AFL漏洞挖掘技术漫谈（一）：用AFL开始你的第一次Fuzzing - FreeBuf互联网安全新媒体平台

freebuf.com/articles/system/191543.html

AFL漏洞挖掘技术漫谈（一）：用AFL开始你的第一次Fuzzing

alphalab 2018-12-17 共**247047**人围观，发现 **5** 个不明物体 系统安全

***本文中涉及到的相关漏洞已报送厂商并得到修复，本文仅限技术研究与讨论，严禁用于非法用途，否则产生的一切后果自行承担。**

一、前言

模糊测试（Fuzzing）技术作为漏洞挖掘最有效的手段之一，近年来一直是众多安全研究人员发现漏洞的首选技术。AFL、LibFuzzer、honggfuzz等操作简单友好的工具相继出现，也极大地降低了模糊测试的门槛。阿尔法实验室的同学近期学习漏洞挖掘过程中，感觉目前网上相关的的资源有些冗杂，让初学者有些无从着手，便想在此对学习过程中收集的一些优秀的博文、论文和工具进行总结与梳理、分享一些学习过程中的想法和心得，同时对网上一些没有涉及到的内容做些补充。

由于相关话题涉及的内容太广，笔者决定将所有内容分成一系列文章，且只围绕AFL这一具有里程碑意义的工具展开，从最简单的使用方法和基本概念讲起，再由浅入深介绍测试完后的后续工作、如何提升Fuzzing速度、一些使用技巧以及对源码的分析等内容。因为笔者接触该领域也不久，内容中难免出现一些错误和纰漏，欢迎大家在评论中指正。

第一篇文章旨在让读者对AFL的使用流程有个基本的认识，文中将讨论如下一些基本问题：

> AFL的基本原理和工作流程；
>
> 如何选择Fuzzing的目标?
>
> 如何获得初始语料库?
>
> 如何使用AFL构建程序？
>
> AFL的各种执行方式；
>
> AFL状态窗口中各部分代表了什么意义？

二、AFL简介

AFL（American Fuzzy Lop）是由安全研究员Michał Zalewski（@lcamtuf）开发的一款基于覆盖引导（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。其工作流程大致如下：

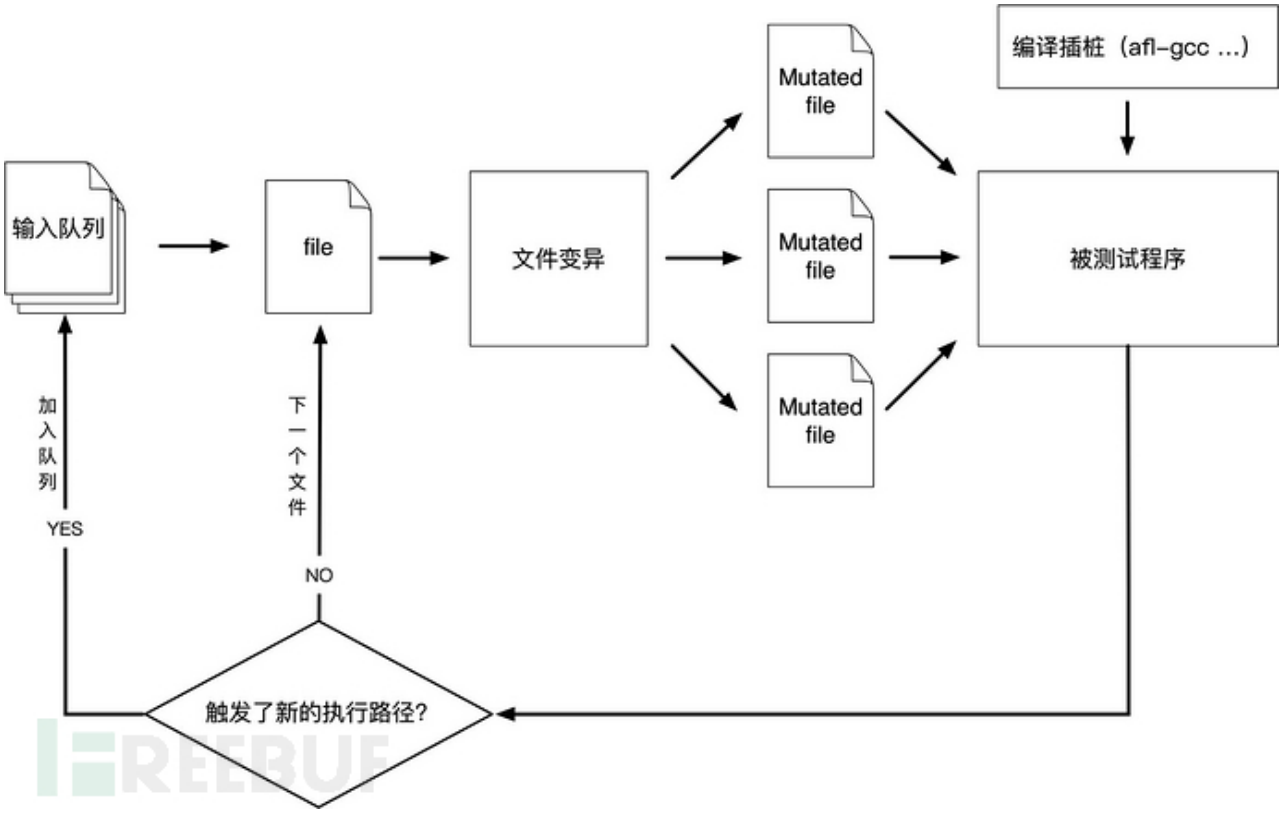
①从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）；

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。



三、选择和评估测试的目标

开始Fuzzing前，首先要选择一个目标。 AFL的目标通常是接受外部输入的程序或库，输入一般来自文件（后面的文章也会介绍如何Fuzzing一个网络程序）。

1. 用什么语言编写

AFL主要用于C/C++程序的测试，所以这是我们寻找软件的最优先规则。（也有一些基于AFL的JAVA Fuzz程序如kelinci、java-afl等，但并不知道效果如何）

2. 是否开源

AFL既可以对源码进行编译时插桩，也可以使用AFL的 QEMU mode 对二进制文件进行插桩，但是前者的效率相对来说要高很多，在Github上很容易就能找到很多合适的项目。

