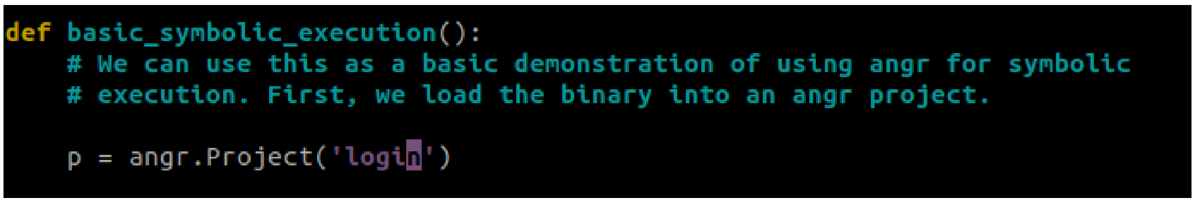
Lab 3: Symbolic Execution for Backdoor Discovery

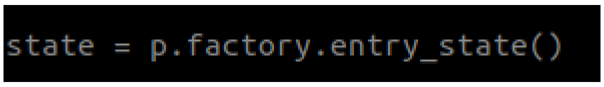
First, we use the example *fauxware* to understand angr's usage to analyze the firmware's backdoor.

We see that in solve.py, we initialize a new angr project object p for binary *fauxware*, then construct it's SimState using constructor entry_state(), create a SimulationManager for that state, then begin execution until we reach a branch state where both branches are satisfiable, get the constraints from sm.active[n].solver.constraints which are passed to constraint solver z3 to produce a set of concrete inputs satisfying them. One of those inputs is the password 'SOSNEAKY'.

Similarly, for our binary *login*, we do the same - create a SimState for the program's entry point using the entry_state constructor - as we know that the the backdoor constraint check is located near the beginning of the program.

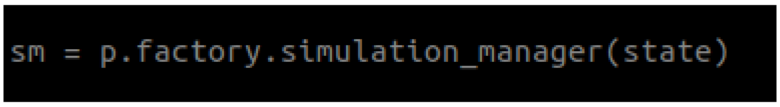


```
def basic_symbolic_execution():
    # We can use this as a basic demonstration of using angr for symbolic
    # execution. First, we load the binary into an angr project.

    p = angr.Project('login')
```



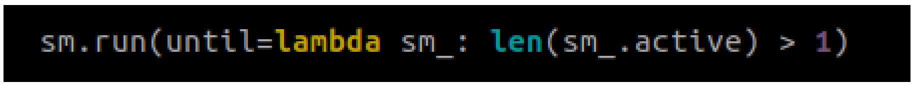
```
state = p.factory.entry_state()
```

Then, we create a SimulationManager object of that state.



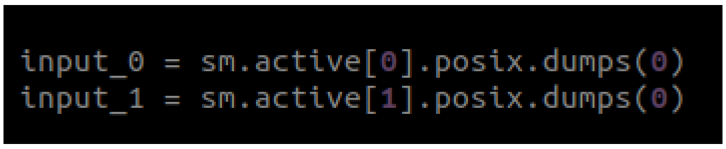
```
sm = p.factory.simulation_manager(state)
```

After which, we begin execution (symbolically execute the program until we reach a branch statement for which both branches are satisfiable).



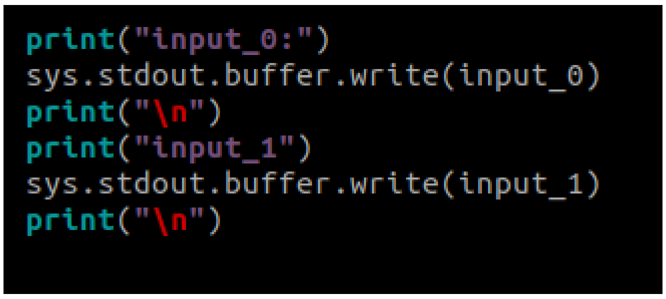
```
sm.run(until=lambda sm_: len(sm_.active) > 1)
```

