Vulnerability Detection: Fuzzing

2018044566 / Seoyeon Park
College of Computing
Hanyang University ERICA
nsa32752@naver.com

Abstract

There are many software vulnerability discovery techniques, fuzzing is the most common technique among them due to its convenience and easy to implement. There are various types of fuzzer, fuzz testing tool. Among the various fuzzers, we explore three versions: the general version, the kernel-targeted version, and the neural network-targeted version. We also explore how to detect software vulnerabilities using fuzzers. If fuzzing is performed by selecting the right fuzzer for the test program and customizing it as the user wants, the effect of fuzzing will be greater.

1 Introduction

Attacks on software vulnerabilities cause serious damages. Especially zero-day attack, which are attacks made at a time when a patch for the vulnerability has not been developed, can result in serious damages. Therefore, discovering software vulnerabilities in advance is important. There are two methods to discover software vulnerabilities: static analysis and dynamic analysis. Static analysis finds vulnerabilities without running a program. Conversely, dynamic analysis finds vulnerabilities while executing a program. It includes validation testing, debugging, and unit testing for the input values generated by developers during development [1].

For dynamic analysis, fuzzing is the most common automated software testing technique for discovering vulnerabilities. Fuzzing was introduced by Professor Barton Miller at the

University of Wisconsin in 1988 [1]. Briefly, fuzzing is random testing. It is the process of generating random inputs and finding inputs that causes crash while executing programs.

The great advantage of fuzzing is easy to implement. If we use static analysis, we must fully understand the program. However, in fuzzing, we don't need fully understand the program. Using fuzzer, a program that performs fuzz testing, we easily perform fuzz testing. Just we monitor the program runs properly. That is why it is very widely used.

Various types such as OS, server, mobile application, embedded devices, SOC, etc. can be used as target program for fuzzing. In this project, we examined AFL, a fuzzer targeting application program, and Syzkaller, which targets OS Kernel. Using AFL, we explore how to analyze the crash found through fuzz testing. We also explore how to find vulnerabilities in kernel through Syzkaller. Furthermore, we examine how to perform fuzz testing in machine learning based programs.

2 OVERVIEW

In section 3, we introduce what is fuzzing and general process of fuzz testing. From section 4 to 6, we introduce three types of fuzzers. In section 4, we study how finds the software vulnerabilities using fuzzer. We use AFL (American Fuzzy Lop), a popular coverage-guided evolutionary fuzzer. From Section 5, we introduce another coverage-guided fuzzer, Syzkaller, which targets OS kernels. In Section 6, we introduce the Tensorfuzz, which targets machine learning programs, and find out why fuzzer targeting machine learning is needed through Tensorfuzz.

3 FUZZING

3.1 FUZZING & FUZZ TESTING

Defined in IETF RFC 2828, vulnerability is a flaw of weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. [13]

Testing for discovering software vulnerabilities can be divided into two methods: Manual search method and automatic search method. For the automatic search method, there are a static analysis and dynamic analysis. Static analysis finds vulnerabilities without running a program. Conversely, dynamic analysis finds vulnerabilities while executing a program. Fuzzing is the most common automated software testing technique. [3] [14]

3.1.1 Fuzzing

Fuzzing is one of the software testing methods to find software vulnerabilities. It is distinguished from other software testing method in that it detects security-related bugs. It makes random input to find the input that causes the program leading to crash or behave strangely. So, fuzzing is also called random testing. The input used in fuzzing is the outside of the program's expected behavior, and fuzzing is not necessary to include an expected input space. Fuzzing aims to detect the outside of program's input. [2]

3.1.2 Fuzz Testing

Fuzz testing is to use fuzzing to test whether the test program violates security policy. [2]

3.2 FUZZ TESTING ALGORITHM [2]

General process of fuzzing consists of 6 steps as follows. This algorithm is general enough to accommodate existing fuzzing technology.

3.2.1 Preprocess

As a step to be performed before fuzzing, test program instrumentation, seed selection and seed trimming are performed.

3.2.1.1 Instrumentation

Instrumentation is divided into two methods: static and dynamics. First, static instrumentation is performed before executing the test program, and instrument the source code. If the test program relies on libraries, additional instrumentations are needed. On the other hand, dynamic instrumentation is performed in the input evaluation step, and the probability of overhead occurrence is lower than static instrumentation. Also, dynamically linked libraries can be easily isolated.

