

# AFL漏洞挖掘技术漫谈（二）：Fuzz结果分析和代码覆盖率 - FreeBuf互联网安全新媒体平台

 [freebuf.com/articles/system/197678.html](https://freebuf.com/articles/system/197678.html)

alphalab

## AFL漏洞挖掘技术漫谈（二）：Fuzz结果分析和代码覆盖率

### 一、前言

阿尔法实验在上一篇文章中向大家介绍了使用AFL开始模糊测试前要做的一些准备工作，以及AFL的几种工作方式，但是并没有提到何时结束测试过程，以及测试完成后又需要做些什么。本文中就继续介绍这些内容，并开始逐步介绍一些AFL相关原理，以下是本文中主要讨论的问题：

1. 何时结束Fuzzing工作
2. afl-fuzz生成了哪些文件
3. 如何对产生的crash进行验证和分类
4. 用什么来评估Fuzzing的结果
5. 代码覆盖率及相关概念
6. AFL是如何记录代码覆盖率的

### 二、Fuzzer工作状态

因为afl-fuzz永远不会停止，所以何时停止测试很多时候就是依靠afl-fuzz提供的状态来决定的。除了前面提到过的通过状态窗口、afl-whatsup查看afl-fuzz状态外，这里再补充几种方法。

#### 1. afl-stat

afl-stat是afl-utils这套工具AFL辅助工具中的一个（这套工具中还有其他更好用的程序，后面用到时会做介绍），该工具类似于afl-whatsup的输出结果。

使用前需要一个配置文件，设置每个afl-fuzz实例的输出目录：

```
{
  "fuzz_dirs": [
    "/root/syncdir/SESSION000",
    "/root/syncdir/SESSION001",
    ...
    "/root/syncdir/SESSION00x"
  ]
}
```

然后指定配置文件运行即可：

```
$ afl-stats -c afl-stats.conf
[SESSION000 on fuzzer1]
Alive:    1/1
Execs:    64 m
Speed:    0.3 x/s
Pend:     6588/249
Crashes: 101
[SESSION001 on fuzzer1]
Alive:    1/1
Execs:    105 m
Speed:    576.6 x/s
Pend:     417/0
Crashes: 291
...
```

## 2. 定制afl-whatsup

afl-whatsup是依靠读afl-fuzz输出目录中的fuzzer\_stats文件来显示状态的，每次查看都要需要手动执行，十分麻烦。因此可以对其进行修改，让其实时显示fuzzer的状态。方法也很简答，基本思路就是在所有代码外面加个循环就好，还可以根据自己的喜好做些调整：



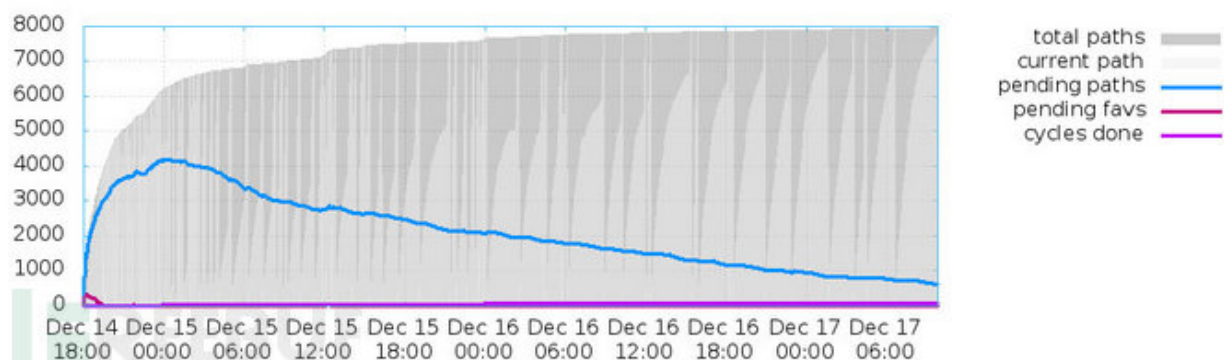
## 3. afl-plot

前面提到的都是基于命令行的工具，如果还想要更直观的结果，可以用afl-plot绘制各种状态指标的直观变化趋势。

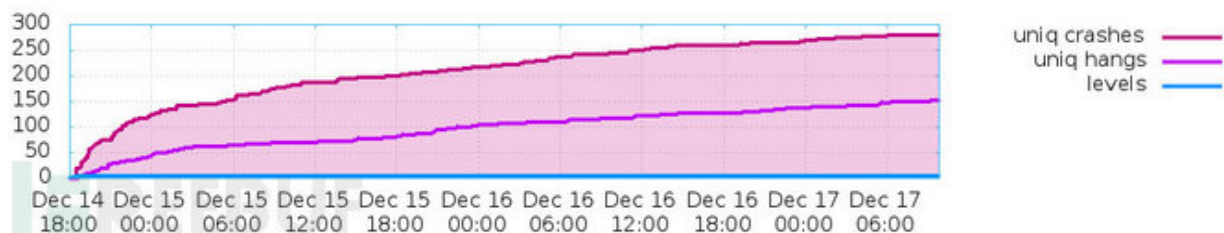
```
$ apt-get install gnuplot
$ afl-plot afl_state_dir graph_output_dir
```

