William Woodruff
Security Engineer

william.woodruff@trailofbits.com
@8x5clPW2 | github.com/woodruffw
Agenda

● About Trail of Bits
● What is fuzzing?
● Current techniques
● Versus other approaches to automated test generation
  ○ Ongoing work at Trail of Bits
● Research developments
About Trail of Bits

- Information security, founded in 2012
- About 50 employees
  - Half remote, half in NYC office
- Research, assurance, and engineering practices
  - Clientele: DARPA, Facebook, Google, LM, Airbnb
- Open source bounties
What is fuzzing?

- An approach to *automated test generation*
  - Humans are bad at writing tests/thinking about invariants
  - Have the machine write and perform them for us!

- Fuzzing randomly tests the *input space* of a program (or a function)
  - Given a function `basename(char *str)`:
    - What happens when `str=NULL`?
    - ...when `strlen(str) >= MAX_PATH`?
    - ...when `str` isn't valid ASCII/UTF8?
  - Fuzzing can help us cover these cases without having to write specific tests!*

*William Woodruff | Trail of Bits | Fuzzing 101 | 11.14.2018*
Fuzzing from 1000 feet

- Goal 1: Generate lots of inputs, as fast as possible
  - Subgoal: inputs should be diffuse, to avoid duplicating work
- Goal 2: Generate high-quality inputs
  - Inputs are high-quality if they activate novel behavior in the program
- Goal 3: Keep track of inputs that cause crashes, and what kinds of crashes they cause
  - Subgoal: deduplicate crashes that are caused by the same bug but different inputs
  - Subgoal: minimize inputs to make eventual triage/remediation simpler

Which goal(s) do we prioritize?
Fuzzing techniques: black-box

- **Black-box fuzzers operate with no knowledge of the target program**
  - Prioritize goal #1: since we don’t know anything about the target, blast it with as many inputs as possible!

- **Examples:**
  - radamsa, zzuf
  - while true; do program < /dev/urandom; done

- **Pros:**
  - We spend most of our time actually running the program, not doing bookkeeping
  - We don’t need our target’s source code (or even to be on the same machine!)
  - Claim: Quantity compensates for quality in terms of empirical results

- **Cons:**
  - We spend most of our time running the program, but with boring test cases
  - Claim: We get stuck in a local maxima, and discover only “shallow” bugs
black-box strengths and weaknesses

int main(void) {
    int x = getw(stdin);
    if (x > 100) crash();
    else whatever();
}

int main(void) {
    int x = getw(stdin);
    if (x == 0xFEEDFACE) crash();
    else whatever();
}

● Which of these programs is the black-box fuzzer going to crash first?
● What would happen if our crash conditions were more complex, or involved nested conditionals?
  ○ What about multiple distinct crashes, at different levels?
Fuzzing techniques: white-box

- **White-box fuzzers operate with (some) knowledge of the target program**
- **Some potential sources of knowledge:**
  - Source: which functions do I/O, touch memory, rely on undefined behavior?
  - Static analysis: does the program link to libraries that contain known vulnerabilities?
  - Specifications: if the program is specified, can we use the spec for counterexamples?
- **Example: american fuzzy lop*, SAGE**
- **Pros:**
  - We can discover “deep” bugs that random inputs would take much longer to hit
  - Claim: Quality compensates for quantity in terms of empirical results (goal #2)
- **Cons:**
  - We need access to the program’s source or specification
White-box fuzzing: static analysis

What’s (potentially) wrong with these functions?

```c
typedef struct {
    int foo;
    int size;
} blob;

void* copy(blob* obj) {
    blob* dup = malloc(sizeof(obj));
    memcpy(dup, obj, sizeof(obj));
    return dup;
}
```

Which of these functions is interesting to a fuzzer?

```c
typedef struct {
    int foo;
    int size;
} blob;

void* copy(blob* obj) {
    blob* dup = malloc(obj->size);
    memcpy(dup, obj, obj->size);
    return dup;
}
```
Fuzzing techniques: grey-box

