Problem 1: Fuzz a simple program

Introduction

Objective

To understand how fuzzing works and how to seed with proper inputs to make fuzzing efficient.

Background

Fuzzing is the automated process of entering random (mutated) data into a program and analyzing the results to find potentially exploitable bug. The software used to automate such a process is called a fuzzer. There are many fuzzer available, but this project is focused on American Fuzzy Lop (AFL). AFL is one of the example of fuzzer that employs genetic algorithms in order to increase code coverage. AFL receive some seed as input and generate random input mutated from the the seed input. AFL is typically used to test application with one small example input.

In AFL the seed becomes very important because it is used as the base for mutation of the randomized input. AFL try to generate potential token from the seed input. With this seed becomes very important in fuzzing using AFL. Additionally, a custom dictionary can be created to improve the fuzzing process. But the addition custom dictionary will affect the overall time taken for fuzzing. So, the custom dictionary should consist of a reasonable-sized token. The suggested token size is between 4 to 16 bytes.

Task

In this experiment a simple C program (`problem1.c`) is provided. The task is to generate all possible execution path using AFL. The callenge will be in determining the correct seed, dictionary and compilation option.

Cases

Bellow are all the generated cases from `problem1.c` along with the seed and compilation option used to trigger the corresponding case.

Case 1

Type	Crash
Seed	`AAAAAAA<?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAA<AAAAAAAAA`
Output	`CASE_01 [nread>16] crashing: null pointer dereference` `Segmentation fault`

In CASE_01 the generated input is one character different from the seed. So AFL only do 1 arithmathic changes. With the generated input the execution will lead to the if condition and falls into the first switch case. This will lead to segmentation fault, because the pointer is assigned to NULL address which means its not assigned to any memory. So when it tries to write using the pointer causes segmentation fault.

Case 2

Type	Timeout
Seed	`AAAAAAA<?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAA<aAAAAAAAA`
Output	`CASE_02 [nread>16] timeout`

In CASE_02 the generated input is one character different from the seed. So AFL only do 1 arithmathic changes. With the generated input the execution will lead to the if condition and falls into the second switch case. Here it leads to timeout because of the execution `sleep(2)` which will sleep for 2 seconds (timeout = 1 second).

Case 3

Type	Crash
Seed	`AAAAAAA<?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAA<*AAAAAAAA`
Output	`CASE_03 [nread>16] assert_failure:` `problem1_n: problem1.c:43: main: Assertion `0' failed.` `Aborted`

In CASE_03 the generated input is one character different from the seed. So AFL only do 1 arithmathic changes. With the generated input the execution will lead to the if condition and falls into the third switch case. It will lead to crash because of `assert(0)` which will be interpreted to assert value of false.

Case 4

There are two cases of CASE_04

Case 4: Loop

Type	Normal Exit (`Exit(0)`)
Seed	`AAAAAAA<?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAA<?AAAAAAAA`
Output	`CASE_04 [nread>16] loop: k=60`

In CASE_04 the generated input is exactly same from the seed. With the generated input the execution will lead to the if condition and falls into the fourth switch case, then the first if condition. This will lead to normal exit from the program

Case 4: Exit

Type	Normal Exit (`Exit(6)`)
Seed	`AAAAAAA<?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAA|?AAAAAAAA`
Output	`CASE_04 [nread>16] exit 6`

In CASE_04 the generated input is one character different from the seed. So AFL only do 1 arithmathic changes. With the generated input the execution will lead to the if condition and falls into the fourth switch case, then the second if condition. This will lead to execution of `exit(6)` (treated as normal execution).

Case 5

Type	Normal Exit (`Exit(0)`)
Seed	`AAAAAA22?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAA22?AAAAAAAA`
Output	`CASE_05 [nread>16] equality condition`

In CASE_05 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the if condition and falls into the fourth switch case, then else condition, then if condition. This will lead to normal execution (`exit(0)`).

Case 6

Type	Normal Exit (`Exit(0)`)
Seed	`AAAAAAed?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAed?AAAAAAAA`
Output	`CASE_06 [nread>16] inequality condition`

In CASE_06 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the if condition and falls into the fourth switch case, then else condition, then else condition. This will lead to normal execution (`exit(0)`).

