

Task 1 - libfuzzer



Project - http-parser (8.8k lines)

Overview

This C-coded HTTP message parser program was used for our fuzz testing. Both requests and answers are parsed by the program. It is intended for use in high-performance HTTP applications. It doesn't perform any syscalls or allocations, buffers any data, and may be stopped at any moment. It only takes around 40 bytes of data per message stream, depending on your architecture (in a web server that is per connection). The data input that is sent for parsing was used in our llvm function as the data of libfuzzer test suite.

Setup for libfuzzer

We had to set up all our testing artifacts and dependencies on a virtual machine running on linux, as some of the libraries required for libfuzz library were not compatible for installation in our Windows or MacOS. Once the VM was ready and running we went through a small program as a tutorial on using this tool to get a better understanding on how to use the libfuzz tool.

Running the following commands on the linux terminal installed all the necessary dependencies and the clang binaries needed to run and understand the tool.

```
# Install git and get this tutorial
sudo apt-get --yes install git
git clone https://github.com/google/fuzzing.git fuzzing

# Get fuzzer-test-suite
git clone https://github.com/google/fuzzer-test-suite.git FTS

./fuzzing/tutorial/libFuzzer/install-deps.sh # Get deps
./fuzzing/tutorial/libFuzzer/install-clang.sh # Get fresh clang binaries
```

Finally we ran the following code to get the report on the small tutorial program.

```
clang++ -g -fsanitize=address,fuzzer fuzzing/tutorial/libFuzzer/fuzz_me.cc
```

```
./a.out
```

Once we understood that llvm function works like a main function with function calls to the fuzzed function, we then needed to run it on our bigger project - “http-parser”.

Choosing our test function

The following step in our exercise required us to locate a function to fuzz test, as libfuzzer focuses on individual functions to assist developers in focusing on the granularity of the programme itself.

```
2422 int
2423 http_parser_parse_url(const char *buf, size_t buflen, int is_connect,
2424                      struct http_parser_url *u)
2425 {
```

We examined the test suite to get insight into the execution of the http url parser function and discovered that it needed the initialization of a struct of http parser url, which may subsequently be used as one of the method's arguments. Furthermore, the function requires an input char* data and the size of the data. This data is processed by the function and will be our target of fuzz , for the data in the LLVMFuzzer function. Finally there is an int *is_connect* which is a *true* or *false* clause for the function, so in our test function we need to run the data with *true* (or 1) and the other test with *false* (or 0). So we created a file named “fuzz_url.c” which contains all the necessary dependent functions for running the http_parser_url and the main LLVM call function which tests the function using random byte input as the data input for the parser. The function call described above is shown below:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    struct http_parser_url u;
    http_parser_url_init(&u);
    http_parser_parse_url((char*)data, size, 0, &u);
    http_parser_parse_url((char*)data, size, 1, &u);

    return 0;
}
```

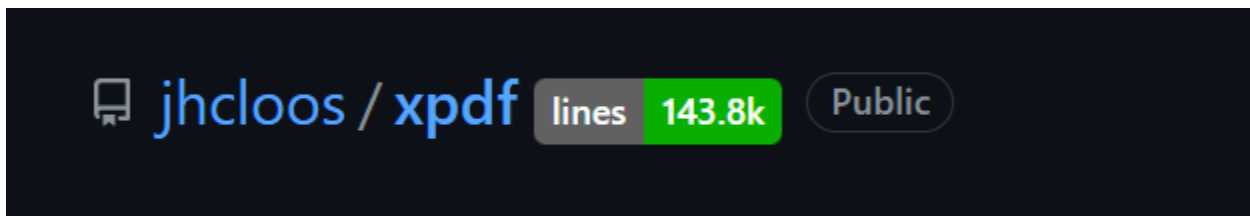
We then ran the following command to run the fuzzer:

```
clang++ -g -fsanitize=address,fuzzer fuzz_url.c
```

Doing so, the program took less than a second to build, producing the file “a.out” which was an executable for the test execution.

We then executed a.out by the command `./a.out` to see the test unfold. We ran the test for over 6 hours with more than 4 billion input executions done by libfuzzer, however no crash was reported for the first project we tested (will attach the whole log of 4 billion execution in the artifacts). We made the limitation of 6 hours and maybe doing it for longer periods might’ve shown us some crashes.

Task 2) AFLplusplus



Project - XPDF (143.8k lines)

Overview

This C-code XPDF program was used for our coverage guided fuzz testing. Xpdf is an open source viewer for Portable Document Format (PDF). In our program we will test a bunch of sample pdfs as a guidance for our input of fuzz testing.

Setup for AFLPlusplus and Xpdf

To set up AFLPlusPlus, we followed the guide in this [link](#). We had to install all the libraries along with some other external libraries like llvm-config-11 - which helped us in visualizing the test. After the installation of AFL++ which took about ~20 minutes and its dependencies, we cloned XPDF from [github](#).

The dependency installation commands are listed below:

```
export LLVM_CONFIG="llvm-config-11"
CC=$HOME/AFLplusplus/afl-clang-fast CXX=$HOME/AFLplusplus/afl-clang-fast++
./configure --prefix="$HOME/fuzzing_xpdf/install/"
make
make install
```

We needed to install XPDF in our system and then make sample pdf files which we had to feed it to the AFL++ fuzzer, which was modified in each of its runs to make the program crash.

To build XPDF, the commands were as follows:

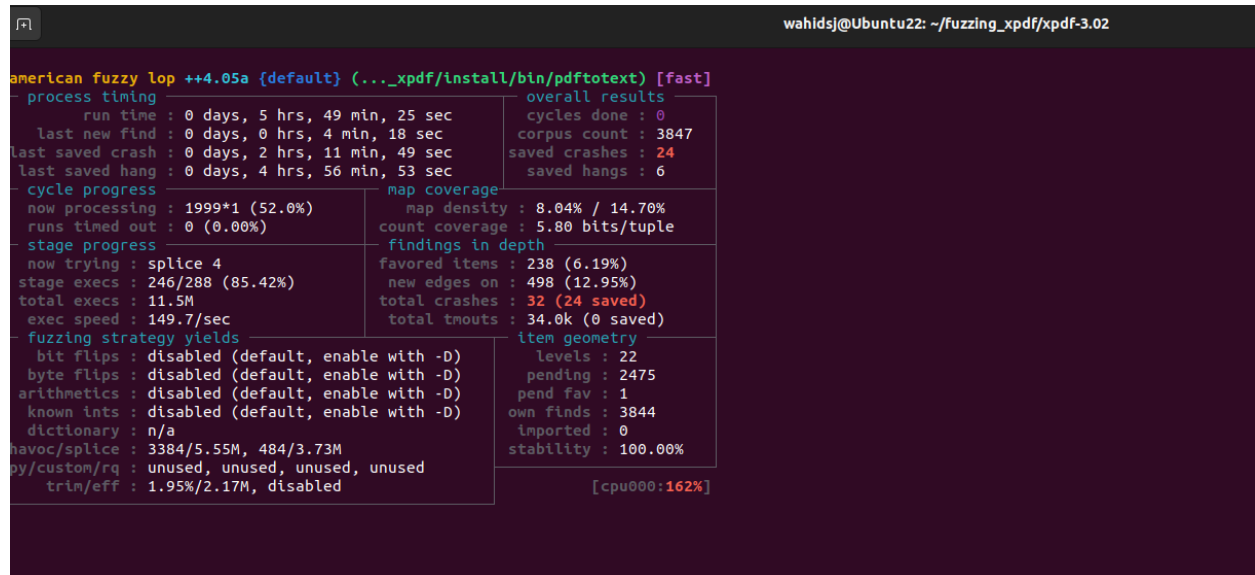
```
cd xpdf-3.02
sudo apt update && sudo apt install -y build-essential gcc
./configure --prefix="$HOME/fuzzing_xpdf/install/"
make
make install
```

The program took quite a long time to build, about ~3 minutes, as it required many external dependencies as well.

After successfully following the above command and installing XPDF, we were ready to run AFL++ fuzzer, with input files selected randomly in pdf format.

