Sistemas de Software Seguros
Segurança de Software
2024/2025
Class Project: Experiments with Fuzzers

1. Objective

The aim of this class project is to experiment with the AFL fuzzer to search for vulnerabilities. In particular, we will be looking for a vulnerability in the GNU coreutils package, which contains several programs for basic file, shell and text manipulation. There is a bug in the *date* and *touch* commands that when exploited can cause a crash (see below).

Vulnerability Details : CVE-2014-9471

The parse_datetime function in GNU coreutils allows remote attackers to cause a denial of service (crash) or possibly execute arbitrary code via a crafted date string, as demonstrated by the "--date=TZ="123"345" @1" string to the touch or date command.

Publish Date : 2015-01-16 Last Update Date : 2017-06-30

2. Fuzzing GNU Coreutils

a) Checking that the vulnerability exists

The application to be fuzzed is the linux package `coreutils-8.21`, which is located at `/home/ss/apps/vulnApp`. You should run the following steps to setup and compile the coreutils:

```
cd apps/vulnApp/coreutils-8.21/
./configure
make
cd src
```

Now, lets check that the `./date` command is working by running the following commands:

```
./date
(this should give the current time)

TZ='America/Los_Angeles' ./date
(this should give the current time in Los Angeles)

./date --date='TZ="America/Los_Angeles" 09:00 next Fri'
(this is another form to give the time in Los Angeles, but for 9AM
next Friday)
```

To terminate, **find the command that exploits** the vulnerability and that causes `./date` to crash.

b) Fuzz the date program

Now let's find out if the AFL fuzzer is able to locate the vulnerability. Before you use AFL, you need to make sure it is compiled. See Annex A to learn how to compile.

Setup the operating system to support AFL:

```
sudo su -
echo core > /proc/sys/kernel/core_pattern
exit
```

Compile coreutils with support for AFL:

```
cd ~/apps/vulnApp/coreutils-8.21/
CC=~/apps/AFL/afl-gcc ./configure
make clean all
```

Create the necessary directories for the operation of AFL:

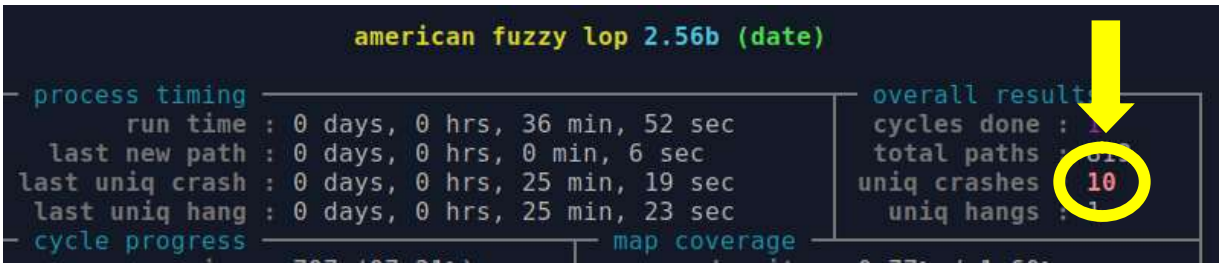
```
mkdir inDir
mkdir outDir
mkdir plotDir
```

Create in directory `inDir` a file called `test` with content " `TZ="1"` ", and then fuzz the target program:

```
~/apps/AFL/afl-fuzz -i ./inDir -o ./outDir -- ./src/date -f @@
```

At this point, AFL should start fuzzing the application. You can find in directory `outDir/crashes` the test files that when provided as input caused the `date` command to

crash, and therefore, that exploit a bug in the application. You can also see how many of these files have been created in the output window of AFL.



