软件安全实验报告

姓名：郭子涵 学号:2312145 班级：信息安全、法学双学位班

报告目录:

1 实验名称

2 实验要求

3 实验背景

4 实验内容

4.1 AFL安装

4.2 代码编写与编译

4.3 AFL测试准备

4.4 启动模糊测试

5 覆盖引导和文件变异

6 心得体会

1 实验名称

AFL模糊测试实验

2 实验要求

根据课本7.4.5章节，复现AFL在KALI下得安装、应用，查阅资料理解覆盖引导和文件变异得概念和含义。

3 实验背景

模糊测试的核心思想是，根据一定的规则，自动或半自动生成的随机数据，然后将产生的数据输入到程序中，并监视程序是否有异常出现，以发现可能的程序错误，如内存泄漏、系统崩溃、未处理的异常等。当一个模糊测试生成器开始启动并运行后，它将自己寻找漏洞，并不需要人工干预，非常有助于发现传统测试方法或手动审计无法检测到的缺陷。

模糊测试包括几个基本的测试步骤：确定被测系统->给定输入->生成测试用例->灌入用例进行测试->监控目标程序情况->输出崩溃日志。

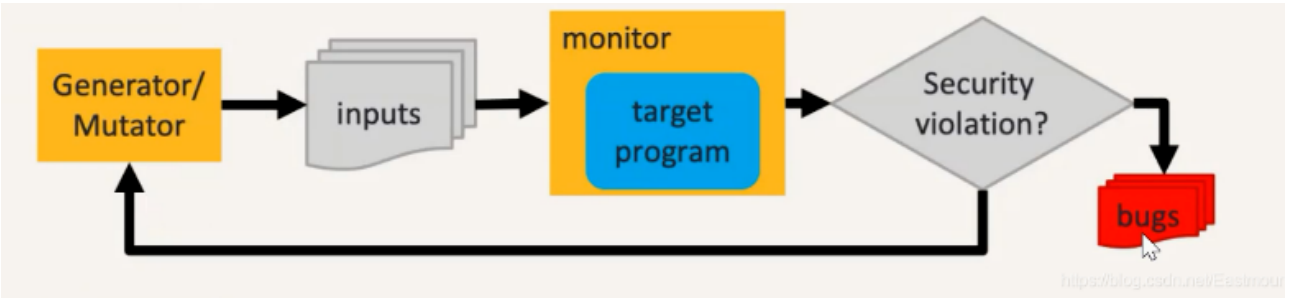


图1：模糊测试基本流程

当前已经有很多开源的模糊测试工具，其中使用较为广泛的是AFL（American Fuzzy Lop），AFL是由安全研究员Michal Zalewski开发的一款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

在执行前，需要对被测程序源码进行插桩，以获知被测程序的运行信息。在执行过程中，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；
2. 选择一些输入文件，作为初始测试集加入输入队列；
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. 上述过程一直循环进行，期间触发crash的文件会被记录下来。

