《漏洞利用及渗透测试基础》实验报告

姓名： 艾明旭 学号：2111033 班级：信息安全一班

**实验名称：**

AFL模糊测试实验

**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. 首先安装AFL

由于sudo apt-get install afl的指令一直报错，故采用其他的方式进行afl的安装

在Kali 2021下，利用如下指令进行安装

1. wget <http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz>

首先使用命令下载安装包（网速有要求）：

1. tar xvf afl-latest.tgz

****解压命令****

1. cd afl-2.52b

打开

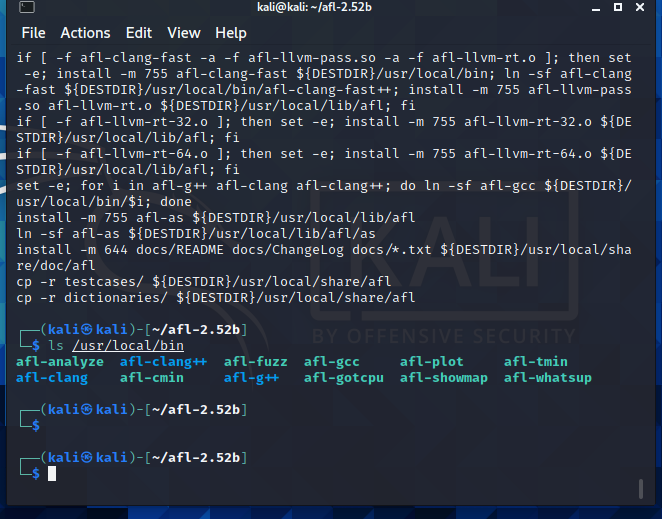
1. sudo make && sudo make install

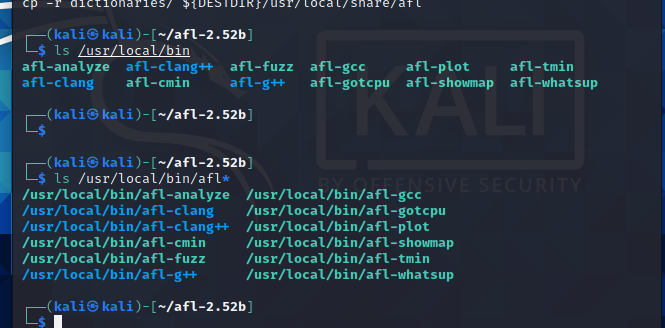
****进入目标文件夹并且编译AFL：****

1. ls /usr/local/bin/afl\*

我们可以在\*\*/usr/local/bin\*\*路径下找到AFL

查看路径可以看到afl安装的文件：ls /usr/bin/afl\*





作用分别为

• afl-gcc和afl-g++分别对应的是gcc和g++的封装。

• afl-clang和afl-clang++分别对应clang的c和c++编译器封装。

• afl-fuzz是AFL的主体，用于对目标程序进行fuzz。

