SAMSUNG

# Fuzzing
## with afl
## (American Fuzzy Lop)

# sudo –u jakub.botwicz whoami

- Principal Security Engineer
  at Samsung R&D Institute in Warsaw, Poland
- Leads a team (one of many in Samsung)
  of security researchers / pentesters
- Performs security assessment of Samsung products
  and software components
- PhD and MSc at Warsaw University of Technology
  Dissertation subject: *Usage of hardware accelerated data classification algorithms in information security*
- 15+ years experience - previously worked as:
  - Developer and architect for vendor of encryption devices
  - Security advisor in Payment card company
  - Security consultant and manager at Big4 company
- Big fan of rock climbing and mountaineering

# sudo –u wojciech.rauner whoami

- Security Engineer @ Samsung R&D Institute Poland
- Area of research: IoT & mobile
- Background: full-stack developer
- Likes to talk about crypto and programming
- Plays CTF in Samsung R&D PL team
- PM me: wojciech.rauner@protonmail.com

# Fuzzing origin

**Fuzzing**

- providing invalid, unexpected or random data as inputs to a computer program

**Infinite monkey theorem**

- a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will eventually type out the entire works of Shakespeare

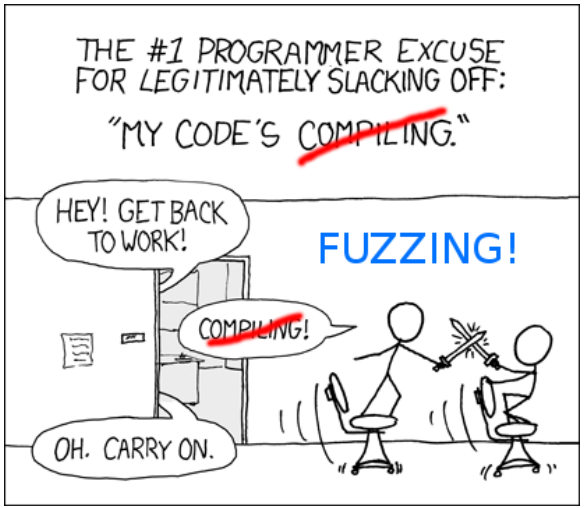
- Similarly, monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will eventually:
  - generate all possible input data
  - finding all bugs
  - exiting vi text editor ☺

Source: Early Office Museum Author: New York Zoological Society

# Fuzzing

**Fuzzing (fuzz testing):**

- automated testing technique
- involves providing invalid, unexpected or random data as inputs to a computer program




Source:Mahesh Paolini-Subramanya
„Fuzzing, and … Deep Learning?"
Original: xkcd „My code's compilimg"

- **Pros:**
  - Identifies issues that no human being could imagine
  - Fuzzing requires mainly CPU time for execution
  - Provides good coverage of tests with minimal manual effort (only if tested code is susceptible to fuzzing)
  - Can be repeated multiple times and conducted by every new version
- **Cons:**
  - Can give false sense of security (if not done properly and result are not analysed correctly)

# Fuzzers classification

- **Payload creation:**
  - Generation-based
  - Mutation-based
- **Payload delivery:**
  - File-based
  - Network-based
- **Approach:**
  - White box (using source code / program specification or docs)
  - Black box
- **Fuzzing techniques:**
  - Smart fuzzer
  - Dumb fuzzer

# Fuzzers classification – afl

- **Payload creation:**
  - Generation-based
  - **Mutation-based**
- **Payload delivery:**
  - **File-based**
  - Network-based
- **Approach:**
  - **White box** (using source code / program specification or docs)
  - Black box
- **Fuzzing techniques:**
  - Smart fuzzer
  - **Dumb fuzzer**

# American Fuzzy Lop

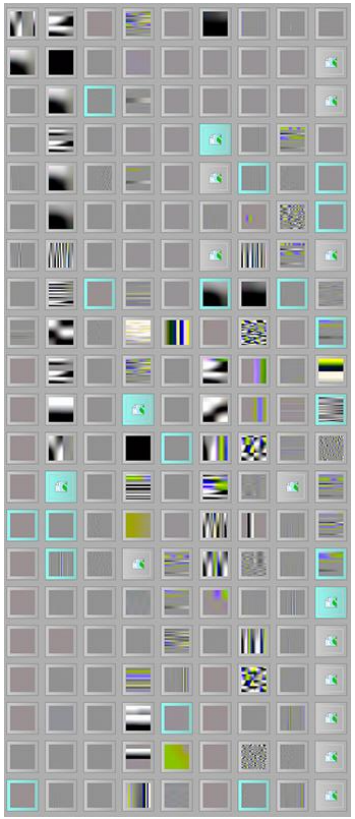
- **Dumb but very efficient fuzzer:**
  - Using files as inputs
  - Mutation-based
  - Coverage analysis
- Created by **Michał Zalewski (lcamtuf)**
  (at that time) Security Engineer in Google
- Registered list of CVEs found using AFL
  GitHub: mrash / afl-cve (Today: **332 CVE**)
- Helped our team identyfing 60+ issues last year  in different open source components

**American Fuzzy Lop**

Source: Wikipedia Author: Lithonius License: Public Domain

# Fuzzing – Pulling JPEGs out of thin air (2014)

- **Article on lcamtuf's blog:**
  - Fuzzing of library parsing JPEGs (djpeg) on 8 CPU core
  - Starting case was text file with only „hello" inside
  - Results:
    - After seconds of fuzzing afl generated JPEG header
    - After six hours first valid JPEG file was produced (blank greyscale image 3x784 pixels)
    - Generated large corpus of different JPEG files



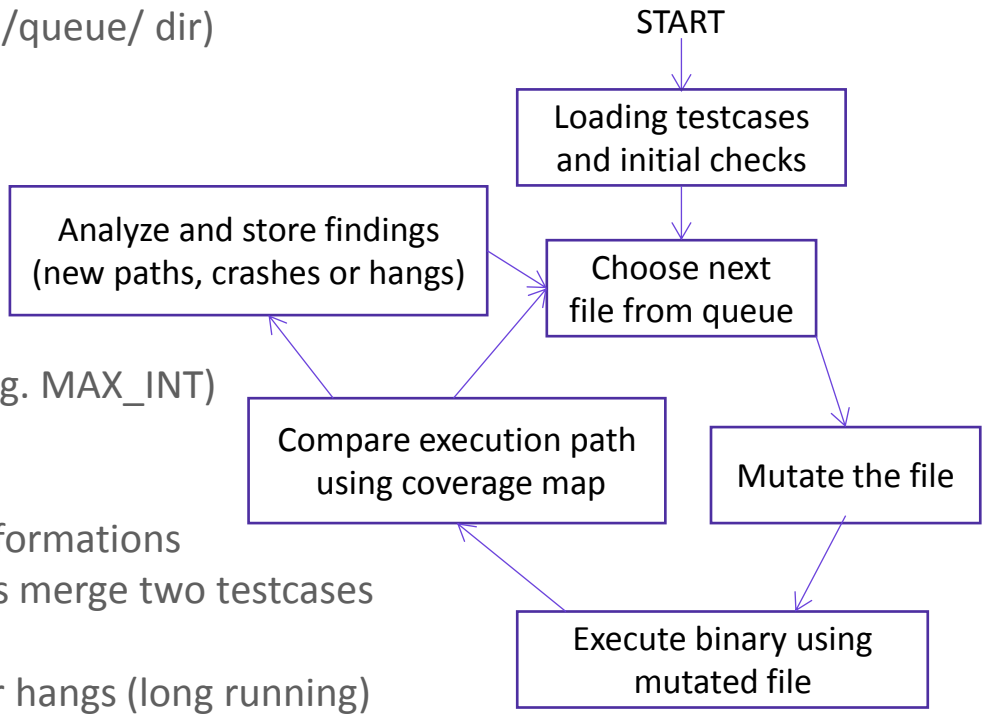
Source: lcamtuf blog, November 07, 2014
Pulling JPEGs out of thin air

# American Fuzzy Lop – basics

- Uses set of selected input files (available in /queue/ dir) each with unique execution „path”

- Performs transformation on favored cases:
  - deterministic:
    - bit-flip, byte-flip
    - arithmetics – change +/- 30
    - known ints – use well-known values (e.g. MAX_INT)
    - trim – truncates the file
  - non-deterministic (random):
    - havoc – stacked multiple random transformations
    - splice – same as above but first transf is merge two testcases

- Detects crashes (e.g. segmentation fault) or hangs (long running)
  - Hangs can be sometimes false-positives when CPU was busy
  - Crashes are almost always real

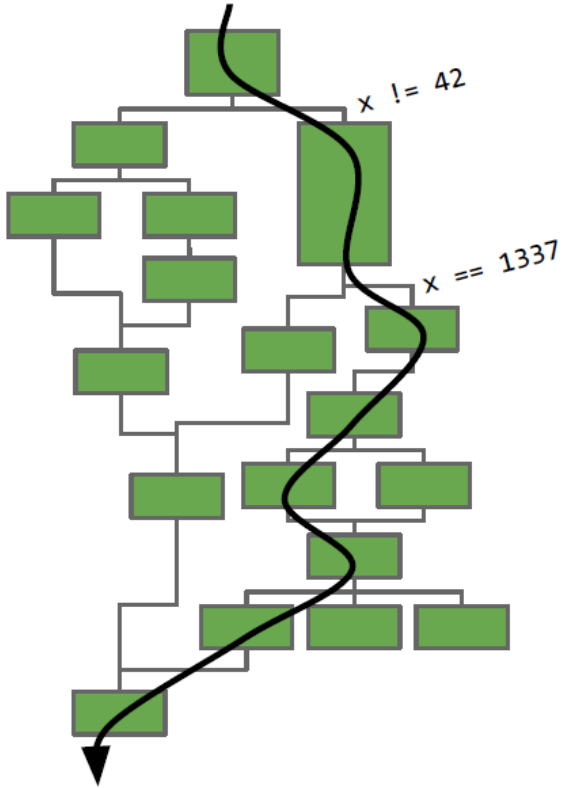


# American Fuzzy Lop – coverage analysis

- Unique paths have at least one unique edge (tuple):
  A->B->C->D->E
  A->B->D->C->E
- Non-unique paths
  Can be different, but use the same set (or sub-set) of edges
  A->B->A->D->E
  A->B->A->B->A->D->E
- Code injected at branch points:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

  where shared_mem is 64 kB region
- There was a presentation on Black Hat'18 describing method to prepare ELF binaries fooling afl by colliding paths
  *„AFL's Blindspot and How to Resist AFL Fuzzing for Arbitrary ELF Binaries"*
  This probably works only with defaults map settings!



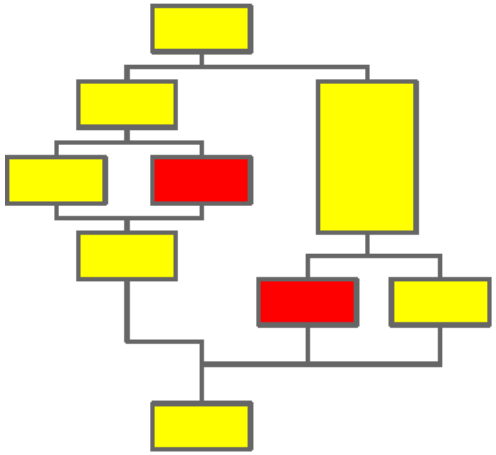

Source: Zardus „25 Years of Program Analysis" (Def Con 25)

# American Fuzzy Lop – modes of work

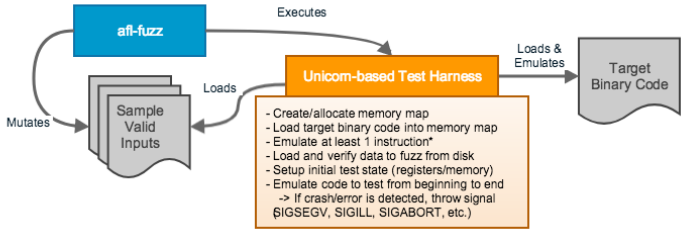
- **Instrumentation mode**
  - Requires source code to be recompiled with afl wrappers (afl-gcc/g++/clang)
  - Fastest method to execute (3-5 time faster than QEmu)
  - Need to recompile for native CPU (false positives and negatives)

- **Emulation mode with QEmu**
  - Can use binaries (where source is not available)
  - Can emulate different CPUs (e.g. ARM)
    - no false positives and negatives caused by CPU diff

- **Unicorn mode**
  - Can use binaries and different CPUs like QEmu
  - Allows to start from specific stored state of CPU



Source: Zardus „25 Years of Program Analysis" (Def Con 25)




Source:Nathan Voss
afl-unicorn: Fuzzing Arbitrary Binary Code
https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf

# American Fuzzy Lop – what is smart in it?

- *„The tool can be thought of as a **collection of hacks**, that have been tested **in practice**, found to be **surprisingly effective**, and have been implemented in **the simplest** and **most robust** way”*

- **Coverage analysis**
  - Construction of coverage map allows fast execution and comparison of paths
- **Favorite paths (new behaviors or frequent generators)**
  - They are preferred in generation of new paths
- **Generating auto-dictionaries**
  - Based on fuzzing frequent words are gathered and used as dictionaries
- **Culling the corpus**
  - Reevaluating cases in the input queue
- **Deduplicating crashes / hangs**
  - Unique crash/hang path contains unique edge or does not contain edge available in all previous crashes / hangs

# American Fuzzy Lop – WARNING before you start

- **Do not use sensitive production systems for fuzzing**
  - Fuzzing can increase rate of hardware or software issues and can make other applications to behave unexpectedly
- **Understand all functions of fuzzed program**
  - If your program creates files (output or temporary), it can fill up whole disc (by space or number of files) or make a mess in your /home/ directory
  - If your program tries to make DNS queries or network connections based on fuzzed inputs, it can mess local traffic
- **Be aware of disc wearing (especially for SSD drives)**
  - Use ramdisk to buffer I/O operations to physical drive

# American Fuzzy Lop – how to start fuzzing (1/2)

1. **Preparing source code and wrapper**

   - **Easiest:** small CLI applications taking input from file
     - Just fuzz ☺

   - **More difficult:** network servers
     - Solutions: change network input to file input or use afl fork that supports network sockets

   - **Very difficult:** operating systems or network stacks
     - Solution: rip out selected modules (e.g. parsing)

2. **Preparing testcases**
   - Use samples from unit tests (if available)
   - Prepare 1-3 valid samples
   - Start with „hello” file ☺

