Constraint Solving

Objectives

- Introduce symbolic/concolic execution as a tertiary means of analyzing compiled binary code.
- Examine how to use symbolic execution to discovery the control flow graph of a program.
- Implement small angr scripts to discover user input to transition to different program paths.

References

- https://angr.io
- https://github.com/angr/angr-doc/blob/master/CHEATSHEET.md

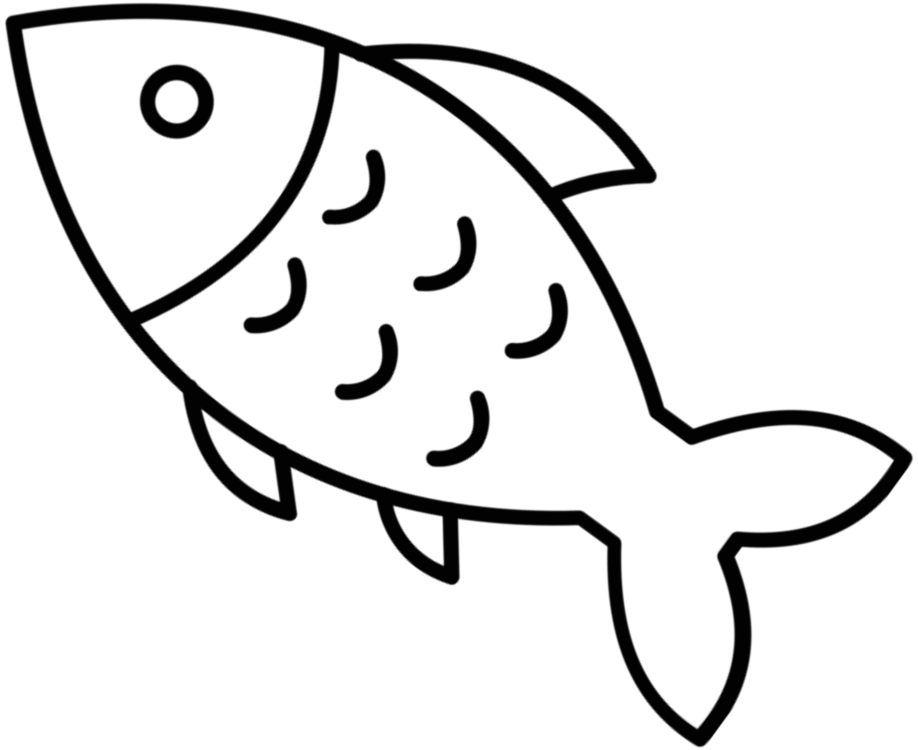
History: Cyber Grand Shellphish

The DARPA Cyber Grand Challenge (CGC) was designed as a Capture The Flag (CTF) competition among autonomous systems without any humans being involved.

The Shellphish team put together a prototype of a system that automatically identifies crashes in binaries using a novel composition of fuzzing and symbolic execution

Mechanical Phish is a highly-available, distributed system that can identify flaws in DECREE binaries, generate exploits (called Proofs Of Vulnerability, or POVs), and patched binaries, without human intervention.

After placing 3rd in the DARPA GCC, Shellphish decided to make our system completely open-source so that others can build upon and improve what they put together.



Text copied from: Cyber Grand Shellphish

History: CrackAddr

Buffer overflow in an email address parsing function of Sendmail [discovered in 2005](). Consisted of a parsing loop using a state machine. At each round of the loop, one byte gets overwritten.

This is known as a **complex loop satisfaction** bug, which required an enormous number of inputs to satisfy the condition

The complexity of this bug made Halvar Flake (Thomas Dullien) from Google Zero argued that automation **could not currently solve bugs like crackaddr.**

Mechanical Phish (aka Angr) was the only framework that solved crackadrr.

