

《漏洞利用及渗透测试基础》实验报告

姓名：王晶 学号：2310420 班级：信息安全法学双学位

实验名称

AFL 模糊测试

实验要求

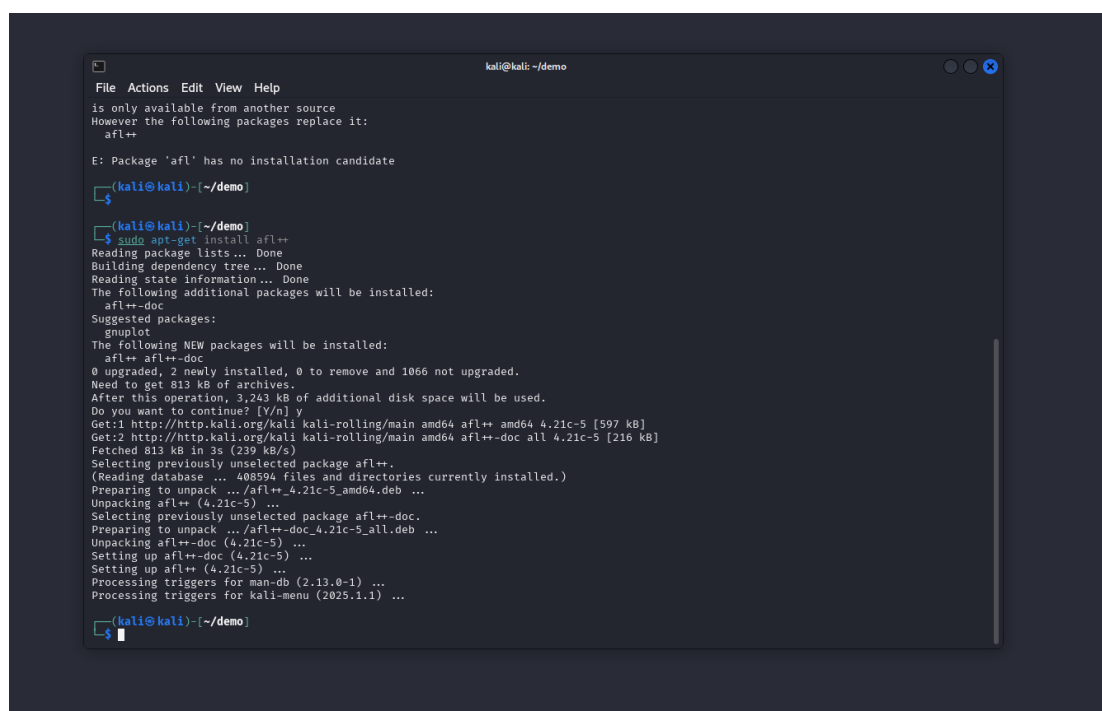
根据课本 7.4.5 章节，复现 AFL 在 KALI 下的安装、应用，查阅资料理解覆盖引导和文件变异的概念和含义。

实验过程

(1) 安装 AFL

首先需要下载 Kali 虚拟机, 在官网地址下载好之后, 在 VMware 内打开即可, 在其中创建一个文件 demo, 启动控制台, 打开终端

输入 `install afl` 命令, 即可下载安装 afl 安装包



```
kali@kali: ~/demo
File Actions Edit View Help
is only available from another source
However the following packages replace it:
afl++
E: Package 'afl' has no installation candidate

(kali@kali)~/demo
$ sudo apt-get install afl++
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  afl++-doc
Suggested packages:
  gnuplot
The following NEW packages will be installed:
  afl++ afl++-doc
0 upgraded, 2 newly installed, 0 to remove and 1066 not upgraded.
Need to get 813 kB of archives.
After this operation, 3,243 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://http.kali.org/kali kali-rolling/main amd64 afl++ amd64 4.21c-5 [597 kB]
Get:2 http://http.kali.org/kali kali-rolling/main amd64 afl++-doc all 4.21c-5 [216 kB]
Fetched 813 kB in 3s (239 kB/s)
Selecting previously unselected package afl++.
(Reading database ... 408594 files and directories currently installed.)
Preparing to unpack .../afl++_4.21c-5_amd64.deb ...
Unpacking afl++ (4.21c-5) ...
Selecting previously unselected package afl++-doc.
Preparing to unpack .../afl++-doc_4.21c-5_all.deb ...
Unpacking afl++-doc (4.21c-5) ...
Setting up afl++-doc (4.21c-5) ...
Setting up afl++ (4.21c-5) ...
Processing triggers for man-db (2.13.0-1) ...
Processing triggers for kali-menu (2025.1.1) ...

(kali@kali)~/demo
$
```