# American Fuzzy Lop – how to start fuzzing (2/2)

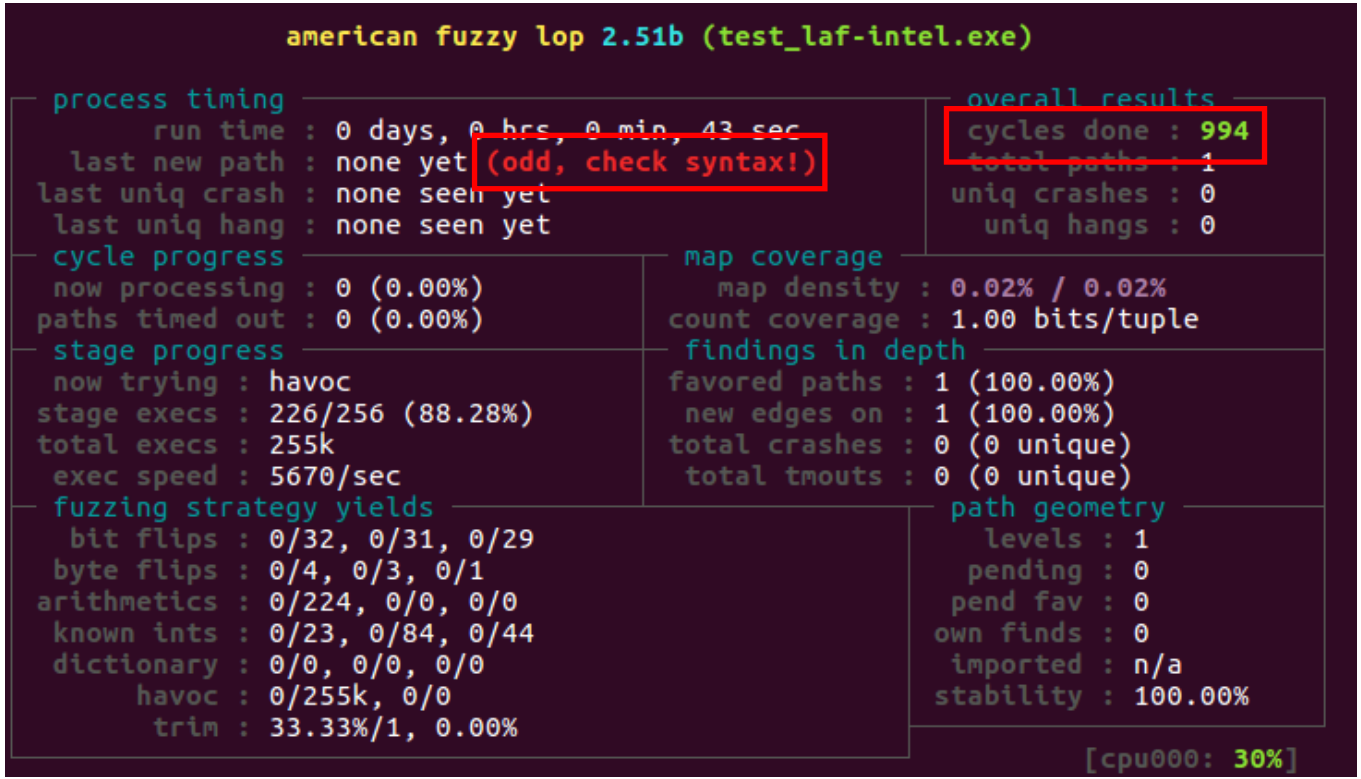
3. **Preparing binary (instrumentation)**
   - Compile using afl-gcc/g++/clang

4. **Starting new fuzzing process**
   - afl-fuzz –i testcase_dir –o findings_dir ./fuzzed_binary @@
     (take the input name from first param)

5. **Continue fuzzing process**
   - After stopping of afl-fuzz (e.g. restart)
   - afl-fuzz –i - …

   (afl will prevent overwriting results of 5+ minutes of fuzzing)

# American Fuzzy Lop – fuzzing in progress



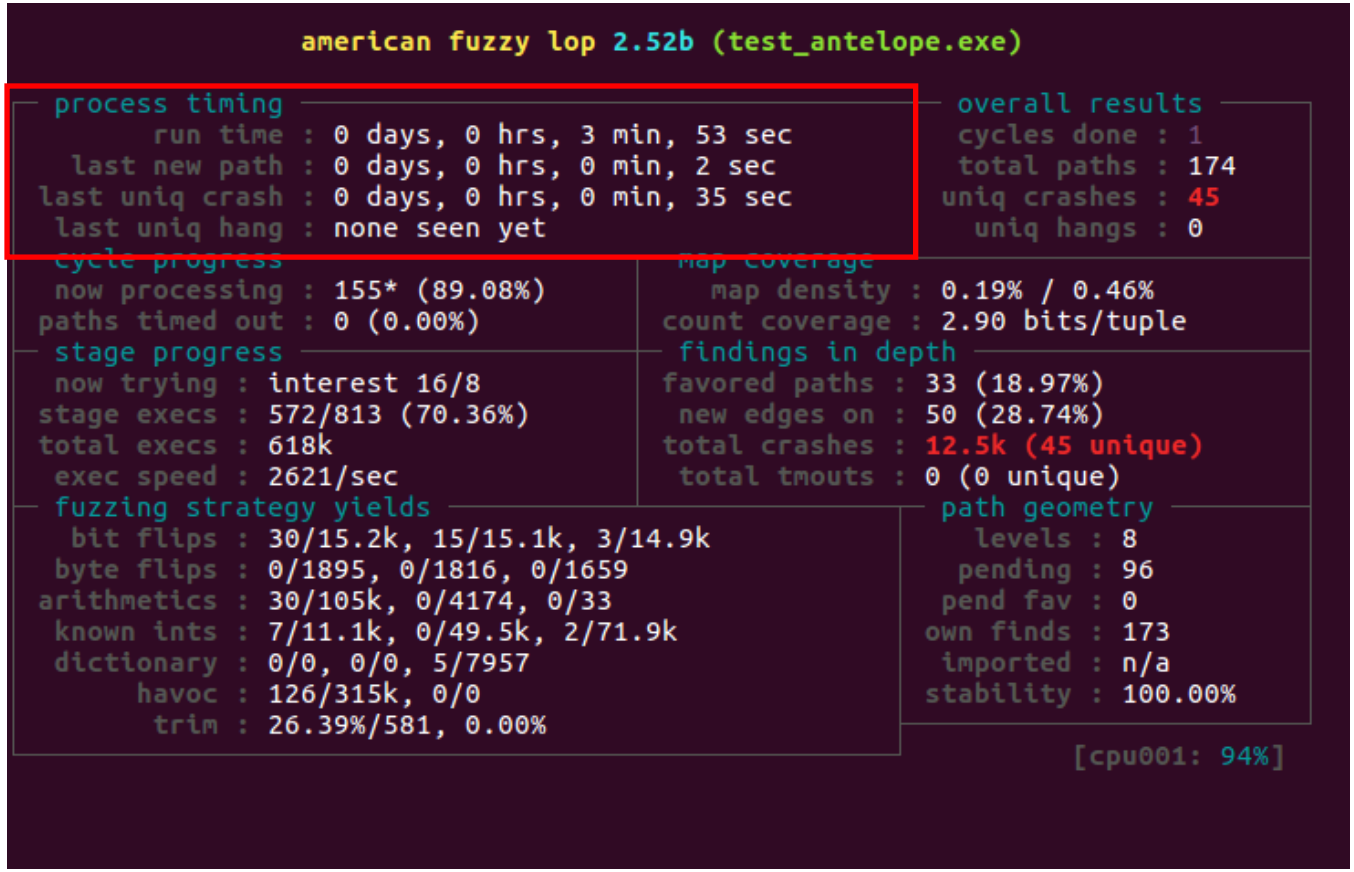

**Color code:**

- **RED** – you should do something about it, usually something strange (no progress or very slow) or crash / hang to be analyzed

**Cycles counter:**

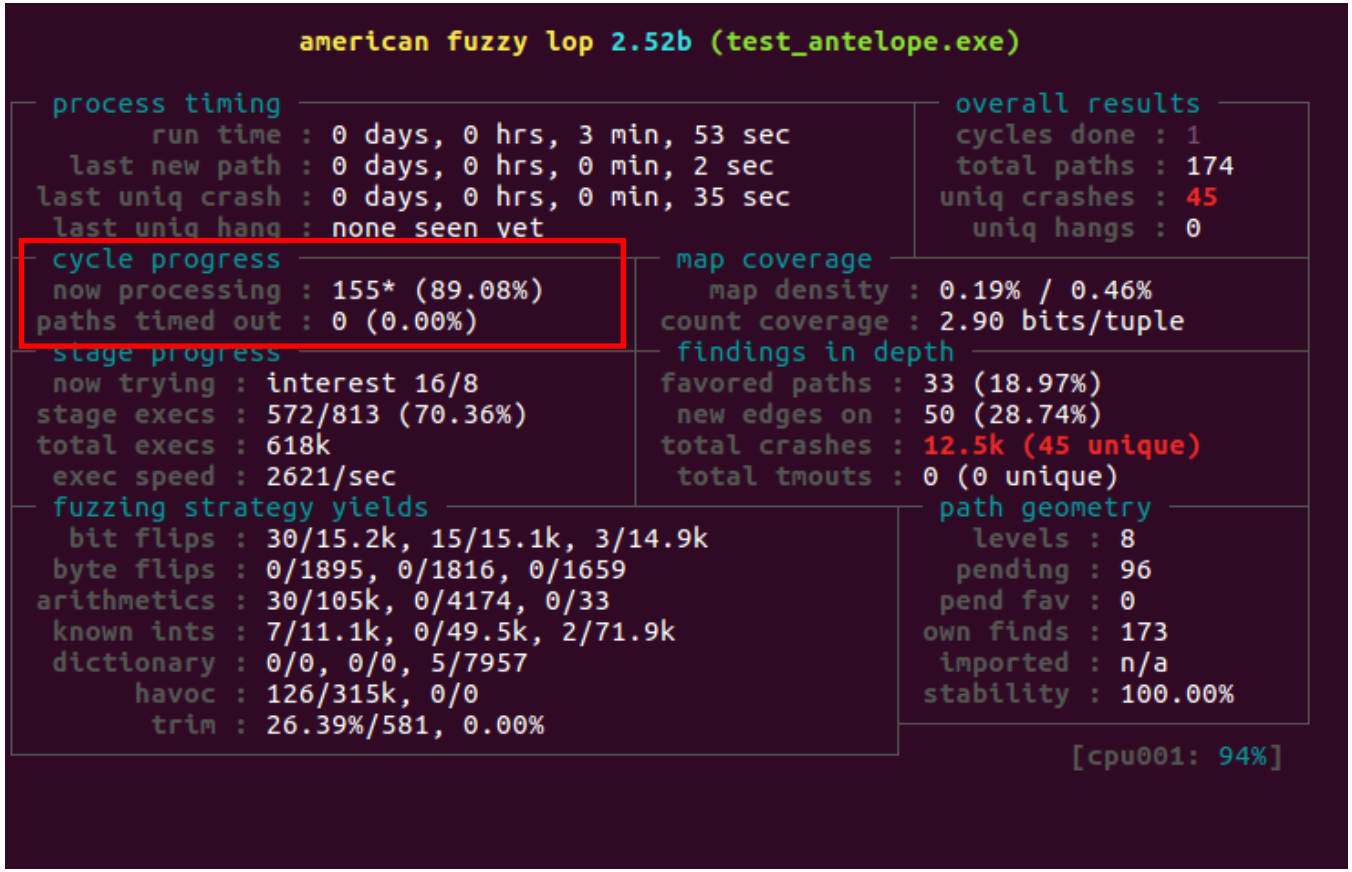
- **MAGENTA** – fuzzing has just started
- **YELLOW** – there are new finds during last cycles
- **BLUE** - there are no new finds during last cycles
- **GREEN** - there are no new finds during last cycles and large number of cycles was performed

