Lightning in a Bottle

25 Years of Fuzzing

What is Fuzzing?

What is Fuzzing?

Fuzzing is lightning in a bottle,
at least it was for Barton Miller.

Barton Miller - Father of Fuzz

Legend has it, Prof. Miller discovered fuzzing during a lightning storm in 1988

His modem, a 1200 baud variety lacking error correction, began to pick up interference from the lightning storm, resulting in garbage characters being injected into his command line

Miller took note as his garbled commands sent unexpected input to common utilities and caused them to crash

Inspired by the storm, Prof. Miller wanted to run an wider experiment so … he assigned his graduate students an exercise

The Fuzz Generator

October 1988 - Bart Miller assigns “The Fuzz Generator” project

The goal of this project is to evaluate the robustness of various UNIX utility programs

First, you will build a fuzz generator. This is a program that will output a random character stream

Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them

For the utilities that break, you will try to determine what type of input cause the break

The Fuzz Generator

October 1988 - Bart Miller assigns “The Fuzz Generator” project

The fuzz generator will generate an output stream of random characters with options for tuning the output.

- only the printable ASCII characters
- all ASCII characters
- include the null (0 byte) character
- generate random length lines (\n terminated strings)
- replay characters in file “name” to output

The Fuzz Generator

October 1988 - Bart Miller assigns “The Fuzz Generator” project

The result of this exercise was ‘fuzz’, one of the world’s first random testing tools

fuzz was used to test 90 console utilities in 7 UNIX varieties - it crashed up to a third of them!

The results were published with tools and data in a 1990 paper titled “An Empirical Study of the Reliability of UNIX Utilities”, coining the term fuzzing in the process.

Fuzzing is Inexpensive

“Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability.”

Fuzz v2

Miller tried again in 1995 with improvements

- X Windows clients
- Network ports
- Memory exhaustion simulation

Crashed as many as 40% of the console utilities and 25% X windows clients

None of the network facing code faulted

Fuzzing is Effective

“Our 1995 study surprised us … the continued prevalence of bugs in the basic UNIX utilities seems a bit disturbing. The simplicity of performing random testing and its demonstrated effectiveness would seem to be **irresistible** to corporate testing groups.”

My Encounter With Fuzzing

My first encounter with fuzzing was around year 2000 as a project with a friend on IRC who suggested we write an argument variable (argv) fuzzer for setuid binaries.

There had traditionally been a number of buffer overflows in argument handling and format string vulnerabilities were a brand new attack vector which was bearing fruit.

This knowledge became shared among hacker circles and it was a common rite of passage after the typical network scanning tools and log cleaners.

Selectively Random

The random fuzzing technique leveraged by Miller's students began to evolve to introduce lists of potentially dangerous input values were injected at random locations in well formed inputs, and **mutational fuzzing** was born which continues to be a foundational fuzzing strategy today.

Meanwhile, many took note that the network services seemed impervious to the random testing attempted by Miller and took a more pragmatic approach - we would need to model protocols and follow the specification closely to create data that would pass the sanity checks in network facing code.

“Smart” Fuzzers

In 2002-2004 a series of fuzzers were released from different researchers taking a similar approach - structured **generational fuzzing**.

These fuzzers were considered “smart” in the sense they are aware of the expected input structure and these structures were tied to value generators which would create permutations of the expected input structure with fuzzy values placed at strategic locations.

Fuzzing is Flexible

- Structured binary parsers
- Text based parsers and tokenizers
- Language parsers
- Complex DOM constructions
- System and Library APIs

The Art of Fuzzing

Developing fuzzing test tools is is a trade-off between ease of setup and time investment in building a complete model of the syntax or interaction logic of a program

Fuzzing interfaces with unstructured inputs will yield limited results

Structured inputs allow for more effective traversal of program states

Defining models of input is tedious and increases the cost of fuzzing

Fuzzing tools should be deployed rapidly and iterated on until the appropriate equilibrium between risk and assurance is met.

The Art of Fuzzing

The intersection of mutation and structure is where the art of fuzzing begins

The Art of Harnessing

The most important factor in fuzzing is input generation. After input generation, it all comes down to harnessing.

Harnessing: The Art in Fuzzing

Generating useful input is important, but some parsers cannot be penetrated by fuzzing external format types or protocols due to complexity, compression, consistency checking, or statefulness.

Many parsers implement complex protocols or structured formats that contain sub parsers that can be attacked directly

A skillful engineer can instrument or interrogate code directly without using external APIs by writing custom harnesses or hooking existing code.

The Fuzzing Renaissance

Feedback driven fuzzing has achieved new heights in fuzzing efficacy

Feedback Driven Fuzzing

Feedback driven fuzzing evaluates the effectiveness of the each iteration of fuzzing

The most common feedback signal is new code coverage. Logical blocks of higher level language are translated to low level basic blocks in code with one or more edges entering or exiting a block

Instrumentation or emulation of the program allows tracking of each edge between basic blocks

Feedback Driven Fuzzing

Feedback may take other forms

- Time spent parsing input
- CPU performance counters
  - instruction count
  - mispredicted branches
- Any other observable side effect that strongly correlates to unique behavior in the target program

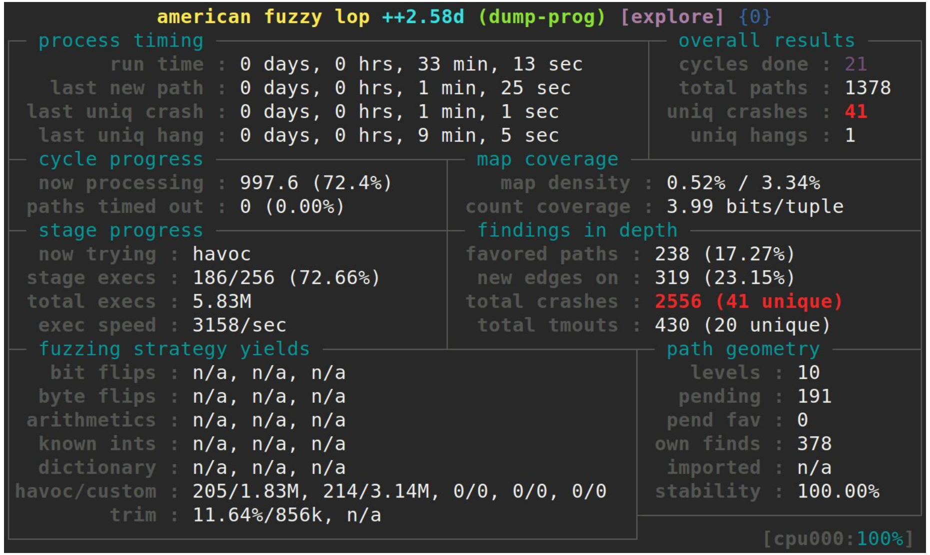
American Fuzzy Lop - Zalewski, 2013

Block based code coverage tracking

- 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 variety of traditional mutation fuzzing strategies

