Fuzzing: The Age of Vulnerability Discovery

Richard Johnson
Fuzzing IO

Workshop on Offensive Technologies 2023
Agenda

Introduction
The Age of Vulnerability Discovery
Advanced Instrumentation
Improved Input Generation
Reaching New Attack Surface
Conclusions
Richard Johnson
Owner, Fuzzing IO

Advanced Fuzzing and Crash Analysis Training
Contract fuzzing harness and security tool development

Contact
rjohnson@fuzzing.io
@richinseattle
Introduction

• In 2020, I discussed fuzzing in "Lightning in a Bottle" reviewing major milestones that led to "The Fuzzing Renaissance" that was kicked off by the creation of effective greybox mutational parser fuzzing with the release of American Fuzzy Lop.
Introduction

• The Renaissance was a historic period of cultural, artistic, political and economic “rebirth” that attempted to surpass ideas and achievements of antiquity. In many ways this is a great metaphor for the rebirth of interest and expanded capabilities of fuzzing starting in 2013.
Introduction

- Today I’d like to reflect on this impact and show how we, as a community at the intersection of academic and professional research, have participated in this so-called Renaissance which has now led to a new age – The Age of Vulnerability Discovery.
Remembering How We Got Here

- Edge transitions are encoded as tuple and tracked in global map
- Includes coverage and frequency
- Simplified genetic algorithm for continual improvement of input generation
- Uses dictionaries and traditional mutation fuzzing strategies
The Age of Vulnerability Discovery
As of February 2023, ClusterFuzz has found ~27,000 bugs in Google and over 8,900 vulnerabilities and 28,000 bugs across 850 projects integrated with OSS-Fuzz.

- ClusterFuzz README
“Evaluating Fuzz Testing”

GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS - 2018

• “We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments.”
“Evaluating Fuzz Testing”

GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS - 2018

• Until 2018, paper evaluations were ad-hoc and suffered from errors
  • Most papers fail to perform multiple runs and used short timeouts
  • Most papers counted crashes using noisy AFL measurements instead of bugs and coverage
  • Most papers did not consider impact of initial seed selection
  • Most papers vary the target set too widely and may have reported selective results
    • COREUTILS / SPEC2000, SYNTHETIC TEST SUITES (CGC), LAVA
“Evaluating Fuzz Testing”

GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS - 2018

• Evaluation conclusions
  
  • multiple trials with statistical tests to distinguish distributions
  
  • a range of benchmark target programs with known bugs (e.g. LAVA-M, CGC, or old programs with bug fixes)
  
  • measurement of performance in terms of known bugs, rather than heuristics based on AFL coverage profiles or stack hashes
  
  • block or edge coverage can be used as a secondary measure
  
  • a consideration of various (well documented) seed choices including empty seed
  
  • timeouts of at least 24 hours, or else justification for less, with performance plotted over time.
“Evaluating Fuzz Testing”

GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS
FuzzBench is a free resource for researchers to run batches of fuzzing runs using their work in a repeatable and comparable framework with other SoTA fuzzers.
# Benchmarking: Observable Success

## Experiment Summary

We show two different aggregates (cross-benchmark) rankings of fuzzers. The first is based on the average of per-benchmark scores, where the score represents the percentage of the highest reached median code-coverage on a given benchmark (higher value is better). The second ranking shows the average rank of fuzzers, after we rank them on each benchmark according to their median reached code-coverage (lower value is better).

### By Avg. Score

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>Average Normalized Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>aflplusplus</td>
<td>96.39</td>
</tr>
<tr>
<td>libaffl</td>
<td>97.74</td>
</tr>
<tr>
<td>aflplusplus</td>
<td>97.68</td>
</tr>
<tr>
<td>Honggfuzz</td>
<td>96.39</td>
</tr>
<tr>
<td>entropic</td>
<td>93.79</td>
</tr>
<tr>
<td>mopt</td>
<td>86.06</td>
</tr>
<tr>
<td>aflsmart</td>
<td>85.30</td>
</tr>
<tr>
<td>eclipser</td>
<td>85.17</td>
</tr>
<tr>
<td>afl</td>
<td>85.06</td>
</tr>
<tr>
<td>aflast</td>
<td>83.63</td>
</tr>
<tr>
<td>libfuzzer</td>
<td>81.99</td>
</tr>
<tr>
<td>fairfuzz</td>
<td>81.13</td>
</tr>
<tr>
<td>centipede</td>
<td>70.48</td>
</tr>
</tbody>
</table>

