

《信息安全综合实践》实验报告

实验名称: Fuzzing 实验

姓名: 黎锦灏 学号: 518021910771 邮箱: ljh2000@sjtu.edu.cn 实验时长: 75 分钟

一、实验目的

1. 掌握 AFL++ 等 Fuzzing 工具的使用方法;
2. 学习如何通过 Fuzzing 技术挖掘程序中存在的安全漏洞。

二、实验内容

序	主题	实验内容
1)	通过 fuzz 技术挖掘程序中的漏洞	发现 Crackme1 的可复现 bug
2)		发现 Crackme2 尽可能多的不同路径
3)		通过 gcov 获取 fuzz 之后 crackme1 和 crackme2 的路径信息, 查看是否走到所有路径
4)	sqlite 漏洞挖掘	通过 Squirrel 发现给定版本的 sqlite 漏洞并分析漏洞成因 (本题选做 20 附加分)

三、实验过程截图 (60 分+20 附加分)

1. 使用 AFL++ 等工具进行模糊测试的重要指令和关键操作截图, 比如编译命令、fuzzer 运行界面、发现的漏洞、gcov 路径信息、漏洞成因分析等。

1) Crackme1

启动并创建名为 Aflplusplus 的 docker 容器, 调用 aflplusplus 的内容, 建好 AFL++ 的运行环境。

```
ljh2000@ubuntu: ~/Desktop
File Edit View Search Terminal Help
ljh2000@ubuntu:~/Desktop$ sudo docker run -ti -v /location/of/your/target:/src aflplusplus/aflplusplus
```

使用 **docker cp** 命令将 crackme1 的源代码文件 crackme.c 复制到容器中。注意: docker 相关的命令都需要 sudo 权限。

```
ljh2000@ubuntu:~/Desktop$ sudo docker cp /home/ljh2000/Desktop/Test/crackme1/crackme.c d9d60360a980:/AFLplusplus
```

输入编译命令: **afl-clang-fast crackme.c -o crackme** 编译 crackme.c, 得到可执行文件 crackme, 可以看到 Aflplusplus 的代码插桩结果。

```
[afl++]root@d9d60360a980:/AFLplusplus# afl-clang-fast crackme.c -o crackme
afl-cc ++3.12c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: LLVM-PCGU
ARD
SanitizerCoveragePCGUARD++3.12c
[+] Instrumented 12 locations with no collisions (non-hardened mode).
[afl++]root@d9d60360a980:/AFLplusplus#
```

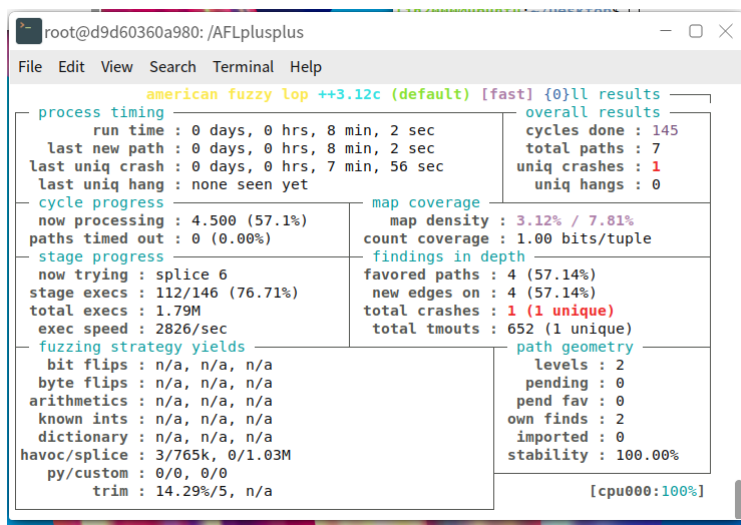
先用 **mkdir in out** 命令创建 in、out 两个文件夹，分别存放 AFL++ 的输入种子文件，和 fuzz 输出的结果（包括 crash 的详细信息）。此时将我根据代码漏洞构造好的输入数据 6667686978563411 输出到种子文件。

```
[afl++]root@d9d60360a980:/AFLplusplus# cd in
[afl++]root@d9d60360a980:/AFLplusplus/in# echo 6667686978563411 | xxd -r -ps > seed
[afl++]root@d9d60360a980:/AFLplusplus/in#
```

输入命令 **afl-fuzz -i in -o out -m none ./crackme @@**，使得 AFL++ 开始 Fuzzing 工作。注意到，@@ 是因为种子文件是自定义的，而不是标准输入文件。

```
[afl++]root@d9d60360a980:/AFLplusplus/in# cd ..
[afl++]root@d9d60360a980:/AFLplusplus# afl-fuzz -i in -o out -m none ./crackme -fsanitize=address @@
```