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit --;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE;  // upperlimit--;
            if ((c == ')') && (!quotation) && roundquote) {
                roundquote = FALSE; upperlimit++;
            }
// If there is sufficient space in the buffer, write the character.
            if (outputIndex < upperlimit) {
                localbuf[outputIndex] = c; outputIndex ++;
            }
        }
        if (roundquote) {
            localbuf[outputIndex] = ')'; outputIndex++;
        }
        if (quotation) {
            localbuf[outputIndex] = '>'; outputIndex++;
        }
    }
```

Code copied from:
http://2015.hackitoergosum.org/slides/HES2015-10-29%20Cracking%20Sendmail%20crackaddr.pdf

Concolic Execution

Concolic execution engines interpret an application, model user input using symbolic variables, track constraints introduced by conditional jumps, and use constraint solvers to create inputs to drive applications.

While these systems are powerful, they suffer from a fundamental problem: if a conditional branch depends on symbolic values, it is often possible to satisfy both the taken and non-taken condition. Thus, the state has to fork and both paths must be explored. This quickly leads to the **well-known path explosion problem**, which is the primary inhibitor of concolic execution techniques.

Symbolic Execution

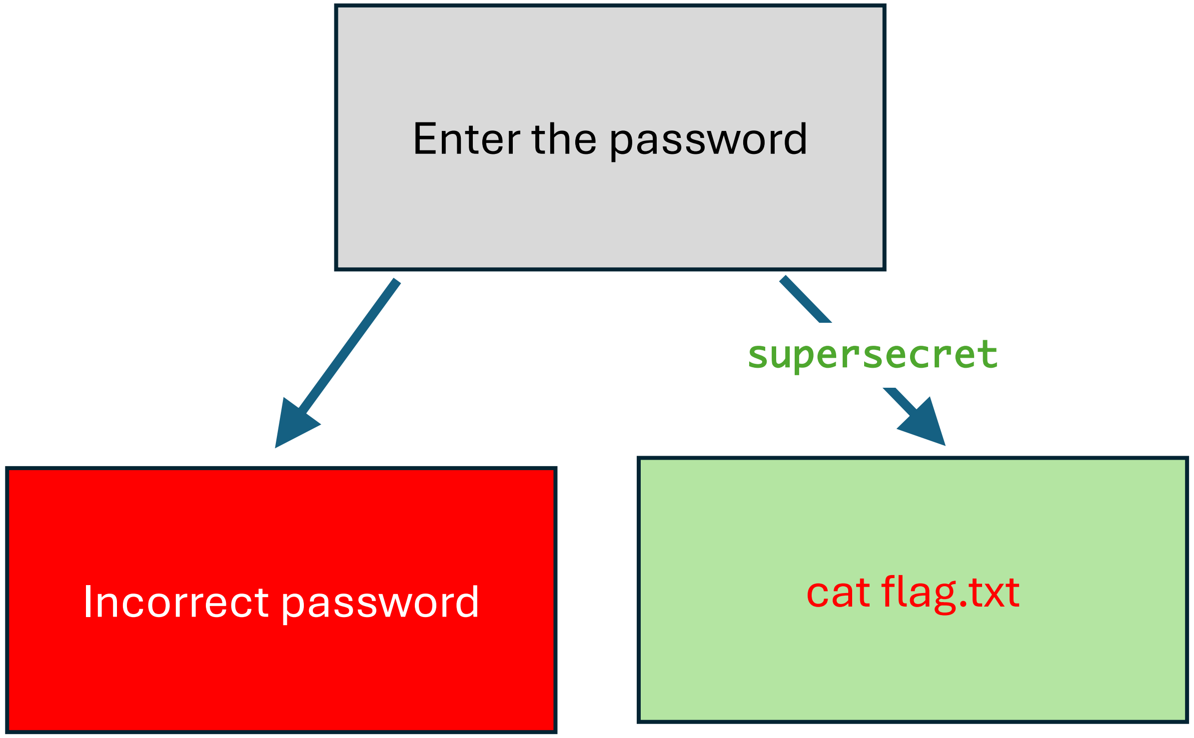
\+ Concrete Execution

Concolic Execution

Text copied from: Driller: Augmenting fuzzing through selective symbolic execution.

Modeling Paths (Program Dead Ends)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
   char password[50];

   printf("Enter the password: ");
   scanf("%49s", password);
   if (strcmp(password, "supersecret") == 0) {
      system("cat flag.txt");
   } else {
      printf("Incorrect password!\n");
   }
   return 0;
}
```



Imagine modeling our program from before. You have two potential outputs (dead-ends) – you either correctly input the password or you do not.

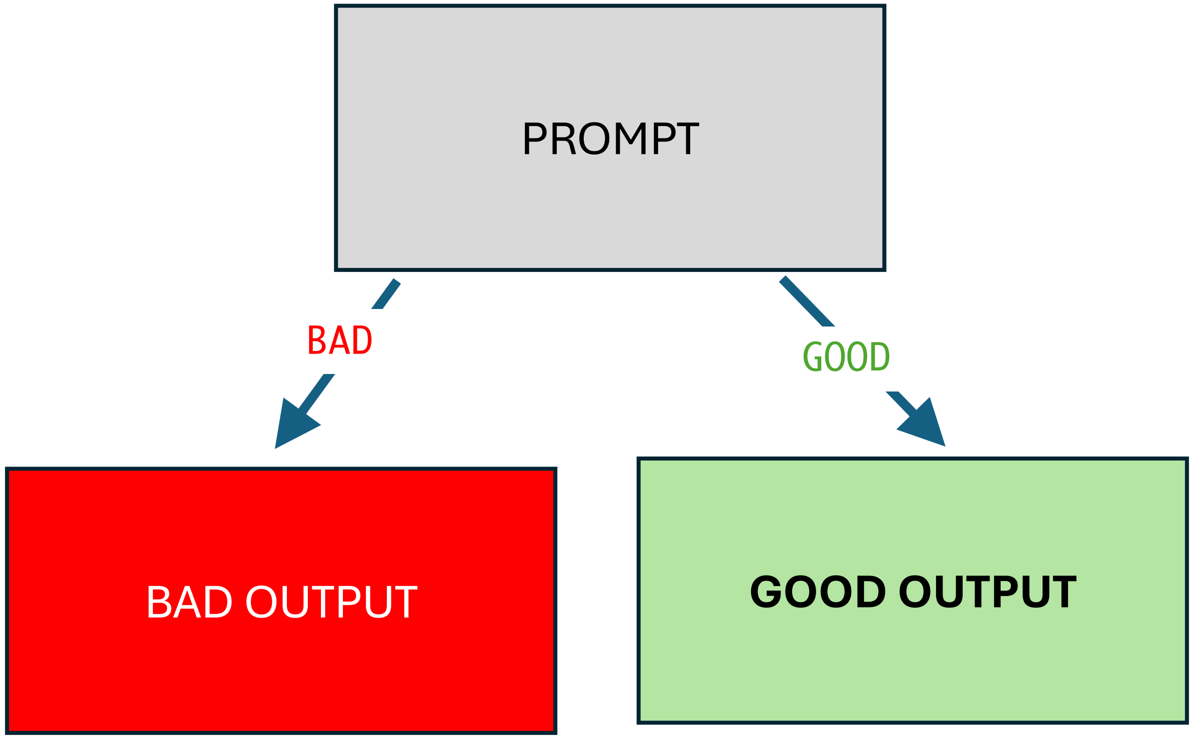
Modeling Paths (Program Dead Ends)

```
import angr, logging
logging.getLogger('angr').setLevel('CRITICAL')

p = angr.Project('./test',main_opts={"base_addr": 0x400000})
state = p.factory.entry_state()
sm = p.factory.simulation_manager(state)

sm.run()

for i in range(len(sm.deadended)):
    msg = f"\t Dead-End #{i}: Input = {sm.deadended[i].posix.dumps(0)}"
    print(msg)
    msg = f"\t Dead-End #{i}: Output = {sm.deadended[i].posix.dumps(1)}"
    print(msg)
```

