Another introduction into radare2

{condret|Lukas}

Overview

- Features
- Components
- Api examples
- Introduction into Esil
- Problems

Features

Radare2 is not just one tool or a conglomeration of several tools.

It provides libs for creating your own tools and there are bindings for many languages.

→ Even if you don't like programming in C, you can still use the R_API.

Features

- Disassembler/Assembler
- Analysis
- Debugger
- Base conversion
- Binary information
- Payload helpers
- ...

Components

- rax2 (base conversion)
- rasm2 (disassembler/assembler)
- rabin2 (binary information)
- radiff2 (binary diffing)
- ragg2/ragg2-cc (payload helper)
- rahash2 (hashing)
- radare2 (all comes together)

rax2

rax2 is a base converter:

hex to ternary:

```
$ rax2 Tx23
1022t
```

bin to int:

```
$ rax2 101010d
42
```

rasm2

rasm2 is an assembler and disassembler for a great number of archs:

Gameboy disassembler:

```
$ rasm2 -a gb -d ff
rst 56
```

Show all supported archs:

```
$ rasm2 -L
_d           8051    PD           8051 Intel CPU
...
```

rabin2

rabin2 provides information about binaries:

/bin/ls:

```
$ rabin2 -I /bin/ls
file/bin/ls
typeEXEC (Executable file)
pic false
canary  true
nx  true
crypto  false
has_va  true
```

rabin2

```
rootelf
class   ELF64
langc
archx86
bits64
machine  AMD x86-64 architecture
os  linux
subsys   linux
endian   little
strip    true
static   false
…
```

rabin2

resolve linked libs:

```
$ rabin2 -l /bin/ls                (lower case L)
[Linked libraries]
libcap.so.2
libacl.so.1
libc.so.6

3 libraries
```

radiff2

radiff2 can do binary diffing and provides creating patchfiles.

Examples? Not here, later.

rahash2

rahash2 can calculate hashes:

calculate sha1 hash of a string:

```
$ rahash2 -a sha1 -s radare2
0x00000000-0x00000007 sha1: 0fc19fa3bbe77e97ece9f0e036444ee16b277a88
```

calculate md5 hash of a file:

```
$ rahash2 -a md5 /bin/r2
/bin/r2: 0x00000000-0x000129d1 md5: 91fb7bf0b5bb8e81a2bc01dc17f4de16
```

ragg2/ragg2-cc

ragg2 helps creating payloads, and provides filters for IDS-circumvention:

create a tiny bin:

```
$ echo "int main() { write (1,\"hi\n\", 3); exit(0); }" > hi.c
$ ragg2-cc hi.c
$ ./hi.c.bin
hi
```

ragg2/ragg2-cc

use xor-encoder for circumvention:

```
$ ragg2 -e xor -c key=32 -B `ragg2-cc -x hi.c`

6a2c596a205be8ffffffffc15e4883c60d301e48ffc6e2f9c9242020204848492a209f21202020
68ad15d0dfdfdf9a232020209821202020 2f25981c2020206010df2f25e3
```

radare2

radare2 is the hex-editor where everything comes together. It provides:

- different io-layers
- debugger
- basic code analysis
- esil-vm (wip)
- ... (no examples here :( )

API-Examples

r_asm and r_anal api:

```
#include <r_asm.h>
#include <r_anal.h>
#include <r_types.h>
#include <stdio.h>

void main () {
	RAsm *a;
	RAsmOp *aop;
	RAnal *anal;
	RAnalOp *anop;
	ut8 hex = 0x00;
	a = r_asm_new ();
	anal = r_anal_new ();
	aop = R_NEW0(RAsmOp);
	anop = r_anal_op_new ();
	r_asm_use (a, "gb");
	r_anal_use (anal, "gb");
	r_anal_op (anal, anop, 0x0, &hex, 1);
	r_asm_set_pc (a, 0x0);
	r_asm_disassemble (a, aop, &hex, 1);
	printf ("0x%02x\t:\t%s\t-\tcycle-length is %i\n", hex, aop->buf_asm, anop->cycles);
	free (aop);
	r_anal_op_free (anop);
	r_asm_free (a);
	r_anal_free (anal);
}
```

API-Examples

```
$ gcc $(pkg-config --libs --cflags r_asm r_anal) ex_asm_anal.c -o ex_asm_anal
$ ./ex_asm_anal
0x00:   nop -   cycle-length is 4
```

Introduction into esil

Esil is the language for the r2 internal vm.

