cover `![cover](cover.jpg)` cover2

Contents

Introduction

This book is an updated version (started by maijin) of the original radare1 book (written by pancake). Which is actively maintained and updated by many contributors over the Internet.

Check the Github site to add new contents or fix typos:

- Github: https://github.com/radareorg/radare2book
- Online: https://book.rada.re/

History

In 2006, Sergi Àlvarez (aka pancake) was working as a forensic analyst. Since he wasn't allowed to use the company software for his personal needs, he decided to write a small tool-a hexadecimal editor-with very basic characteristics:

- be extremely portable (unix friendly, command line, c, small)
- open disk devices, this is using 64bit offsets
- search for a string or hexpair
- review and dump the results to disk

The editor was originally designed to recover a deleted file from an HFS+ partition.

After that, pancake decided to extend the tool to have a pluggable io to be able to attach to processes and implemented the debugger functionalities, support for multiple architectures, and code analysis.

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

The need for scripting showed the fragility of the initial design: a monolithic tool made the API hard to use, and so a deep refactoring was needed. In 2009 radare2 (r2) was born as a fork of radare1. The refactor added flexibility and dynamic features. This enabled much better integration, paving the way to use r2 from different programming languages. Later on, the r2pipe API allowed access to radare2 via pipes from any language.

What started as a one-man project, with some eventual contributions, gradually evolved into a big community-based project around 2014. The number of users was growing fast, and the author-and main developer-had to switch roles from coder to manager in order to integrate the work of the different developers that were joining the project.

Instructing users to report their issues allows the project to define new directions to evolve in. Everything is managed in radare2's GitHub and discussed in the Telegram channel.

The project remains active at the time of writing this book, and there are several side projects that provide, among other things, a graphical user interface (Cutter), a decompiler (r2dec, radeco), Frida integration (r2frida), Yara, Unicorn, Keystone, and many other projects indexed in the r2pm (the radare2 package manager).

Since 2016, the community gathers once a year in r2con, a congress around radare2 that takes place in Barcelona.

The Framework

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

This chapter will give you a quick understanding of them, but you can check the dedicated sections for each tool at the end of this book.

radare2

The main tool of the whole framework. It uses 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, and visualizing. It can be scripted with a variety of languages, including Python, Ruby, JavaScript, Lua, and Perl.

rabin2

A program to extract information from executable binaries, such as ELF, PE, Java CLASS, Mach-O, plus any format supported by r2 plugins. rabin2 is used by the core to get data like exported symbols, imports, file information, cross references (xrefs), library dependencies, and sections.

rasm2

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

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, and others. rahash2 can be used to check the integrity or track changes of big files, memory dumps, or disks.

Examples

```
$ rahash2 file
file: 0x00000000-0x00000007 sha256: 887cfbd0d44aaff69f7bdbedebd282ec96191cce9d7fa7336298a18efc3c7a5a
```

```
$ rahash2 -a md5 file
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.

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:

- Solving 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 with a Socket

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

Debugging a Program Redirecting the stdio into 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 /bin/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, and more. 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
```

Downloading radare2

You can get radare from the GitHub repository: https://github.com/radareorg/radare2

Binary packages are available for a number of operating systems (Ubuntu, Maemo, Gentoo, Windows, iPhone, and so on). But 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.

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

```
$ git clone https://github.com/radareorg/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 lose them!) with:

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

Or send us 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
```

Building with meson + ninja

There is also a work-in-progress support for Meson.

Using clang and ld.gold makes the build faster:

```
CC=clang LDFLAGS=-fuse-ld=gold meson . release --buildtype=release --prefix ~/.local/stow/radare2/release
ninja -C release
# ninja -C release install
```

Helper Scripts

Take a look at the scripts in `sys/`, they 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/local`, but you can specify a different prefix using the argument `--prefix`.

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), OS X, FreeBSD, NetBSD, and OpenBSD (Intel x86 and x86_64)..

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, WinDbg, or Wine as back-ends, and therefore rely on presence of corresponding third-party tools (in case of remote debugging - just on the target machine).

To build on a system using `acr` and `GNU Make` (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 radare2 statically along with all other tools with the command:

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

Meson build

You can use meson + ninja to build:

```
$ sys/meson.py --prefix=/usr --shared --install
```

If you want to build locally:

```
$ sys/meson.py --prefix=/home/$USER/r2meson --local --shared --install
```

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
```

Windows

Radare2 relies on the Meson build system generator to support compilation on all platforms, including Windows. Meson will generate a Visual Studio Solution, all the necessary project files, and wire up the Microsoft Visual C++ compiler for you.

> **tip** You can download nightly binaries from https://ci.appveyor.com/project/radareorg/radare2/history. Be sure to download only from `master` branch!

Prerequisites

- Visual Studio 2015 (or higher)
- Python 3
- Meson
- Git

Step-by-Step

Install Visual Studio 2015 (or higher) Visual Studio must be installed with a Visual C++ compiler, supporting C++ libraries, and the appropriate Windows SDK for the target platform version.

- In the Visual Studio 2015 installer, ensure `Programming Languages > Visual C++` is selected
- In the Visual Studio 2017+ installers, ensure the `Desktop development with C++` workload is selected

If you need a copy of Visual Studio, the Community versions are free and work great.

- Download Visual Studio 2015 Community (registration required)
- Download Visual Studio 2017 Community

Install Python 3 and Meson via Conda It is strongly recommended you install Conda — a Python environment management system — when working with Python on the Windows platform. This will isolate the Radare2 build environment from other installed Python versions and minimize potential conflicts.

Set Up Conda:

1. Download the appropriate Conda (Python 3.x) for your platform (https://conda.io/miniconda.html)
2. Install Conda with the recommended defaults

Create a Python Environment for Radare2 Follow these steps to create and activate a Conda environment named *r2*. All instructions from this point on will assume this name matches your environment, but you may change this if desired.

1. Start > Anaconda Prompt
2. `conda create -n r2 python=3`
3. `activate r2`

Any time you wish to enter this environment, open the Anaconda Prompt and re-issue `activate r2`. Conversely, `deactivate` will leave the environment.

Install Meson

1. Enter the Radare2 Conda environment, if needed (`activate r2`)
2. Download https://github.com/mesonbuild/meson/archive/master.zip
3. `pip install \path\to\downloaded\master.zip`
4. Verify Meson is version 0.48 or higher (`meson -v`)

Install Git for Windows All Radare2 code is managed via the Git version control system and hosted on GitHub.

Follow these steps to install Git for Windows.

1. Download Git for Windows (https://git-scm.com/download/win)

As you navigate the install wizard, we recommend you set these options when they appear: * Use a TrueType font in all console windows * Use Git from the Windows Command Prompt * Use the native Windows Secure Channel library (instead of OpenSSL) * Checkout Windows-style, commit Unix-style line endings (core.autocrlf=true) * Use Windows' default console window (instead of Mintty)

2. Close any previously open console windows and re-open them to ensure they receive the new PATH
3. Ensure `git --version` works

Get Radare2 Code Follow these steps to clone the Radare2 git repository.

1. In your Radare2 Conda environment, navigate to a location where the code will be saved and compiled. This location needs approximately **3-4GiB** of space
2. Clone the repository with `git clone https://github.com/radareorg/radare2.git`

Compile Radare2 Code Follow these steps to compile the Radare2 Code.

Compiled binaries will be installed into the `dest` folder.

1. Enter the Radare2 Conda environment
2. Navigate to the root of the Radare2 sources (`cd radare2`)
3. Initialize Visual Studio tooling by executing the command below that matches the version of Visual Studio installed on your machine and the version of Radare2 you wish to install:

- **Visual Studio 2015:**

  Note: For the 64-bit version change only the `x86` at the very end of the command below to `x64`.

  `"%ProgramFiles(x86)%\Microsoft Visual Studio 14.0\VC\vcvarsall.bat" x86`

- **Visual Studio 2017:**

  Note 1: Change `Community` to either `Professional` or `Enterprise` in the command below depending on the version installed.

  Note 2: Change `vcvars32.bat` to `vcvars64.bat` in the command below for the 64-bit version.

  `"%ProgramFiles(x86)%\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvars32.bat"`

- **Visual Studio Preview:**

  Note 1: Change `Community` to either `Professional` or `Enterprise` in the command below depending on the version installed.

  Note 2: Change `vcvars32.bat` to `vcvars64.bat` in the command below for the 64-bit version.

  `"%ProgramFiles(x86)%\Microsoft Visual Studio\Preview\Community\VC\Auxiliary\Build\vcvars32.bat"`

4. Generate the build system with Meson:

Note 1: Change `debug` to `release` in the command below depending on whether the latest version or release version is desired.

Note 2: If you are using visual studio 2017, you can change swap `vs2015` for `vs2017`.

`meson build --buildtype debug --backend vs2015 --prefix %cd%\dest`

Meson currently requires `--prefix` to point to an absolute path. We use the %CD% pseudo-variable to get the absolute path to the current working directory.

5. Start a build:

   Note: Change `Debug` to `Release` in the command below depending on the version desired.

   `msbuild build\radare2.sln /p:Configuration=Debug /m`

   The `/m[axcpucount]` switch creates one MSBuild worker process per logical processor on your machine. You can specify a numeric value (e.g. `/m:2`) to limit the number of worker processes if needed. (This should not be confused with the Visual C++ Compiler switch `/MP`.)

   If you get an error with the 32-bit install that says something along the lines of `error MSB4126: The specified solution configuration "Debug|x86" is invalid.` Get around this by adding the following argument to the command: `/p:Platform=Win32`

6. Install into your destination folder: `meson install -C build --no-rebuild`

7. Check your Radare2 version: `dest\bin\radare2.exe -v`

Check That Radare2 Runs From All Locations

1. In the file explorer go to the folder Radare2 was just installed in.
2. From this folder go to `dest` > `bin` and keep this window open.
3. Go to System Properties: In the Windows search bar enter `sysdm.cpl`.
4. Go to `Advanced > Environment Variables`.
5. Click on the PATH variable and then click edit (if it exists within both the user and system variables, look at the user version).
6. Ensure the file path displayed in the window left open is listed within the PATH variable. If it is not add it and click `ok`.
7. Log out of your Windows session.
8. Open up a new Windows Command Prompt: type `cmd` in the search bar. Ensure that the current path is not in the Radare2 folder.
9. Check Radare2 version from Command Prompt Window: `radare2 -v`

Android

Radare2 can be cross-compiled for other architectures/systems as well, like Android.

Prerequisites

- Python 3
- Meson
- Ninja
- Git
- Android NDK

Step-by-step

Download and extract the Android NDK Download the Android NDK from the official site and extract it somewhere on your system (e.g. `/tmp/android-ndk`)

Make

Specify NDK base path

```
$ echo NDK=/tmp/android-ndk  > ~/.r2androidrc
```

Compile + create tar.gz + push it to connected android device

```
./sys/android-build.sh arm64-static
```

You can build for different architectures by changing the argument to `./sys/android-build.sh`. Run the script without any argument to see the accepted values.

Meson

Create a cross-file for meson Meson needs a configuration file that describes the cross compilation environment (e.g. `meson-android.ini`). You can adjust it as necessary, but something like the following should be a good starting point:

```
[binaries]
c       = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android28-clang'
cpp     = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android28-clang++'
ar      = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android-ar'
as      = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android-as'
ranlib  = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android-ranlib'
ld      = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android-ld'
strip   = '/tmp/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-android-strip'
pkgconfig = 'false'

[properties]
sys_root = '/tmp/android-ndk/sysroot'
```

```
[host_machine]
system = 'android'
cpu_family = 'arm'
cpu = 'aarch64'
endian = 'little'
```

Compile with meson + ninja Now setup the build directory with meson as usual:

```
$ CFLAGS="-static" LDFLAGS="-static" meson --default-library static --prefix=/tmp/android-dir -Dblob=true build --cross-file ./meson-android.ini
```

A bit of explanation about all the options: * `CFLAGS="-static"`, `LDFLAGS="-static"`, `--default-library static`: this ensure that libraries and binaries are statically compiled, so you do not need to properly set LD_* environment variables in your Android environment to make it find the right libraries. Binaries have everything they need inside. * `-Dblob=true`: it tells meson to compile just one binary with all the needed code for running `radare2`, `rabin2`, `rasm2`, etc. and creates symbolic links to those names. This avoids creating many statically compiled large binaries and just create one that provides all features. You will still have `rabin2`, `rasm2`, `rax2`, etc. but they are just symlinks to `radare2`. * `--cross-file ./meson-android.ini`: it describes how to compile radare2 for Android

Then compile and install the project:

```
$ ninja -C build
$ ninja -C build install
```

Move files to your android device and enjoy At this point you can copy the generated files in /tmp/android-dir to your Android device and running radare2 from it. For example:

```
$ cd /tmp && tar -cvf radare2-android.tar.gz android-dir
$ adb push radare2-android.tar.gz /data/local/tmp
$ adb shell
DEVICE:/ $ cd /data/local/tmp
DEVICE:/data/local/tmp $ tar xvf radare2-android.tar.gz
DEVICE:/data/local/tmp $ ./android-dir/bin/radare2
Usage: r2 [-ACdfLMnNqStuvwzX] [-P patch] [-p prj] [-a arch] [-b bits] [-i file]
          [-s addr] [-B baddr] [-m maddr] [-c cmd] [-e k=v] file|pid|-|--|=
```

User Interfaces

Radare2 has seen many different user interfaces being developed over the years.

Maintaining a GUI is far from the scope of developing the core machinery of a reverse engineering toolkit: it is preferred to have a separate project and community, allowing both projects to collaborate and to improve together - rather than forcing cli developers to think in gui problems and having to jump back and forth between the graphic aspect and the low level logic of the implementations.

In the past, there have been at least 5 different native user interfaces (ragui, r2gui, gradare, r2net, bokken) but none of them got enough maintenance power to take off and they all died.

In addition, r2 has an embedded webserver and ships some basic user interfaces written in html/js. You can start them like this:

```
$ r2 -c=H /bin/ls
```

After 3 years of private development, Hugo Teso; the author of Bokken (python-gtk gui of r2) released to the public another frontend of r2, this time written in c++ and qt, which has been very welcomed by the community.

This GUI was named Iaito, but as long as he prefered not to keep maintaining it, Xarkes decided to fork it under the name of Cutter (name voted by the community), and lead the project. This is how it looks:

- https://github.com/radareorg/cutter.

Basic Radare2 Usage

The learning curve 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 the 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.

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 radare2 opens an executable file, by default it will open the file in Virtual Addressing (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 radare2 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 radare2 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?`. Appending `?*` will show commands starting with the given string, e.g. `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. By pressing `p` or `P` you can scroll different visual mode representations. There is a second most important visual mode - curses-like panels interface, accessible with `V!` command.

Command-line Options

The radare core accepts many flags from the command line.

This is an excerpt from the usage help message:

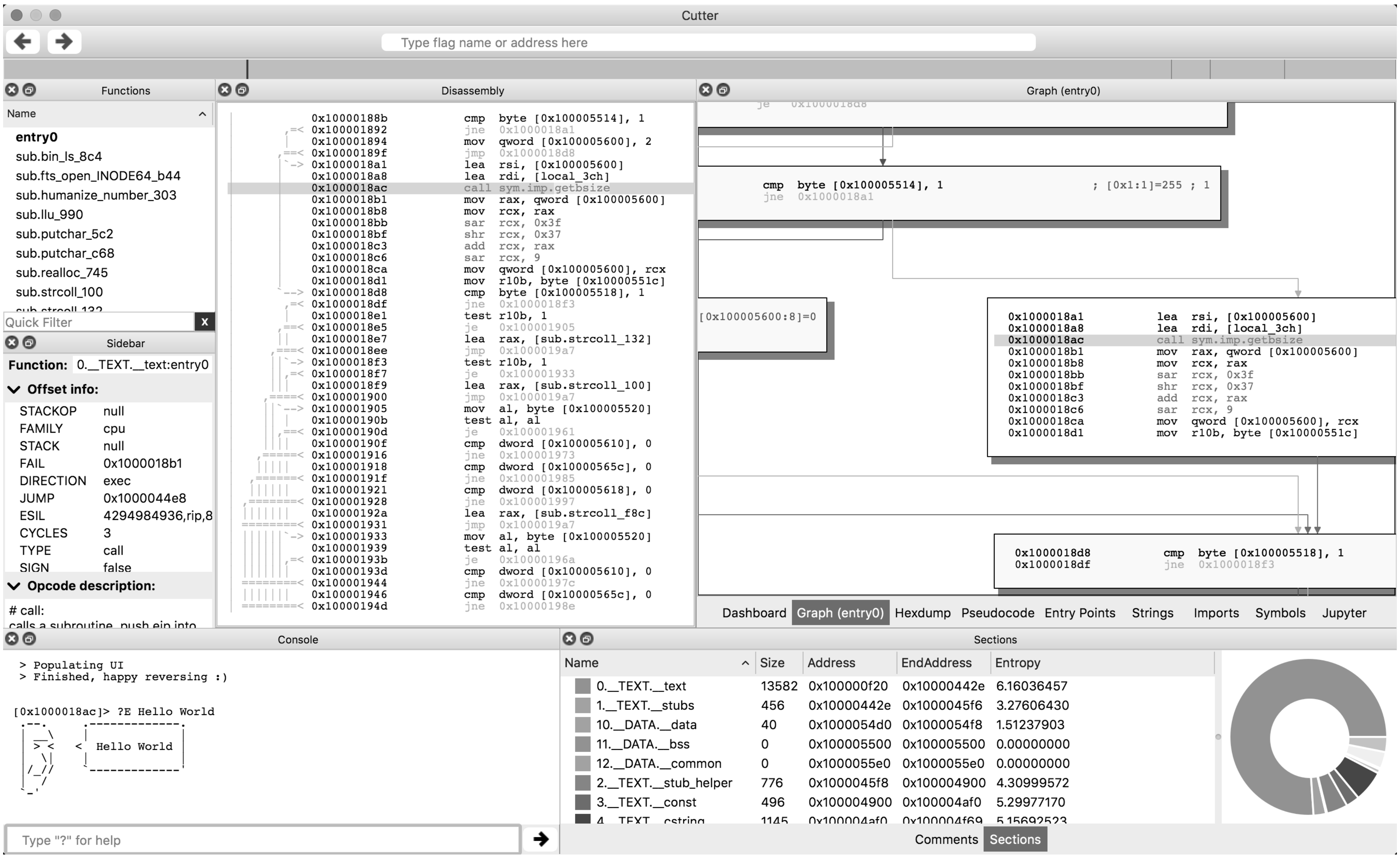

Figure 1: Cutter

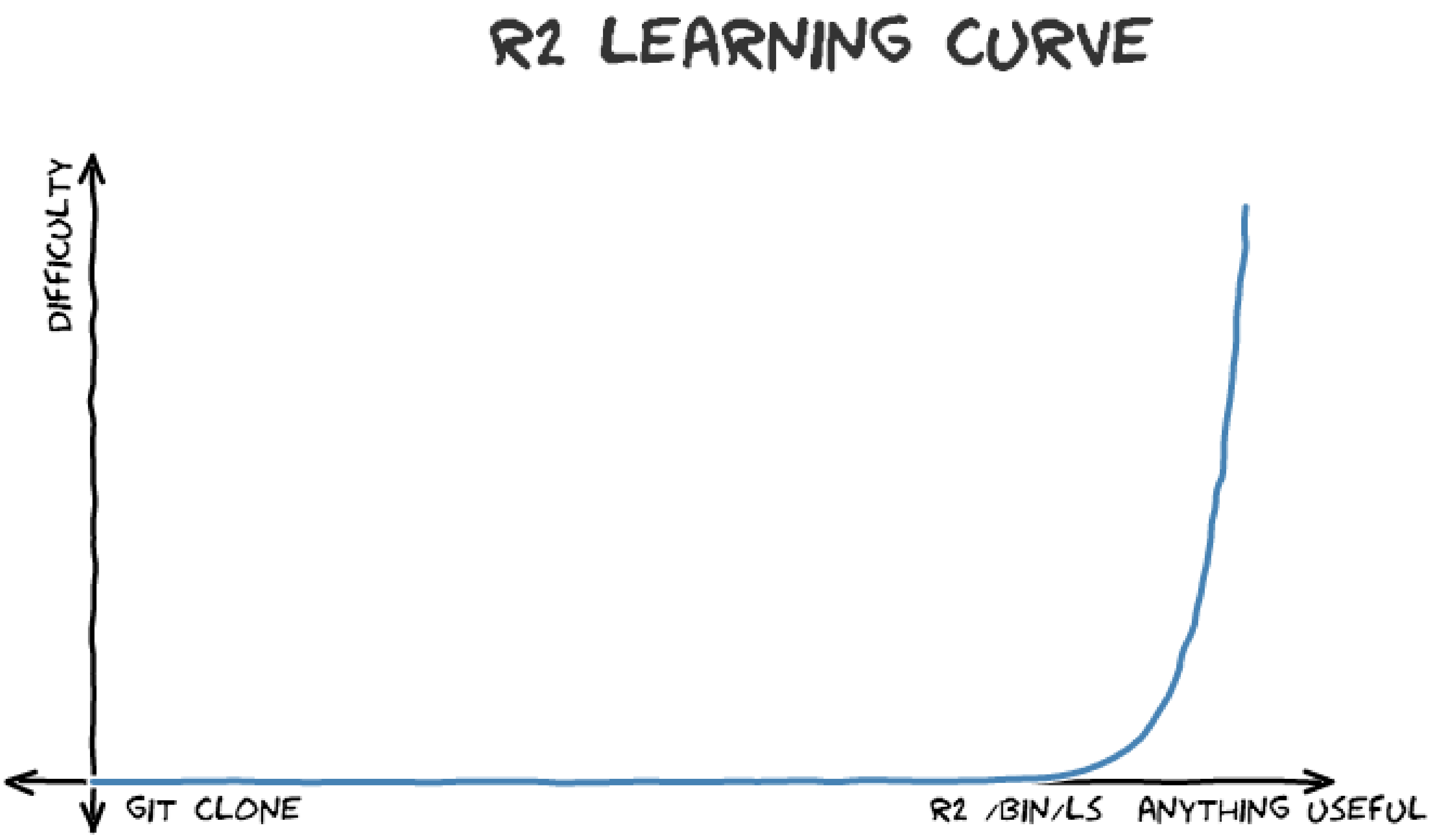

Figure 2: learning_curve

