Radare2book

By @Maijin

Summary

R2 "Book"

Welcome on the Radare2 Book

Introduction

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

Nowadays the project is composed of 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. Allows to open any kind of file from different IO access like disk, network, kernel plugins, remote devices, debugged processes, ... and handle any of them as if they were a simple plain file.

Implements an advanced command line interface for moving around the file, analyzing data, disassembling, binary patching, data comparision, searching, replacing, scripting with Ruby, Python, Lua and Perl, ...

rabin2

Extracts information from executable binaries like ELF, PE, Java CLASS, MACH-O. It's used from the core to get exported symbols, imports, file information, xrefs, library dependencies, sections, ...

rasm2

Commandline assembler and disassembler for multiple architectures (intel[32,64], mips, arm, powerpc, java, msil, ...)

```
$ 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

Implementation of a block-based rahash for small text strings or large disks, supporting multiple algorithms like md4, md5, crc16, crc32, sha1, sha256, sha384, sha512, par, xor, xorpair, mod255, hamdist or entropy.

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

radiff2

Binary diffing utility implementing multiple algorithms. Supports byte-level or delta diffing for binary files and code-analysis diffing to find changes in basic code blocks from radare code analysis or IDA ones using the idc2rdb rsc script.

rafind2

rafind2 is a program to find byte patterns in files

ragg2

Ragg2 is a frontend for r_egg. It's used to compile programs into tiny binaries for x86-32/64 and ARM.

rarun2

Rarun2 is used as a launcher for running programs with different environment, arguments, permissions, directories and overridden default file descriptors. It can be useful for :

- Crackme
- Fuzzing
- Test suite

Getting radare2

You can get radare from the website http://radare.org/ or Github repo https://github.com/radare/radare2

There are binary packages for multiple operating systems and GNU/Linux distributions (Ubuntu, Maemo, Gentoo, Windows, iPhone, etc..) But I hardly encourage you to get the sources and compile them yourself to better understand the dependencies and have the source code and examples more accessible.

I try to publish a new stable release every month and sometimes publish nightly tarballs.

But as always the best way to use a software is to go upstream and pull the development repository which in the case of radare is commonly more stable than the 'stable' releases O:)

To do this you will need Git and type:

```
$ 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 you will have to type the following command in the root of the recently created 'radare2' directory.

```
$ git pull
```

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

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

Or just feed me with a patch

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

Compilation and portability

Currently the core of radare2 can be compiled on many systems and architectures but the main development is done on GNU/Linux and GCC. But it is known to compile with TCC and SunStudio.

People usually wants to use radare as a debugger for reverse engineering, and this is a bit more restrictive portability issue, so if the debugger is not ported to your favorite platform, please, notify it to me or just disable the debugger layer with --without-debugger in the ./configure stage.

Nowadays the debugger layer can be used on Windows, GNU/Linux (intel32, intel64, mips, arm), FreeBSD, NetBSD, OpenBSD (intel32, intel64) and there are plans for Solaris and OSX. And there are some IO plugins to use gdb, gdbremote or wine as backends.

The build system based on ACR/GMAKE.

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

But there is a simple script to do that automatically:

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

Windows compilation

The easy way to compile things for Windows is using MinGW32. The w32 builds distributed in the radare homepage are generated from a GNU/Linux box using MinGW32 and they are tested with Wine.

To compile type:

```
$ CC=i486-mingw32-gcc ./configure --enable-w32 --without-gui
$ make
$ make w32dist
$ zip -r w32-build.zip w32-build
```

The 'i486-mingw32-gcc' compiler is the one I have in my box, you will probably need to change this. MinGW32 will generate a native console application for Windows.

Another possible way to compile radare2 on w32 is using Cygwin, which I dont really recommend at all because of the problems related to the Cygwin librarires makes the program quite hard to be debugged in case of problems.

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 [-dDwntLqv] [-P patch] [-p prj] [-a arch] [-b bits] [-i file] [-s addr] [-B blocks
ize] [-c cmd] [-e k=v] file|-

  -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 promt) 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

Many people requested a sample session of using radare to help in understanding how the shell works and how to perform the most common tasks like disassembling, seeking, binary patching and debugging.

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

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 up the block size to 4K or the size required with the command `b` and move forward or backward at seeks aligned to the block size using the `>` and `<` commands.

