**CS6008 CRYPTOGRAPHY AND NETWORK SECURITY ASSIGNMENT – 4**

**(MODULE-3 EXTERNAL LEARNING)**

**Using libfuzzer and AFL to fuzz our C/C++ code Implementation**

**11.12.2021**

**Done by**

**SABHARI P**

**2018103582**

***Fuzzing:***

The basic idea behind fuzzing is to try massive numbers of random inputs to code in order to trigger a vulnerability. You create a testbench for the code of interest, pair it with a fuzzing engine that generates random data, and launch it on some server somewhere. Hours, days, or weeks later - if your testbench is solid - it comes back with a set of inputs that cause the code to crash. This process may be accelerated by:

• Using a sanitizer: Compiler-supported sanitizers instrument binaries with extra code to check for illegal conditions - such as out-of-bounds memory accesses - that may not cause an immediate crash. This makes the code under test more likely to fail, and thus reduces the fuzzer running time.

• Using coverage-driven fuzzing: fuzzers can monitor program states reached under different inputs, and guide the inputs in a way that tends to produce new (and potentially erroneous) ones.

***Fuzzing Engines***

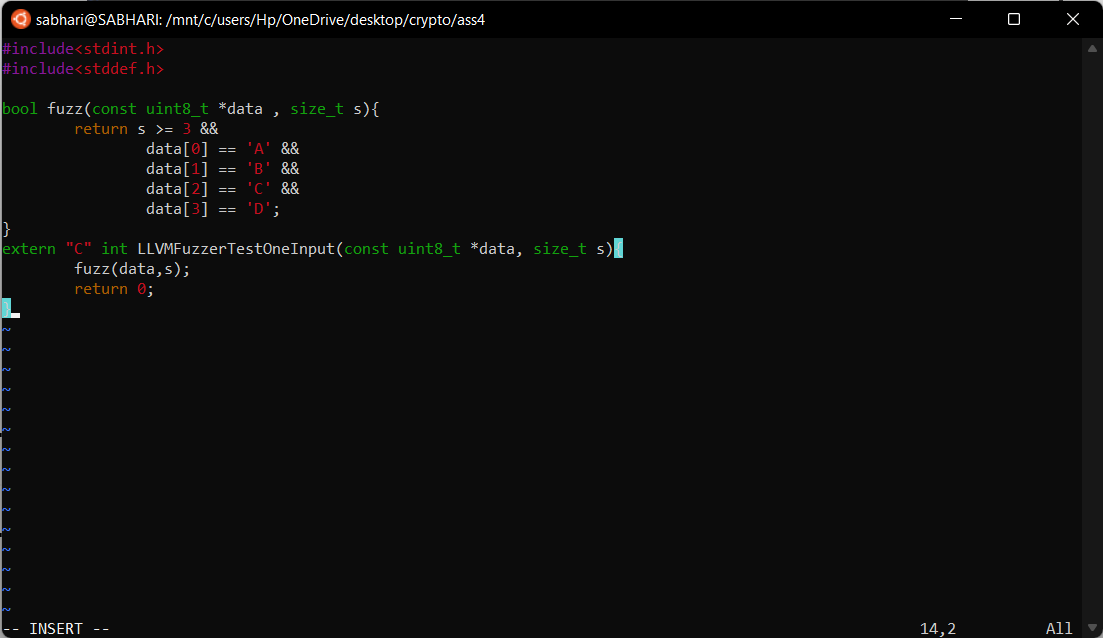
The two fuzzers used are libFuzzer, from the LLVM Core libraries, and the standalone tool American Fuzzy Lop.

libFuzzer can be checked out from LLVM’s Subversion repository and built using their directions. You supply a test driver as a function called LLVMFuzzerTestOneInput with C linkage. The result is a standalone program that exercises the code inside that function. It uses some Clang compiler-supplied instrumentation, via the -fsanitize-coverage option, to monitor which paths are exercised, so gcc is not an option. It must be compiled with -fsanitize=memory to ensure no initialization is missed.AFL is a standalone tool that uses binary rewriting to instrument the code being tested. It supplies wrapper compilers that call either Clang or gcc as necessary. The test driver is written as a main program that takes the random string from standard input, which means each run is a separate process. However, if you use Clang, there is a special “fast” mode that instruments your code as a compiler pass, rather than a final object code rewrite. This means the instrumentation itself can be optimized, producing faster binaries. So clang-fast is used.