```
$ radare2 -h
Usage: r2 [-ACdfLMnNqStuvwzX] [-P patch] [-p prj] [-a arch] [-b bits] [-i file]
          [-s addr] [-B baddr] [-m maddr] [-c cmd] [-e k=v] file|pid|-|--|=
 --           run radare2 without opening any file
 -            same as 'r2 malloc://512'
 =            read file from stdin (use -i and -c to run cmds)
 -=           perform !=! command to run all commands remotely
 -0           print \x00 after init and every command
 -2           close stderr file descriptor (silent warning messages)
 -a [arch]    set asm.arch
 -A           run 'aaa' 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           debug the executable 'file' or running process 'pid'
 -D [backend] enable debug mode (e cfg.debug=true)
 -e k=v       evaluate config var
 -f           block size = file size
 -F [binplug] force to use that rbin plugin
 -h, -hh      show help message, -hh for long
 -H ([var])   display variable
 -i [file]    run script file
 -I [file]    run script file before the file is opened
 -k [OS/kern] set asm.os (linux, macos, w32, netbsd, ...)
 -l [lib]     load plugin file
 -L           list supported IO plugins
 -m [addr]    map file at given address (loadaddr)
 -M           do not demangle symbol names
 -n, -nn      do not load RBin info (-nn only load bin structures)
 -N           do not load user settings and scripts
 -q           quiet mode (no prompt) and quit after -i
 -Q           quiet mode (no prompt) and quit faster (quickLeak=true)
 -p [prj]     use project, list if no arg, load if no file
 -P [file]    apply rapatch file and quit
 -r [rarun2]  specify rarun2 profile to load (same as -e dbg.profile=X)
 -R [rr2rule] specify custom rarun2 directive
 -s [addr]    initial seek
 -S           start r2 in sandbox mode
 -t           load rabin2 info in thread
 -u           set bin.filter=false to get raw sym/sec/cls names
 -v, -V       show radare2 version (-V show lib versions)
 -w           open file in write mode
 -x           open without exec-flag (asm.emu will not work), See io.exec
 -X           same as -e bin.usextr=false (useful for dyldcache)
 -z, -zz      do not load strings or load them even in raw
```

Common usage patterns

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
```

Set the configuration variable:

```
$ r2 -e scr.color=0 blah.bin
```

Debug a program:

```
$ r2 -d ls
```

Use an existing project file:

```
$ r2 -p test
```

Command Format

A general format for radare2 commands is as follows:

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

People who use Vim daily and are familiar with its commands will find themselves at home. You will see this format used throughout the book. 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 you want to use the cmd callback from the I/O plugin you must prefix with `=!`.

Note that a single exclamation mark will run the command and print the output through the RCons API. This means that the execution will be blocking and not interactive. Use double exclamation marks – `!!` – to run a standard system call.

All the socket, filesystem and execution APIs can be restricted with the `cfg.sandbox` configuration variable.

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 standard UNIX pipe `|` is also available in the radare2 shell. You can use it to filter the output of an r2 command with any shell program that reads from stdin, such as `grep`, `less`, `wc`. If you do not want to spawn anything, or you can't, or the target system does not have the basic UNIX tools you need (Windows or embedded users), you can also use the built-in grep (`~`).

See `~?` for help.

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

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

Additionally, you can grep either for columns or for 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 radare2, 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 loops section (iterators) for more information.

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.

Most of the commands offer autocompletion support using `<TAB>` key, for example `s`eek or `f`lags commands. It offers autocompletion using all possible values, taking flag names in this case. Note that it is possible to see the history of the commands using the `!~...` command - it offers a visual mode to scroll through the radare2 command history.

To extend the autocompletion support to handle more commands or enable autocompletion to your own commands defined in core, I/O plugins you must use the `!!!` command.

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 `?`:

```
[0xb7f9d810]> ?vi 0x8048000
134512640
[0xv7f9d810]> ?vi 0x8048000+34
134512674
[0xb7f9d810]> ?vi 0x8048000+0x34
134512692
[0xb7f9d810]> ? 1+2+3-4*3
hex     0xfffffffffffffffa
octal   01777777777777777777772
unit    17179869184.0G
segment fffff000:0ffa
int64   -6
string  "\xfa\xff\xff\xff\xff\xff\xff\xff"
binary  0b1111111111111111111111111111111111111111111111111111111111111010
fvalue: -6.0
float:  nanf
double: nan
trits   0t11112220022122120101211020120210210211201
```

Supported arithmetic operations are:

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

```
[0x00000000]> ?vi 1+2+3
6
```

To use of logical OR should quote the whole command to avoid executing the `|` pipe:

```
[0x00000000]> "? 1 | 2"
hex     0x3
octal   03
unit    3
segment 0000:0003
int32   3
string  "\x03"
binary  0b00000011
fvalue: 2.0
float:  0.000000f
double: 0.000000
trits   0t10
```

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.

Use the `?$?` command to list all the available commands or read the refcard chapter of this book.

```
$$    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)
$b    block size
```

Some more examples:

```
[0x4A13B8C0]> ? $m + $l
140293837812900 0x7f98b45df4a4 03771426427372244 130658.0G 8b45d000:04a4 140293837812900 10100100 140293837812900.0 -0.000000
```

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

Basic Debugger Session

To debug a program, start radare with the `-d` option. Note that 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
$ r2 -a arm -b 16 -d gdb://192.168.1.43:9090
...
```

In the second case, the debugger will fork and load the debugee `ls` program in memory.

It will pause its execution early in `ld.so` dynamic linker. As a result, you will not yet see the entrypoint or any shared libraries at this point.

You can override this behavior by setting another name for an 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 `~/.config/radare2/radare2rc`.

Another way to continue until a specific address is by using the `dcu` command. Which means: "debug continue until" taking the address of the place to stop at. For example:

```
dcu main
```

Be warned that certain malware or other tricky programs can actually execute code before `main()` and thus you'll be unable to control them. (Like the program constructor or the tls initializers)

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 permissions of page at A and size S
> dr eax=33     ; set register value. eax = 33
```

There is another option for debugging in radare, which may be easier: using visual mode.

That way you will neither need to remember many commands nor to keep program state in your mind.

To enter visual debugger mode use `Vpp`:

```
[0xb7f0c8c0]> Vpp
```

The initial view after entering visual mode is a hexdump view of the 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:

```
<Press ':'>
x @ esi
```

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

A frequently used command is `dr`, which is used to read or write values of the target's general purpose registers. For a more compact register value representation you might use `dr=` command. You can also manipulate the hardware and the 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.

See https://github.com/radareorg/radare2/blob/master/DEVELOPERS.md for general help on contributing to radare2.

Configuration

The core reads `~/.config/radare2/radare2rc` while starting. You can add `e` commands to this file to tune the radare2 configuration to your taste.

To prevent radare2 from parsing this file at startup, pass it the `-N` option.

All the configuration of radare2 is done with the `eval` commands. A typical startup configuration file looks like this:

```
$ cat ~/.radare2rc
e scr.color = 1
e dbg.bep   = loader
```

The configuration can also be changed with `-e` <config=value> 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=1 -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 the output to a selected namespace, pass it with an ending dot to `e`. For example, `e file.` will display all variables defined inside the "file" namespace.

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

```
Usage: e [var[=value]]  Evaluable vars
| e?asm.bytes     show description
| e??             list config vars with description
| e a             get value of var 'a'
| e a=b           set var 'a' the 'b' value
| e var=?         print all valid values of var
| e var=??        print all valid values of var with description
| e.a=b           same as 'e a=b' but without using a space
| e,k=v,k=v,k=v   comma separated k[=v]
| e-              reset config vars
| e*              dump config vars in r commands
| e!a             invert the boolean value of 'a' var
| ec [k] [color]  set color for given key (prompt, offset, ...)
| eevar           open editor to change the value of var
| ed              open editor to change the ~/.radare2rc
| ej              list config vars in JSON
| env [k[=v]]     get/set environment variable
| er [key]        set config key as readonly. no way back
| es [space]      list all eval spaces [or keys]
| et [key]        show type of given config variable
| ev [key]        list config vars in verbose format
| evj [key]       list config vars in verbose format in JSON
```

A simpler alternative to the `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:

```
[EvalSpace]

    >  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 the output of any command as ANSI, W32 Console or HTML formats. 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=1' >> ~/.radare2rc
```

Note that enabling colors is not a boolean option. Instead, it is a number because there are different color depth levels. This is:

- 0: black and white
- 1: 16 basic ANSI colors
- 2: 256 scale colors
- 3: 24bit true color

The reason for having such user-defined options is because there's no standard or portable way for the terminal programs to query the console to determine the best configuration, same goes for charset encodings, so r2 allows you to choose that by hand.

Usually, serial consoles may work with 0 or 1, while xterms may support up to 3. RCons will try to find the closest color scheme for your theme when you choose a different them with the `eco` command.

It is possible to configure the 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:

Themes

You can create your own color theme, but radare2 have its own predefined ones. Use the `eco` command to list or select them.

After selecting one, you can compare between the color scheme of the shell and the current theme by pressing Ctrl-Shift and then right arrow key for the toggle.

In visual mode use the `R` key to randomize colors or choose the next theme in the list.

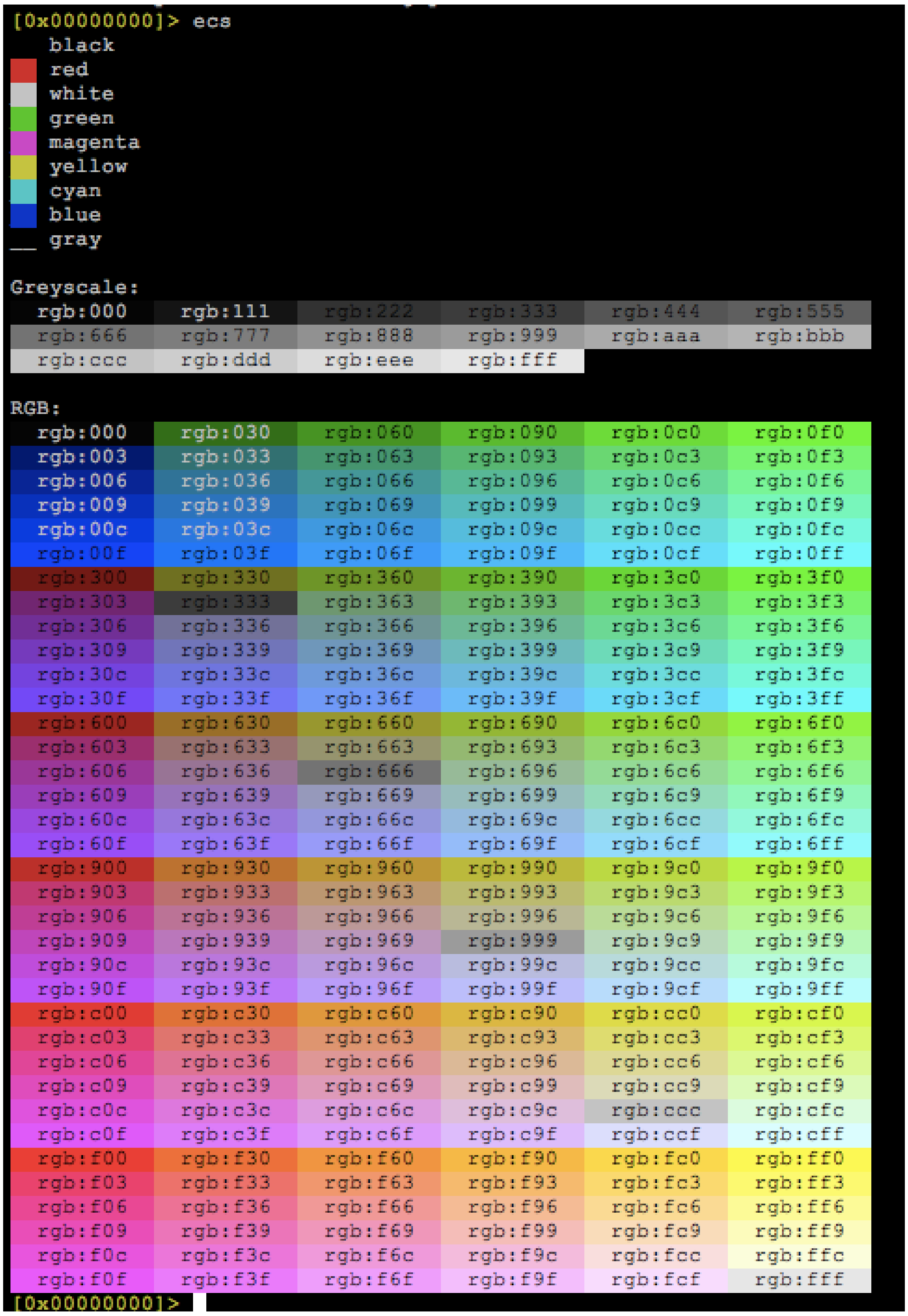
[0x00000000]> ecs
black
red
white
green
magenta
yellow
cyan
blue
gray
Greyscale:
rgb:000 rgb:111 rgb:222 rgb:333 rgb:444 rgb:555
rgb:666 rgb:777 rgb:888 rgb:999 rgb:aaa rgb:bbb
rgb:ccc rgb:ddd rgb:eee rgb:fff
RGB:
rgb:000 rgb:030 rgb:060 rgb:090 rgb:0c0 rgb:0f0
rgb:003 rgb:033 rgb:063 rgb:093 rgb:0c3 rgb:0f3
rgb:006 rgb:036 rgb:066 rgb:096 rgb:0c6 rgb:0f6
rgb:009 rgb:039 rgb:069 rgb:099 rgb:0c9 rgb:0f9
rgb:00c rgb:03c rgb:06c rgb:09c rgb:0cc rgb:0fc
rgb:00f rgb:03f rgb:06f rgb:09f rgb:0cf rgb:0ff
rgb:300 rgb:330 rgb:360 rgb:390 rgb:3c0 rgb:3f0
rgb:303 rgb:333 rgb:363 rgb:393 rgb:3c3 rgb:3f3
rgb:306 rgb:336 rgb:366 rgb:396 rgb:3c6 rgb:3f6
rgb:309 rgb:339 rgb:369 rgb:399 rgb:3c9 rgb:3f9
rgb:30c rgb:33c rgb:36c rgb:39c rgb:3cc rgb:3fc
rgb:30f rgb:33f rgb:36f rgb:39f rgb:3cf rgb:3ff
rgb:600 rgb:630 rgb:660 rgb:690 rgb:6c0 rgb:6f0
rgb:603 rgb:633 rgb:663 rgb:693 rgb:6c3 rgb:6f3
rgb:606 rgb:636 rgb:666 rgb:696 rgb:6c6 rgb:6f6
rgb:609 rgb:639 rgb:669 rgb:699 rgb:6c9 rgb:6f9
rgb:60c rgb:63c rgb:66c rgb:69c rgb:6cc rgb:6fc
rgb:60f rgb:63f rgb:66f rgb:69f rgb:6cf rgb:6ff
rgb:900 rgb:930 rgb:960 rgb:990 rgb:9c0 rgb:9f0
rgb:903 rgb:933 rgb:963 rgb:993 rgb:9c3 rgb:9f3
rgb:906 rgb:936 rgb:966 rgb:996 rgb:9c6 rgb:9f6
rgb:909 rgb:939 rgb:969 rgb:999 rgb:9c9 rgb:9f9
rgb:90c rgb:93c rgb:96c rgb:99c rgb:9cc rgb:9fc
rgb:90f rgb:93f rgb:96f rgb:99f rgb:9cf rgb:9ff
rgb:c00 rgb:c30 rgb:c60 rgb:c90 rgb:cc0 rgb:cf0
rgb:c03 rgb:c33 rgb:c63 rgb:c93 rgb:cc3 rgb:cf3
rgb:c06 rgb:c36 rgb:c66 rgb:c96 rgb:cc6 rgb:cf6
rgb:c09 rgb:c39 rgb:c69 rgb:c99 rgb:cc9 rgb:cf9
rgb:c0c rgb:c3c rgb:c6c rgb:c9c rgb:ccc rgb:cfc
rgb:c0f rgb:c3f rgb:c6f rgb:c9f rgb:ccf rgb:cff
rgb:f00 rgb:f30 rgb:f60 rgb:f90 rgb:fc0 rgb:ff0
rgb:f03 rgb:f33 rgb:f63 rgb:f93 rgb:fc3 rgb:ff3
rgb:f06 rgb:f36 rgb:f66 rgb:f96 rgb:fc6 rgb:ff6
rgb:f09 rgb:f39 rgb:f69 rgb:f99 rgb:fc9 rgb:ff9
rgb:f0c rgb:f3c rgb:f6c rgb:f9c rgb:fcc rgb:ffc
rgb:f0f rgb:f3f rgb:f6f rgb:f9f rgb:fcf rgb:fff
[0x00000000]>

Figure 3: img

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.`

The `e??` command to get help on all the evaluable configuration variables of radare2. As long as the output of this command is pretty large you can combine it with the internal grep `~` to filter for what you are looking for:

e??~color

The Visual mode has an eval browser that is accessible through the `Vbe` command.

asm.arch

Defines the target CPU architecture used for disassembling (`pd`, `pD` commands) and code analysis (`a` command). You can find the list of possible values 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 handmade ones.

asm.bits

Determines width in bits of registers for the 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 set the psuedo syntax in the disassembly. "False" indicates a native one, defined by the current architecture, "true" activates a pseudocode strings format. For example, it'll transform :

```
0x080483ff      e832000000      call 0x8048436
0x08048404      31c0            xor eax, eax
0x08048406      0205849a0408    add al, byte [0x8049a84]
0x0804840c      83f800          cmp eax, 0
0x0804840f      7405            je 0x8048416
```

to

```
0x080483ff      e832000000      0x8048436 ()
0x08048404      31c0            eax = 0
0x08048406      0205849a0408    al += byte [0x8049a84]
0x0804840c      83f800          var = eax - 0
0x0804840f      7405            if (!var) goto 0x8048416
```

It can be useful while disassembling obscure architectures.

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.lines.call

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

asm.lines.out

When defined as "true", the disassembly view will also draw control flow lines that go outside 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 the 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 with programs trace analysis.

asm.bytes

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

asm.sub.reg

A boolean value used to replace register names with arguments or their associated role alias.

For example, if you have something like this:

```
0x080483ea      83c404          add esp, 4
0x080483ed      68989a0408      push 0x8049a98
0x080483f7      e870060000      call sym.imp.scanf
0x080483fc      83c408          add esp, 8
0x08048404      31c0            xor eax, eax
```

This variable changes it to:

```
0x080483ea      83c404          add SP, 4
0x080483ed      68989a0408      push 0x8049a98
0x080483f7      e870060000      call sym.imp.scanf
0x080483fc      83c408          add SP, 8
0x08048404      31c0            xor A0, A0
```

asm.sub.jmp

A boolean value used to substitute jump, call and branch targets in disassembly.

For example, when turned on, it'd display `jal 0x80001a40` as `jal fcn.80001a40` in the disassembly.

asm.sub.rel

A boolean value which substitutes pc relative expressions in disassembly. When turned on, it shows the references as string references.

For example:

```
0x5563844a0181      488d3d7c0e00.  lea rdi, [rip + 0xe7c]    ; str.argv__2d_:__s
```

When turned on, this variable lets you display the above instruction as:

```
0x5563844a0181      488d3d7c0e00.  lea rdi, str.argv__2d_:__s    ; 0x5563844a1004 ; "argv[%2d]: %s\n"
```

asm.sub.section

Boolean which shows offsets in disassembly prefixed with the name of the section or map.

That means, from something like:

```
0x000067ea      488d0def0c01.  lea rcx, [0x000174e0]
```

to the one below, when toggled on.

```
0x000067ea      488d0def0c01.  lea rcx, [fmap.LOAD1.0x000174e0]
```

asm.sub.varonly

Boolean which substitutes the variable expression with the local variable name.

For example: `var_14h` as `rbp - var_14h`, in the disassembly.

cfg.bigendian

Change endianness. "true" means big-endian, "false" is for little-endian. "file.id" and "file.flag" both to be true.

cfg.newtab

If this variable is enabled, help messages will be displayed along with command names in tab completion for commands.

scr.color

This variable specifies the mode for colorized screen output: "false" (or 0) means no colors, "true" (or 1) means 16-colors mode, 2 means 256-colors mode, 3 means 16 million-colors mode. If your favorite theme looks weird, try to bump this up.

scr.seek

This variable accepts a full-featured expression or a pointer/flag (eg. eip). If set, radare will set seek position to its value on startup.

scr.scrollbar

If you have set up any flagzones (`fz?`), this variable will let you display the scrollbar with the flagzones, in Visual mode. Set it to `1` to display the scrollbar at the right end, `2` for the top and `3` to display it at the bottom.

scr.utf8

A boolen variable to show UTF-8 characters instead of ANSI.

cfg.fortunes

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

cfg.fortunes.type

Fortunes are classified by type. This variable determines which types are allowed for displaying when `cfg.fortunes` is `true`, so they can be fine-tuned on what's appropriate for the intended audience. Current types are `tips`, `fun`, `nsfw`, `creepy`.

stack.size

This variable lets you set the size of stack in bytes.

