

《软件安全》实验报告

姓名：刘星宇 学号：2212824 班级：信息安全法学双学位班

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. 配置 Kali2024 环境

在 Kali2024 下, 利用 `sudo apt-get install afl` 安装 AFL

```
(root@kali)-[~]
# sudo apt-get install afl++
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
afl++ is already the newest version (4.08c-1).
0 upgraded, 0 newly installed, 0 to remove and 822 not upgraded.
```

查看 AFL 安装之后利用 `ls /usr/bin/afl*` 可执行的文件

```
(root@kali)-[~]
# ls /usr/bin/afl*
/usr/bin/afl-analyze      /usr/bin/afl-gcc
/usr/bin/afl-c++          /usr/bin/afl-gcc-fast
/usr/bin/afl-cc           /usr/bin/afl-gotcpu
/usr/bin/afl-clang        /usr/bin/afl-ld-lto
/usr/bin/afl-clang++      /usr/bin/afl-lto
/usr/bin/afl-clang-fast   /usr/bin/afl-lto++
/usr/bin/afl-clang-fast++ /usr/bin/afl-network-client
/usr/bin/afl-clang-lto    /usr/bin/afl-network-server
/usr/bin/afl-clang-lto++  /usr/bin/afl-persistent-config
/usr/bin/afl-cmin         /usr/bin/afl-plot
/usr/bin/afl-cmin.bash    /usr/bin/afl-showmap
/usr/bin/afl-fuzz         /usr/bin/afl-system-config
/usr/bin/afl-g++          /usr/bin/afl-tmin
/usr/bin/afl-g++-fast     /usr/bin/afl-whatsup
```

作用分别为：

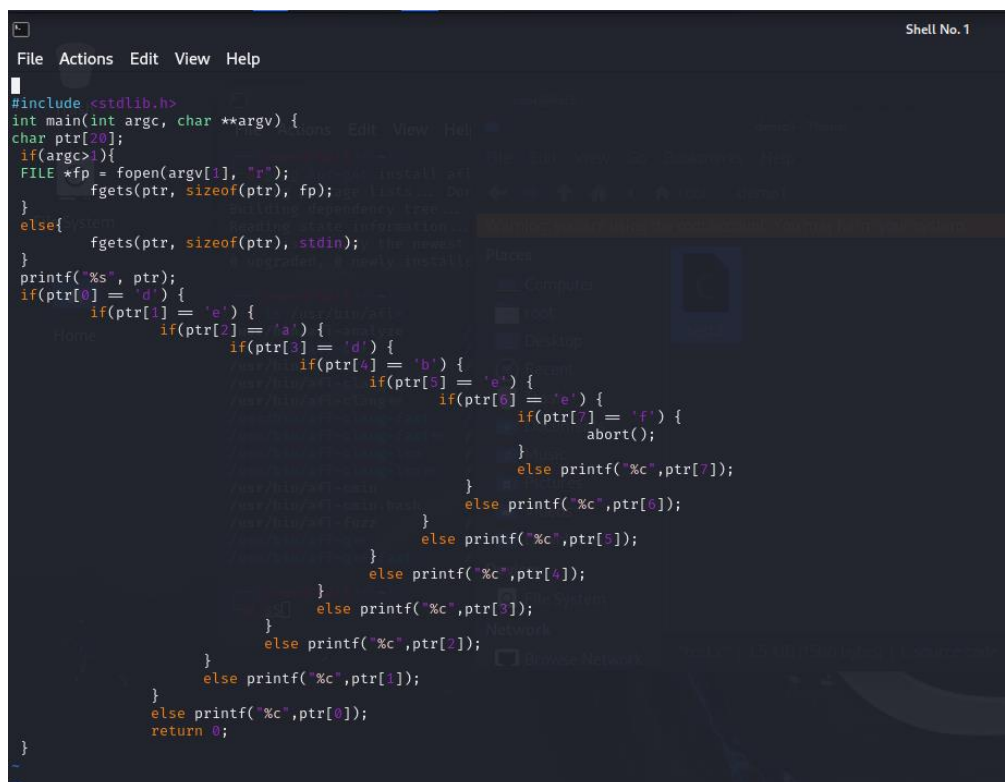
- afl-gcc 和 afl-g++ 分别对应的是 gcc 和 g++ 的封装。
- afl-clang 和 afl-clang++ 分别对应 clang 的 c 和 c++ 编译器封装。
- afl-fuzz 是 AFL 的主体, 用于对目标程序进行 fuzz。
- afl-analyze 可以对用例进行分析, 看能否发现用例中有意义的字段。
- afl-qemu-trace 用于 qemu-mode, 默认不安装, 需要手工执行 qemu-mode 的编译脚本 进行编译。
- afl-plot 生成测试任务的状态图。
- afl-tmin 和 afl-cmin 对用例进行简化。
- afl-whatsup 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