The 'print' command (short: `p`), accepts a second letter to specify the print mode. The most common ones are `px` for printing in hexadecimal, `pd` for disassembling.

To 'write' first open the file with `radare -w`. This should be specified while opening the file. You can then use the `w` command to write strings or `wx` for hexpair strings:

```
> 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 (example: p?).

To enter visual mode press `V<enter>`. To quit visual mode and return to the prompt use the `q` key.

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

While in visual mode you can also insert (alterate bytes) pressing `i` followed by 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 looks something like this:

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

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

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

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

Some examples:

```
ds                    ; call debugger '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 using '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.

The `~` character enables the internal grep function which can be used to filter the output of any command. The usage is quite simple:

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

We 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 first column of the first row matching 'call'
```

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

Expressions

Expressions are mathematical representations of a 64 bit numeric 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 `?`. Here are some examples:

```
[0xB7F9D810]> ? 0x8048000
134512640 0x8048000 01001100000 128.0M 804000:0000 134512640 00000000 1
34512640.0 0.000000
[0xB7F9D810]> ? 0x8048000+34
134512674 0x8048022 01001100042 128.0M 804000:0022 134512674 00100010 1
34512674.0 0.000000
[0xB7F9D810]> ? 0x8048000+0x34
134512692 0x8048034 01001100064 128.0M 804000:0034 134512692 00110100 1
34512692.0 0.000000
[0xB7F9D810]> ? 1+2+3-4*3
-6 0xfffffffffffffffa 01777777777777777777772 17179869183.0G fffff000:0ffa -6
```

The supported arithmetic operations are:

```
+ : addition
- : substraction
* : 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. Here are a few examples:

```
?@?    or stype @@?      ; misc help for '@' (seek), '~' (grep) (see ~??)
?$?          ; show available '$' variables
$$           ; here (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)
```

