

《软件安全》实验报告

姓名：王昱

学号：2212046

班级：信息安全班

一、实验名称：

AFL 模糊测试实验

二、实验要求：

根据课本 7.4.5 章节，复现 AFL 在 KALI 下的安装、应用查阅资料理解覆盖引导和文件变异的概念和含义。

三、实验原理：

AFL 是一款基于覆盖引导（Coverage-guided）的模糊测试工具，它通过记录输入样本的代码覆盖率，从而调整输入样本以提高覆盖率，增加发现漏洞的概率。AFL 主要用于 C/C++ 程序的测试，被测程序有无程序源码均可，有源码时可以对源码进行编译时插桩，无源码可以借助 QEMU 的 User-Mode 模式进行二进制插装。

其工作流程大致如下：

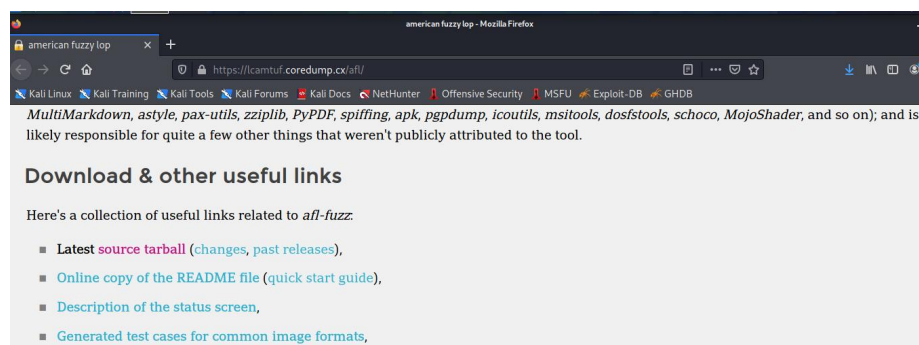
- （1）从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）；
- （2）选择一些输入文件，作为初始测试集加入输入队列（queue）；
- （3）将队列中的文件按一定的策略进行“突变”；
- （4）如果经过变异文件更新了覆盖范围，则将其保留添加到队列中；
- （5）上述过程会一直循环进行，期间触发了 crash 的文件会被记录下来。

四、实验过程：

（一）安装 AFL

由于在 Kali 2021 下使用 `sudo apt-get install afl` 会报错，所以采用了直接从官网下载安装包的方法来进行下载。

1. 打开 <https://lcamtuf.coredump.cx/afl/> 并安装压缩包



2. 解压后在目录中打开终端输入

make

sudo make install

```
(kali@kali)-[~/Downloads/afl-2.52b]
└─$ make
[*] Checking for the ability to compile x86 code...
[*] Everything seems to be working, ready to compile.
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-gcc.c -o afl-gcc -ldl
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc $i; done
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-fuzz.c -o afl-fuzz -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-showmap.c -o afl-showmap -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-tmin.c -o afl-tmin -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-gotcpu.c -o afl-gotcpu -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-analyze.c -o afl-analyze -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" \
  afl-as.c -o afl-as -ldl
ln -sf afl-as as
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" test-instr.c -o test-instr -ldl
echo 0 | ./afl-showmap -m none -q -o .test-instr0 ./test-instr
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[*] All right, the instrumentation seems to be working!
[*] LLVM users: see llvm_mode/README.llvm for a faster alternative to afl-gcc.
[*] All done! Be sure to review README - it's pretty short and useful.
```

```
(kali@kali)-[~/Downloads/afl-2.52b]
└─$ sudo make install
[sudo] password for kali:
[*] Checking for the ability to compile x86 code...
[*] Everything seems to be working, ready to compile.
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" test-instr.c -o test-instr -ldl
echo 0 | ./afl-showmap -m none -q -o .test-instr0 ./test-instr
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[*] All right, the instrumentation seems to be working!
[*] LLVM users: see llvm_mode/README.llvm for a faster alternative to afl-gcc.
[*] All done! Be sure to review README - it's pretty short and useful.

mkdir -p -m 755 ${DESTDIR}/usr/local/bin ${DESTDIR}/usr/local/lib/afl ${DESTDIR}/usr/local/share/doc/afl ${DESTDIR}/usr/local/share/afl
rm -f ${DESTDIR}/usr/local/bin/afl-plot.sh
install -m 755 afl-gcc afl-fuzz afl-showmap afl-tmin afl-gotcpu afl-analyze afl-plot afl-cmin afl-whatsup ${DESTDIR}/usr/local/bin
rm -f ${DESTDIR}/usr/local/bin/afl-as
if [ -f afl-qemu-trace ]; then install -m 755 afl-qemu-trace ${DESTDIR}/usr/local/bin; fi
if [ -f afl-clang-fast -a -f afl-llvm-pass.so -a -f afl-llvm-rt.o ]; then set -e; install -m 755 afl-clang-fast ${DESTDIR}/usr/local/bin; ln -sf afl-clang-fast ${DESTDIR}/usr/local/bin/afl-clang-fast; install -m 755 afl-llvm-pass.so afl-llvm-rt.o ${DESTDIR}/usr/local/lib/afl; fi
if [ -f afl-llvm-rt-32.o ]; then set -e; install -m 755 afl-llvm-rt-32.o ${DESTDIR}/usr/local/lib/afl; fi
if [ -f afl-llvm-rt-64.o ]; then set -e; install -m 755 afl-llvm-rt-64.o ${DESTDIR}/usr/local/lib/afl; fi
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc ${DESTDIR}/usr/local/bin/$i; done
install -m 755 afl-as ${DESTDIR}/usr/local/lib/afl
ln -sf afl-as ${DESTDIR}/usr/local/lib/afl/as
install -m 644 docs/README docs/ChangeLog docs/*.txt ${DESTDIR}/usr/local/share/doc/afl
cp -r testcases/ ${DESTDIR}/usr/local/share/afl
cp -r dictionaries/ ${DESTDIR}/usr/local/share/afl
```