以测试libtiff的情况为例，进入afl-plot输出目录，打开index.html，会看到下面三张图：

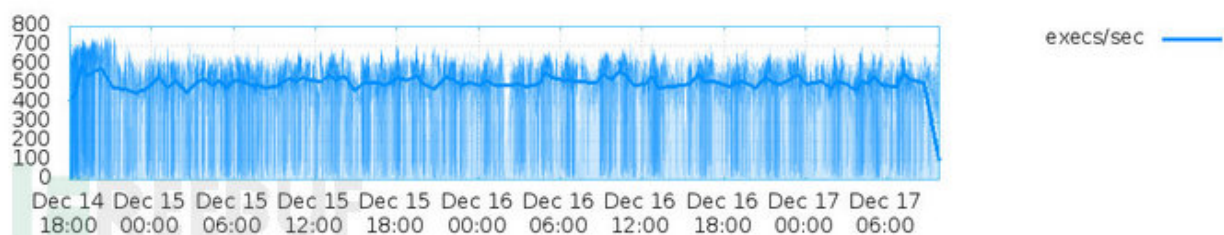
首先是路径覆盖的变化，当**pending fav**的数量变为零并且**total paths**数量基本上没有再增长时，说明fuzzer有新发现的可能性就很小了。



接着是崩溃和超时的变化



最后是执行速度的变化，这里要注意的是，如果随着时间的推移，执行速度越来越慢，有一种可能是由于fuzzer耗尽一些共享资源。



## 4. pythia

笔者在查阅资料的过程中，还发现了pythia这个AFL的扩展项目，虽然不知道效果如何，但这里还是顺便提一提。其特色在于可以估算发现新crash和path概率，其运行界面相比原版的AFL多出了下面几个字段：

process timing	overall results
run time : 0 days, 0 hrs, 34 min, 23 sec	cycles done : 0
last new path : 0 days, 0 hrs, 0 min, 4 sec	current paths : 944
last uniq crash : 0 days, 0 hrs, 27 min, 23 sec	path coverag : 53.8%
last uniq hang : 0 days, 0 hrs, 29 min, 8 sec	uniq crashes : 4
correctness : 1.091064e-04	uniq hangs : 1
fuzzability : 2.019915e+00	effec paths : 7.538

correctness: 在没有发现crash时, 发现一个导致crash输入的概率。

fuzzability: 表示在该程序中发现新路径的难度, 该数值越高代表程序更容易Fuzz。

current paths: 显示当前发现的路径数。

path coverag: 路径覆盖率。

### 三、结束测试

#### 1.何时结束

检查afl-fuzz工作状态的目的是为何时停止测试提供依据, 通常来说符合下面几种情况时就可以停掉了。

(1) 状态窗口中“cycles done”字段颜色变为绿色该字段的颜色可以作为何时停止测试的参考, 随着周期数不断增大, 其颜色也会由洋红色, 逐步变为黄色、蓝色、绿色。当其变为绿色时, 继续Fuzzing下去也很难有新的发现了, 这时便可以通过Ctrl-C停止afl-fuzz。

(2) 距上一次发现新路径(或者崩溃)已经过去很长时间了, 至于具体多少时间还是需要自己把握, 比如长达一个星期或者更久估计大家也都没啥耐心了吧。

(3) 目标程序的代码几乎被测试用例完全覆盖, 这种情况好像很少见, 但是对于某些小型程序应

overall results
cycles done : 7436
total paths : 186
uniq crashes : 23
uniq hangs : 22

process timing
run time : 1 days, 23 hrs, 30 min, 30 sec
last new path : 0 days, 10 hrs, 19 min, 32 sec
last uniq crash : 0 days, 5 hrs, 49 min, 40 sec
last uniq hang : 1 days, 4 hrs, 49 min, 26 sec

该还是可能的，至于如何计算覆盖率将在下面介绍。

(4) 上面提到的pythia提供的各种数据中，一旦**path cover**达到99%（通常来说不太可能），如果不期望再跑出更多crash的话就可以中止fuzz了，因为很多crash可能是因为相同的原因导致的；还有一点就是**correctness**的值达到**1e-08**，根据pythia开发者的说法，这时从上次发现path/uniq crash到下一次发现之间大约需要1亿次执行，这一点也可以作为衡量依据。