Then we construct concrete inputs satisfying constraints:



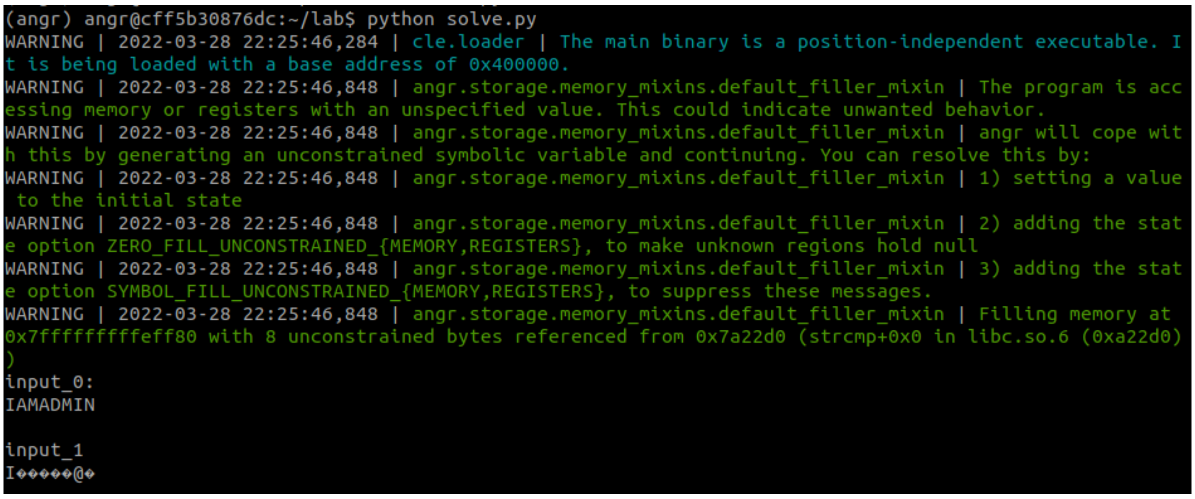
```
input_0 = sm.active[0].posix.dumps(0)
input_1 = sm.active[1].posix.dumps(0)
```

Now, here we differ from *fauxware*. In *fauxware*, we knew the password was 'SOSNEAKY' so we simply check input_0 and input_1 to check if 'SOSNEAKY' is part of those inputs. For login, we don't know the password - so we will print both inputs out to see if there's something resembling a password in there.



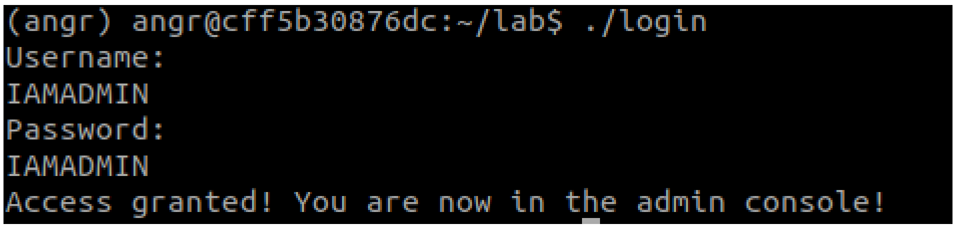
```
print("input_0:")
sys.stdout.buffer.write(input_0)
print("\n")
print("input_1")
sys.stdout.buffer.write(input_1)
print("\n")
```

