Design and Implement Binary Fuzzing based on LibFuzzer

Chun-Ying Huang, Wei-Chieh Chao, Si-Chen Lin, Yi-Hsien Chen
Department of Computer Science, College of Computer Science, National Chiao Tung University

***Abstract*—Libfuzzer is a coverage-guided fuzzing engine. But not like AFL, which can do the fuzzing without source code, Libfuzzer needs source code of the target to compile and do the fuzzing. This paper presents a way to use Libfuzzer to fuzz the binary without source code. Our implementation link the IO of binary with Libfuzzer, and use qemu to collect runtime information.**

I. Introduction

Sample citations [1], [2], [3], [4].

II. Related Work

III. Methodology

A. Link IO together

Normally, when using Libfuzzer, we put the source code need to be test inside `LLVMFuzzerTestOneInput` as describe in the introduction section. But when we want to run a binary, we don't have source code. Hence, what we do is actually putting `execv("our-binary", ..., ...)` inside the `LLVMFuzzerTestOneInput` to invoke our binary. And before execute `execv` we need to use dup2 to link

1. fuzzer test input → binary stdin.
2. binary stdout → `/dev/null`.
3. binary stderr → `/dev/null`

The simplified code is something like below

```
extern "C" int LLVMFuzzerTestOneInput(const
    uint8_t *Data, size_t Size) {

    int P_IN[2]; pipe(P_IN);
    if(Size) write(P_IN[1], Data, Size);
    ...
    int pid = fork();
    if(pid == 0) {
        dup2(P_IN[0], STDIN_FILENO);
        dup2(dev_null, STDOUT_FILENO);
        dup2(dev_null, STDERR_FILENO);
        ...
        execv(..., ..., ...);
    }
    ...
}
```

B. Collect Runtime Information

After we link IO together, the binary should get the input of the libfuzzer test input now. Though it is runnable, Libufuzzer don't have any runtime information like code coverage and will stop running after a few rounds. First, we need to figure out what Libfuzzer collect while running. Libfuzzer collect runtime information through **Clang SanitizerCoverage**, which provides simple code coverage instrumentation and has hook function for customization. Now we use qemu to collect right information for those hook function implemented by LibFuzzer.

1) trace pc guard: LibFuzzer use two array to store the information of code coverage.

1. `__sancov_trace_pc_pcs` : store the program counter of the beginning of a code block.
2. `__sancov_trace_pc_guard_8bit_counters` : store how many times the code block was hit.

```
__sancov_trace_pc_pcs[Idx] = PC;
__sancov_trace_pc_guard_8bit_counters[Idx]++;
```

We need to fill those array for LibFuzzer to work. First, we use radare2 api **r2pipe** to do static analysis and index those code block. While doing static analysis with r2pipe, we filter out addresses we don't want, like library code. Second, we use qemu to check whether we hit a code block, and fill in those value. The qemu command we use is `qemu-x86_64 -D logfile -d in_asm binary`. `in_asm` provides us the assembly code and its address being executed. (see example below)

```
...
----------------
IN:
0x0000004000802c90:  mov    %r11,%rcx
0x0000004000802c93:  sub    %rax,%rcx
0x0000004000802c96:  cmp    $0xb,%rcx
0x0000004000802c9a:  ja     0x4000802dd8
...
```

After parsing qemu output and fill in those value, LibFuzzer is now runnable. But we still got plenty of hook function we need to implement to make it works exactly like it should be.

IV. Evaluation

In this section, we will compare the performance of original Libfuzzer, our Libfuzzer and AFL. We use fuzzer-test-suite, a set of tests for fuzzing engines provided by Google, as the test cases. In order to support our Libfuzzer, we modify the `target.cc` in some test cases. Take `c-ares-CVE-2016-5180` for example, we modified the `extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)` function declaration to `int main()`. And use `read` function to read input data, the size is the length of input data. Check out more detail of patching fuzzer-test-suite in wiki page of our repository. The following specification is our machine information and all experiments were performed on it.

```
virtual private server on Google Cloud Platform
n1-standard-1 (one vCPU, 3.75 GB RAM)
```

A. *Compared with original Libfuzzer*

B. *Compared with AFL*

V. Conclusion

References

[1] K. Serebryany, "libFuzzer a library for coverage-guided fuzz testing," LLVM project, 2015, https://llvm.org/docs/LibFuzzer.html.

[2] R. Swiecki, "honggfuzz," online, 2010, http://honggfuzz.com/.

[3] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2155–2168.

[4] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," 2017, USENIX Security. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany