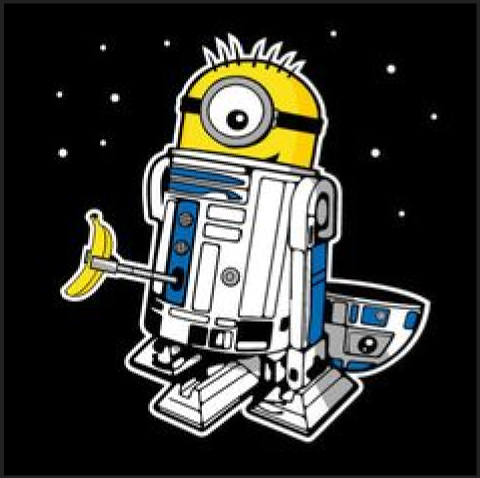
R2M2

RADARE2 + MIASM2 = ♥

@guedou - 09/09/2016



@GUEDOU?

- French
- hobbyist reverser
- network security researcher
  - IPv6, DNS, TLS, BGP, DDoS mitigation, ...
- Scapy co-maintainer
  - Python-based packet manipulation program & library
- neither a radare2 nor miasm2 power user

I needed to implement a rare CPU architecture **easily**

Back in December 2015, only `objdump` knew this architecture

```
binutils$ ./objdump -m mep -b binary -D mister.bin

mister.bin:     file format binary


Disassembly of section .data:

00000000 <.data>:
       0:       08 d8 01 00     jmp 0x100
       4:       18 df 08 00     jmp 0x8e2
[..]
   67c4a:       b0 6f           add $sp,-20
   67c4c:       1a 70           ldc $0,$lp
   67c4e:       12 48           sw $8,0x10($sp)
   67c50:       0e 47           sw $7,0xc($sp)
   67c52:       0a 46           sw $6,0x8($sp)
   67c54:       06 40           sw $0,0x4($sp)
   67c56:       10 07           mov $7,$1
   67c58:       a3 bf           bsr 0x67bfa
   67c5a:       ff 5c           mov $12,-1
   67c5c:       c1 e0 24 00     beq $0,$12,0x67ca4
   67c60:       86 d1 f5 cc     movu $1,0xccf586
```

R2M2 GOALS?

r2m2 is a radare2 plugin that aims to:

- use radare2 as a frontend to miasm2
  - tools, GUI, shortcuts, ...
- use miasm2 as a backend to radare2
  - asm/dis engine, symbolic execution, ...
- be architecture independent

MIASM 101

WHAT IS MIASM?

Python-based reverse engineering framework with many features:

- assembling / disassembling x86 / ARM / MIPS / SH4 / MSP430
- representing assembly semantic using intermediate language
- emulating using JIT
- ...

See the official blog for examples and demos

ASSEMBLING

```
# Create a x86 miasm machine
>>> from miasm2.analysis.machine import Machine
>>> m = Machine("x86_32")

# Get the mnemonic object
>>> mn = m.mn()

# Convert to an internal miasm instruction
>>> instr = mn.fromstring("MOV AX, 1", 32)

# Assemble all variants
>>> mn.asm(instr)
['f\xb8\x01\x00', 'fg\xb8\x01\x00', 'f\xc7\xc0\x01\x00',
'fg\xc7\xc0\x01\x00']
```

DISASSEMBLING

```
# Disassemble all variants
>>> [str(mn.dis(x, 32)) for x in mn.asm(instr)]
['MOV        AX, 0x1',
 'MOV        AX, 0x1',
 'MOV        AX, 0x1',
 'MOV        AX, 0x1']
```

MIASM INTERMEDIATE LANGUAGE

```
# Disassemble a simple ARM instruction
>>> m = Machine("arml")
>>> instr = m.mn.dis("002088e0".decode("hex"), "l")

# Display internal instruction arguments
>>> instr.name, instr.args
('ADD', [ExprId('R2', 32), ExprId('R8', 32), ExprId('R0', 32)]

# Get the intermediate representation architecture object
>>> ira = m.ira()

# Get the instruction miasm intermediate representation
>>> ira.get_ir(instr)
([ExprAff(ExprId('R2', 32),
          ExprOp('+', ExprId('R8', 32), ExprId('R0', 32)))], [
```

