Security in Software Applications
Project 3

Edoardo Ottavianelli
1756005

2021/2022

1 Introduction and Setup

Fuzz testing (fuzzing) is a quality assurance technique used to discover coding errors and security loopholes in software, operating systems or networks. It involves inputting massive amounts of random data, called fuzz, to the test subject in an attempt to make it crash. If a vulnerability is found, a software tool called a fuzzer can be used to identify potential causes. Fuzz testing was originally developed by Barton Miller at the University of Wisconsin in 1989. (Wikipedia)

American Fuzzy Lop is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow. Setting up AFL was really straigthforward. I visited the page http://lcamtuf.coredump.cx/afl, then clicked on the button 'Download'. Downloaded the latest version (afl-2.52b). Then in order to use the tool I executed:

Listing 1: AFL setup

```
$> tar xf afl-latest.tgz
$> cd afl-2.52b/
$> make
$> CC=afl-gcc CXX=afl-g++ ./configure --disable-shared
$> ./afl-fuzz
```

Then I choose ImageMagick 6.7.7-10 (released in 2012) to be tested (https://sourceforge.net/projects/imagemagick/files/old-sources/6.x/6.7/ImageMagick-6.7.7-10.tar.gz/download) and these are the steps performed to install it:

Listing 2: ImageMagick setup

```
$> cd ImageMagick-6.7.7-10/
$> CC=afl-gcc CXX=afl-g++ ./configure && make && make install
$> convert
```

Running afl-fuzz the first time I encountered this error, but it was very easy to fix (following the suggestion in the output):

Listing 3: afl-fuzz error

```
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 12 CPU cores and 2 runnable tasks (utilization: 17%).
[+] Try parallel jobs - see docs/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...

[-] Hmm, your system is configured to send core dump notifications to an
    external utility. This will cause issues: there will be an extended delay
    between stumbling upon a crash and having this information relayed to the
    fuzzer via the standard waitpid() API.

    To avoid having crashes misinterpreted as timeouts, please log in as root
    and temporarily modify /proc/sys/kernel/core_pattern, like so:

    echo core >/proc/sys/kernel/core_pattern

[-] PROGRAM ABORT : Pipe at the beginning of 'core_pattern'
         Location : check_crash_handling(), afl-fuzz.c:7275
```

This helped me also with the parallel fuzzing, looking at the output then I read also the document docs/parallel_fuzzing.txt.

2 Fuzzing

To test the convert functionality, I've used these commands:

Listing 4: afl-fuzz commands

```
$> ./afl-fuzz -M f1 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f2 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f3 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f4 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f5 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f6 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f7 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f8 -m none -i input -o output convert @@ /dev/null
$> ./afl-fuzz -S f9 -m none -i input -o output convert @@ /dev/null
```

The machine used to test was my personal computer, since it has 12 cores, I've used a main fuzzer on a core and then 8 secondary fuzzers to better use the resources. The options used were:

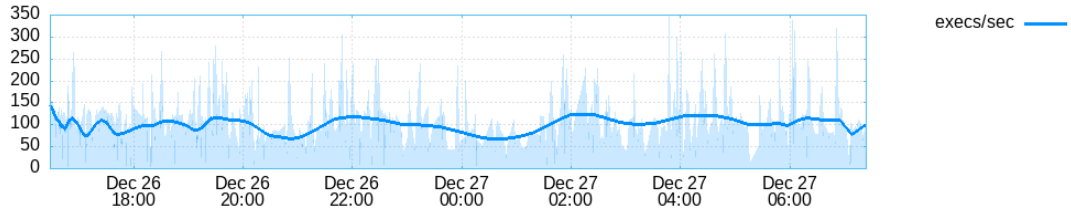
- '-M f1' means main fuzzer called f1.
- '-S fx' means secondary fuzzer called fx
- '-m none' unset the memory limit
- '-i input' set the input directory
- '-o output' set the output directory
- the remaining part tells afl which command should be tested (convert) and the symbols @@ means where put the input to be tested, finally /dev/null means I'm not interested in saving the converted inputs