Fuzzer 运行界面如下：



```
root@d9d60360a980:/AFLplusplus
File Edit View Search Terminal Help

american fuzzy lop ++3.12c (default) [fast] {}ll results

process timing
run time : 0 days, 0 hrs, 8 min, 2 sec
last new path : 0 days, 0 hrs, 8 min, 2 sec
last uniq crash : 0 days, 0 hrs, 7 min, 56 sec
last uniq hang : none seen yet

cycle progress
now processing : 4.500 (57.1%)
paths timed out : 0 (0.00%)

stage progress
now trying : splice 6
stage execs : 112/146 (76.71%)
total execs : 1.79M
exec speed : 2826/sec

fuzzing strategy yields
bit flips : n/a, n/a, n/a
byte flips : n/a, n/a, n/a
arithmetics : n/a, n/a, n/a
known ints : n/a, n/a, n/a
dictionary : n/a, n/a, n/a
havoc/splice : 3/765k, 0/1.03M
py/custom : 0/0, 0/0
trim : 14.29%/5, n/a

overall results
cycles done : 145
total paths : 7
uniq crashes : 1
uniq hangs : 0

map coverage
map density : 3.12% / 7.81%
count coverage : 1.00 bits/tuple

findings in depth
favored paths : 4 (57.14%)
new edges on : 4 (57.14%)
total crashes : 1 (1 unique)
total tmouts : 652 (1 unique)

path geometry
levels : 2
pending : 0
pend fav : 0
own finds : 2
imported : 0
stability : 100.00%

[cpu000:100%]
```

进入 out/default/crashes 文件目录，可以看到 crash 的具体信息，使用 cat 命令发现出现漏洞的输入信息为：fgnixV4ghi。

```

root@d9d60360a980: /AFLplusplus/out/default/crashes
File Edit View Search Terminal Help
dictionary : n/a, n/a, n/a      imported : 0
havoc/splice : 3/784k, 0/1.06M  stability : 100.00%
py/custom : 0/0, 0/0
trim : 14.29%/5, n/a          [cpu000:450%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

[afl++]root@d9d60360a980:/AFLplusplus# cd out
[afl++]root@d9d60360a980:/AFLplusplus/out# cd default
[afl++]root@d9d60360a980:/AFLplusplus/out/default# cd crashes
[afl++]root@d9d60360a980:/AFLplusplus/out/default/crashes# ls
README.txt  id:000000,sig:06,src:000004,time:6072,op:havoc,rep:2
[afl++]root@d9d60360a980:/AFLplusplus/out/default/crashes# cat id:
README.txt
id:000000,sig:06,src:000004,time:6072,op:havoc,rep:2
[afl++]root@d9d60360a980:/AFLplusplus/out/default/crashes# cat id:
README.txt
id:000000,sig:06,src:000004,time:6072,op:havoc,rep:2
[afl++]root@d9d60360a980:/AFLplusplus/out/default/crashes# cat id:~C
[afl++]root@d9d60360a980:/AFLplusplus/out/default/crashes# cat id:000000,sig:06,
src:000004,time:6072,op:havoc,rep:2
fghixV4ghi[afl++]root@d9d60360a980:/AFLplusplus/out/default/crashes#

```

下面简单分析代码漏洞:

在 num 等于 0x12345678 时触发 abort()函数, 故只需构造数据使得 crackme 中的 num 为 0x12345678 即可。注意存储为小数端, 故输入数据需要颠倒。

```

16 int crackme(const char *data,unsigned int size){
17     unsigned int num = *((unsigned int *)data);
18     if(size < sizeof(unsigned int))
19         return 0;
20     if(num == 0x12345678)
21         abort();
22     return 0;
23 }