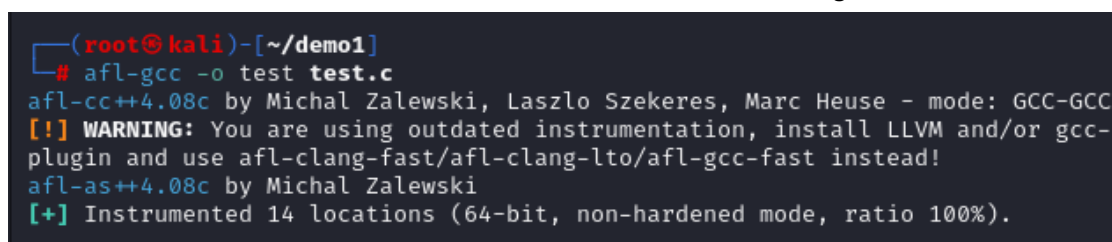
1) 创建本次实验的程序

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 <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[8]);
                        }
                        else printf("%c",ptr[9]);
                    }
                    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**



```
(root@kali)-[~/demo1]
# afl-gcc -o test test.c
afl-cc++4.08c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: GCC-GCC
[!] WARNING: You are using outdated instrumentation, install LLVM and/or gcc-
plugin and use afl-clang-fast/afl-clang-lto/afl-gcc-fast instead!
afl-as++4.08c by Michal Zalewski
[+] Instrumented 14 locations (64-bit, non-hardened mode, ratio 100%).
```

编译后会有插桩符号，使用下面的命令可以验证这一点。

命令: `readelf -s ./test | grep afl`

```
(root@kali)-[~/demo1]
# readelf -s ./test | grep afl
 4: 0000000000001630 0 NOTYPE LOCAL DEFAULT 15 __afl_maybe_log
 6: 0000000000004088 8 OBJECT LOCAL DEFAULT 26 __afl_area_ptr
 7: 0000000000001668 0 NOTYPE LOCAL DEFAULT 15 __afl_setup
 8: 0000000000001640 0 NOTYPE LOCAL DEFAULT 15 __afl_store
 9: 0000000000004090 8 OBJECT LOCAL DEFAULT 26 __afl_prev_loc
10: 000000000000165d 0 NOTYPE LOCAL DEFAULT 15 __afl_return
11: 00000000000040a0 1 OBJECT LOCAL DEFAULT 26 __afl_setup_failur
12: 0000000000001689 0 NOTYPE LOCAL DEFAULT 15 __afl_setup_first
14: 0000000000001951 0 NOTYPE LOCAL DEFAULT 15 __afl_setup_abort
15: 00000000000017a6 0 NOTYPE LOCAL DEFAULT 15 __afl_forkserver
16: 000000000000409c 4 OBJECT LOCAL DEFAULT 26 __afl_temp
17: 0000000000001864 0 NOTYPE LOCAL DEFAULT 15 __afl_fork_resume
18: 00000000000017cc 0 NOTYPE LOCAL DEFAULT 15 __afl_fork_wait_lo
19: 0000000000001949 0 NOTYPE LOCAL DEFAULT 15 __afl_die
20: 0000000000004098 4 OBJECT LOCAL DEFAULT 26 __afl_fork_pid
62: 00000000000040a8 8 OBJECT GLOBAL DEFAULT 26 __afl_global_area_ptr
```

进行下一步之前, 输入如下命令指示系统将 `coredumps` 输出为文件

命令: `echo core > /proc/sys/kernel/core_pattern`

```
(root@kali)-[~/demo1]
# 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

```
(root@kali)-[~/demo1]
# mkdir in out

(root@kali)-[~/demo1]
# echo hello > in/foo
```

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 @@s`