3.用 ls 查看安装是否成功

可以看到文件安装成功

```
(kali@kali)-[~/Downloads/afl-2.52b]
└─$ ls
afl-analyze  afl-as.h      afl-fuzz      afl-gcc.c      afl-showmap    afl-whatsup  debug.h      hash.h      Makefile      testcases
afl-analyze.c  afl-clang    afl-fuzz.c    afl-gotcpu     afl-showmap.c  alloc-inl.h  dictionaries libdislocator qemu_mode    test-instr.c
afl-as         afl-clang++  afl-g++       afl-gotcpu.c   afl-tmin       as           docs         libtokencap  QuickStartGuide.txt types.h
afl-as.c       afl-cmin     afl-gcc       afl-plot       afl-tmin.c     config.h     experimental llvm_mode    README
```

(二) AFL 测试

1.创建本次实验的程序

(1) 编写创建测试代码，生成 test.c 文件

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char ptr[20];
    if(argc>1){
        FILE *fp = fopen(argv[1], "r");
        fgets(ptr, sizeof(ptr), fp);
    }
    else{
        fgets(ptr, sizeof(ptr), stdin);
    }
    printf("%s", ptr);
    if(ptr[0] == 'd') {
        if(ptr[1] == 'e') {
            if(ptr[2] == 'a') {
                if(ptr[3] == 'd') {
                    if(ptr[4] == 'b') {
                        if(ptr[5] == 'e') {
                            if(ptr[6] == 'e') {
                                if(ptr[7] == 'f') { ptr[1] = '\0';
                                    abort();
                                }
                                else printf("%c",ptr[7]);
                            }
                            else printf("%c",ptr[6]);
                        }
                        else printf("%c",ptr[5]);
                    }
                    else printf("%c",ptr[4]);
                }
                else printf("%c",ptr[3]);
            }
            else printf("%c",ptr[2]);
        }
        else printf("%c",ptr[1]);
    }
    else printf("%c",ptr[0]);
}

```

可以看到到程序中有一个 `abort()` 函数，当程序输入“deadbeef”字符串则会调用这个函数，如果传入其他字符会原样输出。

(2) afl 编译.c 文件

使用如下命令编译：

命令：`afl-gcc -o test test.c`

- `afl-gcc` 是对 `gcc` 的封装，包含插桩功能，可以在程序编译中插入便于事后分析的代码。

```

(kali@kali)-[~/demo1]
└─$ afl-gcc -o test test.c
afl-cc 2.52b by <lcantuf@google.com>
afl-as 2.52b by <lcantuf@google.com>
[+] Instrumented 14 locations (64-bit, non-hardened mode, ratio 100%).

```

- 编译后会有插桩符号，使用下面的命令验证：

命令：`readelf -s ./test | grep afl`

