

CAP6135: Programming Project 2: Fuzzing (Spring 2012)

This project is modified from the programming project 2 in Dr. Dawn Song's course "CS161: computer security" in Fall 2008:

<http://inst.eecs.berkeley.edu/~cs161/fa08/>

This is a group project. Each group should have two students. You are responsible to find your partner. Of course, you can do it alone, but it means you will need to work more than other students do.

I have put a tar compressed file on webcourse in this project submission page. You will need to download it and use it for completing this project.

Delivery:

Submit a zip file through WebCourse. The zip file should contain:

1. Two source code files of the mutation-based fuzzer and the protocol-aware fuzzer. Your programs must be programmed using a language that can be run on Eustis (explain clearly how I can compile and run your code in your report!)
2. The input files (modified jpg image) that could trigger each of the bugs you can find. For each bug, just provide one input file. Name the input file as "test-x.jpg" for Bug # x. In this way, I can easily check whether you really found Bug #x by running it by myself!
3. A 3-5 page project report (in PDF). The detailed requirements of this report are discussed at the end of this project description.

Fuzzing:

The goal of this project is to implement a "fuzzer", or fuzz tester. Fuzz testing is one way of discovering security vulnerabilities in any code that processes potentially malicious input. The concept of a fuzzer is simple; it repeatedly runs a program which is being evaluated on a series of automatically generated inputs. If some input causes the program being tested to crash with a memory access violation, the program has failed the test and may have an exploitable security vulnerability. By generating inputs either partly or completely randomly, the fuzzer can rapidly run large numbers of tests without human supervision and also may discover bugs which occur under unusual inputs which the developer may have overlooked.

A *mutation-based fuzzer* takes a valid input for the target program, and works by creating random mutations or changes to generate new test cases. Mutation-based fuzzers are application independent, and so they do not have any knowledge about input format (protocol format) accepted by the target program. In contrast, a *protocol-aware fuzzer* is a fuzzer that is directed towards particular applications, and possesses knowledge about the input format of the target applications. For instance, a fuzzer designed to fuzz web browsers understands HTML syntax, and thus can potentially create invalid test cases to stress the browser's logic. The goal of this project is to design a protocol-aware fuzzer, and empirically compare it with a mutation-based fuzzer.

Implementation:

In teams of two, **you will implement two fuzzers: a mutation-based fuzzer and a protocol-aware fuzzer**, both capable of finding bugs in real-world programs. Each team may implement their fuzzers in the programming language(s) of their choice, as long as they can be executed on the Eustis machine (eustis.eecs.ucf.edu).

As part of the project, we have introduced a number of vulnerabilities into a JPEG to PPM image converter program (called jpegconv) which we will provide to you as a **binary executable**. You can use your fuzzer to uncover some of these bugs. At the end of the project, each team will submit a list of the bugs they found, and the number of distinct bugs found in the test program will form part of each team's grade. Not all of the bugs will be equally easy to discover, however. Whether or not you discover some of the more subtle bugs may depend on the sophistication of your fuzzer design. The fuzzer implemented by each team will be run from the command line on Eustis machine and will be given arguments which specify how it should invoke another program which is to be tested.

Program Specs

The design of the fuzzer will be based around two modes of operation: *search mode* and *replay mode*.

In search mode, the fuzzer will repeatedly invoke the program being tested with a series of inputs, each time checking to see if the program crashes. **If the target application being tested evaluation prints an error message or immediately exits, it is not considered as having a bug.** In general, many of its inputs will be invalid in some way, so in those cases such behavior is appropriate. We are only concerned with discovering inputs which cause the program to crash, which should not happen under any circumstances. For our purposes, we will consider a program to have crashed **if it exited due to signal 11 (SEGV). Before the "jpegconv" program crashes due to a triggered bug, it will outputs "BUG X TRIGGERED" to the screen using stderr.** From this message you can know which bug in the program has been triggered. (I have put the short example code of how to check SEGV in **Start.rtf** file).

When fuzzing, at least some aspects of each new test case generated will be selected randomly. When the fuzzer is operating in search mode, it must ensure that, when making any random choices in constructing a particular input, those choices are based on a **pseudorandom number generator** (PRNG) for which it has saved the seed. If it is determined that a particular input caused the test program to crash, the fuzzer will then output that seed for use in replay mode. In replay mode, the fuzzer will take as an additional argument a seed that it produced while running in search mode. Using that value to seed the PRNG, it will then reproduce and output for reference the exact input that previously caused the test program to crash.

When run in search mode, the fuzzer should have the following command line format:

```
fuzzer --fuzz-file sample.jpg jpegconv test.jpg
```

will first read sample.jpg, modify it (mutations or protocol-based modifications), save to test.jpg and then invoke jpegconv and pass test.jpg as its parameter.