To answer the next questions, it might be useful to read the extra information that is provided as annex of this class project.

c) Confirm crash

Select one of the crashing test files and **confirm that it causes the crash** of the `date` command. To do that, you need to pass as input to the program the contents of the file.

d) Locate the vulnerability

Try **to identify the place in the `date` command source code** where the bug is occurs. To do that, you can use for example the `gdb` debugger or some other tool that you prefer.

e) Determine if the "unique" crashes are really "unique"

Use another of the test files to see if it crashes the program. Although it is called "unique crashes", does it find a different vulnerability?

f) Use AFL_HARDEN to check for bugs

Use the `AFL_HARDEN` vulnerability detection mechanism to determine is further vulnerabilities are found in the code**. You might want to have a look at Annex B.** Note that AFL with `AFL_HARDEN` may only find other inputs that activate the same vulnerability.

Delivery of the Report

The output of the class project is a report answering all the questions and including the justifications for the responses. Each group should deliver the report either by submitting it in

the course moodle page, or if there is some difficulty with this method, by emailing it to TP class professors. The file type should be a pdf.

Deadline: 9 December 2024 (there will be no extensions)

Optional questions --- the groups are NOT required to answer them, but they are interesting follow up work!

Further questions:

- Look into de source code of the date command and **find out what is the cause for the vulnerability** (e.g., is it a buffer overflow?)
- **Explain how the vulnerability could be corrected** in the source code
- Determine if the bug **could also be exploited to execute arbitrary code**

You can explore further capabilities from the fuzzer by reading the documents in the `docs` directory and the file `README.md` in the home directory of AFL. For instance, if you want to see a progress report, do the following:

```
sudo apt-get install gnuplot
~/apps/AFL/afl-plot ./outDir ./plotDir
```

In your browser look at:

```
file:///home/ss/apps/vulnApp/coreutils-8.21/plotDir/index.html
```

You should also notice that the name of the file with the crashing test case provides information about the parent test case. For example, if the file has name:

```
id:000001,sig:06,src:000000,op:arith8,pos:6,val:+24
```

this indicates that the parent test case file was "`src:000000`" and the cause for the crash was signal 6 ("`sig:06`"), which corresponds usually to SIGIOT = SIGABRT (this signal indicates an error detected by the program itself and reported by calling `abort`). To create this test case, the parent suffered mutation "`op:arith8`" on byte "`pos:6`", where this byte became "`val:+24`".

Based on the timestamps of the creation of the crashing test files it is possible to determine when, during the fuzzing execution, the crashing input was found, and therefore, create a graph of its distribution through time (the start time of the fuzzing process can be obtained in file `./outDir/fuzzer_stats`). Moreover, since the mutation operation appears in the file, it is possible to understand which mutators are more effective.

ANNEX

A. Setup the AFL Fuzzer

To setup the AFL fuzzer, you can get the latest version from https://github.com/google/AFL (an older version can be obtained from http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz). To download the latest version, press button “code” and then select "Download ZIP:

Then you need to carry out the following steps to setup AFL:

```
mv Downloads/AFL-master.zip apps/
cd apps/
unzip AFL-master.zip
mv AFL-master AFL
cd AFL/
make
```

To fuzz a program, you need to compile it with the frontend compiler created by AFL (for ex. `afl-gcc`) so that instrumentation is inserted in the executable. Then, to fuzz the program you need to run `afl-fuzz`. The main arguments for `afl-fuzz` are:

```
afl-fuzz [ options ] -- /path/to/fuzzed_app [ ... ]

where the most important options are:
```

```
-i dir        - input directory with input test cases

-o dir        - output directory for fuzzer findings
```

After the "--" characters, the next parameter is the name of the application that we intend to fuzz. This application can have also command line arguments. For programs that take input from a file, use '@@' to mark the location in the target's command line where the input file name should be placed. AFL will substitute this for you:

```
afl-fuzz -i in_dir -o out_dir -- /path/to/program @@
```

Sometimes, you might want to interrupt a fuzzing session and restart it later. You can achieve this by running:

```
afl-fuzz -i- -o out_dir -- /path/to/program @@
```

Further directions to learn more are available in file `docs/life_pro_tips.txt` .

B. Increasing the Chances of Finding Vulnerabilities

As we have seen in previous classes, sometimes there are memory corruption inputs to a program that do *not* cause a crash. In these cases, AFL is unable to detect that a vulnerability was actually activated. To increase the probability that vulnerability activation is detected, AFL allows for a few alternative solutions.

