libFuzzer

Kostya Serebryany <kcc@google.com>
May 2016
Dynamic Tools @ Google

● Sanitizers (dynamic bug detectors)
  ○ AddressSanitizer (ASan) + LeakSanitizer (LSan)
  ○ ThreadSanitizer (TSan)
  ○ MemorySanitizer (MSan)
  ○ UndefinedBehaviorSanitizer (UBSan)

● Fuzzing
  ○ Coverage instrumentation for fuzzing
  ○ libFuzzer

● Kernel testing and fuzzing
  ○ KASAN, KTSAN, Syzkaller

● Compiler-based hardening
  ○ Control Flow Integrity
  ○ SafeStack
Fuzzing is...

- Automatically generate lots of test inputs:
  - crash your code
  - increase code coverage
Vocabulary

- **Target** -- a function that:
  - Consumes an array of bytes
  - Does something non-trivial with these bytes (e.g. parses them)

- **Engine (fuzzer):**
  - A tool that feeds a fuzz target with different random inputs
  - In-process or out-of-process

- **Corpus**
  - A set of valid & invalid inputs for the target
  - Collected manually, by fuzzing, or by crawling
  - Potentially minimized (minimal corpus size with the same coverage)
# Target example: boringssl/+/master/fuzz/privkey.cc

```c
#include <openssl/evp.h>

extern "C"

int LLVMFuzzerTestOneInput(const uint8_t *buf, size_t len) {
    const uint8_t *bufp = buf;
    EVP_PKEY_free(d2i_AutoPrivateKey(NULL, &bufp, len));
    return 0;
}
```
Fuzzing strategies

- Grammar-based Generation
  - Generate random inputs according to grammar rules

- Blind mutations
  - Collect a corpus of representative inputs, apply random mutations to them

- Guided evolutionary mutations
  - Build the target code with coverage instrumentation
  - Run the target on the initial test corpus, collect coverage
  - Run the target on random mutations of the elements of the corpus
  - If new coverage is discovered add the mutation back to the corpus
    - Repeat c.
Guided evolutionary fuzzing engines

- libFuzzer
- AFL
- Honggfuzz
- ...

- go-fuzz for Go programs
libFuzzer overview

- Open-source library, part of LLVM
- Relies on compiler instrumentation to get coverage feedback
- Links with the target
- Works fully in-process
- Usually, should be combined with ASan, MSan, or UBSan

- Demo
Tiny

51  221  1823 FuzzerCrossOver.cpp
424  1383  13309 FuzzerDriver.cpp
140   424  3793 FuzzerIO.cpp
773  2302  24859 FuzzerLoop.cpp
 26    97   943 FuzzerMain.cpp
 275   977  9413 FuzzerMutate.cpp
 202   679  5114 FuzzerSHA1.cpp
  63   191  1715 FuzzerTracePC.cpp
 630  2839 24047 FuzzerTraceState.cpp
 269   952  7225 FuzzerUtil.cpp
  61   197  1836 FuzzerDFSan.h
 186   674  5630 FuzzerFnAdapter.h
  58   291  2198 FuzzerInterface.h
 454  1682 14835 FuzzerInternal.h
  37   137  1277 FuzzerTracePC.h

3649 13046 118017 total
Simple

svn co http://llvm.org/svn/llvm-project/llvm/trunk/lib/Fuzzer

clang++ -c -g -O2 -std=c++11 Fuzzer/*.cpp -IFuzzer

ar ruv libFuzzer.a Fuzzer*.o
OS support

- Linux -- primary platform
- OSX -- works well
- Other -- volunteers welcome!
Has tests (puzzles)

- Fuzzing is random-based, so reliable testing is fun
- Looking for new puzzles!

```c
static volatile int *Null = 0;
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  if (Size > 0 && Data[0] == 'H') {
    if (Size > 1 && Data[1] == 'i') {
      if (Size > 2 && Data[2] == '!' ) {
        std::cout << "Dereferencing NULL\n";
        *Null = 1;
      }
    }
  }
  return 0;
}
```
Depends on compiler

