《软件安全》实验报告

姓名： 王众　学号：2313211

实验名称

AFL模糊测试

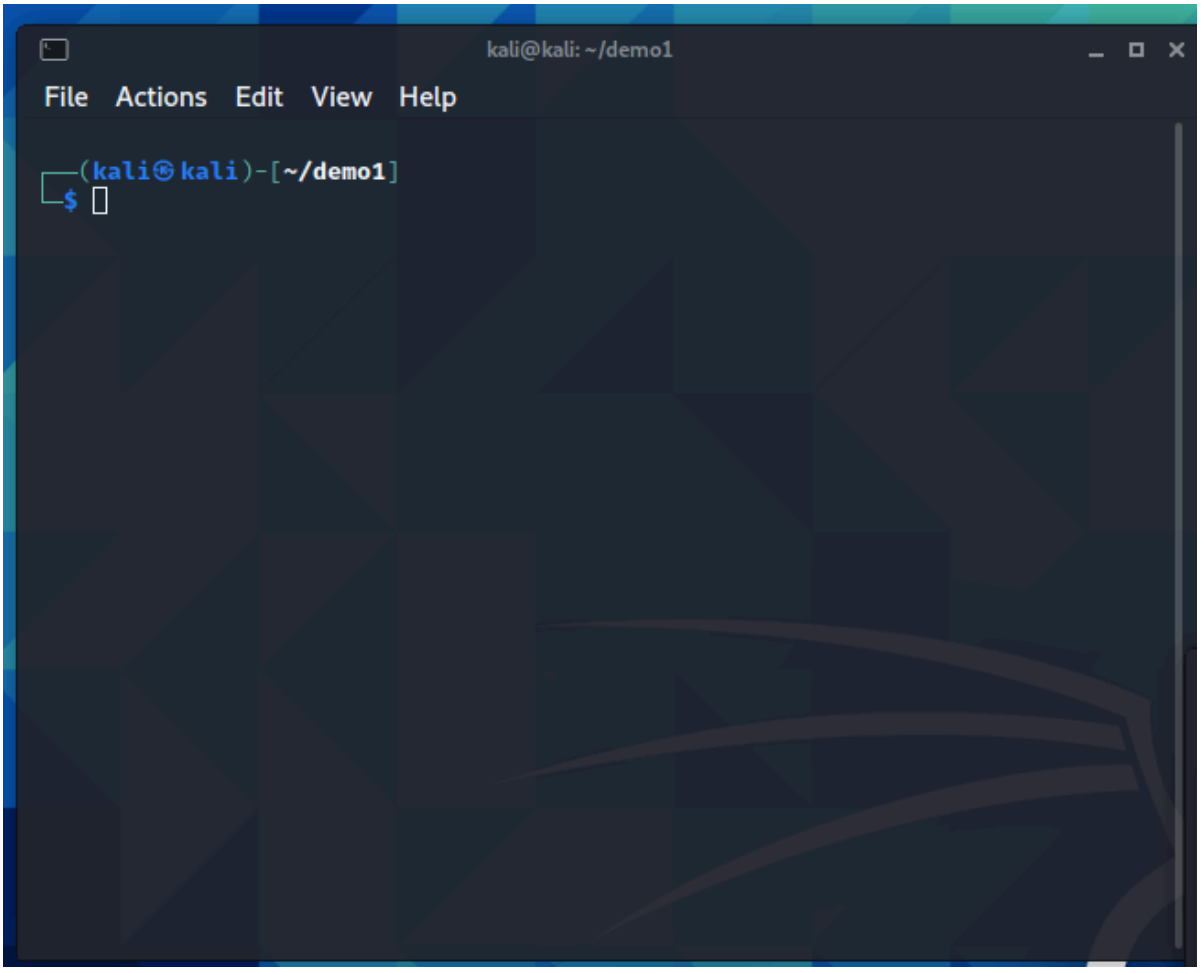
实验要求

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。

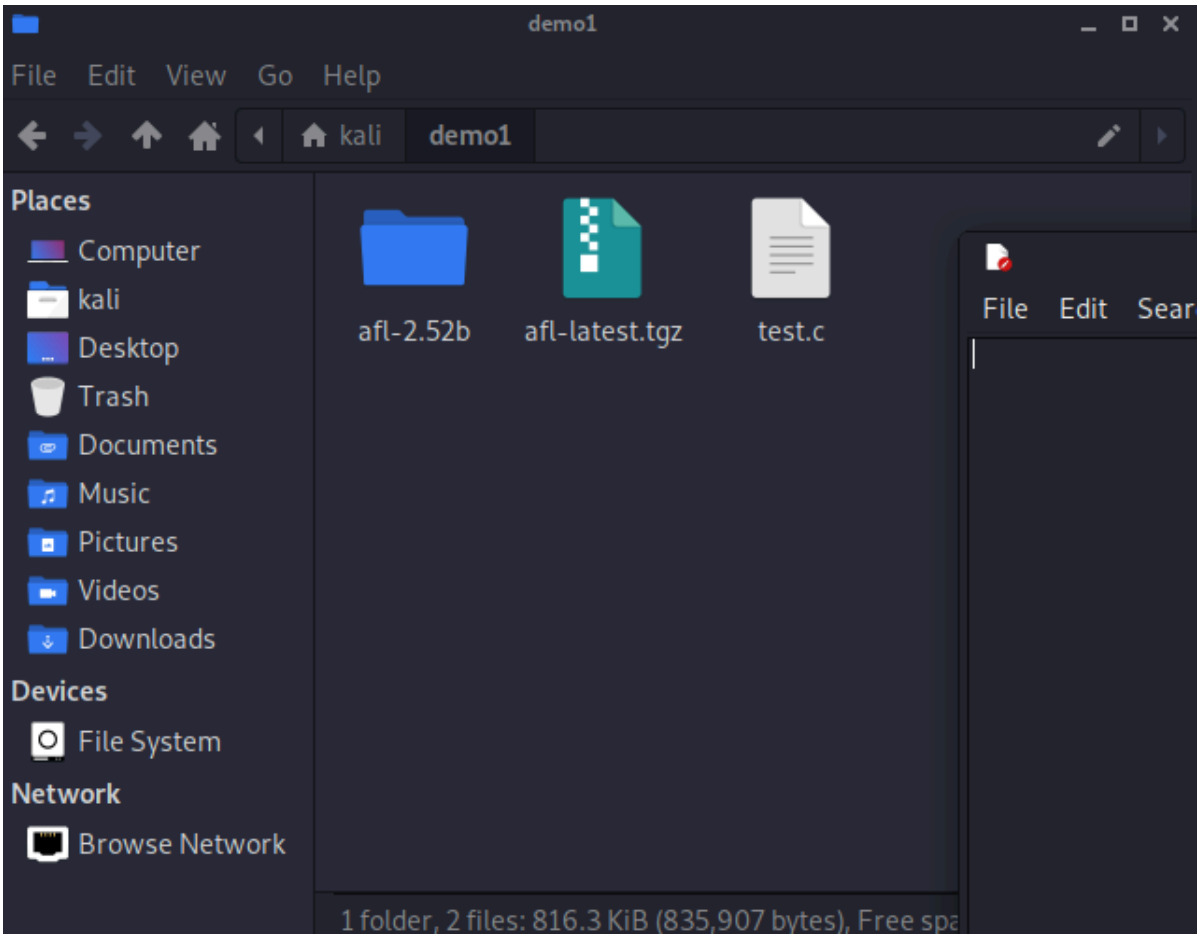
实验过程

1 安装AFL

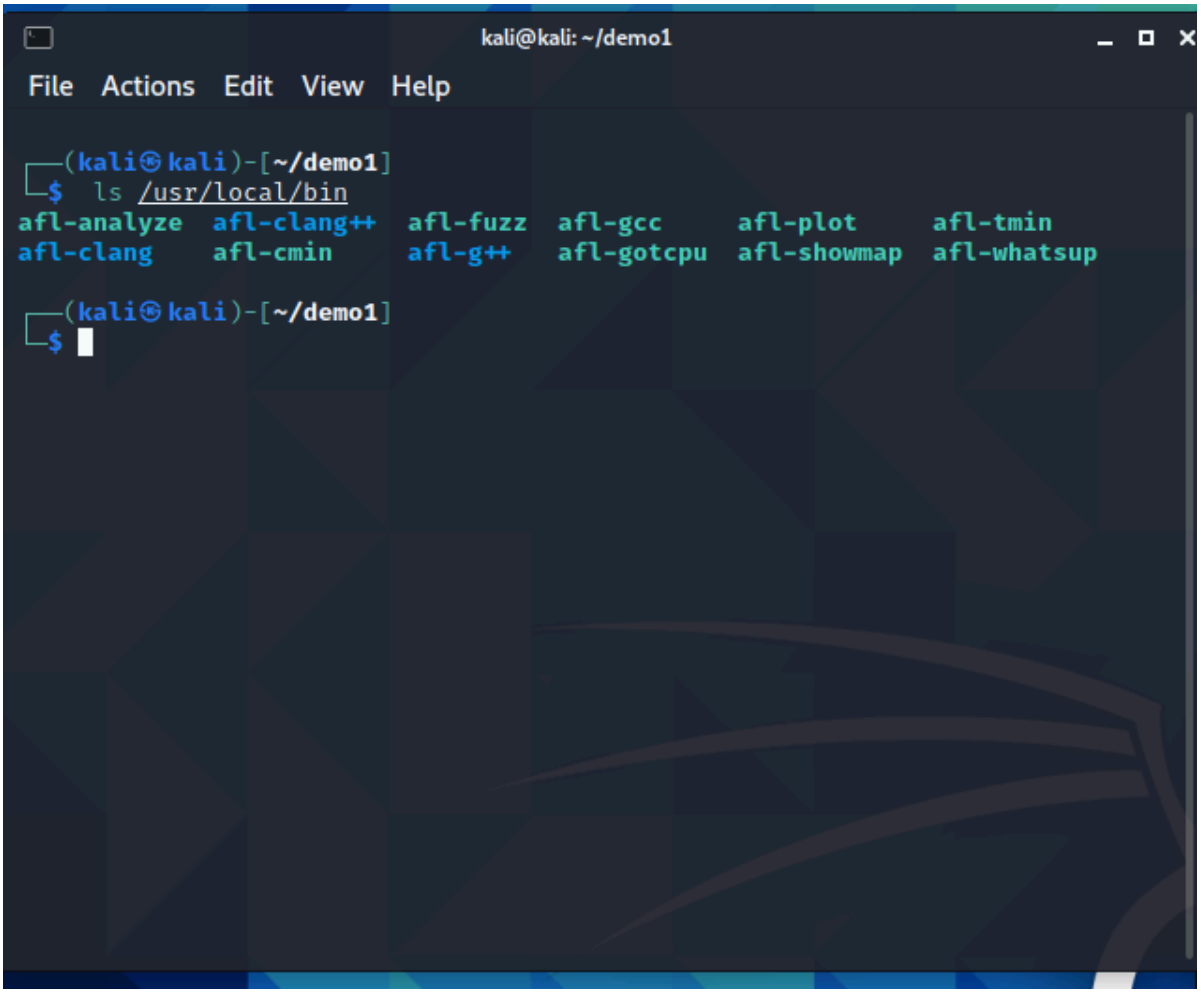
我们先根据实验要求，进入$vmware$ ，开启$kali$ 虚拟机，并在其中创建一个新的文件夹，名字是$demo$。然后我们按下右键，打开终端。



我们在本地下载了文件夹，然后复制到 $kali$虚拟机中去，拖入$demo$ 文件夹。然后我们输入命令行 `tar xvf afl-latest.tgz` ，解压安装包。



我们通过命令行 `cd afl-2.52b` 进入对应的文件夹，然后输入 `sudo make && sudo make install` 来编译AFL。发现编译成功，我们输入命令行 `ls /usr/local/bin` 来验证是否安装成功。



发现安装成功，并且输出了里面含有AFL的 文件。

2 AFL的应用

我们接下来利用安装好的AFL 文件，来复现课本上出现的模糊测试的案例，来进一步加深对于AFL 应用的理解。

2.1 创建测试程序

在 $demo$文件夹中新建一个$test.c$ 文件，并输入我们的源码：

```
#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;
}
```