Files

Use `r2 -H` to list all the environment variables that matter to know where it will be looking for files. Those paths depend on the way (and operating system) you have built r2 for.

```
R2_PREFIX=/usr
MAGICPATH=/usr/share/radare2/2.8.0-git/magic
PREFIX=/usr
INCDIR=/usr/include/libr
LIBDIR=/usr/lib64
LIBEXT=so
RCONFIGHOME=/home/user/.config/radare2
RDATAHOME=/home/user/.local/share/radare2
RCACHEHOME=/home/user/.cache/radare2
LIBR_PLUGINS=/usr/lib/radare2/2.8.0-git
USER_PLUGINS=/home/user/.local/share/radare2/plugins
USER_ZIGNS=/home/user/.local/share/radare2/zigns
```

RC Files

RC files are r2 scripts that are loaded at startup time. Those files must be in 3 different places:

System

radare2 will first try to load /usr/share/radare2/radare2rc

Your Home

Each user in the system can have its own r2 scripts to run on startup to select the color scheme, and other custom options by having r2 commands in there.

- ~/.radare2rc
- ~/.config/radare2/radare2rc
- ~/.config/radare2/radare2rc.d/

Target file

If you want to run a script everytime you open a file, just create a file with the same name of the file but appending `.r2` to it. # 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 the 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 the currently loaded IO plugin (a debugger, for example). Most plugins provide help messages with `=!?` or `=!help`.

```
$ r2 -d /bin/ls
> =!help      ; handled by the IO plugin
```

If a command starts with `!`, posix_system() is called to pass the command to your shell. Check `!?` for more options and usage examples.

```
> !ls         ; run `ls` in the shell
```

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

```
> px 0x17     ; show 0x17 bytes in hexs 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 a 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 the 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: an 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 a semicolon `;`:

```
> px ; dr
```

Using `_`, you can print the result that was obtained by the last command.

```
[0x00001060]> axt 0x00002004
main 0x1181 [DATA] lea rdi, str.argv__2d_:__s
[0x00001060]> _
main 0x1181 [DATA] lea rdi, str.argv__2d_:__s
```

Seeking

To move around the file we are inspecting we will need to change the offset at which we are using the `s` command.

The argument is a math expression that can contain flag names, parenthesis, addition, substraction, multiplication of immediates of contents of memory using brackets.

Some example commands:

```
[0x00000000]> s 0x10
[0x00000010]> s+4
[0x00000014]> s-
[0x00000010]> s+
[0x00000014]>
```

Observe how the prompt offset changes. The first line moves the current offset to the address 0x10.

The second does a relative seek 4 bytes forward.

And finally, the last 2 commands are undoing, and redoing the last seek operations.

Instead of using just numbers, we can use complex expressions, or basic arithmetic operations to represent the address to seek.

To do this, check the ?$? Help message which describes the internal variables that can be used in the expressions. For example, this is the same as doing s+4 .

```
[0x00000000]> s $$+4
```

From the debugger (or when emulating) we can also use the register names as references. They are loaded as flags with the `.dr*` command, which happens under the hood.

```
[0x00000000]> s rsp+0x40
```

Here's the full help of the `s` command. We will explain in more detail below.

```
[0x00000000]> s?
Usage: s    # Help for the seek commands. See ?$? to see all variables
| s                 Print current address
| s.hexoff          Seek honoring a base from core->offset
| s:pad             Print current address with N padded zeros (defaults to 8)
| s addr            Seek to address
| s-                Undo seek
| s-*               Reset undo seek history
| s- n              Seek n bytes backward
| s--[n]            Seek blocksize bytes backward (/=n)
| s+                Redo seek
| s+ n              Seek n bytes forward
| s++[n]            Seek blocksize bytes forward (/=n)
| s[j*=!]           List undo seek history (JSON, =list, *r2, !=names, s==)
| s/ DATA           Search for next occurrence of 'DATA'
| s/x 9091          Search for next occurrence of \x90\x91
| sa [[+-]a] [asz]  Seek asz (or bsize) aligned to addr
| sb                Seek aligned to bb start
| sC[?] string      Seek to comment matching given string
| sf                Seek to next function (f->addr+f->size)
| sf function       Seek to address of specified function
| sf.               Seek to the beginning of current function
| sg/sG             Seek begin (sg) or end (sG) of section or file
| sl[?] [+-]line    Seek to line
| sn/sp ([nkey])    Seek to next/prev location, as specified by scr.nkey
| so [N]            Seek to N next opcode(s)
| sr pc             Seek to register
| ss                Seek silently (without adding an entry to the seek history)
```

```
> 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
```

There are also subcommands of `?` that display the output in one specific format (base 10, base 16 ,…). See `?v` and `?vi`.

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 a test file, let's use a simple `hello_world.c` compiled in Linux ELF format. After we compile it let's open it with radare2:

```
$ r2 hello_world
```

Now we have the command prompt:

```
[0x00400410]>
```

And it is time to go deeper.

Seeking at any position

All seeking commands that take an address as a command parameter can use any numeral base such as hex, octal, binary or decimal.

Seek to an address 0x0. An alternative command is simply `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*.

There's also a command to show 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 radare2 commands will process when not given an explicit size argument. You can temporarily change the block size by specifying a numeric argument to the print commands. For example `px 20`.

```
[0x00000000]> b?
Usage: b[f] [arg]  # Get/Set block size
| b 33     set block size to 33
| b eip+4  numeric argument can be an expression
```

```
| b        display current block size
| b+3      increase blocksize by 3
| b-16     decrease blocksize by 16
| b*       display current block size in r2 command
| bf foo   set block size to flag size
| bj       display block size information in JSON
| 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. To make that work, you have to either run function analysis `af` (which is included in `aa`) or manually seek and define some functions e.g. via `Vd`.

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

You can combine two operations in a single `pdf` command. Except that `pdf` neither uses nor affects global block size.

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

Another way around is to use special variables `$FB` and `$FS` which denote Function's Beginning and Size at the current seek. Read more about Usable variables.

```
[0x00000000]> s sym.main + 0x04
[0x00001ec9]> pD @ $FB !$FS  # disassemble current function
  211: int main (int argc, char **argv, char **envp);
           0x00001ec5      55                 push rbp
           0x00001ec6      4889e5             mov rbp, rsp
           0x00001ec9      4881ecc0000000     sub rsp, 0xc0
...
           0x00001f97      c3                 ret
```

Note: don't put space after `!` size designator. See also Command Format.

Sections

The concept of sections is tied to the information extracted from the binary. We can display this information by using the `i` command.

Displaying information about sections:

```
[0x00005310]> iS
[Sections]
00 0x00000000     0 0x00000000     0 ----
01 0x00000238    28 0x00000238    28 -r-- .interp
02 0x00000254    32 0x00000254    32 -r-- .note.ABI_tag
03 0x00000278   176 0x00000278   176 -r-- .gnu.hash
04 0x00000328  3000 0x00000328  3000 -r-- .dynsym
05 0x00000ee0  1412 0x00000ee0  1412 -r-- .dynstr
06 0x00001464   250 0x00001464   250 -r-- .gnu.version
07 0x00001560   112 0x00001560   112 -r-- .gnu.version_r
08 0x000015d0  4944 0x000015d0  4944 -r-- .rela.dyn
09 0x00002920  2448 0x00002920  2448 -r-- .rela.plt
10 0x000032b0    23 0x000032b0    23 -r-x .init
...
```

As you may know, binaries have sections and maps. The sections define the contents of a portion of the file that can be mapped in memory (or not). What is mapped is defined by the segments.

Before the IO refactoring done by condret, the `S` command was used to manage what we now call maps. Currently the `S` command is deprecated because `iS` and `om` should be enough.

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 `iS`. Use `iS?` to get the help message. To list all created sections use `iS` (or `iSj` to get the json format). The `iS=` will show the region bars in ascii-art.

You can create a new mapping using the `om` subcommand as follows:

```
om fd vaddr [size] [paddr] [rwx] [name]
```

For Example:

```
[0x0040100]> om 4 0x00000100 0x00400000 0x0001ae08 rwx test
```

You can also use `om` command to view information about mapped sections:

```
[0x00401000]> om
 6 fd: 4 +0x0001ae08 0x00000100 - 0x004000ff rwx test
 5 fd: 3 +0x00000000 0x00000000 - 0x0000055f r-- fmap.LOAD0
 4 fd: 3 +0x00001000 0x00001000 - 0x000011e4 r-x fmap.LOAD1
 3 fd: 3 +0x00002000 0x00002000 - 0x0000211f r-- fmap.LOAD2
 2 fd: 3 +0x00002de8 0x00003de8 - 0x0000402f r-- fmap.LOAD3
 1 fd: 4 +0x00000000 0x00004030 - 0x00004037 rw- mmap.LOAD3
```

Use `om?` to get all the possible subcommands. To list all the defined maps use `om` (or `omj` to get the json format or `om*` to get the r2 commands format). To get the ascii art view use `om=`.

It is also possible to delete the mapped section using the `om-mapid` command.

For Example:

```
[0x00401000]> om-6
```

Mapping Files

Radare's I/O subsystem allows you to map the contents of files into the same I/O space used to contain a loaded binary. New contents can be placed at random offsets.

The `o` command permits the user to open a file, this is mapped at offset 0 unless it has a known binary header and then the maps are created in virtual addresses.

Sometimes, we want to rebase a binary, or maybe we want to load or map the file in a different address.

When launching r2, the base address can be changed with the `-B` flag. But you must notice the difference when opening files with unknown headers, like bootloaders, so we need to map them using the `-m` flag (or specifying it as argument to the `o` command).

radare2 is able to open files and map portions of them at random places in memory specifying attributes like permissions and name. It is the perfect basic tooling to reproduce an environment like a core file, a debug session, by also loading and mapping all the libraries the binary depends on.

Opening files (and mapping them) is done using the `o` (open) command. Let's read the help:

```
[0x00000000]> o?
|Usage: o [com- ] [file] ([offset])
| o                         list opened files
| o-1                       close file descriptor 1
| o-!*                      close all opened files
| o--                       close all files, analysis, binfiles, flags, same as !r2 --
| o [file]                  open [file] file in read-only
| o+ [file]                 open file in read-write mode
| o [file] 0x4000 rwx       map file at 0x4000
| oa[-] [A] [B] [filename]  Specify arch and bits for given file
| oq                        list all open files
| o*                        list opened files in r2 commands
| o. [len]                  open a malloc://[len] copying the bytes from current offset
| o=                        list opened files (ascii-art bars)
| ob[?] [lbdos] [...]       list opened binary files backed by fd
| oc [file]                 open core file, like relaunching r2
| of [file]                 open file and map it at addr 0 as read-only
| oi[-|idx]                 alias for o, but using index instead of fd
| oj[?]                     list opened files in JSON format
| oL                        list all IO plugins registered
| om[?]                     create, list, remove IO maps
| on [file] 0x4000          map raw file at 0x4000 (no r_bin involved)
| oo[?]                     reopen current file (kill+fork in debugger)
| oo+                       reopen current file in read-write
| ood[r] [args]             reopen in debugger mode (with args)
| oo[bnm] [...]             see oo? for help
| op [fd]                   prioritize given fd (see also ob)
| ox fd fdx                 exchange the descs of fd and fdx and keep the mapping
```

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 the required file descriptor to it as an argument:

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

You can also view the ascii table showing the list of the opened files:

```
[0x00000000]> ob=
```

Print Modes

One of the key features of radare2 is displaying information in many formats. The goal is to offer a selection of display choices to interpret binary data in the best possible way.

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

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

```
[0x00005310]> p?
|Usage: p[=68abcdDfiImrstuxz] [arg|len] [@addr]
| p[b|B|xb] [len] ([S])   bindump N bits skipping S bytes
| p[iI][df] [len]         print N ops/bytes (f=func) (see pi? and pdi)
| p[kK] [len]             print key in randomart (K is for mosaic)
| p-[?][jh] [mode]        bar|json|histogram blocks (mode: e?search.in)
| p2 [len]                8x8 2bpp-tiles
| p3 [file]               print stereogram (3D)
| p6[de] [len]            base64 decode/encode
| p8[?][j] [len]          8bit hexpair list of bytes
| p=[?][bep] [N] [L] [b]  show entropy/printable chars/chars bars
| pa[edD] [arg]           pa:assemble  pa[dD]:disasm or pae: esil from hex
| pA[n_ops]               show n_ops address and type
| pb[?] [n]               bitstream of N bits
| pB[?] [n]               bitstream of N bytes
| pc[?][p] [len]          output C (or python) format
| pC[aAcdDxw] [rows]      print disassembly in columns (see hex.cols and pdi)
| pd[?] [sz] [a] [b]      disassemble N opcodes (pd) or N bytes (pD)
| pf[?][.nam] [fmt]       print formatted data (pf.name, pf.name $<expr>)
| pF[?][apx]              print asn1, pkcs7 or x509
| pg[?][x y w h] [cmd]    create new visual gadget or print it (see pg? for details)
| ph[?][=|hash] ([len])   calculate hash for a block
```

```
| pj[?] [len]             print as indented JSON
| pm[?] [magic]           print libmagic data (see pm? and /m?)
| po[?] hex               print operation applied to block (see po?)
| pp[?][sz] [len]         print patterns, see pp? for more help
| pq[?][is] [len]         print QR code with the first Nbytes
| pr[?][glx] [len]        print N raw bytes (in lines or hexblocks, 'g'unzip)
| 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]        show variable/pointer/value in memory
| pwd                     display current working directory
| px[?][owq] [len]        hexdump of N bytes (o=octal, w=32bit, q=64bit)
| pz[?] [len]             print zoom view (see pz? for help)
[0x00005310]>
```

Tip: when using json output, you can append the `~{}` to the command to get a pretty-printed version of the output:

```
[0x00000000]> oj
[{"raised":false,"fd":563280,"uri":"malloc://512","from":0,"writable":true,"size":512,"overlaps":false}]
[0x00000000]> oj~{}
[
    {
        "raised": false,
        "fd": 563280,
        "uri": "malloc://512",
        "from": 0,
        "writable": true,
        "size": 512,
        "overlaps": false
    }
]
```

For more on the magical powers of `~` see the help in `?@?`, and the Command Format chapter earlier in the book.

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...
```

Figure 4: hexprint

```
[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...
```

Figure 5: wordprint

Show Hexadecimal Words Dump (32 bits)

8 bits Hexpair List of Bytes

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

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

Figure 6: pxq

Show Hexadecimal Quad-words Dump (64 bits)

Date/Time Formats

Currently supported timestamp output modes are:

```
[0x00404888]> pt?
|Usage: pt [dn]  print timestamps
| pt.  print current time
| pt   print UNIX time (32 bit `cfg.bigendian`) Since January 1, 1970
```

```
| ptd  print DOS time (32 bit `cfg.bigendian`) Since January 1, 1980
| pth  print HFS time (32 bit `cfg.bigendian`) Since January 1, 1904
| ptn  print NTFS time (64 bit `cfg.bigendian`) Since January 1, 1601
```

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

```
[0x08048000]> e cfg.bigendian = false
[0x08048000]> pt 4
29:04:32948 23:12:36 +0000
[0x08048000]> e 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 the output, for example, by year:

```
[0x08048000]> pt ~1974 | wc -l
15
[0x08048000]> pt ~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. Check the manpage for more details, but these are the most important:

```
%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.
%d  The day of the month as a decimal number (range 01 to 31).
%D  Equivalent to %m/%d/%y.  (Yecch-for Americans only).
%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).
%m  The month as a decimal number (range 01 to 12).
%M  The minute as a decimal number (range 00 to 59).
%p  Either "AM" or "PM" according to the given time value.
%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  The time in 24-hour notation (%H:%M:%S).  (SU)
%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.
```

Basic Types

There are print modes available for all basic types. If you are interested in a more complex structure, type `pf??` for format characters and `pf???` for examples:

```
[0x00499999]> pf??
|pf: pf[.k[.f[=v]]|[v]]|[n]|[0|cnt][fmt] [a0 a1 ...]
| Format:
|  b       byte (unsigned)
|  B       resolve enum bitfield (see t?)
|  c       char (signed byte)
|  C       byte in decimal
|  d       0xHEX value (4 bytes) (see 'i' and 'x')
|  D       disassemble one opcode
|  e       temporally swap endian
|  E       resolve enum name (see t?)
|  f       float value (4 bytes)
|  F       double value (8 bytes)
|  i       signed integer value (4 bytes) (see 'd' and 'x')
|  n       next char specifies size of signed value (1, 2, 4 or 8 byte(s))
|  N       next char specifies size of unsigned value (1, 2, 4 or 8 byte(s))
|  o       octal value (4 byte)
|  p       pointer reference (2, 4 or 8 bytes)
|  q       quadword (8 bytes)
|  r       CPU register `pf r (eax)plop`
|  s       32bit pointer to string (4 bytes)
|  S       64bit pointer to string (8 bytes)
|  t       UNIX timestamp (4 bytes)
|  T       show Ten first bytes of buffer
|  u       uleb128 (variable length)
|  w       word (2 bytes unsigned short in hex)
|  x       0xHEX value and flag (fd @ addr) (see 'd' and 'i')
|  X       show formatted hexpairs
|  z       null terminated string
|  Z       null terminated wide string
|  ?       data structure `pf ? (struct_name)example_name`
|  *       next char is pointer (honors asm.bits)
|  +       toggle show flags for each offset
|  :       skip 4 bytes
|  .       skip 1 byte
|  ;       rewind 4 bytes
|  ,       rewind 1 byte
```

Use triple-question-mark `pf???` to get some examples using print format strings.

```
[0x00499999]> pf???
|pf: pf[.k[.f[=v]]|[v]]|[n]|[0|cnt][fmt] [a0 a1 ...]
| Examples:
| pf 3xi foo bar                               3-array of struct, each with named fields: 'foo' as hex, and 'bar' as int
| pf B (BitFldType)arg_name`                   bitfield type
| pf E (EnumType)arg_name`                     enum type
| pf.obj xxdz prev next size name              Define the obj format as xxdz
| pf obj=xxdz prev next size name              Same as above
| pf *z*i*w nb name blob                       Print the pointers with given labels
| pf iwq foo bar troll                         Print the iwq format with foo, bar, troll as the respective names for the fields
| pf 0iwq foo bar troll                        Same as above, but considered as a union (all fields at offset 0)
| pf.plop ? (troll)mystruct                    Use structure troll previously defined
| pfj.plop @ 0x14                              Apply format object at the given offset
```

```
| pf 10xiz pointer length string              Print a size 10 array of the xiz struct with its field names
| pf 5sqw string quad word                    Print an array with sqw struct along with its field names
| pf {integer}? (bifc)                        Print integer times the following format (bifc)
| pf [4]w[7]i                                 Print an array of 4 words and then an array of 7 integers
| pf ic...?i foo bar "(pf xw yo foo)troll" yo Print nested anonymous structures
| pf ;..x                                     Print value located 6 bytes from current offset
| pf [10]z[3]i[10]Zb                          Print an fixed size str, widechar, and var
| pfj +F @ 0x14                               Print the content at given offset with flag
| pf n2                                       print signed short (2 bytes) value. Use N instead of n for printing unsigned values
| pf [2]? (plop)structname @ 0                Prints an array of structs
| pf eqew bigWord beef                        Swap endianness and print with given labels
| pf.foo rr (eax)reg1 (eip)reg2               Create object referencing to register values
| pf tt troll plop                            print time stamps with labels troll and plop
```

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
- `pc*` print 'wx' r2 commands
- `pch` C half-words (2 byte)
- `pcw` C words (4 byte)
- `pcd` C dwords (8 byte)
- `pci` C array of bytes with instructions
- `pca` GAS .byte blob
- `pcA` .bytes with instructions in comments
- `pcs` string
- `pcS` shellscript that reconstructs the bin
- `pcj` json
- `pcJ` javascript
- `pco` Objective-C
- `pcp` python
- `pck` kotlin
- `pcr` rust
- `pcv` JaVa
- `pcV` V (vlang.io)
- `pcy` yara
- `pcz` Swift

If we need to create a .c file containing a binary blob, use the `pc` command, that creates this output. The default size is like in many other commands: the block size, which can be changed with the `b` command.

We can also just temporarily override this block size by expressing it as an argument.

```
[0xB7F8E810]> pc 32
#define _BUFFER_SIZE 32
unsigned char buffer[_BUFFER_SIZE] = {
0x89, 0xe0, 0xe8, 0x49, 0x02, 0x00, 0x00, 0x89, 0xc7, 0xe8, 0xe2, 0xff, 0xff, 0xff, 0x81, 0xc3, 0xd6, 0xa7, 0x01, 0x00, 0x8b, 0x83, 0x00, 0xff, 0xff, 0xf
```

That cstring can be used in many programming languages, not just C.

