# **Notes from Exploring Fuzzing**

## **Instructions**

Make a copy of this document, rename it to “exploring-fuzzing-notes” and move it to your CSE 523 Google Docs collection. If at any point in this exercise you feel stuck, raise your hand and get some guidance. When you reach each GATE below, switch over to the Tracking Progress document and update your position. Try to be efficient with your time.

## **Overview**

Today we will explore fuzzing using the utility afl-fuzz, which is a state of the art open source fuzzer. Keep detailed notes below (place your comments in between the provided horizontal lines); you will be referring to these in the future to do your work.

We will be working in your CSE 523 Ubuntu VM, so start that now and open a terminal window.

**GATE 1**

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:

$ wget <http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz>

$ # Or use the following if the previous gets blocked

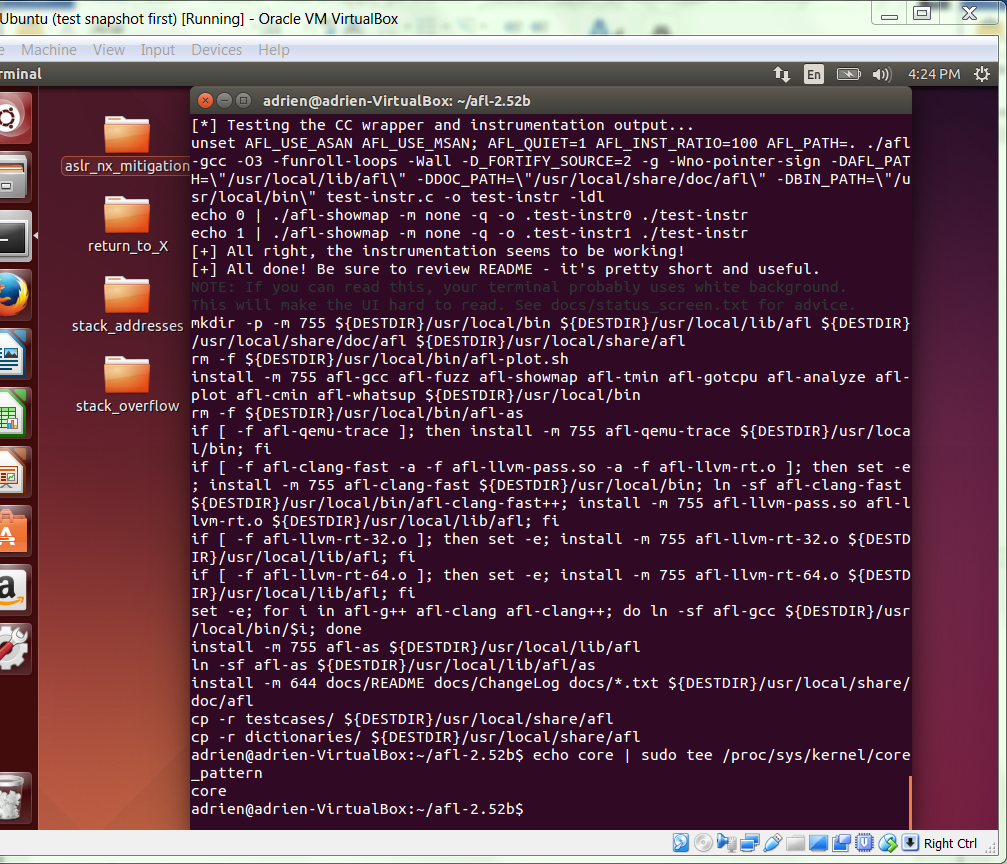
$ wget [http://lcamtuf.coredump.cx/afl/releases/afl-2.52b.tgz](http://lcamtuf.coredump.cx/afl/releases/afl-2.35b.tgz)

$ 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.

**GATE 2**

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);  
  
}

You might recognize this file as stack\_overflow.c from past exercises, with a few minor changes (highlighted in green). Briefly, what is the difference between this version and the original version of stack\_overflow.c? Hint: look at the filename (original can be found in exploring-stack-overflow-notes)

This stack overflow file has a standard input stream and related functional calls.

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 -o so stack\_overflow\_stdin.c

Try running it! Note how the behavior is slightly different from our original. Does this line up with your explanation for the differences between the versions?

Note: the reason we're using a modified version of stack\_overflow.c instead of the original is that 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.