```
(root@kali)-[~/demo1]
# afl-fuzz -i in -o out -- ./test @@s
```

```
american fuzzy lop ++4.08c {default} (./test) [fast]
- process timing
  run time : 0 days, 0 hrs, 0 min, 43 sec
  last new find : 0 days, 0 hrs, 0 min, 6 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
- cycle progress
  now processing : 3.1 (75.0%)
  runs timed out : 0 (0.00%)
- stage progress
  now trying : havoc
  stage execs : 88/1035 (8.50%)
  total execs : 8806
  exec speed : 200.0/sec
- fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 3/8688, 0/0
  py/custom/rq : unused, unused, unused
  trim/eff : 33.33%/1, disabled
- strategy: explore
- state: started :-)
- overall results
  cycles done : 7
  corpus count : 4
  saved crashes : 0
  saved hangs : 0
- map coverage
  map density : 0.00% / 0.00%
  count coverage : 1.00 bits/tuple
- findings in depth
  favored items : 4 (100.00%)
  new edges on : 4 (100.00%)
  total crashes : 0 (0 saved)
  total tmouts : 0 (0 saved)
- item geometry
  levels : 3
  pending : 0
  pend fav : 0
  own finds : 3
  imported : 0
  stability : 100.00%
[cpu000: 50%]
```

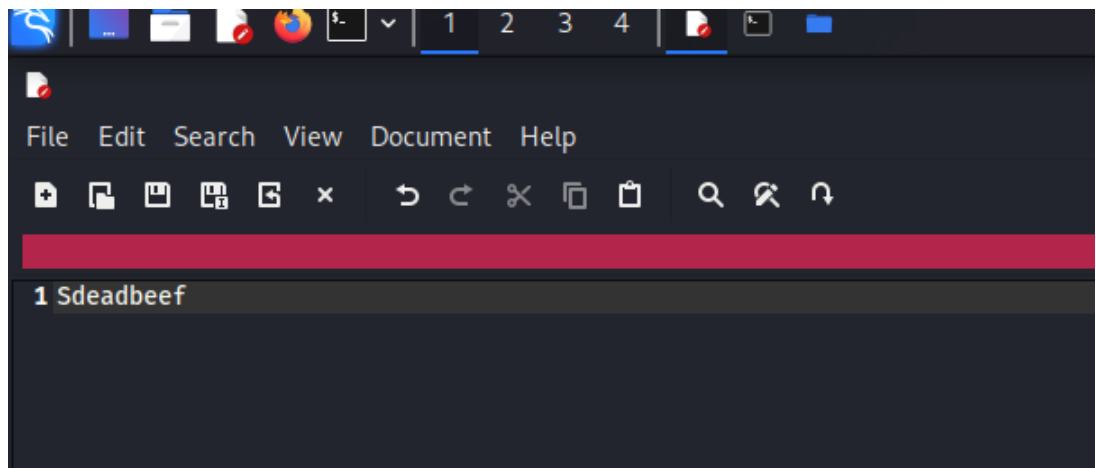
由课本内容可了解到:

- **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，发现有 crash，定位问题。

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。



3. 覆盖引导和文件变异

覆盖引导 (Coverage-Guided)

覆盖引导是一种软件测试和自动化缺陷发现的技术，通常用于动态分析或模糊测试。该技术使用代码覆盖信息来指导测试用例的生成或变异，以提高测试的有效性，从而发现更多的程序缺陷。

在覆盖引导的测试中，系统首先运行一组初始测试用例，并收集这些测试用例执行时的代码覆盖信息。然后，基于这些覆盖信息，系统生成或选择新的测试用例，以覆盖尚未被执行或测试的代码部分。这个过程是迭代的，直到达到某个覆盖标准（如语句覆盖、分支覆盖、条件覆盖等）或时间限制。

