Assignment #2

Analysis:

In order to proceed with this project I decided to use the Python programming language due to ease of use and no demand for high efficiency in order to accomplish this task. Python is a scripting language that is best known for executing all the functionality for the user without numerous lines of code having to be written. This is also ideal in situations where the time frame needed for development is short and not a large number of users will be utilizing the finished product.

Implementation:

I've added a "shebang" in the first line of my code which allows for normal execution of the program within the Eustis machine by calling "./fuzzer.py", permissions may still need to be modified first using "chmod u+x" before execution. I had to "pip install" the "wand" module within my code so that will also need to be included before execution.

Design:

The code consists of three functions: "bug_trigger", "binary_data_modify", and the "main" function. The "bug_trigger" function contains the following:

```
def bug_trigger(binary_data_cross, x):
    from wand.image import Image
    with Image(filename='/home/net/jo798048/homework2/tests/cross.jpg') as img:
        img.save(filename='/home/net/jo798048/homework2/tests/cross_mutated_test_1.jpg')
        binary_data = open('/home/net/jo798048/homework2/tests/cross_mutated_test_1.jpg', 'wb')
        binary_data.write(binary_data_cross)
        binary_data.close()
        cmd_str="./jpeg2bmp cross_mutated_test_1.jpg cross.bmp"
        import subprocess
        child = subprocess.run(cmd_str, shell=True)
        if child.returncode == 139:
            cmd_str="cp cross_mutated_test_1.jpg /home/net/jo798048/homework2/tests/cross_mutated_" + str(x) + ".jpg"
            child = subprocess.run(cmd_str, shell=True)
            print('TEST NUMBER: ' + str(x))
            return 1
        else:
            cmd_str="rm cross_mutated_test_1.jpg"
            child = subprocess.run(cmd_str, shell=True)
            return 0
```

Using the “wand” module, we open the original “cross.jpg” image for editing. We create a copy of “cross.jpg” called “cross_mutated_test_1.jpg” within the same directory of the Eustis machine. The file copy is then prepped for modification of its binary data by specifying “‘wb’” otherwise known in Python as “write bytes”. We passed in an argument from the “binary_data_modify” helper function for our “binary_data_cross parameter”, basically, this variable modifies the data within the image copy and creates our mutated jpg file with potentially error-prone tendencies. Finally, we close the file after we finish our edits. The objective for the next few lines of code is to execute shell commands within the Eustis machine. The “cmd_str” variable is initialized to execute the “jpeg2bmp” program with our mutated test image we created. We need to import “subprocess” for executing linux shell commands. Our “child” variable will run the command using the bash shell and we check for successful triggering of a bug within the “jpeg2bmp” program. If we successfully trigger a bug, then we create a copy of the successful test to the current directory and name it “cross_mutated_(The number of the successful test run).jpg”. We also print out the test number and route the output to a text file for data collection and analysis. We return 1 in order to increase our counter which takes place

within the “main” function outside the “bug_trigger” function. If the bug isn’t triggered, then we simply delete the test file within the same directory and don’t count it as a test run by returning 0.

The “binary_data_modify” helper function contains the following:

```
def binary_data_modify(binary_data_modification, binary_data_cross, counter):
    import random
    specific_spot = random.randint(0, 807)
    new_data = binary_data_cross[:specific_spot] + binary_data_modification
    add_counter = bug_trigger(new_data, counter)
    counter += add_counter
    return counter
```

We import random to prepare the “specific_spot” variable for a random integer value between 0 and 807. Since “cross.jpg” contains 808 bytes worth of data, this gives us the full size and allows for maximum potential of data manipulation to the “cross.jpg” image file. We prep the “cross.jpg” data to be manipulated at a particular offset within its makeup, the spot within the data is modified to contain the information within the “binary_data_modification” variable. The “add_counter” variable will retrieve a 1 or a 0 after executing all of the code within the “bug_trigger” function. We take this value and add it into our “counter” variable before returning it to our “main” function.

Finally, our “main” function contains the following code:

```
def main():
    binary_data_max = b'\xff'
    binary_data_zero = b'\x00'
    binary_data_mid = b'\x05'
    binary_data_cross = b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00`\x00`\x00\x00\xff
    x = 0
    add_counter = 1
    while x != 1650:
        add_counter = binary_data_modify(binary_data_max, binary_data_cross, add_counter)
        add_counter = binary_data_modify(binary_data_mid, binary_data_cross, add_counter)
        add_counter = binary_data_modify(binary_data_zero, binary_data_cross, add_counter)
        x += 1
main()
```

Our “binary_data_max” variable is initialized to contain the max hex byte value of “ff”, the “binary_data_zero” variable contains the minimum hex byte value of “00”, and our “binary_data_mid” variable with contain hex value “05”. The two variables “binary_data_max” and “binary_data_zero” trigger a total of 9 bugs, but the “binary_data_mid” variable is needed to trigger bug #2 by indicating an incorrect number of Huffman tables within the byte code. The “binary_data_cross” variable contains all of the byte data from the “cross.jpg” file. The “x” variable will be used in our “while” loop to commence the trials. The “add_counter” variable is set to 1 for denoting our first successful bug trigger. We conclude the main function by looping for 1,650 trials with 3 different tests, giving us a total of 4,950 different results.

Empirical Results:

With our run of “fuzzer.py”, we achieved a total number of 4,950 tests where only 974 tests ended up being successful, giving us 3,976 bug triggers. The following table contains the statistical occurrence of each bug after executing our code:

Bug Number	Number of occurrences	Chance of occurrence
Bug #1	5	.01%
Bug #2	6	.01%
Bug #3	18	.04%
Bug #4	11	.02%
Bug #6	134	3%
Bug #7	13	.03%
Bug #8	3718	93%
Bug #9	8	.02%
Bug #10	4	.01%
Bug #11	59	1%