Project 2 Description

CS161 Computer Security, Fall 2008

Assigned : 10/27/2008

Due Date : 11/17/2008

1 Overview

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.

We have studied various fuzzing strategies in class. 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.

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.[1] 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 find this program, along with a sample JPEG file to invoke it with, as an archive at :

`/home/ff/cs161/proj2-fa08/proj2-target.tar.gz`

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

[1] We only require that your code compile and run (or just run if it is written in an interpreted language) on the department's instructional Linux machines, specifically, ilinux{1,2,3}.eecs.berkeley.edu. So in particular C, C++, Java, Python, Perl, and Ruby, are all fine.

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. The programs it tests may or may not be graphical, it is only necessary that the fuzzer can start them up from the command line. Each generated input will be either a file to be read by the program under evaluation, or a string to be given as an argument to the program.

2 Project Requirements

2.1 Modes of Operation

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 a consider a program to have crashed if it exited due to signal 11 (SEGV).

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 pseudo-random 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.

2.2 Search Mode

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.

For example,

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

will cause the fuzzer to generate a file filled with HTML data with some name, e.g., test.html, using the file `sample.html` then replace the file argument to `firefox` to have it execute `firefox test.html`. A fuzzer could change random bytes in `sample.html` to generate the test case file `test.html`. Alternatively, it may completely ignore the original contents of `sample.html` and generate a completely new test case, or it may modify `sample.html` in a smarter way to stress the HTML parser in `firefox`.

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.

2.3 Replay Mode

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 sample.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 `sample.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.

2.4 Additional Features

When running in search mode, the fuzzer must also have a mechanism for terminating the program being tested if it runs longer than a specified time. Namely, if the option `--timeout-fail X` is given where `X` a positive real number, then the fuzzer should terminate the program if it runs longer than `X` seconds. The seed used to generate the corresponding input should be printed, indicating the program is presumed to have entered an infinite loop. If the option `--timeout-ok X` is given, the fuzzer should terminate the program if it runs longer than `X` seconds, but should not consider this a failure and thus not print the seed. This latter option is useful for testing interactive programs such a `GIMP` which do not automatically exit after processing their input. At most one of these two options should be specified.

3 Fuzzing Strategies

3.1 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.

3.2 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.

You should target your protocol-aware fuzzer to an image conversion application that converts a JPEG/JFIF image to a ppm image file. Optionally, with careful design plan you can implement your fuzzer to target a wide range of applications – specially image, video, audio playing applications (such as media players, image drawing programs, image converts, music players, and so on). For grading, we will require you to find bugs in a program `jpegconv` provided as a binary by us. For earning the extra credit, you can fuzz any real-world program of your choice and report bugs in them!!

You can design your fuzzing tool with any strategy you like. However, you must have the same command-line interface as the mutation-based fuzzer, with extra command line arguments which should be documented in your submission.

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 online informal descriptions which concisely provide you with information

sufficient to understand the necessary parts of the format. For instance, we found the following link concise and useful :

`http://www.obrador.com/essentialjpeg/headerinfo.htm`.

Wikipedia (`http://en.wikipedia.org/wiki/JPEG`) should be a good starting point as well. We do not expect you to understand anything about JPEG internals like compression (such as huffman coding) and/or quantization techniques – you can simply focus on the format of the file and stress the JPEG file parsing logic fully to uncover enough bugs for full credit.

There are numerous other resources on the web to help you understand the JPEG file format, and you are welcome to read these. In particular, we recommend open source tools `http://hachoir.org/` that dissect a given JPEG file into separate fields and helps you to visually inspect the file format – you might find these useful (and sufficient) to understand the file format before you design your protocol-aware fuzzer.

Hint: Be sure to read the file `START.txt` available in the provided project code directory. You might find useful information there to get started with the protocol-aware fuzzer.

4 Deliverables

There will be two dileverables – a project report and the code itself.

4.1 Project Report

Please submit a project report in PDF format at most 5 pages (8.5×11 inches page) long, in single-column format using any 10 point font or larger. It must contain at least the following information as separate sections:

- **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.
- **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?

4.2 Code

You must submit all your code as an archive with the name `project2-code.tar.gz`. When untarred, the archive should create only one high level directory `project2`. There must be at least two sub-directories `mutation` and `protocol` containing code for the mutation-based fuzzer and protocol-aware fuzzer respectively. There should be a file in each of the two sub-directories called `replay` containing final commands (along with the random seeds and arguments) to trigger the bugs found. Finally, There should be a README describing how to compile the code in the two sub-directories.

Here is a sample directory structure after untarring :

```
project2
project2/mutation/
project2/protocol/
project2/README
project2/mutation/replay.sh
project2/protocol/replay.sh
```

5 Grading

We will test your code on one of the ilinux{1,2,3}.eecs.berkeley.edu machines. So, please ensure that your code compiles cleanly and works on one of those machines.

Please do not wait until the very end to test your code on these machines, as these machines are likely to be overloaded close to the submission deadline.

We will grade you on your design, implementation, empirical evaluation and the results you obtained from your tools. Here is the grading scheme.

- (*2 points*) Design of your mutation-based fuzzer.
- (*2 points*) Implementation of the mutation-based fuzzer. It must compile and should be easy to run for the grader.
- (*4 points*) Design of your protocol-aware fuzzer.
- (*6 points*) Implementation of your protocol-aware fuzzer. It must compile and should be easy to run for the grader.
- (*8 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.
- (*3 points*) Empirical results presented in the project report, and conclusions drawn.
- **Extra Credit** (*5 points*). You can get up to 5 maximum points as extra credit for finding bugs in other real programs. You get *2 points* for reporting the first real-world program in which you've found a bug, and *1 point* for each subsequent program that you report bugs in.