```
[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
```

Strings

Strings are probably one of the most important entry points 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[bijqpsuwWxz+] [N]  Print String
| ps       print string
| ps+[j]   print libc++ std::string (same-endian, ascii, zero-terminated)
| psb      print strings in current block
| psi      print string inside curseek
| psj      print string in JSON format
| psp[j]   print pascal string
| psq      alias for pqs
| pss      print string in screen (wrap width)
| psu[zj]  print utf16 unicode (json)
| psw[j]   print 16bit wide string
| psW[j]   print 32bit wide string
| psx      show string with escaped chars
| psz[j]   print zero-terminated string
```

Most strings are zero-terminated. Below there is an example 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 the 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:

```
$ radare2 /usr/lib/gstreamer-1.0/libgstflv.so
[0x00006020]> aa; pdf @ sym.gst_plugin_flv_get_desc
[x] Analyze all flags starting with sym. and entry0 (aa)
sym.gst_plugin_flv_get_desc ();
[...]
      0x00013830      488d0549db0000  lea rax, section..data.rel.ro ; 0x21380
      0x00013837      c3              ret
[0x00006020]> s section..data.rel.ro
[0x00021380]> pf ii*z*zp*z*z*z*z*z*z major minor name desc init version license source package origin release_datetime
            major : 0x00021380 = 1
            minor : 0x00021384 = 18
             name : (*0x19cf2)0x00021388 = "flv"
             desc : (*0x1b358)0x00021390 = "FLV muxing and demuxing plugin"
             init : 0x00021398 = (qword)0x0000000000013460
          version : (*0x19cae)0x000213a0 = "1.18.2"
          license : (*0x19ce1)0x000213a8 = "LGPL"
           source : (*0x19cd0)0x000213b0 = "gst-plugins-good"
          package : (*0x1b378)0x000213b8 = "GStreamer Good Plugins (Arch Linux)"
           origin : (*0x19cb5)0x000213c0 = "https://www.archlinux.org/"
 release_datetime : (*0x19cf6)0x000213c8 = "2020-12-06"
```

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

```
[0x00005310]> e asm.arch=??
_dAe  _8_16      6502        LGPL3   6502/NES/C64/Tamagotchi/T-1000 CPU
_dAe  _8         8051        PD      8051 Intel CPU
_dA_  _16_32     arc         GPL3    Argonaut RISC Core
a___  _16_32_64  arm.as      LGPL3   as ARM Assembler (use ARM_AS environment)
adAe  _16_32_64  arm         BSD     Capstone ARM disassembler
_dA_  _16_32_64  arm.gnu     GPL3    Acorn RISC Machine CPU
_d__  _16_32     arm.winedbg LGPL2   WineDBG's ARM disassembler
adAe  _8_16      avr         GPL     AVR Atmel
adAe  _16_32_64  bf          LGPL3   Brainfuck
_dA_  _32        chip8       LGPL3   Chip8 disassembler
_dA_  _16        cr16        LGPL3   cr16 disassembly plugin
_dA_  _32        cris        GPL3    Axis Communications 32-bit embedded processor
adA_  _32_64     dalvik      LGPL3   AndroidVM Dalvik
ad__  _16        dcpu16      PD      Mojang's DCPU-16
_dA_  _32_64     ebc         LGPL3   EFI Bytecode
adAe  _16        gb          LGPL3   GameBoy(TM) (z80-like)
_dAe  _16        h8300       LGPL3   H8/300 disassembly plugin
_dAe  _32        hexagon     LGPL3   Qualcomm Hexagon (QDSP6) V6
_d__  _32        hppa        GPL3    HP PA-RISC
_dAe  _0         i4004       LGPL3   Intel 4004 microprocessor
_dA_  _8         i8080       BSD     Intel 8080 CPU
adA_  _32        java        Apache  Java bytecode
_d__  _32        lanai       GPL3    LANAI
...
```

Configuring the Disassembler

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

```
[0x00005310]> e? asm.
asm.anal: Analyze code and refs while disassembling (see anal.strings)
asm.arch: Set the arch to be used by asm
asm.assembler: Set the plugin name to use when assembling
asm.bbline: Show empty line after every basic block
```

```
asm.bits: Word size in bits at assembler
asm.bytes: Display the bytes of each instruction
asm.bytespace: Separate hexadecimal bytes with a whitespace
asm.calls: Show callee function related info as comments in disasm
asm.capitalize: Use camelcase at disassembly
asm.cmt.col: Column to align comments
asm.cmt.flgrefs: Show comment flags associated to branch reference
asm.cmt.fold: Fold comments, toggle with Vz
...
```

Currently there are 136 `asm.` configuration variables so we do not list them all.

Disassembly Syntax

The `asm.syntax` variable is used to change the flavor of the 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 Intermediate 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:

```
[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:

```
[0x00005310]> fs?
|Usage: fs [*] [+-][flagspace|addr] # Manage flagspaces
| fs            display flagspaces
| fs*           display flagspaces as r2 commands
| fsj           display flagspaces in JSON
| fs *          select all flagspaces
| fs flagspace  select flagspace or create if it doesn't exist
| fs-flagspace  remove flagspace
| fs-*          remove all flagspaces
| fs+foo        push previous flagspace and set
| fs-           pop to the previous flagspace
| fs-.          remove the current flagspace
| fsq           list flagspaces in quiet mode
| fsm [addr]    move flags at given address to the current flagspace
| fss           display flagspaces stack
| fss*          display flagspaces stack in r2 commands
| fssj          display flagspaces stack in JSON
| fsr newname   rename selected flagspace
[0x00005310]> fs
0  439 * strings
1   17 * symbols
2   54 * sections
3   20 * segments
4  115 * relocs
5  109 * imports
[0x00005310]>
```

Here there are some command examples:

```
[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`.

Local flags

Every flag name should be unique for addressing reasons. But it is quite a common need to have the flags, for example inside the functions, with simple and ubiquitous names like `loop` or `return`. For this purpose you can use so called "local" flags, which are tied to the function where they reside. It is possible to add them using `f.` command:

```
[0x00003a04]> pd 10
      0x00003a04      48c705c9cc21.  mov qword [0x002206d8], 0xffffffffffffffff ;
[0x2206d8:8]=0
      0x00003a0f      c60522cc2100.  mov byte [0x00220638], 0     ; [0x220638:1]=0
      0x00003a16      83f802         cmp eax, 2
  .-< 0x00003a19      0f84880d0000   je 0x47a7
     0x00003a1f      83f803         cmp eax, 3
 .-< 0x00003a22      740e           je 0x3a32
    0x00003a24      83e801         sub eax, 1
.-< 0x00003a27      0f84ed080000   je 0x431a
    0x00003a2d      e8fef8ffff     call sym.imp.abort           ; void abort(void)
    ; CODE XREF from main (0x3a22)
  > 0x00003a32      be07000000     mov esi, 7
[0x00003a04]> f. localflag @ 0x3a32
[0x00003a04]> f.
0x00003a32 localflag   [main + 210]
[0x00003a04]> pd 10
      0x00003a04      48c705c9cc21.  mov qword [0x002206d8], 0xffffffffffffffff ;
[0x2206d8:8]=0
      0x00003a0f      c60522cc2100.  mov byte [0x00220638], 0     ; [0x220638:1]=0
```

```
      0x00003a16      83f802         cmp eax, 2
   . < 0x00003a19      0f84880d0000   je 0x47a7
      0x00003a1f      83f803         cmp eax, 3
  . < 0x00003a22      740e           je 0x3a32                    ; main.localflag
     0x00003a24      83e801         sub eax, 1
 .  < 0x00003a27      0f84ed080000   je 0x431a
     0x00003a2d      e8fef8ffff     call sym.imp.abort           ; void abort(void)
     ; CODE XREF from main (0x3a22)
 ` >  .localflag:
     ; CODE XREF from main (0x3a22)
 ` > 0x00003a32      be07000000     mov esi, 7
[0x00003a04]>
```

Flag Zones

radare2 offers flag zones, which lets you label different offsets on the scrollbar, for making it easier to navigate through large binaries. You can set a flag zone on the current seek using:

```
[0x00003a04]> fz flag-zone-name
```

Set `scr.scrollbar=1` and go to the Visual mode, to see your flag zone appear on the scrollbar on the right end of the window.

See `fz?` for more information. ## 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 given as a wide-string, 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[1248][+-][n]       increment/decrement byte,word..
| w foobar             write string 'foobar'
| w0 [len]             write 'len' bytes with value 0x00
| w6[de] base64/hex    write base64 [d]ecoded or [e]ncoded string
| wa[?] push ebp       write opcode, separated by ';' (use '"' around the command)
| waf f.asm            assemble file and write bytes
| waF f.asm            assemble file and write bytes and show 'wx' op with hexpair bytes of assembled code
| wao[?] op            modify opcode (change conditional of jump. nop, etc)
| wA[?] r 0            alter/modify opcode at current seek (see wA?)
| wb 010203            fill current block with cyclic hexpairs
| wB[-]0xVALUE         set or unset bits with given value
| wc                   list all write changes
| wc[?][jir+-*?]       write cache undo/commit/reset/list (io.cache)
| wd [off] [n]         duplicate N bytes from offset at current seek (memcpy) (see y?)
| we[?] [nNsxX] [arg]  extend write operations (insert instead of replace)
| wf[fs] -|file        write contents of file at current offset
| wh r2                whereis/which shell command
| wm f0ff              set binary mask hexpair to be used as cyclic write mask
| wo[?] hex            write in block with operation. 'wo?' fmi
| wp[?] -|file         apply radare patch file. See wp? fmi
| wr 10                write 10 random bytes
| ws pstring           write 1 byte for length and then the string
| wt[f][?] file [sz]   write to file (from current seek, blocksize or sz bytes)
| wts host:port [sz]   send data to remote host:port via tcp://
| ww foobar            write wide string 'f\x00o\x00o\x00b\x00a\x00r\x00'
| wx[?][fs] 9090       write two intel nops (from wxfile or wxseek)
| wv[?] eip+34         write 32-64 bit value honoring cfg.bigendian
| wz string            write zero terminated string (like w + \x00)
```

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  ; create sequence from 2 to 255 with step 3
|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
0x7fcd6a891630  4889 e7e8 6839 0000 4989 c48b 05ef 1622
0x7fcd6a891640  005a 488d 24c4 29c2 5248 89d6 4989 e548
0x7fcd6a891650  83e4 f048 8b3d 061a 2200 498d 4cd5 1049
0x7fcd6a891660  8d55 0831 ede8 06e2 0000 488d 15cf e600
[0x7fcd6a891630]> wox 90
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F
0x7fcd6a891630  d819 7778 d919 541b 90ca d81d c2d8 1946
0x7fcd6a891640  1374 60d8 b290 d91d 1dc5 98a1 9090 d81d
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
0x7fcd6a891630  d91b 787a 91cc d91f 1476 61da 1ec7 99a3
0x7fcd6a891640  91de 1a7e d91f 96db 14d9 9593 1401 9593
0x7fcd6a891650  c4da 1a6d e89a d959 9192 9159 1cb1 d959
0x7fcd6a891660  9192 79cb 81da 1652 81da 1456 a252 7c77
```

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 `pz` command, or just use `Z` in the visual mode to toggle the zoom mode.

The cursor can be used to scroll faster through the zoom out view. Pressing `z` again will zoom-in at the 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]> e zoom.byte=h
[0x08049790]> pz // or default pzh
0x00000000  7f00 0000 e200 0000 146e 6f74 0300 0000
0x00000010  0000 0000 0068 2102 00ff 2024 e8f0 007a
0x00000020  8c00 18c2 ffff 0080 4421 41c4 1500 5dff
0x00000030  ff10 0018 0fc8 031a 000c 8484 e970 8648
0x00000040  d68b 3148 348b 03a0 8b0f c200 5d25 7074
0x00000050  7500 00e1 ffe8 58fe 4dc4 00e0 dbc8 b885
```

```
[0x08049790]> e zoom.byte=p
[0x08049790]> pz // or pzp
0x00000000  2f47 0609 070a 0917 1e9e a4bd 2a1b 2c27
0x00000010  322d 5671 8788 8182 5679 7568 82a2 7d89
0x00000020  8173 7f7b 727a 9588 a07b 5c7d 8daf 836d
0x00000030  b167 6192 a67d 8aa2 6246 856e 8c9b 999f
0x00000040  a774 96c3 b1a4 6c8e a07c 6a8f 8983 6a62
0x00000050  7d66 625f 7ea4 7ea6 b4b6 8b57 a19f 71a2
```

```
[0x08049790]> eval zoom.byte = flags
[0x08049790]> pz // or pzf
0x00406e65  48d0 80f9 360f 8745 ffff ffeb ae66 0f1f
0x00406e75  4400 0083 f801 0f85 3fff ffff 410f b600
0x00406e85  3c78 0f87 6301 0000 0fb6 c8ff 24cd 0026
0x00406e95  4100 660f 1f84 0000 0000 0084 c074 043c
0x00406ea5  3a75 18b8 0500 0000 83f8 060f 95c0 e9cd
0x00406eb5  feff ff0f 1f84 0000 0000 0041 8801 4983
0x00406ec5  c001 4983 c201 4983 c101 e9ec feff ff0f
```

```
[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
0x00000020  222a 2330 2b31 2e2a 1714 200d 1512 383d
0x00000030  1e1a 181b 0a10 1a21 2a36 281e 1d1c 0e11
0x00000040  1b2a 2f22 2229 181e 231e 181c 1913 262b
0x00000050  2b30 4741 422f 382a 1e22 0f17 0f10 3913
```

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

```
[0x00003a04]> e? zoom.
zoom.byte: Zoom callback to calculate each byte (See pz? for help)
zoom.from: Zoom start address
zoom.in: Specify  boundaries for zoom
zoom.maxsz: Zoom max size of block
zoom.to: Zoom end address
[0x00003a04]> e zoom.
zoom.byte = h
zoom.from = 0
zoom.in = io.map
zoom.maxsz = 512
```

```
zoom.to = 0
```

Yank/Paste

Radare2 has an internal clipboard to save and write portions of memory loaded from the current io layer.

This clipboard can be manipulated with the `y` command.

The two basic operations are

- copy (yank)
- paste

The yank operation will read N bytes (specified by the argument) into the clipboard. We can later use the `yy` command to paste what we read before into a file.

You can yank/paste bytes in visual mode selecting them with the cursor mode (`Vc`) and then using the `y` and `Y` key bindings which are aliases for `y` and `yy` commands of the command-line interface.

```
[0x00000000]> y?
Usage: y[ptxy] [len] [[@]addr]   # See wd? for memcpy, same as 'yf'.
| y!              open cfg.editor to edit the clipboard
| y 16 0x200      copy 16 bytes into clipboard from 0x200
| y 16 @ 0x200    copy 16 bytes into clipboard from 0x200
| y 16            copy 16 bytes into clipboard
| y               show yank buffer information (srcoff len bytes)
| y*              print in r2 commands what's been yanked
| yf 64 0x200     copy file 64 bytes from 0x200 from file
| yfa file copy   copy all bytes from file (opens w/ io)
| yfx 10203040    yank from hexpairs (same as ywx)
| yj              print in JSON commands what's been yanked
| yp              print contents of clipboard
| yq              print contents of clipboard in hexpairs
| ys              print contents of clipboard as string
| yt 64 0x200     copy 64 bytes from current seek to 0x200
| ytf file        dump the clipboard to given file
| yw hello world  yank from string
| ywx 10203040    yank from hexpairs (same as yfx)
| yx              print contents of clipboard in hexadecimal
| yy 0x3344       paste clipboard
| yz [len]        copy nul-terminated string (up to blocksize) into 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

For most generic reverse engineering tasks like finding the differences between two binary files, which bytes has changed, find differences in the graphs of the code analysis results, and other diffing operations you can just use radiff2:

```
$ radiff2 -h
```

Inside r2, the functionalities exposed by radiff2 are available with the `c` command.

`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]   # Compare
| c [string]               Compare a plain with escaped chars string
| c* [string]              Same as above, but printing r2 commands instead
| c1 [addr]                Compare 8 bits from current offset
| c2 [value]               Compare a word from a math expression
| c4 [value]               Compare a doubleword from a math expression
| c8 [value]               Compare a quadword from a math expression
| cat [file]               Show contents of file (see pwd, ls)
| cc [at]                  Compares in two hexdump columns of block size
| ccc [at]                 Same as above, but only showing different lines
| ccd [at]                 Compares in two disasm columns of block size
| ccdd [at]                Compares decompiler output (e cmd.pdc=pdg|pdd)
| cf [file]                Compare contents of file at current seek
| cg[?] [o] [file]         Graphdiff current file and [file]
| cu[?] [addr] @at         Compare memory hexdumps of $$ and dst in unified diff
| cud [addr] @at           Unified diff disasm from $$ and given address
| cv[1248] [hexpairs] @at  Compare 1,2,4,8-byte (silent return in $?)
| cV[1248] [addr] @at      Compare 1,2,4,8-byte address contents (silent, return in $?)
| cw[?] [us?] [...]        Compare memory watchers
| cx [hexpair]             Compare hexpair string (use '.' as nibble wildcard)
| cx* [hexpair]            Compare hexpair string (output r2 commands)
| cX [addr]                Like 'cc' but using hexdiff output
```

```
| cd [dir]                  chdir
| cl|cls|clear              Clear screen, (clear0 to goto 0, 0 only)
```

To compare memory contents at current seek position against a 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 the `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, be math expressions which use flag names and anything allowed in an expression:

```
[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
```

SDB

SDB stands for String DataBase. It's a simple key-value database that only operates with strings created by pancake. It is used in many parts of r2 to have a disk and in-memory database which is small and fast to manage using it as a hashtable on steroids.

SDB is a simple string key/value database based on djb's cdb disk storage and supports JSON and arrays introspection.

There's also the sdbtypes: a vala library that implements several data structures on top of an sdb or a memcache instance.

SDB supports:

- namespaces (multiple sdb paths)
- atomic database sync (never corrupted)
- bindings for vala, luvit, newlisp and nodejs
- commandline frontend for sdb databases
- memcache client and server with sdb backend
- arrays support (syntax sugar)
- json parser/getter

Usage example

Let's create a database!

```
$ sdb d hello=world
$ sdb d hello
world
```

Using arrays:

```
$ sdb - '[]list=1,2' '[0]list' '[0]list=foo' '[]list' '[+1]list=bar'
1
foo
2
foo
bar
2
```

Let's play with json:

```
$ sdb d g='{"foo":1,"bar":{"cow":3}}'
$ sdb d g?bar.cow
3
$ sdb - user='{"id":123}' user?id=99 user?id
99
```

Using the command line without any disk database:

```
$ sdb - foo=bar foo a=3 +a -a
bar
4
3

$ sdb -
foo=bar
foo
bar
a=3
+a
```

```
4
-a
3
```

Remove the database

```
$ rm -f d
```

So what ?

So, you can now do this inside your radare2 sessions!

Let's take a simple binary, and check what is already *sdbized*.

```
$ cat test.c
int main(){
    puts("Hello world\n");
}
$ gcc test.c -o test
```

```
$ r2 -A ./test
[0x08048320]> k **
bin
anal
syscall
debug
```

```
[0x08048320]> k bin/**
fd.6
[0x08048320]> k bin/fd.6/*
archs=0:0:x86:32
```

The file corresponding to the sixth file descriptor is a x86_32 binary.

