SPQR

Angr & Pwn

vulnerability detection using symbolic execution

by malweisse

About me

Andrea Fioraldi [@andreafioraldi, @malweisse on IRC]

22 years old, student of Engineering in Computer Science.

Interested in binary analysis (symbolic execution, reversing and other pretty stuffs) and binary exploitation (do u know what is double free?).

Capturing flags with TheRomanXpl0it and mHACKeroni.

Not so much skilled trumpeter, mountain bike and trekking lover, Dragon Ball fanboy, homebrewer.

- **Symbolic execution**
- Angr crash course
- Angr and the bugs

Concrete execution

- Single flow of instructions (path) at time
- Can’t evaluate program behaviour

```c
int foo(int a, int b)
{
    int c = 77;

    if(a + b == 42) {
        c = c - b;
    }
    else {
        c = a - c;
    }
    if(c == 38) {
        puts("Well done.");
    }
    else {
        puts("Try again.");
    }
}
```

Symbolic execution

- Explore all possible paths in a program
- Evaluate how inputs affect the choice of a path
- Use ***symbolic*** values as inputs
- Logical expression in function of the symbolic inputs (path constraints and symbolic storage)
- Execution forked on branches
- SMT solver to evaluate that expressions

```c
int foo(int a, int b)
{
	int c = 77;

	if(a + b == 42) {
		c = c - b;
	}
	else {
		c = a - c;
	}
	if(c == 38) {
		puts("Well done.");
	}
	else {
		puts("Try again.");
	}
}
```

Symbolic execution

- [Path #1] Formula: $X + Y = 42$,
  Storage: $c = 77$

- [Path #2] Formula: $\neg(X + Y = 42)$,
  Storage: $c = 77$

Target: line 12

```
1  int foo(int a, int b)
2  {
3      int c = 77;
4
5      if(a + b == 42) {
6          c = c - b;
7      }
8      else {
9          c = a - c;
10     }
11     if(c == 38) {
12         puts("Well done.");
13     }
14     else {
15         puts("Try again.");
16     }
17 }
```

Symbolic execution

- [Path #1.1] Formula: X + Y = 42 ∧ c = 38, Storage: c = 77 − Y
- [Path #1.2] Formula: X + Y = 42 ∧ c != 38, Storage: c = 77 − Y
- [Path #2.1] Formula: ¬(X + Y = 42) ∧ c = 38, Storage: c = X − 77
- [Path #2.2] Formula: ¬(X + Y = 42) ∧ c != 38, Storage: c = X − 77

The interesting paths are 1.1 and 2.1 because they reached our target

```
int foo(int a, int b)
{
    int c = 77;

    if(a + b == 42) {
        c = c - b;
    }
    else {
        c = a - c;
    }
    if(c == 38) {
        puts("Well done.");
    }
    else {
        puts("Try again.");
    }
}
```

Symbolic execution

- [Path #1.1] Formula: $X + Y = 42 \wedge c = 38$, Storage: $c = 77 - Y$

  solve($X + Y = 42 \wedge 77 - Y = 38$)

  $X = 3, Y = 39$

```
int foo(int a, int b)
{
    int c = 77;

    if(a + b == 42) {
        c = c - b;
    }
    else {
        c = a - c;
    }
    if(c == 38) {
        puts("Well done.");
    }
    else {
        puts("Try again.");
    }
}
```

Symbolic execution (what about?)

- Symbolic pointers dereference?
- Loops?
- Exponential increase of the number of paths?
- Environment interaction?
- Non-linear constraints?

Symbolic execution

Angr crash course

Angr and the bugs

Angr WTF

Angr is a binary analysis framework.
Binary loader, emulator, symbolic executor.

http://angr.io/

```
python3 -m pip install angr
```

Angr modules

- ***CLE***, a multi-format binary loader with an intuitive API;

- ***archinfo***, a collection of classes that contain architecture-specific information;

- ***PyVEX***, a wrapper around Valgrind’s VEX IR lifter, used to make the analyses architecture-agnostic;

- ***Claripy***, the angr data backend, a wrapper around the Z3 solver and an interface to abstract concrete and symbolic values handling;

Loading a binary

```python
import angr
# load the example
project = angr.Project("./foo")
```

Simulation

```
# start a new SimulationManager
simgr = project.factory.simulation_manager()
# step
simgr.step()
# step until it branches
simgr.run(until=lambda sm: len(sm.active) != 1)
# check the states that are still active
print (simgr.active)
```

States

```python
state = simgr.active[0]

# a state has plugins, representing registers, memory, etc
print (state.regs.rax)
print (state.memory.load(state.regs.rsp, 8))
# one of the plugins represents the system state
print (state.posix.fd)
# files are backed by a memory region
print (state.posix.fd[0].read_data(8)) #return (value, size)
```

Solver

```
state = simgr.active[0]

# SMT solver
print (state.solver)
addr = state.regs.rsp + 0x100
# each value in angr is represented as an expression tree
v = state.memory.load(addr, 8) + 0x10
print (v)
print (v.op) # __add__
print (v.args) # (other bitvector, 0x10)
# add state constraints
state.add_constraints(v < 0x2a)
state.add_constraints(v > 0x1a)
# concretize a value
print (hex(state.solver.eval(v)))
```

Simulation 2

```python
# create a state starting on foo()
initial_state = project.factory.blank_state()
initial_state.regs.rip = 0x4005c7 # foo address
# create symbolic args
a = initial_state.solver.BVS("sym_a", 64)
b = initial_state.solver.BVS("sym_b", 64)
initial_state.regs.rdi = a
initial_state.regs.rsi = b
# start a new SimulationManager based on initial_state
simgr = project.factory.simulation_manager(initial_state)
# explore using conditions
simgr.explore(find=0x400600, avoid=0x40060e) # well done, try again
print (simgr, simgr.found) # found stash
# conretize found inputs
print (a.args[0], "=", simgr.found[0].solver.eval(a))
print (b.args[0], "=", simgr.found[0].solver.eval(b))
```

Default Stashes

- ***active***, states that are "live"
- ***errored***, states that errored out (actual angr bugs)
- ***found***, states that reached the "find" condition
- ***avoided***, states that hit the "avoid" condition
- ***deadended***, states that produced no successors
- ***unconstrained***, states that jump to unconstrained symbolic values

Symbolic execution

Angr crash course

Angr and the bugs

Stack Buffer Overflow

What is the purpose of a stack buffer overflow?

Control the program counter value overwriting the return address on the stack

Stack Buffer Overflow

DEMO

Thank you!

QUESTIONS?

References

USEFUL PAPERS:

- R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," ACM Comput. Surv., vol. 51, no. 3, 2018
- Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in IEEE Symposium on Security and Privacy, 2016.

BRIEF SUMMARY (the chapter 1 of my thesis):

- https://www.researchgate.net/publication/327655380_Symbolic_Execution_and_Debugging_Synchronization

MORE ABOUT ANGR

- https://docs.angr.io/
- https://www.blackhat.com/docs/us-15/materials/us-15-Kruegel-Using-Static-Binary-Analysis-To-Find-Vulnerabilities-And-Backdoors-In-Firmware.pdf