《软件安全》实验报告

姓名： 邢清画　　学号：2211999　　班级：1023（物联网）

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. 打开虚拟机进入 Kali Linux 系统，新建 demo1，启动控制台，安装 AFL。

利用 sudo apt-get install afl，发现无法安装，因为 Linux 的 AFL 已经停止更新。

使用 wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz 进行安装。

```
┌──(kali㉿kali)-[~/demo1]
└─$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz                                                    4 ⨯
--2024-05-07 09:29:36--  http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
Resolving lcamtuf.coredump.cx (lcamtuf.coredump.cx)... 172.67.180.222, 104.21.83.192, 2606:4700:3036::ac43:b4de, ...
Connecting to lcamtuf.coredump.cx (lcamtuf.coredump.cx)|172.67.180.222|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz [following]
--2024-05-07 09:29:37--  https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
Connecting to lcamtuf.coredump.cx (lcamtuf.coredump.cx)|172.67.180.222|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 835907 (816K) [application/x-gzip]
Saving to: 'afl-latest.tgz'

afl-latest.tgz               100%[=============================================================>] 816.32K   666KB/s    in 1.2s

2024-05-07 09:29:39 (666 KB/s) - 'afl-latest.tgz' saved [835907/835907]


┌──(kali㉿kali)-[~/demo1]
└─$
```

下载完成之后进行解压：

```
┌──(kali㉿kali)-[~/demo1]
└─$ tar xvf afl-latest.tgz
afl-2.52b/
afl-2.52b/afl-as.h
afl-2.52b/libtokencap/
afl-2.52b/libtokencap/libtokencap.so.c
afl-2.52b/libtokencap/README.tokencap
afl-2.52b/libtokencap/Makefile
afl-2.52b/alloc-inl.h
afl-2.52b/config.h
afl-2.52b/test-instr.c
afl-2.52b/afl-analyze.c
afl-2.52b/README
afl-2.52b/afl-showmap.c
afl-2.52b/experimental/
afl-2.52b/experimental/clang_asm_normalize/
afl-2.52b/experimental/clang_asm_normalize/as
afl-2.52b/experimental/README.experiments
afl-2.52b/experimental/distributed_fuzzing/
afl-2.52b/experimental/distributed_fuzzing/sync_script.sh
afl-2.52b/experimental/crash_triage/
afl-2.52b/experimental/crash_triage/triage_crashes.sh
afl-2.52b/experimental/libpng_no_checksum/
afl-2.52b/experimental/libpng_no_checksum/libpng-nocrc.patch
afl-2.52b/experimental/bash_shellshock/
afl-2.52b/experimental/bash_shellshock/shellshock-fuzz.diff
afl-2.52b/experimental/canvas_harness/
```

安装成功后，利用

cd afl-2.52b > ls > sudo make && sudo make install > ls /usr/local/bin > ls /usr/local/bin/afl*

这些语句可以找到 AFL 的一些可执行文件

```
┌──(kali㉿kali)-[~/demo1]
└─$ cd afl-2.52b                                                                                    2

┌──(kali㉿kali)-[~/demo1/afl-2.52b]
└─$ ls
afl-analyze.c  afl-fuzz.c    afl-showmap.c  config.h      experimental   llvm_mode           README
afl-as.c       afl-gcc.c     afl-tmin.c     debug.h       hash.h         Makefile            testcases
afl-as.h       afl-gotcpu.c  afl-whatsup    dictionaries  libdislocator  qemu_mode           test-instr.c
afl-cmin       afl-plot      alloc-inl.h    docs          libtokencap    QuickStartGuide.txt types.h
```

```
┌──(kali㉿kali)-[~/demo1/afl-2.52b]
└─$ sudo make && sudo make install
[sudo] password for kali:
[*] Checking for the ability to compile x86 code...
[+] Everything seems to be working, ready to compile.
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
r/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-gcc.c -o afl-gcc -ldl
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc $i; done
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
r/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-fuzz.c -o afl-fuzz -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
r/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-showmap.c -o afl-showmap -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
r/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-tmin.c -o afl-tmin -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
r/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-gotcpu.c -o afl-gotcpu -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
r/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-analyze.c -o afl-analyze -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/us
```

```
┌──(kali㉿kali)-[~/demo1/afl-2.52b]
└─$ ls /usr/local/bin
afl-analyze  afl-clang++  afl-fuzz  afl-gcc     afl-plot     afl-tmin
afl-clang    afl-cmin     afl-g++   afl-gotcpu  afl-showmap  afl-whatsup

┌──(kali㉿kali)-[~/demo1/afl-2.52b]
└─$ ls /usr/local/bin/afl*
/usr/local/bin/afl-analyze  /usr/local/bin/afl-cmin  /usr/local/bin/afl-gcc     /usr/local/bin/afl-showmap
/usr/local/bin/afl-clang    /usr/local/bin/afl-fuzz  /usr/local/bin/afl-gotcpu  /usr/local/bin/afl-tmin
/usr/local/bin/afl-clang++  /usr/local/bin/afl-g++   /usr/local/bin/afl-plot    /usr/local/bin/afl-whatsup
```