• afl-analyze可以对用例进行分析，看能否发现用例中有意义的字段。

• afl-qemu-trace用于qemu-mode，默认不安装，需要手工执行qemu-mode的编译脚本进行编译。

• afl-plot生成测试任务的状态图。

• afl-tmin和afl-cmin对用例进行简化。

• afl-whatsup用于查看fuzz任务的状态。

• afl-gotcpu用于查看当前CPU 状态。

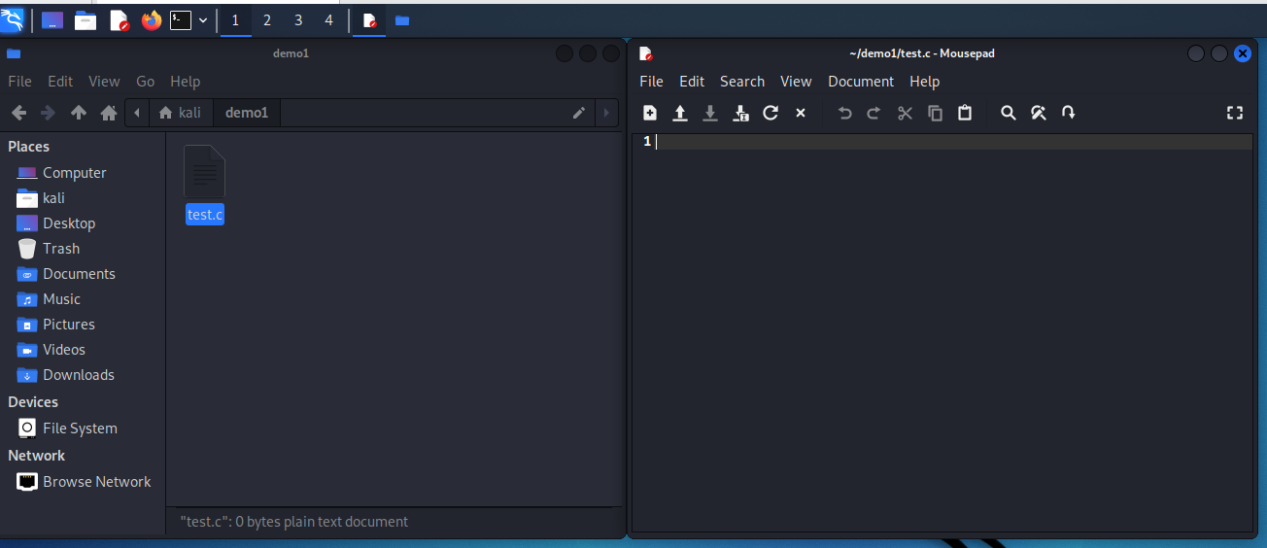
• afl-showmap用于对单个用例进行执行路径跟踪。

1. **进行AFL测试**

以一个白盒模糊测试为例

**1）创建本次实验的程序**

新建文件夹demo，并创建本次实验的程序test.c

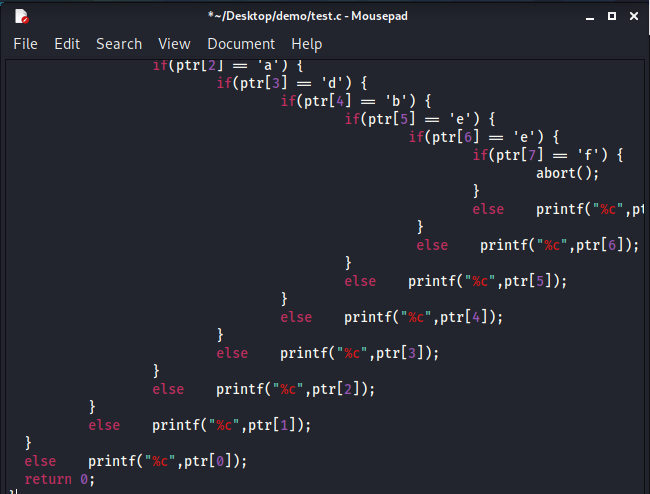


文件中输入以下代码

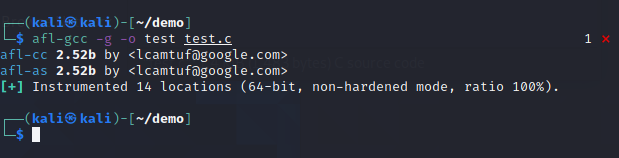
代码编译后得到的程序如果被传入“deadbeef”则会终止，触发异常；

如果传入其他字符会原样输出

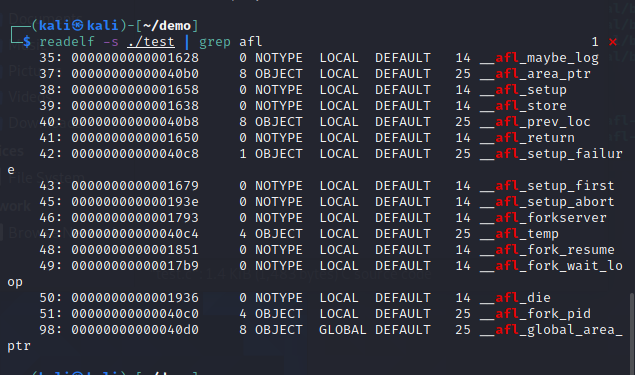
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*\*argv) {  char ptr[20];  if(argc>1){  FILE \*fp = fopen(argv[1], "r");  fgets(ptr, sizeof(ptr), fp);  }  else{  fgets(ptr, sizeof(ptr), stdin);  }  