SYMBOLIC EXECUTION

```
# Add the instruction to the current block
>>> ira.add_instr(instr)

# Display the IR block
>>> for label, bloc in ira.blocs.items():
...     print bloc
...
loc_0000000000000000:0x00000000
        R2 = (R8+R0)

        IRDst = loc_0000000000000004:0x00000004
```

```
# Import the symbolic execution object
>>> from miasm2.ir.symbexec import symbexec

# Create the symbolic execution object
>>> s = symbexec(ira, ira.arch.regs.regs_init)

# Emulate using default registers value
>>> ret = s.emul_ir_bloc(ira, 0)

# Dump modified registers
>>> s.dump_id()
R2 (R0_init+R8_init)
IRDst 0x4  # miasm internal PC
```

```
# Import miasm expression objects
>>> from miasm2.expression.expression import ExprId, ExprInt32

# Affect a value to R0
>>> s.symbols[ExprId("R0", 32)] = ExprInt32(0)
>>> r = s.emul_ir_bloc(ira, 0)
>>> s.dump_id()
R2 R8_init  # the expression was simplified
[..]

# Affect a value to R8
>>> s.symbols[ExprId("R8", 32)] = ExprInt32(0x2807)
>>> r = s.emul_ir_bloc(ira, 0)
>>> s.dump_id()
R2 0x2807  # R0 + R8 = 0 + 0x2807
[..]
```

EMULATION / JIT

Let's build a simple binary to emulate

```
$ cat add.c
int add (int a, int b) { return a+b; }
main () { printf ("add (): %d\n", add (1, 2)); }

$ gcc -m32 -o add add.c

$ ./add
add(): 3
```

Then, build a miasm sandbox to emulate add()

```
$ cat sandbox_r2con.py
from miasm2.analysis.sandbox import Sandbox_Linux_x86_32

# Parse arguments
parser = Sandbox_Linux_x86_32.parser(description="ELF sandboxe
parser.add_argument("filename", help="ELF Filename")
options = parser.parse_args()

# Create sandbox
sb = Sandbox_Linux_x86_32(options.filename, options, globals()

# Get the address of add()
addr = sb.elf.getsectionbyname(".symtab").symbols["add"].value

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

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

# Push arguments on the stack
sb.jitter.push_uint32_t(1)
sb.jitter.push_uint32_t(0x2806)

# Push the address of the implicit breakpoint
sb.jitter.push_uint32_t(0x1337beef)

# Run
sb.jitter.jit.log_mn = True
sb.run(addr)

# Display the result
print "\nadd(): 0x%x" % sb.jitter.cpu.EAX
```

Finally, emulate add()

```
$ python sandbox_r2con.py ./add
080483E4 PUSH       EBP
080483E5 MOV        EBP, ESP
080483E7 MOV        EAX, DWORD PTR [EBP+0xC]
080483EA MOV        EDX, DWORD PTR [EBP+0x8]
080483ED ADD        EAX, EDX
080483EF POP        EBP
080483F0 RET

add(): 0x2807
```

GDB SERVER

```
$ python sandbox_r2con.py ./add -g 2807
Listen on port 2807
```

```
$ gdb
(gdb) target remote localhost:2807
Remote debugging using localhost:2807
0x080483ff in ?? ()
(gdb) info registers eip eax
eip            0x80483ff        0x80483ff
eax            0x0      0

(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x1337beef in ?? ()
(gdb) info registers eip eax
eip            0x1337beef       0x1337beef
eax            0x3      3
```

ADDING A NEW ARCHITECTURE TO MIASM

HIGH-LEVEL CHECKLIST

1. registers in
   miasm2/arch/ARCH/regs.py
2. opcodes in
   miasm2/arch/ARCH/arch.py
3. semantic in
   miasm2/arch/ARCH/sem.py

ADDING A NEW OPCODE IN ARCH.PY

MIPS ADDIU	
Encoding	001001 ss ssst tttt iiii iiii iiii iiii

The opcode is defined as:

```
addop("addiu", [bs("001001"), rs, rt, s16imm], [rt, rs, s16imm
```

The arguments are defined as:

```
rs = bs(l=5, cls=(mips32_gpreg,))
rt = bs(l=5, cls=(mips32_gpreg,))
s16imm = bs(l=16, cls=(mips32_s16imm,))
```

*mips32_** objects implement `encode()` and `decode()` methods that return miasm expressions!

