

# Reversing with Radare2

## Starting Radare

The basic usage is **radare2** *executable* (on some systems you can use **r2** instead of **radare2**); if you want to run radare2 without opening any file, you can use **--** instead of an executable name.

Some command-line options are:

```
-d file          debug executable file
                  Warning: if there exists a script named file.r2,
                  then it gets executed after the others rc-files
-d pid          debug process pid
-A              analyze all referenced code (aaa command)
-R profile.rr2  specifies rarun2 profile (same as
                  -e dbg.profile=profile.rr2)
-w              open file in write mode
-p prj          use project prj
-P              list projects
-h              show help message (-hh the verbose one)
```

Example: **r2 -dA /bin/ls**

## General information

The command **?** prints the help. Command names are hierarchically defined; for instance, all **printing** commands start with **p**. So, to understand what a command does, you can append **?** to a *prefix* of such a command; e.g., to learn what **pdf** does, you can first try **pd?**, then the more general **p?**. You can get recursive help with **?\***; e.g.: **p?\***. Single-line comments can be entered using **#**; e.g. **s # where R we?**. Command **?** can also be used to evaluate an expression and print its result in various format; e.g. **? 5 \* 8+2** (note the space after **?**). Commands **?v/?vi** print result only in hex/decimal. There are also some special **\$**-variables (list them all with: **?\$?**); e.g.:

```
$$    current virtual seek
$b    block size
```

Where an address *addx* is expected, you can provide any expression that evaluates to an address, e.g. a function name or a register name. In this cheatsheet we sometimes use *fn-name*, instead of *addx*, to emphasize that the argument is supposed to be a function starting address. As default address is (usually?) used the current seek: **\$\$**. All commands that:

- accept an optional size (e.g. **pd**), use the current block size by default (see: **b**)
- accept an optional address (e.g., **pdf**), use the current position by default (see: **s**)

## Internal grep-like filtering

You can filter command output by appending **~[!]*str***, to display only rows [not] containing string *str*; e.g. **pdf-rdx** and **pdf-!rdx**. You can further filter by appending

```
:r          to display row r ( $0 \leq r < \#rows$  or, backwards
              with:  $-\#rows \leq r \leq -1$ )
[c1, c2, ...] to display columns c1, c2, ... ( $0 \leq c_i < \#cols$ )
:r[c1, ..., cn] to display columns c1, ..., cn of row r
...          to pipe the output into less-like viewer
...          to pipe the output into HUD viewer
```

Examples: **af1-[0]**, **af1-malloc[0]**, **pdf~:2** and **pdf-mov:2**

There is much more (sorting, counting, ...); see: **~?**

## Shell interaction

Command output can be redirected to a file by appending **>filename** or piped to an external command with **|progname [args]**. Examples: **af1 > all\_functions** and **af1 | wc -l**.

External commands can be run with **!!progname [args]**.

Moreover, backticks can be used to send the output of **r2**-commands as arguments; e.g. **!!echo `? 42`**. Vice versa output of external programs can be used as arguments for internal commands; e.g. **pdf `echo 3` @ `echo entry0`**.

Some common Unix-like commands are implemented as built-ins; e.g. **ls**, **cd**, **pwd**, **mkdir** and **rm**.

## Radare scripting

```
. filename      interpret r2 script filename
.! command     interpret output of command as r2 commands
```

## Python scripting (via r2pipe)

You can script Radare2 with Python, by leveraging *r2pipe*, that can be easily installed (inside any Python 2 virtual environment) with: **pip install r2pipe**.

Then, you can spawn a Python interpreter, from inside **r2**, with:

```
#!pipe python [python-file]
or simply:
```

```
#. python-file
```

Once you are in Python-world, you can connect to **r2** by importing **r2pipe** and initializing some variable, say **r2**, with **r2pipe.open("#!pipe")**, or simply **r2pipe.open()**.

Then you can interact with Radare by invoking method **cmd**; e.g. **print(r2.cmd('pdf @ entry0'))**.

You can make most Radare2 commands output in JSON format by appending a **j**; e.g. **pdfj** (instead of **pdf**).

Method **cmdj** can de-serialize JSON output into Python objects; e.g. **f = r2.cmdj('pdfj @ entry0')**  
**print f['name'], f['addr'], f['ops'][0]['opcode']**

## r2pipe: connecting to other r2 instances

You can connect to any web-listening instance of **r2** by passing **r2pipe.open** a string of the form **'http://host:port'**. By using this approach you get your own seek-cursor: your seek commands won't affect others.