printf("%s", ptr);  if(ptr[0] == 'd') {  if(ptr[1] == 'e') {  if(ptr[2] == '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;  } |



命令：**afl-gcc -o test test.c** 编译产生目标文件 -o test指定文件为test test.c 意为源文件是test.c 使用afl的编译器编译，可以使模糊测试过程更加高效

re

编译后会有插桩符号，使用下面的命令：**readelf -s ./test | grep afl**可以验证这一点（插桩在编译过程中插入便于分析的其他代码，使得找寻漏洞更容易）

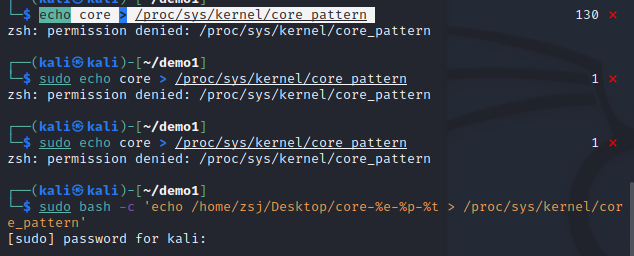
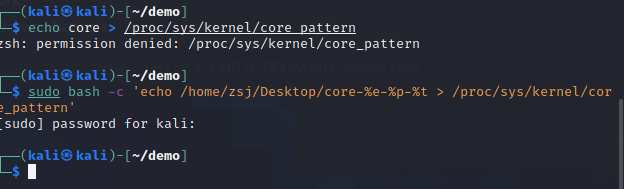


如图，可以看到产生的目标文件test里面有插桩

进行测试之前，需要做一些准备工作：

1. 输入命令：**echo core > /proc/sys/kernel/core\_pattern**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；
2. 创建输入输出文件夹

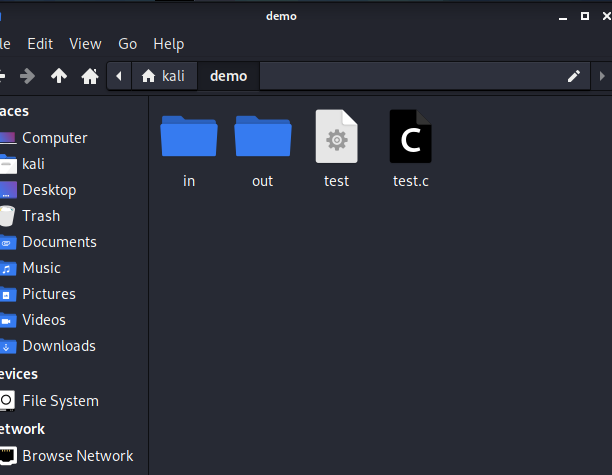
由于用这条命令权限不足无法执行，改用命令：**sudo bash -c 'echo /home/zsj/Desktop/core-%e-%p-%t > /proc/sys/kernel/core\_pattern'**解决



