AFL - American fuzzy lop

List

Description

- **AFL(American Fuzzy Lop)은 테스트 케이스의 코드 적용 범위(Code coverage)를 효율적으로 늘리기 위해 유전자 알고리즘(Genetic algorithm)을 사용하는 fuzzer입니다.**
  - 지원 가능한 OS는 Linux, OpenBSD, FreeBSD, NetBSD의 32bit 및 64bit를 지원합니다.
    - MacOS X 및 Solaris에서도 작동하지만 일부 제약이 있습니다.
  - 지원 가능한 프로그램 언어는 C, C++, Objective C를 지원합니다.
  - 지원 가능한 컴파일러는 gcc, g++, clang, clang++를 지원합니다.
  - 지원 가능한 테스트 방식은 White-box, Black-box를 지원합니다.
    - 리눅스에서는 QEMU 옵션을 사용해서 블랙 박스 바이너리를 퍼징 할 수 있습니다.

Site

- http://lcamtuf.coredump.cx/afl/

Install

```
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
$ tar -xvf afl-latest.tgz
$ cd afl-2.49b/
$ make
$ sudo make install
```

Commands

Command	Description	Basic methods of use
afl-analyze	파일 포맷 분석기	afl-analyze -i <test case file> target_app
afl-clang	clang wrapper	clang 명령어와 동일합니다.
afl-clang++	clang++ wrapper	clang++ 명령어와 동일합니다.
afl-cmin	중복되는 테스트 케이스 제거	afl-cmin -i <test case dir> -o <output dir> target_app
afl-fuzz	AFL의 코드 퍼지	afl-fuzz -i <test case dir> -o <output dir> target_app
afl-g++	g++ wrapper	g++명령어와 동일합니다.
afl-gcc	gcc wrapper	gcc 명령어와 동일합니다.
afl-gotcpu	CPU 선점 비율 출력	afl-gotcpu
afl-plot	진행률 출력- "gnuplot" 설치 필요	afl-plot <afl state dir> <graph output dir>
afl-tmin	테스트 케이스 최소화	afl-tmin -i <test case file> -o <output file> target_app
afl-whatsup	상태 점검 도구	afl-whatsup <afl_sync_dir>

Description of commands

afl-fuzz

- 해당 도구는 바이너리를 대상으로 다양한 퍼징을 시도합니다.

White-box, Black-box test

- **해당 도구는 White-box, Black-box 테스트를 지원합니다.**
  - White-box를 사용하기 위해서 afl에서 제공하는 컴파일러를 이용해 바이너리를 빌드해야 합니다.
  - Black-box를 사용하기 위해 afl에서는 QEMU를 사용하며, -Q 옵션을 이용해 사용할 수 있습니다.
- 다음과 같이 White-box 테스트를 할 수 있습니다.

White-box

```
afl-fuzz -i <test case dir> -o <output dir> target_app
```

- 다음과 같이 black-box 테스트를 할 수 있습니다.

Black-box

```
afl-fuzz -Q -i <test case dir> -o <output dir> target_app
```

데이터 입력 옵션

Standard input

```
afl-fuzz -i <test case dir> -o <output dir> target_app [params...]
```

File input

```
afl-fuzz -i <test case dir> -o <output dir> target_app @@
```

Parallel fuzzing

- **해당 도구는 병렬 퍼징을 지원합니다.**
    - afl-fuzz의 모든 복사본은 하나의 CPU 코어를 차지합니다.
    - 즉,n 코어 시스템에서는 거의 항상 성능 저하없이 거의 동시 n 개의 동시 퍼징 작업을 실행할 수 있습니다
        - afl-gotcpu 도구를 사용하여 확인할 수 있음
    - 사실 멀티 코어 시스템에서 단 하나의 작업에만 의존한다면 하드웨어를 충분히 활용 하지 못할 것 입니다.
    - 병렬 처리가 일반적으로 올바른방법입니다.

Single-system parallelization

- 단일 시스템 병렬화를 이용하기 위해 로컬 시스템의 여러 코어에 단일 작업을 병렬로 연결하기위한 빈 디렉토리("sync dir")를 생성합니다.
    - 해당 디렉토리에 모든 인스턴스가 공유됩니다.
- 다음과 같이 마스터 인스턴스(-M)를 실행합니다.