- Syntax
- Esil basic operations
  - Custom Ops
- Internal vars
- Examples

Esil-Syntax

Esil is quite similar to forth:

- ',' as seperator
- <src>,<dst>,op
- <src0>,<src1>,op
- <dst>,op

Esil basic operations

- mov → =
- cmp → ==
- neg → !
- mul → *
- add → +
- sub → -
- if → ?{
- else → }{
- read → []
- write → =[]
- ...

Esil custom ops

Sometimes esil basic ops are not sufficient for emulation. In that case it is possible to create a custom op in the analysis-plugin.

From libr/include/r_anal.h:

```
typedef int (*RAnalEsilOp)(RAnalEsil *esil);

...

R_API r_anal_esil_set_op (RAnalEsil *esil, const char *op, RAnalEsilOp code);
```

Esil internal vars

The vm provides useful information, such as carry, via internal vars. Those can be ro accessed with the prefix '%'. They are calculated on demand. Most of the esil basic ops store the old and the new value of the destination of the op in the esil-struct.

Internal vars:

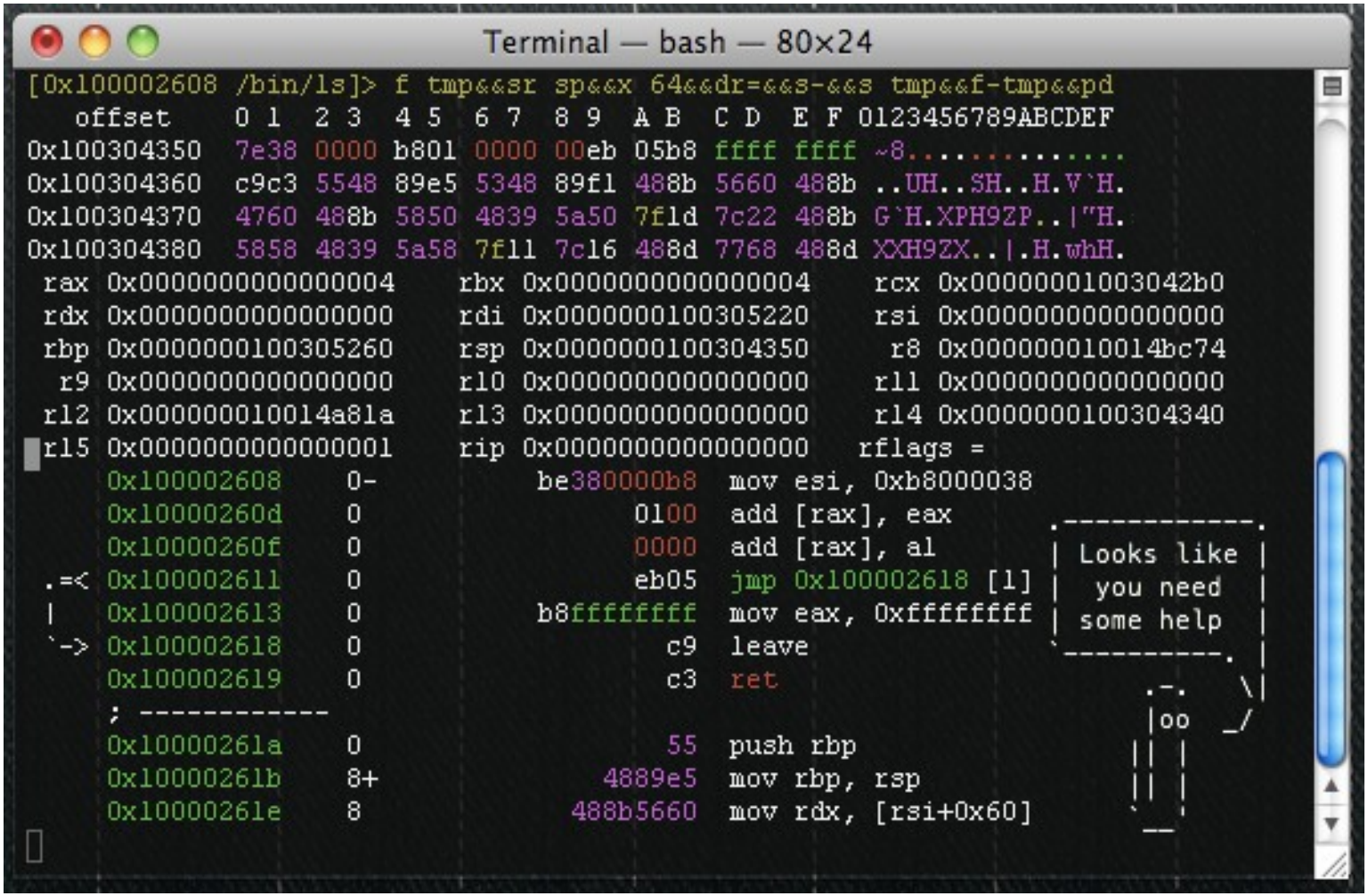
- %z - checks if the new value is zero
- %r - cpu-regsize in bytes
- %p - parity of the new value
- %cx - checks if there was a carry from bit x
- %bx - checks if there was a borrow from bit x

