Fuzzing

Fuzzing

Fuzzing or **fuzz testing** is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.

The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

American Fuzzy Lop (AFL)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.

- **AFL++**
  - AFL++ (AFLplusplus) is a community-maintained fork of AFL created due to the relative inactivity of Google's upstream AFL development since September 2017. It includes new features and speedups.
  - You can use AFL++ rather than original AFL in this homework.

Install

```
git clone https://github.com/google/AFL.git
make && make install
```

- **Fast LLVM-based instrumentation for afl-fuzz**
  - AFL provides another compile tool that could speed up fuzzing. More information please see llvm_mode/README.md
  - Install afl-clang-fast
    - `apt install clang`
    - `cd llvm_mode/`
    - `LLVM_CONFIG=llvm-config-10 make`
    - `cd ..`
    - `make install`

Fuzzing step

- Choose a target
- Write a harness for fuzzing
- Compile the target and the harness with afl-gcc/afl-clang-fast
- Get some data as input seeds of fuzzing
- Start fuzzing... (wait for finding crashes)
- Analysis the crashes

Harness

Instead of fuzzing all source code, we just fuzz some vulnerable funtions. For that, we usually writes a test program to call that functions.

The harness should:

- Run in foreground
- Usefully process input on stdin (or a specified file)
- Feed input to your target
- Exit cleanly
- Sometimes we would directly modify the source code of target for fuzzing

Initial Test Cases (Seeds)

A good seed would help fuzzer find more interesting paths, which may found vulnerability more quickly.

In general, we would find some real inputs that exercise as much of the target as possible.

From the README

- Keep the files small. Under 1 kB is ideal, although not strictly necessary.
- Use multiple test cases only if they are functionally different from each other. There is no point in using fifty different vacation photos to fuzz an image library.

Tips

There are some method making fuzzing effective

- LLVM Mode (`afl-clang-fast`)
  - **Deferred forkserver**: tell AFL where to start each new process
  - **Persistent mode**: don’t fork for each run, just loop around this bit of code
  - See `llvm_mode/README.md`
- Sanitizers
  - Spot things you otherwise wouldn’t
  - The most well known is **ASAN** (AddressSanitizer)
  - To use ASAN, set `AFL_USE_ASAN=1` while compiling

Tips (cont.)

- Hardening
  - The setting is useful for catching non-crashing memory bugs at the expense of a very slight (sub-5%) performance loss.
  - Set `AFL_HARDEN=1` while compiling
- Parallel Fuzzing
  - See `docs/parallel_fuzzing.txt`
- Dictionaries
  - Help the fuzzer to access paths it otherwise wouldn’t
  - https://github.com/AFLplusplus/AFLplusplus/tree/stable/dictionaries
  - https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/fuzzers/dicts

Example

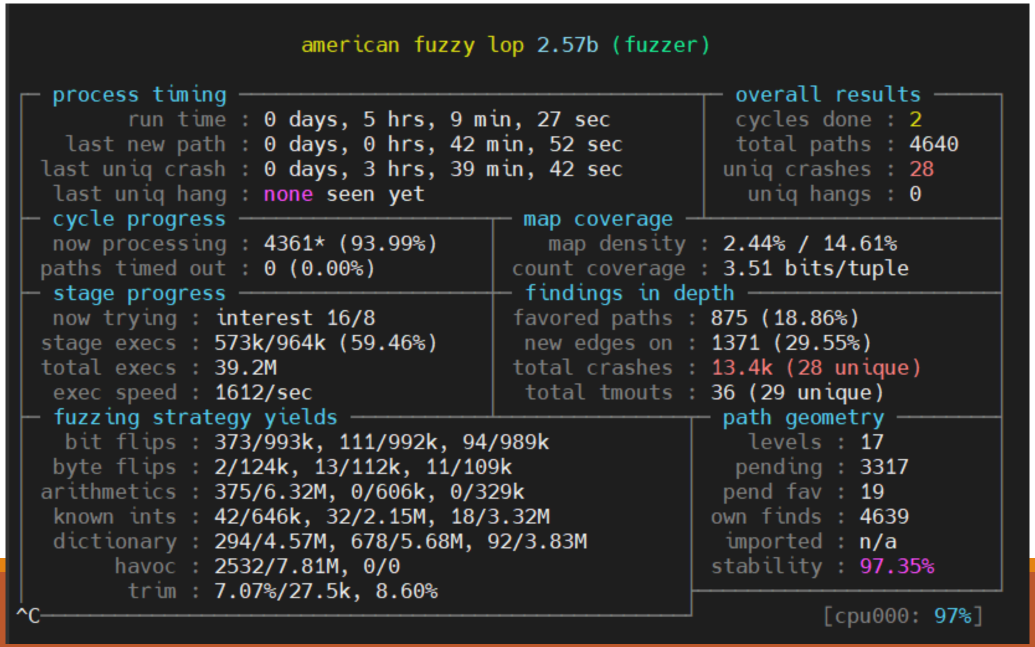
libxml2 is a popular XML library. Let's try and find **CVE-2015-8317**.

