AFL's Blindspot and How to Resist AFL Fuzzing for Arbitrary ELF Binaries

Kang Li
360 Cyber Immunity Lab / Team Disekt / University of Georgia

Collaborators: Yue Yin, Guodong Zhu

#BHUSA / @BLACKHAT Events
About Me

Professor of Computer Science at UGA

Founding Mentor of xCTF and Blue-Lotus

Founder of the Disekt, SecDawgs CTF Teams

2016 DARPA Cyber Grand Challenge Finalist
- Write a simple buggy program
- Assign the binary (without symbols) and expect students to find bugs
- “Rest” until students finish (usually takes hours …)
```c
7 int cb(uchar *out) {
8     int ret = 0;

9     if (out[0] == 'M') {
10        if (out[1] == 'A') {
11            if (out[2] == 'G') {
12                if (out[3] == 'I') {
13                    if (out[4] == 'C') {
14                        if (out[5] == '!') {
15                            ret = 1;
16                            /* printf("You Won!\n"); */
17                            crash();
18                        }
19                    }
20                }
21            }
22        }
23     }
24
25     /* printf("Please Try Again\n"); */
26     return 0;
27 }
```
### American Fuzzy Lop (AFL) Results

**Process Timing**
- Run time: 0 days, 0 hrs, 3 min, 51 sec
- Last new path: 0 days, 0 hrs, 1 min, 7 sec
- Last uniq crash: 0 days, 0 hrs, 0 min, 6 sec
- Last uniq hang: None seen yet

**Cycle Progress**
- Now processing: 5 (71.43%)
- Paths timed out: 0 (0.00%)

**Stage Progress**
- Now trying: havoc
- Stage execs: 2028/2048 (99.02%)
- Total execs: 336k
- Exec speed: 1322/sec

**Fuzzing Strategy Yields**
- Bit flips: 0/320, 0/313, 0/299
- Byte flips: 0/40, 0/33, 0/19
- Arithmetics: 2/2237, 0/197, 0/70
- Known ints: 0/244, 0/880, 0/824
- Dictionary: 0/0, 0/0, 0/0
- Havoc: 3/141k, 2/187k
- Trim: 0.00%/6, 0.00%

**Overall Results**
- Cycles done: 65
- Total paths: 7
- Uniq crashes: 1
- Uniq hangs: 0

**Map Coverage**
- Map density: 0.05% / 0.08%
- Count coverage: 1.00 bits/tuple

**Findings in Depth**
- Favored paths: 7 (100.00%)
- New edges on: 7 (100.00%)
- Total crashes: 1 (1 unique)
- Total tmsgs: 0 (0 unique)

**Path Geometry**
- Levels: 6
- Pending: 0
- Pending fav: 0
- Own finds: 6
- Import: n/a
- Stability: 100.00%
• Fast and Reliable Fuzzing
  edge coverage stored in a compact bitmap (default 64KB)
  low test overhead, simple to use

• Bugs Found in
  Bind, PuTTY, tcpdump, ffmpeg, GnuTLS, libtiff, libpng, ...
  more on the AFL sites (http://lcamtuf.coredump.cx/afl/)

• Widely Used
  by most of the 2016 CGC Finalist Teams
Why to Resist AFL fuzzing

• The deafL tool (this talk)
  • to force students to study binaries (instead of just running AFL)

• Other reasons:
  • to learn AFL’s limitations and to develop better fuzzers

... ...
The Fuzzing Process of AFL

1. Start with sample seed inputs

2. Mutate seed inputs to generate mutants

3. Collect code coverage (CFG edges) Information

4. Save as new seeds if coverage increases

5. Repeat from step 2
AFL Instrumentation

- **if with Source Code** (Compiler-aid Instrumentation, AFL-GCC)
  1. `cur_location = <RANDOM#>;`
  2. `shared_mem[cur_location ^ prev_location]++ ;`
  3. `prev_location = cur_location >> 1;`

- **if with Binary Only** (AFL-QEMU)
  1. `cur_location = (block_address >> 4) ^ (block_address << 8);`
  2. `shared_mem[cur_location ^ prev_location]++ ;`
  3. `prev_location = cur_location >> 1;`
$ readelf testcase_1

Assuming the basic blocks being covered are:

...  
0x428DB2  
0x428E10  
0x428DED  
...

