Exploring Fuzzing

Overview

This lab will explore fuzzing. We will start with writing our own fuzzer, and continue to use afl-fuzz utility, which is a state-of-the-art open-source fuzzer. Before you start, watch the lecture and make sure you understand the concept of fuzzers.

Keep detailed notes below (place your comments in between the provided horizontal lines); You will be working in your Ubuntu 16.4 VM, so start that now and open a terminal window.

Part 1: Writing our own fuzzer

Earlier this semester, you learned that one way to discover a vulnerability is to determine whether you could provide an input that would cause the program to crash. (That's not the last step in the process, but it is certainly one of the important early ones.) The process of providing inputs was handled manually. We will now learn how to automate it.

Copy your ans_check5 binary to a new folder named "binaries", and enter the new directory. Consider the following script.

```
import subprocess, os

# The following variables control the command line
program = "./ans_check6 "
arg_pattern = 'a'
pattern_max = 10

for i in range(pattern_max):
    print "Trying input with length", i
    cs = program + " " + arg_pattern*i
    print "Command: %s" % cs
    print "******************"
    proc = subprocess.Popen([cs], shell=True,
                            stdin=subprocess.PIPE,
                            stdout=subprocess.PIPE)
    print proc.communicate()[0]
    print "******************"
    print "Return value: %i, %s" % (proc.returncode,
```

```
                                        os.strerror(proc.returncode))
    print
```

Store the text above in a file called fiz_beta.py. You can invoke the script as follows.

```
python fiz_beta.py > output.txt
```

Can you see how this script might be used to automate program runs and to discover crash inputs? Change the script so that it causes ans_check5 to crash at least once. Include the lines that you changed below.

I changed line6. Changed the 'pattern_max = 10' variable to 50.
It allows the program to try more cases.

Create a new version of fiz_beta.py, called fiz.py, that accepts `program`, `arg_pattern`, and `pattern_max` parameters as command line parameters. Include your new source code below.

```
import subprocess, os
import sys

# The following variables control the command line
if len(sys.argv) != 4:
    print 'Put 3 arguements'
    print 'fiz.py [program] [arg_pattern] [pattern_max]'
    exit

print '{},{},{},{}'.format(sys.argv[0],sys.argv[1],sys.argv[2],sys.argv[3])
program = sys.argv[1]
arg_pattern = sys.argv[2]
pattern_max = int(sys.argv[3])

for i in range(pattern_max):
    print "Trying input with length", i
    cs = program + " " + arg_pattern*i
    print "Command: %s" % cs
    print "******************"
    proc = subprocess.Popen([cs], shell=True,
```

```
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE)
print proc.communicate()[0]
print "******************"
print "Return value: %i, %s" % (proc.returncode,
                        os.strerror(proc.returncode))
print
```

Part 2: Setting up afl-fuzzer

We now move to use afl-fuzz utility, which is a state of the art open-source fuzzer. See the lecture slides for an overview of afl-fuzz.

The first step is to install and setup the fuzzer that we are going to use, afl-fuzz. First we download and extract the source:

```
$ cd /tmp
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
$ # Or use the following if the previous gets blocked
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-2.52b.tgz
$ tar xzf afl-latest.tgz
$ cd afl-2.52b # assuming current version is 2.52b
```

Now we compile and install it like any other program:

```
$ sudo make install
```

This installs it to a directory already in our PATH environment variable, so we can call it without having to explicitly provide a path.

To complete the setup we need to tell the kernel not to send core dumps to an external utility to avoid crashes being interpreted as hangs due to the added delay.

```
$ echo core | sudo tee /proc/sys/kernel/core_pattern
```

Now our fuzzer is ready! All we're missing is a program to fuzz.

Part 3: Fuzzing a simple program

Part 3.1: Set up

Create a new directory “fuzzing” for this in-class assignment. We will be fuzzing different applications in this directory. Change to your fuzzing directory now.

Inside of your fuzzing base directory, create a new directory titled “stack_overflow” for this particular application. In this directory, create the following C file 'stack_overflow_stdin.c':

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {

  int value = 5;
  char buffer_one[8], buffer_two[8];
  char s[64];

  strcpy(buffer_one, "one");
  strcpy(buffer_two, "two");

  printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
  printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
  printf("[BEFORE] value is at %p and is %d (0x%08x)\n\n", &value, value, value);

  gets(s);
  printf("[STRCPY] copying %d bytes into buffer_two\n\n", strlen(s));
  strcpy(buffer_two, s);

  printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
  printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
  printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);

}
```

Now we need to compile our target program and setup for fuzzing.

We compile our target program with afl-fuzz's gcc wrapper, appropriately called afl-gcc:

```
$ afl-gcc -g -fno-stack-protector -o so stack_overflow_stdin.c
```

Try running it! (And be mindful of the size of the buffer being used.)

Note: afl-fuzz feeds input either through stdin (what we're using here) or by file, not by command line arguments.

Now we need to make input and output directories for afl-fuzz to read from and write to. (Note, if you use a Google Drive folder mapped into your VM, you might get errors; afl-fuzz will need to

create symlinks in these directories. To get around this, simply create your directories elsewhere in a normal directory such as /tmp.)

```
$ mkdir testcases findings
```

Then we add our starting test case. It can be whatever you want as long it doesn't crash the program, but generally the smaller the better.

```
$ echo hi > testcases/in.txt
```

Part 3.2: Fuzzing

Now all we need to do is run afl-fuzz and tell it where to find the starting test cases (-i), where to output its findings (-o) and the target program.