## 2. 输出结果

afl-fuzz的输出目录中存在很多文件，有时想要写一个辅助工具可能就要用到其中的文件。下面以多个fuzz实例并行测试时的同步目录为例：

```
$ tree -L 3
.
├─ fuzzer1
│   ├── crashes
│   │   ├── id:000000,sig:06,src:000019+000074,op:splice,rep:2
│   │   ├── ...
│   │   └─ id:000002,sig:06,src:000038+000125,op:splice,rep:4
│   │   └─ README.txt
│   ├── fuzz_bitmap
│   ├── fuzzer_stats
│   ├── hangs
│   │   └─ id:000000,src:000007,op:flip1,pos:55595
│   ├── plot_data
│   └─ queue
│       └─ id:000000,orig:1.png
├─ ....
│   └─ id:000101,src:fuzzer10,src:000102
└─ fuzzer2
    ├── crashes
    └─ ...
```

queue：存放所有具有独特执行路径的测试用例。

crashes：导致目标接收致命signal而崩溃的独特测试用例。

crashes/README.txt：保存了目标执行这些crash文件的命令行参数。

hangs：导致目标超时的独特测试用例。

fuzzer\_stats：afl-fuzz的运行状态。

plot\_data：用于afl-plot绘图。

## 四、处理测试结果

到了这里，我们可能已经跑出了一大堆的crashes，那么接下来的步骤，自然是确定造成这些crashes的bug是否可以利用，怎么利用？这是另一个重要方面。当然，个人觉得这比前面提到的内容都要困难得多，这需要对常见的二进制漏洞类型、操作系统的安全机制、代码审计和调试等内容



都有一定深度的了解。但如果只是对crash做简单的分析和分类，那么下面介绍的几种方法都可以给我们提供一些帮助。

## 1. crash exploration mode

这是afl-fuzz的一种运行模式，也称为**peruvian rabbit mode**，用于确定bug的可利用性，具体细节可以参考lcamtuf的博客。

```
$ afl-fuzz -m none -C -i poc -o peruvian-were-rabbit_out -- ~/src/LuPng/a.out @@ out.png
```

peruvian were-rabbit 2.52b[pythia] (a.out)			
process timing		overall results	
run time : 0 days, 0 hrs, 15 min, 44 sec		cycles done : 575	
last new path : 0 days, 0 hrs, 2 min, 7 sec		current paths : 8	
last uniq crash : 0 days, 0 hrs, 15 min, 5 sec		path coverag : 1.0%	
last uniq hang : none seen yet		uniq crashes : 4	
correctness : 0.000000e+00		uniq hangs : 0	
fuzzability : -nan		effec paths : infity	
cycle progress		map coverage	
now processing : 6* (75.00%)		map density : 0.05% / 0.23%	
paths timed out : 0 (0.00%)		count coverage : 1.44 bits/tuple	
stage progress		findings in depth	
now trying : havoc		favored paths : 2 (25.00%)	
stage execs : 553/2293 (24.12%)		new edges on : 1 (12.50%)	
total execs : 3.81M		new crashes : 86.6k (4 unique)	
exec speed : 3985/sec		total tmouts : 4 (2 unique)	
fuzzing strategy yields		path geometry	
bit flips : 0/80.2k, 0/80.2k, 8/80.1k		levels : 5	
byte flips : 0/10.0k, 0/10.0k, 0/9996		pending : 0	
arithmetics : 0/558k, 0/107k, 0/10.7k		pend fav : 0	
known ints : 0/62.9k, 0/274k, 0/435k		own finds : 7	
dictionary : 0/0, 0/0, 0/39.7k		imported : n/a	
havoc : 2/986k, 1/1.06M		stability : 100.00%	
trim : 95.92%/1778, 0.00%			
[cpu000:199%]			

举个例子，当你发现目标程序尝试写入\跳转到一个明显来自输入文件的内存地址，那么就可以猜测这个bug应该是可以利用的；然而遇到例如NULL pointer dereferences这样的漏洞就没那么容易判断了。

将一个导致crash测试用例作为afl-fuzz的输入，使用-C选项开启crash exploration模式后，可以快速地产生很多和输入crash相关、但稍有些不同的crashes，从而判断能够控制某块内存地址的长度。这里笔者在实践中没有找到适合的例子，但在一篇文章中发现了一个很不错的例子——tcpdump栈溢出漏洞，crash exploration模式从一个crash产生了42个新的crash，并读取不同大小的相邻内存。

```
$ for i in `ls` ; do ASAN_OPTIONS=symbolize=1 ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
tcpdump -ee -vv -nnr $i 2>&1 ; done | grep READ | sort | uniq -c
    18 READ of size 102 at 0xfffffb27e thread T0
    22 READ of size 102 at 0xfffffb29e thread T0
     1 READ of size 62 at 0xfffffb27e thread T0
     1 READ of size 62 at 0xfffffb29e thread T0
```

