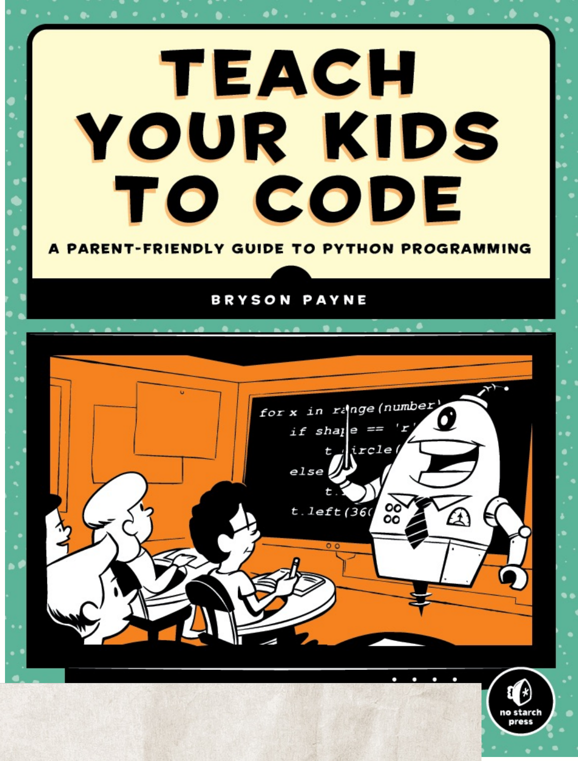
Symbolic Execution for the Win: Pwning CTFs with angr

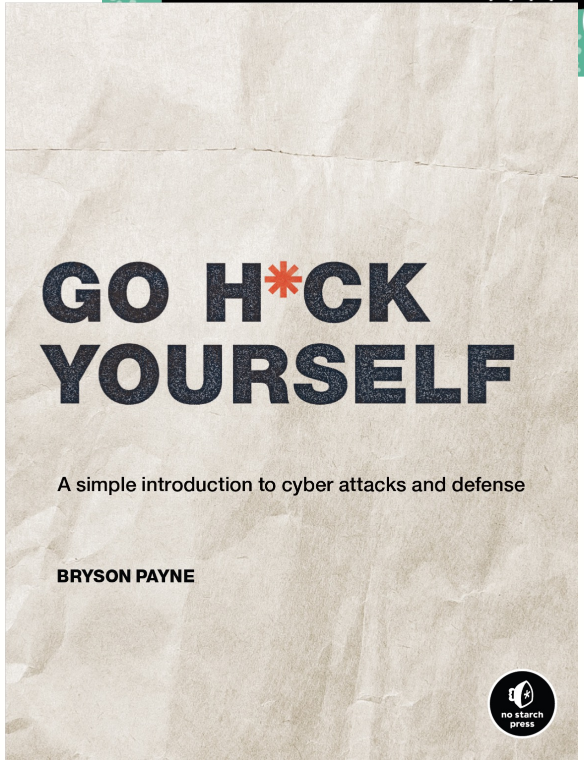
Dr. Bryson Payne, CISSP, CEH, GPEN, GRID, GREM
Professor of Computer Science
Coordinator, Student Cyber Programs

UNG
UNIVERSITY of NORTH GEORGIA™

About Me

- Dr. Payne: Ph.D. in computer science from Georgia State University, 6 years as a CIO, 24 years teaching CS/IS/Cyber in the University System
- Author of *Teach Your Kids to Code*, *Go Hack Yourself*; next book *Hacking for Kids* comes out Jan 2023
- Coach for the #1 2019 & 2020 NSA Codebreaker Challenge
- Coaching Staff for US Cyber Team





UNG
UNIVERSITY of NORTH GEORGIA™

Intro

- Competitions and CTFs **motivate and engage** students in cybersecurity and cyber ops
- Reverse engineering and pwn/binary exploit challenges are common in CTFs, but the tools have a steep learning curve, not all programs teach RE
- angr is a Python framework for analyzing binaries
- Built as part of DARPA Cyber Grand Challenge
- Can be used to solve CTF challenges (and find real vulnerabilities) in *almost* automated fashion

UNG
UNIVERSITY *of* NORTH GEORGIA™

What is angr?

