Lab 1: Basic Fuzzing Framework (BFF)

Summer Semester 2018

Due: May 28th, 8:00 a.m. Eastern Time

Corresponding Lecture: Lesson 3 (Random Testing)

Objective

The goal of this lab is to gain familiarity with random testing and obtain hands-on experience with a fuzzing tool used in industry. Specifically, we will use CERT's Basic Fuzzing Framework (BFF) to fuzz Linux applications. The BFF targets applications that consume files as input by mutating a set of seed files to probe for errors in handling improperly formatted or corrupted files. Of particular interest are errors discovered that could present a security vulnerability, which are labelled as such by the BFF.

In addition to fuzzing common Linux utilities included in the Ubuntu 16 LTS distribution, you will also fuzz a C++ application we provide. Using the output provided by the BFF and the source code for the application, you will find the source of the bugs in the code, and determine how to correct them.

Resources

1. About BFF:

   https://www.cert.org/vulnerability-analysis/tools/bff.cfm

2. About `pnmnorm`:

   http://manpages.ubuntu.com/manpages/precise/man1/pnmnorm.1.html

Synopsis

The general steps that you should take when running a fuzzer are as follows:

1. Decide upon the proper tool for your program under test and its configuration. For this lab, we have selected BFF as the proper tool but you will be responsible for its configuration such as what kinds of fuzzers to use and for how long.
2. Set up the program to be tested. This step includes determining what command and arguments you should use to run the program and picking appropriate seed files.
3. Process the results to isolate and fix bugs. You likely will use a combination of automated test case minimization and manual debugging.

Setup

BFF is pre-installed on the course VM. You will need to download, import, and start the VM to begin the lab. Please refer to the course syllabus for information about the course VM.

A fuzzer usually provides various kinds of debug information such as minimized test cases, call stack information, or memory state information. For this lab we are going to focus on minimization so we want to turn off the `valgrind` option that is turned on in the the pre-installed version of BFF. Valgrind is a memory analysis tool and causes a large performance hit so disabling it is critical to ensure your BFF campaigns run within the time limits.

First, open the `bff.yaml` file and at the bottom under `analyzer` options, change `use_valgrind: True` to `use_valgrind: False`. From the comments, we can see this will also turn off Callgrind, which is a resource-intensive call stack analysis tool included with Valgrind.

Updates to `~/bff/configs/bff.yaml`:

```
analyzer:
 exclude_unmapped_frames: True
 savefailedasserts: False
 use_valgrind: False
 use_pin_calltrace: False
 valgrind_timeout: 120
```

Second, download and extract the lab resources (`bff.zip`) provided on Canvas into the `~/` directory. It should create the following two directories:

1. `~/bff/csvParser` - Contains C++ source code for an application you will fuzz in this lab.
2. `~/bff/seedfiles/csv_examples` - Contains seed files useful for fuzzing this application.

Next, build this application by executing the following command in the `~/bff/csvParser` directory:

```
clang++ -std=c++11 -g -o csvParser csvParser.cpp
```

This should create an executable application named `csvParser` you will fuzz later in this lab.

Demonstration

First, let's run a demo fuzzing campaign using the BFF. Open a terminal emulator and navigate to the `~/bff` directory. Start the fuzzing campaign by executing the `batch.sh` shell script. This will run the default campaign, which fuzzes the ImageMagick utility with a variety of mutated input files.

NOTE: You may see OS popup warnings that applications have stopped working. You may safely close and ignore these warnings. This is true any time you run this tool, not just during the demonstration.

Throughout the test, you will see various mutated seed files successfully execute and some that will not. When a crash occurs, the test case causing the crash will be analyzed further and recorded. If a duplicate crash is encountered, the BFF will recognize the test case signature and avoid redundant analysis. Let the test run for five minutes, then stop the campaign by pressing `CTRL + C` in the terminal emulator.

Next, examine the results of the testing campaign. Navigate to the `~/bff/results` directory in a file manager window. Inside, you should see a result folder for `convert_v5.2.0`. Within this folder, you will see the following (not an exhaustive list):

1. `bff.yaml`: A copy of the campaign file the BFF used to execute the fuzzing campaign.
2. `uniquelog.txt`: A log of the details of unique crashing test cases.
3. `/crashers`: A subfolder containing classification and analysis output per unique test case, including (not an exhaustive list):
    a. Output from GDB
    b. Log output from the test case minimizer
    c. A minimized input to reproduce the crash
    d. The full input that caused the crash
4. `/seedfiles`: A subfolder containing the seed files used during fuzzing.

In this demonstration, you used the default fuzzer, a byte mutator that randomly replaces bytes in the seed files to generate test cases. There are a variety of other fuzzers available that you can select based on the application being tested. You can find descriptions of these fuzzers in the campaign file (`bff.yaml`) comments.
Before proceeding to the rest of the lab, take some time to familiarize yourself with the following key components of the BFF:

1. BFF Readme file (`~/bff/README`): Contains general information about the operation of the tool and a detailed explanation of the demo you just ran.
2. Campaign files (`*.yaml`): These files define the exactly how the BFF should execute a fuzzing campaign. It also contains detailed comments about the various campaign settings and how they determine the behavior of the BFF during a fuzzing campaign.

Lab Instructions

Part 1 - Fuzzing a Common Linux Utility

In the first part of this lab, you will configure the BFF to fuzz a Linux command line utility included as part of the Ubuntu 16 LTS distribution: `pnmnorm`. After running fuzzing campaigns on this utility, you will also analyze the results and report your findings. Your goal is to play role of a tester who seeks to find a crashing input, minimizes the input, and sends reproducible information to a developer. As the tester, you do not have access to the program's source code.