For example:

```
[0x4A13B8C0]> :? $m + $l
140293837812900 0x7f98b45df4a4 03771426427372244 130658.0G 8b45d000:04a
4 140293837812900 10100100 140293837812900.0 -0.000000


[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 0xffffffb3
int   ->  bin           ;  rax2 b30
bin   ->  int           ;  rax2 1010d
float ->  hex           ;  rax2 3.33f
hex   ->  float         ;  rax2 Fx40551ed8
oct   ->  hex           ;  rax2 35o
hex   ->  oct           ;  rax2 Ox12 (O 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]---+
|Eo. .            |
| . . . .         |
|      o          |
|       .         |
|        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 .radarerc.

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

Now the debugger prompt should appear and if you press `enter` (null command) the basic view of the process will be displayed with the stack dump, general purpose registers and disassembly from current program counter (eip on intel).

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 !reg which can be used to get or set values of the general purpose registers. You can also manipulate the hardware and extended/floating registers.

Configuration

Colors

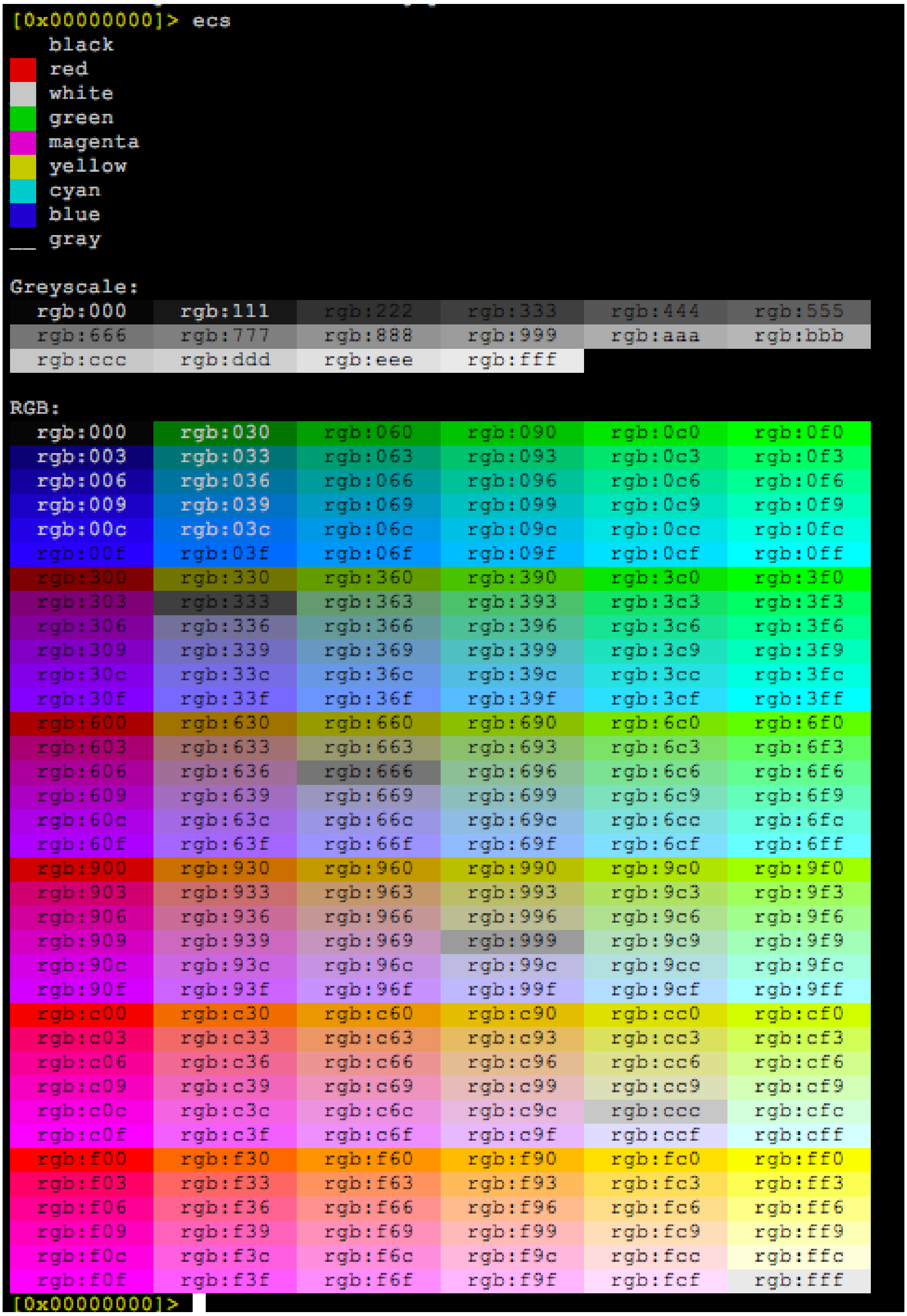
The console access is wrapped by an API that permits to show the output of any command as ANSI, w32 console or HTML (more to come ncurses, pango, ...) this allows the core to be flexible enought to run on limited environments like kernels or embedded devices allowing us to get the feedback from the application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

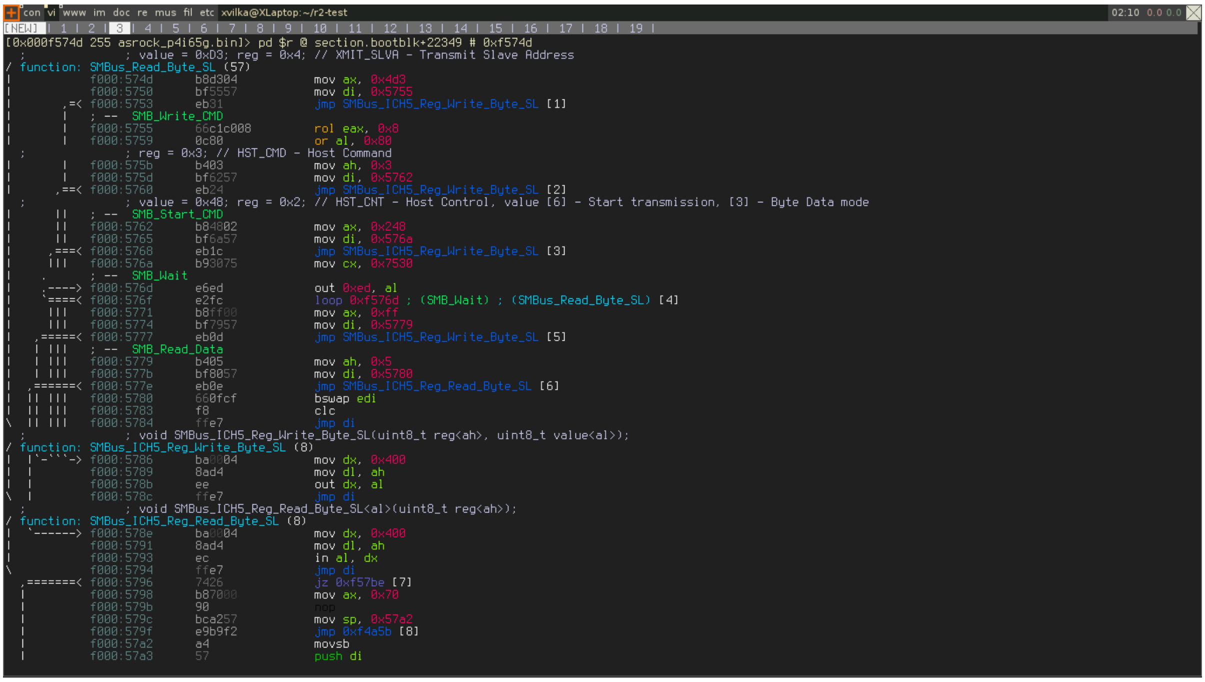
Type `ec` to get a list of all the palete elements. Type `ecs` to show a color palete to pick colors from:



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
```