2. 在 demo1 文件夹中新建文本文件 test.c，复制实验代码。该代码编译后得到的程序如果传入“deadbeef”则会终止，如果输入其他字符则会原样输出。代码如下图所示：

*~/demo1/test.c - Mousepad

File Edit Search View Document Help

```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]);
 }
```

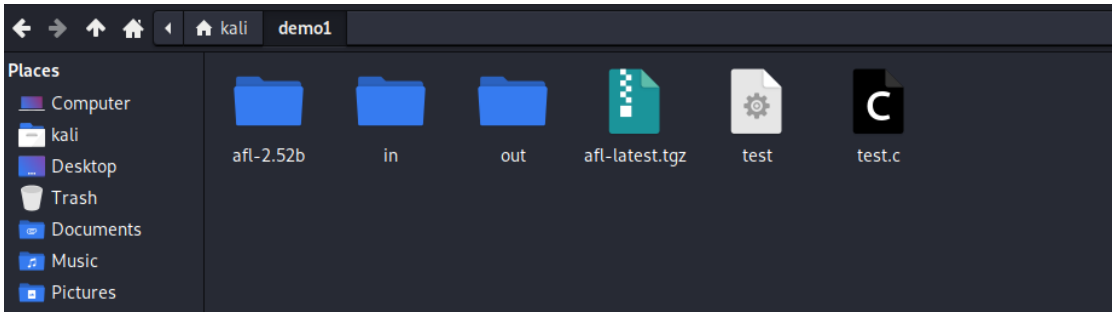
使用 afl 编译器，用 afl-gcc 命令 afl-gcc -o test test.c 创建本次的实验文件 test:

```
┌──(kali㉿kali)-[~/demo1]
└─$ afl-gcc -o test test.c
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 14 locations (64-bit, non-hardened mode, ratio 100%).
```

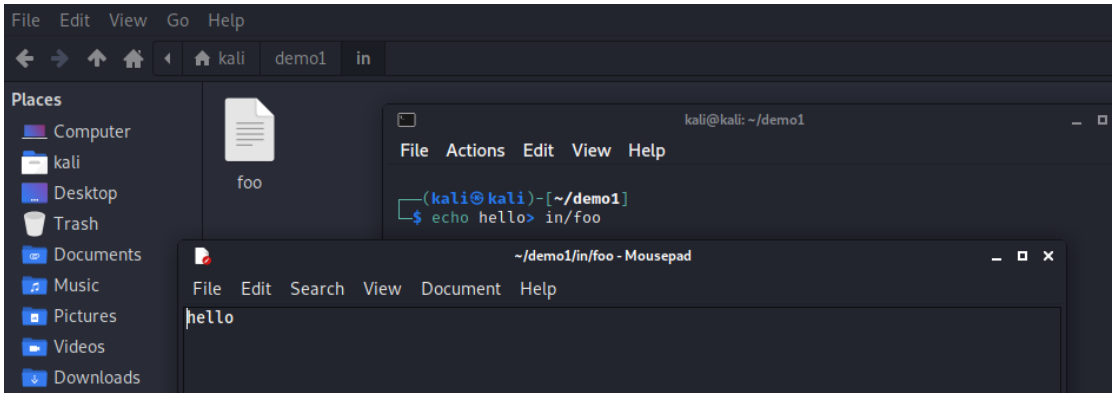
编译之后出现插桩，输入命令：readelf -s ./test | grep afl

```
┌──(kali㉿kali)-[~/demo1]
└─$ readelf -s ./test | grep afl                                        1 ×
    35: 0000000000001628     0 NOTYPE  LOCAL  DEFAULT   14 __afl_maybe_log
    37: 00000000000040b0     8 OBJECT  LOCAL  DEFAULT   25 __afl_area_ptr
    38: 0000000000001658     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup
    39: 0000000000001638     0 NOTYPE  LOCAL  DEFAULT   14 __afl_store
    40: 00000000000040b8     8 OBJECT  LOCAL  DEFAULT   25 __afl_prev_loc
    41: 0000000000001650     0 NOTYPE  LOCAL  DEFAULT   14 __afl_return
    42: 00000000000040c8     1 OBJECT  LOCAL  DEFAULT   25 __afl_setup_failur
e
    43: 0000000000001679     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup_first
    45: 000000000000193e     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup_abort
    46: 0000000000001793     0 NOTYPE  LOCAL  DEFAULT   14 __afl_forkserver
    47: 00000000000040c4     4 OBJECT  LOCAL  DEFAULT   25 __afl_temp
    48: 0000000000001851     0 NOTYPE  LOCAL  DEFAULT   14 __afl_fork_resume
    49: 00000000000017b9     0 NOTYPE  LOCAL  DEFAULT   14 __afl_fork_wait_lo
op
    50: 0000000000001936     0 NOTYPE  LOCAL  DEFAULT   14 __afl_die
    51: 00000000000040c0     4 OBJECT  LOCAL  DEFAULT   25 __afl_fork_pid
    98: 00000000000040d0     8 OBJECT  GLOBAL DEFAULT   25 __afl_global_area_
ptr
```