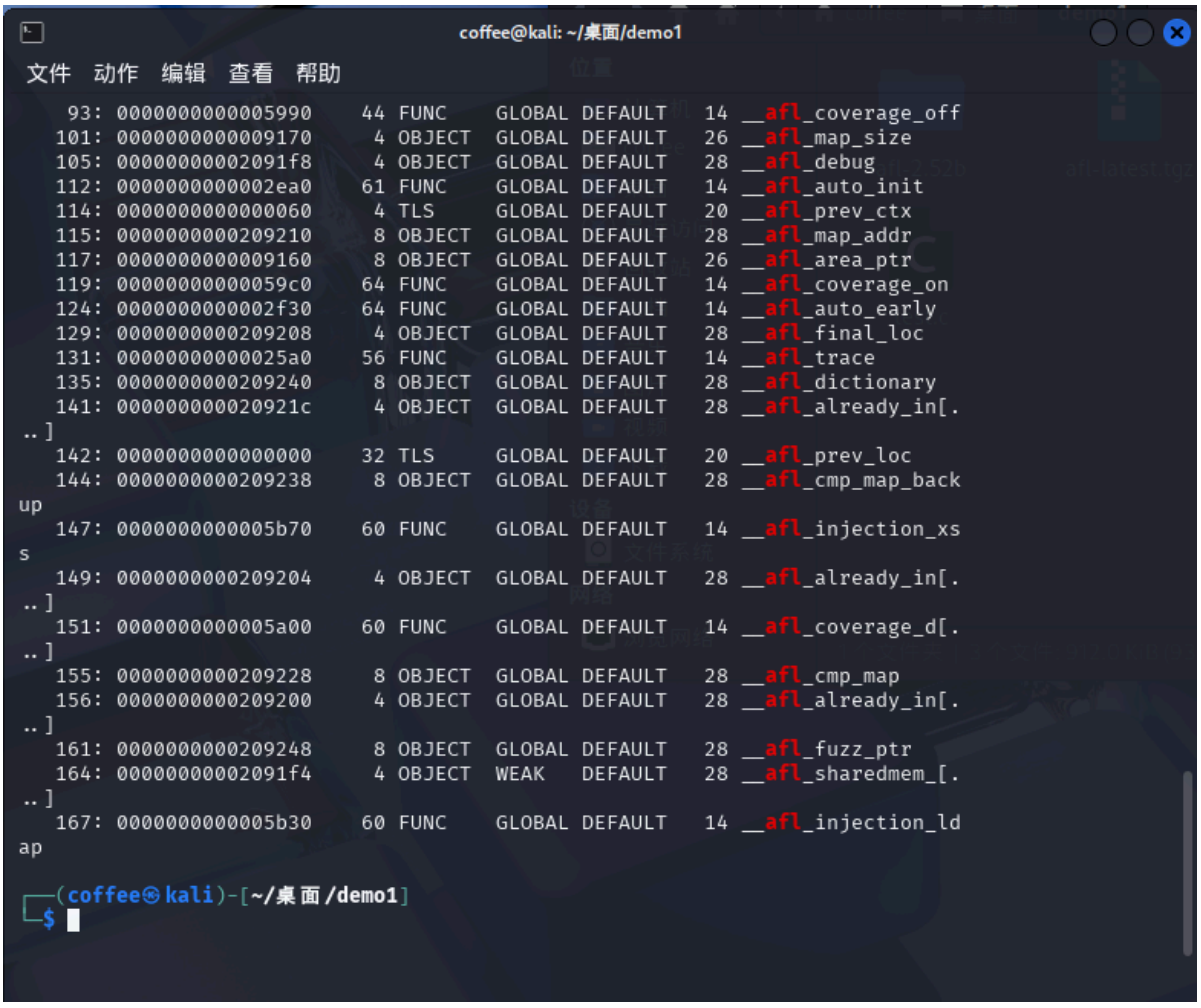
通过分析代码我们可以知道，当输入字符串“ `deadbeef` ”时程序捕捉到一个异常，程序终止。

我们使用 $linus$的编译器进行编译，可以使模糊过程更加高效。我们输入命令行 `afl-gcc -o test test.c`，来对源代码进行编译。发现得到一个编译后的文件。

```
┌──(coffee㉿kali)-[~/桌面/demo1]
└─$ afl-gcc -o test test.c
afl-cc++4.21c 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++4.21c by Michal Zalewski
[+] Instrumented 14 locations (64-bit, non-hardened mode, ratio 100%).

┌──(coffee㉿kali)-[~/桌面/demo1]
└─$ afl-clang-fast -o test test.c
afl-cc++4.21c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: LLVM-PC
GUARD
SanitizerCoveragePCGUARD++4.21c
[+] Instrumented 12 locations with no collisions (non-hardened mode) of which
 are 0 handled and 0 unhandled selects.
```