3. 程序版本

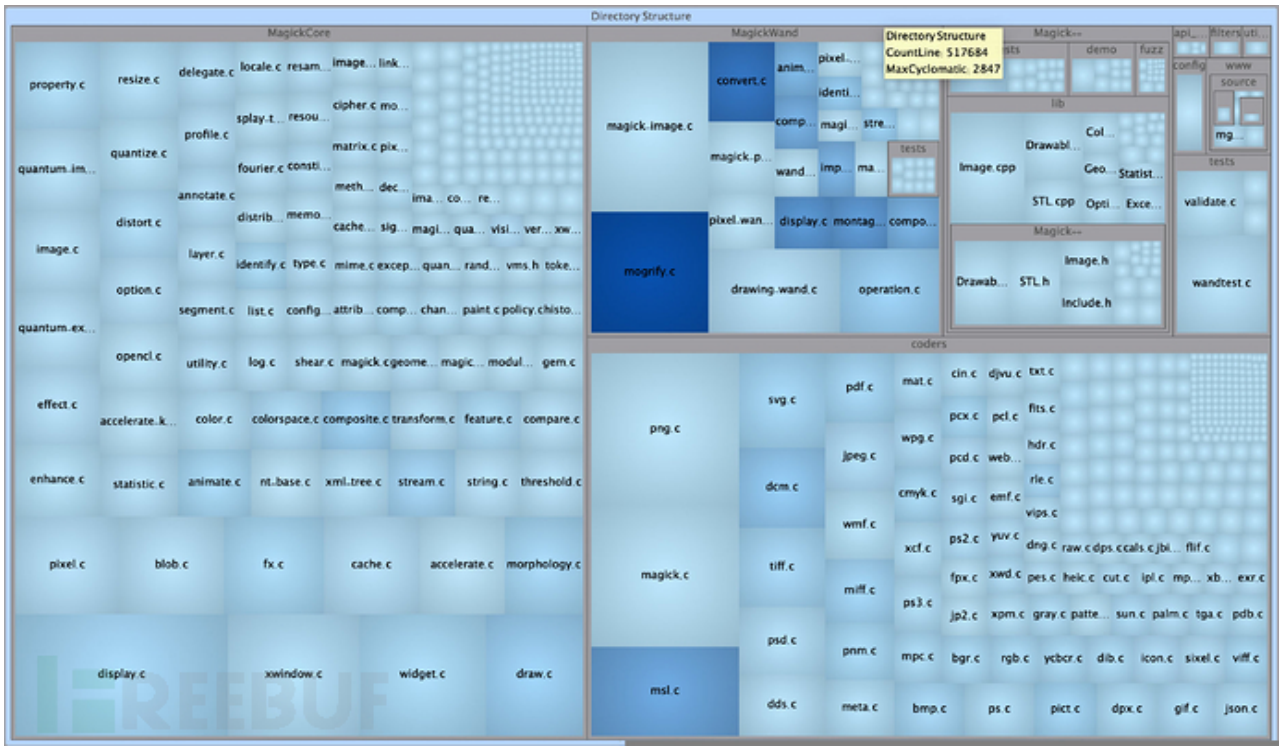
目标应该是该软件的最新版本，不然辛辛苦苦找到一个漏洞，却发现早就被上报修复了就尴尬了。

4. 是否有示例程序、测试用例

如果目标有现成的基本代码示例，特别是一些开源的库，可以方便我们调用该库不用自己再写一个程序；如果目标存在测试用例，那后面构建语料库时也省事儿一点。

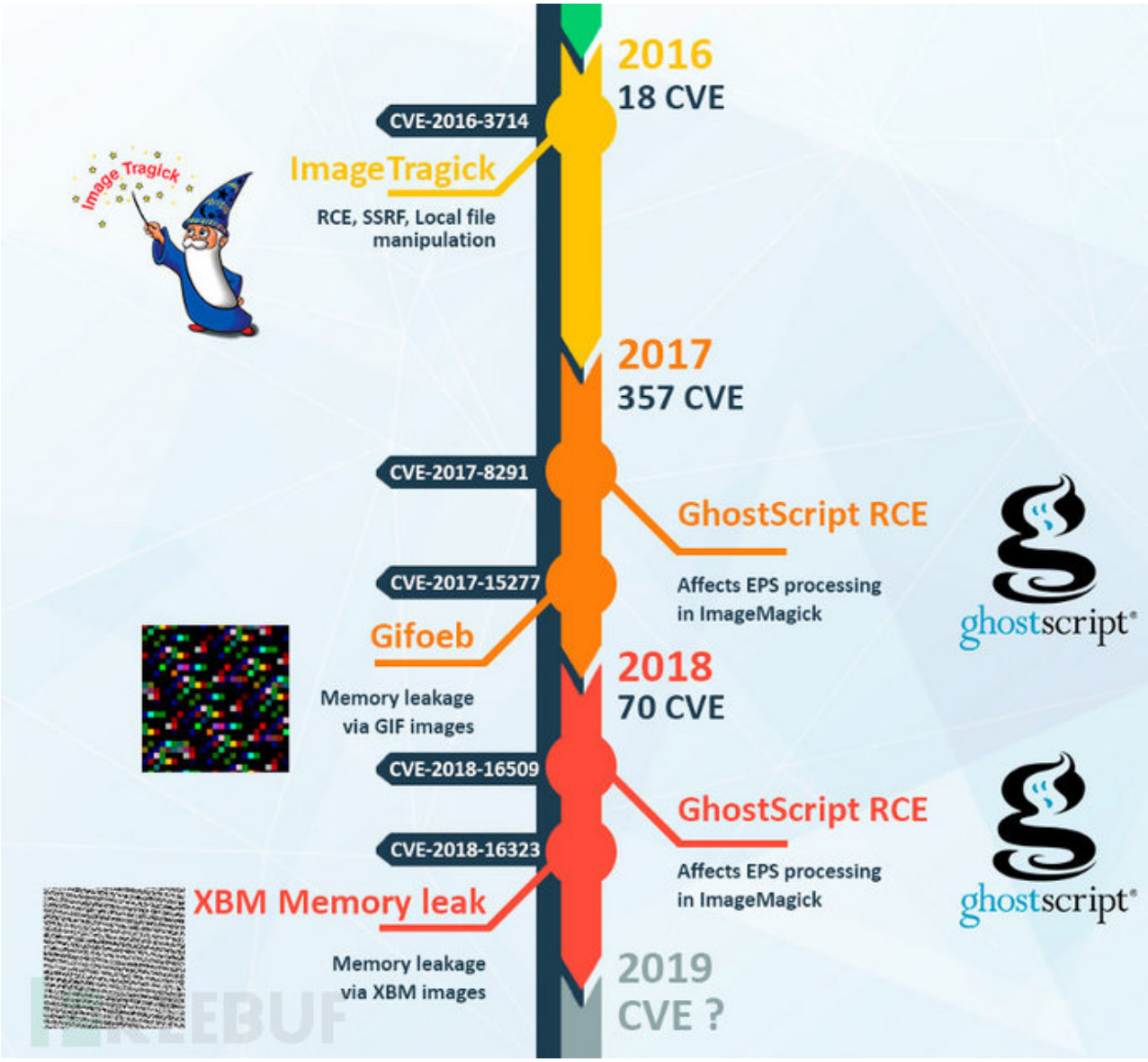
5.项目规模

某些程序规模很大，会被分为好几个模块，为了提高Fuzz效率，在Fuzzing前，需要定义Fuzzing部分。这里推荐一下源码阅读工具Understand，它 treemap 功能，可以直观地看到项目结构和规模。比如下面ImageMagick的源码中，灰框代表一个文件夹，蓝色方块代表了一个文件，其大小和颜色分别反映了行数和文件复杂度。