### By Avg. Rank

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>Average Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>aflplusplus</td>
<td>2.96</td>
</tr>
<tr>
<td>aflplusplus</td>
<td>3.09</td>
</tr>
<tr>
<td>libaffl</td>
<td>5.00</td>
</tr>
<tr>
<td>Honggfuzz</td>
<td>5.26</td>
</tr>
<tr>
<td>entropic</td>
<td>5.74</td>
</tr>
<tr>
<td>eclipser</td>
<td>6.52</td>
</tr>
<tr>
<td>aflsmart</td>
<td>6.70</td>
</tr>
<tr>
<td>afl</td>
<td>7.52</td>
</tr>
<tr>
<td>mopt</td>
<td>7.70</td>
</tr>
<tr>
<td>libfuzzer</td>
<td>8.91</td>
</tr>
<tr>
<td>centipede</td>
<td>9.78</td>
</tr>
<tr>
<td>aflast</td>
<td>9.87</td>
</tr>
<tr>
<td>fairfuzz</td>
<td>9.96</td>
</tr>
<tr>
<td>centipede</td>
<td>10.00</td>
</tr>
</tbody>
</table>

---

**FUZZBENCH**
Benchmarking: Observable Success

freetype2_ftfuzzer summary

Ranking by median reached code coverage

Reached code coverage distribution

Fuzzer (highest median coverage on the left)
Benchmarking: Observable Success

**FUZZBENCH – COVERAGE OVER TIME VIEW**

**Mean code coverage growth over time**

freetype2_ftfuzzer (23h, at least 20 trials/fuzzer)

- The error bands show the 95% confidence interval around the mean code coverage.
Benchmarked: Observable Success

Fuzzbench – Log Time View Differentiator for Longer Fuzzing Campaigns

Mean code coverage growth over time
freetype2-rtfuzzer (23h, at least 20 trials/fuzzer)

* The error bands show the 95% confidence interval around the mean code coverage.
Benchmarking: Observable Success

FUZBBENCH – COMPARING PERFORMANCE OF FUZZER VS PARSER
Fuzz Introspector

LIKE GCOV REPORTS BUT FOR FUZZING!

• Fuzz introspector is a tool to help fuzzer developers to get an understanding of their fuzzer’s performance and identify any potential blockers

• Integrated with OSS-FUZZ
# Fuzz Introspector

**LIKE GCOV REPORTS BUT FOR FUZZING!**

## Project functions overview

<table>
<thead>
<tr>
<th>Func name</th>
<th>Git URL</th>
<th>Functions filename</th>
<th>Arg count</th>
<th>Arg #</th>
<th>Function reach depth</th>
<th>Fuzzers hit count</th>
<th>I Count</th>
<th>BB Count</th>
<th>Cyclomatic complexity</th>
<th>Functions reached</th>
<th>Reached by functions</th>
<th>Accumulated cyclomatic complexity</th>
<th>Undiscovered complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib/tools</td>
<td>LNK</td>
<td>executable</td>
<td>2</td>
<td>[vec *, &quot;vec&quot;]</td>
<td>17</td>
<td>0</td>
<td>65</td>
<td>16</td>
<td>3</td>
<td>1093</td>
<td>0</td>
<td>10283</td>
<td>3</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>1</td>
<td>[struct.HFile *]</td>
<td>17</td>
<td>1</td>
<td>74</td>
<td>22</td>
<td>0</td>
<td>805</td>
<td>1</td>
<td>9403</td>
<td>0</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>1</td>
<td>[struct.HFile *]</td>
<td>17</td>
<td>1</td>
<td>84</td>
<td>23</td>
<td>0</td>
<td>861</td>
<td>1</td>
<td>9505</td>
<td>2</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>2</td>
<td>[vec *, &quot;vec&quot;]</td>
<td>10</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>756</td>
<td>0</td>
<td>7066</td>
<td>196</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>2</td>
<td>[vec *, &quot;vec&quot;]</td>
<td>18</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>735</td>
<td>0</td>
<td>7064</td>
<td>194</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>3</td>
<td>[vec *, &quot;vec&quot;, &quot;vec&quot;]</td>
<td>18</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>735</td>
<td>1</td>
<td>7064</td>
<td>194</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>4</td>
<td>[vec *, &quot;vec&quot;, &quot;vec&quot;, &quot;vec&quot;]</td>
<td>17</td>
<td>0</td>
<td>97</td>
<td>22</td>
<td>0</td>
<td>724</td>
<td>0</td>
<td>7042</td>
<td>192</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>3</td>
<td>[struct.HFile *, &quot;must.hFile&quot;, &quot;must.hFile&quot;, &quot;must.hFile&quot;]</td>
<td>18</td>
<td>1</td>
<td>486</td>
<td>73</td>
<td>27</td>
<td>654</td>
<td>2</td>
<td>6059</td>
<td>10</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>1</td>
<td>[struct.HFile *]</td>
<td>18</td>
<td>1</td>
<td>73</td>
<td>6</td>
<td>7</td>
<td>676</td>
<td>3</td>
<td>6716</td>
<td>10</td>
</tr>
<tr>
<td>libc</td>
<td>LNK</td>
<td>executable</td>
<td>1</td>
<td>[struct.HFile *]</td>
<td>17</td>
<td>1</td>
<td>179</td>
<td>23</td>
<td>7</td>
<td>625</td>
<td>4</td>
<td>6394</td>
<td>10</td>
</tr>
</tbody>
</table>

Showing 1 to 10 of 2,001 entries
Fuzz Introspector

LIKE GCOV REPORTS BUT FOR FUZZING!

