Lec10: Fuzzing and Symbolic Execution

Taesoo Kim

Administrivia

- In-class CTF on Dev 1 (24 hours)!
- Submit your team's challenge by Nov 27
- But submit it early for our feedback!
- Or you can brainstorm your challenge during the office hours!

Emphasis on Exploitation (so far)

- In practice, it's more important to ask: how to find bugs?
  - With source code
  - With only binaries

Two Pre-conditions for Exploitation

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    }
```

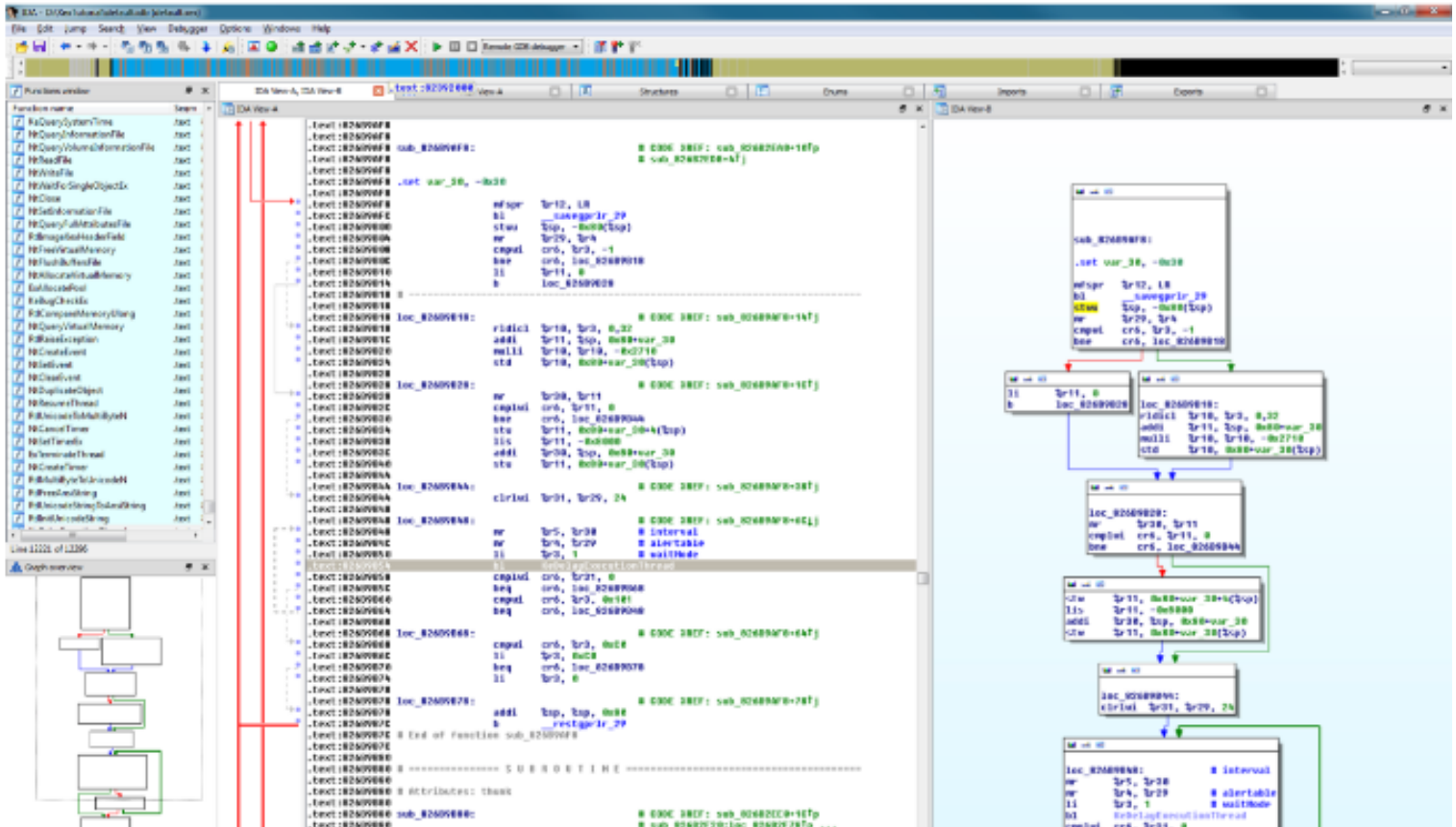
Solution 1: Code Auditing (w/ code)