$ 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

**GATE 3**

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

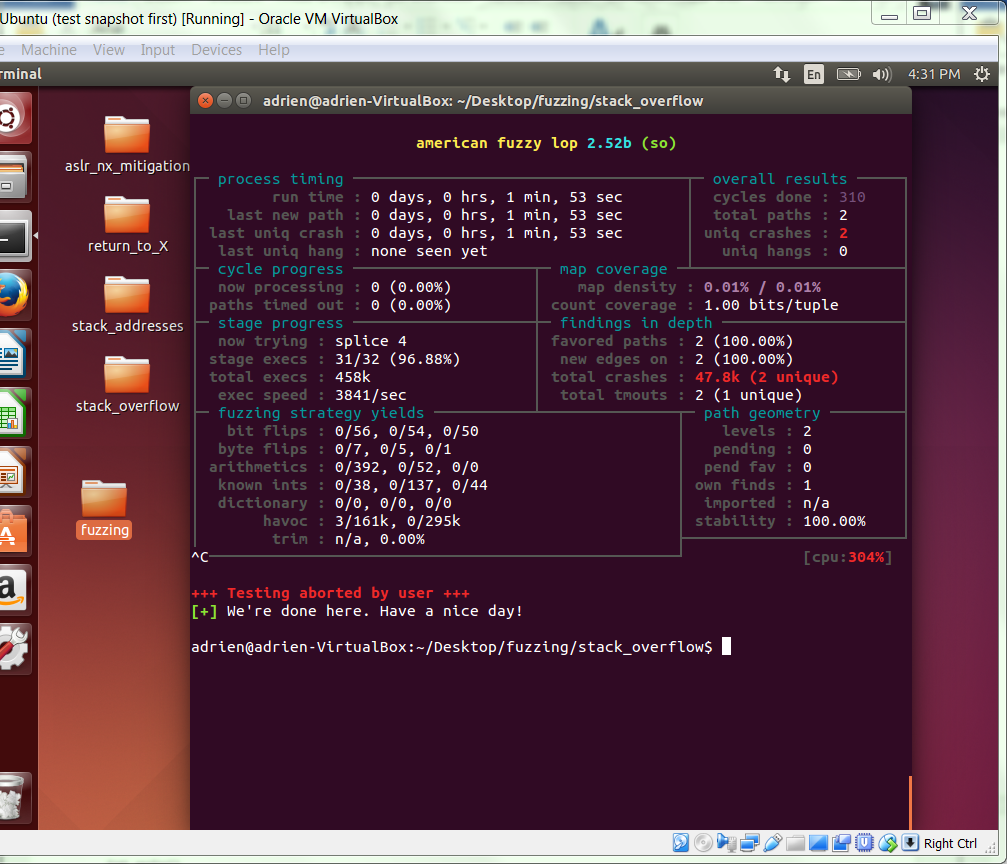
Since our target program is so small, and has such a glaring vulnerability, it shouldn't take long for afl-fuzz to find something ('uniq crashes' is the metric to watch here). Once you're satisfied with the amount of fuzzing done, stop the program with ctrl-c.

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. 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 the two (or more) crashes found by the fuzzer:

A crash happened when an input is oversized.

A crash happened when the variable is oversized (buffer\_two)



