### AFL模糊测试工具

AFL（American Fuzz Lop）是当前最前沿、最先进的模糊测试工具之一，它是由谷歌员工Michal Zalewski开发，支持多平台（ARM、X86、X64）、多系统（Linux、BSD、Windows、MacOS）。自2013年发布以来，它以有效、快速、稳定、易用的优点在安全领域广受称赞。据AFL官网粗略统计，AFL在125套不同种类的著名软件中，发现软件漏洞达260个之多。它使用简便，不需要先行复杂的配置，能无缝处理复杂的现实程序。

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。AFL主要用于C/C++程序的测试，被测程序有无程序源码均可，有源码时可以对源码进行编译时插桩，无源码可以借助QEMU的User-Mode模式进行二进制插装。

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。

流程图如下图7-21所示：

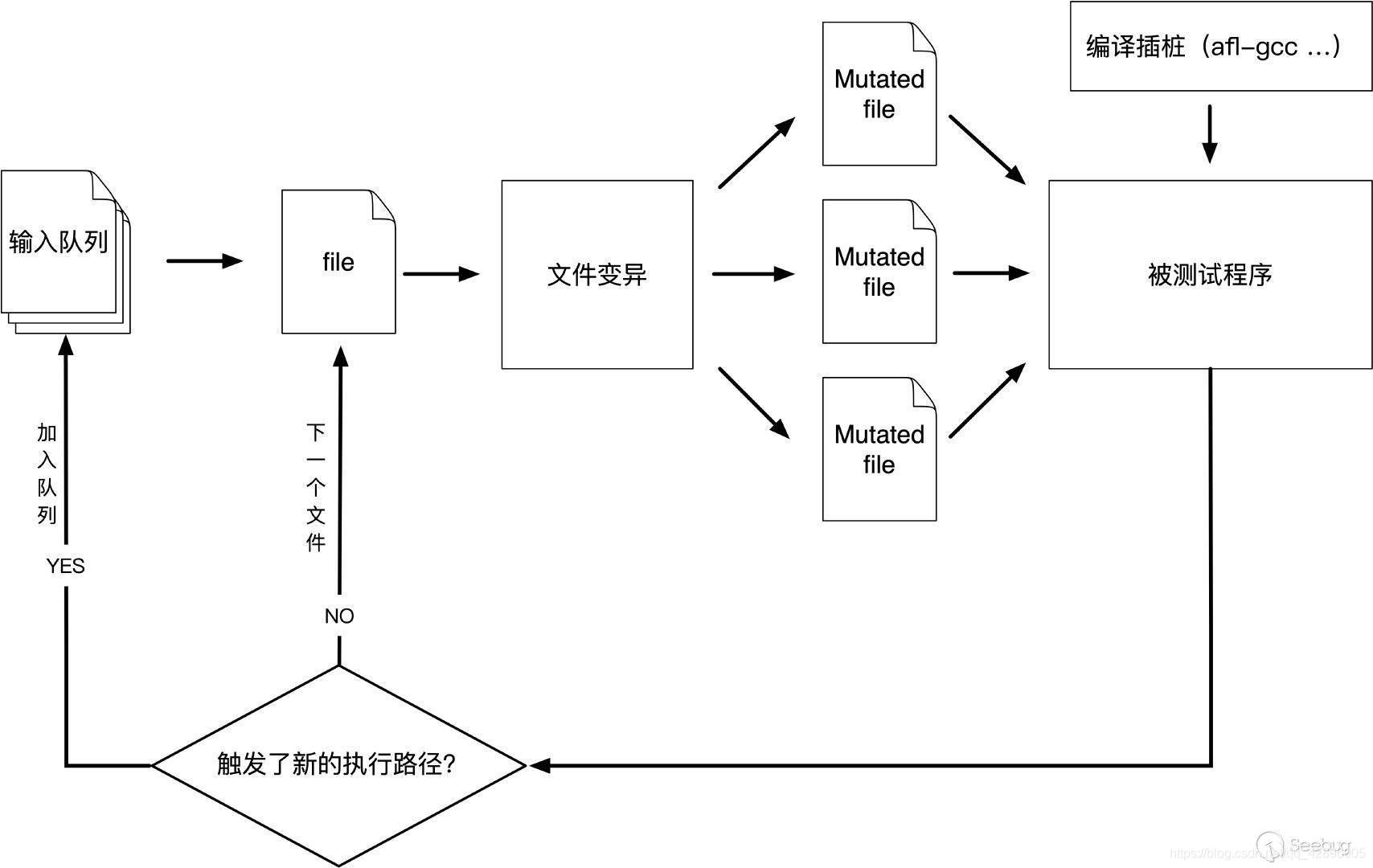


图7-21 AFL工具工作流程图

**1. AFL安装**

在Kali 2021下，利用sudo apt-get install afl即可安装。

查看路径可以看到afl安装的文件：ls /usr/bin/afl\*，如图7-22

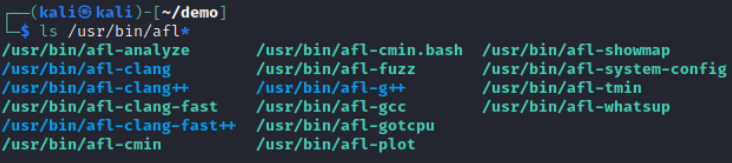


图7-22 查看afl安装的文件

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。

**2. AFL测试**

AFL可以进行白盒测试，也可以进行黑盒测试。

接下来，以一个白盒模糊测试为例，讲解AFL的简单用法。

**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，该代码编译后得到的程序如果被传入“deadbeef”则会终止，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;  } |

