《漏洞利用及渗透测试基础》实验报告

姓名：管昀玫 学号：2013750 班级：计科一班

**实验名称：**

AFL模糊测试实验

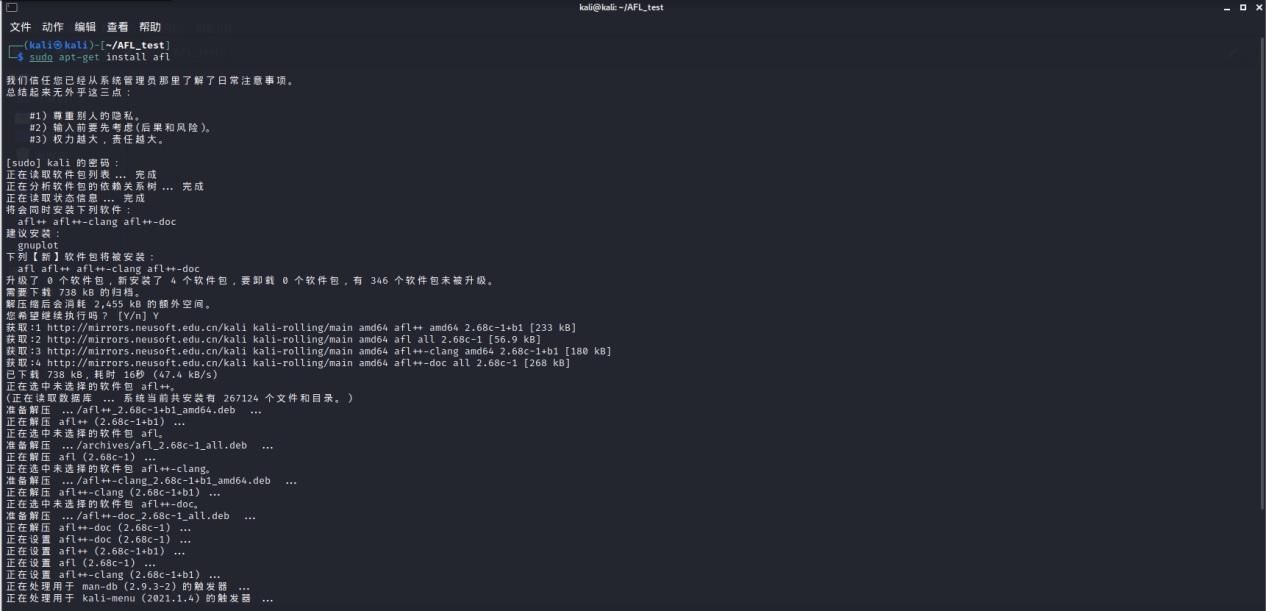
**实验要求：**

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：**

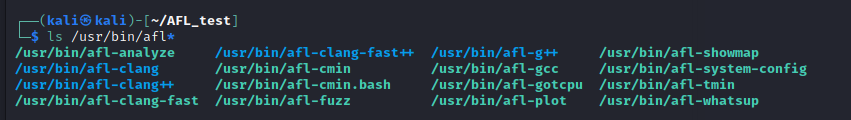
1. kali中的afl安装

打开kali，在主目录下新建一个AFL\_test的文件夹，首先apt-get update，再在该文件夹下用apt-get install的命令安装afl工具，截图如下：