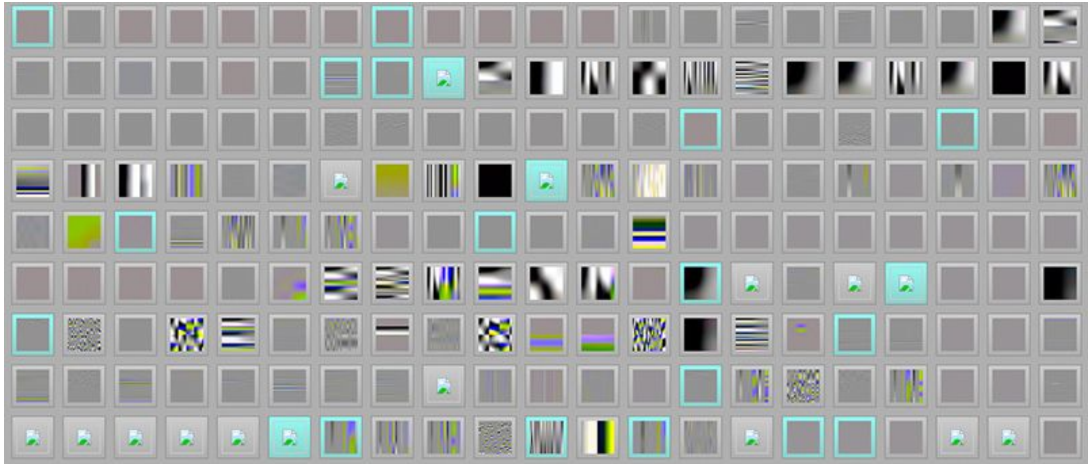


```
            american fuzzy lop ++2.58d (dump-prog) [explore] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 33 min, 13 sec      │  cycles done : 21    │
│   last new path : 0 days, 0 hrs, 1 min, 25 sec       │  total paths : 1378  │
│ last uniq crash : 0 days, 0 hrs, 1 min, 1 sec        │ uniq crashes : 41    │
│  last uniq hang : 0 days, 0 hrs, 9 min, 5 sec        │   uniq hangs : 1     │
├─ cycle progress ───────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 997.6 (72.4%)    │    map density : 0.52% / 3.34%         │
│ paths timed out : 0 (0.00%)        │ count coverage : 3.99 bits/tuple       │
├─ stage progress ───────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                │ favored paths : 238 (17.27%)           │
│ stage execs : 186/256 (72.66%)     │  new edges on : 319 (23.15%)           │
│ total execs : 5.83M                │ total crashes : 2556 (41 unique)       │
│  exec speed : 3158/sec             │  total tmouts : 430 (20 unique)        │
├─ fuzzing strategy yields ──────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                        │    levels : 10         │
│  byte flips : n/a, n/a, n/a                        │   pending : 191        │
│ arithmetics : n/a, n/a, n/a                        │  pend fav : 0          │
│  known ints : n/a, n/a, n/a                        │ own finds : 378        │
│  dictionary : n/a, n/a, n/a                        │  imported : n/a        │
│havoc/custom : 205/1.83M, 214/3.14M, 0/0, 0/0, 0/0  │ stability : 100.00%    │
│        trim : 11.64%/856k, n/a                     ├────────────────────────┘
└────────────────────────────────────────────────────┘          [cpu000:100%]
```

American Fuzzy Lop

American Fuzzy Lop achieved many impressive feats in its first couple years of release.

One striking blog post in 2014 titled “Pulling JPEGs out of thin air” demonstrated how AFL could generate valid JPEG images starting from a single input string: “hello”



American Fuzzy Lop

Nearly every major browser, file parser, network server, and language interpreter has fallen to the powerful strategy employed by American Fuzzy Lop

OpenSSL
Cryptography and SSL/TLS Toolkit
FFmpeg
NGINX
SQLite

American Fuzzy Lop

It’s 25 years after Barton Miller’s fuzz, and this is American Fuzzy Lop playing Super Mario Bros.

A file format has been created to deserialize the possible input keys (left, right, jump, run)

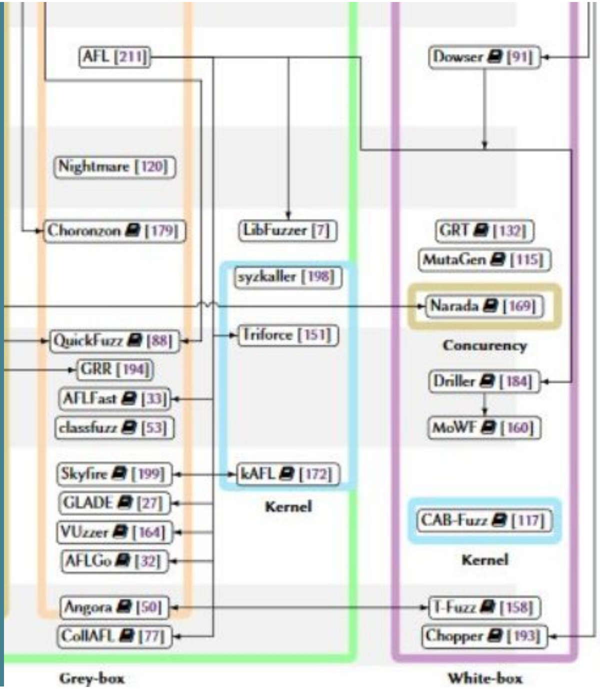
The X coordinate is tracked as progress

When Mario fails, the progress is checked against previous inputs and if it is favorable it is kept and further mutated

Academic Developments

The past 5 years have seen a massive explosion in academic research expanding upon guided fuzzing

- Alternate feedback mechanisms
- Optimized search strategies
- Code transformation
- Feature extraction
- Taint Analysis
- Custom kernels for fuzzing



Cyber Grand Challenge, 2016

Created by The Defense Advanced Research Projects Agency (DARPA) “in order to develop automatic defense systems that can discover, prove, and correct software flaws in real-time”

The Cyber Grand Challenge pitted automated bug hunting tools against an unknown array of targets

For the attack engines, all entries used a combination of guided fuzzing and symbolic execution

Symbolic Execution has been used for Automated Test Case Generation, a precise but limited approach to exercising program logic related to fuzzing

Our hosts, ForAllSecure, had the winning Cyber Grand Challenge entry, Mayhem ForAllSecure

Advanced Harnessing

Forking Servers inject a process scheduler into a target, taking advantage of POSIX fork() system call and copy-on-write for rapid creation of ephemeral test executions

Persistent Fuzzing greatly increases performance and avoids platform constraints around process creation, memory randomization

Emulated Fuzzing enables targeting of embedded IoT, ICS devices, mobile device platforms, etc

Snapshot Fuzzing allows entire operating systems or processes to be restored to their original memory contents, eliminating the possibility of side effects and non-determinism in the execution environment

Desirable Fuzzer Features

Flexible input generation algorithms (examples below from radamsa)

- ab: enhance silly issues in ASCII string data handling
- bd: drop a byte
- bf: flip one bit
- bi: insert a random byte
- br: repeat a byte
- bp: permute some bytes
- bei: increment a byte by one
- bed: decrement a byte by one
- ber: swap a byte with a random one

Desirable Fuzzer Features

Flexible input generation algorithms (examples below from radamsa)

- sr: repeat a sequence of bytes
- sd: delete a sequence of bytes
- ld: delete a line
- lr2: duplicate a line
- li: copy a line closeby
- lr: repeat a line
- ls: swap two lines
- lp: swap order of lines
- lis: insert a line from elsewhere

Desirable Fuzzer Features

Flexible input generation algorithms (examples below from radamsa)