To open a background web-service in **r2** use command **=h&**. You may also want to take a look at configuration variable **http.sandbox**.

## Configuration

```
e??          list all variable names and descriptions
e?[?] var-name show description of var-name
e var-name    show the value of var-name
e var-name =?[?] print valid values of var-name [with descript.].
                  E.g. e asm.arch=??
e             show the value of all variables
eco theme-name select theme; eg. eco solarized
eco           list available themes
b            display current block size
b size       set block size
env [name [=value]] get/set environment variables
```

## Some variables

<b>asm.pseudo</b>	enable pseudo-code syntax (in visual mode, toggle with: <b>\$</b> )
<b>asm.bytes</b>	display bytes of each instruction
<b>asm.describe</b>	show opcode description
<b>asm.cmtright</b>	comments at right of disassembly if they fit
<b>asm.emu</b>	run ESIL emulation analysis on disasm
<b>asm.demangle</b>	Show demangled symbols in disasm
<b>bin.demangle</b>	Import demangled symbols from RBin
<b>cmd.stack</b>	command to display the stack in visual debug mode (Eg: <b>px 32</b> )
<b>dbg.follow.child</b>	continue tracing the child process on fork
<b>io.cache</b>	enable cache for IO changes (AKA non-persistent write-mode)
<b>scr.utf8</b>	show nice UTF-8 chars instead of ANSI (Windows: switch code-page with <b>chcp 65001</b> )
<b>scr.nkey</b>	select seek mode (fun, hit, flag); affects commands <b>n</b> and <b>N</b> during visual mode
<b>scr.wheel</b>	enables mouse-wheel in visual mode
<b>scr.breaklines</b>	break lines in Visual instead of truncating them

## Example: my ~/.radare2rc

```
e asm.bytes=0
e scr.utf8=true
e asm.cmtright=true
e cmd.stack=px 32
e scr.wheel=false
eco solarized
```

## Searching: /

<b>/ <i>str</i></b>	search for string <i>str</i>
<b>/x <i>hstr</i></b>	search for hex-string <i>hstr</i>
<b>/a <i>asm-instr</i></b>	assemble instruction and search for its bytes
<b>/R <i>opcode</i></b>	find ROP gadgets containing opcode; see: <a href="http://radare.today/posts/ropnroll/">http://radare.today/posts/ropnroll/</a>
<b>/A <i>type</i></b>	It seems you need to be in <i>debug</i> mode to use this (?!?) find instructions of type <i>type</i> (/A? for the listof types)

Also: **e??search** for options

## Seeking: s

<b>s</b>	print current position/address
<b>s <i>addx</i></b>	seek to <i>addx</i>
<b>s.. <i>hex</i></b>	changes the least-significant part of current address to <i>hex</i>
<b>s+ <i>n</i> and s- <i>n</i></b>	seek <i>n</i> bytes forward/backward
<b>s++ and s--</b>	seek block-size bytes forward/backward
<b>s-</b>	undo seek
<b>s+</b>	redo seek

## Writing: w

<b>wa</b> <i>asm-instr</i>	assemble and write opcodes; for more instructions the whole command must be quoted: " <b>wa</b> <i>asm-instr</i> <sub>1</sub> ; <i>asm-instr</i> <sub>2</sub> ; ..."
<b>wao</b> ...	replace current instruction; see <b>wao?</b> for details
<b>w[z]</b> <i>str</i>	write string <i>str</i> [and append byte <code>\x00</code> ]
<b>wx</b> <i>hex-pairs</i>	write hex-pairs
<b>wc</b>	list pending changes (see variable <code>io.cache</code> )
<b>wc*</b>	list pending changes in Radare commands
<b>wtf</b> [ <i>file</i> ] [ <i>size</i> ]	write to file

## Analysis (functions and syscalls): a

<b>aaa</b>	analyze ( <b>aa</b> ) and auto-name all functions
<b>afl</b> [ <i>l</i> ]	list functions [with details]
<b>afi</b> <i>fn-name</i>	show verbose info for <i>fn-name</i>
<b>afn</b> <i>new-name addr</i>	(re)name function at address <i>addr</i>
<b>asl</b>	list syscalls
<b>asl</b> <i>name</i>	display syscall-number for <i>name</i>
<b>asl</b> <i>n</i>	display name of syscall number <i>n</i>
<b>afvd</b> <i>var-name</i>	output r2 command for displaying the address and value of arg/local <i>var-name</i>
<b>.afvd</b> <i>var-name</i>	display address and value of <i>var-name</i>
<b>afvn</b> <i>name new-name</i>	rename argument/local variable
<b>afvt</b> <i>name type</i>	change type for given argument/local
<b>axt</b> <i>addr</i>	find data/code references to <i>addr</i>

