Project Description

CS161 Computer Security, Spring 2008

1 Overview

The goal of this project is to implement a "fuzzer", or fuzz tester. Fuzz testing is one way of attempting to discover security vulnerabilities in code implementing a network protocol or otherwise processing 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 overlook.

In teams of four, you will implement a fuzzer capable of finding bugs in real life programs. Each team may implement their fuzzer in the programming language(s) of their choice.[1] As part of the project, we have introduced a number of vulnerabilities into a program 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,

[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. If your team has a strong desire to write your project in a language for which a compiler or interpreter is not available on the instructional Linux machines, let us know and we will try to accommodate you.

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 a 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 or if it runs longer than some specified timeout (indicating a possible infinite loop). Note that it is not considered a bug for the program under evaluation to print an error message or immediately exit. 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).

Various strategies for generating sequences of inputs are possible, from exhausting all possibilities of some short length to producing strings of bytes completely at random. In most cases though, at least some aspects of each input 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.

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.

To specify that the fuzzer should construct an input file for the test program, a special argument `--fuzz-file SPEC` will be given among the `CMDARG`'s. In that case, the fuzzer should generate a file with contents determined by the `SPEC` argument, and replace `--fuzz-file SPEC` with the name of the file when running the command. For example,

```
fuzzer firefox --fuzz-file html-file
```

will cause the fuzzer to generate a file filled with html data with some name, e.g., test.html, then execute `firefox test.html`. Similarly, if the special argument `--fuzz-string SPEC` is given, a string will be generated according to `SPEC` and passed in place of that argument to the program.

The interpretation of the `SPEC` arguments is left up to your design. Since you will want your fuzzer to test different types of programs and possibly test the same program using different strategies for generating inputs, you will need to specify parameters for generating inputs when invoking the fuzzer. The `SPEC` argument provides a means of doing this. For example, you might design your fuzzer to be able to generate inputs suitable for testing firefox, mplayer (a video player), and grep, in which case you would use the `SPEC` argument to distinguish these cases:

```
fuzzer firefox --fuzz-file html-file
fuzzer mplayer --fuzz-file avi-file
fuzzer grep --fuzz-string regex --fuzz-file text-file
```

If after some number of trials in the third example grep crashes with a segmentation fault, the fuzzer should then print the corresponding seed as string (e.g., `24cdb33d7e3d99cb`) 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 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 24cdb33d7e3d99cb grep --fuzz-string regex --fuzz-file text-file
```

might cause the fuzzer to print

```
grep 'dje(nw?2*k)||h{2}' testfile
```

then exit. In replay mode any generated input files should not be removed, so after this example executes the file `testfile` 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 should also recognize several additional options which may be provided before the name of the program to be tested. If the option `--trials N` is given where `N` is a positive integer, then the fuzzer should run `N` tests of the program before exiting. If this option is omitted, the fuzzer should keep testing until it is terminated.

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 firefox which do not automatically exit after processing their input. At most one of these two options should be specified.

2.5 Strategies for Constructing Inputs

The simplest sort of fuzzer would be one that generates completely random strings of bytes as inputs. While it would have the advantage of being easy to write and applicable to any program, it would be very limited in the types of bugs it could discover. For example, many programs read the first four 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. In terms of code coverage, a test with a completely random string of bytes is extremely unlikely to cover more than a tiny portion of the program's code.

Thus, it is necessary to generate inputs with at least some degree of validity to 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 data format read by the program.

One general approach to producing inputs sufficiently valid to exercise significant portions of the code while allowing enough randomness to trigger exceptional cases would be to describe the data format using a probabilistic context-free grammar (PCFG). A PCFG is context-free grammar in which every production is augmented by a probability. Such a grammar can be used to produce random expressions of the context-free grammar. To do so, one may begin with the start symbol, randomly select a production based on their probabilities, then recursively expand the symbols of that production in the same way until a string consisting only of terminals is obtained. By writing a fuzzer which could generate testing inputs from PCFG's, one could easily test any programs with a file format which can be described by a context-free grammar.

Each team may choose to use this overall approach to generating test inputs or some other approach, but careful consideration should be given to

this aspect of the design. The generality and sophistication of the fuzzer design as described by each team in their final design document (and implemented in their code) will form a significant part of the grade. Likely, the most effective fuzzers will use a combination of techniques.

2.6 Using the Fuzzer

At some point we will be releasing a faulty program on which you will test your fuzzer. The program will be a version of the `pstotext(1)` command that we have specially modified to introduce a number of bugs which can cause it to crash. Using your fuzzer, you will attempt to discover inputs which trigger these bugs. At the end of the semester, each team will submit a list of the bug-triggering inputs they discovered along with their source code and other deliverables.

The number of distinct bugs found will form a significant portion of the project grade. To help the teams gauge their progress, when we release the testing program we will state how many distinct bugs it includes. We will also ensure that the testing program provides a means to distinguish which bug (by number) caused a particular crash, so that teams can tell whether they are discovering new bugs or simply triggering the same bugs over and over. The submission for each discovered bug will take the form of a seed and command line which will cause your fuzzer to produce an input which triggers that bug using its replay mode, e.g.,