ADDING A NEW OPCODE IN SEM.PY

Solution#1 - Implement the logic with miasm expressions

```
def addiu(ir, instr, reg_dst, reg_src, imm16):

    expr_src = ExprOp("+", reg_src, imm16.zeroExtend(32))

    return [ExprAff(reg_dst, expr_src)], []
```

Solution#2 - Be lazy, and implement using the *sembuilder*

```
@sbuild.parse
def addiu(reg_dst, reg_src, imm16):
    reg_dst = reg_src + imm16
```

The resulting expression is:

```
>>> ir.get_ir(instr)  # instr being the IR of "ADDIU A0, A1, 2
([ExprAff(ExprId('A0', 32), ExprOp('+', ExprId('A1', 32),
                            ExprInt(uint32(0x2L))))], [])
```

R2 PLUGINS IN PYTHON

RADARE2-BINDINGS BASED PLUGINS

```
$ cat radare2-bindings_plugin_ad.py

from miasm2.analysis.machine import Machine
import r2lang

def miasm_asm(buf):
    # [..]
    return asm_str

def miasm_dis(buf):
    # [..]
    return [dis_len, dis_str]

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

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

def miasm_ad_plugin(a):

    return { "name": "miasm",
             "arch": "miasm",
             "bits": 32,
             "license": "LGPL3",
             "desc": "miasm2 backend with radare2-bindings",
             "assemble": miasm_asm,
             "disassemble": miasm_dis }


r2lang.plugin("asm", miasm_ad_plugin)
```

Quite easy to use

```
$ r2 -i radare2-bindings_plugin_ad.py /bin/ls -qc 'e asm.arch=
            ;-- entry0:
            0x004049de      31ed              XOR          EBP, EBP
            0x004049e0      4989d1            MOV          R9, RDX
            0x004049e3      5e                POP          RSI
            0x004049e4      4889e2            MOV          RDX, RSP
            0x004049e7      4883e4f0          AND          RSP, 0xF
```

As of today, only *assembly* and *disassembly* plugins can be implemented

CFFI BASED PLUGINS

More steps must be taken:

1. call Python from C
2. access r2 structures from Python
3. build a r2 plugin

The CFFI Python module produces a `.so`!

STEP#1 - CALL PYTHON FROM C

Example: convert *argv[1]* in base64 from Python

1 - C side of the world

```
$ cat test_cffi.h
char* base64(char*); // under the hood, a Python function will

$ cat test_cffi.c
#include <stdio.h>
#include "test_cffi.h"

int main(int argc, char** argv)
{
  printf("[C] %s\n", base64(argc>1?argv[1]:"r2con"));
}
```

2 - Python side of the world

```
$ cat cffi_test.py

import cffi
ffi = cffi.FFI()

# Declare the function that will be exported
ffi.embedding_api("".join(open("test_cffi.h").readlines()))

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

```
# /!\ the first part of the code is on the previous slide /!\

# Define the Python module seen from Python
ffi.set_source("python_embedded", '#include "test_cffi.h"')

# Define the Python code that will be called
ffi.embedding_init_code("""
from python_embedded import ffi
@ffi.def_extern()
def base64(s):
    s = ffi.string(s)  # convert to Python string
    print "[P] %s" % s
    return ffi.new("char[]", s.encode("hex"))  # convert to C
""")

ffi.compile()
```

3 - compile

```
$ python cffi_test.py  # build python_embedded.so
$ gcc -o test_cffi test_cffi.c python_embedded.so
```

4 - enjoy

```
$ LD_LIBRARY_PATH=./ ./test_cffi cffi
[P] cffi
[C] Y2ZmaQ==

$ LD_LIBRARY_PATH=./ ./test_cffi
[P] r2con
[C] cjJjb24=
```

STEP#2 - ACCESS R2 STRUCTURES FROM PYTHON

- can't simply use `set_source()` on all r2 headers
  - CFFI C parser (pycparser) does not support all C extensions / dialects
- must *prepare* headers with alternative solutions:
  - use a C preprocessor, aka gcc -E
  - use pycparser and fake headers
  - ***automatically* extract r2 plugins structures**
    - ⇧ r2m2 does that ⇧

In a nutshell

```
// C
RAnalOp test;
set_type((RAnalOp_cffi*)&test, 0x2806);
printf("RAnalOp.type: 0x%x\n", test.type);
```

```
# Python
@ffi.def_extern()
def set_type(r2_op, value):
    r2_analop = ffi.cast("RAnalOp_cffi*", r2_op)
    r2_analop.type = value + 1
```