- **Fuzz Introspector** is a tool to help fuzzer developers to get an understanding of their fuzzer’s performance and identify any potential blockers
- Integrated with OSS-FUZZ
**Fuzzer Challenges**

**THE CRUCIBLE FOR FUZZING AND SOLVER TOOLS**

• For those who are looking to ensure their new fuzzer is able to solve difficult challenges encountered in the field, AFL++ team has created the **Fuzzer Challenges** project

**Testcases:**

- **test-u8** - several chained 8 bits checks
- **test-u16** - several chained 16 bits checks
- **test-u32** - several chained 32 bits checks
- **test-u64** - several chained 64 bits checks
- **test-u128** - several chained 128 bits checks
- **test-u32-cmp** - several chained 32 bit lesser/greater checks
- **text-extint** - llvm ExtInt() tests
- **test-float** - several chained float checks
- **test-double** - several chained double checks
- **test-longdouble** - several chained long double checks
- **test-memcmp** - several chained memcmp checks
- **test-strcmp** - several chained strncasecmp checks
- **test-transform** - different transforming string checks
- **test-crc32** - several chained crc32 checks
Fuzzer Challenges

The Crucible for Fuzzing and Solver Tools

- For those who are looking to ensure their new fuzzer is able to solve difficult challenges encountered in the field, AFL++ team has created the **Fuzzer Challenges** project.
Bugs Exist in New Code

About half of the exploitable vulnerabilities found in Chrome in 2019 were found in code less than a year old.

40% of OSS-Fuzz bug finds were new check-ins before code shipped to stable.

Why? Specifications rapidly change for the infrastructure that runs our businesses and code churn is high.

The best defense is a good offense, which in the case for code owners is integrating continuous fuzzing in the build chain.
Building the Fuzz Chain

FUZZING IN THE COMPILER

- LibFuzzer is now part of Clang/LLVM and Visual Studio
- Go 1.18 has integrated libFuzzer
- Rust has cargo-fuzz for integrated LibFuzzer harnesses. Supports structured fuzzing using the "Arbitrary" crate
Building the Fuzz Chain

FUZZING IN THE RUNTIME

• Jazzer – coverage guided fuzzing for JVM from Code Intelligence

• Wraps native LibFuzzer to fuzz Java, Kotlin, Scala, Clojure

• User creates a library with native implementation of LLVMFuzzerTestOneInput callback and calls Java methods via JNI

• Comes with special sanitizers for detecting injection bugs, user controlled class deserialization, etc
Building the Fuzz Chain

JAZZER FUZZING HARNESS

• Support for Jazzzer fuzz targets is now in OSS-FUZZ and the JUnit testing framework

```java
import com.code_intelligence.jazzer.api.FuzzedDataProvider;

import com.google.json.JsonSanitizer;

public class DenylistFuzzer {
    public static void fuzzerTestOneInput(FuzzedDataProvider data) {
        String input = data.consumeRemainingAsString();
        String output;
        try {
            output = JsonSanitizer.sanitize(input, 10);
        } catch (ArrayIndexOutOfBoundsException e) {
            // ArrayIndexOutOfBoundsException is expected if nesting depth is exceeded.
            return;
        }
        // See https://github.com/OWASP/json-sanitizer#output.
        if (output.contains("<script") || output.contains("<xscript")
            || output.contains("<![CDATA[") || output.contains("]]>")) {
            System.err.println("input : " + input);
            System.err.println("output : " + output);
            throw new IllegalStateException("Output contains forbidden substring");
        }
    }
}
```
Building the Fuzz Chain

JAZZER SUPPORTS CUSTOM SANITIZERS

```java
class PathTraversalSanitizer {
    @MethodHook(type = HookType.AFTER, targetClassName = "java.io.File",
                targetMethod = "<init>", targetMethodDescriptor = "(Ljava/lang/String;)")
    public static void fileConstructorHook(MethodHandle method, Object thisObject, Object[] arguments, int hookId, Object returnValue) {
        File file = (File) thisObject;
        String pathname = (String) arguments[0];
        try {
            // Check whether the canonical path of 'file' lies inside a known list of allowed paths.
            if (!file.getCanonicalPath().startsWith("/expected/path")) {
                // If not, throw a distinctive exception that is reported by Jazzer.
                throw new PotentialPathTraversalException();
            }
        } catch (IOException e) {
            //
        }
    }
```
Fuzzing in the Cloud

SECURITY IS A SHARED RESPONSIBILITY AND GOOGLE IS HERE TO HELP!