**2）创建测试用例**

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**

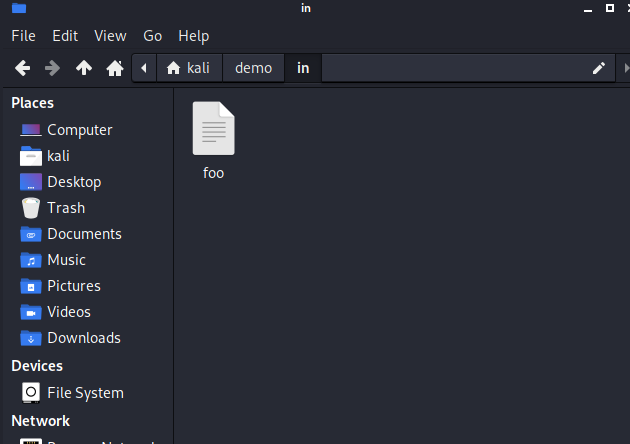
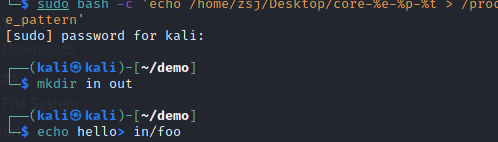


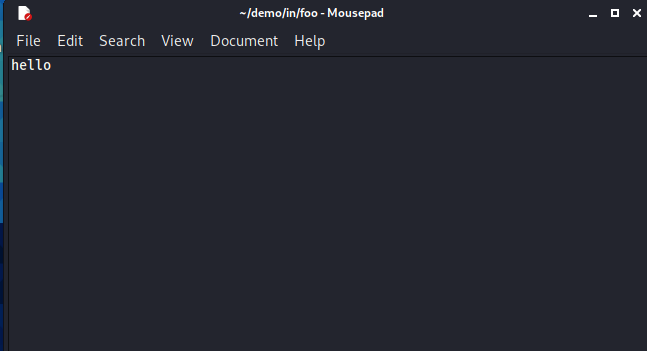
可以看到创建了输入输出文件夹

然后，在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。这就是输入的种子文件，后续AFL会通过这个语料进行变异，构造更多的测试用例



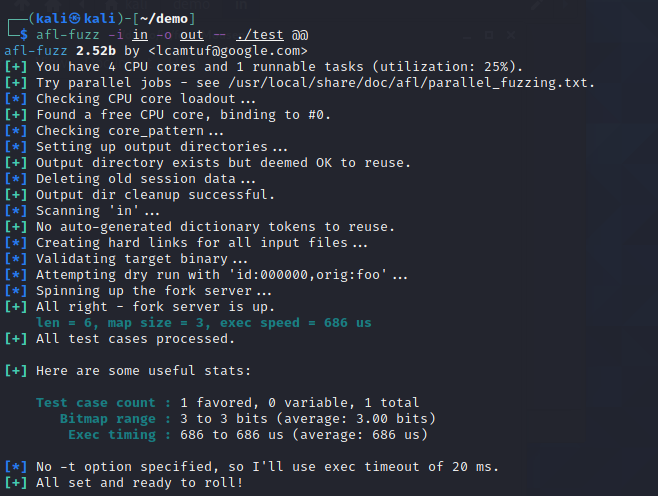


**3）启动模糊测试**

运行如下命令，开始启动模糊测试：

命令：**afl-fuzz -i in -o out -- ./test @@**

**（命令含义为输入为in文件夹，输出为out文件夹，目标文件为test @@表明从文件中读取输入）**



注意：

对那些可以直接从stdin读取输入的目标程序来说，语法如下：

$ ./afl-fuzz -i testcase\_dir -o findings\_dir /path/to/program […params…]