Program readelf’s Control Flow Graph (partial)
$ readelf testcase_1

Assuming the basic blocks being covered are:
...
0x428DB2
0x428E10
0x428DED
...

Program readelf’s Control Flow Graph (partial)
How Coverage Info is Collected in AFL

Assuming the basic blocks being covered are:

```
$ readelf testcase_1

0x428DB2
0x428E10
0x428DED
```

Program `readelf`'s Control Flow Graph (partial)
How Coverage Info is Collected in AFL

$ readelf testcase_1

Assuming the basic blocks being covered are:

... 0x428DB2 0x428E10 0x428DED ...

Program readelf’s Control Flow Graph (partial)
How Coverage Info is Collected in AFL

$ readelf testcase_1

Assuming the basic blocks being covered are:

... 0x428DB2 0x428E10 0x428DED ...

New Coverage Information! testcase_1 saved in afl/queue

Program readelf's Control Flow Graph (partial)
How Coverage Info is Collected in AFL

$ readelf testcase_N

if shared_mem[] is marked with new updates — find an input with a “new interest path”

...
$ readelf testcase_X

Basic blocs being covered:
...
0x428E70
0x428E28
...

If no new updates in shared_mem[], AFL considers no new edges.
$ readelf testcase_X

Basic blocs being covered:

...  
0x428E70  
0x428E28  
...

If no new updates in shared_mem[], AFL considers no new edges.

AFL fails to detect a new path, **testcase_X discarded!**

AFL’s shared_mem[]

Hash conflict occur!

Program *readelf*'s Control Flow Graph (partial)
When we combine symbolic execution with AFL, we found AFL refuses to sync several inputs generated by our symbolic execution engine. Two pairs of conflict edges are shown below.

```
hash(0x41c9b0, 0x41c9d1) = 0x9bd0
hash(0x4189dc, 0x4189e1) = 0x9bd0
hash(0x419509, 0x41951c) = 0xd79
hash(0x419509, 0x41951c) = 0x9bd0
```

AFL’s `shared_mem[]`
What about Using Large Bitmap Sizes?

Bitmap Sizes vs. AFL Speed


1. Large bitmap sizes reduce but do not eliminate hash conflicts.

2. Speed degrades significantly after bitmap size gets larger (than CPU mem cache size)
How to Resist AFL Fuzzing

• Add Complex Path Constraints
e.g.  `if (input * input = long_int_value)`

• Add Delays for Known Invalid Inputs
e.g. insert `sleep()` call to slow down AFL execution

• Add Nondeterministic Events
e.g. dynamic code relocation

Usually Need Source Code
How to Resist AFL Fuzzing

- Add **Complex** Path Constraints
  e.g. `if (input * input = long_int_value)`

- Add Delays for **Known Invalid** Inputs
  e.g. insert sleep() call to slow down AFL execution

- Add **Nondeterministic** Events
  e.g. dynamic code relocation

- **Disturb AFL’s Seed Selection** ← (this talk)

  Reducing AFL’s ability to finding new paths by introducing fake edges to cause hash conflicts

  Target at the AFL-QEMU mode

  Resist through binary rewriting

**Usually Need Source Code**

**Without Source Code**
General Idea of deafL

- Suppress AFL’s ability to mutate seeds and trigger crashes
- The deafL tool — Inject dummy code to a binary to create conflicting hash values to those edges leading toward crashes
The deafL tool needs to provide answers to these 3 questions

• Which edges to target (to create hash conflicts)?

• How to create an edge that has a specific hash value?

• How to inject fake edges to a binary?
Which Edges to Target (for Hash Conflict)?

- A naïve solution:
  - Add fake edges to completely fill AFL’s share_mem[
  - Binaries become too fat and run very slow!

- Need to target only a small set of edges
  - Idea: find those edges that lead to the mutation of crash inputs
Which Edges to Target (for Hash Conflict)?