Examples

Here are a few examples for the gameboy-z80:

jp 0x150	→	0x150,pc,=
cp 0x11	→	17,a,==,%z,Z,=,%b4,H,=,%b8,C,=,1,N,=
xor a	→	a,a,^=,%z,Z,=,0,N,=,0,H,=,0,C,=
push de	→	2,sp,-=,de,sp,=[2]
pop hl	→	sp,[2],hl,=,2,sp,+=
ret Z	→	Z,?{,sp,[2],pc,=,2,sp,+=,}
adc b	→	b,C,+,a,+=,%c3,H,=,%c7,C,=,0,N,=
halt	→	HALT (custom op)

Problems


Terminal — bash — 80×24
[0x100002608 /bin/ls]> f tmp&&sr sp&&x 64&&dr=&&s-&&s tmp&&f-tmp&&pd
offset 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x100304350 7e38 0000 b801 0000 00eb 05b8 ffff ffff ~8..............
0x100304360 c9c3 5548 89e5 5348 89f1 488b 5660 488b ..UH..SH..H.V`H.
0x100304370 4760 488b 5850 4839 5a50 7f1d 7c22 488b G`H.XPH9ZP..|"H.
0x100304380 5858 4839 5a58 7f11 7c16 488d 7768 488d XXH9ZX..|.H.whH.
rax 0x0000000000000004 rbx 0x0000000000000004 rcx 0x00000001003042b0
rdx 0x0000000000000000 rdi 0x0000000100305220 rsi 0x0000000000000000
rbp 0x0000000100305260 rsp 0x0000000100304350 r8 0x0000000010014bc74
r9 0x0000000000000000 r10 0x0000000000000000 r11 0x0000000000000000
r12 0x000000010014a81a r13 0x0000000000000000 r14 0x0000000100304340
r15 0x0000000000000001 rip 0x0000000000000000 rflags =
0x100002608 0- be380000b8 mov esi, 0xb8000038
0x10000260d 0 0100 add [rax], eax
0x10000260f 0 0000 add [rax], al
.=< 0x100002611 0 eb05 jmp 0x100002618 [1]
| 0x100002613 0 b8ffffffff mov eax, 0xffffffff
`-> 0x100002618 0 c9 leave
0x100002619 0 c3 ret
; ------------
0x10000261a 0 55 push rbp
0x10000261b 8+ 4889e5 mov rbp, rsp
0x10000261e 8 488b5660 mov rdx, [rsi+0x60]
Looks like you need some help
.-. \|
|oo _/
|| |
|| |
`--'

Problems

Sometimes you run into a situation where static analysis fails, is not sufficient or it is just not possible to visualize things without confusing the users.

- call/jmp <reg>
- function-signatures
- gameboys halt-instruction

call/jmp <reg> (function-pointers)

Function-pointers are nice, if you are just a programmer and not interrested in reverse-engineering. If you just analyse the opcode, you won't succeed. So you have to focus more on the context to reduce the number of possible destinations. Emulation can be great for this, even if it will take a lot of time to emulate all possible paths.

But how can you visualize all these context information, without confusing the user?