Also, AFL is more mature and has more sophisticated mutation algorithms, and though its one-process-per-test approach is slower, the special Clang support compensates. Address sanitizing seems much faster than memory sanitizing, and you can always re-run all the (unique path) test cases afterwards.

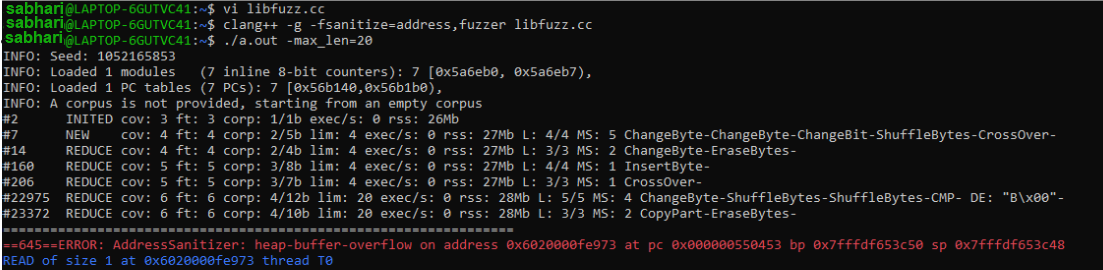
***Explanation with code screenshots:***

***Program: libfuzz.c***

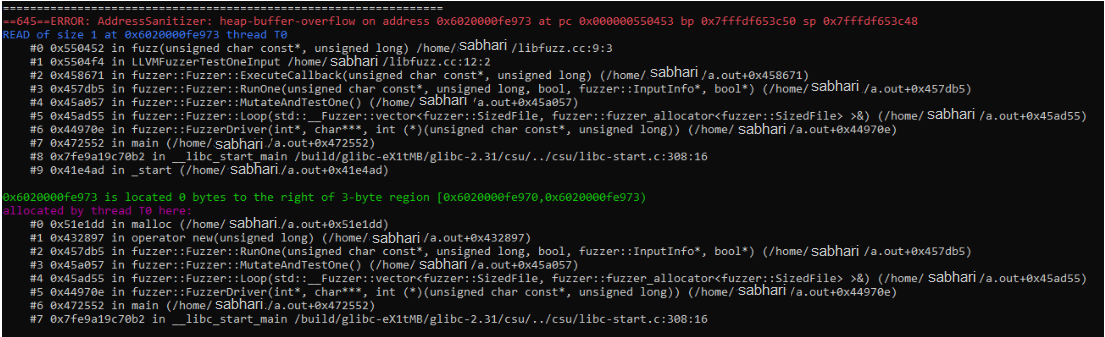
******

When size==3, data[3] does not exist and this is the bug. Lets fuzz the executable and identify bugs. Clang+ is used to add libfuzzer to our program. LLVMFuzzerTestOneInput is the driver of libfuzzer to test fuzz() function. We will compile and see what happens. clang++ -g –fsanitize=address,fuzzer filename is used to compile source file for fuzzing. Max\_len=20 is used to run fuzzer with max input size of 20.

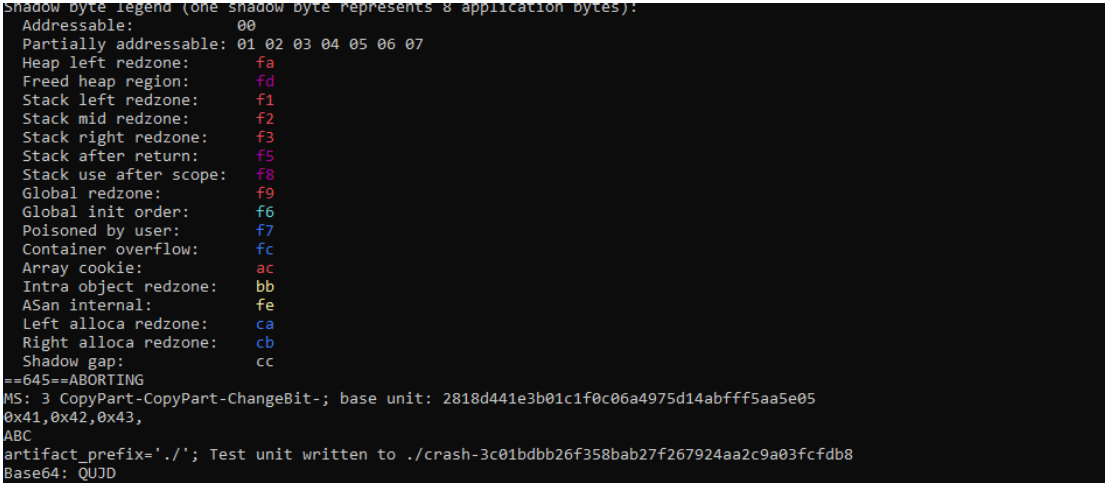
C:\Users\Hp\OneDrive\Desktop\crypto\ass4\img\1.1.png



