HW3 - Fuzz Testing (libfuzzer)

Submission deadline: June 8 - 23:59:59 KST

In the previous HW2, we have seen AFL - a coverage guided fuzzer especially suited to fuzzing file parsers. However, AFL isn't designed to fuzz other types of interfaces. In this HW3, we will study libFuzzer which can fuzz arbitrary APIs.

Your task in this HW3 is to set up libFuzzer and to use it to find bugs in our PNG parsing library.

libFuzzer

libFuzzer comes built-in with the `clang` compiler. It is a coverage-guided mutational fuzzer. Mutational fuzzers randomly mutate existing input to generate new inputs. Coverage-guided fuzzers use newly discovered code paths to determine which inputs are better.

Sanitization is a process of instrumenting memory allocation and accesses to prevent memory bugs. While AFL can work with a sanitizer, it can lead to false memory exhaustion bugs - sanitizers allocate a lot of memory. libFuzzer natively supports sanitizers, and their usage is recommended.

Unlike AFL, libFuzzer requires some setup. The user needs to write fuzzing stubs for every interface they want to test. These stubs take random input from the fuzzer and use it to call the interface being tested. This makes libFuzzer more versitile than AFL - it can test all functions, not only the ones that parse files.

Files

- `libfuzzer`
  - `src`
    - `crc.h`
    - `crc.c`
    - `pngparser.h`
    - `pngparser.c`
    - `fuzzer_load_png.c`
    - `Makefile`
- `seeds`
  - `rgba.png`
  - `palette.png`

The source code for this lab consists of a library we will be fuzzing (`crc.h`, `crc.c`, `pngparser.h`, `pngparser.c`), and a test harness (`fuzzer_load_png.c`). The directory also consists of a Makefile with some of the targets not implemented.

Setting up libFuzzer

1. If it's not already, install `clang` on your machine (`sudo apt install clang` on Ubuntu/Debian/Mint)
2. Open `Makefile`
    1. `CFLAGS` stores the compilation flags.
        1. `-g` compiles debug version of the binaries
        2. `-fsanitize=fuzzer,address` compiles the binaries with the instrumentation (`fuzzer`) and ASan (`address`). Some valid values are:
            1. `-fsanitize=fuzzer` - compile without any sanitization
            2. `-fsanitize=fuzzer,address` - compile with ASan (Address Sanitizer) (We will use this for the lab)
            3. `-fsanitize=fuzzer,undefined` - compile with UBSan (Undefined Behavior Sanitizer)
            4. `-fsanitize=fuzzer,signed-integer-overflow` - compile with a part of UBSan
            5. `-fsanitize=fuzzer,memory` - compile with MSan (Memory Sanitizer)
    2. For every interface that we want to fuzz, we will write a separate libFuzzer stub - a separate fuzzing program.
        1. `fuzzer_load_png` is the fuzzing stub which loads an arbitrary image
        2. Notice `TO DO` lines. Your job will be to implement the missing stubs.
3. Open `fuzzer_load_png.c`
    1. Every libFuzzer stub contains `int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)`. It is the function that the fuzzer calls with the random data. `Data` contains the mutated data, while `Size` denotes its length.
    2. Our interface (`load_png`) expects a file, so we write our data into the file and pass it to the function.
        - N.B. If we run multiple instances of the fuzzer the filename will be the same.
    3. If the fuzzer passes less data than required by the inteface, the stub can just return 0.
    4. If the function takes multiple parameters, we can provide default values as other inputs.
    5. If we want to fuzz multiple parameters at the same time, we can manually split `Data` among them. (e.g. first 4 bytes denote an integer parameter, and the remaining N-4 bytes are the image buffer).
4. Run `make`
5. Run `./fuzzer_load_png -detect_leaks=0 <seed-directory>` and

wait for a crash

6. Fix the bug
    1. Run `gdb ./fuzzer_load_png`
    2. In GDB run: `run <crash-file>`
    3. Debugging in progress…
    4. `current_chunk->chunk_data` is never initialized to NULL, leading to it being freed in the error handling code.
    5. Initialize `current_chunk->chunk_data` to NULL.
    6. Execute `./fuzzer_load_png <crash_file>` to verify that the bug has been fixed
7. Continue fuzzing :)

The Task

1. You are required to implement fuzzers missing from the Makefile:
    1. `fuzzer_load_png_name`
    2. `fuzzer_store_png_rgba`
    3. `fuzzer_store_png_palette`
    4. `fuzzer_store_png_name`
2. Find, report, and fix 5 bugs in Y0L0 PNG
3. For each bug found, copy the libFuzzer output to a `crash.log` file and include it in your submission.

For this phase, we will consider ANYTHING that triggers a crash with ASan a bug: crashes, leaks, invalid frees, invalid accesses…

You are free to fix and report as many bugs as you like. Extra bugs will be used as replacements for the ones which aren't accepted. `png_chunk_valid` has been prepatched to always return 1.

Running `make all` should build all fuzzers with the names identical to their C source files (minus the extension).

Useful libFuzzer flags:

- `-detect_leaks=0` disables leak detection
- `-workers=N` sets the maximum number of concurrently executing processes
- `-runs=N` sets the total number of fuzzing runs (i.e. how many bugs we want to find)
- `-jobs=N` set the total number of fuzzing jobs. Runs are distributed among jobs. Maximally `workers` jobs can be active at a time.

