CAP6135: Programming Project 2: Fuzzing
(Spring 2011)

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. 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 ones 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 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.

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 will have the following interface.

```
fuzzer [OPTION ...] CMD [CMDARG ...]
```

Invoked in this way, the fuzzer will run `CMD` with any listed arguments (`CMDARG`). For example:

```
fuzzer echo foo
```

would cause the fuzzer to repeatedly run echo foo, checking each time to see if it crashes.
If the tested program requires an input file, the fuzzer will support `--fuzz-file` parameter:

```
fuzzer --fuzz-file sample.html firefox test.html
```

will first read sample.html, modify it (mutations or protocol-based modifications), save to test.html and then invoke firefox and pass test.html 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 will 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. At that point, it should print the command line it would have executed, then exit.

For example,

```
fuzzer --replay 24cdb33d7e --fuzz-file sample.html firefox test.html
```

might cause the target application, firefox to print some garbage output and then crash. In replay mode any generated input files should not be removed, so after this example executes the file test.html 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). Note in particular that if your fuzzer sometimes generates inputs or parts of inputs by running through of a sequence of values or through some other deterministic means, then the seed printed will also have to include any necessary information about what step it was on.

Mutation-based Fuzzer

You should first write a simple mutation-based fuzzer. The mutation based fuzzer can take a valid input file 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

Mutation-based fuzzers may be limited in the types of bugs they discover or may be too inefficient. For example, many programs read the first two bytes of an input file, compare them to a “magic number” for the expected file format, and exit immediately with an error message if they do not match. Since a mutation based fuzzer does not understand the file format of the target program, it may generate such invalid inputs for the first two bytes several times, leading to poor code coverage. Thus, it is necessary to generate inputs with at least some degree of validity in

order to find all but the most trivial bugs. Tailoring the inputs generated by a fuzzer to the program being tested so that subtle bugs can be discovered is an art that requires knowledge of the file format read by the program. Therefore, you must write a protocol-aware fuzzer that understands the file format of the target program.

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. You need not understand the complete specification for JPEG file format. In this project, we will restrict our study to a small subset of JPEG headers that are most commonly seen in JPEG images seen on the web. Specifically, we provide you a sample JPEG file named sample.jpg and the header formats appearing in this file 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

Besides these you can also use hachoir, which is a software suite for file analysis. This tool is capable of dissecting JPEG headers and showing you their structure in a programmer-friendly way. The format structure of given JPEG file (sample.jpg) using image dissection tool (hachoir-urwid) is also included as "sample.format", which should be sufficient for completing the project.

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 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.

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.

(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.