CS 395: Binary Exploitation in Linux

Week 9: Z3 and Angr

Installing

- Sudo apt-get update
- pip3 install z3-solver
- pip3 install ipython

We'll install Angr in a bit

Problem: Reverse Engineering is Hard!

- Sometimes you need to reverse engineer some math-heavy code that can get very confusing very fast.
- This includes things like hashing algorithms, CD key generators for video games, etc.:
- Reversing algorithms by hand is slow and difficult, even stepping through with GDB doesn’t help much.
- “*Given the output of this sequence of operations, what must the input have been?*”

Solution: Symbolic Execution and Static Analysis!

- Books on the stuff!
- New and powerful technology enables this technique!
- We're using the Z3 Theorem Prover with Python bindings
- Uses computer magic and super-math to make reverse engineering easier when you have control over a programs input, and want to reach a specific output

Symbolic Execution- a means of analyzing a program to determine what inputs cause what outputs. An interpreter follows the program, taking symbolic inputs rather than actual inputs from a program.

Static Analysis- a method of analyzing a program without executing it. Done by code reading, or in our case, using symbolic execution with Z3

Z3 Demo

- The program takes our input, and manipulates it mathematically and expects a certain output…
- Looks like a system of equations to me!
- Sounds like a job for the Z3 theorem prover!

Decompile: main - (crackme)

```
1
2  undefined8 main(int param_1,long param_2)
3
4  {
5    uint uVar1;
6    uint uVar2;
7    uint uVar3;
8    uint uVar4;
9    uint uVar5;
10
11   puts("I will take 4 numbers less than 20 from you and apply my ~secret hashing~ to them");
12   puts("Give me the input that will produce: 96 1 18 27");
13   if (param_1 != 5) {
14     puts("Invalid input, give me 4 numbers from the command line");
15                     /* WARNING: Subroutine does not return */
16     exit(1);
17   }
18   uVar2 = atoi(*(char **)(param_2 + 8));
19   uVar3 = atoi(*(char **)(param_2 + 0x10));
20   uVar4 = atoi(*(char **)(param_2 + 0x18));
21   uVar5 = atoi(*(char **)(param_2 + 0x20));
22   printf("given numbers: %d %d %d %d\n",(ulong)uVar2,(ulong)uVar3,(ulong)uVar4,(ulong)uVar5);
23   uVar1 = uVar2 * uVar4 * 3;
24   uVar3 = (int)uVar3 % (int)uVar5 - 1;
25   uVar4 = uVar4 + uVar5 * 2;
26   uVar2 = uVar2 * uVar2 * uVar2 + uVar5 * uVar5;
27   if ((((uVar1 == 0x60) && (uVar3 == 1)) && (uVar4 == 0x12)) && (uVar2 == 0x59)) {
28     puts("Congratulations! You win!");
29   }
30   else {
31     puts("Wrong! Try again nerd!");
32   }
33   printf("output numbers: %d %d %d %d\n",(ulong)uVar1,(ulong)uVar3,(ulong)uVar4,(ulong)uVar2);
34   return 0;
35 }
36
```

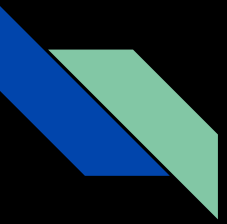
Z3 Solution

- We know the program takes 4 integers as input
- We know they each must be < 20
- We know that after some math, they are expected to be a certain value

Run the script and it tells us the correct input without ever running the binary!

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week9]
└─$ python3 solver.py
sat
[b = 2,
 a = 4,
 c = 8,
 d = 5,
 div0 = [else → 0],
 mod0 = [else → 2]]
```

