

《漏洞利用及渗透测试基础》

姓名：田晋宇 学号：2212039 班级：物联网工程

实验名称

- Angr应用示例实验

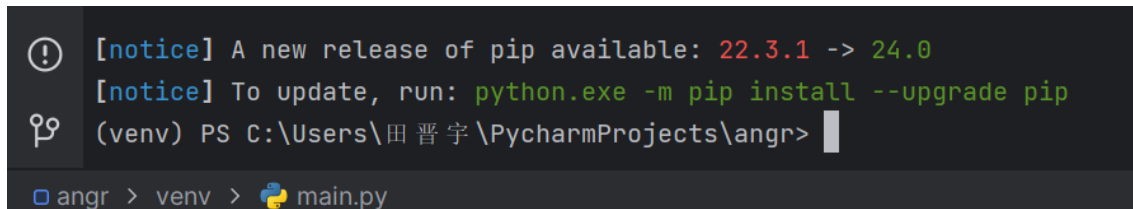
实验要求

- 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

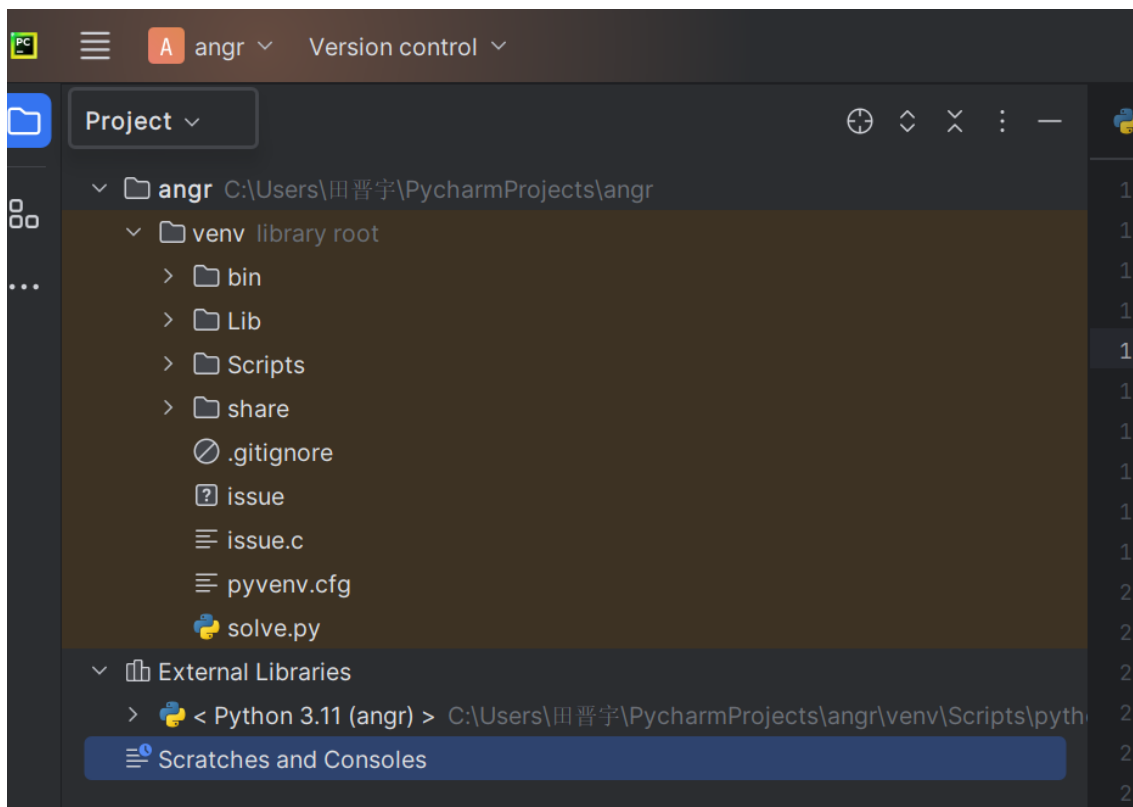
- 配置环境

首先在电脑配置python环境，并用 `pip install angr` 指令安装angr。



```
[notice] A new release of pip available: 22.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Users\田晋宇\PycharmProjects\angr>
```