6. 程序曾出现过漏洞

如果某个程序曾曝出过多次漏洞，那么该程序有仍有很大可能存在未被发现的安全漏洞。如ImageMagick每个月都会发现难以利用的新漏洞，并且每年都会发生一些具有高影响的严重漏洞，图中可以看到仅2017年就有357个CVE！(图源medium.com)



四、构建语料库

AFL需要一些初始输入数据（也叫种子文件）作为Fuzzing的起点，这些输入甚至可以是毫无意义的数据，AFL可以通过启发式算法自动确定文件格式结构。lcamtuf就在博客中给出了一个有趣的例子——对djpeg进行Fuzzing时，仅用一个字符串”hello”作为输入，最后凭空生成大量jpge图像！

尽管AFL如此强大，但如果要获得更快的Fuzzing速度，那么就有必要生成一个高质量的语料库，这一节就解决如何选择输入文件、从哪里寻找这些文件、如何精简找到的文件三个问题。

1. 选择

(1) 有效的输入

尽管有时候无效输入会产生bug和崩溃，但有效输入可以更快的找到更多执行路径。

(2) 尽量小的体积

较小的文件会不仅可以减少测试和处理的时间，也能节约更多的内存，AFL给出的建议是最好

小于1 KB，但其实可以根据自己测试的程序权衡，这在AFL文档的 perf_tips.txt 中有具体说明。

2. 寻找

3. 修剪

网上找到的一些大型语料库中往往包含大量的文件，这时就需要对其精简，这个工作有个术语叫做——语料库蒸馏（Corpus Distillation）。AFL提供了两个工具来帮助我们完成这部工作——afl-cmin 和 afl-tmin 。

(1) 移除执行相同代码的输入文件——afl-cmin

afl-cmin 的核心思想是：**尝试找到与语料库全集具有相同覆盖范围的最小子集**。举个例子：假设有多个文件，都覆盖了相同的代码，那么就丢掉多余的文件。其使用方法如下：

```
$ afl-cmin -i input_dir -o output_dir -- /path/to/tested/program [params]
```

更多的时候，我们需要从文件中获取输入，这时可以使用“@@”代替被测试程序命令行中输入文件名的位置。Fuzzer会将其替换为实际执行的文件：

```
$ afl-cmin -i input_dir -o output_dir -- /path/to/tested/program [params] @@
```

下面的例子中，我们将一个有1253个png文件的语料库，精简到只包含60个文件。

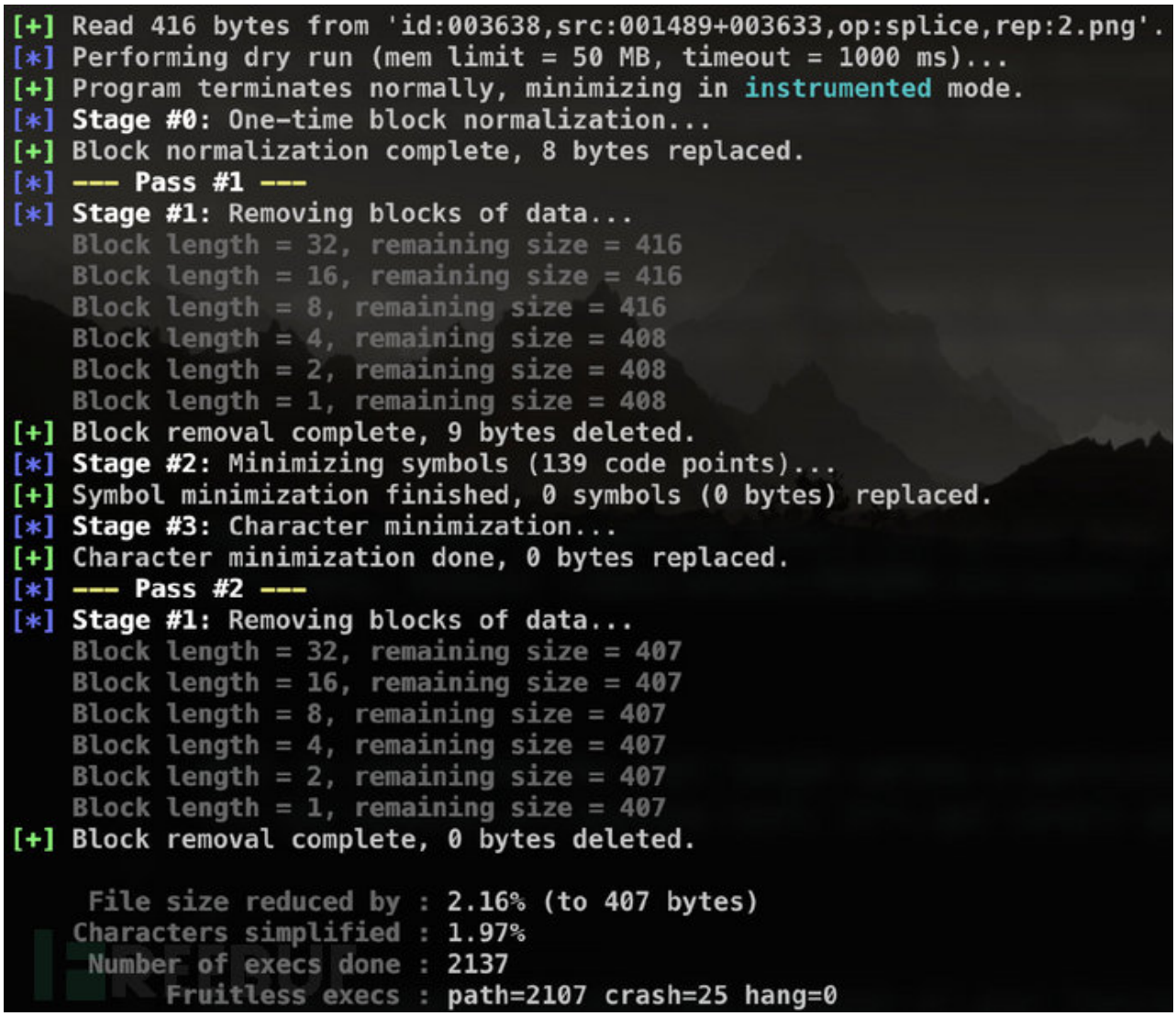
```
[*] Testing the target binary...
[+] OK, 86 tuples recorded.
[*] Obtaining traces for input files in 'png'...
    Processing file 1253/1253...
[*] Sorting trace sets (this may take a while)...
[+] Found 1525 unique tuples across 1253 files.
[*] Finding best candidates for each tuple...
    Processing file 1253/1253...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
    Processing tuple 1525/1525...
[+] Narrowed down to 60 files, saved in 'png-cmin'.
```