```
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 }
```

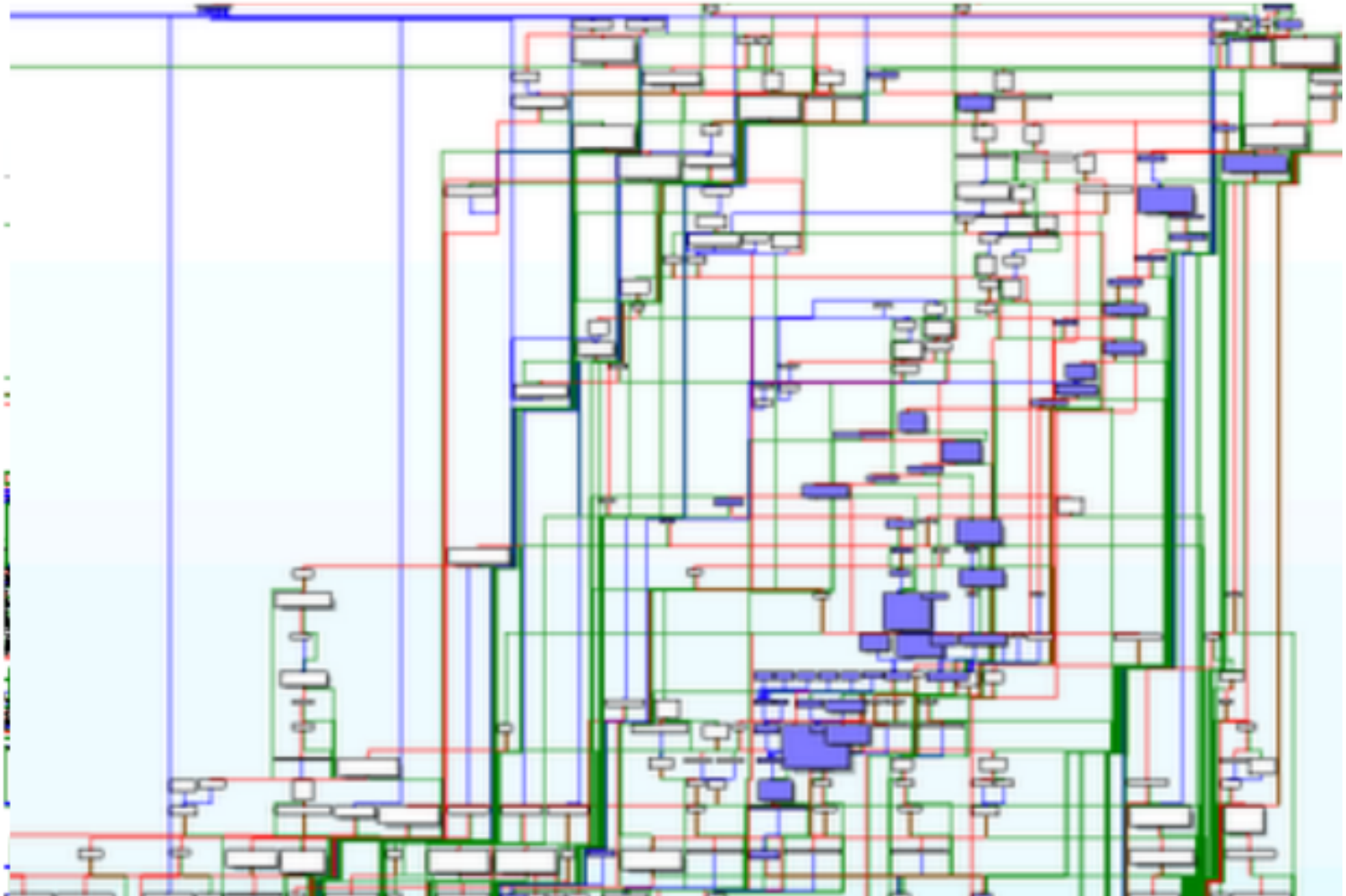
Ref. CVE-2014-1266, Apple sslKeyExchange.c

Solution 2: Static Analysis (on binary)

