Radare2

Command line options

```
-L: List of supported IO plugins
-q: Exit after processing commands
-w: Write mode enabled
-i: Interprets a r2 script
-A: Analize executable at load time (xrefs, etc)
-n: Bare load. Do not load executable info as the entrypoint
-c'cmds': Run r2 and execute commands (eg: r2 -wqc'wx 3c @ main')
-p: Creates a project for the file being analyzed (CC add a comment when opening a
file as a project)
-: Opens r2 with the malloc plugin that gives a 512 bytes memory area to play with
(size can be changed)
    Similar to r2 malloc://512
```

Configuration properties

They can be used in evaluations:`? ${asm.tabs}`

```
e: Returs configuration properties
e <property>: Checks a specific property:
    e asm.tabs => false
e <property>=<value>: Change property value
    e asm.arch=ppc
e? help about a configuration property
    e? cmd.stack
```

You will want to set your favourite options in `~/.radare2rc` since every line there will be interpreted at the beginning of each session. Mine for reference:

```
# Show comments at right of disassembly if they fit in screen
e asm.cmtright=true

# Shows pseudocode in disassembly. Eg mov eax, str.ok = > eax = str.ok
e asm.pseudo = true

# Display stack and register values on top of disasembly view (visual mode)
e cmd.stack = true

# Solarized theme
eco solarized

# Use UTF-8 to show cool arrows that do not look like crap :)
e scr.utf8 = true
```

There is an easier interface accessible from the Visual mode, just typing `Ve`

Basic Commands

Command syntax: `[.][times][cmd][~grep][@[@iter]addr!size][|>pipe]` * `;` Command chaining: `x 3;s+3;pi 3;s+3;pxo 4;` * `|` Pipe with shell commands: `pd | less` * `!` Run shell commands: `!cat /etc/passwd` * `!!` Escapes to shell, run command and pass output to radare buffer * Note: The double exclamation mark tells radare to skip the plugin list to find an IO plugin handling this command to launch it directly to the shell. A single one will walk through the io plugin list. * `` ` `` Radare commands: ``wx `!ragg2 -i exec` `` * `~` grep * `~!` grep -v * `~[n]` grep by columns `afl~[0]` * `~:n` grep by rows `afl~:0`

```
    pi~mov,eax            ; lines with mov or eax
    pi~mov&eax            ; lines with mov and eax
    pi~mov,eax:6          ; 6 first lines with mov or eax
    pd 20~call[0]:0       ; grep first column of the first row matching 'call'
```

- `.cmd` Interprets command output

```
is* prints symbolos
.is* interprets output and define the symbols in radare (normally they are already
loaded if r2 was not invoked with -n)
```

- `..` repeats last commands (same as enter \n)
- `(` Used to define and run macros
- `$` Used to define alias
- `$$`: Resolves to current address
- Offsets (`@`) are absolute, we can use $$ for relative ones `@ $$+4`
- `?` Evaluate expression

```
[0x00000000]> ? 33 +2
35 0x23 043 0000:0023 35 00100011 35.0 0.000000

Note: | and & need to be escaped
```

- `?$?` Help for variables used in expressions
- `$$`: Here
- `$s`: File size
- `$b`: Block size
- `$l`: Opcode length
- `$j`: When `$$` is at a `jmp`, `$j` is the address where we are going to jump to
- `$f`: Same for `jmp` fail address
- `$m`: Opcode memory reference (e.g. mov eax,[0x10] => 0x10)
- `???` Help for `?` command
- `?i` Takes input from stdin. Eg `?i username`
- `??` Result from previous operations
- `?s from to [step]`: Generates sequence from to every

- `?p`: Get physical address for given virtual address
- `?P`: Get virtual address for given physical one
- `?v` Show hex value of math expr

```
?v 0x1625d4ca ^ 0x72ca4247 = 0x64ef968d
?v 0x4141414a - 0x41414140  = 0xa
```

