Modern Fuzzing
Research & Engineering

Andrea Fioraldi
@andreafioraldi
What is Fuzz Testing?

Fuzz Testing, or Fuzzing, is a family of “Software” Testing techniques that involves providing machine-generated inputs to the System Under Test (SUT) in order to satisfy some objectives.
What is Fuzz Testing?

Fuzz Testing, or Fuzzing, is a family of “Software” Testing techniques that involves providing machine-generated inputs to the System Under Test (SUT) in order to satisfy some objectives.
What is Fuzz Testing?

Machine-generated inputs can be of any kind, beyond the classic definition of “unexpected” (by the way, what does it mean?) inputs.
What is Fuzz Testing?

Machine-generated inputs can be of any kind, beyond the classic definition of “unexpected” (by the way, what does it mean?) inputs.

Fuzzing is often considered related to Random Testing, a technique that provides inputs sampled uniform independently from the input space (using a specification maybe, so they are not random bytes in general).
What is Fuzz Testing?

Machine-generated inputs can be of any kind, beyond the classic definition of “unexpected” (by the way, what does it means?) inputs.

Fuzzing is often considered related to Random Testing, a technique that provides inputs sampled uniform independently from the input space (using a specification maybe, so they are not random bytes in general).

But Fuzzing can generate inputs deterministically, or can generate inputs mutating previously generated inputs that makes the sampling from the input space not independent.
Widely discussed SOTA Fuzzing in 2022

- Feedback-driven, mainly Coverage-guided
Widely discussed SOTA Fuzzing in 2022

- Feedback-driven, mainly Coverage-guided
- Can bypass coverage roadblocks (concolic-aided, taint-assisted, RedQueen, ...)

```c
if (input == 0xabadcafe) {
    interesting_code();
}
```
Widely discussed SOTA Fuzzing in 2022

- Feedback-driven, mainly Coverage-guided
- Can bypass coverage roadblocks (concolic-aided, taint-assisted, RedQueen, ...)
- Input models help to fuzz deeper

```
<start> ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= +<factor> | -<factor> | (<expr>) | <integer> |
<integer> ::= <digit><integer> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
Widely discussed SOTA Fuzzing in 2022

- Feedback-driven, mainly Coverage-guided
- Can bypass coverage roadblocks (concolic-aided, taint-assisted, RedQueen, ...)
- Input models help to fuzz deeper
- Can test network interactions
Widely discussed SOTA Fuzzing in 2022

- Feedback-driven, mainly Coverage-guided
- Can bypass coverage roadblocks (concolic-aided, taint-assisted, RedQueen, ...)
- Input models help to fuzz deeper
- Can test network interactions
- Can fuzz userspace programs, kernel, hypervisors, ...
Widely used tools in 2022

LLVM's Libfuzzer

google/honggfuzz

Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based)

Contributors: 68
Issues: 17
Stars: 3k
Forks: 492

havoc/splice: 506/1.05M
py/custom: 0/0, 0/0
trim: 19.25%/53.2k, n/a
We still miss bugs

Yes, even in heavily-fuzzed projects in OSS-Fuzz
Still finding these bugs by hand...

---

**Project Zero**

News and updates from the Project Zero team at Google

**Wednesday, December 1, 2021**

**This shouldn't have happened: A vulnerability postmortem**

Posted by Tavis Ormandy, Project Zero

**Introduction**

This is an unusual blog post. I normally write posts to highlight some hidden attack surface or interesting complex vulnerability class. This time, I want to talk about a vulnerability that is neither of those things. The striking thing about this vulnerability is just how simple it is. This should have been caught earlier, and I want to explore why that didn't happen.

In 2021, all good bugs need a catchy name, so I'm calling this one “BigSig”.

First, let's take a look at the bug, I'll explain how I found it and then try to understand why we missed it for so long.
Still finding these bugs by hand...

**Issue 2272: libxml2: heap-buffer-overflow in xmlBufAdd**

Reported by fwihelm@google.com on Tue, Mar 8, 2022, 4:19 PM GMT+1

libxml2 is vulnerable to a heap-buffer-overflow when xmlBufAdd is called on a very large buffer:

```c
int xmlBufAdd(xmlBufPtr buf, const xmlChar *str, int len) {
    unsigned int needSize;

    needSize = buf->use + len + 2; (A)
    if (needSize > buf->size){
        if (xmlBufResize(buf, needSize)){
            xmlBufMemoryError(buf, "growing buffer");
            return XML_ERR_NO_MEMORY;
        }
    }
}
```

Team at Google

A vulnerability postmortem

In 2021, all good bugs need a catchy name, so I’m calling this one “BigSig”.

First, let’s take a look at the bug, I’ll explain how I found it and then try to understand why we missed it for so long.
**Why?**

- Fuzzers often test only the default configuration
- Fuzzers have input length limits
- Code coverage as feedback is not enough
Why?

- Fuzzers often test only the default configuration
- Fuzzers have input length limits
- Code coverage as feedback is not enough (beware of path explosion!)

○ Fioraldi, D’Elia, Balzarotti. “The Use of Likely Invariants as Feedback for Fuzzers”

○ Mantovani, Fioraldi, Balzarotti. “Fuzzing with Data Dependency Information”

○ Herrera, Payer, Hosking. “DATAFLOW - Towards a Data-Flow-Guided Fuzzer”
int wavlike_msadpcm_init (SF_PRIVATE *psf, int blockalign, int samplesperblock)
{
  MSADPCM_PRIVATE *pms;
  unsigned int pmssize = sizeof (MSADPCM_PRIVATE) + blockalign + 3 * psf->sf.channels * samplesperblock;
  pms->samples = pms->dummydata; // array in pms
  pms->block = (unsigned char*) (pms->dummydata + psf->sf.channels * samplesperblock);
  pms->channels = psf->sf.channels;
  pms->blocksize = blockalign;
  ...
}

...
An Example

```c
int wavlike_msadpcm_init (SF_PRIVATE *psf, int blockalign, int samplesperblock)
{
  MSADPCM_PRIVATE *pms;
  unsigned int pmssize;

  pmssize = sizeof (MSADPCM_PRIVATE) + blockalign + 3 * psf->sf.channels * samplesperblock;

  pms->samples = pms->dummydata;  // array in pms
  pms->block = (unsigned char*) (pms->dummydata + psf->sf.channels * samplesperblock);
  pms->channels = psf->sf.channels;
  pms->blocksize = blockalign;
}
```
static int msadpcm_decode_block (SF_PRIVATE *psf, MSADPCM_PRIVATE *pms)
{
    ... sampleindx = 2 * pms->channels;
    while (blockindx < pms->blocksize)
    {
        bytecode = pms->block [blockindx++];
        pms->samples [sampleindx++] = (bytecode >> 4) & 0x0F; // heap overflow bug
        pms->samples [sampleindx++] = bytecode & 0x0F;
    }
    ...
}
An Example

```c
static int msadpcm_decode_block (SF_PRIVATE *psf, MSADPCM_PRIVATE *pms)
{
    sampleindx = 2 * pms->channels;
    while (blockindx < pms->blocksize)
    {
        bytecode = pms->block [blockindx++];
        pms->samples [sampleindx++] = (bytecode >> 4) & 0x0F; // heap overflow bug
        pms->samples [sampleindx++] = bytecode & 0x0F;
    }
}
```

This only happens when the program is in a specific state, characterized by a small allocation size for the pms buffer and a pms->blocksize value sufficiently high to force the loop to write out of the bounds of the array.

However, none of these requirements can be extracted from code coverage, as there are no branches in the program that involve these thresholds.
An Example