## 2. triage\_crashes

AFL源码的experimental目录中有一个名为triage\_crashes.sh的脚本，可以帮助我们触发收集到的crashes。例如下面的例子中，11代表了SIGSEGV信号，有可能是因为缓冲区溢出导致进程引用了无效的内存；06代表了SIGABRT信号，可能是执行了abort/assert函数或double free导致，这些结果可以作为简单的参考。

```
$ ~/afl-2.52b/experimental/crash_triage/triage_crashes.sh fuzz_out ~/src/LuPng/a.out @@
out.png 2>&1 | grep SIGNAL
+++ ID 000000, SIGNAL 11 +++
+++ ID 000001, SIGNAL 06 +++
+++ ID 000002, SIGNAL 06 +++
+++ ID 000003, SIGNAL 06 +++
+++ ID 000004, SIGNAL 11 +++
+++ ID 000005, SIGNAL 11 +++
+++ ID 000006, SIGNAL 11 +++
...
```

## 3. crashwalk

当然上面的两种方式都过于鸡肋了，如果你想得到更细致的crashes分类结果，以及导致crashes的具体原因，那么crashwalk就是不错的选择之一。这个工具基于gdb的exploitable插件，安装也相对简单，在ubuntu上，只需要如下几步即可：

```
$ apt-get install gdb golang
$ mkdir tools
$ cd tools
$ git clone https://github.com/jfoote/exploitable.git
$ mkdir go
$ export GOPATH=~/tools/go
$ export CW_EXPLOITABLE=~/tools/exploitable/exploitable/exploitable.py
$ go get -u github.com/bnagy/crashwalk/cmd/...
```

crashwalk支持AFL/Manual两种模式。前者通过读取crashes/README.txt文件获得目标的执行命令（前面第三节中提到的），后者则可以手动指定一些参数。两种使用方式如下：

```
#Manual Mode
$ ~/tools/go/bin/cwtriage -root syncdir/fuzzer1/crashes/ -match id -- ~/parse @@
#AFL Mode
$ ~/tools/go/bin/cwtriage -root syncdir -afl
```

两种模式的输出结果都一样，如上图所示。这个工具比前面几种方法要详细多了，但当有大量crashes时结果显得还是十分混乱。

```

---CRASH SUMMARY---
Filename: syncdir/fuzzer11/crashes/id:000016,sig:06,src:000057+000008,op:splice,rep:2
SHA1: ed5480c5ee7db00c5cd8f265b18e9c60ba3a25e1
Classification: EXPLOITABLE
Hash: f750cdb1c5f8f7f1e08f813d9c97f1d.a5311ccbf5e531e25de4dc74587c62cc
Command: /root/src/LuPng/lupng syncdir/fuzzer11/crashes/id:000016,sig:06,src:000057+000008,op:splice,rep:2 /dev/null
Faulting Frame:
  releaseChunk @ 0x000000000402ce2: in /root/src/LuPng/lupng
Disassembly:
Stack Head (10 entries):
  __GI_raise @ 0x00007ffff7828428: in (BL)
  __GI_abort @ 0x00007ffff782a02a: in (BL)
  __libc_message @ 0x00007ffff786a7ea: in (BL)
  malloc_printerr @ 0x00007ffff787337a: in (BL)
  __int_free @ 0x00007ffff787337a: in (BL)
  __GI__libc_free @ 0x00007ffff787753c: in (BL)
  releaseChunk @ 0x000000000402ce2: in /root/src/LuPng/lupng
  luPngReadUC @ 0x000000000402ce2: in /root/src/LuPng/lupng
  luPngReadFile @ 0x0000000004034a7: in /root/src/LuPng/lupng
  main @ 0x000000000401036: in /root/src/LuPng/lupng
Registers:
rax=0x0000000000000000 rbx=0x000000000000005e rcx=0x00007ffff7828428 rdx=0x0000000000000006
rsi=0x00000000000001042 rdi=0x00000000000001042 rbp=0x00007ffff7ffc020 rsp=0x00007ffff7fbc88
r8=0x0000000000000005 r9=0x0000000000000000 r10=0x0000000000000008 r11=0x0000000000000246
r12=0x000000000000005e r13=0x00007ffff7f7be38 r14=0x00007ffff7f7be38 r15=0x0000000000000002
rip=0x00007ffff7828428 efl=0x0000000000000246 cs=0x0000000000000033 ss=0x000000000000002b
ds=0x0000000000000000 es=0x0000000000000000 fs=0x0000000000000000 gs=0x0000000000000000
Extra Data:
  Description: Heap error
  Short description: HeapError (10/22)
  Explanation: The target's backtrace indicates that libc has detected a heap error or that the target was executing
t stopped. This could be due to heap corruption, passing a bad pointer to a heap function such as free(), etc. Since
de buffer overflows, use-after-free situations, etc. they are generally considered exploitable.
---END SUMMARY---