- Current Approach
  - Run AFL first to find crashes
  - Find those inputs that mutate to crashes (call them *targeted seed files*)
  - Find all edges that link between the initial seed inputs and the targeted seed files
Find Target Edges (example)

• Start from an AFL crash file
  crashes/id:000000,sig:11,src:000179+000048,op:splice,rep:2

• Find its parents (where it mutates from)
  queue/id:000179,src:000121+000178,op:splice,rep:4,+cov
  queue/id:000048,src:000000,op:havoc,rep:8
Find Target Edges (example)

- Start from an AFL crash file
  
  crashes/id:000000,sig:11,src:000179+000048,op:splice,rep:2

- Find its parents (where it mutates from)
  
  queue/id:000179,src:000121+000178,op:splice,rep:4,+cov
  queue/id:000048,src:000000,op:havoc,rep:8

- Find all code edges that covered by these parent inputs but not by the initial seed
  
  queue/id:0000:initial_seed_input

---

Sample output of finding target edges

```
[id:000179,src:000121+000178,op:splice,rep:4,+cov]
introduced [9] new edges:
[0x43f032, 0x43f06f] at index [0x5687]
[0x43f06f, 0x43dc22] at index [0x37c1]
[0x4a418e, 0x4a41c7] at index [0x7610]
[0x4a41c7, 0x4a41ea] at index [0x7f90]
[0x4a431f, 0x4a4331] at index [0xc8ab]
[0x4a4331, 0x4a4386] at index [0x68a1]
[0x4a7033, 0x4a7039] at index [0xd402]
[0x4a7039, 0x4a7058] at index [0xb004]
[0x4a7058, 0x4a7070] at index [0xa885]
... ...
```
• Use a `cmp-jne` snippet to fake an edge

• for a given “targeted edge”: 
  `[ blk_A_addr, blk_B_addr ]`

• Assuming we have a starting address to insert code (known `prev_location`), calculating a target address so that

\[
prev_location \oplus cur_location = blk_A_addr \oplus blk_B_addr
\]

• Can generate a nested blob of `cmp-jne` snippets for a list of “targeted edges”.

\[
cur_location = (block_address \gg 4) \oplus (block_address \ll 8); 
shared_mem[cur_location \oplus prev_location]++; 
prev_location = cur_location \gg 1;
\]

\[
{prev_location}:
  cmp \%rsp 0x0
{prev_location}+4:  jne \{cur_location\}
{prev_location}+11:  nop
{prev_location}+12:  nop
......
{cur_location}:
  nop
\]
Injecting Edges with Hash Conflicts

• Code Injection Overview

• Build on the python lief package

• Modify entrypoint to points to inserted code (to fake edges)

• Major changes to code and data
  • Update section table to extend .text size
  • Update all address/offset/data info after the inserted section:
    ○ "dynamic", "rela.dyn", "rela.plt",
      "symtab", "dynsym"
    ○ Pointer references in ".text", ".data",
      ".rodata"

ELF Header
Program Header Table
Section 1
...
.text
Code for fake edges
...
Section N
Section Header Table

entrypoint modified
$ python Deafl.py examples/magic/cb
int cb(uchar *out) {
    int ret = 0;

    if (out[0] == 'M') {
        if (out[1] == 'A') {
            if (out[2] == 'G') {
                if (out[3] == 'I') {
                    if (out[4] == 'C') {
                        if (out[5] == '!') {
                            ret = 1;
                            /* printf("You Won!\n"); */
                            crash();
                        }
                    }
                }
            }
        }
    }