마스터 인스턴스

```
./afl-fuzz -i testcase_dir -o sync_dir -M fuzzer01 [...other stuff...]
```

    - 다음과 같이 보조 인스턴스(-S)를 실행합니다.

보조 인스턴스

```
$ ./afl-fuzz -i testcase_dir -o sync_dir -S fuzzer02 [...other stuff...]
$ ./afl-fuzz -i testcase_dir -o sync_dir -S fuzzer03 [...other stuff...]
```

Multi-system parallelization

- 다중 시스템 병렬화 처리의 기본 작동 원리는 단일 시스템 병렬화에서 설명한 메커니즘과 비슷합니다.
- 두 가지 작업을 수행하는 간단한 스크립트가 필요합니다.
- 로컬 시스템에서 모든 <fuzzer_id> 디렉토리의 "/queue/" 경로 아래에 있는 파일들을 압축합니다.

```
for s in {1..10}; do
      ssh user@host${s} "tar -czf - sync/host${s}_fuzzid*/[qf]*" >host${s}.tgz
    done
```

- 압축된 파일을 모든 컴퓨터에 파일을 배포하고 압축을 해제합니다

```
  for s in {1..10}; do
      for d in {1..10}; do
        test "$s" = "$d" && continue
        ssh user@host${d} 'tar -kxzf -' <host${s}.tgz
      done
    done
```

Parallel fuzzing using AFL

- https://raw.githubusercontent.com/mirrorer/afl/master/docs/parallel_fuzzing.txt

afl-analyze

- **해당 도구는 Test case의 파일 포맷을 분석합니다.**
  - 데이터 스트림으로 부터 순차적으로 전달되는 데이터들을 가져오며, 매 입력마다 바이너리의 동작을 관찰합니다.
- 다음과 같은 정보를 유추 할 수 있습니다.
  - no-op block
  - Critical stream
  - "magic value" 섹션
  - 내용으로 의심되는 영역
  - 길이 필드로 의심되는 영역
  - 체크섬 블럭으로 의심되는 영역
  - 체크썸 또는 Magic 값으로 의심되는 영역
- 다음과 같이 사용할 수 있습니다.

afl-analyze -i testcase/test1.txt ./test

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-analyze -i testcase/test1.txt ./test

afl-analyze 2.49b by <lcamtuf@google.com>


[+] Read 4 bytes from 'testcase/test1.txt'.
[*] Performing dry run (mem limit = 50 MB, timeout = 1000 ms)...
[*] Analyzing input file (this may take a while)...

     01  - no-op block              01  - suspected length field
     01  - superficial content      01  - suspected cksum or magic int
     01  - critical stream          01  - suspected checksummed block
     01  - "magic value" section

[000000]  a  #0a  a  #0a

[+] Analysis complete. Interesting bits: 0.00% of the input file.
[+] We're done here. Have a nice day!

lazenca0x0@ubuntu:~/Documents/AFL/test$
```

afl-cmin

- **해당 도구는 Test case의 중복을 최소화 합니다.**
- 다음과 같이 가장 적합한 test case만을 분리 합니다.
  - 4개의 test case에서 7개의 tuple을 발견했으며, 2개의 파일로 축소했습니다.
  - 여기서 사용된 test case는 아래 "Create to Test cases."에서 생성한 파일을 사용했습니다.

afl-cmin -i testcase/ -o newTestCase/ ./test

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-cmin -i testcase/ -o newTestCase/ ./test
corpus minimization tool for afl-fuzz by <lcamtuf@google.com>

[*] Testing the target binary...
[+] OK, 4 tuples recorded.
[*] Obtaining traces for input files in 'testcase/'...
    Processing file 4/4...
[*] Sorting trace sets (this may take a while)...
[+] Found 7 unique tuples across 4 files.
[*] Finding best candidates for each tuple...
    Processing file 4/4...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
    Processing tuple 7/7...
[+] Narrowed down to 2 files, saved in 'newTestCase/'.

lazenca0x0@ubuntu:~/Documents/AFL/test$ cd newTestCase/
lazenca0x0@ubuntu:~/Documents/AFL/test/newTestCase$ ls -al
total 16
drwxrwxr-x 2 lazenca0x0 lazenca0x0 4096 Aug 15 20:08 .
drwxrwxr-x 5 lazenca0x0 lazenca0x0 4096 Aug 15 20:08 ..
-rw-rw-r-- 2 lazenca0x0 lazenca0x0    4 Aug  9 00:18 test1.txt
-rw-rw-r-- 2 lazenca0x0 lazenca0x0    9 Aug  9 00:19 test3.txt
lazenca0x0@ubuntu:~/Documents/AFL/test/newTestCase$
```

