

软件安全实验报告

姓名：辛浩然 学号：2112514 班级：信息安全、法学

1 实验名称

AFL 模糊测试

2 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。

3 实验过程

3.1 AFL 安装

在 Kali 下，利用 `sudo apt-get install afl` 即可安装。

查看路径可以看到 afl 安装的文件：`ls /usr/bin/afl*`

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

3.2 AFL 测试

3.2.1 创建实验程序

首先，新建文件夹 demo，并创建本次实验的程序 Test.c，该代码编译后得到的程序如果被传入“deadbeef”则会终止，如果传入其他字符会原样输出。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char **argv)
4 {
5     char ptr[20];
6     if (argc > 1)
7     {
8         FILE *fp = fopen(argv[1], "r");
9         fgets(ptr, sizeof(ptr), fp);
10    }
11    else
12    {
13        fgets(ptr, sizeof(ptr), stdin);
14    }
15    printf("%s", ptr);
16    if (ptr[0] == 'd')
17    {
18        if (ptr[1] == 'e')
19        {
20            if (ptr[2] == 'a')
21            {
22                if (ptr[3] == 'd')
23                {
24                    if (ptr[4] == 'b')
25                    {
26                        if (ptr[5] == 'e')
27                        {
28                            if (ptr[6] == 'e')
29                            {
30                                if (ptr[7] == 'f')
31                                {
32                                    abort();
33                                }
34                                else
35                                    printf("%c", ptr[7]);
36                            }
37                            else
38                                printf("%c", ptr[6]);
39                        }
40                        else
41                            printf("%c", ptr[5]);
42                    }

```

```

43         else
44             printf("%c", ptr[4]);
45     }
46     else
47         printf("%c", ptr[3]);
48     }
49     else
50         printf("%c", ptr[2]);
51     }
52     else
53         printf("%c", ptr[1]);
54 }
55 else
56     printf("%c", ptr[0]);
57 return 0;
58 }

```

The screenshot shows a window titled "~/Desktop/demo/test.c - Mousepad". The window contains a C program that reads input from a file or stdin and checks for specific byte patterns. The code is as follows:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char **argv) {
4     char ptr[20];
5     if(argc>1){
6         FILE *fp = fopen(argv[1], "r");
7         fgets(ptr, sizeof(ptr), fp);
8     }
9     else{
10        fgets(ptr, sizeof(ptr), stdin);
11    }
12    printf("%s", ptr);
13    if(ptr[0] = 'd') {
14        if(ptr[1] = 'e') {
15            if(ptr[2] = 'a') {
16                if(ptr[3] = 'd') {
17                    if(ptr[4] = 'b') {
18                        if(ptr[5] = 'e') {
19                            if(ptr[6] = 'e') {
20                                if(ptr[7] = 'f') {
21                                    abort();
22                                }
23                            else printf("%c", ptr[7]);

```

然后，运行命令 `afl-gcc -o test test.c` 使用 afl 的编译器编译，使模糊测试过程更加高效。

```
kali@kali: ~/Desktop/demo
File Actions Edit View Help
(kali@kali)~[~/Desktop/demo]
$ afl-gcc -o test test.c
afl-cc++4.04c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: GCC-GCC
[!] WARNING: You are using outdated instrumentation, install LLVM and/or gcc-
plugin and use afl-clang-fast/afl-clang-lto/afl-gcc-fast instead!
afl-as++4.04c by Michal Zalewski
[+] Instrumented 14 locations (64-bit, non-hardened mode, ratio 100%).
```

