Lec09: Fuzzing and Symbolic Execution

Taesoo Kim

Administrivia

- Three more labs! including NSA code-breaking challenge!
- Please submit your working “exploits” for previous weeks!
- New recitations:
  - Monday: 18:00~19:00, CoC 053 (Oct 29th: S106 Howney Physics)
  - Wednesday: 18:00~19:00, CoC 052
- In-class CTF on Nov 16-17 (24 hours)!
- Due: Find your team members, and let us know ASAP!
- Due: Submit your CTF challenge by Nov 13!

So far, focuses are more on “exploitation”

- More important question: how to find bugs?
  - With source code (we will see in the last lecture!)
  - With only binary

Two Pre-conditions (often much difficult!)

1. Locating a bug (i.e., bug finding)
2. Triggering the bug (i.e., reachability)

```
1  // Q2. How to reach this path?
2  if (magic == 0xdeadbeef) {
3      // Q1. Is this buggy?
4      memcpy(dst, src, len)
5  }
```

Solution 1: Code Auditing (w/ code)

```
1  static OSStatus SSLVerifySignedServerKeyExchange(...) {
2   ...
3   if (err = SSLHashSHA1.update(&hashCtx, &clientRandom))
4       goto fail;
5   if (err = SSLHashSHA1.update(&hashCtx, &serverRandom))
6       goto fail;
7   if (err = SSLHashSHA1.update(&hashCtx, &signedParams))
8       goto fail;
9       goto fail;
10  if (err = SSLHashSHA1.final(&hashCtx, &hashOut)
11      goto fail;
12
13  err = sslRawVerify(...);
14 fail:
15     return err;
16 }
```

Solution 2: Static Analysis (on binary)

- Reverse Engineering (e.g., IDA)

Problem: Too Complex (e.g., browser)

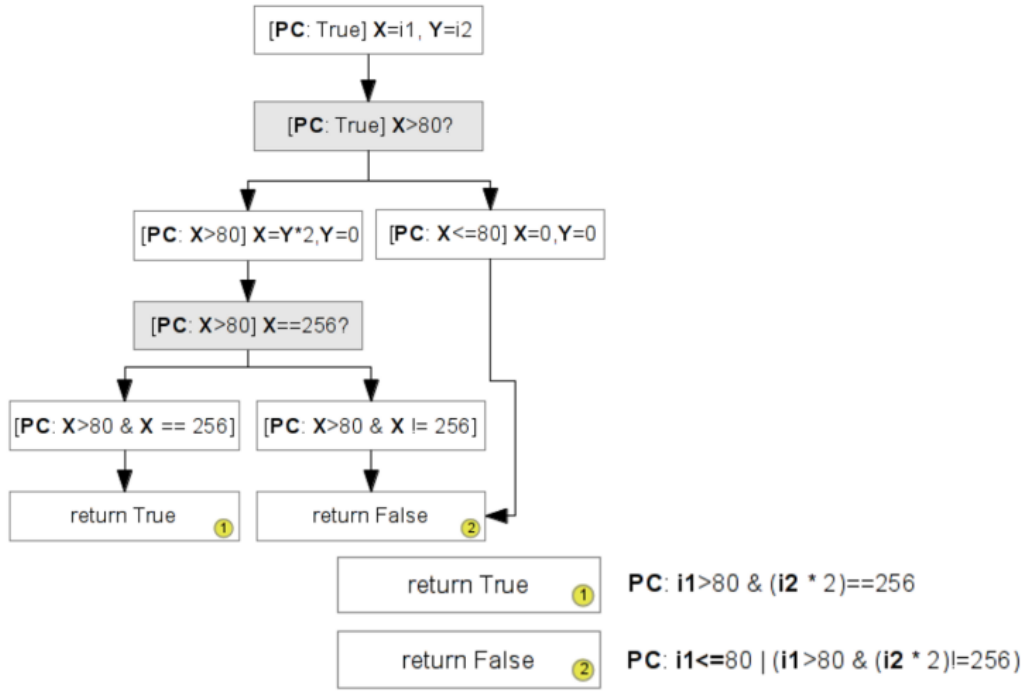
Two Popular Directions

- Symbolic execution (also static)
- Fuzzing (dynamic)

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```



Problem: State Explosion

- Too many path to explore (e.g., strcmp(“hello”, input))
- Too huge state space (e.g., browser? OS?)
- Solving constraints is a hard problem

Today's Topic: Fuzzing

- Two key ideas
  - Reachability is given (since we are executing!)
  - Focus on quickly exploring the path/state
    - How? mutating inputs
    - How/what to mutate? based on code coverage!