3.2.1.2 Seed selection

It is the process of extracting part of the initial input. For example, in MP3 file, the range in which the mutation can be performed is very large. Therefore, it finds part of the inputs that achieve the same coverage as the original one.

3.2.1.3 Seed trimming

It is a step to reduce the size of input. The smaller the size of input, the less memory is consumed and higher processing efficiency can be obtained.

3.2.2 Schedule

After the preprocess step, it is a step of selecting an input to be injected into the test program. The input is selected from the queue, which is a set of useful input for performing fuzzing. This step aims to select the most interesting input for the test program such as having a high probability of finding a unique bug or achieving maximum coverage from the queue.

3.2.3 Input Generation

In this step, using the input which is selected in previous step creates a new input. There are two methods to create new input: Generation-based and mutation-based. First, Generation-based method is to use predefined model. It uses spec of test program such as network protocol and system call template to create suitable test cases. In mutation-based method, mutates the input using bit-flipping, arithmetic mutation, block-based mutation, dictionary-based mutation, etc.

3.2.4 Input Evaluation

In input evaluation step, run the test program and find the input that causes crash or behave strangely. The process of analyzing and reporting the resulting execution is also performed at this step. This step is to remove crash that caused the same as previous crash and leave only a unique crash, and to report the crash to the user. It uses a bug oracle such as a sanitizer to detect crashes such as memory vulnerabilities that cannot be detected during the fuzzing process. Also, priority is given to the test case according to the severity and uniqueness of the test case.

3.2.5 Configuration Update

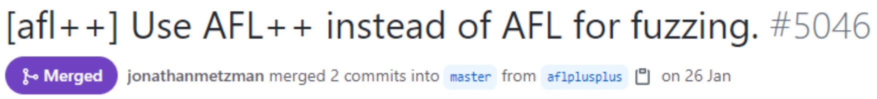
In configuration step, input that achieves new coverage although it doesn't cause a crash or behaved strangely, is put into the queue.

3.2.6 Continue

Fuzz testing repeats the scheduling step to the configuration step.

4 AFL++: MUTATIONAL, COVERAGE-GUIDED FUZZER

AFL++ is mutational coverage-guided fuzzer, which is a combination of AFL and useful functions among additional functions of AFL implemented by individuals. In January 2021, Google's OSS-Fuzz officially committed 'Use AFL++ instead of AFL for fuzzing' to AFL Github.



[afl++] Use AFL++ instead of AFL for fuzzing. #5046

Merged jonathanmetzman merged 2 commits into master from aflplusplus on 26 Jan

Picture 1. Use AFL++ instead of AFL for fuzzing [6]

4.1 AFL

American Fuzzy Lop(AFL) is mutational coverage guided fuzzer. It mutates test cases which is given as a simple and finds a new coverage.

4.1.1 Mutations

AFL mutates test cases to create various test cases. Although the test cases not generate a crash, if new coverage is achieved, it is put in the queue and utilized in next fuzz iteration. There are 10 methods for mutations.