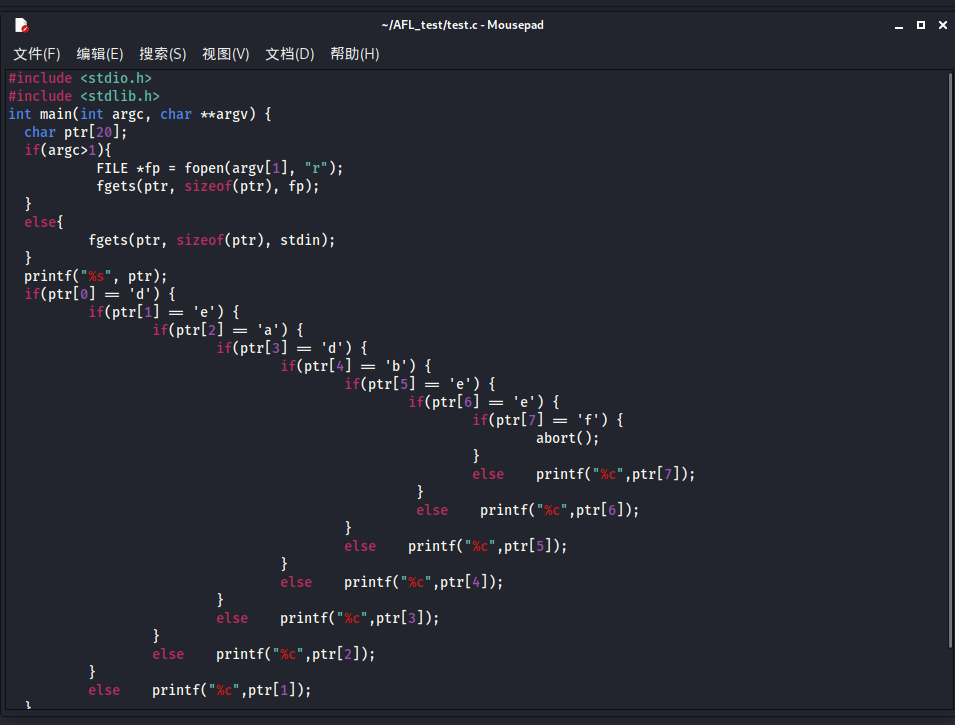
安装完成

查看安装的路径下：



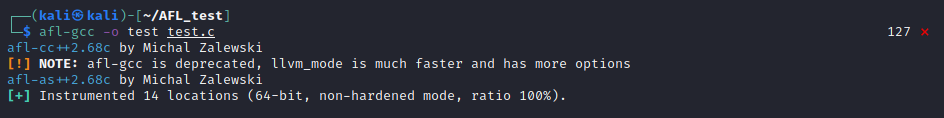
1. 实验准备
   1. 测试代码准备

创建需要测试使用的test.c：



使用afl的编译器编译，可以使模糊测试过程更加高效。

* 1. 编译文件

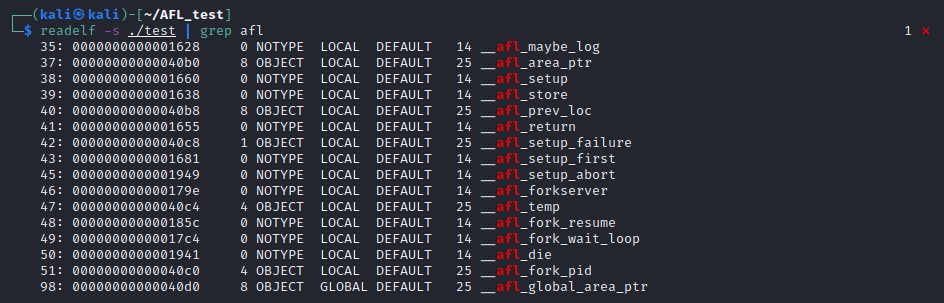


编译成功，可以看见此时的目录下多出了一个编译产生的文件



其中，afl-gcc和普通的gcc不太一样，afl的gcc在编译的时候就会进行一些插桩操作，编译后会有插桩符号。

查看已经完成的插桩功能：



找到了一些关于上述插桩功能的介绍，参考网站：

<https://blog.csdn.net/weixin_43961839/article/details/108630078>

<https://xz.aliyun.com/t/4628#toc-6>

进行下一步之前，还需要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

* 1. 日志等输出目录准备



这里因为只用sudo一直提示权限不够，而使用su进入到root下的时候密码又总是错误，所以修改了一下root的密码，进入到root下才创建成功

在AFL\_test目录下创建in 和 out目录

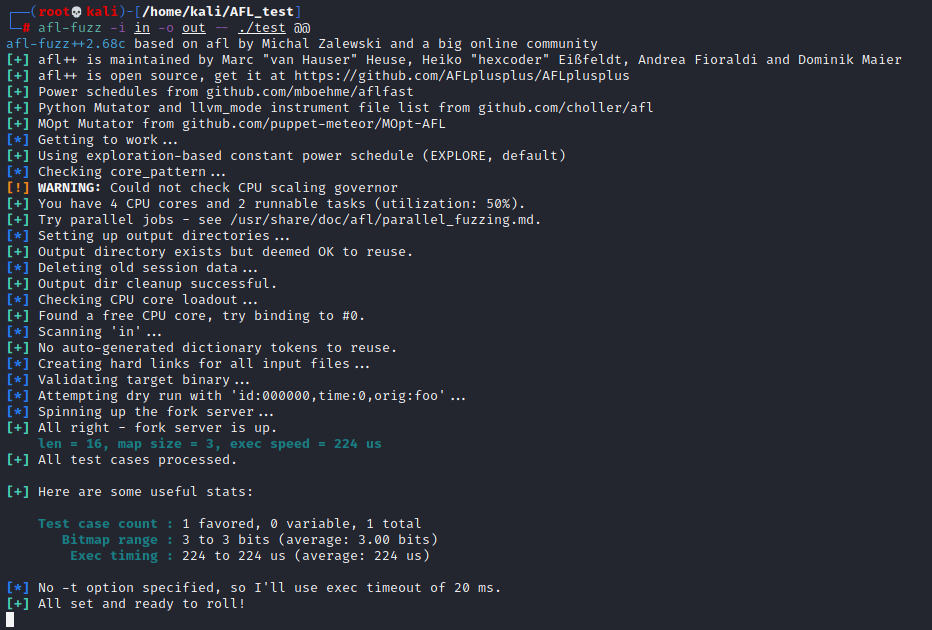


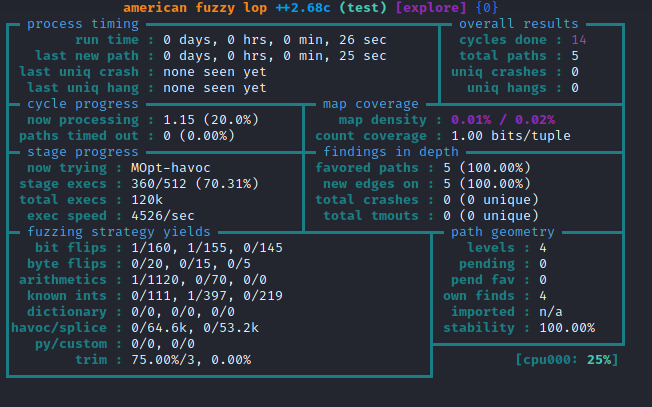
然后，在输入文件夹中创建一个包含字符串“hello”的文件。

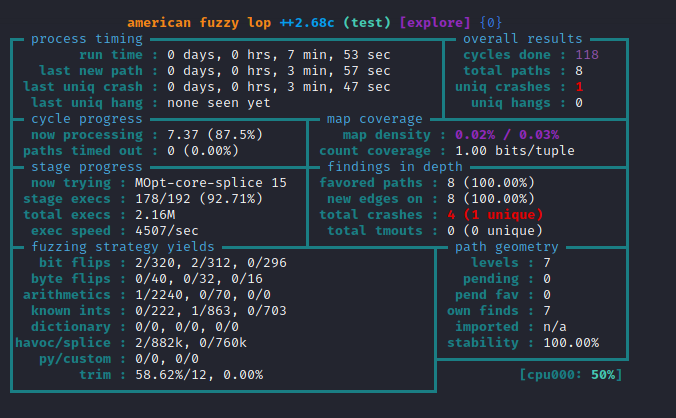
命令：echo hello> in/foo

foo就是我们的测试用例，里面包含初步字符串hello。AFL会通过这个语料进行变异，构造更多的测试用例。

1. 启动测试







通过crashes可以看见，已经找到了触发异常的结果，进入到crashe的文件夹下查看



可以看到是触发异常所需要的条件，模糊测试成功

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。

延展资料：

1. 覆盖引导和文件变异
   1. 覆盖引导

Coverage guided fuzzing (also known as greybox fuzzing) uses program instrumentation to trace the code coverage reached by each input fed to a fuzz target. Fuzzing engines use this information to make informed decisions about which inputs to mutate to maximize coverage.

从上面在google上找到的介绍来看，主要参考的是一个名为“覆盖率”的概念，通过对输入的修改，使得覆盖率达到最高。使用程序工具来跟踪输入到[模糊目标的](https://google.github.io/clusterfuzz/reference/glossary/#fuzz-target)每个输入所达到的代码覆盖率。[模糊引擎](https://google.github.io/clusterfuzz/reference/glossary/#fuzzing-engine)使用此信息来做出明智的决策，以决定要更改哪些输入以最大化覆盖范围。

* 1. 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，下面粗略介绍各个阶段的变异的行为。

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；（翻转也就是0-1,1-0）

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；

interest：    把一些有意思的东西“interesting values”对文件内容进行替换；

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；

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；

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；

一波变异结束后的文件，会在队列结束后下一轮中继续变异下去。AFL状态栏右上角的  cycles done  意味着完成的循环数，每次循环是对整个队列的再一次变异，不过只有第一次 cycle 才会进行  deterministic fuzzing ，之后循环的只有随机性变异了