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



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



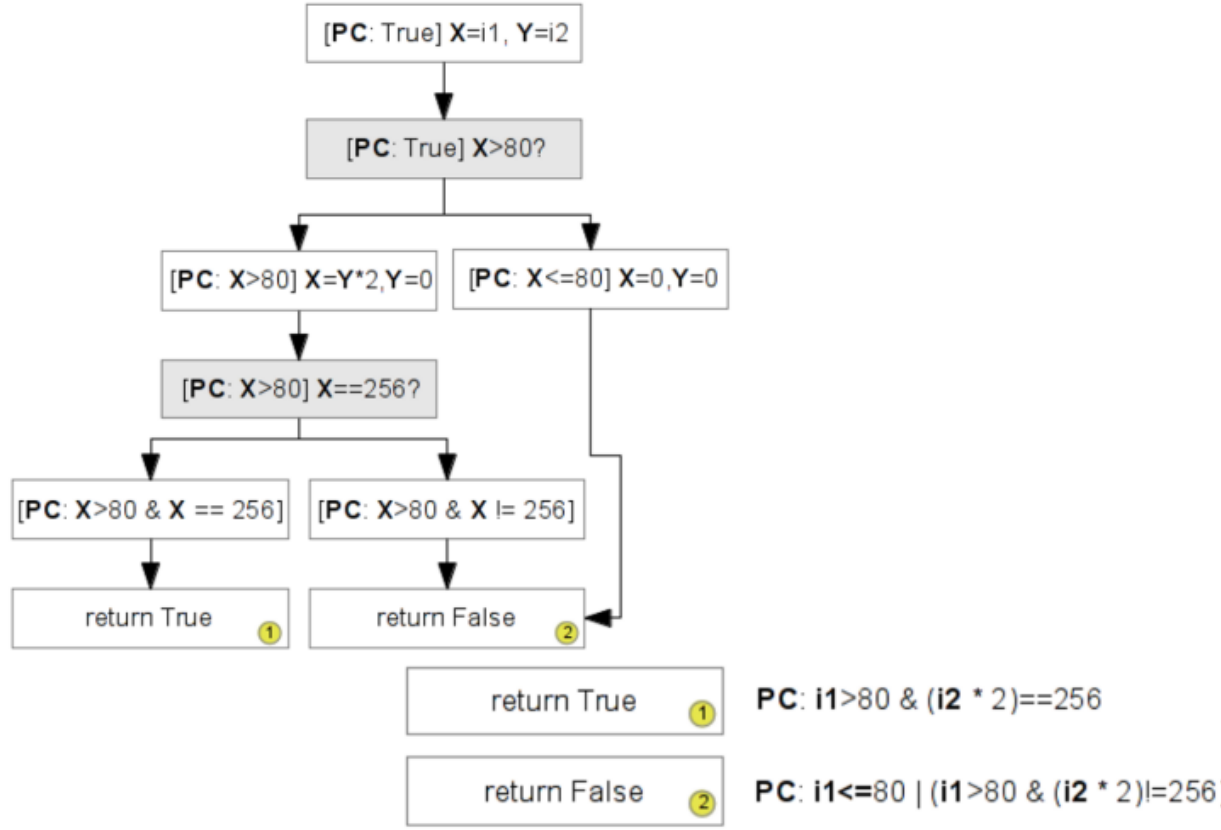
Two Popular Directions

- 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;
}
```