## Graphviz/graph code: ag

<b>ag</b> <i>addr</i>	output graphviz code (BB at <i>addr</i> and children) E.g. view the function graph with: <b>ag \$\$   xdot -</b>
<b>agc</b> <i>addr</i>	callgraph of function at <i>addr</i>
<b>agC</b>	full program callgraph

## Information: i (and S)

<b>i</b>	show info of current file
<b>ie</b>	entrypoint
<b>iz</b> [ <i>z</i> ]	strings in data sections [whole binary]
<b>il</b>	libraries
<b>ii</b>	imports
<b>iS</b>	sections
<b>S</b>	list segments (confusingly called sections!?)

## Printing: p

<b>ps</b> [ <b>@</b> <i>addr</i> ]	print C-string at <i>addr</i> (or current position)
<b>pxr</b> [ <i>n</i> ] [ <b>@</b> <i>addr</i> ]	print with references to flags/code (telescoping)
<b>px</b> [ <i>n</i> ] [ <b>@</b> <i>addr</i> ]	hexdump — note: <b>x</b> is an alias for <b>px</b>
<b>px{h w q}</b> ...	hexdump half-words (16 bits) / words (32 bits) / quad-words (64 bits)
<b>pxl</b> [ <i>n</i> ] [ <b>@</b> <i>addr</i> ]	display <i>n</i> rows of hexdump
<b>px/fmt</b> [ <b>@</b> <i>addr</i> ]	gdb-style printing <i>fmt</i> (in gdb see: <b>help x</b> from r2: <b>!!gdb -q -ex 'help x' -ex quit</b> )
<b>pd</b> [ <i>n</i> ] [ <b>@</b> <i>addr</i> ]	disassemble <i>n</i> instructions
<b>pD</b> [ <i>n</i> ] [ <b>@</b> <i>addr</i> ]	disassemble <i>n</i> bytes
<b>pd -n</b> [ <b>@</b> <i>addr</i> ]	disassemble <i>n</i> instructions backwards
<b>pdf</b> [ <b>@</b> <i>fn-name</i> ]	disassemble function <i>fn-name</i>
<b>pc[p]</b> [ <i>n</i> ] [ <b>@</b> <i>addr</i> ]	dumps in C [Python] format
<b>* <i>addr</i></b> [=value]	shortcut for reading/writing at <i>addr</i>

## Debugging: d

<b>?d</b> <i>opcode</i>	description of <i>opcode</i> (eg. <b>?d jle</b> )
<b>dc</b>	continue (or start) execution
<b>dcu</b> <i>addr</i>	continue until <i>addr</i> is reached
<b>dcs</b> [ <i>name</i> ]	continue until the next syscall (named <i>name</i> , if specified)
<b>dcr</b>	continue until ret (uses step over)
<b>dr=</b>	show general-purpose regs and their values
<b>dro</b>	show previous (old) values of registers
<b>drr</b>	show register references (telescoping)
<b>dr</b> <i>reg-name</i> = <i>value</i>	set register value
<b>drt</b>	list register types
<b>drt</b> <i>type</i>	list registers of type <i>type</i> and their values
<b>db</b>	list breakpoints
<b>db[-]</b> <i>addr</i>	add [remove] breakpoint
<b>doo</b> <i>args</i>	(re)start debugging
<b>ood</b>	synonym for <b>doo</b>
<b>ds</b>	step into
<b>dso</b>	step over
<b>dbt</b>	display backtrace
<b>drx</b>	hardware breakpoints
<b>dm</b>	list memory maps; the asterisk shows where the current offset is
<b>dmp</b>	change page permissions (see: <b>dmp?</b> )

## Types: t

<b>"td</b> <i>C-type-def</i> "	define a new type
<b>t</b> <i>t-name</i>	show type <i>t-name</i> in <b>pf</b> syntax
<b>.t</b> <i>t-name</i> <b>@</b> <i>addr</i>	display the value (of type <i>t-name</i> ) at <i>addr</i>
<b>t</b>	list (base?) types
<b>te / ts / tu</b>	list enums/structs/unions
<b>to</b> <i>file</i>	parse type information from C header file
<b>t1</b> <i>t-name</i>	link <i>t-name</i> to current address
<b>t1</b> <i>t-name</i> = <i>addr</i>	link <i>t-name</i> to address <i>addr</i>
<b>t1</b>	list all links in readable format
<b>tp</b> <i>t-name</i> = <i>addr</i>	cast data at <i>addr</i> to type <i>t-name</i> , and prints it

