CS558: LAB 1: PROGRAM FUZZING

Due 11:59 February 22, via websubmit:

http://cs-websubmit.bu.edu/main.py?courseid=cs558

Note: While the fuzzer is running, the computer it is running on will be unusable, so please do not plan to use your computer during this time. Plan to complete this homework on your own computer and not on a shared lab computer.

Objective

In this assignment you will write a basic fuzzer that can find security vulnerabilities in software. Along the way, you will learn:

- Different types of fuzzer: smart vs. dumb, mutative vs. generative. Advantages of each type.
- The importance of templates selection
- How to revise a fuzzer and template set to increase the probability of crashes

Background

Smart vs. dumb

A smart fuzzer takes longer to write and requires more knowledge of the target format but is also more likely to find bugs once running. A smart fuzzer is effectively required for formats that have internal checksums or validation and for XML. A dumb fuzzer is very unlikely to find bugs in these formats.

Generative vs. mutative

A mutative fuzzer starts with one or more templates and corrupts them. A generative fuzzer creates a fuzzed file or packet from scratch. Mutative fuzzers are generally easier to write, but the effectiveness of the fuzzer is limited by the templates: if a particular code path is not executed while processing any of the templates, any bugs in that code path will be missed. A generative fuzzer is harder to write and often takes longer to find vulnerabilities but is more thorough (more likely to find all possible vulnerabilities), and there is no need to find templates.

Generally, the easiest option is a dumb mutative fuzzer but a smart mutative fuzzer will find more bugs faster. Generative fuzzers are typically only used for relatively simple formats like TCP.

Step 1: Write the fuzzer

Your goal for this assignment is to produce a dumb mutative fuzzer. In this step you will write a program that generates test input files.

- The fuzzer should be executed from the command line as follows:
  `java Fuzzer <num_iterations>`

Here, `num_iterations` is a command line argument that determines how many iterations of the mail loop the fuzzer will run.

- The fuzzer will read the template file named "template" located in the project's directory into memory.
- The fuzzer will iterate for `num_iterations` times. In every iteration you should change each character of the file to a random character with 20% probability (the template is the disk will not be changed in every iteration)
- Every 1000 iterations, the program should add 10 random characters to the end of the template.
- After all iterations, The fuzzer will save the mutated template to a file named "test_file" in the project's directory.
- After writing the output file the fuzzer terminates.
- Make sure the fuzzer is deterministic. That is, if you execute the fuzzer twice with the same parameter, you get the same output file in both executions. If you want, you can use pseudo-random numbers in you fuzzer as follow:
  - `import java.util.Random;`
  - `Random rand = new Random(20071969);`
  - `int pick = rand.nextInt(10);`

  By initializing the object `rand` with a seed, the function `rand.nextInt` will produce the same pseudo-random numbers in every execution.
- Your code should be well documented and understandable. Make sure that each file is understandable in of itself, even without a "readme" file.

Step 2: Test your fuzzer on the provided program

We have provided a sample program that has vulnerabilities and a sample input for that program. The program takes a single argument of a file to be read. We are providing you with C code. As a first step, you should compile this code on your machine to get an executable which you will fuzz. (In reality you of course will not usually have access to the source code. We do it here only help you get an executable that runs on your machine.)

In the next step you will write a test module that uses the fuzzer from step 1 to crash the sample program. The test module will run as follows:

- It will run in a loop until a crashing input file is found
  - In every iteration the test module will run the fuzzer once, increasing by 1 the value of the parameter `num_iterations.`
  - The test module will open the test_file generated using the program.
  - The test module will wait until the program has finished (either wait a few seconds or block on termination of the program).
  - Test if the program crashed (by checking the exit code) and if so output the value of the parameter `num_iterations` that produced the crash.

`Note: Your fuzzer will be tested on programs similar to but different the test program given. It will be expected to work on these programs as well.`

Step 3: Submit

Use the online submission tool to submit the following:

- The code for the fuzzer from Step 1
- A writeup of how you mutate inputs and why
- A text file named "parameter" that contains a single number, value of the parameter `num_iterations` that produced the crash.
- Do not submit the test module from Step 2. Only use the test module to find a good value to write in the file "parameter".

Extra Credit: Run the fuzzer on a real program

Required for graduate students enrolled in the course.

Note: Recall the ethics and regulations of legitimate hacking: Do not interfere with "live" software and data. Do not cause harm or interfere with the experience of other users. Do not try to access information that you are not allowed to access, even if you can.

To this point you have created a fuzzer that can be run on a small test program. For extra credit, test externally available software to test for vulnerabilities. The program must accept external input, such as a file or protocol. It is easiest if you can easily start the program and cause it to process the fuzzed files so that the program runs and loads that file without any prompts or other required interaction. Pick one of the formats handled by that application to target.

Some factors to consider when selecting a program and file format to fuzz:

- More complicated programs and file types are more likely to have vulnerabilities simply due to the volume of code they contain.
- Older programs are more likely to have vulnerabilities.
- Programs made by large companies and/or with large user bases tend to have fewer vulnerabilities, although this is not universal.
- Some file formats are harder to fuzz than others. Formats that contain many internal checksums or that are compressed are hard to fuzz because most fuzz runs will cause a checksum to fail, hiding the error. **The newer Microsoft Office file formats (.docx, .xlsx, .pptx) are hard to fuzz for this reason.**
- Programs that often deal with malformed input, such as web browsers, tend to have fewer vulnerabilities that can be found with this type of fuzzing.
- It may be hard to find templates for an obscure program or format.

Bad applications to target:

- Notepad, MSPaint, and similar programs with very simple input and simple functionality: the input is so simple that you are unlikely to find bugs
- Anything by Microsoft or Adobe: these companies have extensive fuzzing programs, making it unlikely you'll find bugs
- Programs that run within web browsers, like ActiveX controls or NPAPI plugins: the infrastructure to successfully fuzz and detect crashes in plugins is more complicated than fuzzing a program directly.

Good applications to target:

- Anything by a small developer: they are unlikely to have the resources to fuzz extensively
- Older products that are still available, especially anything made before about 2002.
- Open source projects with a small number of active contributors

If the fuzzer doesn't find crashes, you can try to improve it. Some things to try include:

- Increase the aggressiveness, so that more changes are made per iteration.
- Increase the intelligence of the fuzzer, so that it spends more time fuzzing areas of the format/protocol that are likely to have bugs.
- Try different templates or even a different file format.

Don't get discouraged: sometimes you just won't find bugs. In the larger sense, this is good: it means the software is pretty secure!

Extra credit will be given depending on the effort and extra sophistication employed to obtain the results.

Submission policy:

Every submitted assignment MUST include the following information:

- List of collaborators
- List of references used (online material, course nodes, textbooks, wikipedia, etc.)
- Number of late days used on this assignment
- Total number of late days used thus far in the entire semester

If any of this information is missing, at least 20\% of the points for the assignment will automatically be deducted from your assignment. See also discussion on plagiarism below.}