**GATE 4**

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;  
}

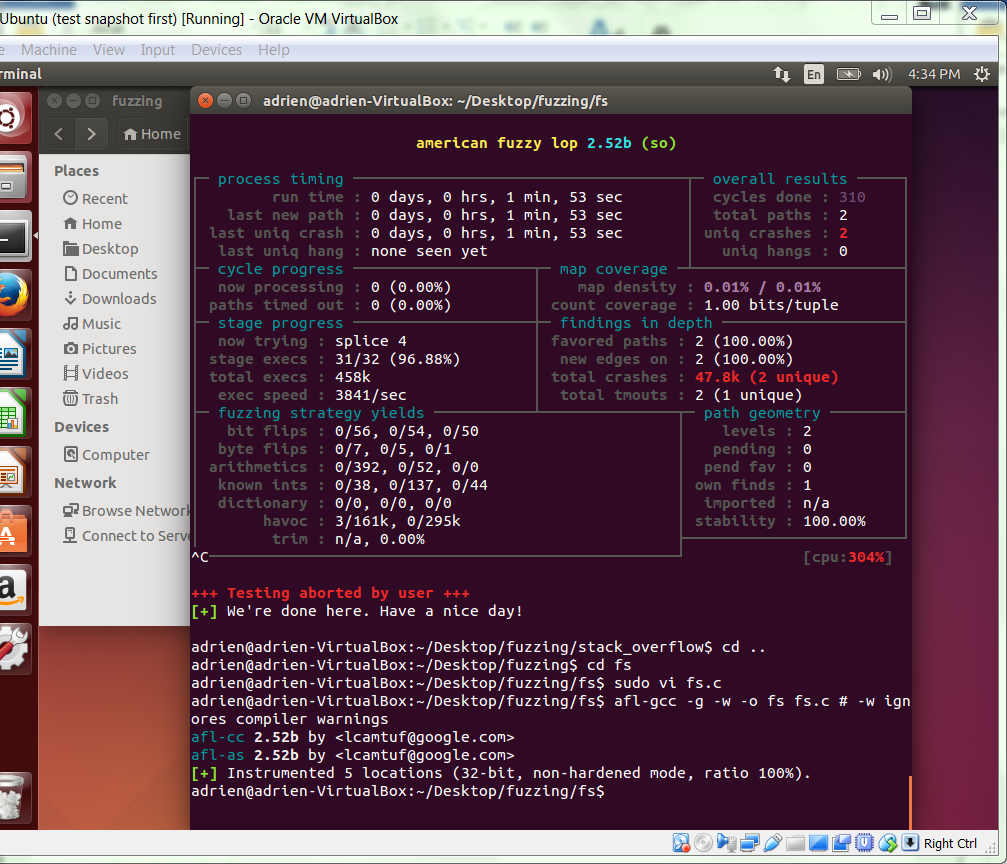
Compile it with afl-gcc:

$ 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)

The program takes the input of maximum of 100 characters and print out the memory address. Oversized portion will be discarded.

There is no obvious vulnerabilities.



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

$ mkdir testcases findings

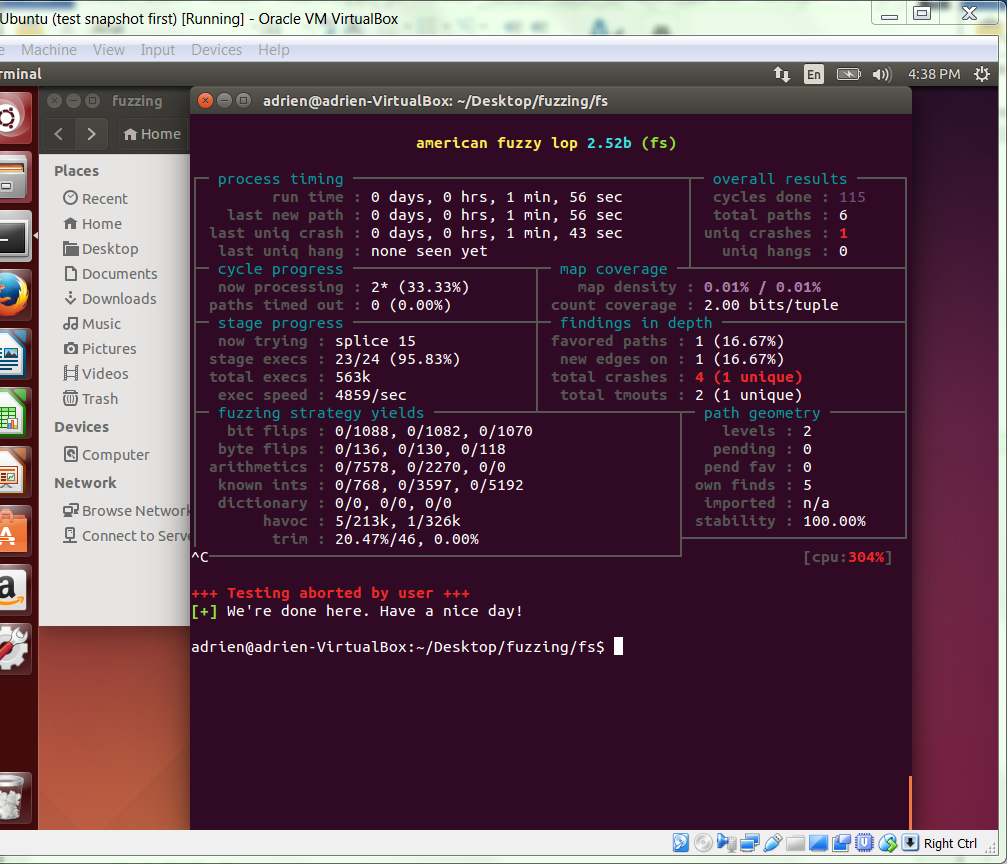
$ echo hello > testcases/in.txt

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?