- td: delete a node
- tr2: duplicate a node
- tr: repeat a path of the parse tree
- uw: try to make a code point too wide
- ui: insert funny unicode
- num: try to modify a textual number
- xp: try to parse XML and mutate it
- ft: jump to a similar position in block
- fn: likely clone data between similar positions
- fo: fuse previously seen data elsewhere

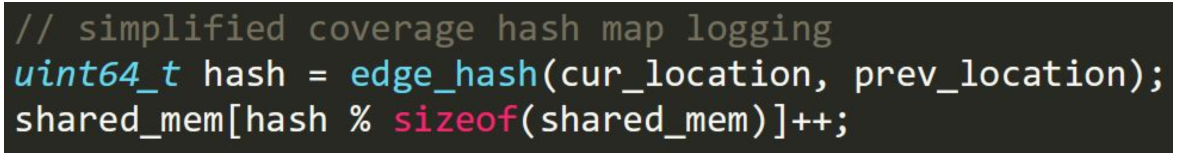
Desirable Fuzzer Features

Token injection - key words or constants

Optional syntax or interaction logic specification

High performance, minimal overhead feedback

- Code coverage with hashed logging for quick lookups or less invasive instrumentation



```
// simplified coverage hash map logging
uint64_t hash = edge_hash(cur_location, prev_location);
shared_mem[hash % sizeof(shared_mem)]++;
```

Desirable Fuzzer Features

Adaptability

- Flexible harnessing via API level input generation
  - AFL outputs to filesystem or STDIN which can be cumbersome

Corpus building over time

- Generated corpus is a valuable resource that should not be discarded
- Regression tests, starting point for future fuzzing, cross-implementation fuzzing

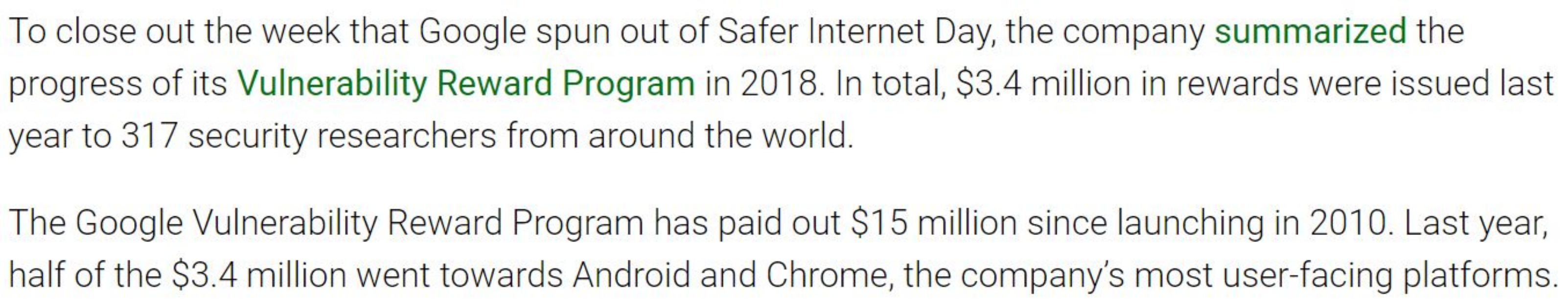
Desirable Fuzzer Features

Repeatable results

- Random seeds values should be recorded and use procedural algorithms based on the initial seed
- Memory randomization should be avoided through system configuration, persistent fuzzing, or snapshot fuzzing
- Non-deterministic behavior should be eliminated through harnessing or controlling sources of non-determinism such as timers, random values, or environmental jitter

Fuzzing in the SDLC

Fuzz now, avoid paying bug bounties later

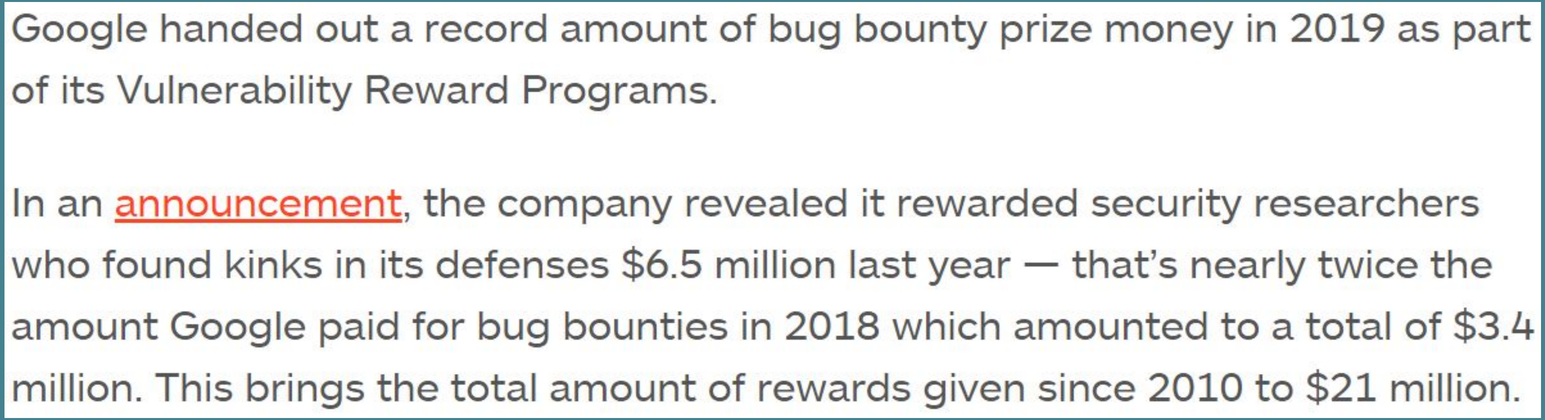


To close out the week that Google spun out of Safer Internet Day, the company summarized the progress of its Vulnerability Reward Program in 2018. In total, $3.4 million in rewards were issued last year to 317 security researchers from around the world.

The Google Vulnerability Reward Program has paid out $15 million since launching in 2010. Last year, half of the $3.4 million went towards Android and Chrome, the company's most user-facing platforms.

Fuzzing in the SDLC

Fuzz now, avoid paying bug bounties later



Google handed out a record amount of bug bounty prize money in 2019 as part of its Vulnerability Reward Programs.

In an announcement, the company revealed it rewarded security researchers who found kinks in its defenses $6.5 million last year — that's nearly twice the amount Google paid for bug bounties in 2018 which amounted to a total of $3.4 million. This brings the total amount of rewards given since 2010 to $21 million.

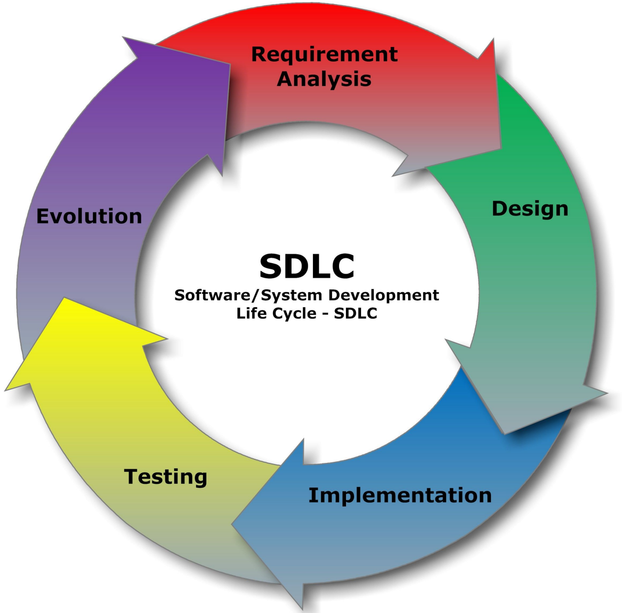
Fuzzing is Software Testing

Fuzzing should be considered as critical to your Software Development Life Cycle as Unit Testing.