afl-tmin

- **해당 도구는 Test case의 최적화를 진행합니다.**
- 최적화되는 내용은 다음과 같습니다.
    - 데이터 블럭의 최소화
    - 기호의 최소화
    - 문자의 최소화
- 다음과 같이 Test case를 최소화 할 수 있습니다.
    - 68개의 문자가 56개로 변경되었습니다.
    - 임의 문자들이 숫자 0(0x30)으로 변경되었습니다.
    - Test case에 기호가 없었기 때문에 기호의 최소화는 진행되지 않았습니다.

afl-tmin -i result/crashes/id₩:000000₩,sig₩:11₩,src₩:000000₩,op₩:havoc₩,rep₩:128 -o testcaseMin ./test

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-tmin -i result/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,
rep\:128 -o testcaseMin ./test
afl-tmin 2.49b by <lcamtuf@google.com>

[+] Read 68 bytes from 'result/crashes/id:000000,sig:11,src:000000,op:havoc,rep:128'.
[*] Performing dry run (mem limit = 50 MB, timeout = 1000 ms)...
[+] Program exits with a signal, minimizing in crash mode.
[*] Stage #0: One-time block normalization...
[+] Block normalization complete, 68 bytes replaced.
[*] --- Pass #1 ---
[*] Stage #1: Removing blocks of data...
    Block length = 4, remaining size = 68
    Block length = 2, remaining size = 56
    Block length = 1, remaining size = 56
[+] Block removal complete, 12 bytes deleted.
[*] Stage #2: Minimizing symbols (1 code point)...
[+] Symbol minimization finished, 0 symbols (0 bytes) replaced.
[*] Stage #3: Character minimization...
[+] Character minimization done, 0 bytes replaced.
[*] --- Pass #2 ---
[*] Stage #1: Removing blocks of data...
    Block length = 4, remaining size = 56
    Block length = 2, remaining size = 56
    Block length = 1, remaining size = 56
[+] Block removal complete, 0 bytes deleted.

     File size reduced by : 17.65% (to 56 bytes)
    Characters simplified : 121.43%
     Number of execs done : 33
          Fruitless execs : path=12 crash=0 hang=0

[*] Writing output to 'testcaseMin'...
[+] We're done here. Have a nice day!
lazenca0x0@ubuntu:~/Documents/AFL/test$ hexdump result/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,
rep\:128
0000000 81b9 ad13 0000 76e1 04ff 007f eee7 ffff
0000010 64ff 0000 798a 9379 7980 7979 7966 e100
0000020 ff76 7fc0 e700 ffee ffff ffff 7f04 e700
0000030 ffee ffff 0064 6900 7979 7993 7979 0079
0000040 0100 ff00
0000044
lazenca0x0@ubuntu:~/Documents/AFL/test$ hexdump testcaseMin
0000000 3030 3030 3030 3030 3030 3030 3030 3030
*
0000038
lazenca0x0@ubuntu:~/Documents/AFL/test$ cat testcaseMin
0000000000000000000000000000000000000000000000000000000000000000
```

afl-gotcpu

- 해당 도구는 afl-fuzz에서 사용중인 cpu 선점 비율을 출력합니다.

afl-gotcpu

```
lazenca0x0@ubuntu:~$ afl-gotcpu
afl-gotcpu 2.49b by <lcamtuf@google.com>
[*] Measuring per-core preemption rate (this will take 1.00 sec)...
    Core #0: CAUTION (231%)