```

## 4. afl-collect

最后重磅推荐的工具便是afl-collect，它也是afl-utils套件中的一个工具，同样也是基于exploitable来检查crashes的可利用性。它可以自动删除无效的crash样本、删除重复样本以及自动化样本分类。使用起来命令稍微长一点，如下所示：

```
$ afl-collect -j 8 -d crashes.db -e gdb_script ./afl_sync_dir ./collection_dir --
/path/to/target --target-opts
```

但是结果就像下面这样非常直观：



```

*** GDB+EXPLOITABLE SCRIPT OUTPUT ***
[00001] output:id:000000,sig:06,src:000012,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00002] output:id:000004,sig:06,src:000048,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00003] output:id:000008,sig:06,src:000067,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00004] output:id:000012,sig:11,src:000021,op:ext_A0,pos:12.....: PROBABLY_NOT_EXPLOITABLE [Source]
[00005] output:id:000016,sig:06,src:000041,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00006] output:id:000020,sig:11,src:000167+000134,op:splice,rep:8.....: EXPLOITABLE [DestAv (8/22)]
[00007] output:id:000001,sig:06,src:000019,op:ext_A0,pos:74.....: EXPLOITABLE [HeapError (10/22)]
[00008] output:id:000005,sig:06,src:000048,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00009] output:id:000009,sig:06,src:000068,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00010] output:id:000013,sig:06,src:000028,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00011] output:id:000017,sig:06,src:000084,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00012] output:id:000021,sig:06,src:000011,op:havoc,rep:2.....: EXPLOITABLE [HeapError (10/22)]
[00013] output:id:000002,sig:11,src:000024,op:ext_A0,pos:12.....: EXPLOITABLE [DestAv (8/22)]
[00014] output:id:000006,sig:06,src:000048,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00015] output:id:000010,sig:06,src:000068,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00016] output:id:000014,sig:06,src:000049,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00017] output:id:000018,sig:06,src:000006+000088,op:splice,rep:4.....: EXPLOITABLE [HeapError (10/22)]
[00018] output:id:000022,sig:06,src:000104,op:havoc,rep:2.....: EXPLOITABLE [HeapError (10/22)]
[00019] output:id:000003,sig:06,src:000039,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00020] output:id:000007,sig:11,src:000067,op:ext_A0,pos:12.....: EXPLOITABLE [DestAv (8/22)]
[00021] output:id:000011,sig:06,src:000083,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00022] output:id:000015,sig:06,src:000001,op:ext_A0,pos:12.....: EXPLOITABLE [HeapError (10/22)]
[00023] output:id:000019,sig:06,src:000147+000149,op:splice,rep:2.....: EXPLOITABLE [HeapError (10/22)]
***
[*] Saving sample classification info to database.
[!] Removed 15 duplicate samples from index. Will continue with 8 remaining samples.
[*] Generating final gdb+exploitable script '/root/collection/gdb_script' for 8 samples...
[*] Copying 8 samples into output directory...

```

## 五、代码覆盖率及其相关概念

代码覆盖率是模糊测试中一个极其重要的概念，**使用代码覆盖率可以评估和改进测试过程，执行到的代码越多，找到bug的可能性就越大**，毕竟，在覆盖的代码中并不能100%发现bug，在未覆盖的代码中却是100%找不到任何bug的，所以本节中就将详细介绍代码覆盖率的相关概念。

### 1. 代码覆盖率（Code Coverage）

代码覆盖率是一种度量代码的覆盖程度的方式，也就是指源代码中的某行代码是否已执行；对二进制程序，还可将此概念理解为汇编代码中的某条指令是否已执行。其计量方式很多，但无论是GCC的GCOV还是LLVM的SanitizerCoverage，都提供函数（function）、基本块（basic-block）、边界（edge）三种级别的覆盖率检测，更具体的细节可以参考LLVM的官方文档。

