SWEN 438 *Special Topic: DevOps*

Laboratory 5: American Fuzzy Lop

The purpose of this laboratory is to get some practical experience with fuzzing using a tool called American Fuzzy Lop (AFL). This is a widely-used tool developed by Google and used, for example, in Project Zero.

Getting Started

The starting point for this lab is to get AFL setup. Since AFL is installed on the ECS machines, the easiest approach is to use them (e.g. via `ssh`) as everything can be completed through a terminal. For example, you can log into `barretts.ecs.vuw.ac.nz` or `regent.ecs.vuw.ac.nz` via `ssh`.

If you prefer to get AFL running on your local machine, there are some notes in the Appendix.

Part 1: Build Example Application

A very simple example application has been provided for this lab. Download and expand the file `example.tgz` from the course website. Then, check that it works by compiling it in the usual fashion, and testing it on one of the example files:

```
> g++ -o main main.cpp
> ./main inputs/test_01.txt
Added 2 words to dictionary...
hello world
```

If you see the above two lines of output from the program, then it is working correctly. You can examine the input files by hand. For example, the input file `test_01.txt` is as follows:

```
world hello
1
0
.
```

The interpretation of this is fairly straightforward. A "dictionary" is populated from the first line of the file. The remaining lines are either indices into this dictionary, or the special operator `.` which generates a newline. Thus, the above program prints word 1 from the dictionary first (i.e. `hello`) then word `0` (i.e. `world`), followed finally by a newline. *You can try out some different input files if you wish.*

Part 2: Fuzz Example Application

At this point, we will run AFL on our application. Since our application has numerous problems (i.e. potential security vulnerabilities), we should be able to find them!

There are two steps to the process: first, we compile our application with AFL using `afl-g++`; second, we run our application using `afl-fuzz`. The first part here allows AFL to insert instrumentation into our program which helps it to see how different inputs affect execution. To test with AFL, run the following:

```
> afl-g++ -o main main.cpp
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 117 locations (64-bit, non-hardened mode, ratio 100%).
```

NOTE: You may observe some an error message here: if you have `.` on your PATH environment variable (or $AFL_PATH), AFL does not like this and you'll need to modify your PATH accordingly.

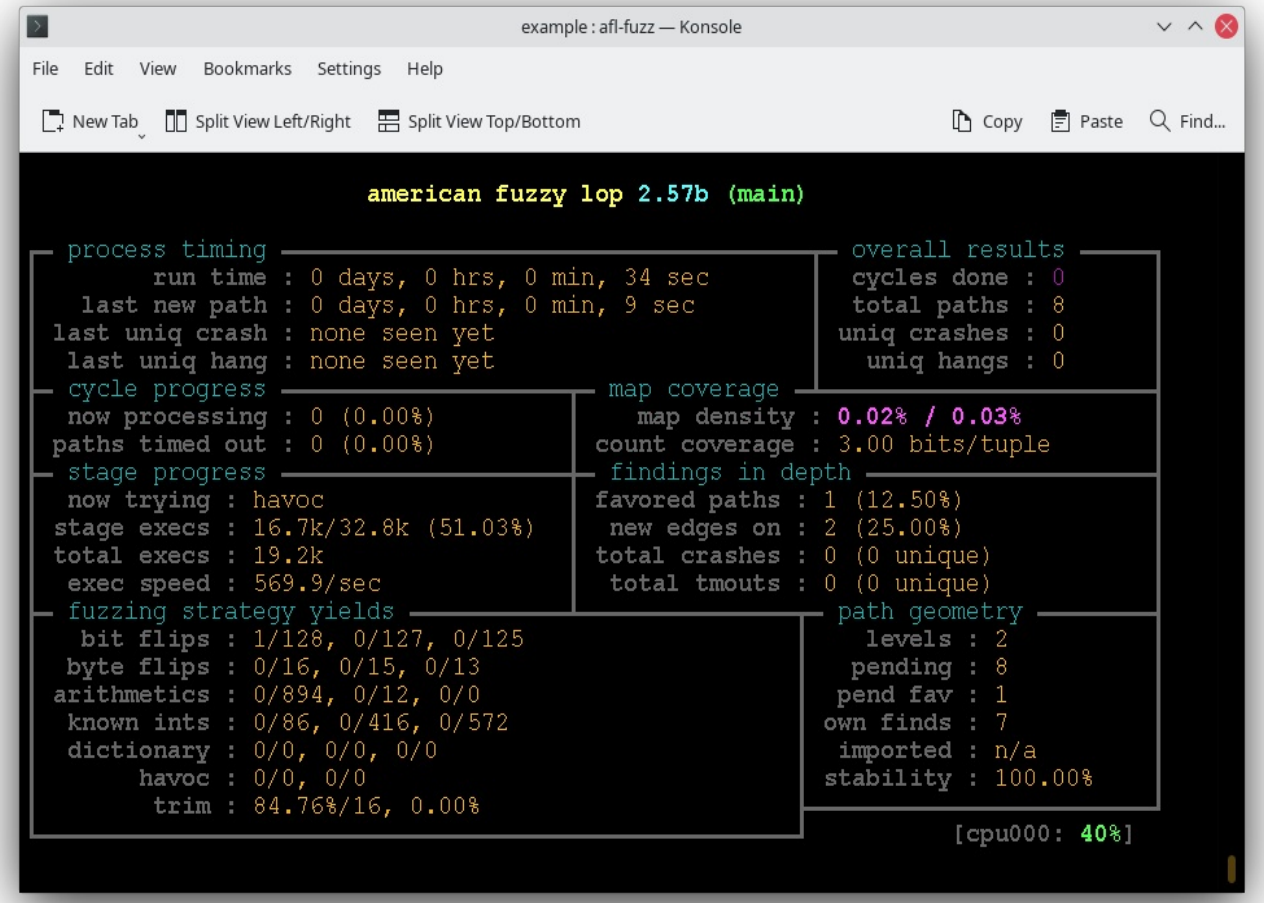
Assuming it has compiled correctly, you can now run AFL as follows:

```
afl-fuzz -i inputs -o outputs ./main @@
```