对从文件读取输入的目标程序来说，要用“@@”，语法如下：

$ ./afl-fuzz -i testcase\_dir -o findings\_dir /path/to/program @@

AFL执行过程

**·process timing**

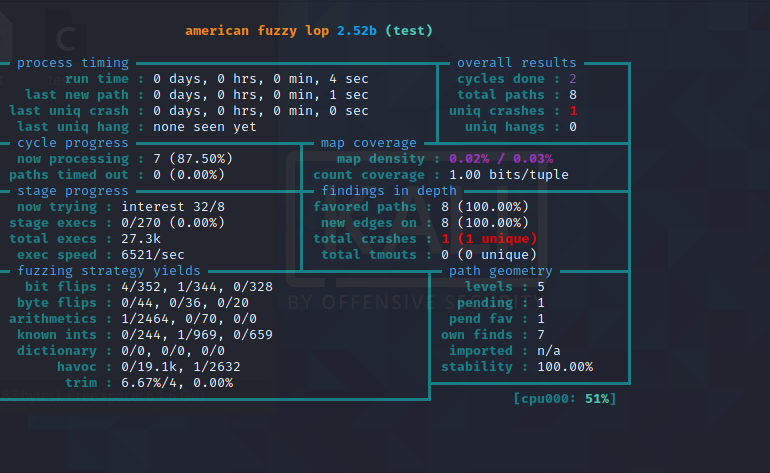
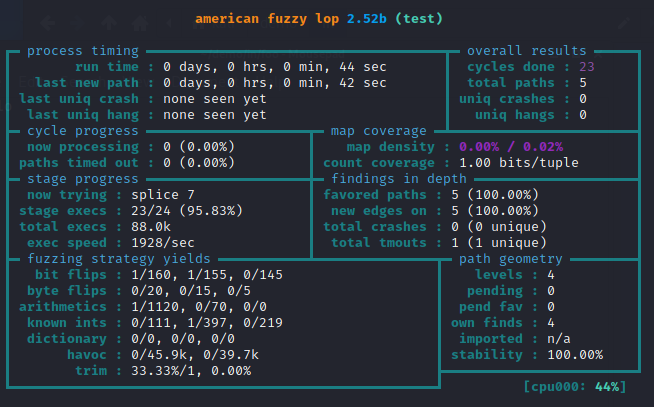
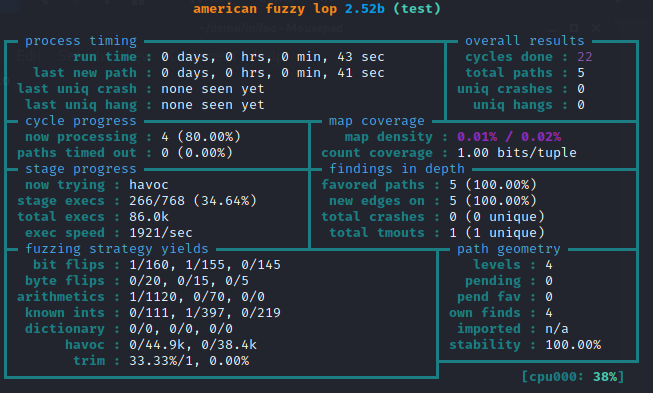
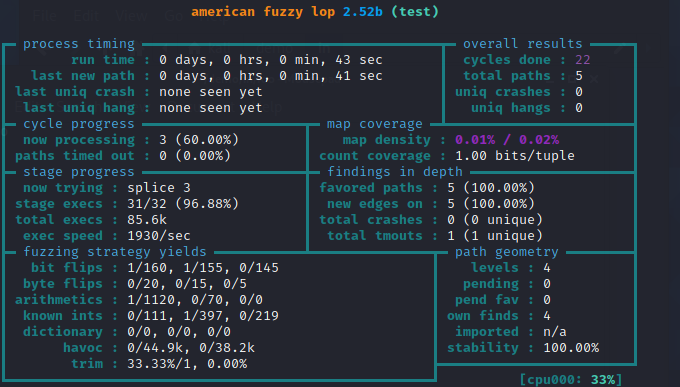
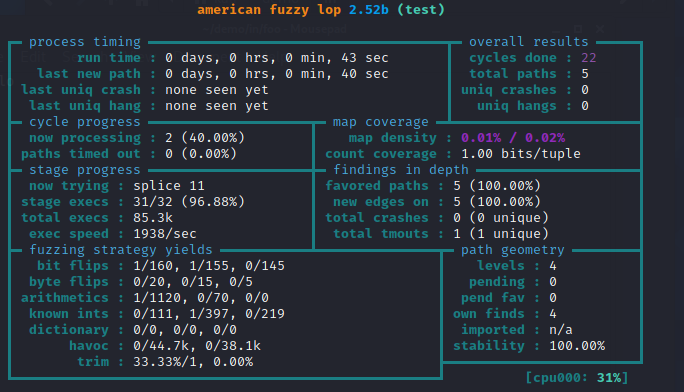
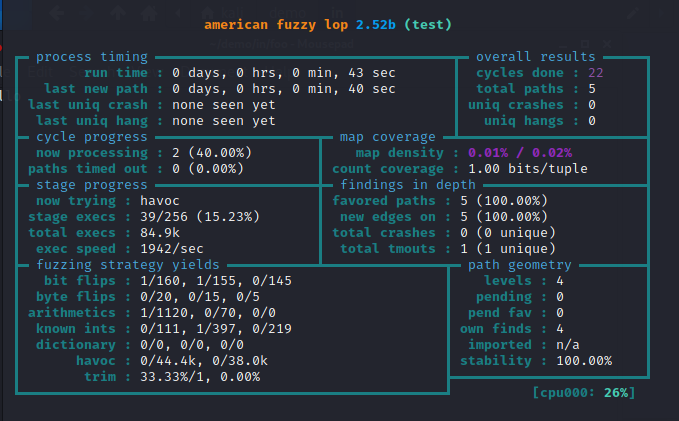
这里展示了当前fuzzer的运行时间、最近一次发现新执行路径的时间、最近一次崩溃的时间、最近一次超时的时间。