The input string that caused the program crash is %n. and this is a format vulnerability. We can hence by using the malicious input string to access things such as private variables on stack or memory and/or modify them.



**GATE 5**

Our final fuzzing target is a much more substantial one. It's a real world open source library called libbfd, it's part of the GNU binutils set of programs, which includes readelf, objdump, strings and others. It examines and manipulates binary files.

Go to your fuzzing base directory and create a new folder called ‘bfd’. In this folder, we first need to download and extract the source.

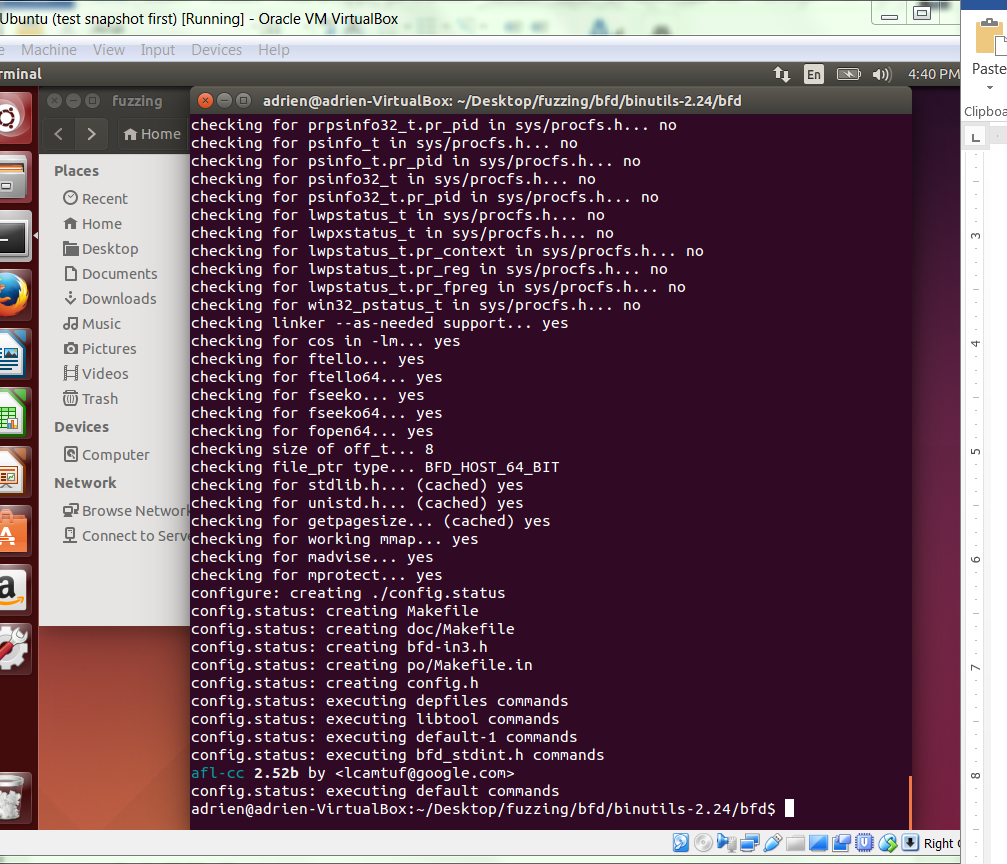
$ wget <http://ftp.gnu.org/gnu/binutils/binutils-2.24.tar.bz2> # Must be v2.24!

$ tar xjf binutils-2.24.tar.bz2

Now we need to build the library:

$ cd binutils-2.24/bfd/

$ CC=afl-gcc ./configure --disable-shared



Note the --disabled-shared. This forces the library to be statically linked instead of dynamically linked, i.e. the library is compiled directly into any binary that uses it. Usually most programs that call a library don't actually include the library in the program binary itself. The library lives in a precompiled standalone '.so' file that is shared by all programs that use it, and is loaded into memory when the calling program is run (i.e. it is dynamically linked). Why do you think static linking is important for fuzzing purposes?

