```
[0x0000818c 255 alxchk_ls]> pd $r @ loc.00008189+3 # 0x818c
┌ (fcn) fcn.000081a9 283
│         0x0000818c    488b1c24      mov rbx, [rsp]
│         0x00008190    488b6c2408    mov rbp, [rsp+0x8]
│         0x00008195    4c8b642410    mov r12, [rsp+0x10]
│         0x0000819a    4c8b6c2418    mov r13, [rsp+0x18]
│         0x0000819f    4c8b742420    mov r14, [rsp+0x20]
│         0x000081a4    4883c428      add rsp, 0x28
│         0x000081a8    c3            ret
│         0x000081a9    0f1f80000000. nop [rax]
│    ┌──> 0x000081b0    498b30        mov rsi, [r8]
│    ╎    0x000081b3    4839ee        cmp rsi, rbp
│    ┌──< 0x000081b6    7427          jz 0x81df [1]
│    ╎    0x000081b8    4889ef        mov rdi, rbp
│    ╎        ; CODE (CALL) XREF from 0x000081f8 (fcn.000081f8)
│    ╎    0x000081bb    41ff542438    call qword [r12+0x38] ; (fcn.000081f8) [2]
│    ╎       fcn.000081f8()
│    ╎    0x000081c0    84c0          test al, al
│    ┌──< 0x000081c2    7514          jnz 0x81d8 [3]
│    ╎    0x000081c4    488b5b08      mov rbx, [rbx+0x8]
│    ╎    0x000081c8    4c8b4308      mov r8, [rbx+0x8]
│    ╎    0x000081cc    4d85c0        test r8, r8
│    └──< 0x000081cf    75df          jnz 0x81b0 [4]
│    ╎    0x000081d1    31f6          xor esi, esi
│    ╎        ; CODE (CALL) XREF from 0x00008189 (fcn.0000805c)
│    ╎    0x000081d3    ebb4          jmp loc.00008189 [5]
│    ╎    0x000081d5    0f1f00        nop [rax]
│    └──> 0x000081d8    4c8b4308      mov r8, [rbx+0x8]
│         0x000081dc    498b30        mov rsi, [r8]
│         0x000081df    4584ed        test r13b, r13b
│         0x000081e2    74a5          jz loc.00008189 [6]
│         0x000081e4    498b4008      mov rax, [r8+0x8]
│         0x000081e8    49c7000000000. mov qword [r8], 0x0
│         0x000081ef    48894308      mov [rbx+0x8], rax
│         0x000081f3    498b442448    mov rax, [r12+0x48]
│         ; CODE (CALL) XREF from 0x000081bb (fcn.0000805c)
┌ (fcn) fcn.000081f8 175
│         0x000081f8    49894008      mov [r8+0x8], rax
│         0x000081fc    4d89442448    mov [r12+0x48], r8
│         0x00008201    eb86          jmp loc.00008189 [7]
│         0x00008203    0f1f440000    nop [rax+rax]
│         0x00008208    48c703000000. mov qword [rbx], 0x0
│         0x0000820f    e975ffffff    jmp loc.00008189 [8]
│         ; CODE (CALL) XREF from 0x00008214 (fcn.00008214)
┌ (fcn) fcn.00008214 147
│         0x00008214    6666662e0f1. o16 nop [cs:rax+rax]
│         ; CODE (CALL) XREF from 0x00005762 (fcn.000041b0)
│         ; CODE (CALL) XREF from 0x00006251 (fcn.000041b0)
│         ; CODE (CALL) XREF from 0x000063d2 (fcn.000041b0)
┌ (fcn) fcn.00008220 135
│         0x00008220    488b07        mov rax, [rdi]
```

Table of Contents

1. introduction
2. Introduction
    i. History
    ii. Overview
    iii. Getting radare2
    iv. Compilation and Portability
    v. Compilation on Windows
    vi. Command-line Flags
    vii. Basic Usage
    viii. Command Format
    ix. Expressions
    x. Rax2
    xi. Basic Debugger Session
    xii. Contributing to radare2
3. Configuration
    i. Colors
    ii. Common Configuration Variables
4. Basic Commands
    i. Seeking
    ii. Block Size
    iii. Sections
    iv. Mapping Files
    v. Print Modes
    vi. Flags
    vii. Write
    viii. Zoom
    ix. Yank/Paste
    x. Comparing Bytes
5. Visual mode
    i. Visual Cursor
    ii. Visual Inserts
    iii. Visual XREFS
    iv. Visual Configuration Editor
6. Searching bytes
    i. Basic Searches
    ii. Configurating the Search
    iii. Pattern Search
    iv. Automatization
    v. Backward Search
    vi. Search in Assembly
    vii. Searching for AES Keys
7. Disassembling

    i. Adding Metadata
    ii. ESIL
8. Rabin2
    i. File Identification
    ii. Entrypoint
    iii. Imports
    iv. Symbols (exports)
    v. Libraries
    vi. Strings
    vii. Program Sections
9. Radiff2
    i. Binary Diffing
10. Rasm2
    i. Assemble
    ii. Disassemble
11. Analysis
    i. Code Analysis
12. Rahash2
    i. Rahash Tool
13. Debugger
    i. Registers
14. Remote Access Capabilities
    i. Remoting Capabilities
15. Plugins
    i. Plugins
16. Crackmes
    i. IOLI
        i. IOLI 0x00
        ii. IOLI 0x01
17. Reference Card

R2 "Book"

Welcome to the Radare2 Book

- Webpage: https://www.gitbook.com/book/radare/radare2book/details

- Online: http://radare.gitbooks.io/radare2book/content/
- PDF: https://www.gitbook.com/download/pdf/book/radare/radare2book
- Epub: https://www.gitbook.com/download/epub/book/radare/radare2book
- Mobi: https://www.gitbook.com/download/mobi/book/radare/radare2book

Introduction

This book aims to cover most usage aspects of radare2. A framework for reverse engineering and analyzing binaries.

--pancake

History

The radare project started in February of 2006, aiming to provide a free and simple command-line interface for a hexadecimal editor supporting 64-bit offsets, to make searches and to help recovering data from hard-disks.

Since then, the project has grown with the aim changed to provide a complete framework for analyzing binaries with some basic *NIX concepts in mind, like famous "everything is a file", "small programs that interact using stdin/stdout" or "keep it simple".

It is mostly a single-person project, but some contributions (in source, patches, ideas or specient) have been made and are really appreciated.

The project is composed of a hexadecimal editor as the central point of the project with assembler/disassembler, code analysis, scripting features, analysis and graphs of code and data, easy unix integration, ...

Overview

The Radare2 project is a set of small command-line utilities that can be used together or independently.

radare2

The core of the hexadecimal editor and debugger. radare2 allows you to open a number of input/output sources as if they were simple, plain files, including disks, network connections, kernel drivers, processes under debugging, and so on.

It implements an advanced command line interface for moving around a file, analyzing data, disassembling, binary patching, data comparison, searching, replacing, visualizing. It can be scripted with a variety of languages, including Ruby, Python, Lua, and Perl.

rabin2

A program to extract information from executable binaries, such as ELF, PE, Java CLASS, and Mach-O. rabin2 is used by the core to get exported symbols, imports, file information, cross references (xrefs), library dependencies, sections, etc.

rasm2

A command line assembler and disassembler for multiple architectures (including Intel x86 and x86-64, MIPS, ARM, PowerPC, Java, and MSIL).

rasm2 Examples

```
$ rasm2 -a java 'nop'
00

$ rasm2 -a x86 -d '90'
nop

$ rasm2 -a x86 -b 32 'mov eax, 33'
b821000000

$ echo 'push eax;nop;nop' | rasm2 -f -
5090
```

rahash2

An implementation of a block-based hash tool. From small text strings to large disks, rahash2 supports multiple algorithms, including MD4, MD5, CRC16, CRC32, SHA1, SHA256, SHA384, SHA512, par, xor, xorpair, mod255, hamdist, or entropy. rahash2 can be used to check the integrity of, or track changes to, big files, memory dumps, and disks.

radiff2

A binary diffing utility that implements multiple algorithms. It supports byte-level or delta diffing for binary files, and code-analysis diffing to find changes in basic code blocks obtained from the radare code analysis, or from the IDA analysis using the rsc idc2rdb script.

rafind2

A program to find byte patterns in files.

ragg2

A frontend for r_egg. ragg2 compiles programs written in a simple high-level language into tiny binaries for x86, x86-64, and ARM.

Examples

```
$ cat hi.r
/* hello world in r_egg */
write@syscall(4);
exit@syscall(1);

main@global(128) {
  .var0 = "hi!\n";
  write(1,.var0, 4);
  exit(0);
}
$ ragg2 -O -F hi.r
$ ./hi
hi!

$ cat hi.c
main() {
  write(1, "Hello0, 6);
  exit(0);
}
$ ragg2 hi.c
$ ./hi.c.bin
Hello
```

rarun2

A launcher for running programs within different environments, with different arguments, permissions, directories, and overridden default file descriptors. rarun2 is useful for:

- Crackmes
- Fuzzing
- Test suites

Sample rarun2 script

```
$ cat foo.rr2
#!/usr/bin/rarun2
program=./pp400
arg0=10
stdin=foo.txt
chdir=/tmp
#chroot=.
./foo.rr2
```

Connecting a Program to a Socket

```
$ nc -l 9999
$ rarun2 program=/bin/ls connect=localhost:9999
```

Debugging a Program by Redirecting IO to Another Terminal

1. open a new terminal and type 'tty' to get a terminal name:

```
$ tty ; clear ; sleep 999999
/dev/ttyS010
```

2. In another terminal, run `r2`:

```
$ r2 -d rarun2 program=/bin/ls stdio=/dev/ttys010
```

rax2

An utility that aims to be a minimalistic mathematical expression evaluator for the shell. It is useful for making base conversions between floating point values, hexadecimal representations, hexpair strings to ASCII, octal to integer, etc. It supports both endianness settings and can be used as an interactive shell if no arguments are given.

Examples

```
$ rax2 1337
0x539

$ rax2 0x400000
4194304

$ rax2 -b 01111001
y

$ rax2 -S radare2
72616461726532

$ rax2 -s 617765736f6d65
awesome
```

Getting radare2

You can get radare from the website, http://radare.org/, or the GitHub repository, https://github.com/radare/radare2.

Binary packages are available for a number of operating systems (Ubuntu, Maemo, Gentoo, Windows, iPhone, and so on). Yet, you are highly encouraged to get the source and compile it yourself to better understand the dependencies, to make examples more accessible and of course to have the most recent version.

A new stable release is typically published every month. Nightly tarballs are sometimes available at http://bin.rada.re/.

The radare development repository is often more stable than the 'stable' releases. To obtain the latest version:

```
$ git clone https://github.com/radare/radare2.git
```

This will probably take a while, so take a coffee break and continue reading this book.

To update your local copy of the repository, use `git pull` anywhere in the radare2 source code tree:

```
$ git pull
```

If you have local modifications of the source, you can revert them (and loose them!) with:

```
$ git reset --hard HEAD
```

Or send me a patch:

```
$ git diff > radare-foo.patch
```

The most common way to get r2 updated and installed system wide is by using:

```
$ sys/install.sh
```

Helper Scripts

Take a look at the sys/* scripts, those are used to automatize stuff related to syncing, building and installing r2 and its bindings.

The most important one is sys/install.sh. It will pull, clean, build and symstall r2 system wide.

Symstalling is the process of installing all the programs, libraries, documentation and data files using symlinks instead of copying the files.

By default it will be installed in /usr, but you can define a new prefix as argument.

This is useful for developers, because it permits them to just run 'make' and try changes without having to run make install again.

Cleaning Up

Cleaning up the source tree is important to avoid problems like linking to old objects files or not updating objects after an ABI change.

The following commands may help you to get your git clone up to date:

```
$ git clean -xdf
$ git reset --hard @~10
$ git pull
```

If you want to remove previous installations from your system, you must run the following commands:

```
$ ./configure --prefix=/usr/local
$ make purge
```

Compilation and Portability

Currently the core of radare2 can be compiled on many systems and architectures, but the main development is done on GNU/Linux with GCC, and on MacOS X with clang. Radare is also known to compile on many different systems and architectures (including TCC and SunStudio).

People often want to use radare as a debugger for reverse engineering. Currently, the debugger layer can be used on Windows, GNU/Linux (Intel x86 and x86_64, MIPS, and ARM), FreeBSD, NetBSD, and OpenBSD (Intel x86 and x86_64). There are plans to support Solaris and MacOS X.

Compared to core, the debugger feature is more restrictive portability-wise. If the debugger has not been ported to your favorite platform, you can disable the debugger layer with the --without-debugger `configure` script option when compiling radare2.

Note that there are I/O plugins that use GDB, GDB Remote, or Wine as back-ends, and therefore rely on presence of corresponding third-party tools.

To build on a system using ACR/GMAKE (e.g. on *BSD systems):

```
$ ./configure --prefix=/usr
$ gmake
$ sudo gmake install
```

There is also a simple script to do this automatically:

```
$ sys/install.sh
```

Static Build

You can build statically radare2 and all the tools with the command:

```
$ sys/static.sh
```

Docker

Radare2 repository ships a Dockerfile that you can use with Docker.

This dockerfile is also used by Remnux distribution from SANS, and is available on the docker registryhub.

Cleaning Up Old Radare2 Installations

```
./configure --prefix=/old/r2/prefix/installation
```

```
make purge
```

Compilation on Windows

The easy way to compile things for Windows is using MinGW32. The w32 builds distributed from the radare homepage are generated from a GNU/Linux box using MinGW32 and they are tested with Wine. Also keep in mind, that MinGW-w64 wan't tested, so no guarantees here.

Be sure to setup your MinGW32 to compile with **thread model: win32**, not **posix**, and target should be **mingw32**.

The following is an example of compiling with MinGW32 (you need to have installed **zip** for Windows):

```
CC=i486-mingw32-gcc ./configure
make
make w32dist
zip -r w32-build.zip w32-build
```

This generates a native, 32-bit console application for Windows. The 'i486-mingw32-gcc' compiler is the one I have in my box, you will probably need to change this.

Cygwin is another possibility; however, issues related to Cygwin libraries can make debugging difficult. But using binary compiled for Cygwin will allow you to use Unicode in the Windows console, and to have 256 colors.

Please, be sure to build radare2 from the same environment you're going to use r2 in. If you are going to use r2 in MinGW32 shell or cmd.exe — you should build r2 in the MinGW32 environment. And if you are going to use r2 in Cygwin — you have to build r2 from the Cygwin shell. Since Cygwin is more UNIX-compatible than MinGW, the radare2 supports more colors and Unicode symbols if build using the former one.

There is a script that automates process of detecting the crosscompiler toolchain configuration, and builds a zip file containing r2 programs and libraries that can be deployed on Windows or Wine:

```
sys/mingw32.sh
```

Mingw-W64

- Download the MSYS2 distribution from the official site: http://msys2.github.io/
- Setup the proxy (if needed):

```
export http_proxy=<myusername>:<mypassword>@zz-wwwproxy-90-v:8080
export https_proxy=$http_proxy
export ftp_proxy=$http_proxy
export rsync_proxy=$http_proxy
export rsync_proxy=$http_proxy
export no_proxy="localhost,127.0.0.1,localaddress,.localdomain.com"
```

- Update packages:

```
pacman --needed -Sy bash pacman pacman-mirrors msys2-runtime mingw-w64-x86_64-toolcha
```

- Close MSYS2, run it again from Start menu and update the rest with

```
pacman -Su
```

- Install the building essentials:

```
pacman -S git make zip gcc
```

- Compile the radare2:

```
./configure ; make ; make w32dist
```

Bindings

To build radare2 bindings, you will need to install Vala (valac) for Windows

Then download valabind and build it:

```
git clone https://github.com/radare/valabind.git valabind
cd valabind
make
make install
```

After you installed valabind, you can build radare2-bindings, for example for Python and Perl:

```
git clone https://github.com/radare/radare2-bindings.git radare2-bindings
cd radare2-bindings
./configure --enable=python,perl
make
make install
```

Command-line Options

The radare core accepts many flags from command line.

An excerpt from usage help message:

```
$ radare2 -h
Usage: r2 [-dDwntLqv] [-P patch] [-p prj] [-a arch] [-b bits] [-i file] [-s addr] [-B blo

   -a [arch]    set asm.arch
   -A           run 'aa' command to analyze all referenced code
   -b [bits]    set asm.bits
   -B [baddr]   set base address for PIE binaries
   -c 'cmd..'   execute radare command
   -C           file is host:port (alias for -c+=http://%s/cmd/)
   -d           use 'file' as a program for debug
   -D [backend] enable debug mode (e cfg.debug=true)
   -e k=v       evaluate config var
   -f           block size = file size
   -h, -hh      show help message, -hh for long
   -i [file]    run script file
   -k [kernel]  set asm.os variable for asm and anal
   -l [lib]     load plugin file
   -L           list supported IO plugins
   -m [addr]    map file at given address
   -n           disable analysis
   -N           disable user settings
   -q           quiet mode (no promt) and quit after -i
   -p [prj]     set project file
   -P [file]    apply rapatch file and quit
   -s [addr]    initial seek
   -S           start r2 in sandbox mode
   -t           load rabin2 info in thread
   -v, -V       show radare2 version (-V show lib versions)
   -w           open file in write mode
```

Common usage patterns of command-line options.

- Open a file in write mode without parsing the file format headers.

  $ r2 -nw file

- Quickly get into an r2 shell without opening any file.

  $ r2 -

Specify which sub-binary you want to select when opening a fatbin file:

```
$ r2 -a ppc -b 32 ls.fat
```

Run a script before showing interactive command-line prompt:

```
$ r2 -i patch.r2 target.bin
```

Execute a command and quit without entering the interactive mode:

```
$ r2 -qc ij hi.bin > imports.json
```

Configure an eval variable:

```
$ r2 -e scr.color=false blah.bin
```

Debug a program:

```
$ r2 -d ls
```

Use an existing project file:

```
$ r2 -p test
```

Basic Radare Usage

The learning curve for radare is usually somewhat steep at the beginning. Although after an hour of using it you should easily understand how most things work, and how to combine various tools radare offers, you are encouraged to read the rest of this book to understand how some non-trivial things work, and to ultimately improve your skills with radare.

Navigation, inspection and modification of a loaded binary file is performed using three simple actions: seek (to position), print (buffer), and alterate (write, append).

The 'seek' command is abbreviated as `s` and accepts an expression as its argument. The expression can be something like `10`, `+0x25`, or `[0x100+ptr_table]`. If you are working with block-based files, you may prefer to set the block size to a required value with `b` command, and seek forward or backwards with positions aligned to it. Use `>` and `<` commands to navigate this way.

The 'print' command is abbreviated as `p` and has a number of submodes — the second letter specifying a desired print mode. Frequent variants include `px` to print in hexadecimal, and `pd` for disassembling.

To be allowed to write files, specify the `-w` option to radare when opening a file. The `w` command can be used to write strings, hexpairs (`x` subcommand), or even assembly opcodes (`a` subcommand). Examples:

```
> w hello world         ; string
> wx 90 90 90 90        ; hexpairs
> wa jmp 0x8048140      ; assemble
> wf inline.bin         ; write contents of file
```

Appending a `?` to a command will show its help message, for example, `p?`.

To enter visual mode, press `V<enter>`. Use `q` to quit visual mode and return to the prompt. In visual mode you can use HJKL keys to navigate (left, down, up, and right, respectively). You can use these keys in cursor mode toggled by `c` key. To select a byte range in cursor mode, hold down `SHIFT` key, and press navigation keys HJKL to mark your selection. While in visual mode, you can also overwrite bytes by pressing `i`. You can press `TAB` to switch between the hex (middle) and string (right) columns. Pressing `q` inside the hex panel returns you to visual mode.

Command Format

A general format for radare commands is as follows:

```
[.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ;
```

Commands are identified by a single case-sensitive character [a-zA-Z]. To repeatedly execute a command, prefix the command with a number:

```
px    # run px
3px   # run px 3 times
```

The `!` prefix is used to execute a command in shell context. If a single exclamation mark is used, commands will be sent to the system() hook defined in currently loaded I/O plugin. This is used, for example, by the ptrace I/O plugin, which accepts debugger commands from radare interface.

A few examples:

```
ds                   ; call the debugger's 'step' command
px 200 @ esp         ; show 200 hex bytes at esp
pc > file.c          ; dump buffer as a C byte array to file.c
wx 90 @@ sym.*       ; write a nop on every symbol
pd 2000 | grep eax   ; grep opcodes that use the 'eax' register
px 20 ; pd 3 ; px 40  ; multiple commands in a single line
```

The `@` character is used to specify a temporary offset at which the command to its left will be executed. The original seek position in a file is then restored. For example, `pd 5 @ 0x100000fce` to disassemble 5 instructions at address 0x100000fce.

The `~` character enables internal grep-like function used to filter output of any command. For example:

```
pd 20~call           ; disassemble 20 instructions and grep output for 'call'
```

Additionally, you can either grep for columns or rows:

```
pd 20~call!0         ; get first row
pd 20~call!1         ; get second row
pd 20~call[0]        ; get first column
pd 20~call[1]        ; get second column
```

Or even combine them:

```
pd 20~call[0]!0      ; grep the first column of the first row matching 'call'
```

This internal grep function is a key feature for scripting radare, because it can be used to iterate over a list of offsets or data generated by disassembler, ranges, or any other command. Refer to the macros section (iterators) for more information.