Unit Testing validates a functional specification

Fuzzing validates program safety (within limits)

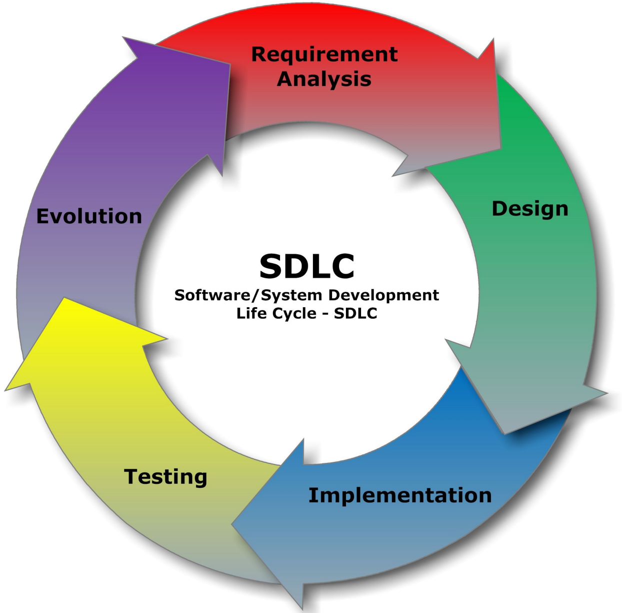
Like Unit Tests, integrating fuzzing early in the development process is much cheaper than to do it retroactively



Fuzzing is Software Testing

Fuzzing enforces defensive programming

Compositional analysis of your software will determine where the security boundaries lie and define which interfaces should be fuzzed

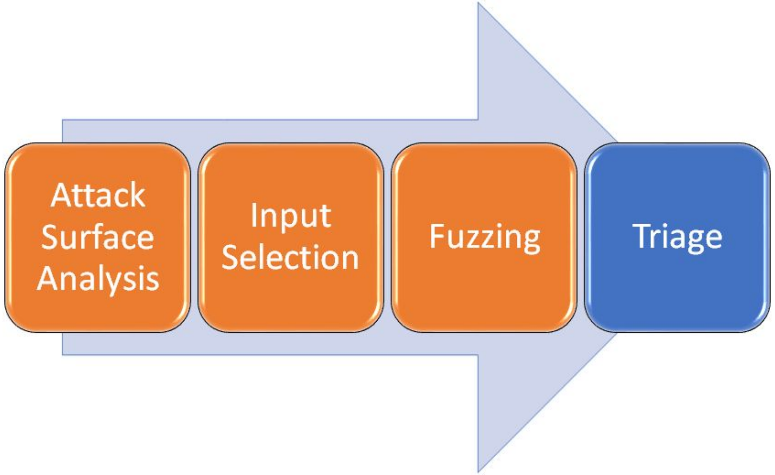


Measuring Fuzzing Coverage

Code coverage is the primary measurement of software unit testing

Not all code interacts with untrusted data. **Attack Surface Analysis** aims to identify what code interacts with user data. We want to measure coverage against that subset of code.

Attack Surface will be activated through selecting appropriate inputs



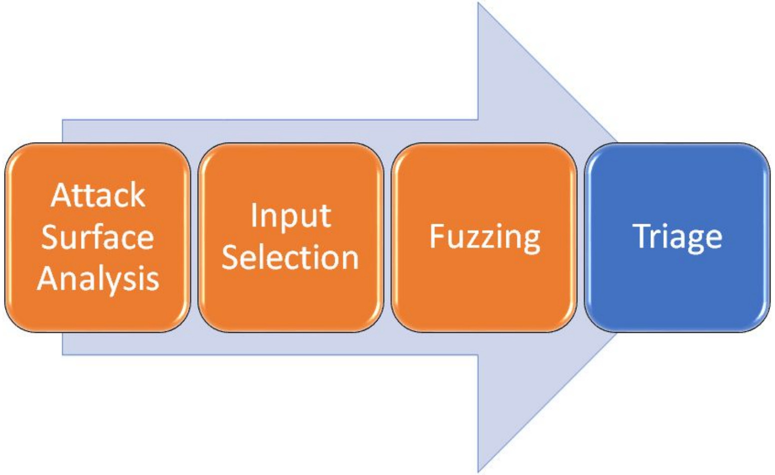
The Fuzzing Process

The APIs that accept untrusted input must be harnessed for fuzzing

The closer to your APIs, the more effective the fuzzer will be

Prefer fuzzing individual feature sets over the entire protocol with a single harness

Fuzzing is a brute force data search, smaller, more focused input is more efficient



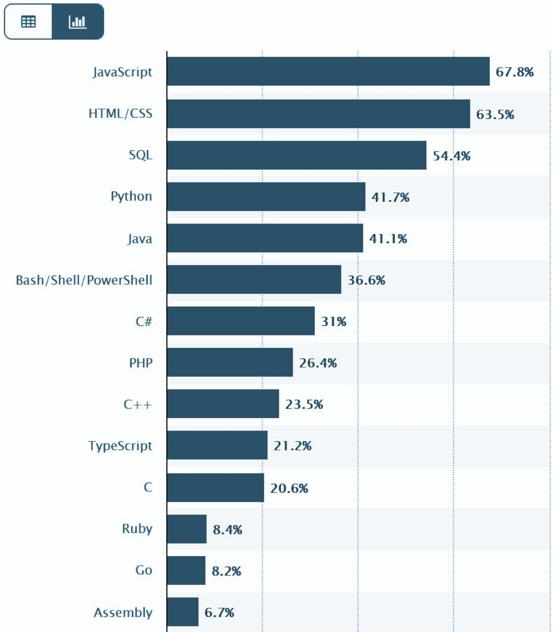
Fuzzing Managed Languages

So far the assumption has been that the target software is "native code" or code that manually manages memory, typically written in C/C++

Managed Language Runtimes typically eliminate the bug classes targeted by bit flipping input mutators

Injection vulnerabilities are most prevalent in managed language so mutations need to have more semantic language awareness