● Build libFuzzer itself with any compiler
  ○ no special flags

● Build the target code with fresh Clang using
  ○ one of the sanitizers (asan, msan, ubsan) and
  ○ -fsanitize-coverage=edge[,8bit-counters,trace-cmp,indirect-calls]
  ○ Usually with -O1 or -O2 for speed
  ○ But -O0 may be better for fuzzing :-(
Corpus

- A fuzzer accepts zero or more corpus dirs
  - ./my-fuzzer
  - ./my-fuzzer DIR1
  - ./my-fuzzer DIR1 DIR2 DIR3

- All corpus dirs will be used as initial seeds

- New inputs will be written to DIR1 (primary corpus)

- DIR2, etc are read-only
As a regression test

- If given input files (not dirs) just runs them w/o fuzzing
  - ./my-fuzzer input-file1 input-file2 input-file3
  - Or: ./my-fuzzer -runs=0 my-corpus-dir

- When fixing a bug don’t forget to add the reproducer to the test suite
**Flags**

- All flags look like `-flag=value (single '-')`
- `-help=1` prints all flags
- Everything with `--` is ignored
  - You may use `--flags` to control the target behavior
  - But it’s better to have several targets
Maximal input length

- When using a corpus, the length of inputs is limited by the size of the largest element in the corpus.
- Without corpus, the default max length is 64 bytes.
- Override with -max_len=N
Many CPUs

- **-jobs=N**: runs N jobs total, up to #CPUs/2 simultaneously
  - Starts a new job after some job dies

- **-jobs=N -workers=M**: run N jobs total, M jobs simultaneously

- Each process periodically re-reads the primary corpus dir, i.e. fuzzing processes cooperate
INFO: Seed: 1118835970

#0  READ  units: 2 exec/s: 0

#2  INITED cov: 2 bits: 2 units: 1 exec/s: 0

#1024  pulse  cov: 2 bits: 2 units: 1 exec/s: 1234

#3730  NEW  cov: 3 bits: 3 units: 2 exec/s: 1200 L: 5 MS: 3 ChangeBit-...

#3793  NEW  cov: 4 bits: 4 units: 3 exec/s: 1205 L: 1 MS: 1 CrossOver-

#4327  NEW  cov: 5 bits: 5 units: 4 exec/s: 1209 L: 8 MS: 5 ShuffleBytes-

#6594  NEW  cov: 6 bits: 6 units: 5 exec/s: 1199 L: 2 MS: 2 EraseByte-
Failure

- On every kind of failure:
  - Prints stack trace (more info, if available)
  - Dumps the reproducer on disk
  - Exits the process
ASan, MSan, UBSan

- Reports failure on any `{ASan, MSan, UBSan}`-ish bug
- With UBSan, use `-fno-sanitize-recover=undefined` (crash on first bug)

```plaintext
#0  READ  units: 1 exec/s: 0
#1  INITED cov: 3 bits: 3 units: 1 exec/s: 0
#2  NEW   cov: 4 bits: 4 units: 2 exec/s: 0 L: 64 MS: 0
=================================================================
==5799==ERROR: AddressSanitizer: heap-buffer-overflow ...
READ of size 1 at 0x60200000eaf3 thread T0
    #0 0x4e34d3 in LLVMFuzzerTestOneInput
artifact_prefix='./'; Test unit written to ./crash-94d56771416d9bdc058af41360e9ce01a8efdb84
```
Signals

- **SIGSEGV, SIGBUS, SIGABRT, SIGILL, SIGFPE**
  - Reports failure

  ```
  ==7667==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
  #0 0x4e36da in LLVMFuzzerTestOneInput
  artifact_prefix='./'; Test unit written to ./crash-c07078879f59203eeb77b1e2390b60cde5634ce6
  ```

