```
[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      0f1f8000000.   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      49c70000000.   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      48c70300000.   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

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

Authors & Contributors

The radare2 book

Rework by maijin on the original radare book

Contributors:

- Anton Kochkov
- aoighost
- Austin Hartzheim
- David Tomaschik
- DZ_ruyk
- Grigory Rechistov
- hdznrrd
- Jeffrey Crowell
- John
- Judge Dredd
- jvoisin
- Kevin Grandemange
- muzlightbeer
- Peter C
- sghctoma
- SkUaTeR
- TDKPS
- Thanat0s

The original radare book

Original author and greetings from The radare book : pancake

- God. aka Flying Spaghetti Monster
- Nibble (ELF32/64 and PE parser+lot of bugfixes and core work)
- ora8 (w32 port debugger, syscallproxying, hw breakpoints..)
- nopcode guys (for the cons and
- Sexy Pandas (let's pwn the plugs!)
- 48bits (keep up the good work)
- Gerardo (ideas and tips for the book)
- pof (for the crackme tutorial and usability tips)
- Esteve (search engine+some code graph stuff)
- revenge (OSX debugger+mach0 work)
- Lia (4teh luf :)

Introduction

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

--pancake

History

The radare project began in February of 2006 to provide a free and simple command-line hexadecimal editor with support for 64-bit offsets. The intention was to use the tool to perform searches and help recover data from hard-disks.

Since then, the project has evolved to provide a complete framework for analyzing binaries while making use of basic *NIX concepts. Those concepts include the famous "everything is a file," "small programs that interact using stdin/stdout," and "keep it simple" paradigms.

It is mostly a single-person project. However, ideas and source code contributions are greatly appreciated.

The central focus of this project is the hexadecimal editor. Additionally, this project contains an assembler/disassembler, code/data analysis and graphing tools, scripting features, easy Unix integration, and more.

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).

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 -
509090
```

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.

Examples

```
$ rahash2 file
file: 0x00000000-0x00000007 sha256: 887cfbd0d44aaff69f7bdbedebd282ec96191cce9d7fa73362
98a18efc3c7a5a

$ rahash2 file -a md5
file: 0x00000000-0x00000007 md5: d1833805515fc34b46c2b9de553f599d
```

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); //x64 write@syscall(1);
exit@syscall(1); //x64 exit@syscall(60);

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

$ cat hi.c
main@global(0,6) {
  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 - Create a new file containing the following rarun2 profile named foo.rr2:

```
#!/usr/bin/rarun2
program=/bin/ls
stdio=/dev/ttys010
```

3 - Launch the following radare2 command: r2 -R foo.rr2 -d ls

rax2

A minimalistic mathematical expression evaluator for the shell that is useful for making base conversions between floating point values, hexadecimal representations, hexpair strings to ASCII, octal to integer, etc. It also supports 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 automate 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

Mingw32

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 isn'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**. Before the starting of compilation you need to setup git first, for a proper automatic fetching of capstone:

```
git config --global core.autocrlf true
git config --global core.filemode false
```

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.

To simplify the building under Windows/Mingw32 there is a script in radare2 sources: `sys/mingw32.bat`. Simply run it from the cmd.exe (or ConEmu/cmd.exe). It assumes that you have Mingw32 installed in `C:\Mingw` and Git in `C:\Program Files (x86)\Git`. If you want to use another installations, just set `MINGW_PATH` and `GIT_PATH` variables correspondingly:

```
set MINGW_PATH=D:\Mingw32
set "GIT_PATH=E:\Program and Stuff\Git"
sys\mingw32.bat
```

Please, note, that this script should be run from radare2 directory.

Cygwin

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.

Note, Cygwin build require exactly the opposite git configuration, so setup git first, for a proper automatic fetching of capstone:

```
git config --global core.autocrlf false
```

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-tool
  chain
  ```

- 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 patch
```

- Compile the radare2:

```
./configure --with-ostype=windows ; 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
blocksize] [-c cmd] [-e k=v] file|-

   -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 alternate (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 `s++` and `s--` commands to navigate this way.

If r2 opens an executable file, by default it will open the file in VA mode and the sections will be mapped to their virtual addresses. In VA mode, seeking is based on the virtual address and the starting position is set to the entry point of the executable. Using `-n` option you can suppress this default behavior and ask r2 to open the file in non-VA mode for you. In non-VA mode, seeking is based on the offset from the beginning of the file.

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 0.00
0000
[0xB7F9D810]> ? 0x8048000+34
134512674 0x8048022 01001100042 128.0M 804000:0022 134512674 00100010 134512674.0 0.00
0000
[0xB7F9D810]> ? 0x8048000+0x34
134512692 0x8048034 01001100064 128.0M 804000:0034 134512692 00110100 134512692.0 0.00
0000
[0xB7F9D810]> ? 1+2+3-4*3
-6 0xfffffffffffffffa 01777777777777777777772 17179869183.0G fffff000:0ffa -6
```

Supported arithmetic operations are:

- + : addition
- - : subtraction
- * : multiplication
- / : division
- % : modulus
- > : shift right
- < : shift left

Use of logical OR should be escaped using quotes, or it will be mistaken for a pipe opeator:

```
[0x00000000]> "? 1 | 2" 3 0x3 03 3 0000:0003 3 "\x03" 00000011 2.0 2.000000f 2.000000
```

Numbers can be displayed in several formats:

```
0x033   : hexadecimal can be displayed
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 type @@?      ; 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 1402938378129
00 10100100 140293837812900.0 -0.000000


[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 the `-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 overwrite 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.

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 configuration edit looks like this:

```
Eval spaces:

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

For configuration values that can take one of several values, you can use the `=?` operator to get a list of valid values:

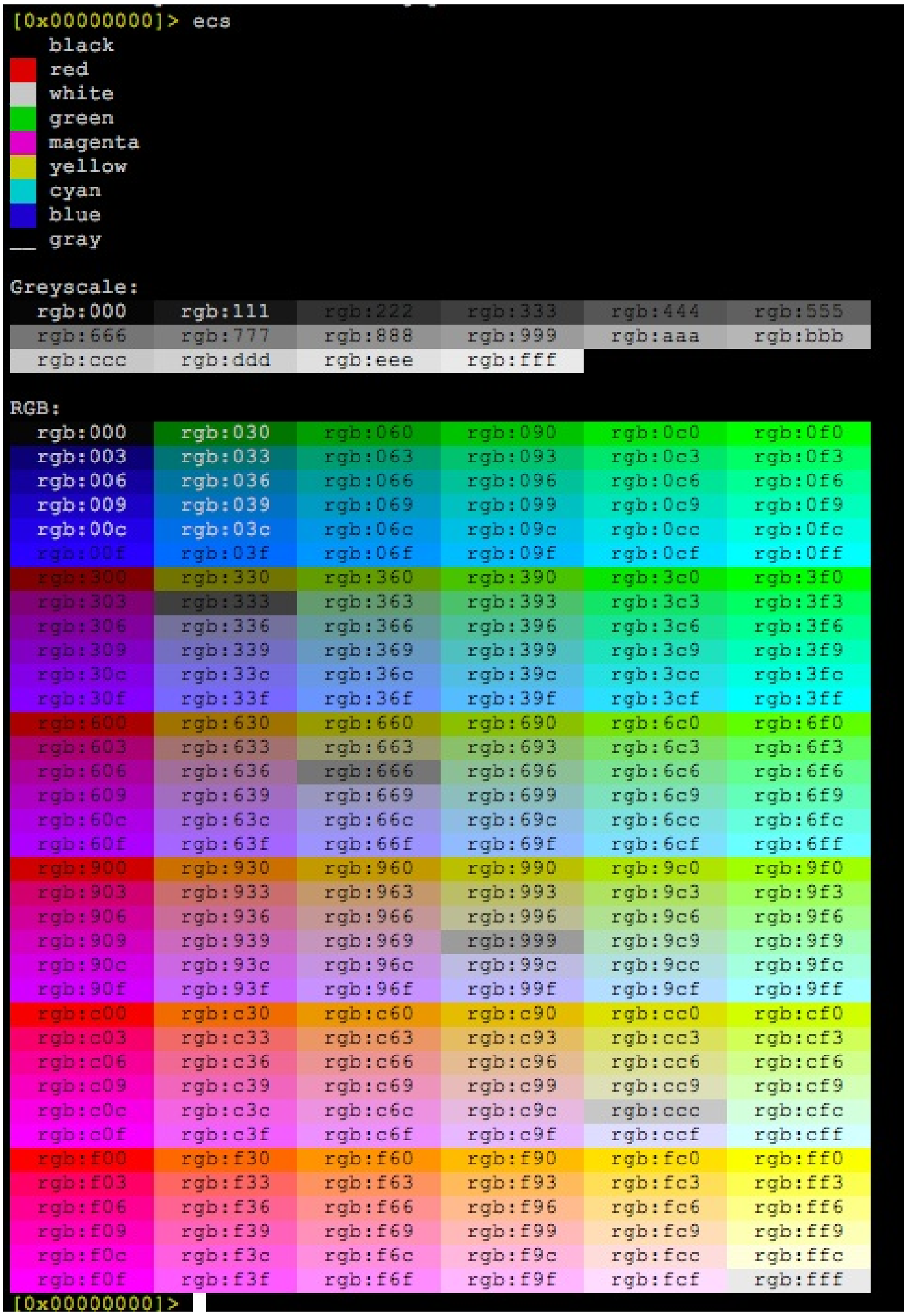
```
[0x00000000]> e scr.nkey =?
scr.nkey = fun, hit, flag
```

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:

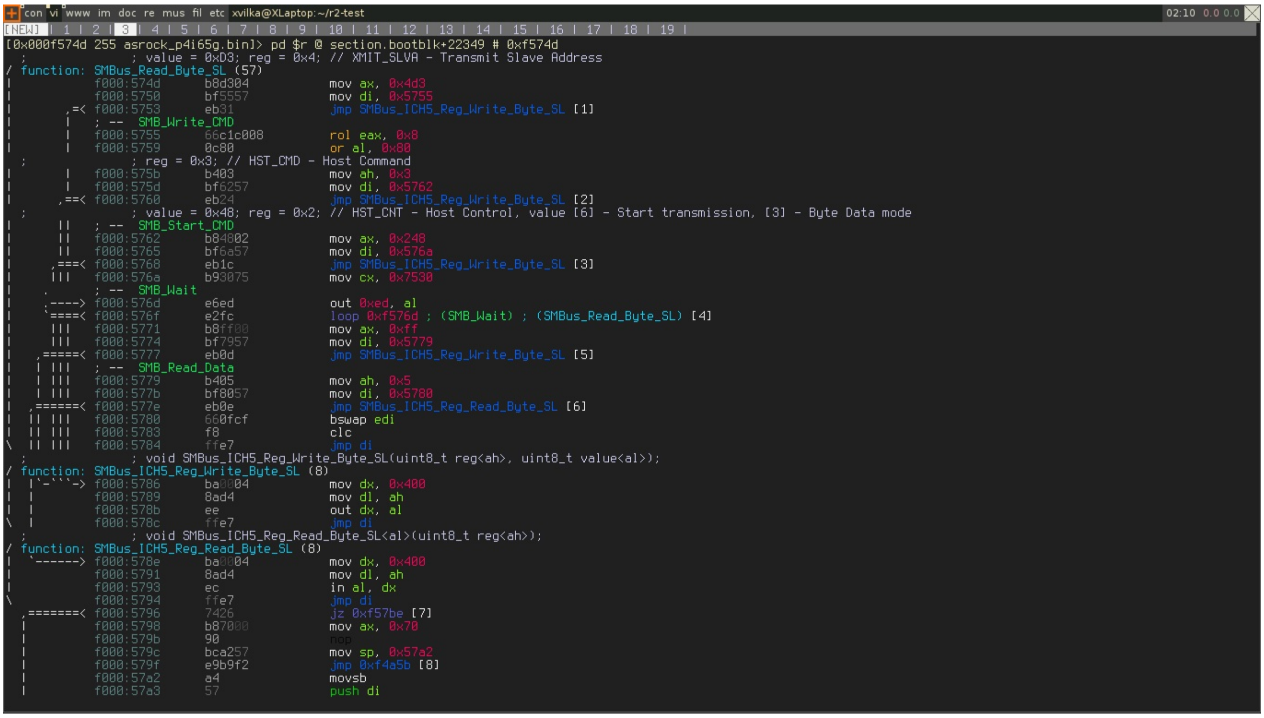
```
$ echo 'e scr.color=true' >> ~/.radare2rc
```

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). You can find the list of possible value by looking at the result of `e asm.arch=?` or `rasm2 -L` . 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", disassembler view will have flags column.

```
asm.linescall
```

If set to "true", draw lines at the left of disassemble 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 disassembled 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.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

The 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.

Open file

As test file lets use some simple hello_world.c compiled in Linux ELF format. After we compiled it lets open it with radare2

```
r2 hello_world
```

Now we have command prompt

```
[0x00400410]>
```

Now we are ready to go deeper.

Seeking at any position

All seeking commands that have address in command parameters can use any base such as hex/octal/binary or decimal.

Seek to address 0x0, alternative command is just `0x0`

```
[0x00400410]> s 0x0
[0x00000000]>
```

Print current address

```
[0x00000000]> s
0x0
[0x00000000]>
```

there is an alternate way to print current position: `?v $$`.

Seek N positions forward, space is optional

```
[0x00000000]> s+ 128
[0x00000080]>
```

Undo last two seeks to return to the initial address

```
[0x00000080]> s-
[0x00000000]> s-
[0x00400410]>
```

we are back at *0x00400410*.

Let's search in the hello_world ELF file 'Hello'. After the search our position will be set at the position of the found string. Remember we can always go back with `s-`.

```
[0x00400410]> s/ Hello
Searching 5 bytes from 0x00400411 to 0x00600928: 48 65 6c 6c 6f
Searching 5 bytes in [0x400411-0x600928]
hits: 1  hit0_0 .. hit0_0
0x004005b4 hit0_0 "Hello"
[0x004005b4]>s-
[0x00400410]>
```

There's also a command for showing the seek history:

```
[0x00400410]> s*
f undo_3 @ 0x400410
f undo_2 @ 0x40041a
f undo_1 @ 0x400410
f undo_0 @ 0x400411
# Current undo/redo position.
f redo_0 @ 0x4005b4
```

Block Size

The block size determines how many bytes Radare commands will process when not given an explicit size argument. 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 you to map contents of files into the same IO space used to contain loaded binary. New contents can be placed at random offsets. This lets you create a static environment which emulate the view you would have when 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                 prioritize 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
```

Prepare a simple layout:

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

    4 libraries
```

Map a file:

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

List mapped files:

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

Print hexadecimal values from /bin/zsh:

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

Unmap files using the `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
 other countries %d/%m/%y is rather common. This means that in international context t
his format is ambiguous and should not be used.) (SU)
%e  Like %d, the day of the month as a decimal number, but a leading zero is replaced
by a space. (SU)
%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-
digit year corresponding to the ISO week number (see %V).  This has the same format an
d value as %Y, except that if the ISO week number belongs to the previous or next year
, that year is used instead. (TZ)
%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 pr
eceded by a blank.  (See also %H.)  (TZ)
%l  The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are pr
eceded by a blank.  (See also %I.)  (TZ)
%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 string
s for the current locale.  Noon is treated as "PM" and midnight as "AM".
%P  Like %p but in lowercase: "am" or "pm" or a corresponding string for the current l
ocale. (GNU)
%r  The time in a.m. or p.m. notation.  In the POSIX locale this is equivalent to %I:%
M:%S %p.  (SU)
%R  The time in 24-hour notation (%H:%M).  (SU) For a version including the seconds, s
ee %T below.
%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 occasional leap seconds.)
%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
with the first Sunday as the first day of week 01.  See also %V and %W.
%V  The ISO 8601 week number (see NOTES) of the current year as a decimal number, rang
e 01 to 53, where week 1 is the first week that has at least 4 days in the new year.
See also %U and %W.(U)
%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
with the first Monday as the first day of week 01.
%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)
. (SU)
%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, 0x
81, 0xc3, 0xd6, 0xa7, 0x01, 0x00, 0x8b, 0x83, 0x00, 0xff, 0xff, 0xff, 0x5a, 0x8d, 0x24
, 0x84, 0x29, 0xc2 };


[0x7fcd6a891630]> pcs
"\x48\x89\xe7\xe8\x68\x39\x00\x00\x49\x89\xc4\x8b\x05\xef\x16\x22\x00\x5a\x48\x8d\x24\
xc4\x29\xc2\x52\x48\x89\xd6\x49\x89\xe5\x48\x83\xe4\xf0\x48\x8b\x3d\x06\x1a
```

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 escaped 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 ) = 0xffffffda
[0x4A13B8C0]> dr
  eax  0xffffffda    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 achieve 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/libgstflumms.so
[0x000028A0]> seek sym.gst_plugin_desc
[0x000185E0]> pf iissxsssss 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 t
he name in disassembly
           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 conceptually similar to bookmarks. They associate a name with a given offset in a file. Flags can be grouped into 'flag spaces'. A flag space is a namespace for flags, grouping together flags of similar characteristics 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 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 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 be give as a widestring, as assembler instructions, or the data may be read in from another file.

Resize the file using the `r` command. It accepts a numeric argument. A positive value sets a new size for the file. A negative one will truncate the file to the current seek position minus N bytes.

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

Write bytes using the `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, each combines the existing data with the new data using an operator. The command is applied to the current block. Supported operators include: 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 by
tes
| 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:

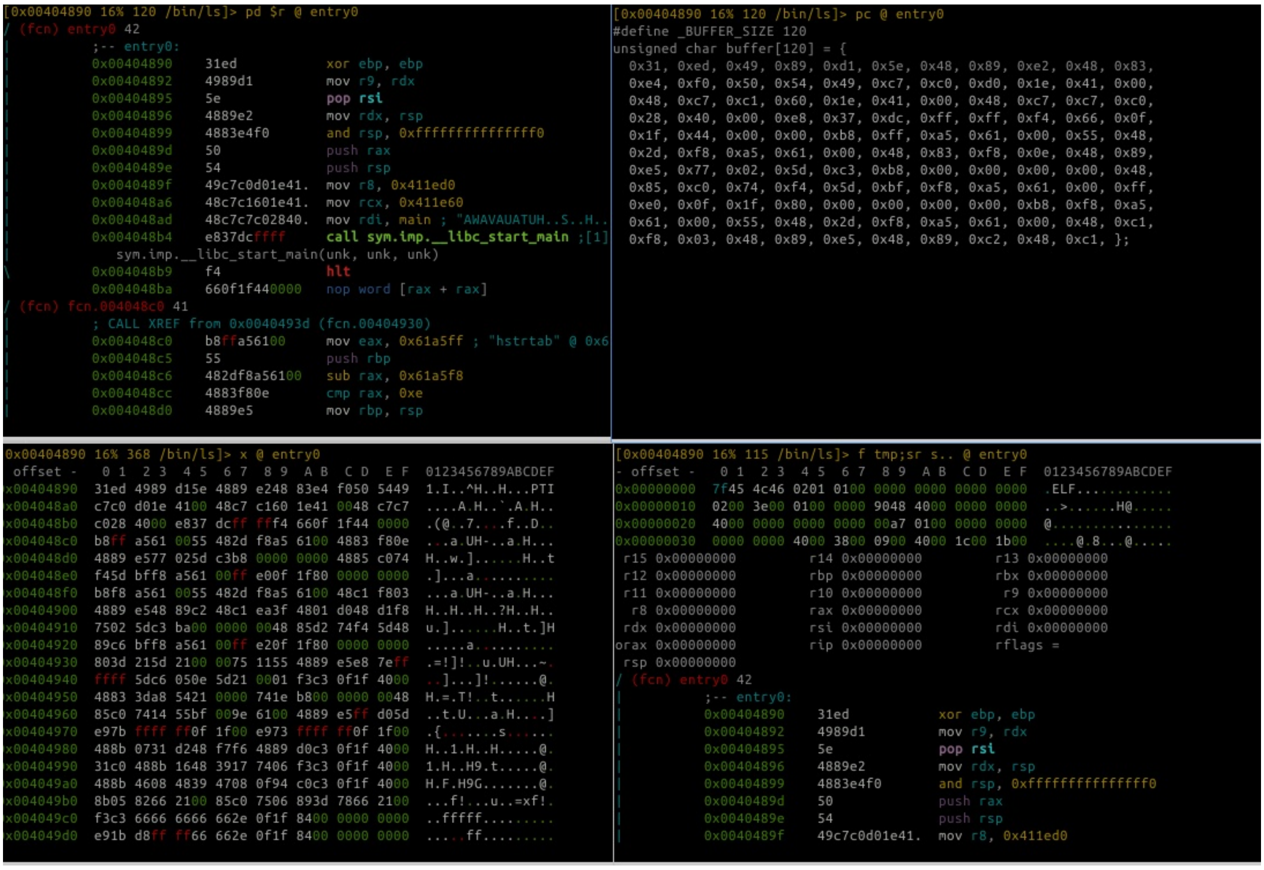
```
[0x00000000]> cx 7f469046

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

You can use the 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`.



print modes aka Panels

The Visual mode uses "print modes" which are basically different panel that you can rotate. By default those are:

↻ Hexdump panel -> Disassembly panel → Debugger panel → Hexadecimal words dump panel → Output C format panel → Op analysis color map panel → Annotated hexdump panel ↺.

Notice that the top of the panel contains the command which is used, for example for the disassembly panel:

```
[0x00404890 16% 120 /bin/ls]> pd $r @ entry0
```

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 Disassembly

Navigation

Move within the Disassembly using `arrows` or `hjkl`. Use `o` to seek directly to a flag or an offset, type it when requested by the prompt: `[offset]>`. Follow a jump or a call using the `number` of your keyboard `[0-9]` and the number on the right in disassembly to follow a call or a jump. In this example typing `1` on the keyboard would follow the call to `sym.imp.__libc_start_main` and therefore, seek at the offset of this symbol.

```
            0x00404894      e857dcffff     call sym.imp.__libc_start_main ;[1]
```

Seek back to the previous location using `u`, `U` will allow you to redo the seek.