1) Before compiling the target program, ensure that the environment variable AFL_HARDEN is set to 1. This will cause AFL to insert further checks when compiling your program, enabling the detection of several extra bugs.

```
export AFL_HARDEN=1
printenv | grep AFL
CC=~/apps/AFL/afl-gcc ./configure
make clean all
~/apps/AFL/afl-fuzz -i ./inDir -o ./outDir -- ./src/date -f @@
```

2) A complementary mechanism to AFL_HARDEN is the library `libdislocator` that comes with the AFL code. This library replaces the original memory allocation functions from libc (e.g., `malloc()`) with functions that are able to detect overflows/underflows and use-after-free/realloc bugs in the heap. To be able to use this library, the target program **cannot be linked** with static libraries. Also notice that this library consumes much more memory than the original functions from libc, and therefore, it **should not be** used in production environments. Run the following steps to use this library.

```
cd ~/apps/AFL/libdislocator/
make
```

```
    cd ~/apps/vulnApp/coreutils-8.21/
    AFL_PRELOAD=/home/ss/apps/AFL/libdislocator/libdislocator.so
~/apps/AFL/afl-fuzz -i ./inDir -o ./outDir -- ./src/date -f @@
```

(Notice: (1) the full path to the libdislocator.so needs to be provided; (2) AFL_PRELOAD and afl-fuzz are in the same command line)

3) Another solution is to take advantage of mechanisms that look for bugs while a program executes, such as the AddressSanitizer (ASAN) and the MemorySanitizer (MSAN). These tools, however, allocate a significant amount of memory in 64-bit architectures, which creates problems to AFL. Therefore, to avoid this problem, it is necessary to compile the target program for a 32-bit architecture (option `-m32` to gcc) and to tell AFL that it should allow for further memory to the child processes (option `-m megs`).

```
    sudo apt-get install gcc-multilib

    unset AFL_HARDEN
    export AFL_USE_ASAN=1
    CC=~/apps/AFL/afl-gcc ./configure CFLAGS='-m32'
    make clean all

 ~/apps/AFL/afl-fuzz -m 800 -i ./inDir -o ./outDir -- ./src/date -f @@
```

You can then see what ASAN/MSAN provides as information by running

```
  ./src/date -f outDir/crashes/id\:00000…rest of the name of the file
```

(Notice: it necessary to install further libc libraries so that the compiler has support for 32-bit architetures; if program was written in c++, you would need to get the libraries for c++ with "sudo apt-get install g++-multilib")

C. Crash Triage Process

There are several steps that must be performed from the discovery of a vulnerability until its final exploitation. AFL provides some help to implement a few of them (see below in bold):

- Seed files -> **Fuzzer** -> **Crashing inputs**
- **Crashing inputs** -> **Minimization** -> **Bucketing** -> Per bug crashes
- Per bug crashes -> automated analysis -> automated triage report
- Automated triage report + input file -> Human using disassemblers and debuggers -> Proof of Concept
- Proof of Concept -> exploit development -> exploit

As we saw in this class project, AFL is quite effective at producing test cases that explore different paths of the program and that cause crashes (*Crashing inputs*). However, after completing the fuzzing operation, we are left with: 1) many test cases that explore different

paths of the target program; some of earlier tests, however, might only cover subsets of the paths that are covered by older test cases and consequently could be eliminated because they are redundant; 2) there are potentially many test cases that cause a crash; nevertheless, these test cases might in fact explore the same bug, and therefore, they could be redundant (notice that if the bug is reached from different program paths, it might be useful to keep multiple inputs because a few paths might be easier to exploit).