- **SIGINT, SIGTERM**
  - Dies peacefully
Timeouts

- `-timeout=N`  
- Sets SIGALRM  
- Checks the time since the current unit started  
- Reports failure on timeout

Artifact prefix='./'; Test unit written to ./timeout-c0a0ad26a634840c67a210fefd0d76577b03a111

```
==7518== ERROR: libFuzzer: timeout after 3 seconds
    #1 0x4e976a in fuzzer::Fuzzer::AlarmCallback() ...
    #3 0x4e35fb in LLVMFuzzerTestOneInput
```
Leaks

- `-detect_leaks=1` (default, requires LeakSanitizer/AddressSanitizer)
- Counts the number of malloc and free calls for every input
- If the numbers don’t match, runs the same input again
- If the numbers are different again, invokes leak checking
- Reports failure on a leak

```
==7958==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 4 byte(s) in 1 object(s) allocated from:
  #0 0x4e105b in operator new(unsigned long)
  #1 0x4e324d in LLVMFuzzerTestOneInput
artifact_prefix='./'; Test unit written to ./leak-b26cdc971c9b551a66ce96d49bd8a070f5f3ce0a
```
OOMs

- With ASan and MSan can not rely on RLIMIT_AS :-(
- OOMs may kill your machine
- -rss_limit_mb=N, default is 2048
- Checks getrusage::ru_maxrss in a separate thread
- Reports failure when limit is exceeded

```
==8686== ERROR: libFuzzer: out-of-memory (used: 2840Mb; limit: 2048Mb)
  #5 0x4e3599 in LLVMFuzzerTestOneInput ...
artifact_prefix='./'; Test unit written to ./oom-c0a0ad26a634840c67a210fefdda76577b03a111
```
Threads

- Threads in the target functions are ok, but ...
  - Coverage data may depend on thread timing, i.e. may be unreliable
  - Extra slowdowns
  - Threads that outlive the target function invocation produce unrelated coverage
  - Thread pools might be ok … but we have little data

- Avoid threads if you can
Logging spam

● A printf deep inside your target function…
  ○ May slow down fuzzing by 10x and more
  ○ May consume lots of disk space

● `--close_fd_mask=[0123]` will close stdout (=1) stderr (=2) or both (=3)

● libFuzzer’s own output will survive

● ASan/MSan/UBSan’s output will be gone (bug, needs fixing)
  ○ Assert failures too

● Avoid printfs in your target!
Minimize or merge corpora

- Merge one or more corpus dirs into the primary corpus
  - `.\my-fuzzer mycorpus newinputs -merge=1`
  - Will only add items that add extra coverage

- Minimize the existing corpus by merging into an empty dir

- Full minimization is NP-hard, but a linear algorithm is a good approximation
Parasitic coverage

- Run a fuzzer for a while, let it converge (stop finding new coverage)
- Restart with the same corpus -- starts with smaller coverage and grows again

- Target produces different coverage on the same input
  - RNG in the target code (e.g. libxml uses RNG for hashing)
  - Using pointer values for hashing -- different code in a hash table is touched

- `print_new_cov_pcs=1` will print all newly covered PCs
CRC

- The target has consistency checking
  - PNG: checks CRC for every chunk

- libFuzzer can crack these sometimes
  - “automatic dictionaries” (below)

- But still very inefficient, need to bypass
Fuzzing build mode

- Proposed common macro:
  FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION

#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION

// Disable RNG, Crypto, and CRC

#endif
Instrumentation Feedback

- Coverage is used to choose interesting mutations
  - Control Flow edges (boolean): \(-\text{fsanitize-coverage}=\text{edge}\)
  - Control Flow edges (counters): \(-\text{fsanitize-coverage}=\text{edge, 8bit-counters}\)
  - Indirect caller-callee pairs: \(-\text{fsanitize-coverage}=\text{edge, indirect-calls}\)
  - Experimental: execution paths, up to 10 edges: \(-\text{fsanitize-coverage}=\text{trace-pc}\)