```

2) Crackme2

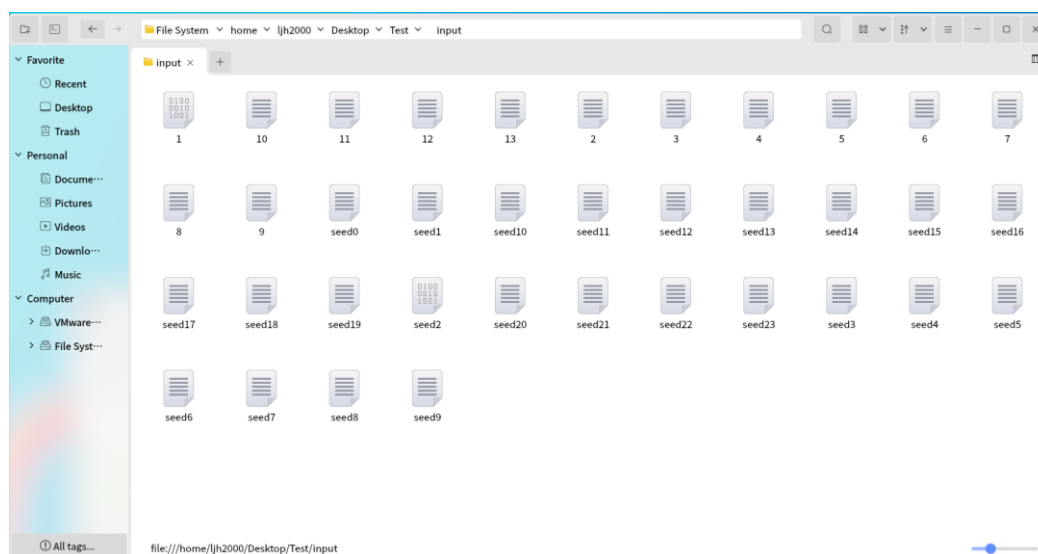
创建新的容器, 使用 docker cp 命令将 crackme.c 复制到容器中, 输入编译命令: afl-clang-fast crackme.c -o crackme 编译 crackme.c, 创建 in、out 文件夹用于输入种子, 输出 Fuzzing 结果文件。

```

ljh2000@ubuntu:~/Desktop$ sudo docker run -ti -v /location/of/your/target:/src a
flplusplus/aflplusplus
[sudo] password for ljh2000:
[afl++]root@625708cd6549:/AFLplusplus# ^C
[afl++]root@625708cd6549:/AFLplusplus# ls
Android.bp      afl-cmin        include
CONTRIBUTING.md afl-cmin.bash   instrumentation
Changelog.md    afl-plot       libAFLDriver.a
Dockerfile      afl-system-config libAFLQemuDriver.a
GNUmakefile     afl-whatsup    qemu_mode
GNUmakefile.gcc_plugin afl-wine-trace src
GNUmakefile.llvm config.h        test
LICENSE         crackme.c      test-instr.c
Makefile        custom_mutators testcases
QuickStartGuide.md dictionaries  types.h
README.md       docs          unicorn_mode
TODO.md         dynamic_list.txt utils
[afl++]root@625708cd6549:/AFLplusplus# mkdir in out
[afl++]root@625708cd6549:/AFLplusplus#

```

我在阅读源码后构造了 36 个种子文件。



输入命令：`afl-fuzz -i input -o out -m none ./crackme @@` 开始 Fuzzing。可以看到一共找到了 **53 条 path**。

```
root@f96494a3a133: /AFLplusplus
File Edit View Search Terminal Help

process timing
run time : 0 days, 17 hrs, 17 min, 36 sec
last new path : 0 days, 16 hrs, 49 min, 47 sec
last uniq crash : none seen yet
last uniq hang : none seen yet

cycle progress
now processing : 50.601 (94.3%)
paths timed out : 0 (0.00%)

stage progress
now trying : havoc
stage execs : 806/3532 (22.82%)
total execs : 11.8M
exec speed : 2654/sec

fuzzing strategy yields
bit flips : n/a, n/a, n/a
byte flips : n/a, n/a, n/a
arithmetics : n/a, n/a, n/a
known ints : n/a, n/a, n/a
dictionary : n/a, n/a, n/a
havoc/splice : 25/4.16M, 5/7.64M
py/custom : 0/0, 0/0
trim : 0.00%/235, n/a

map coverage
map density : 21.09% / 47.66%
count coverage : 1.11 bits/tuple

findings in depth
favored paths : 25 (47.17%)
new edges on : 32 (60.38%)
total crashes : 0 (0 unique)
total tmouts : 3 (3 unique)

path geometry
levels : 8
pending : 6
pend fav : 0
own finds : 30
imported : 0
stability : 100.00%

overall results
cycles done : 61
total paths : 53
uniq crashes : 0
uniq hangs : 0

[cpu000:250%]
```

下面介绍阅读源码后构造数据方式：

注意 `directive=ptr_data+4`，故前四位输入数据无效，在构造数据时用 0000 补位即可。

```

148 // initialize(config);
149 if (data_size > 4 + strlen("crashstring") + 1)
150 {
151     directive = ptr_data + 4;
152     printf("%s\n", directive);
153 } else return -1;

```

为了走到尽可能多的路径，对于每个 if 分支分别构造符合条件判断的数据，作为一个 seed 文件放置在 input 目录下。

```

158 if(!strcmp(directive, "crashstring"))
159 {
160     assert(1!=0);
161     __BUG__;
162 }
163 // int strcmp(const char *s1, const char *s2, size_t n);
164 // Compare at most n bytes of the strings s1 and s2.
165
166 if(!my_strncmp(directive, "!et_pt*on", 9))
167 {
168     __BUG__;
169 }
170

```

以这一 if 分支为例，构造数据如下：

```

// Compare at most n bytes of the strings s1 and s2.
if(!my_strncmp(directive, "!et_pt*on", 9))
{
    __BUG__;
}

```

为了走到尽可能多的路径，构造符合条件作为一个 seed 文件放置在 input 目录下。

3) gcov 获取路径信息：

Crackme1:

首先输入 **gcc -fprofile-arcs -ftest-coverage crackme.c -o crackme**，编译 crackme.c 文件。

```

[afl++]root@d9d60360a980:/AFLplusplus# gcc -fprofile-arcs -ftest-coverage crackme.c -o crackme
[afl++]root@d9d60360a980:/AFLplusplus#