`d` as define

`d` can be used to change the type of data of the current block, several basic types/structures are available as well as more advanced one using `pf` template:

```
d → ...
0x004048f7      48c1e83f       shr rax, 0x3f

d → b
0x004048f7 .byte 0x48
d → B
0x004048f7 .word 0xc148

d → d
0x004048f7 hex length=165 delta=0

0x004048f7  48c1 e83f 4801 c648 d1fe 7415 b800 0000  H..?H..H..t.....

...
```

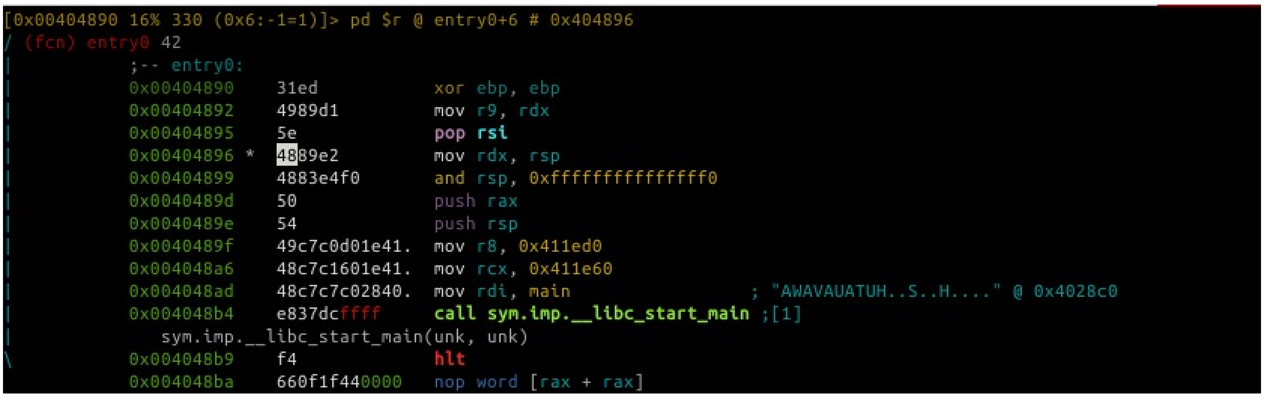
To improve code readability you can change how radare2 presents numerical values in disassembly, by default most of disassembly display numerical value as hexadecimal. Sometimes you would like to view it as a decimal, binary or even custom defined constant. To change value format you can use `d` following by `i` then choose what base to work in, this is the equivalent to `ahi` :

```
d → i → ...
0x004048f7      48c1e83f       shr rax, 0x3f

d → i →  10
0x004048f7      48c1e83f       shr rax, 63

d → i →  2
0x004048f7      48c1e83f       shr rax, '?'
```

Usage of the Cursor for Inserting/Patching...

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

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



The cursor is used to select a range of bytes or simply to point to a byte. You can use the cursor to create a named flag at specifc location. To do so, seek to the required position, then press `f` and enter a name for a flag. If the file was opened in write mode using the `-w` flag or the `o+` command, you can also use the cursor to overwrite a selected range with new values. To do so, select a range of bytes (with HJKL and SHIFT key pressed), then press `i` and enter the hexpair values for the new data. The data will be repeated as needed to fill the range selected. For example:

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

The 10 bytes you have selected will be changed to "12 34 12 34 12 ...".

The Visual Assembler is a feature that provides a live-preview while you type in new opcodes to patch into the disassembly. To use it, seek or place the cursor at the wanted location and hit the 'A' key. To provide multiple opcodes, seperate them with a semicolon, `;`.

XREF

When radare2 has discovered a XREF during the analysis, it will show you the information in the Visual Disassembly using `XREF` tag:

```
             ; DATA XREF from 0x00402e0e (unk)
             str.David_MacKenzie:
```

To see where this string is called press `x`, if you want to jump to the location where the data is used then press the corresponding number [0-9] on your keyboard. (This functionality is similar to `axt`)

`X` corresponds to the reverse operation aka `axf`.

Add a comment

To add a comment press `;`.

Type other commands

Quickly type commands using `:`.

Search

`/`: allows highlighting of strings in the current display. `:cmd` allows you to use one of the "/?" commands that perform more specialized searches.

The HUDS

The "UserFriendly HUD"

The "UserFriendly HUD" can be accessed using the `??` key-combination. This HUD acts as an interactive Cheat Sheet that one can use to more easily find and execute commands. This HUD is particularly useful for new-comers. For experienced users, the other HUDS which are more activity-specific may be more useful.

The "flag/comment/functions/.. HUD"

This HUD can be displayed using the `_` key, it shows a list of all the flags defined and lets you jump to them. Using the keyboard you can quickly filter the list down to a flag that contains a specific pattern.

Tweaking the Disassembly

The disassembly's look-and-feel is controlled using the "asm.* configuration keys, which can be changed using the `e` command. All configuration keys can also be edited through the Visual Configuration Editor.

Visual Configuration Editor

This HUD can be accessed using the `e` key in visual mode. The editor allows you to easily examine and change radare2's configuration. For example, if you want to change something about the disassembly display, select `asm` from the list, navigate to the item you wish to modify it, then select it by hitting `Enter`. If the item is a boolean variable, it will toggle, otherwise you will be prompted to provide a new value.

```
[EvalSpace]

    anal
 >  asm
    bin
    cfg
    cmd
    dbg
    diff
    dir
    esil
    file
    fs
    graph
    hex
    http
    hud
    io
    key
    magic
    pdb
    rap
    rop
    scr
    search
    stack
    time
    zoom

 Sel:asm.arch

/ (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
```

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
```

Following are some example of eval variable related to disassembly.

Examples

asm.arch: Change Architecture && asm.bits: Word size in bits at assembler

You can view the list of all arch using `e asm.arch=?`

```
e asm.arch = dalvik
            0x00404870      31ed4989       cmp-long v237, v73, v137

            0x00404874      d15e4889       rsub-int v14, v5, 0x8948
            0x00404878      e24883e4       ushr-int/lit8 v72, v131, 0xe4
            0x0040487c      f0505449c7c0   +invoke-object-init-range {}, method+18772
;[0]
            0x00404882      90244100       add-int v36, v65, v0
```

```
e asm.bits = 16
            0000:4870      31ed           xor bp, bp
            0000:4872      49             dec cx
            0000:4873      89d1           mov cx, dx
            0000:4875      5e             pop si
            0000:4876      48             dec ax
            0000:4877      89e2           mov dx, sp
```

This latest operation can also be done using `&` in Visual mode.

asm.pseudo: Enable pseudo syntax

```
e asm.pseudo = true
            0x00404870      31ed           ebp = 0
            0x00404872      4989d1         r9 = rdx
            0x00404875      5e             pop rsi
            0x00404876      4889e2         rdx = rsp
            0x00404879      4883e4f0       rsp &= 0xfffffffffffffff0
```

asm.syntax: Select assembly syntax (intel, att, masm...)

```
e asm.syntax = att
            0x00404870      31ed           xor %ebp, %ebp
            0x00404872      4989d1         mov %rdx, %r9
            0x00404875      5e             pop %rsi
            0x00404876      4889e2         mov %rsp, %rdx
            0x00404879      4883e4f0       and $0xfffffffffffffff0, %rsp
```

asm.describe: Show opcode description

```
e asm.describe = true
            0x00404870      31ed           xor ebp, ebp                 ; logical exclu
sive or
            0x00404872      4989d1         mov r9, rdx                  ; moves data fr
om src to dst
            0x00404875      5e             pop rsi                      ; pops last ele
ment of stack and stores the result in argument
            0x00404876      4889e2         mov rdx, rsp                 ; moves data fr
om src to dst
            0x00404879      4883e4f0       and rsp, 0xfffffffffffffff0 ; binary and op
eration between src and dst, stores result on dst
```

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, and hexadecimal values. It automatically creates flags for search hit locations 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 can be seen from the output above, radare2 generates a "hit" flag for every entry found. You can then use the `ps` command to see the strings stored at the offsets marked by the flags in this group, athey ll haves names of the form `hit0_<index>` :

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

You can search for wide-char strings (e.g., unicode letters) using the `/w` command:

```
[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 are searching for a string of hexadecimal values, you're probably better of using the `/x` command:

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

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

```
[0x00000000]> fs
0    0 . strings
1    0 . symbols
2    6 . searches

[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. You can use the `//` command to repeat the last search.

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

Configuring Search Options

The radare2 search engine can be configured through several configuration variables, modifiable with the `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, sectio
n
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 Automation

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, use the `/b` command.

Assembler Search

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

The command `/c jmp [esp]` searches for the specified asm mnemonic:

```
[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]
```

The command `/a jmp eax` assembles a string to machine code, and then searches for the 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 has many options, including target architecture flavor and disassembler variants, among other things.

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

```
[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 the number of input bytes as its argument, instead of the number of opcodes.

The "pseudo" syntax may be somewhat easier for a human to understand than the 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 Disassembly

The typical work involved in reversing binary files makes powerful annotation capabailities essential. Radare offers multiple ways to store and retrieve such metadata.

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 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 the 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 as either code, binary data or string. It is also possible to execute external code at every specified flag location in order to fetch some metadata, such as 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?` family of commands lets you mark a range as one of several kinds of types. Three basic types are: code (disassembly is done using asm.arch), data (an array of data elements) or string. Use the `Cs` comand to define a string, use the `Cd` command for defining an array of data elements, and use the `Cf` command to define more complex data structures like structs.

Annotating data types is most easily done in visual mode, using the "d" key, short for "data type change". To First, use the cursor to select a range of bytes (press `c` key to toggle cursor mode and use HJKL keys to expand selection), then press 'd' to get a menu of possible actions/types. For example, to mark the range as a string, use the 's' option from the menu. You can achieve the same result from the shell using the `Cs` command:

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

The `Cf` 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

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.isOperator()) {
    esilOperators[word](esil);
  } else {
    esil.push (word);
  }
  nextCommand();
}
```

As we can see ESIL uses a stack-based interpreter similar to what is commonly used for calculators. You have two categories of inputs: values and operators. A value simply gets pushed on the stack, an operator then pops values (its arguments if you will) off the stack, performs its operation and pushes its results (if any) back on. We can think of ESIL as a post-fix notation of the operations we want to do.

So let's see an example:

```
4,esp,-=,ebp,esp,=[4]
```

Can you guess what this is? If we take this post-fix notation and transform it back to in-fix we get

```
esp -= 4
4bytes(dword) [esp] = ebp
```

We can see that this corresponds to the x86 instruction `push ebp`! Isn't that cool? The aim is to be able to express most of the common operations performed by CPUs, like binary arithmetic operations, memory loads and stores, processing syscalls etc. This way if we can transform the instructions to ESIL we can see what a program does while it is running even for the most cryptic architectures you definitely don't have a device to debug on for.

Use ESIL

Using visual mode is great to inspect the esil evaluations.

There are 2 environment variables that are important for watching what a program does:

```
[0x00000000]> e asm.emu = true
[0x00000000]> e asm.emustr = true
```

"asm.emu" tells r2 if you want ESIL information to be displayed. If it is set to true you will see comments appear to the right of your disassembly that tell you how the contents of registers and memory addresses are changed by the current instruction. For example if you have an instruction that subtracts a value from a register it tells you what the value was before and what it becomes after. This is super useful so you don't have to sit there yourself and track which value goes where.

One problem with this is that it is a lot of information to take in at once and sometimes you simply don't need it. r2 has a nice compromise for this. That is what the "asm.emustr" variable is for. Instead of this super verbose output with every register value, this only adds really useful information to the output, e.g., strings that are found at addresses a program uses or whether a jump is likely to be taken or not.

The third important variable is "asm.esil". This switches your disassembly to no longer show you the actual disassembled instructions, but instead now shows you corresponding ESIL expressions that describe what the instruction does. So if you want to take a look at how instructions are expressed in ESIL simply set "asm.esil" to true.

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

In visual mode you can also toggle this by simply typing `O`.

ESIL Commands

- "ae" : Evaluate ESIL expression.

  ```
  [0x00000000]> "ae 1,1,+"
  0x2
  [0x00000000]>
  ```

- "aes" : ESIL Step.

  ```
  [0x00000000]> aes
  [0x00000000]>10aes
  ```

- "aeso" : ESIL Step Over.

```
[0x00000000]> aeso
[0x00000000]>10aeso
```

- "aesu" : ESIL Step Until.

```
[0x00001000]> aesu 0x1035
ADDR BREAK
[0x00001019]>
```

- "ar" : Show/modify ESIL registry

```
[0x00001ec7]> ar r_00 = 0x1035
[0x00001ec7]> ar r_00
0x00001035
[0x00001019]>
```

ESIL Instruction Set

Here is the complete instruction set used by the ESIL VM:

ESIL Opcode	Operands	Name	Operation	example
TRAP	src	Trap	Trap signal	
$	src	Syscall	syscall	
$$	src	Instruction address	Get address of current instruction stack=instruction address	
==	src,dst	Compare	stack = (dst == src) ; update_eflags(dst - src)	
<	src,dst	Smaller (signed comparison)	stack = (dst < src) ; update_eflags(dst - src)	[0x0000000]> "ae 1,5, <" 0x0 [0x00000000]> "ae 5,5" 0x0"
		Smaller or Equal	stack = (dst <= src) ;	[0x0000000]> "ae 1,5, <" 0x0

		(signed comparison)	update_eflags(dst - src)	[0x00000000]> "ae 5,5" 0x1"
>	src,dst	Bigger (signed comparison)	stack = (dst > src) ; update_eflags(dst - src)	[0x00000000]> "ae 1,5,>" 0x1 [0x00000000]> "ae 5,5,>" 0x0
>=	src,dst	Bigger or Equal (signed comparison)	stack = (dst >= src) ; update_eflags(dst - src)	[0x00000000]> "ae 1,5,>=" 0x1 [0x00000000]> "ae 5,5,>=" 0x1
<<	src,dst	Shift Left	stack = dst << src	[0x00000000]> "ae 1,1,<<" 0x2 [0x00000000]> "ae 2,1,<<" 0x4
>>	src,dst	Shift Right	stack = dst >> src	[0x00000000]> "ae 1,4,>>" 0x2 [0x00000000]> "ae 2,4,>>" 0x1
<<<	src,dst	Rotate Left	stack=dst ROL src	[0x00000000]> "ae 31,1,<<<" 0x80000000 [0x00000000]> "ae 32,1,<<<" 0x1
>>>	src,dst	Rotate Right	stack=dst ROR src	[0x00000000]> "ae 1,1,>>>" 0x80000000 [0x00000000]> "ae 32,1,>>>" 0x1
&	src,dst	AND	stack = dst & src	[0x00000000]> "ae 1,1,&" 0x1 [0x00000000]> "ae 1,0,&" 0x0 [0x00000000]> "ae 0,1,&" 0x0

				0x0 [0x00000000]> "ae 0,0,&" 0x0
**`	`**	src,dst	OR	stack = dst `
^	src,dst	XOR	stack = dst ^src	[0x00000000]> "ae 1,1,^" 0x0 [0x00000000]> "ae 1,0,^" 0x1 [0x00000000]> "ae 0,1,^" 0x1 [0x00000000]> "ae 0,0,^" 0x0
+	src,dst	ADD	stack = dst + src	[0x00000000]> "ae 3,4,+" 0x7 [0x00000000]> "ae 5,5,+" 0xa
-	src,dst	SUB	stack = dst - src	[0x00000000]> "ae 3,4,-" 0x1 [0x00000000]> "ae 5,5,-" 0x0 [0x00000000]> "ae 4,3,-" 0xffffffffffffffff
*	src,dst	MUL	stack = dst * src	[0x00000000]> "ae 3,4, * " 0xc [0x00000000]> "ae 5,5, * " 0x19
/	src,dst	DIV	stack = dst / src	[0x00000000]> "ae 2,4,/" 0x2 [0x00000000]> "ae 5,5,/" 0x1 [0x00000000]> "ae 5,9,/" 0x1

%	src,dst	MOD	stack = dst % src	2,4,%" 0x0 [0x00000000]> "ae 5,5,%" 0x0 [0x00000000]> "ae 5,9,%" 0x4
!	src	NEG	stack = !!!src	[0x00000000]> "ae 1,!" 0x0 [0x00000000]> "ae 4,!" 0x0 [0x00000000]> "ae 0,!" 0x1
++	src	INC	stack = src++	[0x00000000]> ar r_00=0;ar r_00 0x00000000 [0x00000000]> "ae r_00,++" 0x1 [0x00000000]> ar r_00 0x00000000 [0x00000000]> "ae 1,++" 0x2
--	src	DEC	stack = src--	[0x00000000]> ar r_00=5;ar r_00 0x00000005 [0x00000000]> "ae r_00,--" 0x4 [0x00000000]> ar r_00 0x00000005 [0x00000000]> "ae 5,--" 0x4
+=	src,reg	ADD eq	reg = reg + src	[0x00000000]> ar r_01=5;ar r_00=0;ar r_00 0x00000000 [0x00000000]> "ae r_01,r_00,+=" [0x00000000]> ar r_00 0x00000005 [0x00000000]> "ae 5,r_00,+=" [0x00000000]> ar r_00 0x0000000a
				[0x00000000]> "ae

-=	src,reg	SUB eq	reg = reg - src	r_01,r_00,-=" [0x00000000]> ar r_00 0x00000004 [0x00000000]> "ae 3,r_00,-=" [0x00000000]> ar r_00 0x00000001
*=	src,reg	MUL eq	reg = reg * src	[0x00000000]> ar r_01=3;ar r_00=5;ar r_00 0x00000005 [0x00000000]> "ae r_01,r_00, * =" [0x00000000]> ar r_00 0x0000000f [0x00000000]> "ae 2,r_00, * =" [0x00000000]> ar r_00 0x0000001e
/=	src,reg	DIV eq	reg = reg / src	[0x00000000]> ar r_01=3;ar r_00=6;ar r_00 0x00000006 [0x00000000]> "ae r_01,r_00,/=" [0x00000000]> ar r_00 0x00000002 [0x00000000]> "ae 1,r_00,/=" [0x00000000]> ar r_00 0x00000002
%=	src,reg	MOD eq	reg = reg % src	[0x00000000]> ar r_01=3;ar r_00=7;ar r_00 0x00000007 [0x00000000]> "ae r_01,r_00,%=" [0x00000000]> ar r_00 0x00000001 [0x00000000]> ar r_00=9;ar r_00 0x00000009 [0x00000000]> "ae 5,r_00,%=" [0x00000000]> ar r_00 0x00000004
				[0x00000000]> ar r_00=1;ar r_01=1;ar r_01

<<=	src,reg	Shift Left eq	reg = reg << src	[0x00000000]> "ae r_00,r_01,<<=" [0x00000000]> ar r_01 0x00000002 [0x00000000]> "ae 2,r_01,<<=" [0x00000000]> ar r_01 0x00000008
>>=	src,reg	Shift Right eq	reg = reg << src	[0x00000000]> ar r_00=1;ar r_01=8;ar r_01 0x00000008 [0x00000000]> "ae r_00,r_01,>>=" [0x00000000]> ar r_01 0x00000004 [0x00000000]> "ae 2,r_01,>>=" [0x00000000]> ar r_01 0x00000001
&=	src,reg	AND eq	reg = reg & src	[0x00000000]> ar r_00=2;ar r_01=6;ar r_01 0x00000006 [0x00000000]> "ae r_00,r_01,&=" [0x00000000]> ar r_01 0x00000002 [0x00000000]> "ae 2,r_01,&=" [0x00000000]> ar r_01 0x00000002 [0x00000000]> "ae 1,r_01,&=" [0x00000000]> ar r_01 0x00000000
**`	`=**	src,reg	OR eq	reg = reg `
^=	src,reg	XOR eq	reg = reg ^ src	[0x00000000]> ar r_00=2;ar r_01=0xab;ar r_01 0x000000ab [0x00000000]> "ae r_00,r_01,^=" [0x00000000]> ar r_01 0x000000a9

				0x000000a9 [0x00000000]> "ae 2,r_01,^=" [0x00000000]> ar r_01 0x000000ab
++=	reg	INC eq	reg = reg + 1	[0x00000000]> ar r_00=4;ar r_00 0x00000004 [0x00000000]> "ae r_00,++=" [0x00000000]> ar r_00 0x00000005
--=	reg	DEC eq	reg = reg - 1	[0x00000000]> ar r_00=4;ar r_00 0x00000004 [0x00000000]> "ae r_00,--=" [0x00000000]> ar r_00 0x00000003
!=	reg	NOT eq	reg = !reg	[0x00000000]> ar r_00=4;ar r_00 0x00000004 [0x00000000]> "ae r_00,!=" [0x00000000]> ar r_00 0x00000000 [0x00000000]> "ae r_00,!=" [0x00000000]> ar r_00 0x00000001
---	---	---	---	--
=[] =[*] =[1] =[2] =[4] =[8]	src,dst	poke	*dst=src	[0x00010000]> "ae 0xdeadbeef,0x10000,=[4]," [0x00010000]> pxw 4@0x10000 0x00010000 0xdeadbeef .... [0x00010000]> "ae 0x0,0x10000,=[4]," [0x00010000]> pxw 4@0x10000 0x00010000 0x00000000
[] [*]				[0x00010000]> w test@0x10000 [0x00010000]> "ae 0x10000,[4],"

[2] [4] [8]	src	peek	stack=*src	[0x00010000]> ar r_00=0x10000 [0x00010000]> "ae r_00,[4]," 0x74736574
`	=[] 	=[1] 	=[2] 	=[4]
SWAP		Swap	Swap two top elements	SWAP
PICK	n	Pick	Pick nth element from the top of the stack	2,PICK
RPICK	m	Reverse Pick	Pick nth element from the base of the stack	0,RPICK
DUP		Duplicate	Duplicate top element in stack	DUP
NUM		Numeric	If top element is a reference (register name, label, etc), dereference it and push its real value	NUM
CLEAR		Clear	Clear stack	CLEAR
BREAK		Break	Stops ESIL emulation	BREAK
GOTO	n	Goto	Jumps to Nth ESIL word	GOTO 5
TODO		To Do	Stops execution (reason: ESIL expression not completed)	TODO

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 bor
row from bit 4)
c      - carry, same like above (example: $c7 - checks if carry from bit 7)
p      - parity
r      - regsize ( asm.bits/8 )
[0-9]* - Used to set flags and registers without having any side effects,
         i.e. setting esil_cur, esil_old and esil_lastsz.
         (example: "$0,of,=" to reset the overflow flag)
```

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 `?` operator uses the value of its argument to decide whether to evaluate the expression in curly braces.