NOTE: You may observe an error message related to problems with "on-demand CPU frequency scaling". To workaround this, either follow the instructions given or (perhaps easier) you can set the environment variable `AFL_SKIP_CPUFREQ` as follows:

```
export AFL_SKIP_CPUFREQ=1
```

The command inline tells AFL where to find the *seed inputs* (`-i`), where to place outputs it generates (`-o`) and that it reads from input files (`@@`) rather than `stdin`. The seed inputs are used by AFL to automatically generate other inputs. When AFL is working, you should see a screen which looks roughly as follows:



This indicates that AFL is now testing your application. Observe that this will continue essentially indefinitely! At some point, you'll want to stop it. But, leave it for a while to find some problems (e.g. *at least* 10mins). On my machine it took only a few seconds to find its first crash, but quite a lot longer to find a hang (**NOTE:** for some reason on my machine the crash counter disappeared when it found its first crash, rather than showing e.g. 1).

Once you've stopped AFL, you want to look at the output its provided. In particular, if it did find some problems, then look at the files inside `outputs/crashes` and `outputs/hangs`. You should find they are variations on the test input files provided. For example, this is one that it found:

```
worl@hrworl@hr@1
71
0
.
```

Part 3: Fuzzing a Real-World Application

The next part of the lab is to do something more realistic. In this case, we are going to fuzz a real-world scientific application used for computing Tutte Polynomials.

Getting Setup

Download the latest version (`0.9.18`) of the code from its homepage, and unpack it into a temporary directory:

```
> tar xvzf ~/Downloads/tuttepoly-v0.9.18.tgz
tuttepoly-v0.9.18/
tuttepoly-v0.9.18/tutte/
tuttepoly-v0.9.18/install-sh
tuttepoly-v0.9.18/INSTALL
...
```

This project uses a `configure` script to generate an appropriate `makefile` for building the code. We need to tell the `configure`

script to use `afl-gcc` and `afl-g++` instead of `gcc` and `g++`. We do this by setting two environment variables before running the script:

```
> export CC=afl-gcc
> export CXX=afl-g++
> ./configure
checking for a BSD-compatible install... /bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether the C++ compiler works... yes
checking for C++ compiler default output file name... a.out
...
```