1. Configure and build libxml2 with ASAN
   - `CC=afl-clang-fast ./autogen.sh`
   - `AFL_USE_ASAN=1 make -j 4`
2. Compile the harness
   - `AFL_USE_ASAN=1 afl-clang-fast ./harness.c -I libxml2-2.9.2/include/ libxml2-2.9.2/.libs/libxml2.a -lz -lm -o fuzzer`
3. Create input seeds
   - `mkdir in`
   - `echo "<hi></hi>" > in/hi.xml`

Example (cont.)

4. Start fuzzing
   - `afl-fuzz -i in -o out -m none -x ./xml.dict ./fuzzer @@`
   - According to `docs/notes_for_asan.txt`, ASAN on 64-bit systems requests a lot of memory in a way that can't be easily distinguished from a misbehaving program bent on crashing your system. So we use `-m none` to make fuzzer not limit memory used. But it may allocate a huge amount of memory. See `docs/notes_for_asan.txt` for other solutions for this problem.

5. Wait for the fuzzer finding crashes



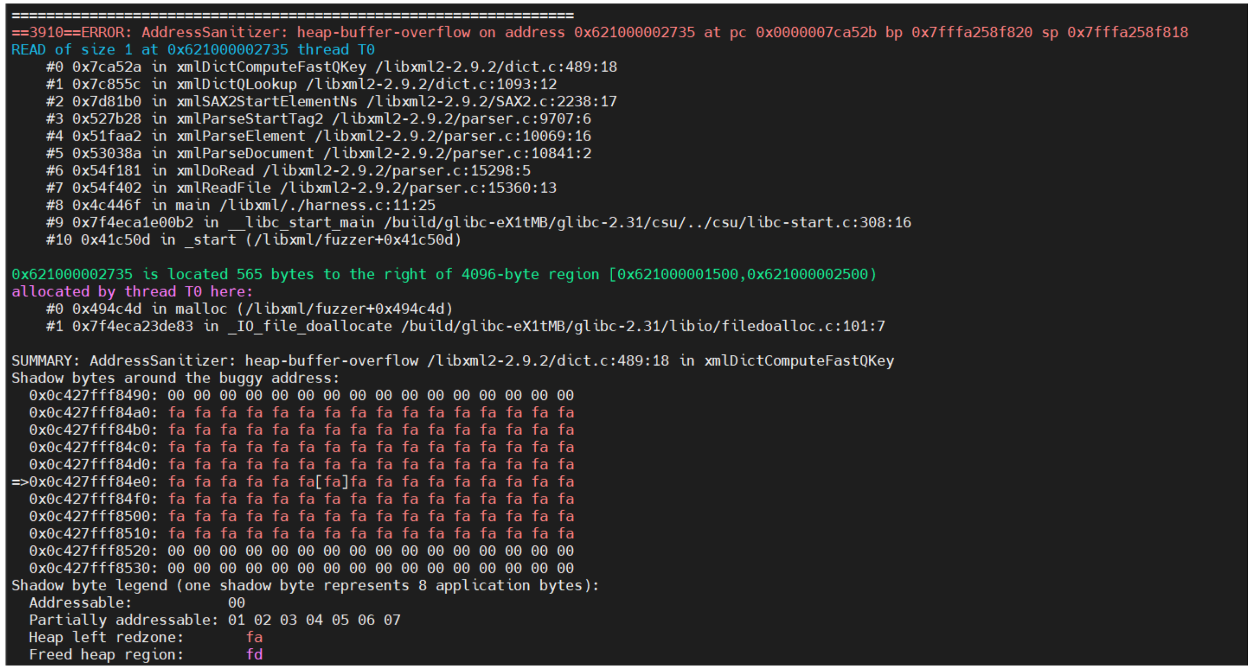
```
                    american fuzzy lop 2.57b (fuzzer)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 5 hrs, 9 min, 27 sec       │  cycles done : 2     │
│   last new path : 0 days, 0 hrs, 42 min, 52 sec      │  total paths : 4640  │
│ last uniq crash : 0 days, 3 hrs, 39 min, 42 sec      │ uniq crashes : 28    │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 4361* (93.99%)    │    map density : 2.44% / 14.61%      │
│ paths timed out : 0 (0.00%)         │ count coverage : 3.51 bits/tuple     │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : interest 16/8         │ favored paths : 875 (18.86%)         │
│ stage execs : 573k/964k (59.46%)    │  new edges on : 1371 (29.55%)        │
│ total execs : 39.2M                 │ total crashes : 13.4k (28 unique)    │
│  exec speed : 1612/sec              │  total tmouts : 36 (29 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 373/993k, 111/992k, 94/989k           │    levels : 17         │
│  byte flips : 2/124k, 13/112k, 11/109k              │   pending : 3317       │
│ arithmetics : 375/6.32M, 0/606k, 0/329k             │  pend fav : 19         │
│  known ints : 42/646k, 32/2.15M, 18/3.32M           │ own finds : 4639       │
│  dictionary : 294/4.57M, 678/5.68M, 92/3.83M        │  imported : n/a        │
│       havoc : 2532/7.81M, 0/0                       │ stability : 97.35%     │
│        trim : 7.07%/27.5k, 8.60%                    ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000: 97%]
```