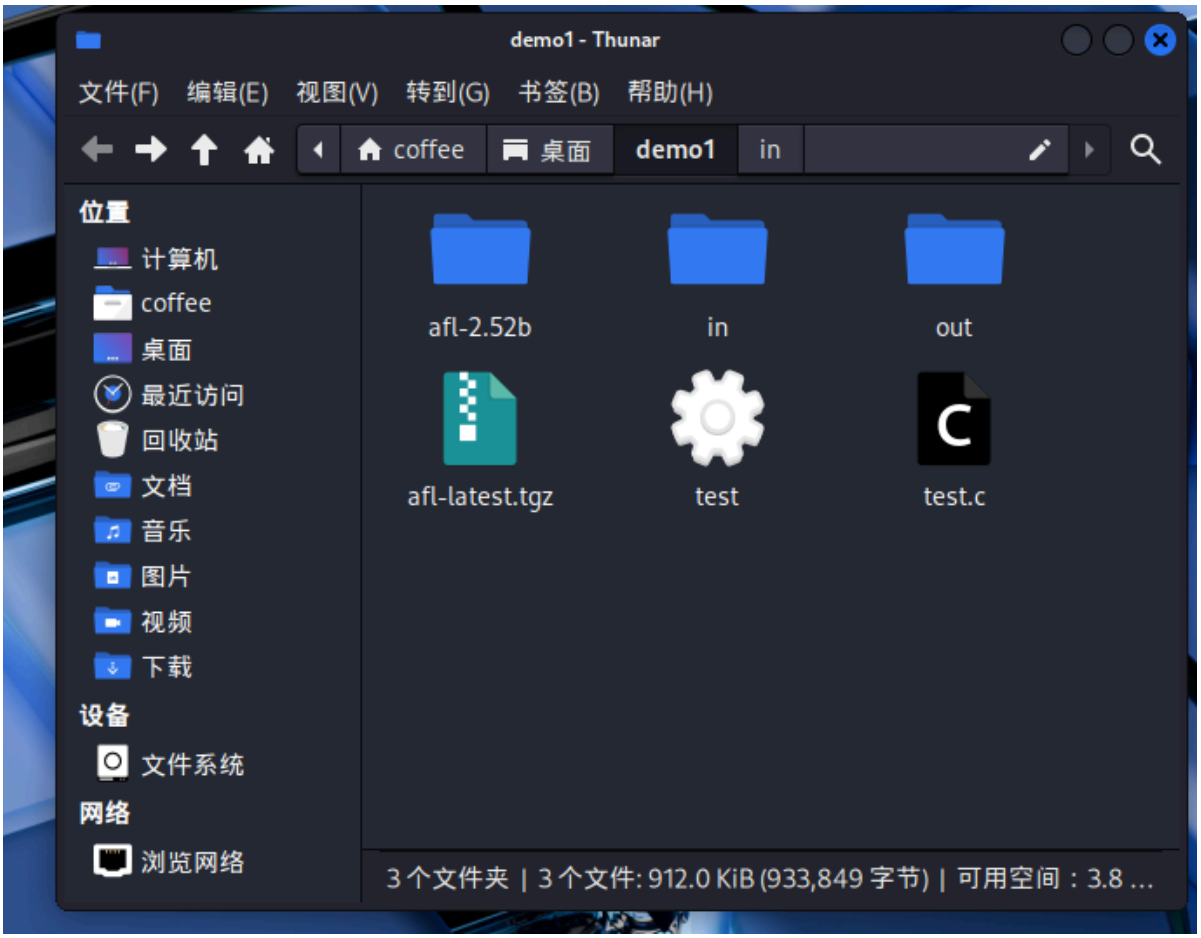
我们接下来用命令行 `readelf -s ./test | grep afl` 来验证插桩符号：



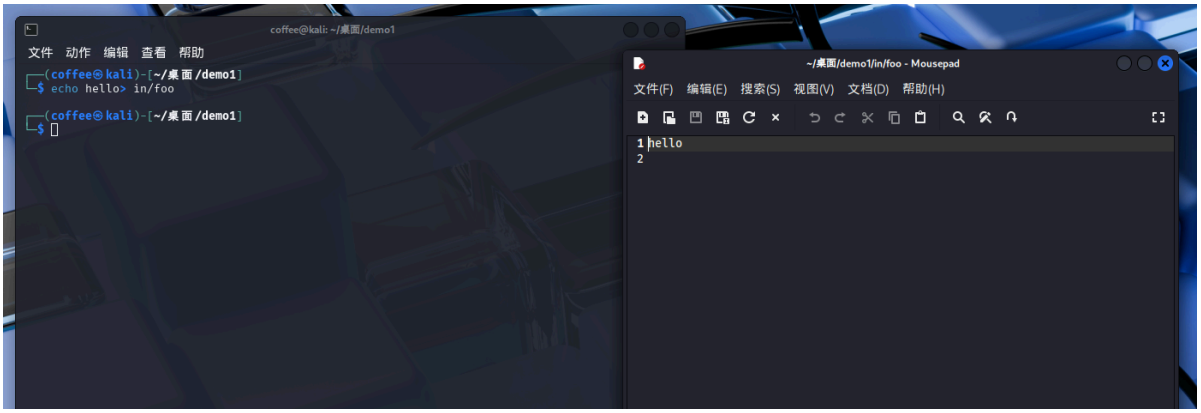
现在我们便创建好了我们本次实验所需要用到的程序，接下来我们需要创建测试用例。

2.2 测试用例的创建

首先，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。命令：`mkdir in out`，结果如下图：