Expressions

Expressions are mathematical representations of 64-bit numerical values. They can be displayed in different formats, be compared or used with all commands accepting numeric arguments. Expressions can use traditional arithmetic operations, as well as binary and boolean ones. To evaluate mathematical expressions prepend them with command `?`. For example:

```
[0xB7F9D810]> ? 0x8048000
134512640 0x8048000 01001100000 128.0M 804000:0000 134512640 00000000 134512640.0 0.00000
[0xB7F9D810]> ? 0x8048000+34
134512674 0x8048022 01001100042 128.0M 804000:0022 134512674 00100010 134512674.0 0.00000
[0xB7F9D810]> ? 0x8048000+0x34
134512692 0x8048034 01001100064 128.0M 804000:0034 134512692 00110100 134512692.0 0.00000
[0xB7F9D810]> ? 1+2+3-4*3
-6 0xfffffffffffffffa 0177777777777777777772 17179869183.0G fffff000:0ffa -6
```

Supported arithmetic operations are:

-
  - : addition
-
  - : substraction
-
  - : multiplication
- / : division
- % : modulus
- : shift right
- < : shift left

Binary operations should be escaped:

- | : logical OR // ("? 0001010 | 0101001")
- \& : logical AND

Values are numbers representable in several formats:

```
0x033   : hexadecimal
3334    : decimal
sym.fo  : resolve flag offset
10K     : KBytes  10*1024
10M     : MBytes  10*1024*1024
```

You can also use variables and seek positions to build complex expressions. Available values include:

```
?@?    or stype @@?      ; misc help for '@' (seek), '~' (grep) (see ~??)
```

```
?$?           ; show available '$' variables
$$            ; here (the current virtual seek)
$l            ; opcode length
$s            ; file size
$j            ; jump address (e.g. jmp 0x10, jz 0x10 => 0x10)
$f            ; jump fail address (e.g. jz 0x10 => next instruction)
$m            ; opcode memory reference (e.g. mov eax,[0x10] => 0x10)
```

Some more examples:

```
[0x4A13B8C0]> :? $m + $l
140293837812900 0x7f98b45df4a4 03771426427372244 130658.0G 8b45d000:04a4 140293837812900


[0x4A13B8C0]> :pd 1 @ +$l
0x4A13B8C2   call 0x4a13c000
```

Rax2

The `rax2` utility comes with the radare framework and aims to be a minimalistic expression evaluator for the shell. It is useful for making base conversions between floating point values, hexadecimal representations, hexpair strings to ascii, octal to integer. It supports endianness and can be used as a shell if no arguments are given.

```
$ rax2 -h

Usage: rax2 [options] [expr ...]
int   ->  hex           ;  rax2 10
hex   ->  int           ;  rax2 0xa
-int  ->  hex           ;  rax2 -77
-hex  ->  int           ;  rax2 0xffffffb3
int   ->  bin           ;  rax2 b30
bin   ->  int           ;  rax2 1010d
float ->  hex           ;  rax2 3.33f
hex   ->  float         ;  rax2 Fx40551ed8
oct   ->  hex           ;  rax2 35o
hex   ->  oct           ;  rax2 Ox12 (O is a letter)
bin   ->  hex           ;  rax2 1100011b
hex   ->  bin           ;  rax2 Bx63
raw   ->  hex           ;  rax2 -S < /binfile
hex   ->  raw           ;  rax2 -s 414141
-b    binstr -> bin     ;  rax2 -b 01000101 01110110
-B    keep base         ;  rax2 -B 33+3 -> 36
-d    force integer     ;  rax2 -d 3 -> 3 instead of 0x3
-e    swap endianness   ;  rax2 -e 0x33
-f    floating point    ;  rax2 -f 6.3+2.1
-h    help              ;  rax2 -h
-k    randomart         ;  rax2 -k 0x34 1020304050
-n    binary number     ;  rax2 -e 0x1234   # 34120000
-s    hexstr -> raw     ;  rax2 -s 43 4a 50
-S    raw -> hexstr     ;  rax2 -S < /bin/ls > ls.hex
-t    tstamp -> str     ;  rax2 -t 1234567890
-x    hash string       ;  rax2 -x linux osx
-u    units             ;  rax2 -u 389289238 # 317.0M
-v    version           ;  rax2 -V
```

Some examples:

```
$ rax2 3+0x80
0x83

$ rax2 0x80+3
131

$ echo 0x80+3 | rax2
131

$ rax2 -s 4142
AB
```

```
$ rax2 -S AB
4142

$ rax2 -S < bin.foo
...

$ rax2 -e 33
0x21000000

$ rax2 -e 0x21000000
33

$ rax2 -k 90203010
+--[0x10302090]---+
|Eo. .            |
| . . . .         |
|      o          |
|       .         |
|        S        |
|                 |
|                 |
|                 |
|                 |
+-----------------+
```

Basic Debugger Session

To debug a program, start radare with `-d` option. You can attach to a running process by specifying its PID, or you can start a new program by specifying its name and parameters: $ pidof mc 32220 $ r2 -d 32220

```
$ r2 -d /bin/ls
```

In the second case, the debugger will fork and load the debuggee `ls` program in memory. It will pause its execution early in `ld.so` dynamic linker. Therefore, do not expect to see an entrypoint or shared libraries at this point. You can override this behavior by setting another name for and entry breakpoint. To do this, add a radare command `e dbg.bep=entry` or `e dbg.bep=main` to your startup script, usually it is `~/.radare2rc`. Be warned though that certain malware or other tricky programs can actually execute code before `main()` and thus you'll be unable to control them.

Below is a list of most common commands used with debugger:

```
> d?          ; get help on debugger commands
> ds 3        ; step 3 times
> db 0x8048920  ; setup a breakpoint
> db -0x8048920 ; remove a breakpoint
> dc          ; continue process execution
> dcs        ; continue until syscall
> dd            ; manipulate file descriptors
> dm          ; show process maps
> dmp A S rwx  ; change page at A with size S protection permissions
> dr eax=33 ; set register value. eax = 33
```

Maybe a simpler method to use debugger in radare is to switch it to visual mode. That way you will not have to remember many commands nor to keep program state in your mind. To enter visual mode use `V`:

```
[0xB7F0C8C0]> V
```

The initial view after entering visual mode is a hexdump view of current target program counter (e.g., EIP for x86). Pressing `p` will allow you to cycle through the rest of visual mode views. You can press `p` and `P` to rotate through the most commonly used print modes. Use F7 or `s` to step into and F8 or `S` to step over current instruction. With the `c` key you can toggle the cursor mode to mark a byte range selection (for example, to later overwirte them with nop). You can set breakpoints with `F2` key.

In visual mode you can enter regular radare commands by prepending them with `:`. For example, to dump a one block of memory contents at ESI: x @ esi

To get help on visual mode, press `?`. To scroll help screen, use arrows. To exit help view, press `q`.

A frequently used command is `dr`, to read or write values of target's general purpose registers. You can also manipulate the hardware and extended/floating point registers.

Contributing

Radare2 Book

If you want to contribute to the Radare2 book, you can do it at the Github repository. Suggested contributions include:

- Crackme writeups
- CTF writeups
- Documentation on how to use Radare2
- Documentation on developing for Radare2
- Conference presentations/workshops using Radare2
- Missing content from the Radare1 book updated to Radare2

Please get permission to port any content you do not own/did not create before you put it in the Radare2 book.

Configuration

The core reads `~/.radare2rc` while starting. You can add `e` commands to this file to tune radare configuration to your taste.

To prevent radare from parsing this file at start, pass it `-n` option. To have a less verbose output for batch mode runs, it is also better to decrease verbosity level with `-v` command-line option.

All the configuration of radare is done with the `eval` commands. A typical startup configuration file looks like this:

```
$ cat ~/.radare2rc
e scr.color = true
e dbg.bep   = loader
```

Configuration can also be changed with `-e` command-line option. This way you can adjust configuration from the command line, keeping the .radare2rc file intact. For example, to start with empty configuration and then adjust `scr.color` and `asm.syntax` the following line may be used:

```
$ radare2 -n -e scr.color=true -e asm.syntax=intel -d /bin/ls
```

Internally, the configuration is stored in a hash table. The variables are grouped in namespaces: `cfg.`, `file.`, `dbg.`, `scr.` and so on.

To get a list of all configuration variables just type `e` in the command line prompt. To limit output to a selected namespace, pass it with an ending dot to `e`. For example, `e file.` will display all variables defined inside "file" namespace.

To get help about `e` command type `e?`:

```
Usage: e[?] [var[=value]]
e?              show this help
e?asm.bytes     show description
e??             list config vars with description
e               list config vars
e-              reset config vars
e*              dump config vars in r commands
e!a             invert the boolean value of 'a' var
er [key]        set config key as readonly. no way back
ec [k] [color]  set color for given key (prompt, offset, ...)
e a             get value of var 'a'
e a=b           set var 'a' the 'b' value
env [k[=v]]     get/set environment variable
```

A simpler alternative to `e` command is accessible from the visual mode. Type `Ve` to enter it, use arrows (up, down, left, right) to navigate the configuration, and `q` to exit it. The start screen for the visual

cofiguration edit looks like this:

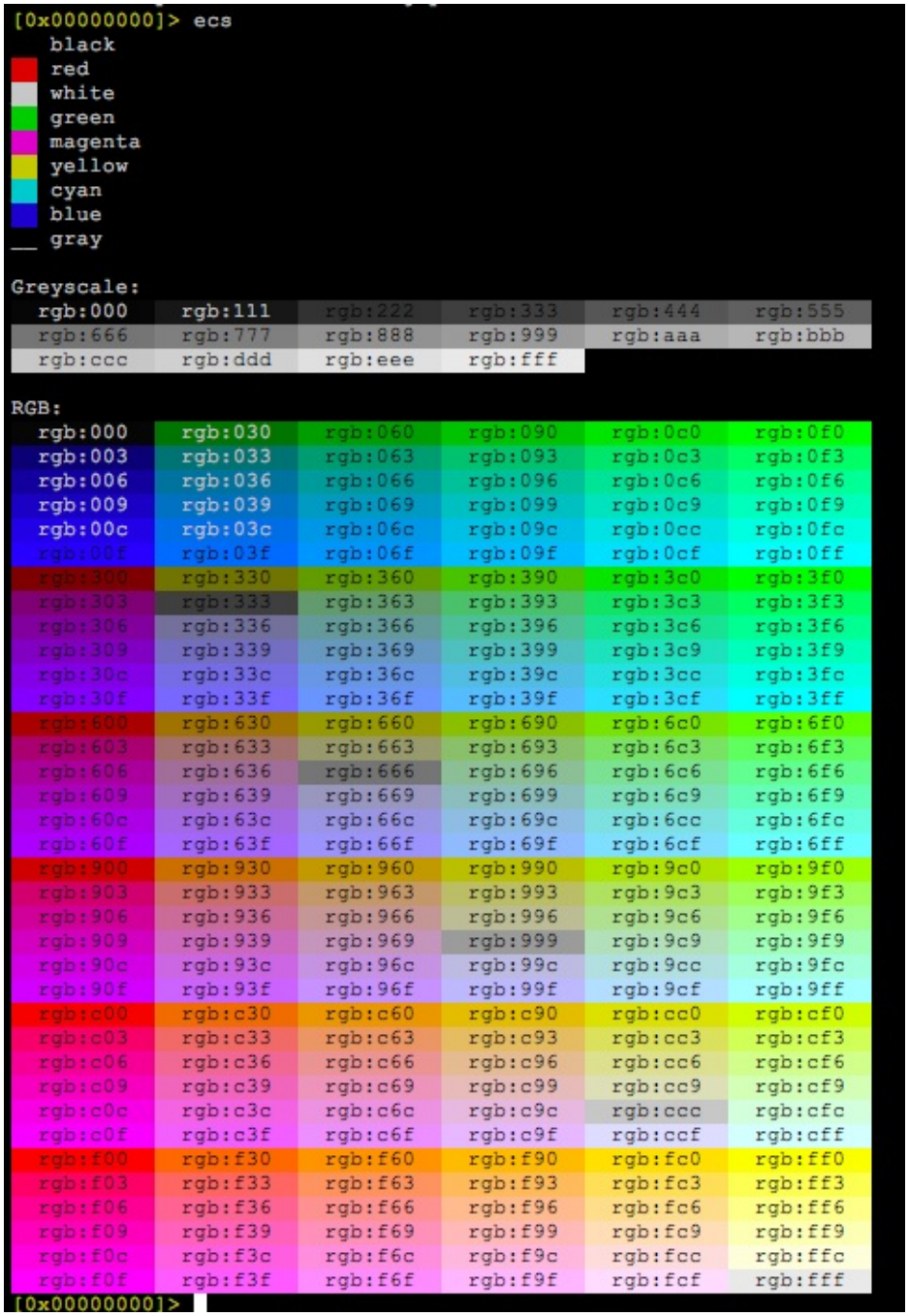
```
Eval spaces:

>  anal
   asm
   scr
   asm
   bin
   cfg
   diff
   dir
   dbg
   cmd
   fs
   hex
   http
   graph
   hud
   scr
   search
   io
```

Colors

Console access is wrapped in API that permits to show output of any command as ANSI, w32 console or HTML formats (more to come: ncurses, Pango etc.) This allows radare's core to run inside environments with limited displaying capabilities, like kernels or embedded devices. It is still possible to receive data from it in your favorite format. To enable colors support by default, add a corresponding configuration option to the .radare2 configuration file:

```
$ echo 'e scr.color=true' >> ~/.radare2rc
```

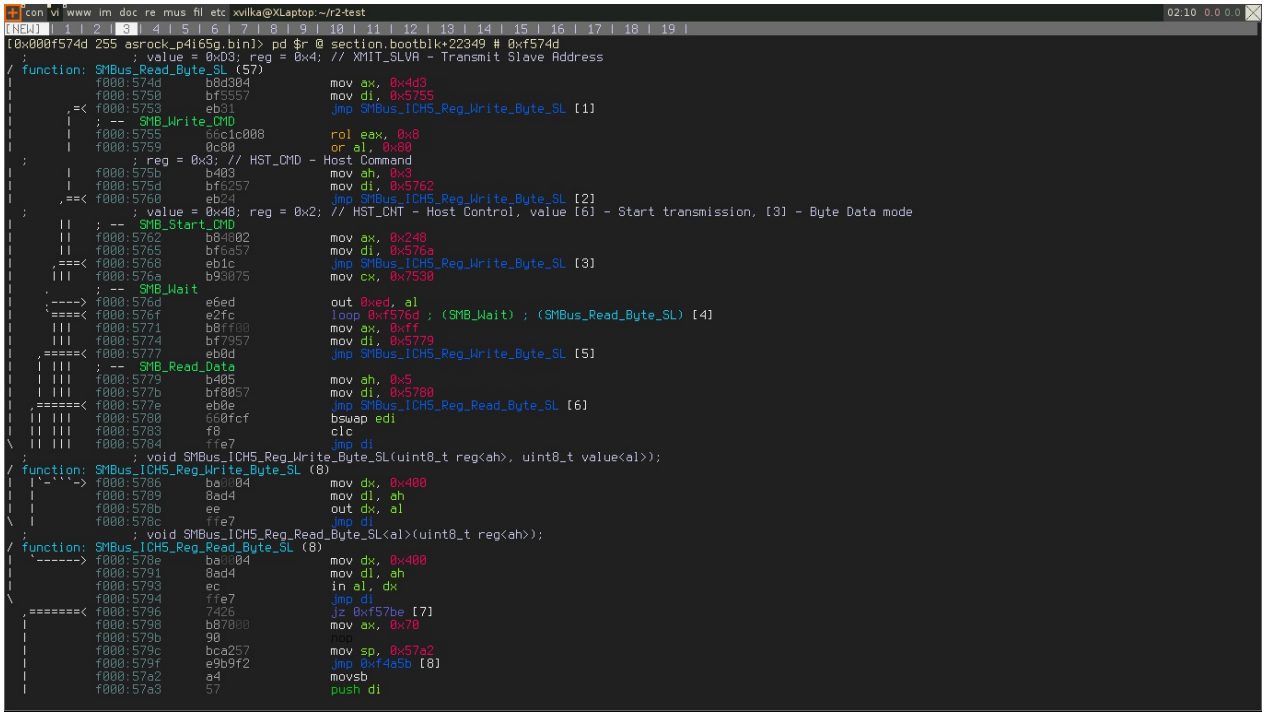
It is possible to configure color of almost any element of disassembly output. For *NIX terminals, r2 accepts color specification in RGB format. To change the console color palette use `ec` command. Type `ec` to get a list of all currently used colors. Type `ecs` to show a color palette to pick colors from:



xvilka theme

```
ec fname rgb:0cf
ec label rgb:0f3
ec math rgb:660
ec bin rgb:f90
ec call rgb:f00
ec jmp rgb:03f
ec cjmp rgb:33c
ec offset rgb:366
ec comment rgb:0cf
ec push rgb:0c0
ec pop rgb:0c0
ec cmp rgb:060
ec nop rgb:000
ec b0x00 rgb:444
```

```
ec b0x7f rgb:555
ec b0xff rgb:666
ec btext rgb:777
ec other rgb:bbb
ec num rgb:f03
ec reg rgb:6f0
ec fline rgb:fc0
ec flow rgb:0f0
```



Common Configuration Variables

Below is a list of the most frequently used configuration variables. You can get a complete list by issuing `e` command without arguments. For example, to see all variables defined in the "cfg" namespace, issue `e cfg.` (mind the ending dot). You can get help on any eval configuration variable by using `??e cfg.`

```
asm.arch
```

Defines target CPU architecture used for disassembling (`pd` , `pD` commands) and code analysis (`a` command). Supported values: `intel32` , `intel64` , `mips` , `arm16` , `arm` `java` , `csr` , `sparc` , `ppc` , `msil` and `m68k` . It is quite simple to add new architectures for disassembling and analyzing code. There is an interface for that. For x86, it is used to attach a number of third-party disassembler engines, including GNU binutils, Udis86 and a few of handmade ones.

```
asm.bits
```

Determines width in bits of registers for current architecture. Supported values: 8, 16, 32, 64. Note that not all target architectures support all combinations for asm.bits.

```
asm.syntax
```

Changes syntax flavor for disassembler between Intel and AT&T. At the moment, this setting affects Udis86 disassembler for Intel 32/Intel 64 targets only. Supported values are `intel` and `att` .

```
asm.pseudo
```

A boolean value to choose a string disassembly engine. "False" indicates a native one, defined by current architecture, "true" activates a pseudocode strings format; for example, it will show `eax=ebx` instead of a `mov eax, ebx` .

```
asm.os
```

Selects a target operating system of currently loaded binary. Usually OS is automatically detected by `rabin -rI` . Yet, `asm.os` can be used to switch to a different syscall table employed by another OS.

```
asm.flags
```

If defined to "true", disasembler view will have flags column.

```
asm.linescall
```

If set to "true", draw lines at the left of dissassemble output (`pd` , `pD` commands) to graphically represent control flow changes (jumps and calls) that are targeted inside current block. Also, see `asm.linesout` .

```
asm.linesout
```

When defined as "true", the disassembly view will also draw control flow lines that go ouside of the block.

```
asm.linestyle
```

A boolean value which changes the direction of control flow analysis. If set to "false", it is done from top to bottom of a block; otherwise, it goes from bottom to top. The "false" setting seems to be a better choice for improved readability, and is the default one.

```
asm.offset
```

Boolean value which controls visibility of offsets for individual disasembled instructions.

```
asm.trace
```

A boolean value that controls displaying of tracing information (sequence number and counter) at the left of each opcode. It is used to assist programs trace analysis.

```
asm.bytes
```

A boolean value used to show or hide displaying of raw bytes of instructions.

```
cfg.bigendian
```

Change endianness. "true" means big-endian, "false" is for little-endian.

```
file.analyze
```

A boolean value. If set, radare will run `.af* @@ sym.` and `.af* @ entrypoint` after resolving the symbols binary loading time. This way, radare will extract maximum of available information from the binary. Note that this configuration item does not affect type of analysis used when opening a project file. This option requires "file.id" and "file.flag" both to be true.

```
scr.color
```

This boolean variable enables or disables colorized screen output.

```
scr.seek
```

This variable accepts an expression, a pointer (eg. eip), etc. If set, radare will set seek position to its value on startup.

```
cfg.fortunes
```

Enables or disables "fortune" messages displayed at each radare start.

Basic Commands

Most command names in radare are derived from action names. They should be easy to remember, as they are short. Actually, all commands are single letters. Subcommands or related commands are specified using the second character of command name. For example, `/ foo` is a command to search plain string, while `/x 90 90` is used to look for hexadecimal pairs.

The general format for a valid command (as explained in the 'Command Format' chapter) looks like this:

```
[[.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
```

For example,

```
> 3s +1024    ; seeks three times 1024 from the current seek
```

If a command starts with `!`, the rest of the string is passed to currently loaded IO plugin (a debugger, for example). If no plugin can handle the command, posix_system() is called to pass the command to your shell. To make sure your command is directly passed to the shell, prefix it with two exclamation signs `!!`.

```
> !help       ; handled by the debugger or shell
> !!ls        ; run `ls` in the shell
```

The meaning of arguments (iter, addr, size) depends on the specific command. As a rule of thumb, most commands take a number as an argument to specify number of bytes to work with, instead of currently defined block size. Some commands accept math expressions, or strings.

```
> px 0x17     ; show 0x17 bytes in hexa at current seek
> s base+0x33 ; seeks to flag 'base' plus 0x33
> / lib       ; search for 'lib' string.
```

The `@` sign is used to specify a temporary offset location or seek position at which the command is executed, instead of current seek position. This is quite useful as you don't have to seek around all the time.

```
> p8 10 @ 0x4010  ; show 10 bytes at offset 0x4010
> f patata @ 0x10 ; set 'patata' flag at offset 0x10
```

Using `@@` you can execute a single command on a list of flags matching the glob. You can think of this as a foreach operation:

```
> s 0
> / lib             ; search 'lib' string
```

```
> p8 20 @@ hit0_*   ; show 20 hexpairs at each search hit
```

The `>` operation is used to redirect output of a command into a file (overwriting it if it already exists).

```
> pr > dump.bin   ; dump 'raw' bytes of current block to file named 'dump.bin'
> f  > flags.txt  ; dump flag list to 'flags.txt'
```

The `|` operation (pipe) is similar to what you are used to expect from it in a *NIX shell: us output of one command as input to another.

```
[0x4A13B8C0]> f | grep section | grep text
0x0805f3b0 512 section._text
0x080d24b0 512 section._text_end
```

You can pass several commands in a single line by separating them with semicolon `;`:

```
> px ; dr
```

Seeking

Current seek position is changed with `s` command. It accepts a math expression as argument. The expression can be composed of shift operations, basic math operations, or memory access operations.

```
[0x00000000]> s?
Usage: s[+-] [addr]
s                 print current address
s 0x320           seek to this address
s-                undo seek
s+                redo seek
s*                list undo seek history
s++               seek blocksize bytes forward
s--               seek blocksize bytes backward
s+ 512            seek 512 bytes forward
s- 512            seek 512 bytes backward
sg/sG             seek begin (sg) or end (sG) of section or file
s.hexoff          Seek honoring a base from core->offset
sa [[+-]a] [asz]  seek asz (or bsize) aligned to addr
sn/sp             seek next/prev scr.nkey
s/ DATA           search for next occurrence of 'DATA'
s/x 9091          search for next occurrence of \x90\x91
sb                seek aligned to bb start
so [num]          seek to N next opcode(s)
sf                seek to next function (f->addr+f->size)
sC str            seek to comment matching given string
sr pc             seek to register

> 3s++        ; 3 times block-seeking
> s 10+0x80   ; seek at 0x80+10
```

If you want to inspect the result of a math expression, you can evaluate it using the `?` command. Simply pass the expression as an argument. The result can be displayed in hexadecimal, decimal, octal or binary formats.

```
> ? 0x100+200
0x1C8 ; 456d ; 710o ; 1100 1000
```

In the visual mode you can press `u` (undo) or `U` (redo) inside the seek history to return back to previous or forward to the next location.

