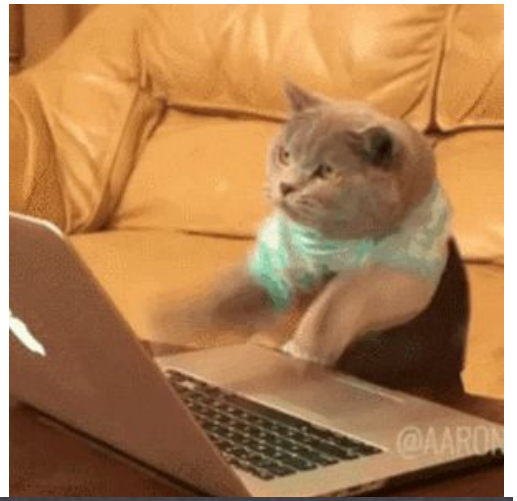
Security Meetup

Newbie fuzzing gains:
Getting the most of your iterations

Presented by @antonio-morales

Dumb fuzzing

- Random mutations and run cases
- No iterations
- Bruteforce



```
$> while true;\
 > do [tested_program] `head -c 100 /dev/urandom;\
 > done
```

“Coverage-guided”

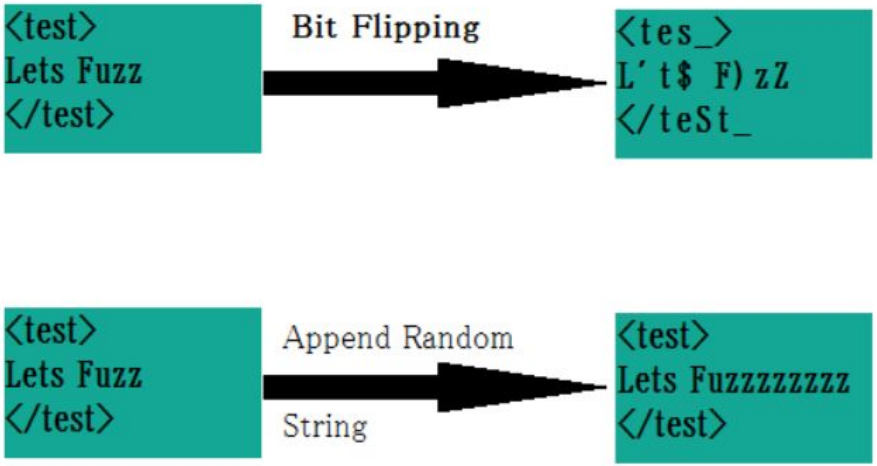
“Evolutionary”

“Mutation-based”

FUZZERS

Mutation-based fuzzer

- A technique that generates fuzz inputs by applying small mutations to existing valid inputs
- No knowledge of the structure of the input is assumed



