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*|*pid*	debug executable *file* or 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; for instance, to learn what `pdf` does, you can first try `pd?`, then the more general `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: $-\#rows \leq r \leq -1$)
`[`*c*`]`	to display column c ($0 \leq c < \#cols$)
`:r[`*c*`]`	to display column c of row r

Examples: `afl~[0]`, `afl~malloc[0]`, `pdf~:2` and `pdf~mov:2`

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).

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‘`.

Python scripting

Assuming that Python extension has been installed (`#!` lists installed extensions) an, interactive Python interpreter can be spawned with `#!python` and a script can be run with `#!python` *script-filename*.

Inside the spawned interpreter `r2` is an *r2pipe* object that can be used to interact with the same instance of Radare, by invoking method `cmd`; e.g. `print(r2.cmd('pdf @ entry0'))`.

In a script the same behaviour can be obtained by `import`ing `r2pipe` and inizializing `r2` with `r2pipe.open("#!pipe")`.

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']
```

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`	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.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`)

Example: my `~/.radare2rc`

```
e asm.bytes=0
e scr.utf8=true
e asm.cmtright=true
e cmd.stack=px 32
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/`

a lot of other commands... TODO!

Also: `e??search` for options

Seeking: `s`

`s`	print current position/address
`s` *addx*	seek to *addx*
`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

Analysis (functions and syscalls): a

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

Graphviz/graph code: ag

ag *addr*	output graphviz code (BB at *addr* and children) E.g. view the function graph with: ag $$ | xdot -
agc *addr*	callgraph of function at *addx*
agC	full program callgraph

Information: i

i	show info of current file
ie	entrypoint
iz	strings in data sections
izz	strings in the whole binary
ii	imports
iS	sections

Printing: p

ps [@ *addx*]	print C-string at *addx* (or current position)
pxr [*n*] [@ *addx*]	print *n* bytes (or block-size), as words, with references to flags and code (telescoping) at *addx* (or current position)
px [*n*] [@ *addx*]	hexdump
pxh ...	hexdump half-words (16 bits)
pxw ...	hexdump words (32 bits)
pxq ...	hexdump quad-words (64 bits)
pxl [*n*] [@ *addx*]	display *n* rows of hexdump
px/*fmt* [@ *addx*]	gdb-style printing *fmt* (in gdb see: help x from r2: !!gdb -q -ex 'help x' -ex quit)
pd [*n*] [@ *addx*]	disassemble *n* instructions
pD [*n*] [@ *addx*]	disassemble *n* bytes
pd -*n* [@ *addx*]	disassemble *n* instructions backwards
pdf [@ *fn-name*]	disassemble function *fn-name*
pdc [@ *fn-name*]	pseudo-disassemble in C-like syntax

Debugging: d

?d *opcode*	description of *opcode* (eg. ?d jle) BUG (?): this doesn't work on Windows
dc	continue (or start) execution
dcu *addx*	continue until *addx* is reached
dcs [*name*]	continue until the next syscall (named *name*, if specified)
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 *reg-name* = *value*	set register value
drt	list register types
drt *type*	list registers of type *type* and their values
db	list breakpoints
db *addx*	add breakpoint
db -*addx*	remove breakpoint
doo *args*	(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 *C-type-def*"	define a new type
t *t-name*	show type *t-name* in pf syntax
.t *t-name* @ *addx*	display the value (of type *t-name*) at *addx*
t	list (base?) types
te	list enums
ts	list structs
tu	list unions
to *file*	parse type information from C header file
tl *t-name*	link *t-name* to current address
tl *t-name* = *addx*	link *t-name* to address *addx*
tl	list all links in readable format

Visual mode: V

Command V enters *visual mode*.

q	exit visual-mode
c	cursor-mode, *tab* switches among stack/regs/disassembly
:	execute a normal-mode command; e.g. :dm
p and P	rotate forward/backward print modes
/*str*	highlight occurences of string *str*
$	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

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
d (a digit)	jump to the target marked [*d*]
m*l* (a letter)	mark the spot with letter *l*
'*l*	jump to mark *l*
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: f

TODO

Comments: C

TODO

Writing: w

wa *asm-instr*	assemble and write opcodes; for more instructions the whole command must be quoted: "wa *asm-instr*$_1$; *asm-instr*$_2$; ..."

Projects: P

TODO

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.