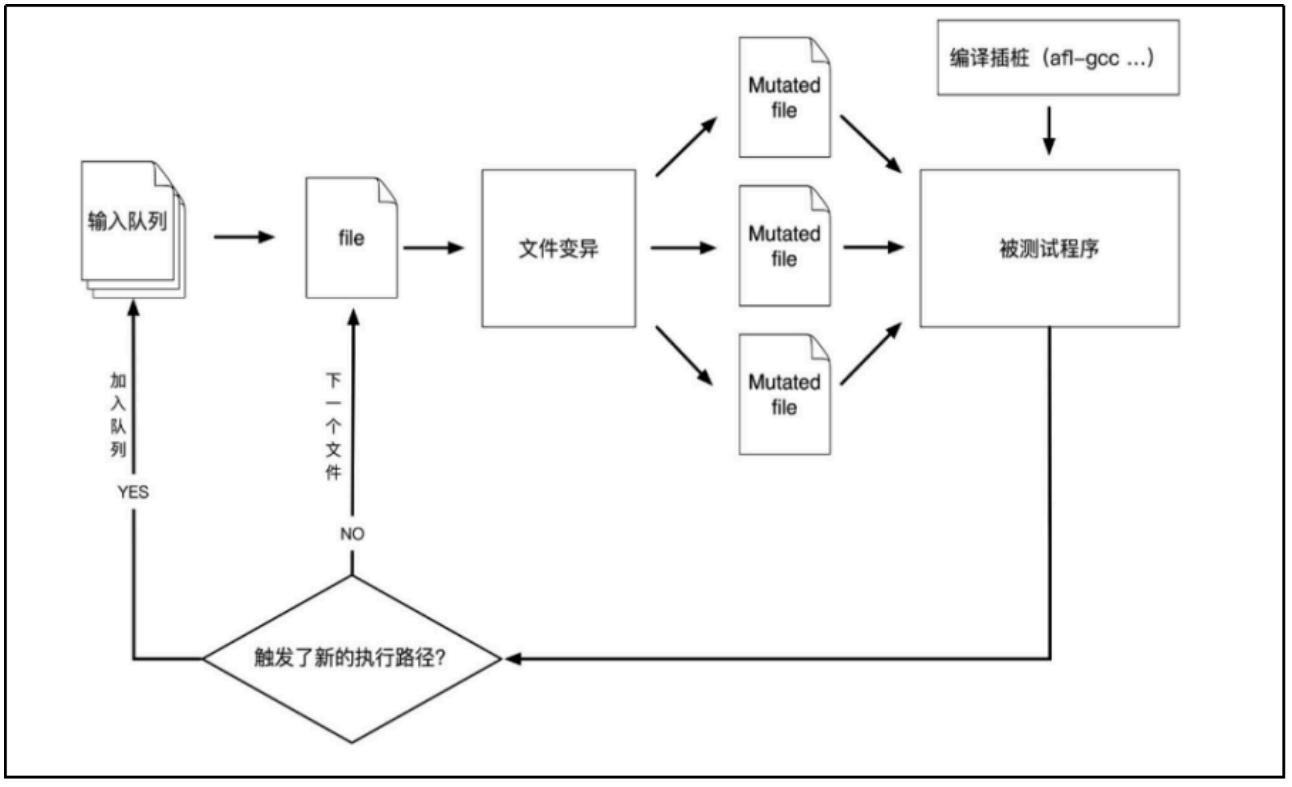


图2：AFL模糊测试基本流程

模糊测试用例的生成算法主要有两种：

1）基于变异：根据已知数据样本，通过变异的方法生成新的测试用例；例如对一个图片文件进行变异，用户需要提供一个相应格式的图片文件，变异生成器会基于该图片进行变异。著名的开源模糊测试工具AFL就是基于变异生成用例。

2）基于生成：根据已知的协议或接口规范，建模并生成测试用例；某些程序可能对输入有严格的规则要求，例如必须是SQL语句、或者给定的协议规范等。测试引擎需要在测试前预先学习对应的语法语义规则，对其进行建模，在此基础上才能变异出有效的测试用例

AFL是采用遗传算法基于变异生成的测试用例，变异的主要类型有下面这几种：

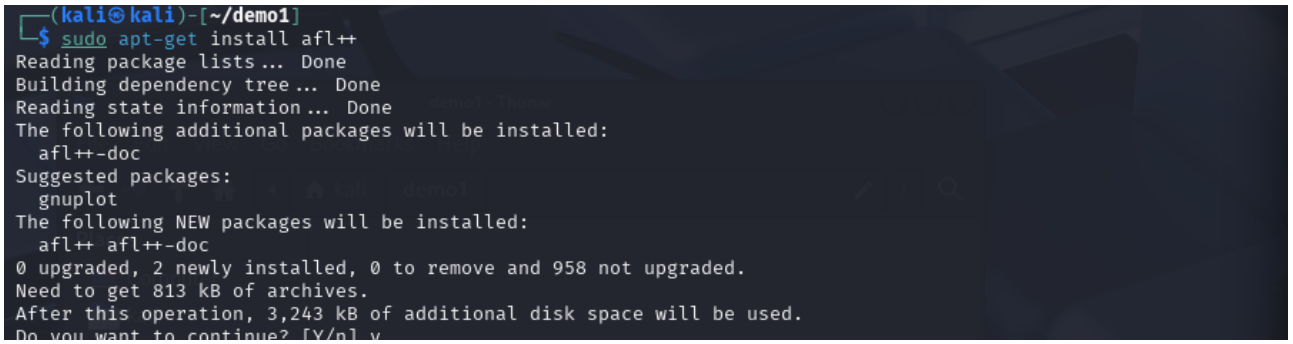
1. Bit flip，按位翻转，1变为0，0变为1
2. Arithmetic，整数加/减算术运算
3. Interest，把一些特殊内容替换到原文件中
4. Dictionary，把自动生成或用户提供的token替换或插入到原文件中
5. Havoc，又称“大破坏”，是前面几种变异的组合
6. Splice，又称“绞接”，将两个文件拼接起来得到一个新文件