    /* printf("Please Try Again\n"); */
    return 0;
}
Peace Disrupted

Total Paths: 7
uniq Crashes: 1
### American Fuzzy Lop 2.52b (new cb)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 24 min, 37 sec</td>
<td>Cycles done: 555</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 23 min, 46 sec</td>
<td>Total paths: 6</td>
</tr>
<tr>
<td>Last uniq crash: none seen yet</td>
<td>Uniq crashes: 0</td>
</tr>
<tr>
<td>Last uniq hang: none seen yet</td>
<td>Uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing: 4 (66.67%)</td>
<td>Map density: 0.06% / 0.09%</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count coverage: 1.00 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now trying: havoc</td>
<td>Favored paths: 6 (100.00%)</td>
</tr>
<tr>
<td>Stage execs: 108/512 (21.09%)</td>
<td>New edges on: 6 (100.00%)</td>
</tr>
<tr>
<td>Total execs: 2.59M</td>
<td>Total crashes: 0 (0 unique)</td>
</tr>
<tr>
<td>Exec speed: 1820/sec</td>
<td>Total timeouts: 0 (0 unique)</td>
</tr>
</tbody>
</table>

**Fuzzing Strategy Yields**

- Bit flips: 0/256, 0/250, 0/238
- Byte flips: 0/32, 0/26, 0/16
- Arithmetics: 2/1790, 0/52, 0/0
- Known ints: 0/190, 0/724, 0/704
- Dictionary: 0/0, 0/0, 0/0
- Havoc: 2/1.11M, 1/1.47M
- Trim: 0.00%/5, 0.00%

**Previous Result:**
- Total paths: 7
- Uniq crashes: 1
Apply $deafL$ to other binaries
### american fuzzy lop 2.52b (tcpdump)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time</td>
<td>Cycles Done: 0</td>
</tr>
<tr>
<td>Last New Path</td>
<td>Total Paths: 298</td>
</tr>
<tr>
<td>Last Unique Crash</td>
<td>Uniq Crashes: 1</td>
</tr>
<tr>
<td>Last Unique Hang</td>
<td>Uniq Hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now Processing: 27</td>
<td>Map Density: 0.46%</td>
</tr>
<tr>
<td>Paths Timed Out: 0</td>
<td>/ 4.74%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now Trying: arith 8/8</td>
<td>Favored Paths: 190</td>
</tr>
<tr>
<td>Stage Execs: 5628/6629</td>
<td>New Edges On: 236</td>
</tr>
<tr>
<td>Total Execs: 370k</td>
<td>Total Crashes: 1</td>
</tr>
<tr>
<td>Exec Speed: 1062/sec</td>
<td>Total Tmouts: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Flips: 60/11.9k, 38/11.9k, 13/11.8k</td>
<td>Levels: 3</td>
</tr>
<tr>
<td>Byte Flips: 0/1486, 1/1470, 1/1438</td>
<td>Pending: 283</td>
</tr>
<tr>
<td>Arithmetics: 63/76.6k, 1/54.4k, 0/30.7k</td>
<td>Pend Fav: 177</td>
</tr>
<tr>
<td>Known Ints: 6/5353, 4/24.5k, 10/43.9k</td>
<td>Own Finds: 297</td>
</tr>
<tr>
<td>Dictionary: 0/0, 0/0, 0/1312</td>
<td>Imported: n/a</td>
</tr>
<tr>
<td>Havoc: 93/85.2k, 0/0</td>
<td>Stability: 100.00%</td>
</tr>
<tr>
<td>Trim: 19.94%/592, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>

---

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
$ python Deafl.py examples/tcpdump_cve2015-3138/tcpdump
No crash found after more than 4 days