Here the error says that heap-overflow has occurred at address 0x6020000fe973 at instruction pc 0x000000550453 bp 0x7fffdf653c50 sp 0x7fffdf653c48.



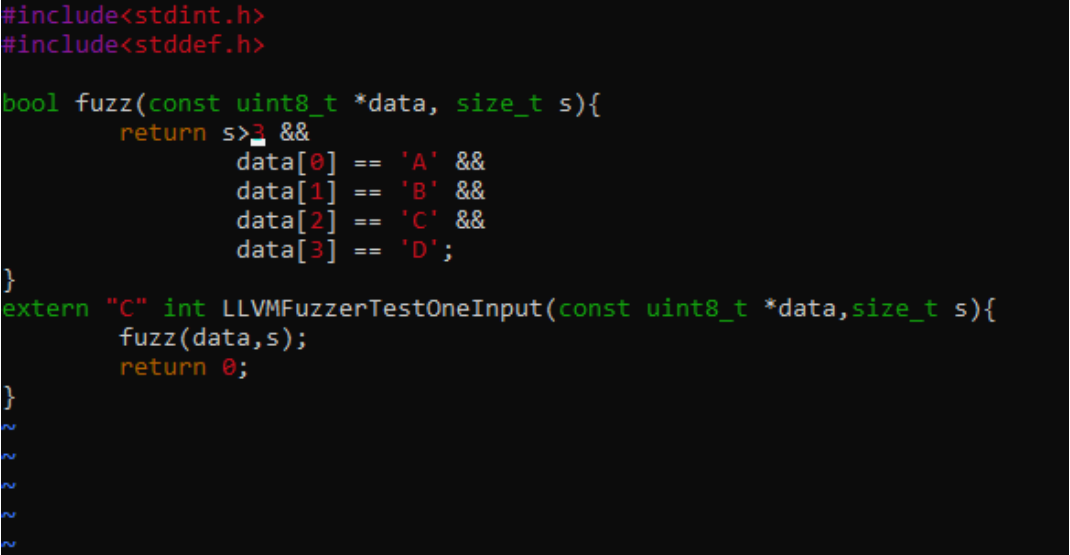


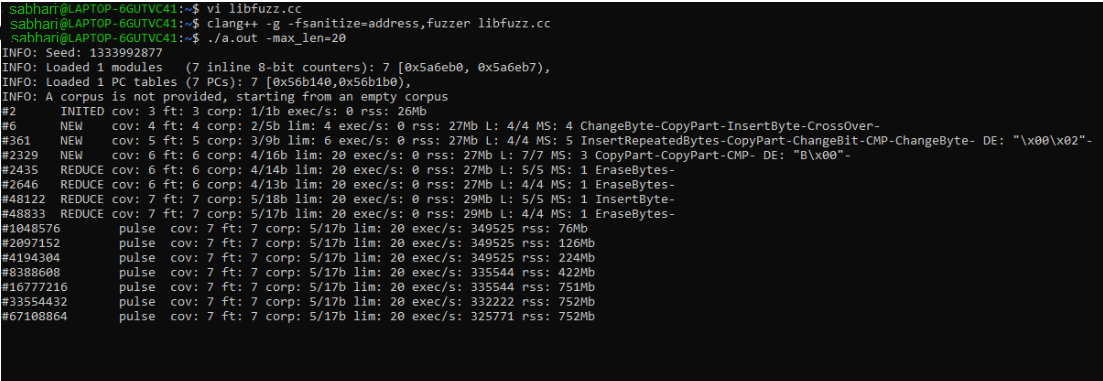


The above image consists of a crash file into which the input that caused heap buffer overflow is written. Using cat command to see the contents of the crash.

