C 语言代码：

c

```
#include <stdio.h>
char u = 0; // 定义了一个字符类型的变量 u，并初始化为 0
int main(void) {
    int i, bits[2] = { 0, 0 }; // 定义了一个整数数组 bits，长度为 2，用来分别统计 bits[0]
    和 bits[1]
    for (i = 0; i < 8; i++) { // 循环 8 次，因为一个字符类型的变量（char）通常占 1 字节，
    也就是 8 位
        bits[(u & (1 << i)) != 0]++;
        // 按位与操作，检查 u 的第 i 位是否为 1
    }
    if (bits[0] == bits[1]) { // 如果 bits[0] 和 bits[1] 相等
        printf("you win!");
    } else {
        printf("you lose!");
    }
}
```

```
    return 0;  
}
```

脚本 solve.py 的代码如下：

```
python  
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 state 之后，通过 solver 求解器，求解 u 的值
# eval_up_to(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_up_to(u, 256)

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

```

solve.py 使用符号执行分析程序路径，探索不同的 u 值来找到满足 "win" 输出的输入。当符号执行探索时，由于 u 是符号化的，符号执行会生成多个路径，并为每个路径计算相应的 u 值。因此，最终会得到多个可能的 u 值，这些值满足程序输出 "win" 的条件。solve.py 的主要步骤如下：

1. 加载二进制文件：通过 angr.Project 加载 issue 程序，auto_load_libs=False 是为了避免自动加载共享库。
2. 初始化符号执行的初始状态：创建一个初始状态，add_options = {angr.options.SYMBOLIC_WRITE_ADDRESSES} 表示启用符号写入地址。这意味着 Angr 将符号化内存写操作的地址，而不是将其具体化。
3. 符号化 u 变量并存储到内存：创建一个符号变量 u，并将其存储到内存地址 0x804a021，这表示变量 u 的值将是符号化的，Anger 会尝试不同的值。
4. 创建模拟执行管理器：创建一个模拟执行管理器（SimulationManager），它负责管理不同的符号执行路径。
5. 定义目标状态和回避状态：correct(state) 用于判断程序输出是否包含 "win"；wrong(state) 用于判断程序输出是否包含 "lose"。
6. 路径探索：在符号执行中，find=correct 表示寻找满足 "win" 输出的路径，avoid=wrong 表示避免满足 "lose" 输出的路径。
7. 求解 u 的可能值：找到符号执行探索中的第一个满足条件的路径，并通过求解器 solver.eval_up_to(u, 256) 来获取符号变量 u 的所有可能值（最多 256 个）。

执行以上 Python 代码，得到如下结果：

The screenshot shows the Python shell interface with the following content:

```

File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)]
] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: E:\Python Code\pythonProject\test001.py =====
[33mWARNING[0m | 2025-05-08 22:55:35, 005 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mThe program is accessing register with an unspecified value. This could indicate unwanted behavior.[0m
[33mWARNING[0m | 2025-05-08 22:55:35, 042 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mangr will cope with this by generating an unconstrained symbolic variable and continuing. You can resolve this by:[0m
[33mWARNING[0m | 2025-05-08 22:55:35, 053 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31m1) setting a value to the initial state[0m
[33mWARNING[0m | 2025-05-08 22:55:35, 060 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31m2) adding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions hold null[0m
[33mWARNING[0m | 2025-05-08 22:55:35, 071 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31m3) adding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messages.[0m
[33mWARNING[0m | 2025-05-08 22:55:35, 080 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mFilling register edi with 4 unconstrained bytes referenced from 0x8048521 (_libc_csu_init+0x1 in issue (0x8048521)))[0m
[33mWARNING[0m | 2025-05-08 22:55:35, 090 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mFilling register ebx with 4 unconstrained bytes referenced from 0x8048523 (_libc_csu_init+0x3 in issue (0x8048523)))[0m
[51, 57, 60, 240, 75, 139, 78, 197, 23, 142, 90, 29, 209, 154, 212, 99, 163, 102, 108, 166, 172, 105, 169, 114, 53, 120, 225, 184, 178, 71, 135, 77, 83, 89, 141, 147, 153, 92, 86, 150, 156, 202, 101, 106, 165, 43, 226, 113, 46, 177, 116, 232, 180, 58, 198, 15, 201, 195, 85, 204, 30, 149, 210, 27, 216, 39, 45, 170, 228, 54]

```

A green box highlights the last line of output, and a green arrow points from it to a green-bordered box containing the text "符号执行的结果".