接着在angr的官方文档中获得实验中所需要的样例。完整路径为 `angr-doc-master.zip\angr-doc-master\examples\sym-write` 并将其导入到pycharm中：



- 复现sym-write的两种方法

1. 求解方法一

issue.c源码：

```

#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;
    }
}

```

solve.py源码:

```

import angr
import claripy

def main():
    p = angr.Project('./issue', load_options={"auto_load_libs": False})
    # By default, all symbolic write indices are concretized.
    state = p.factory.entry_state(add_options={
    angr.options.SYMBOLIC_WRITE_ADDRESSES})
    u = claripy.BVS("u", 8)
    state.memory.store(0x804a021, u)
    sm = p.factory.simulation_manager(state)

    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

    sm.explore(find=correct, avoid=wrong)
    # Alternatively, you can hardcode the addresses.
    # sm.explore(find=0x80484e3, avoid=0x80484f5)
    return sm.found[0].solver.eval_upto(u, 256)

def test():
    good = set()
    for u in range(256):
        bits = [0, 0]
        for i in range(8):
            bits[u & (1 << i) != 0] += 1

```

```

        if bits[0] == bits[1]:
            good.add(u)
        res = main()
        assert set(res) == good

if __name__ == '__main__':
    print(repr(main()))

```

在pycharm中运行，得到如下的结果：

```

C:\Users\田音字\PycharmProjects\angr\venv\Scripts\python.exe C:\Users\田音字\PycharmProjects\angr\venv\solve.py
WARNING | 2024-06-05 14:17:25,986 | angr.storage.memory_mixins.default_filler_mixin | The program is accessing register with an unspecified value. This could indicate unwanted behavior.
WARNING | 2024-06-05 14:17:25,988 | angr.storage.memory_mixins.default_filler_mixin | angr will cope with this by generating an unconstrained symbolic variable and continuing. You can resolve this by specifying the register value.
WARNING | 2024-06-05 14:17:25,988 | angr.storage.memory_mixins.default_filler_mixin | 1) setting a value to the initial state
WARNING | 2024-06-05 14:17:25,988 | angr.storage.memory_mixins.default_filler_mixin | 2) adding the state option ZERO_FILL_UNCONSTRAINED_(MEMORY,REGISTERS), to make unknown regions hold null bytes.
WARNING | 2024-06-05 14:17:25,988 | angr.storage.memory_mixins.default_filler_mixin | 3) adding the state option SYMBOL_FILL_UNCONSTRAINED_(MEMORY,REGISTERS), to suppress these messages.
WARNING | 2024-06-05 14:17:25,988 | angr.storage.memory_mixins.default_filler_mixin | Filling register edi with 4 unconstrained bytes referenced from 0x8048521 (__libc_csu_init+0x1 in issue)
WARNING | 2024-06-05 14:17:25,989 | angr.storage.memory_mixins.default_filler_mixin | Filling register ebx with 4 unconstrained bytes referenced from 0x8048523 (__libc_csu_init+0x3 in issue)
[51, 57, 240, 60, 75, 139, 78, 197, 23, 142, 90, 29, 209, 154, 212, 99, 163, 102, 108, 166, 172, 105, 169, 114, 53, 225, 120, 184, 178, 71, 135, 77, 83, 202, 89, 147, 153, 86, 92, 150, 156, 106, 101, 141, 165, 43, 46, 232, 226, 177, 116, 113, 180, 58, 198, 195, 15, 201, 85, 204, 30, 210, 149, 27, 216, 39, 45, 170, 228, 54]
Process finished with exit code 0

```

```

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

```

对解法一的关键步骤分析：

1. 创建 Angr 项目：

- 我们首先创建一个 Angr 项目，并载入目标二进制文件。在初始化时，我们将 `auto_load_libs` 设置为 `False`，以避免自动加载依赖库。默认情况下，这个选项是 `False`。设置为 `True` 时，Angr 会尝试执行库函数，但这可能会在符号执行过程中带来不必要的复杂性。