Most used programming languages among developers worldwide, as of early 2019



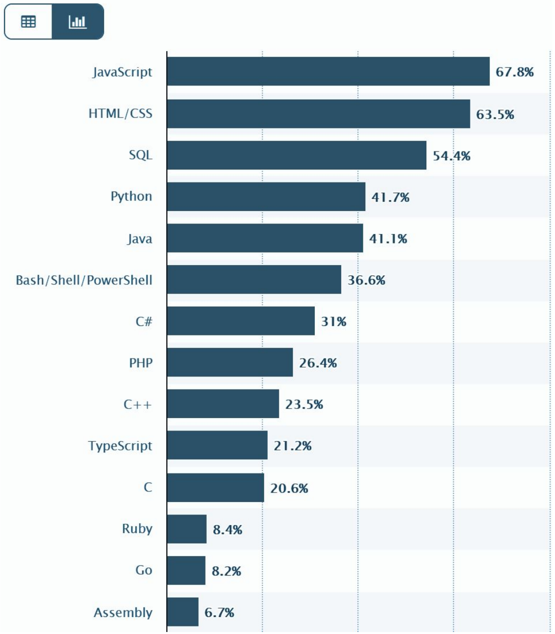
Fuzzing Managed Languages

In the simplest case, keywords or tokens can be injected and sample input formats can be mutated

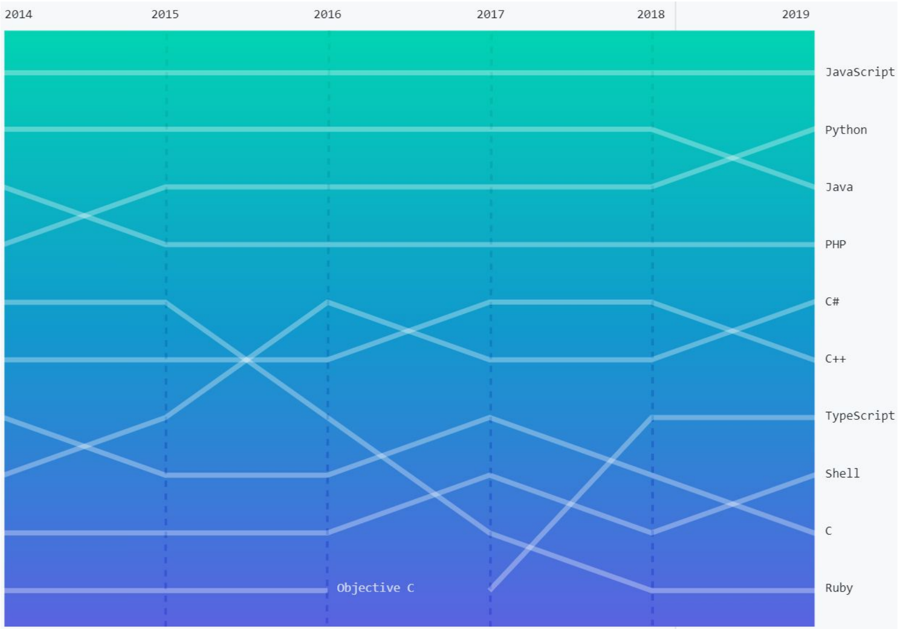
Usually a grammar based generator will be required

Many extensions for AFL exist for targeting managed language, typically the goal is to hit language parser bugs or pass-through APIs that are actually implemented in native code libraries behind the managed APIs

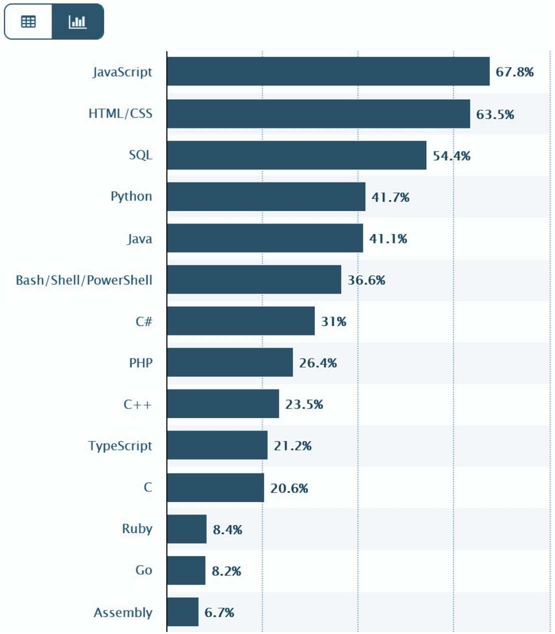
Most used programming languages among developers worldwide, as of early 2019



Fuzzing Managed Languages



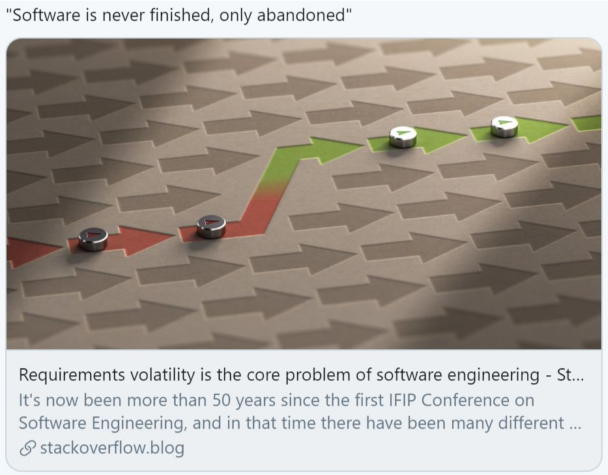
Most used programming languages among developers worldwide, as of early 2019



Fuzzing in the CICD

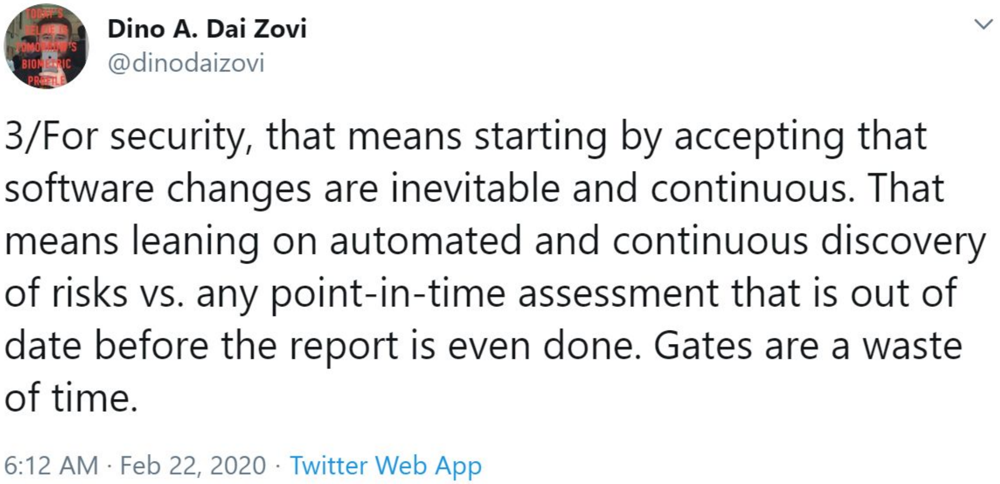
Waterfall is dead, software changes rapidly

Software is Constantly Changing



"Software is never finished, only abandoned"

Requirements volatility is the core problem of software engineering - St...
It's now been more than 50 years since the first IFIP Conference on Software Engineering, and in that time there have been many different ...
🔗 stackoverflow.blog



Dino A. Dai Zovi
@dinodaizovi

3/For security, that means starting by accepting that software changes are inevitable and continuous. That means leaning on automated and continuous discovery of risks vs. any point-in-time assessment that is out of date before the report is even done. Gates are a waste of time.