```

使用 gcov 获取 crackme1 中运行路径信息：

./crackme seed2

gcov crackme.c

```
[afl++]root@d9d60360a980:/AFLplusplus# ./crackme seed2
[afl++]root@d9d60360a980:/AFLplusplus# gcov crackme.c
crackme.gcov:version 'B02*', prefer 'A93*'
crackme.gcvda:version 'B02*', prefer version 'A93*'
File 'crackme.c'
Lines executed:71.43% of 35
Creating 'crackme.c.gcov'
```

使用 **cat crackme.c.gcov** 观察经过的路径，#####表示未经过，数字表示经过的次数。

```
1: 16:int crackme(const char *data,unsigned int size){
1: 17:   unsigned int  num = *((unsigned int *)data);
1: 18:   if(size < sizeof(unsigned int))
#####: 19:       return 0;
1: 20:   if(num == 0x12345678)
#####: 21:       abort();
1: 22:   return 0;
-: 23:}
-: 24:
1: 25:int main(int argc, char *argv[]){
-: 26:
1: 27:   if(argc != 2){
#####: 28:       printf("USAGE: %s InputFile \n\n", argv[0]);
#####: 29:       exit(EXIT_FAILURE);
-: 30:   }
-: 31:
1: 32:   char *inputFile = argv[1];
-: 33:   off_t inSize;
-: 34:
1: 35:   if(access(inputFile, F_OK) == -1){
```

以上指令能得到路径信息以及代码覆盖率，但是很难找到一个种子文件能走完全部路径的情况，所以展示中路径并未覆盖全部代码。

Crackme2:

同样使用类似指令执行 gcov，选取一组种子来观察得到的路径（这里我们选取了 seed12）：

```
-----
[afl++]root@f96494a3a133:/AFLplusplus# gcc -fprofile-arcs -ftest-coverage crackme.c -o crackme
[afl++]root@f96494a3a133:/AFLplusplus# ./crackme input/seed12
30 30 30 30 58 65 52 3D 47 74 49 6F 6E 0A 00
size of input data = 14
start testing!!
Bad magic numberLINE = 123
result = 3

finished testing!!
libgcov profiling error:/AFLplusplus/crackme.gcvda:overwriting an existing profile data with a different timestamp
[afl++]root@f96494a3a133:/AFLplusplus# gcov crackme.c
crackme.gcvno:version 'B02*', prefer 'A93*'
crackme.gcvda:version 'B02*', prefer version 'A93*'
File 'crackme.c'
Lines executed:27.78% of 144
Creating 'crackme.c.gcov'

[afl++]root@f96494a3a133:/AFLplusplus#
```

cat crackme.c.gcov 观察代码覆盖信息：

```
root@f96494a3a133: /AFLplusplus
File Edit View Search Terminal Help
-: 315:#ifdef AFL_PERSISTENT_MODE
-: 316:// while ( __AFL_LOOP(1000)) {
-: 317:#endif // #ifdef AFL_PERSISTENT_MODE
-: 318:
1: 319:     int i = 0;
1: 320:     if (argc == 2) {
1: 321:         global_pkt_buf = read_data_from_file(argv[1], &global_data_size);
-: 322:     } else {
#####: 323:         printf("Wrong number of arguments = %d - should be 1!\n", argc-1);
#####: 324:         return -1;
-: 325:     }
-: 326:
1: 327:     printf("size of input data = %zu\n", global_data_size);
-: 328:
1: 329:     if (global_data_size == 0)
-: 330:     {
#####: 331:         printf("Data sample is empty!\n");
#####: 332:         return -1;
-: 333:     }
-: 334:
-: 335:#endif
-: 336:
1: 337:     printf("start testing!!\n");
-: 338:
```

Lcov 图形化显示:

初始化并创建基准数据文件: **lcov -c -i -d ./ -o init.info**

```
root@f96494a3a133: /AFLplusplus
File Edit View Search Terminal Help
[afl++]root@f96494a3a133:/AFLplusplus# lcov -a init.info -a cover.info -o total.info
Combining tracefiles.
Reading tracefile init.info
Reading tracefile cover.info
Writing data to total.info
Summary coverage rate:
  lines.....: 29.9% (43 of 144 lines)
  functions...: 100.0% (3 of 3 functions)
  branches....: no data found
```

执行编译后的测试文件: **./crackme seed12**

```
Bad magic number
[afl++]root@f96494a3a133:/AFLplusplus# ./crackme input/seed12
30 30 30 30 58 65 52 3D 47 74 49 6F 6E 0A 00
size of input data = 14
start testing!!
Bad magic numberLINE = 123
result = 3
```

收集测试文件运行后产生的覆盖率文件: **lcov -c -d ./ -o cover.info**

```
[afl++]root@f96494a3a133:/AFLplusplus# lcov -c -d ./ -o cover.info
Capturing coverage data from ./
Found gcov version: 9.3.0
Using intermediate gcov format
Scanning ./ for .gda files ...
Found 1 data files in ./
Processing crackme.gcda
/AFLplusplus/crackme.gcno:version 'B02*', prefer 'A93*'
/AFLplusplus/crackme.gcda:version 'B02*', prefer version 'A93*'
Finished .info-file creation
```