(2) 减小单个输入文件的大小——afl-tmin

整体的大小得到了改善，接下来还要对每个文件进行更细化的处理。afl-tmin缩减文件体积的原理这里就不深究了，有机会会在后面文章中解释，这里只给出使用方法（其实也很简单，有兴趣的朋友可以自己搜一搜）。

afl-tmin 有两种工作模式， instrumented mode 和 crash mode 。默认的工作方式是 instrumented mode ，如下所示：

$ afl-tmin -i input_file -o output_file -- /path/to/tested/program [params] @@



如果指定了参数 `-x`，即 `crash mode`，会把导致程序非正常退出的文件直接剔除。

$ afl-tmin -x -i input_file -o output_file -- /path/to/tested/program [params] @@

```
[+] Read 272 bytes from 'id:002247,src:002216,op:flip2,pos:23.png'.
[*] Performing dry run (mem limit = 0 MB, timeout = 1000 ms)...
[+] Program exits with a signal, minimizing in crash mode.
[*] Stage #0: One-time block normalization...
[+] Block normalization complete, 272 bytes replaced.
[*] --- Pass #1 ---
[*] Stage #1: Removing blocks of data...
    Block length = 32, remaining size = 272
[+] Block removal complete, 272 bytes deleted.
[!] WARNING: Down to zero bytes - check the command line and mem limit!
[*] Stage #2: Minimizing symbols (0 code points)...
[+] Symbol minimization finished, 0 symbols (0 bytes) replaced.
[*] Stage #3: Character minimization...
[+] Character minimization done, 0 bytes replaced.
[*] --- Pass #2 ---
[*] Stage #1: Removing blocks of data...
    Block length = 1, remaining size = 0
[+] Block removal complete, 0 bytes deleted.

     File size reduced by : 100.00% (to 0 bytes)
    Characters simplified : 27200.00%
     Number of execs done : 78
          Fruitless execs : path=0 crash=0 hang=0
```

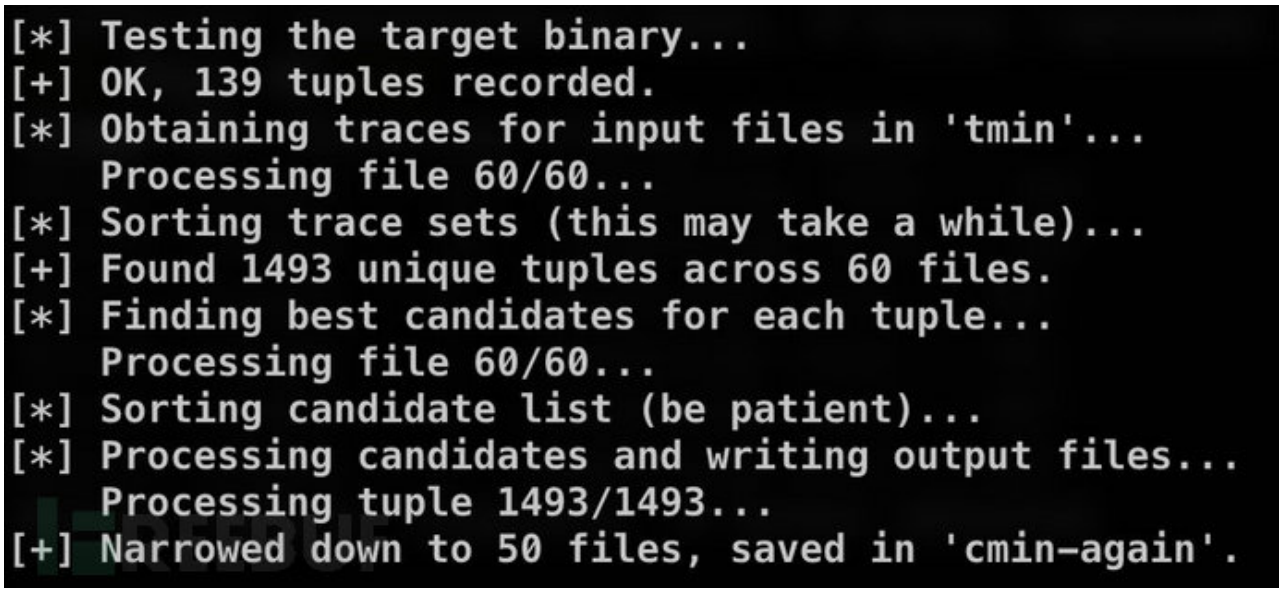
afl-tmin 接受单个文件输入，所以可以用一条简单的shell脚本批量处理。如果语料库中文件数量特别多，且体积特别大的情况下，这个过程可能花费几天甚至更长的时间！

```
for i in *; do afl-tmin -i $i -o tmin-$i -- ~/path/to/tested/program [params] @@; done;
```

下图是经过两种模式的修剪后，语料库大小的变化：

```
# root @ ubuntu in ~/png-cmin [0:56:07]
$ du && cd ../tmin && du && cd ../tmin-crash && du
316K    .
308K    .
184K    .
```

这时还可以再次使用 afl-cmin ，发现又可以过滤掉一些文件了。



五、构建被测试程序

前面说到，AFL从源码编译程序时进行插桩，以记录代码覆盖率。这个工作需要使用其提供的两种编译器的wrapper编译目标程序，和普通的编译过程没有太大区别，本节就只简单演示一下。