## Visual mode: V (q exits)

<b>c</b>	cursor-mode, <i>tab</i> switches among stack/regs/disassembly
<b>+</b> / <b>-</b>	increment/decrement current byte
<b>:</b>	execute a normal-mode command; e.g. <b>:dm</b>
<b>p</b> and <b>P</b>	rotate forward/backward print modes
<b>/str</b>	highlight occurrences of string <i>str</i>
<b>\$</b>	toggle pseudo-syntax
<b>O</b>	toggle ESIL-asm
<b>;</b>	add/remove comments (to current offset)
<b>x</b>	browse xrefs-to current offset
<b>X</b>	browse xrefs-from current function
<b>_</b>	browse flags
<b>d</b>	define function, end-function, rename, ...
<b>di{b o d h s}</b>	define immediate bin/oct/dec/hex or str
<b>V</b>	enter block-graph viewer
<b>A</b>	enter visual-assembler (preview must be confirmed)
<b>n / N</b>	seek next/previous function/flag/hit (see <b>scr.nkey</b> )
<b>i</b>	enter insert mode
<b>e</b>	configures internal variables

## Seeking (in Visual Mode)

<b>.</b>	seeks to program counter
<b>Enter</b>	on jump/call instructions, follow target address
<b>u</b>	undo
<b>U</b>	redo
<b>o</b>	go/seek to given offset
<b>O</b>	seek to beginning of current function
<b>d</b> (a non-zero digit)	jump to the target marked [ <i>d</i> ]
<b>m</b> (a letter)	mark the spot with letter <i>l</i>
<b>'l</b>	jump to mark <i>l</i>
<b>n / N</b>	jump to next/previous function

## Debugging (in Visual Mode)

<b>b</b> or <b>F2</b>	toggle breakpoint
<b>F4</b>	run to cursor
<b>s</b> or <b>F7</b>	step-into
<b>S</b> or <b>F8</b>	step-over
<b>F9</b>	continue

## Flags (AKA “bookmarks”): f

Note: in order to get your defined <i>names</i> appear in disassembly, you must include a prefix ( <b>fun</b> , <b>sub</b> , <b>obj</b> , ...); e.g. <b>f obj.foo @ 0x1234</b>	
<b>f</b> <i>name</i> <b>@</b> <i>addr</i>	or
<b>f</b> <i>name</i> = <i>addr</i>	associate name <i>name</i> to address <i>addr</i>
<b>f-</b> <b>@</b> <i>addr</i>	remove the association at address <i>addr</i>
<b>f-</b> <i>name</i>	remove the association with name <i>name</i>

## Comments: C

<b>CC</b>	list all comments in human friendly form
<b>CCu</b> <i>text</i> [ <b>@</b> <i>addr</i> ]	set (update?) comment <i>text</i> at <i>addr</i>
<b>CC</b> <i>text</i> [ <b>@</b> <i>addr</i> ]	append comment <i>text</i> at <i>addr</i>
<b>CC-</b> [ <b>@</b> <i>addr</i> ]	remove comment at <i>addr</i>
<b>CC.</b> [ <b>@</b> <i>addr</i> ]	show comment at <i>addr</i>
<b>CC!</b> [ <b>@</b> <i>addr</i> ]	edit comment using <b>cfg.editor</b> (vim, ...)

## Projects: P [unstable feature]

<b>P1</b>	list all projects
<b>Ps</b> [ <i>prj-name</i> ]	save project <i>prj-name</i>
<b>Po</b> <i>prj-name</i>	open project <i>prj-name</i>
<b>Pd</b> <i>prj-name</i>	delete project <i>prj-name</i>
<b>Pc</b> <i>prj-name</i>	show project script to console

## Running in different environments: rarun2

**rarun2** is used as a launcher for running programs with different environment, arguments, permissions, directories and overridden default file-descriptors. Usage:

**rarun2** [-t|*script-name.rr2*] [*directives*] [--] [*prog-name*] [*args*]  
**rarun2 -t** shows the terminal name, say *α*, and wait for a connection from another process. For instance, from another terminal, you can execute **rarun2 stdio=α program=/bin/sh** (use **stdin/stdout** to redirect one stream only).  
**rarun2** supports *a lot* of directives, see the man page.

---

Copyright ©2017 by zxgio; cheat-sheet built on October 27, 2017  
This cheat-sheet may be freely distributed under the terms of the GNU General Public License; the latest version can be found at:  
<https://github.com/zxgio/r2-cheatsheet/>