- `?l str`: Returns the length of string
- `@@`: Used for iteractions

```
wx ff @@10 20 30      Writes ff at offsets 10, 20 and 30
wx ff @@`?s  1 10 2`  Writes ff at offsets 1, 2 and 3
wx 90 @@ sym.*        Writes a nop on every symbol
```

Positioning

```
s address: Move cursor to address or symbol
    s-5 (5 bytes backwards)
    s- undo seek
    s+ redo seek
```

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)

```
b size: Change block size
```

JSON Output

Most of commands such as (i)nfo and (p)rint commands accept a `j` to print their output in `json`

```
[0x100000d78]> ij
{"bin":{"type":"mach0","class":"MACH064","endian":"little","machine":"x86 64
all","arch":"x86","os":"osx","lang":"c","pic":true,"canary":false,"nx":false,"crypt
o":false,"va":true,"bits":64,"stripped":true,"static":false,"linenums":false,"syms"
:false,"relocs":false},"core":{"type":"Executable file","os":"osx","arch":"x86 64
all","bits":64,"endian":"little","file":"/bin/ls","fd":6,"size":34640,"mode":"r--",
"block":256,"uri":"/bin/ls","format":"mach064"}}
```

Analyze

```
aa: Analyze all (fcns + bbs) same that running r2 with -A
```

```
ahl <length> <range>: fake opcode length for a range of bytes
ad: Analyze data
    ad@rsp (analize the stack)
```

Function analysis (normal mode)

```
af: Analyze functions
afl: List all functions
    number of functions: afl~?
afi: Returns information about the functions we are currently at
afr: Rename function: structure and flag
afr off: Restore function name set by r2
afn: Rename function
    afn strlen 0x080483f0
af-: Removes metadata generated by the function analysis
af+: Define a function manually given the start address and length
    af+ 0xd6f 403 checker_loop
axt: Returns cross references to (xref to)
axf: Returns cross references from (xref from)
```

Function analysis (visual mode)

```
d, f: Function analysis
d, u: Remove metadata generated by function analysis
```

Opcode analysis:

```
ao x: Analize x opcodes from current offset
a8 bytes: Analize the instruction represented by specified bytes
```

Information

```
iI: File info
iz: Strings in data section
izz: Strings in the whole binary
iS: Sections
    iS~w returns writable sections
is: Symbols
    is~FUNC exports
il: Linked libraries
ii: Imports
ie: Entrypoint
```

Mitigations:

```
i~pic : check if the binary has position-independent-code
i~nx : check if the binary has non-executable stack
```

```
i~canary : check if the binary has canaries
```

Get function address in GOT table: `pd 1 @ sym.imp<funct>` Returns a `jmp [addr]` where `addr` is the the address of function in the GOT. Similar to `objdump -R | grep <func>`

Print

```
psz n @ offset: Print n zero terminated String
px n @ offset: Print hexdump (or just x) of n bytes
pxw n @ offset: Print hexdump of n words
    pxw size@offset  prints hexadecimal words at address
pd n @ offset: Print n opcodes disassambled
pD n @ offset: Print n bytes disassembled
pi n @ offset: Print n instructions disassambeled (no address, XREFs, etc. just
instrunctions)
pdf @ offset: Print disassembled function
    pdf~XREF (grep: XREFs)
    pdf~call (grep: calls)
pcp n @ offset: Print n bytes in python string output.
    pcp 0x20@0x8048550
    import struct
    buf = struct.pack ("32B",
    0x55,0x89,0xe5,0x83,0xzz,0xzz,0xzz,0xzz,0xf0,0x00,0x00,
    0x00,0x00,0xc7,0x45,0xf4,0x00,0x00,0x00,0x00,0xeb,0x20,
    0xc7,0x44,0x24,0x04,0x01,0x00,0x00,0x00,0xzz,0xzz)
p8 n @ offset: Print n bytes (8bits) (no hexdump)
pv: Print file contents as IDA bar and shows metadata for each byte (flags , ...)
pt: Interpret data as dates
pf: Print with format
pf.: list all formats
p=: Print entropy ascii graph
```