>>> CAUTION: You may still have 1 core available. <<<

lazenca0x0@ubuntu:~$
```

afl-plot

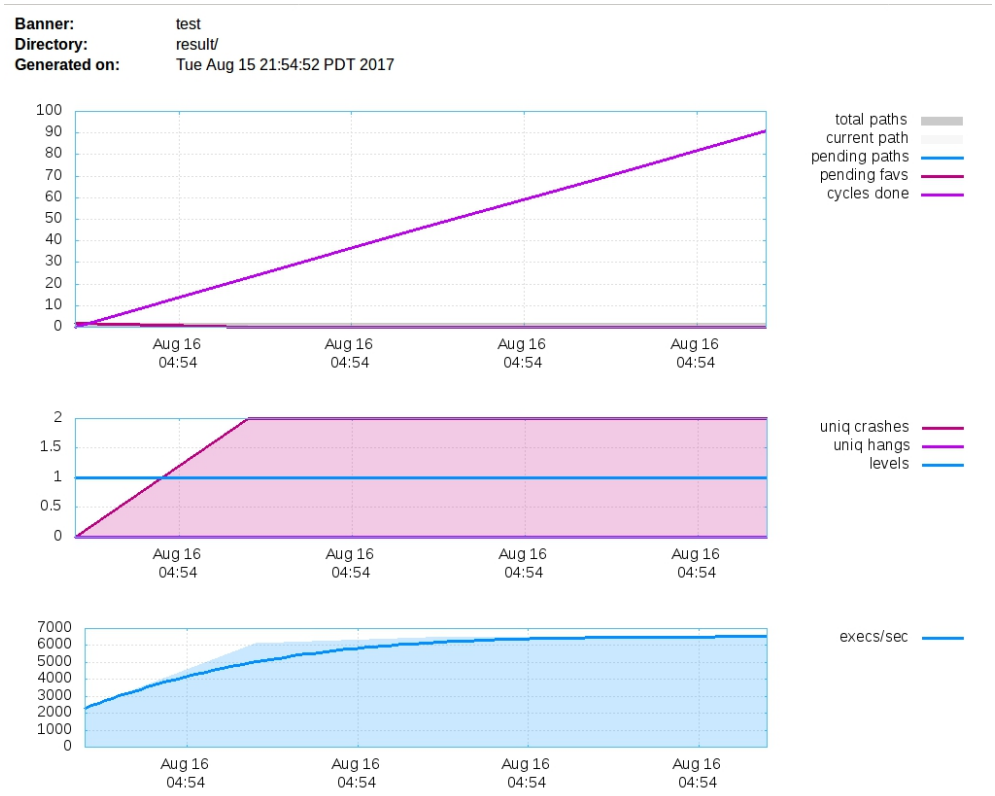
- **해당 도구는 fuzz의 진행 상황을 그래프로 출력합니다.**
- 다음과 같이 해당 프로그램을 실행하면, index.html 파일이 생성됩니다.

afl-plot result/ graph/

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-plot result/ graph/
progress plotting utility for afl-fuzz by <lcamtuf@google.com>

[*] Generating plots...
[*] Generating index.html...
[+] All done - enjoy your charts!
lazenca0x0@ubuntu:~/Documents/AFL/test$
```

index.html



alf-whatsup

- **해당 도구는 병렬 퍼징을 진행했을 때 개별 fuzzer의 상태를 확인할 수 있습니다.**

afl-whatsup result/

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-whatsup result/
status check tool for afl-fuzz by <lcamtuf@google.com>

Individual fuzzers
==================

>>> fuzzer1 (0 days, 0 hrs) <<<

  cycle 1, lifetime speed 1 execs/sec, path 0/2 (0%)
  pending 2/2, coverage 0.01%, no crashes yet

>>> fuzzer2 (0 days, 0 hrs) <<<

  cycle 1, lifetime speed 1 execs/sec, path 0/2 (0%)
  pending 2/2, coverage 0.01%, no crashes yet

Summary stats
=============

       Fuzzers alive : 2
      Total run time : 0 days, 0 hours
         Total execs : 0 million
    Cumulative speed : 2 execs/sec
       Pending paths : 4 faves, 4 total
  Pending per fuzzer : 2 faves, 2 total (on average)
       Crashes found : 0 locally unique

lazenca0x0@ubuntu:~/Documents/AFL/test$
```

Example

Example code

- **아래 코드는 다음과 같은 동작을 합니다.**
  - 사용자로 부터 ID, Password를 입력받습니다.
  - 입력받은 값이 프로그램이 요구하는 값과 일치하면 "Success"를 출력합니다.
  - 그렇지 않을 경우 "Fail"을 출력합니다.
- 여기서 중요한 부분은 사용자로부터 입력 받는 값에 대한 길이 제한이 없다는 것입니다.
  - 이로 인해 Stack Buffer Overflow가 발생합니다.

test.c

