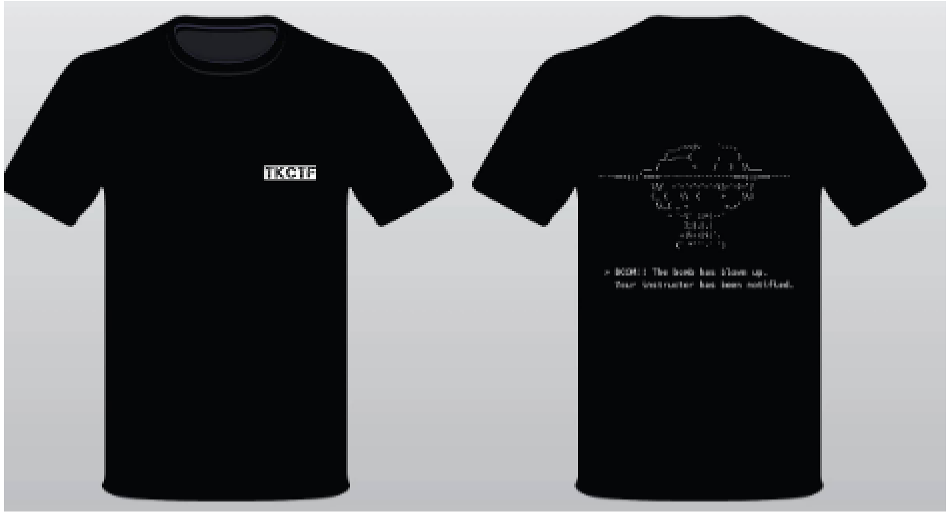
Lec10: Fuzzing and Symbolic Execution

Taesoo Kim

Administrivia


TKCTF
> BOOM!!! The bomb has blown up.
Your instructor has been notified.

- In-class CTF on Nov 22-23 (24 hours)!
- Due: Submit your CTF challenge by Nov 14!
- But submit it early for our feedback!

Emphasis on Exploitation (so far)

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

Two Pre-conditions for Exploitation

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

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

Solution 1: Code Auditing (w/ code)

```
static OSStatus SSLVerifySignedServerKeyExchange(...) {
 ...
 if (err = SSLHashSHA1.update(&hashCtx, &clientRandom))
     goto fail;
 if (err = SSLHashSHA1.update(&hashCtx, &serverRandom))
     goto fail;
 if (err = SSLHashSHA1.update(&hashCtx, &signedParams))
     goto fail;
     goto fail;
 if (err = SSLHashSHA1.final(&hashCtx, &hashOut)
     goto fail;

 err = sslRawVerify(...);
```

Solution 2: Static Analysis (on binary)

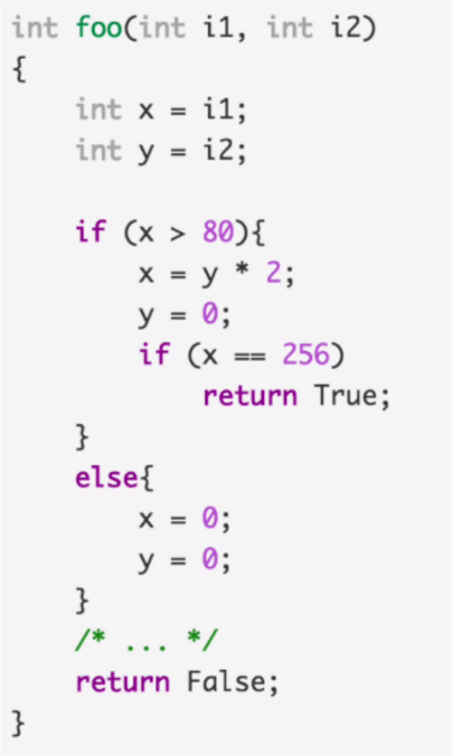
- Reverse Engineering (e.g., IDA or Ghidra)