创建输入（in）输出（out）文件夹，用来存储测试所需的相关输入输出信息



使用命令 echo hello> in/foo，在 in 文件夹中写入 hello



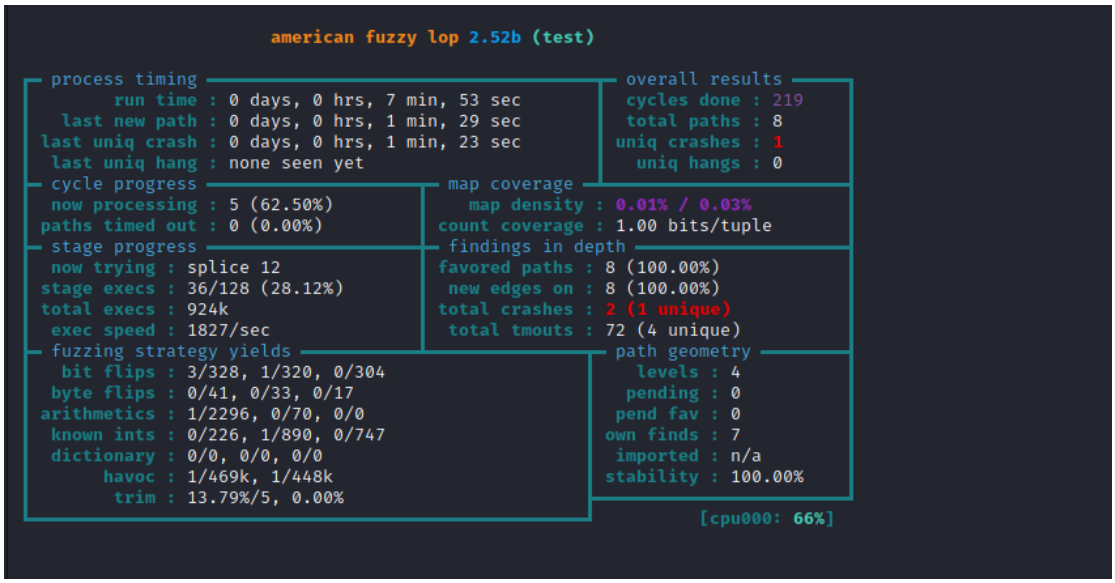
3. 有输入文件之后启动测试

运行 afl-fuzz -i in -o out -- ./test @@ 命令，开始启动模糊测试：

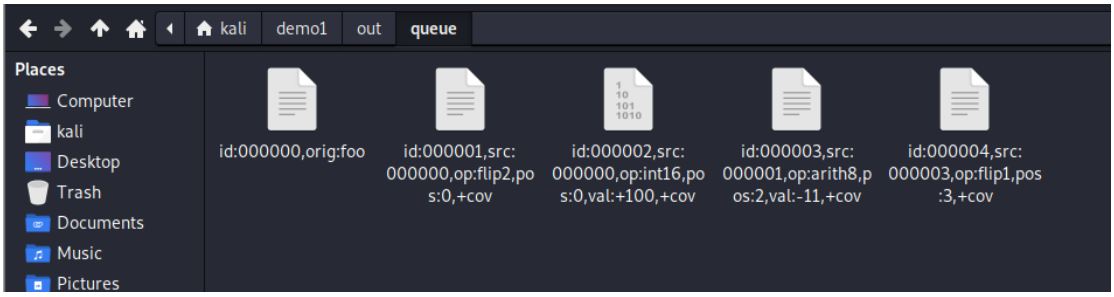
```
┌──(kali㉿kali)-[~/demo1]
└─$ afl-fuzz -i in -o out -- ./test @@
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 4 CPU cores and 1 runnable tasks (utilization: 25%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:foo'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 6, map size = 3, exec speed = 584 us
[+] All test cases processed.

[+] Here are some useful stats:
```

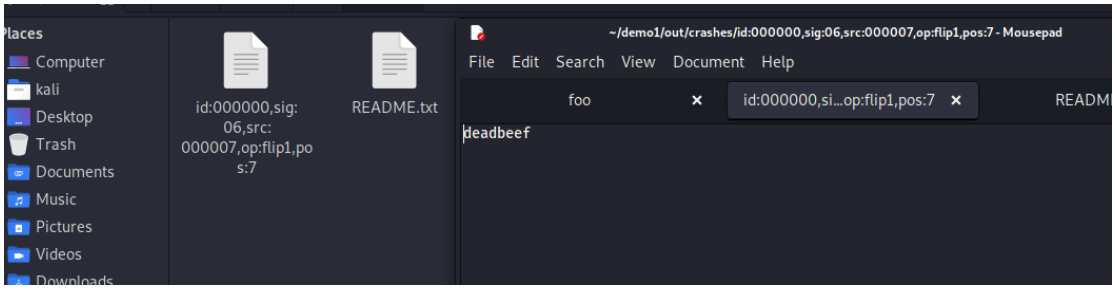
运行一段时间后，发现 total crashes 变为非 0 个数，即出现中断



在队列中可以看到已经产生了很多关于触发异常的记录



在 out 文件夹中，打开 crashes 文件夹，打开错误日志，观察到字符串“deadbeef”。



心得体会：