For either of the two fuzzers you implement, if after some number of trials your target application crashes with a segmentation fault, the fuzzer should then print the corresponding seed

as string (e.g., 24cdb33d7e) on a single line with no spaces on standard output. Note that you will probably want your fuzzer to remove generated input files after each test case so they do not accumulate.

When the argument `--replay SEED` is given as an option appearing before the name of the command to be run, the fuzzer should operate in replay mode. In this case, it should seed its PRNG with the seed `SEED`, then produce input files and input strings as it normally would. The command line should be:

```
fuzzer --replay 24cdb33d7e --fuzz-file sample.jpg jpegconv test.jpg
```

In replay mode the last generated input file `test.jpg` should not be removed, so after this example executes the file `test.jpg` should remain in the working directory for inspection.

The format of `SEED` is up to you, provided it contains all the information necessary for your fuzzer to reconstruct the same input (when run with the appropriate arguments).

Mutation-based Fuzzer

You should first write a simple mutation-based fuzzer. The mutation based fuzzer can take a valid input file (such as the `sample.jpg`) via the `--fuzz-file` argument, and use it as a base case for generating new test cases. To generate a new test case, you may randomly modify one or more bytes in the input file, or generate completely new random data and add it anywhere to the file.

Once implemented, you should run your fuzzer in search mode on the binary JPEG to PPM converter program (`jpegconv`) provided by us. You should report how many bugs did this fuzzer find, along with the number of trials and number of different valid fuzz files you used to find these bugs. As mentioned, when you find a crash, you should store the seed used to generate the crash case so that we can replay your tool on your valid input file to regenerate the crash.

Protocol-aware Fuzzer

To use the protocol knowledge about your target application you may need to understand the file format for JPEG (JFIF) image file format (JPEG File Interchange Format) to some extent. For your convenience, we have printed the format structure for the `sample.jpg` file as "`sample.format`". You need not understand the complete specification for JPEG file format. The header formats appearing in `sample.format` should be sufficient to uncover enough bugs to receive full credit.

There are numerous resources on JPEG specifications online, so look for them. Such as

1. <http://en.wikipedia.org/wiki/JPEG>
2. <http://www.obrador.com/essentialjpeg/headerinfo.htm>

Help

When you uncompress the tar file on webcourse, besides the bugged `jpegconv` executable code, you will also find a normal example picture `sample.JPG`, its format file `sample.format`, and a help file `START.rtf`. The help file will tell you more detailed information on the project.

Project Report Requirement

Please submit a project report on 3-5 pages. Consider this to be also a software engineering project and include sections such as Analysis, Design and Implementation.

Design

Briefly explain your strategy used to implement the mutation-based and protocol-aware fuzzing tools. Specifically, you should point out the advantages that your design/implementation provides. For instance, you may consider the following design points:

- Useful heuristics to generate good test inputs.
- Extensibility to other protocols.
- Scalability to large programs.

Empirical results

Show two graphs describing the results from running the mutation-based fuzzer and the protocol-aware fuzzer respectively. We suggest that you plot the number of trials on the x-axis versus the number of bugs found on the y-axis, to indicate how effective each tool was in finding bugs; however, you may find different parameters more meaningful to plot. Be sure to clearly state how many distinct bugs did the two tools find and how many trial runs were necessary to uncover the bugs for each of the two tools.

In addition, use a table to show the SEED values for bugs found by your Mutation-based and Protocol-aware fuzzers.

Show the screenshot image of each bug you have triggered. When you find an input modified image file 'test.jpg' that could trigger Bug #1, name the file as 'test-1.jpg', save it. Then when you run:

```
$ ./jpegconv -ppm -outfile foo.ppm test-1.jpg
```

The jpegconv should crash and print out:

```
BUG 1 TRIGGERED
```

Conclusion

The goal of the project is to decide which strategy is most effective for fuzzing media file handling applications. Based on your empirical evaluation, tell us which strategy did you find to be most effective, and how do your empirical results support this claim?

Grading

(32 points)

There are at least 10 bugs in the sample program. You need to find any 8 of the 10 to receive full credit. For each bug found by your tools in the sample program given to you, you'll be awarded 1 point. You can receive a maximum of 8 points. *A bug is confirmed to be found only if you have a screenshot image showing the crash and triggering of the bug.*

(8 points)

Design of your mutation-based fuzzer.

(8 points)

Implementation of the mutation-based fuzzer. It must compile and should be easy to run.
Must include executable and readme files.
Readme file giving execution details (command line parameters. SEED etc) for reproducing your results, i.e., Bugs).

(16 points)

Design of your protocol-aware fuzzer.

(24 points)

Implementation of your protocol-aware fuzzer. It must compile and should be easy to run.
Must include executable and readme files.
Readme file giving execution details (command line parameters. SEED etc) for reproducing your results, i.e., Bugs).

(12 points)

Empirical results presented in the project report, and conclusions drawn.