Block Size

The block size determines how many bytes Radare commands will process. All commands will work with this constraint. You can temporally change the block size by specifying a numeric argument to the print commands. For example `px 20`.

```
[0xB7F9D810]> b?
Usage: b[f] [arg]
b         display current block size
b+3       increase blocksize by 3
b-16      decrement blocksize by 16
b 33      set block size to 33
b eip+4   numeric argument can be an expression
bf foo    set block size to flag size
bm 1M     set max block size
```

The `b` command is used to change the block size:

```
[0x00000000]> b 0x100   ; block size = 0x100
[0x00000000]> b +16     ;  ... = 0x110
[0x00000000]> b -32     ;  ... = 0xf0
```

The `bf` command is used to change the block size to value specified by a flag. For example, in symbols, the block size of the flag represents the size of the function.

```
[0x00000000]> bf sym.main    ; block size = sizeof(sym.main)
[0x00000000]> pd @ sym.main  ; disassemble sym.main
...
```

You can combine two operations in a single one (`pdf`):

```
 [0x00000000]> pdf @ sym.main
```

Sections

Firmware images, bootloaders and binary files usually place various sections of a binary at different addresses in memory. To represent this behavior, radare offers the `S` command.

Here's the help message:

```
[0xB7EE8810]> S?
Usage: S[?-.*=adlr] [...]
S               ; list sections
S.              ; show current section name
S?              ; show this help message
S*              ; list sections (in radare commands)
S=              ; list sections (in nice ascii-art bars)
Sa[-] [arch] [bits] [[off]] ; Specify arch and bits for given section
Sd [file]       ; dump current section to a file (see dmd)
Sl [file]       ; load contents of file into current section (see dml)
Sr [name]       ; rename section on current seek
S [off] [vaddr] [sz] [vsz] [name] [rwx] ; add new section
S-[id|0xoff|*]  ; remove this section definition
```

You can specify a section in a single line:

```
# Add new section
S [off] [vaddr] [sz] [vsz] [name] [rwx]
```

For example:

```
[0x00404888]> S 0x00000100 0x00400000 0x0001ae08 0001ae08 test rwx
```

Displaying information about sections:

```
# List sections
[0x00404888]> S

[00] . 0x00000238 r-- va=0x00400238 sz=0x0000001c vsz=0000001c .interp
[01] . 0x00000254 r-- va=0x00400254 sz=0x00000020 vsz=00000020 .note.ABI_tag
[02] . 0x00000274 r-- va=0x00400274 sz=0x00000024 vsz=00000024 .note.gnu.build_id
[03] . 0x00000298 r-- va=0x00400298 sz=0x00000068 vsz=00000068 .gnu.hash
[04] . 0x00000300 r-- va=0x00400300 sz=0x00000c18 vsz=00000c18 .dynsym


# List sections (in nice ascii-art bars)
[0xB7EEA810]> S=

...
25  0x0001a600 |----------------------------#| 0x0001a608 --- .gnu_debuglink
26  0x0001a608 |----------------------------#| 0x0001a706 --- .shstrtab
27* 0x00000000 |################################| 0x0001ae08 rwx ehdr
```

```
=>  0x00004888 |-----^-----------------------| 0x00004988
```

The first three lines are sections and the last one (prefixed by `=>`) is the current seek location.

To remove a section definition, simply prefix the name of the section with `-`:

```
[0xB7EE8810]> S -.dynsym
```

Mapping Files

Radare IO system allows to map contents of files into the same IO space used to contain loaded binary. New contents can be placed at random offsets. This is useful to open multiple files in a single view or to 'emulate' a static environment similar to what you would have using a debugger where the program and all its libraries are loaded in memory and can be accessed.

Using the `S` (sections) command you can define base address for each library to be loaded.

Mapping files is done using the `o` (open) command. Let's read the help:

```
[0x00000000]> o?
Usage: o[com- ] [file] ([offset])
o                  list opened files
oc [file]          open core file, like relaunching r2
oo                 reopen current file (kill+fork in debugger)
oo+                reopen current file in read-write
o 4                priorize io on fd 4 (bring to front)
o-1                close file index 1
o /bin/ls          open /bin/ls file in read-only
o+/bin/ls          open /bin/ls file in read-write mode
o /bin/ls 0x4000   map file at 0x4000
on /bin/ls 0x4000  map raw file at 0x4000 (no r_bin involved)
om[?]              create, list, remove IO maps
```

To prepare a simple layout:

```
$ rabin2 -l /bin/ls
    [Linked libraries]
    libselinux.so.1
    librt.so.1
    libacl.so.1
    libc.so.6

    4 libraries
```

To map a file:

```
[0x00001190]> o /bin/zsh 0x499999
```

To list mapped files:

```
[0x00000000]> o
- 6 /bin/ls @ 0x0 ; r
- 10 /lib/ld-linux.so.2 @ 0x100000000 ; r
- 14 /bin/zsh @ 0x499999 ; r
```

To print hexadecimal values from /bin/zsh:

```
[0x00000000]> px @ 0x499999
```

To unmap files use `o-` command. Pass required file descriptor to it as an argument:

```
[0x00000000]> o-14
```

Print Modes

One of the key features of radare is displaying information in many formats. The goal is to offer a selection of displaying choices to best interpret binary data.

Binary data can be represented as integers, shorts, longs, floats, timestamps, hexpair strings, or more complex formats like C structures, disassembly listings, decompilations, be a result of an external processing...

Below is a list of available print modes listed by `p?`:

```
[0x08049AD0]> p?
Usage: p[=68abcdDfiImrstuxz] [arg|len]
p=[bep?] [blks]  show entropy/printable chars/chars bars
p2 [len]         8x8 2bpp-tiles
p6[de] [len]     base64 decode/encode
p8 [len]         8bit hexpair list of bytes
pa[ed] [hex asm] assemble (pa) or disasm (pad) or esil (pae) from hexpairs
p[bB] [len]      bitstream of N bytes
pc[p] [len]      output C (or python) format
p[dD][lf] [l]    disassemble N opcodes/bytes (see pd?)
pf[?|.nam] [fmt] print formatted data (pf.name, pf.name $<expr>)
p[iI][df] [len]  print N instructions/bytes (f=func) (see pi? and pdi)
pm [magic]       print libmagic data (pm? for more information)
pr [len]         print N raw bytes
p[kK] [len]      print key in randomart (K is for mosaic)
ps[pwz] [len]    print pascal/wide/zero-terminated strings
pt[dn?] [len]    print different timestamps
pu[w] [len]      print N url encoded bytes (w=wide)
pv[jh] [mode]      bar|json|histogram blocks (mode: e?search.in)
p[xX][owq] [len] hexdump of N bytes (o=octal, w=32bit, q=64bit)
pz [len]         print zoom view (see pz? for help)
pwd              display current working directory
```

Hexadecimal View

`px` gives a user-friendly output showing 16 pairs of numbers per row with offsets and raw representations:

```
[0x00404888]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00404888  31ed 4989 d15e 4889 e248 83e4 f050 5449  1.I..^H..H...PTI
0x00404898  c7c0 4024 4100 48c7 c1b0 2341 0048 c7c7  ..@$A.H...#A.H..
0x004048a8  d028 4000 e83f dcff fff4 6690 662e 0f1f  .(@..?....f.f...
```

Show Hexadecimal Words Dump (32 bits)

```
[0x00404888]> pxw
0x00404888  0x8949ed31 0x89485ed1 0xe48348e2 0x495450f0  1.I..^H..H...PTI
```

```
0x00404898  0x2440c0c7 0xc7480041 0x4123b0c1 0xc7c74800  ..@$A.H...#A.H..
0x004048a8  0x004028d0 0xffdc3fe8 0x9066f4ff 0x1f0f2e66  .(@..?....f.f...

[0x00404888]> e cfg.bigendian
false

[0x00404888]> e cfg.bigendian = true

[0x00404888]> pxw
0x00404888  0x31ed4989 0xd15e4889 0xe24883e4 0xf0505449  1.I..^H..H...PTI
0x00404898  0xc7c04024 0x410048c7 0xc1b02341 0x0048c7c7  ..@$A.H...#A.H..
0x004048a8  0xd0284000 0xe83fdcff 0xfff46690 0x662e0f1f  .(@..?....f.f...
```

8 bits Hexpair List of Bytes

```
[0x00404888]> p8 16
31ed4989d15e4889e24883e4f0505449
```

Show Hexadecimal Quad-words Dump (64 bits)

```
[0x08049A80]> pxq
0x00001390  0x65625f6b63617473  0x646e6962006e6967   stack_begin.bind
0x000013a0  0x616d6f6474786574  0x7469727766006e69   textdomain.fwrit
0x000013b0  0x6b636f6c6e755f65  0x6d63727473006465   e_unlocked.strcm
...
```

Date/Time Formats

Currently supported timestamp output modes are:

```
[0x00404888]> pt?
|Usage: pt[dn?]
| pt      print unix time (32 bit cfg.big_endian)
| ptd     print dos time (32 bit cfg.big_endian)
| ptn     print ntfs time (64 bit !cfg.big_endian)
| pt?     show help message
```

For example, you can 'view' the current buffer as timestamps in the ntfs time:

```
[0x08048000]> eval cfg.bigendian = false
[0x08048000]> pt 4
29:04:32948 23:12:36 +0000
[0x08048000]> eval cfg.bigendian = true
[0x08048000]> pt 4
20:05:13001 09:29:21 +0000
```

As you can see, the endianness affects the result. Once you have printed a timestamp, you can grep output, for example, by year value:

```
[0x08048000]> pt | grep 1974 | wc -l
15
[0x08048000]> pt | grep 2022
27:04:2022 16:15:43 +0000
```

The default date format can be configured using the `cfg.datefmt` variable. Formatting rules for it follow the well known strftime(3) format. An excerpt from the strftime(3) manpage:

```
%a  The abbreviated name of the day of the week according to the current locale.
%A  The full name of the day of the week according to the current locale.
%b  The abbreviated month name according to the current locale.
%B  The full month name according to the current locale.
%c  The preferred date and time representation for the current locale.
%C  The century number (year/100) as a 2-digit integer. (SU)
%d  The day of the month as a decimal number (range 01 to 31).
%D  Equivalent to %m/%d/%y.  (Yecch—for Americans only.  Americans should note that in ot
%e  Like %d, the day of the month as a decimal number, but a leading zero is replaced by
%E  Modifier: use alternative format, see below. (SU)
%F  Equivalent to %Y-%m-%d (the ISO 8601 date format). (C99)
%G  The ISO 8601 week-based year (see NOTES) with century as a decimal number.  The 4-dig
%g  Like %G, but without century, that is, with a 2-digit year (00-99). (TZ)
%h  Equivalent to %b.  (SU)
%H  The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I  The hour as a decimal number using a 12-hour clock (range 01 to 12).
%j  The day of the year as a decimal number (range 001 to 366).
%k  The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are prece
%l  The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are prece
%m  The month as a decimal number (range 01 to 12).
%M  The minute as a decimal number (range 00 to 59).
%n  A newline character. (SU)
%O  Modifier: use alternative format, see below. (SU)
%p  Either "AM" or "PM" according to the given time value, or the corresponding strings f
%P  Like %p but in lowercase: "am" or "pm" or a corresponding string for the current loca
%r  The time in a.m. or p.m. notation.  In the POSIX locale this is equivalent to %I:%M:%
%R  The time in 24-hour notation (%H:%M).  (SU) For a version including the seconds, see
%s  The number of seconds since the Epoch, 1970-01-01 00:00:00  +0000 (UTC). (TZ)
%S  The second as a decimal number (range 00 to 60).  (The range is up to 60 to allow for
%t  A tab character. (SU)
%T  The time in 24-hour notation (%H:%M:%S).  (SU)
%u  The day of the week as a decimal, range 1 to 7, Monday being 1.  See also %w.  (SU)
%U  The week number of the current year as a decimal number, range 00 to 53, starting wit
%V  The ISO 8601 week number (see NOTES) of the current year as a decimal number, range 0
%w  The day of the week as a decimal, range 0 to 6, Sunday being 0.  See also %u.
%W  The week number of the current year as a decimal number, range 00 to 53, starting wit
%x  The preferred date representation for the current locale without the time.
%X  The preferred time representation for the current locale without the date.
%y  The year as a decimal number without a century (range 00 to 99).
%Y  The year as a decimal number including the century.
%z  The +hhmm or -hhmm numeric timezone (that is, the hour and minute offset from UTC). (
%Z  The timezone name or abbreviation.
%+  The date and time in date(1) format. (TZ) (Not supported in glibc2.)
%%  A literal '%' character.
```

Basic Types

There are print modes available for all basic types. If you are interested in a more complex structure, just type : `pf?`. The list of the print modes for basic types (`pf?`):

```
Usage: pf[.key[.field[=value]]|[ val]]|[times][format] [arg0 arg1 ...]
Examples:
  pf 10xiz pointer length string
  pf {array_size}b @ array_base
  pf.             # list all formats
  pf.obj xxdz prev next size name
  pf.obj          # run stored format
  pf.obj.name     # show string inside object
  pf.obj.size=33  # set new size
 Format chars:
  e - temporally swap endian
  f - float value (4 bytes)
  c - char (signed byte)
  b - byte (unsigned)
  B - show 10 first bytes of buffer
  i - %i integer value (4 bytes)
  w - word (2 bytes unsigned short in hex)
  q - quadword (8 bytes)
  p - pointer reference (2, 4 or 8 bytes)
  d - 0x%08x hexadecimal value (4 bytes)
  D - disassemble one opcode
  x - 0x%08x hexadecimal value and flag (fd @ addr)
  z - \0 terminated string
  Z - \0 terminated wide string
  s - 32bit pointer to string (4 bytes)
  S - 64bit pointer to string (8 bytes)
  * - next char is pointer (honors asm.bits)
  + - toggle show flags for each offset
  : - skip 4 bytes
  . - skip 1 byte
```

Some examples are below:

```
[0x4A13B8C0]> pf i
0x00404888 = 837634441

[0x4A13B8C0]> pf
0x00404888 = 837634432.000000
```

High-level Languages Views

Valid print code formats for human-readable languages are:

```
pc     C
pcs    string
pcj    json
pcJ    javascript
```

```
pcp    python
pcw    words (4 byte)
pcd    dwords (8 byte)

[0xB7F8E810]> pc 32
#define _BUFFER_SIZE 32
unsigned char buffer[_BUFFER_SIZE] = {
0x89, 0xe0, 0xe8, 0x49, 0x02, 0x00, 0x00, 0x89, 0xc7, 0xe8, 0xe2, 0xff, 0xff, 0xff, 0x81,


[0x7fcd6a891630]> pcs
"\x48\x89\xe7\xe8\x68\x39\x00\x00\x49\x89\xc4\x8b\x05\xef\x16\x22\x00\x5a\x48\x8d\x24\xc4
```

Strings

Strings are probably one of the most important entrypoints when starting to reverse engineer a program, because they usually reference information about functions' actions (asserts, debug or info messages...) Therefore radare supports various string formats:

```
[0x00000000]> ps?
|Usage: ps[zpw] [N]Print String
| ps   print string
| psi  print string inside curseek
| psb  print strings in current block
| psx  show string with scaped chars
| psz  print zero terminated string
| psp  print pascal string
| psu  print utf16 unicode (json)
| psw  print wide string
| psj  print string in JSON format
```

Most strings are zero-terminated. Here is an example by using the debugger to continue the execution of a program until it executes the 'open' syscall. When we recover the control over the process, we get the arguments passed to the syscall, pointed by %ebx. In the case of the 'open' call, it is a zero terminated string which we can inspect using `psz`.

```
[0x4A13B8C0]> dcs open
0x4a14fc24 syscall(5) open ( 0x4a151c91 0x00000000 0x00000000 ) = 0xfffffffda
[0x4A13B8C0]> dr
  eax  0xfffffffda    esi  0xffffffff    eip    0x4a14fc24
  ebx  0x4a151c91    edi  0x4a151be1    oeax   0x00000005
  ecx  0x00000000    esp  0xbfbedb1c    eflags 0x200246
  edx  0x00000000    ebp  0xbfbedbb0    cPaZstIdor0 (PZI)
[0x4A13B8C0]>
[0x4A13B8C0]> psz @ 0x4a151c91
/etc/ld.so.cache
```

Print Memory Contents

It is also possible to print various packed data types using the `pf` command:

```
[0xB7F08810]> pf xxS @ rsp
0x7fff0d29da30 = 0x00000001
0x7fff0d29da34 = 0x00000000
0x7fff0d29da38 = 0x7fff0d29da38 -> 0x0d29f7ee /bin/ls
```

This can be used to look at the arguments passed to a function. To achive this, simply pass a 'format memory string' as an argument to `pf`, and temporally change current seek position / offset using `@`. It is also possible to define arrays of structures with `pf`. To do this, prefix the format string with a numeric value. You can also define a name for each field of the structure by appending them as a space-separated arguments list.

```
[0x4A13B8C0]> pf 2*xw pointer type @ esp
0x00404888 [0] {
   pointer :
(*0xffffffff8949ed31)      type : 0x00404888 = 0x8949ed31
   0x00404890 = 0x48e2
}
0x00404892 [1] {
(*0x50f0e483)    pointer : 0x00404892 = 0x50f0e483
     type : 0x0040489a = 0x2440
}
```

A practical example for using `pf` on a binary of a GStreamer plugin:

```
$ radare ~/.gstreamer-0.10/plugins/libgstflummms.so
[0x000028A0]> seek sym.gst_plugin_desc
[0x000185E0]> pf iissxssssss major minor name desc _init version \
 license source package origin
     major : 0x000185e0 = 0
     minor : 0x000185e4 = 10
      name : 0x000185e8 = 0x000185e8 flumms
      desc : 0x000185ec = 0x000185ec Fluendo MMS source
     _init : 0x000185f0 = 0x00002940
   version : 0x000185f4 = 0x000185f4 0.10.15.1
   license : 0x000185f8 = 0x000185f8 unknown
    source : 0x000185fc = 0x000185fc gst-fluendo-mms
   package : 0x00018600 = 0x00018600 Fluendo MMS source
    origin : 0x00018604 = 0x00018604 http://www.fluendo.com
```

Disassembly

The `pd` command is used to disassemble code. It accepts a numeric value to specify how many instructions should be disassembled. The `pD` command is similar but instead of a number of instructions, it decompiles a given number of bytes.

```
 d : disassembly N opcodes   count of opcodes
 D : asm.arch disassembler   bsize bytes

 [0x00404888]> pd 1
```

```
            ;-- entry0:
            0x00404888    31ed         xor ebp, ebp
```

Selecting Target Architecture

The architecture flavor for disassembler is defined by the `asm.arch` eval variable. You can use `e asm.arch = ?` to list all available architectures.

```
[0xB7F08810]> e asm.arch = ?

_d  16         8051        PD      8051 Intel CPU
_d  16 32      arc         GPL3    Argonaut RISC Core
ad  16 32 64   arm         GPL3    Acorn RISC Machine CPU
_d  16 32 64   arm.cs      BSD     Capstone ARM disassembler
_d  16 32      arm.winedbg LGPL2   WineDBG's ARM disassembler
_d  16 32      avr         GPL     AVR Atmel
ad  32         bf          LGPL3   Brainfuck
_d  16         cr16        LGPL3   cr16 disassembly plugin
_d  16         csr         PD      Cambridge Silicon Radio (CSR)
ad  32 64      dalvik      LGPL3   AndroidVM Dalvik
ad  16         dcpu16      PD      Mojang's DCPU-16
_d  32 64      ebc         LGPL3   EFI Bytecode
_d  8          gb          LGPL3   GameBoy(TM) (z80-like)
_d  16         h8300       LGPL3   H8/300 disassembly plugin
_d  8          i8080       BSD     Intel 8080 CPU
ad  32         java        Apache  Java bytecode
_d  16 32      m68k        BSD     Motorola 68000
_d  32         malbolge    LGPL3   Malbolge Ternary VM
ad  32 64      mips        GPL3    MIPS CPU
_d  16 32 64   mips.cs     BSD     Capstone MIPS disassembler
_d  16 32 64   msil        PD      .NET Microsoft Intermediate Language
_d  32         nios2       GPL3    NIOS II Embedded Processor
_d  32 64      ppc         GPL3    PowerPC
_d  32 64      ppc.cs      BSD     Capstone PowerPC disassembler
ad             rar         LGPL3   RAR VM
_d  32         sh          GPL3    SuperH-4 CPU
_d  32 64      sparc       GPL3    Scalable Processor Architecture
_d  32         tms320      LGPLv3  TMS320 DSP family
_d  32         ws          LGPL3   Whitespace esotheric VM
_d  16 32 64   x86         BSD     udis86 x86-16,32,64
_d  16 32 64   x86.cs      BSD     Capstone X86 disassembler
a_  32 64      x86.nz      LGPL3   x86 handmade assembler
ad  32         x86.olly    GPL2    OllyDBG X86 disassembler
ad  8          z80         NC-GPL2 Zilog Z80
```

Configuring the Disassembler

There are multiple options which can be used to configure the output of disassembler. All these options are described in `e? asm.`

```
              asm.os: Select operating system (kernel) (linux, darwin, w32,..)
           asm.bytes: Display the bytes of each instruction
      asm.cmtflgrefs: Show comment flags associated to branch referece
```

```
          asm.cmtright: Show comments at right of disassembly if they fit in screen
          asm.comments: Show comments in disassembly view
            asm.decode: Use code analysis as a disassembler
             asm.dwarf: Show dwarf comment at disassembly
              asm.esil: Show ESIL instead of mnemonic
            asm.filter: Replace numbers in disassembly using flags containing a dot in the
             asm.flags: Show flags
            asm.lbytes: Align disasm bytes to left
             asm.lines: If enabled show ascii-art lines at disassembly
         asm.linescall: Enable call lines
          asm.linesout: If enabled show out of block lines
        asm.linesright: If enabled show lines before opcode instead of offset
        asm.linesstyle: If enabled iterate the jump list backwards
         asm.lineswide: If enabled put an space between lines
            asm.middle: Allow disassembling jumps in the middle of an instruction
            asm.offset: Show offsets at disassembly
            asm.pseudo: Enable pseudo syntax
              asm.size: Show size of opcodes in disassembly (pd)
          asm.stackptr: Show stack pointer at disassembly
            asm.cycles: Show cpu-cycles taken by instruction at disassembly
              asm.tabs: Use tabs in disassembly
             asm.trace: Show execution traces for each opcode
             asm.ucase: Use uppercase syntax at disassembly
            asm.varsub: Substitute variables in disassembly
              asm.arch: Set the arch to be usedd by asm
            asm.parser: Set the asm parser to use
            asm.segoff: Show segmented address in prompt (x86-16)
               asm.cpu: Set the kind of asm.arch cpu
           asm.profile: configure disassembler (default, simple, gas, smart, debug, full)
             asm.xrefs: Show xrefs in disassembly
         asm.functions: Show functions in disassembly
            asm.syntax: Select assembly syntax
            asm.nbytes: Number of bytes for each opcode at disassembly
         asm.bytespace: Separate hex bytes with a whitespace
              asm.bits: Word size in bits at assembler
        asm.lineswidth: Number of columns for program flow arrows
```

Disassembly Syntax

The `asm.syntax` variable is used to change flavor of assembly syntax used by a disassembler engine. To switch between Intel and AT&T representations:

```
e asm.syntax = intel
e asm.syntax = att
```

You can also check `asm.pseudo`, which is an experimental pseudocode view, and `asm.esil` which outputs ESIL ('Evaluable Strings Intermedate Language'). ESIL's goal is to have a human-readable representation of every opcode semantics. Such representations can be evaluated (interpreted) to emulate effects of individual instructions.