(三) Angr 第二种求解方法:

第二种方法的脚本 solve2.py 的代码如下：

```

python
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)))

```

solve2.py 通过在特定地址处设置钩子来控制程序的行为，符号执行从特定的内存地址开始，并通过 sm.explore 来寻找目标地址 0x080484DB。由于符号执行路径的限制和钩子的作用，最终只会生成一个 u 的值，而不是多个值。solve2.py 的主要步骤如下：

1. 加载二进制文件：同样，使用 angr.Project 加载目标程序。
2. 定义钩子（Hook）：定义了一个名为 hook_demo 的钩子函数。在符号执行过程中，当程序执行到特定地址时，hook_demo 将被触发，强制将 eax 寄存器设置为 0。使用 p.hook 在地址 0x08048485 设置钩子，钩子的作用是修改寄存器的值。
3. 初始化符号执行状态：创建了一个符号执行状态，并将其设置为从地址 0x0804846B 开始。add_options={"SYMBOLIC_WRITE_ADDRESSES"} 仍然启用符号写地址选项。
4. 符号化 u 并存储到内存：和 solve.py 中一样，创建一个符号变量 u，并将其存储到内存地址 0x0804A021。
5. 创建模拟执行管理器并探索路径：创建模拟执行管理器并使用 sm.explore 来探索程序路径，直到找到目标地址 0x080484DB。
6. 求解符号变量 u 的值：从找到的路径中获取符号变量 u 的值。

执行以上 Python 代码，得到如下结果：

The screenshot shows the output of the Python code in an IDLE shell. It includes several warning messages from the angr storage module, such as 'The program is accessing memory with an unspecified value', 'mangr will cope with this by generating an unconstrained symbolic variable and continuing', and 'adding the state option SYMBOL_FILL_UNCONSTRAINED_MEMORY, REGISTERS'. At the bottom, the value '226' is printed, which is highlighted with a green box and labeled '符号执行的结果' (Execution Result).

```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: E:\Python Code\pythonProject\test001.py =====
>>> [33mWARNING[0m | 2025-05-08 23:08:09,325 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mThe program is accessing memory with an unspecified value. This could indicate unwanted behavior.[0m
>>> [33mWARNING[0m | 2025-05-08 23:08:09,340 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mangr will cope with this by generating an unconstrained symbolic variable and continuing. You can resolve this by:[0m
>>> [33mWARNING[0m | 2025-05-08 23:08:09,350 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31m1) setting a value to the initial state[0m
>>> [33mWARNING[0m | 2025-05-08 23:08:09,356 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31m2) adding the state option ZERO_FILL_UNCONSTRAINED_MEMORY, REGISTERS, to make unknown regions hold null[0m
>>> [33mWARNING[0m | 2025-05-08 23:08:09,367 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31m3) adding the state option SYMBOL_FILL_UNCONSTRAINED_MEMORY, REGISTERS, to suppress these messages.[0m
>>> [33mWARNING[0m | 2025-05-08 23:08:09,383 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mFilling memory at 0x7fff0000 with 4 unconstrained bytes referenced from 0x8048472 (main+0x7 in issue (0x8048472))[0m
>>> [33mWARNING[0m | 2025-05-08 23:08:09,397 | [31mangr.storage.memory_mixins.default.filler.mixin[0m | [31mFilling register ebp with 4 unconstrained bytes referenced from 0x8048475 (main+0xa in issue (0x8048475))[0m
226
>>> | 符号执行的结果 | Ln: 13 Col: 0