How well fuzzing can explore all paths?

```
1  int foo(int i1, int i2) {
2    int x = i1;
3    int y = i2;
4
5    if (x > 80) {
6      x = y * 2;
7      y = 0;
8      if (x == 256) {
9  *     __builtin_trap();
10       return 1;
11     }
12   } else {
13     x = 0; y = 0;
14   }
15   return 0;
16 }
```

DEMO: LibFuzzer

```
// $ clang -fsanitize=fuzzer ex.cc
// $ ./a.out
extern "C" int
LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  if (size < 8)
    return 0;

  int i1, i2;
  i1 = *(int *)(&data[0]);
  i2 = *(int *)(&data[4]);
  foo(i1, i2);

  return 0;
}
```

Game Changing Fact: Speed

- In this example,
  - Symbolic execution explores/checks just two conditions
  - Fuzzing requires 256 times (by scanning values from 0 to 256)
- What if fuzzer is an order of magnitude faster (say, 10k times)?
- In fact, LibFuzzer was much faster thanks to lots of heuristics!

Importance of High-quality Corpus

- In fact, fuzzing is really bad at exploring paths
  - e.g., if (a == 0xdeadbeef)
- So, paths should be (or mostly) given by corpus (sample inputs)
  - e.g., pdf files utilizing full features
  - but, not too many! (do not compromise your performance)
- A fuzzer will trigger the exploitable state
  - e.g., len in malloc()

AFL (American Fuzzy Lop)

- VERY well-engineered fuzzer w/ lots of heuristics

Examples of Mutation Techniques

- interest: -1, 0x8000000, 0xffff, etc
- bitflip: flipping 1,2,3,4,8,16,32 bits
- havoc: random tweak in fixed length
- extra: dictionary, etc
- etc

Key Idea 1: Map Input to State Transitions

- Input → [IPs] (problem?)

Key Idea 1: Map Input to State Transitions

- Input → [IPs] (problem?)
- Input → map[IPs % len] (problem? A→B vs B→A)

Key Idea 1: Map Input to State Transitions

- Input → [IPs] (problem?)
- Input → map[IPs % len] (problem? A→B vs B→A)
- Input → map[(prevIP >> 1 ^ curIP) % len] (problem?)

Key Idea 1: Map Input to State Transitions

- Input → [IPs] (problem?)
- Input → map[IPs % len] (problem? A→B vs B→A)
- Input → map[(prevIP >> 1 ^ curIP) % len] (problem?)
- Input → map[(rand1 >> 1 ^ rand2) % len]

Key Idea 2: Avoiding Redundant Paths

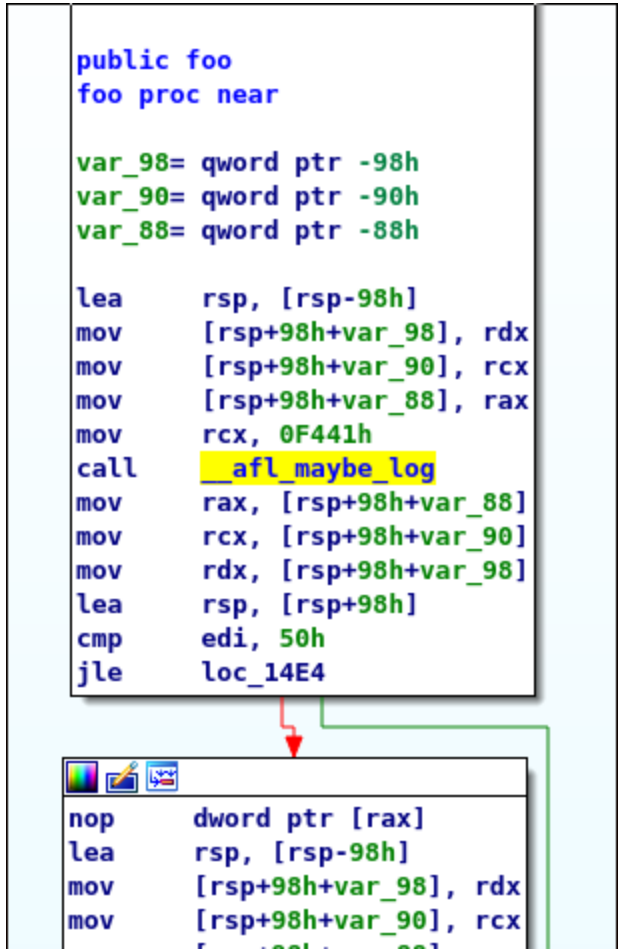
- If you see the duplicated state, throw out
  - e.g., i1 = 1, 2, 3
- If you see the new path, keep it for further exploration
  - e.g., i1 = 81

How to Create Mapping?