Case 7

Type	Crash
Seed	`AAAAAAAAwAAAAAAAamer`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAAAwAAAAAAAamerican fuzzy lop`
Output	`CASE_07 Oh no, I've been caught, excellent!` `Aborted`

CASE_07 is quite hard to triggered. The condition require a very exact sentence of 'american fuzzy lop' here to trigger seed `AAAAAAAAwAAAAAAAamer` is needed (the closest seed can be given to AFL) then in addtional custom dictionary need to be used to complete the sentence in this case token `` `ican fuzzy lop` `` is added as token so it can generate the exact string.
This will lead to `abort()` statement which cause aborted execution (crash).

Case 8

Type	Crash
Seed	`AAAAAAAAwAAAAAAAamer`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAAAwAAAAAAAafl`
Output	`CASE_08 oops, I surrender` `Aborted`

Similar to CASE_07, CASE_08 have a very strict condition to be triggered. But to trigger CASE_08 seed `AAAAAAAAwAAAAAAAamer` is enough (without dictionary) but AFL need to do multiple mutation to generate the desired input. It will require 2 cycle of execution (seed → another seed → CASE_08). This can be improved by adding dictionary (require 1 cycle).
This will lead to `abort()` statement which cause aborted execution (crash).

Case 9

Type	Normal Exit (`return 0`)
Seed	`AAAAAAAAwAAAAAAAamer`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAAAwAAAAAAAamer`
Output	`CASE_09 nothing found`

In CASE_09 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the if condition and falls into the fifth switch case, then else condition. This will lead to normal execution (`return 0`).

Case 10

Type	Crash
Seed	`AAAAAAA<?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1 -fsanitize=address`
Generated Input	`AAAAAAA<AAAAAAAA`
Output	`CASE_10 can you catch me?` `Floating point exception`

CASE_10 is a random cases. It will access `buf[2048]` which out of bound of the pointer. Using ASan (address sanitizer) the execution will be possible. However CASE_10 generated input is the same as CASE_11 generated input beause both crash on CASE_12. So, AFL treat them as the same execution path. CASE_10 also will be triggered if the content of `buf[2048]` is non zero. In other word the case is triggered when at the execution time the memory address of `buf[2048]` is used in another application and contain value non-zero. When compiled with `afl-gcc` the division by zero will yield `Floating point exception`. So the program will crash.

Case 11

Type	Crash
Seed	AAAAAAA<?AAAAAAAA
Compile Flag	`afl-gcc problem1.c -o problem1 -fsanitize=address`
Generated Input	AAAAAAA<AAAAAAAA
Output	`CASE_11 and me?` `Floating point exception`

Similar to CASE_10, CASE_11 is not always executed with the same input. It depend on the content of `buf[2048]` which is outside the control of the program. Oppositely CASE_11 will be triggered in case `buf[2048]` is zero.
When compiled with `afl-gcc` the division by zero will yield `Floating point exception`. So the program will crash.

Case 12

Type	Normal Exit (`return 0`)
Seed	AAAAAAA<?AAAAAAAA
Compile Flag	`afl-clang problem1.c -o problem1`
Generated Input	AAAAAAA<AAAAAAAA
Output	`CASE_11 and me?` `CASE_12 so you've survived others, Here is a bonus value: 8286224`

CASE_12 is always executed after CASE_10 or CASE_11. But, with `afl-gcc` the execution of CASE_12 will cause `Floating point exception` (division by zero). So to trigger CASE_12 `afl-clang` need to be used to compile the program. In c program division by zero is treated as undefined so it up to the compiler to interpret the meaning. In `afl-gcc` it is treated as Floating point exception so it will lead to crash. But in `afl-clang` it is interpreted differently so it will not lead to crash (it will print random value). If option `-fsanitize=address` is used the value will be the same as content `buf[2048]`.
If using `afl-clang` this will result in normal execution.

Case 13

Type	Normal Exit (`return 0`)
Seed	AAAAAA AAAandsome
Compile Flag	afl-gcc problem1.c -o problem1
Generated Input	AAAAAA AAAandsome
Output	CASE_13 [nread>16] common

In CASE_13 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the if condition and falls into the default value of the switch case. This will lead to normal execution (`return 0`).