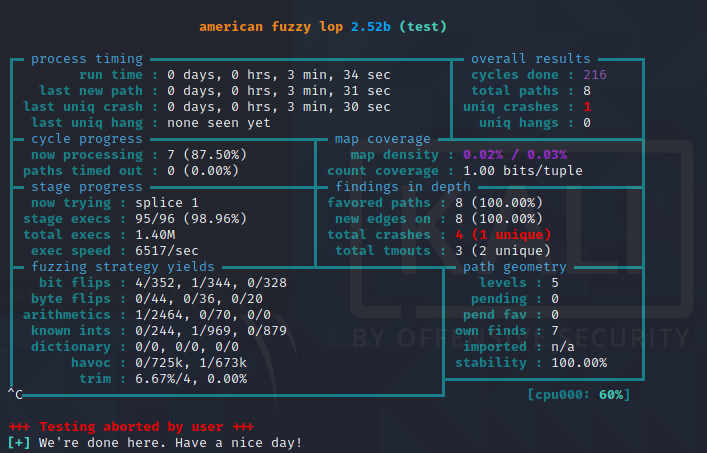
**·overall results**

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。

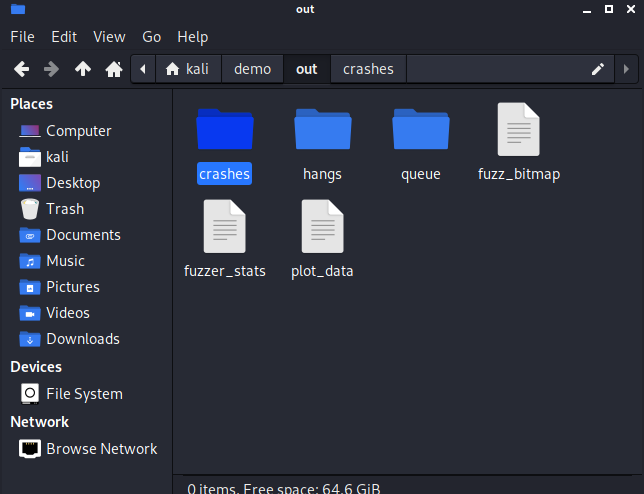
·**stage progress**

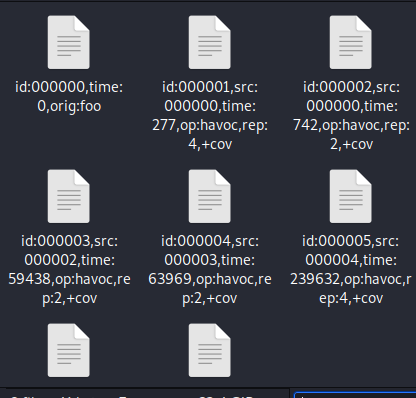
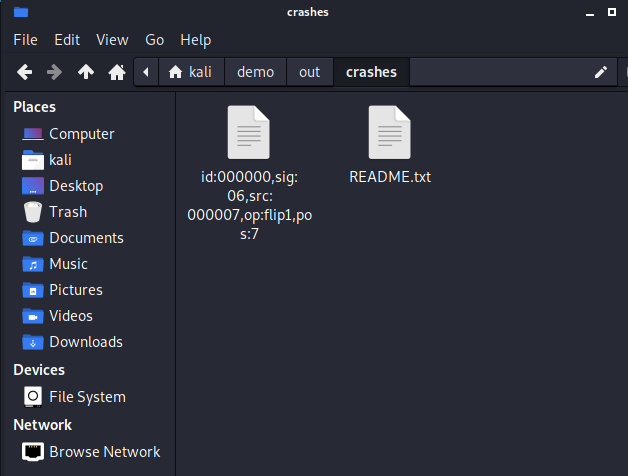
这里包括正在测试的fuzzing策略、进度、目标的执行总次数、目标的执行速度。执行速度可以直观地反映当前跑的快不快，如果速度过慢，比如低于500次每秒，那么测试时间就会变得非常漫长。如果发生了这种情况，那么我们需要进一步调整优化我们的fuzzing。



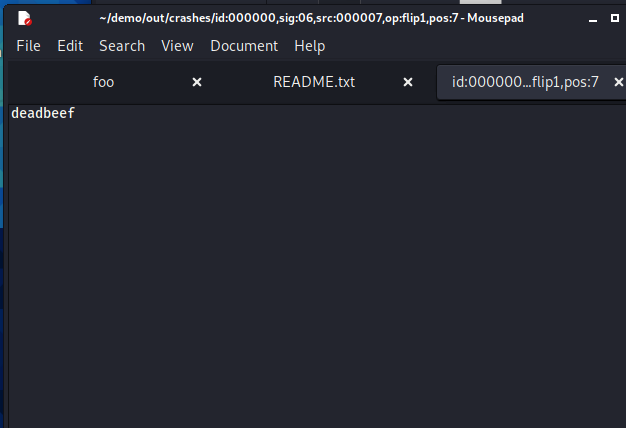


在out文件夹下的crashes子文件夹里面是我们产生crash的样例，hangs里面是产生超时的样例，queue里面是每个不同执行路径的测试用例。