• OSS-FUZZ is an amazing resource. Hundreds of opensource projects receive free fuzzing with libFuzzer and AFL++ engines (or Jazzer, go fuzz, cargo fuzz, etc)

• Google is the top contributor to opensource fuzzing tooling and free compute to perform fuzz testing in the cloud

• OSS-Fuzz also offers a bounty program for writing fuzzing harnesses including rewards for security bugs found by contributed harnesses
# AFL++ - still the best general fuzzer

**AFL++ - Marc Heuse, Heiko Eissfeldt, Andrea Fioraldi, Dominik Maier**

- Persistent fuzzing, libFuzzer harness compat, inmem testcases
- LAF-Intel / CmpCov
- CmpLog / Redqueen
- Higher performance binary fuzzing
- Nyx, QEMU, Unicorn, Frida, QDBI

```plaintext
<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>process timing</td>
<td>run time: 0 days, 0 hrs, 1 min, 0 sec</td>
</tr>
<tr>
<td></td>
<td>last new find: 0 days, 0 hrs, 0 min, 0 sec</td>
</tr>
<tr>
<td></td>
<td>last saved crash: 0 days, 0 hrs, 0 min, 5 sec</td>
</tr>
<tr>
<td></td>
<td>last saved hang: none seen yet</td>
</tr>
<tr>
<td>cycle progress</td>
<td>now processing: 227.0 (32.4%)</td>
</tr>
<tr>
<td></td>
<td>runs timed out: 0 (0.00%)</td>
</tr>
<tr>
<td>stage progress</td>
<td>now trying: havoc</td>
</tr>
<tr>
<td></td>
<td>stage execs: 3450/12.8k (26.95%)</td>
</tr>
<tr>
<td></td>
<td>total execs: 152k</td>
</tr>
<tr>
<td></td>
<td>exec speed: 2547/sec</td>
</tr>
<tr>
<td>fuzzing strategy yields</td>
<td>bit flips: disabled (default, enable with -D)</td>
</tr>
<tr>
<td></td>
<td>byte flips: disabled (default, enable with -D)</td>
</tr>
<tr>
<td></td>
<td>arithmetics: disabled (default, enable with -D)</td>
</tr>
<tr>
<td></td>
<td>known ints: disabled (default, enable with -D)</td>
</tr>
<tr>
<td></td>
<td>dictionary: n/a</td>
</tr>
<tr>
<td></td>
<td>havoc/splice: 522/119k, 174/20.8k</td>
</tr>
<tr>
<td></td>
<td>py/custom/rq: unused, unused, unused, unused</td>
</tr>
<tr>
<td></td>
<td>trim/eff: 46.45%/4163, disabled</td>
</tr>
</tbody>
</table>
```

```plaintext
<table>
<thead>
<tr>
<th>Results</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>overall results</td>
<td>cyc.es done: 0</td>
</tr>
<tr>
<td></td>
<td>corpus count: 700</td>
</tr>
<tr>
<td></td>
<td>saved crashes: 5</td>
</tr>
<tr>
<td></td>
<td>saved hangs: 0</td>
</tr>
<tr>
<td>map coverage</td>
<td>map density: 0.97% / 10.98%</td>
</tr>
<tr>
<td></td>
<td>count coverage: 1.85 bits/tuple</td>
</tr>
<tr>
<td>findings in depth</td>
<td>favored items: 421 (60.14%)</td>
</tr>
<tr>
<td></td>
<td>new edges on: 506 (72.29%)</td>
</tr>
<tr>
<td></td>
<td>total crashes: 76 (5 saved)</td>
</tr>
<tr>
<td></td>
<td>total timeouts: 0 (0 saved)</td>
</tr>
<tr>
<td>item geometry</td>
<td>levels: 7</td>
</tr>
<tr>
<td></td>
<td>pending: 674</td>
</tr>
<tr>
<td></td>
<td>pend fav: 462</td>
</tr>
<tr>
<td></td>
<td>own finds: 698</td>
</tr>
<tr>
<td></td>
<td>imported: 0</td>
</tr>
<tr>
<td></td>
<td>stability: 100.00%</td>
</tr>
</tbody>
</table>
```

[cpu000: 75%]
AFL++ - still the best general fuzzer

- Persistent fuzzing, libFuzzer harness compat, inmem testcases
- LAF-Intel / CmpCov
- CmpLog / Redqueen
- Higher performance binary fuzzing
- Nyx, QEMU, Unicorn, Frida, QDBI
LibAFL gives you many of the benefits of an off-the-shelf fuzzer, while being completely customizable. Some highlight features currently include:

- **fast**: We do everything we can at compile time, keeping runtime overhead minimal. Users reach 120k execs/sec in frida-mode on a phone (using all cores).
- **scalable**: Low Level Message Passing, LLMP for short, allows LibAFL to scale almost linearly over cores, and via TCP to multiple machines.
- **adaptable**: You can replace each part of LibAFL. For example, `BytesInput` is just one potential form input; feel free to add an AST-based input for structured fuzzing, and more.
- **multi platform**: LibAFL was confirmed to work on Windows, MacOS, Linux, and Android on x86_64 and aarch64. LibAFL can be built in `no_std` mode to inject LibAFL into obscure targets like embedded devices and hypervisors.
- **bring your own target**: We support binary-only modes, like Frida-Mode, as well as multiple compilation passes for sourced-based instrumentation. Of course it’s easy to add custom instrumentation backends.
## LibAFL: Modular Fuzzer Design