C:\Users\Hp\OneDrive\Desktop\crypto\ass4\img\6.png

***Program after removal of discovered bug:***

******

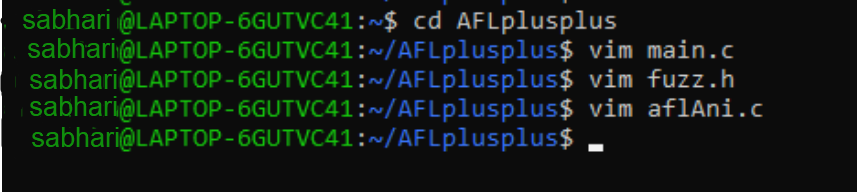


after removing bug, we found that libfuzzer is fuzzing atleast 67108864 input with a memory of 752 mb.

***AFL***

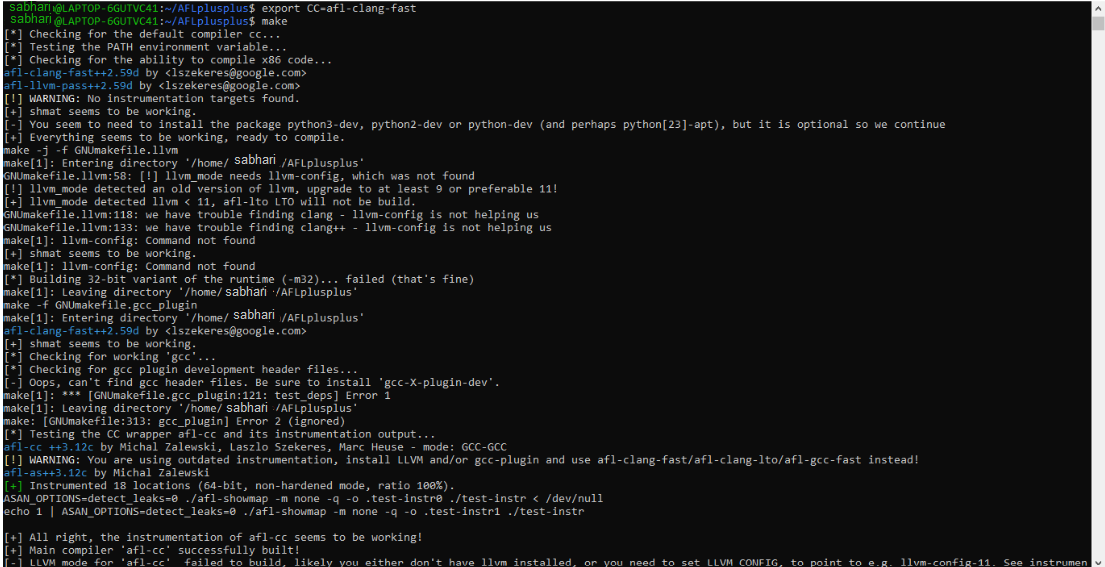
***Explanation with code screen shots:***

aflAni.c is the c file which has been written with several vulnerabilities.

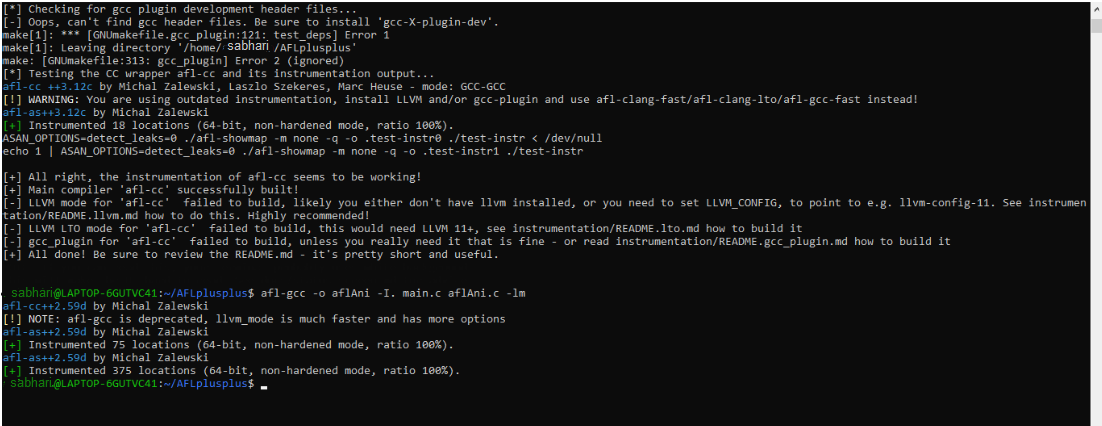


For compiling AFL, we first have to configure clang by using the following command: export CC=afl-clang-fast as they can’t directly accept command line option.