1. 覆盖引导与文件变异的理解

在 AFL 模糊测试的上下文中，“覆盖引导”（coverage-guided）不同于面向对象编程中的覆盖概念。在 AFL 中，覆盖引导是指利用程序执行时收集的路径覆盖信息来优化后续的测试用例生成。这种方法能够有效地指导模糊测试工具生成那些探索新代码路径的测试数据，从而提高测试的全面性和发现错误的概率。

文件变异在 AFL 中指的是基于现有测试用例生成新测试用例的过程，其中原始的测试数据会经历各种变化，如位翻转、插入、删除等，以产生潜在能触发未探索代码路径的新数据。这些变异操作是自动化的，能够在短时间内生成大量的测试用例。

2. AFL 的工作原理与实验步骤

AFL 通过结合遗传算法和确定性模糊测试技术来有效地生成覆盖新代码路径的测试用例。其工作流程包括：
准备阶段： 对目标程序进行插桩处理，这一步是为了让 AFL 能够在运行时收集必要的执行路径信息。
运行 AFL： 输入插桩后的程序，AFL 开始自动地生成测试用例并执行，不断重复这一过程。
监控与调试： AFL 自动监控程序执行情况，并在探索到新的代码路径时保存相应的测试用例。此外，AFL 提供了多种工具以帮助用户分析和调试测试结果。
漏洞分析： 当 AFL 发现可能导致程序崩溃的输入时，用户需要对这些情况进行分析，确定是否存在安全漏洞，并进行相应的分类和修复。

3. 实验体会

通过这次实验，我不仅加深了对 Linux 系统的熟悉程度，也深入理解了 AFL 模糊测试工具的工作原理及其应用。AFL 通过自动化的测试用例生成和覆盖率优化，极大地提高了发现程序安全漏洞的效率和可能性，相较于传统的手动测试方法，它能更快地识别和修复潜在的安全风险。

总结而言，AFL 是一个强大的模糊测试工具，它的高效和自动化特性使得安全测试更加全面和可靠。通过实际操作和应用，我对模糊测试的理论与实践都有了更深刻的认识和体会。