### 2. 基本块（Basic Block）

缩写为BB，指一组顺序执行的指令，BB中第一条指令被执行后，后续的指令也会被全部执行，每个BB中所有指令的执行次数是相同的，也就是说一个BB必须满足以下特征：

只有一个入口点，BB中的指令不是任何跳转指令的目标。

只有一个退出点，只有最后一条指令使执行流程转移到另一个BB

将上面的程序拖进IDA，可以看到同样被划分出了4个基本块：

```
W = 0;  
X = X + Y;  
if( X > Z ) {  
    Y = X;  
    X++;  
} else {  
    Y = Z;  
    Z++;  
}  
W = X + Z;
```



```
W = 0;  
X = X + Y;  
if( X > Z ) {
```

```
    Y = X;  
    X++;
```

```
    Y = Z;  
    Z++;
```

```
W = X + Z;
```

```

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     [rbp+var_C], edx
mov     edx, [rbp+var_4]
add     edx, [rbp+var_8]
mov     [rbp+var_4], edx
mov     edx, [rbp+var_4]
cmp     edx, [rbp+var_C]
jle     loc_100000F46

```

```

mov     eax, [rbp+var_C]
mov     [rbp+var_8], eax
mov     eax, [rbp+var_4]
add     eax, 1
mov     [rbp+var_4], eax
jmp     loc_100000F55

```

```

loc_100000F46:
mov     eax, [rbp+var_C]
mov     [rbp+var_8], eax
mov     eax, [rbp+var_C]
add     eax, 1
mov     [rbp+var_C], eax

```

```

loc_100000F55:
mov     eax, [rbp+var_4]
add     eax, [rbp+var_C]
pop     rbp
retn
_func endp

```

REEBUF

### 3. 边 (edge)

AFL的技术白皮书中提到fuzzer通过插桩代码捕获边 (edge) 覆盖率。那么什么是edge呢？我们可以将程序看成一个控制流图 (CFG)，图的每个节点表示一个基本块，而edge就被用来表示在基本块之间的跳转。知道了每个基本块和跳转的执行次数，就可以知道程序中的每个语句和分支的执行次数，从而获得比记录BB更细粒度的覆盖率信息。



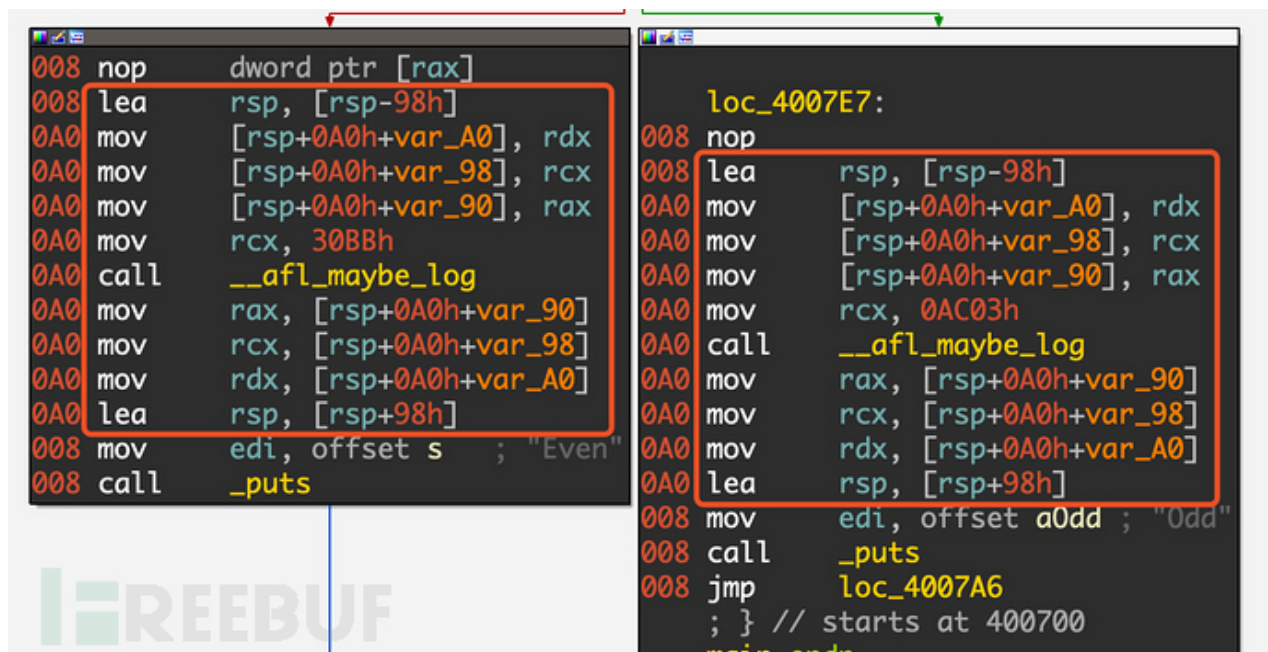
#### 4. 元组 (tuple)

