HW2 Instructions - Fuzz-testing (Fuzzing)

Submission deadlines:

- Deadline: 2025.11.20 - 23:59:59 KST

Fuzz-testing (Fuzzing)

Introduction

Fuzz-testing is a testing method where programs are bombarded with semi-valid input. Semi-valid inputs are those which pass non-interesting checks in the code (checksum, hash, etc.) to reach the potentially problematic code regions. Programs which perform the testing are called fuzzers. The two main components of a fuzzer are:

- an input generator
- a fuzzing oracle

The input generator provides the tested program with inputs which should trigger new, unexplored or unexpected behavior. A fuzzing oracle reports when some abnormal behavior has happened.

AFL (American Fuzzy Lop)

While it possible to test programs with completely random input, it is inefficient. Many of the test cases will cause the program to fail sanity checks, and exit before reaching important code segments. Fuzzers will, therefore, use various metrics and feedback to smartly choose inputs.

AFL uses simple machine learning for input generation (genetic algorithms). Successful inputs (those that lead to interesting behaviour) are kept and combined to produce more (hopefully) successful inputs. AFL can also mutate them (e.g. flip some bits), or generate random input (havoc mode) to add fresh genes to the population. Usually valid input files are provided as seeds (starting files) to the fuzzer.

What is AFL's measure of success for an input? It is new code coverage. It has been determined that the number of bugs correlates strongly with the amount of code. By exploring previously unexecuted code, the fuzzer has a higher chance of encountering previously unseen bugs.

The fuzzing oracle for AFL uses crashes as indicators for bugs. If the program crashes, AFL knows that it has encountered a bug. However, one bug can cause crashes for many different inputs. AFL performs *deduplication* to group crashes based on their cause. This isn't always successful and the number of reported unique crashes is usually inflated.

HW2 - Phase 1 (AFL)

Your task in this phase of HW2 is to set up AFL and to use it to find bugs in our PNG parsing library.

Files

- `/src`
  - `crc.h`
  - `crc.c`
  - `pngparser.h`
  - `pngparser.c`
  - `size.c`
  - `Makefile`
- `/seeds`
  - `rgba.png`
  - `palette.png`
- `INSTRUCTION_AFL.md`

The source code for this HW consists of a library we will be fuzzing (`crc.h`, `crc.c`, `pngparser.h`, `pngparser.c`) and of the test harness (`size.c`). The test harness is just a small program that executes the code that we want to test. The directory contains a makefile as well.

Program `size` takes an image name as input and outputs the dimensions of the image: `size test_image.png`.

Setting up AFL

1) Download AFL from https://github.com/google/AFL
2) Build and install AFL (`make`, `make install`)
3) Go to the `/src` directory
4) Code changes are sometimes needed when fuzzing.
    1) Find the function `is_png_chunk_valid` in pngparser.c
    2) Edit it so it always returns 1
5) Edit the makefile so the compiler used is `afl-gcc`
6) Run make
7) Create two directories in `/src`: `afl_in` and `afl_out`
8) Copy the two image files from `/seeds` to `/afl_in`
9) Run `afl-fuzz -i afl_in -o afl_out <path_to_size> @@`
10) Follow the instructions on screen to run AFL. You may need to change some system settings (disable crash reporting and change the CPU governor)
11) Please handle permission-related issues outside the container.
12) Let AFL run until you are satisfied with the number of unique crashes

The Task

1) You are required to find and fix at least 3 bugs in `pngparser.c`
2) For each bug provide a POC (generated by AFL) and a Markdown file in the required format
3) Submit your answers to the following questions in the file `ANSWERS.md`:
    1) Why did you need to change `is_png_chunk_valid`?
    2) Why did we give you exactly TWO seeds for fuzzing?
    3) Why did you have to use `afl-gcc` to compile the source (and not e.g. ordinary gcc)?
    4) How many crashes in total did AFL produce? How many unique crashes?
    5) Why are hangs counted as bugs in AFL? Which type of attack can they be used for?
    6) Which interface of `libpngparser` remains untested by AFL (take a look at `pngparser.h`)?
    7) How long did you run AFL for? If you run it for twice as long, do you expect to find twice as many bugs? Why?

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. If you find bugs in other source files, they will be accepted as well.

Remember that you need to recompile the binaries with normal GCC and a `-g` flag to debug them. Pass the AFL crashing input as an argument and debug the program in GDB.

Submission format

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

- `src/`
    - `crc.h`
    - `crc.c`
    - `pngparser.h`
    - `pngparser.c` - with your fixes and `is_png_chunk_valid` patched to return 1
    - `size.c`
    - `Makefile` - if you added more files, please provide your makefile. You should use gcc and keep the names of targets and generated binaries.
- `reports/`
    - `ANSWERS.md`
    - `00/`
        - `DESCRIPTION.md`
        - `<AFL-POC>`
    - `01/`
        - `DESCRIPTION.md`

    * <AFL-POC>
  - 02/
    * DESCRIPTION.md
    * <AFL-POC>

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 the 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.