CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing

WS 2023/2024

Prof. Dr. Andreas Zeller
Leon Bettscheider
José Antonio Zamudio Amaya

Exercise 7 (10 Points)

Due: 31. December 2021 23:59

> The lecture is based on The Fuzzing Book, an *interactive textbook that allows you to try out code right in your web browser*.
> The Fuzzing Book code is additionally available as a Python pip package. To work on the exercises, please install the package locally:
> `pip3 install fuzzingbook`

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you may not delete any provided ones. You can verify whether your submission is valid by executing `verify.py`:

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise 7-1: Fuzz it like industry (10 Points)

Preparation

AFL is a well-known and industry approved coverage-guided greybox fuzzer that works on C and C++ programs. Being one of the first fuzzing tools that popularized coverage-guided fuzzing, it has found bugs in a wide range of programs as can be read on the author's website.

In this exercise, you will fuzz the popular XML parsing library libxml2 using AFL. Finding a bug in the latest version of libxml2 is hard, as it has been fuzzed for many years (and it is still being fuzzed). Instead, you will concentrate on an older release.

To get started, you should use the `Dockerfile` provided in the exercise directory. If you are new to docker, please read the official introduction pages. This exercise's `Dockerfile` is based on Ubuntu 20.04, installs the required packages, downloads and builds AFL, and checks out the libxml2 version (v2.8.0) you will fuzz.

To build the docker image, change to the directory where the `Dockerfile` is located, and then run the command:

```
docker build -t sectest/ex71:1.0 .
```

This process can take a while to finish. Once it is done, run an interactive docker container based on this image using the command:

```
docker run -it sectest/ex71:1.0 bash
```

You can now interact with the bash shell as if it was a native Ubuntu system.

AFL provides a modified version of the `clang` compiler. It can be used just like the original `clang` compiler (or other C/C++ compilers such as gcc/g++), except that it injects instrumentation code that AFL requires to get coverage feedback during runtime. The AFL-infused compiler can be found at `/AFL/afl-clang-fast`.

Next, build libxml2 by performing the steps below:

```
cd /libxml2
CC=/AFL/afl-clang-fast ./autogen.sh
AFL_USE_ASAN=1 make -j4
```

At this point, you should have a compiled version of libxml2 stored at `/libxml2/libxml2/.libs/libxml2.a`. This build has both AFL instrumentations (for runtime coverage feedback), and ASAN (AddressSanitizer) instrumentations.

Why do we use sanitizers? Sometimes, a bug triggered by a fuzzer becomes immediately apparent. For instance, a fuzzer input might trigger a segmentation fault, which makes the operating system terminate the process (and thus the fuzzer knows that it has detected a bug). However, most runtime errors are more subtle and do not immediately require operating system intervention. For example, an out-of-bounds write might modify the value of a neighboring variable in memory without triggering a crash. Such errors, and more, can be detected by AddressSanitizer. For more information, have a look at the documentation.

Libraries typically expose many different functions which might or might not be suitable for fuzzing. To fuzz a library, usually the first step is to look for a function that could be interesting for fuzzing, for instance because it invokes complex code. Once such a function has been determined, a *fuzz harness* needs to be implemented. In essence, the harness takes an input from the fuzzer, and transforms it into a representation that the function under test expects. During this step, our job as programmers is to encode the knowledge we have about the function's expected input format.

a. A fuzz harness (2 Points)

To illustrate the concept, we provide you with a fuzz harness, which performs the following steps to test libxml2's parser:

- Process a XML file stored in memory using the xmlReadMemory function.
- Clean up the memory using the xmlFreeDoc function if parsing was successful.

For more background information on these functions, feel free to take a look at the parsing examples at http://xmlsoft.org/examples/.

The following fuzz harness is already stored in the docker container at `/libxml2/harness.c`.

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include "include/libxml/tree.h"
#include "include/libxml/parser.h"

#define SIZE 1000
```

```
void harness(char* buffer, ssize_t length) {
        const char dummy_xml_name[] = "noname.xml";
        xmlDocPtr doc = xmlReadMemory(buffer, length, dummy_xml_name, NULL, 0);
        if (doc != NULL) {
                xmlFreeDoc(doc);
        }
}

int main(int argc, char** argv) {
        char input[SIZE];
        while (__AFL_LOOP(1000)) {
                ssize_t length = read(STDIN_FILENO, input, SIZE);
                harness(input, length);
        }
        return 0;
}
```

Use the following command to build and link the fuzz harness, resulting in an executable at `/libxml2/fuzzer`. The command should be executed from directory `/libxml2`.