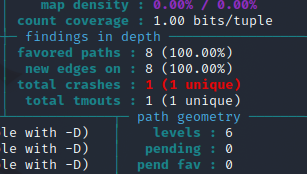


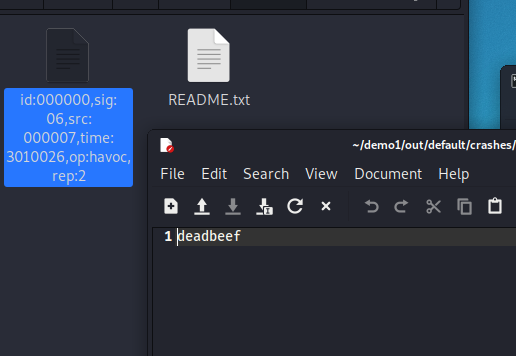
如图，测试字符串“deadbeef”与异常字符串“deadbeef”完全匹配，得到正确的结果（AFL加了插桩，因此在遍历的时候是有策略的）



**4）分析crash**

观察fuzzing结果，如有crash，定位问题。





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。

**心得体会：**

对于kali—linux的使用和其中的一些指令的运用有了一定的了解

对于afl的多种安装方式和其中可能出现的虚拟机网络问题有了一定的修复经验和异常处理能力

对于在kali-linux当中新建c语言程序文件和利用afl运行有了一定的掌握

对于AFL模糊测试的原理与执行过程有了一定的了解和掌握

知道了AFL插桩在编译过程中插入便于分析的其他代码，使得找寻漏洞更容易

对于afl分析当中出现的crash有了一定的修复能力