USING MIASM TO FUZZ BINARIES WITH AFL

@guedou - 22/06/2017 - BeeRumP
WHAT IS AFL?

A smart fuzzer that uses code coverage

- needs an initial corpus
  - ~20 different mutations strategies
  - only keep mutated inputs that modify coverage

- source instrumentation to discover new paths
  - `afl-as` injects ASM after branches, labels, ...
  - uses shm to talk to `afl-fuzz`

- Linux/*BSD only

- as easy to install as typing `make`

See [http://lcamtuf.coredump.cx/afl/](http://lcamtuf.coredump.cx/afl/)
$ cat crash.c
typedef void (*function)();
void crash(char *data) {

   // The magic word is BeeR
   if (data[0] == 'B' && data[1] == 'e' && data[2] == data[1]) {
         printf("ko\n");
         function f = (function) *data;
         f(); // Please crash!
      } else printf("!!\n");
   } else printf("ok\n");
}
A SIMPLE MAIN()

cat test.c
// Typical AFL wrapper
int main() {
    char buffer[BUFFER_SIZE];

    // Clear the buffer content
    memset(buffer, 0, BUFFER_SIZE);

    // Read from stdin
    read(0, buffer, BUFFER_SIZE);

    crash(buffer);
}
AFL SOURCE
INSTRUMENTATION
USE AFL-(GCC|CLANG)

• only works on x86 =/

$ mkdir testcases findings
$ echo "A" > testcases/test0

$ afl-gcc -o test_instr test.c crash.c
$ afl-fuzz -i testcases/ -o findings/ -- ./test_instr

~6000 exec/s
USE AFL-CLANG-FAST - LLVM MODE

• clang instrumentation: no more ASM
  ▪ CPU-independent
• advantages:
  ▪ deferred instrumentation: __AFL_INIT
  ▪ persistent mode: __AFL_LOOP
    ◦ less fork() calls
A PERSISTENT MODE MAIN()

cat test-AFL_LOOP.c
// AFL persistent mode wrapper
int main() {
    char buffer[BUFFER_SIZE];

    while (__AFL_LOOP(1000)) {
        // Clear the buffer content
        memset(buffer, 0, BUFFER_SIZE);

        // Read from stdin
        read(0, buffer, BUFFER_SIZE);

        crash(buffer);
    }
}
$ cd llvm_mode; make; cd ..

$ afl-clang-fast -o test-AFL_LOOP test-AFL_LOOP.c crash.c
$ afl-fuzz -i testcases/ -o findings/ -- ./test-AFL_LOOP

~24000 exec/s
FUZZING A BINARY
DUMB MODE

- no instrumentation =/

$ gcc -o test_binary test.c crash.c
$ afl-fuzz -i testcases/ -o findings/ -n -- ./test_binary

~2000 exec/s
QEMU MODE

• qemu instrumented with AFL code coverage tricks

$ cd qemu_mode; ./build_qemu_support.sh; cd ..