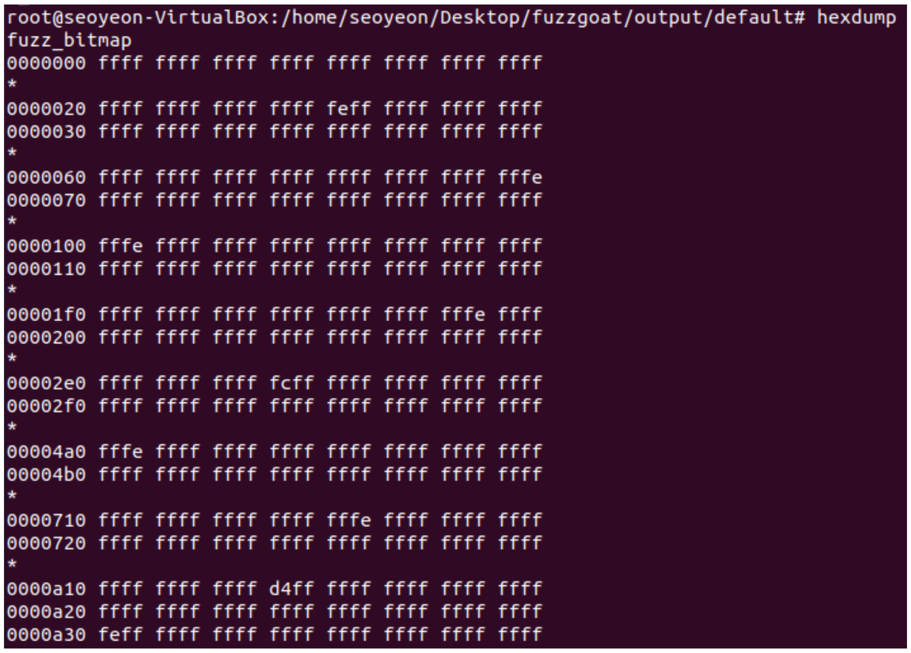
Stage	Operations	Effect
Deterministic/ Havoc	Bitflips	Flip single bit
Deterministic/ Havoc	Interesting values	NULL, -1, 0 etc.
Deterministic/ Havoc	Overwrite	Replace with random
Deterministic	Addition/Subtraction byte	Add/ Subtract random value
Deterministic	Extras overwrite/insertion (Dictionary)	Extras: strings scraped from binary
Havoc	Random Value	Insert random value
Havoc	Deletion	Delete from parent

Table 1: methods of mutations [4]

The mutation method can be largely divided into a deterministic method and a havoc (non-deterministic) method, and there is also a splice method which two test cases are combined into one and then the havoc method is applied. In AFL, first fuzz iteration should perform deterministic mutation stage. After first fuzz iteration, AFL performs havoc stage. In havoc stage, if the test case does not discover a crash or new coverage, a splice step is performed. Splice step differs from the havoc step in that it combines and uses random two inputs in the queue. [4] [9]

4.1.2 Coverage guided feedback

AFL measures the number of times the branch runs in each fuzz iteration. Through this, path explosion is prevented. This number is counted in the bitmap method, and each byte represents a branch. The bitmap can be checked in fuzz_bitmap file which is in output folder defined by user as follows.



```
root@seoyeon-VirtualBox:/home/seoyeon/Desktop/fuzzgoat/output/default# hexdump
fuzz_bitmap
0000000 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000020 ffff ffff ffff ffff feff ffff ffff ffff
0000030 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000060 ffff ffff ffff ffff ffff ffff ffff fffe
0000070 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000100 fffe ffff ffff ffff ffff ffff ffff ffff
0000110 ffff ffff ffff ffff ffff ffff ffff ffff
*
00001f0 ffff ffff ffff ffff ffff ffff fffe ffff
0000200 ffff ffff ffff ffff ffff ffff ffff ffff
*
00002e0 ffff ffff ffff fcff ffff ffff ffff ffff
00002f0 ffff ffff ffff ffff ffff ffff ffff ffff
*
00004a0 fffe ffff ffff ffff ffff ffff ffff ffff
00004b0 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000710 ffff ffff ffff ffff fffe ffff ffff ffff
0000720 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000a10 ffff ffff ffff d4ff ffff ffff ffff ffff
0000a20 ffff ffff ffff ffff ffff ffff ffff ffff
0000a30 feff ffff ffff ffff ffff ffff ffff ffff
```

Picture 2: fuzz_bitamp

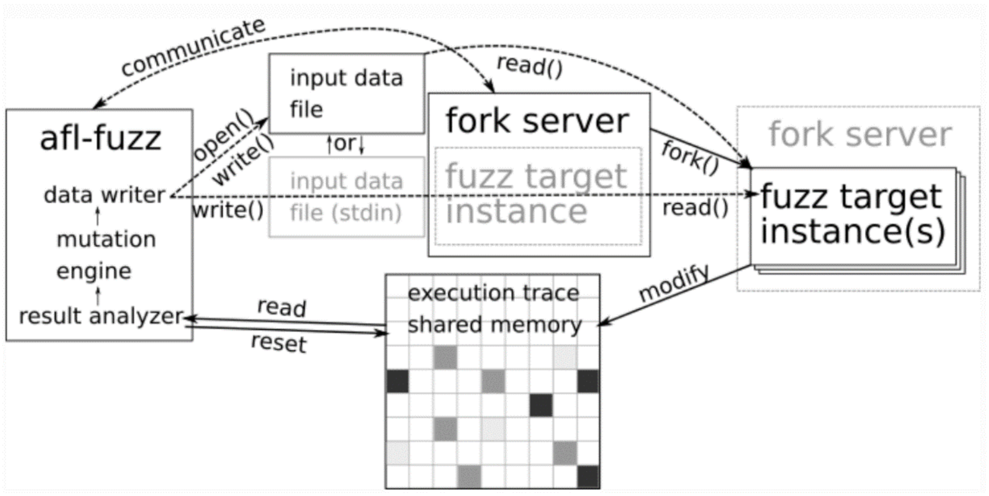
Each branch is given a random two-byte constant ID. When each branch is reached, XOR operation is performed on the previous branch ID and the ID currently reached. Applying hashing function in XOR value to determine which item in the bitmap represents the corresponding branch combination, and each time a particular branch combination is executed, the appropriate byte increases within the bitmap.

