

Reversing with Radare2

Starting Radare

The basic usage is **radare2** *exe* (on some systems you can use simply **r2** instead of **radare2**). If there exists a script named *exe.r2*, then it gets executed after the others rc-files. 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
-d pid      debug process pid
-A          analyze all referenced code (aaa command)
-r profile.rr2 specifies r2run2 profile (same as
            -e dbg.profile=profile.rr2)
-w          open file in write mode
-p [prj]    list projects / use project prj
-h          show help message (-hh the verbose one)
```

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

Running in different environments: r2run2

r2run2 runs programs with different environments, arguments, permissions, directories and overridden default file-descriptors. Usage: **r2run2** [-t|script-name.rr2] [directives] [--] [prog-name] [args] **r2run2** -t shows the terminal name, say *α*, and wait for a connection from another process. For instance, from another terminal, you can execute **r2run2** **stdio=α program=/bin/sh** (use **stdin/stdout** to redirect one stream only). Run **r2run2 -h** to get a sample *.rr2* file. **r2run2** supports a lot of directives, see the man page for details.

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**)

Commands can be chained by using **;;**; e.g. **s fun; pd 2**.

A single command can be applied to each element of a sequence by using **@@**; e.g. **axt @@ str.***, see **@@?**.

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          display row r (0 ≤ r < #rows or, backwards
            with: −#rows ≤ r ≤ −1)
[c1[,c2,...]] display columns c1,c2,... (0 ≤ ci < #cols)
:r[c1,...,cn] display columns c1,...,cn of row r
..          pipe output into less-like viewer
...         pipe into HUD, which filters space separated strings
```

Examples: **afl~[0]**, **afl~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 **|progrname [args]**. Examples: **afl > all_functions** and **afl | wc -l**.

External commands can be run with **!!progrname [args]**. Note: if a command starts with a single **!**, the rest of the string is passed to currently loaded IO plugin (only if no plugin can handle the command, it is passed to the shell).

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

Some variables

```
asm.bytes    display bytes of each instruction
asm.describe show opcode description
asm.cmt.right comments at right of disassembly if they fit
asm.emu      run ESIL emulation analysis on disasm
asm.demangle Show demangled symbols in disasm
asm.shortcut Shortcut (e.g. [1],[2],...) position in visual mode
bin.baddr    base address of the binary
cmd.bp       command to run when a breakpoint is hit;
            e.g. cmd.bp=!!program
cmd.stack    command to display the stack in visual
            debug mode (Eg: px 32)
dbg.follow.child continue tracing the child process on fork
dbg.funcarg  display func. arguments in visual mode [experimental]
dbg.slow     show stack and regs in visual mode, in a slow but
            verbose (e.g. telescoping) mode; check column mode
dbg.trace    trace program execution (check also asm.trace)
io.cache     enable cache for IO (=non-persistent write-mode)
scr.utf8     show nice UTF-8 chars instead of ANSI
            (Windows: switch code-page with chcp 65001)
scr.utf8.curvy show curved UTF-8 corners (requires scr.utf8)
scr.nkey     select seek mode; affects n/N in visual mode
scr.breaklines break lines in Visual instead of truncating them
scr.html     disassembly outputs in HTML syntax
```

Searching: /

```
/ str        search for string str
/c instr     search for instruction instr
/x hstr      search for hex-string hstr
/a asm-instr assemble instruction and search for its bytes
/R[/] opcode find ROP gadgets [with r.e.] containing opcode;
            see: http://radare.today/posts/ropnroll/
/A type      find instructions of type type (/A? for the listof types)
Also: e search.in=?? and e??search for options
```

Seeking: s

```
s           print current position/address
s addx      seek to addx
s.. hex     changes least-significant part of current address to hex
s+ n and s- n seek n bytes forward/backward
s++ and s-- seek block-size bytes forward/backward
s- and s+   undo/redo seek
s=          list seek history
s*          list seek history as r2-commands
```

Writing: w

wa <i>asm-instr</i>	assemble+write opcodes; quote the whole command for more instructions: " wa <i>instr</i> ₁ ; <i>instr</i> ₂ ; ..."
wao ...	replace current instruction; see wao? for details
w[z] <i>str</i>	write string <i>str</i> [and append byte <code>\x00</code>]
wx <i>hex-pairs</i>	write hex-pairs
wc	list pending changes (see variable <code>io.cache</code>)
wtf [<i>file</i>] [<i>size</i>]	write to file
wop0 <i>v</i>	print offset of <i>v</i> inside De Bruijn pattern; equiv. to ragg2 -q v ; to produce a pattern: ragg2 -r -P size

Analysis (functions and syscalls): a

aaa	analyze (aa) and auto-name functions
aod <i>opcode</i>	description of <i>opcode</i> (eg. aod jle)
afl1 [<i>l</i>]	list functions [with details]
afi <i>fn-name</i>	show verbose info for <i>fn-name</i>
afn <i>new-name addr</i>	(re)name function at address <i>addr</i>
asl	list syscalls
asl <i>name</i>	display syscall-number for <i>name</i>
asl <i>n</i>	display name of syscall number <i>n</i>
afvd <i>var-name</i>	output r2 command for displaying the address and value of arg/local <i>var-name</i>
.afvd <i>var-name</i>	display address and value of <i>var-name</i>
afvn <i>new-name old-name</i>	rename argument/local variable
afvt <i>name type</i>	change type for given argument/local
afv- <i>name</i>	removes variable <i>name</i>
axt <i>addr</i>	find data/code references to <i>addr</i>
ahi { <i>b d h o r S s</i> } @ <i>addr</i>	define binary/decimal/hex/octal/IP/ syscall/string base for immediate