合并基准数据和执行测试文件后生成的覆盖率数据:

lcov -a init.info -a cover.info -o total.info

```
root@f96494a3a133: /AFLplusplus
File Edit View Search Terminal Help
[afll+]root@f96494a3a133:/AFLplusplus# lcov -a init.info -a cover.info -o total.info
Combining tracefiles.
Reading tracefile init.info
Reading tracefile cover.info
Writing data to total.info
Summary coverage rate:
  lines.....: 29.9% (43 of 144 lines)
  functions...: 100.0% (3 of 3 functions)
  branches....: no data found
```

通过 final.info 生成 html 文件:

genhtml -o cover_report --legend --title "lcov" --prefix=./ final.info

```
root@f96494a3a133: /AFLplusplus
File Edit View Search Terminal Help
[afll+]root@f96494a3a133:/AFLplusplus# genhtml -o cover_report --legend --title "lcov" --prefix=./ final.info
Reading data file final.info
Found 1 entries.
Using user-specified filename prefix "."
Writing .css and .png files.
Generating output.
Processing file ./AFLplusplus/crackme.c
Writing directory view page.
Overall coverage rate:
  lines.....: 29.9% (43 of 144 lines)
  functions...: 100.0% (3 of 3 functions)
```

使用 ls 查看当前目录:

```
root@f96494a3a133: /AFLplusplus# ls
Android.bp      QuickStartGuide.md  config.h          dictionaries      libAFLDriver.a      total.info
CONTRIBUTING.md  README.md            cover.info        docs              libAFLQemuDriver.a  types.h
ChangeLog.md     TODO.md              crackme           dynamic_list.txt  out                 unicorn_mode
Dockerfile       afl-cmin             crackme.c         final.info        out_old             utils
GNUMakefile      afl-cmin.bash        crackme.gcov     include           qemu_mode           src
GNUMakefile.gcc_plugin  afl-plot            crackme.gcd      instrumentation  test
GNUMakefile.llvm  afl-system-config   crackme.gcno     input             test-instr.c
LICENSE          afl-whatsup          custom_mutators  instrumentation  testcases
Makefile         afl-wine-trace
```

通过 firefox 打开 **cover_report/index.html** 查看代码覆盖率如下:

```
LCOV - code coverage report
Current view: top level
Test: lcov
Date: 2021-04-10 03:32:05
Legend: Rating: low: < 75 % medium: >= 75 % high: >= 90 %

Directory: /AFLplusplus

Line Coverage: 29.9 % 43 / 144
Functions: 100.0 % 3 / 3

Hit Total Coverage
Lines: 43 144 29.9 %
Functions: 3 3 100.0 %

Generated by: LCOV version 1.14
```

