Take-home Experiment
American Fuzzy Lop (AFL)

1. Downloaded Virtual box and installed ubuntu-20.04.3-desktop-amd64 on it.
   Note: Following steps are performed on the virtual box
2. Downloaded the AFL from github
3. Downloaded the vulnerable code C code
4. Compile and execution of test-incr.c file without using AFL

```
aish@aish-VirtualBox:~/Documents/AFL-master$ gcc test-instr.c
aish@aish-VirtualBox:~/Documents/AFL-master$ ./a.out
```

```
aish@aish-VirtualBox:~/Documents/AFL-master$ ./a.out hehe

A non-zero value? How quaint!
aish@aish-VirtualBox:~/Documents/AFL-master$ ./a.out 747474

A non-zero value? How quaint!
aish@aish-VirtualBox:~/Documents/AFL-master$ ./a.out
10
A non-zero value? How quaint!
aish@aish-VirtualBox:~/Documents/AFL-master$ ./a.out
0
Looks like a zero to me!
aish@aish-VirtualBox:~/Documents/AFL-master$
```

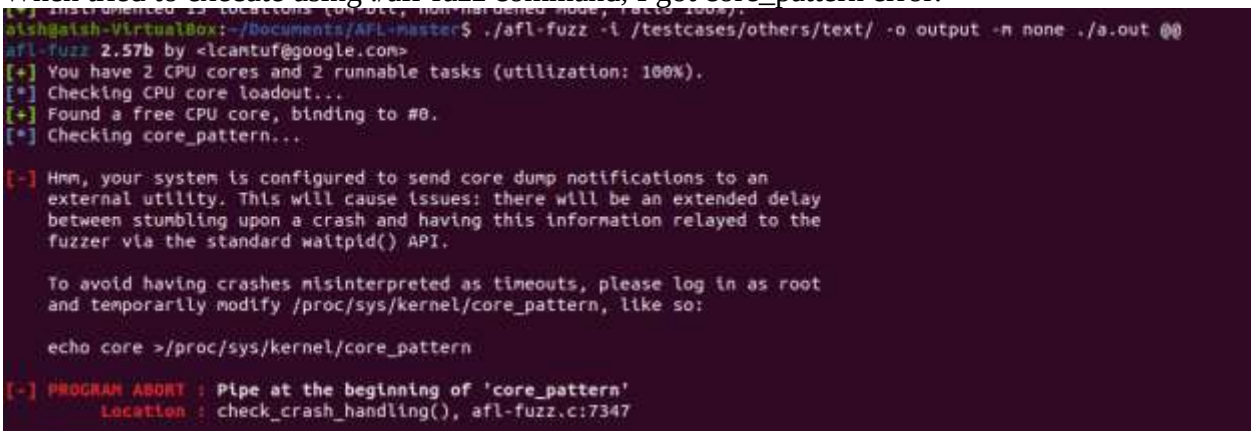
5. Install the AFL using make command

6. Compile the program / library to be fuzzed using afl-gcc.
   # `CC=/path/to/afl-gcc CXX=/path/to/afl-g++ ./configure --disable-shared`
   # make clean all
7. ./afl-gotcpu will give the number of core and their status, I have assigned 2 cores to my virtual box hence 2 cores are available.

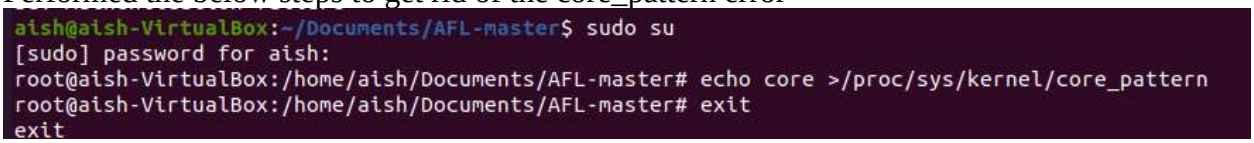
```
root@aish-VirtualBox:/home/aish/Documents/AFL-master# ./afl-gotcpu
afl-gotcpu 2.57b by <lcamtuf@google.com>
[*] Measuring per-core preemption rate (this will take 1.00 sec)...
    Core #0: AVAILABLE (100%)
    Core #1: AVAILABLE (101%)

>>> PASS: You can run more processes on 2 cores. <<<

root@aish-VirtualBox:/home/aish/Documents/AFL-master#
```

8. Compile AFL with the file test-instr.c. This file is provided with the AFL package for testing purpose. And then execute it

