

Software Security Lab on Fuzzing techniques

Download [this tar file](#) from the Moodle web page
Read the [instructions](#) to get the fuzzing tools ...

Part 1 – Grey box fuzzing with AFL

Exercise 1: handling AFL

Go to the directory `AFL-example` and look at the file `example1.c`.
This code takes an input file (as a command line argument) containing an integer and use this integer value as an upper bound to access a fix-sized buffer.

Compile it (with AFL instrumentation) and run it using the following command:
`run-afl.sh example1.c`

Look at the directory `out/crashes` containing the input file(s) leading to crashe(s).

Exercise 2: push it to the limits ...

Write a few examples of (small) vulnerable C code of your own and check if AFL is able (or not) to spot the problems. Try to understand and to provide some explanation when it is not the case.

First you can start with the provided file `example2.c`.

Can you find "by hand" (simply looking at the code) an input which may lead to a crash?
Does this crash occur when running this program (without using AFL)? And when compiled with Adsan (`-fsanitize=address`)?
Run it with AFL to see if the problem is detected or not (with and without Adsan) ...

You can then consider small code examples containing a vulnerability which is not a "direct" stack buffer overflow. It could be for instance:

- a heap-based buffer overflow
- a use-after-free
- a "non-obvious" memory error (e.g., resulting from some arithmetic overflow or unexpected type cast)

Exercise 3: run it on larger example

The code given in directory `JsonParser` implements a Json parser, allowing to parse a Json file and pretty-prints its content on the screen. [Json](#) is a human-readable file format for data interchange. Json syntax and examples of Json files can be found [here](#).

The code provided contains several security vulnerabilities, the objective here being to find and understand them using AFL.

Q0. Compile the code provided (using the command make). You should get 2 executable files:

- `jsonparser` (instrumented for AFL)
- `jsonparser_ASAN` (instrumented for AFL and AddressSanitizer) .

Run (without AFL) the `jsonparser` executable giving as argument the few examples provided in directory `regular_input`, e.g.: `./jsonparser regular-input/ex1`

Q1. Have a look at the code to briefly understand its structure and try to spot by hand potential vulnerability issues (locate where and how memory is allocated and accessed, possible buffer overflows, arithmetic overflows, etc.).

Q2. Try to (randomly) find crashes running some hand-made examples (you may actually find easily some crashes !). Keep your "winning" inputs (the ones leading to a crash)

Q3. Try now a more effective crash detection technique using AFL using either:

- `./run-json-afl_ASAN.sh` (for the ASAN version)
- `./run-json-afl.sh` (for the non ASAN one)

Q4 (crash analysis). For each (unique) crash found:

- without using AFL run the program (instrumented with AdSan) with the corresponding crash input
- identify the bug
- is this bug a security vulnerability ? Is it likely to be exploitable ?

Rk: You can also use a debugger (like `gdb`) to get more information about the crashes.

Part 2 – Symbolic execution with PathCrawler

For this part you should use the online version of PathCrawler:

<http://pathcrawler-online.com:8080/>

Remarks:

- do not forget to upload again the tar file to run it with PatchCrawler after each change on your machine
- you can put several functions in a same file and fuzz them independently

In the following we will work with examples provided on directory `PathCrawler-Examples`

Exercise 1

Q1. Upload the tar file `Example1.tar` and fuzz function `f` located in file `example1.c`

Take the time you need to understand the various outputs produced under the “Test Case” menu, namely:

- the **test cases**, and their corresponding execution paths in the source code;
- the **path predicates** associated to each of these test cases
- the **test input**, i.e. the concrete values found by the solver satisfying each path predicate.

Q2. Modify this example in order to add it a *non feasible* execution path, that will be not covered/explored by any test case. Check the result by running this example with PathCrawler.

Exercise 2

The file `example2.c` contains several functions containing code variants accessing a buffer `T` depending on the value on their input parameter `i`. Depending on the value of `i` these functions may (or may not) contain a buffer overflow.

Q1. Execute successively all these functions with PathCrawler and try to understand the results provided (with respect to the bug detection).

Exercise 3

Try examples from your own corresponding to the following scenarios:

1. a function containing a vulnerability induced by an arithmetic overflow
2. a function containing a vulnerability not found by PathCrawler because of a non decidable constraints ...
3. a function containing a vulnerability inside a loop body
4. a function containing a vulnerability occurring after a "for" loop with a fixed number of iteration (i.e., not depending on the function parameters)
5. a function containing a vulnerability occurring after a "for" loop with a variable number of iteration (i.e., depending on a function parameter).

Part 3 – Application to a concrete example

The link below describes two vulnerabilities (leading to a CVE) found in a part of the code of the Grub2 Linux Bootloader:

<https://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html>

The objective is to see how much a fuzzing tool like AFL and/or PathCrawler may help to find such vulnerabilities.

To do so:

1. Read the explanations provided on the link above ...
2. Get the code of the `grub_username_get` function provided in the previous link
3. Make it executable:
 - replace "grub" library calls by standart ones (`grub_memset` by `memset`, `grub_isprint` by `is_print`, `grub_printf` by `printf`, etc.)
 - provide a main function allowing to read a string from an input file, store it on a fixed-size buffer and call `grub_username_get`

When your code can be compiled and executed properly on simple examples, try to fuzz it using AFL or PathCrawler in order to find the vulnerabilities mentioned (or other ones !)