使用 `ls` 命令, 查看 afl 下的一些文件, 即可查看是否安装成功

```
Processing triggers for man-db (2.13.0-1) ...
Processing triggers for kali-menu (2025.1.1) ...

(kali@kali)~[/demo]
$ ls /usr/bin/afl*
/usr/bin/afl-addseeds /usr/bin/afl-clang-fast /usr/bin/afl-fuzz /usr/bin/afl-ld-lto /usr/bin/afl-plot
/usr/bin/afl-analyze /usr/bin/afl-clang-fast++ /usr/bin/afl-g++ /usr/bin/afl-lto /usr/bin/afl-showmap
/usr/bin/afl-c++ /usr/bin/afl-clang-lto /usr/bin/afl-gcc /usr/bin/afl-lto++ /usr/bin/afl-system-config
/usr/bin/afl-ccc /usr/bin/afl-clang-lto++ /usr/bin/afl-gcc-fast /usr/bin/afl-network-client /usr/bin/afl-tmin
/usr/bin/afl-clang /usr/bin/afl-cmin /usr/bin/afl-g++-fast /usr/bin/afl-network-server /usr/bin/afl-whatsup
/usr/bin/afl-clang++ /usr/bin/afl-cmin.bash /usr/bin/afl-gotcpu /usr/bin/afl-persistent-config

(kali@kali)~[/demo]
$
```