AFL需要一些初始输入数据（也称种子文件）作为模糊测试的起点，这些输入可以是毫无意义的数据，本次实验以“hello”为种子文件。AFL通过上述方式自动确定文件的格式和结构。当输入队列中的全部文件都完成变异测试，则完成了一个周期，如果用户不停止执行，种子文件将会不断变异下去

4 实验内容

4.1 AFL安装

由于系统源中afl已经不在维护，被新的afl++所取代，因此我们安装afl++



查看AFL的可执行文件，如下图所示，作用分别是：

- 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

```
┌──(kali㉿kali)-[~/demo1]
└─$ ls /usr/bin/afl*
/usr/bin/afl-addseeds       /usr/bin/afl-gcc-fast
/usr/bin/afl-analyze        /usr/bin/afl-g++-fast
/usr/bin/afl-c++            /usr/bin/afl-gotcpu
/usr/bin/afl-cc             /usr/bin/afl-ld-lto
/usr/bin/afl-clang          /usr/bin/afl-lto
/usr/bin/afl-clang++        /usr/bin/afl-lto++
/usr/bin/afl-clang-fast     /usr/bin/afl-network-client
/usr/bin/afl-clang-fast++   /usr/bin/afl-network-server
/usr/bin/afl-clang-lto      /usr/bin/afl-persistent-config
/usr/bin/afl-clang-lto++    /usr/bin/afl-plot
/usr/bin/afl-cmin           /usr/bin/afl-showmap
/usr/bin/afl-cmin.bash      /usr/bin/afl-system-config
/usr/bin/afl-fuzz           /usr/bin/afl-tmin
/usr/bin/afl-g++            /usr/bin/afl-whatsup
/usr/bin/afl-gcc
```

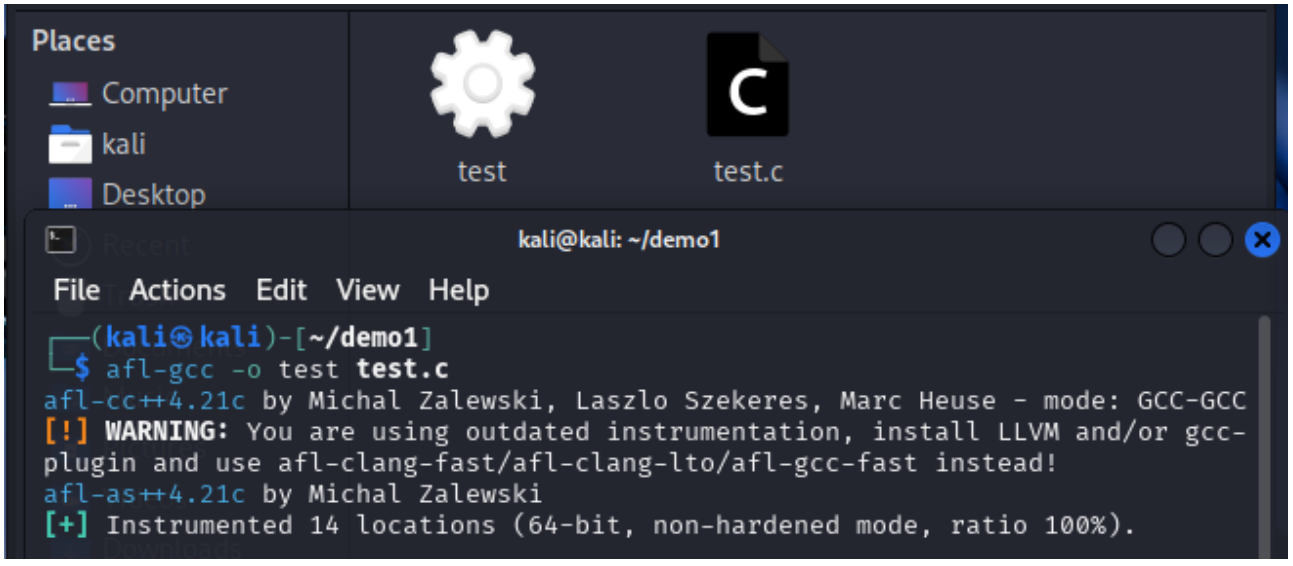
4.2 代码编写与编译

编写test.c文件如下，本段代码从文件或标准输入读取最多19 个字符（char ptr[20]）打印读取的字符串如果读取到的字符串前8个字符依次为 'd' 'e' 'a' 'd' 'b' 'e' 'e' 'f'，也就是字符串 **"deadbeef"**，程序会调用**abort() —— 人为制造崩溃**否则，它会打印出检查失败的那个字符（从 ptr[0]到ptr[7]依次检查）

