Angr

Tristan Wiesepape

What is Angr

- Python framework that does static and dynamic symbolic analysis of binaries
- Analyzes compiled code (not source)
- Doesn’t actually run the code

What can I do with Angr

- Determine what input is needed to reach some part of the code
- Determine what input is needed to get some output
- Determine what input is needed to control RIP

How does Angr analyze code

- Separates the code into basic blocks
  - Basic blocks are sections of code with no jumps
  - This forms a graph, where basic blocks are vertices and jumps are edges
- Angr analyzes code using a list of states
  - States represent a single point in time

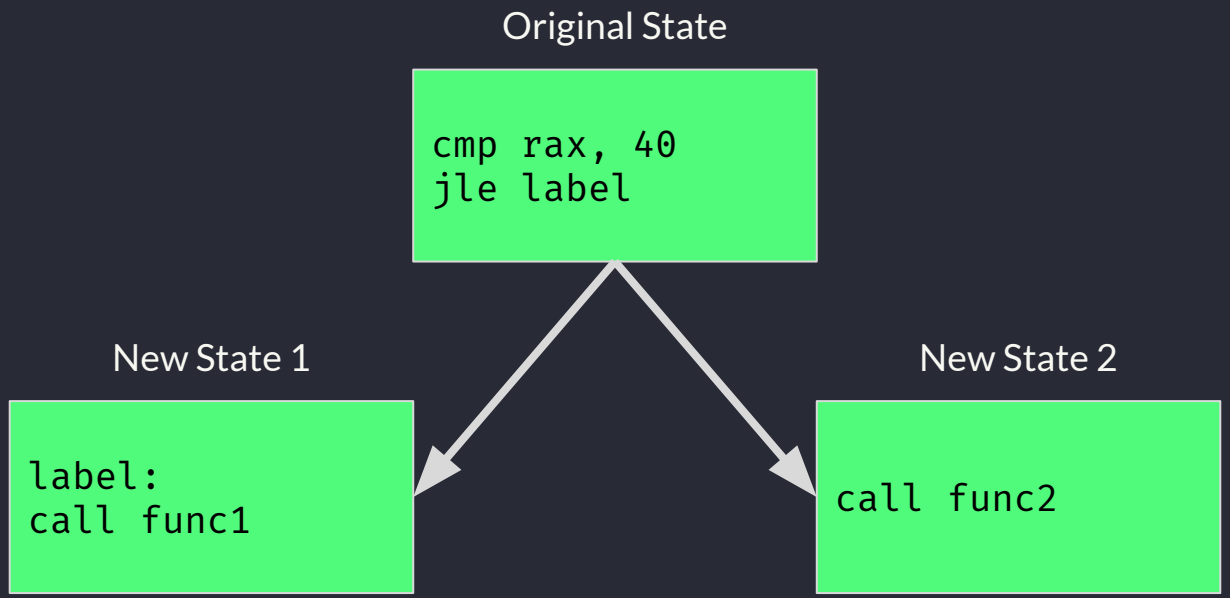
States

- States contain the entire state of the program, including memory, registers, and IO
- Angr analyzes programs as a tree of states
- It starts of with a single initial state, which represents the program at the start of execution
- We can then use the step function to find which states are possible from our current state

How exactly does the step function work?

- Calling step simulates the next basic block
- If the basic block is completely predictable, this results in a single new state
- If the basic block is not predictable this results in multiple new states
    - This can happen when the program does things like calling a libc function, or branching based on unknown data

Example


Original State
cmp rax, 40
jle label
New State 1
label:
call func1
New State 2
call func2

How do multiple states work?

- Having multiple states represents different possibilities
- Each state “remembers” under what conditions it would be run
  - For example, in the previous example state 1 would run if rax is less than 40, while state 2 would run if rax is more than 40
- Angr uses z3 to combine multiple conditions, and even detect impossible ones

How is this useful?

- We can use this to analyze and simplify complex conditions
- Angr can “reason backwards” using the eval function
  - Given some state it will tell us what input is needed to reach that state
- We can keep stepping until we reach some useful state, then look at the conditions to see how to reach this state
- Angr has algorithms for this built in

explore

- Explore is given a address, and told to figure out how to get to it
- Most commonly you would give it the address of get_flag, or some similar function
- It can also be given addresses to avoid as an optimization method

Misc details

- Claripy is used to create symbolic bitvectors
  - This is used to represent unknown inputs / user controlled data
- Stashes
  - Stashes are lists of states
  - Active: default stash containing currently running states
  - Deadended: state cannot continue to run, most commonly due to an error or calling exit
  - Unconstrained: the instruction pointer is based on user data. Angr cannot continue simulating at this point, but you can probably use this to do some kind of buffer overflow attack
  - Found: states that reach the address you are searching for with explore

Example Problem

- Go to forever.isss.io and try to solve angry in the reversing section
- Read angr docs to get more details on exactly how to load and simulate binaries
- Or read the foreverCTF tools guide for angr for more concrete details on angr
  - https://forever.isss.io/challenges#Angr-75
- Or come to the front of the room, and look at the annotated solution for the problem