Submission format

Submit a zip archive with the name `submission.zip`. It should have the following structure:

- `libfuzzer`
  - `src`
    - `crc.h`
    - `crc.c`
    - `pngparser.h`
    - `pngparser.c` - with your fixes and `is_png_chunk_valid` patched to return 1
    - `fuzzer_load_png.c`
    - `fuzzer_load_png_name.c`
    - `fuzzer_store_png_name.c`
    - `fuzzer_store_png_rgba.c`
    - `fuzzer_store_png_palette.c`
    - `Makefile`
  - `reports`
    - `DESCRIPTION_00.md`
    - `poc_00.bin`
    - `crash_00.log`
    - `DESCRIPTION_01.md`
    - `poc_01.bin`
    - `crash_01.log`
    - `DESCRIPTION_02.md`
    - `poc_02.bin`
    - `crash_02.log`
    - `DESCRIPTION_03.md`
    - `poc_03.bin`
    - `crash_03.log`
    - `DESCRIPTION_04.md`
    - `poc_04.bin`
    - `crash_04.log`

`poc.bin` is the file generated by libFuzzer which triggers the specific bug you found and fixed. Those are available in the seeds dir you provided to libFuzzer. Copy the PoC from the fuzzer's directory into your submission archive and rename it to poc.bin for each bug.

Please make sure that:

1. Your code compiles properly on Ubuntu 18.04
2. You have actually zipped all of your files
3. That you don't have previous versions of your files in the archive
4. You haven't included other students' work by mistake

Bug descriptions should follow the specification from HW1.

Sample Bug Report

Name

Uninitialized local variables

Description

The loop iteration counters, `i` and `j`, are not initialized and the behavior of the loop is, thus, undefined.

Affected Lines

In `filter.c:17` and `filter.c:18`

Expected vs Observed

We expect that the loops process over all the pixels in the image by iterating over every row, and every pixel in that row, starting from index 0.The loop counters are not initialized and are thus not guaranteed to start at 0. This makes the behavior of the grayscale filter undefined.

Steps to Reproduce

Command

```
./filter poc.png out.png grayscale
```

Proof-of-Concept Input (if needed)

(attached: poc.png)

Suggested Fix Description

Initialize the `i` and `j` counters to 0 in the loop setup. This allows the loop to iterate over all the image pixels to apply the grayscale filter.

Y0L0 PNG Format

Y0L0 PNG format is a subset of the PNG file format. It consists of a PNG file signature, followed by mandatory PNG chunks:

- IHDR chunk
- Optional PLTE chunk
- One or more IDAT chunks
- IEND chunk

All multibyte data chunk fields are stored in the big-endian order (e.g. 4-byte integers, CRC checksums). IHDR must follow the file signature. If the palette is used to denote color, PLTE chunk must appear before IDAT chunks. IDAT chunks must appear in an uninterrupted sequence. The image data is the concatenation of the data stored in all IDAT chunks. IEND must be the last chunk in an image. All other chunk types are simply ignored.

Y0L0 PNG File Signature

Y0L0 PNG files start with the byte sequence: `137 80 78 71 13 10 26 10`

Y0L0 PNG Chunks

Y0L0 PNG chunks have the following structure:

- Length (4 bytes) denotes the length of the data stored in the chunk
- Chunk type (4 bytes) identifies the chunk type (IHDR, PLTE, IDAT or IEND). The type is encoded as a 4 byte sequence.
- Chunk data (Length bytes) stores the actual chunk data
- CRC code (4 bytes) is a checkcode that is calculated over chunk type and chunk data

All fields are consecutive and in the given order.

IHDR Chunk

IHDR chunk must appear as the first chunk following the file signature. It has the type `IHDR` and the following structure of the chunk data:

- Width (4 bytes)
- Height (4 bytes)
- Bit depth (1 byte)
- Color type (1 byte)
- Compression method (1 byte)
- Reserved (2 bytes)

All fields are consecutive and in the given order.

Bit Depth

The only supported bit-depth is 8 bits (1 byte). This refers to 1 byte per color channel in the RGBA color mode, and to 1 byte per palette index in PLTE color mode.

Color Type

Color type field denotes the way color data is stored in the image. The only supported values are 3 (palette) and 6 (RGBA).

- Palette denotes that we expect to find a PLTE chunk in the image. In the IDAT chunk data colors are not represented as RGBA tuples, but as indices in the palette table. Every offset has the length of bit-depth. If the pixel has the color of 0x123456, and the palette has the color {R:0x12, G:0x34, B: 0x56} at the position 5, the value 5 will be stored in the image.
- RGBA mode represents colors in IDAT data as a sequence of 4 values per pixel, each one bit-depth in size. Every value corresponds to the intensity of a RGBA (red-green-blue-alpha) channel.

Compression Method

The only supported compression method is the deflate algorithm signified by value 0.

Reserved

Reserved for future use.

PLTE Chunk

PLTE chunk must appear before the IDAT chunk if the palette is used to encode color. Its type is encoded with `PLTE`. The chunk data is an array of PLTE entries which are defined as:

- Red (1 byte)
- Green (1 byte)
- Blue (1 byte)

The length field of the PLTE chunk needs to be divisible by 3.

IDAT Chunk

The type of the IDAT chunk is encoded by the bytes `IDAT`. If multiple IDAT chunks exist, they must all occur in sequence. The image data is the concatenation of the data stored in all the IDAT chunks in the order in which they appeared. It is compressed using the deflate algorithm and is stored in the zlib file format:

- Compression type and flags (1 byte)
- Flags and check bits (1 byte)
- Compressed data (n bytes)
- Adler-32 checksum (4 bytes)

This data is used as an input for the inflate algorithm and is handled by zlib. Inflated data is stored left-to-right, top-to-bottom. Every row (scanline) begins with a byte denoting the start of the line. This byte must be 0. Pixel information for the scanline follows this byte. It consists either of the palette index (1 byte), or of the RGBA pixel value (4 bytes), depending on the color format.

All fields of structures are consecutive and in the given order.

IEND Chunk

IEND chunk appears as the last chunk in the file. Its type is `IEND`. It stores no data and the length field is 0.