| corpus | Corpuses contain the testcases, either in memory, on disk, or somewhere else. |
| events | Eventmanager manages all events that go to other instances of the fuzzer. |
| executors | Executors take input, and run it in the target. |
| feedbacks | The feedbacks reduce observer state after each run to a single `is_interesting`-value. If a testcase is interesting, it may be added to a Corpus. |
| fuzzer | The Fuzzer is the main struct for a fuzz campaign. |
| generators | Generators may generate bytes or, in general, data, for inputs. |
| inputs | Inputs are the actual contents sent to a target for each execution. |
| monitors | Keep stats, and display them to the user. Usually used in a broker, or main node, of some sort. |
| mutators | Mutators mutate input during fuzzing. |
| observers | Observers give insights about runs of a target, such as coverage, timing, stack depth, and more. |
| prelude | The purpose of this module is to alleviate imports of many components by adding a glob import. |
| schedulers | Schedule the access to the Corpus. |
| stages | A Stage is a technique used during fuzzing, working on one `crate::corpus::Corpus` entry, and potentially altering it or creating new entries. A well-known Stage, for example, is the mutational stage, running multiple `crate::mutators::Mutator` s against a `crate::corpus::Testcase`, potentially storing new ones, according to `crate::feedbacks::Feedback`. Other stages may enrich `crate::corpus::Testcase` s with metadata. |
| state | The fuzzer, and state are the core pieces of every good fuzzer |
LibAFL: Modular Fuzzer Design

• LibAFL is a modular fuzzing system that provides mix and match components for instrumentation, data generation, and runtime configuration

• Many backends are supported - LLVM SanitizerCoverage, Frida, QEMU user and system, TinyInst, and more

• In the near future, AFL++ will be a frontend for LibAFL

• See paper from CCS'22 “LibAFL: A Framework to Build Modular and Reusable Fuzzers"
Advanced Instrumentation
Fuzzers need to know when a fault has occurred. Memory corruption makes this trivial but other bug classes may require specific checkers.
Sanitizers for Sanity

HEARTBLEED IMPACTED 66% OF WWW SERVERS

- Heartbleed taught us that fuzzers and checkers work in harmony to find deep bugs
- Today’s best practices for fuzzing will include at least one worker node with sanitizers enabled.
- AddressSanitizer is only one of many
Sanitizers for Sanity

ADDRESS SANITIZER LIFTED THE SHADOW

• Use after free (dangling ptr deref)
• Heap buffer overflow
• Stack buffer overflow
• Global buffer overflow
• Use after return
• Use after scope
• Initialization order bugs
• Memory leaks
Sanitizers for Sanity

LLVM SANITIZER FAMILY

• The LLVM Sanitizer family has grown to include:
  • **Undefined Behavior Sanitizer**
  • Hardware-assisted Address Sanitizer
  • Thread Sanitizer (races and deadlocks)
  • Memory Sanitizer (uninitialized memory)

UNDEFINED BEHAVIOR SANITIZER

• Array subscript out of bounds, where the bounds can be statically determined
• Bitwise shifts that are out of bounds for their data type
• Dereferencing misaligned or null pointers
• Signed integer overflow
• Conversion to, from, or between floating-point types which would overflow the destination
Sanitizers for Sanity

**LLVM Sanitizer Family**

- The LLVM Sanitizer family has grown to include:
  - Undefined Behavior Sanitizer
  - **Hardware-assisted Address Sanitizer**
  - Thread Sanitizer (races and deadlocks)
  - Memory Sanitizer (uninitialized memory)

**Hardware-Assisted Address Sanitizer**

- ASAN uses bitmask for shadow memory and redzone for overflows, and quarantines for UAF
- HWASAN uses hardware memory tagging
- AArch64 uses top-byte-ignore (TBI) pointer tagging
- x86_64 will use Linear Address Masking (LAM) or page aliasing. Linear addresses use 48bit (L4) or 57bit (L5) addresses, leaving room for tagging.
- Some detections become probabilistic
Sanitizers for Sanity

LLVM SANITIZER FAMILY

- Sanitizers are now supported in:
- Clang 3.1+, GCC 4.8+
- Visual Studio 2019+
- LibFuzzer, AFL++, honggfuzz
- Kernels: KASAN
- Emulators: QEMU, Frida
Unconstrained Progress

WHEN YOU CAN’T BEAT THEM, UNJOIN THEM

• Code logic is structured as a series of constraints or comparisons that decide which code path to follow

• For a long time, it was assumed constraint solvers were the solution

• Constraint solvers require **heavy analysis** including **lifting** target **code** to intermediate languages to **extract the semantics** of each instruction. Solving requires converting the series of instructions to Boolean algebra in a format compatible with SMTLIB2 or domain specific encoding – **computationally expensive** for quick iteration
Unconstrained Progress

**WHEN YOU CAN’T BEAT THEM, UNJOIN THEM**

- LAF-Intel was the first fuzzing technology to approach the problem of constraints through code transformation.

