Lab 4: Shell Scripting & Data Processing

Benedict Reuschling

February 21, 2019

1 Working with Calendar Dates

Superstitious people think that friday, the 13. of a month is an unlucky day. Regardless of whether you believe in that or not, write a script that does the following: echo all months and years for the next 50 years, on which there is a friday on the 13. The output should be done in tabular form. The system provides you with two calendar programs (`cal(1)` bzw. `ncal(1)`) which you can use to get the information you need.

Once your program is working, extend it to let the user pass the start date (in the form `MMYYYY`) as the first parameter and the number of years to look into the future as the second argument.

2 License Plate Generator

Write a shell script that generates possible combinations of license plates for residents of the city of Darmstadt. Users can enter a partial license plate and the script will return all possible combinations for missing characters, denoted by underscores (_). For example, the user enters the following:

```
License plate: DA-_C 12_
DA-BC 120
DA-BC 121
DA-BC 122
...
DA-FC 120
DA-FC 121
...
DA-QC 129
```

License plates must begin with `DA` or `DI` with a combination of letters or numbers containing

- one or two characters of the series B, F, G, I, O or Q and a number between 1 and 999 **or**
- two characters and a number between 100 and 999

Invalid license plates should be rejected (i.e. `XY-Z1_3` or `DI-ABC_`) when entered and the generator should only run on valid input.

3 Finding Problems using Fuzz Testing

To test programs for robustness against different inputs, so called fuzzers can be used. They can generate random, invalid input that is being fed into programs to see how the programs behave. When the program crashes repeatedly for certain inputs, it can be an indication that there is a bug in the program. The fuzzer logs which input produced which kind of results (i.e. endless loops, crashes) to provide programmers with hints about which inputs need better validation. When the programmer has fixed the bug, another fuzzer run can check whether the problem is solved. Note that fuzzers only find certain kinds of errors and problems, which are often simple. When fuzzers don't find anything, it does not mean that the software is free of other types of bugs.

3.1 Setup

We will be using a fuzzer called American Fuzzy Lop[1] to test programs for invalid input. These steps are best done from the command-line.

1. Download and install the program using `pkg install afl` (as root or using `sudo`).
2. Create a new directory called `afl` in your home directory with two subdirectories: `tests` and `findings`. The `findings` directory will later contain the results of our fuzzing run and `tests` will be filled shortly with basic inputs that the fuzzer will feed to the program.
3. Copy the file `mylist.cpp` that was supplied as part of this lab into the `afl` directory. You can use `fetch` with the direct URL to the file on my homepage to start the download.
4. Look at the source file, using either `cat`, `less` or an editor like `vim`. You'll notice that the indendation is not what it should be like. Use `indent` to pretty-print that source file (check out the man page for various formatting options). Create a patch (in unified format) between the old and new file.
5. Compile the program using either `afl-clang++` or `afl-g++` and check whether it compiles without errors (the command is similar to regular clang/gcc compiles). Use the `-o` switch to create a binary with a useful name.
6. Run the program (using the name you just gave it), provide some inputs and note the exit condition. This should make you familiar with the program in general, what it does (and what it does not do), what inputs it takes, etc.

3.2 Running the Tests

After you've set up the fuzzer, created the necessary directories and checked that the program compiles, it is now time to check whether is has any errors.

1. Create a file within the `tests` directory that contains the exit condition for the program that you found out by running the program (basically, the inputs for one run on a single line). This will be used to initialize the fuzzer.

[1] `http://lcamtuf.coredump.cx/afl/`

2. Start the fuzzing run by invoking the fuzzer program:

```
$ afl-fuzz -i tests -o findings ./<your-compiled-binary>
```

   The program will launch an overview screen that displays how many crashes or hangs it has found already. Let it run for a while (ca. 3 min.) and then quit using Ctrl + C. The longer you let the fuzzer run, the more likely it is that multiple error conditions are found.

   Note that this is a slow fuzzing run and there are methods to run the fuzzing in parallel. But for our purposes, it is sufficient at this speed.

3. Check the `findings` directory. Of particular interest are the `crashes` and `hangs` directories. Each file in there represents an individual test run against the program. The contents of the files show how the original input file from `tests` was changed (mutated) and then used as input to your program.

   Some statistics about the last fuzzer run are contained in `fuzzer_stats`. The `queue` directory contains further tests that were queued up to be used as next inputs.

4. Use the `help` option to `afl-plot` to see how you can create a nice graphical representation of your test run, based on the information contained in `findings/plot_data`. Save this output in a separate directory, as it will be overwritten the next time you start another fuzzer run.

3.3 Fixing the Bug

1. Now that you know for which inputs the program will show undesired behaviour, you can go ahead and fix the program. First, verify that the inputs created by the fuzzer are actually affecting the program (i.e. causing endless loops).

2. Change the source code to handle *one* of the errors found. Refresh your knowledge about proper input validation in C++ when necessary to fix the bug.

3. Re-run the fuzzer and see whether the error is still present or has been fixed. Re-create the plots as well.

4. When you've fixed the bug, demonstrate that your version of the program does not have *one* of the bugs found by the fuzzer. Create a patch using the `diff` program in unified diff format between the original program and your modifications.

   You can also show the old graphs versus the new ones for comparison. Firefox or Chrome can display the PNG files.