AFL deterministic mutations

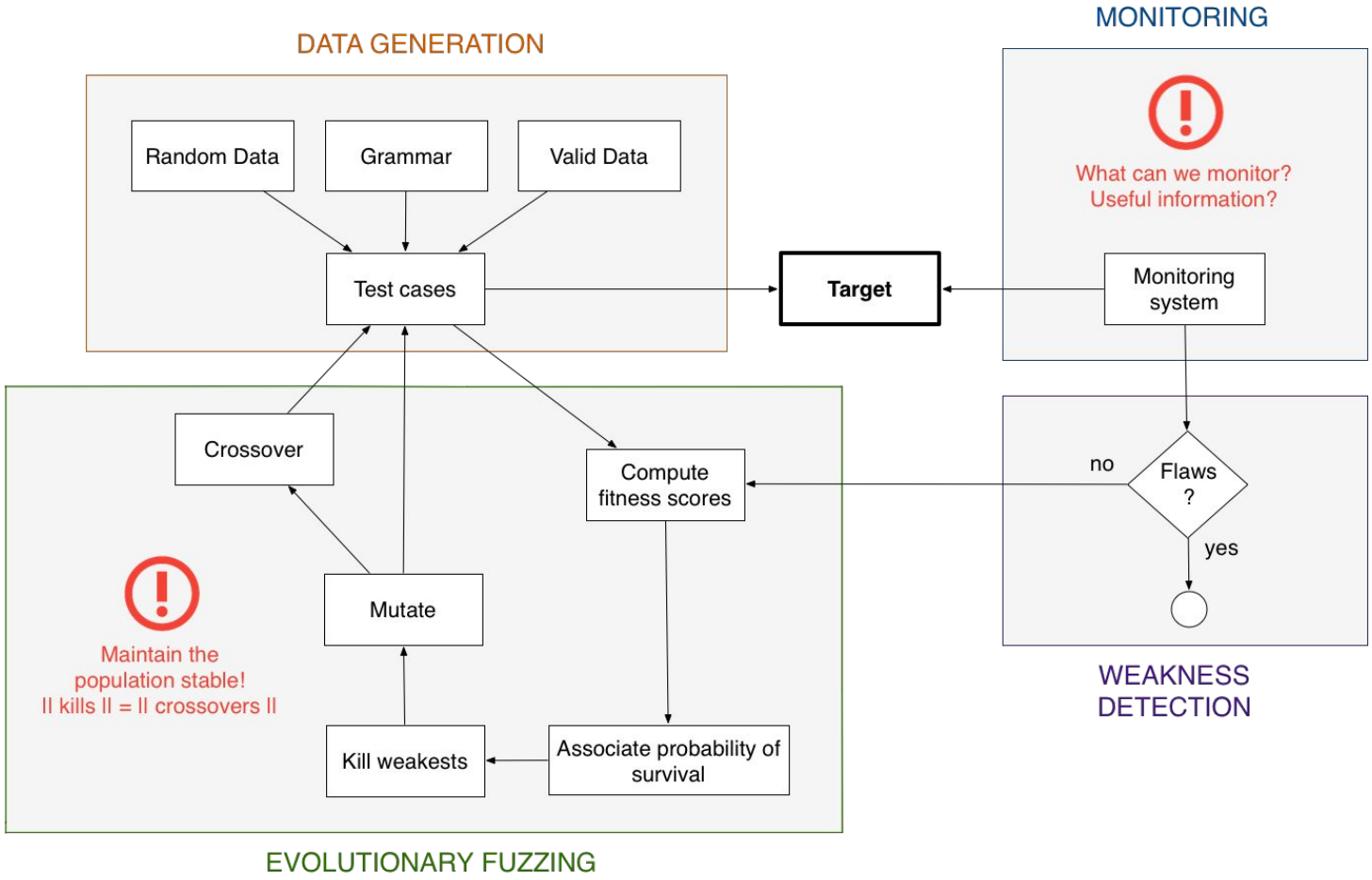
For every byte of the input file:

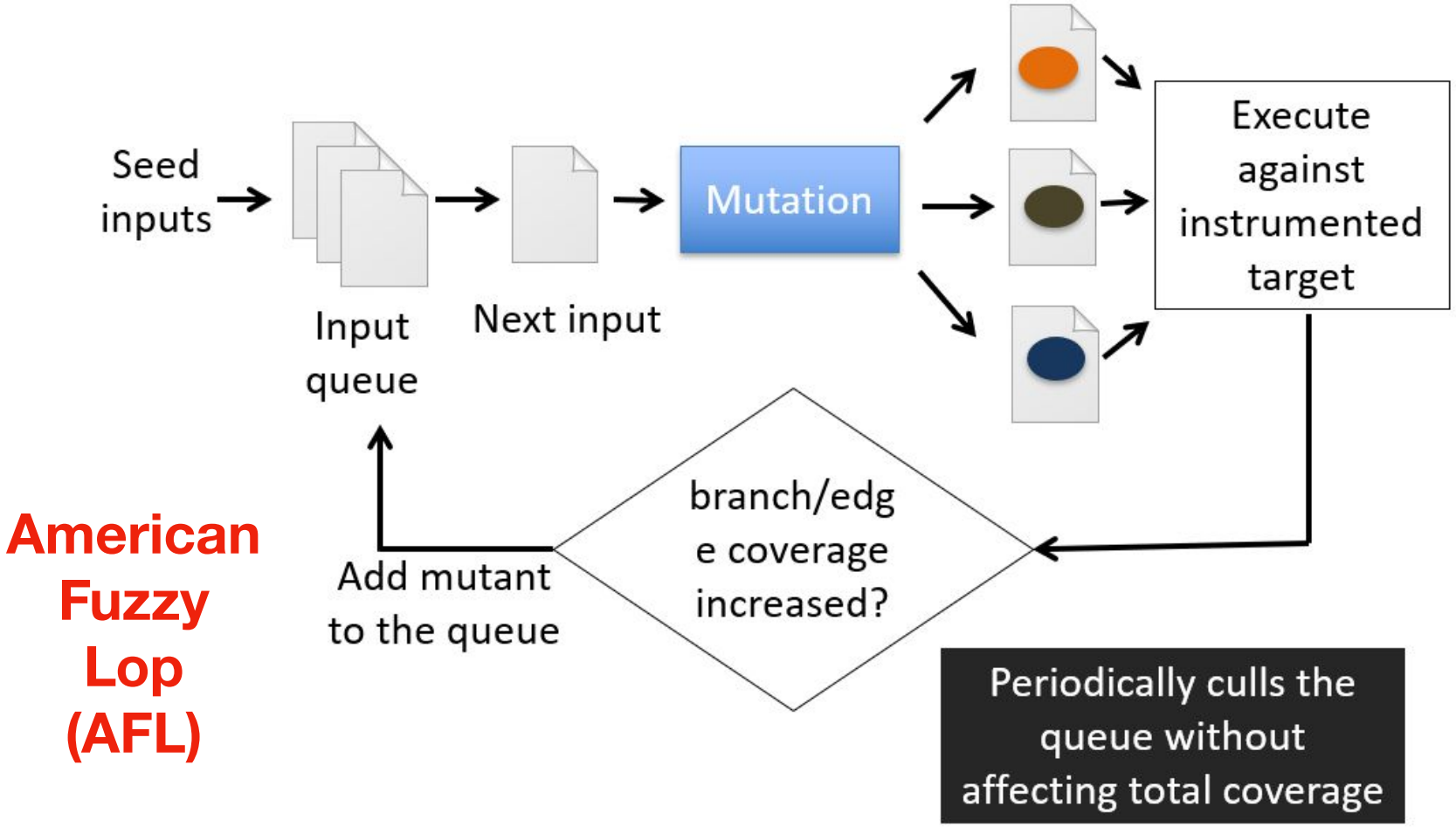
- **Bit flipping**: Sequential and ordered 1,2,4,8,16,32 bit flips.
- **Simple Arithmetics**: increments and decrements of existing integer values (8,16,32 bits integers).
- **Known integers**: Hardcoded set of integer edge cases (-1, 256, 1024, MAX_INT-1, etc).