1. Is the value zero? -> Skip it.
2. Is the value non-zero? -> Evaluate it.

```
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 `<code>,TRAP` 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 ex: $z
indexOf("$")       ->    syscall ex: 1,$
indexOf("TRAP")    ->    can trap
indexOf('++')      ->    has iterator
indexOf('--')      ->    count to zero
indexOf("?{")      ->    conditional
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.

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,--,0,GOTO

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,+=
        ┌─< 0x100001105    e950350000   0x465a,rip,= ;[1]
        │   0x10000110a    55           8,rsp,-=,rbp,rsp,=[8]
        │   0x10000110b    4889e5       rsp,rbp,=
        │   0x10000110e    488d4668     rsi,104,+,rax,=
        │   0x100001112    488d7768     rdi,104,+,rsi,=
        │   0x100001116    4889c7       rax,rdi,=
        │   0x100001119    5d           rsp,[8],rbp,=,8,rsp,+=
       ┌──< 0x10000111a    e93b350000   0x465a,rip,= ;[1]
       ││   0x10000111f    55           8,rsp,-=,rbp,rsp,=[8]
       ││   0x100001120    4889e5       rsp,rbp,=
       ││   0x100001123    488b4f60     rdi,96,+,[8],rcx,=
       ││   0x100001127    4c8b4130     rcx,48,+,[8],r8,=
       ││   0x10000112b    488b5660     rsi,96,+,[8],rdx,=
       ││   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
    ┌─────< 0x100001141    eb11         0x1154,rip,= ;[2]
    │└────> 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,=

Radare2 provides a wide set of a features to automate boring work. It ranges from the simple sequencing of the commands to the calling scripts/another programs via IPC (Inter-Process Communication), called r2pipe.

As mentioned a few times before there is an ability to sequence commands using `;` semicolon operator.

```
[0x00404800]> pd 1 ; ao 1
           0x00404800      b827e66100     mov eax, 0x61e627      ; "tab"
address: 0x404800
opcode: mov eax, 0x61e627
prefix: 0
bytes: b827e66100
ptr: 0x0061e627
refptr: 0
size: 5
type: mov
esil: 6415911,rax,=
stack: null
family: cpu
[0x00404800]>
```

It simply runs the second command after finishing the first one, like in a shell.

The second important way to sequence the commands is with a simple pipe `|`

```
ao|grep address
```

Note, the `|` pipe only can pipe output of r2 commands to external (shell) commands, like system programs or builtin shell commands. There is a similar way to sequence r2 commands, using the backtick operator ```, which works in the same way it does in a shell.

For example, we want to see a few bytes of the memory at the address referred to by the 'mov eax, addr' instruction. We can do that without jumping to it, using a sequence of commands:

```
[0x00404800]> pd 1
              0x00404800      b827e66100     mov eax, 0x61e627      ; "tab"
[0x00404800]> ao
address: 0x404800
opcode: mov eax, 0x61e627
prefix: 0
bytes: b827e66100
ptr: 0x0061e627
refptr: 0
size: 5
type: mov
esil: 6415911,rax,=
stack: null
family: cpu
[0x00404800]> ao~ptr[1]
0x0061e627
0
[0x00404800]> px 10 @ `ao~ptr[1]`
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x0061e627  7461 6200 2e69 6e74 6572                 tab..inter
[0x00404800]>
```

And of course it's possible to redirect the output of an r2 command into a file, using the `>` and `>>` commands

```
[0x00404800]> px 10 @ `ao~ptr[1]` > example.txt
[0x00404800]> px 10 @ `ao~ptr[1]` >> example.txt
```

The `?$?` command describes several helpful variables you can use to do similar actions even more easily, like the `$v` "immediate value" variable, or the `$m` opcode memory reference variable.

One of the most common task in automation is looping through something, there are multiple ways to do this in radare2.

We can loop over flags:

```
@@ flagname-regex
```

For example, we want to see function information with `afi` command:

```
[0x004047d6]> afi
#
offset: 0x004047d0
name: entry0
size: 42
realsz: 42
stackframe: 0
call-convention: amd64
cyclomatic-complexity: 1
bits: 64
type: fcn [NEW]
num-bbs: 1
edges: 0
end-bbs: 1
call-refs: 0x00402450 C
data-refs: 0x004136c0 0x00413660 0x004027e0
code-xrefs:
data-xrefs:
locals:0
args: 0
diff: type: new
[0x004047d6]>
```

Now let's say, for example, that we'd like see a particular field from this output for all functions found by analysis. We can do that with a loop over all function flags (whose names begin with `fcn.`):

```
[0x004047d6]> afi @@ fcn.* ~name
```

This command will extract the `name` field from the `afi` output of every flag with a name matching the regexp `fcn.*` .

We can also loop over a list of offsets, using the following syntax:

```
@@=1 2 3 ... N
```

For example, say we want to see the opcode information for 2 offsets: the current one, and at current + 2:

```
[0x004047d6]> ao @@=$$ $$+2
address: 0x4047d6
opcode: mov rdx, rsp
prefix: 0
bytes: 4889e2
refptr: 0
size: 3
type: mov
esil: rsp,rdx,=
stack: null
family: cpu
address: 0x4047d8
opcode: loop 0x404822
prefix: 0
bytes: e248
refptr: 0
size: 2
type: cjmp
esil: 1,rcx,-=,rcx,?{,4212770,rip,=,}
jump: 0x00404822
fail: 0x004047da
stack: null
cond: al
family: cpu
[0x004047d6]>
```

Note we're using the `$$` variable which evaluates to the current offset. Also note that `$$+2` is evaluated before looping, so we can use the simple arithmetic expressions.

A third way to loop is by having the offsets be loaded from a file. This file should contain one offset per line.

```
[0x004047d0]> ?v $$ > offsets.txt
[0x004047d0]> ?v $$+2 >> offsets.txt
[0x004047d0]> !cat offsets.txt
4047d0
4047d2
[0x004047d0]> pi 1 @@.offsets.txt
xor ebp, ebp
mov r9, rdx
```

radare2 also offers various `foreach` constructs for looping. One of the most useful is for looping through all the instructions of a function:

```
[0x004047d0]> pdf
╒ (fcn) entry0 42
│                ; UNKNOWN XREF from 0x00400018 (unk)
│                ; DATA XREF from 0x004064bf (sub.strlen_460)
│                ; DATA XREF from 0x00406511 (sub.strlen_460)
│                ; DATA XREF from 0x0040b080 (unk)
│                ; DATA XREF from 0x0040b0ef (unk)
│                0x004047d0      31ed           xor ebp, ebp
│                0x004047d2      4989d1         mov r9, rdx
│                0x004047d5      5e             pop rsi
│                0x004047d6      4889e2         mov rdx, rsp
│                0x004047d9      4883e4f0       and rsp, 0xfffffffffffffff0
│                0x004047dd      50             push rax
│                0x004047de      54             push rsp
│                0x004047df      49c7c0c03641.  mov r8, 0x4136c0
│                0x004047e6      48c7c1603641.  mov rcx, 0x413660      ; "AWA..AVI..AU
I..ATL.%.. "
0x00413660  ; "AWA..AVI..AUI..ATL.%.. "
│                0x004047ed      48c7c7e02740.  mov rdi, main          ; "AWAVAUATUH..
S..H...." @
0x4027e0
│                0x004047f4      e857dcffff     call sym.imp.__libc_start_main
╘                0x004047f9      f4             hlt
[0x004047d0]> pi 1 @@i
mov r9, rdx
pop rsi
mov rdx, rsp
and rsp, 0xfffffffffffffff0
push rax
push rsp
mov r8, 0x4136c0
mov rcx, 0x413660
mov rdi, main
call sym.imp.__libc_start_main
hlt
```

In this example the command `pi 1` runs over all the instructions in the current function (entry0).

The last kind of looping lets you loop through predefined iterator types:

- symbols
- imports
- registers
- threads
- comments
- functions
- flags

This is done using the `@@@` command. The previous example of listing information about functions can also be done using the `@@@` command:

```
[0x004047d6]> afi @@@ functions ~name
```

This will extract `name` field from `afi` output and will output a huge list of function names. We can choose only the second column, to remove the redundant `name:` on every line:

```
[0x004047d6]> afi @@@ functions ~name[1]
```

Apart from simple sequencing and looping, radare2 allows to write simple macros, using this construction:

```
[0x00404800]> (qwe, pd 4, ao)
```

This will define a macro called 'qwe' which runs sequentially first 'pd 4' then 'ao'. Calling the macro using syntax `.(macro)` is simple:

```
[0x00404800]> (qwe, pd 4, ao)
[0x00404800]> .(qwe)
    0x00404800      b827e66100     mov eax, 0x61e627      ; "tab"
    0x00404805      55             push rbp
    0x00404806      482d20e66100   sub rax, section_end.LOAD1
    0x0040480c      4889e5         mov rbp, rsp

address: 0x404800
opcode: mov eax, 0x61e627
prefix: 0
bytes: b827e66100
ptr: 0x0061e627
refptr: 0
size: 5
type: mov
esil: 6415911,rax,=
stack: null
family: cpu
[0x00404800]>
```

To list available macroses simply call `(*` :

```
[0x00404800]> (*
(qwe , pd 4, ao)
```

And if want to remove some macro, just add '-' before the name:

```
[0x00404800]> (-qwe)
Macro 'qwe' removed.
[0x00404800]>
```

Moreover, it's possible to create a macro that takes arguments, which comes in handy in some simple scripting situations. To create a macro that takes arguments you simply add them to macro definition. Be sure, if you're using characters like ';', to quote the whole command for proper parsing.

```
[0x00404800]
[0x004047d0]> "(foo x y,pd $0; s +$1)"
[0x004047d0]> .(foo 5 6)
        ;-- entry0:
        0x004047d0      31ed           xor ebp, ebp
        0x004047d2      4989d1         mov r9, rdx
        0x004047d5      5e                pop rsi
        0x004047d6        4889e2         mov rdx, rsp
        0x004047d9        4883e4f0       and rsp, 0xfffffffffffffff0
[0x004047d6]>
```

As you can see, the arguments are named by index, starting from 0: $0, $1, ...

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, the generated symbols list format is similar to the imports list. Use the `-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=stdo
ut
addr=0x0021a600 off=0x0021a600 ord=115 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=_eda
ta
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=__pr
ogname
addr=0x0021a630 off=0x0021a630 ord=119 fwd=NONE sz=8 bind=UNKNOWN type=OBJECT name=pro
gram_invocation_name
addr=0x0021a600 off=0x0021a600 ord=121 fwd=NONE sz=0 bind=GLOBAL type=NOTYPE name=__bs
s_start
addr=0x0021a630 off=0x0021a630 ord=122 fwd=NONE sz=8 bind=GLOBAL type=OBJECT name=__pr
ogname_full
addr=0x0021a600 off=0x0021a600 ord=123 fwd=NONE sz=8 bind=UNKNOWN type=OBJECT name=pro
gram_invocation_short_name
addr=0x00002178 off=0x00002178 ord=124 fwd=NONE sz=0 bind=GLOBAL type=FUNC name=_init
```

With the `-sr` option rabin2 produces a radare2 script instead. It can later be 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 the `-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 the 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 direct 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=sr
c/ls.c
addr=0x00012490 off=0x00012490 ordinal=001 sz=26 len=26 section=.rodata type=A string=
sort_type != sort_version
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
 %*s
addr=0x000124c5 off=0x000124c5 ordinal=005 sz=10 len=10 section=.rodata type=A string=
%*s, %*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=
cannot access %s
addr=0x000124e5 off=0x000124e5 ordinal=008 sz=29 len=29 section=.rodata type=A string=
cannot read symbolic link %s
addr=0x00012502 off=0x00012502 ordinal=009 sz=10 len=10 section=.rodata type=A string=
unlabeled
```

With the `-zr` option, this information is represented as a 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 the `-S` option gives complete information about the sections of an executable. For each section the index, offset, size, alignment, type and permissions, are shown. The next example demonstrates this:

```
$ 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 the `-Sr` option, rabin2 will flag the 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 @ 0x00000
274
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 the 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
```

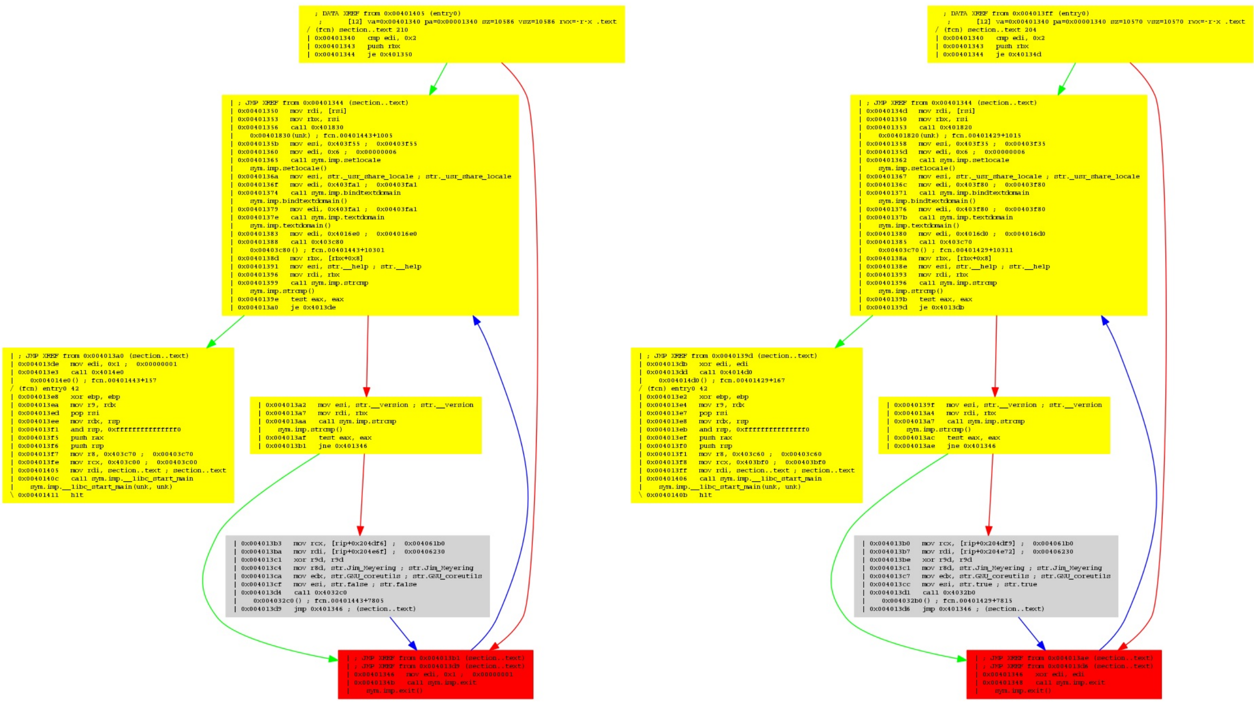
If you want more concrete data, it's also possible to count the differences, with the `-c` option:

```
$ 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
.__libc_start_main
         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 disassembler 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
```

Purpose

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

Syntax of the language

The code of r_egg is compiled as in a flow. It is a one-pass compiler;

this means that you have to define the proper stackframe size at the

beginning of the function, and you have to define the functions in

order to avoid getting compilation errors.

The compiler generates assembly code for x86-{32,64} and arm. But it aims

to support more platforms. This code is the compiled with r_asm and

injected into a tiny binary with r_bin.

You may like to use r_egg to create standalone binaries, position-

independent raw eggs to be injected on running processes or to patch

on-disk binaries.

The generated code is not yet optimized, but it's safe to be executed

at any place in the code.

Preprocessor

Aliases

Sometimes you just need to replace at compile time a single entity on

multiple places. Aliases are translated into 'equ' statements in assembly

language. This is just an assembler-level keyword redefinition.

```
AF_INET@alias(2);
```

```
printf@alias(0x8053940);
```

Includes

Use `cat(1)` or the preprocessor to concatenate multiple files to be compiled.

It's not a task of a compiler to look for external sources, so it's a

delegated task right now.. but we will probably add native support for

spp (merge into)

TODO: this is not yet implemented

```
INCDIR@alias("/usr/include/ragg2");
```

```
sys-osx.r@include(INCDIR);
```

Hashbang

eggs can use a hashbang to make them executable.

```
$ head -n1 hello.r
```

```
#!/usr/bin/ragg2 -X
```

```
$ ./hello.r
```

```
Hello World!
```

Main

The execution of the code is done as in a flow. The first function to be

defined will be the first one to be executed. If you want to run main()

just do like this:

```
#!/usr/bin/ragg2 -X
```

```
main();
```

```
...
```

```
main@global(128,64) {
```

```
...
```

Function definition

You may like to split up your code into several code blocks. Those blocks

are bound to a label followed by root brackets '{ ... }'

Function signatures

`name@type(stackframesize,staticframesize) { body }`

`name` : name of the function to define

`type` : see function types below

`stackframesize` : get space from stack to store local variables

`staticframesize` : get space from stack to store static variables (strings)

`body` : code of the function

Function types

`alias` Used to create aliases

`data` ; the body of the block is defined in .data

`inline` ; the function body is inlined when called

`global` ; make the symbol global

`fastcall` ; function that is called using the fast calling convention

`syscall` ; define syscall calling convention signature

Syscalls

r_egg offers a syntax sugar for defining syscalls. The syntax is like this:

`exit@syscall(1);`

`@syscall() {`

`` ` : mov eax, `` .arg```

`: int 0x80`

`}`

`main@global() {`

`exit (0);`

`}`

Libraries

At the moment there is no support for linking r_egg programs to system libraries. but if you inject the code into a program (disk/memory) you

can define the address of each function using the @alias syntax.

Core library

There's a work-in-progress libc-like library written completely in r_egg

Variables

`.arg`

`.arg0`

`.arg1`

`.arg2`

`.var0`

`.var2`

`.fix`

`.ret ; eax for x86, r0 for arm`

`.bp`

`.pc`

`.sp`

Arrays

Supported as raw pointers. TODO: enhance this feature

Tracing

Sometimes r_egg programs will break or just not work as expected. Use the

'trace' architecture to get a arch-backend call trace:

```
$ ragg2 -a trace -s yourprogram.r
```

Pointers

TODO: Theorically '*' is used to get contents of a memory pointer.

Virtual registers

TODO: a0, a1, a2, a3, sp, fp, bp, pc

Return values

The return value is stored in the a0 register, this register is set when

calling a function or when typing a variable name without assignment.

```
$ cat test.r
add@global(4) {
.var0 = .arg0 + .arg1;
.var0;
}
main@global() {
add (3,4);
}

$ ragg2 -F -o test test.r
$ ./test
$ echo $?
7
```

Traps

Each architecture have a different instruction to break the execution of

the program. REgg language captures calls to 'break()' to run the emit_trap

callback of the selected arch. The

`break()` ; --> compiles into 'int3' on x86

`break;` --> compiles into 'int3' on x86

Inline assembly

Lines prefixed with ':' char are just inlined in the output assembly.

```
: jmp 0x8048400
: .byte 33,44
```

Labels

You can define labels using the `:` keyword like this:

`:label_name:`

`/* loop forever */`

`goto(label_name`)

Control flow

`goto (addr)` -- branch execution

`while (cond)`

`if (cond)`

`break ()` -- executes a trap instruction

Comments

Supported syntax for comments are:

`/* multiline comment */'`

`// single line comment`

`# single line comment`

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_ ;  0x080
487cf
|      |   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.Youmustgiveapasswordforuset
hisprogram_ ;  0x0804880c
|     |    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 disassembly of the `main()` function (`pdf`).

Obtaining Hashes within Radare2 Session

To calculate a checksum of current block when running radare2, use the 'ph' command. Pass an algorithm name to it as a parameter. An example session:

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

You can use all hashing algorithms supported by `rahash2`:

```
[0x00404890]> ph?
md5
sha1
sha256
sha384
sha512
crc16
crc32
md4
xor
xorpair
parity
entropy
hamdist
pcprint
mod255
xxhash
adler32
luhn
```

The `ph` command accepts an optional numeric argument to specify length of byte range to be hashed, instead of default block size. For example:

```
[0x08049A80]> ph md5 32
9b9012b00ef7a94b5824105b7aaad83b
[0x08049A80]> ph md5 64
a71b087d8166c99869c9781e2edcf183
[0x08049A80]> ph 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 very large files because rahash2 buffers the whole input in memory before computing the hash.

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: 978317d58e3ed046305df92a19f7d3e0bfcb3c70cad979f
24fee289ed1d266b0
/bin/ls: 0x00000000-0x0001ae08 sha384: 9e946efdbebb4e0ca00c86129ce2a71ee734ac30b620336
c381aa929dd222709e4cf7a800b25fbc7d06fe3b184933845
/bin/ls: 0x00000000-0x0001ae08 sha512: 076806cedb5281fd15c21e493e12655c55c52537fc1f36e
641b57648f7512282c03264cf5402b1b15cf03a20c9a60edfd2b4f76d4905fcec777c297d3134f41f
/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.

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

Communication between radare and the debugger IO layer is wrapped into `system()` calls, which accept a string as an argument, and executes it as a command. An answer is then buffered in the 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 basic help for the 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
```

Getting Started

Small session in radare2 debugger

- `r2 -d /bin/ls` : Opens radare2 with file `/bin/ls` in debugger mode using the radare2 native debugger, but does not run the program. You'll see a prompt (radare2) - all examples are from this prompt.
- `db flag` : place a breakpoint at flag, where flag can be either an address or a function name
- `db - flag` : remove the breakpoint at flag, where flag can be either an address or a function name
- `db` : show list of breakpoint
- `dc` : run the program
- `dr` : Show registers state
- `drr` : Show registers references (telescoping) (like peda)
- `ds` : Step into instruction
- `dso` : Step over instruction
- `dbt` : Display backtrace
- `dm` : Show memory maps
- `dk <signal>` : Send KILL signal to child
- `ood` : reopen in debug mode
- `ood arg1 arg2` : reopen in debug mode with arg1 and arg2

Registers

The registers are part of a user area stored in the context structure used by the scheduler. This structure can be manipulated to get and set the 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 'rip'
0x7f0f2dbae630

[0x4A13B8C0]> dr rip = esp   ; set 'rip' 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
oeax = 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 as a server process which is controlled by a local radare2 process. This is possible because everything uses radare's IO subsystem which abstracts access to system(), cmd() and all basic IO operations so to work over a 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 make this clearer. A typical remote session might look 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]> =-
```

You can also redirect radare output to a TCP or UDP server (such as `nc -l`). First, Add the server with '=+ tcp://' or '=+ udp://', then you can redirect the output of a command to be sent to the server:

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

The `=<' command will send the output from the execution of`cmd` to the remote connection number N (or the last one used if no id specified).