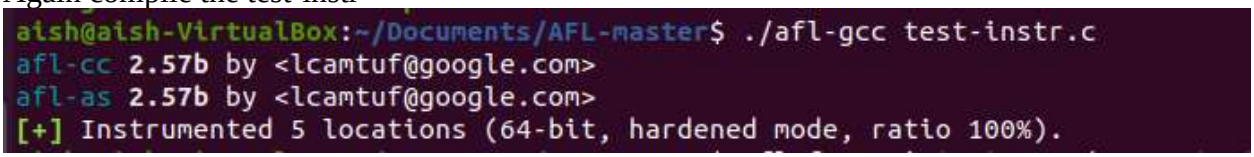
9. When tried to execute using ./afl-fuzz command, I got core_pattern error.



10. Performed the below steps to get rid of the core_pattern error



11. Again compile the test-instr

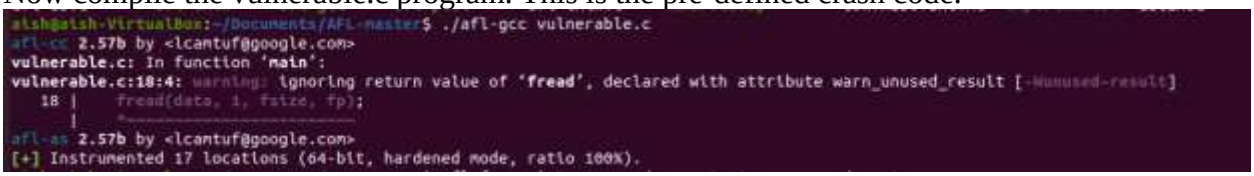


12. Execute the ./a.out file of test-instr.c using afl-fuzz command
./afl-fuzz -i testcases/ -o output -m none ./a.out @@

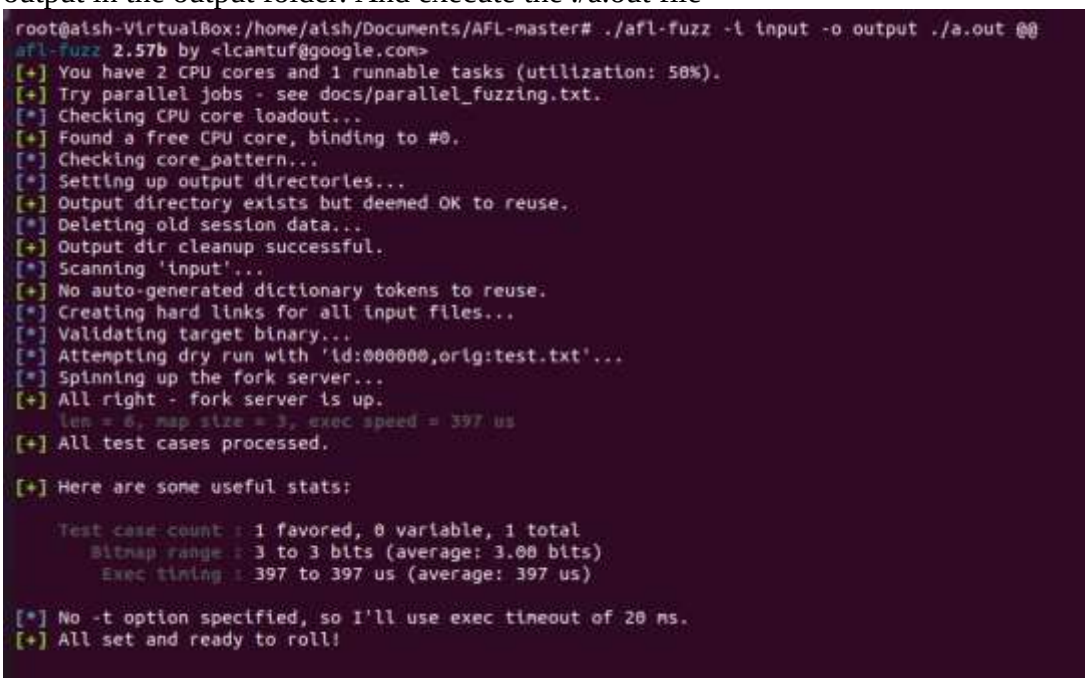
```
root@aish-VirtualBox:/home/aish/Documents/AFL-master# ./afl-gcc test-instr.c
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 5 locations (64-bit, non-hardened mode, ratio 100%).
root@aish-VirtualBox:/home/aish/Documents/AFL-master# ./afl-fuzz -i input -o output ./a.out @@
afl-fuzz 2.57b by <lcamtuf@google.com>
[+] You have 2 CPU cores and 3 runnable tasks (utilization: 150%).
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'input'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:test.txt'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 6, map size = 2, exec speed = 288 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 1 total
       Bitmap range : 2 to 2 bits (average: 2.00 bits)
        Exec timing : 288 to 288 us (average: 288 us)

[*] No -t option specified, so I'll use exec timeout of 20 ms.
[+] All set and ready to roll!
```