Since the size of this bitmap is limited, collision occurs. To prevent this as much as possible, AFL performs seed trimming to reduce test cases and improve execution speed during the coverage feedback process. [10]

4.1.3 Seed scheduling

To find useful test case which has a possibility to discover new coverage, AFL uses evolutionary algorithm, Evolutionary algorithm applies genetic transformation such as mutation or recombination to select a test case that is mode likely to discover new coverage. This assumes that the test case created in this way is more likely to be effective.

4.1.4 How to work AFL [5]



Picture 3: how to work AFL [5]

STEP 1) Run the program which fuzz in afl-fuzz

STEP 2) As the program is executed, it communicates with AFL using pipe and makes fork server. Then, the target instance is executed as a forkcall()

STEP 3) The test case is delivered to the target program.

STEP 4) After execution is completed, the execution result is recorded in the shared memory.

STEP 5) Afl-fuzz reads the records left by the target program and created new inputs.

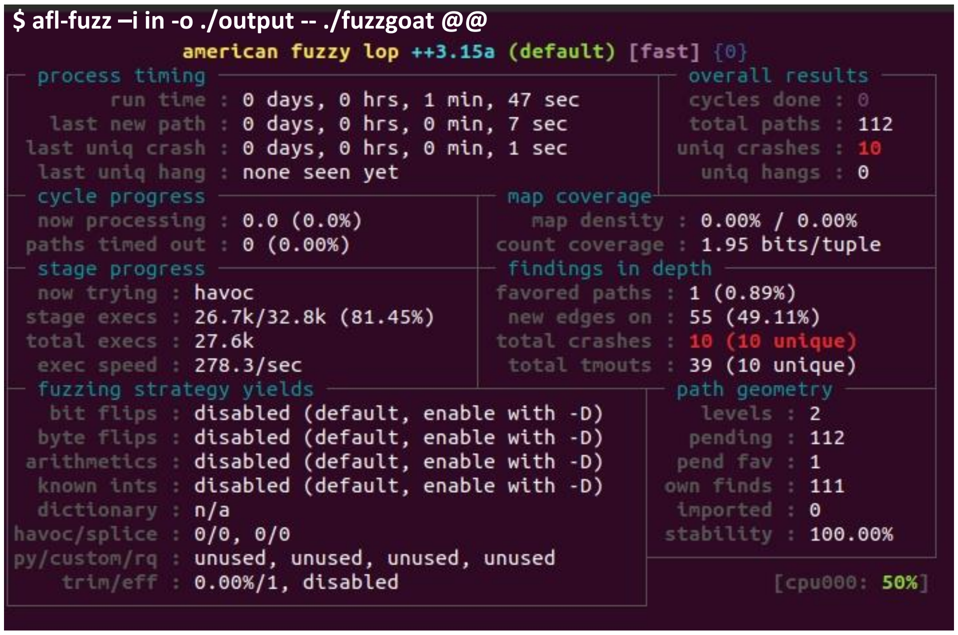
STEP 6) The newly created input enters the program again and is executed.

4.1.4.1 *forkserver*

When executing the target program, execve() system call is used to execute the program. In this case, since the program initialization process is continuously performed, a forksever is used to reduce the load generated by this. When injecting the test case into the target program, after specifying an input value, the target program forks itself. The child process then executes the test cases, and the parent process waits for the child process to end. Using this method, fuzzer does not need to repeat the program initialization step when re-executing the program every time.