If you look in the output from `configure` you should see that it refers to `afl-gcc` and `afl-g++`. At this point, we need to `make` the project as before:

```
> make
make  all-recursive
make[1]: Entering directory '/home/djp/scratch/tuttepoly-v0.9.18'
Making all in nauty
make[2]: Entering directory '/home/djp/scratch/tuttepoly-v0.9.18/nauty'
afl-gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT naugraph.o -MD -MP -MF .deps/naugraph.Tpo -c -o naugraph
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 755 locations (64-bit, non-hardened mode, ratio 100%).
mv -f .deps/naugraph.Tpo .deps/naugraph.Po
afl-gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT nautaux.o -MD -MP -MF .deps/nautaux.Tpo -c -o nautaux.o
...
```

Some warnings are emitted as this is a relatively old code base, but we can ignore them here. We can test its working by running the code on one of the example input files:

```
> tutte/tutte examples/edge10
G[1] := {0--1,0--2,0--3,0--4,0--5,0--6, ... }
TP[1] := 23*y + 109*y^2 + 244*y^3 + 338*y^4 + ... + 10*x^8 + 1*x^9 :
```

Fuzzing Tutte

We're now ready to fuzz the `tutte` program, but there are bunch of things we need to know:

1. **(Inputs)** There are a number of input files in the "`examples`" directory which can be used for seeds. I'd suggest making a fresh directory "`inputs`" and copying over one or two files only (e.g. `edge10`).
2. **(Memory)** The `tutte` program (by default) attempts to pre-allocate `512m` of RAM on initialisation. By default, `afl-fuzz` limits the amount of memory child process can allocated to `50m`. We need to change one of these so they are compatible, otherwise nothing useful will happen (the `tutte` program will just abort everytime).
3. **(Timeout)** The `afl-fuzz` tool limits the execution time of child processes to `20ms` by default. This is not long enough for the `tutte` program to complete on most of the example inputs.

Taking all of this into account, here is an example fuzzing command line we can use:

```
> afl-fuzz -t 60000 -m600M -i inputs -o outputs tutte/tutte @@
```

This sets the memory limit for child processes to `600M` and a timeout of `60000ms`. Fuzzing should then begin, but you will notice it takes a fair bit longer.

NOTE: If you see `odd, check syntax` in the status window whilst fuzzing then that means something is wrong, and its probably not going to find anything.

Another example command line we can use is the following:

```
> afl-fuzz -t 60000 -m64M -i inputs -o outputs tutte/tutte -c32M @@
```

This limits the memory allowed for child processes to `64M` and instructs the `tutte` program to preallocate only `32M` of RAM. *This means it will take longer to return a result on the larger examples.* But, it also cycles the cache more frequently.

At this stage, your goal is find some problematic inputs (either a crash or a hang is sufficient). *This may take some time*. For example, the tool did find some problems after about 6hours of fuzzing. However, you may be able to speed this up by tweaking the parameters.

NOTE: finding a *hang* is sufficient for this lab as it is unknown whether a *crash* even exists!

Submission and Marking

For this lab, please submit at least one crash file for each application to illustrate detected problems via the ECS Submission system https://apps.ecs.vuw.ac.nz/submit/SWEN438/Laboratory_5.

Full marks will be earned for meeting the following criteria:

1. Evidence that AFL has been used to conduct experiments on a simple application
2. Evidence that AFL has been used to conduct experiments on a real-world application

Appendix

Setting up AFL on UNIX and MacOS

For those on a UNIX-like environment, its pretty easy to get AFL up and running by building from the source. You will need a C compiler installed (hence, on Mac you will need the XCode command line utilities installed). The instructions are given below:

1. **(Download)** Clone the AFL git repository
2. **(Build)** Within the AFL directory, run `make`.
3. **(Configure)** Set the `AFL_PATH` environment variable to the repositories location (e.g. `setenv AFL_PATH dir` for `csh` or `export AFL_PATH=dir` for `bash`).
4. **(Test)** Check the tool works by running `$AFL_PATH/afl-g++`. You should see the tool generates some output.

Setting up AFL on Windows

For those on a Windows environment, there are a few options:

1. **(Virtual Machine)** You can setup a virtual machine (e.g. `VirtualBox`) image for Ubuntu 20.04 (or similar). Then, follow the instructions above.
2. **(WinAFL)** For those who are adventurous you can try to build AFL from source. A fork of AFL is available specifically aimed for Windows machines WinAFL.