```

示例 C 语言代码由于默认 `u = 0`, 执行结果为"you lose!", 将以上两种方法得到的结果赋值给 `u` 后再执行就会得到结果"you win!"。

```
char u = 226;

Microsoft Visual Studio 调试 × + ▾

you win!
E:\C++Code\Project2\x64\Release\Project2.exe (进程 53556)已退出,
按任意键关闭此窗口. . .|
```

四、心得体会

两个约束求解器最终都能够找到使程序输出"you win!"的 `u` 值，但是方法有所不同。solve.py 和 solve2.py 的对比如下：

特性	solve.py	solve2.py
目标	寻找多个使程序输出 "you win!" 的 <code>u</code> 值	寻找一个使程序输出 "you win!" 的 <code>u</code> 值
符号执行路径探索	不受限制, 探索多个路径	受钩子限制, 探索较少路径
符号化写地址	启用符号化写地址选项, 符号化 <code>u</code> 并存储在内存地址 <code>0x804a021</code>	启用符号化写地址选项, 符号化 <code>u</code> 并存储在内存地址 <code>0x0804A021</code>
钩子 (Hook)	没有钩子	在特定地址 <code>0x08048485</code> 设置了钩子, 修改寄存器 <code>eax</code>
路径约束	无路径限制, 可以自由探索符号执行路径	符号执行路径受到钩子的限制, 限制了符号执行的路径选择
最终生成的 <code>u</code> 值数量	生成多个 <code>u</code> 值, 最多返回 256 个可能的解	只生成一个 <code>u</code> 值
求解方法	符号执行多路径, 计算 <code>u</code> 的多个可能值	通过钩子和路径探索, 最终找到一个解
符号执行结果输出	通过 <code>solver.eval_upto(u, 256)</code> 获取多个解	通过 <code>st.solver.eval(u)</code> 获取一个解
探索目标	寻找 <code>b'win'</code> 输出, 避免 <code>b'lose'</code> 输出	寻找目标地址 <code>0x080484DB</code>
路径控制方式	通过 <code>sm.explore(find=correct, avoid=wrong)</code> 控制路径探索	通过 <code>sm.explore(find=0x080484DB)</code> 控制路径探索

如何使用 Angr?

1. 加载项目

```
import angr
project = angr.Project('./binary', auto_load_libs=False)
```

2. 创建初始状态

```
state = project.factory.entry_state()
```

或者指定起始地址或符号化输入:

```
state = project.factory.blank_state(addr=0x08048000)
```

3. 定义符号变量

```
import claripy
input_var = claripy.BVS('input', 8 * 20) # 20 字节的符号输入
state.memory.store(0xdeadbeef, input_var)
```

4. 创建 SimulationManager 并探索路径

```
sm = project.factory.simulation_manager(state)
sm.explore(find=lambda s: b"win" in s.posix.dumps(1))
```

5. 提取解

```
found = sm.found[0]
solution = found.solver.eval(input_var, cast_to=bytes)
```

如何解决一些实际问题？

1. 逆向工程自动化

应用场景：分析闭源程序的算法（如加密协议、许可证验证逻辑）。

案例：自动提取恶意软件的关键行为逻辑，或破解 CTF 中的逆向题目。

2. 漏洞挖掘与利用生成

应用场景：发现软件中的漏洞（如栈溢出、UAF），并生成触发漏洞的输入。

案例：自动化挖掘嵌入式设备固件中的漏洞。

3. 路径求解与约束满足

应用场景：自动生成满足特定条件的输入，使程序执行到目标分支。

案例：绕过程序的复杂条件检查。

4. 恶意软件分析

应用场景：分析加壳或混淆的恶意代码，识别其行为。

案例：解密勒索软件的加密密钥生成逻辑。

5. 程序验证与合规性检查

应用场景：验证程序是否满足特定安全属性（如内存安全）。

案例：确保自动驾驶系统的固件不存在未处理的异常分支。

6. CTF 竞赛辅助

应用场景：自动化解决 CTF 中的逆向工程和漏洞利用题目。

案例：通过符号执行直接求解 Flag，无需手动逆向。