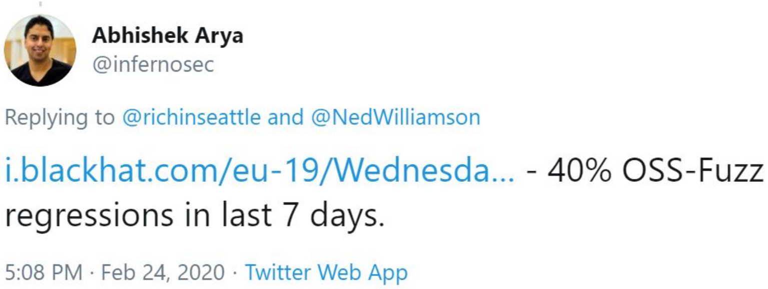
6:12 AM · Feb 22, 2020 · Twitter Web App

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

Specifications rapidly change for the infrastructure that runs our businesses and I'm sure your specifications do too



Abhishek Arya
@infernosec

Replying to @richinseattle and @NedWilliamson

i.blackhat.com/eu-19/Wednesda... - 40% OSS-Fuzz regressions in last 7 days.

5:08 PM · Feb 24, 2020 · Twitter Web App

Continuous Fuzzing is Achievable

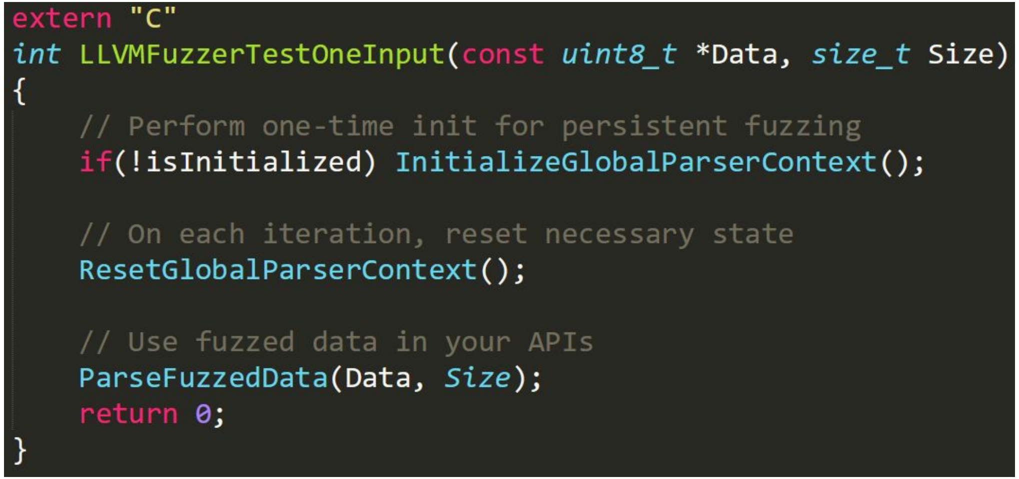
- Write fuzzers
- Build fuzzers
- Fuzz at scale
- Triage crashes
- Improve fuzzers

LLVM libFuzzer

libFuzzer brings AFL style guided fuzzing into the compiler toolchain

AFL mutates files, libFuzzer calls a user supplied callback with mutated data

This is more suitable for CICD and converting unit tests into fuzzers



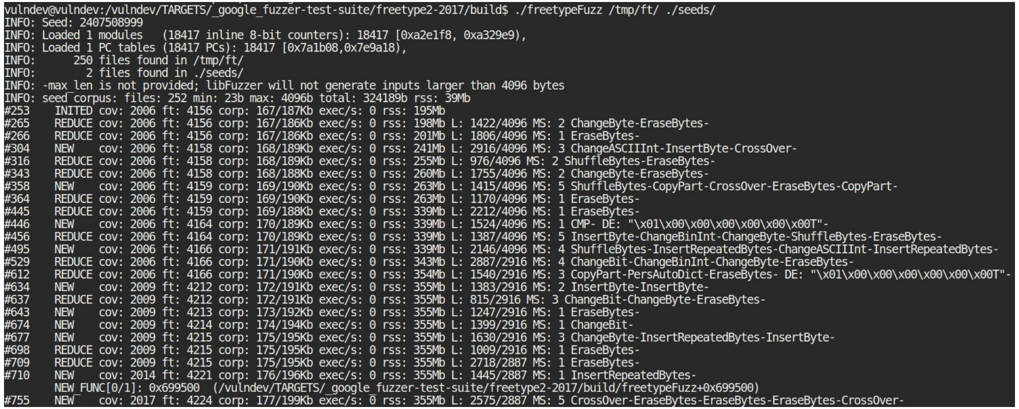
```
extern "C"
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)
{
    // Perform one-time init for persistent fuzzing
    if(!isInitialized) InitializeGlobalParserContext();

    // On each iteration, reset necessary state
    ResetGlobalParserContext();

    // Use fuzzed data in your APIs
    ParseFuzzedData(Data, Size);
    return 0;
}
```