- Multibyte comparisons are rewritten as a series of sequential single-byte comparisons.

- 32bit: $1/(2^{32}) \rightarrow 1/(2^8 \times 4)$. $4,294,967,296$ vs $1,024$.

- CmpCov is the updated version of LAF-Intel seen in fuzzers today.
Focused Mutation

Cheap Alternative to Taint Tracking and Symbolic Execution

- RedQueen was the first fuzzing technology to efficiently correlate values observed in comparisons as being derived directly from the input.

- Solves magic bytes and checksums by spending time fuzzing input byte offsets seen in the comparison instructions without needing taint tracking or symbolic execution.

- CmpLog is the updated implementation seen in fuzzing tools today.
Improved Input Generation
“Grammar mutators are able to trigger deep bugs that are near impossible to find with code coverage guided fuzzers”
vanHauser, Berlin 2022
Grammars

CONTEXT FREE GRAMMARS

• DEFENSICS
• Domato
• Nautilus
• GrammaTron
• AFL++ ATNWalk
• Fzero / F1 Fuzzer

STRUCTURED / API GRAMMARS

• Peach (AFLSmart)
• libprotobuf-mutator
• Fuzzili
• SyzLang
• KernelFuzz
Searching for Approximate Grammar

Fuzzing is a cheap grammar extraction

- It has long been known that high quality grammars can accurately model syntax and interaction logic to perform testing.
- Grammars are tedious to produce, and implementations may deviate from the specification.
- The simplest improvement to bit flipping is to use a dictionary of tokens.
- **DICT2FILE** is a LLVM plugin that ships with AFL++ that can extract constant comparisons at compile time. Generating a dictionary can be as simple as that!
Grammars are a Browser’s Best Friend

DOM FUZZING WITH DOMATO - IVAN FRATRIC / GOOGLE

- A few simple components can compromise a browser...
- grammar.py - Generic BNF parser and generative fuzzer
- *.txt, *.html - HTML/CSS/js grammar and .html template
- generator.py - Test case generator for DOM fuzzing
Grammars are a Browser’s Best Friend

Domato uses an eBNF inspired context-free grammar

$ python generator.py --output_dir out --no_of_files 3
Running on ClusterFuzz
Output directory: out
Number of samples: 3
Writing a sample to out\fuzz-0.html 08/30/2022 07:40 PM 455,255 fuzz-0.html
Writing a sample to out\fuzz-1.html 08/30/2022 07:40 PM 463,901 fuzz-1.html
Writing a sample to out\fuzz-2.html 08/30/2022 07:40 PM 460,684 fuzz-2.html
Grammars are a Browser’s Best Friend

Domato uses an eBNF inspired context-free grammar

- `<cssproperty_z-index>` = `<fuzzint>`
- `<cssproperty_z-index>` = `auto`
- `<cssproperty_z-index>` = `inherit`
- `<cssproperty_zoom>` = `<percentage>%`
- `<cssproperty_zoom>` = `<fuzzint>`
- `<cssproperty_zoom>` = `<float>`
- `<cssproperty_zoom>` = `auto`
- `<cssproperty_zoom>` = `reset`
Grammars are a Browser’s Best Friend

DOM FUZZING WITH DOMATO - IVAN FRATRIC / GOOGLE

To inject lines of code into the grammar, a special syntax is used

```
!begin lines
<new element> = document.getElementById("<string min=97 max=122>";
<element>.doSomething();
!end lines
```

This will expand types as well as generate new variable names
Grammars are a Browser’s Best Friend

DOM FUZZING WITH DOMATO - IVAN FRATRIC / GOOGLE

To inject lines of code into the generator, another syntax is used

!begin function createbody
    n = int(context['size'])
    ret_val = 'a' * n
!end function

<body> = <call function=createbody>
Grammars are a Browser’s Best Friend

• To fuzz the Chrome V8 JavaScript JIT, Samuel created a custom intermediate language that represents dependencies and mutations
• Fuzzilli has found vulnerabilities in every JavaScript engine with a JIT
• The key to Fuzzilli is that it can lift JavaScript source from disk to IL and instrument the AST prior to the JIT engine having access to the code
• JIT vulns have been the primary attack vector on browsers for 5 years
• An Intermediate Language fuzzer is a form of grammar driven mutator
Grammars are a Browser’s Best Friend

**LIBPROTOBUF-MUTATOR VS THE CHROME SANDBOX**

- Ned Williamson showed that APIs can be encoded as ProtoBuf definitions and found Chrome sandbox vulns in “Attacking Chrome IPC” (2019)

- LibProtoBufMutator allows users to describe basic types and structures and use metaprogramming for fuzzing

```protobuf
massage MyFormat {
  oneof a_or_b {
    MessageA message_a = 1;
    MessageB message_b = 2;
  }
  required MessageC message_c = 3;
}
```
Grammars for APIs

