Homework 1: Fuzz testing & Permissions

This homework is due **September 10, 2019** at **10 PM**. You will have a budget of five late days (24-hour periods) over the course of the semester that you can use to turn assignments in late without penalty and without needing to ask for an extension. You may use a maximum of two late days per assignment. Once your late days are used up, extensions will only be granted in extraordinary circumstances.

We encourage you to discuss the problems and your general approach with other students in the class. However, the answers you turn in must be your own original work, and you must adhere to the Code of Academic Integrity. Solutions should be submitted electronically **via Canvas** with the template at the end of this document.

Manually reviewing source code for vulnerabilities can be laborious and time consuming, and outsiders typically cannot do it at all for closed-source software. For these reasons, both attackers and defenders often use an automated form of vulnerability discovery called "fuzz testing" or "fuzzing" that attempts to find edge-cases that the application developers failed to account for. In fuzzing, the analyst creates a program (a "fuzzer") that emulates a user and rapidly provides many different automatically generated inputs to the target application while monitoring for anomalous behavior (e.g., crashes or corrupted return data). When an input consistently causes anomalous behavior, the fuzzer stores it so that the analyst can investigate the problem. The anomalous behavior may be a sign that there is an exploitable vulnerability in the code path that the input exercises.
It is not usually feasible to test with every possible input, but a clever input generation algorithm can increase the odds that the fuzzer will trigger a bug. For instance, many fuzzers start with a set of valid inputs and then corrupt them by making randomized changes, additions, or deletions. Other fuzzers instrument part of the source code to understand which code paths are being exercised and which are not. This helps the fuzzer come up with better test cases and provide useful statistics to the analyst.
In this homework, you will be using a security-oriented fuzzer called *American Fuzzy Lop* (AFL) to help you discover potential vulnerabilities in real code bases. Throughout the homework you will find questions related to access control in Linux and Fuzzing. Answer those questions and turn them in as part of the homework following the instructions at the end.

Setup.

For this assignment you will need a Linux machine. For convenience you can use the VM image that we have provided you for Project 1. You can find the instructions on how to set it up and

run it in the Project 1 document (https://cis.upenn.edu/~sga001/classes/cis331f19/project1/proj1.pdf). The only change you need to make is to increase its memory, since the VM was originally configured with 512 MB of working memory, and AFL will need a little bit more. 2 GB should suffice. If your machine does not have 2 GB of memory, you can use the Moore 100 lab machines which have VirtualBox installed. To change, follow these steps:

1. Open virtualbox.
2. Click on the CIS331 VM, then go to Settings, and then System
3. Increase Base Memory to 2 GB.

Once you have your Linux setup working, download and install AFL and git into your VM:

```
sudo apt-get install git
wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
tar -xzvf afl-latest.tgz
cd afl-2.52b
make
sudo make install
```

Next, you will download the two code bases that will serve as our case studies, and setup AFL to fuzz each of them.

Note: you will be fuzzing code written by people who are not affiliated with this class. We chose these projects essentially at random; we only made sure that their licenses allowed their use in this homework, and that they were easy to to Fuzz with AFL. There is no guarantee that you will find vulnerabilities. If you do, you are free to report any potential bugs that you find to the authors of these tools, but please wait until after the deadline has passed.

Part 1: Fuzzing a JSON parser

Get the library "json-parser" from github.

```
git clone https://github.com/udp/json-parser
cd json-parser
```

Confirm that you are working with the latest version (commit e6426ae…):

```
git log | head -1
```

Compiling with AFL. Compile json-parser with AFL, using the *address sanitizer* (ASAN) option. ASAN is available starting with LLVM 3.1 and gcc 4.8 and helps detect memory errors by replacing `mallocs` and `frees`, and changing the behavior of memory accesses to perform additional checks. For more details on how ASAN works, you can check out: https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

```
AFL_USE_ASAN=1 CC=afl-gcc CXX=afl-g++ ./configure
AFL_USE_ASAN=1 make
```

This creates the static library `libjsonparser.a`.

Creating a harness. Unfortunately, we cannot directly fuzz this library. We need a *harness*, which is code that enables us to automate the testing of this library. In particular, our harness needs to take inputs from stdin, call the appropriate library functions, and then exit cleanly.
The good news is that the authors of json-parser have also included an example application that uses this library and takes a filename as input. This code is the perfect harness! Compile it using AFL and link the `libjsonparser.a` static library and the `libm` math library (which json-parser uses for operations like pow).

```
AFL_USE_ASAN=1 afl-gcc examples/test_json.c -I. libjsonparser.a -lm -o test_json
```

At this point you now have an instrumented `test_json` binary.

