《软件安全》实验报告

姓名：蒋枘言　学号：2313546　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、实验过程

（一）安装 AFL

首先按照步骤在 Kali Linux 虚拟机里安装 AFL。安装成功后，可以发现有许多对应不同功能的文件，如下图所示。

```
┌──(kali㉿kali)-[~/demo]
└─$ ls /usr/local/bin
afl-analyze  afl-clang++  afl-fuzz  afl-gcc     afl-plot     afl-tmin
afl-clang    afl-cmin     afl-g++   afl-gotcpu  afl-showmap  afl-whatsup
```

（二）创建本次实验的程序

新建文件夹 demo，创建本次实验的程序 test.c，程序内容如下图所示。

*~/demo/test.c - Mousepad

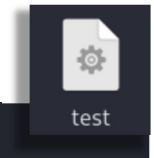
File Edit Search View Document Help

```
#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') {
                                                          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;
}
```

该程序通过文件（argv[1]）或标准输入（stdin）读取数据到 ptr 缓冲区。fgets(ptr, sizeof(ptr), fp) 限制了最多读取 19 字节（sizeof(ptr)=20）（防止缓冲区溢出）。当输入字符串的前 8 字节为 **deadbeef** 时，程序会调用 abort() 崩溃。

执行命令 `afl-gcc -o test test.c`，使用 AFL 的编译器编译，生成了名为 test 的可执行文件。




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

gcc 编译器和 afl-gcc 的核心区别如下表所示：

特性	gcc	afl-gcc
主要用途	常规程序编译	针对模糊测试的编译（插桩支持覆盖率反馈）
代码插桩	无	插入 AFL 的代码插桩，记录执行路径
适用场景	普通开发、发布构建	模糊测试（如漏洞挖掘、代码覆盖率优化）
输出程序	无特殊功能	包含插桩逻辑，可与 AFL 交互反馈覆盖率信息

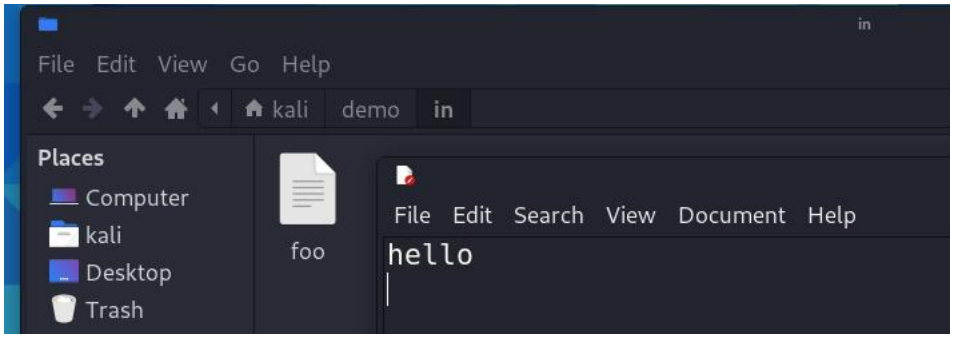
执行 `readelf -s ./test | grep afl` 命令后得到如下图所示的内容：这些符号是 AFL 插桩的标志，表明 test 程序已为模糊测试优化。

```
┌──(kali㉿kali)-[~/demo]
└─$ readelf -s ./test | grep afl
    35: 0000000000001628     0 NOTYPE  LOCAL  DEFAULT   14 __afl_maybe_log
    37: 00000000000040b0     8 OBJECT  LOCAL  DEFAULT   25 __afl_area_ptr
    38: 0000000000001658     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup
    39: 0000000000001638     0 NOTYPE  LOCAL  DEFAULT   14 __afl_store
    40: 00000000000040b8     8 OBJECT  LOCAL  DEFAULT   25 __afl_prev_loc
    41: 0000000000001650     0 NOTYPE  LOCAL  DEFAULT   14 __afl_return
    42: 00000000000040c8     1 OBJECT  LOCAL  DEFAULT   25 __afl_setup_failure
    43: 0000000000001679     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup_first
    45: 000000000000193e     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup_abort
    46: 0000000000001793     0 NOTYPE  LOCAL  DEFAULT   14 __afl_forkserver
    47: 00000000000040c4     4 OBJECT  LOCAL  DEFAULT   25 __afl_temp
    48: 0000000000001851     0 NOTYPE  LOCAL  DEFAULT   14 __afl_fork_resume
    49: 00000000000017b9     0 NOTYPE  LOCAL  DEFAULT   14 __afl_fork_wait_loop
    50: 0000000000001936     0 NOTYPE  LOCAL  DEFAULT   14 __afl_die
    51: 00000000000040c0     4 OBJECT  LOCAL  DEFAULT   25 __afl_fork_pid
    98: 00000000000040d0     8 OBJECT  GLOBAL DEFAULT   25 __afl_global_area_ptr
```