```
#include <stdio.h>
#include <string.h>

int main(void){
        char login[16];
        char password[16];

        printf("Login : ");
        scanf("%s",login);
        printf("Password : ");
        scanf("%s",password);

        if(strcmp(login,"root") == 0){
                if(strcmp(password,"toor") == 0){
                        printf("Success.\n");
                        return 0;
                }
        }
        printf("Fail.\n");
        return 1;
}
```

Create to Test cases.

- **다음과 같이 Test case를 생성합니다.**
  - ID가 틀린 경우
  - Password가 틀린 경우
  - ID, Password 모두 틀린 경우
  - ID, Password 정확한 경우

Create to test cases.

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ mkdir testcase
lazenca0x0@ubuntu:~/Documents/AFL/test$ cd testcase
lazenca0x0@ubuntu:~/Documents/AFL/test$ echo -e "a\toor" > test1.txt
lazenca0x0@ubuntu:~/Documents/AFL/test$ echo -e "root\na" > test2.txt
lazenca0x0@ubuntu:~/Documents/AFL/test$ echo -e "a\na" > test3.txt
lazenca0x0@ubuntu:~/Documents/AFL/test$ echo -e "root\toor" > test4.txt
lazenca0x0@ubuntu:~/Documents/AFL/test$
```

White-box testing

Build using afl-gcc.

- **다음과 같이 AFL에서 제공하는 컴파일러를 이용해 빌드합니다.**
  - 빌드된 파일이 정상적으로 동작하는 것을 확인 할 수 있습니다.
  - 테스트를 위해 Canary를 제거 합니다. (-fno-stack-protector)

afl-gcc -o test test.c

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-gcc -fno-stack-protector -o test test.c
afl-cc 2.49b by <lcamtuf@google.com>
test.c: In function 'main':
test.c:9:2: warning: ignoring return value of 'scanf', declared with attribute warn_unused_result [-Wunused-
result]
  scanf("%s",login);
  ^
test.c:11:2: warning: ignoring return value of 'scanf', declared with attribute warn_unused_result [-Wunused-
result]
  scanf("%s",password);
  ^
afl-as 2.49b by <lcamtuf@google.com>
[+] Instrumented 8 locations (64-bit, non-hardened mode, ratio 100%).
lazenca0x0@ubuntu:~/Documents/AFL/test$ ./test
Login : root
Password : toor
Success.
lazenca0x0@ubuntu:~/Documents/AFL/test$ ./test
Login : a
Password : a
Fail.
lazenca0x0@ubuntu:~/Documents/AFL/test$
```

Run afl-fuzz

- **다음과 같이 AFL을 실행하면 "uniq crashes"를 발견할 수 있습니다.**
  - 테스트 프로그램에서 2개의 Uniq crashes를 발견하였습니다.
- 옵션은 다음과 같습니다.
  - -i : Test case가 저장된 디렉토리 경로
  - -o : 탐지된 결과를 저장할 디렉토리 경로

Run AFL-fuzz

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ echo core > /proc/sys/kernel/core_pattern
lazenca0x0@ubuntu:~/Documents/AFL/test$ mkdir result
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-fuzz -i testcase/ -o result/ ./test
afl-fuzz 2.49b by <lcamtuf@google.com>
[+] You have 1 CPU core and 2 runnable tasks (utilization: 200%).
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'testcase/'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:test1.txt'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 4, map size = 34, exec speed = 1428 us
[*] Attempting dry run with 'id:000001,orig:test2.txt'...
    len = 7, map size = 37, exec speed = 596 us
[*] Attempting dry run with 'id:000002,orig:test3.txt'...
    len = 14, map size = 38, exec speed = 740 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 3 favored, 0 variable, 3 total
       Bitmap range : 34 to 38 bits (average: 36.33 bits)
        Exec timing : 596 to 1428 us (average: 921 us)

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

                        american fuzzy lop 2.49b (test)

 process timing  overall results
        run time : 0 days, 0 hrs, 0 min, 17 sec         cycles done : 16
   last new path : none yet (odd, check syntax!)        total paths : 3
 last uniq crash : 0 days, 0 hrs, 0 min, 11 sec        uniq crashes : 2
  last uniq hang : none seen yet                         uniq hangs : 0
 cycle progress  map coverage
  now processing : 1 (33.33%)            map density : 0.06% / 0.07%
 paths timed out : 0 (0.00%)          count coverage : 1.00 bits/tuple
 stage progress  findings in depth
  now trying : havoc                  favored paths : 3 (100.00%)
 stage execs : 136/256 (53.12%)        new edges on : 3 (100.00%)
 total execs : 29.8k                  total crashes : 242 (2 unique)
  exec speed : 1729/sec                total tmouts : 0 (0 unique)
 fuzzing strategy yields  path geometry
   bit flips : 0/176, 0/173, 0/167                     levels : 1
  byte flips : 0/22, 0/19, 0/13                      pending : 0
 arithmetics : 0/1228, 0/148, 0/0                   pend fav : 0
  known ints : 0/118, 0/532, 0/572                 own finds : 0
  dictionary : 0/0, 0/0, 0/24                       imported : n/a
       havoc : 2/13.6k, 0/12.9k                    stability : 100.00%
        trim : 14.29%/4, 0.00%
^C             [cpu:313%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
lazenca0x0@ubuntu:~/Documents/AFL/test$
```