AFL randomized mutations

- **Block deletion**
- **Block duplication** (overwrite or insertion)
- **Block memset**

- **Test case splicing**: involves taking two different input files from the queue that differs in at least two locations and splicing them at a random location in the middle.


DATA GENERATION
Random Data
Grammar
Valid Data
Test cases
Target
MONITORING
What can we monitor?
Useful information?
Monitoring system
Crossover
Compute fitness scores
Mutate
Maintain the population stable!
II kills II = II crossovers II
Kill weakests
Associate probability of survival
EVOLUTIONARY FUZZING
no
Flaws ?
yes
WEAKNESS DETECTION



American Fuzzy Lop (AFL)

Coverage measurement

Gcov & Lcov:
Statement/Node coverage

- **Gcov** is a test coverage program that is provided with GCC.
- Gcov analyses the **number of time each line of program** is executed during a run.
- We may then talk of **statement/node coverage.**
- **Lcov** is a graphical front-end for GCC's coverage testing tool gcov.

LCOV - code coverage report

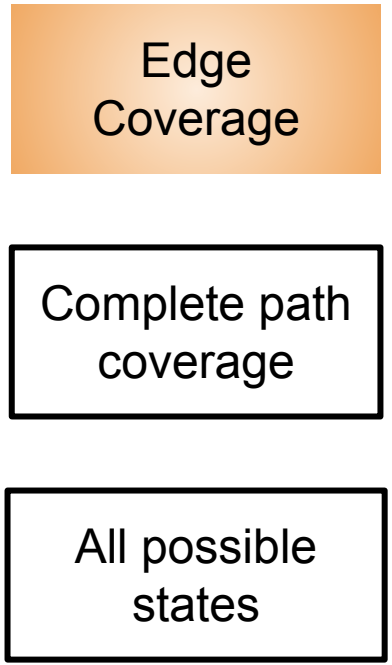
				Hit	Total	Coverage
Current view:	top level - /mnt/ex/pseint/pseint		Lines:	2686	3587	74.9 %
Test:	ads_test.info		Functions:	199	268	74.3 %
Date:	2016-03-30 16:15:30					

Filename	Line Coverage			Functions	
Ejecutar.cpp		96.9 %	440 / 454	100.0 %	3 / 3
LangSettings.cpp		35.8 %	24 / 67	33.3 %	3 / 9
LangSettings.h		88.6 %	31 / 35	90.0 %	9 / 10
SynCheck.cpp		92.0 %	970 / 1054	90.9 %	20 / 22
case_map.cpp		1.5 %	1 / 67	33.3 %	2 / 6
global.cpp		100.0 %	3 / 3	100.0 %	2 / 2
intercambio.cpp		25.1 %	51 / 203	53.8 %	14 / 26
intercambio.h		100.0 %	5 / 5	85.7 %	6 / 7
main.cpp		46.4 %	115 / 248	80.0 %	4 / 5
new_evaluar.cpp		85.1 %	428 / 503	100.0 %	16 / 16
new_funciones.cpp		68.0 %	123 / 181	61.3 %	19 / 31
new_funciones.h		100.0 %	23 / 23	100.0 %	9 / 9
new_memoria.cpp		100.0 %	15 / 15	100.0 %	3 / 3
new_memoria.h		89.2 %	157 / 176	97.1 %	34 / 35
new_programa.cpp		100.0 %	1 / 1	100.0 %	2 / 2
new_programa.h		92.3 %	72 / 78	91.3 %	21 / 23
utils.cpp		58.8 %	163 / 277	78.3 %	18 / 23