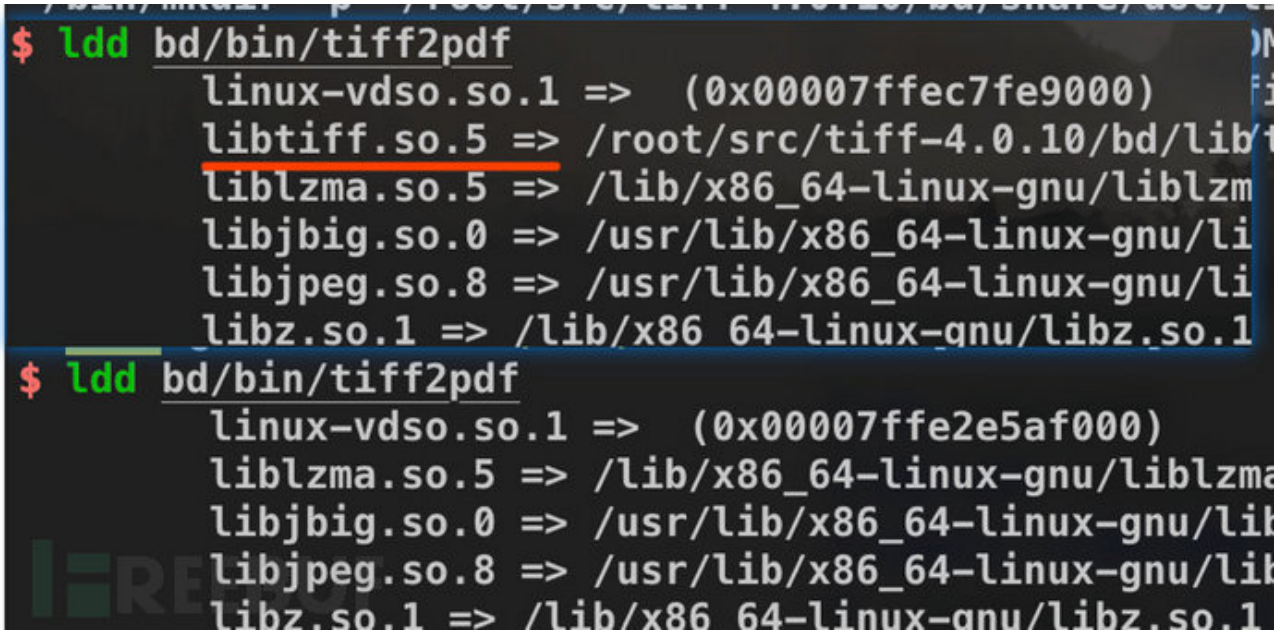
1. afl-gcc模式

`afl-gcc` / `afl-g++` 作为 `gcc` / `g++` 的wrapper，它们的用法完全一样，前者会将接收到的参数传递给后者，我们编译程序时只需要将编译器设置为 `afl-gcc` / `afl-g++` 就行，如下面演示的那样。如果程序不是用autoconf构建，直接修改 `Makefile` 文件中的编译器为 `afl-gcc/g++` 也行。

```
$ ./configure CC="afl-gcc" CXX="afl-g++"
```

在Fuzzing共享库时，可能需要编写一个简单demo，将输入传递给要Fuzzing的库（其实大多数项目中都自带了类似的demo）。这种情况下，可以通过设置 `LD_LIBRARY_PATH` 让程序加载经过AFL插桩的.so文件，不过最简单的方法是静态构建，通过以下方式实现：

```
$ ./configure --disable-shared CC="afl-gcc" CXX="afl-g++"
```



2. LLVM模式

LLVM Mode模式编译程序可以获得更快的Fuzzing速度，进入 `llvm_mode` 目录进行编译，之后使用 `afl-clang-fast` 构建序程序即可，如下所示：

```
$ cd llvm_mode$ apt-get install clang$ export LLVM_CONFIG=`which llvm-config` && make && cd ..$ ./configure --disable-shared CC="afl-clang-fast" CXX="afl-clang-fast++"
```

笔者在使用高版本的clang编译时会报错，换成clang-3.9后通过编译，如果你的系统默认安装的clang版本过高，可以安装多个版本然后使用 `update-alternatives` 切换。

六、开始Fuzzing

`afl-fuzz` 程序是AFL进行Fuzzing的主程序，用法并不难，但是其背后巧妙的工作原理很值得研究，考虑到第一篇文章只是让读者有个初步的认识，这节只简单的演示如何将Fuzzer跑起来，其他具体细节这里就暂时略过。

1. 白盒测试

(1) 测试插桩程序

编译好程序后，可以选择使用 `afl-showmap` 跟踪单个输入的执行路径，并打印程序执行的输出、捕获的元组（tuples），tuple用于获取分支信息，从而衡量衡量程序覆盖情况，下一篇文章中会详细的解释，这里可以先不用管。

```
$ afl-showmap -m none -o /dev/null -- ./build/bin/imagew 23.bmp out.png[*] Executing './build/bin/imagew'...-- Program output begins --23.bmp -> out.pngProcessing: 13x32-- Program output ends --[+] Captured 1012 tuples in '/dev/null'.
```

使用不同的输入，正常情况下 `afl-showmap` 会捕获到不同的tuples，这就说明我们的的插桩是有效的，还有前面提到的 `afl-cmin` 就是通过这个工具来去掉重复的输入文件。

$ $ afl-showmap -m none -o /dev/null -- ./build/bin/imagew 111.pgm out.png[*] Executing './build/bin/imagew'...-- Program output begins --111.pgm -> out.pngProcessing: 7x7-- Program output ends --[+] Captured 970 tuples in '/dev/null'.

(2) 执行fuzzer

在执行 `afl-fuzz` 前，如果系统配置为将核心转储文件（core）通知发送到外部程序。 将导致将崩溃信息发送到Fuzzer之间的延迟增大，进而可能将崩溃被误报为超时，所以我们得临时修改 `core_pattern` 文件，如下所示：