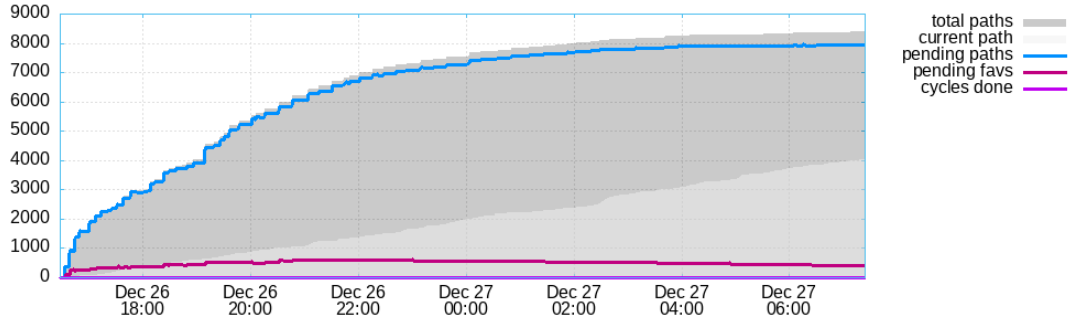
To test the application I've used the file kitty.png provided by afl in the testcases directory. The fuzzers ran for 14 hours (5 days and 13 hours cumulatively) and the charts in the following pages were produced using afl-plot (./afl-plot output/fx /Desktop/afl-plot1-fx). Since ImageMagick creates in the /tmp folder really heavy trash files, and since (due to the fuzzing testing nature) the program was executed many times, the files could fill the computer memory really fast; so I've used the command

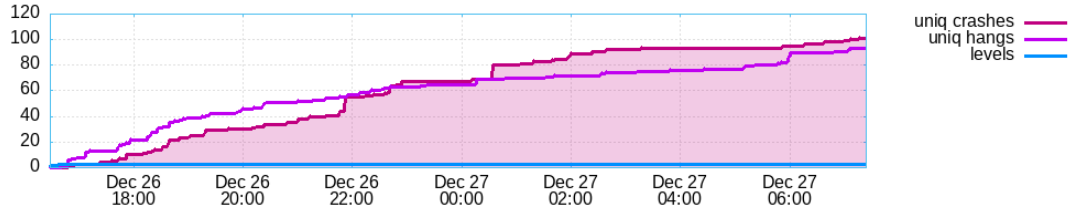
```
watch -n 3 find /tmp -maxdepth 1 -name 'magick-*' -delete >/dev/null 2>&1
```

I thought 10 seconds could be also okay, but actually the program created gigabytes and gigabytes of data in seconds, so I've switched to every 3 seconds.