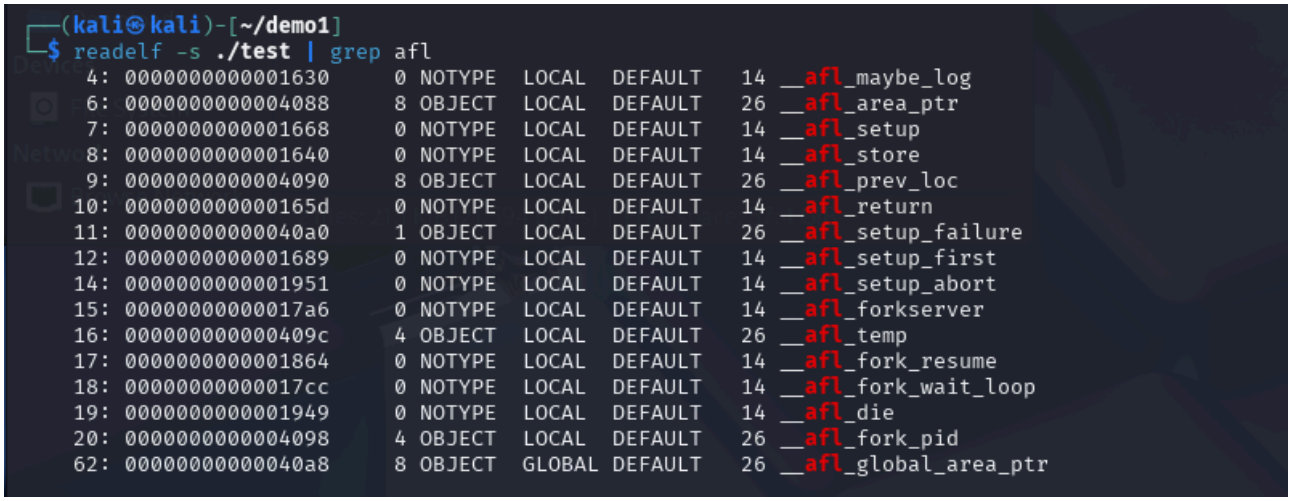
```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char ptr[20];
    if(argc>1){
        FILE *fp = fopen(argv[1], "r");
        fgets(ptr, sizeof(ptr), fp);
    }
    else{
        fgets(ptr, sizeof(ptr), stdin);
    }
 printf("%s", ptr);
 if(ptr[0] == 'd') {
    if(ptr[1]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

使用afl-gcc，这是一个包含插桩功能的编译器，输入 **afl-gcc -o test test.c** 命令编译生成test文件



使用 **readelf -s ./test | grep afl** 命令验证， afl-gcc编译后，在 ELF 符号表中看到的一系列以 `__afl_` 开头的符号，都是 AFL 自动插桩生成的辅助函数和变量，它们是 AFL 在程序中嵌入的**覆盖率跟踪、分支记录和 forkserver 支持机制的一部分**。



主要分为四大类：

1. **afl__area__ptr，**afl__prev__loc，___afl__global__area__ptr这些变量用于路径覆盖率的计算，AFL 通过这些变量来判断程序是否走了“新路径”。

2. **afl__maybe__log，**afl_store，___afl_return这些函数是 AFL 插桩进程在每个基本块插入的逻辑，用于执行路径记录。
3. **afl__forkserver，**afl_fork_resume，**afl__fork__wait__loop，**afl_fork_pid用来避免频繁启动新进程，极大提升 fuzzing 效率。
4. 以及一些其他辅助控制项

4.3 AFL测试准备

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。

```
┌──(kali㉿kali)-[~/demo1]
└─$ echo core | sudo tee /proc/sys/kernel/core_pattern

