CAP6135: Programming Project 2: Fuzzing
(Spring 2013)

(Assigned Feb. 18th, due on Canvas by Mar. 17th midnight
Have 15% bonus points if you finish and submit it by Mar. 3rd midnight and have no revision after that)

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/

I have put a .zip file on Canvas in this project submission page. You will need to download it, extract its included three files for completing this project. Once you put those three files in your Eustis account and if the 'jpegconv' is not executable, you can use command:

```
$chmod u+x  jpegconv
```

To change it to become executable.

Delivery:
Submit a .zip file through UCF Canvas. The zip file should contain:

1. Source code file of your fuzzer. Your programs must be programmed using a language that can be run on Eustis, such as C, Java, or Python (explain clearly how I can compile and run your code on Eustis machine in your report!) You can include multiple fuzzer codes where each of them is used to discover one or several specific bugs.
2. The input files (modified jpg images based on the sample.jpg) that could trigger each of the bugs you can find. For each bug, just provide one input image file. Name the input file as "test-x.jpg" for Bug # x. In this way, I can easily check whether you really found Bug #x by running it by myself!
3. 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.

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:
You will implement a fuzzer that is a combination of mutation-based fuzzing and protocol-aware fuzzing, capable of finding bugs in a provided real-world program 'jpegconv'. Each student may implement his/her fuzzer in the programming language(s) of their choice (such as C,

C++, Java, Python), as long as the fuzzer can be executed on the Eustis machine (eustis.eecs.ucf.edu).

As explained in the fuzz Test example in Lecture 11 (02/18/2013 class), we have deliberately introduced a number of vulnerabilities into the 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 student will submit a list of the bugs they found, and the number of distinct bugs found in the test program will form part of each student'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 (what knowledge of the jpeg file format you have used).

The fuzzer will repeatedly invoke the program being tested (jpegconv) with a series of inputs (modified image files based on sample.jpg), each time checking to see if the jpegconv crashes. **If the target application being tested prints an error message or immediately exits, it is not considered as having a bug.** 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.

Mutation-based Fuzzing
You should first write a simple mutation-based fuzzer, taking the valid input file sample.jpg via the --fuzz-file argument, and use it as a base case for generating new test image files. To generate a new test file, 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.

Protocol-aware Fuzzing
Use the protocol knowledge about file format for JPEG (JFIF) image file format (JPEG File Interchange Format) to some extent. For your convenience, we have printed the format structure for the sample.jpg file as "sample.format". You need not understand the complete specification for JPEG file format. The header formats appearing in sample.format 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

When you uncompress the tar file on webcourse, besides the bugged `jpegconv` executable code, you will also find a normal example picture sample.JPG, its format file sample.format.

How to save generated image files that trigger some bugs?

In most cases, your generated image file (based on the sample.jpg) cannot trigger any bug. Since you need to generate tens of thousands mutated image file as input to the jpegconv for fuzzing, there is no disk space for you to save all those test input images in Eustis machine. So in your fuzzer, only when you determine that the jpegconv generates signal 11 (segmentation fault) then you save the generated image file.

In addition, you need to match the jpegconv stderr output message "BUG X TRIGGERED" to the corresponding saved image file in order to know which bug is triggered by which image file. One simple way is when executing 'jpegconv' command, you can use redirect (introduced in Lecture 11) to save the "BUG X TRIGGERED" messages to a text file. There are other more intelligent ways to do it, and more intelligent way to name those saved image files.

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 fuzzing test. Specifically, you should point out how you mutate the sample.jpg, including which byte or bytes you modify, what values you use, and which bug is triggered by modifying which format field (if you use jpeg format), etc.

Empirical results

Show one or several graphs or tables describing the results from running the fuzzer. Show how many inputs you have tried in fuzzing, the number of bugs you have found (which bugs you have found). For each bug you found, how many different image input files have triggered the bug.

When you find an input modified image file 'test.jpg' that could trigger Bug #1, name the file as 'test-1.jpg', save it (for each bug, you only need to save one image input for final submission; of course, during fuzzing test, you may have saved multiple image files that generate the same bug). Then when you run:

```
$ ./jpegconv -ppm -outfile foo.ppm test-1.jpg
```

The jpegconv should crash and print out:

```
BUG 1 TRIGGERED
```

Show the screenshot image(s) of each bug you have triggered using the above command.

Grading

(70 points)

There are at least 10 bugs in the sample program. You need to find any 8 of the 10 to receive full credit. *A bug is confirmed to be found only if you have a screenshot image showing the crash and the printout text by jpegconv showing triggering of the bug*.