Flags

Flags are similar to bookmarks. They represent a certain offset in a file. Flags can be grouped in 'flag spaces'. A flag space is something like a namespace for flags. They are used to group flags of similar characteristic or type. Examples for flag spaces: sections, registers, symbols.

To create a flag type:

```
[0x4A13B8C0]> f flag_name @ offset
```

You can remove a flag by appending the `-` character to command. Most of commands accept `-` as argument-prefix as an indication to delete something.

```
[0x4A13B8C0]> f- flag_name
```

To switch between or create new flagspaces use the `fs` command:

```
# List flag spaces
[0x4A13B8C0]> fs

00   symbols
01   imports
02   sections
03   strings
04   regs
05   maps

[0x4A13B8C0]> fs symbols ; select only flags in symbols flagspace
[0x4A13B8C0]> f          ; list only flags in symbols flagspace
[0x4A13B8C0]> fs *       ; select all flagspaces
[0x4A13B8C0]> f myflag   ; create a new flag called 'myflag'
[0x4A13B8C0]> f- myflag  ; delete the flag called 'myflag'
```

You can rename flags with `fr`.

Writing Data

Radare can manipulate with a loaded binary file in many ways. You can resize the file, move and copy/paste bytes, insert new bytes (shifting data to the end of the block or file), or simply overwrite bytes. New data may come from another file contents, be a widestring, or even be represented as inline assembler statement.

To resize, use the `r` command. It accepts a numeric argument. A positive value sets new size to a file. A negative one will strip N bytes from the current seek, making the file smaller.

```
r 1024      ; resize the file to 1024 bytes
r -10 @ 33  ; strip 10 bytes at offset 33
```

To write bytes, use `w` command. It accepts multiple input formats, like inline assembly, endian-friendly dwords, files, hexpair files, wide strings:

```
[0x00404888]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| w foobar     write string 'foobar'
| wh r2        whereis/which shell command
| wr 10        write 10 random bytes
| ww foobar    write wide string 'f\x00o\x00o\x00b\x00a\x00r\x00'
| wa push ebp  write opcode, separated by ';' (use '"' around the command)
| waf file     assemble file and write bytes
| wA r 0       alter/modify opcode at current seek (see wA?)
| wb 010203    fill current block with cyclic hexpairs
| wc[ir*?]     write cache undo/commit/reset/list (io.cache)
| wx 9090      write two intel nops
| wv eip+34    write 32-64 bit value
| wo? hex      write in block with operation. 'wo?' fmi
| wm f0ff      set binary mask hexpair to be used as cyclic write mask
| ws pstring   write 1 byte for length and then the string
| wf -|file    write contents of file at current offset
| wF -|file    write contents of hexpairs file here
| wp -|file    apply radare patch file. See wp? fmi
| wt file [sz] write to file (from current seek, blocksize or sz bytes)
```

Some examples:

```
[0x00000000]> wx 123456 @ 0x8048300
[0x00000000]> wv 0x8048123 @ 0x8049100
[0x00000000]> wa jmp 0x8048320
```

Write Over

The `wo` command (write over) has many subcommands. It is applied to the current block. Supported operations: XOR, ADD, SUB...

```
[0x4A13B8C0]> wo?
|Usage: wo[asmdxoArl24] [hexpairs] @ addr[:bsize]
|Example:
|  wox 0x90   ; xor cur block with 0x90
|  wox 90     ; xor cur block with 0x90
|  wox 0x0203 ; xor cur block with 0203
|  woa 02 03  ; add [0203][0203][...] to curblk
|  woe 02 03
|Supported operations:
|  wow  ==  write looped value (alias for 'wb')
|  woa  +=  addition
|  wos  -=  substraction
|  wom  *=  multiply
|  wod  /=  divide
|  wox  ^=  xor
|  woo  |=  or
|  woA  &=  and
|  woR  random bytes (alias for 'wr $b'
|  wor  >>= shift right
|  wol  <<= shift left
|  wo2  2=  2 byte endian swap
|  wo4  4=  4 byte endian swap
```

It is possible to implement cipher-algorithms using radare core primitives and `wo`. A sample session performing xor(90) + add(01, 02):

```
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  4889 e7e8 6839 0000 4989 c48b 05ef 1622  H...h9..I......"
0x7fcd6a891640  005a 488d 24c4 29c2 5248 89d6 4989 e548  .ZH.$.).RH..I..H
0x7fcd6a891650  83e4 f048 8b3d 061a 2200 498d 4cd5 1049  ...H.=..".I.L..I
0x7fcd6a891660  8d55 0831 ede8 06e2 0000 488d 15cf e600  .U.1......H.....


[0x7fcd6a891630]> wox 90
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  d819 7778 d919 541b 90ca d81d c2d8 1946  ..wx..T........F
0x7fcd6a891640  1374 60d8 b290 d91d 1dc5 98a1 9090 d81d  .t`.............
0x7fcd6a891650  90dc 197c 9f8f 1490 d81d 95d9 9f8f 1490  ...|............
0x7fcd6a891660  13d7 9491 9f8f 1490 13ff 9491 9f8f 1490  ................


[0x7fcd6a891630]> woa 01 02
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  d91b 787a 91cc d91f 1476 61da 1ec7 99a3  ..xz.....va.....
0x7fcd6a891640  91de 1a7e d91f 96db 14d9 9593 1401 9593  ...~............
0x7fcd6a891650  c4da 1a6d e89a d959 9192 9159 1cb1 d959  ...m...Y...Y...Y
0x7fcd6a891660  9192 79cb 81da 1652 81da 1456 a252 7c77  ..y....R...V.R|w
```

Zoom

The zoom is a print mode that allows you to get a global view of the whole file or a memory map on a single screen. In this mode, each byte represents `file_size/block_size` bytes of the file. Use the `pO` (zoom out print mode) to enter this mode, or just toggle `z` in the visual mode to zoom-out/zoom-in.

The cursor can be used to scroll faster through the zoom out view. Pressing `z` again will zoom-in where at new cursor position.

```
[0x004048c5]> pz?
|Usage: pz [len] print zoomed blocks (filesize/N)
| e zoom.maxsz  max size of block
| e zoom.from   start address
| e zoom.to     end address
| e zoom.byte   specify how to calculate each byte
| pzp           number of printable chars
| pzf           count of flags in block
| pzs           strings in range
| pz0           number of bytes with value '0'
| pzF           number of bytes with value 0xFF
| pze           calculate entropy and expand to 0-255 range
| pzh           head (first byte value); This is the default mode
```

Let's see some examples:

```
[0x08049790]> pz // or default pzh

0x00000000  7f00 0000 e200 0000 146e 6f74 0300 0000  .........not....
0x00000010  0000 0000 0068 2102 00ff 2024 e8f0 007a  .....h!... $...z
0x00000020  8c00 18c2 ffff 0080 4421 41c4 1500 5dff  ........D!A...].
0x00000030  ff10 0018 0fc8 031a 000c 8484 e970 8648  .............p.H
0x00000040  d68b 3148 348b 03a0 8b0f c200 5d25 7074  ..1H4.......]%pt
0x00000050  7500 00e1 ffe8 58fe 4dc4 00e0 dbc8 b885  u.....X.M.......


[0x08049790]> e zoom.byte=p
[0x08049790]> pO // or pzp

0x00000000  2f47 0609 070a 0917 1e9e a4bd 2a1b 2c27  /G..........*.,'
0x00000010  322d 5671 8788 8182 5679 7568 82a2 7d89  2-Vq....Vyuh..}.
0x00000020  8173 7f7b 727a 9588 a07b 5c7d 8daf 836d  .s.{rz...{\}...m
0x00000030  b167 6192 a67d 8aa2 6246 856e 8c9b 999f  .ga..}..bF.n....
0x00000040  a774 96c3 b1a4 6c8e a07c 6a8f 8983 6a62  .t....l..|j...jb
0x00000050  7d66 625f 7ea4 7ea6 b4b6 8b57 a19f 71a2  }fb_~.~....W..q.

[0x08049790]> eval zoom.byte = flags
[0x08049790]> pO // or pzf

0x00406e65  48d0 80f9 360f 8745 ffff ffeb ae66 0f1f  H...6..E.....f..
0x00406e75  4400 0083 f801 0f85 3fff ffff 410f b600  D.......?...A...
0x00406e85  3c78 0f87 6301 0000 0fb6 c8ff 24cd 0026  <x..c.......$..&
0x00406e95  4100 660f 1f84 0000 0000 0084 c074 043c  A.f..........t.<
```

```
0x00406ea5  3a75 18b8 0500 0000 83f8 060f 95c0 e9cd  :u..............
0x00406eb5  feff ff0f 1f84 0000 0000 0041 8801 4983  ...........A..I.
0x00406ec5  c001 4983 c201 4983 c101 e9ec feff ff0f  ..I...I.........

[0x08049790]> e zoom.byte=F
[0x08049790]> pO // or pzF

0x00000000  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000010  0000 2b5c 5757 3a14 331f 1b23 0315 1d18  ..+\WW:.3..#....
0x00000020  222a 2330 2b31 2e2a 1714 200d 1512 383d  "*#0+1.*.. ...8=
0x00000030  1e1a 181b 0a10 1a21 2a36 281e 1d1c 0e11  .......!*6(.....
0x00000040  1b2a 2f22 2229 181e 231e 181c 1913 262b  .*/"")..#.....&+
0x00000050  2b30 4741 422f 382a 1e22 0f17 0f10 3913  +0GAB/8*."....9.
```

You can limit zooming to a range of bytes instead of the whole bytespace. Change `zoom.from` and `zoom.to` eval variables:

```
[0x465D8810]> e zoom.
zoom.byte = f
zoom.from = 0
zoom.maxsz = 512
zoom.to = 118368???
```

Yank/Paste

You can yank/paste bytes in visual mode using the `y` and `Y` key bindings which are aliases for `y` and `yy` commands of command-line interface. These commands operate on an internal buffer which stores copies of bytes taken starting from the current seek position. You can write this buffer back to different seek position using `yy` command:

```
[0x00000000]> y?
|Usage: y[ptxy] [len] [[@]addr]
| y               show yank buffer information (srcoff len bytes)
| y 16            copy 16 bytes into clipboard
| y 16 0x200      copy 16 bytes into clipboard from 0x200
| y 16 @ 0x200    copy 16 bytes into clipboard from 0x200
| yp              print contents of clipboard
| yx              print contents of clipboard in hexadecimal
| yt 64 0x200     copy 64 bytes from current seek to 0x200
| yf 64 0x200 file copy 64 bytes from 0x200 from file (opens w/ io), use -1 for all bytes
| yfa file copy   copy all bytes from from file (opens w/ io)
| yy 0x3344       paste clipboard
```

Sample session:

```
[0x00000000]> s 0x100    ; seek at 0x100
[0x00000100]> y 100      ; yanks 100 bytes from here
[0x00000200]> s 0x200    ; seek 0x200
[0x00000200]> yy         ; pastes 100 bytes
```

You can perform a yank and paste in a single line by just using the `yt` command (yank-to). The syntax is as follows:

```
[0x4A13B8C0]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  0123456789AB
0x4A13B8C0, 89e0 e839 0700 0089 c7e8 e2ff ...9........
0x4A13B8CC, ffff 81c3 eea6 0100 8b83 08ff ............
0x4A13B8D8, ffff 5a8d 2484 29c2           ..Z.$.).

[0x4A13B8C0]> yt 8 0x4A13B8CC @ 0x4A13B8C0

[0x4A13B8C0]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  0123456789AB
0x4A13B8C0, 89e0 e839 0700 0089 c7e8 e2ff ...9........
0x4A13B8CC, 89e0 e839 0700 0089 8b83 08ff ...9........
0x4A13B8D8, ffff 5a8d 2484 29c2           ..Z.$.).
```

Comparing Bytes

`c` (short for "compare") allows you to compare arrays of bytes from different sources. The command accepts input in a number of formats, and then compares it against values found at current seek position.

```
[0x00404888]> c?
|Usage: c[?dfx] [argument]
| c  [string]    Compares a plain with escaped chars string
| cc [at] [(at)] Compares in two hexdump columns of block size
| c4 [value]     Compare a doubleword from a math expression
| c8 [value]     Compare a quadword from a math expression
| cx [hexpair]   Compare hexpair string
| cX [addr]      Like 'cc' but using hexdiff output
| cf [file]      Compare contents of file at current seek
| cg[o] [file]   Graphdiff current file and [file]
| cu [addr] @at  Compare memory hexdumps of $$ and dst in unified diff
| cw[us?] [...]  Compare memory watchers
| cat  [file]    Show contents of file (see pwd, ls)
| cl|cls|clear   Clear screen, (clear0 to goto 0, 0 only)
```

To compare memory contents at current seek position against given string of values, use `cx`:

```
[0x08048000]> p8 4
7f 45 4c 46

[0x08048000]> cx 7f 45 90 46
Compare 3/4 equal bytes
0x00000002 (byte=03)   90 ' '  ->  4c 'L'
[0x08048000]>
```

Another subcommand of `c` command is `cc` which stands for "compare code". To compare a byte sequence with a sequence in memory:

```
[0x4A13B8C0]> cc 0x39e8e089 @ 0x4A13B8C0
```

To compare contents of two functions specified by their names:

```
[0x08049A80]> cc sym.main2 @ sym.main
```

`c8` compares a quadword from the current seek (in the example below, 0x00000000) against a math expression:

```
[0x00000000]> c8 4

Compare 1/8 equal bytes (0%)
0x00000000 (byte=01)   7f ' '  ->  04 ' '
```

```
0x00000001 (byte=02)   45 'E'  ->  00 ' '
0x00000002 (byte=03)   4c 'L'  ->  00 ' '
```

The number parameter can of course also be a math expressions which uses flag names etc:

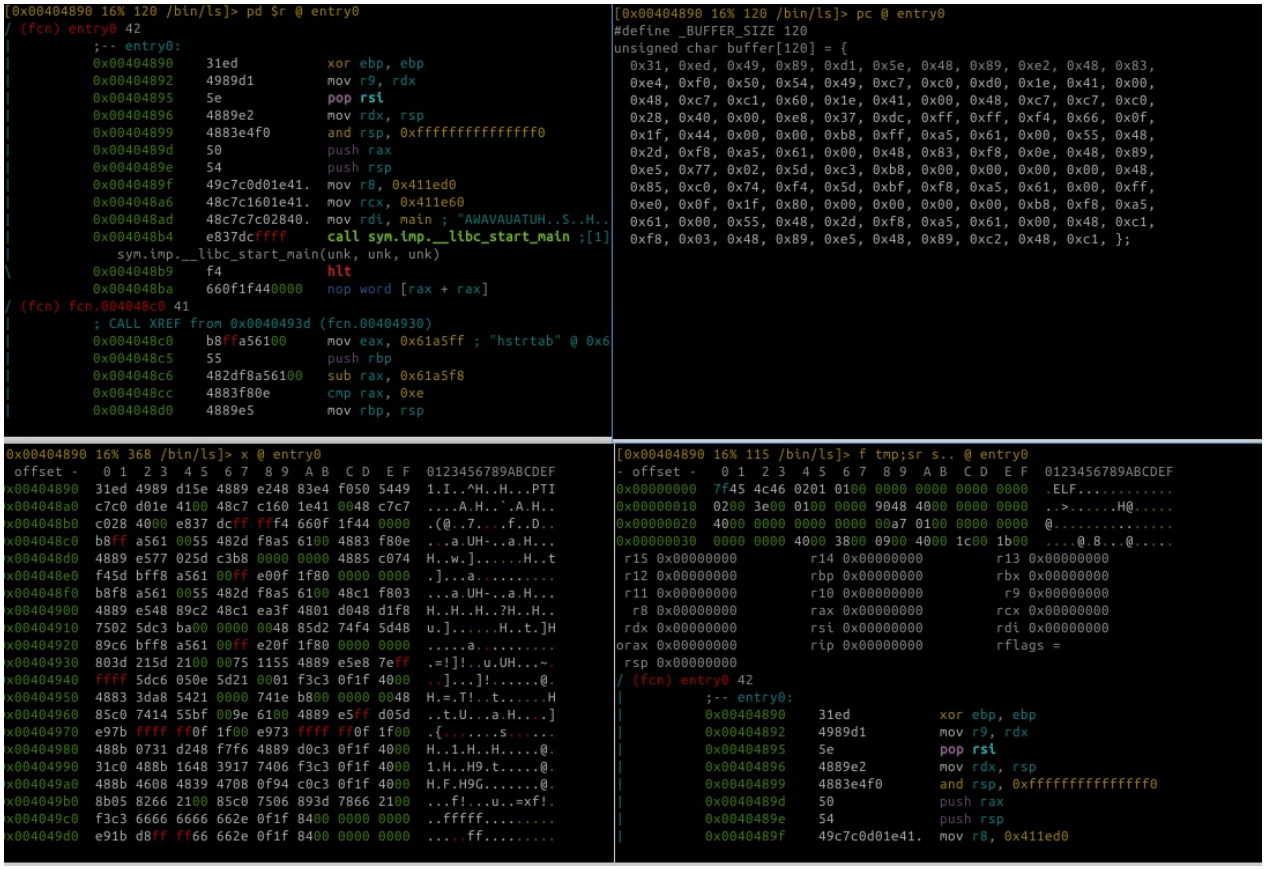
```
[0x00000000]> cx 7f469046

Compare 2/4 equal bytes
0x00000001 (byte=02)   45 'E'  ->  46 'F'
0x00000002 (byte=03)   4c 'L'  ->  90 ' '
```

You can use compare command to find differences between a current block and a file previously dumped to a disk:

```
r2 /bin/true
[0x08049A80]> s 0
[0x08048000]> cf /bin/true
Compare 512/512 equal bytes
```

Visual Mode

The visual mode is a more user-friendly interface alternative to radare2's command-line prompt. It uses HJKL or arrow keys to move around data and code, has a cursor mode for selecting bytes, and offers numerous key bindings to simplify debugger use. To enter visual mode, use `V` command. To exit from it back to command line, press `q`.