[sudo] password for kali:
core
```

创建in和out两个文件夹，分别存储模糊测试所需的输入和输出相关的内容，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，AFL会通过这个语料进行变异，构造更多的测试用例。

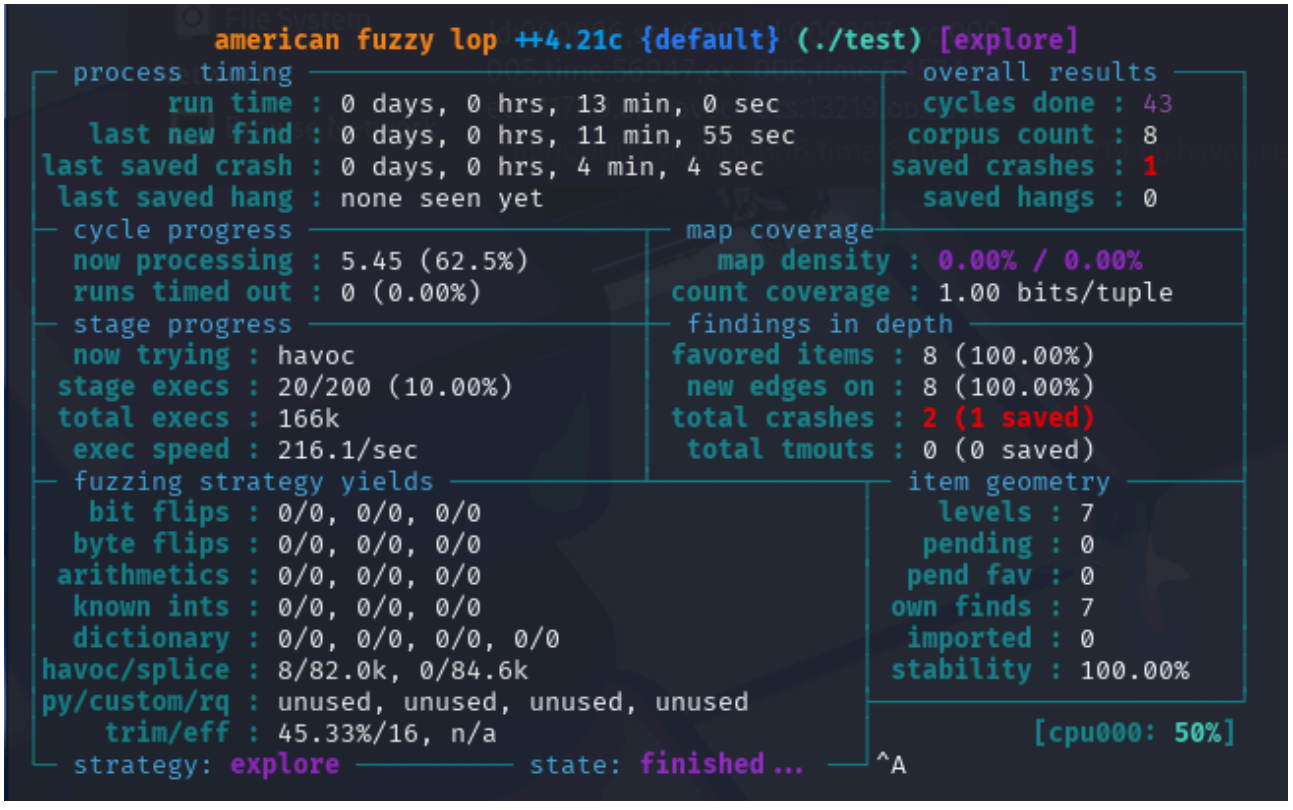
```
┌──(kali㉿kali)-[~/demo1]
└─$ mkdir in out