```
[0x08048320]> k anal/meta/*
meta.s.0x80484d0=12,SGVsbG8gd29ybGQ=
[...]
[0x08048320]> ?b64- SGVsbG8gd29ybGQ=
Hello world
```

Strings are stored encoded in base64.

More Examples

List namespaces

```
k **
```

List sub-namespaces

```
k anal/**
```

List keys

```
k *
k anal/*
```

Set a key

```
k foo=bar
```

Get the value of a key

```
k foo
```

List all syscalls

```
k syscall/*~^0x
```

List all comments

```
k anal/meta/*~.C.
```

Show a comment at given offset:

```
k %anal/meta/[1]meta.C.0x100005000
```

Dietline

Radare2 comes with the lean readline-like input capability through the lean library to handle the command edition and history navigation. It allows users to perform cursor movements, search the history, and implements autocompletion. Moreover, due to the radare2 portability, dietline provides the uniform experience among all supported platforms. It is used in all radare2 subshells - main prompt, SDB shell, visual prompt, and offsets prompt. It also implements the most common features and keybindings compatible with the GNU Readline.

Dietline supports two major configuration modes : Emacs-mode and Vi-mode.

It also supports the famous `Ctrl-R` reverse history search. Using `TAB` key it allows to scroll through the autocompletion options.

Autocompletion

In the every shell and radare2 command autocompletion is supported. There are multiple modes of it - files, flags, and SDB keys/namespaces. To provide the easy way to select possible completion options the scrollable popup widget is available. It can be enabled with `scr.prompt.popup`, just set it to the `true`.

Emacs (default) mode

By default dietline mode is compatible with readline Emacs-like mode key bindings. Thus active are:

Moving

- `Ctrl-a` - move to the beginning of the line
- `Ctrl-e` - move to the end of the line
- `Ctrl-b` - move one character backward
- `Ctrl-f` - move one character forward

Deleting

- `Ctrl-w` - delete the previous word
- `Ctrl-u` - delete the whole line
- `Ctrl-h` - delete a character to the left
- `Ctrl-d` - delete a character to the right
- `Alt-d` - cuts the character after the cursor

Killing and Yanking

- `Ctrl-k` - kill the text from point to the end of the line.
- `Ctrl-x` - kill backward from the cursor to the beginning of the current line.
- `Ctrl-t` - kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as forward-word.
- `Ctrl-w` - kill the word behind point, using white space as a word boundary. The killed text is saved on the kill-ring.
- `Ctrl-y` - yank the top of the kill ring into the buffer at point.
- `Ctrl-]` - rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

History

- `Ctrl-r` - the reverse search in the command history

Vi mode

Radare2 also comes with in vi mode that can be enabled by toggling `scr.prompt.vi`. The various keybindings available in this mode are:

Entering command modes

- `ESC` - enter into the control mode
- `i` - enter into the insert mode

Moving

- `j` - acts like up arrow key
- `k` - acts like down arrow key
- `a` - move cursor forward and enter into insert mode
- `I` - move to the beginning of the line and enter into insert mode
- `A` - move to the end of the line and enter into insert mode
- `^` - move to the beginning of the line
- `0` - move to the beginning of the line
- `$` - move to the end of the line
- `h` - move one character backward
- `l` - move one character forward

Deleting and Yanking

- `x` - cuts the character
- `dw` - delete the current word
- `diw` - deletes the current word.
- `db` - delete the previous word
- `D` - delete the whole line
- `dh` - delete a character to the left
- `dl` - delete a character to the right
- `d$` - kill the text from point to the end of the line.
- `d^` - kill backward from the cursor to the beginning of the current line.
- `de` - kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as forward-word.
- `p` - yank the top of the kill ring into the buffer at point.
- `c` - acts similar to d based commands, but goes into insert mode in the end by prefixing the commands with numbers, the command is performed multiple times.

If you are finding it hard to keep track of which mode you are in, just set `scr.prompt.mode=true` to update the color of the prompt based on the vi-mode.

Visual Mode

The visual mode is a more user-friendly interface alternative to radare2's command-line prompt. It allows easy navigation, 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`.

Navigation

Navigation can be done using HJKL or arrow keys and PgUp/PgDown keys. It also understands usual Home/End keys. Like in Vim the movements can be repeated by preceding the navigation key with the number, for example `5j` will move down for 5 lines, or `2l` will move 2 characters right.

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** → **Hex-less hexdump 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
 ??       show the user-friendly hud
 %        in cursor mode finds matching pair, or toggle autoblocksz
 @        redraw screen every 1s (multi-user view)
 ^        seek to the begining of the function
 !        enter into the visual panels mode
 _        enter the flag/comment/functions/.. hud (same as VF_)
 =        set cmd.vprompt (top row)
 |        set cmd.cprompt (right column)
```

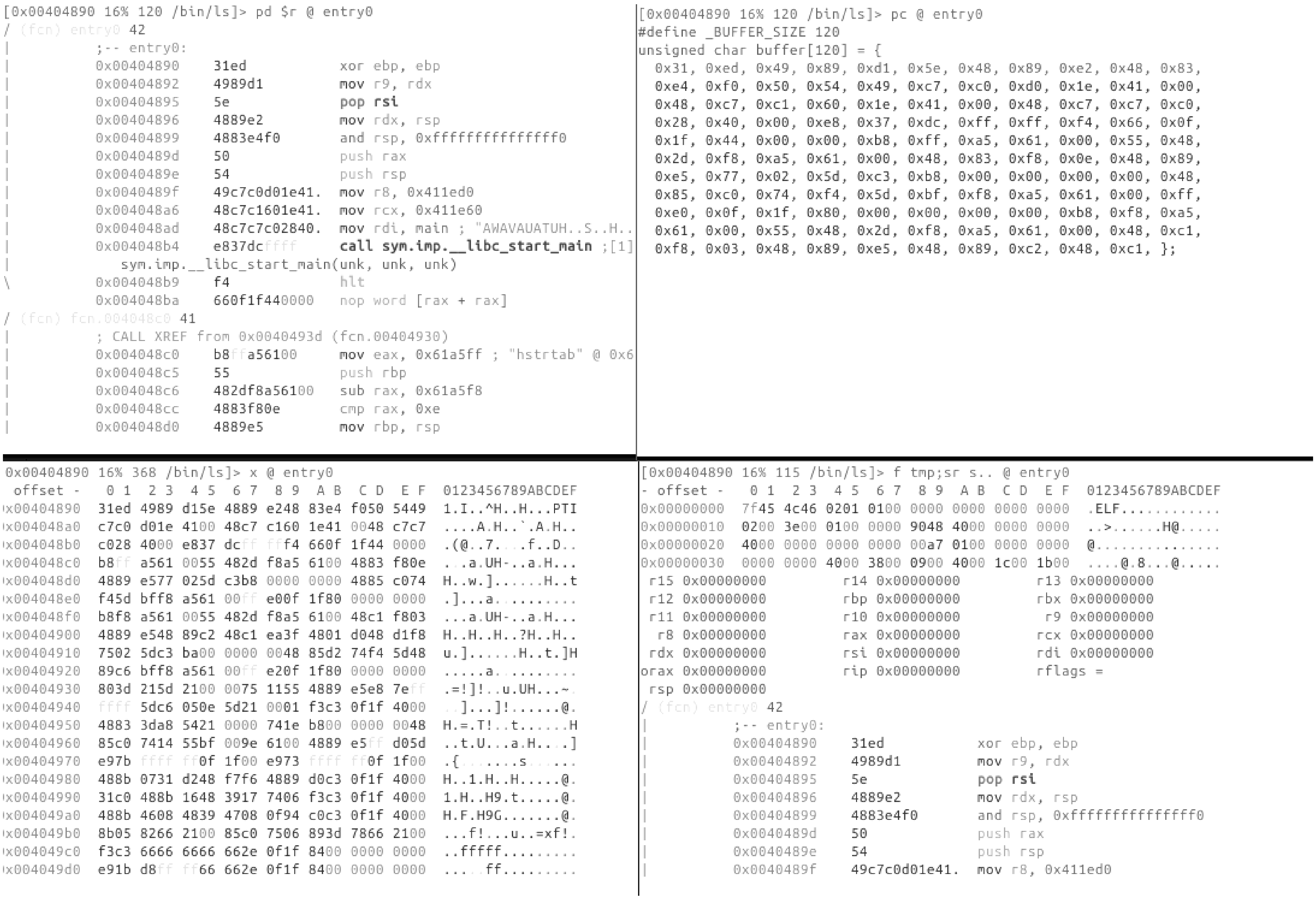

Figure 7: Visual Mode

```
 .         seek to program counter
 \         toggle visual split mode
 "         toggle the column mode (uses pC..)
 /         in cursor mode search in current block
 :cmd      run radare command
 ;[-]cmt   add/remove comment
 0         seek to beginning of current function
 [1-9]     follow jmp/call identified by shortcut (like ;[1])
 ,file     add a link to the text file
 /*+-[]    change block size, [] = resize hex.cols
 </>       seek aligned to block size (seek cursor in cursor mode)
 a/A       (a)ssemble code, visual (A)ssembler
 b         browse symbols, flags, configurations, classes, ...
 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)
 g         go/seek to given offset
 O         toggle asm.pseudo and asm.esil
 p/P       rotate print modes (hex, disasm, debug, words, buf)
 q         back to radare shell
 r         refresh screen / in cursor mode browse comments
 R         randomize color palette (ecr)
 sS        step / step over
 t         browse types
 T         enter textlog chat console (TT)
 uU        undo/redo seek
 v         visual function/vars 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         fold/unfold comments in disassembly
 Z         toggle zoom mode
 Enter     follow address of jump/call
Function Keys: (See 'e key.'), defaults to:
  F2       toggle breakpoint
```

```
F4      run to cursor
F7      single step
F8      step over
F9      continue
```

Visual Disassembly

Navigation

Move within the Disassembly using arrow keys or `hjkl`. Use `g` 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
...
```

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.

```
[0x00404890 16% 330 (0x6:-1=1)]> pd $r @ entry0+6 # 0x404896
/ (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
|           0x0040489d    50             push rax
|           0x0040489e    54             push rsp
|           0x0040489f    49c7c0d01e41.  mov r8, 0x411ed0
|           0x004048a6    48c7c1601e41.  mov rcx, 0x411e60
|           0x004048ad    48c7c7c02840.  mov rdi, main                ; "AWAVAUATUH..S..H...." @ 0x4028c0
|           0x004048b4    e837dcffff     call sym.imp.__libc_start_main ;[1]
|              sym.imp.__libc_start_main(unk, unk)
\           0x004048b9    f4             hlt
            0x004048ba    660f1f440000   nop word [rax + rax]
```

Figure 8: Cursor at 0x00404896

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 instructions 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 instructions, separate them with semicolons, `;`.

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`.

Function Argument display

To enable this view use this config var `e dbg.funcarg = true`

Figure 9: funcarg

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.

Hud input mode can be closed using ^C. It will also exit when backspace is pressed when the user input string is empty.

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.

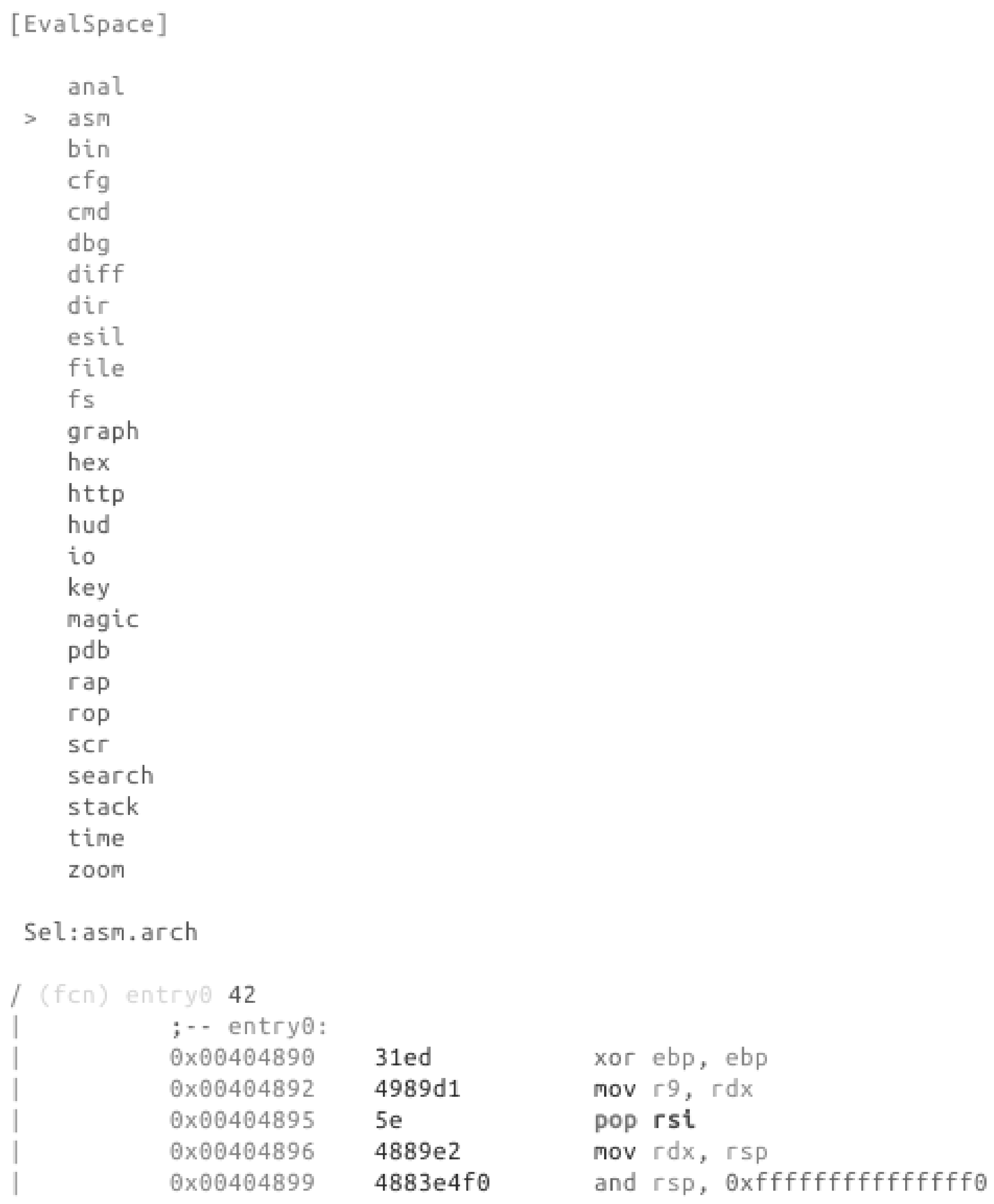

Figure 10: First Select asm

Example switch to pseudo disassembly:

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=?`

```
[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
```

Figure 11: Pseudo disassembly disabled

```
[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
```

Figure 12: Pseudo disassembly enabled

```
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  xor ebp, ebp   ; logical exclusive or
0x00404872  mov r9, rdx    ; moves data from src to dst
0x00404875  pop rsi        ; pops last element of stack and stores the result in argument
0x00404876  mov rdx, rsp   ; moves data from src to dst
0x00404879  and rsp, -0xf  ; binary and operation between src and dst, stores result on dst
```

Visual Assembler

You can use Visual Mode to assemble code using `A`. For example let's replace the `push` by a `jmp`:

Notice the preview of the disassembly and arrows:

You need to open the file in writing mode (`r2 -w` or `oo+`) in order to patch the file. You can also use the cache mode: `e io.cache = true` and `wc?`.

Remember that patching files in debug mode only patch the memory not the file.

Visual Configuration Editor

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

Example switch to pseudo disassembly:

Visual Panels

Concept

Visual Panels is characterized by the following core functionalities:

1. Split Screen
2. Display multiple screens such as Symbols, Registers, Stack, as well as custom panels
3. Menu will cover all those commonly used commands for you so that you don't have to memorize any of them

CUI met some useful GUI as the menu, that is Visual Panels.

Panels can be accessed by using `v` or by using `!` from the visual mode.

Overview

Commands

```
|Visual Ascii Art Panels:
| |      split the current panel vertically
| -      split the current panel horizontally
| :      run r2 command in prompt
| ;      add/remove comment
| _      start the hud input mode
| \      show the user-friendly hud
| ?      show this help
| !      run r2048 game
| .      seek to PC or entrypoint
| *      show decompiler in the current panel
| "      create a panel from the list and replace the current one
| /      highlight the keyword
| (      toggle snow
| &      toggle cache
| [1-9]  follow jmp/call identified by shortcut (like ;[1])
| ' '    (space) toggle graph / panels
| tab    go to the next panel
```

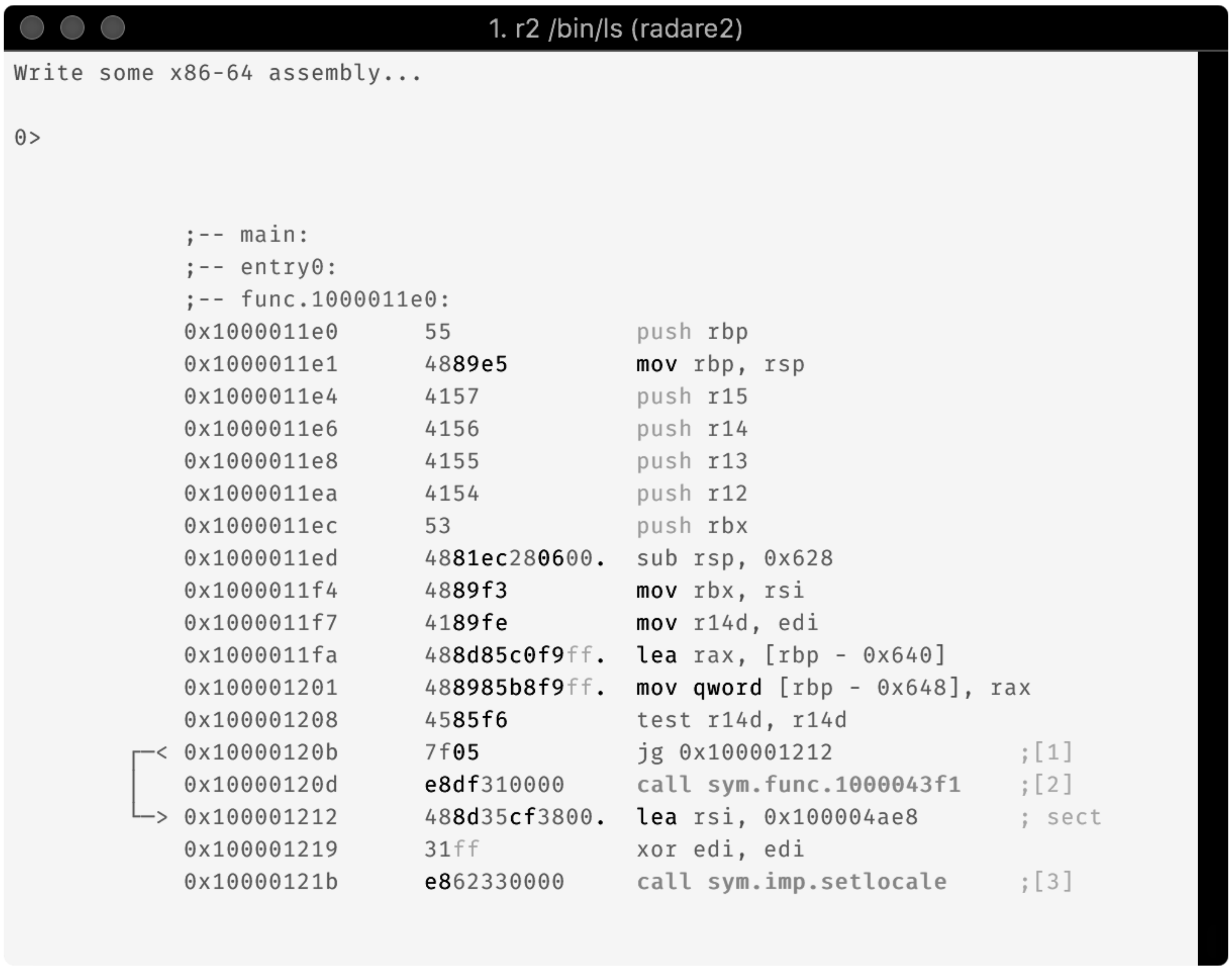

Figure 13: Before

```
1. r2 /bin/ls (radare2)
Write some x86-64 assembly...

2> jmp 0x1000011ec
* eb0a

        ┌─< ;-- main:
        ┌─< ;-- entry0:
        ┌─< ;-- func.1000011e0:
        ┌─< 0x1000011e0      eb0a           jmp 0x1000011ec            ;[1]
        │   0x1000011e2      89e5           mov ebp, esp
        │   0x1000011e4      4157           push r15
        │   0x1000011e6      4156           push r14
        │   0x1000011e8      4155           push r13
        │   0x1000011ea      4154           push r12
        └─> 0x1000011ec      53             push rbx
            0x1000011ed      4881ec280600.  sub rsp, 0x628
            0x1000011f4      4889f3         mov rbx, rsi
            0x1000011f7      4189fe         mov r14d, edi
            0x1000011fa      488d85c0f9ff.  lea rax, [rbp - 0x640]
            0x100001201      488985b8f9ff.  mov qword [rbp - 0x648], rax
            0x100001208      4585f6         test r14d, r14d
        ┌─< 0x10000120b      7f05           jg 0x100001212             ;[2]
        │   0x10000120d      e8df310000     call sym.func.1000043f1    ;[3]
        └─> 0x100001212      488d35cf3800.  lea rsi, 0x100004ae8       ; sect
            0x100001219      31ff           xor edi, edi
            0x10000121b      e862330000     call sym.imp.setlocale     ;[4]
```

Figure 14: After

```
[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
```

Figure 15: First Select asm

```
[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
```

Figure 16: Pseudo disassembly disabled

```
[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
```

Figure 17: Pseudo disassembly enabled

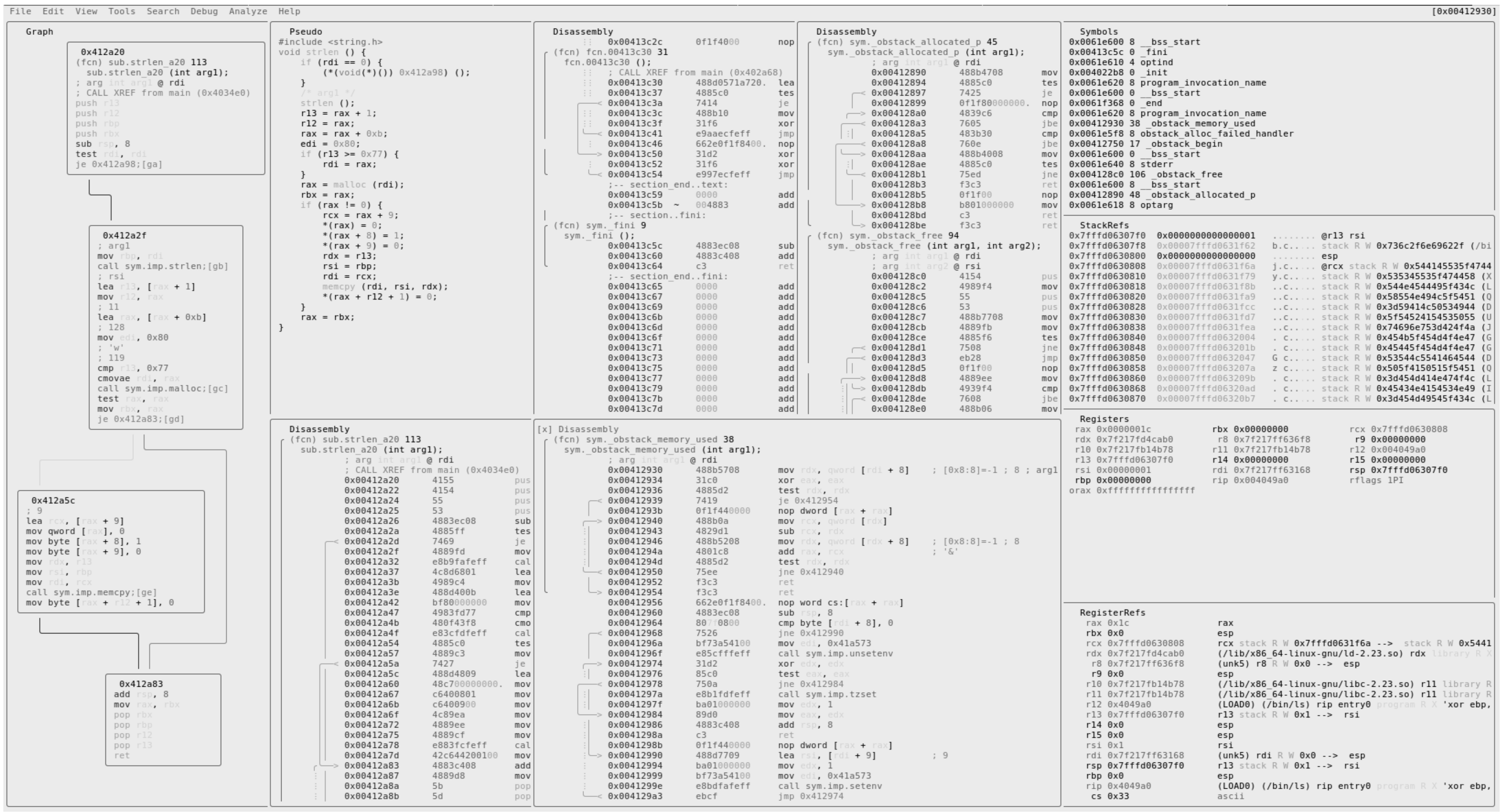

Figure 18: Panels Overview