We can use angr to explore the potential dead-ends and the input that caused each dead-end.

```
python3 dead-end.py
        --0
        [+] Deadend 0: Input = b'\x10\x04\x01@@\x01\x01\x08\x01\x01….'
        --1
        [+] Deadend 1: Input = b'supersecret\x00\x00\x00\x00\x00\x00….
```

Modeling Paths (Program Dead Ends)



A lot of CTF challenges have a similar type of format. They prompt the user for information and then either display a good or bad message.

Angr’s Explore Functionality

- Angr has a high-level function named explore()

- We can use it to search with the following parameters

  - find: goal or target addresses or conditions
  - avoid: addresses or conditions that should be avoided
  - num_find: limits the number of states to find

Example Problem: BabyREEE

```
00001080  int32_t main(int32_t argc, char** argv, char** envp)

0000108f      puts(str: "Hello! Welcome to SEETF. Please …  ")
000010ae      int128_t var_d8
000010ae      __builtin_memcpy(dest: &var_d8, src: "\x98\x00\x00\x00\x8b\x00\x00\x00\x88\x00\x00\x00\xc3\x00
00001176      void buf
00001176      fgets(buf: &buf, n: 0x80, fp: stdin)
00001187      int32_t rax_2
00001187      if (strlen(&buf) != 0x35)
00001192          printf(format: "Flag wrong. Try again.")
00001197          rax_2 = 1
00001187      else
000011ac          puts(str: "Good work! Your flag is the corr…  ")
000011b8          puts(str: "On to the flag check itself...")
000011c5          int64_t rdx_1 = 0
000011c7          int64_t rax_4 = strlen(&buf) - 1
000011e9          while (true)
000011e9              int32_t r8_1 = rdx_1.d
000011ef              if (rax_4 == rdx_1)
000011f8                  puts(str: "Success! Go get your points, cha…  ")
000011fd                  rax_2 = 0
000011ff                  break
000011d4              char rsi = (*(&var_d8 + (rdx_1 << 2))).b
000011de              char rcx_3 = (*(&buf + rdx_1) + 0x45) ^ rdx_1.b
000011e0              rdx_1 = rdx_1 + 1
000011e7              if (rsi != rcx_3)
0000120d                  printf(format: "Flag check failed at index: %d", zx.q(r8_1), rdx_1, rcx_3)
00001212                  rax_2 = 1
00001217                  break
000011a4      return rax_2
```

BAD OUTPUT

GOOD OUTPUT

```python
import angr
import sys
from pwn import *

GOOD = args.GOOD
BAD  = args.BAD
BASE = 0x400000

def main(argv):
  path_to_binary = args.BIN

  project = angr.Project(path_to_binary, main_opts={"base_addr": BASE})
  initial_state = project.factory.entry_state(
    add_options = { angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
                    angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS}
  )
  simulation = project.factory.simgr(initial_state)

  def good_path(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return GOOD.encode() in stdout_output

  def bad_path(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return BAD.encode() in stdout_output

  simulation.explore(find=good_path, avoid=bad_path)

 if simulation.found:
    msg = f"Solution: {simulation.found[0].posix.dumps(0).decode()}"
    print(msg)

if __name__ == '__main__':
  main(sys.argv)
```

```
 python3 re-chall.py BIN=./chall GOOD="Success" BAD="wrong"
Solution: {SEE{0n3_5m411_573p_81d215e8b81ae10f1c08168207fba396}
```

Knowing this, we wrote a small angr script to solve all similar challenges.