More than node coverage

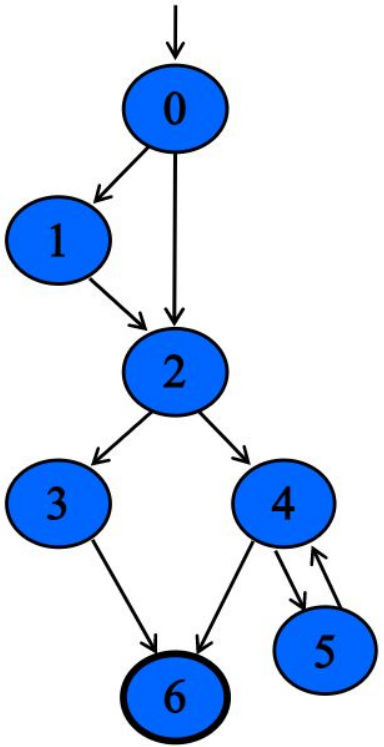
There are other coverage criterias that can be used:

- **Edge coverage**:
  - Has every edge in the CFG been executed?
  - LibFuzzer (native LLVM SanitizerCoverage)
  - AFL (stored in 64kb bitmap)
- Complete path coverage
- All possible states



Edge coverage

Coverage = { (0,1), (0,2), (1,2),
(2,3), (2,4), (3,6), (4,5), (4,6), (5,4) }



More than node coverage

There are other coverage criterias that can be used:

- Edge coverage:
- **Complete path coverage:**
  - Require that all possible execution paths are covered
  - It is not feasible if the graph has a loop
- All possible states

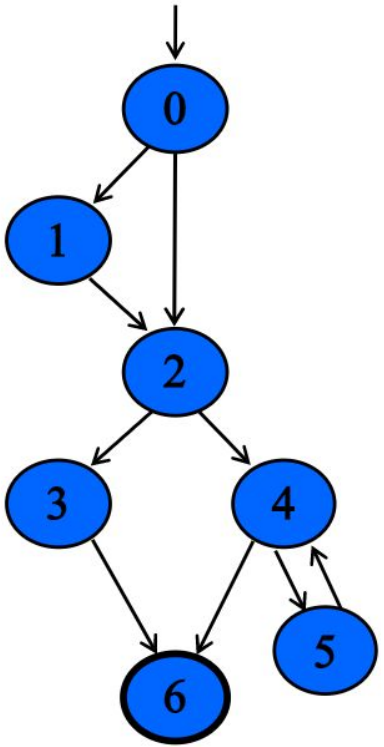
Edge Coverage

Complete path coverage

All possible states

Complete path coverage

Coverage = { (0,1,2,3,6),
(0,1,2,4,6), (0,1,2,4,5,4,6),
(0,1,2,4,5,4,5,4,6), ... }



More than node coverage

There are other coverage criterias that can be used:

- Edge coverage:
- Complete path coverage
- **All possible states:**
  - All combinations of variable values has been tested
  - Each memory state has been reached