4.2 How to use AFL++

To perform fuzz testing using AFL++, 'fuzz goat' was used as test program. Fuzz goat is C program, which contains intentionally created memory vulnerabilities. [11] When performing AFL++, fuzzing progress can be checked through the following status screen.



```
$ afl-fuzz –i in -o ./output -- ./fuzzgoat @@
                  american fuzzy lop ++3.15a (default) [fast] {0}
─ process timing ─────────────────────────────────────┬─ overall results ─
       run time : 0 days, 0 hrs, 1 min, 47 sec        │  cycles done : 0
  last new path : 0 days, 0 hrs, 0 min, 7 sec         │  total paths : 112
last uniq crash : 0 days, 0 hrs, 0 min, 1 sec         │ uniq crashes : 10
 last uniq hang : none seen yet                       │   uniq hangs : 0
─ cycle progress ─────────────────┬─ map coverage─────┴──────────────────
 now processing : 0.0 (0.0%)      │    map density : 0.00% / 0.00%
paths timed out : 0 (0.00%)       │ count coverage : 1.95 bits/tuple
─ stage progress ─────────────────┼─ findings in depth ──────────────────
 now trying : havoc               │ favored paths : 1 (0.89%)
stage execs : 26.7k/32.8k (81.45%)│  new edges on : 55 (49.11%)
total execs : 27.6k               │ total crashes : 10 (10 unique)
 exec speed : 278.3/sec           │  total tmouts : 39 (10 unique)
─ fuzzing strategy yields ────────┴─────────────┬─ path geometry ────────
  bit flips : disabled (default, enable with -D) │   levels : 2
 byte flips : disabled (default, enable with -D) │  pending : 112
arithmetics : disabled (default, enable with -D) │ pend fav : 1
 known ints : disabled (default, enable with -D) │ own finds : 111
 dictionary : n/a                                │  imported : 0
havoc/splice : 0/0, 0/0                          │ stability : 100.00%
py/custom/rq : unused, unused, unused, unused    ├────────────────────────
    trim/eff : 0.00%/1, disabled                 │          [cpu000: 50%]
```

Picture 4: Status screen in AFL++

4.2.1 Status screen [8]

1) Seed scheduling: AFL++ seed scheduling method is based on AFLFast. FAST is the default scheduling method.

2) Overall results_Cycles done: The number of times that fuzzer checked and fuzzed all test cases and returned to the beginning.

3) Findings in depth_total crashes: The number of crashes while performing fuzz testing. The crash is store in the output folder defined by user as follows. Even if it is a crash that occurs in another input, but same crash result appears, it is stored only for one crash. That is, only unique crash is stored in the folder.