编译后会有插桩符号，使用命令 `readelf -s ./test | grep afl` 验证如下：

```
(kali@kali)~[~/Desktop/demo]
$ readelf -s ./test | grep afl
4: 0000000000001630 0 NOTYPE LOCAL DEFAULT 15 __afl_maybe_log
6: 0000000000004090 8 OBJECT LOCAL DEFAULT 26 __afl_area_ptr
7: 0000000000001668 0 NOTYPE LOCAL DEFAULT 15 __afl_setup
8: 0000000000001640 0 NOTYPE LOCAL DEFAULT 15 __afl_store
9: 0000000000004098 8 OBJECT LOCAL DEFAULT 26 __afl_prev_loc
10: 000000000000165d 0 NOTYPE LOCAL DEFAULT 15 __afl_return
11: 00000000000040a8 1 OBJECT LOCAL DEFAULT 26 __afl_setup_failur
e
12: 0000000000001689 0 NOTYPE LOCAL DEFAULT 15 __afl_setup_first
14: 0000000000001951 0 NOTYPE LOCAL DEFAULT 15 __afl_setup_abort
15: 00000000000017a6 0 NOTYPE LOCAL DEFAULT 15 __afl_forkserver
16: 00000000000040a4 4 OBJECT LOCAL DEFAULT 26 __afl_temp
17: 0000000000001864 0 NOTYPE LOCAL DEFAULT 15 __afl_fork_resume
18: 00000000000017cc 0 NOTYPE LOCAL DEFAULT 15 __afl_fork_wait_lo
op
19: 0000000000001949 0 NOTYPE LOCAL DEFAULT 15 __afl_die
20: 00000000000040a0 4 OBJECT LOCAL DEFAULT 26 __afl_fork_pid
62: 00000000000040b0 8 OBJECT GLOBAL DEFAULT 26 __afl_global_area_
ptr
```

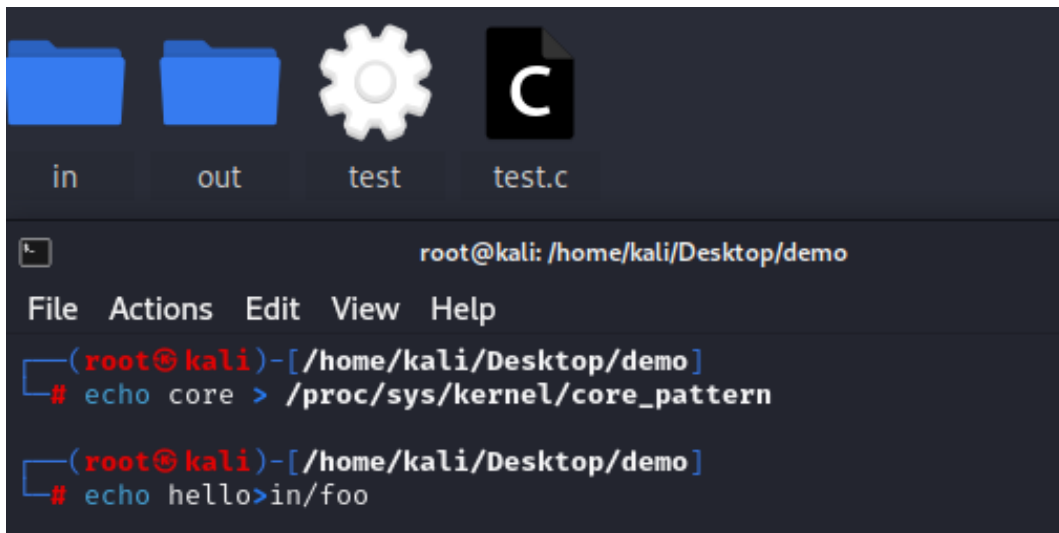
最后，输入如下命令指示系统将 coredumps 输出为文件，而不是将它们发送到特定的崩溃处理程序应用程序。

命令：`echo core > /proc/sys/kernel/core_pattern`

3.2.2 创建测试用例

首先，使用命令 `mkdir in out` 创建两个文件夹 in 和 out，分别存储模糊测试所需的输入和输出相关的内容。

然后，使用命令 `echo hello> in/foo` 在输入文件夹中创建一个包含字符串“hello”的文件。foo 就是测试用例，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。



