

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 **p**rinting 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 between **?** and the expression). There are also some special **\$**-variables (list all of them with: **?\$?**):

```
$$    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: **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 **|progname [args]**. Examples: **afl > all_functions** and **afl | wc -l**.

External commands can be run with **!!progname [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).

External commands can also be run with **#!pipe**, see below in *Python scripting*.

The output of external programs can be used as arguments for internal commands by using back-ticks to enclose the invocation of external commands; e.g. **pdf 'echo 3' @ 'echo entry0'**.

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

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

```
/ str          search for string str
/x hstr        search for hex-string hstr
/a asm-instr   assemble instruction and search for its bytes
/R opcode      find ROP gadgets containing opcode;
               see: http://radare.today/posts/ropnroll/
               It seems you need to be in debug mode to use this (?!?)
```

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

Seeking: s

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

Writing: w

wa <i>asm-instr</i>	assemble and write opcodes; for more instructions the whole command must be quoted: "wa <i>asm-instr</i> ₁ ; <i>asm-instr</i> ₂ ; ..."
w <i>str</i>	write string <i>str</i>
wz <i>str</i>	write string <i>str</i> and append byte \x00
wx <i>hex-pairs</i>	write hex-pairs

Analysis (functions and syscalls): a

aaa	analyze (aa) and auto-name all functions
afl [1]	list functions [with details]
afi <i>fn-name</i>	show verbose info for <i>fn-name</i>
afn <i>new-name addr</i>	name function at address <i>addr</i>
afn <i>new-name old-name</i>	rename function
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>name new-name</i>	rename argument/local variable
afvt <i>name type</i>	change type for given argument/local
axt <i>addr</i>	find data/code references to <i>addr</i>

Graphviz/graph code: ag

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

Information: i (and S)

i	show info of current file
ie	entrypoint
iz	strings in data sections
izz	strings in the whole binary
il	libraries
ii	imports
iS	sections
S	list segments (confusingly called sections!?)

Printing: p

ps [<i>@ addr</i>]	print C-string at <i>addr</i> (or current position)
pxr [<i>n</i>] [<i>@ addr</i>]	print <i>n</i> bytes (or block-size), as words, with references to flags and code (telescoping) at <i>addr</i> (or current position)
px [<i>n</i>] [<i>@ addr</i>]	hexdump — note: x is an alias for px
pxh ...	hexdump half-words (16 bits)
pxw ...	hexdump words (32 bits)
pxq ...	hexdump quad-words (64 bits)
pxl [<i>n</i>] [<i>@ addr</i>]	display <i>n</i> rows of hexdump
px/fmt [<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
pD [<i>n</i>] [<i>@ addr</i>]	disassemble <i>n</i> bytes
pd -n [<i>@ addr</i>]	disassemble <i>n</i> instructions backwards
pdf [<i>@ fn-name</i>]	disassemble function <i>fn-name</i>
pdc [<i>@ fn-name</i>]	pseudo-disassemble in C-like syntax

Debugging: d

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

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 @ addr</i>	display the value (of type <i>t-name</i>) at <i>addr</i>
t	list (base?) types
te	list enums
ts	list structs
tu	list 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 = addr</i>	link <i>t-name</i> to address <i>addr</i>
t1	list all links in readable format

Visual mode: V

Command V	enters <i>visual mode</i> .
q	exit visual-mode
c	cursor-mode, <i>tab</i> switches among stack/regs/disassembly
:	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 pseudo-syntax
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, ...
V	enter block-graph viewer
A	enter visual-assembler
n / N	seek next/previous function/flag/hit (see scr.nkey)

Seeking (in Visual Mode)

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

Debugging (in Visual Mode)

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

Flags (AKA “bookmarks”): f

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

CC	list all comments in human friendly form
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 cfg.editor (vim, ...)

Projects: P

P1	list all projects
Ps [<i>prj-name</i>]	save project <i>prj-name</i>
Po <i>prj-name</i>	open project <i>prj-name</i>
Pd <i>prj-name</i>	delete project <i>prj-name</i>

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.