- Instrumentation is used to guide mutations
  - memcmp/strcmp
    - run-time interception, cheap
  - Cmp instructions
    - compile-time, expensive
    - \(-\text{fsanitize-coverage}=\text{trace-cmp}\)
Manual dictionaries

- -dict=<file>
- Token-based (xml, C++, SQL, PDF, …) or magic-based (png) inputs
- Dictionary entries are injected into inputs while mutating
- Huge speedup in fuzzing when applicable, the idea is stolen from AFL

# Lines starting with '#' and empty lines are ignored.
# Adds "blah" (w/o quotes) to the dictionary.
kw1="blah"
# Use \ for backslash and " for quotes.
kw2="\ac\dc"
# Use \xAB for hex values
kw3="\xF7\xF8"
# the name of the keyword followed by '=' may be omitted:
"foo\0Abar"
Automatic dictionaries

- The fuzzer intercepts `memcmp`, `strcmp`, etc.
- Inserts the `memcmp` arguments into a temporary dictionary.
- If an entry from the temporary dictionary causes a good mutation, the entry is added to the per-process permanent dictionary.
- This feature can be used to generate manual dictionaries (experimental).

- `--fsanitize-coverage=trace-cmp` (compile-time) + `--use_traces=1`
  - Acts as if every cmp instruction is `memcmp`, very slow.
  - Sometimes does miracles.
Mutations

- Erase/Insert/Change/Shuffle Bit/Byte/Bytes
- CrossOver (aka splice)
- Inject token from a dictionary
- Change an ASCII integer (e.g. 123 => 2465357635)
- Easy to add more
User-defined mutators

- Imagine fuzzing a service that consumes valid protocol buffers
- Naive protobuf fuzzing will only stress the parser
- Solution: user-defined mutator

```c
// Optional user-provided custom mutator.
// Mutates raw data in [Data, Data+Size) inplace.
// Returns the new size, which is not greater than MaxSize.
// Given the same Seed produces the same mutation.
size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size, size_t MaxSize,
                               unsigned int Seed);
```
libFuzzer vs AFL

- AFL is amazing too!
  - AFL can reuse libFuzzer’s fuzzing targets.
  - libFuzzer can not reuse out-of-process AFL targets w/o extra work
  - Same corpus can be used for both

- libFuzzer
  - Is fully in-process, i.e. sometimes 100x faster
  - Tightly integrated with asan, msan, ubsan, lsan, and sanitizer coverage
  - Simpler to use and deploy (strictly IMHO)

- Different mutations and coverage instrumentation -- different results
  - One of them is stuck, the other still manages to find new coverage. And vice versa.

- Use both!
libFuzzer + AFL

- libFuzzer stole the idea and syntax of AFL’s dictionaries
- AFL introduced (partially) in-process mode after (because of?) libFuzzer
- AFL can now use compiler instrumentation: `-fsanitize-coverage=trace-pc`
  - Both Clang and GCC
In Chrome

- **Integrated** with Chrome source base and ClusterFuzz

- 85+ targets, 121+ bugs (75+ fixed). Since Q1’16
  - Many targets are added by the code owners
  - No effort to automate fuzzing for the code owner

- ASAN, MSAN, parts of UBSAN

- Later in 2016: AFL for the same targets
Other trophies

- GLIBC: https://sourceware.org/glibc/wiki/FuzzingLibc
- MUSL LIBC
- pugixml
- PCRE: Search for “LLVM fuzzer” in ChangeLog; also in bugzilla
- ICU
- Freetype
- Harfbuzz
- SQLITE
- Python
- Linux Kernel’s BPF verifier
- Capstone: [1] [2]
- Radare2: [1]
- WOFF2: [1]
- LLVM: Clang, Clang-format, libc++, llvm-as