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

Two Popular Directions

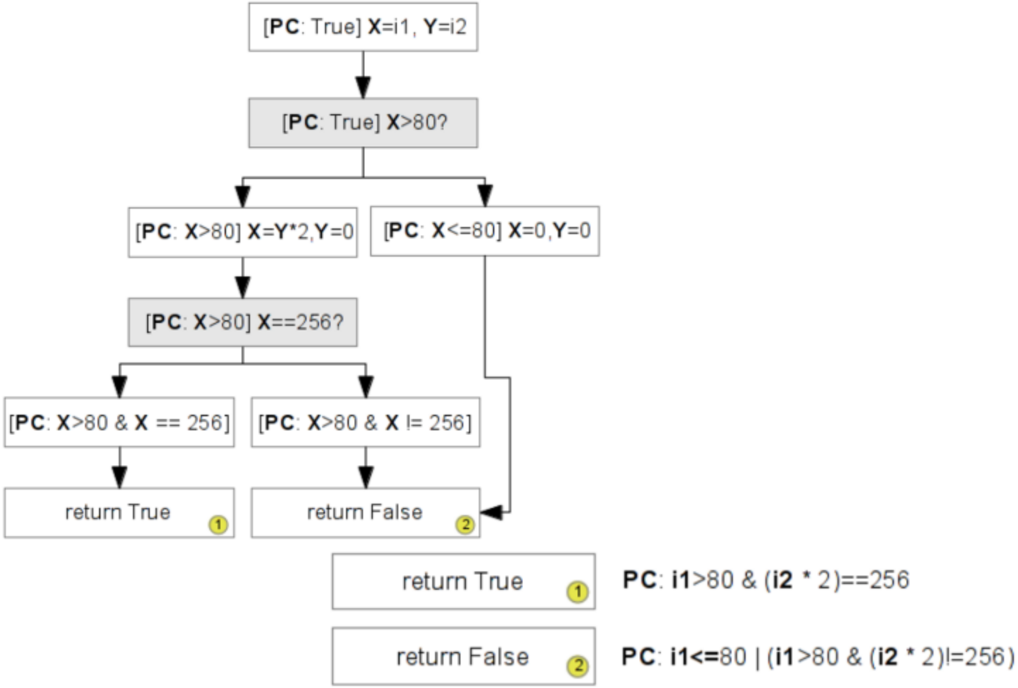
- Symbolic execution (static)
- Fuzzing (dynamic)

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;
}
```



[**PC**: True] **X**=i1, **Y**=i2

[**PC**: True] **X**>80?

[**PC**: **X**>80] **X**=**Y***2,**Y**=0

[**PC**: **X**<=80] **X**=0,**Y**=0

[**PC**: **X**>80] **X**==256?

[**PC**: **X**>80 & **X** == 256]

[**PC**: **X**>80 & **X** != 256]

return True ①

return False ②

return True ① **PC**: **i1**>80 & (**i2** * 2)==256

return False ② **PC**: **i1<=**80 | (**i1**>80 & (**i2** * 2)!=256)

Problem: State Explosion

- Too many path to explore (e.g., strcmp(input1, input2))
- Too huge input/state space (e.g., browser? OS?)
- Solving constraints is a hard problem (i.e., slow)

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?

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

  if (x > 80) {
    x = y * 2;
    y = 0;
    if (x == 256) {
*     __builtin_trap();
      return 1;
    }
  } else {
    x = 0; y = 0;
  }
  return 0;
}
```

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;
}
```

DEMO: Afl

```c
// $ afl-gcc ex-afl.c
// $ afl-fuzz -i input -o output ./a.out
int main(int argc, char* argv[]) {
  int i1 = 0;
  int i2 = 0;

  read(0, &i1, sizeof(i1));
  read(0, &i2, sizeof(i2));

  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

Idea 1: Map Input to State Transitions

- Input → [IPs] (problem?)

Idea 1: Map Input to State Transitions

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

Idea 1: Map Input to State Transitions

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

Idea 1: Map Input to State Transitions

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

Idea 1: Map Input to State Transitions

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

Idea 2: Avoiding Redundant Paths

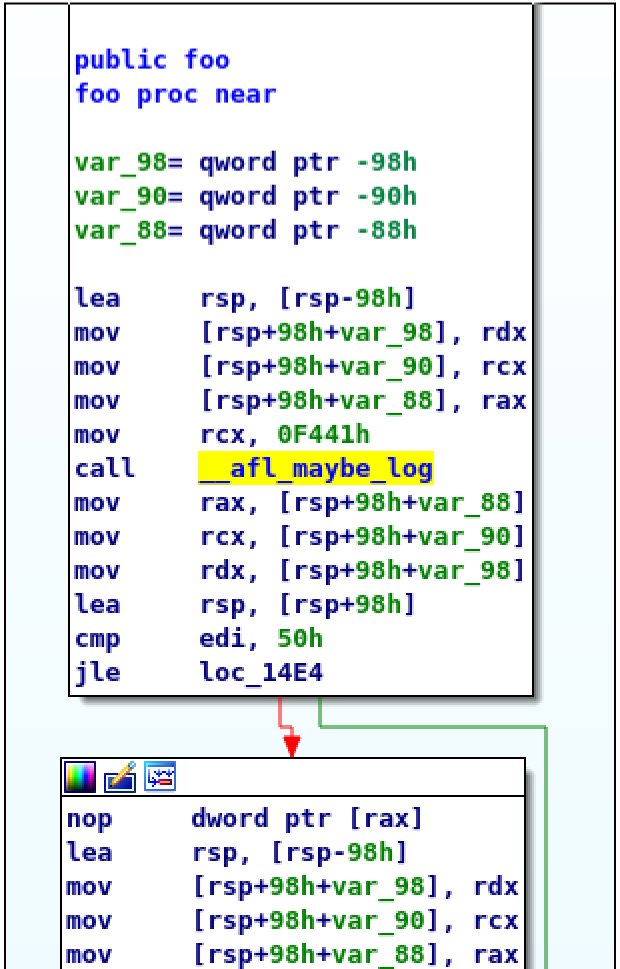
- If you see the duplicated state, throw out
  - e.g., i1 = 1, 2, 3
- If you see the new path, keep it for further exploration
  - e.g., i1 = 81

How to Create Mapping?