```c
pmssize = sizeof (MSADPCM_PRIVATE) + blockalign + 3 * psf->sf.channels * samplesperblock
```
Why?

- Fuzzers often tests only the default configuration
- Fuzzers have input length limits
- Code coverage as feedback is not enough
- Harnessing to cover all the code is hard (especially for devs)
Why?

- Harnessing to cover all the code is hard (especially for devs)
  - We can generate them automatically
  - Ispoglou, Austin, Mohan, Payer. “FuzzGen: Automatic Fuzzer Generation”
Why?

- Harnessing to cover all the code is hard (especially for devs)
  - We can generate them automatically
  - We need introspection of what the fuzzer can cover

- Fuzz Introspector (https://github.com/ossf/fuzz-introspector)
Why?

- Harnessing to cover all the code is hard (especially for devs)
  - We can generate them automatically
  - We need introspection of what the fuzzer can cover

Fuzz Introspector (https://github.com/ossf/fuzz-introspector)
Beyond memory corruption bugs

A SQL injection is not causing a segfault in your application
Several paths are SOTA

- **Differential fuzzing**
  - CryptoFuzz (https://github.com/guidovranken/cryptofuzz)
  - Maier, Fäßler, Seifert. “Uncovering Smart Contract VM Bugs Via Differential Fuzzing”
Several paths are SOTA

- **Differential fuzzing**
  - Cryptofuzz ([https://github.com/guidovranken/cryptofuzz](https://github.com/guidovranken/cryptofuzz))
  - Maier, Fäßler, Seifert. “Uncovering Smart Contract VM Bugs Via Differential Fuzzing”

- **Custom bug detectors**
  - Handwritten bug detectors, useful for memory safe languages (e.g. Java, [https://www.code-intelligence.com/blog/log4j-bug-detectors](https://www.code-intelligence.com/blog/log4j-bug-detectors))
  - Custom sanitizers (e.g. [https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49053](https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49053))
  - Mining invariants and automatic insertion of assertions
    - Daikon, Purify,
Several paths are SOTA

- **Differential fuzzing**
  - Cryptofuzz ([https://github.com/guidovranken/cryptofuzz](https://github.com/guidovranken/cryptofuzz))
  - Maier, Fäßler, Seifert. “Uncovering Smart Contract VM Bugs Via Differential Fuzzing”

- **Custom bug detectors**
  - Handwritten bug detectors, useful for memory safe languages (e.g. Java) ([https://www.code-intelligence.com/blog/log4j-bug-detectors](https://www.code-intelligence.com/blog/log4j-bug-detectors))
  - Custom sanitizers (e.g. [https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49053](https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49053))
  - Mining invariants and automatic insertion of assertions
    - Daikon, Purify, …
Can we do better?

- Improve invariants mining, the coverage problem causes too many false positive and locally valid constraints unsuitable for fuzzing.
Can we do better?

- Improve invariants mining, the coverage problem causes too many false positive and locally valid constraints unsuitable for fuzzing.
- Build large databases of bug patterns (?)
Can we do better?

- Improve invariants mining, the coverage problem causes too many false positive and locally valid constraints unsuitable for fuzzing.

- Build large databases of bug patterns (?)

- Maybe it’s time to start approaching program analysis problems with ML without the “wanna find something to apply this model” bias.
Wanna build a fuzzer and compare with the others?

Good luck.
Problem: Fuzzers Fragmentation

From https://fuzzing-survey.org/
**Cause: Monolithic Codebases**

Fuzzers are designed to be tools⇒ Designed to be tools
⇒ Not designed with code reuse in mind⇒ Hard to extend

Many fuzzers are incompatible forks of others (usually AFL)
This makes them incompatible with orthogonal techniques
How to Create a Fuzzer Then?

- Fork an existing fuzzer (the n-th AFL-something)
- Create a custom fuzzer from scratch
Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers.
Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers.
- Reinventing the wheel, you will code the same code to do that same thing that all others do again and again.
Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers.
- Reinventing the wheel, you will code the same code to do that same thing that all others do again and again.
- Naive design, typically just a mutator.