The esil-vm is a great example for this problem.

call/jmp <reg> (function-pointers)

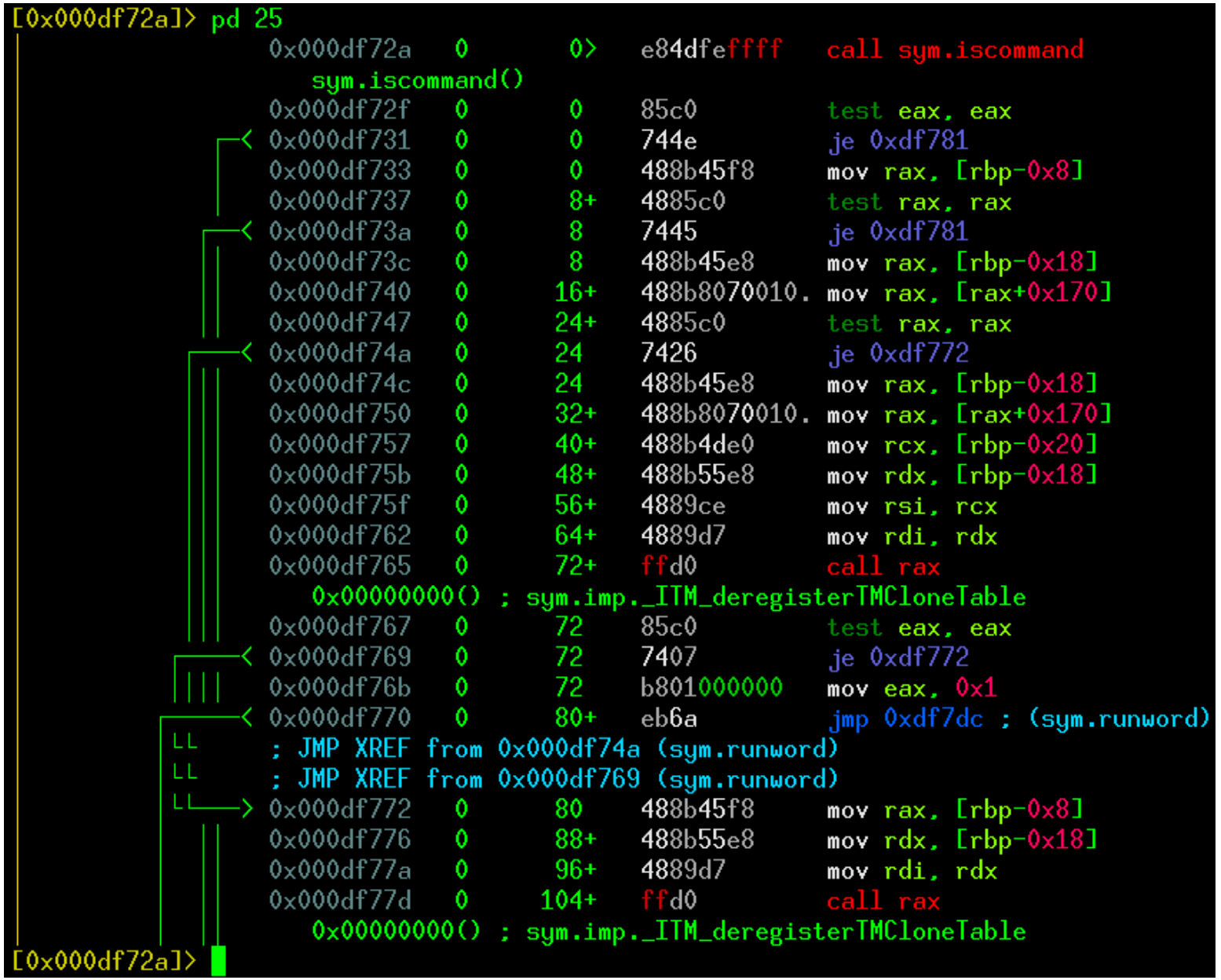
call/jmp <reg> : esil-vm

The esil-vm right now consists of 26 basic ops and 67 extended ops. The op-parsing and execution happens in r_anal_esil_parse(), runword() and iscommand().