4.3 Debugging crash using Address Sanitizer

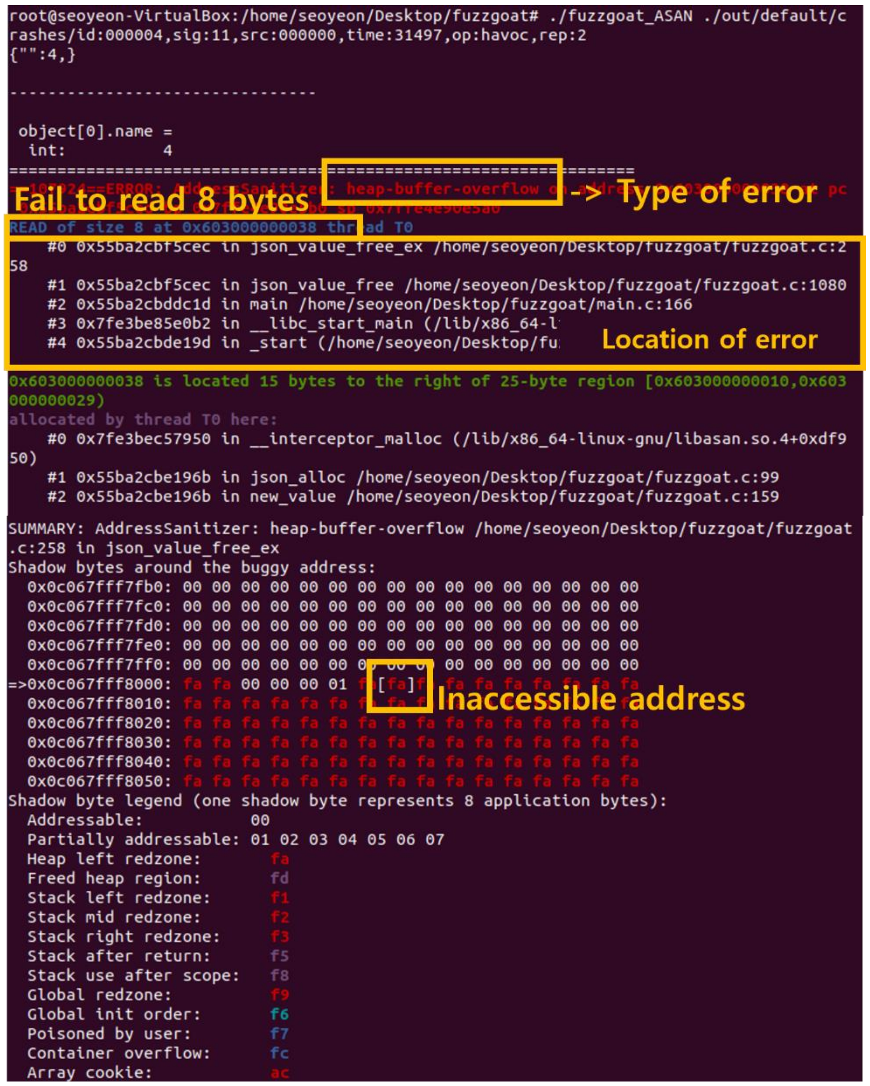
Crash occurrence can be reproduced using crash found through fuzzing as input of the target program. In this process, it is possible to analyze the location and cause of the crash using an address sanitizer.

4.3.1 Address sanitizer

Address sanitizer is a detector of memory error for C/C++. Detectable error type lists as follows: Use after free, heap buffer overflow, stack buffer overflow, global buffer overflow, use after return, use after scope, memory leaks, etc. [7]

4.3.2 How to analyze crash using Address sanitizer

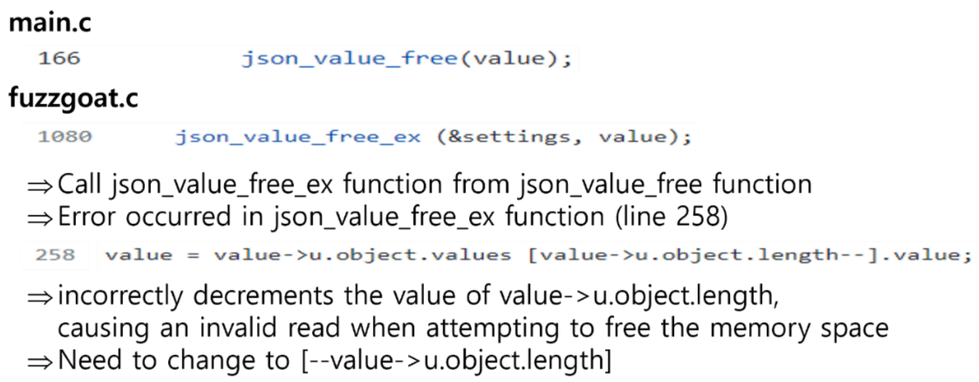
After building by setting Address sanitizer option in the target program, the following screen can be viewed by executing the test program.



```
root@seoyeon-VirtualBox:/home/seoyeon/Desktop/fuzzgoat# ./fuzzgoat_ASAN ./out/default/c
rashes/id:000004,sig:11,src:000000,time:31497,op:havoc,rep:2
{"":4,}

-------------------------------

 object[0].name =
  int:          4
=================================================================
==107924==ERROR: Add[illegible]: heap-buffer-overflow o[illegible] pc
[illegible] sp 0x7ffe4e90e5a0
READ of size 8 at 0x603000000038 thread T0
    #0 0x55ba2cbf5cec in json_value_free_ex /home/seoyeon/Desktop/fuzzgoat/fuzzgoat.c:2
58
    #1 0x55ba2cbf5cec in json_value_free /home/seoyeon/Desktop/fuzzgoat/fuzzgoat.c:1080
    #2 0x55ba2cbddc1d in main /home/seoyeon/Desktop/fuzzgoat/main.c:166
    #3 0x7fe3be85e0b2 in __libc_start_main (/lib/x86_64-l
    #4 0x55ba2cbde19d in _start (/home/seoyeon/Desktop/fu

0x603000000038 is located 15 bytes to the right of 25-byte region [0x603000000010,0x603
000000029)
allocated by thread T0 here:
    #0 0x7fe3bec57950 in __interceptor_malloc (/lib/x86_64-linux-gnu/libasan.so.4+0xdf9
50)
    #1 0x55ba2cbe196b in json_alloc /home/seoyeon/Desktop/fuzzgoat/fuzzgoat.c:99
    #2 0x55ba2cbe196b in new_value /home/seoyeon/Desktop/fuzzgoat/fuzzgoat.c:159
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/seoyeon/Desktop/fuzzgoat/fuzzgoat
.c:258 in json_value_free_ex
Shadow bytes around the buggy address:
  0x0c067fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c067fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c067fff8000: fa fa 00 00 00 01 fa[fa]fa fa fa fa fa fa fa fa
  0x0c067fff8010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
```