具体到AFL的实现中，使用二元组(branch\_src, branch\_dst)来记录**当前基本块 + 前一基本块**的信息，从而获取目标的执行流程和代码覆盖情况，伪代码如下：

```
cur_location = <COMPILE_TIME_RANDOM>;//用一个随机数标记当前基本块
shared_mem[cur_location ^ prev_location]++;//将当前块和前一块异或保存到shared_mem[]
prev_location = cur_location >> 1;//cur_location右移1位区分从当前块到当前块的转跳
```

实际插入的汇编代码，如下图所示，首先保存各种寄存器的值并设置ecx/rcx，然后调用 `__afl_maybe_log`，这个方法的内容相当复杂，这里就不展开讲了，但其主要功能就和上面的伪代码相似，用于记录覆盖率，放入一块共享内存中。





## 六、计算代码覆盖率

了解了代码覆盖率相关的概念后，接下来看看如何计算我们的测试用例对前面测试目标的代码覆盖率。

这里需要用到的工具之一是**GCOV**，它随gcc一起发布，所以不需要再单独安装，和afl-gcc插桩编译的原理一样，gcc编译时生成插桩的程序，用于在执行时生成代码覆盖率信息。

另外一个工具是**LCOV**，它是GCOV的图形前端，可以收集多个源文件的gcov数据，并创建包含使用覆盖率信息注释的源代码HTML页面。

最后一个工具是afl-cov，可以快速帮助我们调用前面两个工具处理来自afl-fuzz测试用例的代码覆盖率结果。在ubuntu中可以使用 `apt-get install afl-cov` 安装afl-cov，但这个版本似乎不支持分支覆盖率统计，所以还是从Github下载最新版本为好，下载完无需安装直接运行目录中的Python脚本即可使用：

```
$ apt-get install lcov
$ git clone https:
$ ./afl-cov/afl-cov -V
afl-cov-0.6.2
```

还是以Fuzz libtiff为例，计算Fuzzing过程的代码覆盖率流程如下：

第一步，使用gcov重新编译源码，在CFLAGS中添加 `"-fprofile-arcs"` 和 `"-ftest-coverage"` 选项，可以在 `--prefix` 中重新指定一个新的目录以免覆盖之前afl插桩的二进制文件。

```
$ make clean
$ ./configure --prefix=/root/tiff-4.0.10/build-cov CC="gcc" CXX="g++" CFLAGS="-fprofile-arcs -ftest-coverage" --disable-shared
$ make
$ make install
```

第二步，执行afl-cov。其中-d选项指定afl-fuzz输出目录；—live用于处理一个还在实时更新的AFL目录，当afl-fuzz停止时，afl-cov将退出；—enable-branch-coverage用于开启边缘覆盖率（分支覆盖率）统计；-c用于指定源码目录；最后一个-e选项用来设置要执行的程序和参数，其中的AFL\_FILE和afl中的“@@”类似，会被替换为测试用例，LD\_LIBRARY\_PATH则用来指定程序的库文件。

```
$ cd ~/tiff-4.0.10
$ afl-cov -d ~/syncdir --live --enable-branch-coverage -c . -e "cat AFL_FILE |
LD_LIBRARY_PATH=./build-cov/lib ./build-cov/bin/tiff2pdf AFL_FILE"
```

成功执行的结果如下所示：

```
[+] AFL test case: id:000176,src:000001,op:havoc,rep:4 (176 / 4754), cycle: 0
lines.....: 12.0% (3572 of 29864 lines)
functions...: 18.4% (214 of 1166 functions)
branches...: 14.3% (1785 of 12501 branches)
[+] AFL test case: id:000177,src:000001,op:havoc,rep:2,+cov (177 / 4754), cycle: 0
lines.....: 12.0% (3576 of 29864 lines)
functions...: 18.4% (214 of 1166 functions)
branches...: 14.3% (1788 of 12501 branches)

Coverage diff id:000176,src:000001,op:havoc,rep:4 id:000177,src:000001,op:havoc,rep:2,+cov
diff id:000176,src:000001,op:havoc,rep:4 -> id:000177,src:000001,op:havoc,rep:2,+cov
Src file: /root/project-gcov/tiff-4.0.10/libtiff/tif_dirread.c
New 'line' coverage: 349
New 'line' coverage: 350
New 'line' coverage: 351
New 'line' coverage: 356
```

我们可以通过—live选择，在fuzzer运行的同时计算覆盖率，也可以在测试结束以后再进行计算，最后会得到一个像下面这样的html文件。它既提供了概述页面，显示各个目录的覆盖率；也可以在点击进入某个目录查看某个具体文件的覆盖率。

点击进入每个文件，还有更详细的数据。每行代码前的数字代表这行代码被执行的次数，没有执行过的代码会被红色标注出来。

## LCOV - code coverage report

Current view: top level

Test: trace.lcov\_info\_final

Date: 2018-12-13 22:37:35

	Hit	Total	Coverage
Lines:	8445	29864	28.3 %
Functions:	403	1166	34.6 %
Branches:	5096	12501	40.8 %