And then we pass command ‘make’ to compile.

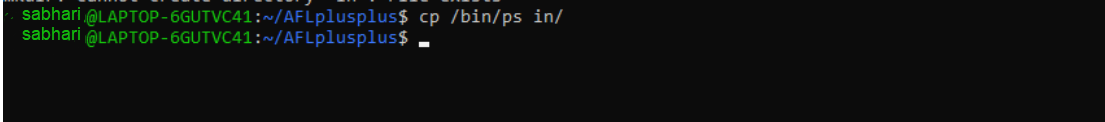


Then afl-gcc –o aflAni –i. main.c aflAni.c –lm (afl-gcc is a command, aflAni is a .c file).

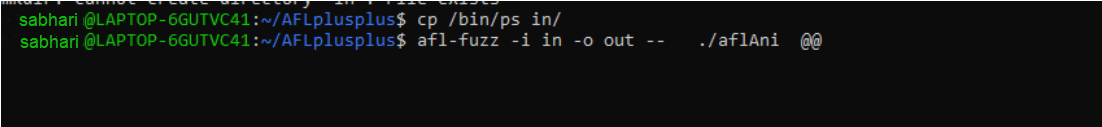


We have set input and output directories namely ‘in’ and ‘out’ respectively. mkdir in and mkdir out. It is then filled with garbage values

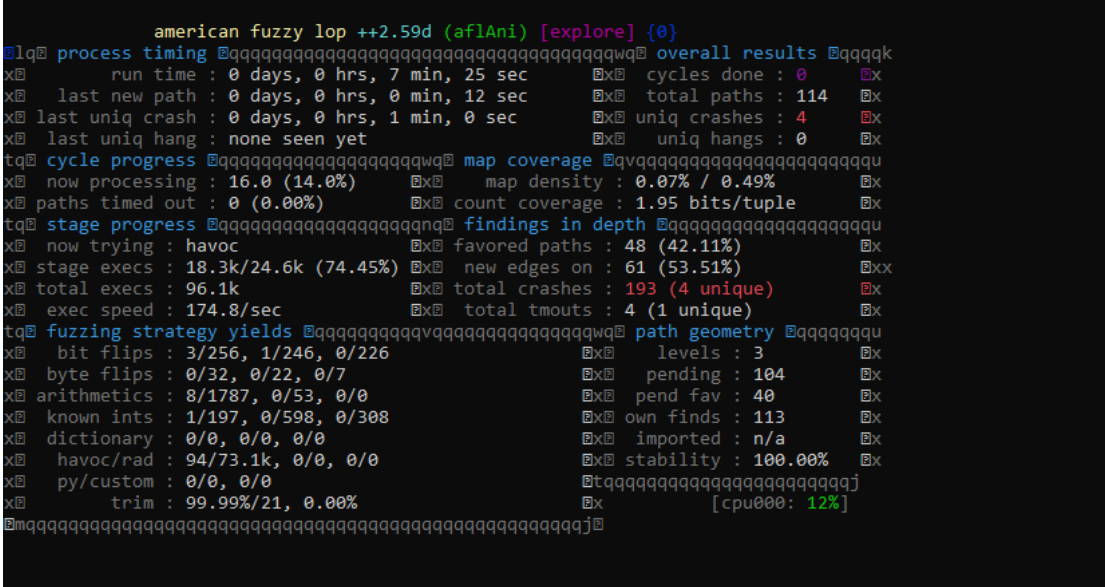
cp /bin/ps in/.