# American Fuzzy Lop – fuzzing in progress



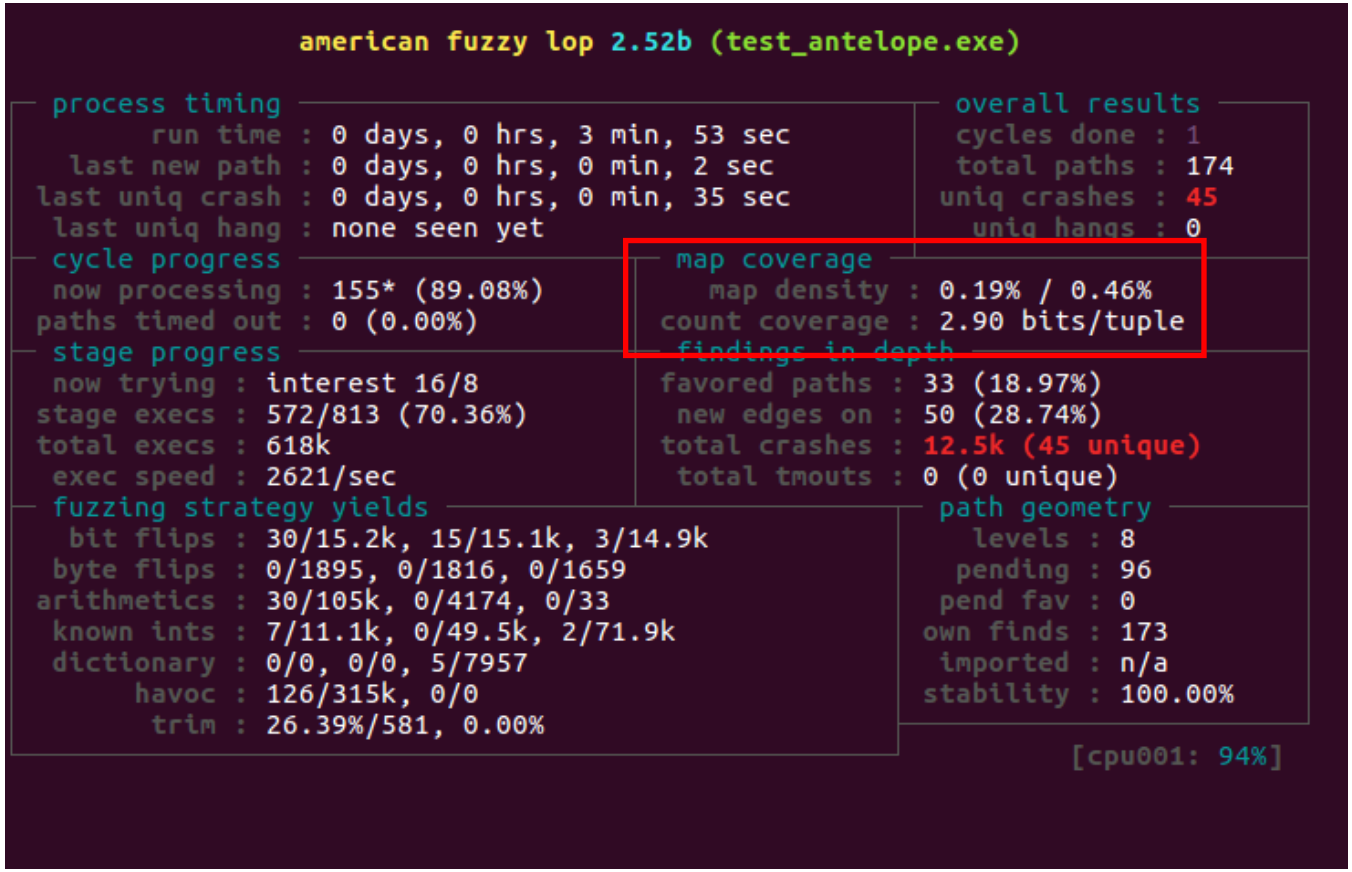

**Process timing:**
how long fuzzing works
time elapsed since last result

# American Fuzzy Lop – fuzzing in progress




**Now processing:**
ID of current testcase
n* - current testcase
is not „favored”

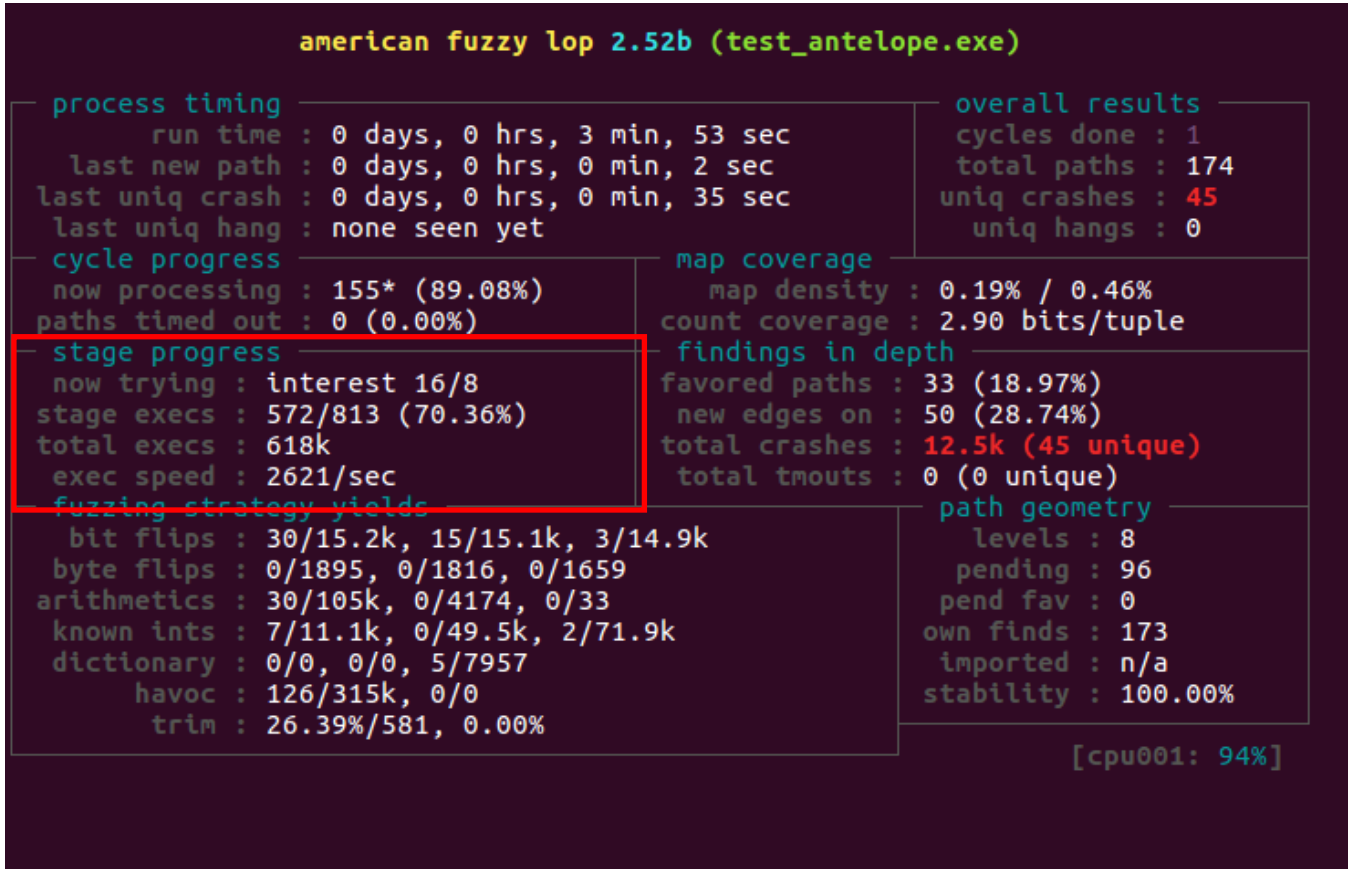
# American Fuzzy Lop – fuzzing in progress




For very large binaries coverage map can be saturated and path collisions can occur.

If you see any red colors in this box , read the docs how to increase size of the map!

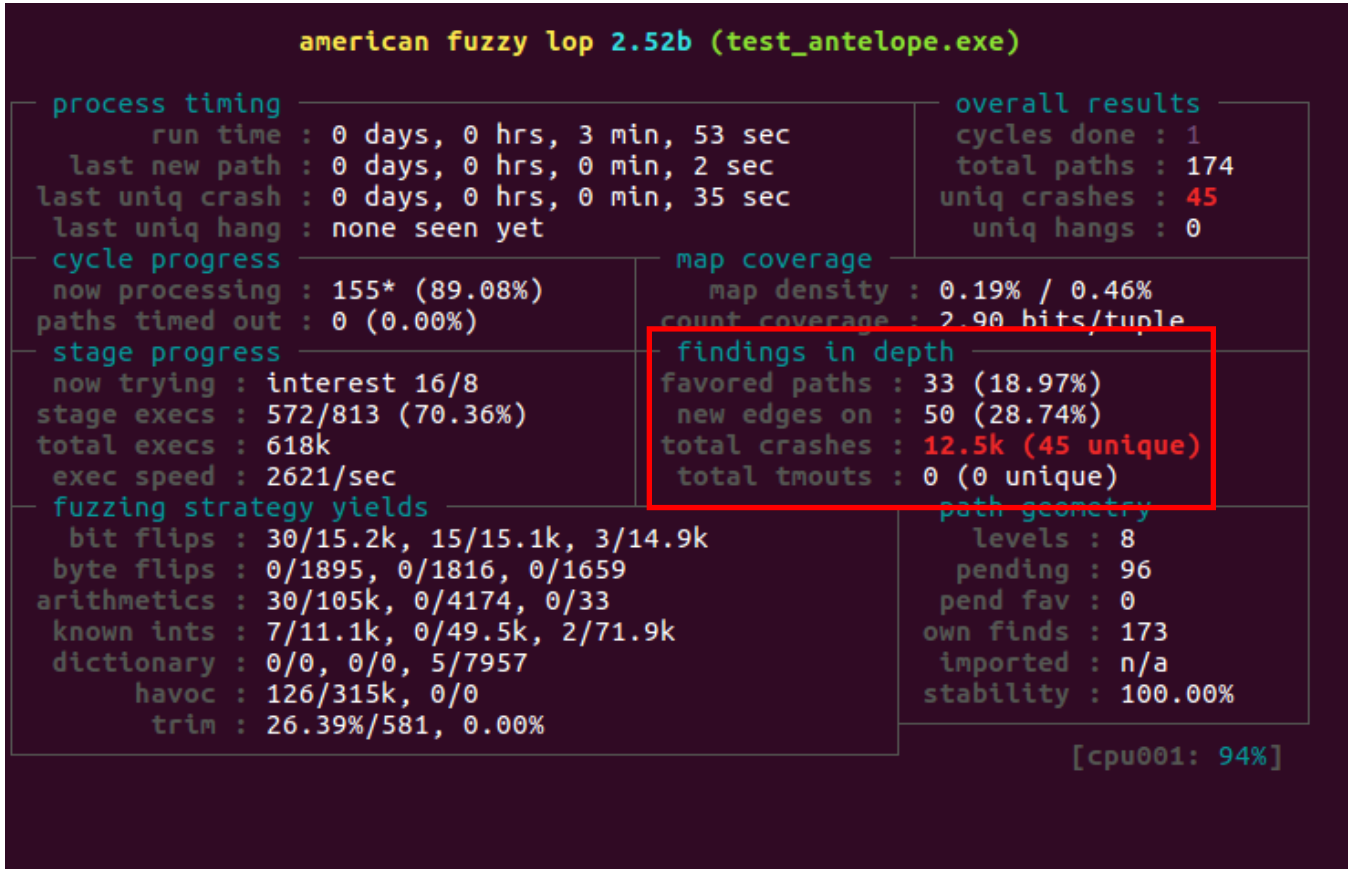
# American Fuzzy Lop – fuzzing in progress




**Stages:**

- calibration – initial checks
- trim Length/Stepover
- bitflip Length/Stepover
- arith Length/Stepover=8
- interest Length/Stepover=8
- extras (user or auto dict)
- havoc – multiple stacked ops
- splice – last-resort stage after cycle with no new paths similarly to havoc but start with splice of two random files
- sync (for parallel fuzzing) – sync with other workers

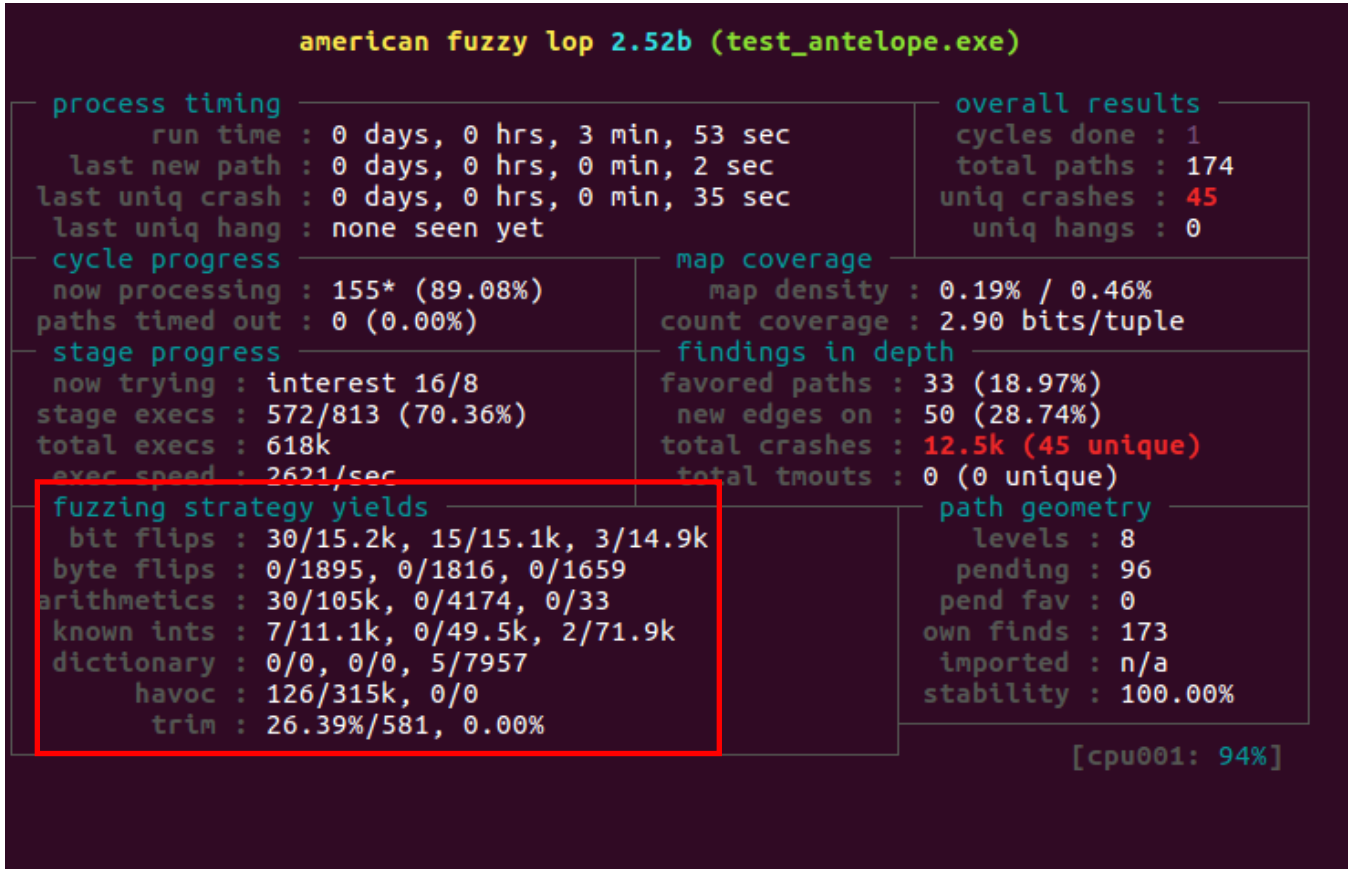
# American Fuzzy Lop – fuzzing in progress