```
2  from z3 import *
3  one = Int('a')
4  two = Int('b')
5  three = Int('c')
6  four = Int('d')
7  s = Solver()
8
9  #add x < 20 constraint
10 s.add(one < 20)
11 s.add(two < 20)
12 s.add(three < 20)
13 s.add(four < 20)
14
15 #copy programs hashing math
16 s.add(3*(one*three) == 96)
17 s.add((two%four)-1 == 1)
18 s.add(three + 2*four == 18)
19 s.add(four*four + one*one*one == 89)
20
21 #show answer
22 print(s.check())
23 print(s.model())
```



Problem: I want to Analyze a Binary with Unforgiving Intensity

- Z3 is too hard/limited
- We want to write more powerful exploit scripts
- I wanna use bleeding-edge tech!
- What do all the researchers use?

Solution: Angr, the Holy Grail of Binary Analysis

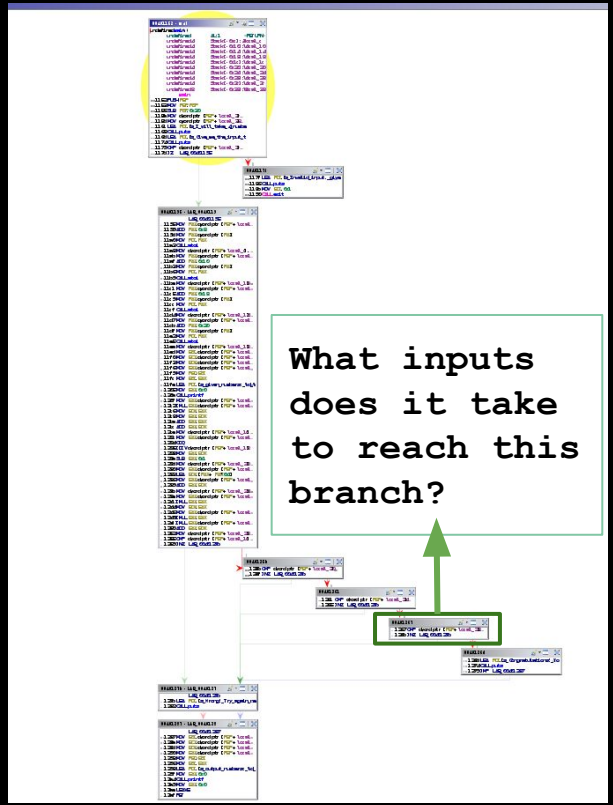
- Uses Z3 under the hood to do symbolic execution
- Feature list miles long
- Check out the API
- Check out their website
- Utilizes dark magic to do crazy stuff

Download in a virtual environment!

Dependencies clash with Z3.

What Makes Angr Different?

- Angr lets us **emulate** a binary and symbolically execute it, a technique called Concolic testing
- We can give Angr the address of our “win condition,” and have it emulate what inputs it would take to reach it.
- Unlimited power



Installing Angr: Virtualenv

- I made a nifty script that will help install Angr: “./angr_install.sh”
- Z3 and Angr have conflicting dependencies! But we want to use both.
- To get around this we use virtualenv to create a virtual environment specially for Angr. This environment has a clean Python install, so we can install Angr in a place that doesn’t conflict with our main environment.

```
┌──(cs395㉿kali)-[~/Desktop/CS395]
└─$ mkvirtualenv angr_env
created virtual environment CPython3.9.1.final.0-64 in 619ms
  creator CPython3Posix(dest=/home/cs395/.virtualenvs/angr_env, clear=False, no_vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=/home/cs395/.local/share/virtualenv)
    added seed packages: pip==21.0.1, setuptools==52.0.0, wheel==0.36.2
  activators BashActivator,CShellActivator,FishActivator,PowerShellActivator,PythonActivator,XonshActivator
virtualenvwrapper.user_scripts creating /home/cs395/.virtualenvs/angr_env/bin/predeactivate
virtualenvwrapper.user_scripts creating /home/cs395/.virtualenvs/angr_env/bin/postdeactivate
virtualenvwrapper.user_scripts creating /home/cs395/.virtualenvs/angr_env/bin/preactivate
virtualenvwrapper.user_scripts creating /home/cs395/.virtualenvs/angr_env/bin/postactivate
virtualenvwrapper.user_scripts creating /home/cs395/.virtualenvs/angr_env/bin/get_env_details

(angr_env) ┌──(cs395㉿kali)-[~/Desktop/CS395]
└─$ deactivate
```

Angr Demo

- The program takes our input, and manipulates it mathematically and expects a certain output…
- Looks like a system of equations to me!
- Sounds like a job for ~~the Z3 theorem prover!~~ Angr!

Decompile: main - (crackme)