Black-box testing

Install library files

- **Black box 테스트를 진행하기 위해 다음과 같은 설정이 필요합니다.**
    - **설치 라이브러리 :** libini-config-dev, libtool-bin, automake, bison, libglib2.0-dev, qemu

Install library files

```
lazenca0x0@ubuntu:~/Documents/AFL/afl-2.49b$ apt-get install libini-config-dev libtool-bin automake bison
libglib2.0-dev qemu -y
lazenca0x0@ubuntu:~/Documents/AFL/afl-2.49b$ cd qemu_mode/
lazenca0x0@ubuntu:~/Documents/AFL/afl-2.49b/qemu_mode/$ ./build_qemu_support.sh
lazenca0x0@ubuntu:~/Documents/AFL/afl-2.49b/qemu_mode/$ cd ..
lazenca0x0@ubuntu:~/Documents/AFL/afl-2.49b$ sudo make install
```

Build using gcc

- **다음과 같이 gcc를 이용해 빌드합니다.**

Build using gcc

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ gcc -fno-stack-protector -o test test.c
```

Run afl-fuzz

- **다음과 같이 Black box test를 진행할 수 있습니다.**
    - Black box test를 진행하기 위해 -Q 옵션만 추가하면 됩니다.
    - White box test와 같이 2개의 uniq crashes를 발견하였습니다.

Run AFL-fuzz

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ afl-fuzz -Q -i testcase/ -o result/ ./test
afl-fuzz 2.49b by <lcamtuf@google.com>
[+] You have 1 CPU core and 3 runnable tasks (utilization: 300%).
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'testcase/'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:test1.txt'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 4, map size = 33, exec speed = 1898 us
[*] Attempting dry run with 'id:000001,orig:test2.txt'...
    len = 6, map size = 33, exec speed = 1048 us
[!] WARNING: No new instrumentation output, test case may be useless.
[*] Attempting dry run with 'id:000002,orig:test3.txt'...
    len = 9, map size = 36, exec speed = 790 us
[*] Attempting dry run with 'id:000003,orig:test4.txt'...
    len = 6, map size = 33, exec speed = 806 us
[!] WARNING: No new instrumentation output, test case may be useless.
[+] All test cases processed.

[!] WARNING: Some test cases look useless. Consider using a smaller set.
[+] Here are some useful stats:

    Test case count : 2 favored, 0 variable, 4 total
       Bitmap range : 33 to 36 bits (average: 33.75 bits)
        Exec timing : 790 to 1898 us (average: 1135 us)

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

                        american fuzzy lop 2.49b (test)

 process timing  overall results
        run time : 0 days, 0 hrs, 0 min, 10 sec         cycles done : 5
   last new path : none yet (odd, check syntax!)        total paths : 4
 last uniq crash : 0 days, 0 hrs, 0 min, 2 sec         uniq crashes : 2
  last uniq hang : none seen yet                         uniq hangs : 0
 cycle progress  map coverage
  now processing : 1* (25.00%)           map density : 0.05% / 0.06%
 paths timed out : 0 (0.00%)          count coverage : 1.00 bits/tuple
 stage progress  findings in depth
  now trying : splice 7               favored paths : 2 (50.00%)
 stage execs : 30/32 (93.75%)          new edges on : 2 (50.00%)
 total execs : 16.2k                  total crashes : 1204 (2 unique)
  exec speed : 1533/sec                total tmouts : 0 (0 unique)
 fuzzing strategy yields  path geometry
   bit flips : 0/128, 0/124, 0/116                     levels : 1
  byte flips : 0/16, 0/12, 0/4                       pending : 0
 arithmetics : 0/890, 0/176, 0/0                    pend fav : 0
  known ints : 0/80, 0/336, 0/176                  own finds : 0
  dictionary : 0/0, 0/0, 0/2                        imported : n/a
       havoc : 1/7936, 1/6184                      stability : 100.00%
        trim : 42.86%/4, 0.00%
             [cpu:303%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

lazenca0x0@ubuntu:~/Documents/AFL/test$
```