**Findings:**

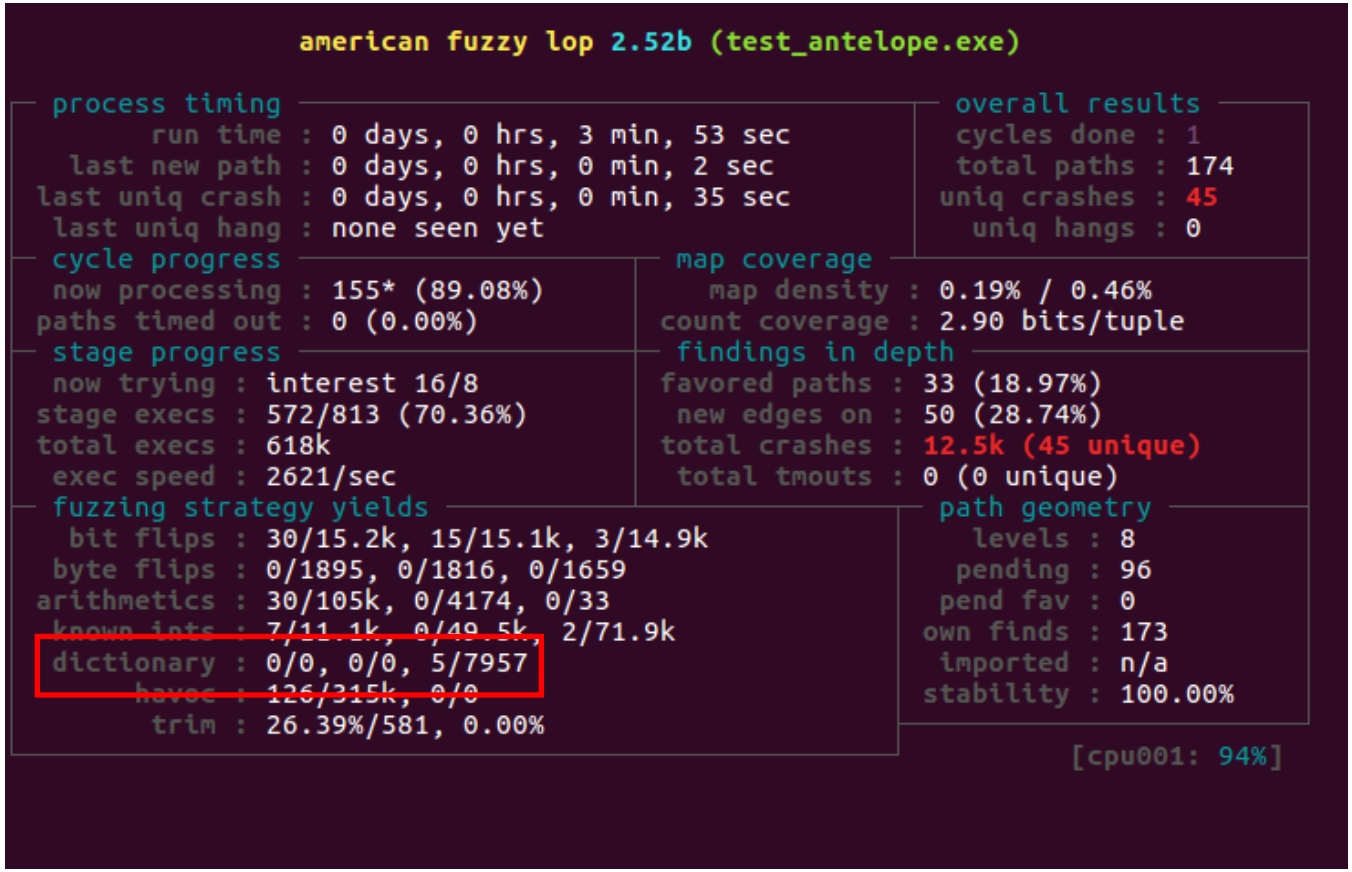
- favored – paths selected as priority ones based on fuzzing heuristics (minimization) When „favored paths" drops to zero, afl not doing fast progress.
- new edges on – paths resulting in better edge coverage
- total timeouts – timeouts are becoming hangs after multiple executions and exceeding larger timeout

# American Fuzzy Lop – fuzzing in progress




**Yields:**
tuples: 30/15.2k
First value shows number of successful usages of strategy (new path was found)
Second number shows number of all tries of this strategy

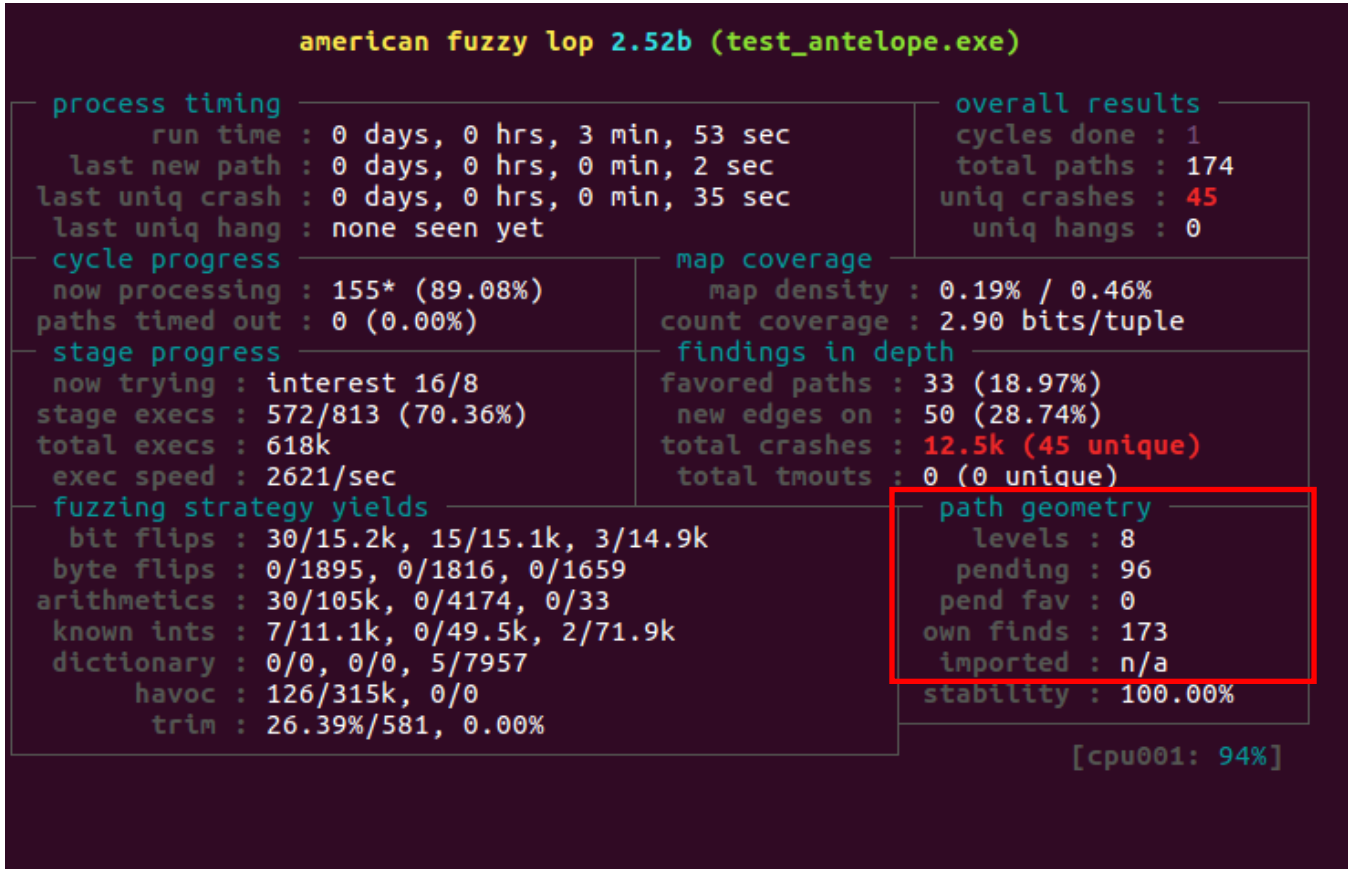
# American Fuzzy Lop – fuzzing in progress




For dictionaries:
third tuple is for auto-generated dictionaries

You can see content of dict in:
./queue/.state/auto_extras/

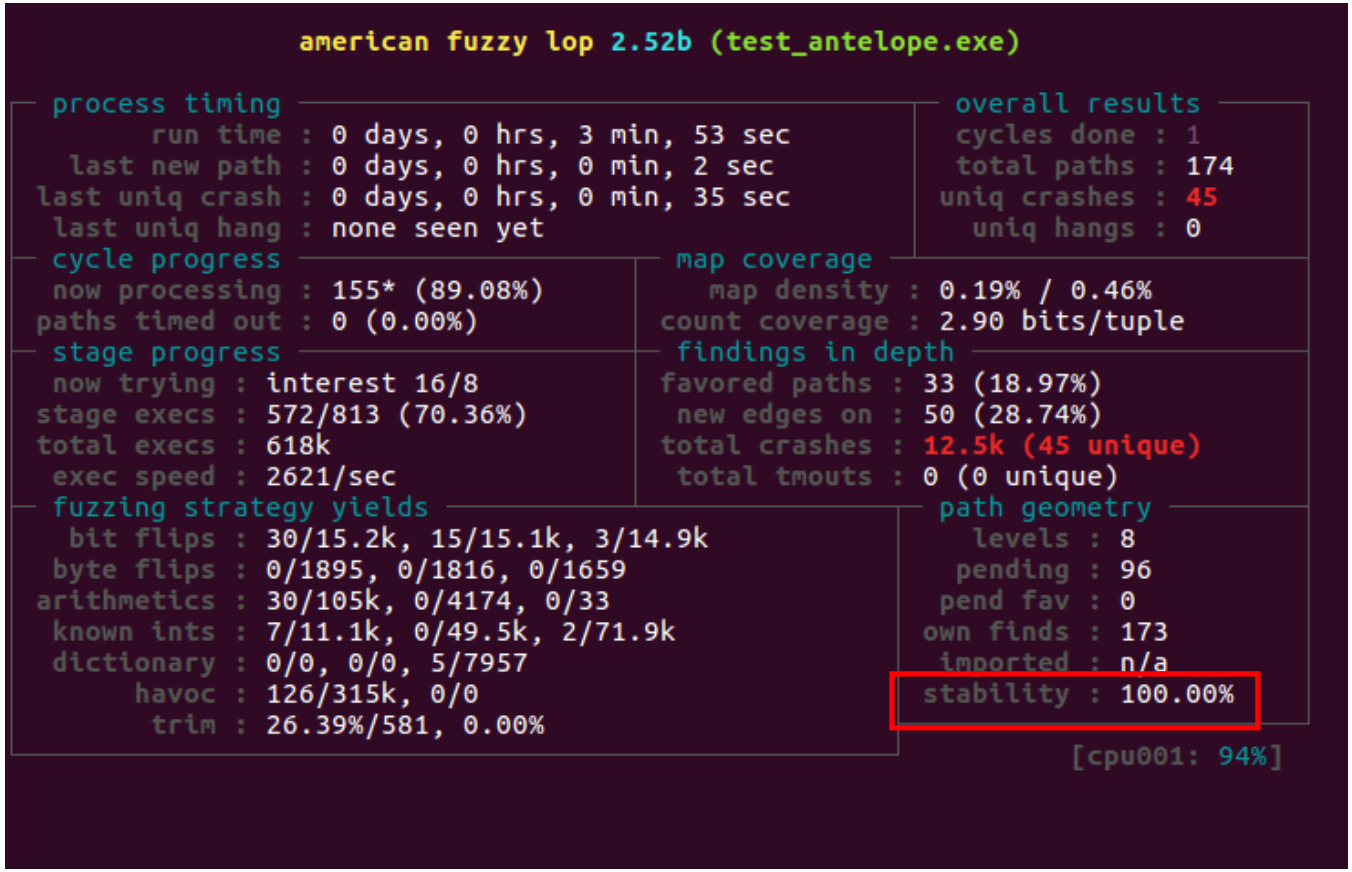
# American Fuzzy Lop – fuzzing in progress




**Path geometry:**

- levels:
level 1 are initial testcases,
level n are inputs derived
from testcases at level (n-1)
- pending – new test cases
not used yet in fuzzing
- pending fav –pending cases
selected as priority ones
- own finds – new paths found
by this fuzzing instance
- imported – new paths imported
from other fuzzing instances

# American Fuzzy Lop – fuzzing in progress

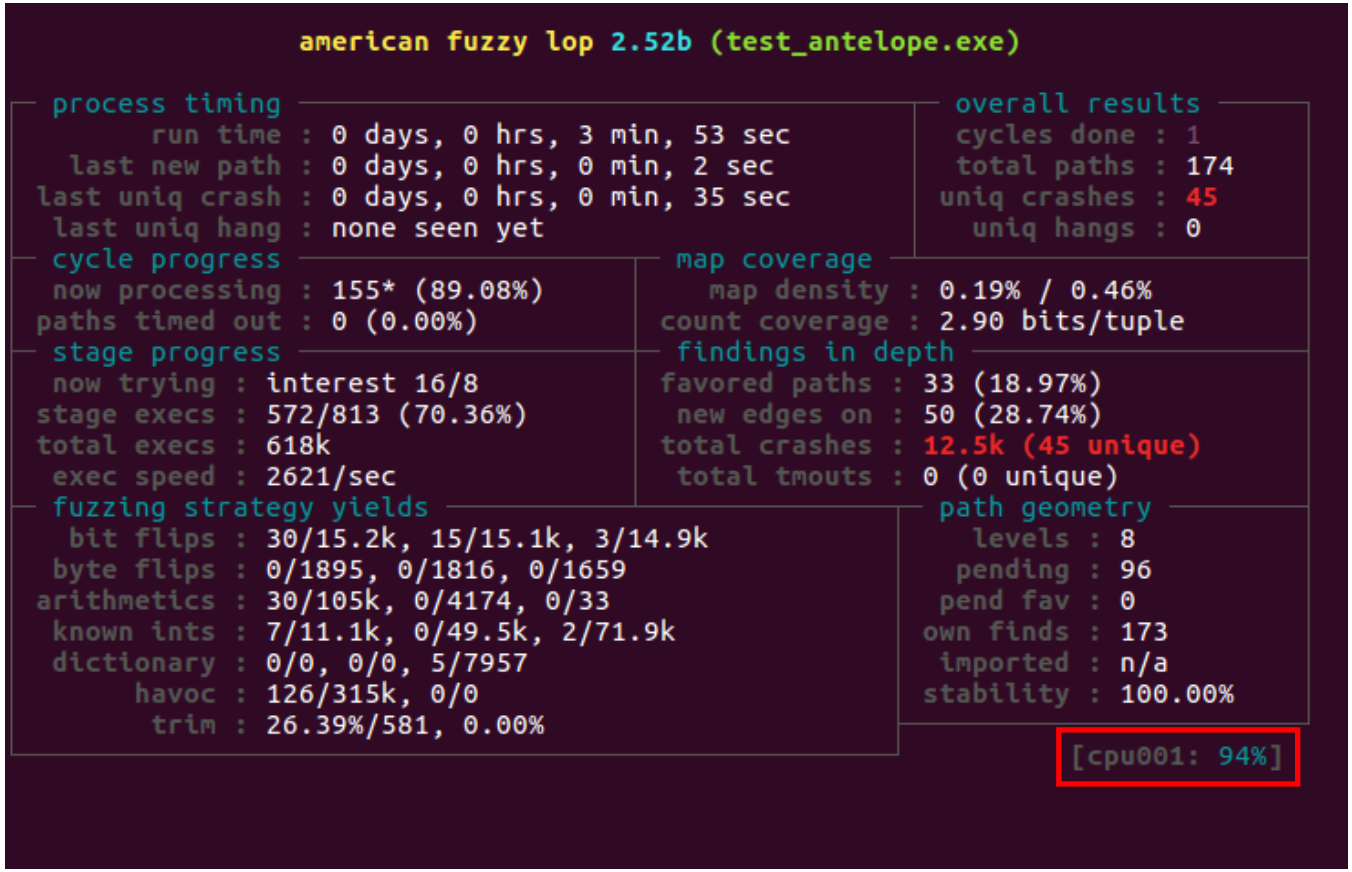



**Stability:**
100% - always the same output and execution path
for the same input

If stability lower than 100%:

- exec depends on random functions
- unitialized memory is used
- persistent resources are used (temporary files or shared memory objects)
- multiple threads depends on exec order
- persistent mode is used

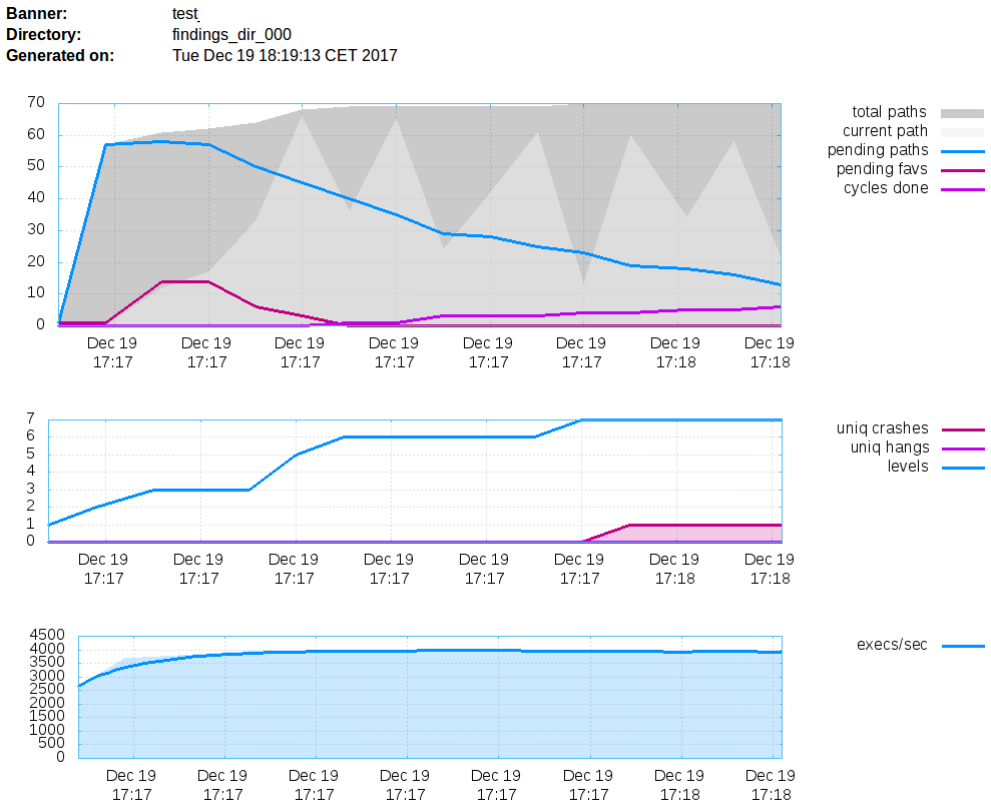
# American Fuzzy Lop – fuzzing in progress




CPU allocated and overall load of this system
- **RED** – system is overalloaded

If you want more detailed and accurate information use afl-gotcpu
(will be described later)

# afl standard tools – afl-plot



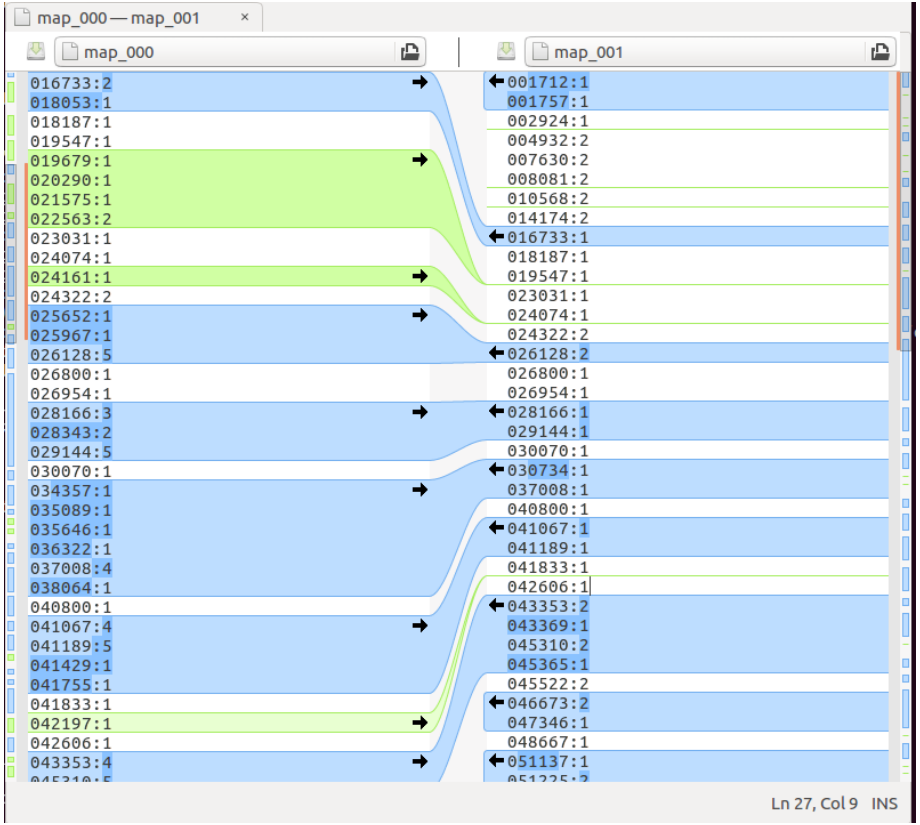

Generated using: *afl-plot findings_dir plot_dir*

Shows multiple metrics over whole fuzzing process

When number of pending favs goes to zero and increase of total paths is almost none fuzzer is less likely to produce any new results

Slowdown of execution over time shows that fuzzers can be exhausting some shared resource (e.g. disk)

# afl standard tools – afl-showmap

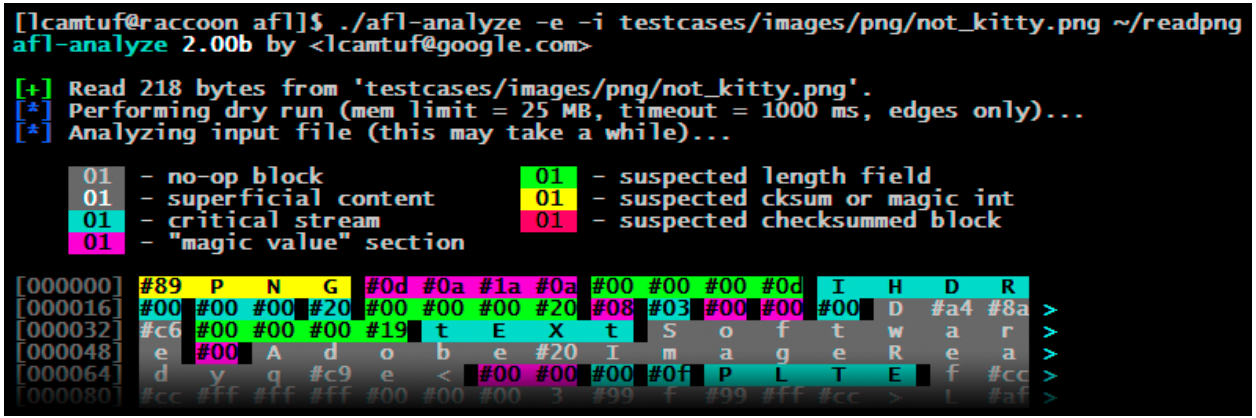



Shows coverage map in raw format

Useful only in debugging purposes
(manual comparison of paths)
or for automated analysis

# afl standard tools – afl-analyze

- Can be useful to analyze protocol packets or unknown format input files
- Usually results are not stable (change on multiple runs)




Source: lcamtuf Automatically inferring file syntax with afl-analyze
https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html

# Fuzzing – analysing crashes / hangs

- **Minimise testcases – afl-tmin**
  - Reduces size of input file so that issue still occurs
- **Using sanitizers (e.g. ASAN – Address Sanitizer)**
  - Provides more detailed information about crash
  - Sample Makefiles generates two binaries (w / wo ASAN)
- **Debug the issue**
  - Use you favourite debugging method – e.g. gdb / printf ☺
- **Fix the vulnerability and start fuzzing again**
  - afl will not find previously found crashes
- **Minimise the corpus – afl-cmin**
  - Selectes only unique paths from given samples

Source: Defusing the situation U.S. Air Forces Central Command News
License: Public domain

# Fuzzing – afl-tmin

**BEFORE**

```
1  UHELP ERuseddYCT
2  TYPE a[SOH]stringUDmeXSKD paghP
3  XCXMKDa□nMDHon value
4  USER tr□ng
5  RM;NELame
6  CDPWDUL P
7  XNOO`+UP
8  RM& □□lna[SOH]stri□gKDvalueSKD XCUPn□L pathname
9  1□□□PSV@□□[EOT]□□□□□□□□□LOT
10 r□□\n
11 SIZE nathnGmtV□tad
12 SMODETRUpm
13 nEkT  o^PRTvhluc
14 SITE□□□ng[DLE][DLE]e
15 [DLE]TE□
16     ng
17 cMC[EOT]□□□□□□NELP
18 HED[SOH] strin□PWD
```

**AFTER**

```
1  TYPE a
2  USER
3  SIZE X
```

# Fuzzing – afl-tmin

**BEFORE (352 chars, 18 FTP commands)**

```
1  UHELP ERuseddYCT
2  TYPE a[SOH]stringUDmeXSKD paghP
3  XCXMKDa□nMDHon value
4  USER tr□ng
5  RM;NELame
6  CDPWDUL P
7  XNOO`+UP
8  RM& □□lna[SOH]stri□gKDvalueSKD XCUPn□L pathname
9  1□□□PSV@□□[EOT]□□□□□□□□□LOT
10 r□□\n
11 SIZE nathnGmtV□tad
12 SMODETRUpm
13 nEkT  o^PRTvhluc
14 SITE□□□ng[DLE][DLE]e
15 [DLE]TE□
16     ng
17 cMC[EOT]□□□□□□NELP
18 HED[SOH] strin□PWD
```

**AFTER (20 chars, 3 FTP commands)**

```
1  TYPE a
2  USER
3  SIZE X
```

# Fuzzing – Tips & Tricks (how to find more issues)

- **Analyse the coverage**
  - Sample Makefile help to use lcov
- **Increase the coverage**
  - Manualy prepare testcase that allow afl to find new paths
  - Modify the code to create new paths

***LCOV - code coverage report***

| | | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| **Current view:** | top level - servers_root | **Lines:** | 1087 | 2233 | 48.7 % |
| **Test:** | coverage.info | **Functions:** | 66 | 115 | 57.4 % |
| **Date:** | 2017-12-08 14:33:05 | | | | |

| Filename | Line Coverage | | Functions | |
|---|---|---|---|---|
| aes.c | 0.0 % | 0 / 169 | 0.0 % | 0 / 5 |
| base64.c | 47.6 % | 30 / 63 | 50.0 % | 1 / 2 |
| bignum.c | 54.1 % | 396 / 732 | 62.8 % | 27 / 43 |
| rsa.c | 28.2 % | 57 / 202 | 45.5 % | 5 / 11 |
| sha2.c | 75.1 % | 157 / 209 | 50.0 % | 5 / 10 |
| test_root.c | 79.4 % | 50 / 63 | 50.0 % | 2 / 4 |
| trm_verifier.c | 30.6 % | 53 / 173 | 46.7 % | 7 / 15 |
| x509parse.c | 55.3 % | 344 / 622 | 76.0 % | 19 / 25 |

Generated by: LCOV version 1.12

```
        :
      320 : static int x509_get_name( unsigned char **p,
        :                             unsigned char *end,
        :                             x509_name *cur )
        : {
        :         int ret, len;
        :         unsigned char *end2;
        :         x509_buf *oid;
        :         x509_buf *val;
        :
      320 :         if( ( ret = asn1_get_tag( p, end, &len,
        :                                   ASN1_CONSTRUCTED | ASN1_SET ) ) != 0 )
        :         {
        0 :                 return( XYSSL_ERR_X509_CERT_INVALID_NAME | ret );
        :         }
        :
      320 :         end2 = end;
      320 :         end  = *p + len;
        :
      320 :         if( ( ret = asn1_get_tag( p, end, &len,
        :                                   ASN1_CONSTRUCTED | ASN1_SEQUENCE ) ) != 0 )
        :         {
        0 :                 return( XYSSL_ERR_X509_CERT_INVALID_NAME | ret );
        :         }
        :
      320 :         if( *p + len != end )
        :                 return( XYSSL_ERR_X509_CERT_INVALID_NAME |
        :                         XYSSL_ERR_ASN1_LENGTH_MISMATCH );
        :
      320 :         oid = &cur->oid;
      320 :         oid->tag = **p;
        :
      640 :         if( ( ret = asn1_get_tag( p, end, &oid->len, ASN1_OID ) ) != 0 )
        :         {
        0 :                 return( XYSSL_ERR_X509_CERT_INVALID_NAME | ret );
        :         }
        :
```

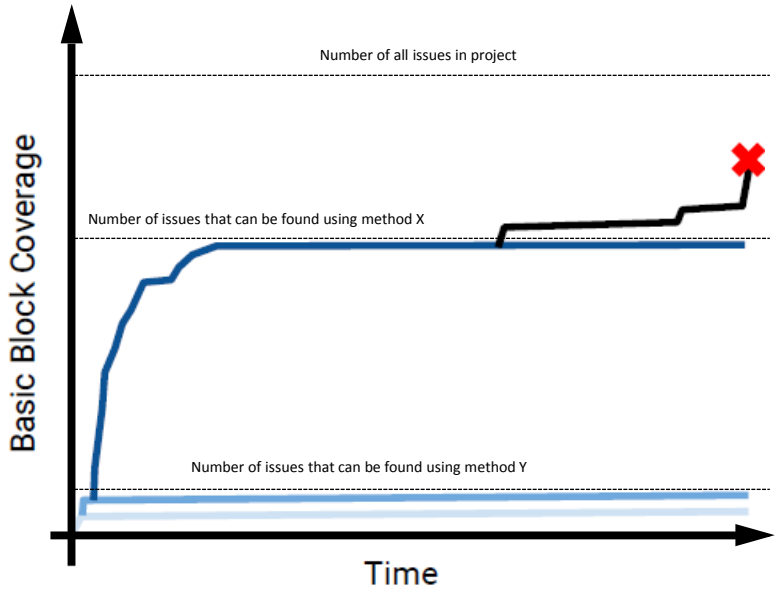
# Fuzzing – when to finish?