```
$ afl-fuzz -i testcases -o findings ./so
```

Since our target program is so small, and has such a glaring vulnerability, it shouldn't take long for afl-fuzz to find something ('uniq crashes' is the metric to watch here). Once you're satisfied with the amount of fuzzing done, stop the program with ctrl-c.

We now need to explore afl's findings. Conveniently, afl stores all inputs that caused a crash in findings/crashes/. The files are named in the following way: <crash id #>,<exit signal>,<path #>,<operation type>,<repetition #>.

Try running our program with some of the produced inputs. (Note that if FILE is a crash file, you can provide it as input to our program with `./so < FILE`.) There should be at least two unique crashes you've found with two different causes. If not, try fuzzing for a bit longer. Identify and discuss what you do or do not understand about the two crashes found by the fuzzer:

- First one Is like below.

[03/30/22]seed@VM:Byeongchan$ ./so
< ./findings/crashes/id\:000000\,sig\:06\,src\:000000\,op\:havoc\,rep\:16
[BEFORE] buffer_two is at 0xbf930018 and contains 'two'
[BEFORE] buffer_one is at 0xbf930010 and contains 'one'
[BEFORE] value is at 0xbf93000c and is 5 (0x00000005)

[STRCPY] copying 26 bytes into buffer_two

*** buffer overflow detected ***: ./so terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(+0x67377)[0xb74e9377]
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x68)[0xb7579548]

```
/lib/i386-linux-gnu/libc.so.6(+0xf5738)[0xb7577738]
/lib/i386-linux-gnu/libc.so.6(+0xf4d2f)[0xb7576d2f]
./so[0x80486ad]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7)[0xb749a637]
./so[0x8048715]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1060889    /home/seed/fuzzing/stack_overflow/so
08049000-0804a000 r--p 00000000 08:01 1060889    /home/seed/fuzzing/stack_overflow/so
```

- Second one is like below.

```
[03/30/22]seed@VM:Byeongchan$ ./so
< ./findings/crashes/id\:000001\,sig\:11\,src\:000000\,op\:havoc\,rep\:128
[BEFORE] buffer_two is at 0xbfcb8ea8 and contains 'two'
[BEFORE] buffer_one is at 0xbfcb8ea0 and contains 'one'
[BEFORE] value is at 0xbfcb8e9c and is 5 (0x00000005)

[STRCPY] copying 0 bytes into buffer_two

[AFTER] buffer_two is at 0xbfcb8ea8 and contains ''
[AFTER] buffer_one is at 0xbfcb8ea0 and contains 'one'
[AFTER] value is at 0xbfcb8e9c and is 5 (0x00000005)
Segmentation fault
[03/30/22]seed@VM:Byeongchan$
```

- Identify and discuss what you do or do not understand about the two crashes found by the fuzzer

I can understand first one because it tried to copy more than the ‘buffer_two’ variable can hold. But I can’t understand second one because the input was less than 8 bytes. Moreover, from the output of the execution, there’s nothing to copy. It’s so weird cause second input file did have some contents on the file.

Part 3.3: Set up and fuzz a second program

Return to your fuzzing base directory, and create a new directory called ‘fs’ for this particular application. In this directory, create the following file 'fs.c':

```
#include  <stdio.h>
```

```
int main()
{
        char buf[100];
        char s[100];
        int x = 1;
        fgets(s, 100, stdin);
        snprintf(buf, sizeof buf, s);
        printf("Buffer size is: (%d) \nData input: %s \n", strlen(buf), buf );
        printf("X equals: %d/ in hex: %x\nMemory address for x: (%p) \n", x, x, &x);
        return 0;
}
```

Compile it with afl-gcc:

```
$ afl-gcc -g -w -o fs fs.c # -w ignores compiler warnings
```

Briefly, what does this program do? Run it and try some inputs. Are there any obvious vulnerabilities to you? (no is an acceptable answer)

[03/30/22]seed@VM:Byeongchan$ ./fs < input
Buffer size is: (10)
Data input: ddxx4llaa

X equals: 1/ in hex: 1
Memory address for x: (0xbfe99f80)
[03/30/22]seed@VM:Byeongchan$

[03/30/22]seed@VM:Byeongchan$ ./fs < input
Buffer size is: (99)
Data input:
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccc
X equals: 1/ in hex: 1
Memory address for x: (0xbff1b390)

- The program takes input stream and save it to ‘s’ variable. And then copy ‘s’ variable to a ‘buf’ variable. Lastly, the program prints out ‘buf’ size, its content and the ‘x’ variable’s information.
- I tried less than 100 characters and more than 100 characters and both worked fine.
- I don’t think it has some vulnerabilities because it uses ‘snprintf’ to limit the size of a copy.

Again we setup our directories and initial test case for fuzzing:

```
$ mkdir testcases findings
$ echo hello > testcases/in.txt
```

Run afl in the same way as before, exiting once again when you're satisfied with the results:

```
$ afl-fuzz -i testcases -o findings ./fs
```

Like before, look at the input(s) that result in crashes and try running them. (There will probably be only one input that causes a crash.) Do you notice anything about the input that might be different than other inputs we have used? Can you find the program's vulnerability? Hint: it's not a type of vulnerability we've covered in depth in this class. Can you think of a way to use it for something malicious?

Do you notice anything about the input that might be different than other inputs we have used?

- ⇨ I noticed that the input causing crash is like '%n['. I googled about '%n' and I found out that '%n' has an ability to write into a designated address about how many bytes have written so far. If I can write something into a variable in the program, that means I can write more than the variable can hold, and finally I can reach out the return address of the next function, in this case it will be a 'fgets function'.

Can you find the program's vulnerability?

- ⇨ When getting some inputs from the users, we need to filter the input. Since there is no filtering about user input, there could be a vulnerability.

Can you think of a way to use it for something malicious?

- ⇨ I'm not sure about this, but If I can make input payload like 'A*n + %n + &x', I can edit the return address of the next function call.

COMPLETE