Plugins

IO plugins

All access to files, network, debugger, etc. is wrapped by an IO abstraction layer that allows radare to treat all data as if it were just a 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
.class, show files with: zip://foo.apk/, open all files with zipall:// (BSD)
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 stri
ng=IOLI Crackme Level 0x00\n
vaddr=0x08048581 paddr=0x00000581 ordinal=001 sz=11 len=10 section=.rodata type=a stri
ng=Password:
vaddr=0x0804858f paddr=0x0000058f ordinal=002 sz=7 len=6 section=.rodata type=a string
=250382
vaddr=0x08048596 paddr=0x00000596 ordinal=003 sz=19 len=18 section=.rodata type=a stri
ng=Invalid Password!\n
vaddr=0x080485a9 paddr=0x000005a9 ordinal=004 sz=16 len=15 section=.rodata type=a stri
ng=Password OK :)\n
```

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 stri
ng=IOLI Crackme Level 0x00\n
```

Here we have the prompt for the password.

```
vaddr=0x08048581 paddr=0x00000581 ordinal=001 sz=11 len=10 section=.rodata type=a stri
ng=Password:
```

This is the error on entering an invalid password.

```
vaddr=0x08048596 paddr=0x00000596 ordinal=003 sz=19 len=18 section=.rodata type=a stri
ng=Invalid Password!\n
```

This is the message on the password being accepted.

```
vaddr=0x080485a9 paddr=0x000005a9 ordinal=004 sz=16 len=15 section=.rodata type=a stri
ng=Password OK :)\n
```

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
=250382
```

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 stri
ng=IOLI Crackme Level 0x01\n
vaddr=0x08048541 paddr=0x00000541 ordinal=001 sz=11 len=10 section=.rodata type=a stri
ng=Password:
vaddr=0x0804854f paddr=0x0000054f ordinal=002 sz=19 len=18 section=.rodata type=a stri
ng=Invalid Password!\n
vaddr=0x08048562 paddr=0x00000562 ordinal=003 sz=16 len=15 section=.rodata type=a stri
ng=Password OK :)\n
```

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 f
ind strings
[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 ;
 [0x8048528:4]=0x494c4f49  ; "IOLI Crackme Level 0x01." @ 0x8048528
|           0x08048407    e810ffffff    call sym.imp.printf
|              sym.imp.printf(unk)
|           0x0804840c    c7042441850. mov dword [esp], str.Password_ ; [0x8048541:4]=0
x73736150  ; "Password: " @ 0x8048541
|           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]=0x490
06425  ; "%d" @ 0x804854c
|           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 ; [0x80
4854f:4]=0x61766e49  ; "Invalid Password!." @ 0x804854f
|       |   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 ; [0x80485
62:4]=0x73736150  ; "Password OK :)." @ 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.

.intro

After a few years of missing out on wargames at Hacktivity, this year I've finally found the time to begin, and almost finish (yeah, I'm quite embarrassed about that unfinished webhack :)) one of them. There were 3 different games at the conf, and I've chosen the one that was provided by avatao. It consisted of 8 challenges, most of them being basic web hacking stuff, one sandbox escape, one simple buffer overflow exploitation, and there were two reverse engineering exercises too. You can find these challenges on https://platform.avatao.com.

.radare2

I've decided to solve the reversing challenges using radare2, a free and open source reverse engineering framework. I have first learned about r2 back in 2011. during a huge project, where I had to reverse a massive, 11MB statically linked ELF. I simply needed something that I could easily patch Linux ELFs with. Granted, back then I've used r2 alongside IDA, and only for smaller tasks, but I loved the whole concept at first sight. Since then, radare2 evolved a lot, and I was planning for some time now to solve some crackmes with the framework, and write writeups about them. Well, this CTF gave me the perfect opportunity :)

Because this writeup aims to show some of r2's features besides how the crackmes can be solved, I will explain every r2 command I use in blockquote paragraphs like this one:

> ***r2 tip:*** Always use ? or -h to get more information!

If you know r2, and just interested in the crackme, feel free to skip those parts! Also keep in mind please, that because of this tutorial style I'm going to do a lot of stuff that you just don't do during a CTF, because there is no time for proper bookkeeping (e.g. flag every memory area according to its purpose), and with such small executables you can succeed without doing these stuff.

A few advice if you are interested in learning radare2 (and frankly, if you are into RE, you should be interested in learning r2 :)):

The framework has a lot of supplementary executables and a vast amount of functionality - and they are very well documented. I encourage you to read the available docs, and use the built-in help (by appending a ? to any command) extensively! E.g.:

```
[0x00000000]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
|%var =valueAlias for 'env' command
| *off[=[0x]value]     Pointer read/write data/values (see ?v, wx, wv)
| (macro arg0 arg1)    Manage scripting macros
| .[-|(m)|f|!sh|cmd]   Define macro or load r2, cparse or rlang file
| = [cmd]              Run this command via rap://
| /                    Search for bytes, regexps, patterns, ..
| ! [cmd]              Run given command as in system(3)
| # [algo] [len]       Calculate hash checksum of current block
| #!lang [..]          Hashbang to run an rlang script
| a                    Perform analysis of code
| b                    Get or change block size

...

[0x00000000]> a?
|Usage: a[abdefFghoprxstc] [...]
| ab [hexpairs]     analyze bytes
| aa                analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| ac [cycles]       analyze which op could be executed in [cycles]
| 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 anal.depth=1

...
```

Also, the project is under heavy development, there is no day without commits to the GitHub repo. So, as the readme says, you should always use the git version!

Some highly recommended reading materials:

- Cheatsheet by pwntester
- Radare2 Book
- Radare2 Blog
- Radare2 Wiki

.first_steps

OK, enough of praising r2, lets start reversing this stuff. First, you have to know your enemy:

```
[0x00 avatao]$ rabin2 -I reverse4
pic      false
canary   true
nx       true
crypto   false
va       true
intrp    /lib64/ld-linux-x86-64.so.2
bintype  elf
class    ELF64
lang     c
arch     x86
bits     64
machine  AMD x86-64 architecture
os       linux
subsys   linux
endian   little
stripped true
static   false
linenum  false
lsyms    false
relocs   false
rpath    NONE
binsz    8620
```

> ***r2 tip:*** rabin2 is one of the handy tools that comes with radare2. It can be used to extract information (imports, symbols, libraries, etc.) about binary executables. As always, check the help (rabin2 -h)!

So, its a dynamically linked, stripped, 64bit Linux executable - nothing fancy here. Let's try to run it:

```
[0x00 avatao]$ ./reverse4
?
Size of data: 2623
pamparam
Wrong!

[0x00 avatao]$ "\x01\x00\x00\x00" | ./reverse4
Size of data: 1
```

OK, so it reads a number as a size from the standard input first, than reads further, probably "size" bytes/characters, processes this input, and outputs either "Wrong!", nothing or something else, presumably our flag. But do not waste any more time monkeyfuzzing the executable, let's fire up r2, because in asm we trust!

```
[0x00 avatao]$ r2 -A reverse4
 -- Heisenbug: A bug that disappears or alters its behavior when one attempts to probe
 or isolate it.
[0x00400720]>
```

> ***r2 tip:*** The -A switch runs *aa* command at start to analyze all referenced code, so we will have functions, strings, XREFS, etc. right at the beginning. As usual, you can get help with *?*.

It is a good practice to create a project, so we can save our progress, and we can come back at a later time:

```
[0x00400720]> Ps avatao_reverse4
avatao_reverse4
[0x00400720]>
```

> ***r2 tip:*** You can save a project using Ps [file], and load one using Po [file]. With the -p option, you can load a project when starting r2.

We can list all the strings r2 found:

```
[0x00400720]> fs strings
[0x00400720]> f
0x00400e98 7 str.Wrong_
0x00400e9f 27 str.We_are_in_the_outer_space_
0x00400f80 18 str.Size_of_data:__u_n
0x00400f92 23 str.Such_VM__MuCH_reV3rse_
0x00400fa9 16 str.Use_everything_
0x00400fbb 9 str.flag.txt
0x00400fc7 26 str.You_won__The_flag_is:__s_n
0x00400fe1 21 str.Your_getting_closer_
[0x00400720]>
```

> ***r2 tip***: r2 puts so called flags on important/interesting offsets, and organizes these flags into flagspaces (strings, functions, symbols, etc.) You can list all flagspaces using *fs*, and switch the current one using *fs [flagspace]* (the default is *, which means all the flagspaces). The command *f* prints all flags from the currently selected flagspace(s).

OK, the strings looks interesting, especially the one at 0x00400f92. It seems to hint that this crackme is based on a virtual machine. Keep that in mind!

These strings could be a good starting point if we were talking about a real-life application with many-many features. But we are talking about a crackme, and they tend to be small and simple, and focused around the problem to be solved. So I usually just take a look at the entry point(s) and see if I can figure out something from there. Nevertheless, I'll show you how to find where these strings are used:

```
[0x00400720]> axt @@=`f~[0]`
d 0x400cb5 mov edi, str.Size_of_data:__u_n
d 0x400d1d mov esi, str.Such_VM__MuCH_reV3rse_
d 0x400d4d mov edi, str.Use_everything_
d 0x400d85 mov edi, str.flag.txt
d 0x400db4 mov edi, str.You_won__The_flag_is:__s_n
d 0x400dd2 mov edi, str.Your_getting_closer_
```

> ***r2 tip***: We can list crossreferences to addresses using the *axt [addr]* command (similarly, we can use *axf* to list references from the address). The @@ is an iterator, it just runs the command once for every arguments listed.
>
> The argument list in this case comes from the command *f~[0]*. It lists the strings from the executable with *f*, and uses the internal grep command ~ to select only the first column (*[0]*) that contains the strings' addresses.

.main

As I was saying, I usually take a look at the entry point, so let's just do that:

```
[0x00400720]> s main
[0x00400c63]>
```

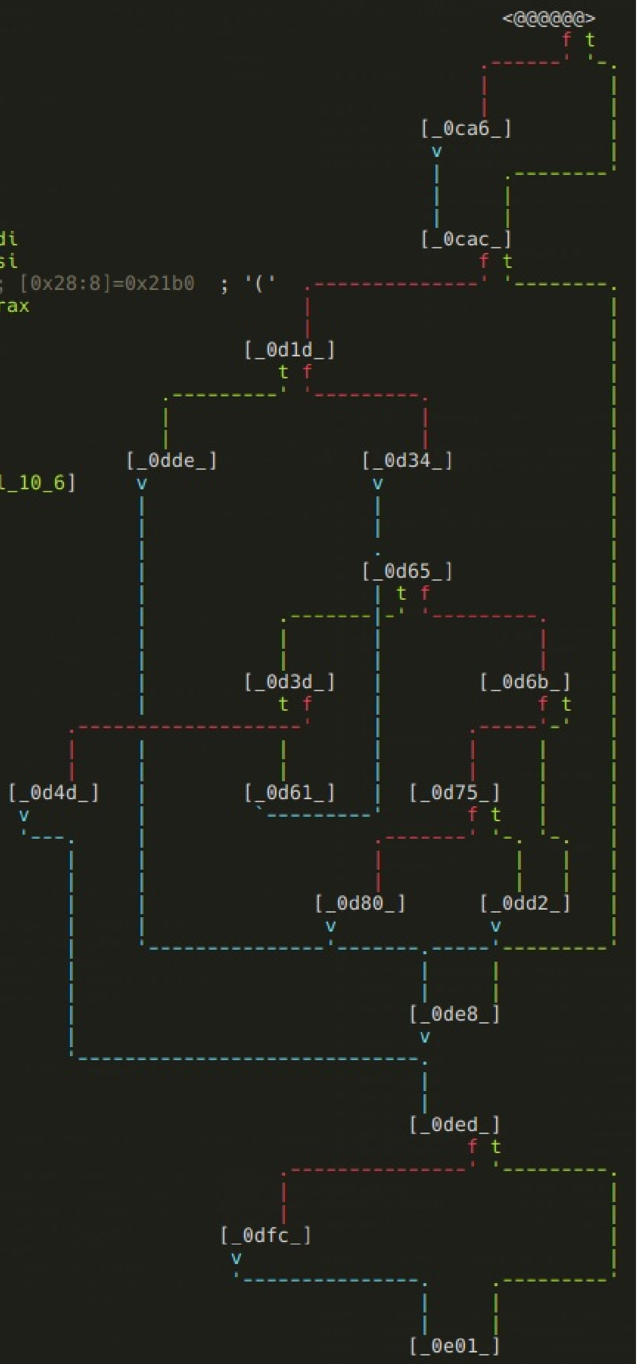
> ***r2 tip:*** You can go to any offset, flag, expression, etc. in the executable using the *s* command (seek). You can use references, like *$$* (current offset), you can undo (*s-*) or redo (*s+*) seeks, search strings (*s/ [string]*) or hex values (*s/x 4142*), and a lot of other useful stuff. Make sure to check out *s?*!

Now that we are at the beginning of the main function, we could use *p* to show a disassembly (*pd*, *pdf*), but r2 can do something much cooler: it has a visual mode, and it can display graphs similar to IDA, but way cooler, since they are ASCII-art graphs :)

> ***r2 tip:*** The command family *p* is used to print stuff. For example it can show disassembly (*pd*), disassembly of the current function (*pdf*), print strings (*ps*), hexdump (*px*), base64 encode/decode data (*p6e*, *p6d*), or print raw bytes (*pr*) so you can for example dump parts of the binary to other files. There are many more functionalities, check *?*!

R2 also has a minimap view which is incredibly useful for getting an overall look at a function:

```
[0x00400c63]> VV @ main (nodes 18 edges 25 zoom 100%) BB mouse:canvas-y movements-speed:5

0x400c63:
(fcn) main 416
; arg int arg_1         @ rbp+0x8
; arg int arg_375       @ rbp+0xbb8
; var int local_0_1     @ rbp-0x1
; var int local_1       @ rbp-0x8
; var int local_8       @ rbp-0x40
; var int local_9       @ rbp-0x48
; var int local_10      @ rbp-0x50
; var int local_10_4    @ rbp-0x54
; var int local_10_6    @ rbp-0x56
; DATA XREF from 0x0040073d (main)
0x00400c63   55                 push rbp
0x00400c64   4889e5             mov rbp, rsp
0x00400c67   4883ec70           sub rsp, 0x70
0x00400c6b   897d9c             mov dword [rbp - 0x64], edi
0x00400c6e   48897590           mov qword [rbp - 0x70], rsi
0x00400c72   64488b042528.      mov rax, qword fs:[0x28] ; [0x28:8]=0x21b0  ; '('
0x00400c7b   488945f8           mov qword [rbp-local_1], rax
0x00400c7f   31c0               xor eax, eax
0x00400c81   488d45aa           lea rax, [rbp-local_10_6]
0x00400c85   ba02000000         mov edx, 2
0x00400c8a   4889c6             mov rsi, rax
0x00400c8d   bf00000000         mov edi, 0
0x00400c92   b800000000         mov eax, 0
0x00400c97   e824faffff         call sym.imp.read ;[a]
0x00400c9c   0fb745aa           movzx eax, word [rbp-local_10_6]
0x00400ca0   663db80b           cmp ax, 0xbb8
0x00400ca4   7606               jbe 0x400cac ;[b]
```



> ***r2 tip:*** With command *V* you can enter the so-called visual mode, which has several views. You can switch between them using *p* and *P*. The graph view can be displayed by hitting *V* in visual mode (or using *VV* at the prompt).
>
> Hitting *p* in graph view will bring up the minimap. It displays the basic blocks and the connections between them in the current function, and it also shows the disassembly of the currently selected block (marked with @@@@@ on the minimap). You can select the next or the previous block using the ** and the ** keys respectively. You can also select the true or the false branches using the *t* and the *f* keys.
>
> It is possible to bring up the prompt in visual mode using the *:* key, and you can use *o* to seek.

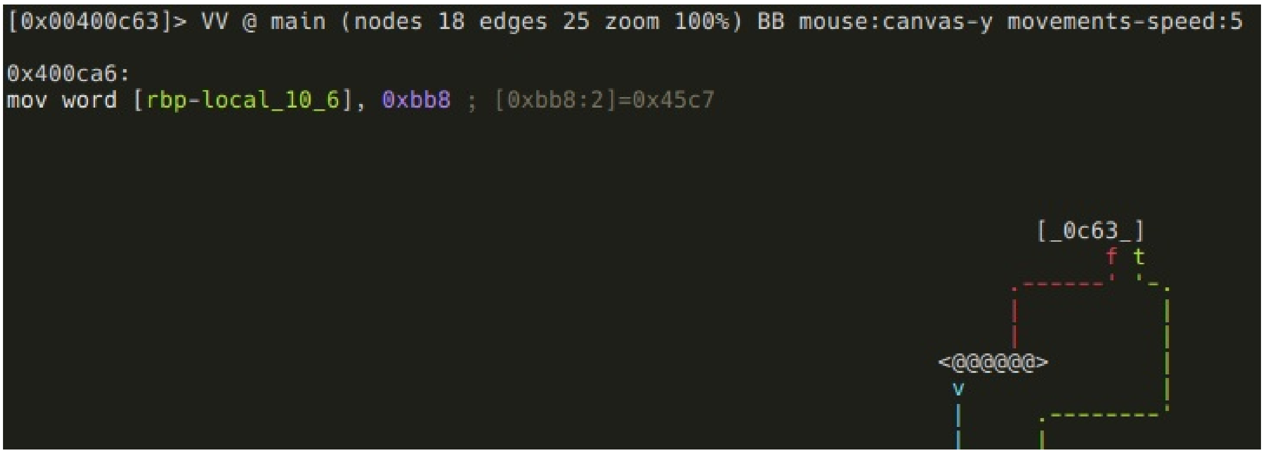
Lets read main node-by-node! The first block looks like this:

We can see that the program reads a word (2 bytes) into the local variable named *local_10_6*, and than compares it to 0xbb8. Thats 3000 in decimal, btw:

```
[0x00400c63]> ? 0xbb8
3000 0xbb8 05670 2.9K 0000:0bb8 3000 10111000 3000.0 0.000000f 0.000000
```

> ***r2 tip***: yep, *?* will evaluate expressions, and print the result in various formats.

If the value is greater than 3000, then it will be forced to be 3000:



There are a few things happening in the next block:

First, the "Size of data: " message we saw when we run the program is printed. So now we know that the local variable *local_10_6* is the size of the input data - so lets name it accordingly (remember, you can open the r2 shell from visual mode using the *:* key!):

```
:> afvn local_10_6 input_size
```

> ***r2 tip***: The *af* command family is used to analyze functions. This includes manipulating arguments and local variables too, which is accessible via the *afv* commands. You can list function arguments (*afa*), local variables (*afv*), or you can even rename them (*afan*, *afvn*). Of course there are lots of other features too - as usual: use the "?", Luke!

After this an *input_size* bytes long memory chunk is allocated, and filled with data from the standard input. The address of this memory chunk is stored in *local_10* - time to use *afvn* again:

```
:> afvn local_10 input_data
```

We've almost finished with this block, there are only two things remained. First, an 512 (0x200) bytes memory chunk is zeroed out at offset 0x00602120. A quick glance at XREFS to this address reveals that this memory is indeed used somewhere in the application:

```
:> axt 0x00602120
d 0x400cfe mov edi, 0x602120
d 0x400d22 mov edi, 0x602120
d 0x400dde mov edi, 0x602120
d 0x400a51 mov qword [rbp - 8], 0x602120
```

Since it probably will be important later on, we should label it:

```
:> f sym.memory 0x200 0x602120
```

> ***r2 tip***: Flags can be managed using the *f* command family. We've just added the flag sym.memory to a 0x200 bytes long memory area at 0x602120. It is also possible to remove (*f-name*), rename (*fr [old] [new]*), add comment (*fC [name] [cmt]*) or even color (*fc [name] [color]*) flags.

While we are here, we should also declare that memory chunk as data, so it will show up as a hexdump in disassembly view:

```
:> Cd 0x200 @ sym.memory
```

> ***r2 tip***: The command family *C* is used to manage metadata. You can set (*CC*) or edit (*CC*) comments, declare memory areas as data (*Cd*), strings (*Cs*), etc. These commands can also be issued via a menu in visual mode invoked by pressing *d*.

The only remaining thing in this block is a function call to 0x400a45 with the input data as an argument. The function's return value is compared to "*", and a conditional jump is executed depending on the result.

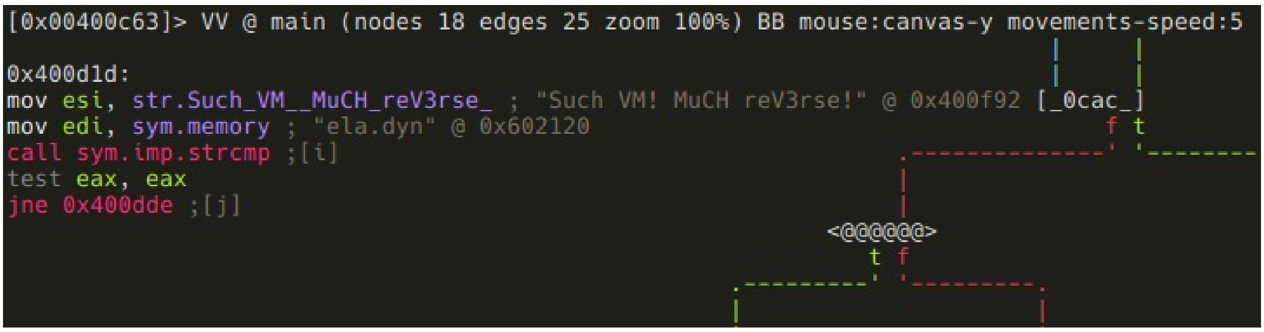
Earlier I told you that this crackme is probably based on a virtual machine. Well, with that information in mind, one can guess that this function will be the VM's main loop, and the input data is the instructions the VM will execute. Based on this hunch, I've named this function *vmloop*, and renamed *input_data* to *bytecode* and *input_size* to *bytecode_length*. This is not really necessary in a small project like this, but it's a good practice to name stuff according to their purpose (just like when you are writing programs).

```
:> af vmloop 0x400a45
:> afvn input_size bytecode_length
:> afvn input_data bytecode
```

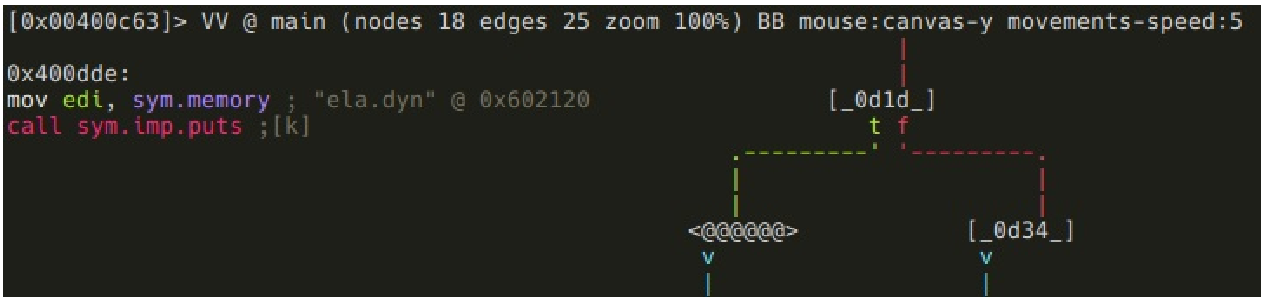
> ***r2 tip***: The *af* command is used to analyze a function with a given name at the given address. The other two commands should be familiar from earlier.