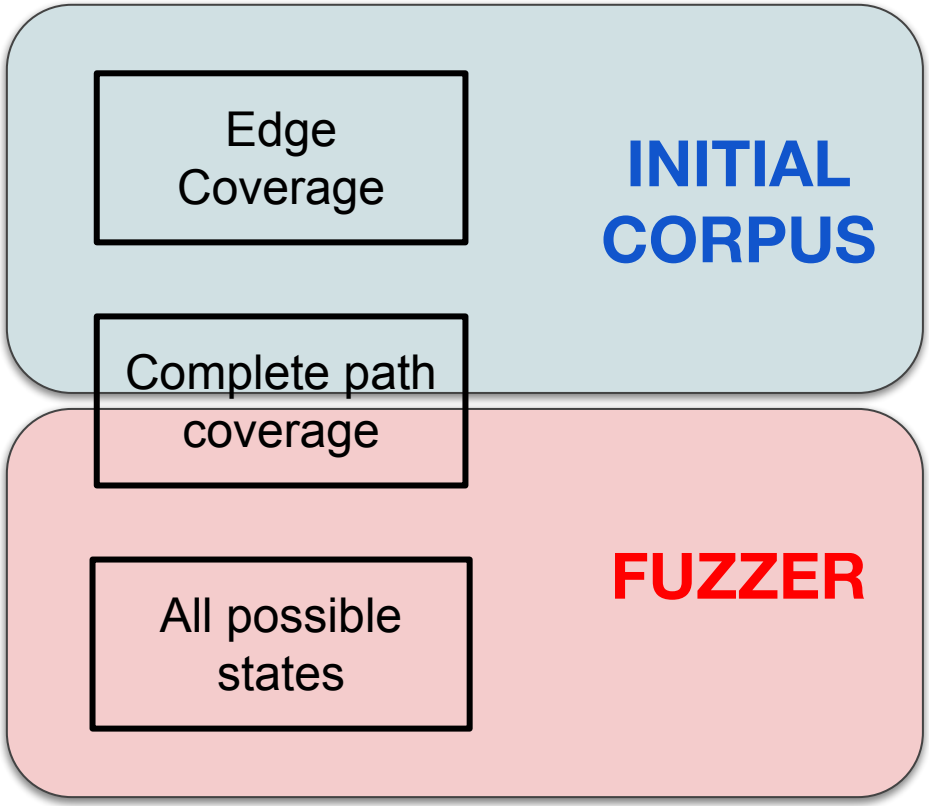
Edge Coverage

Complete path coverage

All possible states

Ideal starting scenario

- Ideally the initial corpus should cover **all code lines** and a **reasonable amount of execution paths** (all functionalities of the program).
- The task of the fuzzer is to find **“Weird paths”** and **unexpected program states.**
- We should also use a **minimal corpus size** and **input size.**

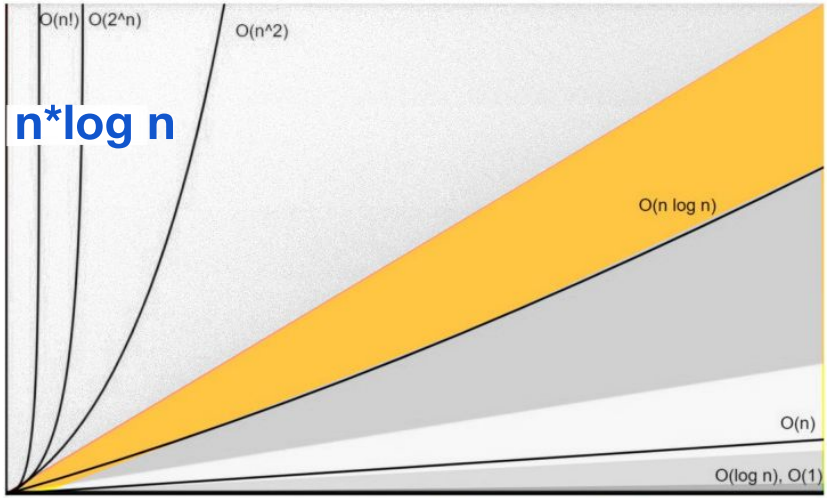


Size matters

- The **number of mutations** on each stage is correlated with the **size of current input file** (bit/byte mutations)

- Bigger files are usually **more time-consuming**, due to program will have to process more information.

Name	Size	Kind
HB50 cupcakes.JPG	2 MB	JPEG image
Roller Skating.JPG	1.3 MB	JPEG image
50HBJukebox2.jpg	720 KB	JPEG image
Facebook.tiff	399 KB	TIFF image
7_days_to_enrol.png	173 KB	PNG image
JoggingShoes.jpg	71 KB	JPEG image