```
fuzzer --replay 0b9e40b5c5895e58 pstotext --fuzz-file mangledpostscript2
```

The `pstotext(1)` program reads a Postscript document and attempts to extract ASCII text as output, so a large part of the project will consist of designing the fuzzer to produce documents which are likely to thoroughly test a Postscript interpreter. Postscript is a simple, stack-based programming language interpreted by printers and document viewers to render text and graphics on a page. A simple Postscript document is shown below.

```
%!PS
/Courier findfont
20 scalefont
setfont
72 500 moveto
(Hello world!) show
showpage
```

Students are encouraged to view some Postscript documents in a text editor and read a little about the Postscript language online (Wikipedia is a good start) before considering the design of their fuzzer. The bugs in the testing program will be of varying subtlety. Some will be easy to find with a rudimentary fuzzer, while others will not likely be uncovered except by a fuzzer which does a good job of producing documents which stress the interpreter.

Additionally, teams are encouraged to try their fuzzer on a variety of real programs of their choice. If any bugs are found in programs other than the testing program, the team should include the name of the program tested, seed, and command line used in their final submission, in the same way the results for the testing program are submitted. Any such discoveries will be rewarded with significant extra credit.

3 Deadlines, Deliverables, and Grading

There will be three submissions over the course of the project, two milestone submissions and a final submission. The first milestone will be due April 2 and will consist of a one page design document, timetable for completing the project, and breakdown of work among the members of your group. The design document should discuss the general strategy your fuzzer will use to generate test inputs, the strategy for generating Postscript in particular, and any specific techniques you intend to use. After the first milestone we will hold individual meetings with each team to discuss their design.

The second milestone will be due April 23 and will consist of updates to these and an initial submission of working code. The fuzzer submitted for the second milestone should implement the entire interface described in this document and be able to run programs with some sort of random input (a single random byte is fine), detect when they exit due to signal 11, print the seed in that case, and reconstruct the input from the seed in replay mode. At this point, the fuzzer does yet need not to implement any sophisticated method generating of inputs.

After the second milestone, each team will be able to devote their remaining time on the project to generating Postscript output and discovering bugs in the test program and other programs. The final project submission will be due May 12 and will consist of the final version of the design document and a tarball containing the fuzzer source code and testing data for the discovered bugs. More precise details on the files to be be present in the tarball and

how the testing data is to be formatted will be released before then.

Deliverables:

Milestone 1 Draft design document, timetable for completing the project, breakdown of labor within the team

Milestone 2 Updated design document, timetable, and labor breakdown, and working fuzzer

Final submission Final design document, final fuzzer code, list of seeds for bugs found in testing program, and (optionally) list of seeds for bugs found in other programs

The projects will be graded based on the following criteria:

- 5%: Milestone 1
- 5%: Milestone 2
- 50%: Number of bugs found in our faulty program
- 40%: Evaluation of the general design of your fuzzer (based especially on the design document)
- Extra credit: Bugs found in other programs other than the provided faulty program