Write

```
wx: Write hex values in current offset
    wx 123456
    wx ff @ 4
wa: Write assembly
    wa jnz 0x400d24
wc: Write cache commit
wv: Writes value doing endian conversion and padding to byte
wo[x]: Write result of operation
    wow 11223344 @102!10
        write looped value from 102 to 102+10
        0x00000066  1122 3344 1122 3344 1122 0000 0000 0000
    wox 0x90
        XOR the current block with 0x90. Equivalent to wox 0x90 $$!$b (write from
current position, a whole block)
    wox 67 @4!10
        XOR from offset 4 to 10 with value 67
wf file: Writes the content of the file at the current address or specified offset
```

```
(ASCII characters only)
wF file: Writes the content of the file at the current address or specified offset
wt file [sz]: Write to file (from current seek, blocksize or sz bytes)
    Eg: Dump ELF files with wt @@ hit0* (after searching for ELF headers: \x7fELF)
woO 41424344 : get the index in the De Bruijn Pattern of the given word
```

Flags

Flags are labels for offsets. They can be grouped in namespaces as `sym` for symbols ...

```
f: List flags
f label @ offset: Define a flag `label` at offset
    f str.pass_len @ 0x804999c
f -label: Removes flag
fr: Rename flag
fd: Returns position from nearest flag (looking backwards). Eg => entry+21
fs: Show all flag spaces
fs flagspace: Change to the specified flag space
```

yank & paste

```
y n: Copies n bytes from current position
y: Shows yank buffer contentent with address and length where each entry was copied
from
yp: Prints yank buffer
yy offset: Paste the contents of the yank buffer at the specified offset
yt n target @ source: Yank to. Copy n bytes fromsource to target address
```

Visual Mode:

`V` enters visual mode

```
q: Exits visual mode
hjkl: move around (or HJKL) (left-down-up-right)
o: go/seek to given offset
?: Help
.: Seek EIP
<enter>: Follow address of the current jump/call
:cmd: Enter radare commands. Eg: x @ esi
d[f?]: Define cursor as a string, data, code, a function, or simply to undefine it.
    dr: Rename a function
    df: Define a function
v: Get into the visual code analysis menu to edit/look closely at the current
function.
p/P: Rotate print (visualization) modes
    hex, the hexadecimal view
    disasm, the disassembly listing
        Use numbers in [] to follow jump
        Use "u" to go back
    debug, the debugger
    words, the word-hexidecimal view
    buf, the C-formatted buffer
    annotated, the annotated hexdump.
c: Changes to cursor mode or exits the cursor mode
    select: Shift+[hjkl]
    i: Insert mode
    a: assembly inline
    A: Assembly in visual mode
    y: Copy
    Y: Paste
    f: Creates a flag where cursor points to
    <tab> in the hexdump view to toggle between hex and strings columns
V: View ascii-art basic block graph of current function
W: WebUI
x, X: XREFs to current function. ("u" to go back)
t: track flags (browse symbols, functions..)
gG: Begging or end of file
HUD
    _ Show HUD
    backspace: Exits HUD
    We can add new commands to HUD in: radare2/shlr/hud/main
;[-]cmt: Add/remove comment
m<char>: Define a bookmark
'<char>: Go to previously defined bookmark
```

ROP

```
/R opcodes: Search opcodes
    /R pop,pop,ret
/Rl opcodes: Search opcodes and print them in linear way
    /Rl jmp eax,call ebx
/a: Search assembly
    /a jmp eax
pda: Returns a library of gadgets that can be use. These gadgets are obtained by
disassmbling byte per byte instead of obeying to opcode length
```

Search depth can be configure with following properties:

```
e search.roplen = 4  (change the depth of the search, to speed-up the hunt)
```