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

- Key ideas
  1. Execute the program with pseudo-random inputs (i.e., input corpus)
  2. Check if the input crashes (i.e., crashing input)
  3. Observe the program execution (i.e., code coverage)
  4. Make a better guess for the input (i.e., input mutation)

→ Reachability is free since we are executing with a concrete input!

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


```
1  // $ clang++ -g -fsanitize=fuzzer ex.cc
2  // $ ./a.out
3  #include <stddef.h>
4  #include <stdint.h>
5
6  extern "C" int
7  LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
8    if (size < 8)
9      return 0;
10
11   int i1, i2;
12   i1 = *(int *)(&data[0]);
13   i2 = *(int *)(&data[4]);
14   foo(i1, i2);
15
16   return 0;
17 }
```

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;
}
```

DEMO: tut01/crackme0x00 (Ref. AFLplusplus)!


```
// -Q: for qemu wrapper
$ afl-fuzz -i input -o output -Q ./crackme0x00
```

→ Q. how to find the password?

Let's Compare Two Approaches

- To crash this example, both need to meet `i1 > 80` and `i2 == 128`:
  - Symbolic execution needs to resolve just two conditions (sounds easy)
  - Fuzzing needs to scan an entire **int** to find a proper `i2` (`i1` is easy)
- However, LibFuzzer/AFL/AFLplusplus are much faster thanks to numerous heuristics!
- Also, testing one fuzzing input is x10k faster!

→ Q. what about `strcmp(buf, "250382")` in `crackme0x00`?

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()

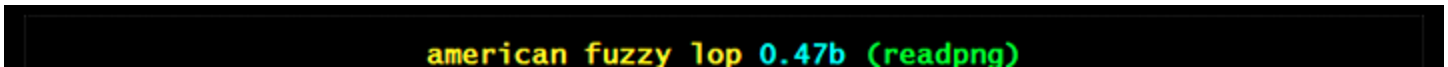
AFL (American Fuzzy Lop)

- VERY well-engineered fuzzer w/ lots of heuristics

american fuzzy lop (2.52b)

My latest book, Practical Doomsday, is now out. Please check it out!

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrum
interesting test cases that trigger new internal states in the targeted binary. This substantially impro
synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-



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[blockIPs % len] (problem? A→B vs B→A)

Idea 1: Map Input to State Transitions

- Input → [IPs] (problem?)
- Input → map[blockIPs % len] (problem? A→B vs B→A)
- Input → map[((prevBlockIP >> 1) ^ curBlockIP) % len] (problem?)

Idea 1: Map Input to State Transitions

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

Idea 1: Map Input to State Transitions

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

→ It also normalize (or bucketize) the map to handle loops better!

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?

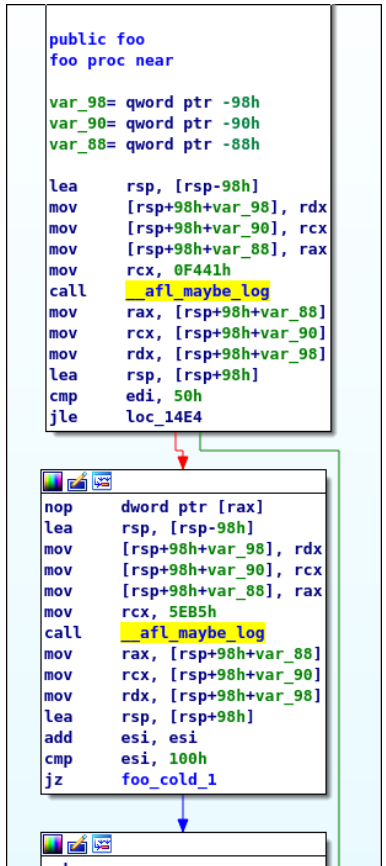
- Instrumentation
  - Source code → by compiler (e.g., gcc, clang)
  - Binary → via binary instrumentation (e.g., QEMU)