Picture 5: crash analysis result using address sanitizer

The result analyzed by the address sanitizer can be checked through code as follows:



main.c

```
166            json_value_free(value);
```

fuzzgoat.c

```
1080      json_value_free_ex (&settings, value);
```

⇒Call json_value_free_ex function from json_value_free function
⇒Error occurred in json_value_free_ex function (line 258)

```
258  value = value->u.object.values [value->u.object.length--].value;
```

⇒incorrectly decrements the value of value->u.object.length,
causing an invalid read when attempting to free the memory space
⇒Need to change to [--value->u.object.length]

Picture 6: code where the crash is occurred

5 SYZKALLER: KERNEL FUZZER

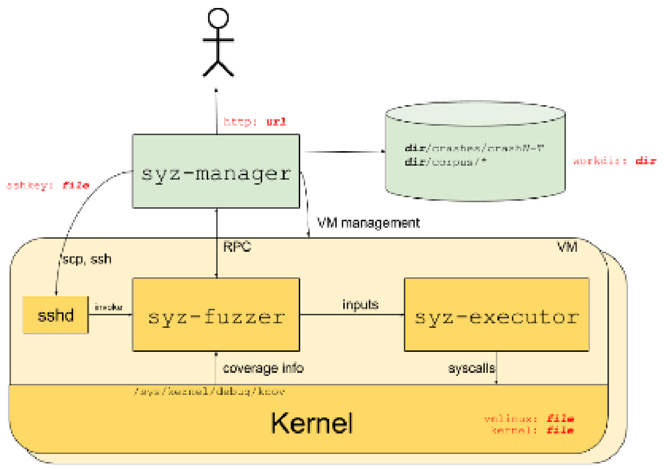
Syzkaller is coverage-guided kernel fuzzer. It aims to find bugs inside kernel. Syzkaller uses sanitizer coverage (tracing mode) and KCOV for coverage collection. When fuzzer is executed in AFL++, user can be checked the progress of fuzzing through status screen. In Syzkaller, user can be checked the status of fuzzing through the web. [12]

5.1 SYSTEMCALL IN SYZKALLER

Since input of kernel is a system call, the test program in the syzkaller means a series of system calls. Syzkaller also uses its own language, Syzlang, to describe system calls. This is because even the same system call can provide very different functions depending on its use and context. For example, ioctl system call interface is generally same, but performs completely different functions between different devices. Using syzlang, arguments and interfaces of ioctl can be logically distinguished. This definition of syzkaller system call can be found in the /sys/[OS]/*.txt file. [12]

5.2 SYZKALLER COMPONENTS [12]

Syzkaller runs in a way that three components interact: Syz-manager, Syz-fuzzer and Syz-executor. Regardless of how deep the testing is performed in the kernel part, Syz-manager,



syz-fuzzer and syz-executor should always be configured the same. The structure of Syzkaller is shown in the following picture.

Picture 7: Structure of Syzkaller [12]

5.2.1 Syz-manager

When fuzz testing is performed using Syzkaller, syz-manager executes first. Syz-manager starts a syz-fuzzer process inside of VMs.