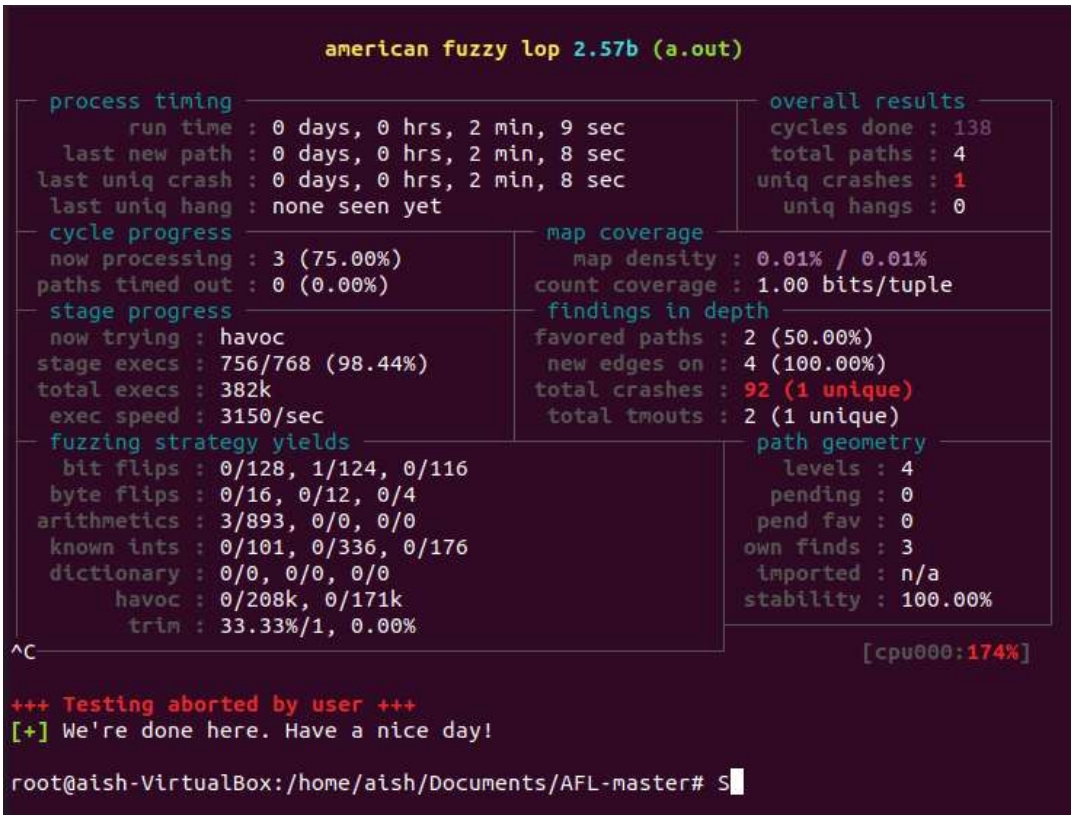
```
root@aish-VirtualBox:/home/aish/Documents/AFL-master# ./afl-fuzz -i input -o output ./a.out @@
afl-fuzz 2.57b by <lcamtuf@google.com>
[+] You have 2 CPU cores and 3 runnable tasks (utilization: 150%).
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'input'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:test.txt'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 6, map size = 2, exec speed = 288 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 1 total
       Bitmap range : 2 to 2 bits (average: 2.00 bits)
        Exec timing : 288 to 288 us (average: 288 us)

[*] No -t option specified, so I'll use exec timeout of 20 ms.
[+] All set and ready to roll!
```