Check the crash input: you can manually check that the discovered test cases that cause a crash in fact exploit a bug. To achieve this, simply run the target program with one of the test cases stored in the directory: `./outDir/crash`.

```
./src/date -f outDir/crashes/id\:000000…rest of the name of the file
```

The following command helps to run all test cases in the queue. This helpful if for example the program was fuzzed without using any of the mechanisms that enhance vulnerability detection, and now you want to check all the test cases with one of these mechanisms (e.g., ASAN):

```
ls outDir/queue | grep id | xargs -I{} ./src/date -f outDir/queue/{}
```

Minimization: the goal here is to make the following steps of the analysis easier by reducing the size (in number of bytes) of the test cases, while ensuring that all paths that were found continue to be covered. For example, we would like that each of the crashing input files only contains the bytes that are required to cause the crash. AFL comes with the tool `afl-tmin` (***afl test-case minimizer***) that helps to perform this task. The parameters to this tool are like the ones for `afl-fuzz`.

The following command helps to eliminate redundancies in the queue of the tests cases:

```
mkdir outDir/min_queue
ls outDir/queue | grep id | xargs -I{} ~/apps/AFL/afl-tmin -m 800 -i
./outDir/queue/{} -o ./outDir/min_queue/{} -- ./src/date -f @@
```

The following command helps to remove redundancies in the crash tests cases:

```
mkdir outDir/min_crash
ls outDir/crashes | grep id | xargs -I{} ~/apps/AFL/afl-tmin -m 800 -i
./outDir/crashes/{} -o ./outDir/min_crash/{} -- ./src/date -f @@
```

Bucketing: Once input files have been minimized, it is helpful to get rid of the test cases that explore similar paths (or subpaths) as other test cases. Similarly, it is useful to remove any duplicate input files that trigger a crash in the same bug. AFL comes with the tool `afl-cmin` (***afl corpus minimizer***) that helps to perform this task. The parameters to this tool are like the ones for `afl-fuzz`.

The following command minimizes the number of test cases in the queue while achieving the same coverage:

(Note: when doing this minimization, make sure the target program is not compiled with the ASAN (or similar) mechanism because `afl-cmin` might be able to reduce further the number of test cases --- before recompiling do not forget to unset the AFL_* environment variables).

```
mkdir outDir/min_c_queue
```

```
~/apps/AFL/afl-cmin -i ./outDir/min_queue -o ./outDir/min_c_queue -- ./src/date -f @@
```

The following command would perform something similar for the crash test cases. However, you may get an error "*Error: no traces obtained from test cases, check syntax*!" if the tool is not capable of collecting some of the necessary information to perform the analysis:

```
mkdir outDir/min_c_crash
~/apps/AFL/afl-cmin -i ./outDir/min_crash -o ./outDir/min_c_crash -- ./src/date -f @@
```

Finding alternative crash inputs: sometimes it is helpful when producing an exploit to understand what alternative code paths can be used to reach the crashing bug. AFL has a special exploration mode that takes one or more crashing test cases as input, and uses its fuzzing strategies to quickly enumerate other code paths that can be reached in the program while keeping it in the crashing state. Sometimes, however, the resulting test cases end up being similar to the existing ones in terms of the executed paths, and so the minimization tools above can be helpful to discard duplicate test cases.

```
mkdir alternative_crashDir
~/apps/AFL/afl-fuzz -C -i ./outDir/crashes -o ./alternative_crashDir -- ./src/date -f @@
```

(Note: in directory `alternative_crashDir/crashes` there will be the alternative test cases that still cause the program to crash, and that potentially explore alternative paths)

Searching for the most relevant input bytes: AFL has a tool that attempts to identify the bytes in the input test case that appear to be critical. The tool `afl-analyze` sequentially flips bytes in the input test case and observes the behavior of the target program. Then, it color-codes the input test case based on which sections appear to be critical.

```
~/apps/AFL/afl-analyze -i ./outDir/crashes/id_..._rest_of_file_name -- ./src/date -f @@
```

Get coverage information: AFL has a tool called `afl-showmap` that provides information about the coverage of the bitmap when the target program is executed with a given input test case. This information is useful to get a rough idea about the increasing coverage that is attained as AFL produces new test cases that go deeper and deeper into the target program (the *number of tuples* corresponds approximately to the number of edges that are covered in the program control flow graph):

```
mkdir outDir/traces_showmap
~/apps/AFL/afl-showmap -o outDir/traces_showmap/trace_1_file -- ./src/date -f ./outDir/queue/id_..._rest_of_file_name
```