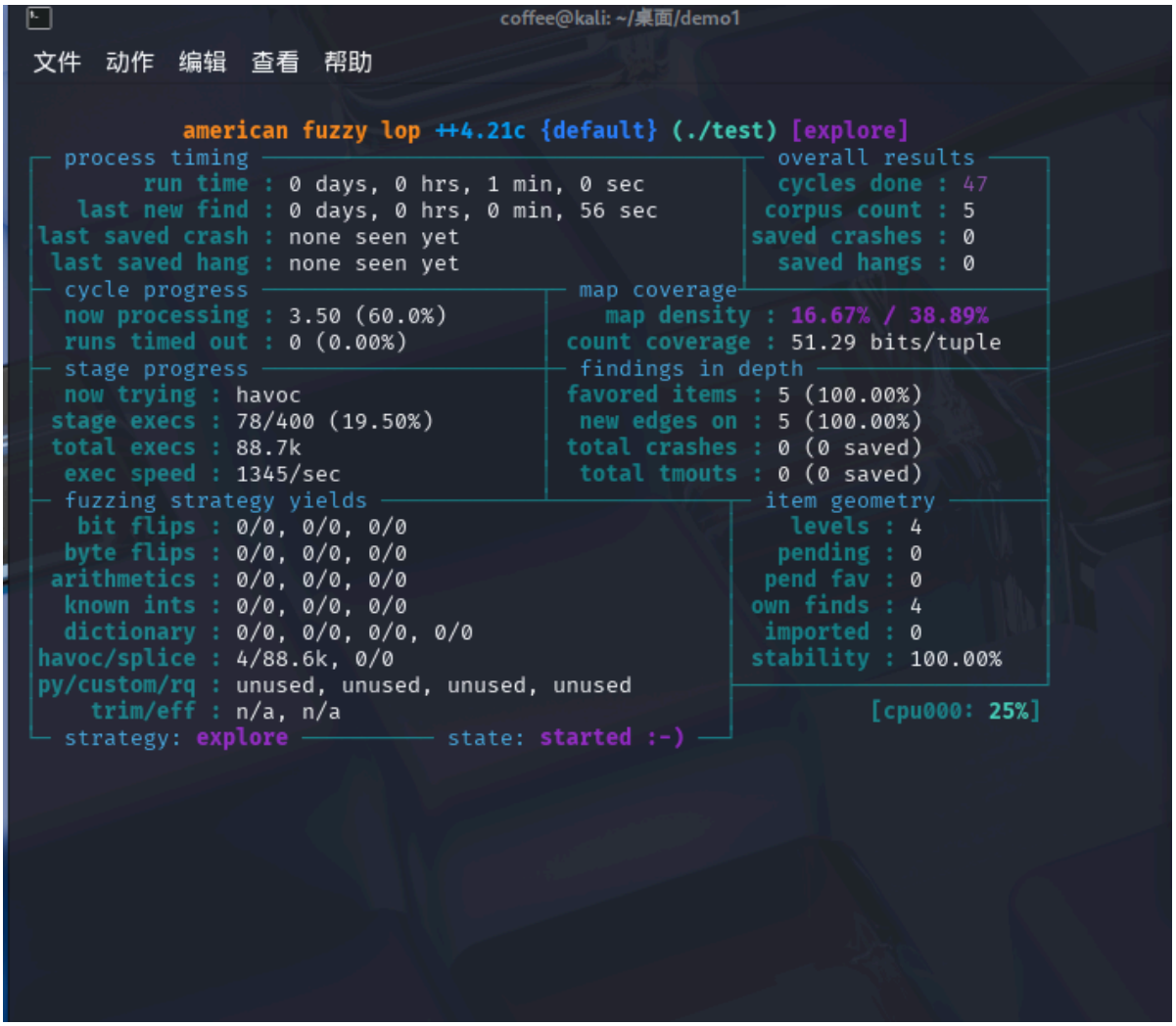
然后，在输入文件夹中创建一个包含字符串“ `hello` ”的文件。 命令行：`echo hello> in/foo` 。`foo` 就是我们的测试用例，里面包含初步字符串 `hello` 。 AFL会通过这个语料进行编译，构造出更多 的测试用例。