```
shell$ LD_LIBRARY_PATH=./ ./test_r2
RAnalOp.type: 0x2807
```

See r2m2 source code for a whole example

STEP#3 - BUILD A R2 PLUGIN

The r2 Wiki shows how to make a r_asm plugin

```
#include <r_asm.h>
#include <r_lib.h>
#include "r2_cffi.h"
#include "cffi_ad.h"

static int disassemble(RAsm *u, RAsmOp *o, const ut8 *b, int l
  python_dis(b, l, (RAsmOp_cffi*)o);
  return o->size;
}

static int assemble(RAsm *u, RAsmOp *o, const char *b) {
  python_asm(b, (RAsmOp_cffi*)o);
  return p->size;
}

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

```c
// /!\ the first part of the code is on the previous slide /!\

RAsmPlugin r_asm_plugin_cffi = {
  .name = "cffi",
  .arch = "cffi",
  .license = "LGPL3",
  .bits = 32,
  .desc = "cffi",
  .disassemble = disassemble,
  .assemble = assemble

};

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

```
// /!\ the other parts of the code are on the previous slides

#ifndef CORELIB
struct r_lib_struct_t radare_plugin = {
  .type = R_LIB_TYPE_ASM,
  .data = &r_asm_plugin_cffi
};
#endif
```

R2M2

(at last!)

WHAT IS R2M2?

- uses everything described so far to bring miasm2 to radare2!
- keeps most of the smart logics in miasm2
  - r2m2 aims to be architecture independent
  - uses the R2M2_ARCH env variable to specify the arch
- provides two r2 plugins:
  - ad: <u>a</u>ssembly & <u>d</u>isassembly
  - Ae: <u>A</u>nalysis & <u>e</u>sil

```
r2m2$ rasm2 -L |grep r2m2
adAe  32         r2m2        LGPL3   miasm2 backend
```

R2M2_AD - THE EASY PLUGIN

- simple CFFI / C wrapper around a miasm2 `Machine()`
- provides miasm2 assembly & disassembly features to radare2

MIPS32 assembly/disassembly with rasm2:

```
r2m2$ export R2M2_ARCH=mips32l; rasm2 -a r2m2 'addiu a0, a1, 2
ADDIU       A0, A1, 0x2
```

miasm2 MSP430 in r2 with random instructions:

```
r2m2$ R2M2_ARCH=msp430 r2 -a r2m2 -qc 'woR; pd 5' -
            0x00000000      07fa              and.w      R10, R7
            0x00000002      47ad              dadd.b     R13, R7
            0x00000004      f05e0778          add.b      @R14+, 0
            0x00000008      f46d81ed          addc.b     @R13+, 0
            0x0000000c      3fdc              bis.w      @R12+, R
```

miasm2 x86-64 on `/bin/ls`:

```
r2m2$ R2M2_ARCH=x86_64 r2 -a r2m2 /bin/ls -qc 'pd 7 @0x00404a1
            0x00404a1c      4883f80e        CMP         RAX, 0xE
            0x00404a20      4889e5          MOV         RBP, RSP
            0x00404a23      761b            JBE         0x1D
            0x00404a25      b800000000      MOV         EAX, 0x0
            0x00404a2a      4885c0          TEST        RAX, RAX
            0x00404a2d      7411            JZ          0x13
            0x00404a2f      5d              POP         RBP
```

Where does these jumps go?

R2M2_AE - THE CHALLENGING ONE

Use miasm2 to **automatically**

- find branches
- find function calls
- split blocks
- emulate instructions
- ...

HOW?

Step#1 - use miasm2 expressions and internal methods

- `breakflow()`, `dstflow()`, `is_subcall()`

```
# r2m2 incomplete example
if instr.is_subcall():
    if isinstance(instr.arg, ExprInt):
        analop.type = R_ANAL_OP_TYPE_CALL
        analop.jump = address + int(instr.arg)
    else:
        analop.type = R_ANAL_OP_TYPE_UCALL
```

A simple MIPS32 output

```
r2m2$ R2M2_ARCH=mips32b rasm2 -a r2m2 'j 0x4; nop' -B > j_nop.

r2m2$ R2M2_ARCH=mips32b r2 -a r2m2 j_nop.bin -qc 'pd 2'
        ,=< 0x00000000      08000001        J           0x4
        `-> 0x00000004      00000000        NOP
```

A more complex output - r2 vs r2m2