（三）创建测试用例

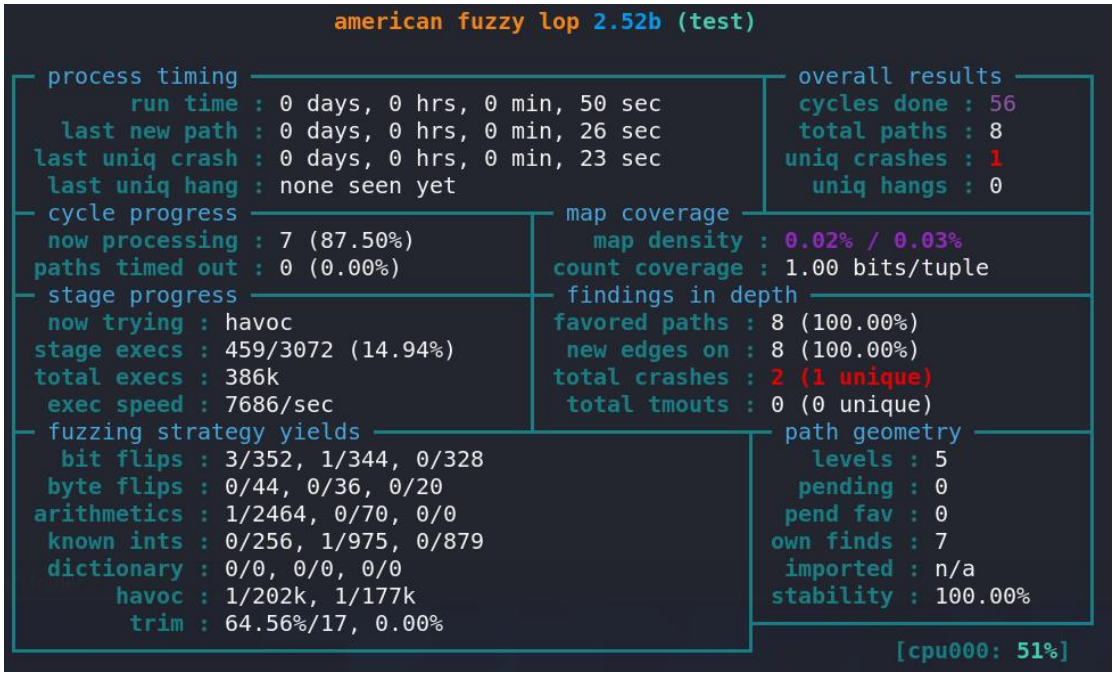
首先，执行命令 `mkdir in out`，创建两个文件夹 in 和 out，分别存储模糊测试所需的输

入和输出相关的内容。然后，执行命令 `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 就是我们的测试用例，里面包含初步字符串“hello”。in/foo 是**种子文件**，AFL 会以其为基础生成随机输入，即通过这个语料进行变异，构造更多的测试用例。



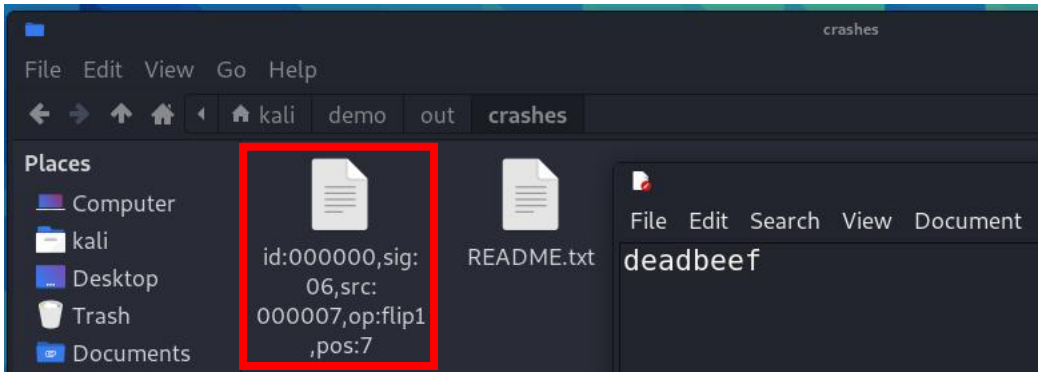
（四）启动模糊测试

执行命令 `afl-fuzz -i in -o out -- ./test @@` ，启动模糊测试，结果如下（未全部完成）：



可以看到界面被分成了很多个板块。①process timing 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。②overall 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，我们便可以通过 Ctrl-C，中止当前的 fuzzing。③stage 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，那么我们需要进一步调整优化我们的 fuzzing。

找到一个 crash 后，在 out/crashs 里面打开第一个文件，发现正是字符串“deadbeef”。可以将得到的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。



四、心得体会

通过本次实验，我对 AFL 模糊测试的核心概念和基本操作有了初步认识，主要收获如下：

1.AFL 工具链的初体验

从安装 AFL 到使用 afl-gcc 编译插桩程序，直观感受到模糊测试工具对代码的改造能力。通过对比普通 gcc 与 afl-gcc 生成的符号表（readelf 命令），理解了插桩如何为覆盖率反馈提供数据支持，这是 AFL 高效生成测试用例的基础。

2.模糊测试流程的实践验证

通过创建种子文件（in/foo）和运行 afl-fuzz，亲历了“输入变异-监控路径-捕获崩溃”的完整流程。首次看到 AFL 界面中的覆盖率统计与崩溃记录时，深刻体会到自动化测试在边缘场景触发上的强大能力。

3.种子文件的重要性

实验中简单的 hello 字符串竟能衍生出触发崩溃的 deadbeef 输入，这让我意识到种子文件的质量直接影响测试效率。

4.安全测试的启发

虽然本次实验仅模拟了一个简单的崩溃场景，但让我认识到模糊测试在漏洞挖掘中的实际价值。它不仅是工具的使用，更是一种通过程序行为反推潜在风险的思维方式。