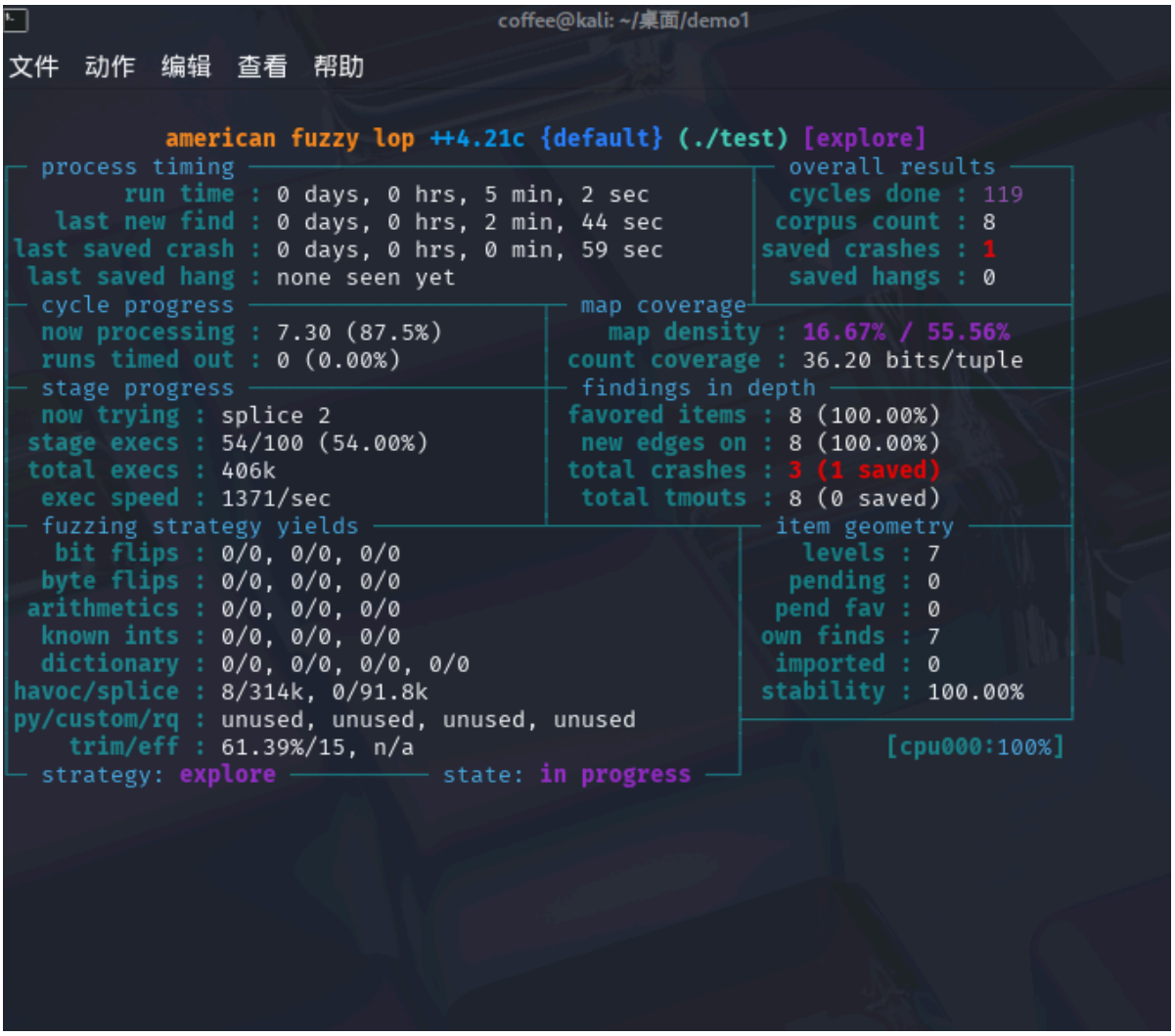


至此，我们就创建好了测试用例，接着就是启动模糊测试，然后观察结果。

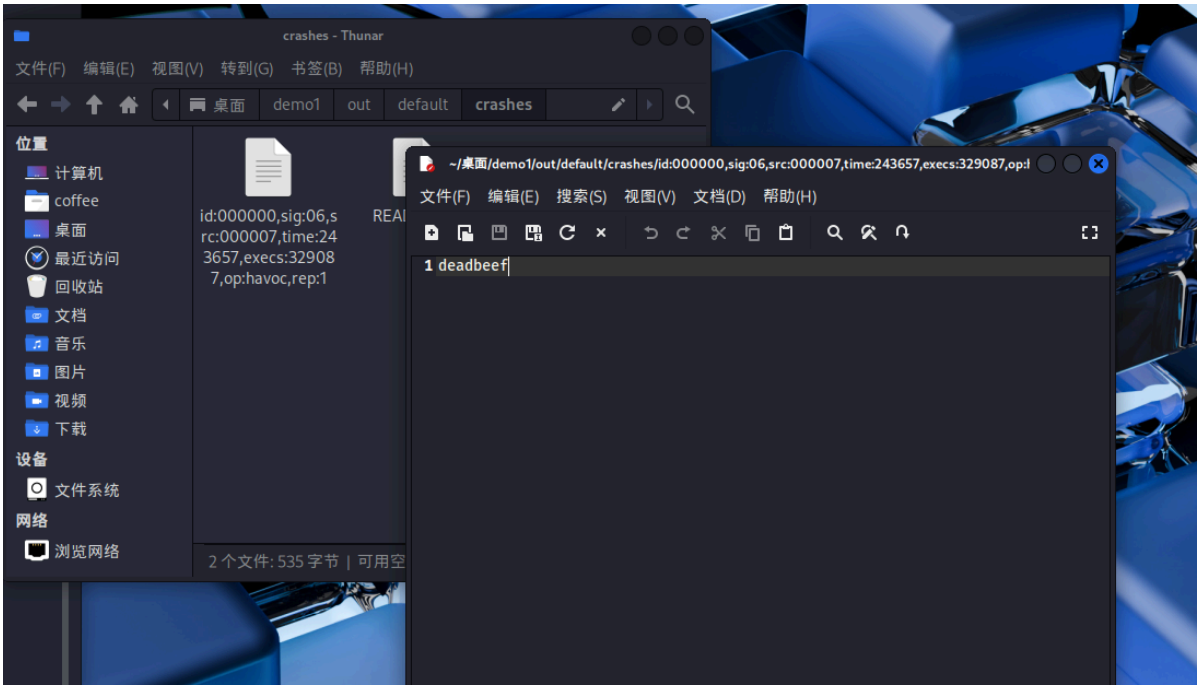
2.3 启动模糊测试

我们使用如下的命令行 `afl-fuzz -i in -o out -- ./test @@` 来进行模糊测试的启动：


coffee@kali: ~/桌面/demo1
文件 动作 编辑 查看 帮助
american fuzzy lop ++4.21c {default} (./test) [explore]
process timing
run time : 0 days, 0 hrs, 1 min, 0 sec
last new find : 0 days, 0 hrs, 0 min, 56 sec
last saved crash : none seen yet
last saved hang : none seen yet
overall results
cycles done : 47
corpus count : 5
saved crashes : 0
saved hangs : 0
cycle progress
now processing : 3.50 (60.0%)
runs timed out : 0 (0.00%)
map coverage
map density : 16.67% / 38.89%
count coverage : 51.29 bits/tuple
stage progress
now trying : havoc
stage execs : 78/400 (19.50%)
total execs : 88.7k
exec speed : 1345/sec
findings in depth
favored items : 5 (100.00%)
new edges on : 5 (100.00%)
total crashes : 0 (0 saved)
total tmouts : 0 (0 saved)
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 : 4/88.6k, 0/0
py/custom/rq : unused, unused, unused, unused
trim/eff : n/a, n/a
item geometry