Getting Help

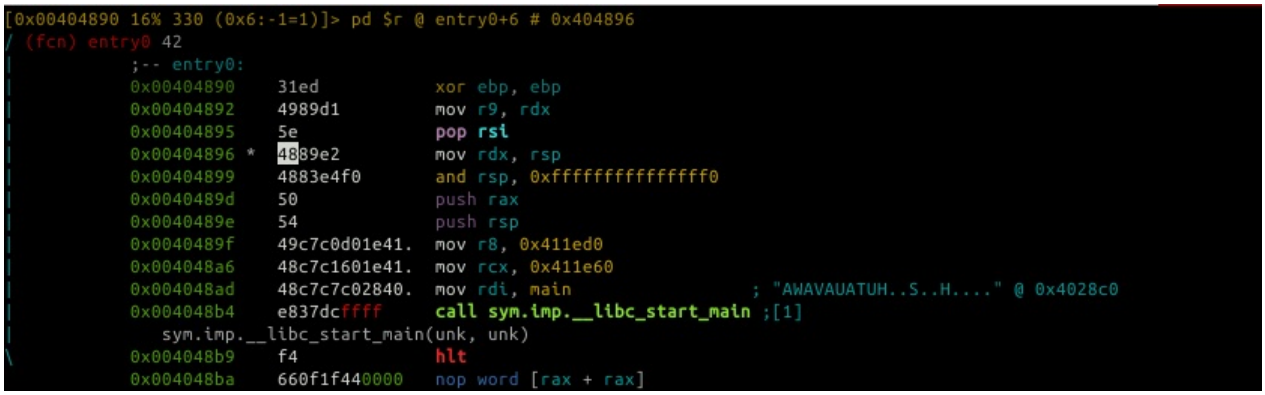
To see help on all key bindings defined for visual mode, press `?`:

```
Visual mode help:
 ?         show this help or manpage in cursor mode
 &         rotate asm.bits between supported 8, 16, 32, 64
 %         in cursor mode finds matching pair, otherwise toggle autoblocksz
 @         set cmd.vprompt to run commands before the visual prompt
 !         enter into the visual panels mode
 _         enter the hud
 =         set cmd.vprompt (top row)
 |         set cmd.cprompt (right column)
 .         seek to program counter
 /         in cursor mode search in current block
 :cmd      run radare command
 ;[-]cmt   add/remove comment
 /*+-[]    change block size, [] = resize hex.cols
 >||<      seek aligned to block size
```

```
  a/A      (a)ssemble code, visual (A)ssembler
  b        toggle breakpoint
  c/C      toggle (c)ursor and (C)olors
  d[f?]    define function, data, code, ..
  D        enter visual diff mode (set diff.from/to)
  e        edit eval configuration variables
  f/F      set/unset or browse flags. f- to unset, F to browse, ..
  gG       go seek to begin and end of file (0-$s)
  hjkl     move around (or HJKL) (left-down-up-right)
  i        insert hex or string (in hexdump) use tab to toggle
  mK/'K    mark/go to Key (any key)
  M        walk the mounted filesystems
  n/N      seek next/prev function/flag/hit (scr.nkey)
  o        go/seek to given offset
  O        toggle asm.esil
  p/P      rotate print modes (hex, disasm, debug, words, buf)
  q        back to radare shell
  r        browse anal info and comments
  R        randomize color palette (ecr)
  sS       step / step over
  T        enter textlog chat console (TT)
  uU       undo/redo seek
  v        visual code analysis menu
  V        (V)iew graph using cmd.graph (agv?)
  wW       seek cursor to next/prev word
  xX       show xrefs/refs of current function from/to data/code
  yY       copy and paste selection
  z        toggle zoom mode
  Enter    follow address of jump/call
Function Keys: (See 'e key.'), defaults to:
  F2       toggle breakpoint
  F7       single step
  F8       step over
  F9       continue
```

Visual Cursor Mode

Pressing lowercase `c` toggles the cursor mode. When this mode is active, currently selected byte (or byte range) is highlighted by having a highlighted background.



The cursor is used to select a range of bytes or simply to point to a byte. You can use the latter to create a named flag. Seek to required position, then press `f` and enter a name for a flag. If you select a range of bytes (with HJKL and SHIFT key pressed), and file write mode has been enabled with `-w` radare2 option, you can press `i` and then enter a byte array to overwrite selected range with new values, used as circular buffer. For example:

```
<select 10 bytes in visual mode using SHIFT+HJKL>
<press 'i' and then enter '12 34'>
```

10 bytes you have selected will be changed to "12 34" repeated in a circular manner: 12 34 12 34 12 34 12 34 12 34. A byte range selection can be used together with the `d` key to set associated data type: a string, code or, word, or to perform other actions as indicated in the menu presented on the key press:

```
B    set as short word (2 bytes)
c    set as code
C    define flag color (fc)
d    set as data
e    end of function
f    analyze function
F    format
j    merge down (join this and next functions)
k    merge up (join this and previous function)
h    highlight word
m    manpage for current call
q    quit/cancel operation
r    rename function
R    find references /r
s    set string
S    set strings in current block
u    undefine metadata here
w    set as 32bit word
W    set as 64bit word
q    quit this menu
```

This can be used to enhance disassembly view, to add metadata or to set code boundaries for cases when instructions are intermixed with data. In cursor mode, you can set a block size by simply moving cursor to position you want and then pressing `_` to invoke HUD menu. Then change block size.

Insert in Visual Mode

Remember that, to be able to actually edit files loaded in radare2, you have to start it with `-w` option. Otherwise a file is opened in read-only mode.

The cursor mode allows you to manipulate with data at nibble-level (4 bits chunks), like it is done in most of hexadecimal editors. Press `TAB` to switch between hexadecimal and ASCII columns of the hexadecimal dump view. After you press `i` key, you are prompted for an hexpair string. If `a` is pressed, enter an assembler expression, which will be translated to machine code and then inseters at the chosen offset.

Visual Xrefs

radare2 implements many user-friendly features, helping to walk through target assembly code, in visual mode. One of them is accessible by pressing the `x` key. A popup menu appears with a list of cross-references (a.k.a xref), either data or code, defined for current seek position. After pressing a number key from this menu list, you will be transfered to a correspoding offset for that xref in the file. Suppose you have moved your cursor to a position in code where the following list of xrefs is defined:

```
|  .....-> ; CODE (CALL) XREF from 0x00402b98 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402ba0 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402ba9 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402bd5 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402beb (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402c25 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402c31 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402c40 (fcn.004028d0)
|  .....-> ; CODE (CALL) XREF from 0x00402c51 (fcn.004028d0)
```

After pressing `x` you will see a menu listing the same xrefs along with numbers in brackets:

```
[GOTO XREF]>
[0] CODE (CALL) XREF 0x00402b98 (loc.00402b38)
[1] CODE (CALL) XREF 0x00402ba0 (loc.00402b38)
[2] CODE (CALL) XREF 0x00402ba9 (loc.00402b38)
[3] CODE (CALL) XREF 0x00402bd5 (loc.00402b38)
[4] CODE (CALL) XREF 0x00402beb (loc.00402b38)
[5] CODE (CALL) XREF 0x00402c25 (loc.00402b38)
[6] CODE (CALL) XREF 0x00402c31 (loc.00402b38)
[7] CODE (CALL) XREF 0x00402c40 (loc.00402b38)
[8] CODE (CALL) XREF 0x00402c51 (loc.00402b38)
[9] CODE (CALL) XREF 0x00402c60 (loc.00402b38)
```

By pressing a number key from `0` to `9`, you can choose to move to a new position in the file assosiated with that corss-reference.

The history of seek positions is saved. You can always get back to a previous view in the file by pressing `u` key.

Visual Configuration Editor

`Ve` or `e` in visual mode allows you to edit radare2 configuration visually. For example, if you want to change the assembly display just select `asm` in the list and choose your assembly display flavor.

Example switch to pseudo disassembly:


[EvalSpace < Variables: asm.arch]
asm.functions = true
asm.indent = false
asm.lbytes = true
asm.lines = true
asm.linescall = false
asm.linesout = true
asm.linesright = false
asm.linesstyle = false
asm.lineswide = false
asm.lineswidth = 7
asm.maxrefs = 5
asm.middle = false
asm.nbytes = 6
asm.offset = true
asm.os = linux
asm.parser = x86.pseudo
> asm.pseudo = false
asm.reloff = false
asm.section = false
asm.segoff = false
asm.size = false
asm.stackptr = false
asm.syntax = intel
asm.tabs = 0
asm.trace = false
asm.tracespace = false
asm.ucase = false
asm.vars = true
asm.varsub = true
asm.varxs = false
asm.xrefs = true
Selected: asm.pseudo (Enable pseudo syntax)
/ (fcn) entry0 42
;-- entry0:
0x00404890 31ed xor ebp, ebp
0x00404892 4989d1 mov r9, rdx
0x00404895 5e pop rsi
0x00404896 4889e2 mov rdx, rsp
0x00404899 4883e4f0 and rsp, 0xfffffffffffffff0


[EvalSpace < Variables: asm.arch]
asm.functions = true
asm.indent = false
asm.lbytes = true
asm.lines = true
asm.linescall = false
asm.linesout = true
asm.linesright = false
asm.linesstyle = false
asm.lineswide = false
asm.lineswidth = 7
asm.maxrefs = 5
asm.middle = false
asm.nbytes = 6
asm.offset = true
asm.os = linux
asm.parser = x86.pseudo
> asm.pseudo = true
asm.reloff = false
asm.section = false
asm.segoff = false
asm.size = false
asm.stackptr = false
asm.syntax = intel
asm.tabs = 0
asm.trace = false
asm.tracespace = false
asm.ucase = false
asm.vars = true
asm.varsub = true
asm.varxs = false
asm.xrefs = true
Selected: asm.pseudo (Enable pseudo syntax)
/ (fcn) entry0 42
;-- entry0:
0x00404890 31ed ebp = 0
0x00404892 4989d1 r9 = rdx
0x00404895 5e pop rsi
0x00404896 4889e2 rdx = rsp
0x00404899 4883e4f0 rsp &= 0xfffffffffffffff0

Searching for Bytes

The radare2 search engine is based on work done by esteve, plus multiple features implemented on top of it. It supports multiple keyword searches, binary masks, hexadecimal. It automatically flags search hit entries to ease future referencing. Search is initiated by `/` command.

```
[0x00000000]> /?
Usage: /[amx/] [arg]
/ foo\x00       search for string `foo\0`
/w foo          search for wide string `f\0o\0o\0`
/wi foo         search for wide string ignoring case `f\0o\0o\0`
/! ff           search for first occurrence not matching
/i foo          search for string `foo` ignoring case
/e /E.F/i       match regular expression
/x ff0033       search for hex string
/x ff..33       search for hex string ignoring some nibbles
/x ff43 ffd0    search for hexpair with mask
/d 101112       search for a deltified sequence of bytes
/!x 00          inverse hexa search (find first byte != 0x00)
/c jmp [esp]    search for asm code (see search.asmstr)
/a jmp eax      assemble opcode and search its bytes
/A              search for AES expanded keys
/r sym.printf   analyze opcode reference an offset
/R              search for ROP gadgets
/P              show offset of previous instruction
/m magicfile    search for matching magic file (use blocksize)
/p patternsize  search for pattern of given size
/z min max      search for strings of given size
/v[?248] num    look for a asm.bigendian 32bit value
//              repeat last search
/b              search backwards
```

Because everything is treated as a file in radare2, it does not matter whether you search in a socket, a remote device, in process memory, or a file.

Basic Search

A basic search for a plain text string in a file would be something like:

```
$ r2 -q -c "/ lib" /bin/ls
Searching 3 bytes from 0x00400000 to 0x0041ae08: 6c 69 62
hits: 9
0x00400239 hit0_0 "lib64/ld-linux-x86-64.so.2"
0x00400f19 hit0_1 "libselinux.so.1"
0x00400fae hit0_2 "librt.so.1"
0x00400fc7 hit0_3 "libacl.so.1"
0x00401004 hit0_4 "libc.so.6"
0x004013ce hit0_5 "libc_start_main"
0x00416542 hit0_6 "libs/"
0x00417160 hit0_7 "lib/xstrtol.c"
0x00417578 hit0_8 "lib"
```

As it can been seen from the output above, radare2 generates a "hit" flag for every entry found. Then you can use `ps` command to see strings at offsets stored in this group of flags:

```
[0x00404888]> / ls
...
[0x00404888]> ps @ hit0_0
lseek
```

You can also search for wide-char strings (e.g., ones containing zeros between letters) using the `/w` in this way:

```
[0x00000000]> /w Hello
0 results found.
```

To perform a case-insensitive search for strings use `/i`:

```
[0x0040488f]> /i Stallman
Searching 8 bytes from 0x00400238 to 0x0040488f: 53 74 61 6c 6c 6d 61 6e
[# ]hits: 004138 < 0x0040488f  hits = 0
```

It is possible to specify hexadecimal escape sequences in the search string by prepending them with "\x":

```
[0x00000000]> / \x7FELF
```

But, if you look for a string of hexadecimal values, you would most likely prefer to pass it as hexpairs. Use `/x` command:

```
[0x00000000]> /x 7F454C46
```

Once the search is done, results are stored in the `search` flag space.

```
[0x00000000]> f
0x00000135 512 hit0_0
0x00000b71 512 hit0_1
0x00000bad 512 hit0_2
0x00000bdd 512 hit0_3
0x00000bfb 512 hit0_4
0x00000f2a 512 hit0_5
```

To remove "hit" flags after you do not need them anymore, use the `f@-hit*` command.

Often, during long search sessions, you will need to launch the latest search more than once. Rather than type the same string over and over again, you will probably prefer to use the `//` command.

```
[0x00000f2a]> //     ; repeat last search
```

Configuring Search Options

The radare2 search engine can be configured through several configuration variables, modifiable with `e` command.

```
e cmd.hit = x         ; radare2 command to execute on every search hit
e search.distance = 0 ; search string distance
e search.in = [foo]   ; search scope limit. Supported values: raw, block, file, section
e search.align = 4    ; only show search results aligned by specified boundary.
e search.from = 0     ; start address
e search.to = 0       ; end address
e search.asmstr = 0   ; search for string instead of assembly
e search.flags = true ; if enabled, create flags on hits
```

The `search.align` variable is used to limit valid search hits to certain alignment. For example, with `e search.align=4` you will see only hits found at 4-bytes aligned offsets.

The `search.flags` boolean variable instructs the search engine to flag hits so that they can be referenced later. If a currently running search is interrupted with `Ctrl-C` keyboard sequence, current search position is flagged with "search_stop".

Pattern Matching Search

The `/p` command allows you to apply repeated pattern searches on IO backend storage. It is possible to identify repeated byte sequences without explicitly specifying them. The only command's parameter sets minimum detectable pattern length. Here is an example:

```
[0x00000000]> /p 10
```

This command output will show different patterns found and how many times each of them is encountered.

Search Automatization

The `cmd.hit` eval variable is used to define a radare2 command to be executed when a matching entry is found by the search engine. If you want to run several commands, separate them with `;`. Alternatively, you can arrange them in a separate script, and then invoke it as a whole with `. script-file-name` command. For example:

```
[0x00404888]> e cmd.hit = p8 8
[0x00404888]> / lib
Searching 3 bytes from 0x00400000 to 0x0041ae08: 6c 69 62
hits: 9
0x00400239 hit4_0 "lib64/ld-linux-x86-64.so.2"
31ed4989d15e4889
0x00400f19 hit4_1 "libselinux.so.1"
31ed4989d15e4889
0x00400fae hit4_2 "librt.so.1"
31ed4989d15e4889
0x00400fc7 hit4_3 "libacl.so.1"
31ed4989d15e4889
0x00401004 hit4_4 "libc.so.6"
31ed4989d15e4889
0x004013ce hit4_5 "libc_start_main"
31ed4989d15e4889
0x00416542 hit4_6 "libs/"
31ed4989d15e4889
0x00417160 hit4_7 "lib/xstrtol.c"
31ed4989d15e4889
0x00417578 hit4_8 "lib"
31ed4989d15e4889
```

Searching Backwards

To search backwards, just use `/b` command.

Assembler Search

If you want to search for a certain assembler opcodes, you can either use `/c` or `/a` commands.

- `/c jmp [esp]` - search for specified asm mnemonics

  [0x00404888]> /c jmp qword [rdx] f hit_0 @ 0x0040e50d # 2: jmp qword [rdx] f hit_1 @ 0x00418dbb # 2: jmp qword [rdx] f hit_2 @ 0x00418fcb # 3: jmp qword [rdx] f hit_3 @ 0x004196ab # 6: jmp qword [rdx] f hit_4 @ 0x00419bf3 # 3: jmp qword [rdx] f hit_5 @ 0x00419c1b # 3: jmp qword [rdx] f hit_6 @ 0x00419c43 # 3: jmp qword [rdx]

- `/a jmp eax` - assemble string to machine code, and then search for resulting bytes.

  [0x00404888]> /a jmp eax hits: 1 0x004048e7 hit3_0 ffe00f1f8000000000b8

Searching for AES Keys

Thanks to Victor Muñoz, radare2 now has support of the algorithm he developed, capable of finding expanded AES keys with `/Ca` command. It searches from current seek position up to the `search.distance` limit, or until end of file is reached. You can interrupt current search by pressing `Ctrl-C`. For example, to look for AES keys in physical memory of your system:

```
$ sudo r2 /dev/mem
[0x00000000]> /Ca
0 AES keys found
```

Disassembling

Disassembling in radare is just a way to represent an array of bytes. It is handled as a special print mode within `p` command.

In the old times, when the radare core was smaller, the disassembler was handled by an external rsc file. That is, radare first dumped current block into a file, and then simply called `objdump` configured to disassemble for Intel, ARM etc... It was a working solution, but it was inefficient as it repeated the same actions over and over, because there were no caches. As a result, scrolling was terribly slow. Nowadays, the disassembler support is one of the basic features of radare. It now allows many options, including target architecture flavor, disassembler variants, among other things.

To see disassembly, use the `pd` command. It accepts a numeric argument to specify how many opcodes of current block you want to see. Most of commands in radare consider current block size as a default limit for data input. If you want to disassemble more bytes, you should use the `b` command to set new block size.

```
[0x00000000]> b 100    ; set block size to 100
[0x00000000]> pd       ; disassemble 100 bytes
[0x00000000]> pd 3     ; disassemble 3 opcodes
[0x00000000]> pD 30    ; disassemble 30 bytes
```

The `pD` command works like `pd` but accepts number of input bytes, instead of number of opcodes, as its parameter.

The "pseudo" syntax may be somewhat easier for a human to understand than default assembler notations. But it can become annoying if you read lots of code. To play with it:

```
[0x00405e1c]> e asm.pseudo = true
[0x00405e1c]> pd 3
          ; JMP XREF from 0x00405dfa (fcn.00404531)
          0x00405e1c    488b9424a80. rdx = [rsp+0x2a8]
          0x00405e24    64483314252. rdx ^= [fs:0x28]
          0x00405e2d    4889d8       rax = rbx

[0x00405e1c]> e asm.syntax = intel
[0x00405e1c]> pd 3
          ; JMP XREF from 0x00405dfa (fcn.00404531)
          0x00405e1c    488b9424a80. mov rdx, [rsp+0x2a8]
          0x00405e24    64483314252. xor rdx, [fs:0x28]
          0x00405e2d    4889d8       mov rax, rbx

[0x00405e1c]> e asm.syntax=att
[0x00405e1c]> pd 3
          ; JMP XREF from 0x00405dfa (fcn.00404531)
          0x00405e1c    488b9424a80. mov 0x2a8(%rsp), %rdx
          0x00405e24    64483314252. xor %fs:0x28, %rdx
          0x00405e2d    4889d8       mov %rbx, %rax
```

Adding Metadata to Disasembly

Typical work of reversing binary files makes the task of taking notes and adding annotations on top of disassembly, data views etc. very important. Radare offers multiple ways to store and retrieve such metadata information.

By following common basic *NIX principles it is easy to write a small utility in a scripting language which uses `objdump`, `otool`, etc. to obtain information from a binary and to import it into radare. For an example, take a look at one of many scripts that are distributed with radare, e.g., `idc2r.py`. To use it, invoke it as `idc2r.py file.idc > file.r2`. It reads an IDC file exported from an IDA Pro database and produces an r2 script containing the same comments, names of functions etc. You can import resulting 'file.r2' by using the dot `.` command of radare:

```
[0x00000000]> . file.r2
```

The `.` command is used to interpret Radare commands from external sources, including files and program output. For example, to omit generation of an intermediate file and import the script directly you can use this combination:

```
[0x00000000]> .!idc2r.py < file.idc
```

The `C` command is used to manage comments and data conversions. You can define a range of program's bytes to be interpreted either as code, binary data or string. It is possible to define flags and execute external code in certain seek position to fetch a comment from an external file or database.

Here's the help:

```
[0x00404cc0]> C?
|Usage: C[-LCvsdfm?] [...]
| C*                            List meta info in r2 commands
| C- [len] [@][ addr]           delete metadata at given address range
| CL[-] [addr|file:line [addr] ]  show 'code line' information (bininfo)
| Cl  file:line [addr]          add comment with line information
| CC[-] [comment-text]    add/remove comment. Use CC! to edit with $EDITOR
| CCa[-at]|[at] [text]    add/remove comment at given address
| Cv[-] offset reg name   add var substitution
| Cs[-] [size] [[addr]]   add string
| Ch[-] [size] [@addr]    hide data
| Cd[-] [size]            hexdump data
| Cf[-] [sz] [fmt..]      format memory (see pf?)
| Cm[-] [sz] [fmt..]      magic parse (see pm?)
[0x00404cc0]>


[0x00000000]> CCa 0x0000002 this guy seems legit

[0x00000000]> pd 2
           0x00000000    0000         add [rax], al
```

```
    ;      this guy seems legit
            0x00000002    0000         add [rax], al
```

The `C` command allows to change type for a byte range. Three basic types are: code (disassembly is done using asm.arch), data (a byte array) or string.

It is easier to manage data types conversion in the visual mode, because the action is bound to "d" key, short for "data type change". Use the cursor to select a range of bytes (press `c` key to toggle cursor mode and use HJKL keys to expand selection) and then press 'ds' to convert it to a string. Alternatively, you can use the `Cs` command from the shell:

```
[0x00000000]> f string_foo @ 0x800
[0x00000000]> Cs 10 @ string_foo
```

The folding/unfolding support is quite premature, but the idea comes from "folder" concept found in Vim editor. You can select a range of bytes in the disassembly view and press '<' to fold them in a single line, or '>' to unfold them. This is used to improve readability by hiding unimportant portions of code/data.

The `Cm` command is used to define a memory format string (the same used by the `pf` command). Here's a example:

```
[0x7fd9f13ae630]> Cf 16 2xi foo bar
[0x7fd9f13ae630]> pd
            ;-- rip:
            0x7fd9f13ae630 format 2xi foo bar {
0x7fd9f13ae630 [0] {
   foo : 0x7fd9f13ae630 = 0xe8e78948
   bar : 0x7fd9f13ae634 = 14696
}
0x7fd9f13ae638 [1] {
   foo : 0x7fd9f13ae638 = 0x8bc48949
   bar : 0x7fd9f13ae63c = 571928325
}
} 16
            0x7fd9f13ae633    e868390000   call 0x7fd9f13b1fa0
               0x7fd9f13b1fa0() ; rip
            0x7fd9f13ae638    4989c4       mov r12, rax
```

It is possible to define structures with simple oneliners. See 'print memory' for more information.

All these "C*" commands can also be accessed from the visual mode by pressing 'd' (data conversion) key.