```
| Enter  start Zoom mode
| a      toggle auto update for decompiler
| b      browse symbols, flags, configurations, classes, ...
| c      toggle cursor
| C      toggle color
| d      define in the current address. Same as Vd
| D      show disassembly in the current panel
| e      change title and command of current panel
| f      set/add filter keywords
| F      remove all the filters
| g      go/seek to given offset
| G      go/seek to highlight
| i      insert hex
| hjkl   move around (left-down-up-right)
| HJKL   move around (left-down-up-right) by page
| m      select the menu panel
| M      open new custom frame
| n/N    seek next/prev function/flag/hit (scr.nkey)
| p/P    rotate panel layout
| q      quit, or close a tab
| Q      close all the tabs and quit
| r      toggle callhints/jmphints/leahints
| R      randomize color palette (ecr)
| s/S    step in / step over
| t/T    tab prompt / close a tab
| u/U    undo / redo seek
| w      start Window mode
| V      go to the graph mode
| xX     show xrefs/refs of current function from/to data/code
| z      swap current panel with the first one
```

Basic Usage

Use `tab` to move around the panels until you get to the targeted panel. Then, use `hjkl`, just like in vim, to scroll the panel you are currently on. Use `S` and `s` to step over/in, and all the panels should be updated dynamically while you are debugging. Either in the Registers or Stack panels, you can edit the values by inserting hex. This will be explained later. While hitting `tab` can help you moving between panels, it is highly recommended to use `m` to open the menu. As usual, you can use `hjkl` to move around the menu and will find tons of useful stuff there. You can also press `"` to quickly browse through the different options View offers and change the contents of the selected panel.

Split Screen

`|` is for the vertical and `-` is for the horizontal split. You can delete any panel by pressing `X`.

Split panels can be resized from Window Mode, which is accessed with `w`.

Window Mode Commands

```
|Panels Window mode help:
| ?      show this help
| ??     show the user-friendly hud
| Enter  start Zoom mode
| c      toggle cursor
| hjkl   move around (left-down-up-right)
| JK     resize panels vertically
| HL     resize panels horizontally
| q      quit Window mode
```

Edit Values

Either in the Register or Stack panel, you can edit the values. Use `c` to activate cursor mode and you can move the cursor by pressing `hjkl`, as usual. Then, hit `i`, just like the insert mode of vim, to insert a value.

Tabs

Visual Panels also offer tabs to quickly access multiple forms of information easily. Press `t` to enter Tab Mode. All the tabs numbers will be visible in the top right corner.

By default you will have one tab and you can press `t` to create a new tab with the same panels and `T` to create a new panel from scratch.

For traversing through the tabs, you can type in the tab number while in Tab Mode.

And pressing `-` deletes the tab you are in.

Saving layouts

You can save your custom layout of your visual panels either by picking the option 'Save Layout' from the File menu of the menu bar or by running:

```
v= test
```

Where `test` is the name with which you'd like to save it.

You can open a saved layout by passing the name as the parameter to `v`:

```
v test
```

More about that can be found under `v?`. # 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: /[!bf] [arg]Search stuff (see 'e??search' for options)
|Use io.va for searching in non virtual addressing spaces
| / foo\x00               search for string 'foo\0'
| /j foo\x00              search for string 'foo\0' (json output)
| /! ff                   search for first occurrence not matching, command modifier
| /!x 00                  inverse hexa search (find first byte != 0x00)
| /+ /bin/sh              construct the string with chunks
| //                      repeat last search
| /a jmp eax              assemble opcode and search its bytes
| /A jmp                  find analyzed instructions of this type (/A? for help)
| /b                      search backwards, command modifier, followed by other command
| /B                      search recognized RBin headers
| /c jmp [esp]            search for asm code matching the given string
| /ce rsp,rbp             search for esil expressions matching
| /C[ar]                  search for crypto materials
| /d 101112               search for a deltified sequence of bytes
| /e /E.F/i               match regular expression
| /E esil-expr            offset matching given esil expressions %%= here
| /f                      search forwards, command modifier, followed by other command
| /F file [off] [sz]      search contents of file with offset and size
| /g[g] [from]            find all graph paths A to B (/gg follow jumps, see search.count and
anal.depth)
| /h[t] [hash] [len]      find block matching this hash. See ph
| /i foo                  search for string 'foo' ignoring case
| /m magicfile            search for matching magic file (use blocksize)
| /M                      search for known filesystems and mount them automatically
| /o [n]                  show offset of n instructions backward
| /O [n]                  same as /o, but with a different fallback if anal cannot be used
| /p patternsize          search for pattern of given size
| /P patternsize          search similar blocks
| /r[erwx][?] sym.printf  analyze opcode reference an offset (/re for esil)
| /R [grepopcode]         search for matching ROP gadgets, semicolon-separated
| /s                      search for all syscalls in a region (EXPERIMENTAL)
| /v[1248] value          look for an `cfg.bigendian` 32bit value
| /V[1248] min max        look for an `cfg.bigendian` 32bit value in range
| /w foo                  search for wide string 'f\0o\0o\0'
| /wi foo                 search for wide string ignoring case 'f\0o\0o\0'
| /x ff..33               search for hex string ignoring some nibbles
| /x ff0033               search for hex string
| /x ff43:ffd0            search for hexpair with mask
| /z min max              search for strings of given size
```

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.

note that '/' *starts multiline comment. It's not for searching. type* '/' to end comment.

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, and they will have 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
```

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

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

If you want to mask some nibble during the search you can use the symbol `.` to allow any nibble value to match:

```
[0x00407354]> /x 80..80
0x0040d4b6 hit3_0 800080
0x0040d4c8 hit3_1 808080
0x004058a6 hit3_2 80fb80
```

You may not know some bit values of your hexadecimal pattern. Thus you may use a bit mask on your pattern. Each bit set to one in the mask indicates to search the bit value in the pattern. A bit set to zero in the mask indicates that the value of a matching value can be 0 or 1:

```
[0x00407354]> /x 808080:ff80ff
0x0040d4c8 hit4_0 808080
0x0040d7b0 hit4_1 808080
0x004058a6 hit4_2 80fb80
```

You can notice that the command `/x 808080:ff00ff` is equivalent to the command `/x 80..80`.

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]   ; pecify search boundarie. Supported values are listed under e search.in=??
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` configuration 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

Sometimes you want to find a keyword backwards. This is, before the current offset, to do this you can seek back and search forward by adding some search.from/to restrictions, or use the `/b` command.

```
[0x100001200]> / nop
0x100004b15 hit0_0 .STUWabcdefghiklmnopqrstuvwxbin/ls.
0x100004f50 hit0_1 .STUWabcdefghiklmnopqrstuwx1] [file .
[0x100001200]> /b nop
[0x100001200]> s 0x100004f50p
[0x100004f50]> /b nop
0x100004b15 hit2_0 .STUWabcdefghiklmnopqrstuvwxbin/ls.
[0x100004f50]>
```

Note that `/b` is doing the same as `/`, but backward, so what if we want to use `/x` backward? We can use `/bx`, and the same goes for other search subcommands:

```
[0x100001200]> /x 90
0x100001a23 hit1_0 90
0x10000248f hit1_1 90
0x1000027b2 hit1_2 90
0x100002b2e hit1_3 90
0x1000032b8 hit1_4 90
0x100003454 hit1_5 90
0x100003468 hit1_6 90
0x10000355b hit1_7 90
0x100003647 hit1_8 90
0x1000037ac hit1_9 90
0x10000389c hit1_10 90
0x100003c5c hit1_11 90

[0x100001200]> /bx 90
[0x100001200]> s 0x10000355b
[0x10000355b]> /bx 90
0x100003468 hit3_0 90
0x100003454 hit3_1 90
0x1000032b8 hit3_2 90
0x100002b2e hit3_3 90
0x1000027b2 hit3_4 90
0x10000248f hit3_5 90
0x100001a23 hit3_6 90
[0x10000355b]>
```

Assembler Search

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

The command `/ad/ jmp [esp]` searches for the specified category of assembly mnemonic:

```
[0x00404888]> /ad/ 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
```

If you are simply looking for plaintext AES keys in your binary, `/Ca` will not find them, but you might want to search with `is~AES` instead if the programmer left those hints for you:

```
[0x00000000]> /Ca
Searching 0 byte in [0x100000-0x1f0000]
hits: 0
Searching 0 byte in [0x196e4-0x1b91c]
hits: 0
Searching 0 byte in [0x194b4-0x196e4]
hits: 0
Searching 0 byte in [0x8000-0x114b4]
hits: 0
[0x00000000]> is~AES
```

```
010 0x000096d4 0x000196d4 GLOBAL    OBJ   16 AES_KEY
```

Other than that, AES keys might show up in different ways in the binary: encrypted, hidden by another encrypting routine, so there's no absolute way other than understanding the binary being analized. For instance, `p=e` might give some hints if high(er) entropy sections are found trying to cover up a hardcoded secret. As an example on entropy searching, since radare 3.2.0, there's the possibility to delimit entropy sections for later use like so:

```
[0x00000000]> b
0x100
[0x00000000]> b 4096
[0x00000000]> /s
0x00100000 - 0x00101000 ~ 5.556094
0x014e2c88 - 0x014e3c88 ~ 0.000000
0x01434374 - 0x01435374 ~ 6.332087
0x01435374 - 0x0144c374 ~ 3.664636
0x0144c374 - 0x0144d374 ~ 1.664368
0x0144d374 - 0x0144f374 ~ 4.229199
0x0144f374 - 0x01451374 ~ 2.000000
(...)
[0x00000000]> /s*
f entropy_section_0 0x00001000 0x00100000
f entropy_section_1 0x00001000 0x014e2c88
f entropy_section_2 0x00001000 0x01434374
f entropy_section_3 0x00017000 0x01435374
f entropy_section_4 0x00001000 0x0144c374
f entropy_section_5 0x00002000 0x0144d374
f entropy_section_6 0x00002000 0x0144f374
```

The blocksize is increased to 4096 bytes from the default 100 bytes so that the entropy search `/s` can work on reasonably sized chunks for entropy analysis. The sections flags can be applied with the dot operator, `./s*` and then looped through `px 32 @@ entropy*`.

Moreover AES keys might be referenced from strings or pointed from the imports, for instance, so the `/` and other search-related commands can come in handy in this regard.

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 or other supported architectures.

It was a working and unix friendly solution, but it was inefficient as it repeated the same expensive actions over and over, because there were no caches. As a result, scrolling was terribly slow.

So there was a need to create a generic disassembler library to support multiple plugins for different architectures. We can list the current loaded plugins with

```
$ rasm2 -L
```

Or from inside radare2:

```
> e asm.arch=??
```

This was many years before capstone appeared. So r2 was using udis86 and olly disassemblers, many gnu (from binutils).

Nowadays, the disassembler support is one of the basic features of radare. It now has many options, endianness, 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 capabilities essential. Radare offers multiple ways to store and retrieve such metadata.

By following common basic UNIX principles, it is easy to write a small utility in a scripting language which uses `objdump`, `otool` or any other existing utility to obtain information from a binary and to import it into radare. For example, take a look at `idc2r.py` shipped with radare2ida. 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 and other data. 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
```

Please keep in mind that importing IDA Pro metadata from IDC dump is deprecated mechanism and might not work in the future. The recommended way to do it - use python-idb-based `ida2r2.py` which opens IDB files directly without IDA Pro installed.

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.

There are many different metadata manipulation commands, here is the glimpse of all of them:

```
[0x00404cc0]> C?
| Usage: C[-LCvsdfm*?][*?] [...]   # Metadata management
| C                                              list meta info in human friendly form
| C*                                             list meta info in r2 commands
| C*.                                            list meta info of current offset in r2 commands
| C- [len] [[@]addr]                             delete metadata at given address range
| C.                                             list meta info of current offset in human friendly form
| CC! [@addr]                                    edit comment with $EDITOR
| CC[?] [-] [comment-text] [@addr]               add/remove comment
| CC.[addr]                                      show comment in current address
| CCa[-at]|[at] [text] [@addr]                   add/remove comment at given address
| CCu [comment-text] [@addr]                     add unique comment
| CF[sz] [fcn-sign..] [@addr]                    function signature
| CL[-][*] [file:line] [addr]                    show or add 'code line' information (bininfo)
| CS[-][space]                                   manage meta-spaces to filter comments, etc..
| C[Cthsdmf]                                     list comments/types/hidden/strings/data/magic/formatted in human friendly form
| C[Cthsdmf]*                                    list comments/types/hidden/strings/data/magic/formatted in r2 commands
| Cd[-] [size] [repeat] [@addr]                  hexdump data array (Cd 4 10 == dword [10])
| Cd. [@addr]                                    show size of data at current address
| Cf[?][-] [sz] [0|cnt][fmt] [a0 a1...] [@addr]  format memory (see pf?)
| Ch[-] [size] [@addr]                           hide data
| Cm[-] [sz] [fmt..] [@addr]                     magic parse (see pm?)
| Cs[?] [-] [size] [@addr]                       add string
| Ct[?] [-] [comment-text] [@addr]               add/remove type analysis comment
| Ct.[@addr]                                     show comment at current or specified address
| Cv[bsr][?]                                     add comments to args
| Cz[@addr]                                      add string (see Cs?)
```

Simply to add the comment to a particular line/address you can use `Ca` command:

```
[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". 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 syntax used by the `pf` command). Here's an 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
0x7fd9f13ae638    4989c4        mov r12, rax
```

The `[sz]` argument to `Cf` is used to define how many bytes the struct should take up in the disassembly, and is completely independent from the size of the data structure defined by the format string. This may seem confusing, but has several uses. For example, you may want to see the formatted structure displayed in the disassembly, but still have those locations be visible as offsets and with raw bytes. Sometimes, you find large structures, but only identified a few fields, or only interested in specific fields. Then, you can tell r2 to display only those fields, using the format string and using 'skip' fields, and also have the disassembly continue after the entire structure, by giving it full size using the `sz` argument.

Using `Cf`, it's easy to define complex structures with simple oneliners. See `pf?` for more information. Remember that all these `C` commands can also be accessed from the visual mode by pressing the `d` (data conversion) key. Note that unlike `t` commands `Cf` doesn't change analysis results. It is only a visual boon.

Sometimes just adding a single line of comments is not enough, in this case radare2 allows you to create a link for a particular text file. You can use it with `CC,` command or by pressing `,` key in the visual mode. This will open an `$EDITOR` to create a new file, or if filename does exist, just will create a link. It will be shown in the disassembly comments:

```
[0x00003af7 11% 290 /bin/ls]> pd $r @ main+55 # 0x3af7
 0x00003af7  call sym.imp.setlocale        ;[1] ; ,(locale-help.txt) ; char *setlocale(int category, const char *locale)
 0x00003afc  lea rsi, str.usr_share_locale ; 0x179cc ; "/usr/share/locale"
 0x00003b03  lea rdi, [0x000179b2]         ; "coreutils"
 0x00003b0a  call sym.imp.bindtextdomain   ;[2] ; char *bindtextdomain(char *domainname, char *dirname)
```

Note `,(locale-help.txt)` appeared in the comments, if we press `,` again in the visual mode, it will open the file. Using this mechanism we can create a long descriptions of some particular places in disassembly, link datasheets or related articles.

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. 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.

Using ESIL

r2's visual mode is great to inspect the ESIL evaluations.

There are 3 environment variables that are important for watching what a program does:

```
[0x00000000]> e emu.str = 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 `emu.str` variable is for (`asm.emustr` on <= 2.2). 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	Name	Operation	example
TRAP	src	Trap	Trap signal		

∗∗|src|Interrupt|interrupt|0x80, ()** | src | Syscall | syscall | rax,() **$$** | src | Instruction address | Get address of current instructionstack=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> "ae 5,5"0x0" **<=** | src,dst | Smaller or Equal (signed comparison) | stack = (dst <= src) ; update_eflags(dst - src) | [0x0000000]> "ae 1,5,<" 0x0> "ae 5,5"0x1" **>** | src,dst | Bigger (signed comparison) | stack = (dst > src) ; update_eflags(dst - src) | > "ae 1,5,>"0x1> "ae 5,5,>"0x0 **>=** | src,dst | Bigger or Equal (signed comparison) | stack = (dst >= src) ; update_eflags(dst - src) | > "ae 1,5,>="0x1> "ae 5,5,>="0x1 **«** | src,dst | Shift Left | stack = dst « src | > "ae 1,1,«"0x2> "ae 2,1,«"0x4 **»** | src,dst | Shift Right | stack = dst » src | > "ae 1,4,»"0x2> "ae 2,4,»"0x1 **«<** | src,dst | Rotate Left | stack=dst ROL src | > "ae 31,1,«<"0x80000000> "ae 32,1,«<"0x1 **»>** | src,dst | Rotate Right | stack=dst ROR src | > "ae 1,1,»>"0x80000000> "ae 32,1,»>"0x1 **&** | src,dst | AND | stack = dst & src | > "ae 1,1,&"0x1> "ae 1,0,&"0x0> "ae 0,1,&"0x0> "ae 0,0,&"0x0 **|** | src,dst | OR | stack = dst | src | > "ae 1,1,|"0x1> "ae 1,0,|"0x1> "ae 0,1,|"0x1> "ae 0,0,|"0x0 **^** | src,dst | XOR | stack = dst ^src | > "ae 1,1,^"0x0> "ae 1,0,^"0x1> "ae 0,1,^"0x1> "ae 0,0,^"0x0 **+** | src,dst | ADD | stack = dst + src | > "ae 3,4,+"0x7> "ae 5,5,+"0xa **-** | src,dst | SUB | stack = dst - src | > "ae 3,4,-"0x1> "ae 5,5,-"0x0> "ae 4,3,-"0xffffffffffffffff ***** | src,dst | MUL | stack = dst * src | > "ae 3,4,*"0xc> "ae 5,5,*"0x19 **/** | src,dst | DIV | stack = dst / src | > "ae 2,4,/"0x2> "ae 5,5,/"0x1> "ae 5,9,/"0x1

% | src,dst | MOD | stack = dst % src | > “ae 2,4,%”0x0> “ae 5,5,%”0x0> “ae 5,9,%”0x4 **~** | bits,src | SIGNEXT | stack = src sign extended | > “ae 8,0x80,~”0xffffffffffffff80 **~/** | src,dst | SIGNED DIV | stack = dst / src (signed) | > “ae 2,-4,~/”0xfffffffffffffffe **~%** | src,dst | SIGNED MOD | stack = dst % src (signed) | > “ae 2,-5,~%”0xffffffffffffffff **!** | src | NEG | stack = !!!src | > “ae 1,!”0x0> “ae 4,!”0x0> “ae 0,!”0x1 **++** | src | INC | stack = src++ | > ar r_00=0;ar r_000x00000000> “ae r_00,++”0x1> ar r_000x00000000> “ae 1,++”0x2 **–** | src | DEC | stack = src– | > ar r_00=5;ar r_000x00000005> “ae r_00,–”0x4> ar r_000x00000005> “ae 5,–”0x4 **=** | src,reg | EQU | reg = src | > “ae 3,r_00,=”> aer r_000x00000003> “ae r_00,r_01,=”> aer r_010x00000003 **:=** | src,reg | weak EQU | reg = src without side effects | > “ae 3,r_00,:=”> aer r_000x00000003> “ae r_00,r_01,:=”> aer r_010x00000003 **+=** | src,reg | ADD eq | reg = reg + src | > ar r_01=5;ar r_00=0;ar r_000x00000000> “ae r_01,r_00,+=”> ar r_000x00000005> “ae 5,r_00,+=”> ar r_000x0000000a **-=** | src,reg | SUB eq | reg = reg - src | > “ae r_01,r_00,-=”> ar r_000x00000004> “ae 3,r_00,-=”> ar r_000x00000001 ***=** | src,reg | MUL eq | reg = reg * src | > ar r_01=3;ar r_00=5;ar r_000x00000005> “ae r_01,r_00,*=”> ar r_000x0000000f> “ae 2,r_00,*=”> ar r_000x0000001e **/=** | src,reg | DIV eq | reg = reg / src | > ar r_01=3;ar r_00=6;ar r_000x00000006> “ae r_01,r_00,/=”> ar r_000x00000002> “ae 1,r_00,/=”> ar r_000x00000002 **%=** | src,reg | MOD eq | reg = reg % src | > ar r_01=3;ar r_00=7;ar r_00 0x00000007 > “ae r_01,r_00,%=” > ar r_00 0x00000001 > ar r_00=9;ar r_00 0x00000009 > “ae 5,r_00,%=” > ar r_00 0x00000004 **«=** | src,reg | Shift Left eq | reg = reg « src | > ar r_00=1;ar r_01=1;ar r_010x00000001> “ae r_00,r_01,«=”> ar r_010x00000002> “ae 2,r_01,«=”> ar r_010x00000008 **»=** | src,reg | Shift Right eq | reg = reg « src | > ar r_00=1;ar r_01=8;ar r_010x00000008> “ae r_00,r_01,»=”> ar r_010x00000004> “ae 2,r_01,»=”> ar r_010x00000001 **&=** | src,reg | AND eq | reg = reg & src | > ar r_00=2;ar r_01=6;ar r_010x00000006> “ae r_00,r_01,&=”> ar r_010x00000002> “ae 2,r_01,&=”> ar r_010x00000002> “ae 1,r_01,&=”> ar r_010x00000000 **|=** | src,reg | OR eq| reg = reg | src | > ar r_00=2;ar r_01=1;ar r_010x00000001> “ae r_00,r_01,|=”> ar r_010x00000003> “ae 4,r_01,|=”> ar r_010x00000007 **^=** | src,reg | XOR eq | reg = reg ^ src | > ar r_00=2;ar r_01=0xab;ar r_010x000000ab> “ae r_00,r_01,^=”> ar r_010x000000a9> “ae 2,r_01,^=”> ar r_010x000000ab **++=** | reg | INC eq | reg = reg + 1 | > ar r_00=4;ar r_000x00000004> “ae r_00,++=”> ar r_000x00000005 **–=** | reg | DEC eq | reg = reg - 1 | > ar r_00=4;ar r_000x00000004> “ae r_00,–=”> ar r_000x00000003 **!=** | reg | NOT eq | reg = !reg | > ar r_00=4;ar r_000x00000004> “ae r_00,!=”> ar r_000x00000000> “ae r_00,!=”> ar r_000x00000001 — | — | — | — | ——————————————- =[]=[*]=[1]=[2]=[4]=[8] | src,dst | poke |*dst=src | > “ae 0xdeadbeef,0x10000,=[4],”> pxw 4@0x100000x00010000 0xdeadbeef ….> “ae 0x0,0x10000,=[4],”> pxw 4@0x100000x00010000 0x00000000 [][*][1][2][4][8] | src | peek | stack=*src | > w test@0x10000> “ae 0x10000,[4],”0x74736574> ar r_00=0x10000> “ae r_00,[4],”0x74736574 |=[]|=[1]|=[2]|=[4]|=[8] | reg | nombre | code | > > SWAP | | Swap | Swap two top elements | SWAP 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 provides by default a set of helper operations for calculating flags. They fulfill their purpose by comparing the old and the new value of the dst operand of the last performed eq-operation. On every eq-operation (e.g. `=`) ESIL saves the old and new value of the dst operand. Note, that there also exist weak eq operations (e.g. `:=`), which do not affect flag operations. The `==` operation affects flag operations, despite not being an eq operation. Flag operations 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: 4,$b - checks if borrow from bit 4)
c       - carry, same like above (example: 7,$c - checks if carry from bit 7)
o       - overflow
p       - parity
r       - regsize ( asm.bits/8 )
s       - sign
ds      - delay slot state
jt      - jump target
js      - jump target set
```