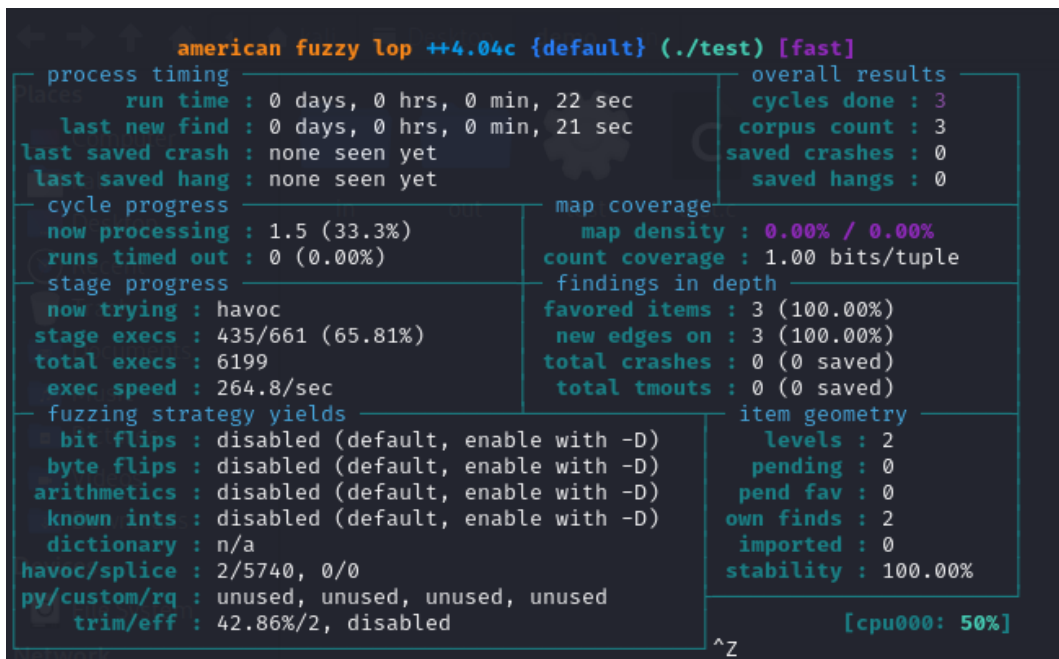
```
root@kali: /home/kali/Desktop/demo
File Actions Edit View Help
(root@kali)-[/home/kali/Desktop/demo]
# echo core > /proc/sys/kernel/core_pattern
(root@kali)-[/home/kali/Desktop/demo]
# echo hello>in/foo
```

3.2.3 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 @@

启动模糊测试后，可以看到如下运行的界面：



```
american fuzzy lop ++4.04c {default} (./test) [fast]
process timing
  run time : 0 days, 0 hrs, 0 min, 22 sec
  last new find : 0 days, 0 hrs, 0 min, 21 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress
  now processing : 1.5 (33.3%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 435/661 (65.81%)
  total execs : 6199
  exec speed : 264.8/sec
fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 2/5740, 0/0
  py/custom/rq : unused, unused, unused
  trim/eff : 42.86%/2, disabled
overall results
  cycles done : 3
  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 : 2
  pending : 0
  pend fav : 0
  own finds : 2
  imported : 0
  stability : 100.00%
[cpu000: 50%]
```