LLVM libFuzzer

libFuzzer brings AFL style guided fuzzing into the compiler toolchain

AFL mutates files, libFuzzer calls a user supplied callback with mutated data

This is more suitable for CICD and converting unit tests into fuzzers

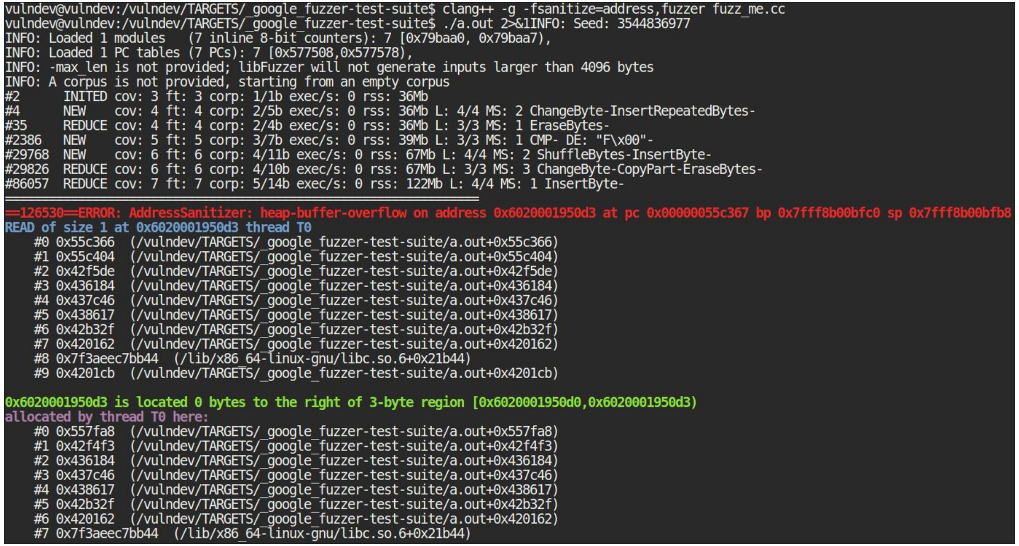


```
vulndev@vulndev:/vulndev/TARGETS/_google_fuzzer-test-suite/freetype2-2017/build$ ./freetypeFuzz /tmp/ft/ ./seeds/
INFO: Seed: 2407508999
INFO: Loaded 1 modules   (18417 inline 8-bit counters): 18417 [0xa2e1f8, 0xa329e9),
INFO: Loaded 1 PC tables (18417 PCs): 18417 [0x7a1b08,0x7e9a18),
INFO:      250 files found in /tmp/ft/
INFO:        2 files found in ./seeds/
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: seed_corpus: files: 252 min: 23b max: 4096b total: 324189b rss: 39Mb
#253    INITED cov: 2006 ft: 4156 corp: 167/187Kb exec/s: 0 rss: 195Mb
#265    REDUCE cov: 2006 ft: 4156 corp: 167/186Kb exec/s: 0 rss: 198Mb L: 1422/4096 MS: 2 ChangeByte-EraseBytes-
#266    REDUCE cov: 2006 ft: 4156 corp: 167/186Kb exec/s: 0 rss: 201Mb L: 1806/4096 MS: 1 EraseBytes-
#304    NEW    cov: 2006 ft: 4158 corp: 168/189Kb exec/s: 0 rss: 241Mb L: 2916/4096 MS: 3 ChangeASCIIInt-InsertByte-CrossOver-
#316    REDUCE cov: 2006 ft: 4158 corp: 168/189Kb exec/s: 0 rss: 255Mb L: 976/4096 MS: 2 ShuffleBytes-EraseBytes-
#343    REDUCE cov: 2006 ft: 4158 corp: 168/188Kb exec/s: 0 rss: 260Mb L: 1755/4096 MS: 2 ChangeByte-EraseBytes-
#358    NEW    cov: 2006 ft: 4159 corp: 169/190Kb exec/s: 0 rss: 263Mb L: 1415/4096 MS: 5 ShuffleBytes-CopyPart-CrossOver-EraseBytes-CopyPart-
#364    REDUCE cov: 2006 ft: 4159 corp: 169/190Kb exec/s: 0 rss: 263Mb L: 1170/4096 MS: 1 EraseBytes-
#445    REDUCE cov: 2006 ft: 4159 corp: 169/188Kb exec/s: 0 rss: 339Mb L: 2212/4096 MS: 1 EraseBytes-
#446    NEW    cov: 2006 ft: 4164 corp: 170/189Kb exec/s: 0 rss: 339Mb L: 1524/4096 MS: 1 CMP- DE: "\x01\x00\x00\x00\x00\x00\x00T"-
#456    REDUCE cov: 2006 ft: 4164 corp: 170/189Kb exec/s: 0 rss: 339Mb L: 1387/4096 MS: 5 InsertByte-ChangeBinInt-ChangeByte-ShuffleBytes-EraseBytes-
#495    NEW    cov: 2006 ft: 4166 corp: 171/191Kb exec/s: 0 rss: 339Mb L: 2146/4096 MS: 4 ShuffleBytes-InsertRepeatedBytes-ChangeASCIIInt-InsertRepeatedBytes-
#529    REDUCE cov: 2006 ft: 4166 corp: 171/190Kb exec/s: 0 rss: 343Mb L: 2887/2916 MS: 4 ChangeBit-ChangeBinInt-ChangeByte-EraseBytes-
#612    REDUCE cov: 2006 ft: 4166 corp: 171/190Kb exec/s: 0 rss: 354Mb L: 1540/2916 MS: 3 CopyPart-PersAutoDict-EraseBytes- DE: "\x01\x00\x00\x00\x00\x00\x00T"-
#634    NEW    cov: 2009 ft: 4212 corp: 172/191Kb exec/s: 0 rss: 355Mb L: 1383/2916 MS: 2 InsertByte-InsertByte-
#637    REDUCE cov: 2009 ft: 4212 corp: 172/191Kb exec/s: 0 rss: 355Mb L: 815/2916 MS: 3 ChangeBit-ChangeByte-EraseBytes-
#643    NEW    cov: 2009 ft: 4213 corp: 173/192Kb exec/s: 0 rss: 355Mb L: 1247/2916 MS: 1 EraseBytes-
#674    NEW    cov: 2009 ft: 4214 corp: 174/194Kb exec/s: 0 rss: 355Mb L: 1399/2916 MS: 1 ChangeBit-
#677    NEW    cov: 2009 ft: 4215 corp: 175/195Kb exec/s: 0 rss: 355Mb L: 1630/2916 MS: 3 ChangeByte-InsertRepeatedBytes-InsertByte-
#698    REDUCE cov: 2009 ft: 4215 corp: 175/195Kb exec/s: 0 rss: 355Mb L: 1009/2916 MS: 1 EraseBytes-
#709    REDUCE cov: 2009 ft: 4215 corp: 175/195Kb exec/s: 0 rss: 355Mb L: 2718/2887 MS: 1 EraseBytes-
#710    NEW    cov: 2014 ft: 4221 corp: 176/196Kb exec/s: 0 rss: 355Mb L: 1445/2887 MS: 1 InsertRepeatedBytes-
        NEW_FUNC[0/1]: 0x699500  (/vulndev/TARGETS/_google_fuzzer-test-suite/freetype2-2017/build/freetypeFuzz+0x699500)
#755    NEW    cov: 2017 ft: 4224 corp: 177/199Kb exec/s: 0 rss: 355Mb L: 2575/2887 MS: 5 CrossOver-EraseBytes-EraseBytes-EraseBytes-CrossOver-
```

LLVM Address Sanitizer

Address Sanitizer is a memory debugger with precise memory tracking that traps on:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Double-free, invalid free
- Use after return/scope