After renaming local variables, flagging that memory area, and renaming the VM loop function the disassembly looks like this:

So, back to that conditional jump. If *vmloop* returns anything else than "*", the program just exits without giving us our flag. Obviously we don't want that, so we follow the false branch.



Now we see that a string in that 512 bytes memory area (*sym.memory*) gets compared to "Such VM! MuCH reV3rse!". If they are not equal, the program prints the bytecode, and exits:

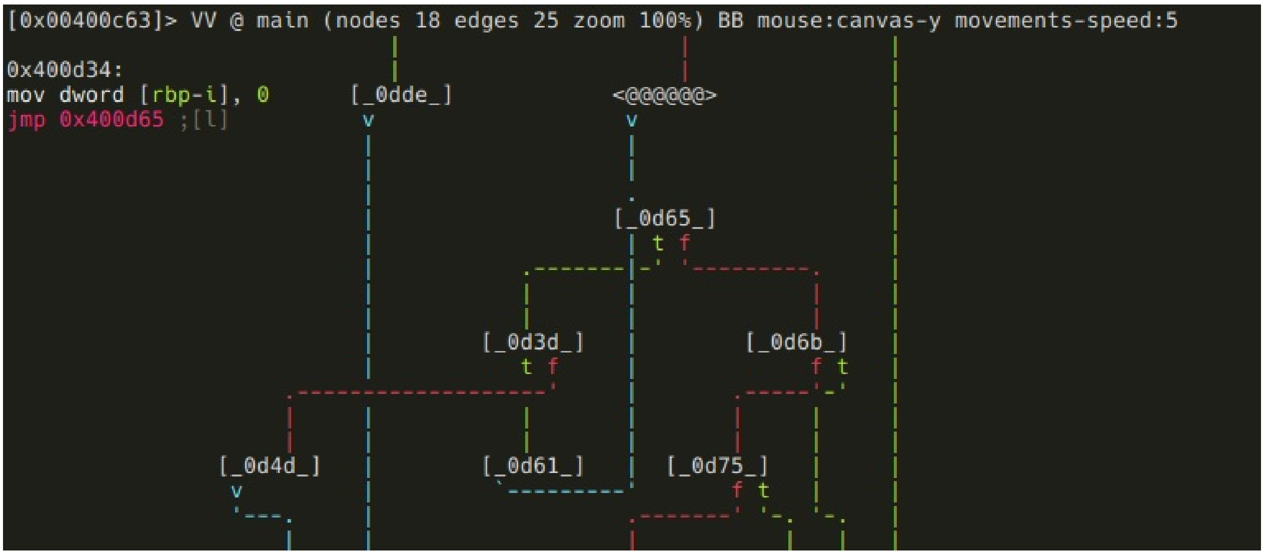


OK, so now we know that we have to supply a bytecode that will generate that string when executed. As we can see on the minimap, there are still a few more branches ahead, which probably means more conditions to meet. Lets investigate them before we delve into *vmloop*!

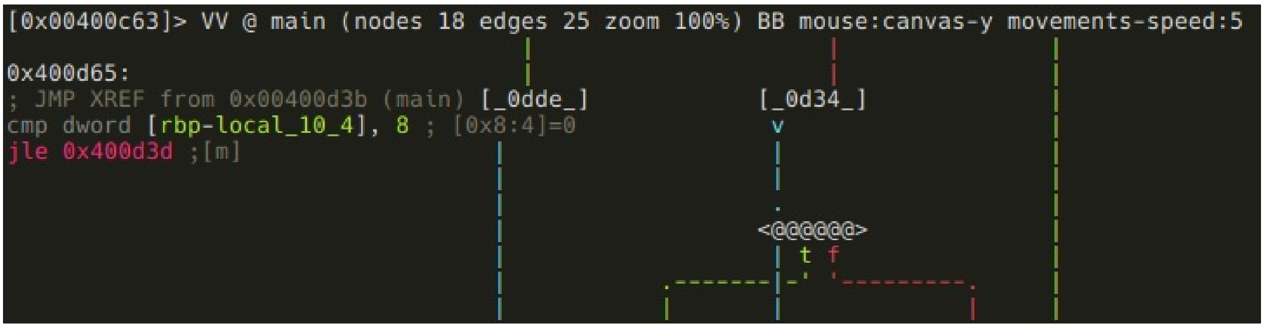
If you take a look at the minimap of the whole function, you can probably recognize that there is some kind of loop starting at block *[0d34]*, and it involves the following nodes:

- [*0d34*]
- [*0d65*]
- [*0d3d*]
- [*0d61*]

Here are the assembly listings for those blocks. The first one puts 0 into local variable *local_10_4*:



And this one compares *local_10_4* to 8, and executing a conditional jump based on the result:

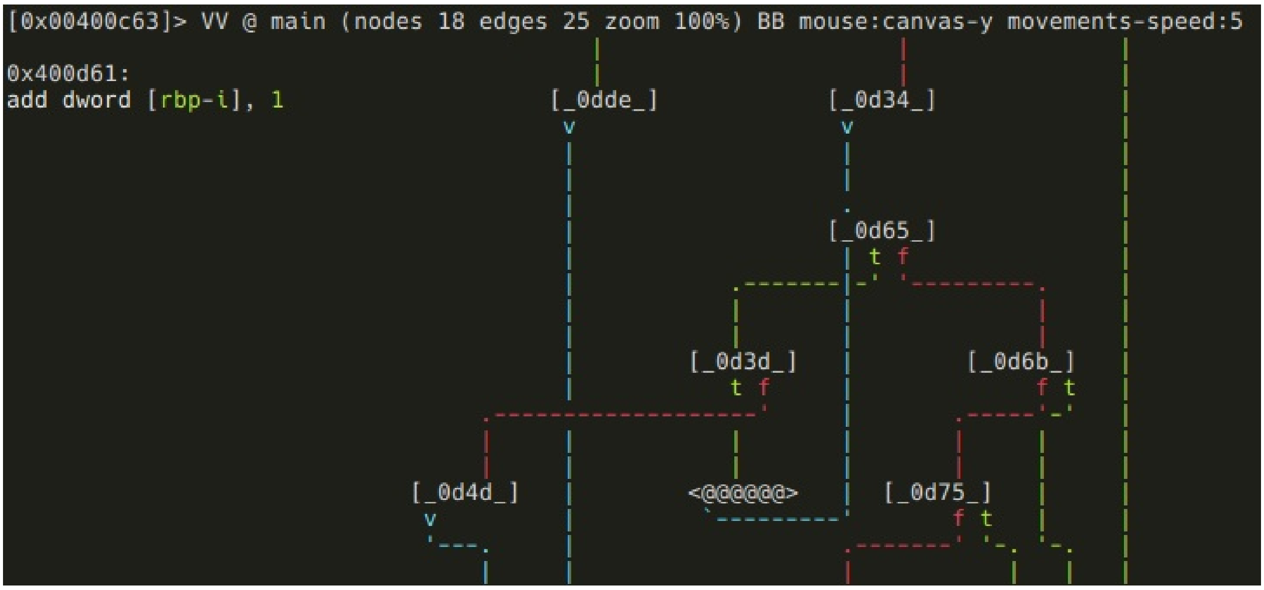


It's pretty obvious that *local_10_4* is the loop counter, so lets name it accordingly:

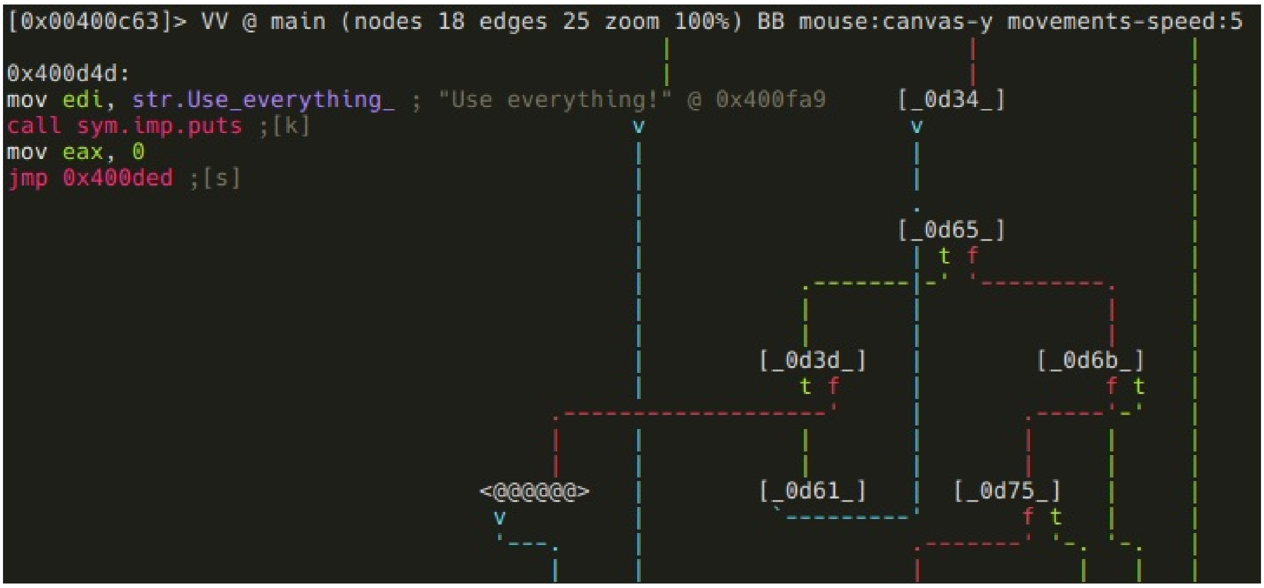
```
:> afvn local_10_4 i
```

Next block is the actual loop body:

The memory area at 0x6020e0 is treated as an array of dwords (4 byte values), and checked if the ith value of it is zero. If it is not, the loop simply continues:



If the value is zero, the loop breaks and this block is executed before exiting:



It prints the following message: Use everything!" As we've established earlier, we are dealing with a virtual machine. In that context, this message probably means that we have to use every available instructions. Whether we executed an instruction or not is stored at 0x6020e0 - so lets flag that memory area:

```
:> f sym.instr_dirty 4*9 0x6020e0
```

Assuming we don't break out and the loop completes, we are moving on to some more checks:

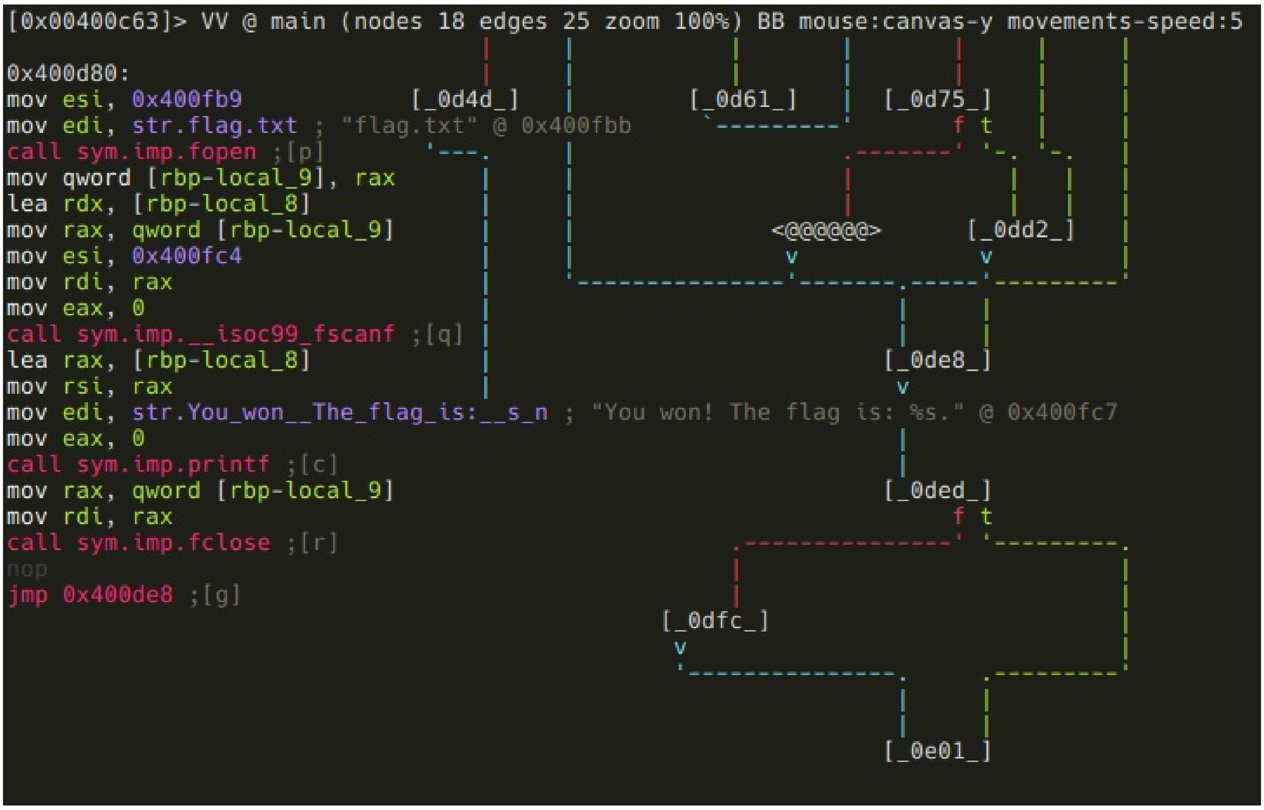
This piece of code may look a bit strange if you are not familiar with x86_64 specific stuff. In particular, we are talking about RIP-relative addressing, where offsets are described as displacements from the current instruction pointer, which makes implementing PIE easier. Anyways, r2 is nice enough to display the actual address (0x602104). Got the address, flag it!

```
:> f sym.good_if_ne_zero 4 0x602104
```

Keep in mind though, that if RIP-relative addressing is used, flags won't appear directly in the disassembly, but r2 displays them as comments:

If *sym.good_if_ne_zero* is zero, we get a message ("Your getting closer!"), and then the program exits. If it is non-zero, we move to the last check:

Here the program compares a dword at 0x6020f0 (again, RIP-relative addressing) to 9. If its greater than 9, we get the same "Your getting closer!" message, but if it's lesser, or equal to 9, we finally reach our destination, and get the flag:



As usual, we should flag 0x6020f0:

```
:> f sym.good_if_le_9 4 0x6020f0
```

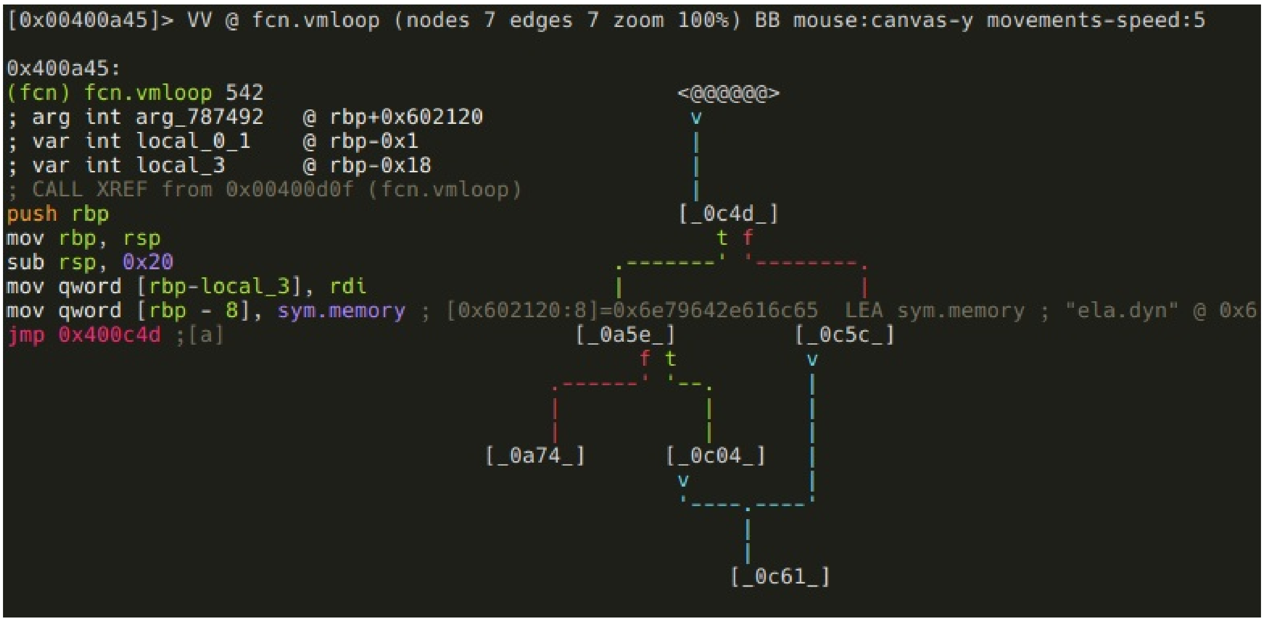
Well, it seems that we have fully reversed the main function. To summarize it: the program reads a bytecode from the standard input, and feeds it to a virtual machine. After VM execution, the program's state have to satisfy these conditions in order to reach the goodboy code:

- *vmloop*'s return value has to be "*"
- *sym.memory* has to contain the string "Such VM! MuCH reV3rse!"
- all 9 elements of *sym.instr_dirty* array should not be zero (probably means that all instructions had to be used at least once)
- *sym.good_if_ne_zero* should not be zero
- *sym.good_if_le_9* has to be lesser or equal to 9

This concludes our analysis of the main function, we can now move on to the VM itself.

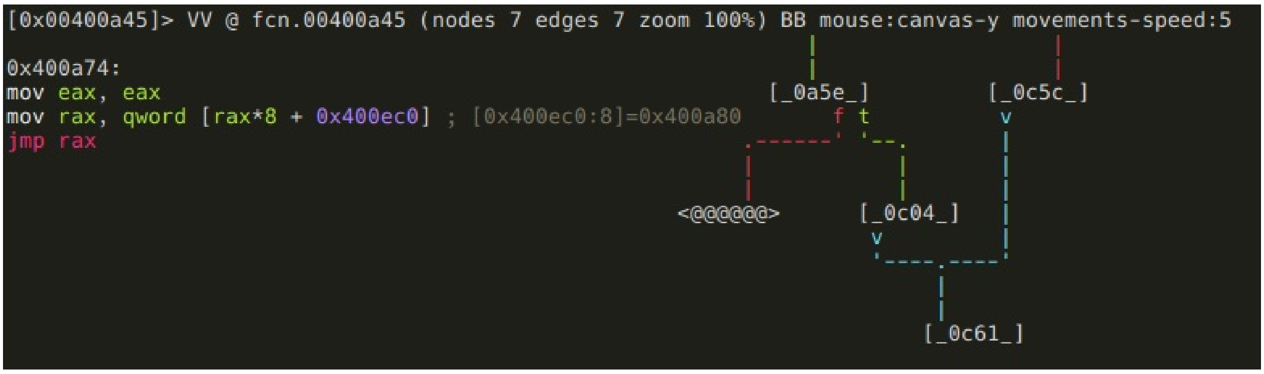
.vmloop

```
[offset]> fcn.vmloop
```



```
[0x00400a45]> VV @ fcn.vmloop (nodes 7 edges 7 zoom 100%) BB mouse:canvas-y movements-speed:5

0x400a45:
(fcn) fcn.vmloop 542
; arg int arg_787492   @ rbp+0x602120
; var int local_0_1    @ rbp-0x1
; var int local_3      @ rbp-0x18
; CALL XREF from 0x00400d0f (fcn.vmloop)
push rbp
mov rbp, rsp
sub rsp, 0x20
mov qword [rbp-local_3], rdi
mov qword [rbp - 8], sym.memory ; [0x602120:8]=0x6e79642e616c65  LEA sym.memory ; "ela.dyn" @ 0x6
jmp 0x400c4d ;[a]
```


Well, that seems disappointingly short, but no worries, we have plenty to reverse yet. The thing is that this function uses a jump table at 0x00400a74,



```
[0x00400a45]> VV @ fcn.00400a45 (nodes 7 edges 7 zoom 100%) BB mouse:canvas-y movements-speed:5

0x400a74:
mov eax, eax
mov rax, qword [rax*8 + 0x400ec0] ; [0x400ec0:8]=0x400a80
jmp rax
```


and r2 can't yet recognize jump tables (Issue 3201), so the analysis of this function is a bit incomplete. This means that we can't really use the graph view now, so either we just use visual mode, or fix those basic blocks. The entire function is just 542 bytes long, so we certainly could reverse it without the aid of the graph mode, but since this writeup aims to include as much r2 wisdom as possible, I'm going to show you how to define basic blocks.