(2) AFL 测试应用

使用安装好的 AFL 文件, 来复现课本上的模糊测试案例

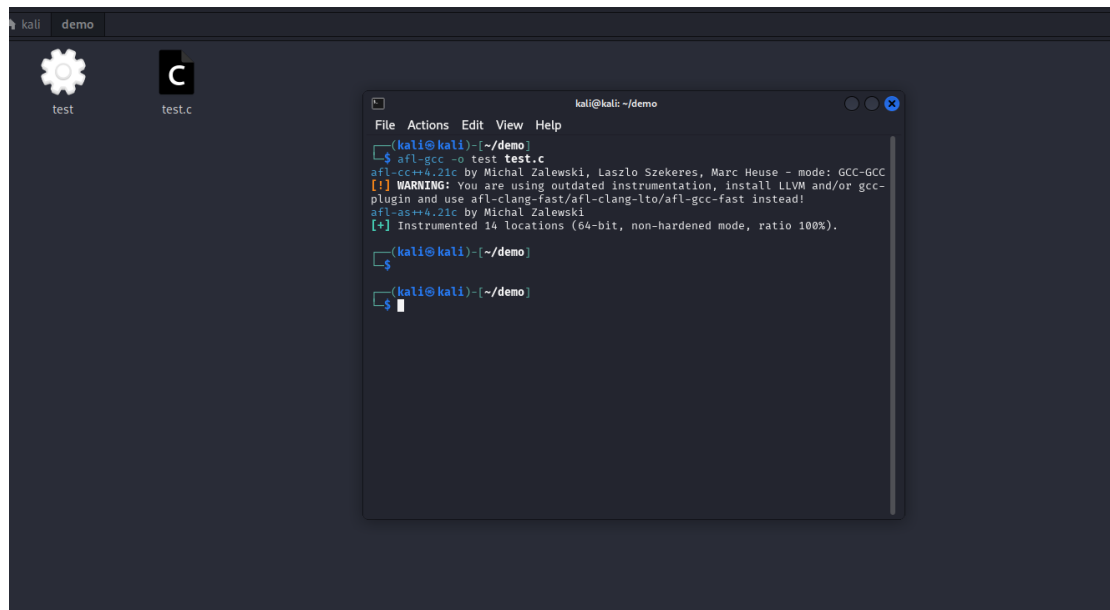
新建一个测试文件 test.c, 输入课本上的代码

```
#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') {
                                    abort();
                                } else {
                                    printf("%c", ptr[7]);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

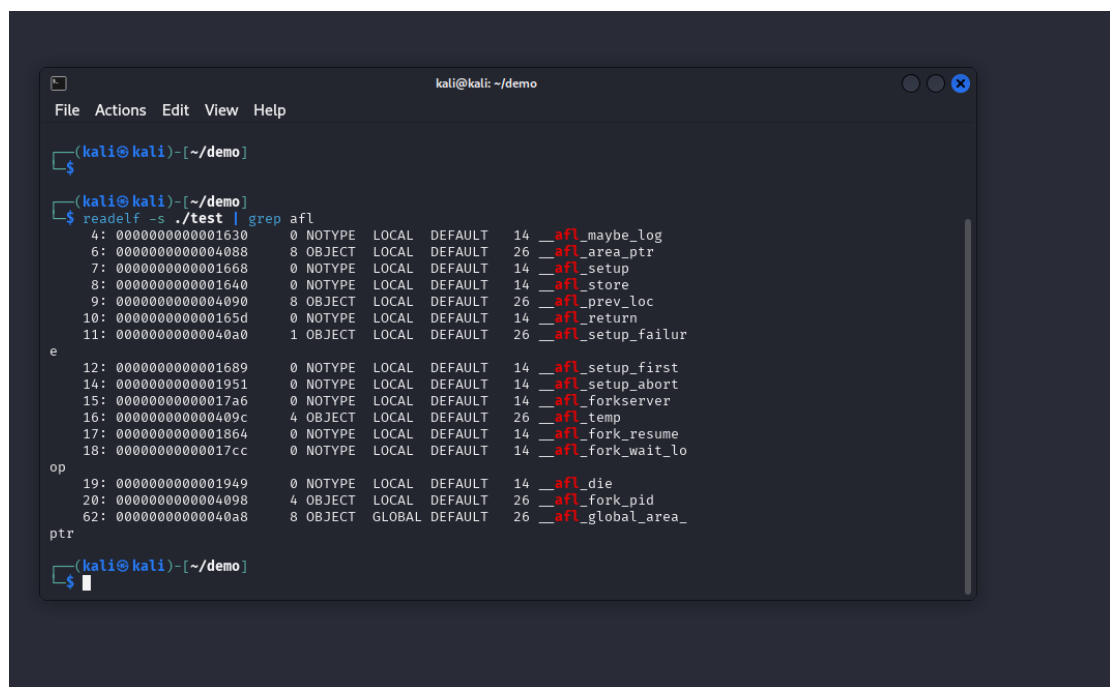
```
        } else {
            printf("%c", ptr[6]);
        }
    } else {
        printf("%c", ptr[5]);