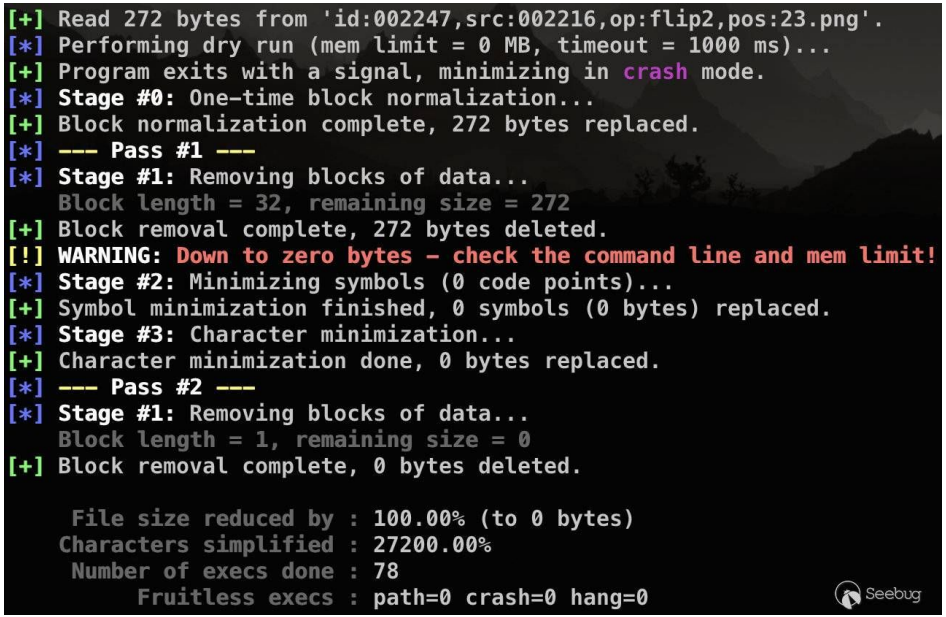
Input size minimization

- In the context of a single test case, the following should be maximized:

$$\frac{|program\ states\ explored|}{input\ size}$$

- We should reach the highest byte-to-coverage feature radio

- AFL: **afl-tmin** tool



Corpus minimization

- Minimized initial corpus is other of the key factors during fuzzing process.
- It should be verified that edge coverage is not reduced (and execution paths!!)
- Libfuzzer: “**merge=1**” option
- AFL: **afl-cmin** tool

```
$ afl-cmin -i Trimmed/ -o Pcap-corpus/ -- tcpdump -ee -vv -nnr @@
corpus minimization tool for afl-fuzz by <lcamtuf@google.com>

[*] Testing the target binary...
[+] OK, 395 tuples recorded.
[*] Obtaining traces for input files in 'Trimmed/'...
    Processing file 1514745/1514745...
[*] Sorting trace sets (this may take a while)...
[+] Found 9566 unique tuples across 1514745 files.
[*] Finding best candidates for each tuple...
    Processing file 1514745/1514745...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
    Processing tuple 9566/9566...
[+] Narrowed down to 273 files, saved in 'Pcap-corpus/'.
```

Overcoming obstacles

“Stuck” fuzzer

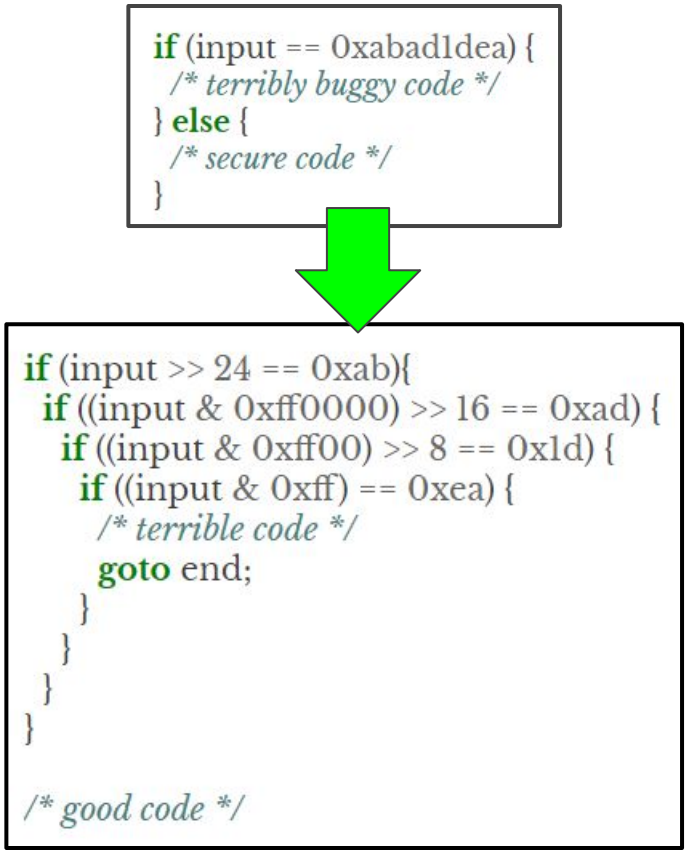
- In this example, we can see that the buggy code is only executed if the variable holds the value “0xabad1dea”.
- This in turn is very unlikely because the value in input is mainly generated by the fuzzer performing byte mutations.
- At this point, the fuzzer needs some help to surpass the conditional statement.

```
if (input == 0xabad1dea) {
  /* terribly buggy code */
} else {
  /* secure code */
}
```

Splitting up comparisons

- To overcome this issue we can **split up comparisons into smaller ones** which should guide the fuzzer towards the correct value.
- Now, as soon as the fuzzer guesses the first byte correctly it will execute the nested-if condition and therefore **discover a new path.**
- This will repeat for the other if-statements.

https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/



VLC vulnerabilities discovered by the Semmle security research team

By: **Semmle Team**
August 19, 2019

Category	Technical Difficulty	Reading time
Security	easy	6 min

Today, the VideoLAN team announced a new release of VLC, fixing 11 vulnerabilities reported by Antonio Morales Maldonado from the Semmle security research team.