```
1
2  undefined8 main(int param_1,long param_2)
3
4  {
5    uint uVar1;
6    uint uVar2;
7    uint uVar3;
8    uint uVar4;
9    uint uVar5;
10
11   puts("I will take 4 numbers less than 20 from you and apply my ~secret hashing~ to them");
12   puts("Give me the input that will produce: 96 1 18 27");
13   if (param_1 != 5) {
14     puts("Invalid input, give me 4 numbers from the command line");
15                     /* WARNING: Subroutine does not return */
16     exit(1);
17   }
18   uVar2 = atoi(*(char **)(param_2 + 8));
19   uVar3 = atoi(*(char **)(param_2 + 0x10));
20   uVar4 = atoi(*(char **)(param_2 + 0x18));
21   uVar5 = atoi(*(char **)(param_2 + 0x20));
22   printf("given numbers: %d %d %d %d\n",(ulong)uVar2,(ulong)uVar3,(ulong)uVar4,(ulong)uVar5);
23   uVar1 = uVar2 * uVar4 * 3;
24   uVar3 = (int)uVar3 % (int)uVar5 - 1;
25   uVar4 = uVar4 + uVar5 * 2;
26   uVar2 = uVar2 * uVar2 * uVar2 + uVar5 * uVar5;
27   if ((((uVar1 == 0x60) && (uVar3 == 1)) && (uVar4 == 0x12)) && (uVar2 == 0x59)) {
28     puts("Congratulations! You win!");
29   }
30   else {
31     puts("Wrong! Try again nerd!");
32   }
33   printf("output numbers: %d %d %d %d\n",(ulong)uVar1,(ulong)uVar3,(ulong)uVar4,(ulong)uVar2);
34   return 0;
35 }
36
```

Angr Solution

- Create a Project, turn off auto_load_libs because it's unnecessary 90% of the time
- BVS = Bit Vector Symbol, think of these like the variables from 7th grade algebra, these symbols are the values angr is trying to "solve for."

```
1 import angr
2 from claripy import *
3 import IPython
4
5 proj = angr.Project("crackme",load_options={'auto_load_libs':False})
6 num1 = BVS("num1",16)
7 num2 = BVS("num2",16)
8 num3 = BVS("num3",16)
9 num4 = BVS("num4",16)
10 state = proj.factory.entry_state(argc=5,args=["crackme",num1,num2,num3,num4])
11 simgr = proj.factory.simgr(state)
12
13 simgr.explore(find=0x0040126d)
14 solver = simgr.found[0].solver
15 print(solver.eval(num1,cast_to=bytes))
16 print(solver.eval(num2,cast_to=bytes))
17 print(solver.eval(num3,cast_to=bytes))
18 print(solver.eval(num4,cast_to=bytes))
19 IPython.embed()
```

Angr Solution cont.

- Creates an entry SimState which is an emulation of what the program would look like if it were just starting, we pass it our BVSs through args[] because they are taken from the command line

```
1 import angr
2 from claripy import *
3 import IPython
4
5 proj = angr.Project("crackme",load_options={'auto_load_libs':False})
6 num1 = BVS("num1",16)
7 num2 = BVS("num2",16)
8 num3 = BVS("num3",16)
9 num4 = BVS("num4",16)
10 state = proj.factory.entry_state(argc=5,args=["crackme",num1,num2,num3,num4])
11 simgr = proj.factory.simgr(state)
12
13 simgr.explore(find=0x0040126d)
14 solver = simgr.found[0].solver
15 print(solver.eval(num1,cast_to=bytes))
16 print(solver.eval(num2,cast_to=bytes))
17 print(solver.eval(num3,cast_to=bytes))
18 print(solver.eval(num4,cast_to=bytes))
19 IPython.embed()
```

Angr Solution cont. 2

- Creates a SimulationManager that manages states, and allows you to step through and explore for a specific address. We use it to simulate what inputs would be needed to hit a success condition.