We get the output:



```
(angr) angr@cff5b30876dc:~/lab$ python solve.py
WARNING | 2022-03-28 22:25:46,284 | cle.loader | The main binary is a position-independent executable. I
t is being loaded with a base address of 0x400000.
WARNING | 2022-03-28 22:25:46,848 | angr.storage.memory_mixins.default_filler_mixin | The program is acc
essing memory or registers with an unspecified value. This could indicate unwanted behavior.
WARNING | 2022-03-28 22:25:46,848 | angr.storage.memory_mixins.default_filler_mixin | angr will cope wit
h this by generating an unconstrained symbolic variable and continuing. You can resolve this by:
WARNING | 2022-03-28 22:25:46,848 | angr.storage.memory_mixins.default_filler_mixin | 1) setting a value
 to the initial state
WARNING | 2022-03-28 22:25:46,848 | angr.storage.memory_mixins.default_filler_mixin | 2) adding the stat
e option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions hold null
WARNING | 2022-03-28 22:25:46,848 | angr.storage.memory_mixins.default_filler_mixin | 3) adding the stat
e option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messages.
WARNING | 2022-03-28 22:25:46,848 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7fffffffffeff80 with 8 unconstrained bytes referenced from 0x7a22d0 (strcmp+0x0 in libc.so.6 (0xa22d0)
)
input_0:
IAMADMIN

input_1
I������@�
```

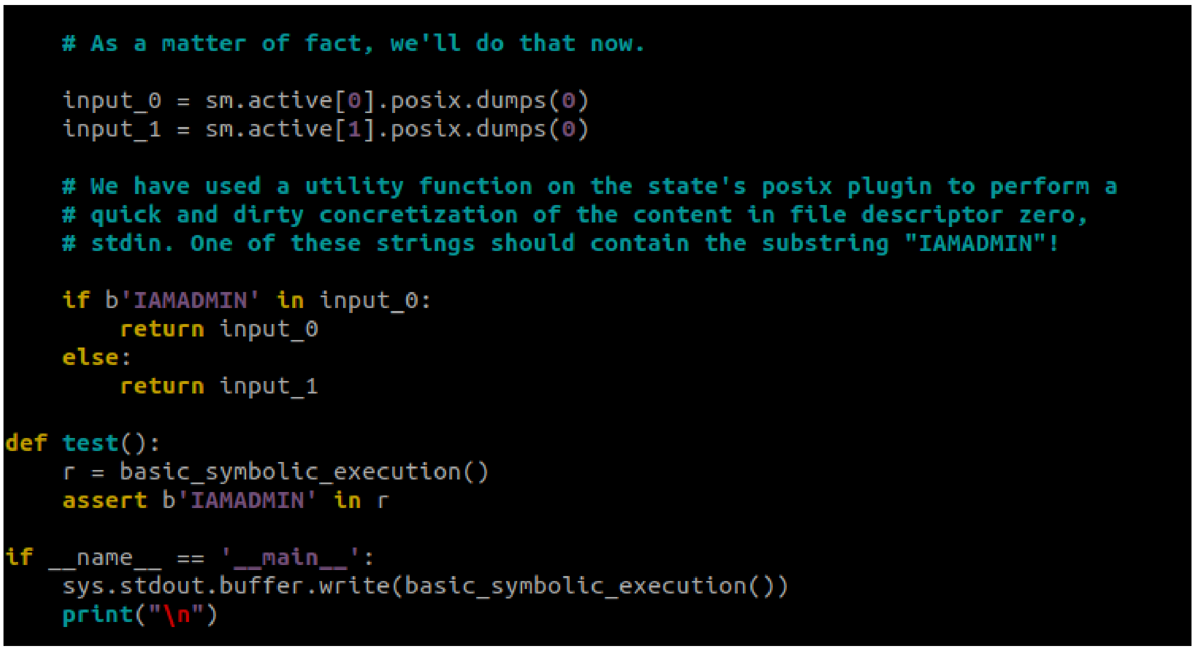
We see that input_0 equals "IAMADMIN". Let us try using this password with the binary ***login***. For that, first we need to give executable permissions to the binary using *chmod +x*.



```
(angr) angr@cff5b30876dc:~/lab$ ./login
Username:
IAMADMIN
Password:
IAMADMIN
Access granted! You are now in the admin console!
```