ESIL

From the wiki page of radare2 at Github

ESIL stands for 'Evaluable Strings Intermediate Language'. It aims to describe a Forth-like representation for every target CPU opcode semantics. ESIL representations can be evaluated (interpreted) in order to emulate individual instructions. Each command of an ESIL expression is separated by a comma. Its virtual machine can be described as this:

```
    while ((word=haveCommand())) {
      if (word.isKeyword()) {
        esilCommands[word](esil);
      } else {
        esil.push (evaluateToNumber(word));
      }
      nextCommand();
    }
```

ESIL commands are operations that pop values from the stack, perform calculations and push result (if any) to the stack. The aim is to be able to express most of common operations performed by CPUs, like binary arithmetic operations, memory loads and stores, processing syscalls etc.

Use ESIL

```
[0x00000000]> e asm.esil = true
```

Syntax and Commands

====== A target opcode is translated into a comma separated list of ESIL expressions.

```
xor eax, eax    ->    0,eax,=,1,zf,=
```

Memory access is defined by brackets operation:

```
mov eax, [0x80480]   ->   0x80480,[],eax,=
```

Default operand size is determined by size of operation destination.

```
movb $0, 0x80480     ->   0,0x80480,=[1]
```

The `?` command checks whether the rest of the expression after it evaluates to zero or not. If it is zero,

the following expression is skipped, otherwise it is evaluated. `%` prefix indicates internal variables.

```
cmp eax, 123  ->   123,eax,==,%z,zf,=
jz eax        ->   zf,?{,eax,eip,=,}
```

If you want to run several expressions under a conditional, put them in curly braces:

```
zf,?{,eip,esp,=[],eax,eip,=,%r,esp,-=,}
```

Whitespaces, newlines and other chars are ignored. So the first thing when processing a ESIL program is to remove spaces:

```
esil = r_str_replace (esil, " ", "", R_TRUE);
```

Syscalls need special treatment. They are indicated by '$' at the beginning of an expression. You can pass an optional numeric value to specify a number of syscall. An ESIL emulator must handle syscalls. See (r_esil_syscall).

Arguments Order for Non-associative Operations

As discussed on IRC, current implementation works like this:

```
a,b,-      b - a
a,b,/=     b /= a
```

This approach is more readable, but it is less stack-friendly.

Special Instructions

NOPs are represented as empty strings. As it was said previously, syscalls are marked by '$' command. For example, '0x80,$'. It delegates emulation from the ESIL machine to a callback which implements syscalls for a specific OS/kernel.

Traps are implemented with the `<trap>,<code>,$$` command. They are used to throw exceptions for invalid instructions, division by zero, memory read error, etc.

Quick Analysis

Here is a list of some quick checks to retrieve information from an ESIL string. Relevant information will be probably found in the first expression of the list.

```
indexOf('[')       ->    have memory references
indexOf("=[")      ->    write in memory
indexOf("pc,=")    ->    modifies program counter (branch, jump, call)
```

```
indexOf("sp,=")    ->    modifies the stack (what if we found sp+= or sp-=?)
indexOf("=")       ->    retrieve src and dst
indexOf(":")       ->    unknown esil, raw opcode ahead
indexOf("%")       ->    accesses internal esil vm flags
indexOf("$")       ->    syscall
indexOf("$$")      ->    can trap
indexOf('++')      ->    has iterator
indexOf('--')      ->    count to zero
indexOf("?{")      ->    conditional
indexOf("LOOP")    ->    is a loop (rep?)
equalsTo("")       ->    empty string, means: nop (wrong, if we append pc+=x)
```

Common operations:

- Check dstreg
- Check srcreg
- Get destinaion
- Is jump
- Is conditional
- Evaluate
- Is syscall

CPU Flags

CPU flags are usually defined as single bit registers in the RReg profile. They and sometimes found under the 'flg' register type.

ESIL Flags

ESIL VM has an internal state flags that are read only and can be used to export those values to the underlying target CPU flags. It is because the ESIL VM always calculates all flag changes, while target CPUs only update flags under certain conditions or at specific instructions.

Internal flags are prefixed with '%' character.

```
z - zero flag, only set if the result of an operation is 0
b - borrow, this requires to specify from which bit (example: %b4 - checks if borrow from
c - carry, same like above (example: %c7 - checks if carry from bit 7)
p - parity
r - regsize ( asm.bits/8 )
```

Variables

Properties of the VM variables:

1. They have no predefined bit width. This way it should be easy to extend them to 128, 256 and 512 bits later, e.g. for MMX, SSE, AVX, Neon SIMD.
2. There can be unbound number of variables. It is done for SSA-form compatibility.

3. Register names have no specific syntax. They are just strings.
4. Numbers can be specified in any base supported by RNum (dec, hex, oct, binary ...)
5. Each ESIL backend should have an associated RReg profile to describe the ESIL register specs.

Bit Arrays

What to do with them? What about bit arithmetics if use variables instead of registers?

Arithmetics

1. ADD ("+")
2. MUL ("*")
3. SUB ("-")
4. DIV ("/")
5. MOD ("%")

Bit Arithmetics

1. AND "&"
2. OR "|"
3. XOR "^"
4. SHL "<<"
5. SHR ">>"
6. ROL "<<<"
7. ROR ">>>"
8. NEG "!"

Floating Point Support

TODO

Handling x86 REP Prefix in ESIL

ESIL specifies that the parsing control-flow commands must be uppercase. Bear in mind that some architectures have uppercase register names. The corresponding register profile should take care not to reuse any of the following:

```
3,SKIP   - skip N instructions. used to make relative forward GOTOs
3,GOTO   - goto instruction 3
LOOP     - alias for 0,GOTO
BREAK    - stop evaluating the expression
STACK    - dump stack contents to screen
CLEAR    - clear stack
```

Usage example:

rep cmpsb

cx,!,?{,BREAK,},esi,[1],edi,[1],==,?{,BREAK,},esi,++,edi,++,cx,--,LOOP

Unimplemented/unhandled Instructions

Those are expressed with the 'TODO' command. which acts as a 'BREAK', but displays a warning message describing that an instruction is not implemented and will not be emulated. For example:

```
fmulp ST(1), ST(0)      =>      TODO,fmulp ST(1),ST(0)
```

ESIL Disassembly Example:

```
[0x1000010f8]> e asm.esil=true
[0x1000010f8]> pd $r @ entry0
   ;      [0] va=0x1000010f8 pa=0x000010f8 sz=13299 vsz=13299 rwx=-r-x 0.__text
            ;-- section.0.__text:
            0x1000010f8    55           8,rsp,-=,rbp,rsp,=[8]
            0x1000010f9    4889e5       rsp,rbp,=
            0x1000010fc    4883c768     104,rdi,+=
            0x100001100    4883c668     104,rsi,+=
            0x100001104    5d           rsp,[8],rbp,=,8,rsp,+=
          |   0x10000110a    55           8,rsp,-=,rbp,rsp,=[8]
          |   0x10000110b    4889e5       rsp,rbp,=
          |   0x100001112    488d7768     rdi,104,+,rsi,=
          |   0x100001116    4889c7       rax,rdi,=
          |   0x100001119    5d           rsp,[8],rbp,=,8,rsp,+=
          ||   0x10000111f    55           8,rsp,-=,rbp,rsp,=[8]
          ||   0x100001120    4889e5       rsp,rbp,=
          ||   0x100001123    488b4f60     rdi,96,+,[8],rcx,=
          ||   0x100001127    4c8b4130     rcx,48,+,[8],r8,=
          ||   0x10000112f    b801000000   1,eax,= ;  0x00000001
          ||   0x100001134    4c394230     rdx,48,+,[8],r8,==,cz,?=
        ,=< 0x100001138    7f1a         sf,of,!,^,zf,!,&,?{,0x1154,rip,=,} ;[2]
      ,=<  0x10000113a    7d07         of,!,sf,^,?{,0x1143,rip,} ;[3]
      ||||   0x10000113c    b8ffffffff   0xffffffff,eax,= ;  0xffffffff
      |`---> 0x100001143    488b4938     rcx,56,+,[8],rcx,=
      |  |||   0x100001147    48394a38     rdx,56,+,[8],rcx,==,cz,?=
```

Introspection

To ease ESIL parsing we should have a way to express introspection expressions to extract data we want. For example, we may want to get the target address of a jump. The parser for ESIL expressions should offer API to make it possible to extract information by analyzing the expressions easily.

```
>  ao~esil,opcode
opcode: jmp 0x10000465a
esil: 0x10000465a,rip,=
```

We need a way to retrieve the numeric value of 'rip'. This is a very simple example, but there are more complex, like conditional ones. We need expressions to be able to get:

- opcode type
- destination of jump
- condition depends on
- all regs modified (write)
- all regs accessed (read)

API HOOKS

It is important for emulation to be able to setup hooks in parser, so we can extend it to implement analysis without having to change parser again and again. That is, every time an operation is about to be executed, a user hook is called. It can be used to determine if rip is going to change, or if the instruction updates stack, etc. Later, we can split that callback into several ones to have an event-based analysis API that may be extended in js like this: esil.on('regset', function(){.. esil.on('syscall', function(){esil.regset('rip'

For the API, see functions hook_flag_read(), hook_execute(), hook_mem_read(). A callback should return true if you want to override the action taken for a callback. For example, to deny memory reads in a region, or voiding memory writes, effectively making it read-only. Return false or 0 if you want to trace ESIL expression parsing.

Other operations that require bindings to external functionalities to work. In this case, r_ref and r_io. This must be defined when initializing the esil vm.

- Io Get/Set Out ax, 44 44,ax,:ou
- Selectors (cs,ds,gs...) Mov eax, ds:[ebp+8] Ebp,8,+,:ds,eax,=

Rabin2 — Show Properties of a Binary

Under this bunny-arabic-like name, radare hides a powerful tool to handle binary files, to get information on imports, sections, headers etc. Rabin2 can present it in several formats accepted by other tools, including radare2 itself. Rabin2 understands many file formats: Java CLASS, ELF, PE, Mach-O, etc., and it is able to obtain symbol import/exports, library dependencies, strings of data sections, xrefs, entrypoint address, sections, architecture type.

```
$ rabin2 -h

Usage: rabin2 [-ACdehHiIjlLMqrRsSvVxzZ] [-@ addr] [-a arch] [-b bits]
              [-B addr] [-c F:C:D] [-f str] [-m addr] [-n str] [-N len]
              [-o str] [-O str] file
 -@ [addr]       show section, symbol or import at addr
 -A              list archs
 -a [arch]       set arch (x86, arm, .. or <arch>_<bits>)
 -b [bits]       set bits (32, 64 ...)
 -B [addr]       override base address (pie bins)
 -c [fmt:C:D]    create [elf,mach0,pe] with Code and Data hexpairs (see -a)
 -C              list classes
 -d              show debug/dwarf information
 -e              entrypoint
 -f [str]        select sub-bin named str
 -g              same as -SMRevsiz (show all info)
 -h              this help
 -H              header fields
 -i              imports (symbols imported from libraries)
 -I              binary info
 -j              output in json
 -l              linked libraries
 -L              list supported bin plugins
 -m [addr]       show source line at addr
 -M              main (show address of main symbol)
 -n [str]        show section, symbol or import named str
 -N [minlen]     force minimum number of chars per string (see -z)
 -o [str]        output file/folder for write operations (out by default)
 -O [str]        write/extract operations (-O help)
 -q              be quiet, just show fewer data
 -r              radare output
 -R              relocations
 -s              symbols (exports)
 -S              sections
 -v              use vaddr in radare output (or show version if no file)
 -x              extract bins contained in file
 -z              strings (from data section)
 -zz             strings (from raw bins [e bin.rawstr=1])
 -Z              guess size of binary program
```

File Properties Identification

File type identification is done using `-I`. With this option, rabin2 prints information on a binary's type, its encoding, endianness, class, operating system, etc.:

```
$ rabin2 -I /bin/ls
file    /bin/ls
type    EXEC (Executable file)
pic false
has_va  true
root    elf
class   ELF64
lang    c
arch    x86
bits    64
machine AMD x86-64 architecture
os  linux
subsys  linux
endian  little
strip   true
static  false
linenum false
lsyms   false
relocs  false
rpath   NONE
```

To make rabin2 output information in format that the main program, radare2, can understand, pass `-Ir` option to it:

```
$ rabin2 -Ir /bin/ls
e file.type=elf
e cfg.bigendian=false
e asm.os=linux
e asm.arch=x86
e anal.arch=x86
e asm.bits=64
e asm.dwarf=true
```

Code Entrypoints

The `-e` option passed to rabin2 will show entrypoints for given binary. Two examples:

```
$ rabin2 -e /bin/ls
[Entrypoints]
addr=0x00004888 off=0x00004888 baddr=0x00000000

1 entrypoints

$ rabin2 -er /bin/ls
fs symbols
f entry0 @ 0x00004888
s entry0
```

Imports

Rabin2 is able to find imported objects by an executable, as well as their offsets in its PLT. This information is useful, for example, to understand what external function is invoked by `call` instruction. Pass `-i` flag to rabin to get a list of imports. An example:

```
$ rabin2 -i /bin/ls |head
[Imports]
ordinal=001 plt=0x000021b0 bind=GLOBAL type=FUNC name=__ctype_toupper_loc
ordinal=002 plt=0x000021c0 bind=GLOBAL type=FUNC name=__uflow
ordinal=003 plt=0x000021d0 bind=GLOBAL type=FUNC name=getenv
ordinal=004 plt=0x000021e0 bind=GLOBAL type=FUNC name=sigprocmask
ordinal=005 plt=0x000021f0 bind=GLOBAL type=FUNC name=raise
ordinal=006 plt=0x00002210 bind=GLOBAL type=FUNC name=localtime
ordinal=007 plt=0x00002220 bind=GLOBAL type=FUNC name=__mempcpy_chk
ordinal=008 plt=0x00002230 bind=GLOBAL type=FUNC name=abort
ordinal=009 plt=0x00002240 bind=GLOBAL type=FUNC name=__errno_location
(...)
```

Symbols (Exports)

With rabin2, generated symbols list format is similar to imports list. Use `-s` option to get it:

```
$ rabin2 -s /bin/ls | head
[Symbols]
addr=0x0021a610 off=0x0021a610 ord=114 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=stdout
addr=0x0021a600 off=0x0021a600 ord=115 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=_edata
addr=0x0021b388 off=0x0021b388 ord=116 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=_end
addr=0x0021a600 off=0x0021a600 ord=117 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=__progn
addr=0x0021a630 off=0x0021a630 ord=119 fwd=NONE sz=8 bind=UNKNOWN type=OBJECT name=progra
addr=0x0021a600 off=0x0021a600 ord=121 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=__bss_s
addr=0x0021a630 off=0x0021a630 ord=122 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=__progn
addr=0x0021a600 off=0x0021a600 ord=123 fwd=NONE sz=8 bind=UNKNOWN type=OBJECT name=progra
addr=0x00002178 off=0x00002178 ord=124 fwd=NONE sz=0 bind=GLOBAL type=FUNC name=_init
```

With `-sr` option rabin2 produces a radare2 script instead. It can be later passed to the core to automatically flag all symbols and to define corresponding byte ranges as functions and data blocks.

```
$ rabin2 -sr /bin/ls

fs symbols
Cd 8 @ 0x0021a610
f sym.stdout 8 0x0021a610
f sym._edata 0 0x0021a600
f sym._end 0 0x0021b388
Cd 8 @ 0x0021a600
f sym.__progname 8 0x0021a600
Cd 8 @ 0x0021a630
f sym.program_invocation_name 8 0x0021a630
f sym.__bss_start 0 0x0021a600
```

List Libraries

Rabin2 can list libraries used by a binary with `-l` option:

```
$ rabin2 -l /bin/ls
[Linked libraries]
libselinux.so.1
librt.so.1
libacl.so.1
libc.so.6

4 libraries
```

If you compare outputs of `rabin2 -l` and `ldd`, you will notice that rabin2 lists fewer libraries than `ldd`. The reason is that rabin2 does not follow and does not show dependencies of libraries. Only original binary dependencies are shown.

Strings

The `-z` option is used to list readable strings found in the .rodata section of ELF binaries, or the .text section of PE files. Example:

```
$ rabin2 -z /bin/ls |head
addr=0x00012487 off=0x00012487 ordinal=000 sz=9 len=9 section=.rodata type=A string=src/l
addr=0x00012490 off=0x00012490 ordinal=001 sz=26 len=26 section=.rodata type=A string=sor
addr=0x000124aa off=0x000124aa ordinal=002 sz=5 len=5 section=.rodata type=A string= %lu
addr=0x000124b0 off=0x000124b0 ordinal=003 sz=7 len=14 section=.rodata type=W string=%*lu
addr=0x000124ba off=0x000124ba ordinal=004 sz=8 len=8 section=.rodata type=A string=%s %*
addr=0x000124c5 off=0x000124c5 ordinal=005 sz=10 len=10 section=.rodata type=A string=%*s
addr=0x000124cf off=0x000124cf ordinal=006 sz=5 len=5 section=.rodata type=A string= ->
addr=0x000124d4 off=0x000124d4 ordinal=007 sz=17 len=17 section=.rodata type=A string=can
addr=0x000124e5 off=0x000124e5 ordinal=008 sz=29 len=29 section=.rodata type=A string=can
addr=0x00012502 off=0x00012502 ordinal=009 sz=10 len=10 section=.rodata type=A string=unl
```

With `-zr` option, this information is represented as radare2 commands list. It can be used in a radare2 session to automatically create a flag space called "strings" pre-populated with flags for all strings found by rabin2. Furthermore, this script will mark corresponding byte ranges as strings instead of code.

```
$ rabin2 -zr /bin/ls |head
fs strings
f str.src_ls.c 9 @ 0x00012487
Cs 9 @ 0x00012487
f str.sort_type__sort_version 26 @ 0x00012490
Cs 26 @ 0x00012490
f str._lu 5 @ 0x000124aa
Cs 5 @ 0x000124aa
f str.__lu_ 14 @ 0x000124b0
Cs 7 @ 0x000124b0
f str._s__s 8 @ 0x000124ba
(...)
```

Program Sections

Rabin2 called with `-S` option gives complete information about sections of an executable. For each section its index, offset, size, alignment, type and permissions, are shown. The next example demonstrates it.

```
$ rabin2 -S /bin/ls
 [Sections]
idx=00 addr=0x00000238 off=0x00000238 sz=28 vsz=28 perm=-r-- name=.interp
idx=01 addr=0x00000254 off=0x00000254 sz=32 vsz=32 perm=-r-- name=.note.ABI_tag
idx=02 addr=0x00000274 off=0x00000274 sz=36 vsz=36 perm=-r-- name=.note.gnu.build_id
idx=03 addr=0x00000298 off=0x00000298 sz=104 vsz=104 perm=-r-- name=.gnu.hash
idx=04 addr=0x00000300 off=0x00000300 sz=3096 vsz=3096 perm=-r-- name=.dynsym
idx=05 addr=0x00000f18 off=0x00000f18 sz=1427 vsz=1427 perm=-r-- name=.dynstr
idx=06 addr=0x000014ac off=0x000014ac sz=258 vsz=258 perm=-r-- name=.gnu.version
idx=07 addr=0x000015b0 off=0x000015b0 sz=160 vsz=160 perm=-r-- name=.gnu.version_r
idx=08 addr=0x00001650 off=0x00001650 sz=168 vsz=168 perm=-r-- name=.rela.dyn
idx=09 addr=0x000016f8 off=0x000016f8 sz=2688 vsz=2688 perm=-r-- name=.rela.plt
idx=10 addr=0x00002178 off=0x00002178 sz=26 vsz=26 perm=-r-x name=.init
idx=11 addr=0x000021a0 off=0x000021a0 sz=1808 vsz=1808 perm=-r-x name=.plt
idx=12 addr=0x000028b0 off=0x000028b0 sz=64444 vsz=64444 perm=-r-x name=.text
idx=13 addr=0x0001246c off=0x0001246c sz=9 vsz=9 perm=-r-x name=.fini
idx=14 addr=0x00012480 off=0x00012480 sz=20764 vsz=20764 perm=-r-- name=.rodata
idx=15 addr=0x0001759c off=0x0001759c sz=1820 vsz=1820 perm=-r-- name=.eh_frame_hdr
idx=16 addr=0x00017cb8 off=0x00017cb8 sz=8460 vsz=8460 perm=-r-- name=.eh_frame
idx=17 addr=0x00019dd8 off=0x00019dd8 sz=8 vsz=8 perm=-rw- name=.init_array
idx=18 addr=0x00019de0 off=0x00019de0 sz=8 vsz=8 perm=-rw- name=.fini_array
idx=19 addr=0x00019de8 off=0x00019de8 sz=8 vsz=8 perm=-rw- name=.jcr
idx=20 addr=0x00019df0 off=0x00019df0 sz=512 vsz=512 perm=-rw- name=.dynamic
idx=21 addr=0x00019ff0 off=0x00019ff0 sz=16 vsz=16 perm=-rw- name=.got
idx=22 addr=0x0001a000 off=0x0001a000 sz=920 vsz=920 perm=-rw- name=.got.plt
idx=23 addr=0x0001a3a0 off=0x0001a3a0 sz=608 vsz=608 perm=-rw- name=.data
idx=24 addr=0x0001a600 off=0x0001a600 sz=3464 vsz=3464 perm=-rw- name=.bss
idx=25 addr=0x0001a600 off=0x0001a600 sz=8 vsz=8 perm=---- name=.gnu_debuglink
idx=26 addr=0x0001a608 off=0x0001a608 sz=254 vsz=254 perm=---- name=.shstrtab

27 sections
```

With `-Sr` option, rabin2 will flag start/end of every section, and will pass the rest of information as a comment.

```
$ rabin2 -Sr /bin/ls
fs sections
S 0x00000238 0x00000238 0x0000001c 0x0000001c .interp 4
f section..interp 28 0x00000238
f section_end..interp 0 0x00000254
CC [00] va=0x00000238 pa=0x00000238 sz=28 vsz=28 rwx=-r-- .interp @ 0x00000238
S 0x00000254 0x00000254 0x00000020 0x00000020 .note.ABI_tag 4
f section..note.ABI_tag 32 0x00000254
f section_end..note.ABI_tag 0 0x00000274
CC [01] va=0x00000254 pa=0x00000254 sz=32 vsz=32 rwx=-r-- .note.ABI_tag @ 0x00000254
S 0x00000274 0x00000274 0x00000024 0x00000024 .note.gnu.build_id 4
f section..note.gnu.build_id 36 0x00000274
```

```
f section_end..note.gnu.build_id 0 0x00000298
CC [02] va=0x00000274 pa=0x00000274 sz=36 vsz=36 rwx=-r-- .note.gnu.build_id @ 0x00000274
S 0x00000298 0x00000298 0x00000068 0x00000068 .gnu.hash 4
f section..gnu.hash 104 0x00000298
f section_end..gnu.hash 0 0x00000300
CC [03] va=0x00000298 pa=0x00000298 sz=104 vsz=104 rwx=-r-- .gnu.hash @ 0x00000298
S 0x00000300 0x00000300 0x00000c18 0x00000c18 .dynsym 4
f section..dynsym 3096 0x00000300
f section_end..dynsym 0 0x00000f18
CC [04] va=0x00000300 pa=0x00000300 sz=3096 vsz=3096 rwx=-r-- .dynsym @ 0x00000300
S 0x00000f18 0x00000f18 0x00000593 0x00000593 .dynstr 4
f section..dynstr 1427 0x00000f18
f section_end..dynstr 0 0x000014ab
CC [05] va=0x00000f18 pa=0x00000f18 sz=1427 vsz=1427 rwx=-r-- .dynstr @ 0x00000f18
S 0x000014ac 0x000014ac 0x00000102 0x00000102 .gnu.version 4
f section..gnu.version 258 0x000014ac
f section_end..gnu.version 0 0x000015ae
(...)
```

Binary Diffing

This section is based on http://radare.today article "binary diffing"

Without any parameters, `radiff2` by default shows what bytes are changed and their corresponding offsets:

```
$ radiff2 genuine cracked
0x000081e0 85c00f94c0 => 9090909090 0x000081e0
0x0007c805 85c00f84c0 => 9090909090 0x0007c805

$ rasm2 -d 85c00f94c0
test eax, eax
sete al
```

Notice how the two jumps are nop'ed.