Case 14

Type	Crash
Seed	AAAAAAA<?AAAAAAAA
Compile Flag	afl-gcc problem1.c -o problem1
Generated Input	��AAAA�AAAA�A
Output	CASE_14 [nread<=16] crashing: null pointer dereference Segmentation fault

In CASE_14 the generated input is mutation of the seed. With the generated input the execution will lead to the else condition and falls into the first switch case. This will lead to segmentation fault, because the pointer is assigned to NULL address which means its not assigned to any memory. So when it tries to write using the pointer causes segmentation fault.

Case 15

Type	Timeout
Seed	AAAAAAA<?AAAAAAAA
Compile Flag	afl-gcc problem1.c -o problem1
Generated Input	AA�ayua0a�
Output	CASE_15 [nread<=16] timeout

In CASE_15 the generated input is mutation of the seed. With the generated input the execution will lead to the else condition and falls into the second switch case. Here it leads to timeout because of the execution `sleep(2)` which will sleep for 2 seconds (timeout = 1 second).

Case 16

Type	Crash
Seed	aaaaaaa<?
Compile Flag	afl-gcc problem1.c -o problem1
Generated Input	�GGGGGGG*GGG��
Output	CASE_16 [nread<=16] assert_failure: problem1_n: problem1.c:104: main: Assertion `0' failed. Aborted

In CASE_16 the generated input is mutation of the seed. With the generated input the execution will lead to the else condition and falls into the third switch case. It will lead to crash because of `assert(0)` which will be interpreted to assert value of false.

Case 17

Type	Normal Exit (`Exit(0)`)
Seed	`aaaaaaa<?`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`aaaaaaa<?`
Output	`CASE_17 [nread<=16] loop: k=60`

In CASE_17 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the else condition and falls into the fourth switch case, then if condition. This will lead to normal execution (`exit(0)`).

Case 18

Type	Normal Exit (`Exit(6)`)
Seed	`AAAAAAed?AAAAAAAA`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`TAAAAAey?A`
Output	`CASE_18 [nread<=16] exit 6`

In CASE_18 the generated input is mutated from the seed. With the generated input the execution will lead to the else condition and falls into the fourth switch case, then second if condition. This will lead to normal execution (`exit(6)`) (treated as normal execution)

Case 19

Type	Normal Exit (`Exit(0)`)
Seed	`AAAAAA@$?`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAA@$?`
Output	`CASE_19 [nread<=16] equality condition`

In CASE_19 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the else condition and falls into the fourth switch case, then else condition, then if condition. This will lead to normal execution (`exit(0)`)

Case 20

Type	Normal Exit (`Exit(0)`)
Seed	`AAAAAAAA?`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`AAAAAAAA?`
Output	`CASE_20 [nread<=16] inequality condition`

In CASE_18 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the else condition and falls into the fourth switch case, then else condition, then else condition. This will lead to normal execution (`exit(0)`)

Case 21

Type	Normal Exit (`return 0`)
Seed	`A`
Compile Flag	`afl-gcc problem1.c -o problem1`
Generated Input	`A`
Output	`CASE_21 [nread<16] common`

In CASE_18 the generated input is exatcly the same from the seed. With the generated input the execution will lead to the else condition and falls into the default value of the switch case. This will lead to normal execution (`return 0`)

Conclusion

From this experiment we learn that fuzzing is very effective to test a program. However it wil require more resource to trigger edge cases. To improve the performance fuzzer, custom dictionary (or rules) need to be added. Different compilation option also can lead to different execution path. So, choosing the suitable compilation option is needed. As for seeding the optimal seed is the one that lead to as many as possible normal execution path. So the fuzzer will perform more efficient. And the size of the input need to be as small as possible (AFL will help to trim the seed file to the smallest possible seed that lead to the same execution path).

Reference

- https://www.wired.com/2016/06/hacker-lexicon-fuzzing/
- http://www.geeknik.net/4rzj8nz7n
- https://github.com/mirrorer/afl