覆盖引导的测试技术被广泛用于自动化软件测试和模糊测试工具中，如 AFL (American Fuzzy Lop)、LibFuzzer 等。

文件变异 (File Mutation)

文件变异通常是在模糊测试 (Fuzz Testing) 或变异测试 (Mutation Testing) 中使用的技术。在模糊测试中，文件变异是指对输入文件（如文本文件、二进制文件等）进行随机或系统性的修改，以生成大量的变异文件，这些文件被用作程序的输入来触发潜在的错误或异常。

在变异测试中，文件变异则是指对源代码文件进行微小的修改（即“变异”），以生成新的程序变体。然后，这些变体被编译并执行，以检查它们是否会产生与原始程序不同的行为或结果。如果某个变体在给定测试用例下产生了与原始程序不同的输出，并且这个不同的输出被确定为错误，那么就认为原始程序中存在一个缺陷。

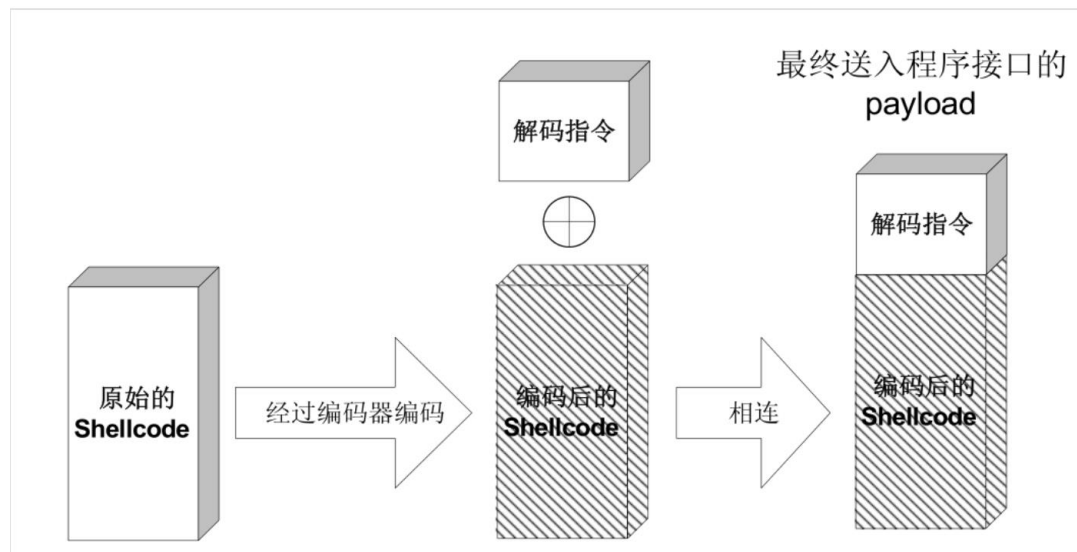
文件变异技术可以帮助测试人员生成大量的测试输入或程序变体，以发现可能隐藏在程序中的错误或异常。然而，这种技术也面临着挑战，如如何生成有意义的变异、如何评估变异的有效性等。

心得体会：

在这次实验中，我反复安装了四次 Kali，终于掌握了如何正确的安装虚拟机，Kali 的安装比 XP 略难一些，其中遇到了下载版本错误，选项错误，遗忘密码等诸多困难，通过自行查阅资料，找到了解决办法，我深刻体会到了细节的重要性。

同时我也发现模糊测试通常是一个耗时的过程，需要长时间的运行才能发现潜在的问题。随着时间的推移，AFL 逐渐发现了目标程序中的一些问题。这使我认识到，在软件开发和测试过程中，持续性和耐心同样重要。

通过此次实验，我更初步了解了 AFL 框架的用法，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，后续我也将积极的的了解和关注。



上图非常清晰的展示了 shellcode 编码的原理。

Shellcode 编码的原理主要是基于一种转换过程，其目的在于将原始的 shellcode 转换为一种能够在特定环境中执行且不易被检测到的形式。这种转换通常涉及对 shellcode 中的字节序列进行重新排列、替换或加密，以避免安全机制的限制和检测。

原理的核心在于，原始的 shellcode 可能包含一些在特定环境下被禁止或限制使用的字符或字节序列。为了绕过这些限制，我们可以对 shellcode 进行编码，将其转换为一个看似无害或不容易被识别的形式。

然而，仅仅对 shellcode 进行编码是不够的。因为编码后的 shellcode 在执行前需要被解码回原始形式。因此，编码过程中通常还会生成一个解码器，这个解码器会在 shellcode 执行前被注入到目标环境中，负责将编码后的 shellcode 解码回原始形式。这样，当解码器执行完毕后，原始的 shellcode 就可以被正确地执行，从而实现攻击者的目的。

2. 解码思想

解码主要通过解码器进行，解码代码。所生成的解码器会与编码后的 shellcode 联合执行。例如本次实验的解码器，默认 EAX 在 shellcode 开始时对准 shellcode 起始位置，之后的代码将每次将 shellcode 的代码异或特定 key（下例为 0x44）后重新覆盖原先 shellcode 的代码。末尾，放一个空指令 0x90 作为结束符。