For bulk processing, you may want to have a higher-level overview of differences. This is why radare2 is able to compute the distance and the percentage of similarity between two files with the `-s` option:

```
$ radiff2 -s /bin/true /bin/false
similarity: 0.97
distance: 743
```

If you want more concrete data, it's also possible to count the differences, with the `-c` option:

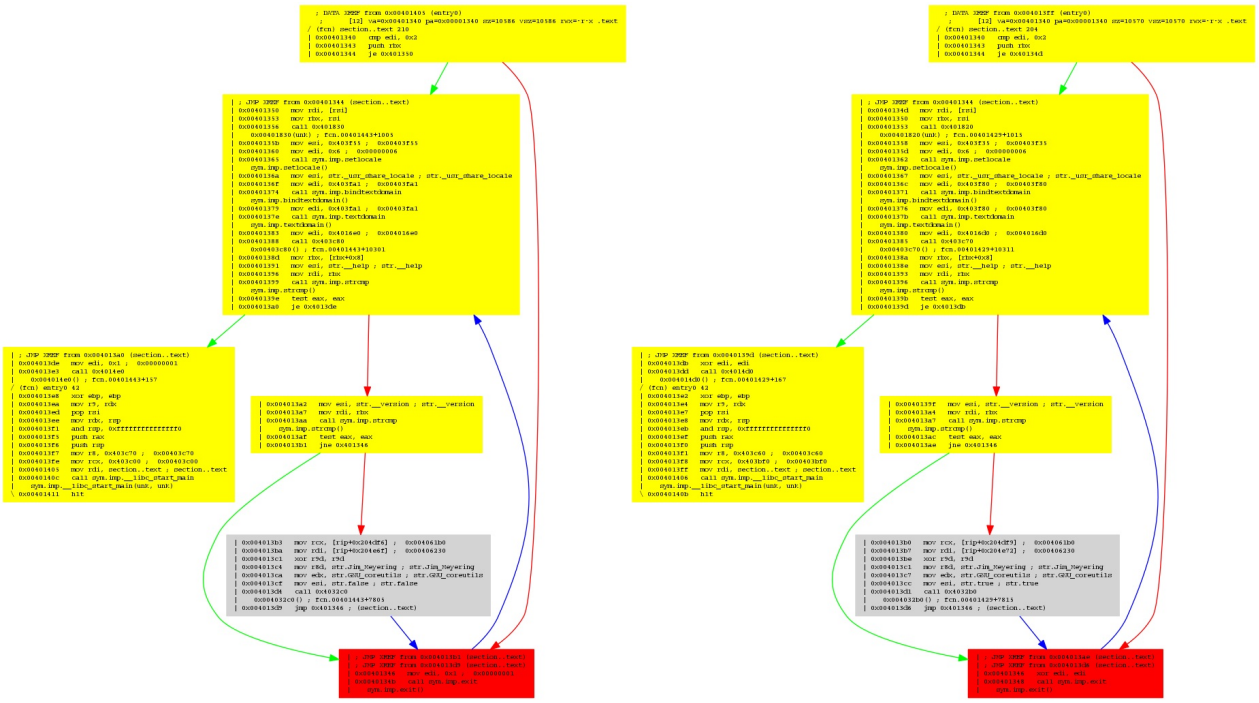
```
$ radiff2 -c genuine cracked
2
```

If you are unsure whether you are dealing with similar binaries, with `-C` flag you can check there are matching functions. It this mode, it will give you three columns for all functions: "First file offset", "Percentage of matching" and "Second file offset".

```
$ radiff2 -C /bin/false /bin/true
         entry0  0x4013e8 |   MATCH  (0.904762) | 0x4013e2  entry0
         sym.imp.__libc_start_main  0x401190 |   MATCH  (1.000000) | 0x401190  sym.imp.__
         fcn.00401196  0x401196 |   MATCH  (1.000000) | 0x401196  fcn.00401196
         fcn.0040103c  0x40103c |   MATCH  (1.000000) | 0x40103c  fcn.0040103c
         fcn.00401046  0x401046 |   MATCH  (1.000000) | 0x401046  fcn.00401046
         [...]
```

And now a cool feature : radare2 supports graph-diffing, à la DarunGrim, with the `-g` option. You can either give it a symbol name, of specify two offsets, if the function you want to diff is named differently in compared files. For example, `radiff2 -g main /bin/true /bin/false | xdot -` will show differences

in `main()` function of Unix `true` and `false` programs. You can compare it to `radiff2 -g main /bin/false /bin/true` (Notice the order of the arguments) to get the two versions. This is the result:



Parts in yellow indicate that some offsets do not match. The grey piece means a perfect match. The red one highlights a strong difference. If you look closely, you will see that the left part of the picture has `mov edi, 0x1; call sym.imp.exit`, while the right one has `xor edi, edi; call sym.imp.exit`.

Binary diffing is an important feature for reverse engineering. It can be used to analyze security updates, infected binaries, firmware changes and more...

We have only shown the code analysis diffing functionality, but radare2 supports additional types of diffing between two binaries: at byte level, deltified similarities, and more to come.

We have plans to implement more kinds of bindiffing algorithms into r2, and why not, add support for ascii art graph diffing and better integration with the rest of the toolkit.

Rasm2

`rasm2` is an inline assembler/disassembler. Initially, `rasm` tool was designed to be used for binary patching. It is main function is get bytes corresponding to given machine instruction opcode.

```
$ rasm2 -h
Usage: rasm2 [-CdDehLBvw] [-a arch] [-b bits] [-o addr] [-s syntax]
             [-f file] [-F fil:ter] [-i skip] [-l len] 'code'|hex|-
 -a [arch]    Set architecture to assemble/disassemble (see -L)
 -b [bits]    Set cpu register size (8, 16, 32, 64) (RASM2_BITS)
 -c [cpu]     Select specific CPU (depends on arch)
 -C           Output in C format
 -d, -D       Disassemble from hexpair bytes (-D show hexpairs)
 -e           Use big endian instead of little endian
 -f [file]    Read data from file
 -F [in:out]  Specify input and/or output filters (att2intel, x86.pseudo, ...)
 -h           Show this help
 -i [len]     ignore/skip N bytes of the input buffer
 -k [kernel]  Select operating system (linux, windows, darwin, ..)
 -l [len]     Input/Output length
 -L           List supported asm plugins
 -o [offset]  Set start address for code (default 0)
 -O [file]    Output file name (rasm2 -Bf a.asm -O a)
 -s [syntax]  Select syntax (intel, att)
 -B           Binary input/output (-l is mandatory for binary input)
 -v           Show version information
 -w           What's this instruction for? describe opcode
 If '-l' value is greater than output length, output is padded with nops
 If the last argument is '-' reads from stdin
```

Plugins for supported target architectures can be listed with the `-L` option. Knowing a plugin name, you can use it by specifying its name to the `-a` option

```
$ rasm2 -L
_d  16         8051        PD      8051 Intel CPU
_d  16 32      arc         GPL3    Argonaut RISC Core
ad  16 32 64   arm         GPL3    Acorn RISC Machine CPU
_d  16 32 64   arm.cs      BSD     Capstone ARM disassembler
_d  16 32      arm.winedbg LGPL2   WineDBG's ARM disassembler
_d  16 32      avr         GPL     AVR Atmel
ad  32         bf          LGPL3   Brainfuck
_d  16         cr16        LGPL3   cr16 disassembly plugin
_d  16         csr         PD      Cambridge Silicon Radio (CSR)
ad  32 64      dalvik      LGPL3   AndroidVM Dalvik
ad  16         dcpu16      PD      Mojang's DCPU-16
_d  32 64      ebc         LGPL3   EFI Bytecode
_d  8          gb          LGPL3   GameBoy(TM) (z80-like)
_d  16         h8300       LGPL3   H8/300 disassembly plugin
_d  8          i8080       BSD     Intel 8080 CPU
ad  32         java        Apache  Java bytecode
_d  32         m68k        BSD     Motorola 68000
_d  32         malbolge    LGPL3   Malbolge Ternary VM
ad  32 64      mips        GPL3    MIPS CPU
```

```
_d  16 32 64   mips.cs     BSD     Capstone MIPS disassembler
_d  16 32 64   msil        PD      .NET Microsoft Intermediate Language
_d  32         nios2       GPL3    NIOS II Embedded Processor
_d  32 64      ppc         GPL3    PowerPC
_d  32 64      ppc.cs      BSD     Capstone PowerPC disassembler
ad             rar         LGPL3   RAR VM
_d  32         sh          GPL3    SuperH-4 CPU
_d  32 64      sparc       GPL3    Scalable Processor Architecture
_d  32         tms320      LGPLv3  TMS320 DSP family
_d  32         ws          LGPL3   Whitespace esotheric VM
_d  16 32 64   x86         BSD     udis86 x86-16,32,64
_d  16 32 64   x86.cs      BSD     Capstone X86 disassembler
a_  32 64      x86.nz      LGPL3   x86 handmade assembler
ad  32         x86.olly    GPL2    OllyDBG X86 disassembler
ad  8          z80         NC-GPL2 Zilog Z80
```

Note that "ad" in the first column means both assembler and disasembler are offered by a corresponding plugin. "*d" indicates disassembler, "a*" means only assembler is available.

Assembler

`rasm2` can be used from the command-line to quickly copy-paste hexpairs that represent a given machine instruction.

```
$ rasm2 -a x86 -b 32 'mov eax, 33'
b821000000

$ echo 'push eax;nop;nop' | rasm2 -f -
5090
```

Rasm2 is used by radare2 core to write bytes using `wa` command.

The assembler understands the following input languages and their flavors: x86 (Intel and AT&T variants), olly (OllyDBG syntax), powerpc (PowerPC), arm and java. For Intel syntax, rasm2 tries to mimic NASM or GAS.

There are several examples in the rasm2 source code directory. Consult them to understand how you can assemble a raw binary file from a rasm2 description.

```
$ cat selfstop.rasm
;
; Self-Stop shellcode written in rasm for x86
;
; --pancake
;

.arch x86
.equ base 0x8048000
.org 0x8048000  ; the offset where we inject the 5 byte jmp

selfstop:
  push 0x8048000
  pusha
  mov eax, 20
  int 0x80

  mov ebx, eax
  mov ecx, 19
  mov eax, 37
  int 0x80
  popa
  ret
;
; The call injection
;

  ret

[0x00000000]> e asm.bits = 32
[0x00000000]> wx `!rasm2 -f a.rasm`
[0x00000000]> pd 20
```

```
        0x00000000    6800800408   push 0x8048000 ;  0x08048000
        0x00000005    60           pushad
        0x00000006    b814000000   mov eax, 0x14 ;  0x00000014
        0x0000000b    cd80         int 0x80
           syscall[0x80][0]=?
        0x0000000d    89c3         mov ebx, eax
        0x0000000f    b913000000   mov ecx, 0x13 ;  0x00000013
        0x00000014    b825000000   mov eax, 0x25 ;  0x00000025
        0x00000019    cd80         int 0x80
           syscall[0x80][0]=?
        0x0000001b    61           popad
        0x0000001c    c3           ret
        0x0000001d    c3           ret
```

Disassembler

Passing the `-d` option to rasm2 allows you to disassemble a hexpair string:

```
$ rasm2 -a x86 -b 32 -d '90'
nop
```

Data and Code Analysis

There are different commands to perform data and code analysis, to extract useful information from a binary, like pointers, string references, basic blocks, opcode data, jump targets, xrefs, etc. These operations are handled by the `a` (analyze) command family:

```
|Usage: a[?adfFghoprsx]
| a8 [hexpairs]    analyze bytes
| aa               analyze all (fcns + bbs)
| ad               analyze data trampoline (wip)
| ad [from] [to]   analyze data pointers to (from-to)
| ae [expr]        analyze opcode eval expression (see ao)
| af[rnbcsl?+-*]   analyze Functions
| aF               same as above, but using graph.depth=1
| ag[?acgdlf]      output Graphviz code
| ah[?lba-]        analysis hints (force opcode size, ...)
| ao[e?] [len]     analyze Opcodes (or emulate it)
| ap               find and analyze function preludes
| ar[?ld-*]        manage refs/xrefs (see also afr?)
| as [num]         analyze syscall using dbg.reg
| at[trd+-*?] [.]  analyze execution Traces
|Examples:
| f ts @ `S*~text:0[3]`; f t @ section..text
| f ds @ `S*~data:0[3]`; f d @ section..data
| .ad t t+ts @ d:ds
```

Code Analysis

Code analysis is a common technique used to extract information from assembly code. Radare uses internal data structures to identify basic blocks, function trees, to extract opcode-level information etc. The most common radare2 analysis command sequence is:

```
[0x08048440]> aa
[0x08048440]> pdf @ main

           ; DATA XREF from 0x08048457 (entry0)
/ (fcn) fcn.08048648 141
|          ;-- main:
|          0x08048648    8d4c2404     lea ecx, [esp+0x4]
|          0x0804864c    83e4f0       and esp, 0xfffffff0
|          0x0804864f    ff71fc       push dword [ecx-0x4]
|          0x08048652    55           push ebp
|          ; CODE (CALL) XREF from 0x08048734 (fcn.080486e5)
|          0x08048653    89e5         mov ebp, esp
|          0x08048655    83ec28       sub esp, 0x28
|          0x08048658    894df4       mov [ebp-0xc], ecx
|          0x0804865b    895df8       mov [ebp-0x8], ebx
|          0x0804865e    8975fc       mov [ebp-0x4], esi
|          0x08048661    8b19         mov ebx, [ecx]
|          0x08048663    8b7104       mov esi, [ecx+0x4]
|          0x08048666    c744240c000. mov dword [esp+0xc], 0x0
|          0x0804866e    c7442408010. mov dword [esp+0x8], 0x1 ;  0x00000001
|          0x08048676    c7442404000. mov dword [esp+0x4], 0x0
|          0x0804867e    c7042400000. mov dword [esp], 0x0
|          0x08048685    e852fdffff   call sym..imp.ptrace
|             sym..imp.ptrace(unk, unk)
|          0x0804868a    85c0         test eax, eax
|      ,=< 0x0804868c    7911         jns 0x804869f
|      |   0x0804868e    c70424cf870. mov dword [esp], str.Don_tuseadebuguer_ ;  0x080487
|      |   0x08048695    e882fdffff   call sym..imp.puts
|      |      sym..imp.puts()
|      |   0x0804869a    e80dfdffff   call sym..imp.abort
|      |      sym..imp.abort()
|      `-> 0x0804869f    83fb02       cmp ebx, 0x2
|     ,==< 0x080486a2    7411         je 0x80486b5
|     |    0x080486a4    c704240c880. mov dword [esp], str.Youmustgiveapasswordforusethis
|     |    0x080486ab    e86cfdffff   call sym..imp.puts
|     |       sym..imp.puts()
|     |    0x080486b0    e8f7fcffff   call sym..imp.abort
|     |       sym..imp.abort()
|     `--> 0x080486b5    8b4604       mov eax, [esi+0x4]
|          0x080486b8    890424       mov [esp], eax
|          0x080486bb    e8e5feffff   call fcn.080485a5
|             fcn.080485a5() ; fcn.080484c6+223
|          0x080486c0    b800000000   mov eax, 0x0
|          0x080486c5    8b4df4       mov ecx, [ebp-0xc]
|          0x080486c8    8b5df8       mov ebx, [ebp-0x8]
|          0x080486cb    8b75fc       mov esi, [ebp-0x4]
|          0x080486ce    89ec         mov esp, ebp
|          0x080486d0    5d           pop ebp
|          0x080486d1    8d61fc       lea esp, [ecx-0x4]
```

```
  \          0x080486d4    c3           ret
```

In this example, we analyze the whole file (`aa`) and then print disasembly of the `main()` function (`pdf`).

Obtaining Hashes within Radare2 Session

To calculate a checksum of current block when running rarare2, use the '#' command. Pass an algorithm name to it as a parameter. An example session:

```
$ radare2 /bin/ls
[0x08049790]> bf entry0
[0x08049790]> #md5
d2994c75adaa58392f953a448de5fba7
```

You can use all hashing algorithms supported by `rahash2`: md4, md5, crc16, crc32, sha1, sha256, sha384, sha512, par, xor, xorpair, mod255, hamdist, entropy. The `#` command accepts an optional numeric argument to specify length of byte range to be hashed, instead of default block size. For example:

```
[0x08049A80]> #md5 32
9b9012b00ef7a94b5824105b7aaad83b
[0x08049A80]> #md5 64
a71b087d8166c99869c9781e2edcf183
[0x08049A80]> #md5 1024
a933cc94cd705f09a41ecc80c0041def
[0x08049A80]>
```

Rahash2

The rahash2 tool can be used to calculate checksums and has functions of byte streams, files, text strings.

```
$ rahash2 -h
Usage: rahash2 [-rBhLkv] [-b sz] [-a algo] [-s str] [-f from] [-t to] [file] ...
-a algo     comma separated list of algorithms (default is 'sha256')
-b bsize    specify the size of the block (instead of full file)
-B          show per-block hash
-e          swap endian (use little endian)
-f from     start hashing at given address
-i num      repeat hash N iterations
-S seed     use given seed (hexa or s:string) use ^ to prefix
-k          show hash using the openssh's randomkey algorithm
-q          run in quiet mode (only show results)
-L          list all available algorithms (see -a)
-r          output radare commands
-s string   hash this string instead of files
-t to       stop hashing at given address
-v          show version information
```

To obtain an MD5 hash value of a text string, use the `-s` option:

```
$ rahash2 -q -a md5 -s 'hello world'
5eb63bbbe01eeed093cb22bb8f5acdc3
```

It is possible to calculate hash values for contents of files. But do not attempt to do it for large files, like complete disks. Before starting a calculation, rahash2 copies the whole input into a memory buffer.

To apply all algorithms known to rahash2, use `all` as an algorithm name:

```
$ rahash2 -a all /bin/ls
/bin/ls: 0x00000000-0x0001ae08 md5: b5607b4dc7d896c0fab5c4a308239161
/bin/ls: 0x00000000-0x0001ae08 sha1: c8f5032c2dce807c9182597082b94f01a3bec495
/bin/ls: 0x00000000-0x0001ae08 sha256: 978317d58e3ed046305df92a19f7d3e0bfcb3c70cad979f24f
/bin/ls: 0x00000000-0x0001ae08 sha384: 9e946efdbebb4e0ca00c86129ce2a71ee734ac30b620336c38
/bin/ls: 0x00000000-0x0001ae08 sha512: 076806cedb5281fd15c21e493e12655c55c52537fc1f36e641
/bin/ls: 0x00000000-0x0001ae08 crc16: 4b83
/bin/ls: 0x00000000-0x0001ae08 crc32: 6e316348
/bin/ls: 0x00000000-0x0001ae08 md4: 3a75f925a6a197d26bc650213f12b074
/bin/ls: 0x00000000-0x0001ae08 xor: 3e
/bin/ls: 0x00000000-0x0001ae08 xorpair: 59
/bin/ls: 0x00000000-0x0001ae08 parity: 01
/bin/ls: 0x00000000-0x0001ae08 entropy: 0567f925
/bin/ls: 0x00000000-0x0001ae08 hamdist: 00
/bin/ls: 0x00000000-0x0001ae08 pcprint: 23
/bin/ls: 0x00000000-0x0001ae08 mod255: 1e
/bin/ls: 0x00000000-0x0001ae08 xxhash: 138c936d
/bin/ls: 0x00000000-0x0001ae08 adler32: fca7131b
```

Debugger

Debuggers are implemented as IO plugins. Therefore, radare can handle different URI types for spawning, attaching and controlling processes. The complete list of IO plugins can be viewed with `r2 -L`. Those that have "d" in the first column ("rwd") support debugging. For example:

```
r_d  debug       Debug a program or pid. dbg:///bin/ls, dbg://1388 (LGPL3)
rwd  gdb         Attach to gdbserver, 'qemu -s', gdb://localhost:1234 (LGPL3)
```

There are different backends for many target architectures and operating systems, e.g., GNU/Linux, Windows, MacOS X, (Net,Free,Open)BSD and Solaris.

A process memory is treated as a plain file. All mapped memory pages of a debugged program and its libraries can be readed and interpreted as code, data structures etc.

Communication between radare and debugger IO layer is wrapped into `system()` calls, which accepts a string as an argument, and executes it as a command. An answer is then buffered in output console, its contents can be additionally processed by a script. This is how radare handles single `!` and double `!!` exclamation mark commands for calling `system()`:

```
[0x00000000]> ds
[0x00000000]> !!ls
```

The double exclamation mark `!!` tells radare to skip the IO plugin list, and to pass the rest of the command directly to shell. Using the single `!` to prepend a command will cause a walk through the IO plugin list to find one that handles it.