Example (cont.)

6. Analysis the crashes
   - AFL will put the found crash in `out/crashes`. You can feed it to the harness with gdb.

- According to the debug information, we know where the crash occurred. After tracing the source code, we can know it is **CVE-2015-8317**.
- You may find other crashes after fuzzing, you can try to analysis why these crash occurred. (It might a CVE or not)



```
=================================================================
==3910==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x621000002735 at pc 0x0000007ca52b bp 0x7fffa258f820 sp 0x7fffa258f818
READ of size 1 at 0x621000002735 thread T0
    #0 0x7ca52a in xmlDictComputeFastQKey /libxml2-2.9.2/dict.c:489:18
    #1 0x7c855c in xmlDictQLookup /libxml2-2.9.2/dict.c:1093:12
    #2 0x7d81b0 in xmlSAX2StartElementNs /libxml2-2.9.2/SAX2.c:2238:17
    #3 0x527b28 in xmlParseStartTag2 /libxml2-2.9.2/parser.c:9707:6
    #4 0x51faa2 in xmlParseElement /libxml2-2.9.2/parser.c:10069:16
    #5 0x53038a in xmlParseDocument /libxml2-2.9.2/parser.c:10841:2
    #6 0x54f181 in xmlDoRead /libxml2-2.9.2/parser.c:15298:5
    #7 0x54f402 in xmlReadFile /libxml2-2.9.2/parser.c:15360:13
    #8 0x4c446f in main /libxml/./harness.c:11:25
    #9 0x7f4eca1e00b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16
    #10 0x41c50d in _start (/libxml/fuzzer+0x41c50d)

0x621000002735 is located 565 bytes to the right of 4096-byte region [0x621000001500,0x621000002500)
allocated by thread T0 here:
    #0 0x494c4d in malloc (/libxml/fuzzer+0x494c4d)
    #1 0x7f4eca23de83 in _IO_file_doallocate /build/glibc-eX1tMB/glibc-2.31/libio/filedoalloc.c:101:7

SUMMARY: AddressSanitizer: heap-buffer-overflow /libxml2-2.9.2/dict.c:489:18 in xmlDictComputeFastQKey
Shadow bytes around the buggy address:
  0x0c427fff8490: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fff84a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff84b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff84c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff84d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c427fff84e0: fa fa fa fa fa fa[fa]fa fa fa fa fa fa fa fa fa
  0x0c427fff84f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff8500: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff8510: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c427fff8520: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c427fff8530: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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
```