    }
} else {
    printf("%c", ptr[4]);
}
} else {
    printf("%c", ptr[3]);
}
} else {
    printf("%c", ptr[2]);
}
} else {
    printf("%c", ptr[1]);
}
} else {
    printf("%c", ptr[0]);
}
return 0;
}
```

对 C 文件进行编译, 输入命令为 `afl-gcc -o test test.c`

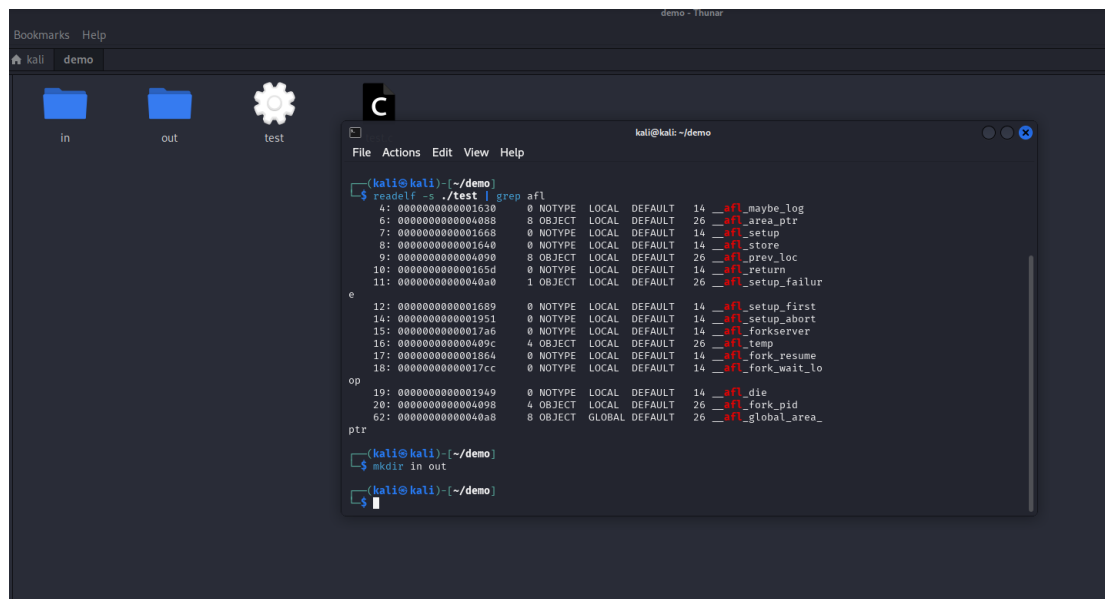


产生目标 test 可执行文件

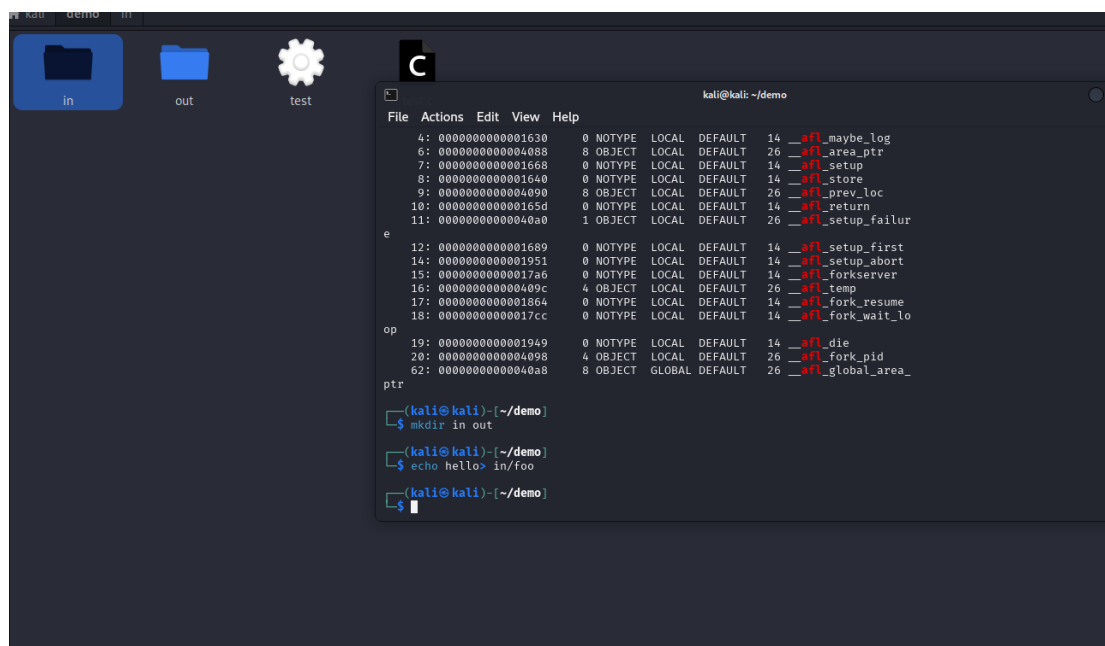
可以测试插桩符号, 输入 `readelf -s ./test | grep afl`



之后创建输入输出文件 `mkdir in out`



在输入文件夹内, 创建一个 foo 文件夹, 里面包含"hello"字符串 echo hello>
in/foo



之后启动测试