- Grey-box fuzzers use *dynamic instrumentation* to gain knowledge of the target program.
- Things we can instrument:
  - Basic blocks/CFG edges: does a given input cause us to execute unique BBs/edges? How does the tuple of all BBs/edges change as we mutate an input?
- Examples: american fuzzy lop*, libFuzzer (LLVM)
- Pros:
  - We can approximate the benefits of white-box fuzzing without needing source code.
  - Claim: With lightweight instrumentation (AFL), we get empirically better/more results than either white or black-box fuzzers.
- Cons:
  - Instrumentation adds runtime overhead, requires that we modify the program being tested (either at compile or runtime), introduces correctness concerns*
Grey-box fuzzing: basic block instrumentation

```
int main(void) {
    int x = getw(stdin);
    int y = 0;

    if (x > 10) {
        y = 1;
        if (x > 100) {
            y = 10;
            if (x > 1000) {
                y = 100;
                crash();
                return 3;
            }
        }
    }
    return 2;
} 
```

Use changes to the activated basic blocks to search the program space:

1. Given an input, can we minimize it and produce the same chain of basic blocks?
2. Once minimized, can we activate new basic blocks along the same chain?
Demo: AFL
How effective is fuzzing?

Extremely! Even black- and grey-box:

- Microsoft SAGE: Hundreds of bugs found in Windows 7 [1]
- AFL: Firefox, Safari, OpenSSL, OpenSSH, Android, glibc, many more [2]

How do black/white/grey box strategies stack up?

How do individual fuzzers compare?

- Not a lot of statistical research, or even standardized evaluation techniques!
Other approaches to test generation

- **Formal verification and countermodeling**
  - Program’s spec might be formally verified, but implementation may not be!
    - Generate test cases that should always fail, according to the formal spec
    - Grammar-based fuzzing

- **Symbolic and “concolic” (symbolic + concrete) execution**
  - Identify input-controlled variables and symbolize them, then do constraint solution
    - Apply an SMT solver like Z3! [5]
      - “Which values of variable x cause the program to take the else branch?”
      - If the input space is small, try all possible values of x!

- **No clear line between fuzzing and many other generation strategies**
  - SAGE is “white-box”, but uses symbolic information for feedback
  - One property: fuzzing implies an element of randomness
Research developments

- **Hardware event-based feedback:**
  - Cache misses, page faults, instruction counts, time spent in kernel space, ...
  - Lower performance impact vs. coverage guidance, better results than black-box

- **Path and depth estimation**
  - “How much of the program’s (interesting) space have we covered so far?”
    - STADS: Software Testing as Species Discovery (Böhme, 2018)

- **CPU and kernel-space fuzzing:**
  - Undocumented isns, ring violations, kernel memory safety violations
  - CPU: sandsifter (Battelle)
  - Kernel: trinity, syzkaller (Google), kernel-fuzzers (Oracle), kAFL
XNU (iOS/macOS) Kernel RCE

https://lgtm.com/blog/apple_xnu_icmp_error_CVE-2018-4407
Ongoing work at Trail of Bits

- **Manticore**: Symbolic execution for x86(_64), ARMv7, EVM bytecode [6]
  - Input generation, instruction tracing
- **DeepState**: Drop-in gtest compatible symbolic execution + fuzzing [7]
- **Echidna**: Grammar-based fuzzing/property testing for EVM [8]
- **Sienna Locomotive**: Coverage-guided black-box fuzzing for Windows
  - Integrated crash triage and vulnerability estimation
- **Toolchain advancements**:
  - **Etheno**: JSON RPC multiplexer for running multiple Ethereum analysis tools [9]
  - **McSema and remill**: Binary lifting (assembly to LLVM) and translation [10, 11]
    - Can be used to make a binary compatible with libFuzzer!
Demo: Manticore
Sources

[2]: http://lcamtuf.coredump.cx/afl/
[5]: https://github.com/Z3Prover/z3
[6]: https://github.com/trailofbits/manticore
[7]: https://github.com/trailofbits/deepstate
[8]: https://github.com/trailofbits/echidna
[9]: https://github.com/trailofbits/etheno
[10]: https://github.com/trailofbits/mcsema
[11]: https://github.com/trailofbits/remill
Additional Resources

- “Super Awesome Fuzzing: Part One”
- https://github.com/CENSUS/choronzon
- https://github.com/MozillaSecurity/dharma
- https://github.com/aoh/blab