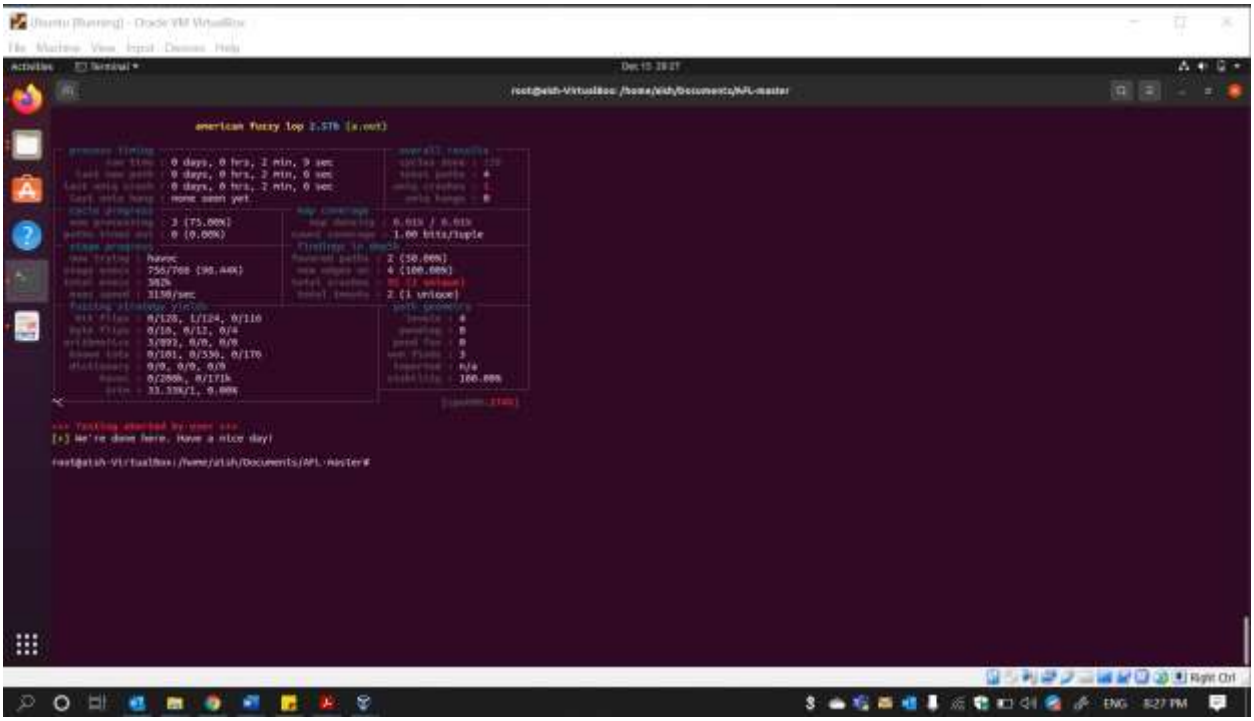
13. No crashes were present in the code.
14. Now compile the vulnerable.c program. This is the pre-defined crash code.



15. The code compiled successfully with 1 warning and no errors
16. Now execute the code with ./afl-fuzz with testcases in testcase folder and the generated output in the output folder. And execute the ./a.out file







17. From the output we can check that there was 1 vulnerability in the code. Total 64 inputs were checked and there was 1 unique crash with 44 failed input tests.

****

Question1: What is Fuzz Testing (basic idea) and its challenges? Which method does AFL use to solve the problems? (Hint: see its documentation)

Answer:

Fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software; it is responsible for most of the remote code execution and privilege escalation bugs found to date in security-critical software. American Fuzzy Lop is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm.

Challenges: fuzzing is also relatively shallow; blind, random mutations make it very unlikely to reach certain code paths in the tested code, leaving some vulnerabilities firmly outside the reach of this technique.

1st Solution to overcome the Challenge:

There have been numerous attempts to solve this problem. One of the early approaches - pioneered by Tavis Ormandy - is corpus distillation. The method relies on coverage signals to select a subset of interesting seeds from a massive, high-quality corpus of candidate files, and then fuzz them by traditional means. The approach works exceptionally well but requires such a corpus to be readily available. In addition, block coverage measurements provide only a very simplistic understanding of program state and are less useful for guiding the fuzzing effort in the long haul.

2nd Solution to overcome the Challenge:

Other, more sophisticated research has focused on techniques such as program flow analysis ("concolic execution"), symbolic execution, or static analysis. All these methods are extremely promising in experimental settings but tend to suffer from reliability and performance problems in practical uses - and currently do not offer a viable alternative to "dumb" fuzzing techniques.

Question 2: Briefly explain the overall algorithm of AFL approach.