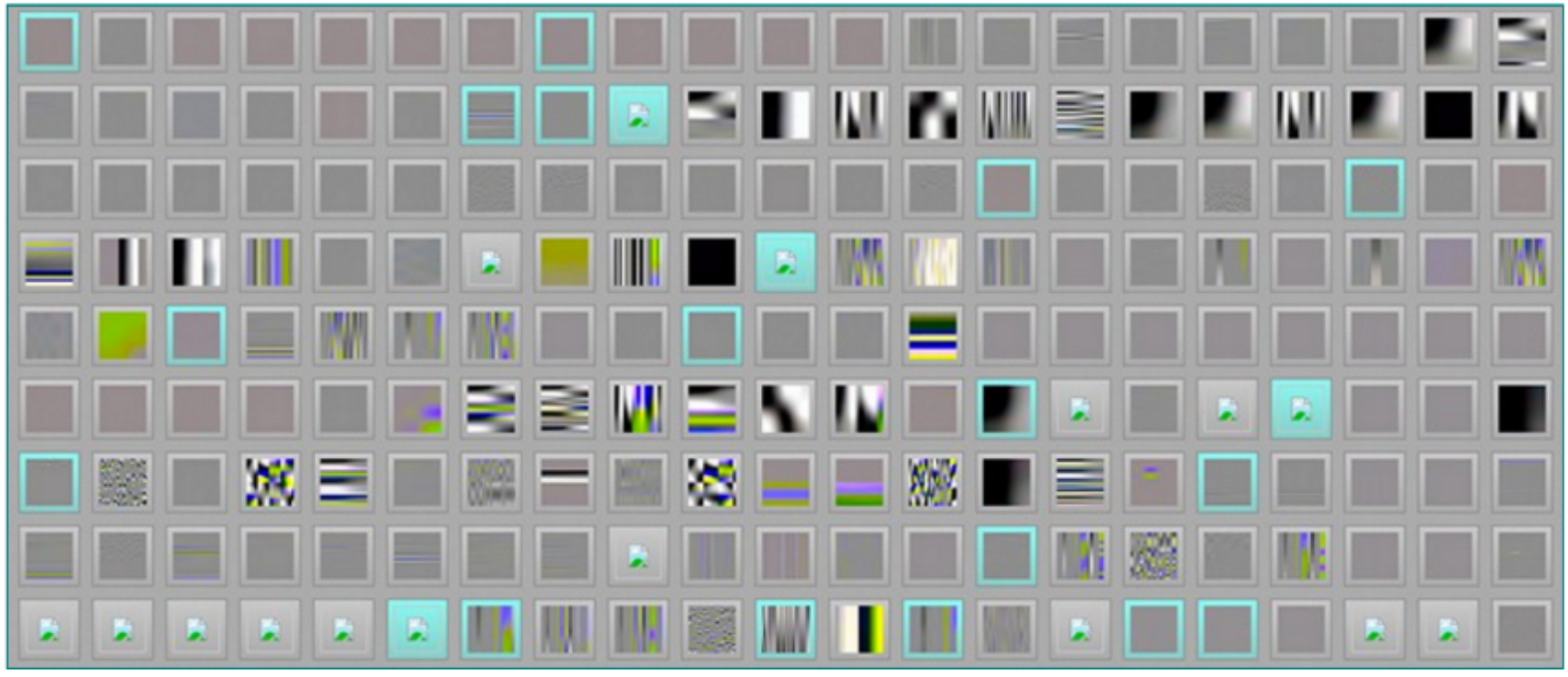
- Instrumentation
  - Source code → compiler (e.g., gcc, clang)
  - Binary → QEMU

```
1  if (block_address > elf_text_start
2      && block_address < elf_text_end) {
3    cur_location = (block_address >> 4) ^ (block_address << 8)
4    shared_mem[cur_location ^ prev_location] ++;
5    prev_location = cur_location >> 1;
6  }
```

Source Code Instrumentation



AFL Arts



Ref. http://lcamtuf.coredump.cx/afl/

Other Types of Fuzzer

- Radamsa: syntax-aware fuzzer
- Cross-fuzz: function syntax for Javascript
- langfuzz: fuzzing program languages
- Driller/QSYM: fuzzing + symbolic execution

Today's Tutorial

- In-class tutorial:
  - Fuzzing with AFL
  - Fuzzing with LibFuzzer

```
$ scp -P 9007 lab07@computron.gtisc.gatech.edu:fuzzing.tar.xz .
$ unxz fuzzing.tar.xz
$ docker load -i fuzzing.tar
$ docker run --privileged -it fuzzing /bin/bash

$ git pull
$ cat README
```

References

-Sanitize, Fuzz, and Harden Your C++ Code