```
(kali㉿kali)-[~/demo1]
$ readelf -s ./test | grep afl
35: 00000000000001628      0 NOTYPE  LOCAL  DEFAULT 14  __afl_maybe_log
37: 000000000000040b0      8 OBJECT  LOCAL  DEFAULT 25  __afl_area_ptr
38: 00000000000001658      0 NOTYPE  LOCAL  DEFAULT 14  __afl_setup
39: 00000000000001638      0 NOTYPE  LOCAL  DEFAULT 14  __afl_store
40: 000000000000040b8      8 OBJECT  LOCAL  DEFAULT 25  __afl_prev_loc
41: 00000000000001650      0 NOTYPE  LOCAL  DEFAULT 14  __afl_return
42: 000000000000040c8      1 OBJECT  LOCAL  DEFAULT 25  __afl_setup_failure
43: 00000000000001679      0 NOTYPE  LOCAL  DEFAULT 14  __afl_setup_first
45: 0000000000000193e      0 NOTYPE  LOCAL  DEFAULT 14  __afl_setup_abort
46: 00000000000001793      0 NOTYPE  LOCAL  DEFAULT 14  __afl_forkserver
47: 000000000000040c4      4 OBJECT  LOCAL  DEFAULT 25  __afl_temp
48: 00000000000001851      0 NOTYPE  LOCAL  DEFAULT 14  __afl_fork_resume
49: 000000000000017b9      0 NOTYPE  LOCAL  DEFAULT 14  __afl_fork_wait_loop
50: 00000000000001936      0 NOTYPE  LOCAL  DEFAULT 14  __afl_die
51: 000000000000040c0      4 OBJECT  LOCAL  DEFAULT 25  __afl_fork_pid
98: 000000000000040d0      8 OBJECT  GLOBAL  DEFAULT 25  __afl_global_area_ptr
```

(3) 指示系统将 coredumps 输出为文件

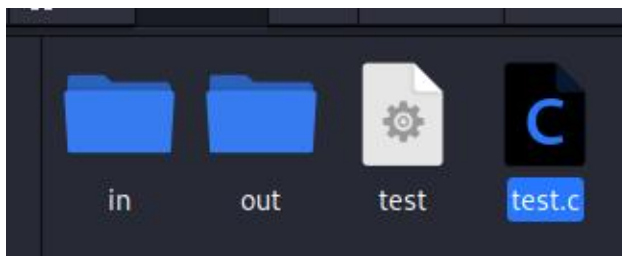
echo core > /proc/sys/kernel/core_pattern

```
(kali㉿kali)-[~/demo1]
$ echo core > /proc/sys/kernel/core_pattern
zsh: permission denied: /proc/sys/kernel/core_pattern
```

2. 创建测试用例

(1) 创建两个文件夹 in 和 out 存储模糊测试所需的输入和输出相关的内容

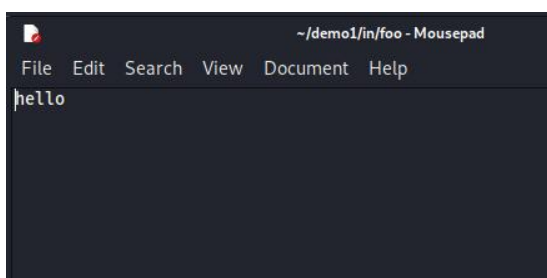
命令: mkdir in out



(2) 输入文件夹中创建一个包含字符串“hello”的文件

命令: echo hello> in/foo

- foo 就是我们的测试用例，里面包含初步字符串 hello。AFL 会通过这个语料进行变异，构造更多的测试用例。



3.启动模糊测试

- 运行如下命令，开始启动模糊测试：

命令：afl-fuzz -i in -o out -- ./test @@

- 模糊测试界面如下：

如图已经找到了一个 crash：

```
american fuzzy lop 2.52b (test)

process timing
  run time : 0 days, 0 hrs, 4 min, 28 sec
  last new path : 0 days, 0 hrs, 3 min, 58 sec
  last uniq crash : 0 days, 0 hrs, 3 min, 50 sec
  last uniq hang : none seen yet

cycle progress
  now processing : 4 (50.00%)
  paths timed out : 0 (0.00%)
  stage progress
    now trying : havoc
    stage execs : 74/384 (19.27%)
    total execs : 1.00M
    exec speed : 3788/sec
  fuzzing strategy yields
    bit flips : 2/352, 1/344, 1/328
    byte flips : 0/44, 0/36, 0/20
    arithmetics : 1/2464, 0/120, 0/0
    known ints : 0/242, 1/976, 0/879
    dictionary : 0/0, 0/0, 0/0
    havoc : 2/495k, 0/500k
    trim : 39.13%/9, 0.00%

map coverage
  map density : 0.01% / 0.03%
  count coverage : 1.00 bits/tuple

findings in depth
  favored paths : 8 (100.00%)
  new edges on : 8 (100.00%)
  total crashes : 1 (1 unique)
  total tmoouts : 0 (0 unique)

path geometry
  levels : 6
  pending : 0
  pend fav : 0
  own finds : 7
  imported : n/a
  stability : 100.00%

overall results
  cycles done : 151
  total paths : 8
  uniq crashes : 1
  uniq hangs : 0