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

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

Source Code Instrumentation



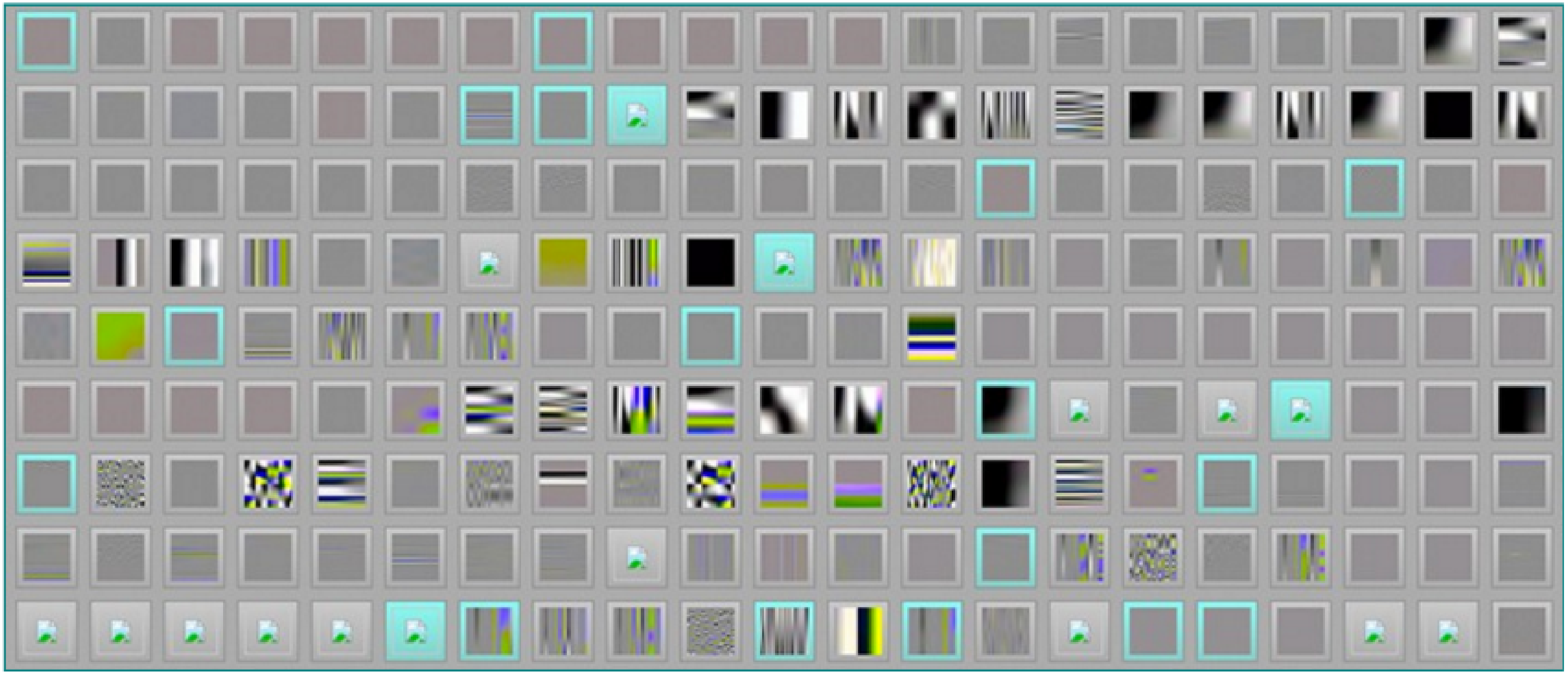
```
public foo
foo proc near

var_98= qword ptr -98h
var_90= qword ptr -90h
var_88= qword ptr -88h

lea     rsp, [rsp-98h]
mov     [rsp+98h+var_98], rdx
mov     [rsp+98h+var_90], rcx
mov     [rsp+98h+var_88], rax
mov     rcx, 0F441h
call    __afl_maybe_log
mov     rax, [rsp+98h+var_88]
mov     rcx, [rsp+98h+var_90]
mov     rdx, [rsp+98h+var_98]
lea     rsp, [rsp+98h]
cmp     edi, 50h
jle     loc_14E4
```

```
nop     dword ptr [rax]
lea     rsp, [rsp-98h]
mov     [rsp+98h+var_98], rdx
mov     [rsp+98h+var_90], rcx
mov     [rsp+98h+var_88], rax
```

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

- Fuzzing with AFL/LibFuzzer
- Fuzzing with Angr/KLEE (optional)

```
$ scp -r lab10@3.95.14.86:/home/lab10 .
$ cd lab10
$ docker build -t fuzzing .
$ docker run --privileged -it fuzzing /bin/bash

or

$ open http://www.dropbox.com/s/6ucge6gp74yjumj/fuzzing.tar.zx
$ unxz fuzzing.tar.xz
$ docker load -i fuzzing.tar
$ docker run --privileged -it fuzzing /bin/bash
```

References

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