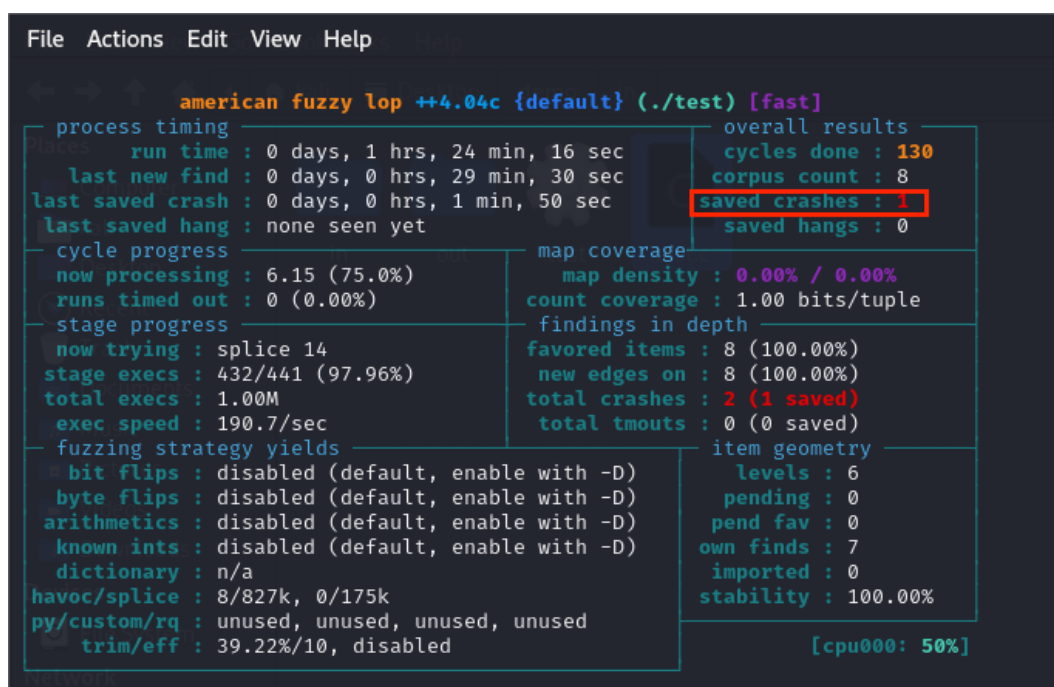
界面中：

- process timing: 展示当前 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: 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，便可以通过 Ctrl-C，中止当前的 fuzzing。
- stage progress: 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，那么需要进一步调整优化 fuzzing。

3.2.4 分析 crash

运行至产生 saved crash:



```
File Actions Edit View Help

american fuzzy lop ++4.04c {default} (./test) [fast]

process timing
  run time : 0 days, 1 hrs, 24 min, 16 sec
  last new find : 0 days, 0 hrs, 29 min, 30 sec
  last saved crash : 0 days, 0 hrs, 1 min, 50 sec
  last saved hang : none seen yet

overall results
  cycles done : 130
  corpus count : 8
  saved crashes : 1
  saved hangs : 0

cycle progress
  now processing : 6.15 (75.0%)
  runs timed out : 0 (0.00%)

map coverage
  map density : 0.00% / 0.00%
  count coverage : 1.00 bits/tuple

stage progress
  now trying : splice 14
  stage execs : 432/441 (97.96%)
  total execs : 1.00M
  exec speed : 190.7/sec

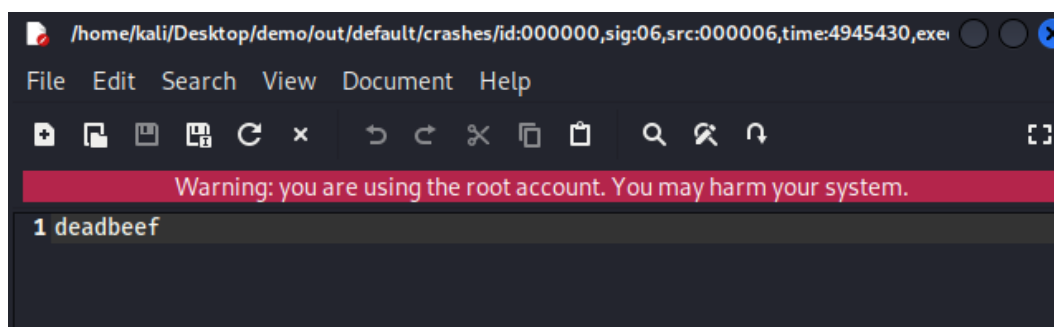
findings in depth
  favored items : 8 (100.00%)
  new edges on : 8 (100.00%)
  total crashes : 2 (1 saved)
  total tmouts : 0 (0 saved)

fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 8/827k, 0/175k
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 39.22%/10, disabled

item geometry
  levels : 6
  pending : 0
  pend fav : 0
  own finds : 7
  imported : 0
  stability : 100.00%

[cpu000: 50%]
```

在 out 文件夹下的 crashes 子文件夹里面是产生 crash 的样例，hangs 里面是产生超时的样例，queue 里面是每个不同执行路径的测试用例。打开 crash:



```
/home/kali/Desktop/demo/out/default/crashes/id:000000,sig:06,src:000006,time:4945430,exe:
File Edit Search View Document Help

Warning: you are using the root account. You may harm your system.

1 deadbeef
```

通常，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。

3.3 覆盖引导和文件变异

AFL 是基于覆盖引导的模糊测试工具，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。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。

覆盖引导是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。

文件变异是指在模糊测试中，对输入文件进行变异以产生新的测试用例。文件变异应该具有启发性判断，避免浪费资源并减少不必要的消耗。

4 心得体会

通过本次 AFL 模糊测试，对 AFL 模糊测试流程和原理有了更深入的理解和认识。