2. 初始化模拟状态：

- 我们使用 `entry_state()` 函数初始化一个 `SimState` 对象 `state`。这个对象模拟了程序运行时的内存、寄存器、文件系统数据、符号信息等动态变化的数据。此外，也可以使用 `blank_state()` 函数来初始化 `state`，并通过参数 `addr` 指定程序的起始运行地址。

3. 符号化变量：

- 将需要求解的变量进行符号化。特别地，这里的符号化变量会被存储到二进制文件的特定存储区中。

4. 创建模拟管理器：

- 我们创建一个模拟管理器（Simulation Manager），用于管理程序执行。初始化的 `state` 通过符号执行可以生成一系列的 `states`，模拟管理器的作用就是对这些 `states` 进行管理。

5. 符号执行与状态判定：

- 通过符号执行获取所需的状态。在示例程序中，我们定义了期望的状态，即在符号执行后，程序输出的字符串中包含 `"win"` 而不包含 `"lose"`。这个状态是通过两个函数来定义的，函数

检查符号执行的输出 `state.posix.dumps(1)` 中是否包含 `"win"` 或 `"lose"` 字符串。

6. 求解符号变量:

- 获得期望状态后，通过求解器 (solver) 计算符号变量 `u` 的具体值。

在代码中，语句 `state.memory.store(0x804a021, u)` 是将符号变量 `u` 存储到指定的内存地址。这个地址实际上是二进制文件中 `.bss` 段中变量 `u` 的地址。我们可以使用 IDA Pro 等工具来查找源代码中变量 `u` 在二进制文件中的地址。在示例中，`u` 在 `.bss` 段中的地址是 `0x804a021`。

找到这个地址后，我们可以对整个二进制文件进行符号执行，以找出符号变量 `u` 的具体值。换句话说，我们通过符号执行模拟程序运行，进而求解符号变量在程序特定状态下的值。

2. 求解方法二

```
#!/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

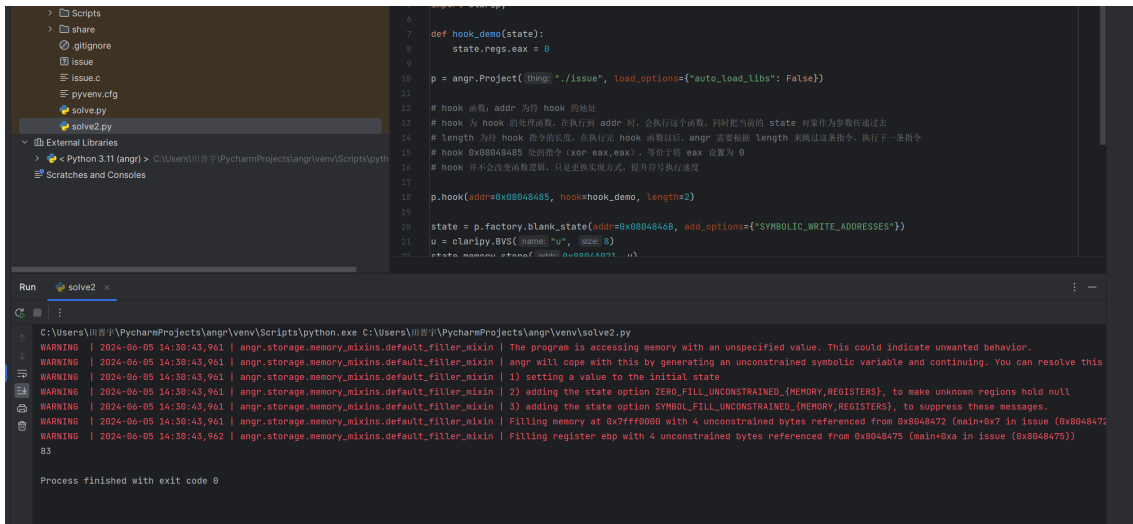
p.hook(addr=0x08048485, hook=hook_demo, length=2)

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)))
```

在pycharm中运行该代码，运行结果如图所示：



第二种解法与第一种解法的区别：

1. 使用 Hook 函数：

- 在新代码中，我们使用了一个 Hook 函数，将地址 `0x08048485` 处的长度为 2 的指令替换为自定义的 `hook_demo` 函数。尽管原始指令 `xor eax, eax` 和 `hook_demo` 函数中的 `state.regs.eax = 0` 的效果是相同的，但这样做可以演示如何通过 Hook 来替换一些复杂的系统函数调用（如 `printf`），从而提升符号执行的性能。