But first, lets analyze what we already have! First, *rdi* is put into local_3. Since the application is a 64bit Linux executable, we know that *rdi* is the first function argument (as you may have recognized, the automatic analysis of arguments and local variables was not entirely correct), and we also know that *vmloop*'s first argument is the bytecode. So lets rename local_3:

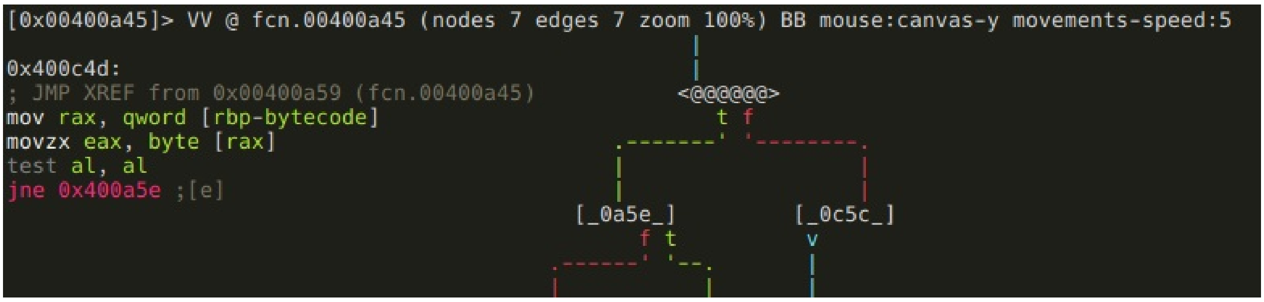
```
:> afvn local_3 bytecode
```

Next, *sym.memory* is put into another local variable at *rbp-8* that r2 did not recognize. So let's define it!

```
:> afv 8 memory qword
```

> ***r2 tip***: The *afv [idx] [name] [type]* command is used to define local variable at [frame pointer - idx] with the name [name] and type [type]. You can also remove local variables using the *afv- [idx]* command.

In the next block, the program checks one byte of bytecode, and if it is 0, the function returns with 1.



If that byte is not zero, the program subtracts 0x41 from it, and compares the result to 0x17. If it is above 0x17, we get the dreaded "Wrong!" message, and the function returns with 0. This basically means that valid bytecodes are ASCII characters in the range of "A" (0x41) through "X" (0x41 + 0x17). If the bytecode is valid, we arrive at the code piece that uses the jump table:

The jump table's base is at 0x400ec0, so lets define that memory area as a series of qwords:

```
[0x00400a74]> s 0x00400ec0
[0x00400ec0]> Cd 8 @@=`?s $$ $$+8*0x17 8`
```

> ***r2 tip***: Except for the *?s*, all parts of this command should be familiar now, but lets recap it! *Cd* defines a memory area as data, and 8 is the size of that memory area. @@ is an iterator that make the preceding command run for every element that @@ holds. In this example it holds a series generated using the *?s* command. *?s* simply generates a series from the current seek (*$$*) to current seek + 8*0x17* (*$$+8*0x17*) with a step of 8.

This is how the disassembly looks like after we add this metadata:

```
[0x00400ec0]> pd 0x18
            ; DATA XREF from 0x00400a76 (unk)
            0x00400ec0 .qword 0x0000000000400a80
            0x00400ec8 .qword 0x0000000000400c04
            0x00400ed0 .qword 0x0000000000400b6d
            0x00400ed8 .qword 0x0000000000400b17
            0x00400ee0 .qword 0x0000000000400c04
            0x00400ee8 .qword 0x0000000000400c04
            0x00400ef0 .qword 0x0000000000400c04
            0x00400ef8 .qword 0x0000000000400c04
            0x00400f00 .qword 0x0000000000400aec
            0x00400f08 .qword 0x0000000000400bc1
            0x00400f10 .qword 0x0000000000400c04
            0x00400f18 .qword 0x0000000000400c04
            0x00400f20 .qword 0x0000000000400c04
            0x00400f28 .qword 0x0000000000400c04
            0x00400f30 .qword 0x0000000000400c04
            0x00400f38 .qword 0x0000000000400b42
            0x00400f40 .qword 0x0000000000400c04
            0x00400f48 .qword 0x0000000000400be5
            0x00400f50 .qword 0x0000000000400ab6
            0x00400f58 .qword 0x0000000000400c04
            0x00400f60 .qword 0x0000000000400c04
            0x00400f68 .qword 0x0000000000400c04
            0x00400f70 .qword 0x0000000000400c04
            0x00400f78 .qword 0x0000000000400b99
```

As we can see, the address 0x400c04 is used a lot, and besides that there are 9 different addresses. Lets see that 0x400c04 first!

```
[0x00400ec0]> pd 4 @ 0x400c04
            ;-- not_instr:
|           0x00400c04    bf980e4000      mov edi, str.Wrong_ ; "Wrong!" @ 0x400e98
|           0x00400c09    e862faffff      call sym.imp.puts
|           0x00400c0e    b800000000      mov eax, 0
|           0x00400c13    eb4c            jmp 0x400c61
[0x00400ec0]>
```

We get the message "Wrong!", and the function just returns 0. This means that those are not valid instructions (they are valid bytecode though, they can be e.g. parameters!) We should flag 0x400c04 accordingly:

```
[0x00400ec0]> f not_instr @ 0x0000000000400c04
```

As for the other offsets, they all seem to be doing something meaningful, so we can assume they belong to valid instructions. I'm going to flag them using the instructions' ASCII values:

```
[0x00400ec0]> f instr_A @ 0x0000000000400a80
[0x00400ec0]> f instr_C @ 0x0000000000400b6d
[0x00400ec0]> f instr_D @ 0x0000000000400b17
[0x00400ec0]> f instr_I @ 0x0000000000400aec
[0x00400ec0]> f instr_J @ 0x0000000000400bc1
[0x00400ec0]> f instr_P @ 0x0000000000400b42
[0x00400ec0]> f instr_R @ 0x0000000000400be5
[0x00400ec0]> f instr_S @ 0x0000000000400ab6
[0x00400ec0]> f instr_X @ 0x0000000000400b99
```

Ok, so these offsets were not on the graph, so it is time to define basic blocks for them!

> ***r2 tip***: You can define basic blocks using the *afb+* command. You have to supply what function the block belongs to, where does it start, and what is its size. If the block ends in a jump, you have to specify where does it jump too. If the jump is a conditional jump, the false branch's destination address should be specified too.

We can get the start and end addresses of these basic blocks from the full disasm of *vmloop*.

```
[0x00400ec0]> pdf @ fcn.vmloop
/ (fcn) fcn.00400a45 542
|           ; arg int arg_787492   @ rbp+0x602120
|           ; var int local_0_1    @ rbp-0x1
|           ; var qword memory       @ rbp-0x8
|           ; var dword instr_ptr_step @ rbp-0xc
|           ; var int bytecode     @ rbp-0x18
|           ;-- fcn.vmloop:
|           0x00400a45    55             push rbp
|           0x00400a46    4889e5         mov rbp, rsp
|           0x00400a49    4883ec20       sub rsp, 0x20
|           0x00400a4d    48897de8       mov qword [rbp-bytecode], rdi
|           0x00400a51    48c745f82021.  mov qword [rbp-memory], sym.memory
|       ,=< 0x00400a59    e9ef010000     jmp 0x400c4d
|       |   0x00400a5e    488b45e8       mov rax, qword [rbp-bytecode]
|       |   0x00400a62    0fb600         movzx eax, byte [rax]
|       |   0x00400a65    0fbec0         movsx eax, al
|       |   0x00400a68    83e841         sub eax, 0x41
|       |   0x00400a6b    83f817         cmp eax, 0x17
|      ,==< 0x00400a6e    0f8790010000   ja 0x400c04
|      ||   0x00400a74    89c0           mov eax, eax
|      ||   0x00400a76    488b04c5c00e.  mov rax, qword [rax*8 + 0x400ec0]
|      ||   0x00400a7e    ffe0           jmp rax
|      ||   ;-- instr_A:
|      ||   0x00400a80    8b055a162000   mov eax, dword [rip + 0x20165a]
|      ||   0x00400a86    83c001         add eax, 1
|      ||   0x00400a89    890551162000   mov dword [rip + 0x201651], eax
|      ||   0x00400a8f    488b45e8       mov rax, qword [rbp-bytecode]
|      ||   0x00400a93    488d5002       lea rdx, [rax + 2]
|      ||   0x00400a97    488b45e8       mov rax, qword [rbp-bytecode]
|      ||   0x00400a9b    4883c001       add rax, 1
|      ||   0x00400a9f    4889d6         mov rsi, rdx
|      ||   0x00400aa2    4889c7         mov rdi, rax
|      ||   0x00400aa5    e863fdffff     call fcn.0040080d
|      ||   0x00400aaa    c745f4030000.  mov dword [rbp-instr_ptr_step], 3
|     ,===< 0x00400ab1    e95f010000     jmp 0x400c15
|     |||   ;-- instr_S:
|     |||   0x00400ab6    8b0528162000   mov eax, dword [rip + 0x201628]
|     |||   0x00400abc    83c001         add eax, 1
|     |||   0x00400abf    89051f162000   mov dword [rip + 0x20161f], eax
|     |||   0x00400ac5    488b45e8       mov rax, qword [rbp-bytecode]
|     |||   0x00400ac9    488d5002       lea rdx, [rax + 2]
|     |||   0x00400acd    488b45e8       mov rax, qword [rbp-bytecode]
|     |||   0x00400ad1    4883c001       add rax, 1
|     |||   0x00400ad5    4889d6         mov rsi, rdx
|     |||   0x00400ad8    4889c7         mov rdi, rax
|     |||   0x00400adb    e8bffdffff     call fcn.0040089f
|     |||   0x00400ae0    c745f4030000.  mov dword [rbp-instr_ptr_step], 3
|    ,====< 0x00400ae7    e929010000     jmp 0x400c15
|    ||||   ;-- instr_I:
|    ||||   0x00400aec    8b05f6152000   mov eax, dword [rip + 0x2015f6]
|    ||||   0x00400af2    83c001         add eax, 1
|    ||||   0x00400af5    8905ed152000   mov dword [rip + 0x2015ed], eax
|    ||||   0x00400afb    488b45e8       mov rax, qword [rbp-bytecode]
|    ||||   0x00400aff    4883c001       add rax, 1
|    ||||   0x00400b03    4889c7         mov rdi, rax
|    ||||   0x00400b06    e856feffff     call fcn.00400961
|    ||||   0x00400b0b    c745f4020000.  mov dword [rbp-instr_ptr_step], 2
|   ,=====< 0x00400b12    e9fe000000     jmp 0x400c15
|   |||||   ;-- instr_D:
|   |||||   0x00400b17    8b05cf152000   mov eax, dword [rip + 0x2015cf]
|   |||||   0x00400b1d    83c001         add eax, 1
|   |||||   0x00400b20    8905c6152000   mov dword [rip + 0x2015c6], eax
|   |||||   0x00400b26    488b45e8       mov rax, qword [rbp-bytecode]
|   |||||   0x00400b2a    4883c001       add rax, 1
|   |||||   0x00400b2e    4889c7         mov rdi, rax
|   |||||   0x00400b31    e806feffff     call fcn.0040093c
|   |||||   0x00400b36    c745f4020000.  mov dword [rbp-instr_ptr_step], 2
|  ,======< 0x00400b3d    e9d3000000     jmp 0x400c15
|  ||||||   ;-- instr_P:
|  ||||||   0x00400b42    8b05a8152000   mov eax, dword [rip + 0x2015a8]
|  ||||||   0x00400b48    83c001         add eax, 1
|  ||||||   0x00400b4b    89059f152000   mov dword [rip + 0x20159f], eax
|  ||||||   0x00400b51    488b45e8       mov rax, qword [rbp-bytecode]
|  ||||||   0x00400b55    4883c001       add rax, 1
|  ||||||   0x00400b59    4889c7         mov rdi, rax
|  ||||||   0x00400b5c    e825feffff     call fcn.00400986
|  ||||||   0x00400b61    c745f4020000.  mov dword [rbp-instr_ptr_step], 2
| ,=======< 0x00400b68    e9a8000000     jmp 0x400c15
| |||||||   ;-- instr_C:
| |||||||   0x00400b6d    8b0581152000   mov eax, dword [rip + 0x201581]
| |||||||   0x00400b73    83c001         add eax, 1
| |||||||   0x00400b76    890578152000   mov dword [rip + 0x201578], eax
| |||||||   0x00400b7c    488b45e8       mov rax, qword [rbp-bytecode]
| |||||||   0x00400b80    4883c001       add rax, 1
| |||||||   0x00400b84    0fb600         movzx eax, byte [rax]
| |||||||   0x00400b87    0fbec0         movsx eax, al
| |||||||   0x00400b8a    890530152000   mov dword [rip + 0x201530], eax
| |||||||   0x00400b90    c745f4020000.  mov dword [rbp-instr_ptr_step], 2
| ========< 0x00400b97    eb7c           jmp 0x400c15
| |||||||   ;-- instr_X:
| |||||||   0x00400b99    8b0559152000   mov eax, dword [rip + 0x201559]
| |||||||   0x00400b9f    83c001         add eax, 1
| |||||||   0x00400ba2    890550152000   mov dword [rip + 0x201550], eax
| |||||||   0x00400ba8    488b45e8       mov rax, qword [rbp-bytecode]
| |||||||   0x00400bac    4883c001       add rax, 1
| |||||||   0x00400bb0    4889c7         mov rdi, rax
| |||||||   0x00400bb3    e867feffff     call fcn.00400a1f
| |||||||   0x00400bb8    c745f4020000.  mov dword [rbp-instr_ptr_step], 2
| ========< 0x00400bbf    eb54           jmp 0x400c15
| |||||||   ;-- instr_J:
| |||||||   0x00400bc1    8b0535152000   mov eax, dword [rip + 0x201535]
| |||||||   0x00400bc7    83c001         add eax, 1
| |||||||   0x00400bca    89052c152000   mov dword [rip + 0x20152c], eax
| |||||||   0x00400bd0    488b45e8       mov rax, qword [rbp-bytecode]
| |||||||   0x00400bd4    4883c001       add rax, 1
| |||||||   0x00400bd8    4889c7         mov rdi, rax
| |||||||   0x00400bdb    e8d8fdffff     call fcn.004009b8
| |||||||   0x00400be0    8945f4         mov dword [rbp-instr_ptr_step], eax
| ========< 0x00400be3    eb30           jmp 0x400c15
| |||||||   ;-- instr_R:
| |||||||   0x00400be5    8b0515152000   mov eax, dword [rip + 0x201515]
| |||||||   0x00400beb    83c001         add eax, 1
| |||||||   0x00400bee    89050c152000   mov dword [rip + 0x20150c], eax
| |||||||   0x00400bf4    488b45e8       mov rax, qword [rbp-bytecode]
| |||||||   0x00400bf8    4883c001       add rax, 1
| |||||||   0x00400bfc    0fb600         movzx eax, byte [rax]
| |||||||   0x00400bff    0fbec0         movsx eax, al
| ========< 0x00400c02    eb5d           jmp 0x400c61
| ||||||    ;-- not_instr:
| |||||`--> 0x00400c04    bf980e4000     mov edi, str.Wrong_
| ||||| |   0x00400c09    e862faffff     call sym.imp.puts
| ||||| |   0x00400c0e    b800000000     mov eax, 0
| ========< 0x00400c13    eb4c           jmp 0x400c61
| `````---> 0x00400c15    8b45f4         mov eax, dword [rbp-instr_ptr_step]
|       |   0x00400c18    4898           cdqe
|       |   0x00400c1a    480145e8       add qword [rbp-bytecode], rax
|       |   0x00400c1e    488b05631420.  mov rax, qword [rip + 0x201463]
|       |   0x00400c25    483d20216000   cmp rax, sym.memory
| ========< 0x00400c2b    720f           jb 0x400c3c
|       |   0x00400c2d    488b05541420.  mov rax, qword [rip + 0x201454]
|       |   0x00400c34    483d20236000   cmp rax, section_end..bss
| ========< 0x00400c3a    7211           jb 0x400c4d
| --------> 0x00400c3c    bf9f0e4000     mov edi, str.We_are_in_the_outer_space_
|       |   0x00400c41    e82afaffff     call sym.imp.puts
|       |   0x00400c46    b800000000     mov eax, 0
| ========< 0x00400c4b    eb14           jmp 0x400c61
| ------`-> 0x00400c4d    488b45e8       mov rax, qword [rbp-bytecode]
|           0x00400c51    0fb600         movzx eax, byte [rax]
|           0x00400c54    84c0           test al, al
|           0x00400c56    0f8502feffff   jne 0x400a5e
|           0x00400c5c    b801000000     mov eax, 1
| --------> 0x00400c61    c9             leave
\           0x00400c62    c3             ret
[0x00400ec0]>
```

As I've mentioned previously, the function itself is pretty short, and easy to read, especially with our annotations. But a promise is a promise, so here is how we can create the missing bacic blocks for the instructions:

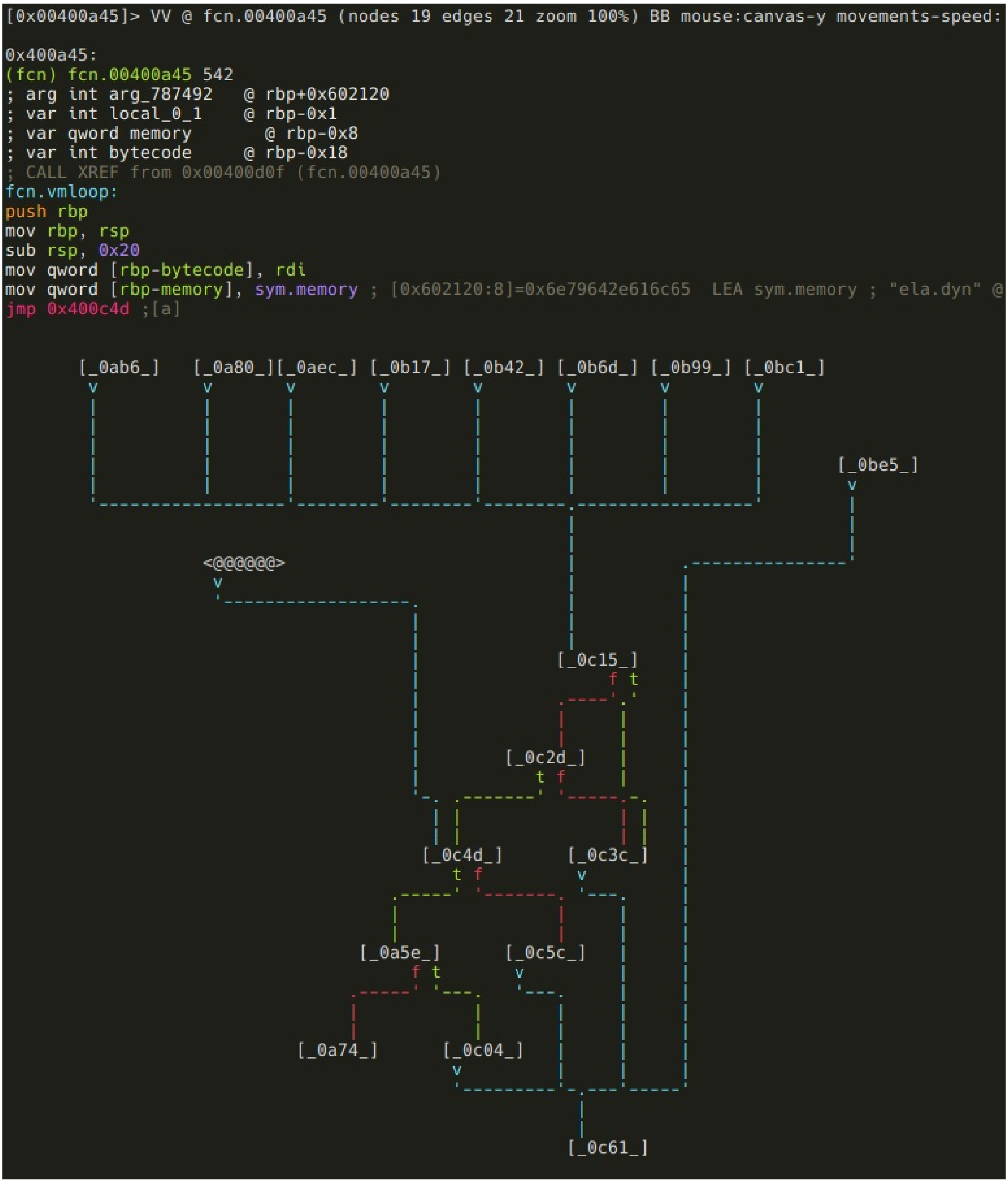
```
[0x00400ec0]> afb+ 0x00400a45 0x00400a80 0x00400ab6-0x00400a80 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400ab6 0x00400aec-0x00400ab6 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400aec 0x00400b17-0x00400aec 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400b17 0x00400b42-0x00400b17 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400b42 0x00400b6d-0x00400b42 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400b6d 0x00400b99-0x00400b6d 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400b99 0x00400bc1-0x00400b99 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400bc1 0x00400be5-0x00400bc1 0x400c15
[0x00400ec0]> afb+ 0x00400a45 0x00400be5 0x00400c04-0x00400be5 0x400c15
```

It is also apparent from the disassembly that besides the instructions there are three more basic blocks. Lets create them too!

```
[0x00400ec0]> afb+ 0x00400a45 0x00400c15 0x00400c2d-0x00400c15 0x400c3c 0x00400c2d
[0x00400ec0]> afb+ 0x00400a45 0x00400c2d 0x00400c3c-0x00400c2d 0x400c4d 0x00400c3c
[0x00400ec0]> afb+ 0x00400a45 0x00400c3c 0x00400c4d-0x00400c3c 0x400c61
```

Note that the basic blocks starting at 0x00400c15 and 0x00400c2d ending in a conditional jump, so we had to set the false branch's destination too!