- **Finishing conditions:**
  - No new paths or hangs and crashes were generated during last time (e.g. week)

  and
  - Code is (almost) fully covered by test cases

  or
  - Use Pythia to analyze metrics (paths coverage)

  or
  - We don't have more time for fuzzing ☺




Source: Zardus „25 Years of Program Analysis" (Def Con 25)

# Fuzzing AQL (Antelope Query Language) – (1/2)

- **AQL basics**
  - AQL is simplified version of SQL for IoT devices
  - Library from Contiki OS (open-source project)
  - Processing steps of query:
    1. Tokenization (splitting raw text into tokens)
    2. Syntax analysis (checking whether query is correct)
    3. Running the query on database

- **Lessons learned**
  - First issues were found in tokenization (crash)
  - Afterwards: long fuzzing with no results
    afl is weak in creating AQL queries!
  - Solution 1: use afl dictionary (sets of SQL keywords)
    - A) built-in dict for SQL (3x more keywords than AQL)
    - B) adjust builtin dict for AQL
  - Solution 2: generate AQL from binary data

# Fuzzing AQL (Antelope Query Language) – (2/2)

- **Solution 2: generate AQL from binary data**
  - Generated AQL needs to be **stable** (same binary -> same AQL)

  - Simplest idea for AQL generation:
    1. Create a const table with all AQL types of tokens (keywords, operators and symbols)
    2. Convert binary data – each byte % NR_TOKENS -> token

  - Generates a long valid AQL files for fuzzing opening new paths
  - In a few minutes new crashes ☺

# American Fuzzy Lop – Excercises

- Prerequisites:
  Install provided Docker file

- You will receive source code AQL engine
  (taken from Contiki OS) with wrapper
  – all ready to start fuzzing ☺

- Part 1 – basic fuzzing with „hello” txt file
- Part 2 – fuzzing with SQL file and dictionary
- Part 3 – analysing crashes
- Part 4 – fuzzing with another SQL file and generator

# American Fuzzy Lop – Excercises (part 1)

1. Unpack the *afl_aql.zip*
2. Analyse *Makefile* but don't touch anything yet ☺
3. Run *make dox*
4. Open *dox_doc.html* in browser
5. Analyse file antelope/test_aql.c (wrapper) – function *main()*
6. Run *make all*
7. Open and analyse testcases in directory testcase_dir_000
8. Run *make fuzz-init*
9. Wait 2-3 minutes
10. If you already found crashes, you are lucky! Try to gamble today!
11. Observe testcases in directory *finding_dir_000/queue/*
12. Run *make plot*
13. Open *afl_plot.html* in browser
14. Run *make lcov*
15. Open *lcov_doc.html* in browser
16. Analyse coverage of different files and graphs

# American Fuzzy Lop – Excercises (part 2)

1. Open and analyse testcases in directory testcase_dir_001
2. Open *Makefile*
3. Change parameter *TESTCASE_DIR* to testcase_dir_001
4. Change parameter *FINDINGS_DIR* to findings_dir_001
5. Check parameter *AFL_DICT* (whether path to sql.dict is valid)
6. Run *make fuzz-init-dict*
7. Wait 2-3 minutes
8. Observe testcases in directory *finding_dir_001/queue/*
9. Run *make plot*
10. Open *afl_plot.html* in the browser
11. Analyse graphs in comparison to previous fuzzing

12. Analyse **crashes** (next page)!

# American Fuzzy Lop – Excercises (part 3)

1. If you still haven't found any crashes, this is not your lucky day!
   Change parameter *TESTCASE_DIR* to *testcase_dir_002*
   a. Change parameter *FINDINGS_DIR* to *findings_dir_002*
   b. Run *make all* and *make fuzz-init-dict*

2. Run afl-tmin on your case
   *afl-tmin –i crash_000.sql –o crash_000_min.sql ./test_aql.exe @@*
3. Analyse differences between both files
4. Minimise another crash case and compare with the previous one

5. Select crash file that you want to analyse
6. Analyse the name of the crash file (e.g. generation method)

# American Fuzzy Lop – Excercises (part 4)

1. Open *Makefile*
   a. Change param *FIXES* to following version *–DPATCH_1 –DPATCH_2*
   b. Change parameter *TESTCASE_DIR* to testcase_dir_003
   c. Change parameter *FINDINGS_DIR* to findings_dir_003
   d. Uncomment param *GENERATOR*
2. Run *make all* (recompile using new params)
3. Run *make fuzz-ini3*
4. Analyse file antelope/test_aql.c (wrapper) – function *convert_raw_data()*
   This is a generator of AQLs from raw data.
5. Wait until crashes (can take more time)

6. Observe testcases in *finding_dir_003/queue/* (run with *test_aql.exe* to see AQL)
7. Run *make plot*
8. Open *afl_plot.html* in the browser
9. Run *make lcov*
10. Analyse coverage and graphs in comparison to previous fuzzing exercises

# afl execution modes (from slowest to fastest)

- **No fork server (full execution) mode**
  - Run afl with AFL_NO_FORKSERV=1
  - 30% slower than basic mode
  - Usable only for programs messing with fork server (e.g. using threads in very weird way)
- **Basic fork server mode**
  - Standard version without using any flags
- **Deferred instrumentation**
  - Fork is performed after selected initialization tasks
  - Requires usage of LLVM/clang
- **Persistent mode**
  - Process is reused without forking
  - Inspired by libfuzzer (in-process fuzzing)
  - Requires usage of LLVM/clang

# LLVM mode

- **Deferred instrumentation**
  - Speed up: depends on the application – can be none or 5x
    - Usage:

```
int main(int argc, char *argv[])
{
        /* Perform initilization of application */
        /* This is going to be run only once!!! */

        #ifdef __AFL_HAVE_MANUAL_CONTROL
          __AFL_INIT();
        #endif

        /* FUZZ here */

            /* Read input data. */
            /* Execute code to be fuzzed. */
            /* Reset state. */

        /* Exit normally */
        return(0);
}
```

- **Persistent mode**
  - Speed up: 3-7x
    - Usage:

```
int main(int argc, char *argv[])
{
        while (__AFL_LOOP(1000)) {

            /* Read input data. */
            /* Execute code to be fuzzed. */
            /* Reset state. */

        }

        /* Exit normally */
        return(0);
}
```

**It is recommended to use both!**

# LAF LLVM Passes

- Changes compare operations to multiple steps
  - allows coverage based fuzzing of compares
    - functions (only strcmp and memcpy)

- Distributed as patch to afl LLVM mode
- Turned on with flags:
  export LAF_SPLIT_SWITCHES=1
  export LAF_TRANSFORM_COMPARES=1 (only strcmp, memcmp)
  export LAF_SPLIT_COMPARES=1

- Compiled code will be slower (deoptimized),
  so not recommended to use ONLY LAF binaries!

- See article for more details:
  https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/

# Fuzzing – Tips & Tricks (assertions and diff fuzz)

- **Using assertion**
  In C <assert.h> void assert(int expression)
  In C++ static_assert() or BOOST_STATIC_ASSERT_MSG()
  Triggered assertions are treated as crashes because they abort()
  So assertion can be used for identifying by afl weird cases or values

- **Differential fuzzing**
  Using additional reference implementation and checking differences
  Recommended to instrument only tested implementation
  Useful for testing cryptography or network packet processing

# Fuzzing – Tips & Tricks (speeding up fuzzing)

- **Testcases should be small**
  - Fuzzing is performed byte by byte so empty spaces in files waste CPU time
- **Remove unecessary code**
  - Loading data / creating structures that are not used in parsing
- **Analyse and increase performance**
  - Use gprof to analyse performance issues
- **Eliminate bottlenecks**
  - Remove checksums / crypto operations
- **Instrument only fuzzed part of codes**
- **Check OS configuration**

(see afl documentation: *Tips for performance optimization)*

# afl crash exploration mode (Peruvian rabbit mode)

- **Requirements**
  - All input files need to be crashes
    (no hangs and normal cases)
- **Usage**
  - Run afl-fuzz with –C
- **Possible usages**
  - Run crash exploration mode and generate new crashes
  - Fix the initial crash(es)
  - Check whether patch fixed all generated crashes

**or**

  - Use crash triage tool
    (e.g. crashwalk with exploitable plugin)
  - Look for „more exploitable” crashes than initial crash(es)

# afl Tips & Tricks – Environment Variables

For afl-gcc/clang/clang-fast:

- AFL_CC, AFL_CXX, AFL_AS – allows you to use alternate compile tools
- AFL_DONT_OPTIMIZE – sets -O0 instead of default -O3 (sometimes generates compilation errors)
- AFL_USE_ASAN, …MSAN – enables ASAN or MSAN without flags directly to compiler
- AFL_INST_RATIO – sets probability of instrumenting branches (lower this for very large binaries)
- AFL_HARDEN – enables code hardening (useful for catching non-crashing errors, sub 5% perf loss)

For afl-fuzz:

- AFL_SKIP_CRASHES – afl will tolerate crashing files in the input queue
  This is useful when you want to start fuzzing with large set of testcases without fixing crashes before (e.g. using testcases generated without ASAN for fuzzing ASAN binaries)
- AFL_PRELOAD – the same as LD_PRELOAD but afl binary will not be preloaded

# Sanitizers

- **Set of dynamic analysis tools designed by Google for LLVM**
  - **ASAN – Address Sanitizer** – detects memory errors
  - **LSAN – Leak Sanitizer** – detects memory leaks
  - **MSAN – Memory Sanitizer** – detects operations on unitialized data
  - TSAN – Thread Sanitizer – detects races between threads
  - UBSAN – Undefined Behaviour Sanitizer
  - Only ASAN/LSAN (x)or MSAN can be used with afl
  - Santizers cannot be used in QEmu mode
  - See the presentation „LCU14 201 – Binary Analysis Tools” for more details!
    https://www.slideshare.net/linaroorg/lcu14-201-binary-analysis-tools
    https://www.youtube.com/watch?v=QIu601HYwSA

# Sanitizers – Address Sanitizer (ASAN)

- Detects all major types of memory errors
  - heap/stack/global buffer overflows
  - use after free / return
  - double free / invalid free

- Wraps all memory access functions like strlen
- Marks (poisons) memory regions around allocated buffers

- Even one-byte overread are detected!

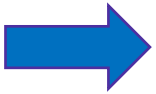
  But…
  - Some crashes are not-reproducible in the wild
  - Most of issues identified only by ASAN are not exploitable
  - Slowdown: 2-5 times
  - Can exhaust memory while fuzzing –
    afl warns that some crashes can be false-positives

# Sanitizers - Address Sanitizer limitations

- Memory overflows/underflows inside structures will not be found!

- Example:

```
struct aql_adt {
  char relations[AQL_RELATION_LIMIT][RELATION_NAME_LENGTH + 1];
  aql_attribute_t attributes[AQL_ATTRIBUTE_LIMIT];
  aql_aggregator_t aggregators[AQL_ATTRIBUTE_LIMIT];
  attribute_value_t values[AQL_ATTRIBUTE_LIMIT];
  index_type_t index_type;
  uint8_t relation_count;
  uint8_t attribute_count;
  uint8_t value_count;
  uint8_t optype;
  uint8_t flags;
  void *lvm_instance;
};
```



```
struct aql_adt {
  char relations[AQL_RELATION_LIMIT][RELATION_NAME_LENGTH + 1];
       char canary_01;
  aql_attribute_t attributes[AQL_ATTRIBUTE_LIMIT];
       char canary_02;
  aql_aggregator_t aggregators[AQL_ATTRIBUTE_LIMIT];
       char canary_03;
  attribute_value_t values[AQL_ATTRIBUTE_LIMIT];
       char canary_04;
  index_type_t index_type;
  uint8_t relation_count;
  uint8_t attribute_count;
  uint8_t value_count;
  uint8_t optype;
  uint8_t flags;
  void *lvm_instance;
};
```

- Solution:
  1. manually add „canary” between buffers inside structs
     (works only if the application not depending on struct layout – e.g. as protocol packets)
  2. set canary value at the beginning
  3. check canary value after each operation

# Sanitizers – Leak Sanitizer (LSAN)

- Detects memory leaks (allocated memory that is not released before end of program)
- Integrated with ASAN (can be used separately with *–fsanitize=leak*)
- Errors can be suppressed by flag *LSAN_OPTIONS=supressions=file.txt*

- Cannot be used in persistent mode

- Example error:

```
==7829==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x42c0c5 in __interceptor_malloc /usr/home/hacker/llvm/projects/compiler-rt/lib/asan/asan_malloc_linux.cc:74
    #1 0x43ef81 in main /usr/home/hacker/memory-leak.c:6
    #2 0x7fef044b876c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226

SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

# Sanitizers – Memory Sanitizer (MSAN)

- Detects operations on unitialized  memory regions and variables
- All code (including libraries) must be compiled with MSAN
- Example error:

```
==6726==  WARNING: MemorySanitizer: UMR (uninitialized-memory-read)
    #0 0x7fd1c2944171 in main umr.cc:6
    #1 0x7fd1c1d4676c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
  ORIGIN: heap allocation:
    #0 0x7f5872b6a31b in operator new[](unsigned long) msan_new_delete.cc:39
    #1 0x7f5872b62151 in main umr.cc:4
    #2 0x7f5871f6476c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
```

# Library preloading (LD_PRELOAD / AFL_PRELOAD)

- Allows to change implementations of standard library functions (e.g. sleep()) to own implementation (e.g. no sleeping)
- afl uses AFL_PRELOAD (only fuzzed binary is preloaded)
- Preeny (by Zardus) – set of useful preload libraries
- Examples:
  - defork – disables fork()
  - de(s)rand – disables (s)rand
  - desock – uses console as sock input
  - desleep – (u)sleep do nothing
  - crazyrealloc – every realloc is move
  - patch – patches program at load time

# Fuzzing programs using Random Number Generators

- **Problem 1:**
  Crashes depending on RNG output can be difficult to reproduce
- **Problem 2:**
  afl has problem with binaries using *srand(time(&t))*
  - They seem to be **stable** (giving the same results) when executed in the same second