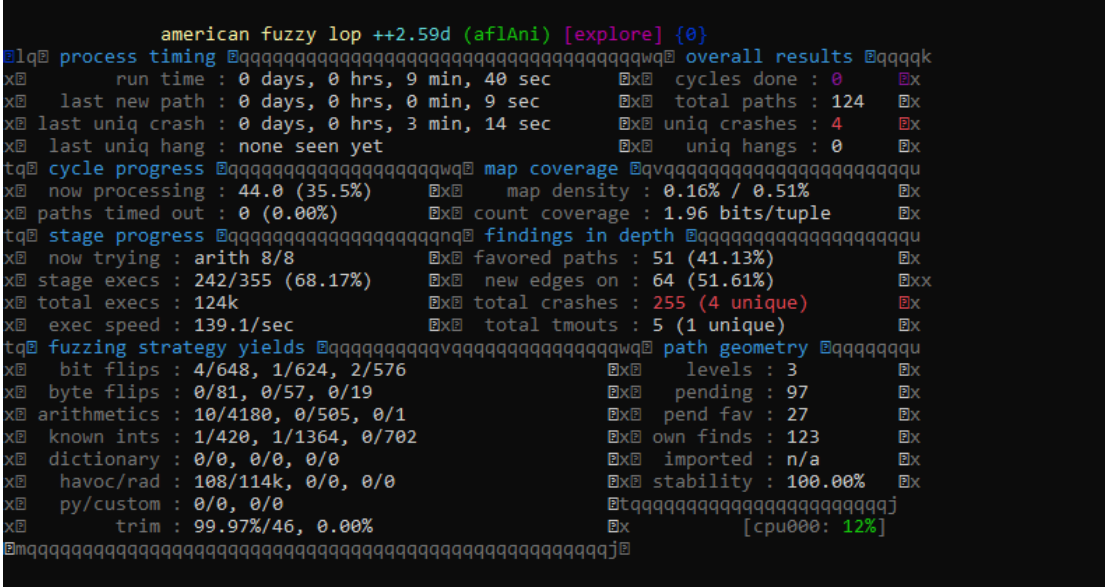


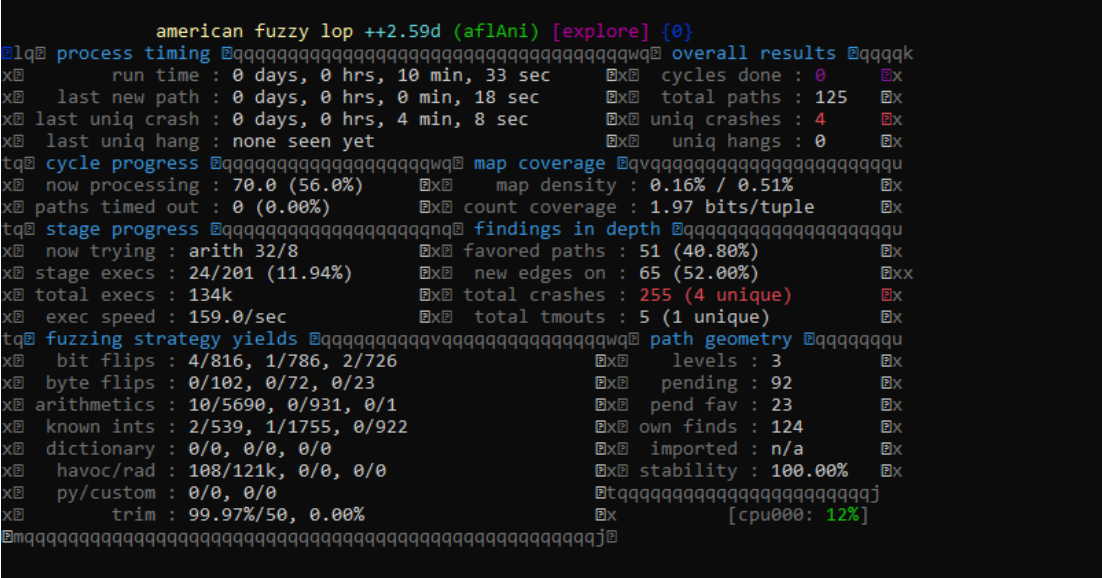
Then afl-fuzz -i in -o out -- ./aflAni @@ command is used to start fuzzing.