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, the 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, interrupts are marked by ‘$'command.Forexample,'0x80$,’. It delegates emulation from the ESIL machine to a callback which implements interrupt handler for a specific OS/kernel/platform.

Traps are implemented with the `TRAP` command. They are used to throw exceptions for invalid instructions, division by zero, memory read error, or any other needed by specific architectures.

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, aka 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 are 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 Unit Support

At the moment of this writing, ESIL does not yet support FPU. But you can implement support for unsupported instructions using r2pipe. Eventually we will get proper support for multimedia and floating point.

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. They act 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
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,=
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 the data that we want. For example, we may want to get the target address of a jump. The parser for ESIL expressions should offer an 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 a 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 the parser, so we can extend it to implement analysis without having to change it again and again. That is, every time an operation is about to be executed, a user hook is called. It can be used for example to determine if `RIP` is going to change, or if the instruction updates the stack. Later, we can split that callback into several ones to have an event-based analysis API that may be extended in JavaScript like this:

```
esil.on('regset', function(){..
esil.on('syscall', function(){esil.regset('rip'
```

For the API, see the functions `hook_flag_read()`, `hook_execute()` and `hook_mem_read()`. A callback should return true or 1 if you want to override the action that it takes. 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 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,=
  ```

Data and Code Analysis

Radare2 has a very rich set of commands and configuration options to perform data and code analysis, to extract useful information from a binary, like pointers, string references, basic blocks, opcode data, jump targets, cross references and much more. These operations are handled by the `a` (analyze) command family:

```
|Usage: a[abdefFghoprxstc] [...]
| aa[?]              analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| a8 [hexpairs]      analyze bytes
| ab[b] [addr]       analyze block at given address
| abb [len]          analyze N basic blocks in [len] (section.size by default)
| abt [addr]         find paths in the bb function graph from current offset to given address
| 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[?]              analyze Functions
| aF                 same as above, but using anal.depth=1
| ag[?] [options]    draw graphs in various formats
| ah[?]              analysis hints (force opcode size, ...)
| ai [addr]          address information (show perms, stack, heap, ...)
| an [name] [@addr]  show/rename/create whatever flag/function is used at addr
| ao[?] [len]        analyze Opcodes (or emulate it)
| aO[?] [len]        Analyze N instructions in M bytes
| ap                 find prelude for current offset
| ar[?]              like 'dr' but for the esil vm. (registers)
| as[?] [num]        analyze syscall using dbg.reg
| av[?] [.]          show vtables
| ax[?]              manage refs/xrefs (see also afx?)
```

In fact, a namespace is one of the biggest in radare2 tool and allows to control very different parts of the analysis:

- Code flow analysis
- Data references analysis
- Using loaded symbols
- Managing different type of graphs, like CFG and call graph
- Manage variables
- Manage types
- Emulation using ESIL VM
- Opcode introspection
- Objects information, like virtual tables

Code Analysis

Code analysis is a common technique used to extract information from assembly code.

Radare2 has different code analysis techniques implemented in the core and available in different commands.

As long as the whole functionalities of r2 are available with the API as well as using commands. This gives you the ability to implement your own analysis loops using any programming language, even with r2 oneliners, shellscripts, or analysis or core native plugins.

The analysis will show up the internal data structures to identify basic blocks, function trees and to extract opcode-level information.

The most common radare2 analysis command sequence is `aa`, which stands for "analyze all". That all is referring to all symbols and entry-points. If your binary is stripped you will need to use other commands like `aaa`, `aab`, `aar`, `aac` or so.

Take some time to understand what each command does and the results after running them to find the best one for your needs.

```
[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_ ;  0x080487cf
| |   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.Youmustgiveapasswordforusethisprogram_ ;  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`). The `aa` command belongs to the family of auto analysis commands and performs only the most basic auto analysis steps. In radare2 there are many different types of the auto analysis commands with a different analysis depth, including partial emulation: `aa`, `aaa`, `aab`, `aaaa`, … There is also a mapping of those commands to the r2 CLI options: `r2 -A`, `r2 -AA`, and so on.

It is a common sense that completely automated analysis can produce non sequitur results, thus radare2 provides separate commands for the particular stages of the analysis allowing fine-grained control of the analysis process. Moreover, there is a treasure trove of configuration variables for controlling the analysis outcomes. You can find them in `anal.*` and `emu.*` cfg variables' namespaces.

Analyze functions

One of the most important "basic" analysis commands is the set of `af` subcommands. `af` means "analyze function". Using this command you can either allow automatic analysis of the particular function or perform completely manual one.

```
[0x00000000]> af?
Usage: af
| af ([name]) ([addr])                  analyze functions (start at addr or $$)
| afr ([name]) ([addr])                 analyze functions recursively
| af+ addr name [type] [diff]           hand craft a function (requires afb+)
| af- [addr]                            clean all function analysis data (or function at addr)
| afa                                   analyze function arguments in a call (afal honors dbg.funcarg)
| afb+ fcnA bbA sz [j] [f] ([t]( [d]))  add bb to function @ fcnaddr
| afb[?] [addr]                         List basic blocks of given function
| afbF([0|1])                           Toggle the basic-block 'folded' attribute
| afB 16                                set current function as thumb (change asm.bits)
| afC[lc] ([addr])@[addr]               calculate the Cycles (afC) or Cyclomatic Complexity (afCc)
| afc[?] type @[addr]                   set calling convention for function
| afd[addr]                             show function + delta for given offset
| afF[1|0|]                             fold/unfold/toggle
| afi [addr|fcn.name]                   show function(s) information (verbose afl)
| afj [tableaddr] [count]               analyze function jumptable
| afl[?] [ls*] [fcn name]               list functions (addr, size, bbs, name) (see afll)
| afm name                              merge two functions
| afM name                              print functions map
| afn[?] name [addr]                    rename name for function at address (change flag too)
| afna                                  suggest automatic name for current offset
| afo[?j] [fcn.name]                    show address for the function name or current offset
```

```
| afs[!] ([fcnsign])                     get/set function signature at current address (afs! uses cfg.editor)
| afS[stack_size]                        set stack frame size for function at current address
| afsr [function_name] [new_type]        change type for given function
| aft[?]                                 type matching, type propagation
| afu addr                               resize and analyze function from current address until addr
| afv[absrx]?                            manipulate args, registers and variables in function
| afx                                    list function references
```

You can use `afl` to list the functions found by the analysis.

There are a lot of useful commands under `afl` such as `aflj`, which lists the function in JSON format and `aflm`, which lists the functions in the syntax found in makefiles.

There's also `afl=`, which displays ASCII-art bars with function ranges.

You can find the rest of them under `afl?`.

Some of the most challenging tasks while performing a function analysis are merge, crop and resize. As with other analysis commands you have two modes: semi-automatic and manual. For the semi-automatic, you can use `afm <function name>` to merge the current function with the one specified by name as an argument, `aff` to readjust the function after analysis changes or function edits, `afu <address>` to do the resize and analysis of the current function until the specified address.

Apart from those semi-automatic ways to edit/analyze the function, you can hand craft it in the manual mode with `af+` command and edit basic blocks of it using `afb` commands. Before changing the basic blocks of the function it is recommended to check the already presented ones:

```
[0x00003ac0]> afb
0x00003ac0 0x00003b7f 01:001A 191 f 0x00003b7f
0x00003b7f 0x00003b84 00:0000 5 j 0x00003b92 f 0x00003b84
0x00003b84 0x00003b8d 00:0000 9 f 0x00003b8d
0x00003b8d 0x00003b92 00:0000 5
0x00003b92 0x00003ba8 01:0030 22 j 0x00003ba8
0x00003ba8 0x00003bf9 00:0000 81
```

Hand craft function

before start, let's prepare a binary file first, for example:

```c
int code_block()
{
  int result = 0;

  for(int i = 0; i < 10; ++i)
    result += 1;

  return result;
}
```

then compile it with `gcc -c example.c -m32 -O0 -fno-pie`, we will get the object file `example.o`. open it with radare2.

since we haven't analyzed it yet, the `pdf` command will not print out the disassembly here:

```
$ r2 example.o
[0x08000034]> pdf
p: Cannot find function at 0x08000034
[0x08000034]> pd
            ;-- section..text:
            ;-- .text:
            ;-- code_block:
            ;-- eip:
            0x08000034      55             push ebp                    ; [01] -r-x section size 41 named .text
            0x08000035      89e5           mov ebp, esp
            0x08000037      83ec10         sub esp, 0x10
            0x0800003a      c745f8000000.  mov dword [ebp - 8], 0
            0x08000041      c745fc000000.  mov dword [ebp - 4], 0
        ,=< 0x08000048      eb08           jmp 0x8000052
       .--> 0x0800004a      8345f801       add dword [ebp - 8], 1
       :|   0x0800004e      8345fc01       add dword [ebp - 4], 1
       :`-> 0x08000052      837dfc09       cmp dword [ebp - 4], 9
       `==< 0x08000056      7ef2           jle 0x800004a
            0x08000058      8b45f8         mov eax, dword [ebp - 8]
            0x0800005b      c9             leave
            0x0800005c      c3             ret
```

our goal is to hand craft a function with the following structure

create a function at 0x8000034 named code_block:

```
[0x8000034]> af+ 0x8000034 code_block
```

In most cases, we use jump or call instructions as code block boundaries. so the range of first block is from `0x08000034 push ebp` to `0x08000048 jmp 0x8000052`. use `afb+` command to add it.

```
[0x08000034]> afb+ code_block 0x8000034 0x800004a-0x8000034 0x8000052
```

note that the basic syntax of `afb+` is `afb+ function_address block_address block_size [jump] [fail]`. the final instruction of this block points to a new address(jmp 0x8000052), thus we add the address of jump target (0x8000052) to reflect the jump info.

the next block (0x08000052 ~ 0x08000056) is more likeyly an if conditional statement which has two branches. It will jump to 0x800004a if `jle-less or equal`, otherwise (the fail condition) jump to next instruction – 0x08000058.:

```
[0x08000034]> afb+ code_block 0x8000052 0x8000058-0x8000052 0x800004a 0x8000058
```

follow the control flow and create the remaining two blocks (two branches) :

```
[0x08000034]> afb+ code_block 0x800004a 0x8000052-0x800004a 0x8000052
[0x08000034]> afb+ code_block 0x8000058 0x800005d-0x8000058
```

check our work:

```
[0x08000034]> afb
0x08000034 0x0800004a 00:0000 22 j 0x08000052
0x0800004a 0x08000052 00:0000 8 j 0x08000052
0x08000052 0x08000058 00:0000 6 j 0x0800004a f 0x08000058
0x08000058 0x0800005d 00:0000 5
[0x08000034]> VV
```

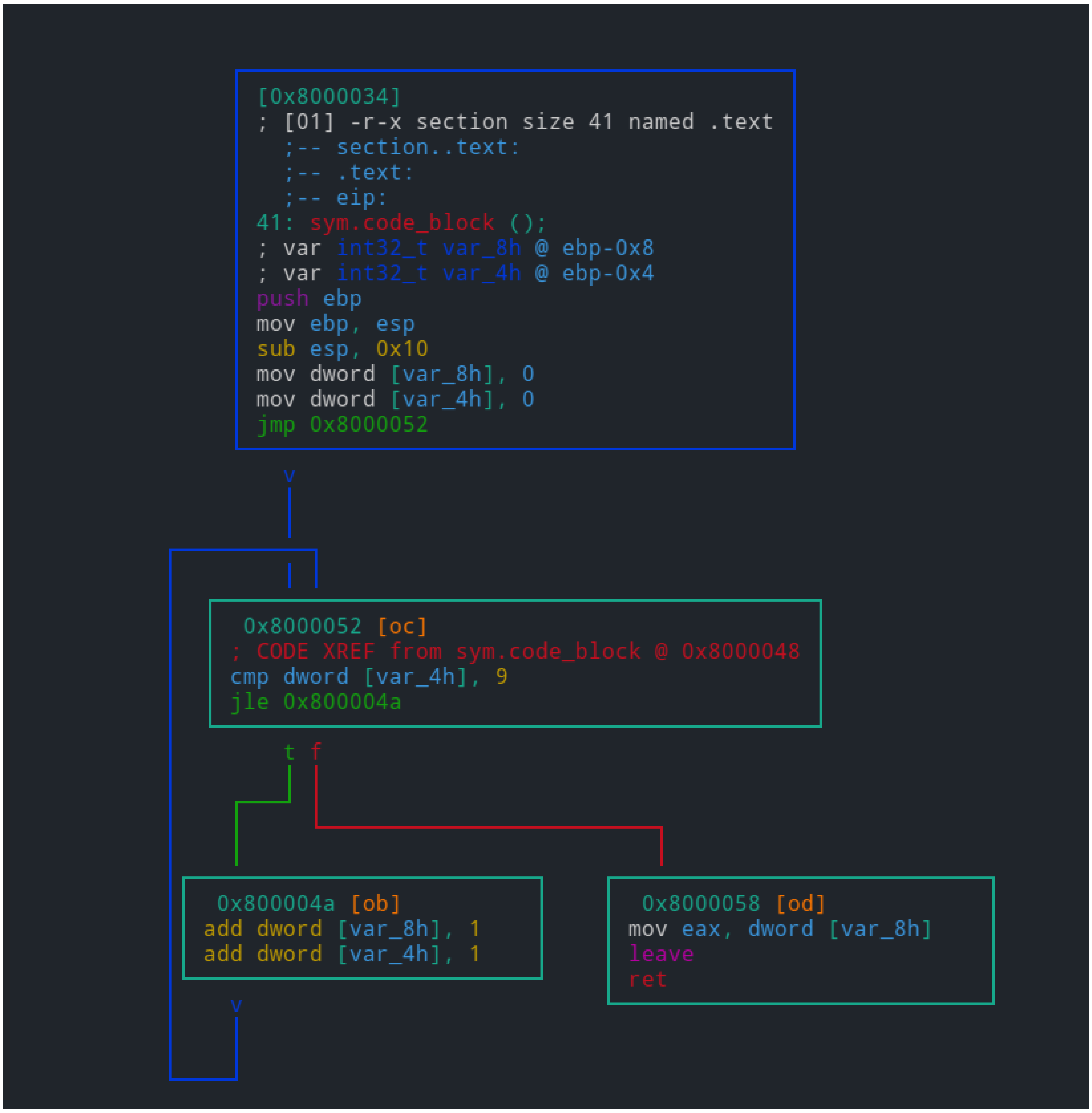

Figure 19: analyze_one

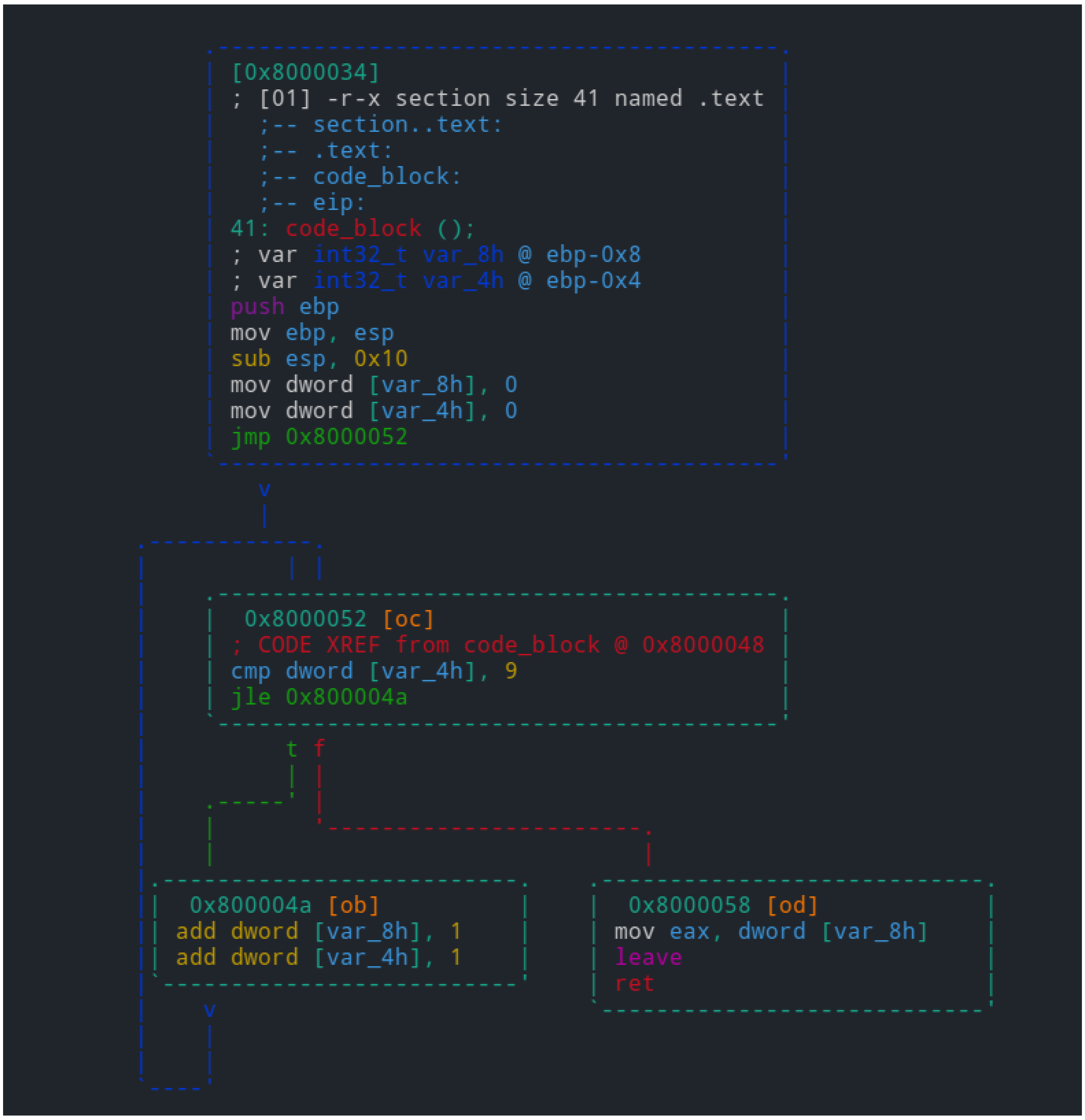

Figure 20: handcraft_one

There are two very important commands for this: `afc` and `afB`. The latter is a must-know command for some platforms like ARM. It provides a way to change the "bitness" of the particular function. Basically, allowing to select between ARM and Thumb modes.