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 dissassemble 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 ouside 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 colorized 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 begining of the program

Basic Commands

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 3
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_ta
g
[02] . 0x00000274 r-- va=0x00400274 sz=0x00000024 vsz=00000024 .note.gnu.bui
ld_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 r
wx 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 relaunching r2
oo                reopen current file (kill+fork in debugger)
```

```
oo+              reopen current file in read-write
o 4              priorize 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[=68abcdDfilmrstuxz] [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 dcff 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 0xfff46690 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 other countries %d/%m/%y is rather common. This means that in internation
al context this format is ambiguous and should not be used.) (SU)
%e  Like %d, the day of the month as a decimal number, but a leading zero is replac
ed by a space. (SU)
%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-digit year corresponding to the ISO week number (see %V).  This has the sam
e format and value as %Y, except that if the ISO week number belongs to the previo
us or next year, that year is used instead. (TZ)
%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
preceded by a blank.  (See also %H.)  (TZ)
%l  The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are p
receded by a blank.  (See also %I.)  (TZ)
%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 strin
gs for the current locale.  Noon is treated as "PM" and midnight as "AM".
%P  Like %p but in lowercase: "am" or "pm" or a corresponding string for the current
locale. (GNU)
%r  The time in a.m. or p.m. notation.  In the POSIX locale this is equivalent to %I:%M
:%S %p.  (SU)
%R  The time in 24-hour notation (%H:%M).  (SU) For a version including the second
s, see %T below.
%s  The number of seconds since the Epoch, 1970-01-01 00:00:00  +0000 (UTC). (T
Z)
%S  The second as a decimal number (range 00 to 60).  (The range is up to 60 to all
ow for occasional leap seconds.)
%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, star
ting with the first Sunday as the first day of week 01.  See also %V and %W.
%V  The ISO 8601 week number (see NOTES) of the current year as a decimal numb
er, range 01 to 53, where week 1 is the first week that has at least 4 days in the new
year.  See also %U and %W.(U)
%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, star
ting with the first Monday as the first day of week 01.
%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 fr
```

```
om UTC). (SU)
%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: JSON, C, Python, Cstring (pcj, pc, pcp, pcs) 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, 0x
81, 0xc3, 0xd6, 0xa7, 0x01, 0x00, 0x8b, 0x83, 0x00, 0xff, 0xff, 0xff, 0x5a, 0x8d, 0x24,
0x84, 0x29, 0xc2 };


[0x7fcd6a891630]> pcs
"\x48\x89\xe7\xe8\x68\x39\x00\x00\x49\x89\xc4\x8b\x05\xef\x16\x22\x00\x5a\x48\x8d
\x24\xc4\x29\xc2\x52\x48\x89\xd6\x49\x89\xe5\x48\x83\xe4\xf0\x48\x8b\x3d\x06\x1a
```

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 ) = 0xffffffda
[0x4A13B8C0]> dr
  eax  0xffffffda    esi  0xffffffff    eip    0x4a14fc24
  ebx  0x4a151c91    edi  0x4a151be1    oeax   0x00000005
  ecx  0x00000000    esp  0xbfbedb1c    eflags 0x200246
  edx  0x00000000    ebp  0xbfbedbb0    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 achive this, simply pass a 'format memory string' as an argument to `pf` and temporally 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 :
(*0xffffffff8949ed31)      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 iissxsssss 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            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 esotheric 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
name in disassembly
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 Intermedate 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 curren block. This is for example a XOR, ADD, SUB, ...

