

0x0000818c 255 alxchk_ls1> pd \$r @ loc.00008189+3 # 0x818c

(fcn) fcn.000081a9 283

```
0x0000818c 488b1c24 mov rbx, [rsp]
0x00008190 488b6c2408 mov rbp, [rsp+0x8]
0x00008195 4c8b642410 mov r12, [rsp+0x10]
0x0000819a 4c8b6c2418 mov r13, [rsp+0x18]
0x0000819f 4c8b742420 mov r14, [rsp+0x20]
0x000081a4 4883c428 add rsp, 0x28
0x000081a8 c3 ret
0x000081a9 0f1f800000 nop [rax]
0x000081b0 498b30 mov rsi, [r8]
0x000081b3 4839ee cmp rsi, rbp
0x000081b6 7427 jz 0x81df [1]
0x000081b8 4889ef mov rdi, rbp
; CODE (CALL) XREF from 0x000081f8 (fcn.000081f8)
0x000081bb 41ff542438 call qword [r12+0x38] ; (fcn.000081f8) [2]
fcn.000081f8()
0x000081c0 84c0 test al, al
0x000081c2 7514 jnz 0x81d8 [3]
0x000081c4 488b5b08 mov rbx, [rbx+0x8]
0x000081c8 4c8b4308 mov r8, [rbx+0x8]
0x000081cc 4d85c0 test r8, r8
0x000081cf 75df jnz 0x81b0 [4]
0x000081d1 31f6 xor esi, esi
; CODE (CALL) XREF from 0x00008189 (fcn.0000805c)
0x000081d3 ebb4 jmp loc.00008189 [5]
0x000081d5 0f1f00 nop [rax]
0x000081d8 4c8b4308 mov r8, [rbx+0x8]
0x000081dc 498b30 mov rsi, [r8]
0x000081df 4584ed test r13b, r13b
0x000081e2 74a5 jz loc.00008189 [6]
0x000081e4 498b4008 mov rax, [r8+0x8]
0x000081e8 49c7000000 mov qword [r8], 0x0
0x000081ef 48894308 mov [rbx+0x8], rax
0x000081f3 498b442448 mov rax, [r12+0x48]
; CODE (CALL) XREF from 0x000081bb (fcn.0000805c)
```

(fcn) fcn.000081f8 175

```
0x000081f8 49894008 mov [r8+0x8], rax
0x000081fc 4d89442448 mov [r12+0x48], r8
0x00008201 eb86 jmp loc.00008189 [7]
0x00008203 0f1f440000 nop [rax+rax]
0x00008208 48c7030000 mov qword [rbx], 0x0
0x0000820f e975ffffff jmp loc.00008189 [8]
; CODE (CALL) XREF from 0x00008214 (fcn.00008214)
```

(fcn) fcn.00008214 147

```
0x00008214 6666662e0f1. o16 nop [cs:rax+rax]
; CODE (CALL) XREF from 0x00005762 (fcn.000041b0)
; CODE (CALL) XREF from 0x00006251 (fcn.000041b0)
; CODE (CALL) XREF from 0x000063d2 (fcn.000041b0)
```

(fcn) fcn.00008220 135

```
0x00008220 488b07 mov rax, [rdi]
```

Table of Contents

1. [introduction](#)
2. [Introduction](#)
 - i. [History](#)
 - ii. [Overview](#)
 - iii. [Getting radare](#)
 - iv. [Compilation and portability](#)
 - v. [Windows compilation](#)
 - vi. [Commandline flags](#)
 - vii. [Basic usage](#)
 - viii. [Command format](#)
 - ix. [Expressions](#)
 - x. [Rax2](#)
 - xi. [Basic debugger session](#)
 - xii. [Contributing](#)
3. [Configuration](#)
 - i. [Colors](#)
 - ii. [Common configuration variables](#)
4. [Basic Commands](#)
 - i. [Seeking](#)
 - ii. [Block Size](#)
 - iii. [Sections](#)
 - iv. [Mapping Files](#)
 - v. [Print Modes](#)
 - vi. [Flags](#)
 - vii. [Write](#)
 - viii. [Zoom](#)
 - ix. [Yank/Paste](#)
 - x. [Comparing Bytes](#)
5. [Visual mode](#)
 - i. [Visual cursor](#)
 - ii. [Visual inserts](#)
 - iii. [Visual XREFS](#)
 - iv. [Visual Configuration editor](#)
6. [Searching bytes](#)
 - i. [Basic Searches](#)
 - ii. [Configurating the search](#)
 - iii. [Pattern Search](#)
 - iv. [Automatization](#)
 - v. [Backward Search](#)
 - vi. [Search in assembly](#)
 - vii. [Searching AES Keys](#)
7. [Disassembling](#)

- i. [Adding metadata](#)
 - ii. [ESIL](#)
- 8. [Rabin2](#)
 - i. [File identification](#)
 - ii. [Entrypoint](#)
 - iii. [Imports](#)
 - iv. [Symbols \(exports\)](#)
 - v. [Libraries](#)
 - vi. [Strings](#)
 - vii. [Program sections](#)
- 9. [Radiff2](#)
 - i. [Binary Diffing](#)
- 10. [Rasm2](#)
 - i. [Assemble](#)
 - ii. [Disassemble](#)
- 11. [Analysis](#)
 - i. [Code analysis](#)
- 12. [Rahash2](#)
 - i. [Rahash tool](#)
- 13. [Debugger](#)
 - i. [Registers](#)
- 14. [Remoting capabilities](#)
 - i. [Remoting Capabilities](#)
- 15. [Plugins](#)
 - i. [Plugins](#)
- 16. [Crackmes](#)
 - i. [IOLI](#)
 - i. [IOLI 0x00](#)
 - ii. [IOLI 0x01](#)
- 17. [Reference Card](#)

R2 "Book"

Welcome to the Radare2 Book

- Webpage: <https://www.gitbook.com/book/maijin/radare2book/details>
- Online: <http://maijin.gitbooks.io/radare2book/content/>
- PDF: <https://www.gitbook.com/download/pdf/book/maijin/radare2book>
- Epub: <https://www.gitbook.com/download/epub/book/maijin/radare2book>
- Mobi: <https://www.gitbook.com/download/mobi/book/maijin/radare2book>

Introduction

This book aims to cover most usage aspects of radare2. A framework for reverse engineering and analyzing binaries.

--pancake

History

The radare project started in February of 2006 aiming to provide a free and simple command line interface for a hexadecimal editor supporting 64 bit offsets to make searches and recovering data from hard-disks.

Since then, the project has grown with the aim changed to provide a complete framework for analyzing binaries with some basic *NIX concepts in mind like 'everything is a file', 'small programs that interact together using stdin/out' or 'keep it simple'.

It's mostly a single-person project, but some contributions (in source, patches, ideas or species) have been made and are really appreciated.

The project is composed of a hexadecimal editor as the central point of the project with assembler/disassembler, code analysis, scripting features, analysis and graphs of code and data, easy unix integration, ...

Overview

Today, the radare project is a set of small utilities that can be used together or independently from the command line.

radare2

The core of the hexadecimal editor and debugger. radare2 allows you to open a number of input/output sources as if they were simple, plain files, including disks, networks, kernel plug-ins, processes being debugged, and so on.

It implements an advanced command line interface for moving around the file, analyzing data, disassembling, binary patching, data comparison, searching, replacing, and scripting with a variety of languages, including Ruby, Python, Lua, and Perl.

rabin2

Extracts information from executable binaries, such as ELF, PE, Java CLASS, and Mach-O. rabin2 is used from the core to get exported symbols, imports, file information, cross references (xrefs), library dependencies, sections, etc.

rasm2

A command line assembler and disassembler for multiple architectures (including Intel x86 and x86-64, MIPS, ARM, PowerPC, Java, and MSIL)

Examples

```
$ rasm2 -a java 'nop'
00

$ rasm2 -a x86 -d '90'
nop

$ rasm2 -a x86 -b 32 'mov eax, 33'
b821000000

$ echo 'push eax;nop;nop' | rasm2 -f -
5090
```

rahash2

An implementation of a block-based hash. From small text strings to large disks, rahash2 supports multiple algorithms, including MD4, MD5, CRC16, CRC32, SHA1, SHA256, SHA384, SHA512, par, xor, xorpair, mod255, hamdist, or entropy.

rahash2 can be used to check the integrity of, or track changes to, big files, memory dumps, and disks.

radiff2

A binary diffing utility that implements multiple algorithms. It supports byte-level or delta diffing for binary files, and code-analysis diffing to find changes in basic code blocks from the radare code analysis, or from the IDA analysis using the rsc idc2rdb script.

rafind2

A program to find byte patterns in a file.

ragg2

A frontend for r_egg. ragg2 compiles programs into tiny binaries for x86, x86-64, and ARM.

Examples

```
$ cat hi.r
/* hello world in r_egg */
write@syscall(4);
exit@syscall(1);

main@global(128) {
    .var0 = "hi!\n";
    write(1, .var0, 4);
    exit(0);
}
$ ragg2 -O -F hi.r
$ ./hi
hi!

$ cat hi.c
main() {
    write(1, "Hello0, 6);
    exit(0);
}
$ ragg2 hi.c
$ ./hi.c.bin
Hello
```

rarun2

A launcher for running programs with a different environment, arguments, permissions, directories, and overridden, default file descriptors. rarun2 is useful for:

- Crackmes
- Fuzzing
- Test suites

Sample rarun2 script


```
$ cat foo.rr2
#!/usr/bin/rarun2
program=./pp400
arg0=10
stdin=foo.txt
chdir=/tmp
#chroot=.
./foo.rr2
```

Connecting a program to a socket

```
$ nc -l 9999
$ rarun2 program=/bin/ls connect=localhost:9999
```

Debugging a program redirecting io to another terminal

1. open a new terminal and type 'tty' to get

```
$ tty ; clear ; sleep 999999
/dev/ttyS010
```

2. In another terminal run r2

```
$ r2 -d rarun2 program=/bin/ls stdio=/dev/ttys010
```

rax2

A utility that aims to be a minimalistic expression evaluator for the shell. It is useful for making base conversions between floating point values, hexadecimal representations, hexpair strings to ascii, octal to integer. It supports endianness and can be used as a shell if no arguments are given.

Examples

```
$ rax2 1337
0x539

$ rax2 0x400000
4194304

$ rax2 -b 01111001
y

$ rax2 -S radare2
72616461726532

$ rax2 -s 617765736f6d65
awesome
```

Getting radare2

You can get radare from the website, <http://radare.org/>, or the GitHub repository, <https://github.com/radare/radare2>.

Binary packages are available for a number of operating systems (Ubuntu, Maemo, Gentoo, Windows, iPhone, and so on). I highly encourage you to get the source and compile it yourself, to better understand the dependencies and make the examples more accessible.

A new stable release is typically published every month. Nightly tarballs are sometimes available <http://bin.rada.re/>.

The radare development repository is often more stable than the 'stable' releases. To obtain the latest version:

```
$ git clone https://github.com/radare/radare2.git
```

This will probably take a while, so take a coffee break and continue reading this paper.

To update your local copy of the repository, use git pull in the root of the radare2 directory:

```
$ git pull
```

If you have local modifications of the source, you can revert them with:

```
$ git reset --hard HEAD
```

Or send me a patch:

```
$ git diff > radare-foo.patch
```

Compilation and portability

The main radare2 development is performed with GNU/Linux and GCC; however, the core of radare2 can be compiled on many systems and architectures (including TCC and Sun Studio).

People often want to use radare as a debugger for reverse engineering. Currently, the debugger layer can be used on Windows, GNU/Linux (Intel x86 and x86-64, MIPS, and ARM), FreeBSD, NetBSD, and OpenBSD (Intel x86 and x86-64). There are plans for supporting Solaris and Mac OS X.

The debugger feature is more restrictive portability wise. If the debugger has not been ported to your favorite platform, please notify me or disable the debugger layer with the `--without-debugger` configure script option when compiling radare2.

Note that there are some I/O plug-ins to use GDB, GDB Remote, or Wine as back-ends.

To build on a system using ACR/GMAKE:

```
$ ./configure --prefix=/usr
$ gmake
$ sudo gmake install
```

There is also a simple script to do this automatically:

```
$ sys/install.sh
```

Static build

You can build statically radare2 and all the tools with the command:

```
$ sys/static.sh
```

Docker

Radare2 repository ships a [Dockerfile](#) that you can use with Docker.

This dockerfile is also use by Remnux distribution from SANS and is available on the docker [registryhub](#).

Cleaning up old r2 installations

```
./configure --prefix=/old/r2/prefix/installation
make purge
```

Windows compilation

We recommend using MinGW32 to compile radare for Windows. The 32-bit Windows builds distributed on the radare homepage are generated from a GNU/Linux system using MinGW32. They are tested with Wine.

The following is an example of compiling with MinGW32 (you need to have installed **zip** for Windows):

```
$ ./configure
$ make
$ make w32dist
$ zip -r w32-build.zip w32-build
```

This generates a native, 32-bit console application for Windows.

Cygwin is another possibility; however, issues relating to Cygwin libraries can make debugging difficult in case of problems.

Please, be sure to build radare2 from that environment you're going to use r2 in. If you're going to use r2 in MinGW32 shell or cmd.exe - you need to build r2 in the MinGW32 environment. And if you're going to use r2 in Cygwin - you need to build from the Cygwin shell as well. Since Cygwin is more UNIX compatible radare2 supports here more colors and unicode symbols.

Bindings

To build radare2 bindings you will need to install [Vala \(valac\) for Windows](#)

Then download [valabind](#) and build it:

```
$ git clone https://github.com/radare/valabind.git valabind
$ cd valabind
$ make
$ make install
```

After you installed valabind, you can build radare2-bindings, for example for Python and Perl:

```
$ git clone https://github.com/radare/radare2-bindings.git radare2-bindings
$ cd radare2-bindings
$ ./configure --enable=python,perl
$ make
$ make install
```

Commandline flags

The core accepts multiple flags from the command line to change some configuration or start with different options.

Here's the help message:

```
$ radare2 -h
Usage: r2 [-dWntLqv] [-P patch] [-p prj] [-a arch] [-b bits] [-i file] [-s addr] [-B blo

    -a [arch]      set asm.arch
    -A            run 'aa' command to analyze all referenced code
    -b [bits]      set asm.bits
    -B [baddr]     set base address for PIE binaries
    -c 'cmd..'     execute radare command
    -C            file is host:port (alias for -c+=http://%s/cmd/)
    -d            use 'file' as a program for debug
    -D [backend]  enable debug mode (e cfg.debug=true)
    -e k=v         evaluate config var
    -f            block size = file size
    -h, -hh       show help message, -hh for long
    -i [file]      run script file
    -k [kernel]    set asm.os variable for asm and anal
    -l [lib]       load plugin file
    -L            list supported IO plugins
    -m [addr]      map file at given address
    -n            disable analysis
    -N            disable user settings
    -q            quiet mode (no prompt) and quit after -i
    -p [prj]       set project file
    -P [file]      apply rapatch file and quit
    -s [addr]      initial seek
    -S            start r2 in sandbox mode
    -t            load rabin2 info in thread
    -V, -v        show radare2 version (-V show lib versions)
    -w            open file in write mode
```

Basic usage

The learning curve for radare is usually somewhat steep at the beginning. Although after an hour of using it you will easily understand how most things work, and how to combine the various tools radare offers. I strongly encourage you to read the rest of this book to help you understand how everything works and to improve your skills.

Navigating a binary file is done using three simple actions: seek, print, and alterate.

The 'seek' command is abbreviated as `s` and accepts an expression as its argument. This expression can be something like `10`, `+0x25`, or `[0x100+ptr_table]`. If you are working with block-based files, you may prefer to set the block size to 4K or the size required with the `b` command, and move forward or backwards at seeks aligned to the block size using the `>` and `<` commands.

The 'print' command is abbreviated as `p` and accepts a letter to specify the print mode. The most common ones are `px` for printing in hexadecimal, and `pd` for disassembling.

To 'write', specify the `-w` option when opening the file, `radare -w`. The `w` command can be used to write strings (no letter specified), hexpairs (an `x` specified), or even assembly opcodes (an `a` specified):

```
> w hello world           ; string
> wx 90 90 90 90         ; hexpairs
> wa jmp 0x8048140        ; assemble
> wf inline.bin          ; write contents of file
```

Appending a `?` to the command will show its help message, for example, `p?`.

To enter visual mode, press `v<enter>`. Use `q` to quit visual mode and return to the prompt.

In visual mode you can use the hjkl keys to navigate (left, down, up, and right, respectively). You can use these keys in cursor mode (`c`). To select keys in cursor mode, hold down the shift key while using any of the hjkl keys.

While in visual mode you can also insert (alterate bytes) by pressing `i`, followed by `tab` to switch between the hex or string column. Pressing `q` inside the hex panel returns you to visual mode.

Command format

The general format for commands is as follows:

```
[.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
```

Commands are identified by a single character [a-zA-Z]. To repeatedly execute a command, prefix the command with a number:

```
px      # run px
3px     # run px 3 times
```

The `!` prefix is used to execute a command in shell context. If a single exclamation is used, commands will be sent to the `system()` hook defined in the currently-loaded I/O plug-in. This is used, for example, in the `ptrace` I/O plug-in, which accepts debugger commands from this interface.

Some examples:

```
ds                ; call the debugger's 'step' command
px 200 @ esp      ; show 200 hex bytes at esp
pc > file.c       ; dump buffer as a C byte array to file
wx 90 @@ sym.*    ; write a nop on every symbol
pd 2000 | grep eax ; grep opcodes that use the 'eax' register
px 20 ; pd 3 ; px 40 ; multiple commands in a single line
```

The `@` character is used to specify a temporary offset at which the command to its left will be executed. For example, `pd 5 @ 0x100000fce` will start disassembling at address `0x100000fce`.

The `~` character enables the internal grep function used to filter the output of any command. For example:

```
pd 20~call        ; disassemble 20 instructions and grep for 'call'
```

You can either grep for columns or rows:

```
pd 20~call!0      ; get first row
pd 20~call!1      ; get second row
pd 20~call[0]      ; get first column
pd 20~call[1]      ; get second column
```

Or even combine them:

```
pd 20~call[0]!0    ; grep the first column of the first row matching 'call'
```

The internal `grep` function is a key feature for scripting radare, because it can be used to iterate over a list of offsets or data processed from disassembly, ranges, or any other command. Refer to the macros section (iterators) for more information.

Expressions

Expressions are mathematical representations of a 64-bit numerical value which can be displayed in different formats, compared or used with all commands as a numeric argument. Expressions support multiple, basic arithmetic operations as well as some binary and boolean ones. The command used to evaluate these mathematical expressions is `?`. For example:

```
[0xB7F9D810]> ? 0x8048000
134512640 0x8048000 010011000000 128.0M 804000:0000 134512640 00000000 134512640.0 0.00000
[0xB7F9D810]> ? 0x8048000+34
134512674 0x8048022 01001100042 128.0M 804000:0022 134512674 00100010 134512674.0 0.00000
[0xB7F9D810]> ? 0x8048000+0x34
134512692 0x8048034 01001100064 128.0M 804000:0034 134512692 00110100 134512692.0 0.00000
[0xB7F9D810]> ? 1+2+3-4*3
-6 0xfffffffffffffffffa 01777777777777777777777777777772 17179869183.0G fffff000:0ffa -6
```

The supported arithmetic operations are:

```
+ : addition
- : subtraction
* : multiplication
/ : division
% : modulus
> : shift right
< : shift left
```

Binary operations should be escaped:

```
\| : logical OR // ("? 0001010 | 0101001")
\& : logical AND
```

Values are numbers expressable in various formats:

```
0x033 : hexadecimal
3334 : decimal
sym.fo : resolve flag offset
10K : KBytes 10*1024
10M : MBytes 10*1024*1024
```

You can also use variables and seeks to build more complex expressions. For example:


```
?@? or stype @@? ; misc help for '@' (seek), '~' (grep) (see ~??)
?$? ; show available '$' variables
$$ ; here (the current virtual seek)
$l ; opcode length
$s ; file size
```

```
$j      ; jump address (e.g. jmp 0x10, jz 0x10 => 0x10)
$f      ; jump fail address (e.g. jz 0x10 => next instruction)
$m      ; opcode memory reference (e.g. mov eax,[0x10] => 0x10)
```

Some more examples:

```
[0x4A13B8C0]> :? $m + $l
140293837812900 0x7f98b45df4a4 03771426427372244 130658.0G 8b45d000:04a4 140293837812900

[0x4A13B8C0]> :pd 1 @ +$l
0x4A13B8C2    call 0x4a13c000
```



Rax2

The `rax2` utility comes with the radare framework and aims to be a minimalistic expression evaluator for the shell. It is useful for making base conversions between floating point values, hexadecimal representations, hexpair strings to ascii, octal to integer. It supports endianness and can be used as a shell if no arguments are given.

```
$ rax2 -h

Usage: rax2 [options] [expr ...]
int   -> hex           ; rax2 10
hex   -> int           ; rax2 0xa
-int  -> hex           ; rax2 -77
-hex  -> int           ; rax2 0xffffffffb3
int   -> bin           ; rax2 b30
bin   -> int           ; rax2 1010d
float -> hex           ; rax2 3.33f
hex   -> float         ; rax2 Fx40551ed8
oct   -> hex           ; rax2 35o
hex   -> oct           ; rax2 0x12 (0 is a letter)
bin   -> hex           ; rax2 1100011b
hex   -> bin           ; rax2 Bx63
raw   -> hex           ; rax2 -S < /binfile
hex   -> raw           ; rax2 -s 414141
-b    binstr -> bin     ; rax2 -b 01000101 01110110
-B    keep base        ; rax2 -B 33+3 -> 36
-d    force integer    ; rax2 -d 3 -> 3 instead of 0x3
-e    swap endianness  ; rax2 -e 0x33
-f    floating point    ; rax2 -f 6.3+2.1
-h    help             ; rax2 -h
-k    randomart        ; rax2 -k 0x34 1020304050
-n    binary number    ; rax2 -e 0x1234 # 34120000
-s    hexstr -> raw     ; rax2 -s 43 4a 50
-S    raw -> hexstr     ; rax2 -S < /bin/ls > ls.hex
-t    tstamp -> str     ; rax2 -t 1234567890
-x    hash string      ; rax2 -x linux osx
-u    units            ; rax2 -u 389289238 # 317.0M
-v    version          ; rax2 -V
```

Some examples:

```
$ rax2 3+0x80
0x83

$ rax2 0x80+3
131

$ echo 0x80+3 | rax2
131

$ rax2 -s 4142
AB
```

```

$ rax2 -S AB
4142

$ rax2 -S < bin.foo
...

$ rax2 -e 33
0x21000000

$ rax2 -e 0x21000000
33

$ rax2 -k 90203010
+--[0x10302090]---+
|E0. .          |
| . . . .      |
|      0        |
|      .        |
|      S        |
|               |
|               |
|               |
|               |
+-----+

```

Basic debugger session

To start debugging a program use the `-d` flag and append the PID or the program path with arguments.

```
$ r2 -d /bin/ls
```

The debugger will fork and load the `ls` program in memory stopping the execution in the `ld.so`, so don't expect to see the entrypoint or the mapped libraries at this point. To change this you can define a new 'break entry point' adding `e dbg.bep=entry` or `dbg.bep=main` to your `~/.radare2rc`.

But take care on this, because some malware or programs can execute code before the main.

Here's a list of the most common commands for the debugger:

```
> d?          ; get help on debugger commands
> ds 3         ; step 3 times
> db 0x8048920 ; setup a breakpoint
> db -0x8048920 ; remove a breakpoint
> dc          ; continue process execution
> dcs         ; continue until syscall
> dd          ; manipulate file descriptors
> dm          ; show process maps
> dmp A S rwx ; change page at A with size S protection permissions
> dr eax=33 ; set register value. eax = 33
```

The easiest way to use the debugger is from the Visual mode. That way you will not need to remember many commands nor keep states in your mind.

```
[0xB7F0C8C0]> V
```

After entering this command a hexdump of the current eip will be shown. Now press `p` one time to get into the debugger view. You can press `p` and `P` to rotate through the most commonly used print modes.

Use `F7` or `s` to step into and `F8` or `S` to step over.

With the `c` key you can toggle the cursor mode to enable the selection of a range of bytes to nop them or set breakpoints using the `F2` key.

In visual mode you can enter commands with `:` to dump buffer contents like

```
x @ esi
```

To get help in visual mode press `?`.

At this point the most common commands are `dr` which can be used to get or set values of the general

purpose registers. You can also manipulate the hardware and extended/floating registers.

Contributing

Radare2 Book

If you want to contribute to the Radare2 book, you can do so at the [Github repository](#). Suggested contributions include:

- Crackme writeups
- CTF writeups
- Documentation on how to use Radare2
- Documentation on developing for Radare2
- Conference presentations/workshops using Radare2
- Missing content from the Radare1 book updated to Radare2

Please get permission to port any content you do not own/did not create before you put it in the Radare2 book.

Configuration

The core reads `~/.radare2rc` while starting, so you can setup there some `e` commands to set it up in your favorite way.

To avoid parsing this file, use `-n` and to get a cleaner output for using radare in batch mode maybe is better to just drop the verbosity with `-v`.

All the configuration of radare is done with the `eval` command which allows the user to change some variables from an internal hashtable containing string pairs.

The most common configuration looks like this:

```
$ cat ~/.radare2rc
e scr.color = true
e dbg.bep   = loader
```

These configurations can be also defined using the `-e` flag of radare while loading it, so you can setup different initial configurations from the commandline without having to change to rc file.

```
$ radare -n -e scr.color=true -e asm.syntax=intel -d /bin/ls
```

All the configuration is stored in a hash table grouped by different root names (cfg., file., dbg., ..)

To get a list of the configuration variables just type `e` in the prompt. All the basic commands can be reduced to a single char. You can also list the configuration variables of a single eval configuration group ending the command argument with a dot `.`.

There are two enhanced interfaces to help users to interactively configure this hashtable. One is called `emenu` and provides a shell for walking through the tree and change variables.

To get a help about this command you can type `e?`:

```
Usage: e[?] [var[=value]]
e?                show this help
e?asm.bytes       show description
e??              list config vars with description
e                list config vars
e-               reset config vars
e*               dump config vars in r commands
e!a              invert the boolean value of 'a' var
er [key]         set config key as readonly. no way back
ec [k] [color]   set color for given key (prompt, offset, ...)
e a              get value of var 'a'
e a=b            set var 'a' the 'b' value
env [k[=v]]      get/set environment variable
```


There is a easier `e` interface accessible from the Visual mode, just typing `ve` after entering this mode.

```
Eval spaces:
```

```
> anal
  asm
  scr
  asm
  bin
  cfg
  diff
  dir
  dbg
  cmd
  fs
  hex
  http
  graph
  hud
  scr
  search
  io
```

Most of the eval tree is quite stable, so don't expect hard changes on this area.

I encourage you to experiment a bit on this to fit the interface to your needs.

Colors

The console access is wrapped by an API that permits to show the output of any command as ANSI, w32 console or HTML (more to come ncurses, pango, ...) this allows the core to be flexible enough to run on limited environments like kernels or embedded devices allowing us to get the feedback from the application in our favourite format.

To start, we'll enable the colors by default in our rc file:

```
$ echo 'e scr.color=true' >> ~/.radare2rc
```

You can configure the colors to be used in almost every element in your disassembly. r2 supports rgb colors in unix terminals and allows to change the console color palettes using the `ec` command.

Type `ec` to get a list of all the palette elements. Type `ecs` to show a color palette to pick colors from:

```
[0x00000000]> ecs
black
red
white
green
magenta
yellow
cyan
blue
gray

Greyscale:
rgb:000  rgb:111  rgb:222  rgb:333  rgb:444  rgb:555
rgb:666  rgb:777  rgb:888  rgb:999  rgb:aaa  rgb:bbb
rgb:ccc  rgb:ddd  rgb:eee  rgb:fff

RGB:
rgb:000  rgb:030  rgb:060  rgb:090  rgb:0c0  rgb:0f0
rgb:003  rgb:033  rgb:063  rgb:093  rgb:0c3  rgb:0f3
rgb:006  rgb:036  rgb:066  rgb:096  rgb:0c6  rgb:0f6
rgb:009  rgb:039  rgb:069  rgb:099  rgb:0c9  rgb:0f9
rgb:00c  rgb:03c  rgb:06c  rgb:09c  rgb:0cc  rgb:0fc
rgb:00f  rgb:03f  rgb:06f  rgb:09f  rgb:0cf  rgb:0ff
rgb:030  rgb:330  rgb:360  rgb:390  rgb:3c0  rgb:3f0
rgb:033  rgb:333  rgb:363  rgb:393  rgb:3c3  rgb:3f3
rgb:036  rgb:336  rgb:366  rgb:396  rgb:3c6  rgb:3f6
rgb:039  rgb:339  rgb:369  rgb:399  rgb:3c9  rgb:3f9
rgb:03c  rgb:33c  rgb:36c  rgb:39c  rgb:3cc  rgb:3fc
rgb:03f  rgb:33f  rgb:36f  rgb:39f  rgb:3cf  rgb:3ff
rgb:060  rgb:630  rgb:660  rgb:690  rgb:6c0  rgb:6f0
rgb:063  rgb:633  rgb:663  rgb:693  rgb:6c3  rgb:6f3
rgb:066  rgb:636  rgb:666  rgb:696  rgb:6c6  rgb:6f6
rgb:069  rgb:639  rgb:669  rgb:699  rgb:6c9  rgb:6f9
rgb:06c  rgb:63c  rgb:66c  rgb:69c  rgb:6cc  rgb:6fc
rgb:06f  rgb:63f  rgb:66f  rgb:69f  rgb:6cf  rgb:6ff
rgb:090  rgb:930  rgb:960  rgb:990  rgb:9c0  rgb:9f0
rgb:093  rgb:933  rgb:963  rgb:993  rgb:9c3  rgb:9f3
rgb:096  rgb:936  rgb:966  rgb:996  rgb:9c6  rgb:9f6
rgb:099  rgb:939  rgb:969  rgb:999  rgb:9c9  rgb:9f9
rgb:09c  rgb:93c  rgb:96c  rgb:99c  rgb:9cc  rgb:9fc
rgb:09f  rgb:93f  rgb:96f  rgb:99f  rgb:9cf  rgb:9ff
rgb:c00  rgb:c30  rgb:c60  rgb:c90  rgb:cc0  rgb:cf0
rgb:c03  rgb:c33  rgb:c63  rgb:c93  rgb:cc3  rgb:cf3
rgb:c06  rgb:c36  rgb:c66  rgb:c96  rgb:cc6  rgb:cf6
rgb:c09  rgb:c39  rgb:c69  rgb:c99  rgb:cc9  rgb:cf9
rgb:c0c  rgb:c3c  rgb:c6c  rgb:c9c  rgb:ccc  rgb:cfc
rgb:c0f  rgb:c3f  rgb:c6f  rgb:c9f  rgb:ccf  rgb:cff
rgb:f00  rgb:f30  rgb:f60  rgb:f90  rgb:fc0  rgb:ff0
rgb:f03  rgb:f33  rgb:f63  rgb:f93  rgb:fc3  rgb:ff3
rgb:f06  rgb:f36  rgb:f66  rgb:f96  rgb:fc6  rgb:ff6
rgb:f09  rgb:f39  rgb:f69  rgb:f99  rgb:fc9  rgb:ff9
rgb:f0c  rgb:f3c  rgb:f6c  rgb:f9c  rgb:fcc  rgb:ffc
rgb:f0f  rgb:f3f  rgb:f6f  rgb:f9f  rgb:fcf  rgb:fff

[0x00000000]>
```

xvilka theme

```
ec fname rgb:0cf
ec label rgb:0f3
ec math rgb:660
ec bin rgb:f90
ec call rgb:f00
ec jmp rgb:03f
ec cjmp rgb:33c
ec offset rgb:366
ec comment rgb:0cf
ec push rgb:0c0
ec pop rgb:0c0
ec cmp rgb:060
ec nop rgb:000
ec b0x00 rgb:444
```

```

ec b0x7f rgb:555
ec b0xff rgb:666
ec btext rgb:777
ec other rgb:bbb
ec num rgb:f03
ec reg rgb:6f0
ec fline rgb:fc0
ec flow rgb:0f0

```

```

con: vi www im doc re mus fl etc xvlika@XLaptop:~/r2-test
[NAME] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
[0x000f574d 255 asrock_p4i65g.bin] > pd $r 0 section.bootblk+22349 # 0xf574d
; value = 0xd3; reg = 0x4; // XMIT_SLAVE - Transmit Slave Address
/ function: SMBus_Read_Byte_SL (57)
| 0000:574d b8d304 mov ax, 0xd3
| 0000:5750 bf5557 mov di, 0x5755
| 0000:5753 eb31 jmp SMBus_ICH5_Reg_Write_Byte_SL [1]
| ; -- SMB_Write_CMD
| 0000:5755 66c1c008 rol eax, 0x8
| 0000:5759 0c80 or al, 0x80
| ; reg = 0x3; // HST_CMD - Host Command
| 0000:575b b403 mov ah, 0x3
| 0000:575d bf6257 mov di, 0x5762
| 0000:5760 eb24 jmp SMBus_ICH5_Reg_Write_Byte_SL [2]
| ; value = 0x48; reg = 0x2; // HST_CNT - Host Control, value [6] - Start transmission, [3] - Byte Data mode
| ; -- SMB_Start_CMD
| 0000:5762 d84802 mov ax, 0x248
| 0000:5765 bf6a57 mov di, 0x576a
| 0000:5768 eb1c jmp SMBus_ICH5_Reg_Write_Byte_SL [3]
| 0000:576a b93075 mov cx, 0x7530
| ; -- SMB_Wait
| 0000:576d e6ed out 0xed, al
| 0000:576f e2fc loop 0xf576d ; (SMB_Wait) ; (SMBus_Read_Byte_SL) [4]
| 0000:5771 b8ff00 mov ax, 0xff
| 0000:5774 bf7957 mov di, 0x5779
| 0000:5777 eb04 jmp SMBus_ICH5_Reg_Write_Byte_SL [5]
| ; -- SMB_Read_Data
| 0000:5779 b405 mov ah, 0x5
| 0000:577b bf8057 mov di, 0x5780
| 0000:577e eb0e jmp SMBus_ICH5_Reg_Read_Byte_SL [6]
| 0000:5780 90f0cf bswap edi
| 0000:5783 f8 cll
| 0000:5784 ffe7 jmp di
| ; void SMBus_ICH5_Reg_Write_Byte_SL(uint8_t reg<ah>, uint8_t value<al>);
/ function: SMBus_ICH5_Reg_Write_Byte_SL (8)
| 0000:5786 ba0004 mov dx, 0x400
| 0000:5789 8ad4 mov di, ah
| 0000:578b ee out dx, al
| 0000:578c ffe7 jmp di
| ; void SMBus_ICH5_Reg_Read_Byte_SL<ah>>(uint8_t reg<ah>);
/ function: SMBus_ICH5_Reg_Read_Byte_SL (8)
| 0000:578e ba0004 mov dx, 0x400
| 0000:5791 8ad4 mov di, ah
| 0000:5793 ec in al, dx
| 0000:5794 ffe7 jmp di
| 0000:5796 7426 jz 0xf57be [7]
| 0000:579b b87000 mov ax, 0x70
| 0000:579c bca257 mov sp, 0x57a2
| 0000:579f e9b9f2 jmp 0xf4a5b [8]
| 0000:57a2 a4 movsb
| 0000:57a3 57 push di

```

Common configuration variables

Here's a list of the most common eval configuration variables, you can get the complete list using the `e` command without arguments or just use `e cfg.` (ending with dot, to list all the configuration variables of the cfg. space). You can get help on any eval configuration variable using : `??e cfg.` for example

```
asm.arch
```

Defines the architecture to be used while disassembling (pd, pD commands) and analyzing code (`a` command). Currently it handles `intel32` , `intel64` , `mips` , `arm16` , `arm` , `java` , `csr` , `sparc` , `ppc` , `msil` and `m68k` .

It is quite simple to add new architectures for disassembling and analyzing code, so there is an interface adapted for the GNU disassembler and others for udis86 or handmade ones.

```
asm.bits
```

This variable will change the `asm.arch` one (in radare1) and viceversa (is determined by asm.arch). It determines the size in bits of the registers for the selected architecture. This is 8, 16, 32, 64.

```
asm.syntax
```

Defines the syntax flavour to be used while disassembling. This is currently only targeting the udis86 disassembler for the x86 (32/64 bits). The supported values are `intel` or `att` .

```
asm.pseudo
```

Boolean value that determines which string disassembly engine to use (the native one defined by the architecture) or the one filtered to show pseudocode strings. This is `eax=ebx` instead of a `mov eax, ebx` for example.

```
asm.os
```

Defines the target operating system of the binary to analyze. This is automatically defined by `rabin -rI` and it's useful for switching between the different syscall tables and perform different depending on the OS.

```
asm.flags
```

If defined to `true` shows the flags column inside the disassembly.

```
asm.linescall
```

Draw lines at the left of the offset in the disassemble print format (pd, pD) to graphically represent jumps and calls inside the current block.

```
asm.linesout
```

When defined as `true`, also draws the jump lines in the current block that goes outside of this block.

```
asm.linestyle
```

Can get `true` or `false` values and makes the line analysis be performed from top to bottom if false or bottom to top if true. `false` is the optimal and default value for readability.

```
asm.offset
```

Boolean value that shows or hides the offset address of the disassembled opcode.

```
asm.profile
```

Set how much information is showed to the user on disassembly. Can get the values `default`, `simple`, `gas`, `smart`, `debug`, `full`.

This eval will modify other asm. variables to change the visualization properties for the disassembler engine. `simple` asm.profile will show only offset+opcode, and `debug` will show information about traced opcodes, stack pointer delta, etc..

```
asm.trace
```

Show tracing information at the left of each opcode (sequence number and counter). This is useful to read execution traces of programs.

```
asm.bytes
```

Boolean value that shows or hides the bytes of the disassembled opcode.

```
cfg.bigendian
```

Choose the endian flavour `true` for big, `false` for little.

```
file.analyze
```

Runs `.af* @@ sym.` and `.af* @ entrypoint`, after resolving the symbols while loading the binary, to determine the maximum information about the code analysis of the program. This will not be used while opening a project file, so it is preloaded. This option requires `file.id` and `file.flag` to be true.

```
scr.color
```

This boolean variable allows to enable or disable the colored output

```
scr.seek
```

This variable accepts an expression, a pointer (eg. `eip`), etc. radare will automatically seek to make sure its value is always within the limits of the screen.

```
cfg.fortunes
```

Enables or disables the 'fortune' message at the beginning of the program

Basic commands

Most command names in radare are derived from action names. They should be easy to remember as well as short. Therefore command names are all single letters. Subcommands or related commands are described using a second character. For example `/ foo` for searching plain strings or `/x 90 90` to look for hexpair strings.

The format of valid command (as explained in 'Command format' chapter) looks something like this:

```
[.][times][cmd][~grep][@[iter]addr!size][>pipe] ; ...
> 3s +1024      ; seeks three times 1024 from the current seek
```

If the command starts with `!` the string is passed to the currently loaded IO plugin (the debugger for example). If no plugin handles the command, `posix_system()` is called to hand the command to your shell. You can also force this by prefixing your command with two exclamation marks `!!` to make sure your command is directly handed to the shell.

```
> !help          ; handled by the debugger or shell
> !!ls           ; runs ls in the shell
```

The `[arg]`-part depends on the specific command. As a rule of thumb, most commands take a number as an argument to specify the number of bytes to work on instead of block size. Other commands accept math expressions, or strings.

```
> px 0x17        ; show 0x17 bytes in hexa at current seek
> s base+0x33     ; seeks to flag 'base' plus 0x33
> / lib          ; search for 'lib' string.
```

The `@` command is used to specify a temporal offset location / seek at which the command is executed. This is quite useful so you don't have to seek around all the time.

```
> p8 10 @ 0x4010 ; show 10 bytes at offset 0x4010
> f patata @ 0x10 ; set 'patata' flag at offset 0x10
```

Using the `@@` command you can execute a single command on a list of flags matching the glob. You can think of it as a foreach operation:

```
> s 0
> / lib          ; search 'lib' string
> p8 20 @@ hit0_* ; show 20 hexpairs at each search hit
```

The `>` operation is used to redirect the output of a command into a file (truncating it to 0 if it already

exist)

```
> pr > dump.bin ; dump 'raw' bytes of current block to 'dump.bin' file  
> f > flags.txt ; dump flag list to 'flags.txt'
```

The `|` operation (pipe) behaves similar to what you're used from the *NIX shell: use the output of one command as the input to another.

```
[0x4A13B8C0]> f | grep section | grep text  
0x0805f3b0 512 section._text  
0x080d24b0 512 section._text_end
```

Using the `;` char you can concatenate multiple commands on a single line:

```
> px ; dr
```

Seeking

Seeking is done using the `s` command. It accepts a math expression as argument which can be composed of shift operations, basic math operations or memory access operations.

```
[0x00000000]> s?
Usage: s[+-] [addr]
s                print current address
s 0x320         seek to this address
s-              undo seek
s+              redo seek
s*              list undo seek history
s++            seek blocksize bytes forward
s--            seek blocksize bytes backward
s+ 512          seek 512 bytes forward
s- 512          seek 512 bytes backward
sg/sG           seek begin (sg) or end (sG) of section or file
s.hexoff        Seek honoring a base from core->offset
sa [[+-]a] [asz] seek asz (or bsize) aligned to addr
sn/sp           seek next/prev scr.nkey
s/ DATA        search for next occurrence of 'DATA'
s/x 9091         search for next occurrence of \x90\x91
sb              seek aligned to bb start
so [num]         seek to N next opcode(s)
sf              seek to next function (f->addr+f->size)
sC str          seek to comment matching given string
sr pc           seek to register

> 3s++          ; 3 times block-seeking
> s 10+0x80     ; seek at 0x80+10
```

If you want to inspect the result of a math expression you can evaluate it using the `?` command. Simply pass the expression as an argument. The result can be displayed in hexadecimal, decimal, octal or binary.

```
> ? 0x100+200
0x1C8 ; 456d ; 710o ; 1100 1000
```

In visual mode you can press `u` (undo) or `U` (redo) inside the seek history.

Block size

The block size is the default view size for radare. All commands will work with this constraint, but you can always temporally change the block size just giving a numeric argument to the print commands for example (px 20)

```
[0xB7F9D810]> b?  
Usage: b[f] [arg]  
b          display current block size  
b+3        increase blocksize by 3  
b-16       decrement blocksize by 16  
b 33       set block size to 33  
b eip+4    numeric argument can be an expression  
bf foo     set block size to flag size  
bm 1M      set max block size
```

The `b` command is used to change the block size:

```
[0x00000000]> b 0x100    ; block size = 0x100  
[0x00000000]> b +16     ; ... = 0x110  
[0x00000000]> b -32     ; ... = 0xf0
```

The `bf` command is used to change the block size to the one specified by a flag. For example in symbols, the block size of the flag represents the size of the function.

```
[0x00000000]> bf sym.main ; block size = sizeof(sym.main)  
[0x00000000]> pd @ sym.main ; disassemble sym.main  
...
```

You can perform these two operations in a single one (pdf):

```
[0x00000000]> pdf @ sym.main
```

Sections

Firmware images, bootloaders and binary files usually load various sections of a binary to different addresses in memory.

To represent this behavior, radare offers the `s` command.

Here's the help message:

```
[0xB7EE8810]> S?
Usage: S[?-.*=adlr] [...]
S                ; list sections
S.               ; show current section name
S?               ; show this help message
S*               ; list sections (in radare commands)
S=               ; list sections (in nice ascii-art bars)
Sa[-] [arch] [bits] [[off]] ; Specify arch and bits for given section
Sd [file]        ; dump current section to a file (see dmd)
Sl [file]        ; load contents of file into current section (see dml)
Sr [name]        ; rename section on current seek
S [off] [vaddr] [sz] [vsz] [name] [rwx] ; add new section
S-[id|0xoff|*]   ; remove this section definition
```

You can specify a section in a single line in this way:

```
S [off] [vaddr] [sz] [vsz] [name] [rwx] ; add new section
```

For example:

```
[0x00404888]> S 0x00000100 0x00400000 0x0001ae08 0001ae08 test rwx
```

Displaying the section information:

```
[0x00404888]> S ; list sections

[00] . 0x00000238 r-- va=0x00400238 sz=0x0000001c vsz=0000001c .interp
[01] . 0x00000254 r-- va=0x00400254 sz=0x00000020 vsz=00000020 .note.ABI_tag
[02] . 0x00000274 r-- va=0x00400274 sz=0x00000024 vsz=00000024 .note.gnu.build_id
[03] . 0x00000298 r-- va=0x00400298 sz=0x00000068 vsz=00000068 .gnu.hash
[04] . 0x00000300 r-- va=0x00400300 sz=0x00000c18 vsz=00000c18 .dynsym

[0xB7EEA810]> S= ; list sections (in nice ascii-art bars)

...
25 0x0001a600 |-----#| 0x0001a608 --- .gnu_debuglink
26 0x0001a608 |-----#| 0x0001a706 --- .shstrtab
27* 0x00000000 |#####| 0x0001ae08 rwx ehdr
=> 0x00004888 |-----^-----| 0x00004988
```

The first three lines are sections and the last one (prefixed by `=>`) is the current seek location.

To remove a section definition simply prefix the name of the section with `-` :

```
[0xB7EE8810]> S - .dynsym
```

Mapping files

Radare IO allows you to virtually map contents of files into the same IO space as you loaded binary at random offsets. This is useful to open multiple files in a single view or to 'emulate' an static environment similar to what you would have using a debugger where the program and all its libraries are loaded in memory and can be accessed.

Using the `s ections` command you'll be able to define different base addresses for each library loaded.

Mapping files is done using the `o` (open) command. Let's read the help:

```
[0x00000000]> o?
Usage: o[com- ] [file] ([offset])
o                list opened files
oc [file]        open core file, like relauching r2
oo              reopen current file (kill+fork in debugger)
oo+             reopen current file in read-write
o 4             prioritize io on fd 4 (bring to front)
o-1             close file index 1
o /bin/ls        open /bin/ls file in read-only
o+/bin/ls        open /bin/ls file in read-write mode
o /bin/ls 0x4000 map file at 0x4000
on /bin/ls 0x4000 map raw file at 0x4000 (no r_bin involved)
om[?]           create, list, remove IO maps
```

Let's prepare a simple layout:

```
$ rabin2 -l /bin/ls
[Linked libraries]
libselinux.so.1
librt.so.1
libacl.so.1
libc.so.6

4 libraries
```

Map a file:

```
[0x00001190]> o /bin/zsh 0x499999
```

Listing mapped files:

```
[0x00000000]> o
- 6 /bin/ls @ 0x0 ; r
- 10 /lib/ld-linux.so.2 @ 0x100000000 ; r
- 14 /bin/zsh @ 0x499999 ; r
```

Print some hexadecimal values from /bin/zsh

```
[0x00000000]> px @ 0x499999
```

To unmap these files simply use the `o-` command giving the file descriptor as argument:

```
[0x00000000]> o-14
```

3.5 Print modes

One of the key features of radare is displaying information in various formats. The goal is to offer a selection of displaying choices to best interpret binary data.

Binary data can be represented as integers, shorts, longs, floats, timestamps, hexpair strings, or more complex formats like C structures, disassembly, decompilations, external processors, ..

Here's a list of the available print modes listable using `p?` :

```
[0x08049AD0]> p?
Usage: p[=68abcdDfiImrstuxz] [arg|len]
p=[bep?] [blks] show entropy/printable chars/chars bars
p2 [len] 8x8 2bpp-tiles
p6[de] [len] base64 decode/encode
p8 [len] 8bit hexpair list of bytes
pa[ed] [hex asm] assemble (pa) or disasm (pad) or esil (pae) from hexpairs
p[bB] [len] bitstream of N bytes
pc[p] [len] output C (or python) format
p[dD][lf] [l] disassemble N opcodes/bytes (see pd?)
pf[?|.nam] [fmt] print formatted data (pf.name, pf.name $<expr>)
p[iI][df] [len] print N instructions/bytes (f=func) (see pi? and pdi)
pm [magic] print libmagic data (pm? for more information)
pr [len] print N raw bytes
p[kK] [len] print key in randomart (K is for mosaic)
ps[pwz] [len] print pascal/wide/zero-terminated strings
pt[dn?] [len] print different timestamps
pu[w] [len] print N url encoded bytes (w=wide)
pv[jh] [mode] bar|json|histogram blocks (mode: e?search.in)
p[xX][owq] [len] hexdump of N bytes (o=octal, w=32bit, q=64bit)
pz [len] print zoom view (see pz? for help)
pwd display current working directory
```

3.5.1 Hexadecimal

User-friendly way:

```
[0x00404888]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00404888 31ed 4989 d15e 4889 e248 83e4 f050 5449 1.I..^H..H...PTI
0x00404898 c7c0 4024 4100 48c7 c1b0 2341 0048 c7c7 ..@$A.H...#A.H..
0x004048a8 d028 4000 e83f dcf8 fff4 6690 662e 0f1f .(@..?....f.f...
```

Show hexadecimal words dump (32bit)

```
[0x00404888]> pxw
0x00404888 0x8949ed31 0x89485ed1 0xe48348e2 0x495450f0 1.I..^H..H...PTI
0x00404898 0x2440c0c7 0xc7480041 0x4123b0c1 0xc7c74800 ..@$A.H...#A.H..
0x004048a8 0x004028d0 0xffdc3fe8 0x9066f4ff 0x1f0f2e66 .(@..?....f.f...
```



```
[0x00404888]> e cfg.bigendian
false

[0x00404888]> e cfg.bigendian = true

[0x00404888]> pxw
0x00404888 0x31ed4989 0xd15e4889 0xe24883e4 0xf0505449 1.I..^H..H...PTI
0x00404898 0xc7c04024 0x410048c7 0xc1b02341 0x0048c7c7 ..@$A.H...#A.H..
0x004048a8 0xd0284000 0xe83fdcff 0xffff46690 0x662e0f1f .(@..?....f.f...
```

8bit hexpair list of bytes

```
[0x00404888]> p8 16
31ed4989d15e4889e24883e4f0505449
```

Show hexadecimal quad-words dump (64bit)

```
[0x08049A80]> pxq
0x00001390 0x65625f6b63617473 0x646e6962006e6967 stack_begin.bind
0x000013a0 0x616d6f6474786574 0x7469727766006e69 textdomain.fwrit
0x000013b0 0x6b636f6c6e755f65 0x6d63727473006465 e_unlocked.strcm
...
```

3.5.2 Date formats

The current supported timestamp print modes are:

```
[0x00404888]> pt?
|Usage: pt[dn?]
| pt      print unix time (32 bit cfg.big_endian)
| ptd     print dos time (32 bit cfg.big_endian)
| ptn     print ntfs time (64 bit !cfg.big_endian)
| pt?     show help message
```

For example, you can 'view' the current buffer as timestamps in ntfs time:

```
[0x08048000]> eval cfg.bigendian = false
[0x08048000]> pt 4
29:04:32948 23:12:36 +0000
[0x08048000]> eval cfg.bigendian = true
[0x08048000]> pt 4
20:05:13001 09:29:21 +0000
```

As you can see, the endianness effect on the print formats. Once you have printed a timestamp you can grep the results by the year for example:

```
[0x08048000]> pt | grep 1974 | wc -l
15
```

```
[0x08048000]> pt | grep 2022
27:04:2022 16:15:43 +0000
```

The default date format can be configured using the `cfg.datefmt` variable. The field definitions follow the well known `strftime(3)` format.

Excerpt from the `strftime(3)` manpage:

```
%a  The abbreviated name of the day of the week according to the current locale.
%A  The full name of the day of the week according to the current locale.
%b  The abbreviated month name according to the current locale.
%B  The full month name according to the current locale.
%c  The preferred date and time representation for the current locale.
%C  The century number (year/100) as a 2-digit integer. (SU)
%d  The day of the month as a decimal number (range 01 to 31).
%D  Equivalent to %m/%d/%y. (Yecch—for Americans only. Americans should note that in ot
%e  Like %d, the day of the month as a decimal number, but a leading zero is replaced by
%E  Modifier: use alternative format, see below. (SU)
%F  Equivalent to %Y-%m-%d (the ISO 8601 date format). (C99)
%G  The ISO 8601 week-based year (see NOTES) with century as a decimal number. The 4-dig
%g  Like %G, but without century, that is, with a 2-digit year (00-99). (TZ)
%h  Equivalent to %b. (SU)
%H  The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I  The hour as a decimal number using a 12-hour clock (range 01 to 12).
%j  The day of the year as a decimal number (range 001 to 366).
%k  The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are prece
%l  The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are prece
%m  The month as a decimal number (range 01 to 12).
%M  The minute as a decimal number (range 00 to 59).
%n  A newline character. (SU)
%O  Modifier: use alternative format, see below. (SU)
%p  Either "AM" or "PM" according to the given time value, or the corresponding strings f
%P  Like %p but in lowercase: "am" or "pm" or a corresponding string for the current loca
%r  The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to %I:%M:%
%R  The time in 24-hour notation (%H:%M). (SU) For a version including the seconds, see
%s  The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). (TZ)
%S  The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for
%t  A tab character. (SU)
%T  The time in 24-hour notation (%H:%M:%S). (SU)
%u  The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w. (SU)
%U  The week number of the current year as a decimal number, range 00 to 53, starting wit
%V  The ISO 8601 week number (see NOTES) of the current year as a decimal number, range 0
%w  The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.
%W  The week number of the current year as a decimal number, range 00 to 53, starting wit
%x  The preferred date representation for the current locale without the time.
%X  The preferred time representation for the current locale without the date.
%y  The year as a decimal number without a century (range 00 to 99).
%Y  The year as a decimal number including the century.
%z  The +hhmm or -hhmm numeric timezone (that is, the hour and minute offset from UTC). (
%Z  The timezone name or abbreviation.
%+  The date and time in date(1) format. (TZ) (Not supported in glibc2.)
%%  A literal '%' character.
```

3.5.3 Basic types

There are print modes available for all basic types. If you are interested in a more complex structure or just type : `pf?`

Here's the list of the print (`pf?`) modes for basic types:

```
Usage: pf[.key[.field[=value]]][ val]][times][format] [arg0 arg1 ...]
```

Examples:

```
pf 10xiz pointer length string
pf {array_size}b @ array_base
pf.                # list all formats
pf.obj xxdz prev next size name
pf.obj             # run stored format
pf.obj.name        # show string inside object
pf.obj.size=33     # set new size
```

Format chars:

```
e - temporally swap endian
f - float value (4 bytes)
c - char (signed byte)
b - byte (unsigned)
B - show 10 first bytes of buffer
i - %i integer value (4 bytes)
w - word (2 bytes unsigned short in hex)
q - quadword (8 bytes)
p - pointer reference (2, 4 or 8 bytes)
d - 0x%08x hexadecimal value (4 bytes)
D - disassemble one opcode
x - 0x%08x hexadecimal value and flag (fd @ addr)
z - \0 terminated string
Z - \0 terminated wide string
s - 32bit pointer to string (4 bytes)
S - 64bit pointer to string (8 bytes)
* - next char is pointer (honors asm.bits)
+ - toggle show flags for each offset
: - skip 4 bytes
. - skip 1 byte
```

Let's see some examples:

```
[0x4A13B8C0]> pf i
0x00404888 = 837634441

[0x4A13B8C0]> pf
0x00404888 = 837634432.000000
```

3.5.4 Source (asm, C)

Valid print code formats are:

```
pc      C
pcs     string
pcj     json
pcJ     javascript
pcp     python
```

```
pcw    words (4 byte)
pcd    dwords (8 byte)

[0xB7F8E810]> pc 32
#define _BUFFER_SIZE 32
unsigned char buffer[_BUFFER_SIZE] = {
0x89, 0xe0, 0xe8, 0x49, 0x02, 0x00, 0x00, 0x89, 0xc7, 0xe8, 0xe2, 0xff, 0xff, 0xff, 0x81,

[0x7fcd6a891630]> pcs
"\x48\x89\xe7\xe8\x68\x39\x00\x00\x49\x89\xc4\x8b\x05\xef\x16\x22\x00\x5a\x48\x8d\x24\xc4
```

3.5.5 Strings

Strings are probably one of the most important entrypoints when starting to reverse engineer a program because they are usually referencing information about the functions actions (asserts, debug or info messages, ...).

Therefore radare supports various string formats:

```
[0x00404888]> ps?
|Usage: ps[zpw] [N]
| ps  = print string
| psb = print strings in current block
| psx = show string with scaped chars
| psz = print zero terminated string
| psp = print pascal string
| psw = print wide string
```

Most strings will be zero-terminated. Here's an example by using the debugger to continue the execution of the program until it executes the 'open' syscall. When we recover the control over the process, we get the arguments passed to the syscall, pointed by %ebx. In the case of the 'open' call, this parameter is a zero terminated string which we can inspect using `psz`.

```
[0x4A13B8C0]> dcs open
0x4a14fc24 syscall(5) open ( 0x4a151c91 0x00000000 0x00000000 ) = 0xffffffffda
[0x4A13B8C0]> dr
  eax 0xffffffffda  esi 0xfffffffffff  eip 0x4a14fc24
  ebx 0x4a151c91   edi 0x4a151be1   oeax 0x00000005
  ecx 0x00000000   esp 0xbfbdb1c   eflags 0x200246
  edx 0x00000000   ebp 0xbfbdbbb0  cPaZstIdor0 (PZI)
[0x4A13B8C0]>
[0x4A13B8C0]> psz @ 0x4a151c91
/etc/ld.so.cache
```

3.5.6 Print memory

It is also possible to print various packed data types using the `pf` command.

```
[0xB7F08810]> pf xxS @ rsp
0x7fff0d29da30 = 0x00000001
0x7fff0d29da34 = 0x00000000
0x7fff0d29da38 = 0x7fff0d29da38 -> 0x0d29f7ee /bin/ls
```

This can for instance be used to look at the arguments passed to a function. To achieve this, simply pass a 'format memory string' as an argument to `pf` and temporarily change the current seek position / offset using `@`.

It is also possible to define arrays of structures with `pf`. To do this, prefix the format string with a numeric value.

You can also define a name for each field of the structure by appending them as a space-separated argument list.

```
[0x4A13B8C0]> pf 2*xw pointer type @ esp
0x00404888 [0] {
  pointer :
  (*0xfffffffff8949ed31)      type : 0x00404888 = 0x8949ed31
  0x00404890 = 0x48e2
}
0x00404892 [1] {
  (*0x50f0e483)      pointer : 0x00404892 = 0x50f0e483
  type : 0x0040489a = 0x2440
}
```

A practical example for using `pf` on a binary of a GStreamer plugin:

```
$ radare ~/.gstreamer-0.10/plugins/libgstflumms.so
[0x000028A0]> seek sym.gst_plugin_desc
[0x000185E0]> pf iisxsssss major minor name desc _init version \
license source package origin
  major : 0x000185e0 = 0
  minor : 0x000185e4 = 10
  name : 0x000185e8 = 0x000185e8 flumms
  desc : 0x000185ec = 0x000185ec Fluendo MMS source
  _init : 0x000185f0 = 0x00002940
version : 0x000185f4 = 0x000185f4 0.10.15.1
license : 0x000185f8 = 0x000185f8 unknown
source : 0x000185fc = 0x000185fc gst-fluendo-mms
package : 0x00018600 = 0x00018600 Fluendo MMS source
origin : 0x00018604 = 0x00018604 http://www.fluendo.com
```

3.5.7 Disassembly

The `pd` command is used to disassemble code. It accepts a numeric value to specify how many opcodes should be disassembled. The `pd` command is similar but instead of a number of instructions it decompiles a given number of bytes.

```

d : disassembly N opcodes    count of opcodes
D : asm.arch disassembler    bsize bytes

[0x00404888]> pd 1
;-- entry0:
0x00404888      31ed          xor ebp, ebp

```

3.5.8 Selecting the architecture

The architecture flavour for the disassembly is defined by the `asm.arch` eval variable. You can use `e asm.arch = ?` to list all available architectures.

```

[0xB7F08810]> e asm.arch = ?

_d 16      8051      PD      8051 Intel CPU
_d 16 32    arc      GPL3    Argonaut RISC Core
ad 16 32 64  arm      GPL3    Acorn RISC Machine CPU
_d 16 32 64  arm.cs   BSD     Capstone ARM disassembler
_d 16 32     arm.winedbg LGPL2  WineDBG's ARM disassembler
_d 16 32     avr      GPL     AVR Atmel
ad 32       bf       LGPL3    Brainfuck
_d 16       cr16     LGPL3    cr16 disassembly plugin
_d 16       csr      PD       Cambridge Silicon Radio (CSR)
ad 32 64     dalvik   LGPL3    AndroidVM Dalvik
ad 16       dcpu16   PD       Mojang's DCPU-16
_d 32 64     ebc      LGPL3    EFI Bytecode
_d 8        gb       LGPL3    GameBoy(TM) (z80-like)
_d 16       h8300    LGPL3    H8/300 disassembly plugin
_d 8        i8080    BSD      Intel 8080 CPU
ad 32       java     Apache   Java bytecode
_d 16 32     m68k     BSD      Motorola 68000
_d 32       malbolge LGPL3    Malbolge Ternary VM
ad 32 64     mips     GPL3     MIPS CPU
_d 16 32 64  mips.cs  BSD     Capstone MIPS disassembler
_d 16 32 64  msil     PD       .NET Microsoft Intermediate Language
_d 32       nios2    GPL3     NIOS II Embedded Processor
_d 32 64     ppc      GPL3     PowerPC
_d 32 64     ppc.cs  BSD     Capstone PowerPC disassembler
ad 32       rar      LGPL3    RAR VM
_d 32       sh       GPL3     SuperH-4 CPU
_d 32 64     sparc    GPL3     Scalable Processor Architecture
_d 32       tms320   LGPLv3   TMS320 DSP family
_d 32       ws       LGPL3    Whitespace esoteric VM
_d 16 32 64  x86      BSD      udis86 x86-16,32,64
_d 16 32 64  x86.cs   BSD     Capstone X86 disassembler
a_ 32 64     x86.nz   LGPL3    x86 handmade assembler
ad 32       x86.olly  GPL2     OllyDBG X86 disassembler
ad 8        z80      NC-GPL2  Zilog Z80

```

3.5.9 Configuring the disassembler

There are multiple options that can be used to configure the output of the disassembler, all these options

are described using `e? asm.`

```

asm.os: Select operating system (kernel) (linux, darwin, w32,..)
asm.bytes: Display the bytes of each instruction
asm.cmtflgrefs: Show comment flags associated to branch referece
asm.cmtright: Show comments at right of disassembly if they fit in screen
asm.comments: Show comments in disassembly view
asm.decode: Use code analysis as a disassembler
asm.dwarf: Show dwarf comment at disassembly
asm.esil: Show ESIL instead of mnemonic
asm.filter: Replace numbers in disassembly using flags containing a dot in the
asm.flags: Show flags
asm.lbytes: Align disasm bytes to left
asm.lines: If enabled show ascii-art lines at disassembly
asm.linescall: Enable call lines
asm.linesout: If enabled show out of block lines
asm.linesright: If enabled show lines before opcode instead of offset
asm.linesstyle: If enabled iterate the jump list backwards
asm.lineswide: If enabled put an space between lines
asm.middle: Allow disassembling jumps in the middle of an instruction
asm.offset: Show offsets at disassembly
asm.pseudo: Enable pseudo syntax
asm.size: Show size of opcodes in disassembly (pd)
asm.stackptr: Show stack pointer at disassembly
asm.cycles: Show cpu-cycles taken by instruction at disassembly
asm.tabs: Use tabs in disassembly
asm.trace: Show execution traces for each opcode
asm.ucase: Use uppercase syntax at disassembly
asm.varsub: Substitute variables in disassembly
asm.arch: Set the arch to be usedd by asm
asm.parser: Set the asm parser to use
asm.segoff: Show segmented address in prompt (x86-16)
asm.cpu: Set the kind of asm.arch cpu
asm.profile: configure disassembler (default, simple, gas, smart, debug, full)
asm.xrefs: Show xrefs in disassembly
asm.functions: Show functions in disassembly
asm.syntax: Select assembly syntax
asm.nbytes: Number of bytes for each opcode at disassembly
asm.bytespace: Separate hex bytes with a whitespace
asm.bits: Word size in bits at assembler
asm.lineswidth: Number of columns for program flow arrows

```

3.5.10 Disassembly syntax

The syntax variable is used to influence the flavor of assembly syntax the disassembler engine outputs.

```

e asm.syntax = intel
e asm.syntax = att

```

You can also check `asm.pseudo` which is an experimental pseudocode view and `asm.esil` which outputs ESIL ('Evaluable Strings Intermediate Language'). It aims to output a human readable representation of every opcode. Those representations can be evaluated in order to emulate the code.

Flags

Flags are similar to bookmarks. They represent a certain offset in the file. Flags can be grouped in 'flag spaces'. A flag space is something like a namespace for flags. They are used to group flags of similar characteristic or type. Some example of flagspaces could be sections, registers, symbols.

To create a flag just type:

```
[0x4A13B8C0]> f flag_name @ offset
```

You can remove a flag by prefixing its name with `-`. Most commands accept `-` as argument-prefix as a way to delete items.

```
[0x4A13B8C0]> f -flag_name
```

To switch between or create new flagspaces use the `fs` command:

```
[0x4A13B8C0]> fs      ; list flag spaces

00  symbols
01  imports
02  sections
03  strings
04  regs
05  maps

[0x4A13B8C0]> fs symbols ; select only flags in symbols flagspace
[0x4A13B8C0]> f          ; list only flags in symbols flagspace

[0x4A13B8C0]> fs *      ; select all flagspaces
```

You can rename flags with `fr`.

Write

Radare can manipulate a loaded binary file in multiple ways. You can resize the file, move and copy/paste bytes, insert new bytes (shifting data to the end of the block or file) or simply overwrite bytes at a address, contents of a file, a widestring or even inline assembling an opcode.

To resize use the `r` command which accepts a numeric argument. A positive value sets the new size to the file. A negative one will strip N bytes from the current seek, down-sizing the file.

```
r 1024      ; resize the file to 1024 bytes
r -10 @ 33  ; strip 10 bytes at offset 33
```

To write bytes use the `w` command. It accepts multiple input formats like inline assembly, endian-friendly dwords, files, hexpair files, wide strings:

```
[0x00404888]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| w foobar      write string 'foobar'
| wh r2         whereis/which shell command
| wr 10         write 10 random bytes
| ww foobar     write wide string 'f\x00o\x00o\x00b\x00a\x00r\x00'
| wa push ebp   write opcode, separated by ';' (use '"' around the command)
| waf file      assemble file and write bytes
| wA r 0        alter/modify opcode at current seek (see wA?)
| wb 010203     fill current block with cyclic hexpairs
| wc[ir*?]     write cache undo/commit/reset/list (io.cache)
| wx 9090       write two intel nops
| wv eip+34     write 32-64 bit value
| wo? hex       write in block with operation. 'wo?' fmi
| wm f0ff       set binary mask hexpair to be used as cyclic write mask
| ws pstring    write 1 byte for length and then the string
| wf -|file     write contents of file at current offset
| wF -|file     write contents of hexpairs file here
| wp -|file     apply radare patch file. See wp? fmi
| wt file [sz]  write to file (from current seek, blocksize or sz bytes)
```

Some examples:

```
[0x00000000]> wx 123456 @ 0x8048300
[0x00000000]> wv 0x8048123 @ 0x8049100
[0x00000000]> wa jmp 0x8048320
```

3.8.1 Write over with operation

The `wo` command (write operation) accepts multiple kinds of operations that can be applied on the current block. This is for example a XOR, ADD, SUB, ...

```
[0x4A13B8C0]> wo?
|Usage: wo[asmdxoAr124] [hexpairs] @ addr[:bsize]
|Example:
|  wox 0x90 ; xor cur block with 0x90
|  wox 90 ; xor cur block with 0x90
|  wox 0x0203 ; xor cur block with 0203
|  woa 02 03 ; add [0203][0203][...] to curblk
|  woe 02 03
|Supported operations:
|  wow == write looped value (alias for 'wb')
|  woa += addition
|  wos -= subtraction
|  wom *= multiply
|  wod /= divide
|  wox ^= xor
|  woo |= or
|  woA &= and
|  woR random bytes (alias for 'wr $b')
|  wor >=> shift right
|  wol <=< shift left
|  wo2 2= 2 byte endian swap
|  wo4 4= 4 byte endian swap
```

This way it is possible to implement cipher-algorithms using radare core primitives.

A sample session doing a xor(90) + addition(01 02):

```
[0x7fcd6a891630]> px
- offset -      0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  4889 e7e8 6839 0000 4989 c48b 05ef 1622  H...h9..I....."
0x7fcd6a891640  005a 488d 24c4 29c2 5248 89d6 4989 e548  .ZH.$.) .RH..I..H
0x7fcd6a891650  83e4 f048 8b3d 061a 2200 498d 4cd5 1049  ...H.=..".I.L..I
0x7fcd6a891660  8d55 0831 ede8 06e2 0000 488d 15cf e600  .U.1.....H.....

[0x7fcd6a891630]> wox 90
[0x7fcd6a891630]> px
- offset -      0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  d819 7778 d919 541b 90ca d81d c2d8 1946  ..wx..T.....F
0x7fcd6a891640  1374 60d8 b290 d91d 1dc5 98a1 9090 d81d  .t`.....
0x7fcd6a891650  90dc 197c 9f8f 1490 d81d 95d9 9f8f 1490  ...|.....
0x7fcd6a891660  13d7 9491 9f8f 1490 13ff 9491 9f8f 1490  ....

[0x7fcd6a891630]> woa 01 02
[0x7fcd6a891630]> px
- offset -      0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  d91b 787a 91cc d91f 1476 61da 1ec7 99a3  ..XZ.....va.....
0x7fcd6a891640  91de 1a7e d91f 96db 14d9 9593 1401 9593  ...~.....
0x7fcd6a891650  c4da 1a6d e89a d959 9192 9159 1cb1 d959  ...m...Y...Y...Y
0x7fcd6a891660  9192 79cb 81da 1652 81da 1456 a252 7c77  ..y....R...V.R|w
```

Zoom

The zoom is a print mode that allows you to get a global view of the whole file or memory map in a single screen. Each byte represents `file_size/block_size` bytes of the file. Use the `p0` (zoom out print mode) to use it, or just toggle `z` in the visual mode to zoom-out/zoom-in.

The cursor can be used to scroll faster through the zoom out view and pressing `z` again to zoom-in where the cursor points.

```
[0x004048c5]> pz?
|Usage: pz [len] print zoomed blocks (filesize/N)
| e zoom.maxsz  max size of block
| e zoom.from   start address
| e zoom.to     end address
| e zoom.byte   specify how to calculate each byte
| pzp          number of printable chars
| pzf          count of flags in block
| pzs          strings in range
| pz0          number of bytes with value '0'
| pzF          number of bytes with value 0xFF
| pze          calculate entropy and expand to 0-255 range
| pzh          head (first byte value); This is the default mode
```

For example. let's see some examples:

```
[0x08049790]> pz // or default pzh

0x00000000  7f00 0000 e200 0000 146e 6f74 0300 0000  ....not....
0x00000010  0000 0000 0068 2102 00ff 2024 e8f0 007a  ....h!... $.z
0x00000020  8c00 18c2 ffff 0080 4421 41c4 1500 5dff  ....D!A...].
0x00000030  ff10 0018 0fc8 031a 000c 8484 e970 8648  ....p.H
0x00000040  d68b 3148 348b 03a0 8b0f c200 5d25 7074  ..1H4.....]%pt
0x00000050  7500 00e1 ffe8 58fe 4dc4 00e0 dbc8 b885  u.....X.M.....

[0x08049790]> e zoom.byte=p
[0x08049790]> p0 // or pzp

0x00000000  2f47 0609 070a 0917 1e9e a4bd 2a1b 2c27  /G.....*.,'
0x00000010  322d 5671 8788 8182 5679 7568 82a2 7d89  2-Vq...Vyh..}.
0x00000020  8173 7f7b 727a 9588 a07b 5c7d 8daf 836d  .s.{rz...{\}...m
0x00000030  b167 6192 a67d 8aa2 6246 856e 8c9b 999f  .ga..}.bF.n....
0x00000040  a774 96c3 b1a4 6c8e a07c 6a8f 8983 6a62  .t....l..|j...jb
0x00000050  7d66 625f 7ea4 7ea6 b4b6 8b57 a19f 71a2  }fb_~.~....W..q.

[0x08049790]> eval zoom.byte = flags
[0x08049790]> p0 // or pzf

0x00406e65  48d0 80f9 360f 8745 ffff ffeb ae66 0f1f  H...6..E....f..
0x00406e75  4400 0083 f801 0f85 3fff ffff 410f b600  D.....?...A...
0x00406e85  3c78 0f87 6301 0000 0fb6 c8ff 24cd 0026  <x..c.....$.&
0x00406e95  4100 660f 1f84 0000 0000 0084 c074 043c  A.f.....t.<
```

```

0x00406ea5  3a75 18b8 0500 0000 83f8 060f 95c0 e9cd  :u.....
0x00406eb5  feff ff0f 1f84 0000 0000 0041 8801 4983  .....A..I.
0x00406ec5  c001 4983 c201 4983 c101 e9ec feff ff0f  ..I...I.....

```

```
[0x08049790]> e zoom.byte=F
```

```
[0x08049790]> p0 // or pzF
```

```

0x00000000  0000 0000 0000 0000 0000 0000 0000 0000  .....
0x00000010  0000 2b5c 5757 3a14 331f 1b23 0315 1d18  ..+\WW:.3..#....
0x00000020  222a 2330 2b31 2e2a 1714 200d 1512 383d  ".*#0+1.*.. ...8=
0x00000030  1e1a 181b 0a10 1a21 2a36 281e 1d1c 0e11  .....!*6(.....
0x00000040  1b2a 2f22 2229 181e 231e 181c 1913 262b  .*/"")..#.....&+
0x00000050  2b30 4741 422f 382a 1e22 0f17 0f10 3913  +0GAB/8*."....9.

```

You can determine the limits for performing a zoom on a range of bytes of the whole bytespace by using the `zoom.from` and `zoom.to` eval variables.

```

[0x465D8810]> e zoom.
zoom.byte = f
zoom.from = 0
zoom.maxsz = 512
zoom.to = 118368???

```

Yank/Paste

You can yank/paste bytes in visual mode using the `y` and `Y` key bindings which are alias for the `y` and `yy` commands of the shell. These commands operate on an internal buffer which stores N bytes counted from the current seek. You can write-back to another seek using the `yy` command.

```
[0x00000000]> y?
|Usage: y[ptxy] [len] [[@]addr]
| y                show yank buffer information (srcoff len bytes)
| y 16            copy 16 bytes into clipboard
| y 16 0x200      copy 16 bytes into clipboard from 0x200
| y 16 @ 0x200    copy 16 bytes into clipboard from 0x200
| yp             print contents of clipboard
| yx             print contents of clipboard in hexadecimal
| yt 64 0x200     copy 64 bytes from current seek to 0x200
| yf 64 0x200 file copy 64 bytes from 0x200 from file (opens w/ io), use -1 for all bytes
| yfa file copy   copy all bytes from from file (opens w/ io)
| yy 0x3344       paste clipboard
```

Sample session:

```
[0x00000000]> s 0x100      ; seek at 0x100
[0x00000100]> y 100       ; yanks 100 bytes from here
[0x00000200]> s 0x200      ; seek 0x200
[0x00000200]> yy          ; pastes 100 bytes
```

You can perform a yank and paste in a single line by just using the `yt` command (yank-to). The syntax is as follows:

```
[0x4A13B8C0]> x
  offset  0 1  2 3  4 5  6 7  8 9  A B  0123456789AB
0x4A13B8C0, 89e0 e839 0700 0089 c7e8 e2ff ...9.....
0x4A13B8CC, ffff 81c3 eea6 0100 8b83 08ff .....
0x4A13B8D8, ffff 5a8d 2484 29c2          ..Z.$.).

[0x4A13B8C0]> yt 8 0x4A13B8CC @ 0x4A13B8C0

[0x4A13B8C0]> x
  offset  0 1  2 3  4 5  6 7  8 9  A B  0123456789AB
0x4A13B8C0, 89e0 e839 0700 0089 c7e8 e2ff ...9.....
0x4A13B8CC, 89e0 e839 0700 0089 8b83 08ff ...9.....
0x4A13B8D8, ffff 5a8d 2484 29c2          ..Z.$.).
```

Comparing bytes

You can compare data using the `c` command. It accepts an input in various formats and compares the input against the bytes in the current seek.

```
[0x00404888]> c?
|Usage: c[?dfx] [argument]
| c [string]      Compares a plain with escaped chars string
| cc [at] [(at)]  Compares in two hexdump columns of block size
| c4 [value]      Compare a doubleword from a math expression
| c8 [value]      Compare a quadword from a math expression
| cx [hexpair]    Compare hexpair string
| cx [addr]       Like 'cc' but using hexdiff output
| cf [file]       Compare contents of file at current seek
| cg[o] [file]    Graphdiff current file and [file]
| cu [addr] @at   Compare memory hexdumps of $$ and dst in unified diff
| cw[us?] [...]   Compare memory watchers
| cat [file]      Show contents of file (see pwd, ls)
| cl|cls|clear    Clear screen, (clear0 to goto 0, 0 only)
```

An example of memory comparison:

```
[0x08048000]> p8 4
7f 45 4c 46

[0x08048000]> cx 7f 45 90 46
Compare 3/4 equal bytes
0x00000002 (byte=03)  90 ' ' -> 4c 'L'
[0x08048000]>
```

Another subcommand of `c` (compare) command is `cc` which stands for 'compare code'.

```
[0x4A13B8C0]> cc 0x39e8e089 @ 0x4A13B8C0

[0x08049A80]> cc sym.main2 @ sym.main
```

`c8` compares a quadword from the current seek (0x00000000) from a math expression

```
[0x00000000]> c8 4

Compare 1/8 equal bytes (0%)
0x00000000 (byte=01)  7f ' ' -> 04 ' '
0x00000001 (byte=02)  45 'E' -> 00 ' '
0x00000002 (byte=03)  4c 'L' -> 00 ' '
```

The number parameter can of course also be a math expressions using flag names and so on:

```
[0x00000000]> cx 7f469046
```

```
Compare 2/4 equal bytes
```

```
0x00000001 (byte=02)  45 'E'  -> 46 'F'
```

```
0x00000002 (byte=03)  4c 'L'  -> 90 ' '
```

We can use the compare command to compare the current block to a file previously dumped to disk.

```
r2 /bin/true
```

```
[0x08049A80]> s 0
```

```
[0x08048000]> cf /bin/true
```

```
Compare 512/512 equal bytes
```


Visual mode

The visual mode is a user-friendlier interface for the commandline prompt of radare which accepts hjkl movement keys, a cursor for selecting bytes and some keybindings to ease the use of the debugger.

The screenshot displays the Visual mode interface of radare2, showing assembly code and a hex dump side-by-side.

Left Panel (Assembly):

```

[0x00404890 16% 120 /bin/ls]> pd $r @ entry0
(fcn) entry0 42
;-- entry0:
0x00404890 31ed      xor ebp, ebp
0x00404892 4989d1    mov r9, rdx
0x00404895 5e        pop rsi
0x00404896 4889e2    mov rdx, rsp
0x00404899 4883e4f0  and rsp, 0xfffffffffffffff0
0x0040489d 50        push rax
0x0040489e 54        push rsp
0x0040489f 49c7c0d01e41. mov r8, 0x411ed0
0x004048a6 48c7c1601e41. mov rcx, 0x411e60
0x004048ad 48c7c7c02840. mov rdi, main ; "AWAVAUATUH..S..H..
0x004048b4 e837dcffff  call sym.imp._libc_start_main[;1]
    sym.imp._libc_start_main(unk, unk, unk)
0x004048b9 f4        hlt
0x004048ba 660f1f440000 nop word [rax + rax]
(fcn) fcn.004048c0 41
; CALL XREF from 0x0040493d (fcn.00404930)
0x004048c0 b8ffa56100  mov eax, 0x61a5ff ; "hstrtab" @ 0x6
0x004048c5 55        push rbp
0x004048c6 482df8a56100 sub rax, 0x61a5f8
0x004048cc 4883f80e    cmp rax, 0xe
0x004048d0 4889e5    mov rbp, rsp
  
```

Right Panel (Hex Dump):

```

[0x00404890 16% 120 /bin/ls]> pc @ entry0
#define _BUFFER_SIZE 120
unsigned char buffer[120] = {
  0x31, 0xed, 0x49, 0x89, 0xd1, 0x5e, 0x48, 0x89, 0xe2, 0x48, 0x83,
  0xe4, 0xf0, 0x50, 0x54, 0x49, 0xc7, 0xc0, 0xd0, 0x1e, 0x41, 0x00,
  0x48, 0xc7, 0xc1, 0x60, 0x1e, 0x41, 0x00, 0x48, 0xc7, 0xc7, 0xc0,
  0x28, 0x40, 0x00, 0xe8, 0x37, 0xdc, 0xff, 0xff, 0xf4, 0x66, 0x0f,
  0x1f, 0x44, 0x00, 0x00, 0xb8, 0xff, 0xa5, 0x61, 0x00, 0x55, 0x48,
  0x2d, 0xf8, 0xa5, 0x61, 0x00, 0x48, 0x83, 0xf8, 0x00, 0x48, 0x89,
  0xe5, 0x77, 0x02, 0x5d, 0xc3, 0xb8, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xc0, 0x74, 0xf4, 0x5d, 0xbf, 0xf8, 0xa5, 0x61, 0x00, 0xff,
  0xe0, 0xf0, 0x1f, 0x80, 0x00, 0x00, 0x00, 0xb8, 0xf8, 0xa5,
  0x61, 0x00, 0x55, 0x48, 0x2d, 0xf8, 0xa5, 0x61, 0x00, 0x48, 0xc1,
  0xf8, 0x03, 0x48, 0x89, 0xe5, 0x48, 0x89, 0xc2, 0x48, 0xc1, };
  
```

Bottom Left Panel (Hex Dump):

```

0x00404890 16% 368 /bin/ls]> x @ entry0
offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
x00404890 31ed 4989 d15e 4889 e248 83e4 f050 5449 1.I..^H..H...PTI
x004048a0 c7c0 d01e 4100 48c7 c160 1e41 0048 c7c7 ...A.H..A.H..
x004048b0 c028 4000 e837 dcff fff4 660f 1f44 0000 .(0..7...f..D..
x004048c0 b8ff a561 0055 482d f8a5 6100 4883 f80e ...a.UH...a.H...
x004048d0 4889 e577 025d c3b8 0000 0000 4885 c074 H..w.].....H..t
x004048e0 f45d bfff a561 00ff e00f 1f80 0000 0000 .]...a.....
x004048f0 b8f8 a561 0055 482d f8a5 6100 48c1 f803 ...a.UH...a.H...
x00404900 4889 e548 89c2 48c1 ea3f 4801 d048 df8 H..H..H..?H..H..
x00404910 7502 5dc3 ba00 0000 0048 85d2 74f4 5d48 u.].....H..t.)H
x00404920 89c6 bfff a561 00ff e20f 1f80 0000 0000 ....a.....
x00404930 803d 215d 2100 0075 1155 4889 e5e8 7eff .=!]!...u.UH...
x00404940 ffff 5dc6 050e 5d21 0001 f3c3 0f1f 4000 .!...]!.....@
x00404950 4883 3da8 5421 0000 741e b800 0000 0048 H.=.T!..t....H
x00404960 85c0 7414 55bf 009e 6100 4889 e5ff d05d ..t.U...a.H....]
x00404970 e97b ffff f0f0 1f00 e973 ffff f0f0 1f00 .{...a.....S....
x00404980 488b 0731 d248 f7f6 4889 d0c3 0f1f 4000 H..1.H..H....@
x00404990 31c0 480b 1648 3917 7406 f3c3 0f1f 4000 1.H..H9.t....@
x004049a0 488b 4608 4839 4708 0f94 c0c3 0f1f 4000 H.F.H9G.t....@
x004049b0 8b05 8266 2100 85c0 7506 893d 7866 2100 ...f!...u...=xf!
x004049c0 f3c3 6666 6666 662e 0f1f 8400 0000 0000 ..fffff.....
x004049d0 e91b d8ff ff66 662e 0f1f 8400 0000 0000 ....ff.....
  
```

Bottom Right Panel (Hex Dump):

```

[0x00404890 16% 115 /bin/ls]> f tmp;sr s.. @ entry0
offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
x00000000 7f45 4c46 0201 0100 0000 0000 0000 0000 .ELF.....
x00000010 0200 3e00 0100 0000 9048 4000 0000 0000 ..>.....H0.....
x00000020 4000 0000 0000 0000 00a7 0100 0000 0000 0.....
x00000030 0000 0000 4000 3800 0900 4000 1c00 1b00 ....@.8...@.....
r15 0x00000000 r14 0x00000000 r13 0x00000000
r12 0x00000000 rbp 0x00000000 rbx 0x00000000
r11 0x00000000 r10 0x00000000 r9 0x00000000
r8 0x00000000 rax 0x00000000 rcx 0x00000000
rdx 0x00000000 rsi 0x00000000 rdi 0x00000000
orax 0x00000000 rip 0x00000000 rflags =
rsp 0x00000000
(fcn) entry0 42
;-- entry0:
0x00404890 31ed      xor ebp, ebp
0x00404892 4989d1    mov r9, rdx
0x00404895 5e        pop rsi
0x00404896 4889e2    mov rdx, rsp
0x00404899 4883e4f0  and rsp, 0xfffffffffffffff0
0x0040489d 50        push rax
0x0040489e 54        push rsp
0x0040489f 49c7c0d01e41. mov r8, 0x411ed0
  
```

Getting Help

To get a help of all the keybindings hooked in visual mode you can press `? :`

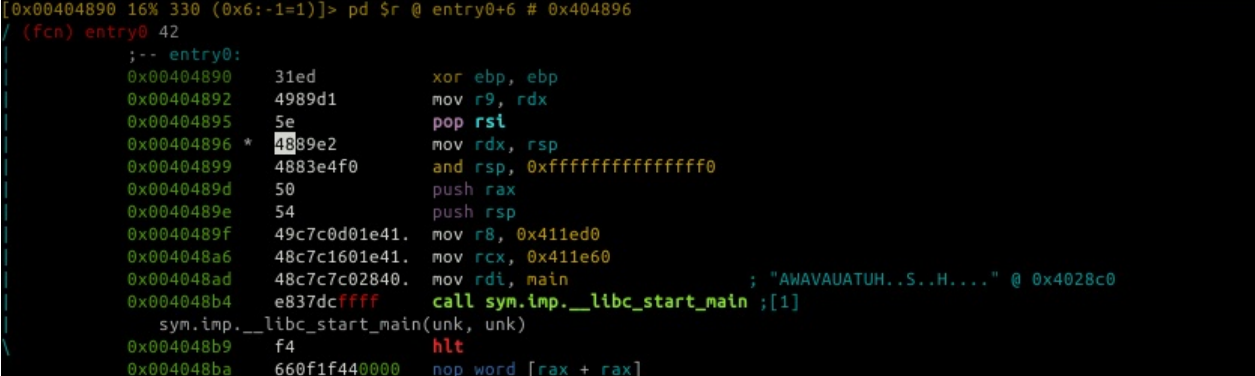
```

Visual mode help:
?      show this help or manpage in cursor mode
_      enter the hud
.      seek to program counter
/      in cursor mode search in current block
:cmd   run radare command
;[-]cmt add/remove comment
/*+[-] change block size, [] = resize hex.cols
>|<   seek aligned to block size
i/a/A  (i)nsert hex, (a)ssemble code, visual (A)ssembler
b/B    toggle breakpoint / automatic block size
c/C    toggle (c)ursor and (C)olors
d[f?]  define function, data, code, ..
D      enter visual diff mode (set diff.from/to)
e      edit eval configuration variables
f/F    set/unset flag
gG     go seek to begin and end of file (0-$s)
  
```

```
h j k l      move around (or H J K L) (left-down-up-right)
mK/'K       mark/go to Key (any key)
M           walk the mounted filesystems
n/N         seek next/prev function/flag/hit (scr.nkey)
o           go/seek to given offset
p/P         rotate print modes (hex, disasm, debug, words, buf)
q           back to radare shell
R           randomize color palette (ecr)
sS          step / step over
t           track flags (browse symbols, functions..)
T           browse anal info and comments
v           visual code analysis menu
V/W         (V)iew graph using cmd.graph (agv?), open (W)ebUI
uU          undo/redo seek
x           show xrefs to seek between them
yY          copy and paste selection
z           toggle zoom mode
Enter       follow address of jump/call
Function Keys: (See 'e key.'), defaults to:
  F2        toggle breakpoint
  F7        single step
  F8        step over
  F9        continue
```

Visual cursor

Pressing lowercase **c** makes the cursor appear or disappear:



```
[0x00404890 16% 330 (0x6:-1=1)]> pd $r @ entry0+6 # 0x404896
(fcn) entry0 42
;-- entry0:
0x00404890 31ed      xor ebp, ebp
0x00404892 4989d1    mov r9, rdx
0x00404895 5e        pop rsi
0x00404896 * 4889e2    mov rdx, rsp
0x00404899 4883e4f0   and rsp, 0xfffffffffffffff0
0x0040489d 50        push rax
0x0040489e 54        push rsp
0x0040489f 49c7c0d01e41. mov r8, 0x411ed0
0x004048a6 48c7c1601e41. mov rcx, 0x411e60
0x004048ad 48c7c7c02840. mov rdi, main ; "AWAVAUATUH..S..H...." @ 0x4028c0
0x004048b4 e837dcfff  call sym.imp.__libc_start_main ;[1]
    sym.imp.__libc_start_main(unk, unk)
0x004048b9 f4        hlt
0x004048ba 660f1f440000 nop word [rax + rax]
```

The cursor is used to select a range of bytes or just point to a byte to flag it (press **f** to create a new flag where the cursor points to)

If you select a range of bytes press **i** and then a byte array to overwrite the selected bytes with the ones you choose in a circular copy way. For example:

```
<select 10 bytes in visual mode using upper hjkl>
<press 'i' and then '12 34'>
```

The 10 bytes selected will become: 12 34 12 34 12 34 12 34 12 34 The byte range selection can be used together with the **d** key to change the data type of the selected bytes into a string, code or a byte array.

That's useful to enhance the disassembly, add metadata or just align the code if there are bytes mixed with code.

In cursor mode you can set the block size by simply moving it to the position you want and pressing **_**. Then change block size.

Visual insert

The insert mode allows you to write bytes at nibble-level like most common hexadecimal editors. In this mode you can press `<tab>` to switch between the hexa and ascii columns of the hexadecimal dump.

To get back to the normal mode, just press `<tab>` to switch to the hexadecimal view and press `q`. (NOTE: if you press `q` in the ascii view...it will insert a `q` instead of quit this mode)

There are other keys for inserting and writing data in visual mode. Basically by pressing `i` key you'll be prompted for an hexpair string or use `a` for writing assembly where the cursor points.

Visual xrefs

radare implements many user-friendly features for the visual interface to walk thru the assembly code. One of them is the `x` key that pops up a menu for selecting the xref (data or code) against the current seek and then jump there. For example when pressing x when looking at those XREF:

```
| ....--> ; CODE (CALL) XREF from 0x00402b98 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402ba0 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402ba9 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402bd5 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402beb (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402c25 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402c31 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402c40 (fcn.004028d0)
| ....--> ; CODE (CALL) XREF from 0x00402c51 (fcn.004028d0)
```

After pressing `x`

```
[GOTO XREF]>
[0] CODE (CALL) XREF 0x00402b98 (loc.00402b38)
[1] CODE (CALL) XREF 0x00402ba0 (loc.00402b38)
[2] CODE (CALL) XREF 0x00402ba9 (loc.00402b38)
[3] CODE (CALL) XREF 0x00402bd5 (loc.00402b38)
[4] CODE (CALL) XREF 0x00402beb (loc.00402b38)
[5] CODE (CALL) XREF 0x00402c25 (loc.00402b38)
[6] CODE (CALL) XREF 0x00402c31 (loc.00402b38)
[7] CODE (CALL) XREF 0x00402c40 (loc.00402b38)
[8] CODE (CALL) XREF 0x00402c51 (loc.00402b38)
[9] CODE (CALL) XREF 0x00402c60 (loc.00402b38)
```

All the calls and jumps are numbered (1, 2, 3...) these numbers are the keybindings for seeking there from the visual mode. All the seek history is stored, by pressing `u` key you will go back in the seek history time :)

Visual Configuration editor

`ve` or `e` in Visual mode allows you to edit radare2 configuration visually. For example if you want to change the assembly display just select `asm` in the list and choose your assembly display flavor.

```
[EvalSpace]
  anal
>  asm
  bin
  cfg
  cmd
  dbg
  diff
  dir
  esil
  file
  fs
  graph
  hex
  http
  hud
  io
  key
  magic
  pdb
  rap
  rop
  scr
  search
  stack
  time
  zoom

Sel:asm.arch

/ (fcn) entry0 42
|      |-- entry0:
|      0x00404890  31ed          xor ebp, ebp
|      0x00404892  4989d1        mov r9, rdx
|      0x00404895  5e           pop rsi
|      0x00404896  4889e2        mov rdx, rsp
|      0x00404899  4883e4f0      and rsp, 0xfffffffffffffff0
```

Example switch to pseudo disassembly:

```
[EvalSpace < Variables: asm.arch]

asm.functions = true
asm.indent = false
asm.lbytes = true
asm.lines = true
asm.linescall = false
asm.linesout = true
asm.linesright = false
asm.linesstyle = false
asm.lineswide = false
asm.lineswidth = 7
asm.maxrefs = 5
asm.middle = false
asm.nbytes = 6
asm.offset = true
asm.os = linux
asm.parser = x86.pseudo
> asm.pseudo = false
asm.reloff = false
asm.section = false
asm.segoff = false
asm.size = false
asm.stackptr = false
asm.syntax = intel
asm.tabs = 0
asm.trace = false
asm.tracespace = false
asm.ucas = false
asm.vars = true
asm.varsub = true
asm.varxs = false
asm.xrefs = true

Selected: asm.pseudo (Enable pseudo syntax)

/ (fcn) entry0 42
|      ;-- entry0:
|      0x00404890      31ed      xor ebp, ebp
|      0x00404892      4989d1     mov r9, rdx
|      0x00404895      5e      pop rsi
|      0x00404896      4889e2     mov rdx, rsp
|      0x00404899      4883e4f0    and rsp, 0xfffffffffffffff0
```

```
[EvalSpace < Variables: asm.arch]
```

```
asm.functions = true
asm.indent = false
asm.lbytes = true
asm.lines = true
asm.linescall = false
asm.linesout = true
asm.linesright = false
asm.linesstyle = false
asm.lineswide = false
asm.lineswidth = 7
asm.maxrefs = 5
asm.middle = false
asm.nbytes = 6
asm.offset = true
asm.os = linux
asm.parser = x86.pseudo
> asm.pseudo = true
asm.reloff = false
asm.section = false
asm.segoff = false
asm.size = false
asm.stackptr = false
asm.syntax = intel
asm.tabs = 0
asm.trace = false
asm.tracespace = false
asm.ucas = false
asm.vars = true
asm.varsub = true
asm.varxs = false
asm.xrefs = true
```

Selected: asm.pseudo (Enable pseudo syntax)

```
(fcn) entry0 42
;-- entry0:
0x00404890 31ed          ebp = 0
0x00404892 4989d1        r9 = rdx
0x00404895 5e           pop rsi
0x00404896 4889e2        rdx = rsp
0x00404899 4883e4f0      rsp &= 0xffffffffffffffff0
```


Searching bytes

The search engine of radare is based on the work done by esteve plus multiple features on top of it that allows multiple keyword searching with binary masks and automatic flagging of results.

This powerful command is `/` .

```
[0x00000000]> /
Usage: /[amx/] [arg]
/ foo\x00      search for string `foo\0`
/w foo         search for wide string `f\0o\0o\0`
/wi foo        search for wide string ignoring case `f\0o\0o\0`
/! ff          search for first occurrence not matching
/i foo         search for string `foo` ignoring case
/e /E.F/i      match regular expression
/x ff0033      search for hex string
/x ff..33      search for hex string ignoring some nibbles
/x ff43 ffd0   search for hexpair with mask
/d 101112      search for a deltified sequence of bytes
/!x 00         inverse hexa search (find first byte != 0x00)
/c jmp [esp]   search for asm code (see search.asmstr)
/a jmp eax     assemble opcode and search its bytes
/A            search for AES expanded keys
/r sym.printf  analyze opcode reference an offset
/R            search for ROP gadgets
/P            show offset of previous instruction
/m magicfile   search for matching magic file (use blocksize)
/p patternsize search for pattern of given size
/z min max     search for strings of given size
/v[?248] num   look for a asm.bigendian 32bit value
//            repeat last search
/b            search backwards
```

With radare everything is handled as a file, it doesn't matters if it is a socket, a remote device, the process memory, etc..

Basic searches

A basic search for a plain string in a whole file would be something like:

```
$ r2 -c "/" lib" -q /bin/ls
Searching 3 bytes from 0x00400000 to 0x0041ae08: 6c 69 62
hits: 9
0x00400239 hit0_0 "lib64/ld-linux-x86-64.so.2"
0x00400f19 hit0_1 "libselinux.so.1"
0x00400fae hit0_2 "librt.so.1"
0x00400fc7 hit0_3 "libacl.so.1"
0x00401004 hit0_4 "libc.so.6"
0x004013ce hit0_5 "libc_start_main"
0x00416542 hit0_6 "libs/"
0x00417160 hit0_7 "lib/xstrtol.c"
0x00417578 hit0_8 "lib"
```

`r2 -q` // quiet mode (no prompt) and quit after -i

As you can see, radare generates a `hit` flag for each search result found. You can just use the `ps` command to visualize the strings at these offsets in this way:

```
[0x00404888]> / ls
...
[0x00404888]> ps @ hit0_0
lseek
```

We can also search wide-char strings (the ones containing zeros between each letter) using the `/w` in this way:

```
[0x00000000]> /w Hello
0 results found.
```

Search for strings ignoring case

```
[0x0040488f]> /i Stallman
Searching 8 bytes from 0x00400238 to 0x0040488f: 53 74 61 6c 6c 6d 61 6e
[# ]hits: 004138 < 0x0040488f hits = 0
```

It is also possible to mix hexadecimal escape sequences in the search string:

```
[0x00000000]> / \x7FELF
```

But if you want to perform an hexadecimal search you will probably prefer an hexpair input with `/x` :

```
[0x00000000]> /x 7F454C46
```

Once the search is done, the results are stored in the `search` flag space.

```
[0x00000000]> f
0x00000135 512 hit0_0
0x00000b71 512 hit0_1
0x00000bad 512 hit0_2
0x00000bdd 512 hit0_3
0x00000bfb 512 hit0_4
0x00000f2a 512 hit0_5
```

To remove these flags, you can just use the `f@-hit*` command.

Sometimes while working long time in the same file you will need to launch the last search more than once and you will probably prefer to use the `//` command instead of typing all the string again.

```
[0x00000f2a]> //      ; repeat last search
```

Configurating the searches

The search engine can be configured by the `e` interface:

```
Configuration:
e cmd.hit = x          ; command to execute on every search hit
e search.distance = 0  ; search string distance
e search.in = [foo]    ; boundaries to raw, block, file, section)
e search.align = 4     ; only catch aligned search hits
e search.from = 0      ; start address
e search.to = 0        ; end address
e search.asmstr = 0    ; search string instead of assembly
e search.flags = true  ; if enabled store flags on keyword hits
```

`search.align` variable is used to determine that the only `valid` search hits must have to fit in this alignment. For example. you can use `e search.align=4` to get only the hits found in 4-byte aligned addresses.

The `search.flag` boolean variable makes the engine setup flags when finding hits. If the search is stopped by the user with a `^C` then a `search_stop` flag will be added.

Pattern search

The search command allows you to throw repeated pattern searches against the IO backend to be able to identify repeated sequences of bytes without specifying them. The only property to perform this search is to manually define the minimum length of these patterns.

Here`s an example:

```
[0x00000000]> /p 10
```

The output of the command will show the different patterns found and how many times they are repeated.

Automatization

The `cmd.hit` eval variable is used to define a command that will be executed when a hit is reached by the search engine. If you want to run more than one command use `;` or `. script-file-name` for including a file as a script.

For example:

```
[0x00404888]> e cmd.hit = p8 8
[0x00404888]> / lib
Searching 3 bytes from 0x00400000 to 0x0041ae08: 6c 69 62
hits: 9
0x00400239 hit4_0 "lib64/ld-linux-x86-64.so.2"
31ed4989d15e4889
0x00400f19 hit4_1 "libselinux.so.1"
31ed4989d15e4889
0x00400fae hit4_2 "librt.so.1"
31ed4989d15e4889
0x00400fc7 hit4_3 "libacl.so.1"
31ed4989d15e4889
0x00401004 hit4_4 "libc.so.6"
31ed4989d15e4889
0x004013ce hit4_5 "libc_start_main"
31ed4989d15e4889
0x00416542 hit4_6 "libs/"
31ed4989d15e4889
0x00417160 hit4_7 "lib/xstrtol.c"
31ed4989d15e4889
0x00417578 hit4_8 "lib"
31ed4989d15e4889
```

Backward search

To search backward just use `/b`

Search in assembly

If you want to search for a certain type of opcodes you can either use `/c` or `/a` :

```
/c jmp [esp]      search for asm code
```

```
[0x00404888]> /c jmp qword [rdx]
f hit_0 @ 0x0040e50d  # 2: jmp qword [rdx]
f hit_1 @ 0x00418dbb  # 2: jmp qword [rdx]
f hit_2 @ 0x00418fcb  # 3: jmp qword [rdx]
f hit_3 @ 0x004196ab  # 6: jmp qword [rdx]
f hit_4 @ 0x00419bf3  # 3: jmp qword [rdx]
f hit_5 @ 0x00419c1b  # 3: jmp qword [rdx]
f hit_6 @ 0x00419c43  # 3: jmp qword [rdx]
```

```
/a jmp eax        assemble opcode and search its bytes
```

```
[0x00404888]> /a jmp eax
hits: 1
0x004048e7 hit3_0 ffe00f1f8000000000b8
```


Searching AES keys

Thanks to Victor Muñoz i have added support to the algorithm he developed to find expanded AES keys. It runs the search from the current seek to the `cfg.limit` or the end of the file. You can always stop the search pressing `^C`.

```
$ sudo r2 /dev/mem  
[0x00000000]> /Ca  
0 AES keys found
```

Disassembling

Disassembling in radare is just a way to represent a bunch of bytes. So it is handled as a print mode with the 'p' command.

In the old times when radare core was smaller. The disassembler was handled by an external rsc file, so radare was dumping the current block into a file, and the script was just calling objdump in a proper way to disassemble for intel, arm, etc...

Obviously this is a working solution, but takes too much cpu for repeating just the same task so many times, because there are no caches and the scrolling was absolutely slow.

Nowadays, the disassembler is one of the basics in radare allowing you to choose the architecture flavour and some To disassemble use the 'pd' command.

The 'pd' command accepts a numeric argument to specify how many opcodes of the current block do you want to disassemble. Most of the commands in radare are restricted by the block size. So if you want to disassemble more bytes you should use the 'b' command to specify the new block size.

```
[0x00000000]> b 100      ; set block size to 100
[0x00000000]> pd         ; disassemble 100 bytes
[0x00000000]> pd 3       ; disassemble 3 opcodes
[0x00000000]> pD 30      ; disassemble 30 bytes
```

The 'pD' command works like 'pd' but gets the number of bytes instead of the number of opcodes.

The 'pseudo' syntax is closer to the humans, but it can be annoying if you are reading lot of code:

```
[0x00405e1c]> e asm.pseudo = true
[0x00405e1c]> pd 3
      ; JMP XREF from 0x00405dfa (fcn.00404531)
0x00405e1c  488b9424a80. rdx = [rsp+0x2a8]
0x00405e24  64483314252. rdx ^= [fs:0x28]
0x00405e2d  4889d8      rax = rbx

[0x00405e1c]> e asm.syntax = intel
[0x00405e1c]> pd 3
      ; JMP XREF from 0x00405dfa (fcn.00404531)
0x00405e1c  488b9424a80. mov rdx, [rsp+0x2a8]
0x00405e24  64483314252. xor rdx, [fs:0x28]
0x00405e2d  4889d8      mov rax, rbx

[0x00405e1c]> e asm.syntax=att
[0x00405e1c]> pd 3
      ; JMP XREF from 0x00405dfa (fcn.00404531)
0x00405e1c  488b9424a80. mov 0x2a8(%rsp), %rdx
0x00405e24  64483314252. xor %fs:0x28, %rdx
0x00405e2d  4889d8      mov %rbx, %rax
```

Adding metadata

The work on binary files makes the task of taking notes and defining information on top of the file quite important. Radare offers multiple ways to retrieve and acquire this information from many kind of file types.

Following some *nix principles becomes quite easy to write a small utility in shellscript that using `objdump`, `otool`, etc.. to get information from a binary and import it into radare just making echo's of the commands script.

You can have a look on one of the many scripts that are distributed with radare like 'idc2r.py':

This script is called with 'idc2r.py file.idc > file.r2'. It reads an IDC file exported from an IDA database and imports the comments and the names of the functions.

We can import the 'file.r2' using the '.' command of radare (similar to the shell):

```
[0x00000000]> . file.r2
```

The command '.' is used to interpret data from external resources like files, programs, etc.. In the same way we can do the same without writing a file.

```
[0x00000000]> .!idc2r.py < file.idc
```

The 'C' command is the one used to manage comments and data conversions. So you can define a range of bytes to be interpreted as code, or a string. It is also possible to define flags and execute code in a certain seek to fetch a comment from an external file or database.

Here's the help:

```
[0x00404cc0]> C?
|Usage: C[-LCvsdfm?] [...]
| C*                               List meta info in r2 commands
| C- [len] [@][ addr]             delete metadata at given address range
| CL[-] [addr|file:line [addr] ]  show 'code line' information (bininfo)
| Cl file:line [addr]             add comment with line information
| CC[-] [comment-text]           add/remove comment. Use CC! to edit with $EDITOR
| CCa[-at][[at] [text]           add/remove comment at given address
| Cv[-] offset reg name          add var substitution
| Cs[-] [size] [[addr]]          add string
| Ch[-] [size] [@addr]           hide data
| Cd[-] [size]                   hexdump data
| Cf[-] [sz] [fmt..]             format memory (see pf?)
| Cm[-] [sz] [fmt..]             magic parse (see pm?)
[0x00404cc0]>
```

```
[0x00000000]> CCa 0x00000002 this guy seems legit
```

```
[0x00000000]> pd 2
      0x00000000    0000    add [rax], al
;      this guy seems legit
      0x00000002    0000    add [rax], al
```

The 'C' command allows us to change the type of data. The three basic types are: code (disassembly using asm.arch), data (byte array) or string.

In visual mode is easier to manage this because it is hooked to the 'd' key trying to mean 'data type change'. Use the cursor to select a range of bytes ('c' key to toggle cursor mode and HJKL to move with selection) and then press 'ds' to convert to string.

You can use the Cs command from the shell also:

```
[0x00000000]> f string_foo @ 0x800
[0x00000000]> Cs 10 @ string_foo
```

The folding/unfolding is quite premature but the idea comes from the 'folder' concepts in vim. So you can select a range of bytes in the disassembly view and press '<' to fold these bytes in a single line or '>' to unfold them. Just to ease the readability of the code.

The Cm command is used to define a memory format string (the same used by the pf command). Here's an example:

```
[0x7fd9f13ae630]> Cf 16 2xi foo bar
[0x7fd9f13ae630]> pd
      ;-- rip:
      0x7fd9f13ae630 format 2xi foo bar {
0x7fd9f13ae630 [0] {
  foo : 0x7fd9f13ae630 = 0xe8e78948
  bar : 0x7fd9f13ae634 = 14696
}
0x7fd9f13ae638 [1] {
  foo : 0x7fd9f13ae638 = 0x8bc48949
  bar : 0x7fd9f13ae63c = 571928325
}
} 16
      0x7fd9f13ae633    e868390000    call 0x7fd9f13b1fa0
      0x7fd9f13b1fa0() ; rip
      0x7fd9f13ae638    4989c4      mov r12, rax
```

This way it is possible to define structures by just using simple oneliners. See 'print memory' for more information.

All those C* commands can also be accessed from the visual mode by pressing 'd' (data conversion) key.

ESIL

From the wiki page of radare2 github

ESIL stands for 'Evaluable Strings Intermediate Language'. It aims to describe a Forth-like representation for every opcode. Those representations can be evaluated in order to emulate code. Each element of an esil expression is separated by a comma. The VM can be described as this:

```
while ((word=haveCommand())) {
  if (word.isKeyword()) {
    esilCommands[word](esil);
  } else {
    esil.push (evaluateToNumber(word));
  }
  nextCommand();
}
```

The esil commands are operations that pop values from the stack, performs some calculations and pushes the result in the stack (if any). They aim to cover all common operations done by CPUs, permitting to do binary operations, memory peeks and pokes, spawning a syscall, etc.

Use ESIL

```
[0x00000000]> e asm.esil = true
```

Syntax

An opcode is translated into a comma separated list of ESIL expressions.

```
xor eax, eax    ->    0,eax,=,1,zf,=
```

Memory access is defined by brackets.

```
mov eax, [0x80480]    ->    0x80480,[],eax,=
```

Default size is the destination of the operation. In this case 8bits, aka 1 byte.

```
movb $0, 0x80480    ->    0,0x80480,=[1]
```

Conditionals are expressed with the '?' char at the beginning of the expression. this checks if the rest of the expression is 0 or not and skips the next expression if doesn't matches. % is the prefix for internal vars.

```
cmp eax, 123  -> 123, eax, ==, %z, zf, =
jz  eax      -> zf, ?{, eax, eip, =, }
```

So.. if you want to run more than one expression under a conditional, you'll have to write it

```
zf, ?{, eip, esp, =[], eax, eip, =, %r, esp, -=, }
```

The whitespace, newlines and other chars are ignored in esil, so the first thing to do is:

```
esil = r_str_replace (esil, " ", "", R_TRUE);
```

Syscalls are specially handled by '\$' at the beginning of the expression. After that char you have an optional numeric value that specifies the number of syscall. The emulator must handle those expressions and 'simulate' the syscalls. (r_esil_syscall)

Order of arguments

As discussed on irc, current implementation works like this:

```
a, b, -      b - a
a, b, /=     b /= a
```

This approach is more readable, but it's less stack-friendly

Special instructions

NOPs are represented as empty strings. Unknown or invalid instructions

Syscalls are implemented with the '0x80,\$' command. It delegates the execution of the esil vm into a callback that implements the syscall for a specific kernel.

Traps are implemented with the `<trap>, <code>, $$` command. They are used to throw exceptions like invalid instructions, division by zero, memory read error, etc.

Quick analysis

Here's a list of some quick checks to retrieve information from an esil string. Relevant information will be probably found in the first expression of the list.

```
indexOf([''])  -> have memory references
indexOf("=[")  -> write in memory
indexOf("pc,=") -> modifies program counter (branch, jump, call)
indexOf("sp,=") -> modifies the stack (what if we found sp+= or sp-=?)
```

```

indexOf("=")      -> retrieve src and dst
indexOf(":")      -> unknown esil, raw opcode ahead
indexOf("%")      -> accesses internal esil vm flags
indexOf("$")      -> syscall
indexOf("$")      -> can trap
indexOf('++')     -> has iterator
indexOf('--')     -> count to zero
indexOf("?{")     -> conditional
indexOf("LOOP")   -> is a loop (rep?)
equalsTo("")      -> empty string, means: nop (wrong, if we append pc+=x)

```

Common operations:

- Check dstreg
- Check srcreg
- Get destinaion
- Is jump
- Is conditional
- Evulate
- Is syscall

CPU Flags

CPU flags are usually defined as 1 bit registers in the RReg profile. and sometimes under the 'flg' register type.

ESIL Flags

ESIL VM have an internal state flags that can are read only and can be used to export those values to the underlaying CPU flags. This is because the ESIL vm defines all the flag changes, while the CPUs only update the flags under certain conditions or specific instructions.

Those internal flags are prefixed by the '%' character.

```

z - zero flag, only set if the result of an operation is 0
b - borrow, this requires to specify from which bit (example: %b4 - checks if borrow from
c - carry, same like above (example: %c7 - checks if carry from bit 7)
p - parity
r - regsize ( asm.bits/8 )

```

Variables

1. No predefined bitness (should be easy to extend them to 128,256 and 512bits, e.g. for MMX, SSE, AVX, Neon)
2. Infinite number (for SSA-form compatibility)
3. Register names have no specific syntax. They are just strings

4. Numbers can be specified in any base supported by RNum (dec, hex, oct, binary ...)
5. Each ESIL backend should have an associated RReg profile to describe the esil register specs

Bitarrays

What to do with them? What about bit arithmetics if use variables instead of registers?

Arithmetics

1. ADD ("+")
2. MUL ("*")
3. SUB ("-")
4. DIV ("/")
5. MOD ("%")

Bit arithmetics

1. AND "&"
2. OR "|"
3. XOR "^"
4. SHL "<<"
5. SHR ">>"
6. ROL "<<<"
7. ROR ">>>"
8. NEG "!"

Floating point

TODO

The x86 REP prefix in ESIL

ESIL specifies that the parsing control-flow commands are in uppercase. Bear in mind that some archs have uppercase register names. The register profile should take care to not reuse any of the following:

```
3,SKIP    - skip N instructions. used to make relative forward GOTOs
3,GOTO    - goto instruction 3
LOOP      - alias for 0,GOTO
BREAK     - stop evaluating the expression
STACK     - dump stack contents to screen
CLEAR     - clear stack
```

Usage example:

rep cmpsb

```
cx,!,?{,BREAK,},esi,[1],edi,[1],==,?{,BREAK,},esi,++,edi,++,cx,--,LOOP
```

Unimplemented/unhandled instructions

Those are expressed with the 'TODO' command. which acts as a 'BREAK', but displaying a warning message describing which instruction is not implemented and will not be emulated.

For example:

```
fmulp ST(1), ST(0)      =>      TODO,fmulp ST(1),ST(0)
```

Disassembly example:

```
[0x1000010f8]> e asm.esil=true
[0x1000010f8]> pd $r @ entry0
;      [0] va=0x1000010f8 pa=0x000010f8 sz=13299 vsz=13299 rwx=-r-x 0.__text
; -- section.0.__text:
0x1000010f8    55          8, rsp, -=, rbp, rsp, =[8]
0x1000010f9    4889e5        rsp, rbp, =
0x1000010fc    4883c768      104, rdi, +=
0x100001100    4883c668      104, rsi, +=
0x100001104    5d          rsp, [8], rbp, =, 8, rsp, +=
| 0x10000110a    55          8, rsp, -=, rbp, rsp, =[8]
| 0x10000110b    4889e5        rsp, rbp, =
| 0x100001112    488d7768      rdi, 104, +, rsi, =
| 0x100001116    4889c7        rax, rdi, =
| 0x100001119    5d          rsp, [8], rbp, =, 8, rsp, +=
|| 0x10000111f    55          8, rsp, -=, rbp, rsp, =[8]
|| 0x100001120    4889e5        rsp, rbp, =
|| 0x100001123    488b4f60      rdi, 96, +, [8], rcx, =
|| 0x100001127    4c8b4130      rcx, 48, +, [8], r8, =
|| 0x10000112f    b801000000    1, eax, = ; 0x00000001
|| 0x100001134    4c394230      rdx, 48, +, [8], r8, ==, cz, ?=
|< 0x100001138    7f1a         sf, of, !, ^, zf, !, &, ?{, 0x1154, rip, =, } ; [2]
|< 0x10000113a    7d07         of, !, sf, ^, ?{, 0x1143, rip, } ; [3]
| || 0x10000113c    b8ffffff      0xffffffff, eax, = ; 0xffffffff
|> 0x100001143    488b4938      rcx, 56, +, [8], rcx, =
| || 0x100001147    48394a38      rdx, 56, +, [8], rcx, ==, cz, ?=
```

Introspection

To ease esil parsing we should have a way to express introspection expressions to extract the data we want. For example. We want to get the target address of a jmp.

The parser for the esil expressions should be implemented in an API to make it possible to extract

information by analyzing the expressions easily.

```
> ao-esil,opcode
opcode: jmp 0x10000465a
esil: 0x10000465a,rip,=
```

We need a way to retrieve the numeric value of 'rip'. This is a very simple example, but there will be more complex, like conditional ones and we need expressions to get:

- opcode type
- destination of jump
- condition depends on
- all regs modified (write)
- all regs accessed (read)

API HOOKS

It is important for emulation to be able to setup hooks in the parser, so we can extend the parser to implement the analysis without having to write the parser again and again. This is, every time an operation is going to be executed we call a user hook which can be used to determine if rip is changing or if the instruction updates the stack. Later, at this level we can split that callback into several ones to have an event based analysis api that may be extended in js like this: `esil.on('regset', function(){..`
`esil.on('syscall', function(){esil.regset('rip'`

we have already them. see `hook_flag_read()` `hook_execute()` `hook_mem_read()` ...

return true if you want to override the action taken for a callback. for example. avoid mem reads in a region or mem writes to make all memory read only.

return false or 0 if you want to trace esil expression parsing. aka emulation ..

Other operations that require bindings to external functionalities to work. In this case `r_ref` and `r_io`. This must be defined when initializing the esil vm.

- Io Get/Set Out ax, 44 44,ax,:ou
- Selectors (cs,ds,gs...) Mov eax, ds:[ebp+8] Ebp,8,+, :ds,eax,=

Rabin2

Under this bunny-arabic-like name, radare hides the power of a wonderful tool to handle binary files and get information to show it in the command line or import it into the core.

Rabin2 is able to handle multiple file formats like Java CLASS, ELF, PE, MACH-O, etc.. and it is able to get symbol import/exports, library dependencies, strings of data sections, xrefs, address of entrypoint, sections, architecture type, etc.

```
$ rabin2 -h

Usage: rabin2 [-ACdehHiIjLLMqRrRsSvVxZZ] [-@ addr] [-a arch] [-b bits]
              [-B addr] [-c F:C:D] [-f str] [-m addr] [-n str] [-N len]
              [-o str] [-O str] file
  -@ [addr]      show section, symbol or import at addr
  -A             list archs
  -a [arch]      set arch (x86, arm, .. or <arch>_<bits>)
  -b [bits]      set bits (32, 64 ...)
  -B [addr]      override base address (pie bins)
  -c [fmt:C:D]   create [elf,mach0,pe] with Code and Data hexpairs (see -a)
  -C            list classes
  -d            show debug/dwarf information
  -e            entrypoint
  -f [str]       select sub-bin named str
  -g            same as -SMRevsiz (show all info)
  -h            this help
  -H            header fields
  -i            imports (symbols imported from libraries)
  -I            binary info
  -j            output in json
  -l            linked libraries
  -L            list supported bin plugins
  -m [addr]      show source line at addr
  -M            main (show address of main symbol)
  -n [str]       show section, symbol or import named str
  -N [minlen]    force minimum number of chars per string (see -z)
  -o [str]       output file/folder for write operations (out by default)
  -O [str]       write/extract operations (-O help)
  -q            be quiet, just show fewer data
  -r            radare output
  -R            relocations
  -s            symbols (exports)
  -S            sections
  -v            use vaddr in radare output (or show version if no file)
  -x            extract bins contained in file
  -z            strings (from data section)
  -zz           strings (from raw bins [e bin.rawstr=1])
  -Z            guess size of binary program
```

File identification

The file identification is done through the `-I` flag, it will output information regarding binary class, encoding, OS, type, etc.

```
$ rabin2 -I /bin/ls
file      /bin/ls
type      EXEC (Executable file)
pic       false
has_va    true
root      elf
class     ELF64
lang      c
arch      x86
bits      64
machine   AMD x86-64 architecture
os        linux
subsys    linux
endian    little
strip     true
static    false
linenum   false
lsyms     false
relocs    false
rpath     NONE
```

As it was said we can add the `-r` flag to use all this information in radare:

```
$ rabin2 -Ir /bin/ls
e file.type=elf
e cfg.bigendian=false
e asm.os=linux
e asm.arch=x86
e anal.arch=x86
e asm.bits=64
e asm.dwarf=true
```

Entrypoint

The flag "-e" lets us know the program entrypoint

```
$ rabin2 -e /bin/ls
[Entrypoints]
addr=0x00004888 off=0x00004888 baddr=0x00000000

1 entrypoints

$ rabin2 -er /bin/ls
fs symbols
f entry0 @ 0x00004888
s entry0
```

Imports

Rabin2 is able to get all the imported objects, as well as their offset at the PLT, this information is quite useful, for example, to recognize which function is called by a call instruction.

```
$ rabin2 -i /bin/ls |head
[Imports]
ordinal=001 plt=0x000021b0 bind=GLOBAL type=FUNC name=__ctype_toupper_loc
ordinal=002 plt=0x000021c0 bind=GLOBAL type=FUNC name=__uflow
ordinal=003 plt=0x000021d0 bind=GLOBAL type=FUNC name=getenv
ordinal=004 plt=0x000021e0 bind=GLOBAL type=FUNC name=sigprocmask
ordinal=005 plt=0x000021f0 bind=GLOBAL type=FUNC name=raise
ordinal=006 plt=0x00002210 bind=GLOBAL type=FUNC name=localtime
ordinal=007 plt=0x00002220 bind=GLOBAL type=FUNC name=__mempcpy_chk
ordinal=008 plt=0x00002230 bind=GLOBAL type=FUNC name=abort
ordinal=009 plt=0x00002240 bind=GLOBAL type=FUNC name=__errno_location
(...)
```

Symbols (exports)

In rabin, symbols list works in a very similar way as exports do.

```
$ rabin2 -s /bin/ls | head
[Symbols]
addr=0x0021a610 off=0x0021a610 ord=114 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=stdout
addr=0x0021a600 off=0x0021a600 ord=115 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=_edata
addr=0x0021b388 off=0x0021b388 ord=116 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=_end
addr=0x0021a600 off=0x0021a600 ord=117 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=__progn
addr=0x0021a630 off=0x0021a630 ord=119 fwd=NONE sz=8 bind=UNKNOWN type=OBJECT name=progra
addr=0x0021a600 off=0x0021a600 ord=121 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=__bss_s
addr=0x0021a630 off=0x0021a630 ord=122 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=__progn
addr=0x0021a600 off=0x0021a600 ord=123 fwd=NONE sz=8 bind=UNKNOWN type=OBJECT name=progra
addr=0x00002178 off=0x00002178 ord=124 fwd=NONE sz=0 bind=GLOBAL type=FUNC name=_init
```

With -r radare core can flag automatically all these symbols and define function and data blocks.

```
$ rabin2 -sr /bin/ls

fs symbols
Cd 8 @ 0x0021a610
f sym.stdout 8 0x0021a610
f sym._edata 0 0x0021a600
f sym._end 0 0x0021b388
Cd 8 @ 0x0021a600
f sym.__progrname 8 0x0021a600
Cd 8 @ 0x0021a630
f sym.program_invocation_name 8 0x0021a630
f sym.__bss_start 0 0x0021a600
```

Libraries

Rabin2 can list the libraries used by a binary with the flag -l.

```
$ rabin2 -l /bin/ls
[Linked libraries]
libselinux.so.1
librt.so.1
libacl.so.1
libc.so.6

4 libraries
```

If you compare the output of 'rabin2 -l' and 'ldd' you will notice that rabin will list less libraries than 'ldd'. The reason is that rabin will not follow the dependencies of the listed libraries, it will just display the ones listed in the binary itself.

Strings

The `-z` flag is used to list all the strings located in the section `.rodata` for ELF binaries, and `.text` for PE ones.

```
$ rabin2 -z /bin/ls |head
addr=0x00012487 off=0x00012487 ordinal=000 sz=9 len=9 section=.rodata type=A string=src/l
addr=0x00012490 off=0x00012490 ordinal=001 sz=26 len=26 section=.rodata type=A string=sor
addr=0x000124aa off=0x000124aa ordinal=002 sz=5 len=5 section=.rodata type=A string= %lu
addr=0x000124b0 off=0x000124b0 ordinal=003 sz=7 len=14 section=.rodata type=W string=%*lu
addr=0x000124ba off=0x000124ba ordinal=004 sz=8 len=8 section=.rodata type=A string=%s %*
addr=0x000124c5 off=0x000124c5 ordinal=005 sz=10 len=10 section=.rodata type=A string=%*s
addr=0x000124cf off=0x000124cf ordinal=006 sz=5 len=5 section=.rodata type=A string= ->
addr=0x000124d4 off=0x000124d4 ordinal=007 sz=17 len=17 section=.rodata type=A string=can
addr=0x000124e5 off=0x000124e5 ordinal=008 sz=29 len=29 section=.rodata type=A string=can
addr=0x00012502 off=0x00012502 ordinal=009 sz=10 len=10 section=.rodata type=A string=unl
```

With `-r` all this information is converted to radare2 commands, which will create a flag space called "strings" filled with flags for all those strings. Furthermore, it will redefine them as strings insted of code.

```
$ rabin2 -zr /bin/ls |head
fs strings
f str.src_ls.c 9 @ 0x00012487
Cs 9 @ 0x00012487
f str.sort_type__sort_version 26 @ 0x00012490
Cs 26 @ 0x00012490
f str._lu 5 @ 0x000124aa
Cs 5 @ 0x000124aa
f str.__lu_ 14 @ 0x000124b0
Cs 7 @ 0x000124b0
f str._s__s 8 @ 0x000124ba
(...)
```

Program sections

Rabin2 give us complete information about the program sections. We can know their index, offset, size, align, type and permissions, as we can see in the next example.

```
$ rabin2 -S /bin/ls
[Sections]
idx=00 addr=0x00000238 off=0x00000238 sz=28 vsz=28 perm=-r-- name=.interp
idx=01 addr=0x00000254 off=0x00000254 sz=32 vsz=32 perm=-r-- name=.note.ABI_tag
idx=02 addr=0x00000274 off=0x00000274 sz=36 vsz=36 perm=-r-- name=.note.gnu.build_id
idx=03 addr=0x00000298 off=0x00000298 sz=104 vsz=104 perm=-r-- name=.gnu.hash
idx=04 addr=0x00000300 off=0x00000300 sz=3096 vsz=3096 perm=-r-- name=.dynsym
idx=05 addr=0x00000f18 off=0x00000f18 sz=1427 vsz=1427 perm=-r-- name=.dynstr
idx=06 addr=0x000014ac off=0x000014ac sz=258 vsz=258 perm=-r-- name=.gnu.version
idx=07 addr=0x000015b0 off=0x000015b0 sz=160 vsz=160 perm=-r-- name=.gnu.version_r
idx=08 addr=0x00001650 off=0x00001650 sz=168 vsz=168 perm=-r-- name=.rela.dyn
idx=09 addr=0x000016f8 off=0x000016f8 sz=2688 vsz=2688 perm=-r-- name=.rela.plt
idx=10 addr=0x00002178 off=0x00002178 sz=26 vsz=26 perm=-r-x name=.init
idx=11 addr=0x000021a0 off=0x000021a0 sz=1808 vsz=1808 perm=-r-x name=.plt
idx=12 addr=0x000028b0 off=0x000028b0 sz=64444 vsz=64444 perm=-r-x name=.text
idx=13 addr=0x0001246c off=0x0001246c sz=9 vsz=9 perm=-r-x name=.fini
idx=14 addr=0x00012480 off=0x00012480 sz=20764 vsz=20764 perm=-r-- name=.rodata
idx=15 addr=0x0001759c off=0x0001759c sz=1820 vsz=1820 perm=-r-- name=.eh_frame_hdr
idx=16 addr=0x00017cb8 off=0x00017cb8 sz=8460 vsz=8460 perm=-r-- name=.eh_frame
idx=17 addr=0x00019dd8 off=0x00019dd8 sz=8 vsz=8 perm=-rw- name=.init_array
idx=18 addr=0x00019de0 off=0x00019de0 sz=8 vsz=8 perm=-rw- name=.fini_array
idx=19 addr=0x00019de8 off=0x00019de8 sz=8 vsz=8 perm=-rw- name=.jcr
idx=20 addr=0x00019df0 off=0x00019df0 sz=512 vsz=512 perm=-rw- name=.dynamic
idx=21 addr=0x00019ff0 off=0x00019ff0 sz=16 vsz=16 perm=-rw- name=.got
idx=22 addr=0x0001a000 off=0x0001a000 sz=920 vsz=920 perm=-rw- name=.got.plt
idx=23 addr=0x0001a3a0 off=0x0001a3a0 sz=608 vsz=608 perm=-rw- name=.data
idx=24 addr=0x0001a600 off=0x0001a600 sz=3464 vsz=3464 perm=-rw- name=.bss
idx=25 addr=0x0001a600 off=0x0001a600 sz=8 vsz=8 perm=---- name=.gnu_debuglink
idx=26 addr=0x0001a608 off=0x0001a608 sz=254 vsz=254 perm=---- name=.shstrtab

27 sections
```

Also, using `-r`, radare will flag the beginning and end of each section, as well as comment each one with the previous information.

```
$ rabin2 -Sr /bin/ls
fs sections
S 0x00000238 0x00000238 0x0000001c 0x0000001c .interp 4
f section..interp 28 0x00000238
f section_end..interp 0 0x00000254
CC [00] va=0x00000238 pa=0x00000238 sz=28 vsz=28 rwx=-r-- .interp @ 0x00000238
S 0x00000254 0x00000254 0x00000020 0x00000020 .note.ABI_tag 4
f section..note.ABI_tag 32 0x00000254
f section_end..note.ABI_tag 0 0x00000274
CC [01] va=0x00000254 pa=0x00000254 sz=32 vsz=32 rwx=-r-- .note.ABI_tag @ 0x00000254
S 0x00000274 0x00000274 0x00000024 0x00000024 .note.gnu.build_id 4
f section..note.gnu.build_id 36 0x00000274
f section_end..note.gnu.build_id 0 0x00000298
```

```
CC [02] va=0x00000274 pa=0x00000274 sz=36 vsz=36 rwx=-r-- .note.gnu.build_id @ 0x00000274
S 0x00000298 0x00000298 0x00000068 0x00000068 .gnu.hash 4
f section..gnu.hash 104 0x00000298
f section_end..gnu.hash 0 0x00000300
CC [03] va=0x00000298 pa=0x00000298 sz=104 vsz=104 rwx=-r-- .gnu.hash @ 0x00000298
S 0x00000300 0x00000300 0x00000c18 0x00000c18 .dynsym 4
f section..dynsym 3096 0x00000300
f section_end..dynsym 0 0x00000f18
CC [04] va=0x00000300 pa=0x00000300 sz=3096 vsz=3096 rwx=-r-- .dynsym @ 0x00000300
S 0x00000f18 0x00000f18 0x00000593 0x00000593 .dynstr 4
f section..dynstr 1427 0x00000f18
f section_end..dynstr 0 0x000014ab
CC [05] va=0x00000f18 pa=0x00000f18 sz=1427 vsz=1427 rwx=-r-- .dynstr @ 0x00000f18
S 0x000014ac 0x000014ac 0x00000102 0x00000102 .gnu.version 4
f section..gnu.version 258 0x000014ac
f section_end..gnu.version 0 0x000015ae
(...)
```

Binary Diffing

Based on radare.today article "[binary diffing](#)"

Without parameter, `radiff2` will by default show what bytes changed, and the corresponding offsets.

```
$ radiff2 genuine cracked
0x000081e0 85c00f94c0 => 9090909090 0x000081e0
0x00007c805 85c00f84c0 => 9090909090 0x00007c805

$ rasm2 -d 85c00f94c0
test eax, eax
sete al
```

Notice how the two jumps are noped.

For bulk processing, you may want to have a higher-overview of the differences. This is why radare2 is able to compute the distance and the percentage of similarity between two files with the `-s` option:

```
$ radiff2 -s /bin/true /bin/false
similarity: 0.97
distance: 743
```

If you want more concrete data, it's also possible to count the differences, with the `-c` option:

```
$ radiff2 -c genuine cracked
2
```

If you're unsure about the fact that you're dealing with similar binaries, you can check if some functions are matching with the `-C` option. The columns being: "First file offset", "Percentage of matching" and "Second file offset".

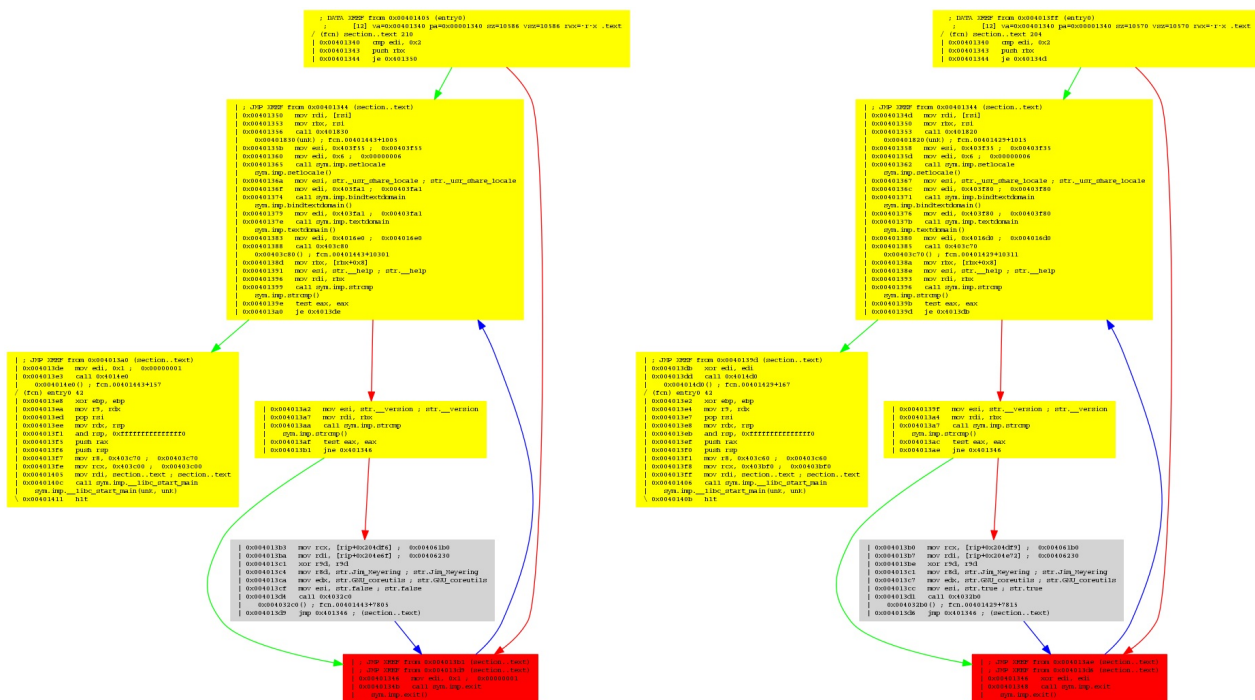
```
$ radiff2 -C /bin/false /bin/true
entry0 0x4013e8 | MATCH (0.904762) | 0x4013e2 entry0
sym.imp.__libc_start_main 0x401190 | MATCH (1.000000) | 0x401190 sym.imp.__
fcn.00401196 0x401196 | MATCH (1.000000) | 0x401196 fcn.00401196
fcn.0040103c 0x40103c | MATCH (1.000000) | 0x40103c fcn.0040103c
fcn.00401046 0x401046 | MATCH (1.000000) | 0x401046 fcn.00401046
[...]
```

And now the cool feature : radare2 supports graph-diffing, à la [DarunGrim](#), with the `-g` option. You can either give a symbol name, or specify two offsets in case the function you want to diff doesn't have the same name in both file.

For example, `radiff2 -g main /bin/true /bin/false | xdot -` will show the differences between the

main function of true and false. You can compare it to `radiff2 -g main /bin/false /bin/true` (Notice the order of the arguments) to get the two versions.

This is the result:



The parts in yellow are indicating that some offsets are not matching, the grey one is a perfect match, while the red one highlight a strong difference. If you look closely, you'll see that the left one is `mov edi, 0x1; call sym.imp.exit`, while the right one is `xor edi, edi; call sym.imp.exit`.

Binary diffing is an important feature for reverse engineering. It can be used to analyze [security updates](#), infected binaries, firmware changes and more..

We have only shown the code analysis diffing functionality, but radare2 supports more sort of diffing between two binaries at byte level, deltified similarities and more to come.

We have plans to implement more kinds of bindiffing functionalities into r2, and why not, add support for ascii art graph diffing and better integration with the rest of the toolkit.

Rasm2

The inline assembler/disassembler. Initially 'rasm' was designed to be used for binary patching, just to get the bytes of a certain opcode. Here's the help

```
$ rasm2 -h
Usage: rasm2 [-CdDehLBvw] [-a arch] [-b bits] [-o addr] [-s syntax]
             [-f file] [-F fil:ter] [-i skip] [-l len] 'code'|hex|-
-a [arch]    Set architecture to assemble/disassemble (see -L)
-b [bits]    Set cpu register size (8, 16, 32, 64) (RASM2_BITS)
-c [cpu]     Select specific CPU (depends on arch)
-C           Output in C format
-d, -D       Disassemble from hexpair bytes (-D show hexpairs)
-e           Use big endian instead of little endian
-f [file]    Read data from file
-F [in:out]  Specify input and/or output filters (att2intel, x86.pseudo, ...)
-h           Show this help
-i [len]     ignore/skip N bytes of the input buffer
-k [kernel] Select operating system (linux, windows, darwin, ..)
-l [len]     Input/Output length
-L           List supported asm plugins
-o [offset]  Set start address for code (default 0)
-O [file]    Output file name (rasm2 -Bf a.asm -O a)
-s [syntax]  Select syntax (intel, att)
-B           Binary input/output (-l is mandatory for binary input)
-v           Show version information
-w           What's this instruction for? describe opcode
If '-l' value is greater than output length, output is padded with nops
If the last argument is '-' reads from stdin
```

Asm supported plugins can be listed with "-L"

```
$ rasm2 -L
_d 16      8051      PD      8051 Intel CPU
_d 16 32   arc      GPL3    Argonaut RISC Core
ad 16 32 64 arm      GPL3    Acorn RISC Machine CPU
_d 16 32 64 arm.cs   BSD     Capstone ARM disassembler
_d 16 32   arm.winedbg LGPL2  WineDBG's ARM disassembler
_d 16 32   avr      GPL     AVR Atmel
ad 32      bf      LGPL3    Brainfuck
_d 16      cr16    LGPL3    cr16 disassembly plugin
_d 16      csr     PD       Cambridge Silicon Radio (CSR)
ad 32 64   dalvik  LGPL3    AndroidVM Dalvik
ad 16      dcpu16  PD       Mojang's DCPU-16
_d 32 64   ebc     LGPL3    EFI Bytecode
_d 8       gb      LGPL3    GameBoy(TM) (z80-like)
_d 16      h8300   LGPL3    H8/300 disassembly plugin
_d 8       i8080   BSD     Intel 8080 CPU
ad 32      java    Apache  Java bytecode
_d 32      m68k    BSD     Motorola 68000
_d 32      malbolge LGPL3    Malbolge Ternary VM
ad 32 64   mips    GPL3     MIPS CPU
_d 16 32 64 mips.cs  BSD     Capstone MIPS disassembler
```

_d	16 32 64	msil	PD	.NET Microsoft Intermediate Language
_d	32	nios2	GPL3	NIOS II Embedded Processor
_d	32 64	ppc	GPL3	PowerPC
_d	32 64	ppc.cs	BSD	Capstone PowerPC disassembler
ad		rar	LGPL3	RAR VM
_d	32	sh	GPL3	SuperH-4 CPU
_d	32 64	sparc	GPL3	Scalable Processor Architecture
_d	32	tms320	LGPLv3	TMS320 DSP family
_d	32	ws	LGPL3	Whitespace esoteric VM
_d	16 32 64	x86	BSD	udis86 x86-16,32,64
_d	16 32 64	x86.cs	BSD	Capstone X86 disassembler
a_	32 64	x86.nz	LGPL3	x86 handmade assembler
ad	32	x86.olly	GPL2	OlllyDBG X86 disassembler
ad	8	z80	NC-GPL2	Zilog Z80

Assemble

It is quite common to use 'rasm2' from the shell. It is a nice utility for copypasting the hexpairs that represent the opcode.

```
$ rasm2 -a x86 -b 32 'mov eax, 33'
b821000000

$ echo 'push eax;nop;nop' | rasm2 -f -
5090
```

Rasm2 is used from radare core to write bytes using 'wa' command.

It is possible to assemble for x86 (intel syntax), olly (olly syntax), powerpc, arm and java. For the intel syntax, rasm tries to use NASM or GAS. You can use the SYNTAX environment variable to choose your favorite syntax: intel or att.

There are some examples in rasm's source directory to assemble a raw file using rasm from a file describing these opcodes.

```
$ cat selfstop.rasm
;
; Self-Stop shellcode written in rasm for x86
;
; --pancake
;

.arch x86
.equ base 0x8048000
.org 0x8048000 ; the offset where we inject the 5 byte jmp

selfstop:
    push 0x8048000
    pusha
    mov eax, 20
    int 0x80

    mov ebx, eax
    mov ecx, 19
    mov eax, 37
    int 0x80
    popa
    ret
;
; The call injection
;

    ret

[0x00000000]> e asm.bits = 32
```



```
[0x00000000]> wx `!rasm2 -f a.rasm`  
[0x00000000]> pd 20  
    0x00000000    6800800408    push 0x8048000 ; 0x08048000  
    0x00000005    60                pushad  
    0x00000006    b814000000    mov eax, 0x14 ; 0x00000014  
    0x0000000b    cd80          int 0x80  
    syscall[0x80][0]=?  
    0x0000000d    89c3          mov ebx, eax  
    0x0000000f    b913000000    mov ecx, 0x13 ; 0x00000013  
    0x00000014    b825000000    mov eax, 0x25 ; 0x00000025  
    0x00000019    cd80          int 0x80  
    syscall[0x80][0]=?  
    0x0000001b    61            popad  
    0x0000001c    c3            ret  
    0x0000001d    c3            ret
```

Disassemble

In the same way as rasm assembler works, giving the '-d' flag you can disassemble an hexpair string:

```
$ rasm2 -a x86 -b 32 -d '90'  
nop
```

Analysis

There are different commands to perform data and code analysis and extract information like pointers, string references, basic blocks, extract opcode information, jump information, xrefs, etc..

Those operations are handled by the root 'a'analyze command:

```
|Usage: a[?adfFghoprsx]
| a8 [hexpairs]      analyze bytes
| aa                 analyze all (fcns + bbs)
| ad                 analyze data trampoline (wip)
| ad [from] [to]      analyze data pointers to (from-to)
| ae [expr]           analyze opcode eval expression (see ao)
| af[rnbcs1?+-*]     analyze Functions
| aF                 same as above, but using graph.depth=1
| ag[?acgdlf]        output Graphviz code
| ah[?lba-]          analysis hints (force opcode size, ...)
| ao[e?] [len]        analyze Opcodes (or emulate it)
| ap                 find and analyze function preludes
| ar[?ld-*]          manage refs/xrefs (see also afr?)
| as [num]            analyze syscall using dbg.reg
| at[trd+-*?] [.]    analyze execution Traces
|Examples:
| f ts @ `S*~text:0[3]`; f t @ section..text
| f ds @ `S*~data:0[3]`; f d @ section..data
| .ad t t+ts @ d:ds
```

Code analysis

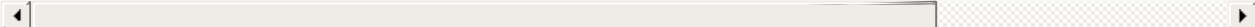
The code analysis is a common technique used to extract information from the assembly code. Radare stores multiple internal data structures to identify basic blocks, function trees, extract opcode-level information and such.

One common radare2 analysis command usage is the following:

```
[0x08048440]> aa
[0x08048440]> pdf @ main

; DATA XREF from 0x08048457 (entry0)
/ (fcn) fcn.08048648 141
|
| ;-- main:
| 0x08048648 8d4c2404 lea ecx, [esp+0x4]
| 0x0804864c 83e4f0 and esp, 0xfffffffff0
| 0x0804864f ff71fc push dword [ecx-0x4]
| 0x08048652 55 push ebp
|
| ; CODE (CALL) XREF from 0x08048734 (fcn.080486e5)
| 0x08048653 89e5 mov ebp, esp
| 0x08048655 83ec28 sub esp, 0x28
| 0x08048658 894df4 mov [ebp-0xc], ecx
| 0x0804865b 895df8 mov [ebp-0x8], ebx
| 0x0804865e 8975fc mov [ebp-0x4], esi
| 0x08048661 8b19 mov ebx, [ecx]
| 0x08048663 8b7104 mov esi, [ecx+0x4]
| 0x08048666 c744240c000. mov dword [esp+0xc], 0x0
| 0x0804866e c7442408010. mov dword [esp+0x8], 0x1 ; 0x00000001
| 0x08048676 c7442404000. mov dword [esp+0x4], 0x0
| 0x0804867e c7042400000. mov dword [esp], 0x0
| 0x08048685 e852fdffff call sym..imp.ptrace
|
| sym..imp.ptrace(unk, unk)
| 0x0804868a 85c0 test eax, eax
| ,=< 0x0804868c 7911 jns 0x0804869f
| | 0x0804868e c70424cf870. mov dword [esp], str.Don_tuseadebuguer_ ; 0x080487
| | 0x08048695 e882fdffff call sym..imp.puts
| | sym..imp.puts()
| | 0x0804869a e80dfdffff call sym..imp.abort
| | sym..imp.abort()
| `-> 0x0804869f 83fb02 cmp ebx, 0x2
| ,==< 0x080486a2 7411 je 0x080486b5
| | 0x080486a4 c704240c880. mov dword [esp], str.Youmustgiveapasswordforusethis
| | 0x080486ab e86cfdffff call sym..imp.puts
| | sym..imp.puts()
| | 0x080486b0 e8f7fcffff call sym..imp.abort
| | sym..imp.abort()
| `--> 0x080486b5 8b4604 mov eax, [esi+0x4]
| 0x080486b8 890424 mov [esp], eax
| 0x080486bb e8e5feffff call fcn.080485a5
|
| fcn.080485a5() ; fcn.080484c6+223
| 0x080486c0 b800000000 mov eax, 0x0
| 0x080486c5 8b4df4 mov ecx, [ebp-0xc]
| 0x080486c8 8b5df8 mov ebx, [ebp-0x8]
| 0x080486cb 8b75fc mov esi, [ebp-0x4]
| 0x080486ce 89ec mov esp, ebp
```

	0x080486d0	5d	pop ebp
	0x080486d1	8d61fc	lea esp, [ecx-0x4]
\	0x080486d4	c3	ret



Rahash2

The it is quite easy to calculate a hash checksum of the current block using the '#' command.

```
$ radare2 /bin/ls
[0x08049790]> bf entry0
[0x08049790]> #md5
d2994c75adaa58392f953a448de5fba7
```

In the same way you can also calculate other hashing algorithms that are supported by 'rahash': md4, md5, crc16, crc32, sha1, sha256, sha384, sha512, par, xor, xorpair, mod255, hamdist, entropy, all.

The '#' command can accept a numeric argument to define the length in bytes to be hashed.

```
[0x08049A80]> #md5 32
9b9012b00ef7a94b5824105b7aaad83b
[0x08049A80]> #md5 64
a71b087d8166c99869c9781e2edcf183
[0x08049A80]> #md5 1024
a933cc94cd705f09a41ecc80c0041def
[0x08049A80]>
```

Rahash2 tool

The rahash tool is the used by radare to realize these calculations. It

```
$ rahash2 -h
Usage: rahash2 [-rBhLkv] [-b sz] [-a algo] [-s str] [-f from] [-t to] [file] ...
-a algo      comma separated list of algorithms (default is 'sha256')
-b bsize     specify the size of the block (instead of full file)
-B           show per-block hash
-e           swap endian (use little endian)
-f from      start hashing at given address
-i num       repeat hash N iterations
-S seed      use given seed (hexa or s:string) use ^ to prefix
-k           show hash using the openssl's randomkey algorithm
-q           run in quiet mode (only show results)
-L           list all available algorithms (see -a)
-r           output radare commands
-s string    hash this string instead of files
-t to        stop hashing at given address
-v           show version information
```

It permits the calculation of the hashes from strings or files.

```
$ rahash2 -q -a md5 -s 'hello world'
5eb63bbbe01eeed093cb22bb8f5acdc3
```

It is possible to hash the full contents of a file . But dont do this for large files like disks or so, because rahash stores the buffer in memory before calculating the checksum instead of doing it progressively.

```
$ rahash2 -a all /bin/ls
/bin/ls: 0x00000000-0x0001ae08 md5: b5607b4dc7d896c0fab5c4a308239161
/bin/ls: 0x00000000-0x0001ae08 sha1: c8f5032c2dce807c9182597082b94f01a3bec495
/bin/ls: 0x00000000-0x0001ae08 sha256: 978317d58e3ed046305df92a19f7d3e0bfc3c70cad979f24f
/bin/ls: 0x00000000-0x0001ae08 sha384: 9e946efdbebb4e0ca00c86129ce2a71ee734ac30b620336c38
/bin/ls: 0x00000000-0x0001ae08 sha512: 076806cedb5281fd15c21e493e12655c55c52537fc1f36e641
/bin/ls: 0x00000000-0x0001ae08 crc16: 4b83
/bin/ls: 0x00000000-0x0001ae08 crc32: 6e316348
/bin/ls: 0x00000000-0x0001ae08 md4: 3a75f925a6a197d26bc650213f12b074
/bin/ls: 0x00000000-0x0001ae08 xor: 3e
/bin/ls: 0x00000000-0x0001ae08 xorpair: 59
/bin/ls: 0x00000000-0x0001ae08 parity: 01
/bin/ls: 0x00000000-0x0001ae08 entropy: 0567f925
/bin/ls: 0x00000000-0x0001ae08 hamdist: 00
/bin/ls: 0x00000000-0x0001ae08 pcprint: 23
/bin/ls: 0x00000000-0x0001ae08 mod255: 1e
/bin/ls: 0x00000000-0x0001ae08 xxhash: 138c936d
/bin/ls: 0x00000000-0x0001ae08 adler32: fca7131b
```

Debugger

The debugger in radare is implemented as an IO plugin. It handles several URIs for creating or attaching to a process. (List can be found on `r2 -L`. For example:

```
r_d debug      Debug a program or pid. dbg:///bin/ls, dbg://1388 (LGPL3)
```

There are different backends for multiple architectures and operating systems like GNU/Linux, Windows, MacOSX, (Net,Free,Open)BSD and Solaris.

The process memory is interpreted by radare as a plain file. So all the mapped pages like the program and the libraries can be readed and interpreted as code, structures, etc..

The rest of the communication between radare and the debugger layer is the wrapped `system()` call that receives a string as argument and executes the given command. The result of the operation is buffered in the output console and this contents can be handled by a scripting language.

This is the reason why radare can handle single and double exclamation marks for calling `system()`.

```
[0x00000000]> ds
[0x00000000]> !!ls
```

The double exclamation mark tells radare to skip the plugin list to find an IO plugin handling this command to launch it directly to the shell. A single one will walk through the io plugin list.

The debugger commands are mostly portable between architectures and operating systems. But radare tries to implement them on all the architectures and OSs injecting shellcodes, or handling exceptions in a special way. For example in mips there's no stepping feature by hardware, so radare has an own implementation using a mix of code analysis and software breakpoints to bypass this limitation.

To get the basic help of the debugger you can just type 'd?':

```
Usage: d[sbhcrbo] [arg]
dh [handler]    list or set debugger handler
dH [handler]    transplant process to a new handler
dd             file descriptors (!fd in r1)
ds[ol] N       step, over, source line
do            open process (reload, alias for 'oo')
dk [sig][=act] list, send, get, set, signal handlers of child
di[s] [arg..]  inject code on running process and execute it (See gs)
dp[=*?t][pid]  list, attach to process or thread id
dc[?]         continue execution. dc? for more
dr[?]         cpu registers, dr? for extended help
db[?]         breakpoints
dbt           display backtrace
dt[?r] [tag]   display instruction traces (dtr=reset)
dm[?*]        show memory maps
dw [pid]       block prompt until pid dies
```


To restart your debugging session, you can type `oo` or `oo+` depending on your desired behavior.

<code>oo</code>	reopen current file (kill+fork in debugger)
<code>oo+</code>	reopen current file in read-write

Registers

The registers are part of the user area stored in the context structure used by the scheduler. This structure can be manipulated to get and set values for those registers, and on intel for example, is possible to directly manipulate the DRx hardware registers to setup hardware breakpoints.

There are different commands to get values of the registers. For the General Purpose ones use:

```
[0x4A13B8C0]> dr
r15 = 0x00000000
r14 = 0x00000000
r13 = 0x00000000
r12 = 0x00000000
rbp = 0x00000000
rbx = 0x00000000
r11 = 0x00000000
r10 = 0x00000000
r9 = 0x00000000
r8 = 0x00000000
rax = 0x00000000
rcx = 0x00000000
rdx = 0x00000000
rsi = 0x00000000
rdi = 0x00000000
oeax = 0x0000003b
rip = 0x7f0f2dbae630
rsp = 0x7fff515923c0

[0x7f0f2dbae630]> dr?rip ; get value of 'eip'
0x7f0f2dbae630

[0x4A13B8C0]> dr eip = esp ; set 'eip' as esp
```

The interaction between the plugin and the core is done by commands returning radare instructions. This is used for example to set some flags in the core to set the values of the registers.

```
[0x7f0f2dbae630]> dr* ; Appending '*' will show radare commands
f r15 1 0x0
f r14 1 0x0
f r13 1 0x0
f r12 1 0x0
f rbp 1 0x0
f rbx 1 0x0
f r11 1 0x0
f r10 1 0x0
f r9 1 0x0
f r8 1 0x0
f rax 1 0x0
f rcx 1 0x0
f rdx 1 0x0
f rsi 1 0x0
f rdi 1 0x0
```

```
f oeax 1 0x3b
f rip 1 0x7fff73557940
f rflags 1 0x200
f rsp 1 0x7fff73557940
```

```
[0x4A13B8C0]> .dr* ; include common register values in flags
```

An old copy of the registers is stored all the time to keep track of the changes done during the execution of the program. This old copy can be accessed with `oregs` .

```
[0x7f1fab84c630]> dro
r15 = 0x00000000
r14 = 0x00000000
r13 = 0x00000000
r12 = 0x00000000
rbp = 0x00000000
rbx = 0x00000000
r11 = 0x00000000
r10 = 0x00000000
r9 = 0x00000000
r8 = 0x00000000
rax = 0x00000000
rcx = 0x00000000
rdx = 0x00000000
rsi = 0x00000000
rdi = 0x00000000
oeax = 0x0000003b
rip = 0x7f1fab84c630
rflags = 0x00000200
rsp = 0x7fff386b5080
```

```
[0x7f1fab84c630]> dr
r15 = 0x00000000
r14 = 0x00000000
r13 = 0x00000000
r12 = 0x00000000
rbp = 0x00000000
rbx = 0x00000000
r11 = 0x00000000
r10 = 0x00000000
r9 = 0x00000000
r8 = 0x00000000
rax = 0x00000000
rcx = 0x00000000
rdx = 0x00000000
rsi = 0x00000000
rdi = 0x7fff386b5080
oeax = 0xffffffffffffffff
rip = 0x7f1fab84c633
rflags = 0x00000202
rsp = 0x7fff386b5080
```

The values in `eax`, `oeax` and `eip` has changed.

To store and restore the register values you can just dump the output of 'dr*' command to disk and then re-interpret it again:

```
[0x4A13B8C0]> dr* > regs.saved ; save registers  
[0x4A13B8C0]> drp regs.saved ; restore
```

In the same way the eflags can be altered by the characters given in this way. Setting to '1' the selected flags:

```
[0x4A13B8C0]> dr eflags = pst  
[0x4A13B8C0]> dr eflags = azsti
```

You can easily get a string representing the last changes in the registers using the 'drd' command (diff registers):

```
[0x4A13B8C0]> drd  
orax = 0x0000003b was 0x00000000 delta 59  
rip = 0x7f00e71282d0 was 0x00000000 delta -418217264  
rflags = 0x00000200 was 0x00000000 delta 512  
rsp = 0x7fffe85a09c0 was 0x00000000 delta -396752448
```

Remoting Capabilities

Radare can work locally or remotely without hard differences. The reason is that everything remains on the IO subsystem that abstracts the access to `system()`, `cmd()` and all basic IO operations thru the network.

Here's the help of the command:

```
[0x00405a04]> =?
|Usage:  =[:!+-=hH] [...] # radare remote command execution protocol
|
|rap commands:
| =                list all open connections
| =<[fd] cmd       send output of local command to remote fd
| =[fd] cmd        exec cmd at remote 'fd' (last open is default one)
| =! cmd           run command via r_io_system
| += [proto://]host add host (default=rap://, tcp://, udp://)
| -=[fd]           remove all hosts or host 'fd'
| ==[fd]           open remote session with host 'fd', 'q' to quit
|
|rap server:
| =:port           listen on given port using rap protocol (o rap://9999)
| =:host:port cmd  run 'cmd' command on remote server
|
|http server:
| =h port          listen for http connections (r2 -qc=H /bin/ls)
| =h-             stop background webserver
| =h*             restart current webserver
| =h& port         start http server in background)
| =H port          launch browser and listen for http
| =H& port         launch browser and listen for http in background
```

You can also display some of the radare2 remoting capabilities by displaying the list of supported IO plugins `radare2 -L`.

lets introduce the command with a little example :) A typical remote session could be:

At remote host1:

```
$ radare2 rap://:1234
```

At remote host2:

```
$ radare2 rap://:1234
```

At localhost:

```
$ radare2 -
```

; Add hosts

```
[0x004048c5]> += rap://<host1>:1234//bin/ls
Connected to: <host1> at port 1234
waiting... ok

[0x004048c5]> =
0 - rap://<host1>:1234//bin/ls
```

You can open remote files in debug mode (or using any io plugin) specifying the uri when adding hosts:

```
[0x004048c5]> += += rap://<host2>:1234/dbg:///bin/ls
Connected to: <host2> at port 1234
waiting... ok
0 - rap://<host1>:1234//bin/ls
1 - rap://<host2>:1234/dbg:///bin/ls
```

Exec commands in host1

```
[0x004048c5]> =0 px
[0x004048c5]> = s 0x666
```

Open a session with host2

```
[0x004048c5]> ==1
fd:6> pi 1
...
fd:6> q
```

Remove hosts (and close connections)

```
[0x004048c5]> -=
```

So, you can init tcp or udp servers, add them with '+= tcp:/' or '+= udp:/', and then redirect to them the radare output. For instance:

```
[0x004048c5]> += tcp://<host>:<port>/
Connected to: <host> at port <port>
5 - tcp://<host>:<port>/
[0x004048c5]> =<5 cmd...
```

The `=<' command will send the result of the execution of the command at the right to the remote connection number N (or the last one used if no id specified).

Plugins

IO plugins

All the access to files, network, debugger, etc.. is wrapped by an IO abstraction layer that allows to interpret all the data as if it was a single file.

IO plugins are the ones used to wrap the open, read, write and 'system' on virtual file systems. You can make radare understand that any thing can be handled as a plain file. A socket connection, a remote radare session, a file, a process, a device, a gdb session, etc..

So, when radare reads a block of bytes, is the task of the IO plugin to get these bytes from any place and put them in the internal buffer. IO plugins are selected while opening a file by its URI. Here'r some examples:

Debugging URIs

```
$ r2 dbg:///bin/ls
$ r2 pid://1927
```

Remote sessions

```
$ r2 rap://:1234
$ r2 rap://<host>:1234//bin/ls
```

Virtual buffers

```
$ r2 malloc:///512
shortcut for
$ r2 -
```

You can get a list of the radare IO plugins by typing `radare -L` :

```
$ r2 -L
rw_ zip      Open zip files apk:///foo.apk//MANIFEST or zip:///foo.apk//theclass/fun.cl
rwd windbg   Attach to a KD debugger (LGPL3)
rw_ sparse   sparse buffer allocation (sparse:///1024 sparse://) (LGPL3)
rw_ shm      shared memory resources (shm:///key) (LGPL3)
rw_ self     read memory from myself using 'self:/' (LGPL3)
rw_ rap      radare network protocol (rap://:port rap://host:port/file) (LGPL3)
rwd ptrace   ptrace and /proc/pid/mem (if available) io (LGPL3)
rw_ procpid  /proc/pid/mem io (LGPL3)
rw_ mmap     open file using mmap:/// (LGPL3)
rw_ malloc   memory allocation (malloc:///1024 hex:///cd8090) (LGPL3)
r__ mach     mach debug io (unsupported in this platform) (LGPL)
```

```
rw_ ihex      Intel HEX file (ihex://eeproms.hex) (LGPL)
rw_ http      http get (http://radare.org/) (LGPL3)
rw_ gzip      read/write gzipped files (LGPL3)
rwd gdb       Attach to gdbserver, 'qemu -s', gdb://localhost:1234 (LGPL3)
r_d debug     Debug a program or pid. dbg:///bin/ls, dbg://1388 (LGPL3)
rw_ bfdbg     BrainFuck Debugger (bfdbg://path/to/file) (LGPL3)
```


Crackmes

Crackmes are the training ground for reverse engineering. This section will go over tutorials on how to defeat various crackmes using r2.

IOLI CrackMes

The IOLI crackme is a good starting point for learning r2. This is a set of tutorials based on the tutorial at [dustri](#)

The IOLI crackmes are available at a locally hosted [mirror](#)

IOLI 0x00

This is the first IOLI crackme, and the easiest one.

```
$ ./crackme0x00
IOLI Crackme Level 0x00
Password: 1234
Invalid Password!
```

The first thing to check is if the password is just plaintext inside the file. In this case, we don't need to do any disassembly, and we can just use rabin2 with the -z flag to search for strings in the binary.

```
$ rabin2 -z ./crackme0x00
vaddr=0x08048568 paddr=0x00000568 ordinal=000 sz=25 len=24 section=.rodata type=a string=
vaddr=0x08048581 paddr=0x00000581 ordinal=001 sz=11 len=10 section=.rodata type=a string=
vaddr=0x0804858f paddr=0x0000058f ordinal=002 sz=7 len=6 section=.rodata type=a string=25
vaddr=0x08048596 paddr=0x00000596 ordinal=003 sz=19 len=18 section=.rodata type=a string=
vaddr=0x080485a9 paddr=0x000005a9 ordinal=004 sz=16 len=15 section=.rodata type=a string=
```

So we know what the following section is, this section is the header shown when the application is run.

```
vaddr=0x08048568 paddr=0x00000568 ordinal=000 sz=25 len=24 section=.rodata type=a string=
```

Here we have the prompt for the password.

```
vaddr=0x08048581 paddr=0x00000581 ordinal=001 sz=11 len=10 section=.rodata type=a string=
```

This is the error on entering an invalid password.

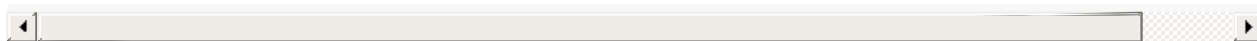
```
vaddr=0x08048596 paddr=0x00000596 ordinal=003 sz=19 len=18 section=.rodata type=a string=
```

This is the message on the password being accepted.

```
vaddr=0x080485a9 paddr=0x000005a9 ordinal=004 sz=16 len=15 section=.rodata type=a string=
```

But what is this? It's a string, but we haven't seen it in running the application yet.

```
vaddr=0x0804858f paddr=0x0000058f ordinal=002 sz=7 len=6 section=.rodata type=a string=25
```



Let's give this a shot.

```
$ ./crackme0x00
IOLI Crackme Level 0x00
Password: 250382
Password OK :)
```

So we now know that 250382 is the password, and have completed this crackme.

IOLI 0x01

This is the second IOLI crackme.

```
$ ./crackme0x01
IOLI Crackme Level 0x01
Password: test
Invalid Password!
```

Let's check for strings with rabin2.

```
$ rabin2 -z ./crackme0x01
vaddr=0x08048528 paddr=0x00000528 ordinal=000 sz=25 len=24 section=.rodata type=a string=
vaddr=0x08048541 paddr=0x00000541 ordinal=001 sz=11 len=10 section=.rodata type=a string=
vaddr=0x0804854f paddr=0x0000054f ordinal=002 sz=19 len=18 section=.rodata type=a string=
vaddr=0x08048562 paddr=0x00000562 ordinal=003 sz=16 len=15 section=.rodata type=a string=
```

This isn't going to be as easy as 0x00. Let's try disassembly with r2.

```
$ r2 ./crackme0x01 -- Use 'zoom.byte=printable' in zoom mode ('z' in Visual mode) to find
[0x08048330]> aa
[0x08048330]> pdf@main
/ (fcn) main 113
|      ; var int local_4 @ ebp-0x4
|      ; DATA XREF from 0x08048347 (entry0)
|      0x080483e4      55          push ebp
|      0x080483e5      89e5        mov ebp, esp
|      0x080483e7      83ec18      sub esp, 0x18
|      0x080483ea      83e4f0      and esp, -0x10
|      0x080483ed      b800000000  mov eax, 0
|      0x080483f2      83c00f      add eax, 0xf
|      0x080483f5      83c00f      add eax, 0xf
|      0x080483f8      c1e804      shr eax, 4
|      0x080483fb      c1e004      shl eax, 4
|      0x080483fe      29c4        sub esp, eax
|      0x08048400      c7042428850. mov dword [esp], str.IOLI_Crackme_Level_0x01_n ; [0
|      0x08048407      e810ffffff  call sym.imp.printf
|      sym.imp.printf(unk)
|      0x0804840c      c7042441850. mov dword [esp], str.Password_ ; [0x8048541:4]=0x73
|      0x08048413      e804ffffff  call sym.imp.printf
|      sym.imp.printf()
|      0x08048418      8d45fc      lea eax, dword [ebp + 0xffffffffc]
|      0x0804841b      89442404    mov dword [esp + 4], eax ; [0x4:4]=0x10101
|      0x0804841f      c704244c850. mov dword [esp], 0x804854c ; [0x804854c:4]=0x490064
|      0x08048426      e8e1feffff  call sym.imp.scanf
|      sym.imp.scanf()
|      0x0804842b      817dfc9a140. cmp dword [ebp + 0xffffffffc], 0x149a
|      ,=< 0x08048432      740e        je 0x8048442
|      | 0x08048434      c704244f850. mov dword [esp], str.Invalid_Password__n ; [0x80485
|      | 0x0804843b      e8dcfeffff  call sym.imp.printf
```

```

|      |      sym.imp.printf()
|      |,==< 0x08048440    eb0c      jmp 0x804844e ; (main)
|      ||      ; JMP XREF from 0x08048432 (main)
|      |`-> 0x08048442    c7042462850. mov dword [esp], str.Password_OK____n ; [0x8048562:
|      |      0x08048449    e8cefeffff call sym.imp.printf
|      |      sym.imp.printf()
|      |      ; JMP XREF from 0x08048440 (main)
|      |`--> 0x0804844e    b8000000000 mov eax, 0
|      |      0x08048453    c9          leave
|      |      0x08048454    c3          ret
\

```

"aa" tells r2 to analyze the whole binary, which gets you symbol names, among things.

"pdf" stands for

- Print
- Disassemble
- Function

This will print the disassembly of the main function, or the main() that everyone knows. You can see several things as well: weird names, arrows, etc.

- imp. stands for imports. Those are imported symbols, like printf()
- str. stands for strings. Those are strings (obviously).

If you look carefully, you'll see a cmp instruction, with a constant, 0x149a. cmp is a compare instruction, and the 0x in front of it specifies it's in base 16, or hex.

```
0x0804842b    817dfc9a140. cmp dword [ebp + 0xffffffffc], 0x149a
```

You can use radare2's ? command to get it in another base.

```
[0x08048330]> ? 0x149a
5274 0x149a 012232 5.2K 0000:049a 5274 10011010 5274.0 0.000000
```

So now we know that 0x149a is 5274 in decimal. Let's try this as a password.

```
$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 5274
Password OK :)
```

Bingo, the password was 5274. In this case, the password function at 0x0804842b was comparing the input against the value, 0x149a in hex. Since user input is usually decimal, it was a safe bet that the input was intended to be in decimal, or 5274. Now, since we're hackers, and curiosity drives us, let's see what

happens when we input in hex.

```
$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 0x149a
Invalid Password!
```

It was worth a shot, but it doesn't work. That's because `scanf` will take the 0 in 0x149a to be a zero, rather than accepting the input as actually being the hex value.

And this concludes IOLI 0x01.

Radare2 Reference Card

Survival Guide

Command	Description
aa	Auto analyze
Content Cell	Content Cell
pdf@fcn(Tab)	Disassemble function
f fcn(Tab)	List functions
f str(Tab)	List strings
fr [flagname] [newname]	Rename flag
psz [offset]	Print string
arf [flag]	Find cross reference for a flag

Flagspaces

Command	Description
fs	Display flagspaces
fs *	Select all flagspaces
fs [sections]	Select one flagspace

Flags

Command	Description
f	List flags
fs *	Select all flagspaces
fs [sections]	Select one flagspace
fj	Display flags in JSON
fl	Show flag length
fx	Show hexdump of flag
fC [name] [comment]	Set flag comment

Information

Command	Description
ii	Information on imports
il	Info on binary

ie	Display entrypoint
iS	Display sections
ir	Display relocations

Print string

Command	Description
psz [offset]	Print zero terminated string
psb [offset]	Print strings in current block
psx [offset]	Show string with escaped chars
psp [offset]	Print pascal string
psw [offset]	Print wide string

Visual mode

Command	Description
V	Enter visual mode
p/P	Rotate modes (hex, disasm, debug, words, buf)
c	Toggle (c)ursor
q	Back to Radare shell
h/j/k/l	Move around (or H/J/K/L) (left-down-up-right)
Enter	Follow address of jump/call
sS	Step/step over
o	Go/seek to given offset
.	Seek to program counter
/	In cursor mode, search in current block
:cmd	Run radare command
:[-]cmt	Add/remove comment
/*+-[Change block size, [= resize hex.cols
> <	Seek aligned to block size
i/a/A	(i)nsert hex, (a)ssemble code, visual (A)ssembler
b/B	Toggle breakpoint / automatic block size
d[f?]	Define function, data, code, ..
D	Enter visual diff mode (set diff.from/to)
e	Edit eval configuration variables
f/F	Set/unset flag
gG	Go seek to begin and end of file (0-\$s)
mK/'K	Mark/go to Key (any key)

M	Walk the mounted filesystems
n/N	Seek next/prev function/flag/hit (scr.nkey)
o	Go/seek to given offset
C	Toggle (C)olors
R	Randomize color palette (ecr)
t	Track flags (browse symbols, functions..)
T	Browse anal info and comments
v	Visual code analysis menu
V/W	(V)iew graph (agv?), open (W)ebUI
uU	Undo/redo seek
x	Show xrefs to seek between them
yY	Copy and paste selection
z	Toggle zoom mode

Searching

Command	Description
/ foo\00	Search for string 'foo\0'
/b	Search backwards
//	Repeat last search
/w foo	Search for wide string 'f\0o\0o\0'
/wi foo	Search for wide string ignoring case
/! ff	Search for first occurrence not matching
/i foo	Search for string 'foo' ignoring case
/e /E.F/i	Match regular expression
/x ff0.23	Search for hex string
/x ff..33	Search for hex string ignoring some nibbles
/x ff43 ffd0	Search for hexpair with mask
/d 101112	Search for a deltified sequence of bytes
/!x 00	Inverse hexa search (find first byte != 0x00)
/c jmp [esp]	Search for asm code (see search.asmstr)
/a jmp eax	Assemble opcode and search its bytes
/A	Search for AES expanded keys
/r sym.printf	Analyze opcode reference an offset
/R	Search for ROP gadgets
/P	Show offset of previous instruction
/m magicfile	Search for matching magic file
/p patternsize	Search for pattern of given size

/z min max	Search for strings of given size
/v[?248] num	Look for a asm.bigendian 32bit value

Saving

Command	Description
Po [file]	Open project
Ps [file]	Save project
Pi [file]	Show project information

Usable variables in expression

Command	Description
\$\$	Here (current virtual seek)
\$o	Here (current disk io offset)
\$s	File size
\$b	Block size
\$w	Get word size, 4 if asm.bits=32, 8 if 64
\$c,\$r	Get width and height of terminal
\$S	Section offset
\$SS	Section size
\$j	Jump address (jmp 0x10, jz 0x10 => 0x10)
\$f	Jump fail address (jz 0x10 => next instruction)
\$l	Number of instructions of current function
\$F	Current function size
\$Jn	Get nth jump of function
\$Cn	Get nth call of function
\$Dn	Get nth data reference in function
\$Xn	Get nth xref of function
\$m	Opcode memory reference (mov eax,[0x10] => 0x10)
\$l	Opcode length
\$e	1 if end of block, else 0
\$ev	Get value of eval config variable
\$?	Last comparison value

License

Based on the Radare 2 reference card by Thanat0s, which is under the GNU GPL. Original license is as follows:

This card may be freely distributed under the terms of the GNU
general public licence – Copyright c by Thanat0s - v0.1 -