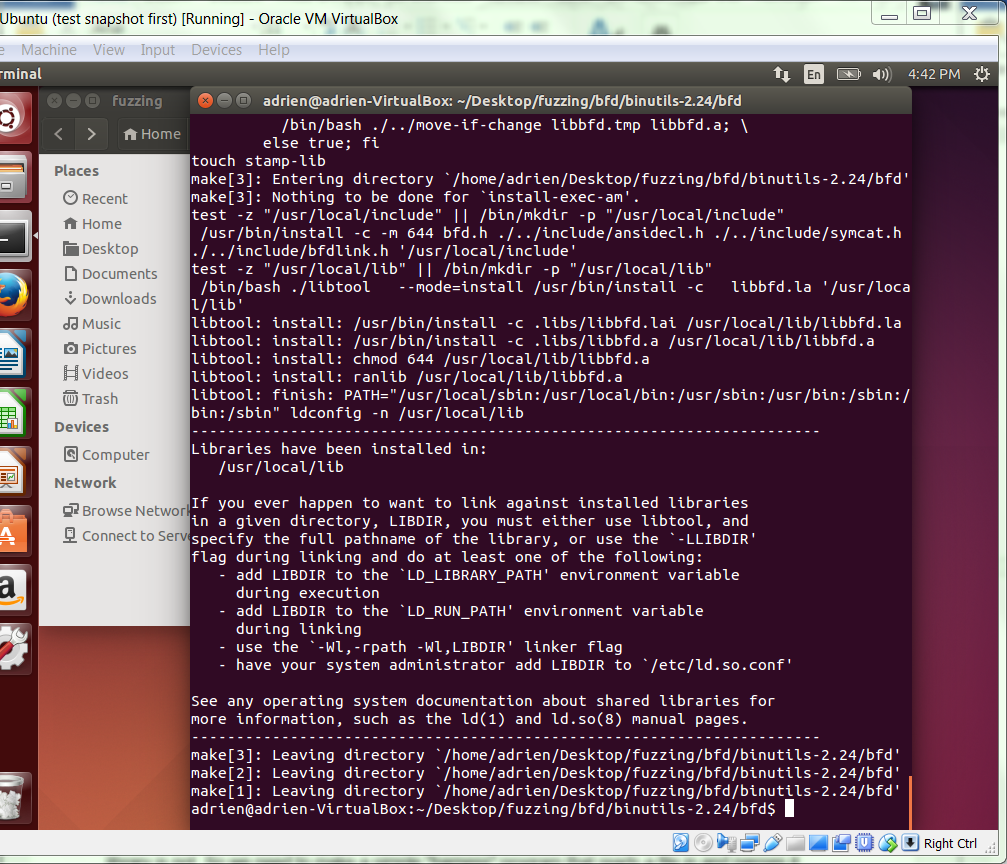
Because in static linking, the size of the executables become greater, compared to the dynamic linking. With static linking, it is enough to include the library referenced by target executable.