```
AFL_USE_ASAN=1 /AFL/afl-clang-fast ./harness.c -I include .libs/libxml2.a -lz -
lm -o fuzzer
```

Run the following commands and copy the output of all commands to file `exercise_1a.txt`.

```
nm /libxml2/fuzzer | grep -E "asan_init|harness|__afl"
ldd /libxml2/fuzzer
file /libxml2/fuzzer
```

Info:

- To be most effective, fuzz harnesses should not contain any nonessential code, such as `printf`'s, which will slow down the fuzzing process.

b. Fuzz it like it's 2012 (3 Points)

Now, run the following commands from directory `/libxml2` to set up the fuzzer run.

First, create the two directories `inputs` and `outputs`.

```
mkdir inputs outputs
```

Store an initial XML testcase to the `inputs` directory.

```
echo "<fuzz></fuzz>" > inputs/seed.xml
```

Finally, execute your fuzzer: (Note that this is one line)

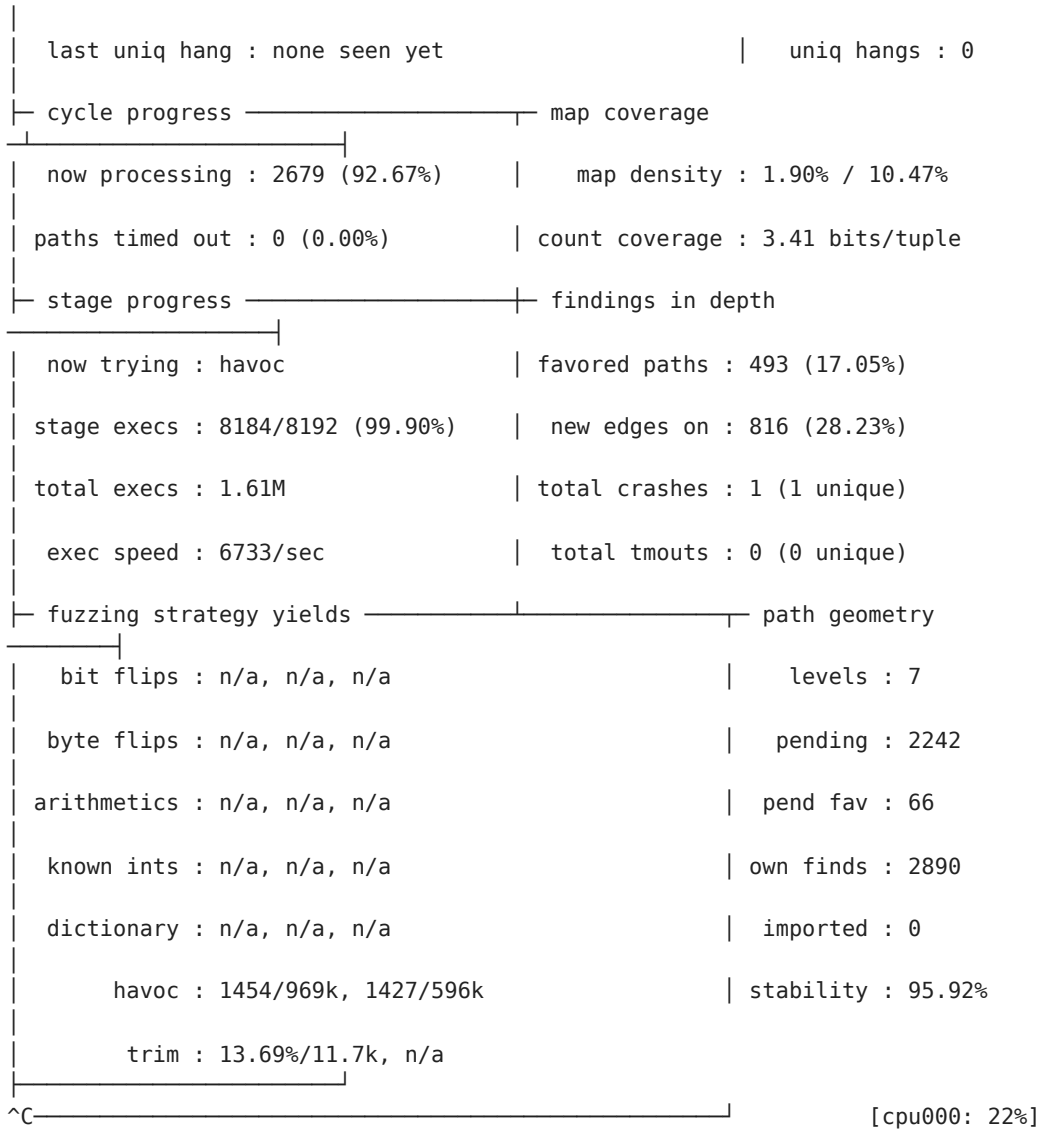
```
/AFL/afl-fuzz -m none -x /AFL/dictionaries/xml.dict -i inputs -o outputs/ -S
fuzzer01 -- /libxml2/fuzzer
```

Let it run for 30 minutes. Does it find a bug? If so, place AddressSanitizer's report in `exercise_1b_asan_report.txt` and the failing test case in `exercise_1b_asan_testcase.txt`. If you found more than one bug, *only submit one*.

Additionally, copy AFL's status screen after running for roughly 30 minutes to `exercise_1b_afl.txt`, e.g.:

```
                        american fuzzy lop 2.57b (fuzzer01)

    ┌─ process timing ─────────────────────────────────────┬─ overall results
    ─────┐
    │        run time : 0 days, 0 hrs, 4 min, 2 sec        │  cycles done : 0
    │
    │   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 2891
    │
    │ last uniq crash : 0 days, 0 hrs, 0 min, 31 sec       │ uniq crashes : 1
```



Note:

- When AFL triggers a crash or ASAN violation, it will put the failing input in a `crashes` directory located in a subdirectory of the `outputs` directory. To produce the ASAN report, you can pass this failing input to the ASAN-instrumented fuzz target, e.g. `/libxml2/fuzzer < id\:000000\,sig\:06\,src\:002318\,op\:havoc\,rep\:32`.
- Make sure that your harness does not contain a bug.

c. Implement a fuzz harness (2 Points)

In this exercise you will write another fuzz harness for libxml2.

The harness should:

- Call `xmlRegexpCompile` on a fuzzer-generated string. For more information, read the function documentation.

You can use the following template, which you can also find in the docker container at `/libxml/harness2.c`:

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include "include/libxml/tree.h"
#include "include/libxml/parser.h"
#include "include/libxml/xmlregexp.h"
```

```c
#define SIZE 1000

void harness(char* buffer, ssize_t length) {
        // Implement your harness here
}

int main(int argc, char** argv) {
        char input[SIZE];
        while (__AFL_LOOP(1000)) {
                ssize_t length = read(STDIN_FILENO, input, SIZE);
                harness(input, length);
        }
        return 0;
}
```

Submit the harness you have implemented in this exercise as `harness2.c`.

Hints:

- A C string can be converted to a `xmlChar` by typecasting. For example: `const xmlChar* a = (const xmlChar *)"abc";`
- Make sure that your harness does not contain a bug.

d. Run your fuzz harness (3 Points)

Now, build the harness you have implemented in Exercise 7-1c. To do this, you can refer to the instructions of Exercise 7-1b again, but make sure to specify the correct fuzzer harness `/libxml/harness2.c`.

Let the fuzzer run for about 30 minutes. Does it find a bug? If so, place AddressSanitizer's report in `exercise_1d_asan_report.txt` and the failing test case in `exercise_1d_asan_testcase.txt`. If you found more than one bug, *only submit one*.

Additionally, copy AFL's status screen after running for roughly 30 minutes to `exercise_1d_afl.txt`.

Notes:

- Give this fuzzer binary a different name - use for instance the name `fuzzer2` rather than `fuzzer`.
- Before running the fuzzer, make sure to empty the `inputs` and `outputs` directories which might be polluted from Exercise 7-1b.
- You should **not** use the `-x /AFL/dictionaries/xml.dict` parameter as in Exercise 7-1b.