5.2.2 Syz-fuzzer

Syz-fuzzer guides fuzzing process which contains input generation, mutation and minimization. Syz-fuzzer starts syz-executor process and sends inputs that trigger new coverage to syz-executor via RPC.

5.2.3 Syz-executor

Syz-executor executes a sequence of system calls. Program which comes from syz-fuzzer is executed by syz-executor, and syz-executor sends results back to syz-fuzzer.

5.3 How syzkaller works [12]

Step 1) Syz-manager starts VMs and RPC service

Step 2) Syz-manager run syz-fuzzer

Step 3) Syz-manager read corpus and send to syz-fuzzer

Step 4) Syz-fuzzer generates, mutates inputs

Step 5) Syz-executor executes program (invoke syscalls)

Step 6) Kcov calculate coverage in kernel

Step 7) Syz-fuzzer retrieve coverage in kernel

Step 8) If coverage increases, report the program as new corpus

Step 9) Goto step 3

6 Tensorfuzz: fuzzing machine learning model

Tensorfuzz is coverage-guided fuzzing for neural networks. It measures coverage by looking at the activations of computation graph. Computational graph is directed graph where the nodes correspond to mathematical operations. Traditional coverage metrics track which lines of code have been executed and which branches have been taken. But, in neural networks because of changes in input and output values, several different inputs execute the same lines of code and take the same branches, yet produce interesting variations in behavior. That is why fuzzer suitable for the neural network is needed. Using Tensorfuzz, we can discover the numerical errors in trained neural networks. [15]

7 Discussion

Fuzzing is a dynamic analysis method which generates random inputs to find vulnerabilities in test programs. As in the example in Section 4, the program's vulnerability could be discovered with a crash found through fuzzing. Like address sanitizer, crash may be useful by other sanitizer and libraries that fit into the program. In addition, Syzkaller was able to perform fuzzing not only on existing system calls but also on system calls defined by users. As a result, the usefulness and effect of fuzzer depend on how the user uses fuzzer. In addition to the fuzzers introduced in this project, there are numerous fuzzers. If fuzzing is performed by selection the right fuzzer for the test program and customizing it as the user wants, the effect of fuzzing will be greater.

8 Conclusion

In this project, we explored fuzzing and fuzzer as a whole.

We first examined the definition of fuzzing and general process of fuzzing. After them, we examined various fuzzers: AFL++, Syzkaller, Tensorfuzz. Using fuzzgoat as a test program, we explored how to use the crash which was discovered while performing AFL++. And through Syzkaller working process, we examined how the kernel performs fuzz testing. Finally, in machine learning program, we explored why fuzzers targeting NN are needed instead of existing fuzzers.

9 Reference

[1] James Fell, "A Review of Fuzzing Tools and Methods", in Originally published in PenTest Magazine on 10th March 2017

[2] Valentine J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, Maveric Woo, "The Art, Science, and Engineering of Fuzzing: A Survey", arXiv:1812.00140v4 [cs.CR] 8 Apr 2019

[3] https://www.youtube.com/watch?v=RDl81Jd83zc, "Samsung Security Tech Forum 2021, Challenges in automated vulnerability discovery through Fuzzing"

[4] https://cpuu.postype.com/post/9180710

[5] https://barro.github.io/2018/06/afl-fuzz-on-different-file-systems/

[6] https://github.com/google/oss-
fuzz/commit/665e4898215c25a47dd29139f46c4f47f8139417

[7] https://github.com/google/sanitizers/wiki/AddressSanitizer

[8] https://afl-1.readthedocs.io/en/latest/user_guide.html

[9] Andrea

, Dominik Maier, Heiko Eißfeldt, Marc Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research", USENIX.

[10] https://groups.google.com/g/afl-users/c/ThkdTNw_los

[11] https://github.com/fuzzstati0n/fuzzgoat

[12] https://github.com/google/syzkaller

[13] Jun Li, Bodong Zhao and Chao Zhang, "Fuzzing: a survey", Li et al. Cybersecurity (2018) 1;6

[14] James Fell, "A Review of Fuzzing Tools and Methods", PenTest Magazine on 10th March 2017

[15] Augustus Odena, Ian Goodfellow, "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing", arXiv:1807.10875v1 [stat.ML] 28 Jul 2018