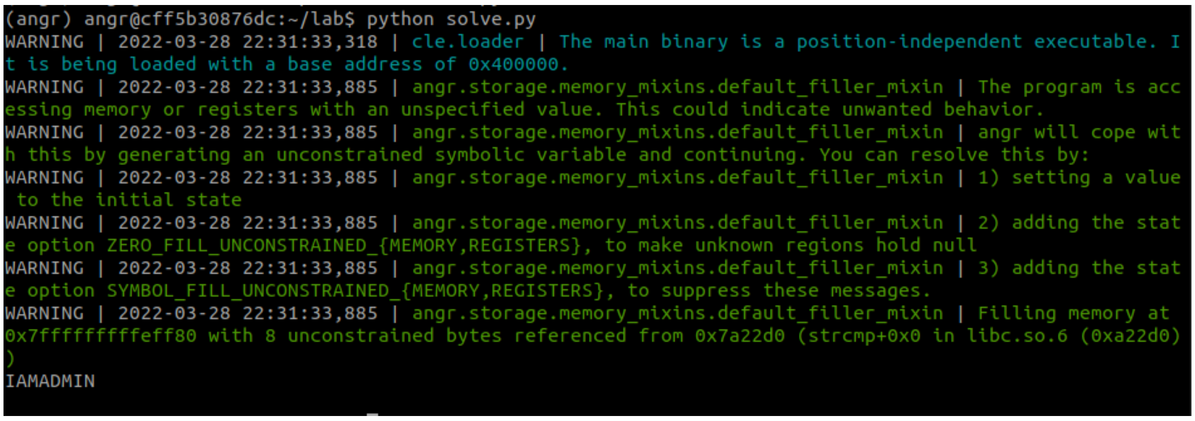
We see that it works.

Then, we edit solve.py so as to print one of the strings input_0 or input_1 such that one of them contains "IAMADMIN".



```
    # As a matter of fact, we'll do that now.

    input_0 = sm.active[0].posix.dumps(0)
    input_1 = sm.active[1].posix.dumps(0)

    # We have used a utility function on the state's posix plugin to perform a
    # quick and dirty concretization of the content in file descriptor zero,
    # stdin. One of these strings should contain the substring "IAMADMIN"!

    if b'IAMADMIN' in input_0:
        return input_0
    else:
        return input_1

def test():
    r = basic_symbolic_execution()
    assert b'IAMADMIN' in r

if __name__ == '__main__':
    sys.stdout.buffer.write(basic_symbolic_execution())
    print("\n")
```

Thus, we get the output:



```
(angr) angr@cff5b30876dc:~/lab$ python solve.py
WARNING | 2022-03-28 22:31:33,318 | cle.loader | The main binary is a position-independent executable. I
t is being loaded with a base address of 0x400000.
WARNING | 2022-03-28 22:31:33,885 | angr.storage.memory_mixins.default_filler_mixin | The program is acc
essing memory or registers with an unspecified value. This could indicate unwanted behavior.
WARNING | 2022-03-28 22:31:33,885 | angr.storage.memory_mixins.default_filler_mixin | angr will cope wit
h this by generating an unconstrained symbolic variable and continuing. You can resolve this by:
WARNING | 2022-03-28 22:31:33,885 | angr.storage.memory_mixins.default_filler_mixin | 1) setting a value
 to the initial state
WARNING | 2022-03-28 22:31:33,885 | angr.storage.memory_mixins.default_filler_mixin | 2) adding the stat
e option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions hold null
WARNING | 2022-03-28 22:31:33,885 | angr.storage.memory_mixins.default_filler_mixin | 3) adding the stat
e option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messages.
WARNING | 2022-03-28 22:31:33,885 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at
0x7ffffffffffeff80 with 8 unconstrained bytes referenced from 0x7a22d0 (strcmp+0x0 in libc.so.6 (0xa22d0)
)
IAMADMIN
```

Thus, "IAMADMIN" is the hardcoded password.

End