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
type EXEC (Executable file)
pic  false
canary  true
nx  true
crypto  false
has_va  true
rabin2

rootelf
class ELF64
langc
arch x86
bits 64
machine AMD x86-64 architecture
os linux
subsys linux
endian little
strip true
static false
...

resolve linked libs:

$ rabin2 -l /bin/ls (lower case L)

[Linked libraries]
libcap.so.2
libacl.so.1
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: 0fc19fa3bbe77e97e9f0e036444ee16b277a88

calculate md5 hash of a file:

$ rahash2 -a md5 /bin/r2
/bin/r2: 0x00000000-0x000129d1 md5: 91fb7bf0b5bb8e81a2bc01dc17fd4de16
ragg2 helps creating payloads, and provides filters for IDS-circumvention:

create a tiny bin:

```bash
$ 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:

```bash
$ ragg2 -e xor -c key=32 -B `ragg2-cc -x hi.c`
```
6a2c596a205be8fffffffffc15e4883c60d301e48ffc6e2f9c924202048492a209f21202020
68ad15d0dfdfdf9a2320202098212020202f25981c2020206010df2f25e3
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 : ( )
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%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 separator
- <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:

```c
typedef int (*RAanalEsilOp)(RAanalEsil *esil);
...
R_API r_anal_esil_set_op (RAanalEsil *esil, const char *op, RAanalEsilOp 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
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
Function-pointers are nice, if you are just a programmer and not interested 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
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 repetition 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

INSTEAD OF REPORTING ALREADY FIXED BUGS

USE R2 FROM GITHUB
I CAN BURN YOUR HOUSE DOWN
WITH RADARE2
EOF

- Questions?
- Ideas?