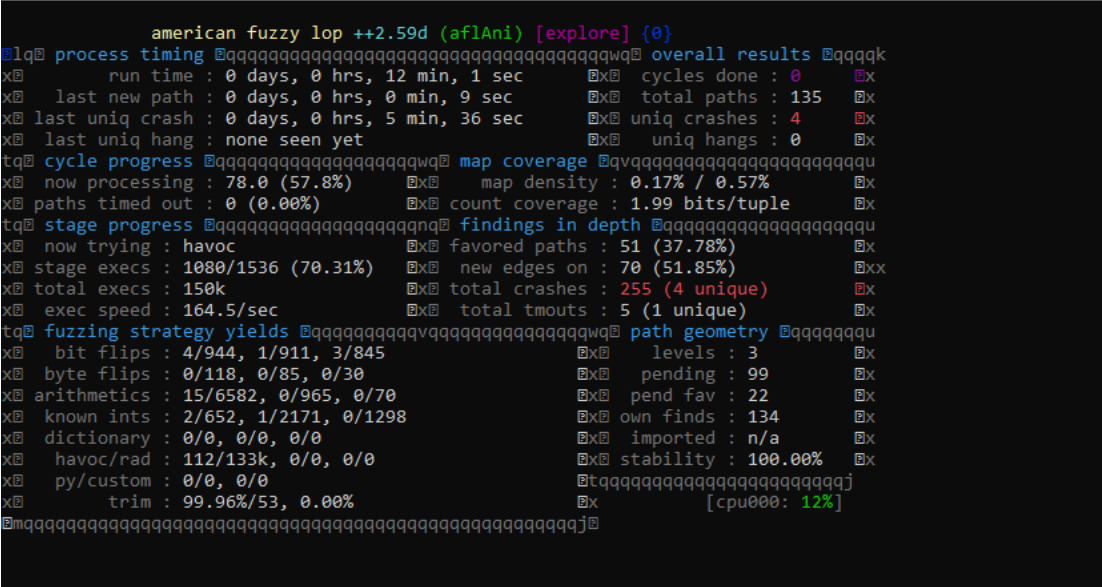


***AFL opens a terminal:***



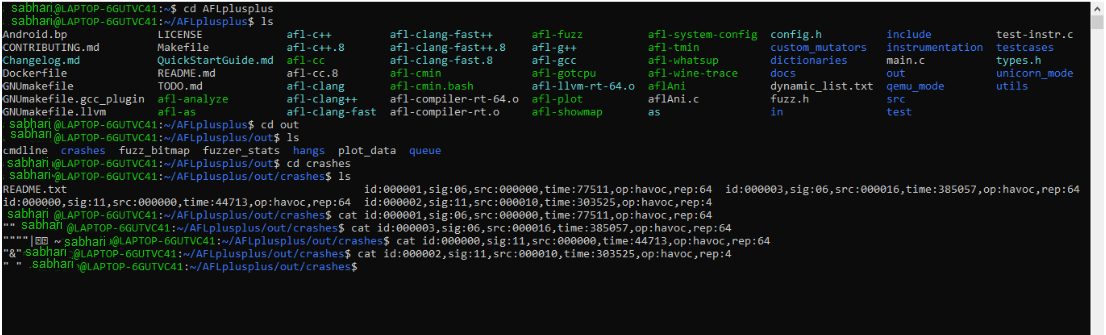




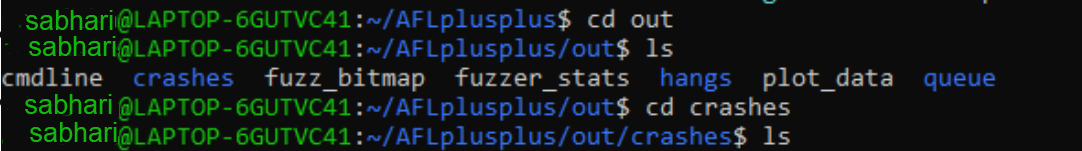


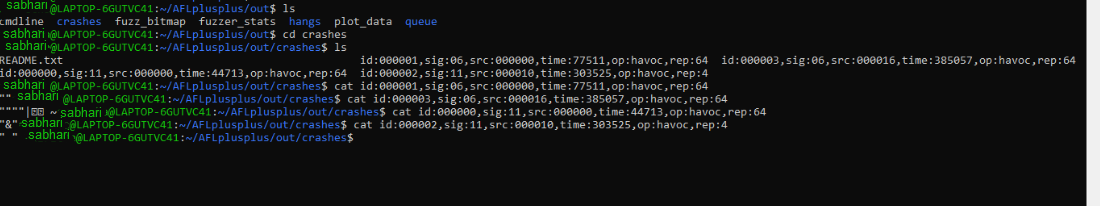
This shows the total crash count as 255 out of which 4 crashes are unique. It discovered 135 unique paths in the time period of 12 minutes 1 second and 150k executions.

After termination of this window, we can examine the crash in the mentioned output directory (here my output directory is out).



We Make ls and check what is there in the output directory. We will find crash directory there. So, we Enter crash directory and list it. It will provide the crash that has occurred!





Thus, using cat command, we can see the contents of the crashes that has occurred!