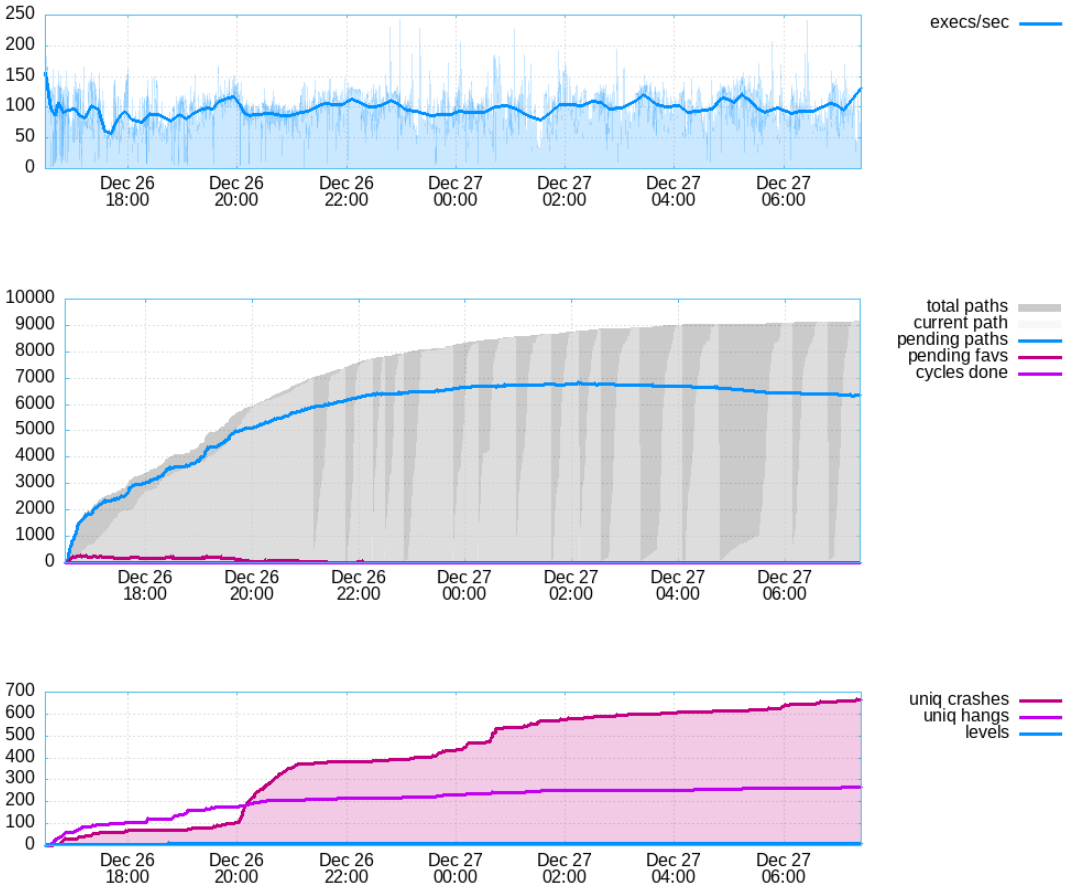
2.1 F1 (main fuzzer) results



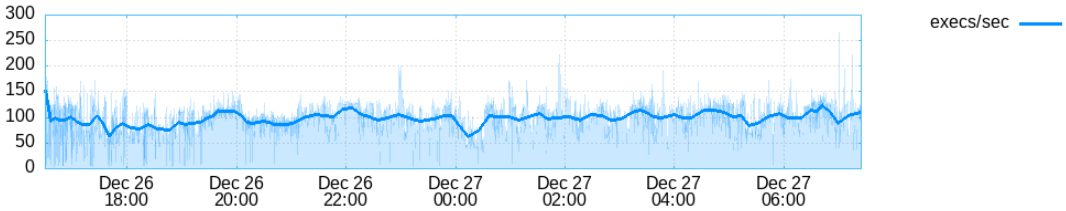


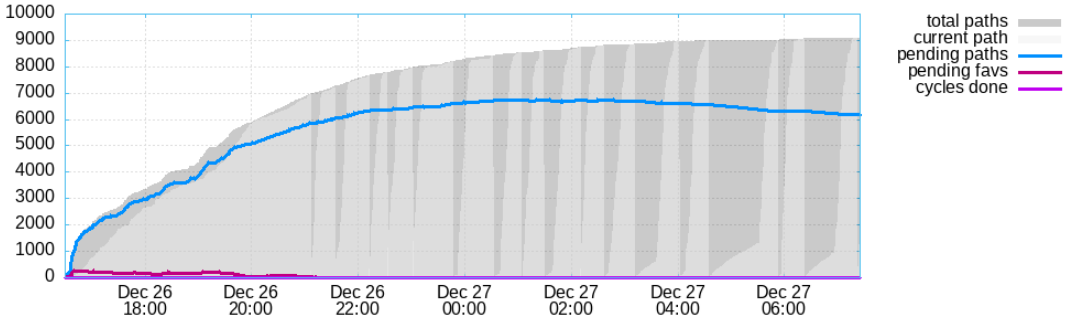


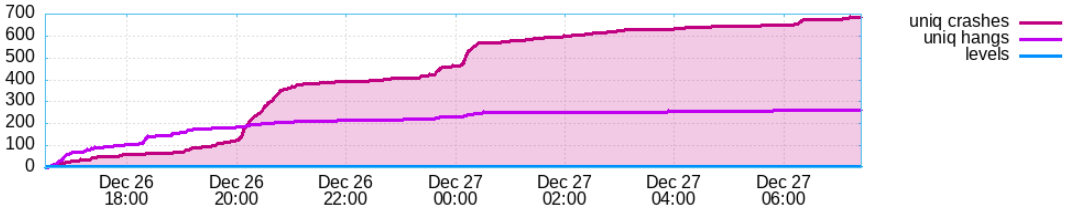
2.2 F2 (secondary fuzzer) results



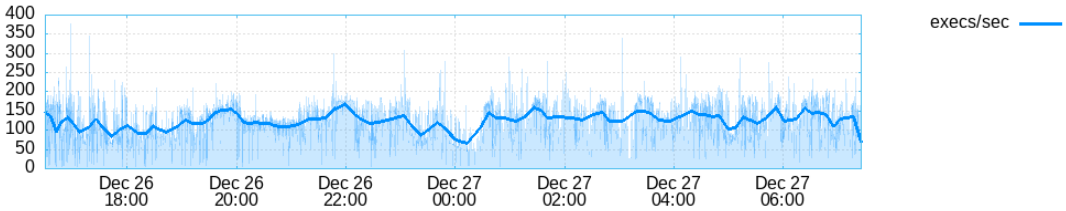
2.3 F3 (secondary fuzzer) results

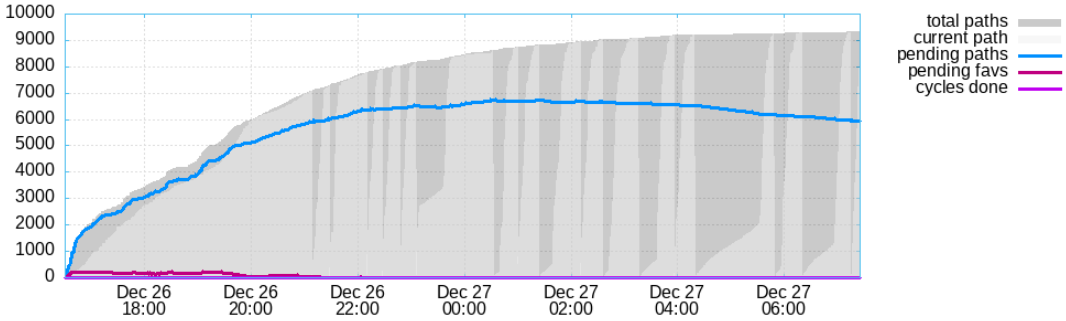


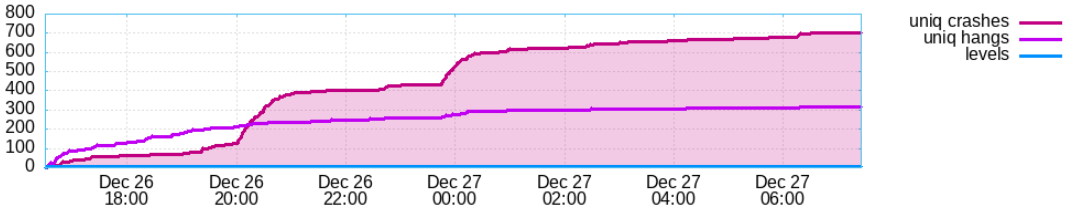




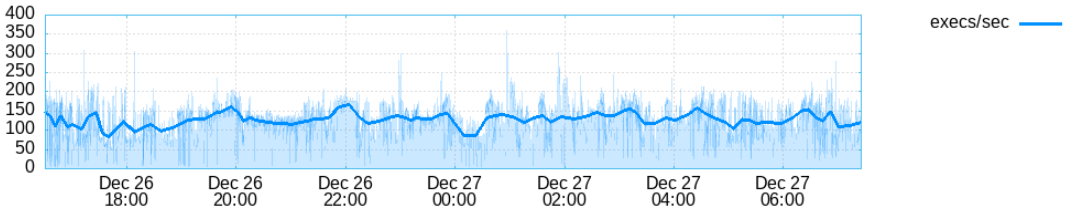
2.4 F4 (secondary fuzzer) results

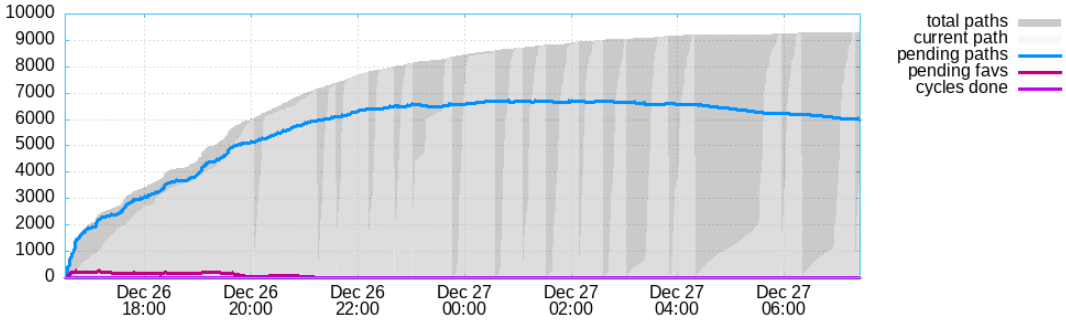


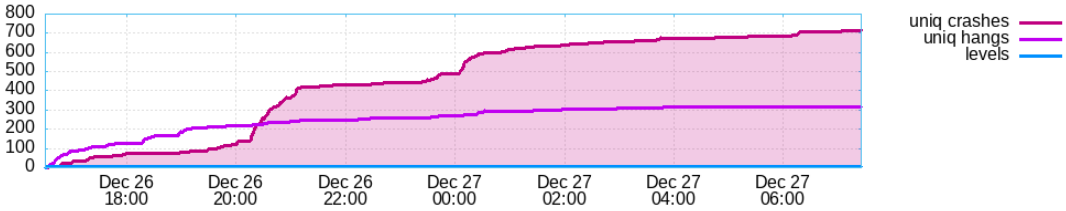




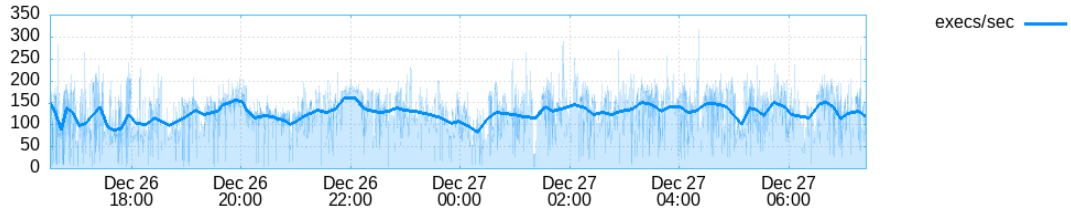
2.5 F5 (secondary fuzzer) results

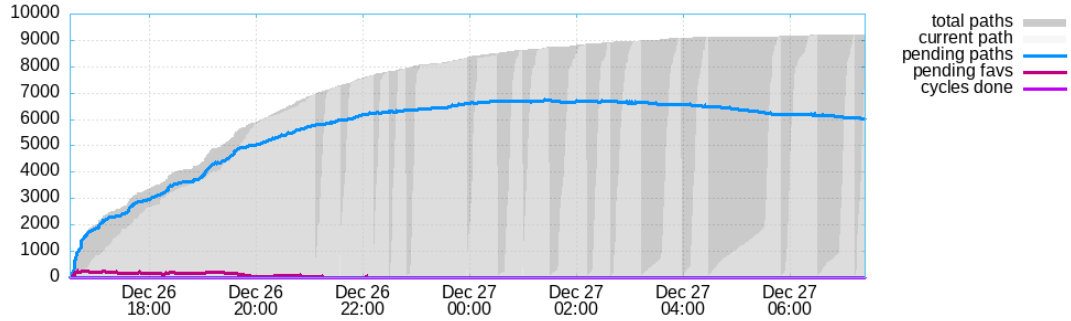


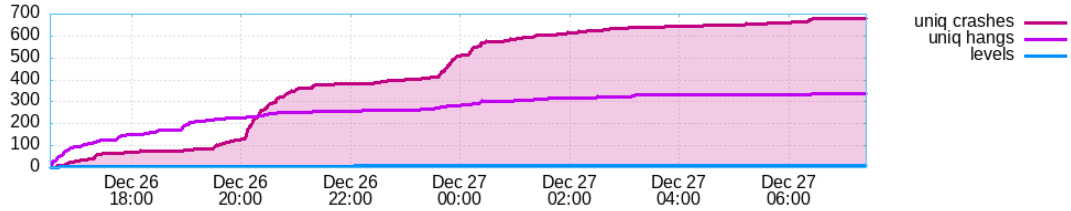




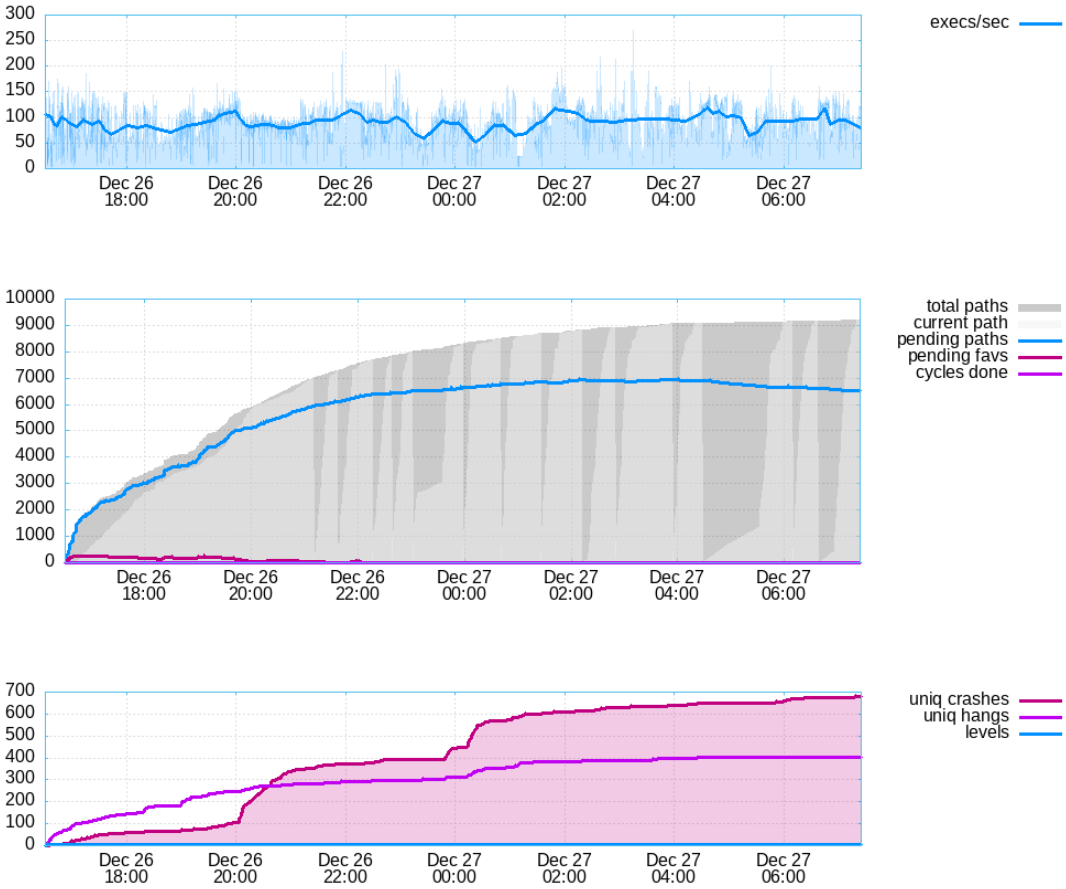
2.6 F6 (secondary fuzzer) results



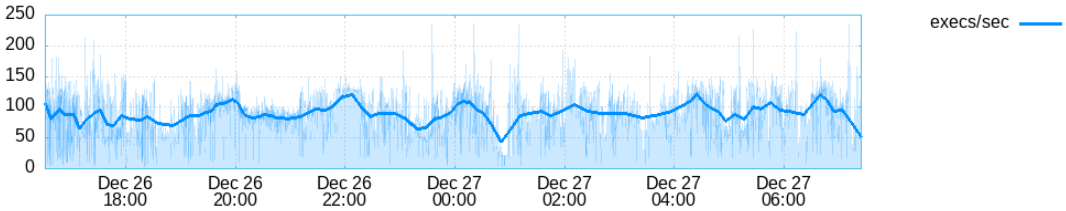