```
1 import angr
2 from claripy import *
3 import IPython
4
5 proj = angr.Project("crackme",load_options={'auto_load_libs':False})
6 num1 = BVS("num1",16)
7 num2 = BVS("num2",16)
8 num3 = BVS("num3",16)
9 num4 = BVS("num4",16)
10 state = proj.factory.entry_state(argc=5,args=["crackme",num1,num2,num3,num4])
11 simgr = proj.factory.simgr(state)
12
13 simgr.explore(find=0x0040126d)
14 solver = simgr.found[0].solver
15 print(solver.eval(num1,cast_to=bytes))
16 print(solver.eval(num2,cast_to=bytes))
17 print(solver.eval(num3,cast_to=bytes))
18 print(solver.eval(num4,cast_to=bytes))
19 IPython.embed()
```

Angr Solution cont. 3

- We access the SimulationManagers found[] list for a state that reached the success condition, then use the (Z3) solver to evaluate our BVSs and see what inputs led to this state. We cast them to bytes for a prettier output.

```
1  import angr
2  from claripy import *
3  import IPython
4
5  proj = angr.Project("crackme",load_options={'auto_load_libs':False})
6  num1 = BVS("num1",16)
7  num2 = BVS("num2",16)
8  num3 = BVS("num3",16)
9  num4 = BVS("num4",16)
10 state = proj.factory.entry_state(argc=5,args=["crackme",num1,num2,num3,num4])
11 simgr = proj.factory.simgr(state)
12
13 simgr.explore(find=0x0040126d)
14 solver = simgr.found[0].solver
15 print(solver.eval(num1,cast_to=bytes))
16 print(solver.eval(num2,cast_to=bytes))
17 print(solver.eval(num3,cast_to=bytes))
18 print(solver.eval(num4,cast_to=bytes))
19 IPython.embed()
```

IPython!

```
19 IPython.embed()
```

- Soups up your Python interpreter, and allows you to embed itself into scripts and interact with your script while it's running.
- Also includes features like:
  - Prettier interpreter
  - Better command/output history
  - Tab completion
  - See documentation of a python object with "?"
  - And more!
- Check out the documentation

```
Python 3.9.1 (default, Dec  8 2020, 07:51:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: sim
Out[1]: <SimulationManager with 1 active, 2 deadended, 1 found>

In [2]: sim.found[0]
Out[2]: <SimState @ 0x401250>

In [3]: sim.found[0].posix.stdin.concret
                                        concrete
                                        concretize()
```

Angr Demo 2

- Probably simpler to do that last one in Z3…
- How about a problem like this that expects multiple passwords?
- Pretend the answers aren't right in front of you
- Hey, this kinda reminds me of the binary bomb from CS367...

```
undefined8 main(void)

{
  int iVar1;
  char local_26 [10];
  char local_1c [10];
  char local_12 [10];

  puts("What\'s the password?");
  fgets(local_12,10,stdin);
  iVar1 = strncmp(local_12,"password",8);
  if (iVar1 == 0) {
    puts("\nWell done, but I still don\'t trust you...");
    puts("What\'s the second password?");
    fgets(local_1c,10,stdin);
    iVar1 = strncmp(local_1c,"secret",6);
    if (iVar1 == 0) {
      puts("\nFantastic! I almost believe you deserve access!");
      puts("What\'s the third password?");
      fgets(local_26,10,stdin);
      iVar1 = strncmp(local_26,"cs395",5);
      if (iVar1 == 0) {
        puts("\nGreat job! You\'re an Angr master!");
                    /* WARNING: Subroutine does not return */
        exit(0);
      }
    }
  }
  fail();
  return 0;
}
```

Angr Solution

- Short and powerful!
- Angr can explore all the way to the success condition, then tell us what input it used to get there

```
1 import angr
2 import IPython
3
4 proj = angr.Project("crackme",load_options={'auto_load_libs':False})
5 state = proj.factory.entry_state()
6 sim = proj.factory.simgr(state)
7
8 sim.explore(find=0x00401250)
9 print(sim.found[0].posix.stdin.concretize())
10 IPython.embed()
```

Good Luck on your Final Project!

I hope you have all enjoyed the class, and that you learned something you found interesting!

Hack The Planet