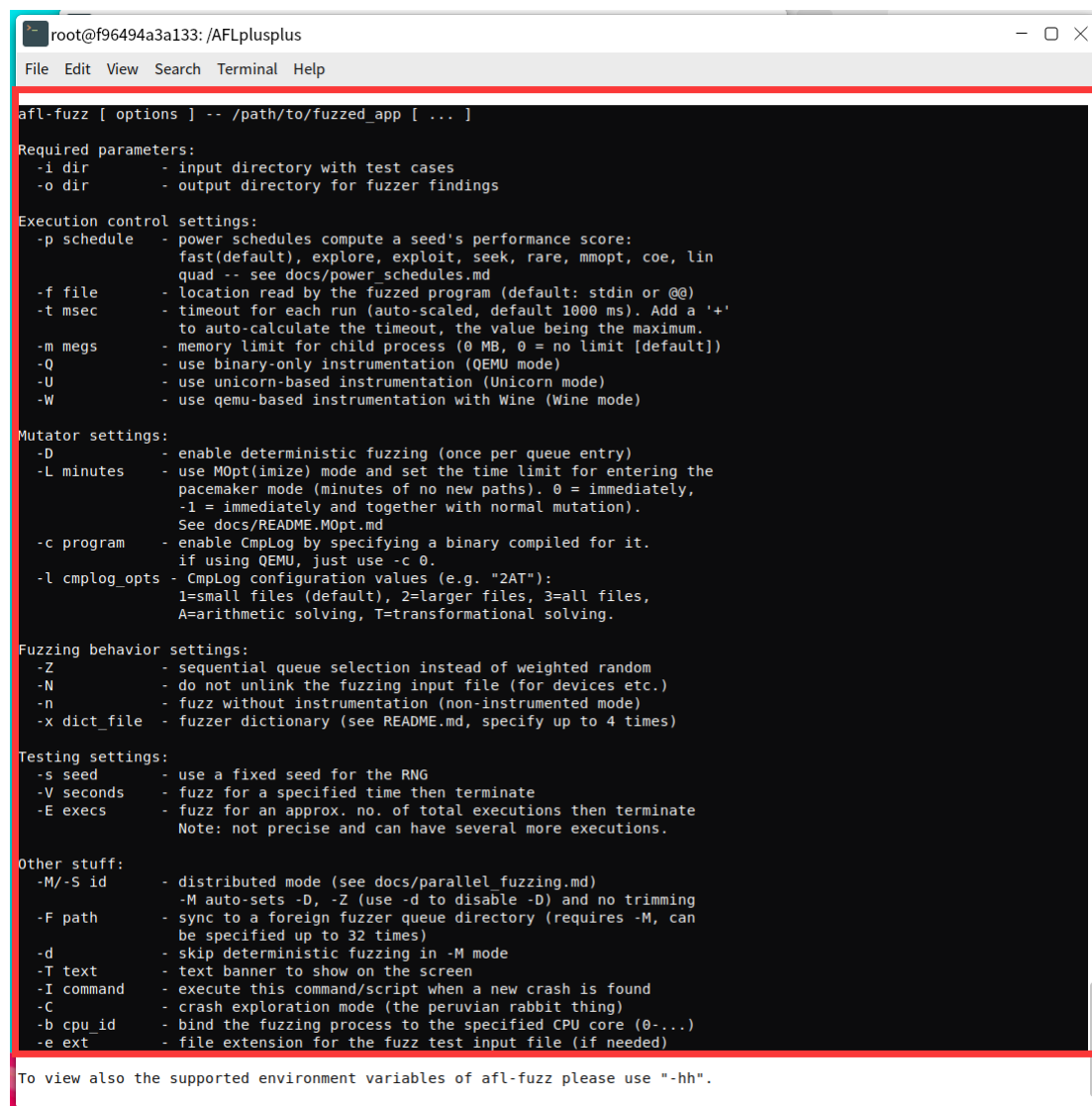
在本次实验中,我选取了 **36** 个种子跑出了 **53** 条路径,上面只展示了使用 **seed12** 的路径信息和覆盖结果。

四、分析和思考 (30 分)

1. 在使用 AFL++ 进行模糊测试的过程中,使用不同的参数是否会对结果产生明显区别,若有,请阐述添加参数的内在意义和原理,若没有,请给出理由。

AFL++进行模糊测试时，不同的参数会对结果产生区别。

下面列举了 afl-fuzz 的参数列表：



```
root@f96494a3a133: /AFLplusplus
File Edit View Search Terminal Help

afl-fuzz [ options ] -- /path/to/fuzzed_app [ ... ]

Required parameters:
-i dir      - input directory with test cases
-o dir      - output directory for fuzzer findings

Execution control settings:
-p schedule - power schedules compute a seed's performance score:
              fast(default), explore, exploit, seek, rare, mmopt, coe, lin
              quad -- see docs/power_schedules.md
-f file     - location read by the fuzzed program (default: stdin or @@)
-t msec     - timeout for each run (auto-scaled, default 1000 ms). Add a '+'
              to auto-calculate the timeout, the value being the maximum.
-m megs     - memory limit for child process (0 MB, 0 = no limit [default])
-Q          - use binary-only instrumentation (QEMU mode)
-U          - use unicorn-based instrumentation (Unicorn mode)
-W          - use qemu-based instrumentation with Wine (Wine mode)

Mutator settings:
-D          - enable deterministic fuzzing (once per queue entry)
-L minutes  - use MOpt(imize) mode and set the time limit for entering the
              pacemaker mode (minutes of no new paths). 0 = immediately,
              -1 = immediately and together with normal mutation).
              See docs/README.MOpt.md
-c program  - enable CmpLog by specifying a binary compiled for it.
              If using QEMU, just use -c 0.
-l cmplog_opts - CmpLog configuration values (e.g. "2AT"):
              1=small files (default), 2=larger files, 3=all files,
              A=arithmetic solving, T=transformational solving.

Fuzzing behavior settings:
-Z          - sequential queue selection instead of weighted random
-N          - do not unlink the fuzzing input file (for devices etc.)
-n          - fuzz without instrumentation (non-instrumented mode)
-x dict_file - fuzzer dictionary (see README.md, specify up to 4 times)

Testing settings:
-s seed     - use a fixed seed for the RNG
-V seconds  - fuzz for a specified time then terminate
-E execs    - fuzz for an approx. no. of total executions then terminate
              Note: not precise and can have several more executions.

Other stuff:
-M/-S id    - distributed mode (see docs/parallel fuzzing.md)
              -M auto-sets -D, -Z (use -d to disable -D) and no trimming
-F path     - sync to a foreign fuzzer queue directory (requires -M, can
              be specified up to 32 times)
-d          - skip deterministic fuzzing in -M mode
-T text     - text banner to show on the screen
-I command  - execute this command/script when a new crash is found
-C          - crash exploration mode (the peruvian rabbit thing)
-b cpu_id   - bind the fuzzing process to the specified CPU core (0-...)
-e ext      - file extension for the fuzz test input file (if needed)

To view also the supported environment variables of afl-fuzz please use "-hh".
```