```c
int main(int argc, char *argv[])
{
   time_t t;
   /* Intializes random number generator using time()
        (number of seconds since 00:00 Jan 1, 1970) */
   srand(time(&t));

   int x = rand();
   printf("Got x = %d x mod 1000 = %d\n", x, x%1000);
   switch(x % 1000) {

        case 0: printf("Got 0\n");
        break;

        case 1: printf("Got 1\n");
        break;

        case 2: assert(0); // crash
        break;
   }
   return(0);
}
```

- **Detection methods:**
  - *grep* the code for „rand”
  - disassemble (*objdump -d*) binary and *grep* for „rand”
  - *ltrace* binary and look for „rand”
- **Fix methods:**
  - manually change RNG seed to static value and recompile
  - remove RNG using LD_PRELOAD / AFL_PRELOAD
    WANT=**2** MOD=**1000** LD_PRELOAD=tools/preeny/x86_64-linux-gnu/desrand.so ./test_random.exe

# Library preloading (LD_PRELOAD / AFL_PRELOAD)

After desock application takes input from console:

- LD_PRELOAD=tools/preeny/x86_64-linux-gnu/desock.so ./htcpcp_server 9999
  < findings_dir_000/crashes/id\:000000\,sig\:11\,src\:000000\,op\:flip1\,pos\:3

Multiple preload libraries are separated with semicolon:

- AFL_PRELOAD=tools/preeny/x86_64-linux-gnu/desock.so:tools/preeny/x86_64-linux-gnu/defork.so
  afl-fuzz -i testcase_dir_000 -o findings_dir_000 ./htcpcp_server 9999

afl fork created by Doug Bridwell can be used for fuzzing network servers

- afl-network/afl-fuzz -i testcase_dir_000 -o findings_dir_001 -N tcp://localhost:9999 ./htcpcp_server 9999

# afl for network fuzzing

afl fork created by Doug Bridwell can be used for fuzzing network servers and clients

- https://github.com/jdbirdwell/afl

Usage for fuzzing server:

- afl-network/afl-fuzz -i testcase_dir_000 -o findings_dir_001 -N tcp://localhost:9999 ./htcpcp_server 9999

Usage for fuzzing client:

- afl-network/afl-fuzz -i … -o … –L -N tcp://localhost:9999 ./htcpcp_client localhost 9999 1 1

Limitations:

- Implements fuzzing only for the first write and ignores all responses and do not send next messages
- Most network servers run as background processes and process requests from many processes - they do not normally exit. A timeout delay is required in order to terminate these processes, and the default timeout used in afl-fuzz is usually too long. This also slows down whole fuzzing!
  Parameters:
  - -D delay_before_write in msec
  - -t timeout_delay in msec

# American Fuzzy Lop – fuzzing on multiple CPUs

- Runs 1 master afl instance and multiple slave instances (each using single CPU core)
- Master instance runs deterministic steps,
  while all slave instances run non-deterministic

- afl instances can be started and stopped at any time
- afl instances synchronize between each other using /sync/ directories

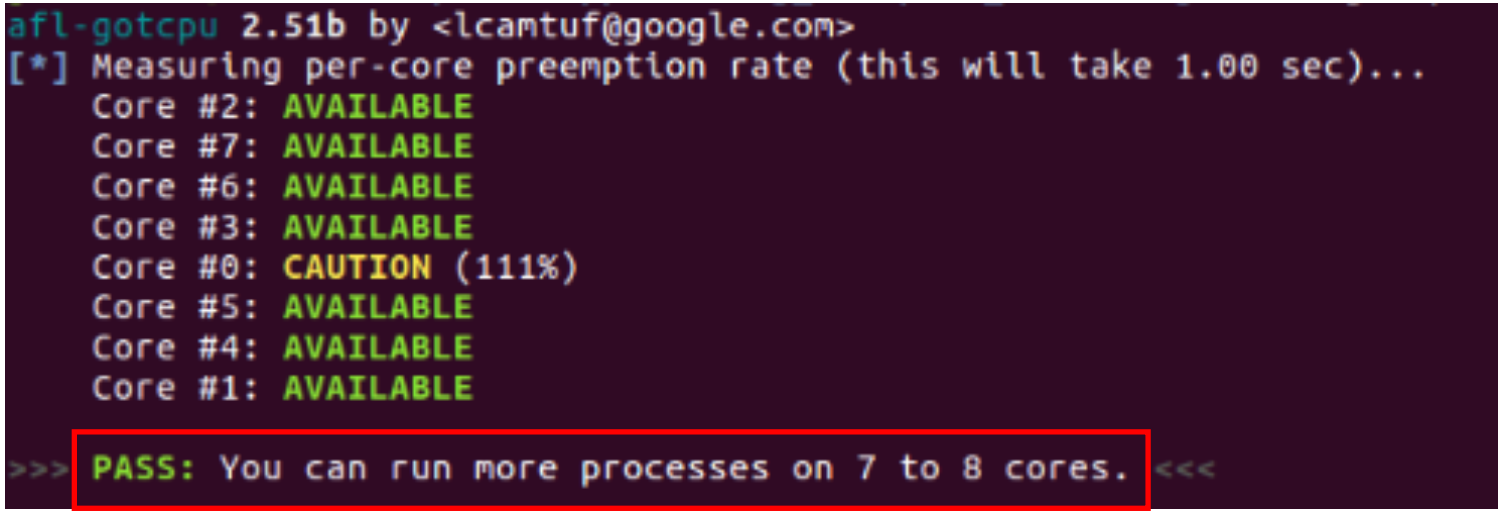
- Different afl branches and different binaries (e.g. with different Sanitizers)
  cooperate without any problems
  Recommended mix at the beginning of fuzzing:
  - Master afl instance using Pythia (to observe fuzzing status and metrics)
  - Multiple instances of „dumb and fast workers” afl-rb (Rare Branches) and afl-fast
  - Single instances of binaries using different Sanitizers
    and „other slow but smart” like LLVM-Compare detectors or Symbolic (driller/angr)

- Over time (less new/favored paths) more „fast workers” can be changed to „slow workers”

# American Fuzzy Lop – afl-whatsup

```
ubuntu@fuzzer:~$ afl-whatsup servers_root/findings_dir_001/
status check tool for afl-fuzz by <lcamtuf@google.com>

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

>>> fuzzer_main (7 days, 0 hrs) <<<

  cycle 2, lifetime speed 227 execs/sec, path 196/544 (36%)
  pending 0/261, coverage 3.26%, no crashes yet


...

>>> fuzzer_s9 (7 days, 0 hrs) <<<

  cycle 1457, lifetime speed 553 execs/sec, path 518/591 (87%)
  pending 0/0, coverage 3.26%, no crashes yet

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

     Fuzzers alive : 25
    Total run time : 175 days, 4 hours
       Total execs : 6739 million
  Cumulative speed : 11123 execs/sec
     Pending paths : 0 faves, 261 total
Pending per fuzzer : 0 faves, 10 total (on average)
     Crashes found : 0 locally unique
```

# American Fuzzy Lop – afl-gotcpu



# American Fuzzy Lop – Sister projects

- Support for other programming languages:
  - Go (Dmitry Vyukov)
  - Java (only GCC Java)
  - OCaml (KC Sivaramakrishnan)
  - Python (Jakub Wilk)
  - Rust (Keegan McAllister)

- Support for other environments:
  - Android (ele7enxxh)
  - Kernel (Linux, FreeBSD, Windows) – syzkaller (Dmitry Vyukov)
  - Kernel (Linux, MacOS, Windows) – kAFL (Sergej Schumilo)
  - TriforceAFL (Tim Newsham and Jesse Hertz)
  - Unicorn (Nathan Voss)
  - Windows binaries – WinAFL (Ivan Fratric)

# American Fuzzy Lop – Sister projects

- Fuzzing preparation support:
  - aflize (Jacek Wielemborek) – build afl versions of Debian packages
  - afl-sid (Jacek Wielemborek) – build and deploy AFL via Docker
  - docker-afl (Ozzy Johnson)

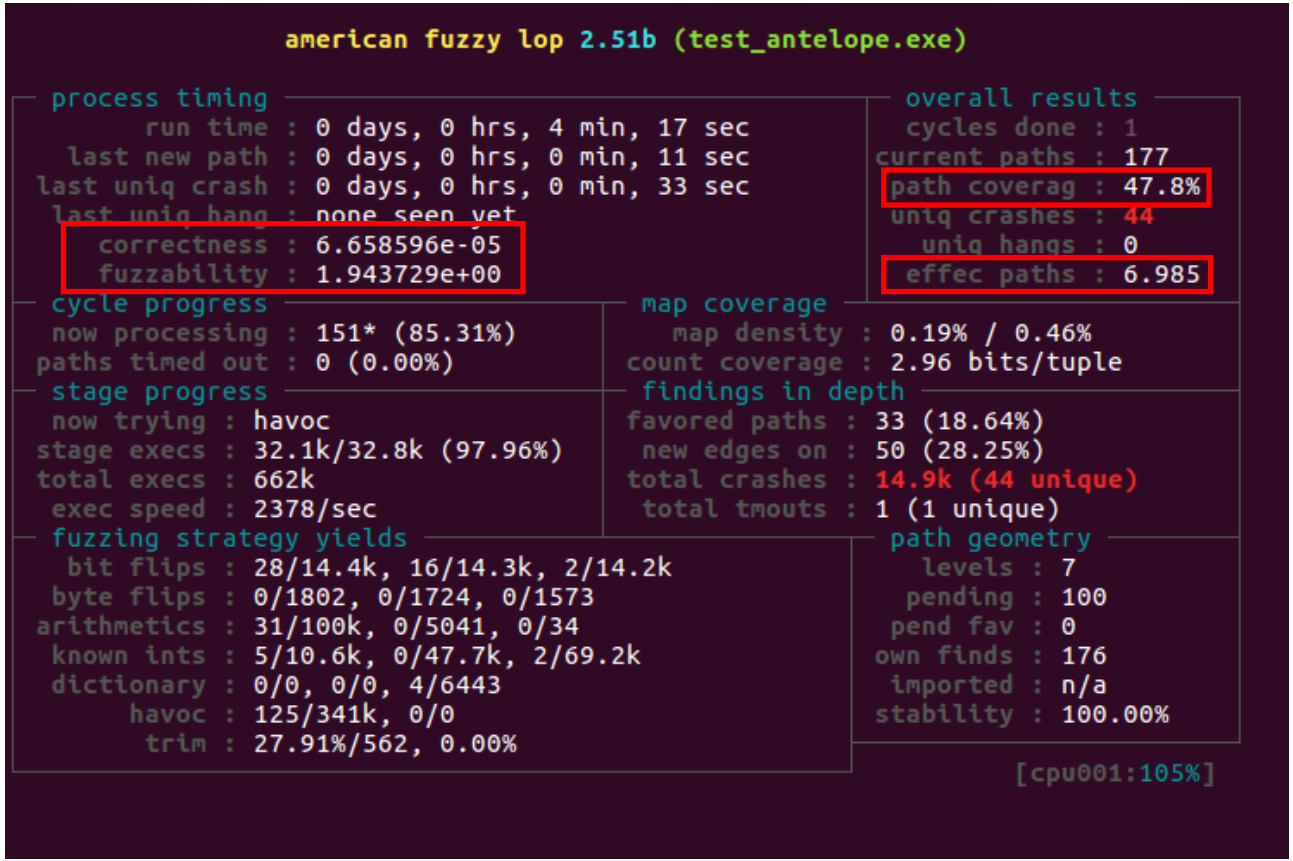
- Fuzzing on multiple cores or servers:
  - Roving (Richo Healey)
  - Distfuzz-AFL (Martijn Bogaard)
  - AFLDFF (quantumvm)
  - AFL Utils (rc0r)

- Triage of crashes:
  - afl-crash-analyzer (Tobias Ospelt)
  - crashwalk (Ben Nagy)
  - exploitable gdb plugin

# American Fuzzy Lop – Sister projects

- Additional tools:
  - afl-monitor (Paul S. Ziegler) – more detailed statistics
  - pythia (Marcel Boehme) – new statistical metrics to measure: completeness of fuzzing, probability to identify new vulnerability

# Pythia (new fuzzing metrics)




Introduces new metrics:

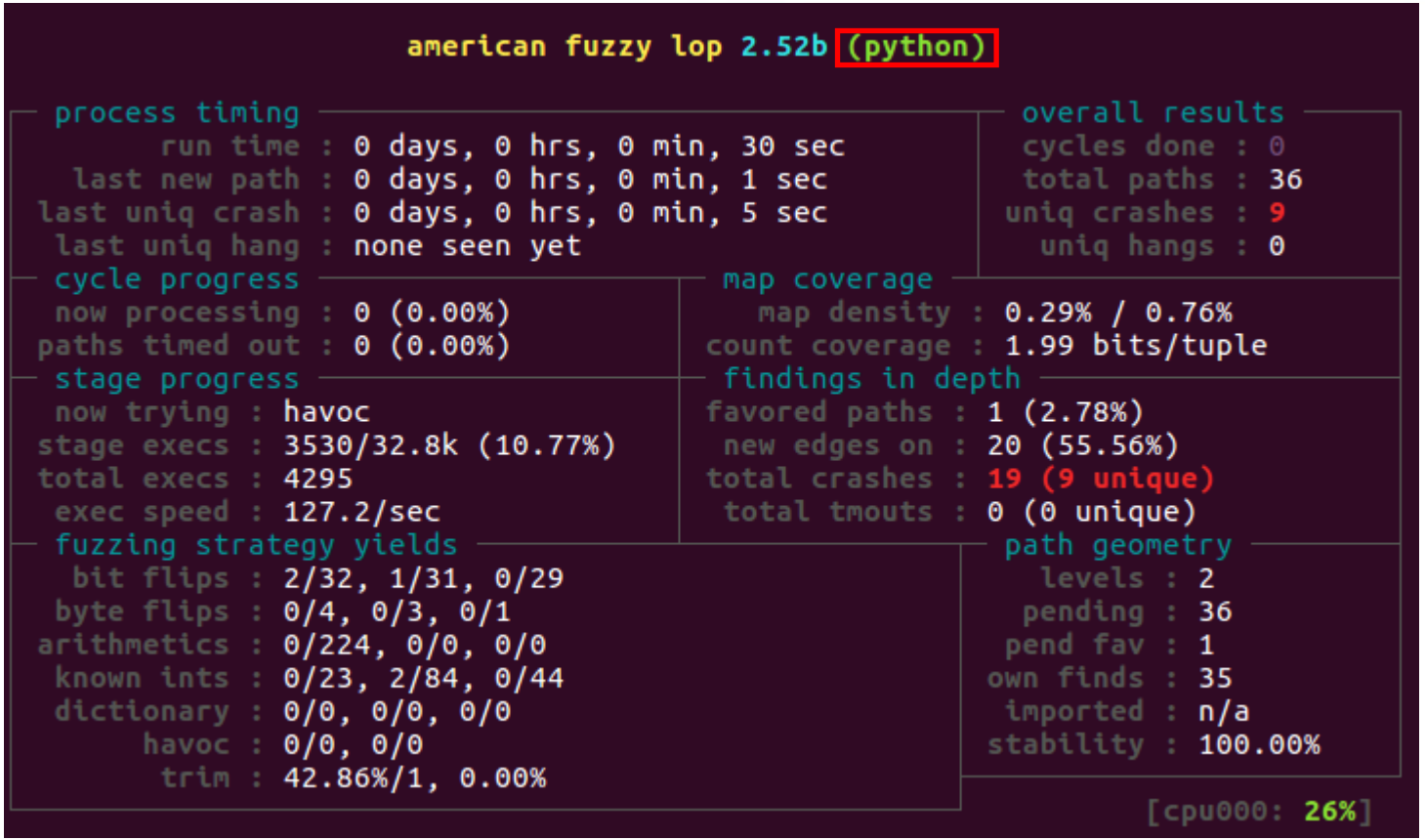
- correctness
- fuzzability
- path coverage
- effective paths

# python-afl

- **How to use:**
  - Install afl and Python 2.6+ or 3.2+
  - Install python-afl (with pip)
  - Prepare wrapper as for afl fuzzing
    - Import afl library
    - Run afl.init at start
  - Run fuzzer:
    *py-afl-fuzz –i testcase_dir*
    *–o findings_dit –m (limit or none)*
    *python wrapper.py @@*