┌──(kali㉿kali)-[~/demo1]
└─$ echo hello> in/foo
```

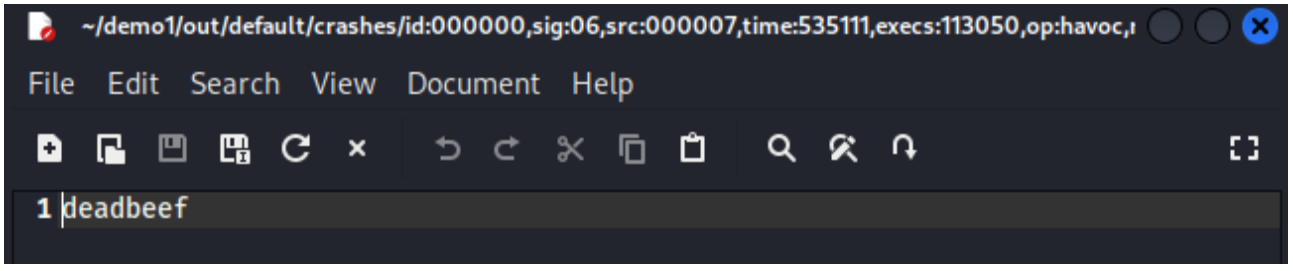
4.4 启动模糊测试

运行命令：**afl-fuzz -i in -o out -- ./test @@**执行测试，可以看到如下界面：

1. 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。
2. 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，继续 fuzzing 下去也不会有什么新的发现。此时，可以通过 Ctrl-C，中止当前的 fuzzing。
3. stage progress：此窗口不断发生变化，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。执行速度直观地反映当前执行快慢，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。

观察Queue文件夹，发现其由……->dead->deadbead->deadbeea不断向我们想找的'crash'逼近，最终在total crashes变红时说明已经找到可以使得程序崩溃的'crash'





与我们程序设计的'deadbeef'相同,验证成功！

5 覆盖引导和文件变异

在本次 AFL 模糊测试实验中，我们接触和实践了两个模糊测试中非常核心的技术概念：**覆盖引导与文件变异**。这两者不仅是 AFL 的工作基础，更体现了现代模糊测试工具从盲目测试走向智能化漏洞挖掘的关键转变。

覆盖引导是通过“程序覆盖信息”来指导测试输入的生成。AFL 在编译阶段利用 **afl-gcc** 对目标程序插桩，嵌入监控代码，使得在每次执行测试用例时，它能够获取程序的执行路径信息，比如哪些基本块被执行，哪些分支被触发。实验中我们使用 **readelf -s ./test | grep afl** 命令、 查看了插桩后生成的符号信息，如 **afl__maybe__log 、afl__area__ptr** 等，这些都是 AFL 插入的用于记录覆盖信息的符号。运行 **afl-fuzz** 命令后，AFL 会不断检测每个变异输入是否能带来新的执行路径，如果某个输入覆盖了之前未到达的代码路径，它就会被认为是“有价值”的，并被保留用于后续更多的变异操作。这种机制极大提高了模糊测试的效率和有效性，因为 AFL 不是盲目尝试所有输入，而是优先探索可能引发更多执行路径的输入。

文件变异是 AFL 模糊测试的另一个核心过程。在实验中，我们指定了输入文件目录 **-i in**，其中放置了初始测试用例**“hello”**。AFL 会从这个种子出发，对其进行各种变异，比如比特翻转、字节插入、删除、复制、边界值替换、算术操作等，生成大量“类似但不同”的新测试用例。变异后的输入再被注入程序中执行，利用覆盖信息判断是否值得保留。变异过程完全自动进行，通过 @@ 占位符自动将输入重定向到程序中，这种机制不仅支持标准输入，还支持文件输入，适应多种程序结构。在我们的实验中，虽然程序本身逻辑可能简单，但通过反复变异与覆盖引导，AFL 仍能不断生成多样化输入，尝试触发不同分支或潜在的崩溃点。

总体来看，这次实验清晰展现了 AFL 模糊测试流程：首先通过插桩方式收集路径覆盖信息（覆盖引导），然后利用初始输入进行高强度的输入扰动（文件变异），再结合覆盖反馈决定下一步的输入演化方向。这种“反馈驱动、自动迭代”的测试方式，使得 AFL 不仅适用于发现浅层逻辑漏洞，更能在无需源码情况下发现复杂的安全隐患。这种自动化与智能化的结合是模糊测试从传统测试走向实用安全工具的关键，也是我们从这次实验中所体会到的最大价值。

6 心得体会

通过本次AFL模糊测试实验，我深入了解了模糊测试技术的核心原理以及AFL工具的使用方式。实验从源码编译、插桩测试程序入手，通过 **afl-gcc** 对目标程序进行编译，插入必要的监测逻辑，以便 AFL 能追踪程序的路径覆盖情况。在使用 **afl-fuzz** 进行测试时，我学会了设置输入种子目录与输出目录，并正确使用命令行参数传递待测程序和输入文件，通过 @@ 占位符实现输入重定向。

过程中遇到的错误，比如命令格式不规范、权限不足等问题，也促使我更深入理解 Linux 系统和Kali虚拟机的运行机制，特别是与 **/proc** 系统调用有关的部分。通过分析 **readelf** 输出，我也初步掌握了 AFL 在二进制文件中插入的符号机制，如 **afl__area__ptr 和** afl_prev_loc等，这些是 AFL 跟踪路径覆盖的关键。

整体实验不仅提升了我对软件安全测试技术的理解，也锻炼了我在实际环境下配置与调试工具的能力，更让我意识到自动化漏洞发现技术在现代软件开发中的重要性与实用价值。