Check for the crash.

- 다음과 같이 발견된 uniq crashes는 result 폴더에 저장되어 있습니다.
    - 해당 파일을 이용해 crash를 재현 할 수 있습니다.

Check for the crash.

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ ls -al result/crashes/
total 20
drwx------ 2 lazenca0x0 lazenca0x0 4096 Aug  9 01:26 .
drwxrwxr-x 5 lazenca0x0 lazenca0x0 4096 Aug  9 01:26 ..
-rw------- 1 lazenca0x0 lazenca0x0   68 Aug  9 01:26 id:000000,sig:11,src:000000,op:havoc,rep:128
-rw------- 1 lazenca0x0 lazenca0x0   86 Aug  9 01:26 id:000001,sig:11,src:000002+000003,op:splice,rep:128
-rw------- 1 lazenca0x0 lazenca0x0  604 Aug  9 01:26 README.txt
lazenca0x0@ubuntu:~/Documents/AFL/test$ ./test < result/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,
rep\:128
Login : Password : Fail.
Segmentation fault
lazenca0x0@ubuntu:~/Documents/AFL/test$ ./test < result/crashes/id\:000001\,sig\:11\,src\:000002+000003\,op\:
splice\,rep\:128
Login : Password : Fail.
Segmentation fault
lazenca0x0@ubuntu:~/Documents/AFL/test$
```

- 다음은 생성 crash 파일의 내용입니다.
    - 저장된 내용은 특별한 의미를 가지지 않습니다.

Crash file

```
lazenca0x0@ubuntu:~/Documents/AFL/test$ hexdump result/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,
rep\:128
0000000 81b9 ad13 0000 76e1 04ff 007f eee7 ffff
0000010 64ff 0000 798a 9379 7980 7979 7966 e100
0000020 ff76 7fc0 e700 ffee ffff ffff 7f04 e700
0000030 ffee ffff 0064 6900 7979 7993 7979 0079
0000040 0100 ff00
0000044
lazenca0x0@ubuntu:~/Documents/AFL/test$ hexdump result/crashes/id\:000001\,sig\:11\,src\:000002+000003\,op\:
splice\,rep\:128
0000000 6f72 746f 0000 0004 5774 aaaa aaaa aaaa
0000010 aa97 aaaa 0000 8000 5774 aaaa 97a4 aaaa
0000020 00aa 0000 7480 aa57 9faa 72aa 6f6f aa74
0000030 aaaa 97a4 aaaa 16aa aaaa 619c aa57 aaaa
0000040 aaaa 97aa aaaa 00aa 0000 aa80 6f6f aaaa
0000050 72aa 6f6f 6f74
0000056
```

Related information

- http://lcamtuf.coredump.cx/afl/README.txt
- http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt
- http://lcamtuf.coredump.cx/afl/technical_details.txt
- https://lcamtuf.blogspot.jp/2014/10/fuzzing-binaries-without-execve.html
- https://lcamtuf.blogspot.jp/2016/02/say-hello-to-afl-analyze.html
- https://raw.githubusercontent.com/mirrorer/afl/master/docs/parallel_fuzzing.txt