```
# -*- coding: utf-8 -*-
import random
import socket
import threading
import unittest
from coapthon.messages.response import Response
from coapthon.messages.request import Request
from coapthon import defines
from coapthon.serializer import Serializer
from plugtest_coapserver import CoAPServerPlugTest
import sys
from coapthon.server.coap import CoAP
import afl

afl.init()
content = open(sys.argv[1], 'r')
data = content.read()

try:
    serializer = Serializer()
    message = serializer.deserialize(data, ("127.0.0.1", 5683))
    if isinstance(message, int):
        print("receive_datagram - BAD REQUEST")
    print("receive_datagram - " + str(message))

except RuntimeError:
    logger.exception("Exception with Executor")
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

# python-afl




afl status screen is identical to the standard one
Only difference is name of fuzzer – python

# python-afl

- **Tips:**
  - Immediate os._exit(0) – allows to speed up (no destructors at end)
  - Python 3 is much slower
  - Persistent mode can be used with: while afl.loop(1000) …

| | Python 2 | Python 3 | Native (C++) |
|---|---|---|---|
| dumb mode | 110/s | 47/s | 1200/s |
| pre-init | 130/s | 46/s | 5800/s |
| deferred | 560/s | 260/s | 6800/s |
| quick exit | 2700/s | 2100/s | 8700/s |
| rewinding persistent mode | 5800/s | 5900/s | - |
| persistent mode | 17000/s | 15000/s | 44000/s |

Source: Jussi Judin Taking a look at python-afl
https://barro.github.io/2018/01/taking-a-look-at-python-afl/

# crashwalk + exploitable plugin

- Deduplicates crashes and heuristically checks exploitation
- Requires Go and exploitable gdb plugin

- Runs with:
  - cwtriage – root . –afl
    (afl mode uses README.txt from /crash/ directory)
  - cwtriage –root . -- ./fuzz_app @@
  - cwdump ./crashwalk.db > triage.txt

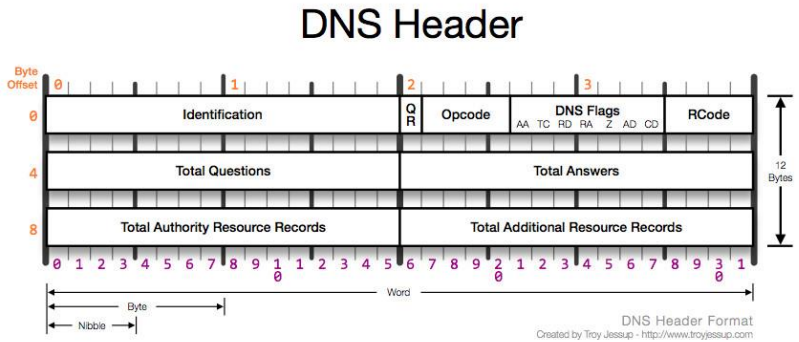
# crashwalk + exploitable plugin – result

- Result:

```
(1 of 8) - Hash: 8563714dca803fd20d4b216f6bbee015.8563714dca803fd20d4b216f6bbee015
---CRASH SUMMARY---
Filename: id:000001,sig:11,src:009013+008893,op:splice,rep:2
SHA1: 5b775abac792ff0d8bf9f0c5a6ff84ab2c5dced9
Classification: EXPLOITABLE
Hash: 8563714dca803fd20d4b216f6bbee015.8563714dca803fd20d4b216f6bbee015
Command: ..my_test id:000001,sig:11,src:009013+008893,op:splice,rep:2
Faulting Frame:
  trans4m_freq_2_time_fxp_2 @ 0x00000000004b65d9: in my_test
Disassembly:
Stack Head (5 entries):
  trans4m_freq_2_time_fxp_2 @ 0x00000000004b65d9: in my_test
...
 main                @ 0x00000000004076c1: in my_test
Registers:
rax=0x000000000028a5f0 rbx=0x0000000000711de2 rcx=0x00007fffffff9d50 rdx=0x0000000009230000
...
Extra Data:
 Description: Access violation on destination operand
 Short description: DestAv (8/22)
  Explanation: The target crashed on an access violation at an address matching the destination operand of the instruction. This likely indi
cates a write access violation, which means the attacker may control the write address and/or value.
---END SUMMARY---
```

# Fuzzing mDNS (Multicast Domain Name Service) server

- **mDNS basics**
  - DNS queries are perfect for fuzzing by afl
    - Small packets with densly packed values
    - No checksums or crypto
  - Server processing:
    1. Startup server (create data structures)
    2. Main loop:
       a. Listen for mDNS queries
       b. Parse incoming queries
       c. Prepare and send response
- **Lessons learned**
  - Whole packet processing (with server startup) took 10x more than parsing queries




| Questions |
| --- |
| Answers |
| Authority |
| Additional information |

# Fuzzing mDNS (Multicast Domain Name Service) server

- **Lessons learned**
  - Whole packet processing (with server startup) took 10x more than parsing queries
  - Approach: first fuzz only parsing function, afterwards whole server using packets generated in first step
  - Vulnerabilities were in both stages of processing

# Fuzzing FTP (File Transfer Protocol) server

- **FTP basics**
  - Server receives a set of commands from client
  - Interacts with filesystem (requests / creates files / dirs)

- **Fuzzing approach**
  - Parsing of single FTP command showed no vulns
  - Implemented idea: fuzzing whole FTP session
    (set of commands in file separated by newline)

- **Lessons learned**
  - Fuzzed server generated huuuge number of trash files / dirs
    - it was required to isolate fuzzed apps creating files
  - Fuzzed server slowed down while number of files were growing
    - it was required to regularly clean sandbox directory
  - Next FTP command generated new path – fuzzing can't finish

```
1  UHELP ERuseddYCT
2  TYPE aSOHstringUDmeXSKD paghP
3  XCXMKDa□nMDHon value
4  USER tr□ng
5  RM;NELame
6  CDPWDUL P
7  XNOO`+UP
8  RM& □□lnaSOHstri□gKDvalueSKD XCUPn□L pathname
9  1□□□PSV@□□EOT□□□□□□□□□LOT
10 r□□\n
11 SIZE nathnGmtV□tad
12 SMODETRUpm
13 nEkT  o^PRTvhluc
14 SITE□□□ngDLEDLEe
15 DLETE□
16     ng
17 cMCEOT□□□□□□NELP
18 HEDSOH strin□PWD
```

# American Fuzzy Lop – Excercises (part 5)

You can choose from below optional excercises or experiment on your own:

- Fuzzing with Sanitizers (Address / Leak / Memory)
- Using LD_PRELOAD / AFL_PRELOAD to fuzz strange applications
- Fuzzing network servers and clients
  - using afl for network fuzzing
  - using desock from Preeny
- Using afl sister projects:
  - Pythia
  - Crashwalk + exploitable
  - Python-afl
  - afl-monitor
- Experiments with:
  - Persistent mode
  - LAF LLVM Passes
  - Crash exploration mode (Peruvian rabbit)

# American Fuzzy Lop – Excercises (part 6) – AQL cont'd

1. Run *make fuzz-asan* (Observe the exec speed in comparison to *make fuzz*)
2. Run the crash using both binaries:
   *test_aql.exe* (without ASAN), *test_aql_asan.exe* (with ASAN)
3. Observe differences in results

4. Gather crashes in the same directory and run crashwalk + exploitable

5. Use *make peruvian-rabbit* to run crash exploration mode

6. Set flag USE_PERSISTENT and run *make* to experiment with persistent mode

7. Analyze Makefile options (init-)parallel-fuzz and run_fuzz.sh script prepared for running afl on multiple CPUs

# AFL with Qemu


Qemu logo, https://wiki.qemu.org/Logo

Fuzzing prebuilt, (possibly) closed-source binaries

- AFL can work with Qemu **user emulation mode**
- Fuzzing binaries without porting

Couple of gotcha's

- It's emulation - memory limit set high
- Performance drop (according to documentation 2x-5x drop)
- Libraries (that you want to analyze) must be staticaly linked
- You won't get information about exact line in which crash happened
- Watch out for malicious binaries – they can interact with your fuzzing system

Usage:

- Afl-fuzz with –Q option, possibly set –m to none or higher

# AFL with Unicorn Engine

Case:

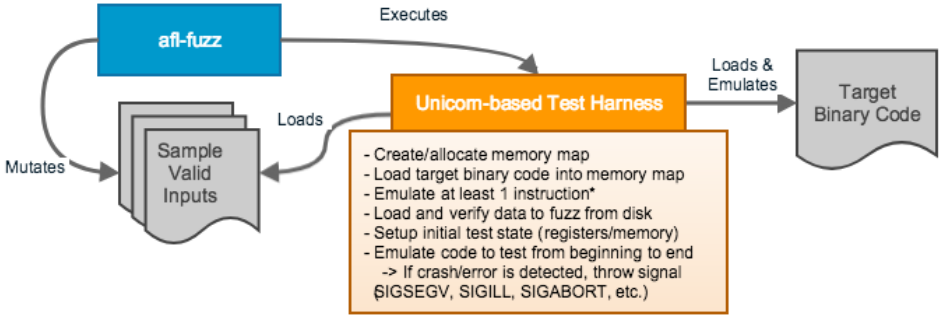
- Obtained binary compiled for other architecture that you work on (e.g. ARM)
  - Optional but preffered: able to debug binary running on the device
- Unable to fuzz it on device (insufficient power)
- Interesting code is tightly coupled with difficult to emulate functions/features

Unicorn engine logo, source: https://www.unicorn-engine.org/

AFL-Unicorn

- Fork of AFL created by Nathan Voss
- Utilizes Unicorn engine to emulate code (https://www.unicorn-engine.org/)
  - Can emulate code built for „Arm, Arm64 (Armv8), M68K, Mips, Sparc, & X86 (include X86_64)” (based on qemu)
  - Has C and Python bindings

# AFL with Unicorn Engine




Source: Nathan Voss, **afl-unicorn: Fuzzing Arbitrary Binary Code,** https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf

Advantages over QEMU mode:

- Can emulate parts of the code (single functions) without complicated setup

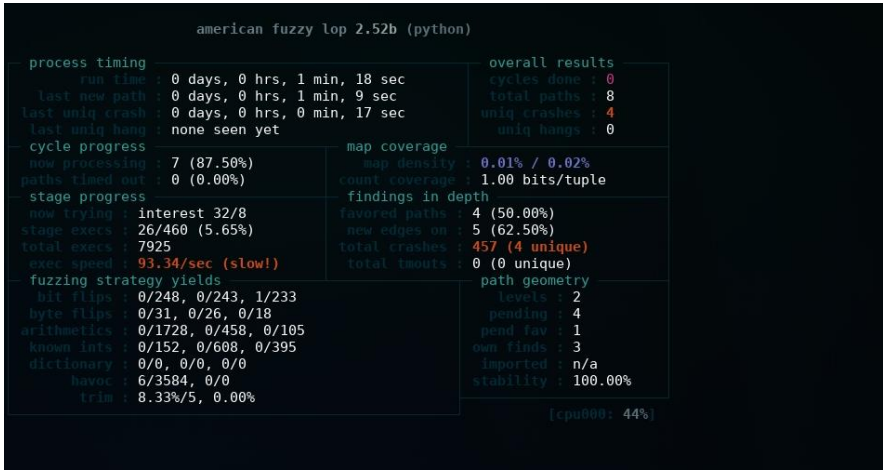
Problems to solve:

- Ommiting parts of the execution that cause crash because of the emulated environment (e.g. calls to kernel)
  - (Optional) Capturing interesting function execution context and data – very useful
- Speed ☹ (at least it scales up with additional cores or with C based test harness)

# AFL Unicorn – provided example

- simple_target.c – provided buggy code
- simple_target.bin – binary built for MIPS architecture
- simple_test_harness.py – test harness with setup (Python bindings) – invokes simple_target.bin in Unicorn engine
- sample_inputs

```
./afl-fuzz -U -i unicorn_mode/samples/simple/sample_inputs \
-o unicorn_mode/samples/simple/sample_output \
-- python unicorn_mode/samples/simple/simple_test_harness.py @@
```



# American Fuzzy Lop – Additional information

- **Project afl-training**
  - Interesting fuzzing tasks (e.g. Heartbleed vuln in OpenSSL)
  - GitHub: ThalesIgnite -> afl-training

- **afl mailing list**
  - https://groups.google.com/forum/#!forum/afl-users

- **Author's (lcamtuf) blog**
  - https://lcamtuf.blogspot.com/

# Dziękujemy!

# Thank You!

SAMSUNG