Fuzzing

Wei Le

September 18, 2023

Security bugs are expensive

- Security bugs can bring \$500-\$100,000 on the open market
- Good bug finders make \$180-\$250/hr consulting
- Google vulnerability reward program
- Google Bughunter hall of fame

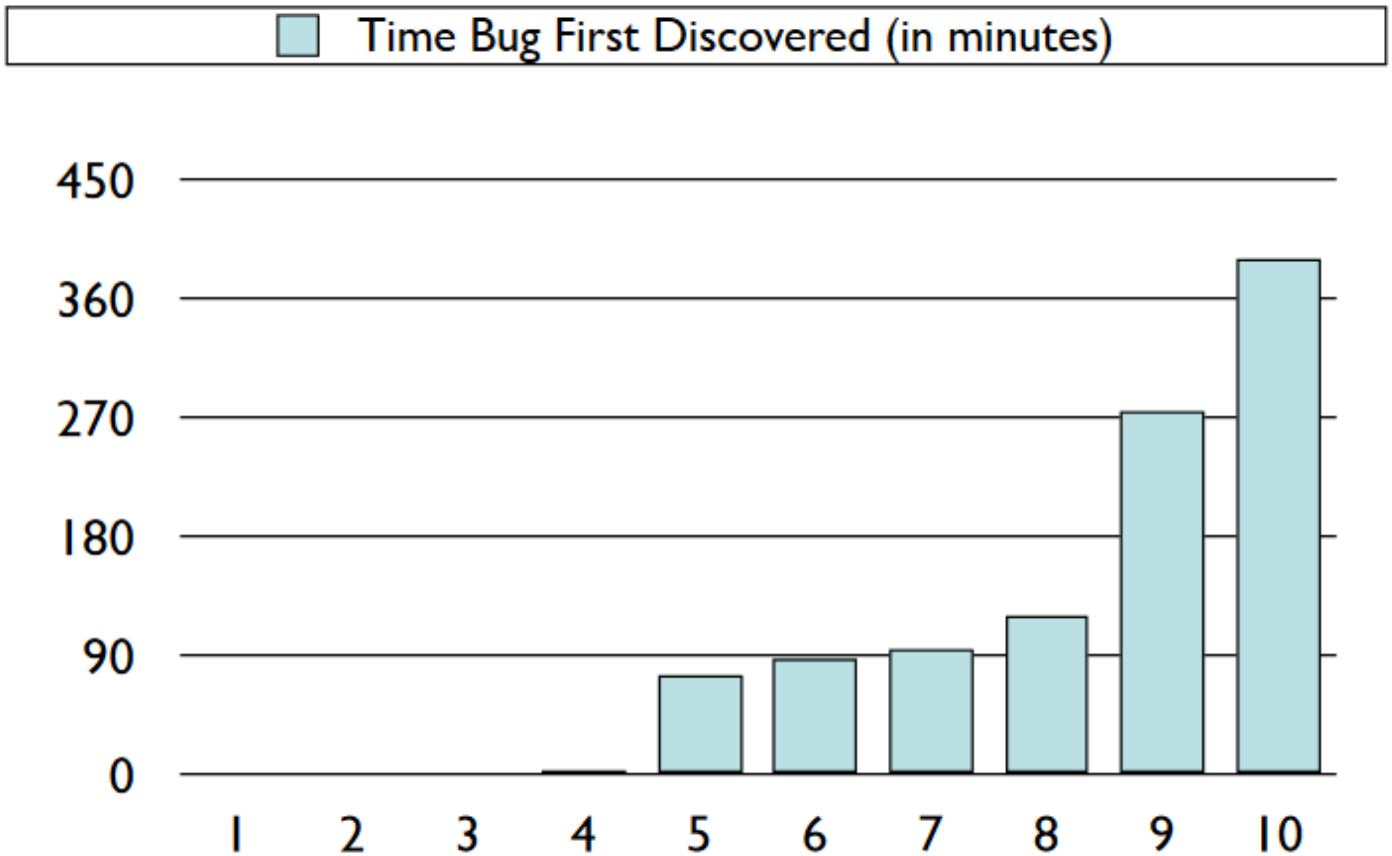
Fuzzing

- early fuzzing, applying to file inputs
- goal: find security bugs, work well for parsers, protocols
- *fuzzing*: generate very bad, invalid inputs to find security bugs

  Fuzz testing is a negative software testing method that feeds malformed and unexpected input data to a program, device, or system with the purpose of finding security-related defects, or any critical flaws leading to denial of service, degradation of service, or other undesired behavior (A. Takanen et al, Fuzzing for Software Security Testing and Quality Assurance, 2008)

- programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called *fuzzers*.