Setting up inputs and outputs. The next step is to figure out where to put the inputs and outputs of the fuzzer. Start by creating a *ramdisk*. A ramdisk is a virtual disk that uses your memory instead of the underlying SSD or HDD as backing. The reason for using a ramdisk here is that AFL is going to write a ton of stuff over and over, and SSDs might experience wear. Memory accesses are also faster than accessing an SSD, so this speeds the fuzzing as well.

```
sudo mkdir /mnt/ramdisk
sudo chown ubuntu:ubuntu /mnt/ramdisk
sudo mount -t tmpfs -o size=256M tmpfs /mnt/ramdisk
```

Question (a): What does `chown` do, and what is one reason to use it here?

You can verify that the ramdisk is setup up properly with: `df -h | tail -n1`
Since you are fuzzing a JSON library, you can leverage the JSON dictionary that AFL already has. This is an optimization to make fuzzing more effective and is not required. You should also *seed* AFL, by providing it some sample inputs. AFL will use these seeds to construct test cases. The json-parser authors have several tests in the `tests` folder, so you can just use those tests as seeds. Create input and output folders in the ramdisk and copy the tests to the input folder.

```
mkdir /mnt/ramdisk/inputs /mnt/ramdisk/outputs
cp tests/*.json /mnt/ramdisk/inputs/
```

To get an idea of how all of these pieces come into play in AFL, see the figure below (Figure 1).