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time: 4 days, 21 hrs, 8 min, 31 sec</td>
<td>cycles done: 22</td>
</tr>
<tr>
<td>last new path: 0 days, 0 hrs, 12 min, 58 sec</td>
<td>total paths: 6245</td>
</tr>
<tr>
<td>last uniq crash: none seen yet</td>
<td>uniq crashes: 0</td>
</tr>
<tr>
<td>last uniq hang: none seen yet</td>
<td>uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing: 2449* (39.22%)</td>
<td>map density: 0.81% / 21.52%</td>
</tr>
<tr>
<td>paths timed out: 0 (0.00%)</td>
<td>count coverage: 2.80 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying: interest 16/8</td>
<td>favored paths: 1937 (31.02%)</td>
</tr>
<tr>
<td>stage execs: 1020/2833 (36.00%)</td>
<td>new edges on: 2528 (40.48%)</td>
</tr>
<tr>
<td>total execs: 201M</td>
<td>total crashes: 0 (0 unique)</td>
</tr>
<tr>
<td>exec speed: 609.2/sec</td>
<td>total tmouts: 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte flips: 40/1.11M, 25/617k, 14/637k</td>
<td>pending: 2259</td>
</tr>
<tr>
<td>arithmetics: 1029/34.0M, 47/29.7M, 11/21.8M</td>
<td>pend fav: 0</td>
</tr>
<tr>
<td>known ints: 181/2.09M, 308/9.33M, 174/17.5M</td>
<td>own finds: 6244</td>
</tr>
<tr>
<td>dictionary: 0/0, 0/0, 354/26.9M</td>
<td>imported: n/a</td>
</tr>
<tr>
<td>havoc: 1988/30.5M, 0/0</td>
<td>stability: 100.00%</td>
</tr>
<tr>
<td>trim: 21.65%/372k, 44.85%</td>
<td></td>
</tr>
</tbody>
</table>

[cpu001: 51%]
$ python Deafl.py examples/objcopy_cve2018-10534/objcopy
With a seed that is similar to the CVE crash input

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time: 0 days, 0 hrs, 1 min, 11 sec</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>last new path: 0 days, 0 hrs, 0 min, 0 sec</td>
<td>total paths: 164</td>
</tr>
<tr>
<td>last uniq crash: 0 days, 0 hrs, 0 min, 46 sec</td>
<td>uniq crashes: 1</td>
</tr>
<tr>
<td>last uniq hang: none seen yet</td>
<td>uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing: 32 (19.51%)</td>
<td>map density: 2.72% / 4.75%</td>
</tr>
<tr>
<td>paths timed out: 0 (0.00%)</td>
<td>count coverage: 1.74 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying: havoc</td>
<td>favored paths: 64 (39.02%)</td>
</tr>
<tr>
<td>stage execs: 2680/6144 (43.62%)</td>
<td>new edges on: 93 (56.71%)</td>
</tr>
<tr>
<td>total execs: 20.6k</td>
<td>total crashes: 1 (1 unique)</td>
</tr>
<tr>
<td>exec speed: 539.6/sec</td>
<td>total tmouts: 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips: n/a, n/a, n/a</td>
<td>levels: 3</td>
</tr>
<tr>
<td>byte flips: n/a, n/a, n/a</td>
<td>pending: 156</td>
</tr>
<tr>
<td>arithmetics: n/a, n/a, n/a</td>
<td>pend fav: 58</td>
</tr>
<tr>
<td>known ints: n/a, n/a, n/a</td>
<td>own finds: 163</td>
</tr>
<tr>
<td>dictionary: n/a, n/a, n/a</td>
<td>imported: n/a</td>
</tr>
<tr>
<td>havoc: 138/11.1k, 20/4712</td>
<td>stability: 100.00%</td>
</tr>
<tr>
<td>trim: 46.79%/825, n/a</td>
<td></td>
</tr>
</tbody>
</table>

CVE 2018-10534
With a seed that is similar to the CVE crash input
• Injected code can be easily identified
  • potentially can be muted by another round of binary rewriting

• Only resist AFL-QEMU
  • may not work with other instrumentation schemes (Intel-PT, PIN, DynamoRIO)

• Only reduce AFL’s ability to explore new paths
  • does not eliminate AFL’s chance to find specific paths
  • no guarantees due to random mutations
• Leverage the Limitation of AFL-QEMU
  • AFL-QEMU only tracks edges in an EFL binary’s 1st code segment
  • Move code to a new code segment to avoid AFL tracking
• Inserting False Termination Signals
  • Abort at normal exit points to generate fake crashes
• AFL’s high efficiency comes from its compact data structure for edge coverage (shared_mem[ ]).

• Hash conflict creates a blindspot for AFL — limits its ability to explore paths.

• The deafL tool — binary rewriting to resist AFL fuzzing.

• Intentionally create hash conflicts for edges that lead to the mutation of crash inputs.
Q&A
kangli.ctf@gmail.com