- angr is a multi-architecture binary analysis toolkit
- Can perform both static and dynamic, concrete and symbolic (or *concolic*) analysis, including:
  - Disassembly
  - Symbolic execution
  - Control-flow analysis
  - Data-dependency analysis
  - Value-set analysis (VSA)
  - Decompilation

Steps in Symbolic Execution w/angr

- Load a binary for analysis
- Translate the binary into intermediate representation
- Perform symbolic exploration of the program's possible states
- Explore to find the states that lead to a win/success state in a CTF
- [Optional: avoid states that lead to loss/failure]

Installing angr in Python

- Kali:

```
#angr
sudo apt install python3-pip
pip install angr
pip install pycparser --force
```

- Windows

```
pip3 install angr
```

- Virtualenv recommended
- Official docs/install instructions:

https://docs.angr.io/introductory-errata/install

All sample files from today

- Challenge binaries are courtesy of Point3's ESCALATE platform
- https://tinyurl.com/CAETechTalk-angr

Simple angr CTF attack:

```
import angr, claripy
project = angr.Project('Lin64_1')
flag = claripy.BVS('flag',8*256) # variable we’re solving for
state = project.factory.entry_state(args=['Lin64_1', flag])
simgr = project.factory.simulation_manager(state)
simgr.explore(find=0x004005c6) # “success” address
print(simgr.found[0].solver.eval(flag, cast_to = bytes))
```

Demo – Ghidra and angr

- Do quick analysis in Ghidra to find “win/success”
- Plug in this address to the simulation manager’s explore method as the “find” address

Refining our angr

- We can clean up the flag to a shorter bit vector
- We can add a list of addresses to **avoid** in the explore method

Faster angr

```
import angr, claripy
project = angr.Project('Lin64_2')
flag = claripy.BVS('flag',8*39)
state = project.factory.entry_state(args=['Lin64_2', flag])
simgr = project.factory.simulation_manager(state)
simgr.explore(find=0x00400896, avoid=[0x4008ac])
print(simgr.found[0].solver.eval(flag,cast_to=bytes))
```

Clean up error messages with options

- Add to the entry_state:
  add_options={
  angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
  angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS}

```
import angr, claripy
project = angr.Project('Lin64_3’)
flag = claripy.BVS('flag',8*39)
state = project.factory.entry_state(args=['Lin64_3',flag],add_options={
   angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
   angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS})
simgr = project.factory.simulation_manager(state)
simgr.explore(find=0x004006f9, avoid=[0x40070f])
print(simgr.found[0].solver.eval(flag, cast_to = bytes).decode('utf-8'))
```

Additional optimizations

- Flag values (and input strings) are usually printable characters, ASCII 0x20-0x7e (space to ~) – most CTFs exclude the space
- We can add constraints to each byte of the flag symbol:

```
for byte in flag.chop(8):
        state.solver.add(byte < 0x7f)
        state.solver.add(byte >= 0x20)
```

Demos

- Clever multi-solver with lambda function based on output
- Windows solvers

What if the flag is stdin input?

```
input_length = 39
input_chars = [claripy.BVS("char_%d" % i, 8) for i in range(input_length)]
input = claripy.Concat(*input_chars)
state = proj.factory.entry_state(args=["./file"], stdin=input)
for byte in input_chars:
  state.solver.add(byte >= 0x20, byte <= 0x7e)
…
print(simgr.found[0].solver.eval(input, cast_to=bytes).decode('utf-8'))
```

Conclusion

- angr is a symbolic execution tool worth introducing to your Reverse Engineering students and CTF competition teams
- angr can be used by novices and experts alike, often in a fraction of the time required with debuggers, disassemblers, and decompilers
- But, you probably still need some basic RE skills (Ghidra)
- All files from today: https://tinyurl.com/CAETechTalk-angr
- Q&A – Thank You!