```
if (block_address > elf_text_start
    && block_address < elf_text_end) {
  cur_location = (block_address >> 4) ^ (block_address <<
;

  shared_mem[cur_location ^ prev_location] ++;
  prev_location = cur_location >> 1;
}
```

Source Code Instrumentation

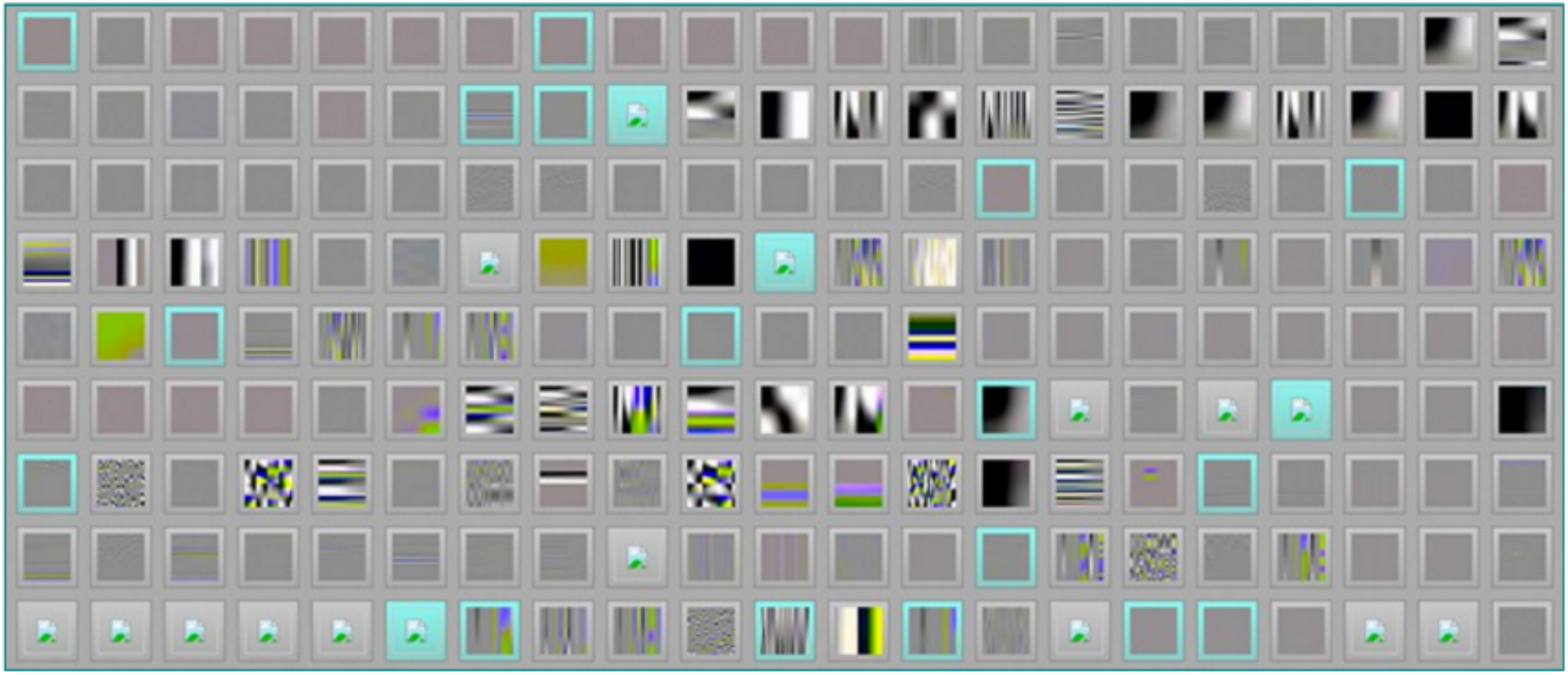

```
1436: mov    rcx,0xf441
143d: call   1520 <__afl_maybe_log>
```



afl_maybe_log()

```
0000000000001440 <__afl_store>:
  # rcx = __afl_cur_loc
  xor    rcx,QWORD PTR [rip+0x2c51]    # rcx = __afl_cur_loc ^
__afl_prev_loc
  xor    QWORD PTR [rip+0x2c4a],rcx    # __afl_prev_loc = __afl_cur_loc
  shr    QWORD PTR [rip+0x2c43],1      # __afl_prev_loc =
__afl_prev_loc >> 1
  add    BYTE PTR [rdx+rcx*1],0x1      # __afl_area_ptr[rcx] += 1
```

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)

```
$ wget https://t.ly/qPOLG
$ unxz fuzzing.tar.xz
$ docker load -i fuzzing.tar
$ docker run --privileged -it fuzzing /bin/bash
```

References

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