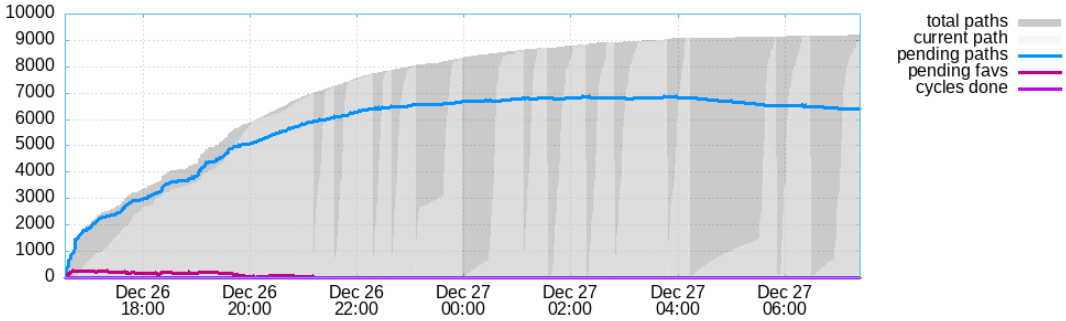


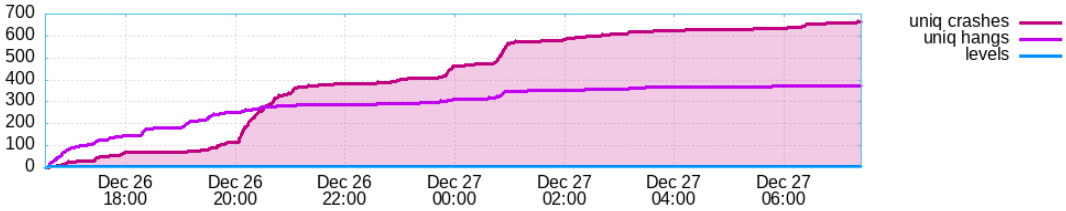
2.7 F7 (secondary fuzzer) results



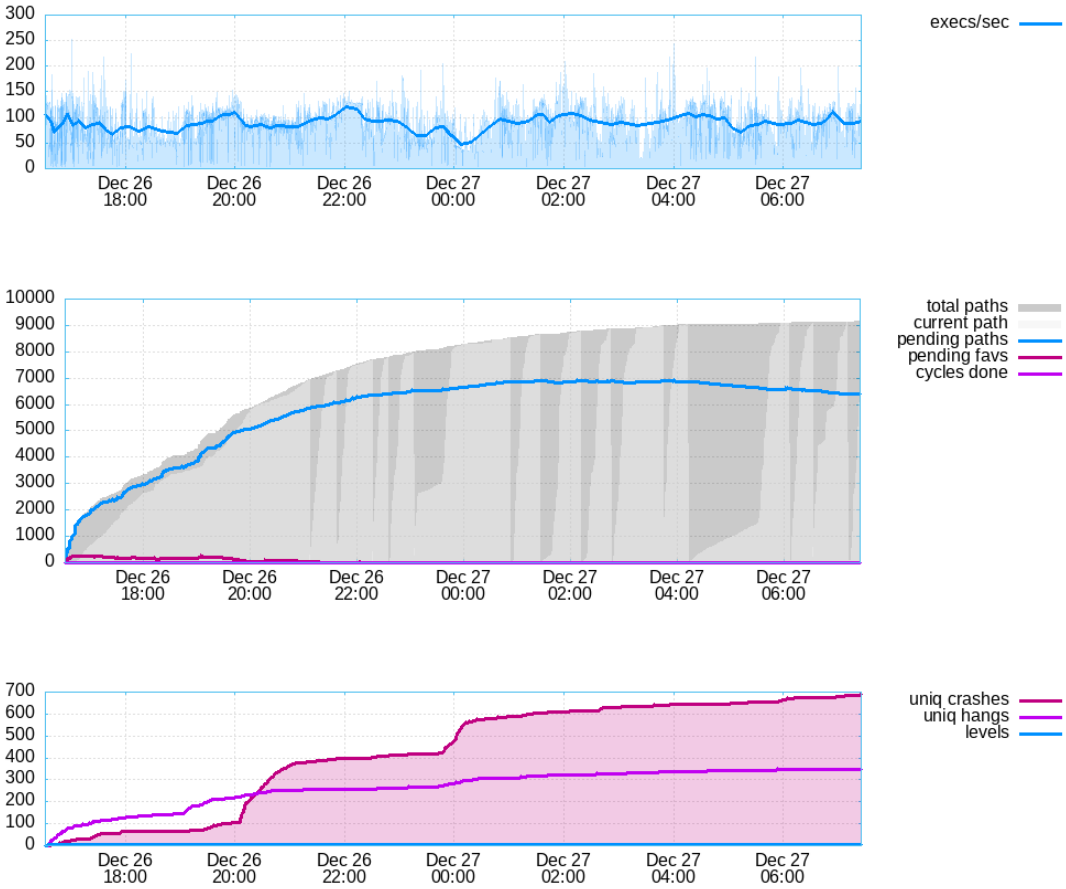
2.8 F8 (secondary fuzzer) results







2.9 F9 (secondary fuzzer) results



3 Conclusions

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

Individual fuzzers
==================

>>> f1 (0 days, 14 hrs) <<<

  cycle 1, lifetime speed 107 execs/sec, path 4013/8389 (47%)
  pending 417/7933, coverage 18.46%, crash count 101 (!)

>>> f2 (0 days, 14 hrs) <<<

  cycle 17, lifetime speed 93 execs/sec, path 9139/9146 (99%)
  pending 2/6342, coverage 18.47%, crash count 663 (!)

>>> f3 (0 days, 14 hrs) <<<

  cycle 17, lifetime speed 96 execs/sec, path 9091/9093 (99%)
  pending 2/6175, coverage 18.47%, crash count 686 (!)

>>> f4 (0 days, 14 hrs) <<<