```
r2$ r2 /bin/ls -qc 'pd 12 @0x00404a1c'
            0x00404a1c      4883f80e       cmp rax, 0xe
            0x00404a20      4889e5         mov rbp, rsp
        ,=< 0x00404a23      761b           jbe 0x404a40
        |   0x00404a25      b800000000     mov eax, 0
        |   0x00404a2a      4885c0         test rax, rax
       ,==< 0x00404a2d      7411           je 0x404a40
       ||   0x00404a2f      5d             pop rbp
       ||   0x00404a30      bf60e66100     mov edi, loc._edata
       ||   0x00404a35      ffe0           jmp rax
       ||   0x00404a37      660f1f840000.  nop word [rax + rax
       ``-> 0x00404a40      5d             pop rbp
            0x00404a41      c3             ret
```

```
r2m2$ R2M2_ARCH=x86_64 r2 -a r2m2 /bin/ls -qc 'pd 12 @0x00404a
            0x00404a1c      4883f80e       CMP        RAX, 0xE
            0x00404a20      4889e5         MOV        RBP, RSP
        ,=< 0x00404a23      761b           JBE        0x1D
        |   0x00404a25      b800000000     MOV        EAX, 0x0
        |   0x00404a2a      4885c0         TEST       RAX, RAX
       ,==< 0x00404a2d      7411           JZ         0x13
       ||   0x00404a2f      5d             POP        RBP
       ||   0x00404a30      bf60e66100     MOV        EDI, loc
       ||   0x00404a35      ffe0           JMP        RAX
```

```
||    0x00404a37      660f1f840000.  NOP        WORD PTR
``-> 0x00404a40      5d             POP        RBP
     0x00404a41      c3             RET
```

Step#2 - convert miasm2 expression to radare2 ESIL

- both achieve the same goal: express instructions semantics
- simple automatic conversions are possible

```
m2 expr -> ExprAff(ExprId("R0", 32), ExprInt(0x2807, 32))
r2 esil -> 0x2807,r0,=
```

- need to dynamically define the radare2 registers profile
  - done thanks to CFFI and miasm2
- some instructions are problematic, as their semantics are complex
  - radare2 limits ESIL to be less than 64 bytes long

What to do with long ESIL expressions?

- drop them
  - weird solution
- truncate them
  - difficult to predict the outcome, but <u>today</u> r2m2 does that
- try to simplify them in r2
  - ⇧ r2m2 should do that, sooner or later ⇧

A simple MIPS32 output

```
r2m2$ R2M2_ARCH=mips32b rasm2 -a r2m2 'j 0x4; nop' -B > j_nop.

r2m2$ R2M2_ARCH=mips32b r2 -a r2m2 j_nop.bin -qc 'e asm.emu=tr
        ,=< 0x00000000      08000001       J          0x4
        `-> 0x00000004      00000000       NOP    n
```

A more complex output

```
R2M2_ARCH=x86_64 r2 -a r2m2 /bin/ls -qc 'e asm.emu=true; pd 12
            0x00404a1c      4883f80e        CMP          RAX, 0xE
            0x00404a20      4889e5          MOV          RBP, RSP
        ,=< 0x00404a23      761b            JBE          0x1D
        |   0x00404a25      b800000000      MOV          EAX, 0x0
        |   0x00404a2a      4885c0          TEST         RAX, RAX
       ,==< 0x00404a2d      7411            JZ           0x13
       ||   0x00404a2f      5d              POP          RBP
       ||   0x00404a30      bf60e66100      MOV          EDI, loc
       ||   0x00404a35      ffe0            JMP          RAX
       ||   0x00404a37      660f1f840000.   NOP          WORD PTR
       ``-> 0x00404a40      5d              POP          RBP
            0x00404a41      c3              RET
```

CURRENT ISSUES & FUTURE WORK

- truncated ESIL: simplify with m2 `expr_simp()`
- calling conventions: specify them dynamically
- redesign r2m2 as regular Python module
  - ease code reuse (for Python or r2pipe plugins)
  - ease unit & regression tests

- add r2m2 to r2pm

CONCLUDING REMARKS

- miasm2 and radare2 are powerful tools
  - combining them turned out to be efficient
- r2m2 is more than *"PoC that works on my laptop"*

```
$ docker run --rm -it -e 'R2M2_ARCH=mips32l' guedou/r
"rasm2 -a r2m2 'addiu a0, a1, 2'"
```

- too good to be true?
  - could be, yet r2m2 is better than nothing

Today, in September 2016, r2m2 allows me to get call graphs

Questions? Comments? Issues? Beers?

https://github.com/guedou/r2m2