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

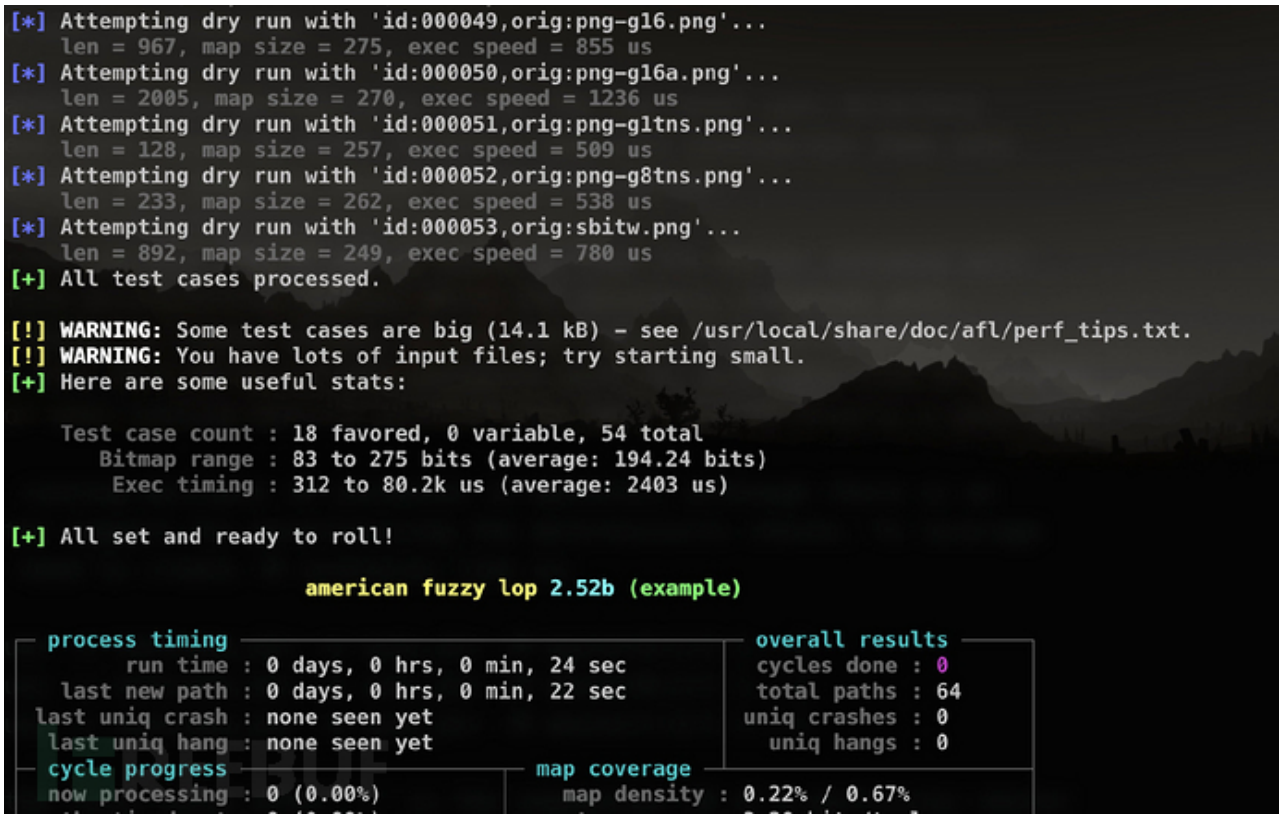
之后就可以执行 `afl-fuzz` 了，通常的格式是：

```
$ afl-fuzz -i testcase_dir -o findings_dir /path/to/program [params]
```

或者使用“@@”替换输入文件，Fuzzer会将其替换为实际执行的文件：

```
$ afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

如果没有什么错误，Fuzzer就正式开始工作了。首先，对输入队列中的文件进行预处理；然后给出对使用的语料库可警告信息，比如下图中提示有个较大的文件（14.1KB），且输入文件过多；最后，开始Fuzz主循环，显示状态窗口。



(3) 使用screen

一次Fuzzing过程通常会持续很长时间，如果这期间运行afl-fuzz实例的终端终端被意外关闭了，那么Fuzzing也会被中断。而通过在 `screen session` 中启动每个实例，可以方便的连接和断开。关于screen的用法这里就不再多讲，大家可以自行查询。

```
$ screen afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

也可以为每个session命名，方便重新连接。

```
$ screen -S fuzzer1$ afl-fuzz -i testcase_dir -o findings_dir /path/to/program [params] @@[detached from 6999.fuzzer1]$ screen -r fuzzer1  ...
```

2. 黑盒测试

所谓黑盒测试，通俗地讲就是对没有源代码的程序进行测试，这时就要用到AFL的QEMU模式了。启用方式和LLVM模式类似，也要先编译。但注意，因为AFL使用的QEMU版本太旧，`util/memfd.c`中定义的函数`memfd_create()`会和glibc中的同名函数冲突，在这里可以找到针对QEMU的patch，之后运行脚本`build_qemu_support.sh`就可以自动下载编译。

```
$ apt-get install libini-config-dev libtool-bin automake bison libglib2.0-dev -y$ cd qemu_mode$ build_qemu_support.sh$ cd .. && make install
```

现在起，只需添加`-Q`选项即可使用QEMU模式进行Fuzzing。

```
$ afl-fuzz -Q -i testcase_dir -o findings_dir /path/to/program [params] @@
```

3. 并行测试

(1) 单系统并行测试

如果你有一台多核心的机器，可以将一个`afl-fuzz`实例绑定到一个对应的核心上，也就是说，机器上有几个核心就可以运行多少`afl-fuzz`实例，这样可以极大的提升执行速度，虽然大家都应该知道自己的机器的核心数，不过还是提一下怎么查看吧：

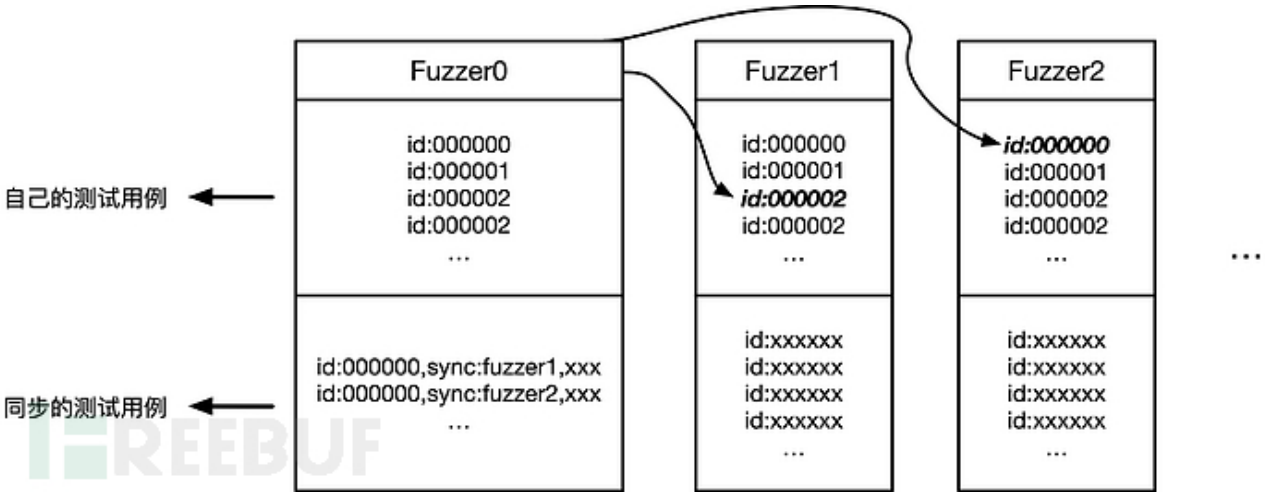
```
$ cat /proc/cpuinfo| grep "cpu cores"| uniq
```