1、使用 AFL++进行模糊测试时添加 **-c 参数**，可以将目标文件中的比较值传入 AFL++。若目标文件在输入之前没有转换数据，添加-c 参数会大幅改善结果。

2、使用 LLVM，并加入 **afl-clang-lto** 或者 **afl-clang-lto++** 命令可以加速 fuzz 的速度。这一命令能提供更好的代码覆盖率，且有可能找到其他命令挖掘不出的漏洞。

3、加入参数 `AFL_LLVM_LAF_ALL`, 设置 `export AFL_LLVM_LAF_ALL=1`, 可以看到 `AFL++` 能更有效的拆分整数、字符串、浮点数。这一参数的优势在于, 能在种子不够优秀且偏小时, 显著提高挖洞的成功率。

2. 挖掘程序漏洞, 除了使用不同的模糊测试工具之外, 还和其他什么因素相关?

- 1、Fuzzing 模糊测试是基于遗传算法的, 对于初始种子的选取非常重要, 优良的种子数据能极大提高程序漏洞挖掘效率。一个优质用例往往能用最少的步数走到最多的路径, 并且初始种子应尽可能小。
- 2、模糊测试分析者的经验与技术水平。待分析的程序有可能代码架构混乱, 难以阅读, 而且大型程序更需要分析者在阅读源码后针对性地设置初始种子、设计测试方法, 这都是挖掘程序漏洞的重要因素。
- 3、挖掘程序漏洞也与操作环境有关, 有些漏洞可能在特定操作系统或特定数据库下才会暴露, 在单一环境下测试时可能无法体现。
- 4、挖掘程序漏洞还与待挖掘的程序本身有关。程序源码中可能存在校验和、HMAC 等部分, 使得 Fuzzing 工作困难, 影响效率与结果。

3. 除了实验中用到的模糊测试工具, 你还知道哪些模糊测试测试工具, 他们的有什么特点? 请介绍两到三种你所了解的模糊测试工具。

我了解的模糊测试工具还有: **Libfuzzer**、**Sulley**、**PeachTech Peach Fuzzer**。

1、Libfuzzer:

Libfuzzer 的特点在于“**演进式模糊测试**”。

Libfuzzer 的目标是产生比传统模糊测试工具更相关的结果, 该工具将模糊输入反馈给目标程序的特定进入点或输入域, 然后根据被测试应用程序对这些查询的反应, 跟踪该代码还触及了哪些其他部分。在获得新信息后, Libfuzzer 修改其查询, 查看自身是否能进一步渗透。

Libfuzzer

```
const size_t N = 1 << 12;

// Define an array of counters that will be understood by libFuzzer
// as extra coverage signal. The array must be:
// * uint8_t
// * in the section named __libfuzzer_extra_counters.
// The target code may declare more than one such array.
//
// Use either `Counters[Idx] = 1` or `Counters[Idx]++;`
// depending on whether multiple occurrences of the event 'Idx'
// is important to distinguish from one occurrence.
```

2、Sulley:

Sulley 的特点在于能一次性无缝运行数日，持续检查应用程序对模糊输入的怪异响应并记录结果。

Sulley 依托硬件平台的并行执行能力，能在无需用户编程的情况下，自动确定测试案例的哪种特定顺序会触发错误。对于在激活模糊测试引擎后有别的任务的用户，在他们数小时或数天后返回时就能得到 Sulley 的所有报告信息。

Sulley

A pure-python fully automated and unattended fuzzing framework.

85 commits

2 branches

0 packages

1 release

11 contributors


GPL-2.0

Branch: master


New pull request

Find file

Clone or download


 pedrammini Update README.md

Latest commit bf#0d1 on 15 Feb 2019

 docs


Corrected exception thrown when the wrong value is used.

rs ago

 examples


Move archived_fuzzies -> examples

rs ago

 installer


Initial import from google-code-svn.

rs ago

 requests


Rename pedram's XXX: naming scheme to TODO:

rs ago

 sulley


Merge pull request #102 from truekonrads/getaddrinfo-af-inet6-support

rs ago

 unit_tests


Fixed primitives "off-by-one" error. This was preventing "full_range"...

rs ago

 utils


Rename pedram's XXX: naming scheme to TODO:

rs ago

 .gitignore

Corrected exception thrown when the wrong value is used.

rs ago

 AUTHORS.txt

Update contributors/authors, and make the session class a bit more re...