2. 符号执行目标改变：

- 符号执行的目标地址变更为 `0x080484DB`。由于源程序中的 `win` 和 `lose` 是互斥的，因此我们只需要给定一个 `find` 条件，即可找到正确的状态。

3. 求解符号变量的改动：

- 最后，使用 `eval(u)` 替代了原来的 `eval_upto`，这将打印出一个具体的结果，而不是一系列可能的值。

Angr解决实际问题的探究

Angr 可以用于逆向工程和分析恶意软件，以了解其行为、提取加密密钥、分析恶意逻辑等。Angr 通过其强大的符号执行、模拟和分析功能，可以解决安全研究、恶意软件分析、自动化测试等多个实际问题。通过上述步骤，用户可以利用 Angr 自动发现漏洞、分析恶意软件行为、生成测试用例，从而提升工作效率和分析深度。

Angr在解决实际问题上主要有如下应用：

- 自动化漏洞发现：**检测软件中的缓冲区溢出等安全漏洞，通过符号执行，自动生成可能触发漏洞的输入数据，减少手动分析和漏洞挖掘的时间，提高漏洞检测的效率和准确性。
- 恶意软件分析：**逆向工程和分析恶意软件，以理解其行为和提取关键信息。自动分析恶意软件的逻辑，提取例如加密密钥等关键数据，识别恶意行为路径，提升恶意软件分析的速度和深度。
- 自动化测试：**生成广泛覆盖程序不同路径的测试用例，进行功能和安全性测试。通过符号执行生成各种输入数据，覆盖尽可能多的程序路径，发现潜在的边缘情况和错误，提高软件的稳定性和安全性。
- 自动化逆向工程：**分析闭源软件以理解其内部工作机制和算法。自动探索程序的执行路径，解析程序的逻辑和功能，减少手动逆向工程的复杂性和工作量。
- 密码学算法破解：**破解简单的加密算法，提取密钥或解密数据。通过符号执行，分析加密和解密逻辑，自动化地找到密钥或解密方法，帮助理解和破解简单的加密方案。

心得体会

1. 符号执行的强大功能

通过本次实验，我深刻体会到了符号执行在程序分析中的强大功能。Angr 框架提供了一个灵活且高效的符号执行平台，使得自动化漏洞发现、恶意软件分析和逆向工程等任务变得更加便捷。符号执行能够模拟程序的多种执行路径，自动生成测试用例和漏洞触发条件，大大减少了手动分析的工作量，提高了效率和准确性。

2. 实践中遇到的挑战

在实验过程中，我遇到了一些挑战。例如，理解和使用 Angr 的各种功能选项需要一定的学习曲线，特别是在设置符号化变量和路径探索策略时。此外，处理复杂的系统调用和外部库函数也需要进行适当的 Hook，以确保符号执行的效率和准确性。这些挑战促使我深入学习相关技术，提升了我的问题解决能力。

3. 工具与手动分析的结合

虽然 Angr 强大且高效，但在某些情况下，结合手动分析仍然是必要的。通过本次实验，我认识到，工具和手动分析各有优势，合理结合使用能够达到最佳效果。例如，在分析复杂的二进制文件时，手动分析可以帮助理解程序逻辑，而 Angr 可以自动探索更多的执行路径和生成测试用例。

4. 提升了对程序安全性的认识

实验过程中，通过自动化漏洞发现和恶意软件分析等实际应用，我更加深刻地认识到程序安全性的重要性。符号执行不仅能帮助发现潜在漏洞，还能提供有效的安全测试方法，为软件开发过程中的安全性保障提供了有力支持。

5. 实验应用的广泛性

通过多个实际案例的学习和实践，我了解到符号执行和 Angr 框架在多个领域的广泛应用，包括漏洞挖掘、恶意软件分析、自动化测试、逆向工程和密码学算法破解等。掌握这些技术为我今后的研究和工作开辟了更多的可能性。