In general, debugger commands are portable between architectures and operating systems. Still, as radare tries to support the same functionality for all target architectures and operating systems, certain things have to be handled separately. They include injecting shellcodes and handling exceptions. For example, in MIPS targets there is no hardware-supported single-stepping feature. In this case, radare2 provides its own implementation for single-step by using a mix of code analysis and software breakpoints.

To get the basic help for debugger, type 'd?':

```
Usage: d[sbhcrbo] [arg]
dh [handler]   list or set debugger handler
dH [handler]   transplant process to a new handler
dd             file descriptors (!fd in r1)
ds[ol] N       step, over, source line
do             open process (reload, alias for 'oo')
dk [sig][=act] list, send, get, set, signal handlers of child
di[s] [arg..]  inject code on running process and execute it (See gs)
dp[=*?t][pid]  list, attach to process or thread id
dc[?]          continue execution. dc? for more
dr[?]          cpu registers, dr? for extended help
```

```
db[?]           breakpoints
dbt             display backtrace
dt[?r] [tag]    display instruction traces (dtr=reset)
dm[?*]          show memory maps
dw [pid]        block prompt until pid dies
```

To restart your debugging session, you can type `oo` or `oo+`, depending on desired behavior.

```
oo                reopen current file (kill+fork in debugger)
oo+               reopen current file in read-write
```

Registers

The registers are part of user area stored in the context structure used by the scheduler. This structure can be manipulated to get and set values of those registers, and, for example, on Intel hosts, it is possible to directly manipulate DR0-DR7 hardware registers to set hardware breakpoints.

There are different commands to get values of registers. For the General Purpose ones use:

```
[0x4A13B8C0]> dr
r15 = 0x00000000
r14 = 0x00000000
r13 = 0x00000000
r12 = 0x00000000
rbp = 0x00000000
rbx = 0x00000000
r11 = 0x00000000
r10 = 0x00000000
r9 = 0x00000000
r8 = 0x00000000
rax = 0x00000000
rcx = 0x00000000
rdx = 0x00000000
rsi = 0x00000000
rdi = 0x00000000
oeax = 0x0000003b
rip = 0x7f20bf5df630
rsp = 0x7fff515923c0

[0x7f0f2dbae630]> dr?rip ; get value of 'eip'
0x7f0f2dbae630

[0x4A13B8C0]> dr eip = esp   ; set 'eip' as esp
```

Interaction between a plugin and the core is done by commands returning radare instructions. This is used, for example, to set flags in the core to set values of registers.

```
[0x7f0f2dbae630]> dr*      ; Appending '*' will show radare commands
f r15 1 0x0
f r14 1 0x0
f r13 1 0x0
f r12 1 0x0
f rbp 1 0x0
f rbx 1 0x0
f r11 1 0x0
f r10 1 0x0
f r9 1 0x0
f r8 1 0x0
f rax 1 0x0
f rcx 1 0x0
f rdx 1 0x0
f rsi 1 0x0
f rdi 1 0x0
```

```
f oeax 1 0x3b
f rip 1 0x7fff73557940
f rflags 1 0x200
f rsp 1 0x7fff73557940


[0x4A13B8C0]> .dr*  ; include common register values in flags
```

An old copy of registers is stored all the time to keep track of the changes done during execution of a program being analyzed. This old copy can be accessed with `oregs`.

```
[0x7f1fab84c630]> dro
r15 = 0x00000000
r14 = 0x00000000
r13 = 0x00000000
r12 = 0x00000000
rbp = 0x00000000
rbx = 0x00000000
r11 = 0x00000000
r10 = 0x00000000
r9 = 0x00000000
r8 = 0x00000000
rax = 0x00000000
rcx = 0x00000000
rdx = 0x00000000
rsi = 0x00000000
rdi = 0x00000000
oeax = 0x0000003b
rip = 0x7f1fab84c630
rflags = 0x00000200
rsp = 0x7fff386b5080


[0x7f1fab84c630]> dr
r15 = 0x00000000
r14 = 0x00000000
r13 = 0x00000000
r12 = 0x00000000
rbp = 0x00000000
rbx = 0x00000000
r11 = 0x00000000
r10 = 0x00000000
r9 = 0x00000000
r8 = 0x00000000
rax = 0x00000000
rcx = 0x00000000
rdx = 0x00000000
rsi = 0x00000000
rdi = 0x7fff386b5080
oeax = 0xffffffffffffffff
rip = 0x7f1fab84c633
rflags = 0x00000202
rsp = 0x7fff386b5080
```

Values stored in eax, oeax and eip have changed.

To store and restore register values you can just dump the output of 'dr*' command to disk and then re-interpret it again:

```
[0x4A13B8C0]> dr* > regs.saved ; save registers
[0x4A13B8C0]> drp regs.saved ; restore
```

EFLAGS can be similarly altered. E.g., setting selected flags:

```
[0x4A13B8C0]> dr eflags = pst
[0x4A13B8C0]> dr eflags = azsti
```

You can get a string which represents latest changes of registers using `drd` command (diff registers):

```
[0x4A13B8C0]> drd
orax = 0x0000003b was 0x00000000 delta 59
rip = 0x7f00e71282d0 was 0x00000000 delta -418217264
rflags = 0x00000200 was 0x00000000 delta 512
rsp = 0x7fffe85a09c0 was 0x00000000 delta -396752448
```

Remote Access Capabilities

Radare can be run locally, or it can be started remotely just the same. It is possiblr because everything uses radare's IO subsystem that abstracts access to system(), cmd() and all basic IO operations through the network.

Help for commands useful for remote access to radare:

```
[0x00405a04]> =?
|Usage:  =[:!+-=hH] [...] # radare remote command execution protocol
|
rap commands:
| =                  list all open connections
| =<[fd] cmd         send output of local command to remote fd
| =[fd] cmd          exec cmd at remote 'fd' (last open is default one)
| =! cmd             run command via r_io_system
| =+ [proto://]host  add host (default=rap://, tcp://, udp://)
| =-[fd]             remove all hosts or host 'fd'
| ==[fd]             open remote session with host 'fd', 'q' to quit
|
rap server:
| =:port             listen on given port using rap protocol (o rap://9999)
| =:host:port cmd    run 'cmd' command on remote server
|
http server:
| =h port            listen for http connections (r2 -qc=H /bin/ls)
| =h-                stop background webserver
| =h*                restart current webserver
| =h& port           start http server in background)
| =H port            launch browser and listen for http
| =H& port           launch browser and listen for http in background
```

You can learn radare2 remote capabilities by displaying the list of supported IO plugins: `radare2 -L`.

A little example should help understanding. A typical remote session can be like this:

At the remote host1:

```
$ radare2 rap://:1234
```

At the remote host2:

```
$ radare2 rap://:1234
```

At localhost:

```
$ radare2 -
```

; Add hosts

```
[0x004048c5]> =+ rap://<host1>:1234//bin/ls
Connected to: <host1> at port 1234
waiting... ok

[0x004048c5]> =
0 - rap://<host1>:1234//bin/ls
```

You can open remote files in debug mode (or using any IO plugin) specifying URI when adding hosts:

```
[0x004048c5]> =+ =+ rap://<host2>:1234/dbg:///bin/ls
Connected to: <host2> at port 1234
waiting... ok
0 - rap://<host1>:1234//bin/ls
1 - rap://<host2>:1234/dbg:///bin/ls
```

To execute commands on host1:

```
[0x004048c5]> =0 px
[0x004048c5]> = s 0x666
```

To open a session with host2:

```
[0x004048c5]> ==1
fd:6> pi 1
...
fd:6> q
```

To remove hosts (and close connections):

```
[0x004048c5]> =-
```

If you can initialize a TCP or UDP server, add it with '=+ tcp://' or '=+ udp://'. Then redirect radare output to them. For instance:

```
 [0x004048c5]> =+ tcp://<host>:<port>/
Connected to: <host> at port <port>
5 - tcp://<host>:<port>/
[0x004048c5]> =<5 cmd...
```

The `=<' command will send result of a command's execution at the right to the remote connection number N (or the last one used if no id specified).

Plugins

IO plugins

All the access to files, network, debugger, etc.. is wrapped by an IO abstraction layer that allows to interpret all the data as if it was a single file.

IO plugins are the ones used to wrap the open, read, write and 'system' on virtual file systems. You can make radare understand anything as a plain file. E.g., a socket connection, a remote radare session, a file, a process, a device, a gdb session, etc..

So, when radare reads a block of bytes, it is the task of an IO plugin to get these bytes from any place and put them into internal buffer. An IO plugin is chosen by a file's URI to be opened. Some examples:

- Debugging URIs

  $ r2 dbg:///bin/ls $ r2 pid://1927

- Remote sessions

  $ r2 rap://:1234 $ r2 rap://:1234//bin/ls

- Virtual buffers

  $ r2 malloc://512 shortcut for $ r2 -

You can get a list of the radare IO plugins by typing `radare2 -L`:

```
$ r2 -L
rw_  zip         Open zip files apk://foo.apk//MANIFEST or zip://foo.apk//theclass/fun.cl
rwd  windbg      Attach to a KD debugger (LGPL3)
rw_  sparse      sparse buffer allocation (sparse://1024 sparse://) (LGPL3)
rw_  shm         shared memory resources (shm://key) (LGPL3)
rw_  self        read memory from myself using 'self://' (LGPL3)
rw_  rap         radare network protocol (rap://:port rap://host:port/file) (LGPL3)
rwd  ptrace      ptrace and /proc/pid/mem (if available) io (LGPL3)
rw_  procpid     /proc/pid/mem io (LGPL3)
rw_  mmap        open file using mmap:// (LGPL3)
rw_  malloc      memory allocation (malloc://1024 hex://cd8090) (LGPL3)
r__  mach        mach debug io (unsupported in this platform) (LGPL)
rw_  ihex        Intel HEX file (ihex://eeproms.hex) (LGPL)
rw_  http        http get (http://radare.org/) (LGPL3)
rw_  gzip        read/write gzipped files (LGPL3)
rwd  gdb         Attach to gdbserver, 'qemu -s', gdb://localhost:1234 (LGPL3)
r_d  debug       Debug a program or pid. dbg:///bin/ls, dbg://1388 (LGPL3)
rw_  bfdbg       BrainFuck Debugger (bfdbg://path/to/file) (LGPL3)
```

Crackmes

Crackmes (from "crack me" challenge) are the training ground for reverse engineering people. This section will go over tutorials on how to defeat various crackmes using r2.

IOLI CrackMes

The IOLI crackme is a good starting point for learning r2. This is a set of tutorials based on the tutorial at dustri

The IOLI crackmes are available at a locally hosted mirror

IOLI 0x00

This is the first IOLI crackme, and the easiest one.

```
$ ./crackme0x00
IOLI Crackme Level 0x00
Password: 1234
Invalid Password!
```

The first thing to check is if the password is just plaintext inside the file. In this case, we don't need to do any disassembly, and we can just use rabin2 with the -z flag to search for strings in the binary.

```
$ rabin2 -z ./crackme0x00
vaddr=0x08048568 paddr=0x00000568 ordinal=000 sz=25 len=24 section=.rodata type=a string=
vaddr=0x08048581 paddr=0x00000581 ordinal=001 sz=11 len=10 section=.rodata type=a string=
vaddr=0x0804858f paddr=0x0000058f ordinal=002 sz=7 len=6 section=.rodata type=a string=25
vaddr=0x08048596 paddr=0x00000596 ordinal=003 sz=19 len=18 section=.rodata type=a string=
vaddr=0x080485a9 paddr=0x000005a9 ordinal=004 sz=16 len=15 section=.rodata type=a string=
```

So we know what the following section is, this section is the header shown when the application is run.

```
vaddr=0x08048568 paddr=0x00000568 ordinal=000 sz=25 len=24 section=.rodata type=a string=
```

Here we have the prompt for the password.

```
vaddr=0x08048581 paddr=0x00000581 ordinal=001 sz=11 len=10 section=.rodata type=a string=
```

This is the error on entering an invalid password.

```
vaddr=0x08048596 paddr=0x00000596 ordinal=003 sz=19 len=18 section=.rodata type=a string=
```

This is the message on the password being accepted.

```
vaddr=0x080485a9 paddr=0x000005a9 ordinal=004 sz=16 len=15 section=.rodata type=a string=
```

But what is this? It's a string, but we haven't seen it in running the application yet.

```
vaddr=0x0804858f paddr=0x0000058f ordinal=002 sz=7 len=6 section=.rodata type=a string=25
```

Let's give this a shot.

```
$ ./crackme0x00
IOLI Crackme Level 0x00
Password: 250382
Password OK :)
```

So we now know that 250382 is the password, and have completed this crackme.

IOLI 0x01

This is the second IOLI crackme.

```
$ ./crackme0x01
IOLI Crackme Level 0x01
Password: test
Invalid Password!
```

Let's check for strings with rabin2.

```
$ rabin2 -z ./crackme0x01
vaddr=0x08048528 paddr=0x00000528 ordinal=000 sz=25 len=24 section=.rodata type=a string=
vaddr=0x08048541 paddr=0x00000541 ordinal=001 sz=11 len=10 section=.rodata type=a string=
vaddr=0x0804854f paddr=0x0000054f ordinal=002 sz=19 len=18 section=.rodata type=a string=
vaddr=0x08048562 paddr=0x00000562 ordinal=003 sz=16 len=15 section=.rodata type=a string=
```

This isn't going to be as easy as 0x00. Let's try disassembly with r2.

```
$ r2 ./crackme0x01 -- Use `zoom.byte=printable` in zoom mode ('z' in Visual mode) to find
[0x08048330]> aa
[0x08048330]> pdf@main
/ (fcn) main 113
|          ; var int local_4 @ ebp-0x4
|          ; DATA XREF from 0x08048347 (entry0)
|          0x080483e4    55           push ebp
|          0x080483e5    89e5         mov ebp, esp
|          0x080483e7    83ec18       sub esp, 0x18
|          0x080483ea    83e4f0       and esp, -0x10
|          0x080483ed    b800000000   mov eax, 0
|          0x080483f2    83c00f       add eax, 0xf
|          0x080483f5    83c00f       add eax, 0xf
|          0x080483f8    c1e804       shr eax, 4
|          0x080483fb    c1e004       shl eax, 4
|          0x080483fe    29c4         sub esp, eax
|          0x08048400    c7042428850. mov dword [esp], str.IOLI_Crackme_Level_0x01_n ; [0
|          0x08048407    e810ffffff   call sym.imp.printf
|             sym.imp.printf(unk)
|          0x0804840c    c7042441850. mov dword [esp], str.Password_ ; [0x8048541:4]=0x73
|          0x08048413    e804ffffff   call sym.imp.printf
|             sym.imp.printf()
|          0x08048418    8d45fc       lea eax, dword [ebp + 0xfffffffc]
|          0x0804841b    89442404     mov dword [esp + 4], eax ; [0x4:4]=0x10101
|          0x0804841f    c704244c850. mov dword [esp], 0x804854c ; [0x804854c:4]=0x490064
|          0x08048426    e8e1feffff   call sym.imp.scanf
|             sym.imp.scanf()
|          0x0804842b    817dfc9a140. cmp dword [ebp + 0xfffffffc], 0x149a
|      ,=< 0x08048432    740e         je 0x8048442
|      |   0x08048434    c704244f850. mov dword [esp], str.Invalid_Password__n ; [0x80485
|      |   0x0804843b    e8dcfeffff   call sym.imp.printf
```

```
|      |      sym.imp.printf()
|     ,==< 0x08048440    eb0c         jmp 0x804844e ; (main)
|     ||   ; JMP XREF from 0x08048432 (main)
|     |`-> 0x08048442    c7042462850. mov dword [esp], str.Password_OK____n ; [0x8048562:
|     |    0x08048449    e8cefeffff   call sym.imp.printf
|     |       sym.imp.printf()
|     |    ; JMP XREF from 0x08048440 (main)
|     `--> 0x0804844e    b800000000   mov eax, 0
|          0x08048453    c9           leave
\          0x08048454    c3           ret
```

"aa" tells r2 to analyze the whole binary, which gets you symbol names, among things.

"pdf" stands for

- Print
- Disassemble
- Function

This will print the disassembly of the main function, or the `main()` that everyone knows. You can see several things as well: weird names, arrows, etc.

- "imp." stands for imports. Those are imported symbols, like printf()
- "str." stands for strings. Those are strings (obviously).

If you look carefully, you'll see a `cmp` instruction, with a constant, 0x149a. `cmp` is an x86 compare instruction, and the 0x in front of it specifies it is in base 16, or hex (hexadecimal).

```
0x0804842b    817dfc9a140. cmp dword [ebp + 0xfffffffc], 0x149a
```

You can use radare2's `?` command to get it in another numeric base.

```
[0x08048330]> ? 0x149a
5274 0x149a 012232 5.2K 0000:049a 5274 10011010 5274.0 0.000000
```

So now we know that 0x149a is 5274 in decimal. Let's try this as a password.

```
$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 5274
Password OK :)
```

Bingo, the password was 5274. In this case, the password function at 0x0804842b was comparing the input against the value, 0x149a in hex. Since user input is usually decimal, it was a safe bet that the input was intended to be in decimal, or 5274. Now, since we're hackers, and curiosity drives us, let's see what

happens when we input in hex.

```
$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 0x149a
Invalid Password!
```

It was worth a shot, but it doesn't work. That's because `scanf()` will take the 0 in 0x149a to be a zero, rather than accepting the input as actually being the hex value.

And this concludes IOLI 0x01.

Radare2 Reference Card

Survival Guide

Command	Description
aa	Auto analyze
Content Cell	Content Cell
pdf@fcn(Tab)	Disassemble function
f fcn(Tab)	List functions
f str(Tab)	List strings
fr [flagname] [newname]	Rename flag
psz [offset]	Print string
arf [flag]	Find cross reference for a flag

Flagspaces

Command	Description
fs	Display flagspaces
fs *	Select all flagspaces
fs [sections]	Select one flagspace

Flags

Command	Description
f	List flags
fs *	Select all flagspaces
fs [sections]	Select one flagspace
fj	Display flags in JSON
fl	Show flag length
fx	Show hexdump of flag
fC [name] [comment]	Set flag comment

Information

Command	Description
ii	Information on imports
iI	Info on binary

ie	Display entrypoint
iS	Display sections
ir	Display relocations

Print string

Command	Description
psz [offset]	Print zero terminated string
psb [offset]	Print strings in current block
psx [offset]	Show string with scaped chars
psp [offset]	Print pascal string
psw [offset]	Print wide string

Visual mode

Command	Description
V	Enter visual mode
p/P	Rotate modes (hex, disasm, debug, words, buf)
c	Toggle (c)ursor
q	Back to Radare shell
hjkl	Move around (or HJKL) (left-down-up-right)
Enter	Follow address of jump/call
sS	Step/step over
o	Go/seek to given offset
.	Seek to program counter
/	In cursor mode, search in current block
:cmd	Run radare command
;[-]cmt	Add/remove comment
/*+-[]	Change block size, [] = resize hex.cols
>||<	Seek aligned to block size
i/a/A	(i)nsert hex, (a)ssemble code, visual (A)ssembler
b/B	Toggle breakpoint / automatic block size
d[f?]	Define function, data, code, ..
D	Enter visual diff mode (set diff.from/to)
e	Edit eval configuration variables
f/F	Set/unset flag
gG	Go seek to begin and end of file (0-$s)
mK/'K	Mark/go to Key (any key)

M	Walk the mounted filesystems
n/N	Seek next/prev function/flag/hit (scr.nkey)
o	Go/seek to given offset
C	Toggle (C)olors
R	Randomize color palette (ecr)
t	Track flags (browse symbols, functions..)
T	Browse anal info and comments
v	Visual code analysis menu
V/W	(V)iew graph (agv?), open (W)ebUI
uU	Undo/redo seek
x	Show xrefs to seek between them
yY	Copy and paste selection
z	Toggle zoom mode

Searching

Command	Description
/ foo\00	Search for string ’foo\0’
/b	Search backwards
//	Repeat last search
/w foo	Search for wide string ’f\0o\0o\0’
/wi foo	Search for wide string ignoring case
/! ff	Search for first occurrence not matching
/i foo	Search for string ’foo’ ignoring case
/e /E.F/i	Match regular expression
/x ff0.23	Search for hex string
/x ff..33	Search for hex string ignoring some nibbles
/x ff43 ffd0	Search for hexpair with mask
/d 101112	Search for a deltified sequence of bytes
/!x 00	Inverse hexa search (find first byte != 0x00)
/c jmp [esp]	Search for asm code (see search.asmstr)
/a jmp eax	Assemble opcode and search its bytes
/A	Search for AES expanded keys
/r sym.printf	Analyze opcode reference an offset
/R	Search for ROP gadgets
/P	Show offset of previous instruction
/m magicfile	Search for matching magic file
/p patternsize	Search for pattern of given size

/z min max	Search for strings of given size
/v[?248] num	Look for a asm.bigendian 32bit value

Saving

Command	Description
Po [file]	Open project
Ps [file]	Save project
Pi [file]	Show project information

Usable variables in expression

Command	Description
$$	Here (current virtual seek)
$o	Here (current disk io offset)
$s	File size
$b	Block size
$w	Get word size, 4 if asm.bits=32, 8 if 64
$c,$r	Get width and height of terminal
$S	Section offset
$SS	Section size
$j	Jump address (jmp 0x10, jz 0x10 => 0x10)
$f	Jump fail address (jz 0x10 => next instruction)
$l	Number of instructions of current function
$F	Current function size
$Jn	Get nth jump of function
$Cn	Get nth call of function
$Dn	Get nth data reference in function
$Xn	Get nth xref of function
$m	Opcode memory reference (mov eax,[0x10] => 0x10)
$l	Opcode length
$e	1 if end of block, else 0
$ev	Get value of eval config variable
$?	Last comparison value

License

This chapter is based on the Radare 2 reference card by Thanat0s, which is under the GNU GPL. Original license is as follows:

```
This card may be freely distributed under the terms of the GNU
general public licence — Copyright c  by Thanat0s - v0.1 -
```