ESIL code emulation: ae

aei [<i>m</i>]	initialize ESIL VM state [stack]
aepc <i>addr</i>	change ESIL PC to <i>addr</i> (aeip sets PC to curseek)
aer ...	handle ESIL registers like dr does
aes [<i>b o</i>]	perform emulated debugger step [back over]
aecu <i>addr</i>	continue until given address

Graphviz/graph code: ag

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

Flags (AKA “bookmarks”): f

fs [<i>name</i>]	display flagspaces [select/create <i>fs name</i>]
fs+ <i>name</i>	push previous flag space and set <i>name</i>
fs-	pop to the previous flag space
f	list flags
f <i>name</i> @ <i>addr</i>	or
f <i>name</i> = <i>addr</i>	associate name <i>name</i> to address <i>addr</i>
f- @ <i>addr</i>	remove the association at address <i>addr</i>
f- <i>name</i>	remove the association with name <i>name</i>

Comments: C

CCu <i>text</i> [@ <i>addr</i>]	set (update?) comment <i>text</i> at <i>addr</i>
CC <i>text</i> [@ <i>addr</i>]	append comment <i>text</i> at <i>addr</i>
CC- [@ <i>addr</i>]	remove comment at <i>addr</i>
CC. [@ <i>addr</i>]	show comment at <i>addr</i>
CC! [@ <i>addr</i>]	edit comment using <code>cfg.editor</code> (vim, ...)

Debugging: d

dc	continue (or start) execution
dcu <i>addr</i>	continue until <i>addr</i> is reached
dcS [<i>name</i>]	continue until the next syscall [<i>name</i>]
dcr	continue until ret (uses step over)
dr=	show general-purpose regs and their values
dro	show previous (old) values of registers
drr	show register references (telescoping)
dr <i>reg-name</i> = <i>value</i>	set register value
drt	list register types
drt <i>type</i>	list registers of type <i>type</i> and their values
db	list breakpoints
db[-] <i>addr</i>	add [remove] breakpoint
doo [<i>args</i>]	(re)start debugging; synonym of ood
ds[o]	step into [over]
dbt	display backtrace (check <code>dbg.btdepth/btalgo</code>)
drx	hardware breakpoints
dm	list memory maps; the asterisk shows where the current offset is
dmh	show heap allocation (see: dmh?)
dmm	list modules (libraries, loaded binaries)
dmi [<i>addr lib</i>] [<i>sym</i>]	list symbols of target lib
dmp	change page permissions (see: dmp?)
dt[d]	list all traces [disassembled]
dd	handle file descriptors (see: dd?)

Types: t

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

Printing: p

ps [@ <i>addr</i>]	print C-string at <i>addr</i> (or current position)
psb [@ <i>addr</i>]	print C-strings at <i>addr</i> (or current block)
pxr [<i>n</i>] [@ <i>addr</i>]	print with references to flags/code (telescoping)
px [<i>n</i>] [@ <i>addr</i>]	hexdump — note: x is an alias for px
px{h w q} ...	hexdump in 16/32/64 bit words
px{H W Q} ...	as the previous one, but one per line
pxl [<i>n</i>] [@ <i>addr</i>]	display <i>n</i> rows of hexdump
pxfmt [@ <i>addr</i>]	gdb-style printing <i>fmt</i> (in gdb see: help x from r2: !!gdb -q -ex 'help x' -ex quit)
pd [<i>n</i>] [@ <i>addr</i>]	disassemble <i>n</i> instructions (backwards if <i>n</i> < 0)
p8 [<i>n</i>] [@ <i>addr</i>]	print bytes
pD [<i>n</i>] [@ <i>addr</i>]	disassemble <i>n</i> bytes
pdf [@ <i>fn-name</i>]	disassemble function <i>fn-name</i>
pc [<i>p</i>] [<i>n</i>] [@ <i>addr</i>]	dumps in C [Python] format
* <i>addr</i> [= <i>value</i>]	shortcut for reading/writing at <i>addr</i>
pf <i>fmt</i> <i>a</i> ₁ [, <i>a</i> ₂ , ...]	formatted print, see pf?? and pf???
pa [<i>d</i>] ...	assemble-to/disassemble-from hex-pairs

Information: i (and S)

i	show info of current file
iz [<i>z</i>]	strings in data sections [whole binary]
i{e i l S SS}	entrypoint/imports/libraries/sections/segments

Visual mode: V (q exits)

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

Seeking (in Visual Mode)

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

Debugging (in Visual Mode)

B (<i>b</i> in older versions) or F2	toggle breakpoint
F4	run to cursor
s or F7	step-into
S or F8	step-over
F9	continue

Projects: P [unstable feature]

P1	list all projects
P{o s d} [<i>prj-name</i>]	open/save/delete project <i>prj-name</i>
Pc <i>prj-name</i>	show project script to console