```
[0x4A13B8C0]> wo?
|Usage: wo[asmdxoArl24] [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  -=  substraction
|  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
```

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 comparision:

```
[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 paramater 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

Visual cursor

Pressing lowercase `c` makes the cursor appear or disappear. 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 enhace 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 popups 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 :)

Searching bytes

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 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.
```

It is also possible to mix hexadecimal scape 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 searchs

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 alignement. 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 searchs 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 Muoz 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]> /A
0 AES keys found
```

Disassembling

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 adquire 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 0x0000002 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 a 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.

Rabin2

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 wich function is called by a call instruction.

```
$ rabin2 -i /bin/ls |head
[Imports]
ordinal=001 plt=0x000021b0 bind=GLOBAL type=FUNC name=__ctype_toupper_lo
c
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=__progname
addr=0x0021a630 off=0x0021a630 ord=119 fwd=NONE sz=8 bind=UNKNOWN typ
e=OBJECT name=program_invocation_name
addr=0x0021a600 off=0x0021a600 ord=121 fwd=NONE sz=0 bind=GLOBAL type=
NOTYPE name=__bss_start
addr=0x0021a630 off=0x0021a630 ord=122 fwd=NONE sz=8 bind=GLOBAL type=
OBJECT name=__progname_full
addr=0x0021a600 off=0x0021a600 ord=123 fwd=NONE sz=8 bind=UNKNOWN typ
e=OBJECT name=program_invocation_short_name
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.__progname 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/ls.c
addr=0x00012490 off=0x00012490 ordinal=001 sz=26 len=26 section=.rodata type=A string=sort_type != sort_version
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 %*s
addr=0x000124c5 off=0x000124c5 ordinal=005 sz=10 len=10 section=.rodata typ
e=A string=%*s, %*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 typ
e=A string=cannot access %s
addr=0x000124e5 off=0x000124e5 ordinal=008 sz=29 len=29 section=.rodata typ
e=A string=cannot read symbolic link %s
addr=0x00012502 off=0x00012502 ordinal=009 sz=10 len=10 section=.rodata typ
e=A string=unlabeled
```

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. Furtheremore, 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.g
nu.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=.d
ynsym
idx=05 addr=0x00000f18 off=0x00000f18 sz=1427 vsz=1427 perm=-r-- name=.dy
nstr
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 @ 0x
00000298
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 @ 0
x00000300
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 @ 0x0
0000f18
S 0x000014ac 0x000014ac 0x00000102 0x00000102 .gnu.version 4
f section..gnu.version 258 0x000014ac
f section_end..gnu.version 0 0x000015ae
(...)
```

Rasm2

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

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, 0xfffffff0
|       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 0x804869f
|    |  0x0804868e    c70424cf870. mov dword [esp], str.Don_tuseadebuguer_ ;  0x0
80487cf
|    |  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 0x80486b5
|   |   0x080486a4    c704240c880. mov dword [esp], str.Youmustgiveapasswordfor
usethisprogram_ ;  0x0804880c
|   |   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

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 openssh'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: c8f5032c2dce807c9182597082b94f01a3bec
495
/bin/ls: 0x00000000-0x0001ae08 sha256: 978317d58e3ed046305df92a19f7d3e0bfc
b3c70cad979f24fee289ed1d266b0
/bin/ls: 0x00000000-0x0001ae08 sha384: 9e946efdbebb4e0ca00c86129ce2a71ee73
4ac30b620336c381aa929dd222709e4cf7a800b25fbc7d06fe3b184933845
/bin/ls: 0x00000000-0x0001ae08 sha512: 076806cedb5281fd15c21e493e12655c55c
52537fc1f36e641b57648f7512282c03264cf5402b1b15cf03a20c9a60edfd2b4f76d49
05fcec777c297d3134f41f
/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
```