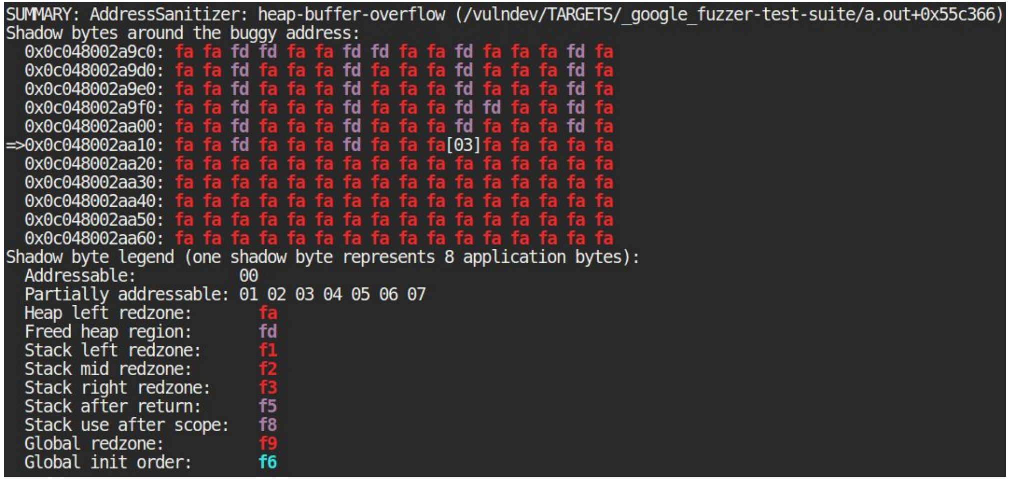


```
vulndev@vulndev:/vulndev/TARGETS/_google_fuzzer-test-suite$ clang++ -g -fsanitize=address,fuzzer fuzz_me.cc
vulndev@vulndev:/vulndev/TARGETS/_google_fuzzer-test-suite$ ./a.out 2>&1INFO: Seed: 3544836977
INFO: Loaded 1 modules   (7 inline 8-bit counters): 7 [0x79baa0, 0x79baa7),
INFO: Loaded 1 PC tables (7 PCs): 7 [0x577508,0x577578),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 3 ft: 3 corp: 1/1b exec/s: 0 rss: 36Mb
#4      NEW    cov: 4 ft: 4 corp: 2/5b exec/s: 0 rss: 36Mb L: 4/4 MS: 2 ChangeByte-InsertRepeatedBytes-
#35     REDUCE cov: 4 ft: 4 corp: 2/4b exec/s: 0 rss: 36Mb L: 3/3 MS: 1 EraseBytes-
#2386   NEW    cov: 5 ft: 5 corp: 3/7b exec/s: 0 rss: 39Mb L: 3/3 MS: 1 CMP- DE: "F\x00"-
#29768  NEW    cov: 6 ft: 6 corp: 4/11b exec/s: 0 rss: 67Mb L: 4/4 MS: 2 ShuffleBytes-InsertByte-
#29826  REDUCE cov: 6 ft: 6 corp: 4/10b exec/s: 0 rss: 67Mb L: 3/3 MS: 3 ChangeByte-CopyPart-EraseBytes-
#86057  REDUCE cov: 7 ft: 7 corp: 5/14b exec/s: 0 rss: 122Mb L: 4/4 MS: 1 InsertByte-
=================================================================
==126530==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020001950d3 at pc 0x00000055c367 bp 0x7fff8b00bfc0 sp 0x7fff8b00bfb8
READ of size 1 at 0x6020001950d3 thread T0
    #0 0x55c366  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x55c366)
    #1 0x55c404  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x55c404)
    #2 0x42f5de  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x42f5de)
    #3 0x436184  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x436184)
    #4 0x437c46  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x437c46)
    #5 0x438617  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x438617)
    #6 0x42b32f  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x42b32f)
    #7 0x420162  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x420162)
    #8 0x7f3aeec7bb44  (/lib/x86_64-linux-gnu/libc.so.6+0x21b44)
    #9 0x4201cb  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x4201cb)

0x6020001950d3 is located 0 bytes to the right of 3-byte region [0x6020001950d0,0x6020001950d3)
allocated by thread T0 here:
    #0 0x557fa8  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x557fa8)
    #1 0x42f4f3  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x42f4f3)
    #2 0x436184  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x436184)
    #3 0x437c46  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x437c46)
    #4 0x438617  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x438617)
    #5 0x42b32f  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x42b32f)
    #6 0x420162  (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x420162)
    #7 0x7f3aeec7bb44  (/lib/x86_64-linux-gnu/libc.so.6+0x21b44)
```

LLVM Address Sanitizer

Address Sanitizer is a memory debugger with precise memory tracking that traps on:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Double-free, invalid free
- Use after return/scope



```
SUMMARY: AddressSanitizer: heap-buffer-overflow (/vulndev/TARGETS/_google_fuzzer-test-suite/a.out+0x55c366)
Shadow bytes around the buggy address:
  0x0c048002a9c0: fa fa fd fd fa fa fd fd fa fa fd fa fa fa fd fa
  0x0c048002a9d0: fa fa fd fa fa fa fd fa fa fa fd fa fa fa fd fa
  0x0c048002a9e0: fa fa fd fa fa fa fd fa fa fa fd fa fa fa fd fa
  0x0c048002a9f0: fa fa fd fa fa fa fd fa fa fa fd fd fa fa fd fa
  0x0c048002aa00: fa fa fd fa fa fa fd fa fa fa fd fa fa fa fd fa
=>0x0c048002aa10: fa fa fd fa fa fa fd fa fa fa[03]fa fa fa fa fa
  0x0c048002aa20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c048002aa30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c048002aa40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c048002aa50: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c048002aa60: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
```

libFuzzer + Address Sanitizer

The design of libFuzzer is setup for integration into build systems

Add additional build steps for fuzzing and include in your CI pipeline

libFuzzer terminates when an error is found, which can be appropriate for blocking a code check in

Google will run libFuzzer harnesses for free through the oss-fuzz project
https://google.github.io/oss-fuzz/

The oss-fuzz repo on github is a great reference for writing harnesses

Fuzzing Orchestration

The last piece to the puzzle besides fuzzing engines and harnessing is orchestration

Google and Mozilla have released the platforms they use to fuzz their browsers and open source dependencies.

ClusterFuzz from Google also runs oss-fuzz

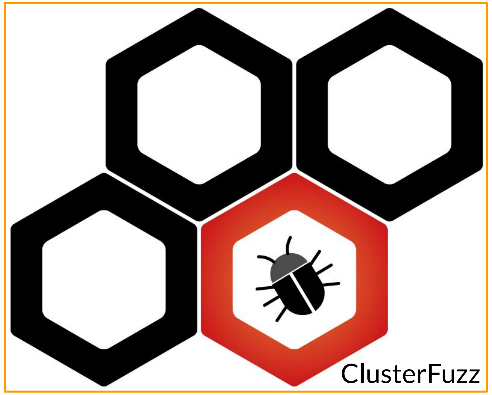
Mozilla provides FuzzManager, CrashManager, Avalanche, Grizzly, etc

ClusterFuzz

FUZZMANAGER

Desirable CICD Features

- High scalability. Google's cluster runs on over 25,000 machines
- Crash deduplication
- Automatic bug filing and closing for issue trackers
- Test case minimization
- Regression finding through bisection
- Statistics for analyzing fuzzer performance and progress
- Easy to use web interface for management and viewing crashes
- Support for coverage guided fuzzing (e.g. libFuzzer and AFL) and black box fuzzing.



Commercially Supported Platforms

Recently, commercially supported platforms have been released

- FuzzBuzz
- FuzzIt
- Microsoft Security Risk Detection

Managing Vulnerability Triage

If fuzzing is successful, then you have another problem.

Security Automation Generates Data

When deployed effectively, security automation will generate a large amount of data that requires expert engineering attention

Typically, developers are initially flooded with results when fuzzing first begins. This can create resistance and should be taken into consideration in late-stage adoption of fuzzing and setting expectations around remediation

Ideally developers will work with security engineers to iterate on their fuzzing harnesses until they are adequate and then security engineers will perform initial triage.

Triage Still Requires Manual Review

There are two approaches currently available for automation of triage

Divide and conquer brute force

- Can locate a code check-in that introduced the bug or minimize an input
- Inefficient and imprecise

Dataflow analysis and taint tracking

- Can determine precise instruction flow leading to crash, heuristics can suggest a patch location
- Complex and new technology

Closing Thoughts

Fuzzing is inexpensive, effective, flexible, and scalable

Fuzzing integrates nicely in CICD through compiler supported frameworks like libFuzzer and oss-fuzz is free

Harnesses should be written by developers alongside unit tests

Orchestration and triage may require custom code & DevOps workflows

Call to Action

Fuzz now, fuzz early

Adopt fuzzing as part of your SDLC and make it as integral as unit testing

Deploy CICD based fuzzing through internal DevOps projects or supported fuzzing services

Talk to us, let us know how we can help!

Thank You, FuzzCon 2020!

FUZZING/IO

Security Automation • Vulnerability Research

Richard Johnson | rjohnson@fuzzing.io