Directory	Line Coverage	Functions	Branches
contrib/addtiff	0.0 % 0 / 479	0.0 % 0 / 13	- 0 / 0
contrib/dha	0.0 % 0 / 379	0.0 % 0 / 7	- 0 / 0
contrib/iptcutil	0.0 % 0 / 288	0.0 % 0 / 8	- 0 / 0
libtiff	40.1 % 6309 / 15718	42.6 % 339 / 795	38.5 % 4228 / 10995
port	0.0 % 0 / 2	0.0 % 0 / 1	- 0 / 0
tools	16.4 % 2136 / 12998	18.7 % 64 / 342	57.6 % 868 / 1506

Filename	Line Coverage	Functions	Branches
mkqstates.c	0.0 % 0 / 77	0.0 % 0 / 3	- 0 / 0
tif_aux.c	38.8 % 66 / 170	72.7 % 8 / 11	27.5 % 44 / 160
tif_close.c	78.4 % 29 / 37	100.0 % 2 / 2	75.0 % 21 / 28
tif_codec.c	33.3 % 7 / 21	33.3 % 1 / 3	50.0 % 4 / 8
tif_color.c	38.5 % 42 / 109	50.0 % 3 / 6	35.7 % 25 / 70
tif_compress.c	31.7 % 38 / 120	36.8 % 7 / 19	21.9 % 7 / 32
tif_dir.c	59.3 % 544 / 918	64.7 % 22 / 34	46.8 % 401 / 857
tif_dirinfo.c	44.6 % 158 / 354	40.0 % 10 / 25	37.9 % 86 / 227
tif_dirread.c	74.5 % 2038 / 2734	77.7 % 73 / 94	67.7 % 1361 / 2009
tif_dirwrite.c	19.3 % 227 / 1179	20.0 % 12 / 60	17.0 % 149 / 876
tif_dumpmode.c	53.5 % 23 / 43	60.0 % 3 / 5	18.8 % 3 / 16
tif_error.c	50.0 % 14 / 28	50.0 % 2 / 4	50.0 % 4 / 8
tif_extension.c	0.0 % 0 / 32	0.0 % 0 / 5	- 0 / 0
tif_fax3.c	75.4 % 515 / 683	85.3 % 29 / 34	74.4 % 834 / 1121
tif_flush.c	22.5 % 9 / 40	100.0 % 2 / 2	13.2 % 5 / 38
tif_getimage.c	14.4 % 217 / 1503	17.5 % 11 / 63	10.2 % 101 / 986
tif_jbig.c	0.0 % 0 / 74	0.0 % 0 / 7	0.0 % 0 / 26
tif_jpeg.c	29.1 % 282 / 970	35.7 % 25 / 70	18.3 % 109 / 595

```

815 : 0 : (long) 1, (long) (already_read + to_read));
816 : 0 : return TIFFReadDirEntryErrAlloc;
817 : : }
818 : 0 : *pdest = new_dest;
819 : :
820 : 0 : bytes_read = TIFFReadFile(tif,
821 : : (char*)pdest + already_read, to_read);
822 : 0 : already_read += bytes_read;
823 [ # # ]: 0 : if (bytes_read != to_read) {
824 : 0 : return TIFFReadDirEntryErrIo;
825 : : }
826 : : }
827 : 0 : return TIFFReadDirEntryErrOk;
828 : : }
829 : :
830 : 1534370 : static enum TIFFReadDirEntryErr TIFFReadDirEntryArrayWithLimit(
831 : : TIFF* tif, TIFFDirEntry* direntry, uint32* count, uint32 desttypesize,
832 : : void** value, uint64 maxcount)
833 : : {
834 : : int typesize;
835 : : uint32 datasize;
836 : : void* data;
837 : : uint64 target_count64;
838 : 1534370 : typesize=TIFFDataWidth(direntry->tdir_type);
839 : :
840 : 1534370 : target_count64 = (direntry->tdir_count > maxcount) ?
841 : 1534370 : maxcount : direntry->tdir_count;
842 : :
843 [ + + ][ - + ]: 1534370 : if ((target_count64==0)|| (typesize==0))
844 : : {
845 : 53331 : *value=0;
846 : 53331 : return(TIFFReadDirEntryErrOk);

```

参考资料

[1] Fuzzing with AFL

[2] INTRO TO AMERICAN FUZZY LOP – FUZZING WITH ASAN AND BEYOND

[3] Clang 9 documentation – SanitizerCoverage

[4] honggfuzz漏洞挖掘技术深究系列

[5]How Much Test Coverage Is Enough For Your Testing Strategy?

**\*本文作者：alphalab，转载请注明来自FreeBuf.COM**