**心得体会：**

通过这次实验，我对AFL有了更进一步的认识：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。已经发现了大量的零日漏洞。

其工作流程大致如下：

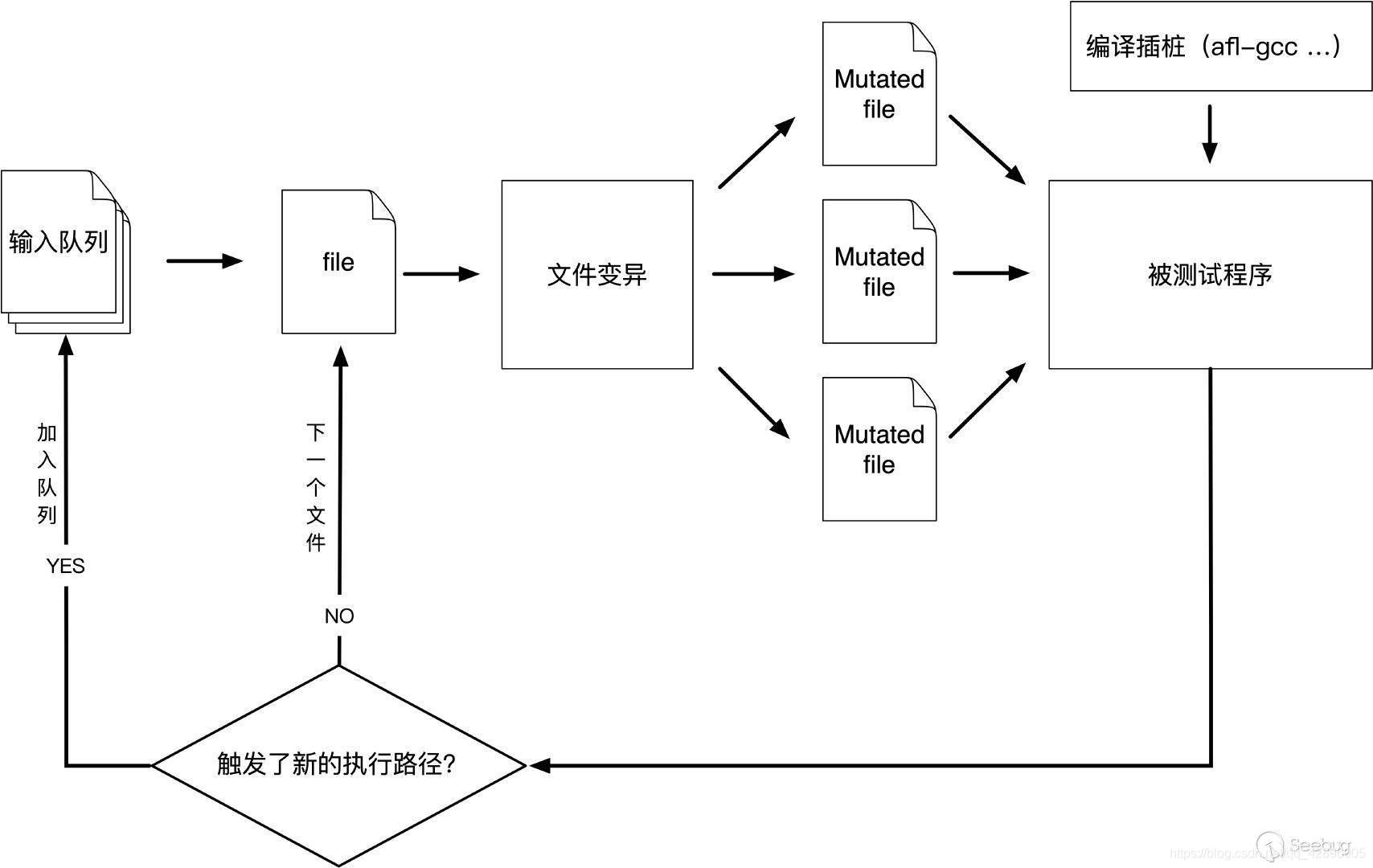
①从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）；

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



除此之外，我对Coverage guide和文件变异策略有了一定的了解，但是了解还不多、不深，并不是很理解其原理。希望以后能更加深入地学习。