An Example Fuzzer Performance



Types of Fuzzing

- Mutation based — "Dumb Fuzzing"
- Generation based — "Smart Fuzzing"
- White box fuzzing — infer the protocol of the input
- Grey box Fuzzing — use the feedback of the code coverage to mutate the input

see the adapted Tal Garfinkel's slides for mutation and generation based fuzzing

How can we do better than mutation-based and generation-based fuzzing

- How many initial inputs should I use?
- Where (which offsets in input) to apply mutation? What values to replace with? – feedback based evolution algorithm
- How to avoid traps (paths always leading to error handling code)? need white box information

Advanced fuzzing between 2012-2018 [CCS 2018]

- 32 papers
- 50 k hours CPU experiments
- State of the art fuzzers: PyFuzz, OSSFuzz (google)

paper	benchmarks	baseline	trials	variance	crash	coverage	seed	timeout
MAYHEM[8]	R(29)				G	?	N	-
FuzzSim[55]	R(101)	B	100	C	S		R/M	10D
Dowser[22]	R(7)	O	?		O		N	8H
COVERSET[45]	R(10)	O			S, G*	?	R	12H
SYMFUZZ[9]	R(8)	A, B, Z			S		M	1H
MutaGen[29]	R(8)	R, Z			S	L	V	24H
SDF[35]	R(1)	Z, O			O		V	5D
Driller[50]	C(126)	A			G	L, E	N	24H
QuickFuzz-1[20]	R(?)		10		?		G	-
AFLFast[6]	R(6)	A	8		C, G*		E	6H, 24H
SeededFuzz[54]	R(5)	O			M	O	G, R	2H
[57]	R(2)	A, O				L, E	V	2H
AFLGo[5]	R(?)	A, O	20		S	L	V/E	8H, 24H
VUzzer[44]	C(63), L, R(10)	A			G, S, O		N	6H, 24H
SlowFuzz[41]	R(10)	O	100		-		R/M/N	
Steelix[33]	C(17), L, R(5)	A, V, O			C, G	L, E, M	N	5H
Skyfire[53]	R(4)	O			?	L, M	R, G	LONG
kAFL[47]	R(3)	O	5		C, G*		V	4D, 12D
DIFUZE[13]	R(7)	O			G*		G	5H
Orthrus[49]	G(4), R(2)	A, L, O	80	C	S, G*		V	>7D
Chizpurfle[27]	R(1)	O			G*		G	-
VDF[25]	R(18)				C	E	V	30D
QuickFuzz-2[21]	R(?)	O	10		G*		G, M	
IMF[23]	R(1)	O			G*	O	G	24H
[59]	S(?)	O	5		G		G	24H
NEZHA[40]	R(6)	A, L, O	100		O		R	
[56]	G(10)	A, L					V	5M
S2F[58]	L, R(8)	A, O			G	O	N	5H, 24H
FairFuzz[32]	R(9)	A	20	C		E	V/M	24H
Angora[10]	L, R(8)	A, V, O	5		G, C	L, E	N	5H
T-Fuzz[39]	C(296), L, R(4)	A, O	3		C, G*		N	24H

Notations for the table

- Benchmarks: what programs they have run
  - R: real-world program (utils programs (string input), image processing programs (file input))
  - C: CGC data set (artificial programs and bugs)
  - L: LAVA-M benchmark (artificial programs and bugs)
  - S: programs with manually injected bugs
  - G: Google fuzzer test suites (real programs and bugs)
- Baseline: which fuzzers they compare against
- Trials: the number of times the fuzzers are run
- Crash: whether the crashes trigger the same bug?
  - G: group crashes based on the stack
  - O: use tools to group
  - C: coverage profile (afl)
  - G: based on the ground truth
  - G*: based on the manual diagnosed ground truth
- Coverage: line (L), branch (E) and method (M)
- Seed:
  - R: randomly sampled seed
  - M: manually constructed seed
  - N/V: non-empty seed given
  - G: generated