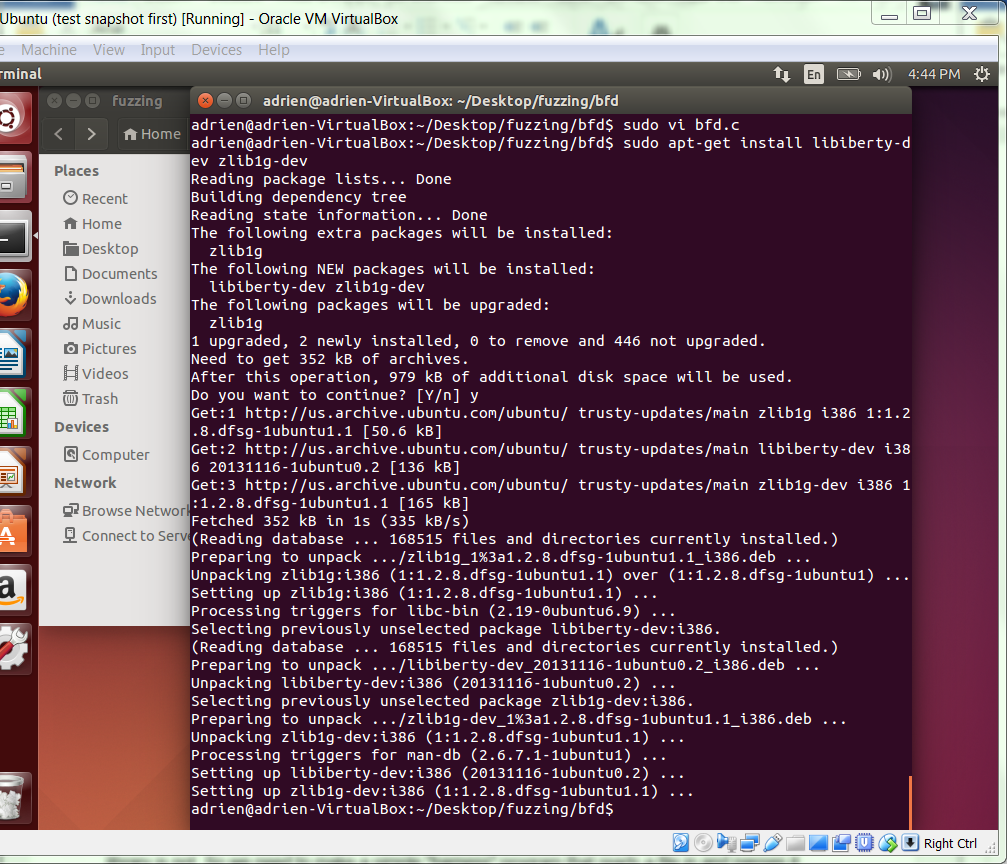
Now we install the library:

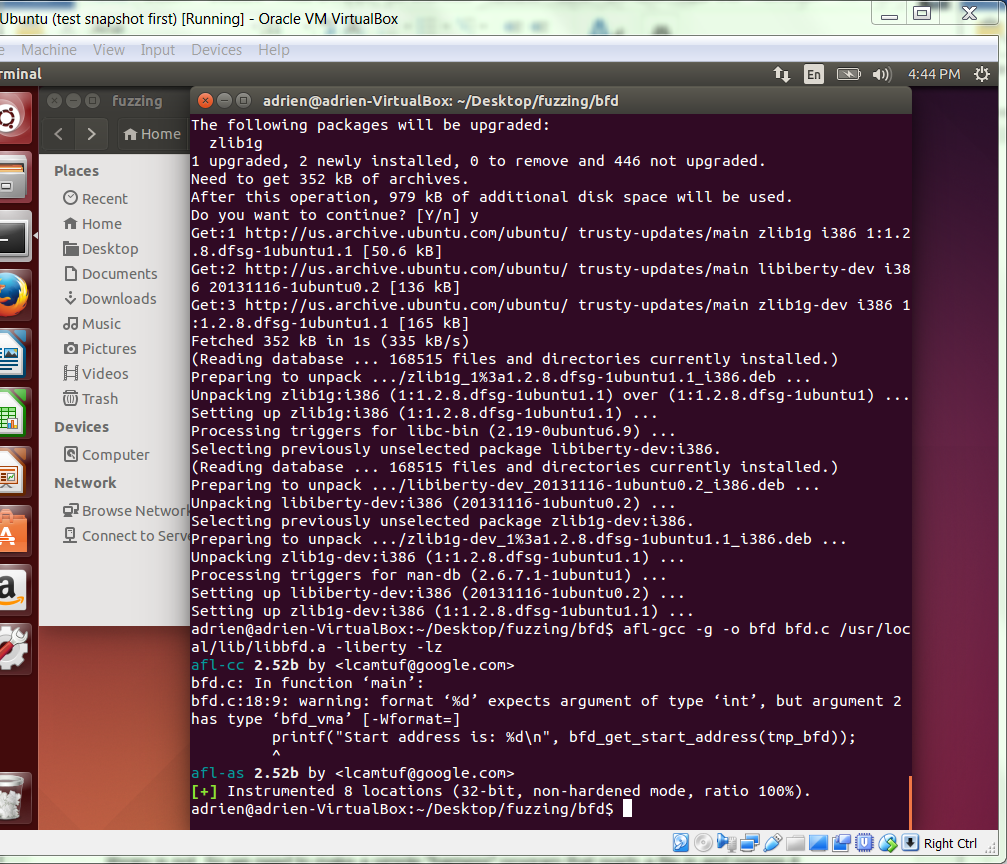
$ sudo make install

$ cd ../../

This puts the compiled (and instrumented) library in /usr/local/lib/.