```
american fuzzy lop ++4.21c {default} (./test) [explore]
process timing
  run time : 0 days, 0 hrs, 0 min, 14 sec
  last new find : 0 days, 0 hrs, 0 min, 13 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress
  now processing : 0.21 (0.0%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 24/37 (64.86%)
  total execs : 4660
  exec speed : 342.2/sec
fuzzing strategy yields
  bit flips : 0/0, 0/0, 0/0
  byte flips : 0/0, 0/0, 0/0
  arithmetics : 0/0, 0/0, 0/0
  known ints : 0/0, 0/0, 0/0
  dictionary : 0/0, 0/0, 0/0, 0/0
  havoc/splice : 2/4590, 0/0
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 70.00%/9, n/a
strategy: explore state: started :-)

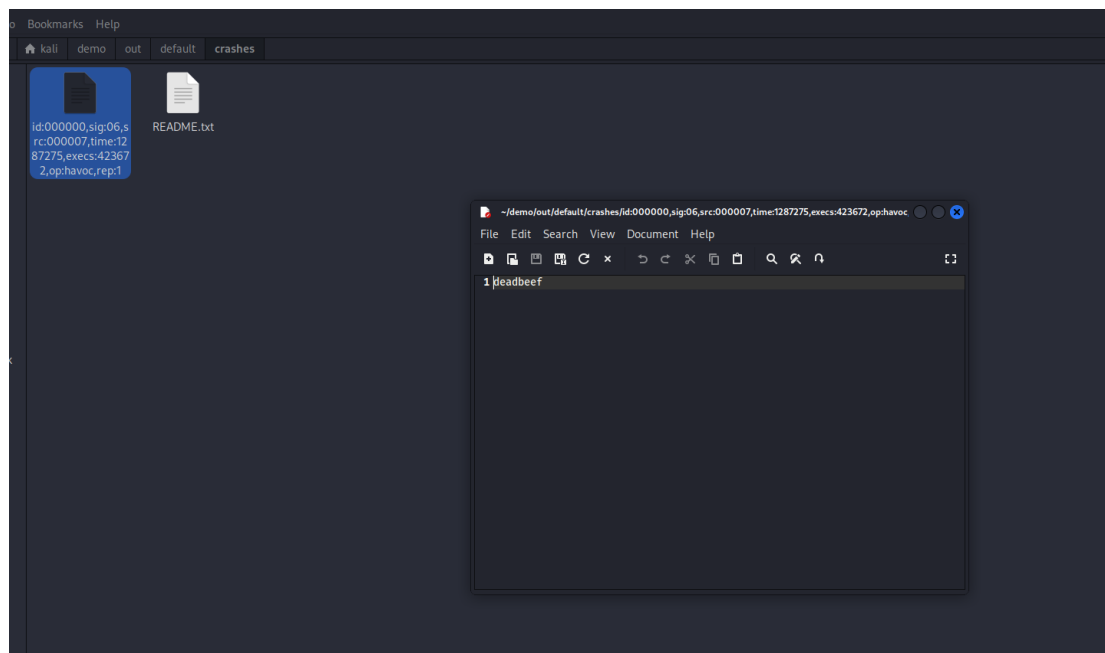
overall results
  cycles done : 13
  corpus count : 3
  saved crashes : 0
  saved hangs : 0
map coverage
  map density : 0.00% / 0.00%
  count coverage : 1.00 bits/tuple
findings in depth
  favored items : 3 (100.00%)
  new edges on : 3 (100.00%)
  total crashes : 0 (0 saved)
  total tmouts : 0 (0 saved)
item geometry
  levels : 3
  pending : 0
  pend fav : 0
  own finds : 2
  imported : 0
  stability : 100.00%
[cpu000: 75%]
```

发现一个 crash 样例

```
american fuzzy lop ++4.21c {default} (./test) [explore]
process timing
  run time : 0 days, 0 hrs, 25 min, 47 sec
  last new find : 0 days, 0 hrs, 6 min, 20 sec
  last saved crash : 0 days, 0 hrs, 4 min, 20 sec
  last saved hang : none seen yet
cycle progress
  now processing : 6.23 (75.0%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 504/600 (84.00%)
  total execs : 509k
  exec speed : 305.7/sec
fuzzing strategy yields
  bit flips : 0/0, 0/0, 0/0
  byte flips : 0/0, 0/0, 0/0
  arithmetics : 0/0, 0/0, 0/0
  known ints : 0/0, 0/0, 0/0
  dictionary : 0/0, 0/0, 0/0, 0/0
  havoc/splice : 8/443k, 0/65.1k
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 98.67%/31, n/a
strategy: explore state: in progress

