Symbolic Execution

Note

I will focus on discussing angr's implementation choices, but there are many other ways to perform given tasks.

Installing angr

- Install Python + pip
- Install dependencies (libffi, maybe others)
    - sudo apt-get install libffi-dev
- Install python virtualenv wrappers
- mkvirtualenv angr
- pip install angr

Goals For This Talk

0x1

- Gain high level understanding of what Symbolic Execution is
- Discover tools for further exploration
- Hopefully inspire you to learn more about it

0x0

- To make you an expert on Symbolic Execution

TL;DR

Login Demo

Context

Static Analysis

- Looking at code
- Not running it
- Good for simple programs or functions
- Allows full understanding of code
- i.e.: IDA Pro

Dynamic Analysis

- Run the code and see what happens
- Useful for complicated code
- i.e.: gdb

Symbolic Analysis

- Simulate running the code
- Don’t provide real input
- Record what happens in sets of statements
- Attempt to solve statements to answer questions

Why Symbolic Execution?

Why I've started learning and you should too.

Awesome CTF Teams

Plaid Parliament of Pwning (Carnegie Mellon University)

BAP (Binary Analysis Platform)

https://github.com/BinaryAnalysisPlatform/bap

http://pwning.net/

ShellPhish (UC Santa Barbara and Northeastern University)

angr

http://angr.io/

http://www.shellphish.net/cli.html

Awesome CTF Teams

Kenshoto (DEFCON CTF Host 2005-2008)

Vivisect/Symboliks

https://github.com/vivisect/vivisect

http://visi.kenshoto.com/viki/MainPage

WTF is Symbolic Execution?

They must be onto something...

“In computer science, symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute.” -- Wikipedia :)

Summary

- Don’t execute the binary, enumerate the paths
- Walk through binary *as-if* executing
- Record constraints based on reads/writes/if-else/etc
- Take every possible path
    - Don’t take “if” side
    - Don’t take “else” side
    - Take both. Always.
- Can start anywhere, end anywhere
- (KEY) Be able to solve the constraints discovered

Nothing New

- “Symbolic Execution and Program Testing” (James C. King, 1976, Commun. ACM)
- Tons of papers on the concept since
- Has stayed mostly in the academic world until recently
- Mostly focuses on testing for bugs and execution paths compiling from source code…
- Not us!

Common Use

- Find paths to code that shouldn't exist
  - Any time you would insert an "assert" statement and hope you never get there
- Used alongside fuzzing but has the ability to be much more thorough
- Binary crash reproduction
  - We know the end state of the crashed program, ask Symbolic Execution to recreate it
- Many others (google it!)

Blackbox Symbolic Analysis

- No source code available (cannot use industry standards like KLEE)
- Mostly useful for binary reversing
  - Malware/Exploitation/lost source code?
- Problems
  - Harder to individually describe inputs as symbolic (worsens path explosion)
  - Coverage problems (won't necessarily find all paths)

Aspects of Symbolic Execution

- Binary Loading
- Intermediate Language Representation
- Intermediate Language Simulation
- Constraint Solving

Binary Loading

- Binary formats contain the information and structure the operating system needs to successfully load the executable and run it

- Binaries come in many formats
  - *nix -- ELF/COFF
  - Windows -- PE/MZ/COM
  - Mac OS -- XCOFF/PEF
  - Atari ST -- GEMDOS :-D

CLE Loads Everything

- CLE performs the following steps
  - Parses executable file (many formats supported)
  - Sets variables and classes that you can use to manually inspect the binary and run other analysis
  - CLE does NOT perform any analysis of it's own. It only loads the file.

CLE Demo

File Info

Intermediate Language Representation

- Most Symbolic Execution applications will use an intermediate language.

- Instead of having to read x86, x86-64, MIPS, ARM, etc, the engine only has to read the intermediate language

- Many assembly languages have strange and undocumented side-effects of commands (such as register and flag interactions)

Intermediate Language Representation

- Utilizing Intermediate Language simplifies the work of the engine to focusing on one language, speeds development, and modularized the approach.
- The act of going from normal assembly (i.e.: x86) to intermediate language is called “lifting”
- Done accurately, this allows for black-box cross compiling (i.e.: no source code translation from x86 to MIPS)

Common Intermediate Languages

- BAP Intermediate Language (CMU)
  - http://users.ece.cmu.edu/~aavgerin/papers/bap-cav-11.pdf

- LLVM-IR (International Consortium)
  - http://llvm.org/releases/2.7/docs/LangRef.html

Valgrind and VEX

- “Valgrind is an instrumentation framework for building dynamic analysis tools”
- Valgrind uses an intermediate language called VEX
  - http://valgrind.org/
- A lot of support for lifting to VEX already exists as the Valgrind project has been around for a while
- Since the main use of Valgrind is to attach to binaries without source code, VEX is a natural solution for Symbolic Execution on black-box binaries.

VEX Demo

Intermediate Language Simulation

- Something needs to execute the intermediate language. (think "*emulator*")
- Execution will create machine states that contain constraints.
- For simple functions, you can execute them entirely in the intermediate language, regardless of their base language (i.e.: x86, ARM, MIPS, etc).

Constraint Solving

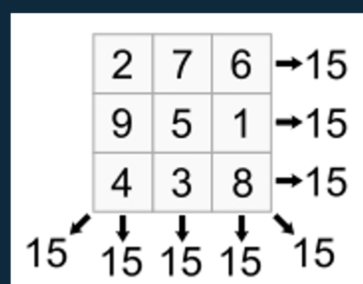
- Simulating the binary naturally creates constraints

- Once a path is found to a part of the code, you can solve the resulting constraints to answer questions

- Microsoft's Z3 Theorem Prover
  - https://github.com/z3prover/z3/wiki
  - One of the top solvers and theorem provers

Constraint Solving

- Example: Magic Square (Demo)



Sudoku Challenge (Demo)

- Create a Sudoku solver using Z3
  - Bonus: Check for and print multiple solutions if they exist

- Create a Sudoku generator using Z3

Login Revisited