Symbolic Execution for Backdoor Discovery

Introduction

Symbolic execution is a powerful software analysis technique for exploring a program's paths. It treats program inputs as symbolic values rather than concrete values, allowing the exploration of program behavior across a wide range of inputs. This lab uses symbolic execution, with the aid of the **angr** framework, to analyze binary code to discover hidden backdoors. A practical application is demonstrated by analyzing the provided "login" binary executable to identify a backdoored password.

Methodology

Environment Setup

- Installed angr in a python environment.
- Acquiring the binary executable with a suspected backdoor and the corresponding username file.

Symbolic Execution Setup

- **Loading the Binary:** Loaded the binary into angr's project structure, which allows for manipulating and analyzing the binary's execution paths.
- **Input Configuration:** Since the objective is to find the backdoor password and the username has no impact, a fixed value is used for the username, and a symbolic variable is created for the password. This approach focuses the symbolic execution on discovering the value that leads to a successful authentication bypass.
- **State Initialization:** Initialized the symbolic execution state with the fixed username and symbolic password as the input, mimicking the program's expectation to receive these values from stdin.

```
#Load the binary file
p = angr.Project('login', auto_load_libs=False)

#Create a symbolic bitvector for the username(8 characters) and the password
fixed_username = b'useruser\n'   #username doesn't matter here, but it should be 8 characters
password = claripy.BVS('password', 8*8)

#Concatanate username and password with a newline
input_data = claripy.Concat(claripy.BVV(fixed_username), password, claripy.BVV(b'\n'))
```

Exploration Strategy

- **Path Exploration:** Used angr's simulation manager to explore the binary's execution paths. The goal was to identify the path that leads to the output message "Access granted! You are now in the admin console!" indicating a successful authentication bypass via the backdoor.
- **Constraint Application:** As the simulation manager explores different paths, it applies constraints based on the execution flow. For paths that do not lead to the desired outcome, constraints are gathered to guide the symbolic execution towards the goal.

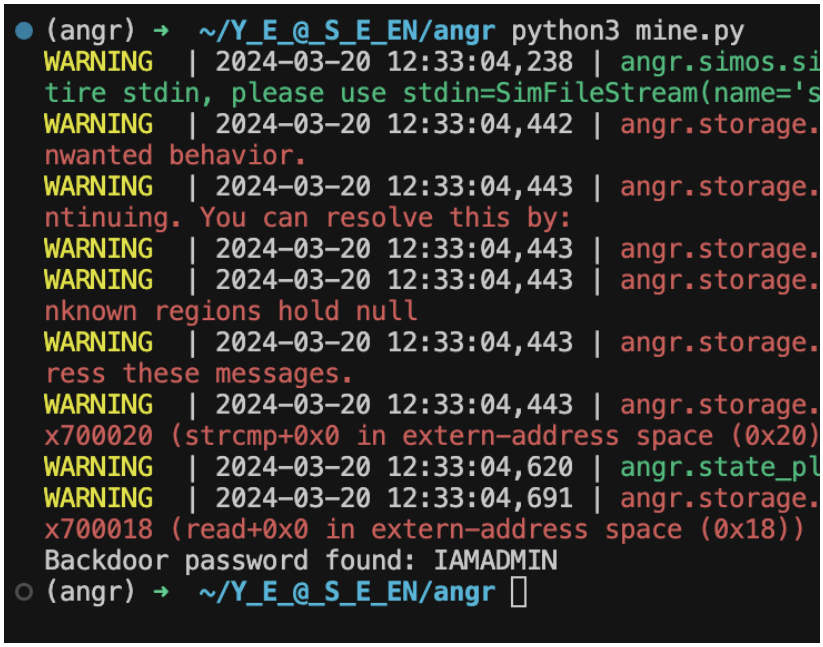
```
state = p.factory.entry_state(stdin=input_data)
simgr = p.factory.simulation_manager(state)
simgr.explore(find=lambda s: b"Access granted! You are now in the admin console!" in s.posix.dumps(1))

if simgr.found:
  found_state = simgr.found[0]

  solution = found_state.solver.eval(password, cast_to=bytes)
  print(f"Backdoor password found: {solution.decode()}")
else:
  print("Backdoor password not found")
```

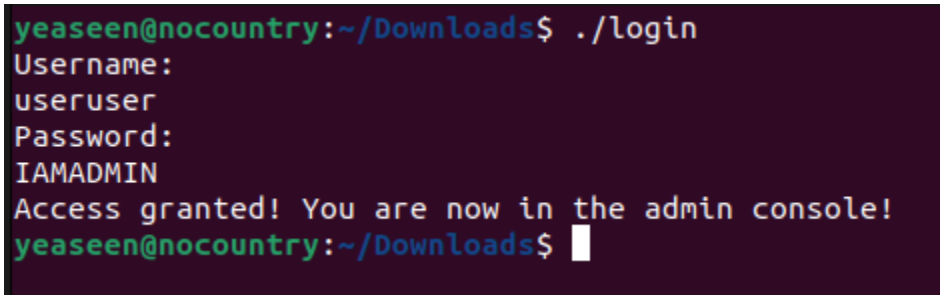
Identifying the Backdoor Password

- **Success Condition Identification:** Among the explored paths, I identified the one(s) that led to the success message. This indicates that the symbolic execution has successfully found a path through the backdoor.
- **Constraint Solving for Password:** Utilized angr's constraint solver to solve the symbolic password variable against the constraints gathered from the path leading to successful authentication. The solution to these constraints is the backdoor password.



Verification and Testing

Backdoor Password Verification: I manually verified the discovered password by running the binary with the found password and a random username. The expected outcome is the success message, confirming the effectiveness of the symbolic execution in discovering the backdoor password.



Conclusion

Utilizing symbolic execution through the **angr** framework efficiently uncovered a backdoor password in a binary executable, demonstrating its effectiveness in revealing hidden vulnerabilities. This approach highlights the importance of advanced security analysis techniques in identifying and mitigating potential threats in software, underscoring the necessity for secure coding practices to safeguard against unauthorized access and ensure application integrity.