rs ago

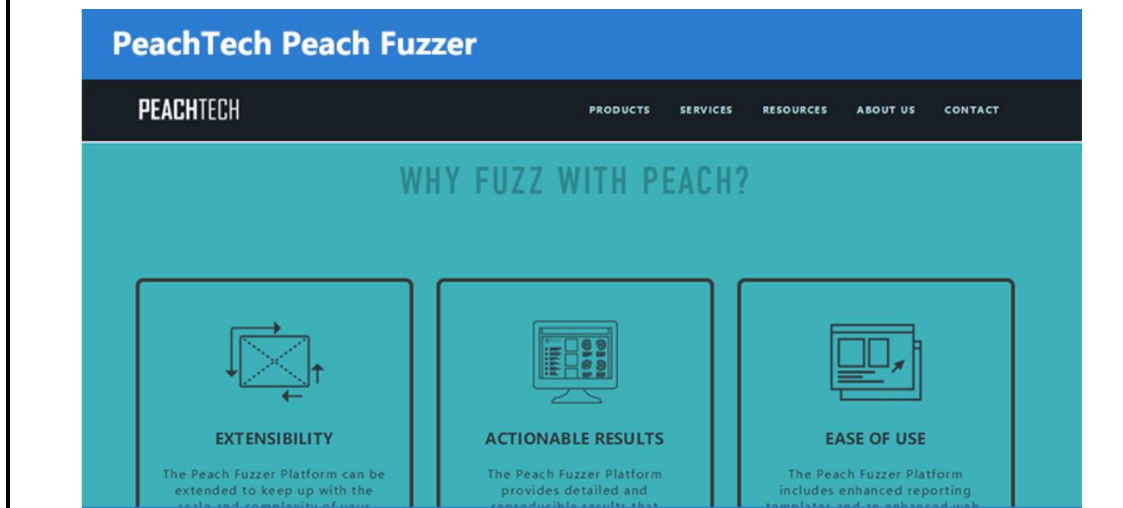
3、PeachTech Peach Fuzzer:

Peach Fuzzer 的特点在于使用 **Peach Pit** 测试定义规范，能让测试者无需考虑平台进行模糊测试。

Peach Pit 是预先编写好的测试定义，包含适用特定目标的规范（比如目标摄入的数据结构，数据流入和流出被测设备或应用程序的方式等），覆盖

Mac、Windows、Linux 等不同平台。Peach Pit 能让测试人员无需设置即可进行模糊测试工作，而且 Peach Pit 的创建非常简单。

Peach Pit 还适用于模糊测试网络协议、嵌入式系统、物联网设备等，Peach Fuzzer 测试的适用面非常广泛。



五、实验总结（收获和心得）（5 分）

在本次实验中，我学习了模糊测试的基本原理，接触了 AFL++ 等 Fuzzing 工具，了解了具体使用方法和工作流程。在运用 Fuzzing 技术挖掘程序漏洞的过程中，我体会到了模糊测试作为自动化测试工具的强大功能。

模糊测试是一种通过向目标系统提供非预期的输入并监视异常结果来发现软件漏洞的方法，这一测试技术主要基于黑盒（或灰盒），Fuzzing 通过自动化生成并执行大量的随机测试用例来发现产品或协议的未知漏洞。Fuzzing 的主要步骤有：种子选取、数据变异、执行程序获取反馈、输入覆盖路径反馈、监控评估并判定用例是否有价值。

当然我也认识到 AFL++ 作为 Fuzzing 工具的不足：在 Crackme1 中，光依赖随机种子几乎不可能得到挖掘到程序漏洞，因为 `abort()` 是 `num` 取得某一特定数据时才会触发。从这里可以看到，AFL 的反馈信息不包含条件信息，对于需满足特定条件的数据、哈希和加解密，完全靠随机碰撞。

我希望在今后的学习中能接触到更多漏洞测试工具和自动化测试技术，丰富知识面的同时也能学习如何检测、全方位保障代码安全。

六、尚存问题或疑问、建议（5 分）

疑问：

- 1、在本次实验中 Crackme1、Crackme2 代码不长，阅读源码较容易，但是实际的大型工程文件过于庞大，依靠人工挑选优质种子代价太大，而随机种子的效率可能太低，在工程应用时该如何构造高效的种子来提高漏洞挖掘效率呢？
- 2、Crackme1 的代码漏洞明显，因为出现了 `abort()` 这一具有显著特征的函数，但是更多的代码漏洞诸如内存泄漏、访问越界等并不能简单通过阅读代码分析，对于这些漏洞应该如何挖掘呢？

建议：

这次实验前老师下发了部分教程和需要搭建的环境，但是在实际应用中，许多同学还是遇到了环境搭建的困难，并且下发的教程不属于入门向的指导，对于初学者来说晦涩难懂，很多模糊测试的原理都是通过课堂讲授以及自己实际操作一段时间后才有所理解，一定程度上影响了听课效率。当然，对于 `docker` 的命令不熟悉，`Linux` 操作指令不够熟练也是影响实践进度的重要原因。

建议每次课程提供搭建好相应环境的虚拟机，供自主搭建环境存在困难或时间紧张的同学使用。另外，希望能在上课前，发放一些相关原理的 `ppt` 和偏向入门向的教程，能让同学们在提前预习后显著提高学习效率。