```
afl-fuzz -i $HOME/fuzzing_xpdf/pdf_examples/ -o $HOME/fuzzing_xpdf/out/ -s 123 --
$HOME/fuzzing_xpdf/install/bin/pdftotext @@ $HOME/fuzzing_xpdf/output
```

AFL running on Ubuntu



```
wahidsj@Ubuntu22: ~/fuzzing_xpdf/xpdf-3.02
american_fuzzy_lop ++4.05a {default} (..._xpdf/install/bin/pdftotext) [fast]
- process timing
  run time : 0 days, 5 hrs, 49 min, 25 sec
  last new find : 0 days, 0 hrs, 4 min, 18 sec
  last saved crash : 0 days, 2 hrs, 11 min, 49 sec
  last saved hang : 0 days, 4 hrs, 56 min, 53 sec
- cycle progress
  now processing : 1999*1 (52.0%)
  runs timed out : 0 (0.00%)
- stage progress
  now trying : splice 4
  stage execs : 246/288 (85.42%)
  total execs : 11.5M
  exec speed : 149.7/sec
- fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 3384/5.55M, 484/3.73M
  py/custom/rq : unused, unused, unused
  trim/eff : 1.95%/2.17M, disabled
- overall results
  cycles done : 0
  corpus count : 3847
  saved crashes : 32
  saved hangs : 6
- map coverage
  map density : 8.04% / 14.70%
  count coverage : 5.80 bits/tuple
- findings in depth
  favored items : 238 (6.19%)
  new edges on : 498 (12.95%)
  total crashes : 32 (24 saved)
  total tmouts : 34.0k (0 saved)
- item geometry
  levels : 22
  pending : 2475
  pend fav : 1
  own finds : 3844
  imported : 0
  stability : 100.00%
[cpu000:162%]
```

After successfully running AFL++ on XPDF for about ~6 hrs, we were able to find **32 crashes** from our initial input feeds, which suggests that there were vulnerabilities in the program. However, AFL++ was very heavy on the system as running this fuzzer for 6 hours made the testing CPU heat up a lot more than when ran libfuzzer. This suggests AFL++ needs more

memory power which makes sense as it continuously produces new mutations of samples from the given samples.

Reflection

While working on this assignment our group faced many challenges. Initially getting the libfuzzer installed in the system was pretty easy and straightforward, however, building and running it on a specific function was the bigger challenge. This is because we had to understand the whole code-base in order to understand how it works and what kind of input does it take. Since most functions are connected to other functions it gets difficult to test on a large scale project with type casting, where each method has different function definitions and structs for most of the inputs. However once the function was set, the test went smoothly as the rest of the program did not need to run but just the function itself. Libfuzzer didn't produce any error which made us wonder maybe it takes more time to produce a crash as it goes byte by byte. For the AFL++, we had a few problems when installing it as there were many dependencies and builds that needed to get the AFL running correctly. Additionally, finding the samples for XPDF to test it was another thing that we needed to do for our test. For the third task we had to read and understand how topological sort works in order to make property based tests for the same.

Effort wise, libfuzzer and Afl++ required a lot of effort and reading of documentation. As we did not perform both fuzz tests on the same project, we cannot compare the benchmark or compare performance. But thinking about ease of use, libfuzzer required a lot of manual work in order to test a function whereas AFL++ ran a coverage fuzz without coding a function or understanding each function. AFL++ also has a better dashboard representation than that of libFuzzer.

While both fuzz tests were able to capture crashes, we believe they are equivalently useful and efficient to find defects within a software.