Answer:

Simplifying a bit, the overall algorithm can be summed up as:

1. Load user-supplied initial test cases into the queue,
2. Take next input file from the queue,
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

The discovered test cases are also periodically culled to eliminate ones that have been obsoleted by newer, higher-coverage finds; and undergo several other instrumentation-driven effort minimization steps. As a side result of the fuzzing process, the tool creates a small, self-contained corpus of interesting test cases. These are extremely useful for seeding other, labor- or resource-intensive testing regimes - for example, for stress-testing browsers, office applications, graphics suites, or closed-source tools.

The fuzzer is thoroughly tested to deliver out-of-the-box performance far superior to blind fuzzing or coverage-only tools.

Question 3: How does AFL measure the code coverage during the fuzzing process? List formula it uses for calculation.

The instrumentation injected into compiled programs captures branch (edge) coverage, along with coarse branch-taken hit counts. The code injected at branch points is essentially equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

The cur_location value is generated randomly to simplify the process of linking complex projects and keep the XOR output distributed uniformly.

The shared_mem[] array is a 64 kB SHM region passed to the instrumented binary by the caller. Every byte set in the output map can be thought of as a hit for a particular (branch_src, branch_dst) tuple in the instrumented code.

The size of the map is chosen so that collisions are sporadic with almost all of the intended targets, which usually sport between 2k and 10k discoverable branch points:

Branch cnt	Colliding tuples	Example targets
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	-

At the same time, its size is small enough to allow the map to be analyzed in a matter of microseconds on the receiving end, and to effortlessly fit within L2 cache.

This form of coverage provides considerably more insight into the execution path of the program than simple block coverage. It trivially distinguishes between the following execution traces:

```
A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)
A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)
```

This aids the discovery of subtle fault conditions in the underlying code, because security vulnerabilities are more often associated with unexpected or incorrect state transitions than with merely reaching a new basic block.

The reason for the shift operation in the last line of the pseudocode shown earlier in this section is to preserve the directionality of tuples (without this, A ^ B would be indistinguishable from B ^ A) and to retain the identity of tight loops (otherwise, A ^ A would be obviously equal to B ^ B).

The absence of simple saturating arithmetic opcodes on Intel CPUs means that the hit counters can sometimes wrap around to zero. Since this is a fairly unlikely and localized event, it's seen as an acceptable performance trade-off.

Question 4: What is fork server? Why does AFL use it during the fuzzing?

Answer:

To improve performance, afl-fuzz uses a "fork server", where the fuzzed process goes through execve(), linking, and libc initialization only once, and is then cloned from a stopped process image by leveraging copy-on-write. The fork server is an integral aspect of the injected instrumentation and simply stops at the first instrumented function to await commands from afl-fuzz. With fast targets, the fork server can offer considerable performance gains, usually between 1.5x and 2x. It is also possible to:

- Use the fork server in manual ("deferred") mode, skipping over larger, user-selected chunks of initialization code. It requires very modest code changes to the targeted program, and With some targets, can produce 10x+ performance gains.
- Enable "persistent" mode, where a single process is used to try out multiple inputs, greatly limiting the overhead of repetitive fork() calls. This generally requires some code changes to the targeted program, but can improve the performance of fast targets by a factor of 5 or more
- Approximating the benefits of in-process fuzzing jobs while still maintaining very robust isolation between the fuzzer process and the targeted binary.

Question: what kind of whitebox fuzzing approach you can use to solve the above problem? How does your approach solve the problem in practice?

```
int foo (int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort (); // error
  return 0;
}
```

Answer:

- Symbolic execution is a software testing technique that is useful to aid the generation of test data and in proving the program quality.
- The execution requires a selection of paths that are exercised by a set of data values. A program, which is executed using actual data, results in the output of a series of values. The IF statement will split the path into 2 paths and the decision is based on the IF condition.
- The common approach for symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph. The conditional statements change the flow of code by branching. While performing the testing, all the branches should be checked atleast once. The flowgraph identifies the decision points and the assignments associated with each flow. By traversing the flow graph from an entry point, a list of assignment statements and branch predicates is produced.
- In symbolic execution, the data is replaced by symbolic values with set of expressions, one expression per output variable. While performing White box path testing, we can give input x= 10 which will make y= 13 thus making the IF condition TRUE. With black box, only 1 input will give error rest 2^32 will not.
- By directly giving the input as x=10, we can check the flow of the program which will lead to error thus exposing the ERROR.

References:

https://github.com/google/AFL/blob/master/docs/technical_details.txt

https://github.com/google/AFL
https://github.com/google/AFL/blob/master/docs/QuickStartGuide.txt
https://www.tutorialspoint.com/software_testing_dictionary/syntax_testing.htm