[cpu:202%]
```

- process timing

这里展示了当前 fuzzer 的运行时间、最近一次发现新执行路径的时间、最近一次崩溃的时间、最近一次超时的时间。

- overall results

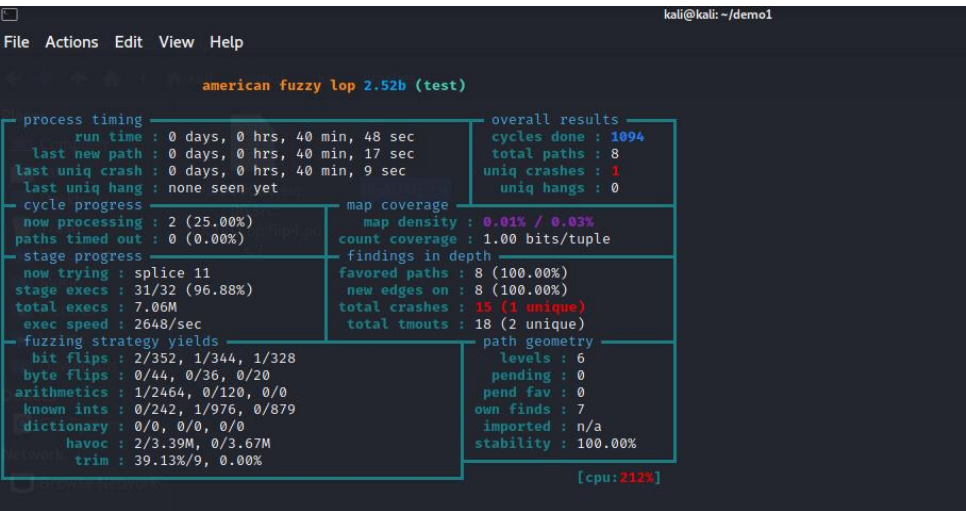
这里包括运行的总周期数、总路径数、崩溃次数、超时次数。其中，总周期数可以用来作为何时停止 fuzzing 的参考。随着不断地 fuzzing，周期数会不断增大，其颜色也会由洋红色，逐步变为黄色、蓝色、绿色。一般来说，当其变为绿色时，代表可执行的内容已经很少了，继续 fuzzing 下去也不会有什么新的发现了。此时，我们便可以通过 Ctrl-C，中止当前的 fuzzing。

- stage progress

这里包括正在测试的 fuzzing 策略、进度、目标的执行总次数、目标的执行速度。执行速度可以直观地反映当前跑的快不快，如果速度过慢，比如低于 500 次每秒，那么测试时间就会变得非常漫长。如果发生了这种情况，那么我们需要进一步调整优化我们的 fuzzing。

- 运行一段时间后：

如图所示 overall results 的周期数已经变为蓝色



```
kali@kali: ~/demo1
File Actions Edit View Help

american fuzzy lop 2.52b (test)

process timing
  run time : 0 days, 0 hrs, 40 min, 48 sec
  last new path : 0 days, 0 hrs, 40 min, 17 sec
  last uniq crash : 0 days, 0 hrs, 40 min, 9 sec
  last uniq hang : none seen yet

cycle progress
  now processing : 2 (25.00%)
  paths timed out : 0 (0.00%)

stage progress
  now trying : splice 11
  stage execs : 31/32 (96.88%)
  total execs : 7.06M
  exec speed : 2648/sec

fuzzing strategy yields
  bit flips : 2/352, 1/344, 1/328
  byte flips : 0/44, 0/36, 0/20
  arithmetics : 1/2464, 0/120, 0/0
  known ints : 0/242, 1/976, 0/879
  dictionary : 0/0, 0/0, 0/0
  havoc : 2/3.39M, 0/3.67M
  trim : 39.13%/9, 0.00%

overall results
  cycles done : 1094
  total paths : 8
  uniq crashes : 1
  uniq hangs : 0