levels : 4
pending : 0
pend fav : 0
own finds : 4
imported : 0
stability : 100.00%
[cpu000: 25%]
strategy: explore
state: started :-)



当观察到产生了一个 $crash$时，我们去$out->crash$ 中可以看到导致本次崩溃的输入：



恰为我们之前分析的“ `deadbeef` ”，验证完毕！

心得体会：

对于AFL覆盖引导的理解：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。

通过实验，进一步了解了 的工作流程：

1.从源码编译程序时进行插桩，以记录代码覆盖率（$CodeCoverage$）；

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。

理解覆盖引导和文档 件变异的概念和含义：

覆盖引导（覆盖率引导）

覆盖引导是AFL模糊测试的核心机制之一。简单来说，AFL会通过在目标程序中插入一些“探针”（插桩），来监控程序运行时哪些代码路径被执行了。比如，程序里有哪些分支（if-else语句）被触发，或者哪些代码块被访问。这些信息就是“覆盖率”。

- **作用**：AFL用覆盖率来判断一个输入是不是“有趣”。如果一个输入能让程序执行到新的代码路径（比如触发了一个之前没跑过的分支），AFL就会认为这个输入有价值，会基于这个输入生成更多变异输入。
- **意义**：覆盖引导让AFL的测试更有针对性，而不是盲目地随机测试。它能帮助AFL更快地探索程序的各种行为，找到隐藏的漏洞。

文档件变异（输入变异）

这里的“文档件”指的是AFL测试时用的输入文件（也叫种子输入），比如一个文本文件、图片文件或者其他数据文件。“变异”就是AFL对这些输入文件进行修改的过程。

- 怎么变异
  ：AFL会用一些策略来改变输入文件，比如：
  - 随机翻转文件中的某些位（比如把0变成1）。
  - 替换文件中的某些字节。
  - 增加或删除文件中的一部分数据。
  - 把两个不同文件的部分拼接起来。
- **作用**：通过这些修改，AFL生成大量新的测试用例，试图让程序处理这些“变异”后的输入时出错（比如崩溃或出现异常）。
- **意义**：变异的过程让AFL能尝试各种可能的输入情况，覆盖更多的程序行为，从而提高发现漏洞的几率。