$ afl-fuzz -i testcases/ -o findings/ -Q -- ./test_binary

~1600 exec/s
QEMU & CROSS FUZZING

- fuzz any QEMU architecture on x86
- uses a lot of RAM =/

```
$ cd ./qemu_mode/; CPU_TARGET=arm ./build_qemu_support.sh
$ afl-qemu-trace ./test_afl_arm_static
Hello beers!
ok

$ afl-fuzz -i testcases/ -o findings/ -Q -m 4096 -- ./test_arm_binary

~1600 exec/s
ON A RASPBERRY PI 3 - MODEL B

- dumb: ~500 exec/s
- llvm: ~1000 exec/s
- AFL_LOOP: ~4000 exec/s
In principle, similar code should be easy to inject into any well-behaved binary-only code (e.g., using DynamoRIO). Conditional jumps offer natural targets for instrumentation, and should offer comparable probe density.

- https://github.com/vrtadmin/moflow/tree/master/afl-dyninst
- https://github.com/ivanfratric/winafl
- https://github.com/mothran/aflpin
FUZZING WITH MIASM
WHAT IS MIASTM?

Python-based RE framework with many awesome features:

- assembly / disassembly x86 / ARM / MIPS / SH4 / MSP430
- instructions semantic using intermediate language
- emulation using JIT
- ease implementing a new architecture
- ...

See [http://miasm.re](http://miasm.re) & [https://github.com/cea-sec/miasm](https://github.com/cea-sec/miasm) for code, examples and demos
HOW?

• Using [https://github.com/jwilk/python-afl](https://github.com/jwilk/python-afl)
  • instrument Python code like AFL to get code coverage data
• Building a miasm sandbox to emulate crash()
A SIMPLE MIASM SANDBOX

$ cat afl_sb_arm.py
from miasm2.analysis.sandbox import Sandbox_Linux_arm
from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE

import sys
import afl

# Parse arguments
parser = Sandbox_Linux_arm.parser(description="ARM ELF sandboxer")
options = parser.parse_args()

# Create sandbox
sb = Sandbox_Linux_arm("test_afl_arm", options, globals())

# !\ the last part of the code is on the next slide /!\ #
# Get the address of crash()
crash_addr = sb.elf.getsectionbyname(".symtab").symbols["crash"].value

# Create the memory page
sb.jitter.vm.add_memory_page(0xF2800, PAGE_READ | PAGE_WRITE, "\x00" * 1024)

while afl.loop():  #<- py-afl magic
    #afl.init()  # <- py-afl magic
    #if 1:
        # Read data from stdin and copy it to memory
        data = sys.stdin.readline()[:28] + "\x00"
        sb.jitter.vm.set_mem(0xF2800, data)
        # Call crash()
        sb.call(crash_addr, 0xF2800)
DUMB MODE

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -n -- python afl

Python jitter: ~8 exec/s

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -n -- python afl

GCC jitter: ~10 exec/s
AFL.INIT()

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -- python afl_sb_arm

Python jitter: ~2 exec/s

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -- python afl_sb_arm

GCC jitter: ~4 exec/s
AFL.LOOP()

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -- python afl_sb_arm

Python jitter: ~10 exec/s

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -- python afl_sb_arm

GCC jitter: ~180 exec/s
SPEEDING THINGS UP!

miasm emulates printf() in Python =/

let's remove printf() calls and recompile it!

$ py-afl-fuzz -m 512 -t 5000 -i testcases/ -o findings/ -- python afl_sb_arm

GCC jitter: ~2500 exec/s
BONUS

HELPING AFL WITH MIASM DSE
KEY CONCEPTS

• AFL & SE:
  - equally good / bad at findings generic / specific solutions

• AFL won't find

```c
unsigned long magic = strtoul(&data[4], 0, 10);
if (magic == 2206)
    printf("Fail ...
");
```

• the plan:
  1. run AFL and stop when it gets stuck
  2. use AFL outputs to solver constraints with miasm DSE
### American Fuzzy Lop 2.43b (test_afl_dse)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 0 min, 5 sec</td>
<td>Cycles done: 17</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 0 min, 2 sec</td>
<td>Total paths: 5</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>
<tr>
<td>Cycle progress</td>
<td>Map Coverage</td>
</tr>
<tr>
<td>Now processing: 4 (80.00%)</td>
<td>Map Density: 0.01% / 0.02%</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count Density: 1.00 bits/tuple</td>
</tr>
<tr>
<td>Stage progress</td>
<td>Findings in depth</td>
</tr>
<tr>
<td>Now trying: havoc</td>
<td>Favored paths: 5 (100.00%)</td>
</tr>
<tr>
<td>Stage execs: 954/8192 (11.65%)</td>
<td>New edges on: 5 (100.00%)</td>
</tr>
<tr>
<td>Total execs: 34.1k</td>
<td>Total crashes: 0 (0 unique)</td>
</tr>
<tr>
<td>Exec speed: 5343/sec</td>
<td>Total tmouts: 0 (0 unique)</td>
</tr>
<tr>
<td>Fuzzing strategy yields</td>
<td>Path Symmetry</td>
</tr>
<tr>
<td>Bit flips: 0/112, 1/107, 0/97</td>
<td>Paths: 1</td>
</tr>
<tr>
<td>Byte flips: 0/14, 0/9, 0/2</td>
<td>Peps: 1</td>
</tr>
<tr>
<td>Arithmetics: 1/783, 0/50, 0/0</td>
<td>Own/inds: 4</td>
</tr>
<tr>
<td>Known ints: 0/80, 0/252, 0/88</td>
<td>Imported: n/a</td>
</tr>
<tr>
<td>Dictionary: 0/0, 0/0, 0/0</td>
<td>Stability: 100.00%</td>
</tr>
<tr>
<td>Havoc: 2/31.5k, 0/0</td>
<td></td>
</tr>
<tr>
<td>Trim: 91.11%/11, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>

+++ Testing aborted by user +++

[+] We're done here. Have a nice day!

```
$ source ve_miasm/bin/activate
(ve_miasm) $  
```
PERSPECTIVES

• generalize the DSE PoC
• instrument a binary using miasm
• pretend that the 'binary' is instrumented
  ▪ use the shm to update the coverage bitmap!
Questions? Beers?

https://guedou.github.io