(10 points)

Empirical results presented in the project report. Report contains clear description.

(10 points)

Submitted compressed file contains image files that could trigger each of found bugs (e.g., if you found 6 bugs, you should have 6 test-x.jpg files in your submission).

(10 points)

Project report contains the screenshot images showing the "BUG X TRIGGERED" for each found bug.

HELP ON PROGRAMMING

```
1. You should first check that ./jpegconv works on your testing
environment.

      $ ./jpegconv -ppm -outfile foo.ppm sample.jpg
      $ ls foo.ppm
      foo.ppm
      $
2. Your fuzzer should generate a mutated/modified image file such as
'YOUR_TEST_CASE.jpg'. Then your fuzzer executes the command-line of
   jpegconv, like this:
      "./jpegconv -ppm -outfile foo.ppm YOUR_TEST_CASE.jpg".

3. You can inspect the sample.jpg file to see the format
   structure. For your convinience, we have printed the format
   structure for that file as "sample.format".

   Each line in the sample.format file signifies a field in the
   JPEG file. For instance, consider the line 2-4 of
   sample.format shown here.

     - 0) start_image: Start of image (SOI) (2 bytes)
       0) header= 0xff: Header (1 byte)
       1) type= 0xd8: Type (1 byte)

    It shows the input format for the first 2 bytes as a tree.
    It states that at offset "0" in the file sample.jpg, an SOI
   field was found. It has two single-byte subfields "header"
   and "type". The values of these are 0xff and 0xd8
   respectively, and so on.

4. You can use any hex editor to check the sample.jpg file. Under Eustis
(Linux) machine, you can use
"$hexdump sample.jpg > hex.txt"

To generate ASCII readable text file 'hex.txt' to show the hexadecimal value
of each byte. The first value in each line is the starting address/position
of the first byte in that line (in hexadecimal value).

5. Information :: You may restrict the scope of your tool to only be
concerned with the headers used in sample.jpg. Specifically, the bugs
introduced in jpegconv, require only code paths resulting from parsing the
image data and following header types:

      * Start of Image (SOI) marker -- two bytes (FFD8)
      * JFIF marker (FFE0)
      * Define Quantization table marker (FFDB)
      * Define Huffman table marker (FFC4)
      * Start of frame marker (FFC0)
      * Start of Scan marker (FFDA)
      * End of Image (EOI) marker (FFD9)
```

6. HINTS AND POSSIBLE DIRECTIONS :
   One strategy is to incrementally add file format knowledge to your fuzzer. For instance, here is one step-by-step way:

- Level 0 : Purely mutation based fuzzer, no information about the JPEG format. You can pick random number of bytes in the jpeg file to change their values to random values.

- Level 1 : Fuzzing image format structure of SOI, EOI header/marker types

- Level 2 : Fuzzing the generic format for most headers/markers. Note that all headers have very similar structure --(0xff , Type Byte (1 byte), Length/size (2bytes), ... Data of the chunk ...)

- Level 3 : Fuzzing the semantics of internal fields of
  APP0 header/markers (header type 0xff E0).

- Level 4 : Fuzzing the semantics of DQT (type 0xff db)
  headers/markers, DHT (type 0xff C4) headers/markers,
  and SOF (header type 0xff C0) headers/markers, and
  SOS (header type 0xff da) headers/markers.

You can then increase the sophistication of your fuzzer to be able to operate at higher levels incrementally.

7. How to call jpegconv repeatedly in your fuzzer without causing the fuzzer to stop execution?

Answer: the fuzzer should not stop if jpegconv crashes, so we can create a child process and the child process run jpegconv. One example is in your fuzzer code:

```
  //Create child process
  pid = fork();
  if(pid == 0){   //this is child process, call execve
      execve(TARGET, args, env);
  }else{
      //this is parent process, wait for child, print seed
      while(waitpid(-1, &status, WNOHANG) > 0){
      if(WTERMSIG(status) == 11){
         printf("BUG TRIGGERED\n");
      }
  }
```

Another way that does not use multi-process is the following code (used in fuzzTest.c in Lecture 11):

```
 sprintf(buffer, "./jpegconv -ppm -outfile foo.ppm %s",
modified_sample_jpeg_file);
  ret = system(buffer);
  wait(&status);
  WEXITSTATUS(ret));
  if (WEXITSTATUS(ret) == 139)
  {
    printf("a bug is triggered!\n");
```

```
    fflush(stdout);
  }
```

The WEXITSTATUS() returns a value that is the signal number that caused the termination plus 128. That's where exit status "139" comes from.