MITRE has issued the following CVE IDs for the vulnerabilities: CVE-2019-14437, CVE-2019-14438, CVE-2019-14438, CVE-2019-14498, CVE-2019-14535, CVE-2019-14534, CVE-2019-14533, CVE-2019-14776, CVE-2019-14778, CVE-2019-14779, CVE-2019-14777, CVE-2019-14970.

About VLC

The VLC Media Player (commonly known as just **VLC**) is a popular media player developed by the VideoLAN project. VLC is available on most platforms (Windows, MacOS, Linux, Android, iOS

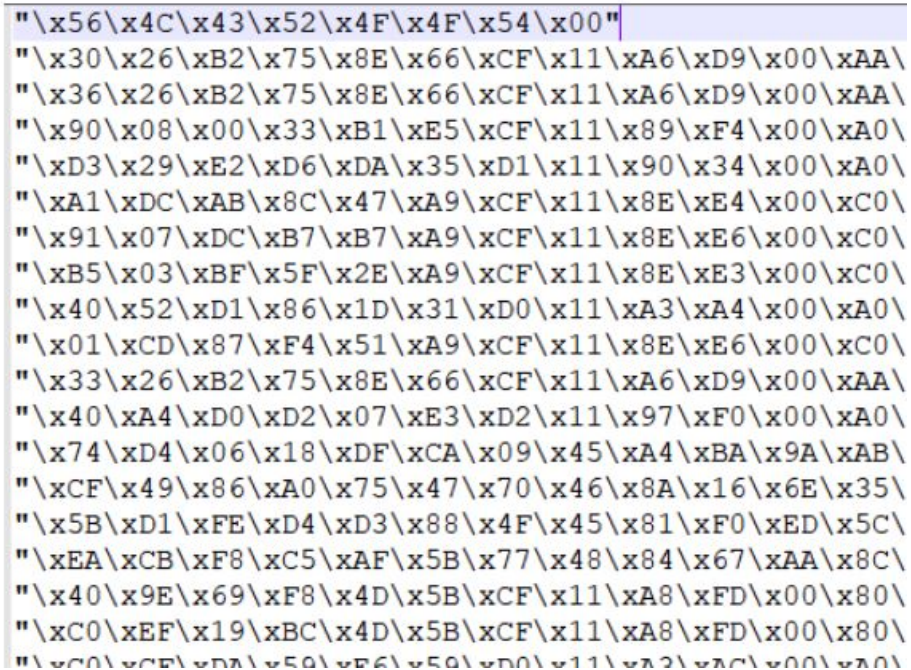
github.com/videolan/vlc/blob/master/modules/demux/asf/libasf_guid.h

```
/* Top-Level object */
static const vlc_guid_t asf_object_header_guid =
{0x75B22630, 0x668E, 0x11CF, {0xA6, 0xD9, 0x00, 0xAA, 0x00, 0x62, 0xCE, 0x6C}};

static const vlc_guid_t asf_object_data_guid =
{0x75B22636, 0x668E, 0x11CF, {0xA6, 0xD9, 0x00, 0xAA, 0x00, 0x62, 0xCE, 0x6C}};

static const vlc_guid_t asf_object_simple_index_guid =
{0x33000890, 0xE5B1, 0x11CF, {0x89, 0xF4, 0x00, 0xA0, 0xC9, 0x03, 0x49, 0xCB}};

static const vlc_guid_t asf_object_index_guid =
{0xD6E229D3, 0x35DA, 0x11D1, {0x90, 0x34, 0x00, 0xA0, 0xC9, 0x03, 0x49, 0xBE}};

/* Header object */
static const vlc_guid_t asf_object_file_properties_guid =
{0x8cabdca1, 0xa947, 0x11cf, {0x8e, 0xe4, 0x00, 0xC0, 0x0C, 0x20, 0x53, 0x65}};

static const vlc_guid_t asf_object_stream_properties_guid =
{0xB7DC0791, 0xA9B7, 0x11CF, {0x8E, 0xE6, 0x00, 0xC0, 0x0C, 0x20, 0x53, 0x65}};

static const vlc_guid_t asf_object_header_extension_guid =
{0x5FBF03B5, 0xA92E, 0x11CF, {0x8E, 0xE3, 0x00, 0xC0, 0x0C, 0x20, 0x53, 0x65}};

static const vlc_guid_t asf_object_codec_list_guid =
{0x86D15240, 0x311D, 0x11D0, {0xA3, 0xA4, 0x00, 0xA0, 0xC9, 0x03, 0x48, 0xF6}};
```

Providing a custom dictionary

- Custom dictionaries can be added in order to provide the fuzzer with a list of complex syntax tokens.