`afl-fuzz`并行Fuzzing，一般的做法是通过`-M`参数指定一个主Fuzzer(`Master Fuzzer`)、通过`-S`参数指定多个从Fuzzer(`Slave Fuzzer`)。

```
$ screen afl-fuzz -i testcases/ -o sync_dir/ -M fuzzer1 -- ./program$ screen afl-fuzz -i testcases/ -o sync_dir/ -S fuzzer2 -- ./program$ screen afl-fuzz -i testcases/ -o sync_dir/ -S fuzzer3 -- ./program  ...
```

这两种类型的Fuzzer执行不同的Fuzzing策略，前者进行确定性测试（deterministic），即对输入文件进行一些特殊而非随机的的变异；后者进行完全随机的变异。

可以看到这里的`-o`指定的是一个同步目录，并行测试中，所有的Fuzzer将相互协作，在找到新的代码路径时，相互传递新的测试用例，如下图中以Fuzzer0的角度来看，它查看其它fuzzer的语料库，并通过比较id来同步感兴趣的测试用例。



`afl-whatsup` 工具可以查看每个fuzzer的运行状态和总体运行概况，加上 `-s` 选项只显示概况，其中的数据都是所有fuzzer的总和。

```
$ afl-whatsup -s syncdir
status check tool for afl-fuzz by <lcamtuf@google.com>

Summary stats
=============

       Fuzzers alive : 12
      Total run time : 70 days, 22 hours
         Total execs : 930 million
    Cumulative speed : 1814 execs/sec
       Pending paths : 10 faves, 1117 total
  Pending per fuzzer : 0 faves, 93 total (on average)
       Crashes found : 0 locally unique
```

还 `afl-gotcpu` 工具可以查看每个核心使用状态。

(2) 多系统并行测试

多系统并行的基本工作原理类似于单系统并行中描述的机制，你需要一个简单的脚本来完成两件事。在本地系统上，压缩每个fuzzer实例目录中 `queue` 下的文件，通过SSH分发到其他机器上解压。