overall results
  cycles done : 298
  corpus count : 8
  saved crashes : 1
  saved hangs : 0
map coverage
  map density : 0.00% / 0.00%
  count coverage : 1.00 bits/tuple
findings in depth
  favored items : 8 (100.00%)
  new edges on : 8 (100.00%)
  total crashes : 3 (1 saved)
  total tmouts : 0 (0 saved)
item geometry
  levels : 7
  pending : 0
  pend fav : 0
  own finds : 7
  imported : 0
  stability : 100.00%
[cpu000: 50%]
```

打开 out 文件夹内的 crash 文件, 打开后发现文件内为 crash 的样例输入, 和预期相符合



模糊测试成功！

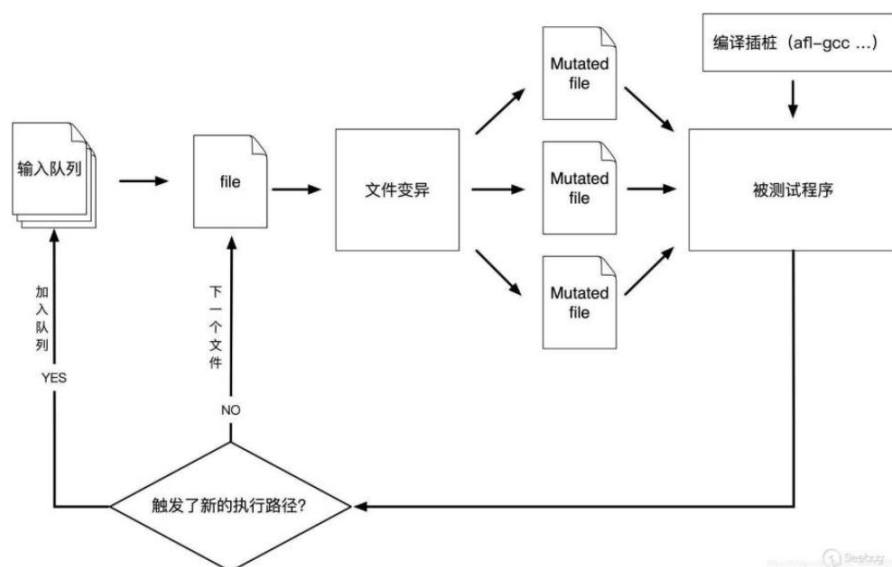
心得体会

(1) 基础知识

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。



(2) 覆盖引导

覆盖引导，即通过向目标程序插桩，统计代码覆盖，反馈给模糊测试引擎（fuzzer，即模糊测试工具），反馈信息用于变异种子，生成更高质量的输入，使得 fuzzer 能够用更好的输入让被测程序达到更高的代码覆盖率。

对于每个目标，fuzzer 都会构建一个输入的语料库，随着 fuzzer 通过变异语料库发现新的输入，覆盖率会不断增长。

(3) 文件变异

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，变异方法如下：

1. bitflip: 按位翻转，每次都是比特位级别的操作，从 1bit 到 32bit，从文件头到文件尾，会产生一些有意思的额外重要数据信息；
2. arithmetic: 与位翻转不同的是，从 8bit 级别开始，而且每次进行的是加减操作，而不是翻转；
3. interest: 把一些有意思的东西 “interesting values” 对文件内容进行替换；
4. dictionary: 用户提供的字典里有 token，用来替换要进行变异的文件内容，如果用户没提供就使用 bitflip 自动生成的 token；
5. havoc: 进行很大程度的杂乱破坏，规则很多，基本上换完就是面目全非的新文件了；

6. splice: 通过将两个文件按一定规则进行拼接, 得到一个效果不同的新文件; bitflip、arithmetic、interest、dictionary 是 deterministic fuzzing 过程, 属于 dumb mode(-d) 和主 fuzzer(-M) 会进行的操作; havoc、splice 与前面不同是存在随机性, 是所有 fuzz 都会进行的变异操作。

文件变异是具有启发性判断的, 应注意“避免浪费, 减少消耗”的原则, 即之前变异应该尽可能产生更大的效果, 比如 eff_map 数组的设计; 同时减少不必要的资源消耗, 变异要及时止损。