Searching

```
/ bytes: Search bytes
    \x7fELF
```

Example: Searching function preludes:

```
push ebp
mov ebp, esp
```

```
Opcodes: 5589e5

/x 5589e5
    [# ]hits: 54c0f4 < 0x0804c600  hits = 1
    0x08049f70 hit0_0 5589e557565383e4f081ec
    0x0804c31a hit0_1 5589e583ec18c704246031
    0x0804c353 hit0_2 5589e583ec1889442404c7
    0x0804c379 hit0_3 5589e583ec08e87cffffff
    0x0804c3a2 hit0_4 5589e583ec18c70424302d

pi 5 @@hit* (Print 5 first instructions of every hit)
```

Its possible to run a command for each hit. Use the `cmd.hit` property:

```
e cmd.hit=px
```

Comments and defines

```
Cd [size]: Define as data
C- [size]: Define as code
Cs [size]: Define as String
Cf [size]: Define as struct
    We can define structures to be shown in the disassmbly
CC: List all comments or add a new comment in console mode
    C* Show all comments/metadata
    CC <comment> add new comment
    CC- remove comment
```

Magic files

```
pm: Print Magic files analysis
    [0x00000000]> pm
    0x00000000 1 ELF 32-bit LSB executable, Intel 80386, version 1
```

Search for magic numbers

```
/m [magicfile]: Search magic number headers with libmagic
```

Search can be controlled with following properties:

```
search.align
search.from (0 = beginning)
search.to (0 = end)
search.asmstr
search.in
```

Yara

Yara can also be used for detecting file signatures to determine compiler types, shellcodes, protections and more.

```
:yara scan
```

Zignatures

Zignatures are useful when dealing with stripped binaries. We can take a non-stripped binary, run zignatures on it and apply it to a different binary that was compiled statically with the same libraries.

```
zg <language> <output file>: Generate signatures
    eg: zg go go.z
Run the generated script to load signatures
    eg: . go.z
z: To show signatures loaded:
```

Zignatures are applied as comments:

```
r2-(pid2)> pd 35 @ 0x08049adb-10
|          0x08049adb   call fcn.0805b030
|             fcn.0805b030(unk, unk, unk, unk) ; sign.sign.b.sym.fmt.Println
|          0x08049ae0   add esp, 0xc
|          0x08049ae3   call fcn.08095580
```

Compare files

```
r2 -m 0xf0000 /etc/fstab    ; Open source file
o /etc/issue                ; Open file2 at offset 0
o                           ; List both files
cc offset: Diff by columns between current offset address and "offset"
```

Graphs

Basic block graphs

```
af: Load function metadata
ag $$ > a.dot: Dump basic block graph to file
ag $$ | xdot: Show current function basic block graph
```

Call graphs

```
af: Load function metadata
agc $$ > b.dot: Dump basic block graph to file
```

Convert .dot in .png

```
dot -Tpng -o /tmp/b.png b.dot
```

Generate graph for file:

```
radiff2 -g main crackme.bin crackme.bin > /tmp/a
xdot /tmp/a
```

Debugger

Start r2 in debugger mode. r2 will fork and attach

```
r2 -d [pid|cmd|ptrace] (if command contains spaces use quotes: r2 -d "ls /")

ptrace://pid (debug backend does not notice, only access to mapped memory)
```