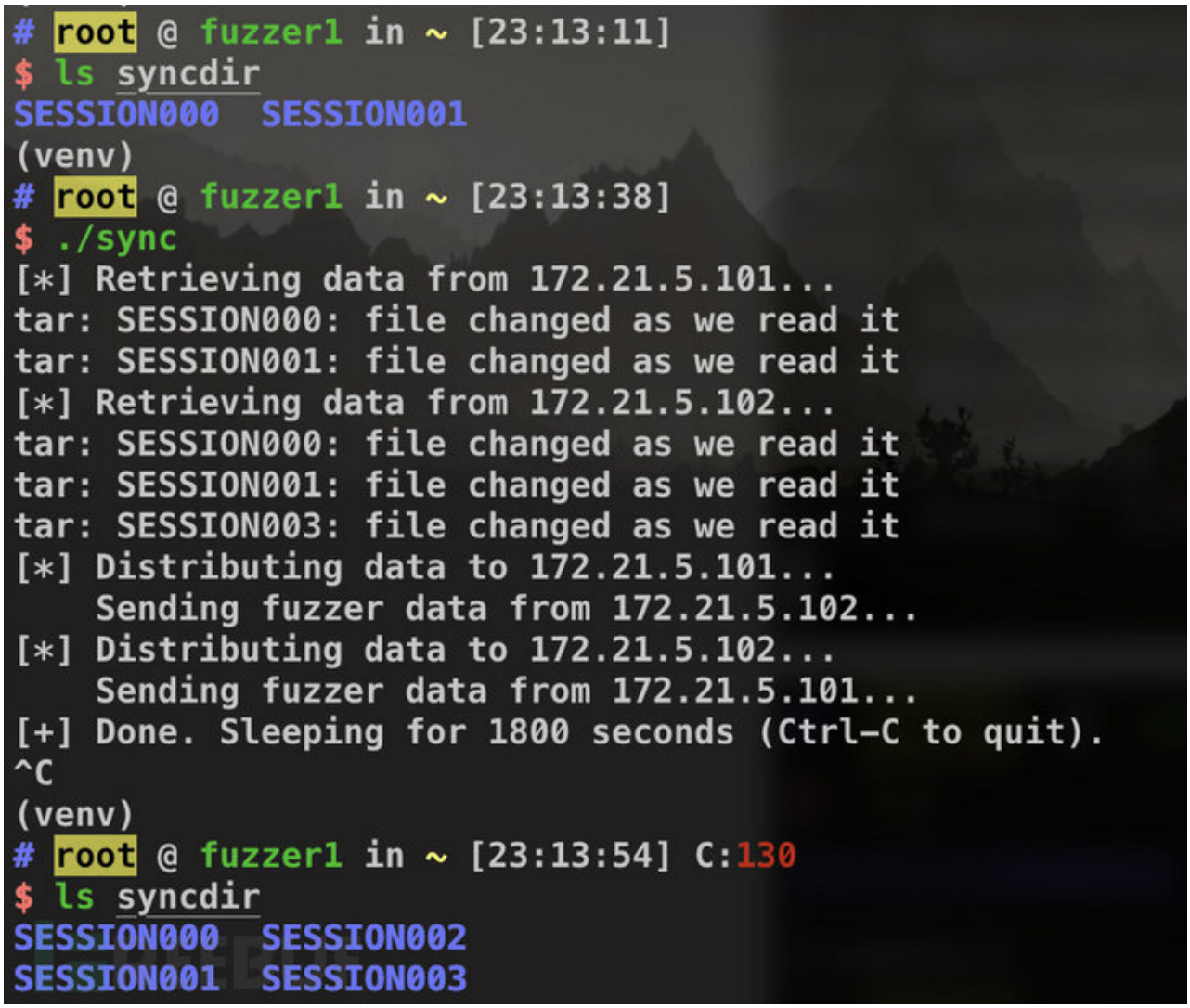
来看一个例子，假设现在有两台机器，基本信息如下：

fuzzer1	fuzzerr2
172.21.5.101	172.21.5.102
运行2个实例	运行4个实例

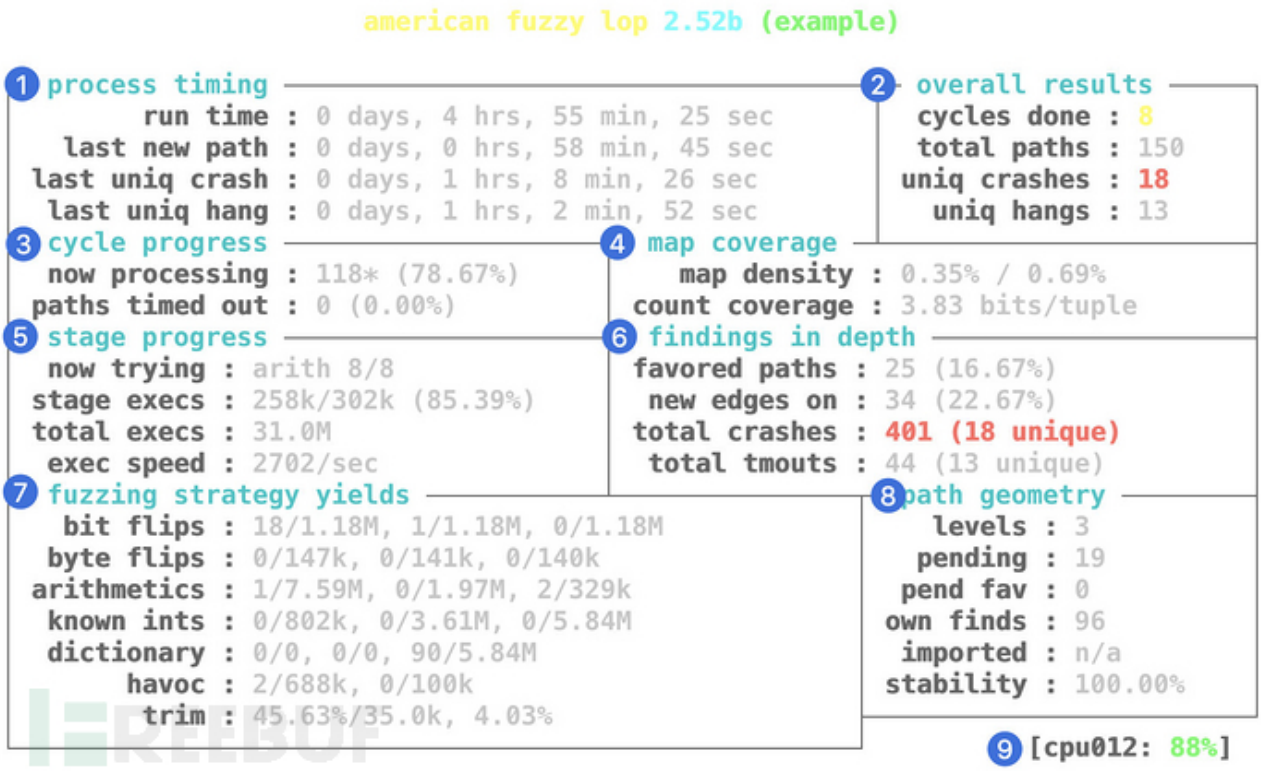
为了能够自动同步数据，需要使用 `authorized_keys` 的方式进行身份验证。现要将fuzzer2中每个实例的输入队列同步到fuzzer1中，可以下面的方式：

#!/bin/sh# 所有要同步的主机FUZZ_HOSTS='172.21.5.101 172.21.5.102'# SSH userFUZZ_USER=root# 同步目录SYNC_DIR='/root/syncdir'# 同步间隔时间SYNC_INTERVAL=$((30 * 60))if ["$AFL_ALLOW_TMP" = ""]; then if ["$PWD" = "/tmp" -o "$PWD" = "/var/tmp"]; then echo "[-] Error: do not use shared /tmp or /var/tmp directories with this script." 1>&2 exit 1 fifirm -rf .sync_tmp 2>/dev/nullmkdir .sync_tmp || exit 1while :; do # 打包所有机器上的数据 for host in $FUZZ_HOSTS; do echo "[*] Retrieving data from ${host}..." ssh -o 'passwordauthentication no' ${FUZZ_USER}@${host} \ "cd '$SYNC_DIR' && tar -czf - SESSION*" >".sync_tmp/${host}.tgz" done # 分发数据 for dst_host in $FUZZ_HOSTS; do echo "[*] Distributing data to ${dst_host}..." for src_host in $FUZZ_HOSTS; do test "$src_host" = "$dst_host" && continue echo " Sending fuzzer data from ${src_host}..." ssh -o 'passwordauthentication no' ${FUZZ_USER}@$dst_host \ "cd '$SYNC_DIR' && tar -xkzf - &>/dev/null" <".sync_tmp/${src_host}.tgz" done done echo "[+] Done. Sleeping for $SYNC_INTERVAL seconds (Ctrl-C to quit)." sleep $SYNC_INTERVAL done

成功执行上述shell脚本后，不仅 `SESSION000` `SESSION002` 中的内容更新了，还将 `SESSION003` `SESSION004` 也同步了过来。



七、认识AFL状态窗口



① Process timing:Fuzzer运行时长、以及距离最近发现的路径、崩溃和挂起经过了多长时间。

② Overall results：Fuzzer当前状态的概述。

③ Cycle progress：我们输入队列的距离。

④ Map coverage：目标二进制文件中的插桩代码所观察到覆盖范围的细节。

⑤ Stage progress：Fuzzer现在正在执行的文件变异策略、执行次数和执行速度。

⑥ Findings in depth：有关我们找到的执行路径，异常和挂起数量的信息。

⑦ Fuzzing strategy yields：关于突变策略产生的最新行为和结果的详细信息。

⑧ Path geometry：有关Fuzzer找到的执行路径的信息。

⑨ CPU load：CPU利用率

八、总结

到此为止，本文已经介绍完了如何开始一次Fuzzing，但这仅仅是一个开始。AFL 的Fuzzing过程是一个死循环，我们需要人为地停止，那么什么时候停止？上面图中跑出的18个特别的崩溃，又如何验证？还有文中提到的各种概念——代码覆盖率、元组、覆盖引导等等又是怎么回事？所谓学非探其花，要自拔其根，学会工具的基本用法后，要想继续进阶的话，掌握这些基本概念相当重要，也有助于理解更深层次内容。所以后面的几篇文章，首先会继续本文中未完成的工作，然后详细讲解重要概念和AFL背后的原理，敬请各位期待。

参考资料

[1]American Fuzzy Lop

[2]Yet another memory leak in ImageMagick

[3]Vulnerability Discovery Against Apple Safari

[4]Pulling JPEGs out of thin air

[5]parallel_fuzzing.txt

[6]Fuzzing workflows; a fuzz job from start to finish

[7]Open Source Fuzzing Tools – ‘Chapter 10 Code Coverage and Fuzzing’

[8]Fuzzing for Software Security Testing and Quality Assurance – ’7.2 Using Code Coverage Information’

***本文作者：alphalab，转载请注明来自FreeBuf.COM**