LIBPROTobuf-MUTATOR VS APIS

- **libprotobuf-mutator** is generalizable to any type of API fuzzing and is integrated directly with LibFuzzer.
- **OSS-FUZZ** has example targets using libprotobuf-mutator to deserialize mutated buffers.

```cpp
// Needed since we use getenv().
#include <stdlib.h>

// Needed since we use std::cout.
#include <iostream>

#include "testing/libfuzzer/proto/lpm_interface.h"

// Assuming the .proto file is path/to/your/proto_file/my_format.proto.
#include "path/to/your/proto_file/my_format.pb.h"

// Put your conversion code here (if needed) and then pass the result to
// your fuzzing code (or just pass "my_format", if your target accepts
// protobufs).

DEFINE_PROTO_FUZZER(const MyFuzzer::MyFormat& my_proto_format) {
  // Convert your protobuf to whatever format your targeted code accepts
  // if it doesn't accept protobufs.
  std::string native_input = convert_to_native_input(my_proto_format);

  // You should provide a way to easily retrieve the native input for
  // a given protobuf input. This is useful for debugging and for seeing
  // the inputs that cause targeted_function to crash (which is the reason we
  // are here!). Note how this is done before targeted_function is called
  // since we can't print after the program has crashed.
  if (getenv("LPM_DUMP_NATIVE_INPUT"))
    std::cout << native_input << std::endl;

  // Now test your targeted code using the converted protobuf input.
  targeted_function(native_input);
}
Grammars for Syscalls

- System Calls are “just” a privileged API between userland and the kernel

Syscall Description Language (syzlang)

```plaintext
open(file ptr[in, filename], flags flags[open_flags]) fd
data(fd fd, buf ptr[out, array[int8]], count bytesize[buf])
close(fd fd)