Custom Fuzzer Engineering Issues

- Lack of code reuse, you will have to spend a lot of time in adapting different techniques from different fuzzers.
- Reinventing the wheel, you will code the same code to do that same thing that all others do again and again.
- Naive design, typically just a mutator.
- Scaling, you cannot adapt it easily to multi-core or -machine.
LibAFL, the fuzzer library.

Advanced Fuzzing Library - Slot your own fuzzers together and extend their features using Rust.

LibAFL is written and maintained by Andrea Floraldi and Dominik Mayer. mail@dmnk.co.

Why LibAFL?

LibAFL gives you many of the benefits of an off-the-shelf fuzzer, while being completely customizable. Some highlight features currently include:

- n/a, n/a, n/a, n/a
- known integers: n/a, n/a, n/a, n/a
- dictionary: n/a, n/a, n/a, n/a
- havoc/splice: 506/1.05M, 1/1
- py/custom: 0/0, 0/0
- trim: 19.25%/53.2k

Contributors

+ 15 contributors

Languages

- Rust 60.3%
- C 36.5%
- C++ 1.7%
- Makefile 0.9%
- Shell 0.4%
- Dockerfile 0.2%
LibAFL is a library for fuzzers that are
- **Fast** (low IPC, runtime overhead)
- **Scalable** (almost linearly to 200+ cores)
- **Portable** (Android, Windows, MacOS, Linux, Kernels, ...)
- **State-of-the-Art** (Hybrid-, Grammar-, Token-, Feedback-Fuzzing)
- **Multi-instrumentation** (binary-only Frida & Qemu, Clang, Python,...)

And, most importantly, **very extendable** with your own components.
MattGorko/Tartiflette
Snapshot fuzzing with KVM and LibAFL
MattGorko/Tartiflette
Snapshots:

epi052/feroxfuzz
A structure-aware HTTP fuzzing library
MattGorko/Tartiflette

A symbolic-model-guided fuzzer for TLS

tlspuffin/tlspuffin

A structure-aware HTTP fuzzing library
Is fuzzer X better than Y?

We don’t know. Really, we can only speculate about this.
Current benchmarking metrics

- Code coverage over time
- Bugs over time
- Speed
- CVEs found (lol)
- Reached coverage for each fuzz case (not so used, IMO useful to benchmark structured mutators)
Standard benchmarks ATM

FuzzBench: Fuzzer Benchmarking As a Service

FuzzBench is a free service that evaluates fuzzers on a wide variety of real-world benchmarks, at Google scale. The goal of FuzzBench is to make it painless to rigorously evaluate fuzzing research and make fuzzing research easier for the community to adopt. We invite members of the research community to contribute their fuzzers and give us feedback on improving our evaluation techniques.

FuzzBench provides:

- An easy API for integrating fuzzers.
- Benchmarks from real-world projects. FuzzBench can use any OSS-Fuzz project as a benchmark.
- A reporting library that produces reports with graphs and statistical tests to help you understand the significance of results.

To participate, submit your fuzzer to run on the FuzzBench platform by following our simple guide. After your integration is accepted, we will run a large-scale experiment using your fuzzer and generate a report comparing your fuzzer to others, such as AFL and libFuzzer. See a sample report.
Magma: A Ground-Truth Fuzzing Benchmark

Magma is a collection of open-source libraries with widespread usage and a long history of security-critical bugs and vulnerabilities. In light of the need for better fuzzer evaluation, we front-ported bugs from previous bug reports to the latest versions of these libraries.

For each ported bug, we added in-line (source-code-level) instrumentation to collect ground-truth information about bugs reached (buggy code executed) and triggered (fault condition satisfied by input). This instrumentation allows a monitoring utility to measure fuzzer progress in real time.

Magma also includes the captain toolset which facilitates the process of building Magma targets and running campaigns.

Check out a sample Magma report and read the paper. Questions, comments, and feedback are welcome!

generate a report comparing your fuzzer to others, such as AFL and libFuzzer. See a sample report.
Can we improve?

- More representative bugs
- “Automated Magma”
- Changing often the targets (maybe from OSSFuzz) to avoid overfitting
- Decent synthetic bugs?
Can we improve?

- More representative bugs
- "Automated Magma"
- Changing often the targets (maybe from OSSFuzz) to avoid overfitting
- Decent synthetic bugs?
Can we improve?

**FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing**

Authors:
Zenong Zhang and Zach Patterson, University of Texas at Dallas; Michael Hicks, University of Maryland and Amazon; Shiyi Wei, University of Texas at Dallas

Distinguished Paper Award Winner

Abstract:
Fuzz testing is an active area of research with proposed improvements published at a rapid pace. Such proposals are assessed empirically: Can they be shown to perform better than the status quo? Such an assessment requires a benchmark of target programs with well-identified, realistic bugs. To ease the construction of such a benchmark, this paper presents FIXREVERTER, a tool that automatically injects realistic bugs in a program. FIXREVERTER takes as input a bugfix pattern which contains both code syntax and semantic conditions. Any code site that matches the specified syntax is undone if the semantic conditions are satisfied, as checked by static analysis, thus (re)introducing a likely bug. This paper focuses on three bugfix patterns, which we call conditional-abort, conditional-construct, and conditional-assign, based on a study of fixes in a corpus of Common Vulnerabilities and Exposures (CVEs). Using FIXREVERTER we have built REVBUGBENCH, which consists of 10 programs into which we have injected nearly 8,000 bugs; the programs are taken from FuzzBench and Binutils, and represent common targets of fuzzing evaluations. We have integrated REVBUGBENCH into the FuzzBench service, and used it to evaluate five fuzzers. Fuzzing performance varies by fuzzer and program, as desired/expected. Overall, 219 unique bugs were reported, 19% of which were detected by just one fuzzer.
Hard engineering problems

There’s a paper about it, problem solved.
Re-implementing things is hard

- Development cost and maintenance

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time : @ days, @ hrs, @ min, 43 sec</td>
<td>cycles done : 15</td>
</tr>
<tr>
<td>last uniq crash : none seen yet</td>
<td>total paths : 703</td>
</tr>
<tr>
<td>last uniq hang : none seen yet</td>
<td>uniq crashes : 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>uniq hangs : 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing : 261*1 (37.1%)</td>
<td>map coverage</td>
</tr>
<tr>
<td>paths timed out : 0 (0.00%)</td>
<td>map density : 5.78% / 13.98%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying : splice 14</td>
<td>favored paths : 114 (16.22%)</td>
</tr>
<tr>
<td>stage execs : 31/32 (96.88%)</td>
<td>new edges on : 167 (23.76%)</td>
</tr>
<tr>
<td>total execs : 2.55M</td>
<td>total crashes : 0 (0 unique)</td>
</tr>
<tr>
<td>exec speed : 61.2k/sec</td>
<td>total tmouts : 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips : n/a, n/a, n/a</td>
<td>levels : 11</td>
</tr>
<tr>
<td>byte flips : n/a, n/a, n/a</td>
<td>pending : 121</td>
</tr>
<tr>
<td>arithmetics : n/a, n/a, n/a</td>
<td>pend fav : 0</td>
</tr>
<tr>
<td>known ints : n/a, n/a, n/a</td>
<td>own finds : 699</td>
</tr>
<tr>
<td>dictionary : n/a, n/a, n/a</td>
<td>imported : n/a</td>
</tr>
<tr>
<td>havoc/splice : 506/1.05M, 193/1.44M</td>
<td>stability : 99.88%</td>
</tr>
<tr>
<td>py/custom : 0/0, 0/0</td>
<td>[cpu0000: 12%]</td>
</tr>
<tr>
<td>trim : 19.25%/53.2k, n/a</td>
<td></td>
</tr>
</tbody>
</table>
Re-implementing things is hard

- Development cost and maintenance
- Re-evaluate techniques to decide if the improvement is worth the effort
Re-implementing things is hard

- Development cost and maintenance
- Re-evaluate techniques to decide if the improvement worths the effort
- Can we do better simply buying more core?
Re-implementing things is hard

- Development cost and maintenance
- Re-evaluate techniques to decide if the improvement is worth the effort
- Can we do better simply by buying more cores?
Re-implementing things is hard

- Development cost and maintenance
- Re-evaluate techniques to decide if the improvement is worth the effort
- Can we do better simply by buying more cores?
- Lack of production-ready engines for tracing/instrumentation of exotic targets
Hard targets
Hard targets
Hard targets
Hard targets

- Usability gap
- Emulation-based fuzzing tools are out-of-date
- We need something like “Step till the break point, put the input in $rdi, snapshot fuzz from here”
Ask more about fuzzing at https://discord.gg/gCraWct
Thanks y’all