- Even when no explicit dictionary is given, afl-fuzz will try to extract existing syntax tokens in the input corpus by watching the instrumentation very closely during deterministic byte flips



CVE-2019-14533

- **Advanced Systems Format (ASF)** is Microsoft's proprietary audio/video container, best known for their most common media types: **WMV** and **WMA.**

- Through the use of a custom dictionary including ASF Object GUIDs, an **Use-After-Free vulnerability** was found in VLC media player. As a result, trying to parse an invalid WMV/ASF (Windows Media Video) file will result in UAF when the video is forwarded.

- This bug could allow an attacker to alter the expected application flow.

CVE-2019-14533

```
static void DemuxEnd( demux_t *p_demux )
{
    demux_sys_t *p_sys = p_demux->p_sys;

    if( p_sys->p_root )
    {
        ASF_FreeObjectRoot( p_demux->s, p_sys->p_root );
        p_sys->p_root = NULL;
        //p_sys->p_fp should also be nulled
    }

    [...]
}
```

Dealing with checksums

- Some protocols or file formats incorporate checksums that will fail if they're modified arbitrarily.
- There are 2 strategies to counter this:
  - **Re-calculate** the checksum on the fuzzed inputs
  - **Patch** the software to disable checksum tests and repair the checksum fields in malformed inputs

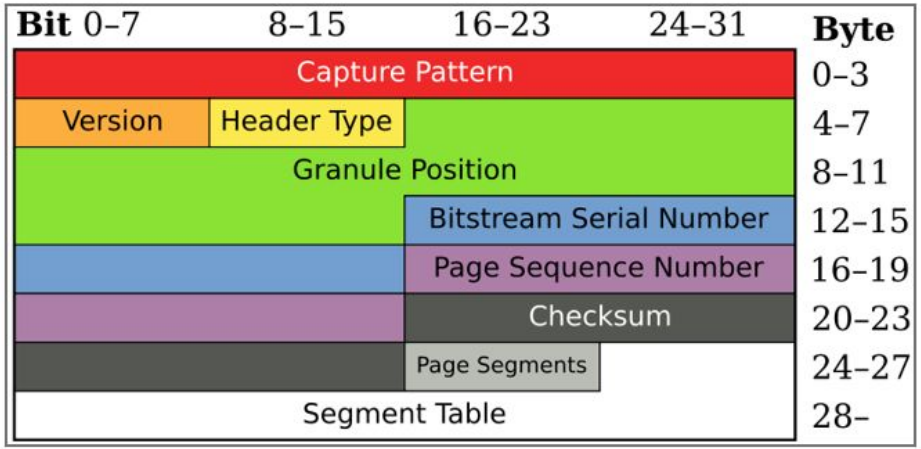
PNG chunk specification.

Name	Size	Description
Length	4 bytes	Length of data field
Type	4 bytes	Chunk type code
Data	n bytes	Data bytes
CRC	4 bytes	CRC of type and data

CVE-2019-14438

- **Ogg** is a free, open container format that can multiplex a number of independent streams for audio, video, subtitles and metadata.

- OGG page header includes a **CRC32 checksum**, that inthe case of VLC, is calculated through **Libogg library**.



CVE-2019-14438

- After disabling CRC32 checksum we were able to find an **OOB write** in “xiph_PackHeaders” function.

- As a result is possible to craft the heap through a OOB read/write attack.

```
/* Compare */
if(memcmp(chksum,page+22,4)){
  /* D'oh.  Mismatch! Corrupt page (or miscapture and not a page
     at all) */
  /* replace the computed checksum with the one actually read in */
  memcpy(page+22,chksum,4);
```

```
for (unsigned i = 0; i < packet_count; i++) {
    if (packet_size[i] > 0) {
        memcpy(current, packet[i], packet_size[i]);
        current += packet_size[i];
    }
}
```

Is just the beginning

Is just the beginning

CVE	Type
CVE-2019-14437	OOB read
CVE-2019-14438	OOB write
CVE-2019-14498	Divide-by-zero
CVE-2019-14535	Divide-by-zero
CVE-2019-14534	NULL Pointer Dereference
CVE-2019-14533	Use-After-Free
CVE-2019-14776	OOB read
CVE-2019-14778	Use-After-Free
CVE-2019-14779	OOB Read
CVE-2019-14777	Use-After-Free
CVE-2019-14970	OOB Write

To be continued...

- In-process fuzzing (persistent mode)
- Custom mutators (Structure-Aware Fuzzing)
- Power Schedules
- Mutation Scheduling

Q & A