To pass arguments:

```
r2 -d rarun2 program=pwn1 arg1=$(python exploit.py)
```

To pass stdin:

```
r2 -d rarun2 program=/bin/ls stdin=$(python exploit.py)
```

Commands

```
do: Reopen program
dp: Shows debugged process, child processes and threads
dc: Continue
dcu <address or symbol>: Continue until symbol (sets bp in address, continua until
bp and remove bp)
dc[sfcp]: Continue until syscall(eg: write), fork, call, program address (To exit a
library)
ds: Step in
dso: Step out
dss: Skip instruction
dr register=value: Change register value
dr(=)?: Show register values
db address: Sets a breakpoint at address
    db sym.main add breakpoint into sym.main
    db 0x804800 add breakpoint
    db -0x804800 remove breakpoint
dsi (conditional step): Eg: "dsi eax==3,ecx>0"
dbt: Shows backtrace
drr: Display in colors and words all the refs from registers or memory
dm: Shows memory map (* indicates current section)
    [0xb776c110]> dm
    sys 0x08048000 - 0x08062000 s r-x /usr/bin/ls
    sys 0x08062000 - 0x08064000 s rw- /usr/bin/ls
    sys 0xb776a000 - 0xb776b000 s r-x [vdso]
    sys 0xb776b000 * 0xb778b000 s r-x /usr/lib/ld-2.17.so
    sys 0xb778b000 - 0xb778d000 s rw- /usr/lib/ld-2.17.so
    sys 0xbfe5d000 - 0xbfe7e000 s rw- [stack]
```

To follow child processes in forks (set-follow-fork-mode in gdb)

```
dcf until a fork happen
then use dp to select what process you want to debug.
```

PEDA like details: `drr;pd 10@-10;pxr 40@esp`

Debug in visual mode

```
toggl breakpoints with F2
single-step with F7 (s)
step-over with F8 (S)
continue with F9
```

WebGUI (Enyo)

```
=h: Start the server
=H: Start server and browser
```

Radare2 suite commandRadare2 suite commands

All suite commands include a `-r` flag to generate instructions for r2

rax2 - Base conversion

```
-e: Change endian
-k: random ASCII art to represent a number/hash. Similar to how SSH represents keys
-s: ASCII to hex
    rax2 -S hola (from string to hex)
    rax2 -s 686f6c61 (from hex to string)
```

```
-S: binary to hex (for files)
```

rahash2 - Entropy, hashes and checksums

```
-a: Specify the algorithm
-b XXX: Block size
-B: Print all blocks
-a entropy: Show file entropy or entropy per block (-B -b 512 -a entropy)
```

radiff2 - File diffing

```
-s: Calculate text distance from two files.
-d: Delta diffing (For files with different sizes. Its not byte per byte)
-C: Code diffing (instead of data)
```

Examples:

```
Diff original and patched on x86_32, using graphdiff algorithm
    radiff2 -a x86 -b32 -C original patched
Show differences between original and patched on x86_32
    radiff2 -a x86 -b32 original patched :
```

rasm2 - Assembly/Disassembly

```
-L: Supported architectures
-a arch instruction: Sets architecture
    rasm2 -a x86 'mov eax,30' => b81e000000
-b tam: Sets block size
-d: Disassembly
    rasm2 -d b81e000000 => mov eax, 0x1e
-C: Assembly in C output
    rasm2 -C 'mov eax,30' => "\xb8\x1e\x00\x00\x00"
-D: Disassemble showing hexpair and opcode
    rasm2 -D b81e0000 => 0x00000000   5               b81e000000  mov eax, 0x1e
-f: Read data from file instead of ARG.
-t: Write data to file
```

rafind2 - Search

```
-Z: Look for Zero terminated strings
-s str: Look for specifc string
```

ragg2 - Shellcode generator, C/opcode compiler

```
-P: Generate De Bruijn patterns
    ragg2 -P 300 -r
-a arch: Configure architecture
-b bits: Specify architecture bits (32/64)
-i shellcode: Specify shellcode to generate
-e encoder: Specify encoder
```

Example:

```
Generate a x86, 32 bits exec shellcode
    ragg2 -a x86 -b 32 -i exec
```

rabin2 - Executable analysis: symbols, imports, strings ...

```
-I: Executable information
-C: Returns classes. Useful to list Java Classes
-l: Dynamic linked libraries
-s: Symbols
-z: Strings
```

rarun2 - Launcher to run programs with different environments, args, stdin, permissions, fds

Examples:

```
r2 -b 32 -d rarun2 program=pwn1 arg1=$(ragg2 -P 300 -r) : runs pwn1 with a De
Bruijn Pattern as first argument, inside radare2's debugger, and force 32 bits
r2 -d rarun2 program=/bin/ls stdin=$(python exploit.py) : runs /bin/ls with the
output of exploit.py directed to stdin
```