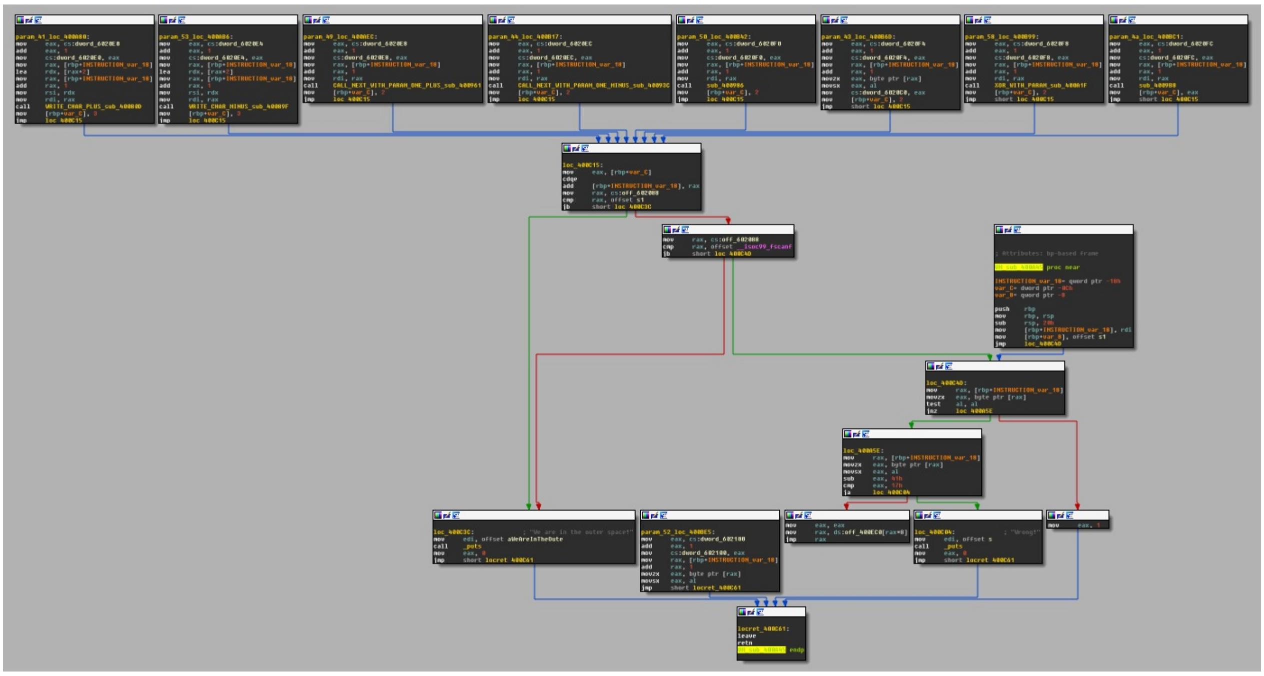
And here is the graph in its full glory after a bit of manual restructuring:



I think it worth it, don't you? :) (Well, the restructuring did not really worth it, because it is apparently not stored when you save the project.)

> ***r2 tip***: You can move the selected node around in graph view using the HJKL keys.

BTW, here is how IDA's graph of this same function looks like for comparison:



As we browse through the disassembly of the *instr_LETTER* basic blocks, we should realize a few things. The first: all of the instructions starts with a sequence like these:

```
0x400a80:
instr_A:
mov eax, dword [rip + 0x20165a] ; [0x6020e0:4]=0x646c6975  LEA sym.instr_dirty ; "uild-id" @ 0x6
add eax, 1
mov dword [rip + 0x201651], eax ; [0x6020e0:4]=0x646c6975  LEA sym.instr_dirty ; "uild-id" @ 0x6
```

```
0x400ab6:
instr_S:
mov eax, dword [rip + 0x201628] ; [0x6020e4:4]=0x64692d  ; "-id" @ 0x6020e4
add eax, 1
mov dword [rip + 0x20161f], eax ; [0x6020e4:4]=0x64692d  ; "-id" @ 0x6020e4
```

It became clear now that the 9 dwords at *sym.instr_dirty* are not simply indicators that an instruction got executed, but they are used to count how many times an instruction got called. Also I should have realized earlier that *sym.good_if_le_9* (0x6020f0) is part of this 9 dword array, but yeah, well, I didn't, I have to live with it... Anyways, what the condition "*sym.good_if_le_9* have to be lesser or equal 9" really means is that *instr_P* can not be executed more than 9 times:

```
0x400b42:
instr_P:
mov eax, dword [rip + 0x2015a8] ; [0x6020f0:4]=0x642e0068  LEA sym.good_if_le_9 ; "h" @ 0x6020f0
add eax, 1
mov dword [rip + 0x20159f], eax ; [0x6020f0:4]=0x642e0068  LEA sym.good_if_le_9 ; "h" @ 0x6020f0
```

Another similarity of the instructions is that 7 of them calls a function with either one or two parameters, where the parameters are the next, or the next two bytecodes. One parameter example:

```
0x400aec:
instr_I:
mov eax, dword [rip + 0x2015f6] ; [0x6020e8:4]=0x756e672e  ; ".gnu.hash" @ 0x6020e8
add eax, 1
mov dword [rip + 0x2015ed], eax ; [0x6020e8:4]=0x756e672e  ; ".gnu.hash" @ 0x6020e8
mov rax, qword [rbp-bytecode]
add rax, 1
mov rdi, rax
call fcn.00400961 ;[i]
mov dword [rbp-instr_ptr_step], 2
jmp 0x400c15 ;[d]
```

And a two parameters example:

```
0x400a80:
instr_A:
mov eax, dword [rip + 0x20165a] ; [0x6020e0:4]=0x646c6975  LEA sym.instr_dirty ; "uild-id" @ 0x6
add eax, 1
mov dword [rip + 0x201651], eax ; [0x6020e0:4]=0x646c6975  LEA sym.instr_dirty ; "uild-id" @ 0x6
mov rax, qword [rbp-bytecode]
lea rdx, [rax + 2] ; 0x2
mov rax, qword [rbp-bytecode]
add rax, 1
mov rsi, rdx
mov rdi, rax
call fcn.0040080d ;[c]
mov dword [rbp-instr_ptr_step], 3
jmp 0x400c15 ;[d]
```

We should also realize that these blocks put the number of bytes they eat up of the bytecode (1 byte instruction + 1 or 2 bytes arguments = 2 or 3) into a local variable at 0xc. r2 did not recognize this local var, so lets do it manually!

```
:> afv 0xc instr_ptr_step dword
```

If we look at *instr_J* we can see that this is an exception to the above rule, since it puts the return value of the called function into *instr_ptr_step* instead of a constant 2 or 3:

```
0x400bc1:
instr_J:
mov eax, dword [rip + 0x201535] ; [0x6020fc:4]=0x74736e79  ; "ynstr" @ 0x6020fc
add eax, 1
mov dword [rip + 0x20152c], eax ; [0x6020fc:4]=0x74736e79  ; "ynstr" @ 0x6020fc
mov rax, qword [rbp-bytecode]
add rax, 1
mov rdi, rax
call fcn.004009b8 ;[m]
mov dword [rbp-instr_ptr_step], eax
jmp 0x400c15 ;[d]
```

And speaking of exceptions, here are the two instructions that do not call functions:

```
0x400be5:
instr_R:
mov eax, dword [rip + 0x201515] ; [0x602100:4]=0x672e0072  ; "r" 0x00602100  ; "r" @ 0x602100
add eax, 1
mov dword [rip + 0x20150c], eax ; [0x602100:4]=0x672e0072  ; "r" 0x00602100  ; "r" @ 0x602100
mov rax, qword [rbp-bytecode]
add rax, 1
movzx eax, byte [rax]
movsx eax, al
jmp 0x400c61 ;[f]
```

This one simply puts the next bytecode (the first the argument) into *eax*, and jumps to the end of *vmloop*. So this is the VM's *ret* instruction, and we know that *vmloop* has to return "*", so "R*" should be the last two bytes of our bytecode.

The next one that does not call a function:

```
0x400b6d:
instr_C:
mov eax, dword [rip + 0x201581] ; [0x6020f4:4]=0x79736e79  ; "ynsym" @ 0x6020f4
add eax, 1
mov dword [rip + 0x201578], eax ; [0x6020f4:4]=0x79736e79  ; "ynsym" @ 0x6020f4
mov rax, qword [rbp-bytecode]
add rax, 1
movzx eax, byte [rax]
movsx eax, al
mov dword [rip + 0x201530], eax ; [0x6020c0:4]=0x65746e69  LEA sym.written_by_instr_C ; "interp"
mov dword [rbp-instr_ptr_step], 2
jmp 0x400c15 ;[d]
```

This is a one argument instruction, and it puts its argument to 0x6020c0. Flag that address!

```
:> f sym.written_by_instr_C 4 @ 0x6020c0
```

Oh, and by the way, I do have a hunch that *instr_C* also had a function call in the original code, but it got inlined by the compiler. Anyways, so far we have these two instructions:

- *instr_R(a1):* returns with *a1*
- *instr_C(a1):* writes *a1* to *sym.written_by_instr_C*

And we also know that these accept one argument,

- instr_I
- instr_D
- instr_P
- instr_X
- instr_J

and these accept two:

- instr_A
- instr_S

What remains is the reversing of the seven functions that are called by the instructions, and finally the construction of a valid bytecode that gives us the flag.

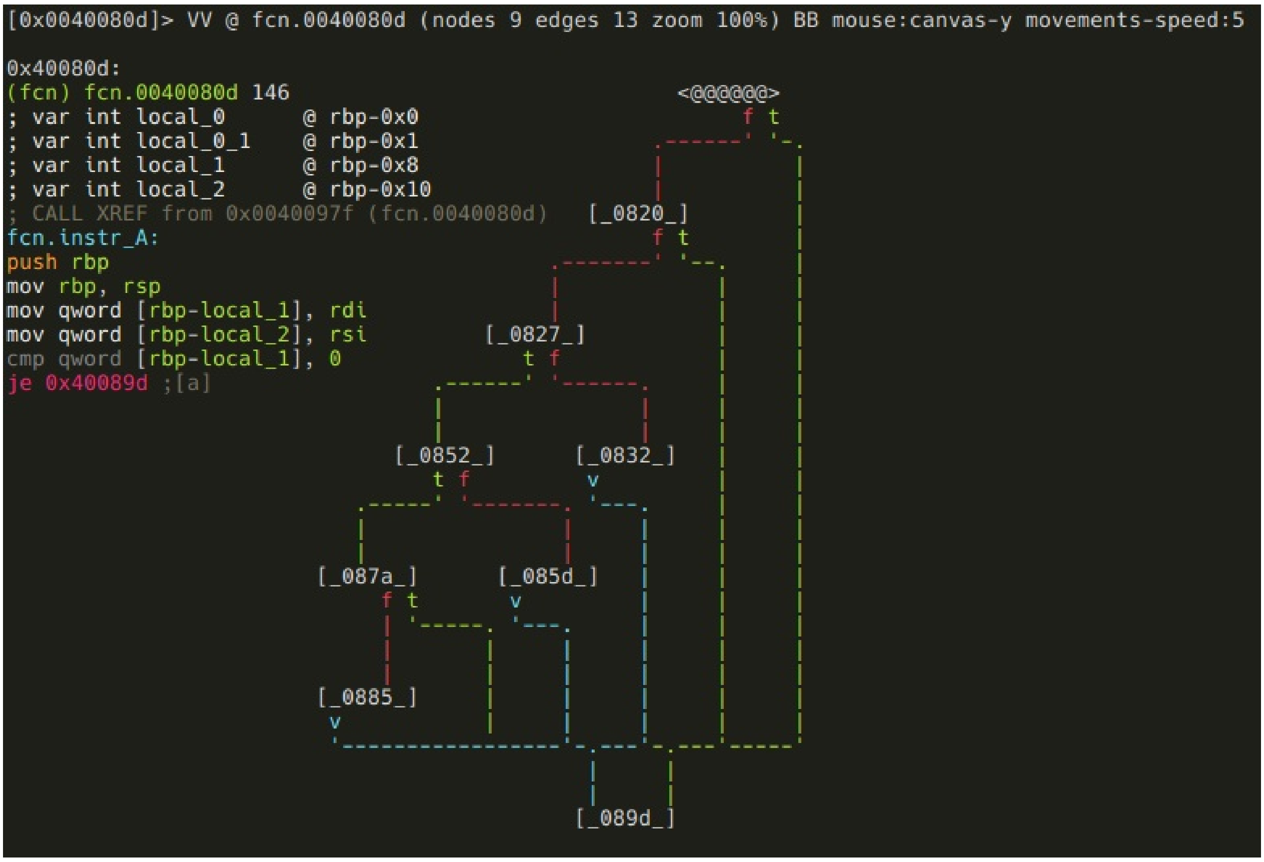
instr_A

The function this instruction calls is at offset 0x40080d, so lets seek there!

```
[offset]> 0x40080d
```

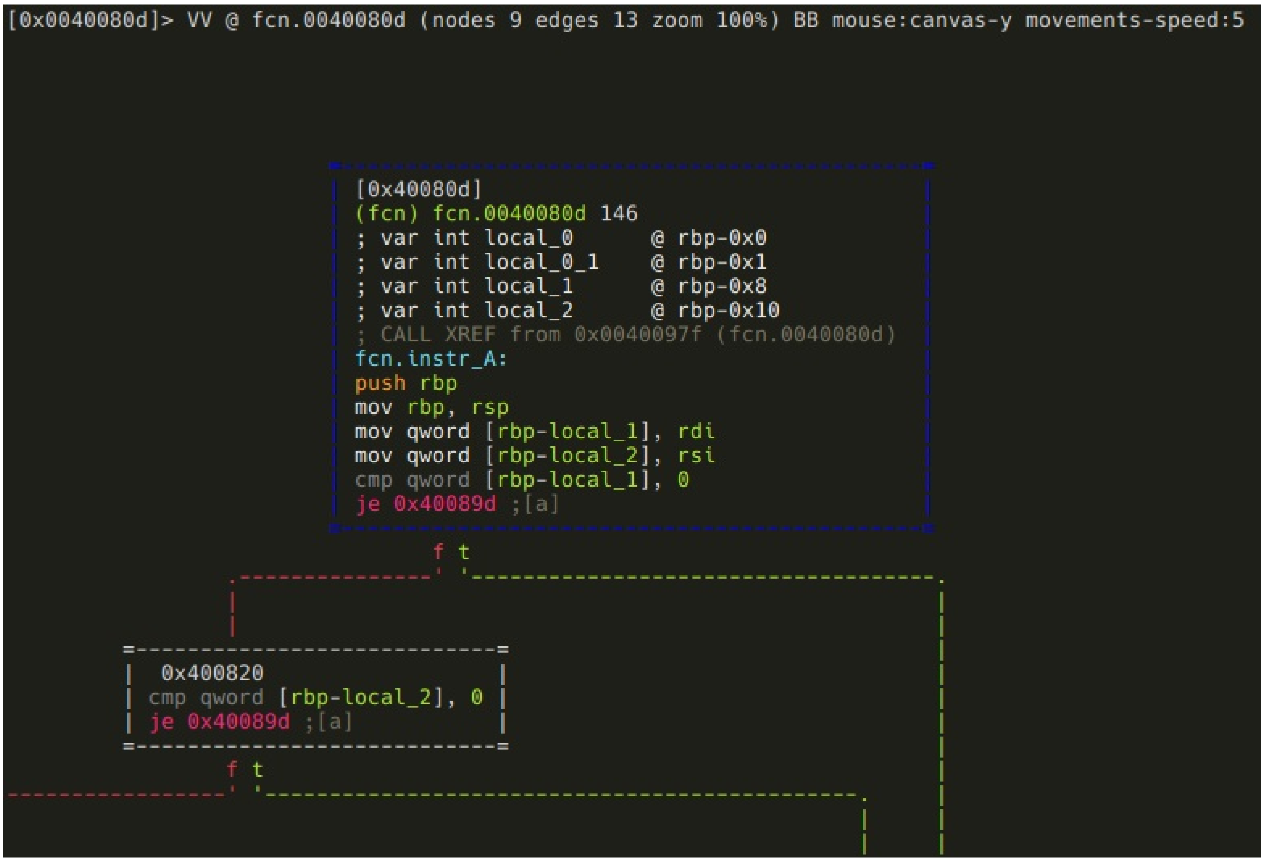
> ***r2 tip:*** In visual mode you can just hit \ when the current line is a jump or a call, and r2 will seek to the destination address.

If we seek to that address from the graph mode, we are presented with a message that says "Not in a function. Type 'df' to define it here. This is because the function is called from a basic block r2 did not recognize, so r2 could not find the function either. Lets obey, and type *df*! A function is indeed created, but we want some meaningful name for it. So press *dr* while still in visual mode, and name this function *instr_A*!



> ***r2 tip:*** You should realize that these commands are all part of the same menu system in visual mode I was talking about when we first used *Cd* to declare *sym.memory* as data.

Ok, now we have our shiny new *fcn.instr_A*, lets reverse it! We can see from the shape of the minimap that probably there is some kind cascading if-then-elif, or a switch-case statement involved in this function. This is one of the reasons the minimap is so useful: you can recognize some patterns at a glance, which can help you in your analysis (remember the easily recognizable for loop from a few paragraphs before?) So, the minimap is cool and useful, but I've just realized that I did not yet show you the full graph mode, so I'm going to do this using full graph. The first basic blocks:

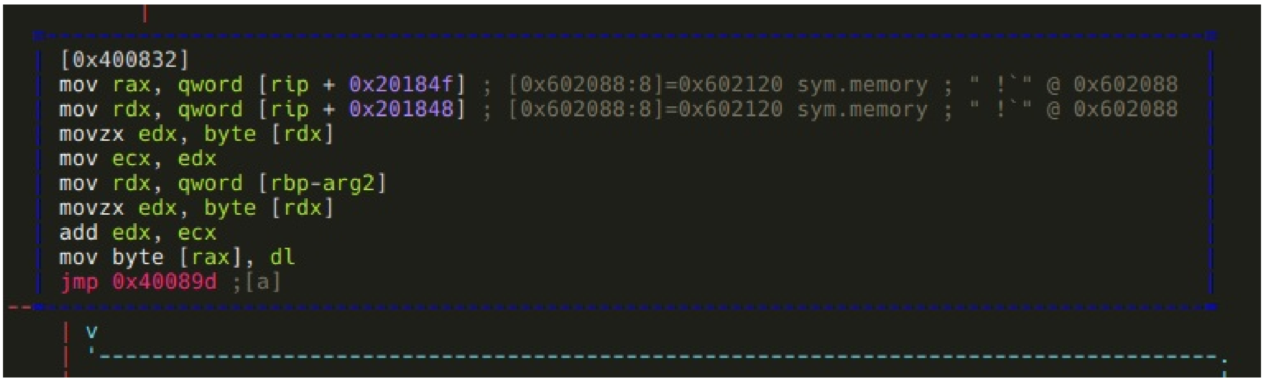


The two function arguments (*rdi* and *rsi*) are stored in local variables, and the first is compared to 0. If it is, the function returns (you can see it on the minimap), otherwise the same check is executed on the second argument. The function returns from here too, if the argument is zero. Although this function is really tiny, I am going to stick with my methodology, and rename the local vars:

```
:> afvn local_1 arg1
:> afvn local_2 arg2
```

And we have arrived to the predicted switch-case statement, and we can see that *arg1*'s value is checked against "M", "P", and "C".

This is the "M" branch:



It basically loads an address from offset 0x602088 and adds *arg2* to the byte at that address. As r2 kindly shows us in a comment, 0x602088 initially holds the address of *sym.memory*, the area where we have to construct the "Such VM! MuCH reV3rse!" string. It is safe to assume that somehow we will be able to modify the value stored at 0x602088, so this "M" branch will be able to modify bytes other than the first. Based on this assumption, I'll flag 0x602088 as *sym.current_memory_ptr*:

```
:> f sym.current_memory_ptr 8 @ 0x602088
```

Moving on to the "P" branch:

```
[0x40085d]
mov rdx, qword [rip + 0x201824] ; [0x602088:8]=0x602120 sym.memory LEA sym.current_memory
mov rax, qword [rbp-arg2]
movzx eax, byte [rax]
movzx eax, al
add rax, rdx
mov qword [rip + 0x201810], rax ; [0x602088:8]=0x602120 sym.memory LEA sym.current_memory
jmp 0x40089d ;[a]
```

Yes, this is the piece of code that allows us to modify *sym.current_memory_ptr*: it adds *arg2* to it.

Finally, the "C" branch:

```
[0x400885]
mov rax, qword [rbp-arg2]
movzx eax, byte [rax]
movzx edx, al
mov eax, dword [rip + 0x20182b] ; [0x6020c0:4]=0x65746e69  LEA sym.written_by_instr_C ; "i
add eax, edx
mov dword [rip + 0x201823], eax ; [0x6020c0:4]=0x65746e69  LEA sym.written_by_instr_C ; "i
```

Well, it turned out that *instr_C* is not the only instruction that modifies *sym.written_by_instr_C*: this piece of code adds *arg2* to it.

And that was *instr_A*, lets summarize it! Depending on the first argument, this instruction does the following:

- *arg1* == "M": adds *arg2* to the byte at *sym.current_memory_ptr*.
- *arg1* == "P": steps *sym.current_memory_ptr* by *arg2* bytes.
- *arg1* == "C": adds *arg2* to the value at *sym.written_by_instr_C*.

