# 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/

# THE TARGET: CRASH()

```
$ 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])
  {
    if (data[1] && data[3] == data[0] + 16)
    {
      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 PERSITENT 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

# OTHER ALTERNATIVES

From afl-as.h:

```
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 MIASM?

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 & https://github.com/cea-sec/miasm for code, examples and demos

# HOW?

- Using 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_arml
from miasm2.jitter.csts import PAGE_READ, PAGE_WRITE

import sys
import afl

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

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

# /!\ the last part of the code is on the next slide /!\ #
```

```python
# /!\ the first part of the code is on the previous 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" *

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 af
```

Python jitter: ~8 exec/s

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

GCC jitter: ~10 exec/s

# AFL.INIT()

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

Python jitter: ~2 exec/s

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

GCC jitter: ~4 exec/s

# AFL.LOOP()

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

Python jitter: ~10 exec/s

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

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_s
```

GCC jitter: ~2500 exec/s

# BONUS

## HELPING AFL WITH MIASM DSE

# KEY CONCEPTS

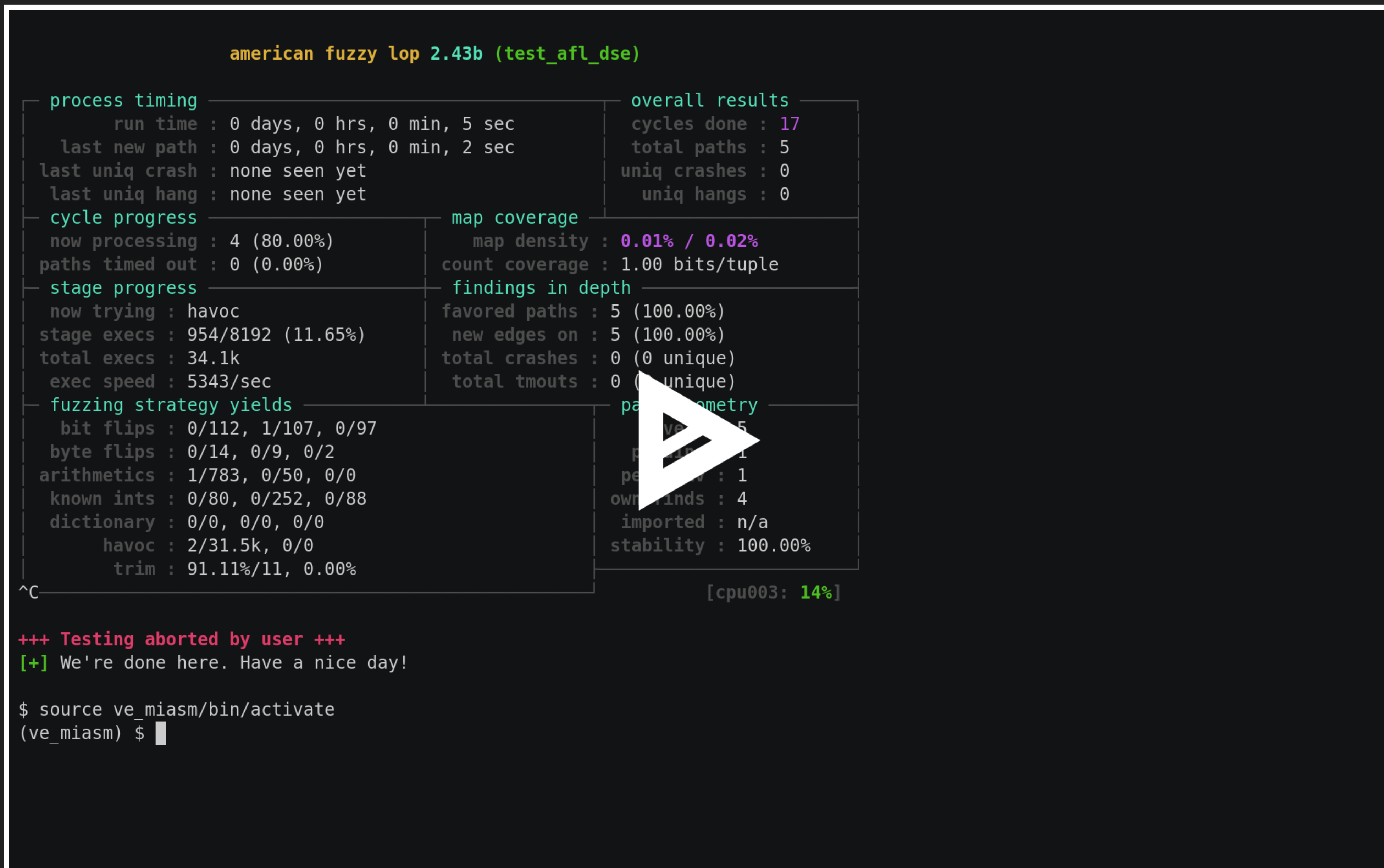
- AFL & SE:
  - equally good / bad at findings generic / specific solutions
- AFL won't find

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

- the plan:

1. run AFL and stop when it gets stuck
2. use AFL outputs to solver constraints with miasm DSE

# DEMO?


american fuzzy lop 2.43b (test_afl_dse)
process timing
run time : 0 days, 0 hrs, 0 min, 5 sec
last new path : 0 days, 0 hrs, 0 min, 2 sec
last uniq crash : none seen yet
last uniq hang : none seen yet
overall results
cycles done : 17
total paths : 5
uniq crashes : 0
uniq hangs : 0
cycle progress
now processing : 4 (80.00%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.01% / 0.02%
count coverage : 1.00 bits/tuple
stage progress
now trying : havoc
stage execs : 954/8192 (11.65%)
total execs : 34.1k
exec speed : 5343/sec
findings in depth
favored paths : 5 (100.00%)
new edges on : 5 (100.00%)
total crashes : 0 (0 unique)
total tmouts : 0
fuzzing strategy yields
bit flips : 0/112, 1/107, 0/97
byte flips : 0/14, 0/9, 0/2
arithmetics : 1/783, 0/50, 0/0
known ints : 0/80, 0/252, 0/88
dictionary : 0/0, 0/0, 0/0
havoc : 2/31.5k, 0/0
trim : 91.11%/11, 0.00%
imported : n/a
stability : 100.00%
[cpu003: 14%]
^C
+++ 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