Some findings through studying these tools

- performance of a fuzzer is largely dependent on the seed, variant seeds should be used, including the empty seed
- runtime ($>$24 hours) for a valid study, e.g., objdump: takes 6 hours to find the first bug
- measure fuzzer performance: coverage, bug trigger ability
- patch the bug and then the fuzzer should run again to see if the crashes happen

Fuzzing algorithm [CCS 2018]

```
corpus ← initSeedCorpus()
queue ← ∅
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
    end if
end while
```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Parameters and design decisions

- **initSeedCorpus**:
  1. use grammar to specify the valid, interesting input
  2. static analysis to identify the structure of the inputs
  3. given a few valid inputs by users
- **mutate**:
  1. use symbolic executor to determine the number of bits of a seed to mutate
  2. data based mutation: bit flipping, random based mutation
  3. use taint analysis to determine how the bit is used, and mutate these important bits
  4. use the parser of the input in the program as a mutator

Parameters and design decisions

- **eval**:
  1. use symbolic execution to guide the mutated seeds to reach branches
  2. transform the program and remove checks on the input to force to reach the new code
  3. runtime analysis to detect errors
- **isInteresting**:
  1. crash
  2. long running time
  3. static analysis before fuzzing to determine for each statement the chance of leading to the deep paths and vulnerabilities
  4. reach a particular area?

AFL

- AFL found 76% more bugs than the baseline (68 vs.16) in the same corpus over a 24 hours period [2016 NDSS]
- Fuzzing performance is different from run to run so rerun it if no crashes found
- AFL using dictionaries of input (generation based fuzzing), also using instrumentation-guided (dynamic information) genetic algorithm (grey box based fuzzing)
- Additional tools: visualization, minimize the test cases, help diagnose crashes – find reachable code from crashes

AFL

Internal

1. Load user-supplied initial test cases into the queue
2. Take next input file from the queue,
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

AFL

Work for instrumented code (so we know which branches are taken during execution):

- set path: CC=/path/to/afl/afl-gcc for c code; CXX=/path/to/afl/afl-g++ for c++; there are other compiles beside gcc, e.g., afl-clang
- prepare compilation: ./configure CC=afl-gcc CXX=afl-g++ LD=afl-gcc–disable-shared
- test library: you will need to statically link the instrumented library
- AFL also works with instrumented binary code using qemu (binary analysis tools)

AFL

Initial test:

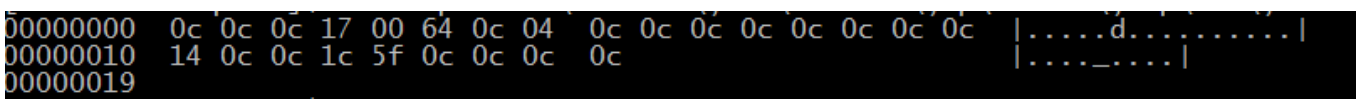
- you can start with one or many
- the input is through "read" read (0, buf, 8) – read 8 bytes from std-in
- ./afl-fuzz -i testcase_dir -o findings_dir – /path/to/tested/program [...params...]
- ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
  For programs that take input from a file, use '@@'

AFL

more info can be found
https://lcamtuf.coredump.cx/afl/README.txt

Output:

- To enable the use of Address Sanitizer you need to set the environment variable AFL_USE_ASAN to 1 during compilation
- output including three folders: queue, crashes and hangs
- test input generated may be not directly readable by you, they are binary files, using hexdump -c :
  - left column is the address of the first byte of the line, in hex.
  - middle column is the bytes, 16 bytes per line.
  - right column is text for the bytes, if printable.



AFL

extend AFL with improved test oracle: modifying the target programs to call abort() when

- Two bignum libraries produce different outputs when given the same fuzzer-generated input,
- An image library produces different outputs when asked to decode the same input image several times in a row,
- A serialization / deserialization library fails to produce stable outputs when iteratively serializing and deserializing fuzzer-supplied data,
- A compression library produces an output inconsistent with the input file when asked to compress and then decompress a particular blob.

References and Further Reading

1. Evaluating fuzz testing [CSS 2018]
2. Driller: Augmenting Fuzzing Through Selective Symbolic Execution (compare fuzzing and symbolic execution) [NDSS 2016]
3. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool For Developer Testing
4. Synthesizing program input grammars [PLDI 2017]