使用afl的编译器编译，可以使模糊测试过程更加高效。

命令：**afl-gcc -o test test.c**

编译后会有插桩（详见第8.2章）符号，使用下面的命令可以验证这一点。

命令：**readelf -s ./test | grep afl，**如图7-23：

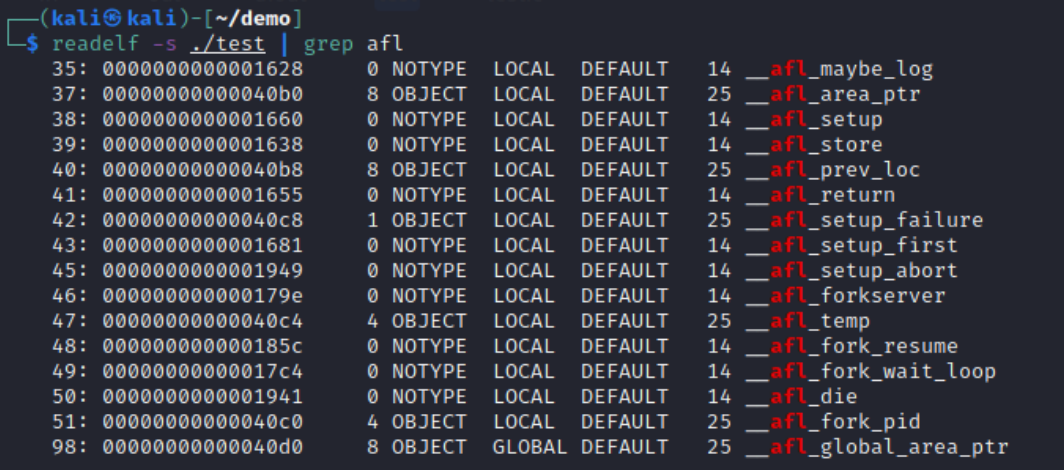


图7-23 验证编译后会有编译后会有插桩

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**

**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**

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。

**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 @@**

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 @@

启动模糊测试后，可以看到如下图7-24的运行界面：

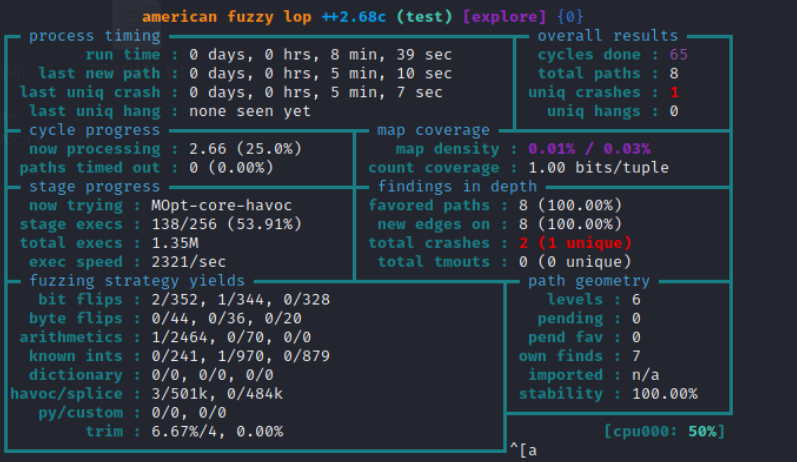


图7-24 WinAFL运行界面

下面对界面进行介绍：

**·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。

**4）分析crash**

观察fuzzing结果，如有crash，定位问题。如图7-25

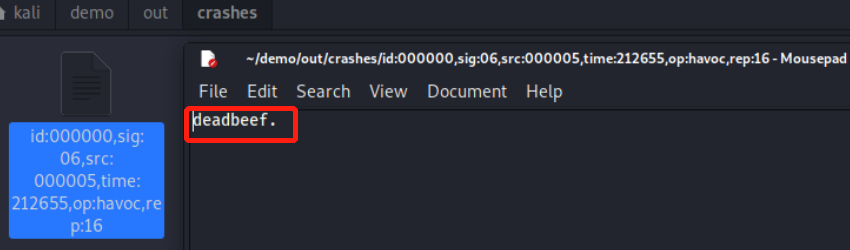


图7-25 out文件夹下crashes子文件夹包含产生crash的样例

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。

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。

**注意：本书只是简单的描述了AFL框架的用法，实际上，AFL模糊测试框架功能非常强大、非常有用，已经发现了大量的零日漏洞，得到了众多研究学者的关注。**