So … let's take a short look at the disassembly:

```
$ r2 /lib/libr_anal.so
```

call/jmp <reg> : esil-vm



call/jmp <reg> : esil-vm

```
[0x000df57c]> pdf
            ; CALL XREF from 0x000df72a (sym.runword)
┌ (fcn) sym.iscommand 170
│           0x000df57c    0       0     55             push rbp
│           0x000df57d    0       8+    4889e5         mov rbp, rsp
│           0x000df580    0      16+    4881ecb0000.   sub rsp, 0xb0
│           0x000df587    0      20+    4889bd68fff.   mov [rbp-0x98], rdi
│           0x000df58e    0      28+    4889b560fff.   mov [rbp-0xa0], rsi
│           0x000df595    0      36+    48899558fff.   mov [rbp-0xa8], rdx
│           0x000df59c    0      44+    488b8560fff.   mov rax, [rbp-0xa0]
│           0x000df5a3    0      52+    4889c7         mov rdi, rax
│           0x000df5a6    0      60>    e8f551faff     call sym.imp.sdb_hash
│              sym.imp.sdb_hash(unk)
│           0x000df5ab    0      60     89c1           mov ecx, eax
│           0x000df5ad    0      68+    488d8570fff.   lea rax, [rbp-0x90]
│           0x000df5b4    0      76+    ba10000000     mov edx, 0x10
│           0x000df5b9    0      84+    4889c6         mov rsi, rax
│           0x000df5bc    0      92+    4889cf         mov rdi, rcx
│           0x000df5bf    0     100>    e80c47faff     call sym.imp.sdb_itoa
│              sym.imp.sdb_itoa()
│           0x000df5c4    0     100     488945f8       mov [rbp-0x8], rax
│           0x000df5c8    0     108+    488b8568fff.   mov rax, [rbp-0x98]
│           0x000df5cf    0     116+    488b8058010.   mov rax, [rax+0x158]
│           0x000df5d6    0     124+    488b55f8       mov rdx, [rbp-0x8]
│           0x000df5da    0     132+    4889d6         mov rsi, rdx
│           0x000df5dd    0     140+    4889c7         mov rdi, rax
│           0x000df5e0    0     148>    e8eb5cfaff     call sym.imp.sdb_num_exists
│              sym.imp.sdb_num_exists()
│           0x000df5e5    0     148     85c0           test eax, eax
│       ┌─< 0x000df5e7    0     148     7436           je 0xdf61f
│       │   0x000df5e9    0     148     488b8568fff.   mov rax, [rbp-0x98]
│       │   0x000df5f0    0     156+    488b8058010.   mov rax, [rax+0x158]
│       │   0x000df5f7    0     164+    488b4df8       mov rcx, [rbp-0x8]
│       │   0x000df5fb    0     172+    ba00000000     mov edx, 0x0
│       │   0x000df600    0     180+    4889ce         mov rsi, rcx
│       │   0x000df603    0     188+    4889c7         mov rdi, rax
│       │   0x000df606    0     196>    e88542faff     call sym.imp.sdb_num_get
│       │      sym.imp.sdb_num_get()
│       │   0x000df60b    0     196     4889c2         mov rdx, rax
│       │   0x000df60e    0     204+    488b8558fff.   mov rax, [rbp-0xa8]
│       │   0x000df615    0     212+    488910         mov [rax], rdx
│       │   0x000df618    0     220+    b801000000     mov eax, 0x1
│       │   0x000df61d    0     228+    eb05           jmp 0xdf624 ; (sym.iscommand)
│       │   ; JMP XREF from 0x000df5e7 (sym.iscommand)
│       └─> 0x000df61f    0     228     b800000000     mov eax, 0x0
│           ; JMP XREF from 0x000df61d (sym.iscommand)
│           0x000df624    0     236+    c9             leave
└           0x000df625    0     228-    c3             ret
```

call/jmp <reg> : esil-vm

The iscommand creates a hash of the op-string and checks if it exists in the sdb-instance. If so, it resolves a function-pointers belonging to the op-string. … → call rax

- It's hard to resolve all possible values for rax at this point
- Even if you can resolve all possible values for rax, you cannot ensure if all of them will be used (custom ops)
- It's hard to visualize them, without spamming

Function-Signatures

Function-Signatures are not so good resolved by r2 at the moment.

If a function with 1 arg is called twice, r2 might show you a wrong signature

GB-Halt-Instruction

Halt on gameboy is nasty, because gameboy is nasty:

- checks if interrupts are enabled
- if so:
  - run the next **byte** twice (this can result in jr-trampolins)
    - endless-loop if the next byte is a halt-instruction
    - r2 cannot handle conditional repitition of only 1 byte
- if not:
  - skip the next **instruction**
    - easy for r2 to handle

Pointers

- http://radare.org/
- http://radare.today/
- https://github.com/radare/radare2/
- irc://irc.freenode.net/radare


CALL
FOR
DEVS!

Reporting bugs


I CAN BURN YOUR HOUSE DOWN
WITH RADARE2
memegenerator.net

EOF

- Questions?
- Ideas?