  cycle 18, lifetime speed 119 execs/sec, path 4480/9319 (48%)
  pending 4/5943, coverage 18.47%, crash count 705 (!)

>>> f5 (0 days, 14 hrs) <<<

  cycle 19, lifetime speed 123 execs/sec, path 1186/9308 (12%)
  pending 1/6013, coverage 18.47%, crash count 715 (!)

>>> f6 (0 days, 14 hrs) <<<

  cycle 18, lifetime speed 122 execs/sec, path 3744/9218 (40%)
  pending 4/6043, coverage 18.47%, crash count 681 (!)

>>> f7 (0 days, 14 hrs) <<<

  cycle 15, lifetime speed 86 execs/sec, path 8991/9197 (97%)
  pending 5/6517, coverage 18.47%, crash count 678 (!)

>>> f8 (0 days, 14 hrs) <<<

  cycle 15, lifetime speed 86 execs/sec, path 8749/9189 (95%)
```

```
  pending 4/6383, coverage 18.47%, crash count 663 (!)

>>> f9 (0 days, 14 hrs) <<<

  cycle 15, lifetime speed 85 execs/sec, path 3409/9144 (37%)
  pending 4/6406, coverage 18.47%, crash count 687 (!)

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

      Fuzzers alive : 9
     Total run time : 5 days, 13 hours
        Total execs : 49 million
    Cumulative speed : 917 execs/sec
       Pending paths : 443 faves, 57755 total
  Pending per fuzzer : 49 faves, 6417 total (on average)
       Crashes found : 5579 locally unique
```

In order to understand the causes of the crashes I put the inputs that afl found causing those crashes in a single file. The inputs are stored and divided by different fuzzers, since I spawned nine different fuzzers, I have nine different folders that I put together using this command:

```
find output/*/crashes* -type f > crashes_input.txt
```

Now, it's trivial to reproduce those crashes to understand better the causes of them. To achieve this I've used this command:

```
for input in $(cat crashes_input.txt); do echo ""; echo "INPUT: $input";
echo -ne "OUTPUT: "; convert $input /dev/null; echo ""; done >
crashes_causes.txt
```

These are the causes producing crashes:

Occurrences	Reason
3129	Segmentation fault
762	malloc()
206	corrupted double-linked list
124	free()
96	double free or corruption
74	Invalid pointer
51	corrupted size vs. prev_size
About 200	Others

This of course means about 9 hundreds crashes were not considered due to their nature (not exploitable), still we have a lot of crashes to analyze.

Now we have the main reasons of the crashes, but we should go deep in understanding which functions are not behaving in the correct way. To achieve

this I've used gdb against each input file. The following is the command used to run gdb to inspect in batch mode all the inputs:

```
for input in $(cat crashes_input.txt); do echo ""; echo "GDB: $input";
gdb -xe "r $input /dev/null" -batch convert; done > gdb_output.txt 2>&1
```

So now we have in this file an easily greppable specific reason for each crash. Using this command I put in files.txt where the crashes spawn up:

```
cat gdb_output.txt | grep "at" | awk -F' at ' '{print $2}'
gdb_output.txt | sed '/^$/d' | sort | uniq -c > files.txt
```

And inspecting the lines indicated in this file I was able to reconstruct the exact mapping of the functions that produce the crashes. This is a complete list:

Occurrences	File	Function
831	coders/pcx.c:550	ReadPCXImage
1	coders/pcx.c:555	ReadPCXImage
63	coders/pcx.c:565	ReadPCXImage
32	coders/pcx.c:583	ReadPCXImage
1	coders/rle.c:395	ReadRLEImage
444	coders/rle.c:427	ReadRLEImage
3	coders/rle.c:447	ReadRLEImage
270	coders/rle.c:532	ReadRLEImage
6	magick/blob.c:2920	ReadBlobByte
1	magick/cache.c:1520	DestroyPixelCacheNexus
5	magick/cache.c:3213	GetVirtualPixelsFromNexus
4	magick/cache.c:4537	QueueAuthenticPixels
3	magick/cache.c:5020	SetPixelCacheNexusPixels
1	magick/exception.c:573	GetLocaleExceptionMessage
1	magick/exception.c:633	InheritException
72	magick/hashmap.c:427	DestroyLinkedList
2	magick/hashmap.c:473	GetLastValueInLinkedList
1	magick/image.c:1311	DestroyImageInfo
6	magick/locale.c:1041	DestroyLocaleNode
3	magick/magic.c:1049	DestroyMagicElement
1	magick/magick.c:789	DestroyMagickNode
3	magick/matrix.c:280	GaussJordanElimination
3	magick/mime.c:328	GetMimeInfo
36	magick/montage.c:286	GetMontageGeometry
1	magick/quantize.c:1561	FloydSteinbergDither
1	magick/quantize.c:2464	PruneChild
1	magick/quantize.c:609	AssignImageColors
1976	magick/quantum.c:133	AcquireQuantumInfo
4	magick/semaphore.c:291	LockSemaphoreInfo
6	magick/splay-tree.c:1490	Splay
7	magick/splay-tree.c:1574	SplaySplayTree
14	magick/splay-tree.c:256	LinkSplayTreeNodes
2	magick/splay-tree.c:263	LinkSplayTreeNodes
8	magick/splay-tree.c:338	GetFirstSplayTreeNode
10	magick/splay-tree.c:692	DestroySplayTree
15	magick/splay-tree.c:693	DestroySplayTree
41	magick/splay-tree.c:707	DestroySplayTree
3	magick/splay-tree.c:719	DestroySplayTree
4	magick/splay-tree.c:841	GetNextValueInSplayTree

Some of the inputs were not considered because not so interesting, so they are not included in this table.

Finally, some of these crashes are well known vulnerabilities, we can also

search them in a CVE database (https://nvd.nist.gov/, the U.S. Government Vulnerability Database). These are some examples:

- **CVE-2014-9844**: (coders/rle.c) The ReadRLEImage function in coders/rle.c in ImageMagick 6.8.9.9 allows remote attackers to cause a denial of service (out-of-bounds read) via a crafted image file.
- **CVE-2016-10050**: (coders/rle.c) Heap-based buffer overflow in the ReadRLEImage function in coders/rle.c in ImageMagick 6.9.4-8 allows remote attackers to cause a denial of service (application crash) or have other unspecified impact via a crafted RLE file.
- **CVE-2016-8862**: (magick/quantum.c:133) The AcquireMagickMemory function in MagickCore/memory.c in ImageMagick before 7.0.3.3 allows remote attackers to have unspecified impact via a crafted image, which triggers a memory allocation failure.
- **CVE-2016-8683**: (coders/pcx.c) The ReadPCXImage function in coders/pcx.c in GraphicsMagick 1.3.25 allows remote attackers to have unspecified impact via a crafted image, which triggers a memory allocation failure and a "file truncation error for corrupt file."
- **CVE-2017-13658**: (magick/image.c) In ImageMagick before 6.9.9-3 and 7.x before 7.0.6-3, there is a missing NULL check in the ReadMATImage function in coders/mat.c, leading to a denial of service (assertion failure and application exit) in the DestroyImageInfo function in MagickCore/image.c.