Python3 Implementation for Fuzzing Test

Brenden Morton

# Design

In this section, I will describe the implementation of the fuzzer. This fuzzer was written in python3. To run the script use the following command:

python3 fuzzer.py [X] > [output text file] 2>&1

The command above will run the fuzzer script X number of times. Then, it will output the STDOUT and the STDERR into the specified output text file. The reason for this is to process the output after the program runs. Initially, the plan was to either use sockets, threads, or subprocesses to stream the STDOUT and STDERR from the terminal into another script to process the error outputs but this was taking up too much time.

The code is quite clear and readable, so it is preferred to read the code, but here is a description of the code in plain text.

Now, I will discuss how the script works. First, I will discuss the main loop.

A fuzzing test consists of trying random modifications to the inputs to the program or software. For this project, we are inputting mutated JPEG files into an executable. Thus, we must be modifying random bytes in the JPEG input file to trigger the bugs set in place. To do this, the script will read in the number of iterations to run to determine how many mutated files to create. Then, a new folder is created to store the results of this execution of the fuzzing script. Lastly, a loop iterates over the number of mutated files from the command line arguments and runs the mutate() function.

For each iteration of the main loop, a single mutated JPEG file will be created from the given original JPEG file. To do this, the bytes of the original JPEG file are read into a variable called bytes\_d. In python3, bytes\_d will be initialized as a byte string with the bytes from the original cross.jpg JPEG file. However, this creates some problems with modifying bytes because a byte string object is immutable. To circumvent this, the bytes\_d byte string object is converted into a list. The list object is mutable and will allow for the modification of specified bytes based on the index in the list. This is ideal since we can then randomly modify positions in the list. This leads to the next part of the mutate() function: generating the number of modifications to make. Initially, I began by only making one-byte modifications for each mutated JPEG file. After playing around with the script a bit, I decided to modify a random number of bytes in the file. Testing the results, I noticed there were minor differences in the number of bugs triggered in the single-byte modification compared to the multi-byte modifications. I decided to keep the multi-byte modifications to increase the randomness of the mutation. A variable num\_modifications is set to a random integer value ranging from one to the total number of bytes in the file. Next, a loop will iterate the number of modifications to make and modify a random position in JPEG with a randomly generated byte. To generate a random position to modify in the list of bytes, a variable rand\_pos is set to a random int between 0 and the index of the last byte or len(bytes\_d) – 1. Now, that a randomly generated position is created, we must generate a random byte to replace the current byte at that position. To do this, I created a function called get\_random\_byte\_replacement() which returns a random byte converted into an integer. This is then able to be placed into the list of bytes which upon writing the mutated JPEG into a new file will be converted properly. Lastly, the byte list is converted back to a bytes string object and is written to a new file.

For the last part of the program, the results of the script are evaluated. When the fuzzer script is run, the contents of the STDOUT and STDERR are sent to a specified output text file. Here, the evaluate\_results() function is able to process the results. The main purpose of this function is to count the number of different bugs triggered and outputting the frequency of bugs triggered. This function also displays the bugs that are not triggered.

# Empirical Results

In this section, I will discuss some results and findings from running the script.

To properly test this script, I began just by doing small number of mutation files, such as 50 and 100 and noticed that bugs were being triggered with some more than others. After some trial and error, I came across that around 1000 runs found most of the bugs. At this point, I knew that just increasing the number of mutated JPEG files to create would results in more bugs being likely to be triggered, I went for 50,000 mutated files. For the results below, I decided to compare the findings between single-byte modifications and multi-byte modifications.

Here are the results from the first run of 50,000 mutated files:

Command:

python3 fuzzer.py 50000 > output.txt 2>&1

Multi-byte Modifications

Random mutli-byte modifications 50,000 sample:

|  |  |
| --- | --- |
| Bug | Frequency |
| 1 | 38 |
| 2 | 21 |
| 3 | 8 |
| 4 | 1273 |
| 5 | 136 |
| 6 | 0 |
| 7 | 39 |
| 8 | 99 |

Total bugs triggered = 1614

Here is a pie-chart of the same data.

Single-byte Modifications

Below is a single-byte modification to compare the types of modifications,

Random single-byte modifications 50,000 sample

|  |  |
| --- | --- |
| Bug | Frequency |
| 1 | 38 |
| 2 | 21 |
| 3 | 8 |
| 4 | 1273 |
| 5 | 136 |
| 6 | 0 |
| 7 | 39 |
| 8 | 99 |

Total bugs triggered = 4363

# Analysis

Comparing the data above in the Empirical Results section, the single-byte modification had significantly more bugs triggered. The multi-byte modification had 1614 total bugs triggered whereas the single-byte modification method had 4363 total bugs triggered. Below is a table with a side-by-side comparison between the two samples:

|  |  |  |
| --- | --- | --- |
| Bug | Multi-byte Freq | Single-byte Freq |
| 1 | 38 | 6 |
| 2 | 21 | 129 |
| 3 | 8 | 40 |
| 4 | 1273 | 1654 |
| 5 | 136 | 283 |
| 6 | 0 | 0 |
| 7 | 39 | 423 |
| 8 | 99 | 1828 |

As seen above, the data from both samples appears to be very similar in that the bugs triggered in its own group follows a similar distribution. However, the single-byte modification produced many more bugs as compared to the multi-byte modification. I cannot precisely determine why this is the case but it appears that this might be a result of the ./jpgbmp executable’s flow in the program. Not knowing how exactly this program works contributes to some of the ambiguity of the reason for such a difference in the results. One reason can be that multi-byte modifications produce symmetric byte modifications that can cancel out the effects of a byte modification. For example, the same position might be modified to the same byte as before. Although this is very unlikely other modifications might even cancel the effects of modifications made to a specific byte. To truly understand this, we would need to analyze the hex dump or byte dump of the jpeg file as well as get more insight on the jpgbmp executable program.