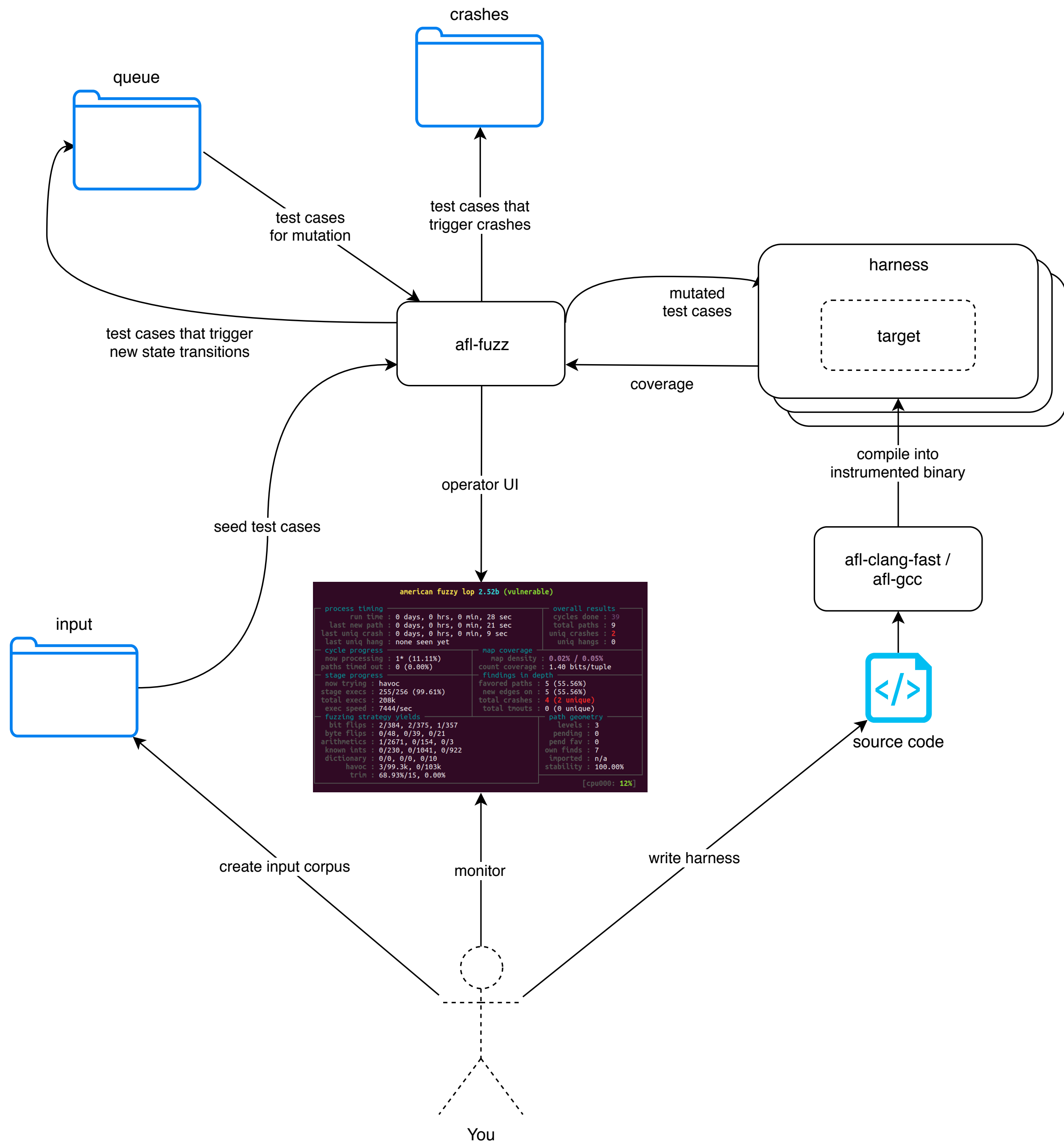

Figure 1: Outline of AFL. Taken from: https://github.com/mykter/afl-training/tree/master/harness. You may also check that URL to learn more about the basics of creating a harness.

You have finally set everything up, and are ready to start fuzzing. Run AFL with the following command (we discuss the flags below):

```
afl-fuzz -m 800 -x /path/to/afl-2.52b/dictionaries/json.dict\
-i /mnt/ramdisk/inputs/ -o /mnt/ramdisk/outputs/ -- ./test_json @@
```

-m: sets the memory limit (in MB)
-x: sets the dictionary (optional)
-i: input folder containing seeds
-o: output folder
@@: this tells AFL that the harness takes file names as inputs instead of the data directly.

Note that the first time you run AFL you will get a message asking you to not output core notifications. It also tells you exactly how to fix it:

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

Question (b): Why does the one liner `sudo echo core > /proc/sys/kernel/core_pattern` not work (try it out)?

Run AFL again. This time the core notifications message should be gone. You should let the fuzzer run for about 5 to 10 mins or until it finds a couple of inputs that crash `test_json`. You can then exit with ctrl-c (but read the question below before you exit). The inputs that crash the program are stored at: `/mnt/ramdisk/outputs/crashes/` Save one of those inputs as `crash-input1.json`, and submit it with the rest of your answers. To test that the produced file really does crash the program (or triggers an address sanitizer error), just test it!

```
./test_json /mnt/ramdisk/outputs/crashes/<input filename>
```

Question (c): For how long did you run AFL, how many unique crashes did it trigger, and how many total paths did it explore before you pressed ctrl-c?

Part 2: Fuzzing a plotting tool

`guff` is a command line tool that plots a set of points to the terminal.
Download `guff` from github, and make sure you are using the latest version (commit a6f11ad…).

```
git clone https://github.com/silentbicycle/guff
cd guff
git log | head -1
```

Compile `guff` using the Makefile and test it (read the provided README for instructions).

```
make guff
```

Note: if you just type `make` (without specifying `guff`) you will likely get an undefined reference error. This is because the Makefile is not linking the math library (`-lm`) in the right place for the test cases target (gcc is very sensitive to the order of flags!). Let this be a lesson. Making every compiler happy is a pain! While this Makefile likely works on the author's machine and the version of gcc that the author has installed, it sadly does not work on ours. If you want to build the test cases, simply move the `TEST_LDFLAGS` to the end of the `test_${PROJECT}` target in the Makefile. That should make gcc happy.
Now recompile `guff` but this time using AFL (set the CC and CXX environment variables, as you did before, followed by make). Create seeds that AFL can use for `guff` and place them in an input directory in the ramdisk (we recommend that you create new input and output directories to avoid mistakenly passing inputs that were meant for the previous program). Fuzz `guff` for 10–15 mins (or until it finds a few inputs that crash the program) and then answer the following questions.

Question (d): What are good seeds to use for `guff`? Provide one sample seed in a file called `guff-seed.txt`.

Question (e): What is the name of one of the files generated by AFL (check the crashes folder in `/mnt/ramdisk/outputs/`). In addition to specifying the name, submit that file as `crash-input2.txt`.

Question (f): What are 2 reasons that would lead AFL to not trigger any crashes in 10 minutes even if the code had bugs? How would you address one of them?

Question (g): What would happen if you were to run AFL on one of the targets from project 1 (give it a try!)? If you think AFL will produce any crash test cases, are those test cases likely to be a solution to that target? Why or why not?

Submission Template

Upload to Canvas a gzipped tarball (`.tar.gz`) named `hw1.tar.gz`. The tarball should contain only the files below:

```
crash-input1.json
guff-seed.txt
crash-input2.txt
answers.txt
```

The format for `answers.txt` should be:

```
# Questions

  a.

  b.

  c.

  d.

  e.

  f.

  g.
```