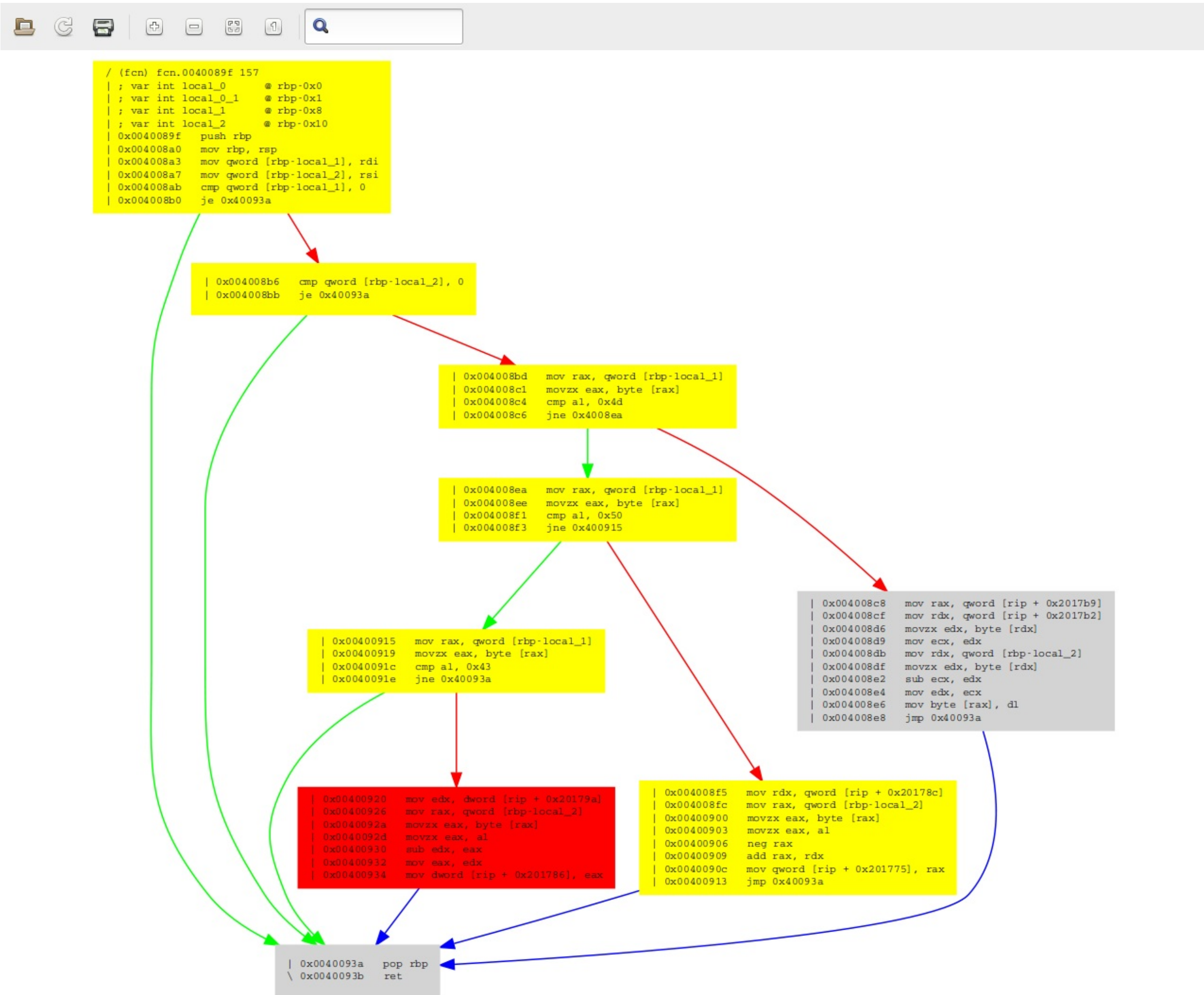
instr_S

This function is not recognized either, so we have to manually define it like we did with *instr_A*. After we do, and take a look at the minimap, scroll through the basic blocks, it is pretty obvious that these two functions are very-very similar. We can use *radiff2* to see the difference.

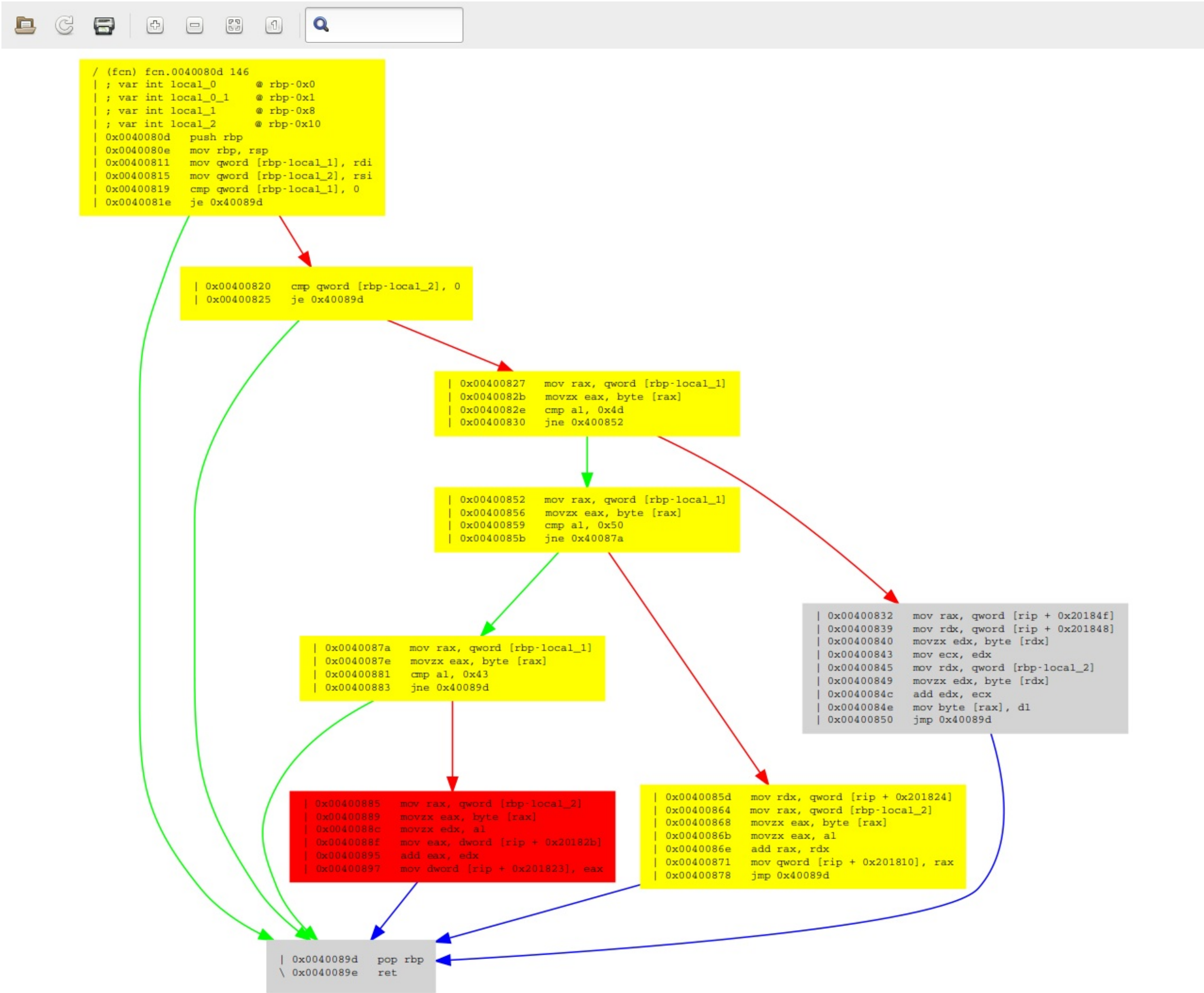
> ***r2 tip:*** radiff2 is used to compare binary files. There's a few options we can control the type of binary diffing the tool does, and to what kind of output format we want. One of the cool features is that it can generate DarumGrim-style bindiff graphs using the *-g* option.

Since now we want to diff two functions from the same binary, we specify the offsets with *-g*, and use reverse4 for both binaries. Also, we create the graphs for comparing *instr_A* to *instr_S* and for comparing *instr_S* to *instr_A*.

```
[0x00 ~]$ radiff2 -g 0x40080d,0x40089f  reverse4 reverse4 | xdot -
```




```
[0x00 ~]$ radiff2 -g 0x40089f,0x40080d  reverse4 reverse4 | xdot -
```



A sad truth reveals itself after a quick glance at these graphs: radiff2 is a liar! In theory, grey boxes should be identical, yellow ones should differ only at some offsets, and red ones should differ seriously. Well this is obviously not the case here - e.g. the larger grey boxes are clearly not identical. This is something I'm definitely going to take a deeper look at after I've finished this writeup.

Anyways, after we get over the shock of being lied to, we can easily recognize that *instr_S* is basically a reverse-*instr_A*: where the latter does addition, the former does subtraction. To summarize this:

- *arg1* == "M": subtracts *arg2* from the byte at *sym.current_memory_ptr*.
- *arg1* == "P": steps *sym.current_memory_ptr* backwards by *arg2* bytes.
- *arg1* == "C": subtracts *arg2* from the value at *sym.written_by_instr_C*.

instr_I

```
[0x00400961]> VV @ fcn.00400961 (nodes 1 edges 0 zoom 100%) BB mouse:canvas-y movements-speed:5

0x400961:
(fcn) fcn.00400961 37
; var int local_0_1    @ rbp-0x1
; var int local_3      @ rbp-0x18
; CALL XREF from 0x004009b1 (fcn.00400961)
fcn.instr_I:
push rbp                                       <@@@@@@>
mov rbp, rsp
sub rsp, 0x18
mov qword [rbp-local_3], rdi
mov byte [rbp-local_0_1], 1
lea rdx, [rbp-local_0_1]
mov rax, qword [rbp-local_3]
mov rsi, rdx
mov rdi, rax
call fcn.0040080d ;[a]
leave
ret
```

This one is simple, it just calls *instr_A(arg1, 1)*. As you may have noticed the function call looks like `call fcn.0040080d` instead of `call fcn.instr_A`. This is because when you save and open a project, function names get lost - another thing to examine and patch in r2!

instr_D

```
[0x0040093c]> VV @ fcn.0040093c (nodes 1 edges 0 zoom 100%) BB mouse:canvas-y movements-speed:5

0x40093c:
(fcn) fcn.0040093c 37
; var int local_0_1    @ rbp-0x1
; var int local_3      @ rbp-0x18
fcn.instr_D:
push rbp
mov rbp, rsp                                   <@@@@@@>
sub rsp, 0x18
mov qword [rbp-local_3], rdi
mov byte [rbp-local_0_1], 1
lea rdx, [rbp-local_0_1]
mov rax, qword [rbp-local_3]
mov rsi, rdx
mov rdi, rax
call fcn.0040089f ;[a]
leave
ret
```

Again, simple: it calls *instr_S(arg1, 1)*.

instr_P

It's local var rename time again!

```
:> afvn local_0_1 const_M
:> afvn local_0_2 const_P
:> afvn local_3 arg1
```

```
[0x00400986]> VV @ fcn.00400986 (nodes 1 edges 0 zoom 100%) BB mouse:canvas-y movements-speed:5

0x400986:
(fcn) fcn.00400986 50
; arg int arg_9_5      @ rbp+0x4d
; arg int arg_10       @ rbp+0x50
; var int const_M      @ rbp-0x1
; var int const_P      @ rbp-0x2
; var int arg1         @ rbp-0x18             <@@@@@@>
fcn.instr_P:
push rbp
mov rbp, rsp
sub rsp, 0x18
mov qword [rbp-arg1], rdi
mov byte [rbp-const_M], 0x4d ; [0x4d:1]=0 ; 'M'
mov byte [rbp-const_P], 0x50 ; [0x50:1]=64 ; 'P'
mov rax, qword [rip + 0x2016e7] ; [0x602088:8]=0x602120 sym.memory LEA sym.current_memory_ptr ;
mov rdx, qword [rbp-arg1]
movzx edx, byte [rdx]
mov byte [rax], dl
lea rax, [rbp-const_P]
mov rdi, rax
call fcn.00400961 ;[a]
leave
ret
```

This function is pretty straightforward also, but there is one oddity: const_M is never used. I don't know why it is there - maybe it is supposed to be some kind of distraction? Anyways, this function simply writes *arg1* to *sym.current_memory_ptr*, and than calls *instr_I("P")*. This basically means that *instr_P* is used to write one byte, and put the pointer to the next byte. So far this would seem the ideal instruction to construct most of the "Such VM! MuCH reV3rse!" string, but remember, this is also the one that can be used only 9 times!

instr_X

Another simple one, rename local vars anyways!

```
:> afvn local_1 arg1
```

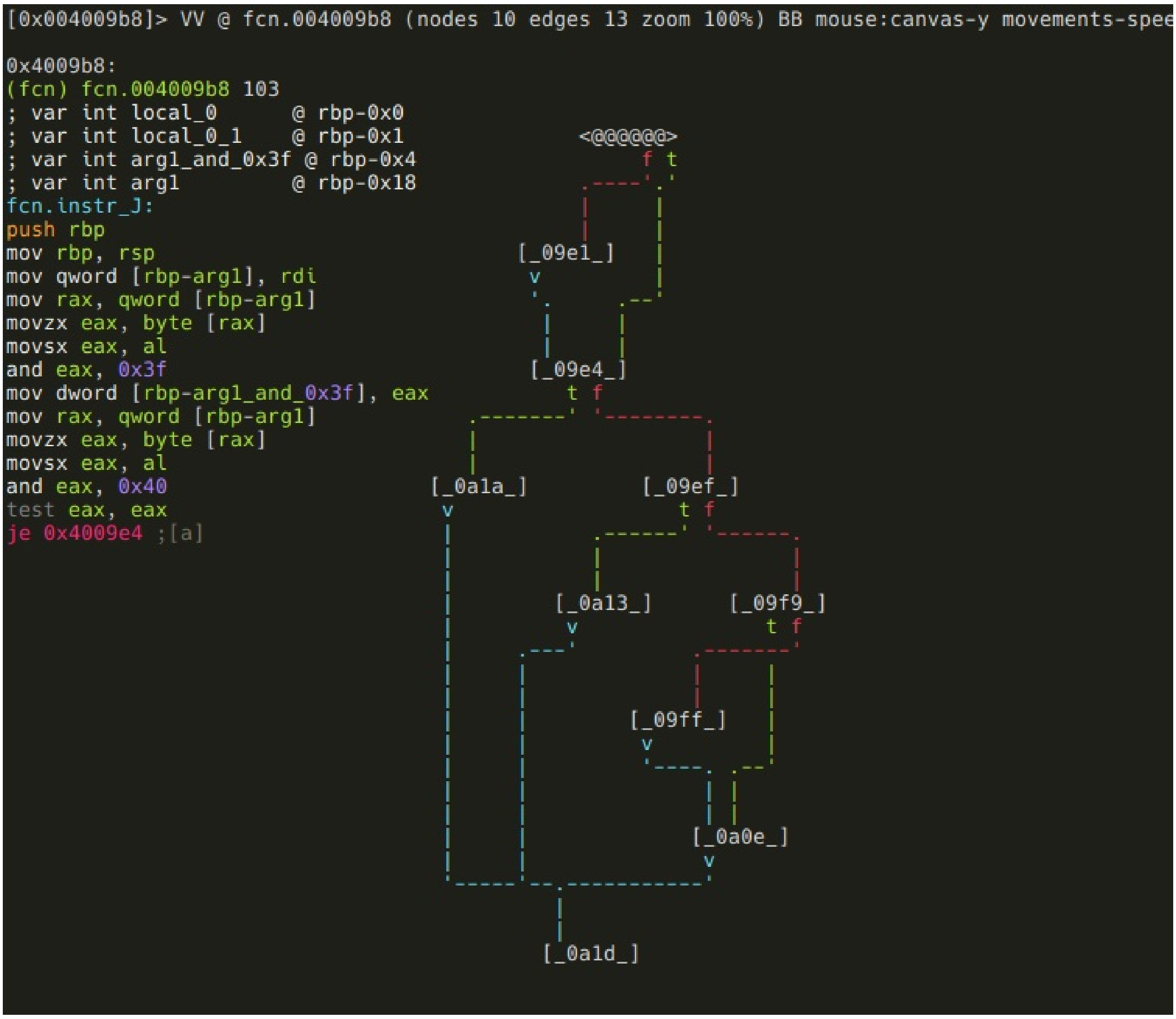
```
[0x00400a1f]> VV @ fcn.00400a1f (nodes 1 edges 0 zoom 100%) BB mouse:canvas-y movements-speed:5

0x400a1f:
(fcn) fcn.00400a1f 38
; var int local_0_1    @ rbp-0x1
; var int arg1         @ rbp-0x8
fcn.instr_X:
push rbp
mov rbp, rsp                                  <@@@@@@>
mov qword [rbp-arg1], rdi
mov rax, qword [rip + 0x20165a] ; [0x602088:8]=0x602120 sym.memory LEA sym.current_memory_ptr ;
mov rdx, qword [rip + 0x201653] ; [0x602088:8]=0x602120 sym.memory LEA sym.current_memory_ptr ;
movzx ecx, byte [rdx]
mov rdx, qword [rbp-arg1]
movzx edx, byte [rdx]
xor edx, ecx
mov byte [rax], dl
pop rbp
ret
```

This function XORs the value at *sym.current_memory_ptr* with *arg1*.

instr_J

This one is not as simple as the previous ones, but it's not that complicated either. Since I'm obviously obsessed with variable renaming:

```
:> afvn local_3 arg1
:> afvn local_0_4 arg1_and_0x3f
```



After the result of *arg1 & 0x3f* is put into a local variable, *arg1 & 0x40* is checked against 0. If it isn't zero, *arg1_and_0x3f* is negated:

The next branching: if *arg1* >= 0, then the function returns *arg1_and_0x3f*,

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed

0x4009e4:
mov rax, qword [rbp-arg1]
movzx eax, byte [rax]
test al, al
jns 0x400a1a ;[b]
```

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed

0x400a1a:
mov eax, dword [rbp-arg1_and_0x3f]
```

else the function branches again, based on the value of *sym.written_by_instr_C*:

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed

0x4009ef:
mov eax, dword [rip + 0x2016cb] ; [0x6020c0:4]=0x65746e69  LEA sym.written_by_instr_C ; "interp
test eax, eax
je 0x400a13 ;[c]
```

If it is zero, the function returns 2,

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed

0x400a13:
mov eax, 2
jmp 0x400a1d ;[d]
```

else it is checked if *arg1_and_0x3f* is a negative number,

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed

0x4009f9:
cmp dword [rbp-arg1_and_0x3f], 0
jns 0x400a0e ;[e]
```

and if it is, *sym.good_if_ne_zero* is incremented by 1:

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed

0x4009ff:
mov eax, dword [rip + 0x2016ff] ; [0x602104:4]=0x762e756e  LEA sym.good_if_ne_zero ; "nu.versio
add eax, 1
mov dword [rip + 0x2016f6], eax ; [0x602104:4]=0x762e756e  LEA sym.good_if_ne_zero ; "nu.versio
```

After all this, the function returns with *arg1_and_0x3f*:

```
[0x004009b8]> VV @ fcn.004009b8 (nodes 10 edges 13 zoom 100%) BB mouse:canvas-y movements-speed
                                                              |         |                       | |
0x400a0e:                                                     |         |                       | |
mov eax, dword [rbp-arg1_and_0x3f]                            |         |                    <@@@@@@>
jmp 0x400a1d ;[d]                                             |         |                       v
                                                              '-----'--.-----------'
                                                                       |
                                                                       |
                                                                   [_0a1d_]
```

.instructionset

We've now reversed all the VM instructions, and have a full understanding about how it works. Here is the VM's instruction set:

Instruction	1st arg	2nd arg	What does it do?
"A"	"M"	arg2	*sym.current_memory_ptr += arg2
	"P"	arg2	sym.current_memory_ptr += arg2
	"C"	arg2	sym.written_by_instr_C += arg2
"S"	"M"	arg2	*sym.current_memory_ptr -= arg2
	"P"	arg2	sym.current_memory_ptr -= arg2
	"C"	arg2	sym.written_by_instr_C -= arg2
"I"	arg1	n/a	instr_A(arg1, 1)
"D"	arg1	n/a	instr_S(arg1, 1)
"P"	arg1	n/a	*sym.current_memory_ptr = arg1; instr_I("P")
"X"	arg1	n/a	*sym.current_memory_ptr ^= arg1

"J"	arg1	n/a	arg1_and_0x3f = arg1 & 0x3f;
			if (arg1 & 0x40 != 0)
			  arg1_and_0x3f *= -1
			if (arg1 >= 0) return arg1_and_0x3f;
			else if (*sym.written_by_instr_C != 0) {
			  if (arg1_and_0x3f < 0)
			    ++*sym.good_if_ne_zero;
			  return arg1_and_0x3f;
			} else return 2;

"C"	arg1	n/a	*sym.written_by_instr_C = arg1
"R"	arg1	n/a	return(arg1)

.bytecode

Well, we did the reverse engineering part, now we have to write a program for the VM with the instruction set described in the previous paragraph. Here is the program's functional specification:

- the program must return "*"
- *sym.memory* has to contain the string "Such VM! MuCH reV3rse!" after execution
- all 9 instructions have to be used at least once
- *sym.good_if_ne_zero* should not be zero
- instr_P is not allowed to be used more than 9 times

Since this document is about reversing, I'll leave the programming part to the fellow reader :) But I'm not going to leave you empty-handed, I'll give you one advice: Except for "J", all of the instructions are simple, easy to use, and it should not be a problem to construct the "Such VM! MuCH reV3rse!" using them. "J" however is a bit complicated compared to the others. One should realize that its sole purpose is to make *sym.good_if_ne_zero* bigger than zero, which is a requirement to access the flag. In order to increment *sym.good_if_ne_zero*, three conditions should be met:

- *arg1* should be a negative number, otherwise we would return early
- *sym.written_by_instr_C* should not be 0 when "J" is called. This means that "C", "AC", or "SC" instructions should be used before calling "J".
- *arg1_and_0x3f* should be negative when checked. Since 0x3f's sign bit is zero, no matter what *arg1* is, the result of *arg1* & 0x3f will always be non-negative. But remember that "J" negates *arg1_and_0x3f* if *arg1* & 0x40 is not zero. This basically means that *arg1*'s 6th bit should be 1 (0x40 = 01000000b). Also, because *arg1_and_0x3f* can't be 0 either, at least one of *arg1*'s 0th, 1st, 2nd, 3rd, 4th or 5th bits should be 1 (0x3f = 00111111b).

I think this is enough information, you can go now and write that program. Or, you could just reverse engineer the quick'n'dirty one I've used during the CTF:

```
\x90\x00PSAMuAP\x01AMcAP\x01AMhAP\x01AM AP\x01AMVAP\x01AMMAP\x01AM!AP\x01AM AP\x01AMMA
P\x01AMuAP\x01AMCAP\x01AMHAP\x01AM AP\x01AMrAP\x01AMeAP\x01AMVAP\x01AM3AP\x01AMrAP\x01
AMsAP\x01AMeIPAM!X\x00CAJ\xc1SC\x00DCR*
```

Keep in mind though, that it was written on-the-fly, parallel to the reversing phase - for example there are parts that was written without the knowledge of all possible instructions. This means that the code is ugly and unefficient.

.outro

Well, what can I say? Such VM, much reverse! :)

What started out as a simple writeup for a simple crackme, became a rather lengthy writeup/r2 tutorial, so kudos if you've read through it. I hope you enjoyed it (I know I did), and maybe even learnt something from it. I've surely learnt a lot about r2 during the process, and I've even contributed some small patches, and got a few ideas of more possible improvements.

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)
$I	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 ぼぺほま by Thanat0s - v0.1 -
```