open_flags = O_RDONLY, O_WRONLY, O_RDWR, O_APPEND
```
When in Doubt, Math it Out

HIGHLY SELECTIVE APPLICATION OF HIGHLY SOPHISTICATED TECH WINS IN THE END

• FuzzBench and other empirical data has deflated the hype around symbolic execution, taint analysis, and other complex analysis

• In a compute constrained environment, it's typically more efficient to use naive high throughput approaches like fuzzing

• Selective use of graph analysis to locate “frontiers” that are guarded by complex constraints can identify when to apply more complex approaches to code exploration
When in Doubt, Math it Out

SYMCC / SYMQEMU

- SYMCC / SYMQEMU are SoTA implementations of concolic execution that can work on large code bases
- Integration exists in AFL++ via custom mutators
- Less than 1% of edges require symex
When in Doubt, Math it Out

SYMCC / SYMQEMU

- SYMCC is based on an LLVM instrumentation and can be efficient for source-based targets.

- However, if source is available then other compiler plugins can extract grammars which will be more efficient than using constraint solving.

- SymQemu is a derivative of the SymCC implementation and works on unmodified binaries and has been shown to be more robust than alternatives.
When in Doubt, Math it Out

**SYMSAN**

- Builds on top of the built in DataFlowSanitizer in LLVM
- Low runtime state management overhead due to efficient storage of high level types
- 2x faster than SymCC/SymQEMU
- Minimal memory overhead

**TRITON-DSE**

- High level API for Triton from Quarkslab uses a unique internal graph representation that is highly efficient
- Available as an extension to AFL++ but not proven to be more capable than SymQEMU
When in Doubt, Math it Out

TRITON-DSE

• “TritonDSE goal is to provide higher-level primitives than Triton. Triton is a low-level framework where one have to provide manually all instructions to be executed symbolically.”

Loader mechanism (based on LIEF, cle or custom ones)
Memory segmentation
Coverage strategies (block, edge, path)
Pointer coverage
Automatic input injection on stdin, argv
Input replay with QBDI
input scheduling (customizable)
sanitizer mechanism
basic heap allocator
some libc symbolic stubs
Reaching New Attack Surface
2019: 2046 Linux kernel bugs found & fixed by SyzKaller (BlueHat IL 2020)

2023: 4535 fixed

Keeping average of 2-3/day

SyzBot Dashboard
https://syzkaller.appspot.com/upstream
Fuzzing Windows

Fuzzing on Windows has many architectural considerations

- No forking – Address Space Randomization causes inconsistencies across runs
- Multi-threaded by default
- Locked access to system resources
- An array of IPC options
- Async behavior via COM interfaces
- Handle based access to subsystems that need to be managed appropriately across iterations
Fuzzing Windows with DBI

Many code coverage engines have propagated to Windows

• The solution for fuzzing windows is either to embed fuzzing in the process using a native JIT for x64->x64 or to use snapshot-oriented fuzzing based on hypervisors or emulation

• Dynamic Binary Instrumentation achieves the first option
  • DynamoRIO
  • Frida
  • Custom JIT via Debugger APIs like TinylInst
Snapshot Fuzzing

A SNAPSHOT IN TIME UNLOCKS THE FUTURE POTENTIAL

• Snapshot fuzzing solves the problems with persistent in-memory fuzzing faced by solutions like LibFuzzer, AFL++ Persistence, DBI based fuzzing, stateful exploration, and many other challenges.

• For the uninitiated, snapshot fuzzing restores state between iterations.

• Each modified page after a checkpoint defined by the fuzzer or initial state saved will be restored from the saved or initial state.

• Solves all problems with global state management, process creation, etc.
Snapshot Fuzzing

A Snapshot in Time Unlocks the Future Potential

• Snapshot fuzzing is not easy-mode and requires expert users
• Workflows are more challenging; initialization of the fuzzing loop is from a runtime state requiring configuration of virtual or hypervisor CPU state
• Snapshots can be generated incrementally to reliably handle and target state transitions
• Snapshots can be migrated to scalable environments
Fuzzing Anything With Emulators

**EMULATE CODE YOU CAN ISOLATE FROM HARDWARE I/O**

- On the graph of performance vs targetability, emulators are the maximum for targetability
- Given enough effort anything can be simulated (ref: “the simulation”)
- In reality, there is an ideal compromise between instrumentation of an “in situ” execution environment and emulation
- Emulation can limit access to external devices, environmental config, etc
SO YOU WANT TO FUZZ A WINDOWS KERNEL DRIVER

• Fuzzing closed source bare metal OS environments is challenging
• The options are to take full system memory snapshots, which will only have the kernel and a single process mapped into memory or to develop a custom hypervisor
• WTF implements the first approach – fuzzing a kernel via a kernel memory snapshot
• My team expanded this to support any OS that runs in QEMU
WTF Fuzzer

SO YOU WANT TO FUZZ A WINDOWS KERNEL DRIVER

• In the “simple” case, a kernel debugger attaches to the runtime state at the point of the call you want to test

• Given the right amount of effort, you can hook any API and respond accordingly

• You can see this in the example multi-packet TLV sample

• Confirmed this in eBPF fuzzing we did last year – multi IOCTL
kAFL / NYX Fuzzer

YOU WANT KASAN AND SYZKALLER ON WINDOWS?

• Pure emulation based kernel fuzzing will find its limits quickly due to device I/O

• The final boss for OS fuzzing is writing a custom hypervisor that resets live memory and execution state in an OS that still has access to virtualized devices

• Using native hypervisor allows use of IntelPT for 5% full system coverage

• Achieved after a 4 year effort from RUB-SysSec in German
YOU WANT KASAN AND SYZKALLER ON WINDOWS?

- kAFL is now forked by Intel and is state of the art for live OS fuzzing
- Nyx today is provided as an API
- Nyx is integrated in AFL++ for fuzzing hypervisor guests!
“GDB Fuzzing”

TRADING OFF PERFORMANCE TO GET THE JOB DONE

• Anything implementing a gdb stub can technically be fuzzed
• JTAG, embedded devices, simulators, hypervisors, etc
• Recently being explored by Andreas Zeller`s group in the ISSTA`23 paper “Fuzzing Embedded Systems Using Debug Interfaces” (GDBFuzz)
• Avatar^2 for firmware that can't be fuzzed in isolation and needs to emulate code while still accessing hardware I/O on embedded devices
Differential Fuzzing

ASK AN ORACLE WHEN HARNESSING IS IMPOSSIBLE

• Differential Fuzzing is the answer when bugs can be found via a round trip through an encoder/decoder

• Differential Fuzzing also allows comparing across implementations of a parser. This is used by tools like SiliFuzz to fuzz CPUs or others to fuzz smart contracts
Differential Fuzzing

ASK AN ORACLE WHEN HARNESSING IS IMPOSSIBLE

• Smart Contract Fuzzing

For inputs, there are three notable things we’ll want to vary:

• The config which determines how the VM should execute (what features and such)
• The BPF program to be executed, which we’ll generate like we do in “smart”
• The initial memory of the VMs

Once we’ve developed our inputs, we’ll also need to think of our outputs:

• The “return state”, the exit code itself or the error state
• The number of instructions executed (e.g., did the JIT program overrun?)
• The final memory of the VMs
Differential Fuzzing

ASK AN ORACLE WHEN HARNESSING IS IMPOSSIBLE

Then, to execute both JIT and the interpreter, we’ll take the following steps:

- The same steps as the first fuzzers:
  - Use the rBPF verification pass (called “check”) to make sure that the VM will accept the input program
  - Initialise the memory, the syscalls, and the entrypoint
  - Create the executable data

- Then prepare to perform the differential testing
  - JIT compile the BPF code (if it fails, fail quietly)
  - Initialise the interpreted VM
  - Initialise the JIT VM
  - Execute both the interpreted and JIT VMs
  - Compare return state, instructions executed, and final memory, and panic if any do not match.
Summary

Fuzzing can go anywhere, test anything, achieve new heights, we are only at the beginning of expanding the impact and reach of fuzzing. There is room in all three areas I focused on, but we are continually growing capabilities and reach for fuzzing.
Thank You, WOOT’23
Questions?

Richard Johnson  |  rjohnson@fuzzing.io  |  @richinseattle

https://fuzzing.io/woot23.pdf