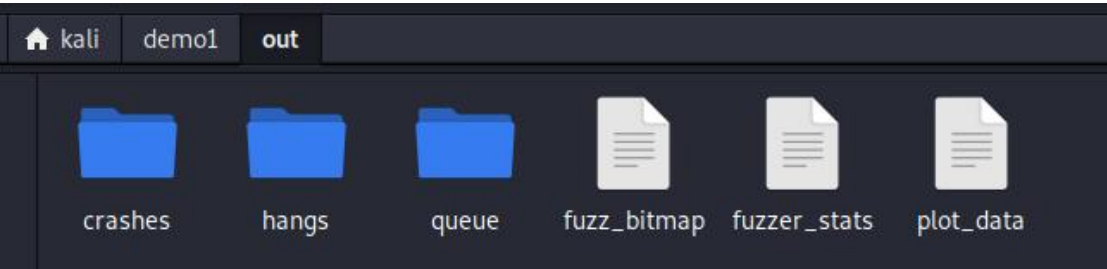
map coverage
  map density : 0.01% / 0.03%
  count coverage : 1.00 bits/tuple

findings in depth
  favored paths : 8 (100.00%)
  new edges on : 8 (100.00%)
  total crashes : 15 (1 unique)
  total tmouts : 18 (2 unique)

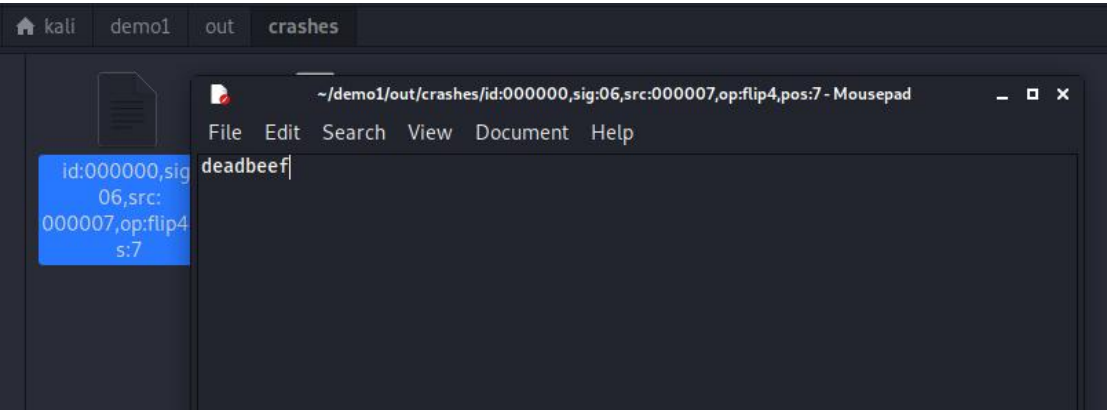
path geometry
  levels : 6
  pending : 0
  pend fav : 0
  own finds : 7
  imported : n/a
  stability : 100.00%

[cpu:212%]
```

4.分析 crash



打开产生的 crash 样例，为 “deadbeef”，说明触发到了 abort() 函数。



得到 crash 样例后，可以将这些样例作为目标测试程序的输入，重新触发异常并跟踪运行状态，进行分析、定位程序出错的原因或确认存在的漏洞类型。

(三) 覆盖引导和文件变异的概念和含义

1.覆盖引导

覆盖率是度量测试完整性的一个手段，是测试有效性的一个度量。通过已执行代码表示，用于可靠性、稳定性以及性能的评测。测试覆盖是对测试完全程度的评测。测试覆盖是由测试需求和测试用例的覆盖或已执行代码的覆盖表示的。建立在对测试结果的

评估和对测试过程中确定的变更请求（缺陷）的分析的基础上。

在 AFL 的 fuzzing 过程中，维护了一个 testcase 队列 queue，每次把队列里的文件取出来之后，对其进行变异，如果这个变异将覆盖率提高了，则将这个文件放入队列中。而通过记录输入样本的代码覆盖率，从而调整输入样本以提高覆盖率的变异方向就是覆盖引导

2.文件变异

AFL 维护了一个队列(queue)，每次从这个队列中取出一个文件，对其进行大量变异，并检查运行后是否会引起目标崩溃、发现新路径等结果。变异的主要类型如下：

- bitflip，按位翻转，1 变为 0，0 变为 1
- arithmetic，整数加/减算术运算
- interest，把一些特殊内容替换到原文件中
- dictionary，把自动生成或用户提供的 token 替换/插入到原文件中
- havoc，中文意思是“大破坏”，此阶段会对原文件进行大量变异
- splice，中文意思是“绞接”，此阶段会将两个文件拼接起来得到一个新的文件

五、心得体会：

通过本次实验，了解了 AFL 框架的用法以及覆盖引导和文件变异的含义，明白了模糊测试的概念和意义。