To configure your campaigns, you will need to edit the campaign file (`bff.yaml`) located in the `~/bff/configs` directory. Seed files to use in fuzzing these utilities have been provided for you in the `~/bff/seedfiles` directory.

Run BFF campaigns on `pnmnorm`. You should run campaigns using multiple fuzzers and your goal is to find two fuzzers that crashes the program. You should try each fuzzer campaign for 5 minutes.

NOTE: You should provide a unique campaign ID for separate campaigns. Otherwise, your results will combined into a single results folder.

You will submit the following items for each program in Part 1. See the "Items to Submit" section at the end of the document for details on how to submit these files:

- The minimized seedfile that caused BFF to crash
- The campaign configuration you ran in BFF to cause the crash found in the `yaml` file
- The crashing output from the `uniquelog.txt` files produced by BFF

Part 2 - Fuzzing a C++ Application

In the second part of this lab, you will configure the BFF to fuzz a C++ application we have provided which contains bugs that we have intentionally introduced. Your goal is to find **two** distinct bugs in the application by identifying fuzzers that are most likely to find the bugs, and configuring the BFF to run campaigns to discover crashing test cases for these bugs. For this part of the lab, you are playing the role of a developer, so you have full access to the source code for the program under test.

First, create a BFF campaign file that targets the `csvParser` application you built in the lab setup. For seedfiles, use the files provided in the `~/bff/seedfiles/csv_examples` directory. The `csvParser` application takes one command line parameter, the file path of a csv formatted file to be parsed. As noted in Part 1, you should use a unique campaign ID to keep each fuzzing campaign separate in the `~/bff/results` folder.

During your fuzzing campaigns, be careful not to kill the campaign with `CTRL + C` until you have found enough unique crashes. If you do not have crashes in your results, you likely need to run BFF for longer. Some programs may require a 10 minute campaign to produce sufficient results. If your campaign keeps reporting the same test case signature (the hexadecimal ID number) or fails to report any failing test cases, this may indicate that your campaign is no longer producing useful results. If this occurs, you will be more likely to generate new failing test cases by using a different fuzzer rather than letting the campaign run longer.

Once you have successfully generated failing test cases, you can analyze the data generated in the `~/bff/results` directory to identify bugs in the application source code. You may also find it helpful to run `csvParser` from the command line against the mutated file to see the crash for yourself. It is very possible that different BFF campaigns will find the same underlying bug in the application. If your generated failing test cases do not identify more than one distinct bug (error made by the programmer at a specific location in code), you will need to run additional campaigns to generate more data to find a second bug.

Another useful tool debugging can be the GDB utility. You can see some outputs from gdb in the `~/bff/results` directory, but you may find it helpful to run the program using gdb. If you're not familiar with gdb, you can find a useful tutorial here. However, using GDB is not required - feel free to debug using whatever method you are most comfortable with.

Once you have identified two distinct bugs in the `csvParser` application, answer the following questions using 100 words or less per question or subquestion:

1. Which fuzzers did you select for running a BFF campaign? Why did you choose these fuzzers? For full credit, make sure to base your answer of "why" on the behavior of each fuzzer. "I selected every fuzzer because I wanted to test them all" is not the type of answer we are expecting.
2. Excluding the verify fuzzer, what are some fuzzers that are ineffective at finding a crash? Why do you think these fuzzers were ineffective at finding crashing test cases?
3. Using the data generated by the BFF campaigns and other debug utilities you ran as a guide, examine the source code in `csvParser.cpp` and answer the following questions for the **first** bug that you found in the `csvParser` application:
    a. Which line of code in `csvParser.cpp` contains the bug?
    b. Explain why the failing input discovered by the BFF caused the application to crash.
    c. How could you fix the bug so that the program will execute properly?

4. Using the data generated by the BFF campaigns you ran as a guide, examine the source code in `csvParser.cpp` and answer the following questions for the **second** bug that you found in the `csvParser` application:
    a. Which line of code in `csvParser.cpp` contains the bug?
    b. Explain why the failing input discovered by the BFF caused the application to crash.
    c. How could you fix the bug so that the program will execute properly?

Items to Submit

Submit the following files in a single compressed file (`.zip` format) named `lab1.zip`:

1. (5 points) The campaign file (named `pnmnorm1.yaml`) you created to run the first successful fuzzing campaign on `pnmnorm`.
2. (5 points) A single file named `pnmnorm_uniquelog1.txt` that is the `uniquelog.txt` file generated by BFF for your first successful fuzzing campaign targeting `pnmnorm`.
3. (5 points) The minimized input file named `pnmnorm1.ppm` or `pnmnorm1.pgm` (the extension will vary depending on which seed file causes the crash) that caused a failure in your first successful fuzzing campaign on `pnmnorm`.
4. (5 points) The campaign file (named `pnmnorm2.yaml`) you created to run the second successful fuzzing campaign on `pnmnorm`.
5. (5 points) A single file named `pnmnorm_uniquelog2.txt` that is the `uniquelog.txt` file generated by BFF for your second successful fuzzing campaign targeting `pnmnorm`.
6. (5 points) The minimized input file named `pnmnorm2.ppm` or `pnmnorm2.pgm` (the extension will vary depending on which seed file causes the crash) that caused a failure in your second successful fuzzing campaign on `pnmnorm`.
7. (5 points) The campaign file (named `csvParser.yaml`) you created to run the fuzzing campaign you felt most effect for finding failing test cases in `csvParser`.
8. (5 points) A single file named `csvParser_uniquelog.txt` combining the `uniquelog.txt` files generated by the BFF for all fuzzing campaigns targeting the `csvParser` application.
9. (60 points) A PDF file named `responses.pdf` containing your answers to the questions asked at the end of Part 2 of this lab.