# Symbolic execution

## Summary

The goal of this laboratory exercise is to obtain hands-on practice with the ANGR symbolic execution framework. In this exercise, we’ll be using ANGR to automatically determine some of the solutions to "defusing" our binary from the bomb lab.



## Preparations

In your Linux VM:

* Install Ghidra per the previous instructions
* Install angr
  + (angr) reuben@VM:~$ pip install angr

## Background

* Instructor will go over some of the slides from the symbolic execution lecture

## Overall sequence

1. Prepare for symbolic execution
2. Plan the symbolic execution tasks and code the angr script to perform it
3. Run symbolic execution for fun and profit

## Step 1-prepare for symbolic execution

* Recall from the bomb lab that the decompiled main() function was similar to the contents below

Table 0.1-main() decompile for bomb binary

|  |
| --- |
| int main(int argc,char \*\*argv)  {  undefined8 uVar1;    if (argc == 1) {  infile = stdin;  }  else {  if (argc != 2) {  \_\_printf\_chk(1,"Usage: %s [<input\_file>]\n",\*argv);  /\* WARNING: Subroutine does not return \*/  exit(8);  }  infile = (FILE \*)fopen(argv[1],"r");  if (infile == (FILE \*)0x0) {  \_\_printf\_chk(1,"%s: Error: Couldn\'t open %s\n",\*argv,argv[1]);  /\* WARNING: Subroutine does not return \*/  exit(8);  }  }  initialize\_bomb();  puts("Welcome to my fiendish little bomb. You have 6 phases with");  puts("which to blow yourself up. Have a nice day!");  uVar1 = read\_line();  phase\_1(uVar1);  phase\_defused();  puts("Phase 1 defused. How about the next one?");  uVar1 = read\_line();  phase\_2(uVar1);  phase\_defused();  puts("That\'s number 2. Keep going!");  uVar1 = read\_line();  phase\_3(uVar1);  phase\_defused();  puts("Halfway there!");  uVar1 = read\_line();  phase\_4(uVar1);  phase\_defused();  puts("So you got that one. Try this one.");  uVar1 = read\_line();  phase\_5(uVar1);  phase\_defused();  puts("Good work! On to the next...");  uVar1 = read\_line();  phase\_6(uVar1);  phase\_defused();  return 0;  } |

* Our initial goal will be to setup ANGR to determine the solution for phase\_1
  + We need to identify the starting and ending addresses for symbolic execution and also what we desire to avoid (i.e., calls to explode\_bomb)
  + Let’s use Ghidra!
* First, let’s open the binary in Ghidra
* Next, we’ll want to locate the addresses for:
  + Start of phase\_1() and its call to explode\_bomb() and alternately, the successful exit
  + The address of explode\_bomb()
  + Address of input\_strings symbol (read\_line() saves user input to this address)

## Step 2-plan for the symbolic execution tasks and implement the code

* Clone this git project

$ git clone <https://gitlab.com/underpantsgnomes/softwaresecurity/symbolicexecutionlab>

* Open the solve.py script in an editor
* Let’s walk through the solve\_flag\_1() function

## Step 3-running symbolic execution for fun and profit

* Run it and you will receive the answers for all of the phases

$ python solve.py

* Enter them for yourself by separately running bomb to confirm

## Step 4-review the other phase solving tasks

* solve\_flag\_2
* solve\_flag\_3
* solve\_flag\_4
* solve\_flag\_5
* solve\_flag\_6
* solve\_secret

## Acknowledgements

* Cory Duplantis-tutorial author
* Rahul Nair, course assistant for JHU-ISI EN.650.660, 2016

## References

* <http://angr.io/>
* <http://ctfhacker.com/ctf/python/symbolic/execution/reverse/radare/2015/11/28/cmu-binary-bomb-flag2.html>
* <https://github.com/angr/angr-doc/blob/master/examples/cmu_binary_bomb/>