**GATE 6**

When trying to fuzz a library, we encounter a problem that we didn't have to worry about when fuzzing a standalone program. You can run a program and see if it crashes, but how do you just run a library? Our fuzzing target has to a be a runnable program, and in its current state our library is not. So we need to make a simple "harness" program that reads a file in and passes it to our target library.

Create the following program 'bfd.c':

#define PACKAGE "libgrive" // Hack to allow library compilation :-/  
#include <stdio.h>  
#include <bfd.h>  
  
int main(int argc, char \*argv[]) {  
 if (argc != 2) return 1;  
 bfd \*tmp\_bfd = NULL;  
 tmp\_bfd = bfd\_openr(argv[1], NULL);  
 if (tmp\_bfd == NULL) {  
 return 2;  
 }  
 if (!bfd\_check\_format (tmp\_bfd, bfd\_object)) {  
 if (bfd\_get\_error() != bfd\_error\_file\_ambiguously\_recognized) {  
 printf("Incompatible format\n");  
 return 3;  
 }  
 }  
 printf("Start address is: %d\n", bfd\_get\_start\_address(tmp\_bfd));  
 return 0;  
}

We need to install a couple dependencies for libbfd before we can compile, specifically libiberty and zlib:

$ sudo apt-get install libiberty-dev zlib1g-dev

$ afl-gcc -g -o bfd bfd.c /usr/local/lib/libbfd.a -liberty -lz

Try running it on a few different file types, including C source, a compiled C program and some other types of your choosing. Briefly, what file types does our program work with and what does the program do? Hint: you can use the 'file' command to get more detailed information on a file's type.

The program works with compiled C program and what it does is to print out the start address of the file. C source and python source are actually incompatible.

**GATE 7**

Let's again set up our input and output directories:

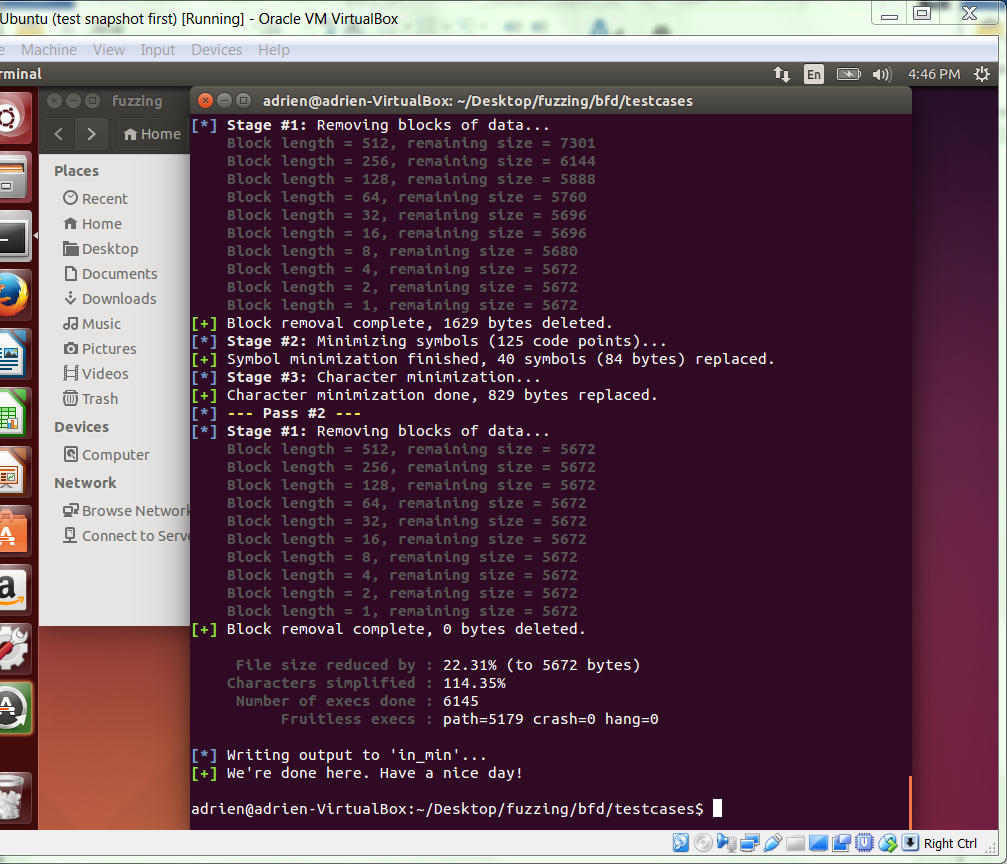
$ mkdir testcases findings

Our starting test case is slightly more complicated this time. Since libbfd works with binary files we should start with a binary file. We can easily make one by compiling an extremely barebones C program:

$ cd testcases

$ echo "int main() {return 0;}" > in.c

$ gcc -o in in.c



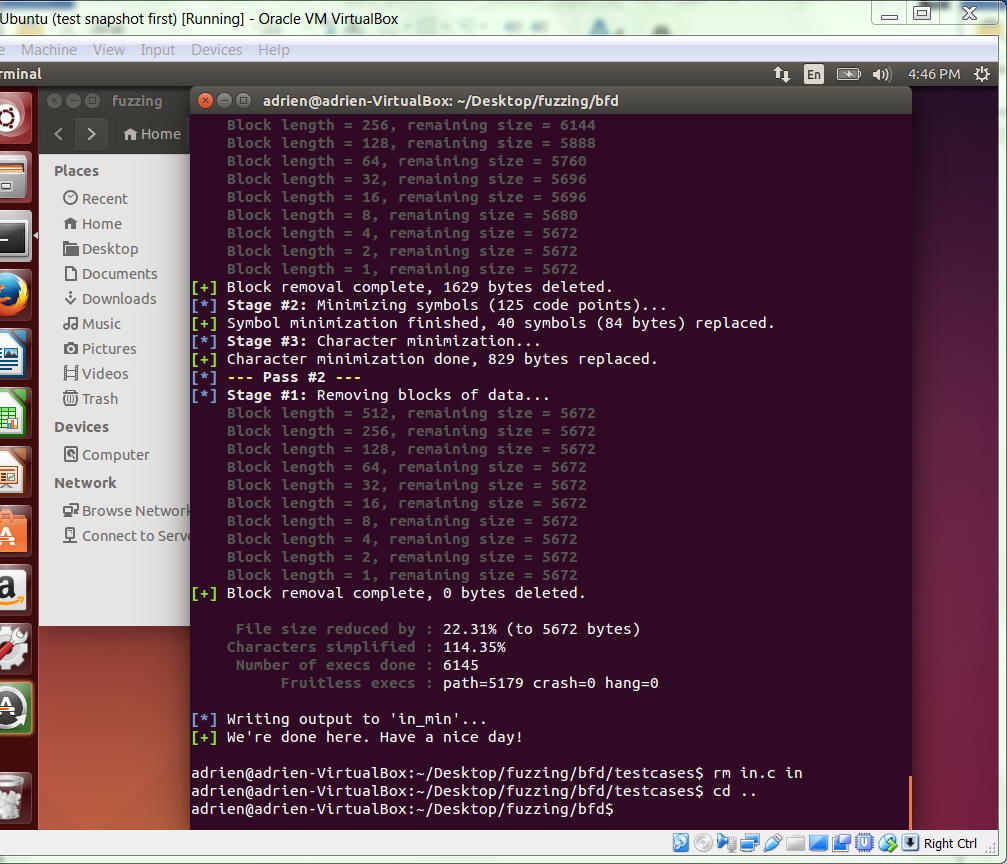
Now as mentioned before, smaller starting test cases are generally better as they usually run faster, reduce complexity, etc. However, we can't really make our starting test case much smaller that easily, so we're going to use afl's built in test case minimizing tool, afl-tmin. afl-tmin takes a test case and our instrumented target program and slowly trims the test case, ensuring each time that the execution path through the target program is unchanged until it can't trim any more. Despite our already barebones starting test case, afl-tmin can trim a good amount away.

The syntax is similar to afl-fuzz itself, provide the input test case (-i), what you want the minimized version to be called (-o) and the target program. Note here since our target program reads a file from a filename passed in as an argument instead of reading from stdin, we put '@@' to tell afl where to put the filename as an argument:

$ afl-tmin -i in -o in\_min -- ../bfd @@

$ rm in.c in # remove source and un-minimized binary file

$ cd ..



**GATE 8**

Now we're ready to fuzz! The rest is left for your last homework of the semester.