`afc` on the other side, allows to manually specify function calling convention. You can find more information on its usage in calling_conventions.

Recursive analysis

There are 5 important program wide half-automated analysis commands:

- `aab` - perform basic-block analysis ("Nucleus" algorithm)
- `aac` - analyze function calls from one (selected or current function)
- `aaf` - analyze all function calls
- `aar` - analyze data references
- `aad` - analyze pointers to pointers references

Those are only generic semi-automated reference searching algorithms. Radare2 provides a wide choice of manual references' creation of any kind. For this fine-grained control you can use `ax` commands.

```
Usage: ax[?d-l*]   # see also 'afx?'
| ax              list refs
| ax*             output radare commands
| ax addr [at]    add code ref pointing to addr (from curseek)
| ax- [at]        clean all refs/refs from addr
| ax-*            clean all refs/refs
| axc addr [at]   add generic code ref
| axC addr [at]   add code call ref
| axg [addr]      show xrefs graph to reach current function
| axg* [addr]     show xrefs graph to given address, use .axg*;aggv
| axgj [addr]     show xrefs graph to reach current function in json format
| axd addr [at]   add data ref
| axq             list refs in quiet/human-readable format
| axj             list refs in json format
| axF [flg-glob]  find data/code references of flags
| axm addr [at]   copy data/code references pointing to addr to also point to curseek (or at)
| axt [addr]      find data/code references to this address
| axf [addr]      find data/code references from this address
| axv [addr]      list local variables read-write-exec references
| ax. [addr]      find data/code references from and to this address
| axff[j] [addr]  find data/code references from this function
| axs addr [at]   add string ref
```

The most commonly used `ax` commands are `axt` and `axf`, especially as a part of various r2pipe scripts. Lets say we see the string in the data or a code section and want to find all places it was referenced from, we should use `axt`:

```
[0x0001783a]> pd 2
;-- str.02x:
; STRING XREF from 0x00005de0 (sub.strlen_d50)
; CODE XREF from 0x00017838 (str.._s_s_s + 7)
0x0001783a     .string "%%%02x" ; len=7
;-- str.src_ls.c:
; STRING XREF from 0x0000541b (sub.free_b04)
; STRING XREF from 0x0000543a (sub.__assert_fail_41f + 27)
; STRING XREF from 0x00005459 (sub.__assert_fail_41f + 58)
; STRING XREF from 0x00005f9e (sub._setjmp_e30)
; CODE XREF from 0x0001783f (str.02x + 5)
0x00017841 .string "src/ls.c" ; len=9
[0x0001783a]> axt
sub.strlen_d50 0x5de0 [STRING] lea rcx, str.02x
(nofunc) 0x17838 [CODE] jae str.02x
```

There are also some useful commands under `axt`. Use `axtg` to generate radare2 commands which will help you to create graphs according to the XREFs.

```
[0x08048320]> s main
[0x080483e0]> axtg
agn 0x8048337 "entry0 + 23"
agn 0x80483e0 "main"
age 0x8048337 0x80483e0
```

Use `axt*` to split the radare2 commands and set flags on those corresponding XREFs.

Also under `ax` is `axg`, which finds the path between two points in the file by showing an XREFs graph to reach the location or function. For example:

```
:> axg sym.imp.printf
- 0x08048a5c fcn 0x08048a5c sym.imp.printf
  - 0x080483e5 fcn 0x080483e0 main
  - 0x080483e0 fcn 0x080483e0 main
    - 0x08048337 fcn 0x08048320 entry0
  - 0x08048425 fcn 0x080483e0 main
```

Use `axg*` to generate radare2 commands which will help you to create graphs using `agn` and `age` commands, according to the XREFs.

Apart from predefined algorithms to identify functions there is a way to specify a function prelude with a configuration option `anal.prelude`. For example, like `e anal.prelude = 0x554889e5` which means

```
push rbp
mov rbp, rsp
```

on x86_64 platform. It should be specified *before* any analysis commands.

Configuration

Radare2 allows to change the behavior of almost any analysis stages or commands. There are different kinds of the configuration options:

- Flow control
- Basic blocks control
- References control
- IO/Ranges
- Jump tables analysis control
- Platform/target specific options

Control flow configuration

Two most commonly used options for changing the behavior of control flow analysis in radare2 are `anal.hasnext` and `anal.jmp.after`. The first one allows forcing radare2 to continue the analysis after the end of the function, even if the next chunk of the code wasn't called anywhere, thus analyzing all of the available functions. The latter one allows forcing radare2 to continue the analysis even after unconditional jumps.

In addition to those we can also set `anal.jmp.indir` to follow the indirect jumps, continuing analysis; `anal.pushret` to analyze `push ...; ret` sequence as a jump; `anal.nopskip` to skip the NOP sequences at a function beginning.

For now, radare2 also allows you to change the maximum basic block size with `anal.bb.maxsize` option . The default value just works in most use cases, but it's useful to increase that for example when dealing with obfuscated code. Beware that some of basic blocks control options may disappear in the future in favor of more automated ways to set those.

For some unusual binaries or targets, there is an option `anal.noncode`. Radare2 doesn't try to analyze data sections as a code by default. But in some cases - malware, packed binaries, binaries for embedded systems, it is often a case. Thus - this option.

Reference control

The most crucial options that change the analysis results drastically. Sometimes some can be disabled to save the time and memory when analyzing big binaries.

- `anal.jmp.ref` - to allow references creation for unconditional jumps
- `anal.jmp.cref` - same, but for conditional jumps
- `anal.datarefs` - to follow the data references in code
- `anal.refstr` - search for strings in data references
- `anal.strings` - search for strings and creating references

Note that strings references control is disabled by default because it increases the analysis time.

Analysis ranges

There are a few options for this:

- `anal.limits` - enables the range limits for analysis operations
- `anal.from` - starting address of the limit range
- `anal.to` - the corresponding end of the limit range
- `anal.in` - specify search boundaries for analysis. You can set it to `io.maps`, `io.sections.exec`, `dbg.maps` and many more. For example:
  - To analyze a specific memory map with `anal.from` and `anal.to`, set `anal.in = dbg.maps`.
  - To analyze in the boundaries set by `anal.from` and `anal.to`, set `anal.in=range`.
  - To analyze in the current mapped segment or section, you can put `anal.in=bin.segment` or `anal.in=bin.section`, respectively.
  - To analyze in the current memory map, specify `anal.in=dbg.map`.
  - To analyze in the stack or heap, you can set `anal.in=dbg.stack` or `anal.in=dbg.heap`.
  - To analyze in the current function or basic block, you can specify `anal.in=anal.fcn` or `anal.in=anal.bb`.

Please see `e anal.in=??` for the complete list.

Jump tables

Jump tables are one of the trickiest targets in binary reverse engineering. There are hundreds of different types, the end result depending on the compiler/linker and LTO stages of optimization. Thus radare2 allows enabling some experimental jump tables detection algorithms using `anal.jmp.tbl` option. Eventually, algorithms moved into the default analysis loops once they start to work on every supported platform/target/testcase. Two more options can affect the jump tables analysis results too:

- `anal.jmp.indir` - follow the indirect jumps, some jump tables rely on them
- `anal.datarefs` - follow the data references, some jump tables use those

Platform specific controls

There are two common problems when analyzing embedded targets: ARM/Thumb detection and MIPS GP value. In case of ARM binaries radare2 supports some auto-detection of ARM/Thumb mode switches, but beware that it uses partial ESIL emulation, thus slowing the analysis process. If you will not like the results, particular functions' mode can be overridden with `afB` command.

The MIPS GP problem is even trickier. It is a basic knowledge that GP value can be different not only for the whole program, but also for some functions. To partially solve that there are options `anal.gp` and `anal.gpfixed`. The first one sets the GP value for the whole program or particular function. The latter allows to "constantify" the GP value if some code is willing to change its value, always resetting it if the case. Those are heavily experimental and might be changed in the future in favor of more automated analysis.

Visuals

One of the easiest way to see and check the changes of the analysis commands and variables is to perform a scrolling in a `Vv` special visual mode, allowing functions preview:

```
-[ functions ]----------------
(a) add     (x)xrefs     (q)quit
(r) rename  (c)calls     (g)go
(d) delete  (v)variables (?)help
>* 0x00005480   43 entry0
0x000150f0   46 sym._obstack_allocated_p
0x00014fe0  179 sym._obstack_begin_1
0x00014fc0  158 sym._obstack_begin
0x00015130   97 sym._obstack_free
0x00015000  283 sym._obstack_newchunk
0x000151a0   36 sym._obstack_memory_used
0x000033f0    6 sym.imp.__ctype_toupper_loc
0x00003400    6 sym.imp.__uflow
0x00003410    6 sym.imp.getenv
0x00003420    6 sym.imp.sigprocmask
0x00003430    6 sym.imp.__snprintf_chk
0x00003440    6 sym.imp.raise
0x00000000   48 sym.imp.free
0x00003450    6 sym.imp.abort
0x00003460    6 sym.imp.__errno_location
0x00003470    6 sym.imp.strncmp
0x00003480    6 sym.imp.localtime_r
0x00003490    6 sym.imp._exit
0x000034a0    6 sym.imp.strcpy
0x000034b0    6 sym.imp.__fpending
0x000034c0    6 sym.imp.isatty
```

```
...
Visual code review (pdf)
╭ (fcn) entry0 43
│   entry0 ();
│           0x00005480      xor ebp, ebp
│           0x00005482      mov r9, rdx
│           0x00005485      pop rsi
│           0x00005486      mov rdx, rsp
│           0x00005489      and rsp, 0xfffffffffffffff0
│           0x0000548d     push rax
│           0x0000548e     push rsp
│           0x0000548f      lea r8, [0x00015e40]
│           0x00005496      lea rcx, [0x00015dd0]
│               0x0000549d      lea rdi, [main]  ; section..text ; 0x3ac0 ; "AWAVAUATUS\x89\xfdH\x89\xf3H\x83
│           0x000054a4      call qword [reloc.__libc_start_main] ; [0x21efc8:8]=0
╰           0x000054aa      hlt
```

Figure 21: vv

When we want to check how analysis changes affect the result in the case of big functions, we can use minimap instead, allowing to see a bigger flow graph on the same screen size. To get into the minimap mode type `VV` then press `p` twice:

This mode allows you to see the disassembly of each node separately, just navigate between them using `Tab` key.

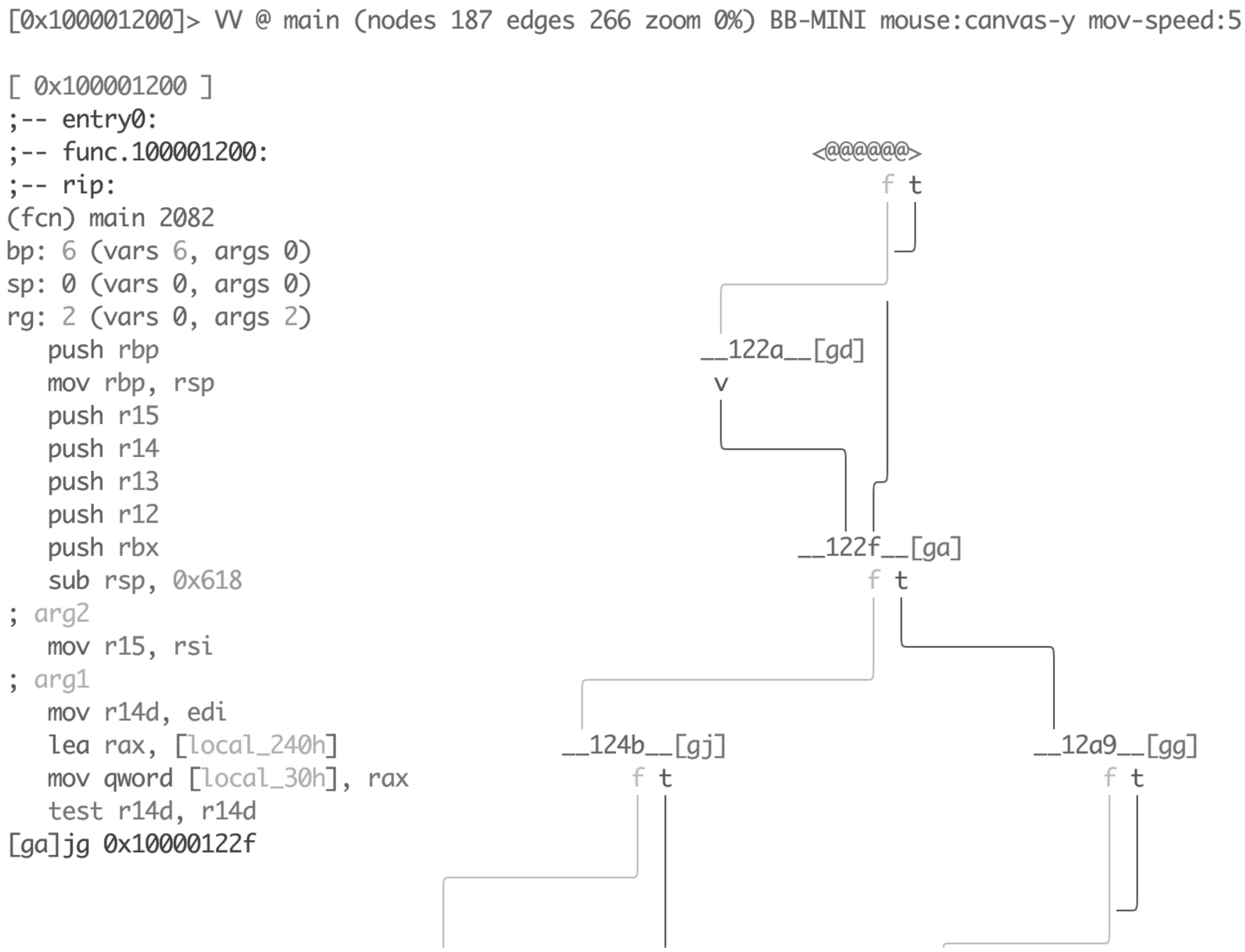

Figure 22: vv2

Analysis hints

It is not an uncommon case that analysis results are not perfect even after you tried every single configuration option. This is where the "analysis hints" radare2 mechanism comes in. It allows to override some basic opcode or meta-information properties, or even to rewrite the whole opcode string. These commands are located under `ah` namespace:

```
Usage: ah[lba-]  Analysis Hints
| ah?                show this help
| ah? offset         show hint of given offset
| ah                 list hints in human-readable format
| ah.                list hints in human-readable format from current offset
| ah-                remove all hints
| ah- offset [size]  remove hints at given offset
| ah* offset         list hints in radare commands format
| aha ppc @ 0x42     force arch ppc for all addrs >= 0x42 or until the next hint
| aha 0 @ 0x84       disable the effect of arch hints for all addrs >= 0x84 or until the next hint
| ahb 16 @ 0x42      force 16bit for all addrs >= 0x42 or until the next hint
| ahb 0 @ 0x84       disable the effect of bits hints for all addrs >= 0x84 or until the next hint
| ahc 0x804804       override call/jump address
| ahd foo a0,33      replace opcode string
| ahe 3,eax,+=       set vm analysis string
| ahf 0x804840       override fallback address for call
| ahF 0x10           set stackframe size at current offset
| ahh 0x804840       highlight this address offset in disasm
| ahi[?] 10          define numeric base for immediates (2, 8, 10, 10u, 16, i, p, S, s)
| ahj                list hints in JSON
| aho call           change opcode type (see aho?) (deprecated, moved to "ahd")
| ahp addr           set pointer hint
| ahr val            set hint for return value of a function
| ahs 4              set opcode size=4
| ahS jz             set asm.syntax=jz for this opcode
| aht [?] <type>     Mark immediate as a type offset (deprecated, moved to "aho")
| ahv val            change opcode's val field (useful to set jmptbl sizes in jmp rax)
```

One of the most common cases is to set a particular numeric base for immediates:

```
[0x00003d54]> ahi?
Usage: ahi [2|8|10|10u|16|bodhipSs] [@ offset]   Define numeric base
| ahi <base>  set numeric base (2, 8, 10, 16)
| ahi 10|d    set base to signed decimal (10), sign bit should depend on receiver size
| ahi 10u|du  set base to unsigned decimal (11)
| ahi b       set base to binary (2)
| ahi o       set base to octal (8)
| ahi h       set base to hexadecimal (16)
| ahi i       set base to IP address (32)
| ahi p       set base to htons(port) (3)
| ahi S       set base to syscall (80)
| ahi s       set base to string (1)

[0x00003d54]> pd 2
0x00003d54      0583000000     add eax, 0x83
0x00003d59      3d13010000     cmp eax, 0x113
[0x00003d54]> ahi d
[0x00003d54]> pd 2
0x00003d54      0583000000     add eax, 131
0x00003d59      3d13010000     cmp eax, 0x113
[0x00003d54]> ahi b
[0x00003d54]> pd 2
0x00003d54      0583000000     add eax, 10000011b
0x00003d59      3d13010000     cmp eax, 0x113
```

It is notable that some analysis stages or commands add the internal analysis hints, which can be checked with `ah` command:

```
[0x00003d54]> ah
 0x00003d54 - 0x00003d54 => immbase=2
[0x00003d54]> ah*
 ahi 2 @ 0x3d54
```

Sometimes we need to override jump or call address, for example in case of tricky relocation, which is unknown for radare2, thus we can change the value manually. The current analysis information about a particular opcode can be checked with `ao` command. We can use `ahc` command for performing such a change:

```
[0x00003cee]> pd 2
0x00003cee      e83d080100     call sub.__errno_location_530
0x00003cf3      85c0           test eax, eax
[0x00003cee]> ao
address: 0x3cee
opcode: call 0x14530
mnemonic: call
prefix: 0
id: 56
bytes: e83d080100
refptr: 0
size: 5
sign: false
type: call
cycles: 3
esil: 83248,rip,8,rsp,-=,rsp,=[],rip,=
jump: 0x00014530
direction: exec
fail: 0x00003cf3
stack: null
family: cpu
stackop: null
[0x00003cee]> ahc 0x5382
[0x00003cee]> pd 2
0x00003cee      e83d080100     call sub.__errno_location_530
0x00003cf3      85c0           test eax, eax
```

```
[0x00003cee]> ao
address: 0x3cee
opcode: call 0x14530
mnemonic: call
prefix: 0
id: 56
bytes: e83d080100
refptr: 0
size: 5
sign: false
type: call
cycles: 3
esil: 83248,rip,8,rsp,-=,rsp,=[],rip,=
jump: 0x00005382
direction: exec
fail: 0x00003cf3
stack: null
family: cpu
stackop: null
[0x00003cee]> ah
 0x00003cee - 0x00003cee => jump: 0x5382
```

As you can see, despite the unchanged disassembly view the jump address in opcode was changed (`jump` option).

If anything of the previously described didn't help, you can simply override shown disassembly with anything you like:

```
[0x00003d54]> pd 2
0x00003d54      0583000000     add eax, 10000011b
0x00003d59      3d13010000     cmp eax, 0x113
[0x00003d54]> "ahd myopcode bla, foo"
[0x00003d54]> pd 2
0x00003d54                     myopcode bla, foo
0x00003d55      830000         add dword [rax], 0
```

Managing variables

Radare2 allows managing local variables, no matter their location, stack or registers. The variables' auto analysis is enabled by default but can be disabled with `anal.vars` configuration option.

The main variables commands are located in `afv` namespace:

```
Usage: afv  [rbs]
| afv*                          output r2 command to add args/locals to flagspace
| afv-([name])                  remove all or given var
| afv=                          list function variables and arguments with disasm refs
| afva                          analyze function arguments/locals
| afvb[?]                       manipulate bp based arguments/locals
| afvd name                     output r2 command for displaying the value of args/locals in the debugger
| afvf                          show BP relative stackframe variables
| afvn [new_name] ([old_name])  rename argument/local
| afvr[?]                       manipulate register based arguments
| afvR [varname]                list addresses where vars are accessed (READ)
| afvs[?]                       manipulate sp based arguments/locals
| afvt [name] [new_type]        change type for given argument/local
| afvW [varname]                list addresses where vars are accessed (WRITE)
| afvx                          show function variable xrefs (same as afvR+afvW)
```

`afvr`, `afvb` and `afvs` commands are uniform but allow manipulation of register-based arguments and variables, BP/FP-based arguments and variables, and SP-based arguments and variables respectively. If we check the help for `afvr` we will get the way two others commands works too:

```
|Usage: afvr [reg] [type] [name]
| afvr                        list register based arguments
| afvr*                       same as afvr but in r2 commands
| afvr [reg] [name] ([type])  define register arguments
| afvrj                       return list of register arguments in JSON format
| afvr- [name]                delete register arguments at the given index
| afvrg [reg] [addr]          define argument get reference
| afvrs [reg] [addr]          define argument set reference
```

Like many other things variables detection is performed by radare2 automatically, but results can be changed with those arguments/variables control commands. This kind of analysis relies heavily on preloaded function prototypes and the calling-convention, thus loading symbols can improve it. Moreover, after changing something we can rerun variables analysis with `afva` command. Quite often variables analysis is accompanied with types analysis, see `afta` command.

The most important aspect of reverse engineering - naming things. Of course, you can rename variable too, affecting all places it was referenced. This can be achieved with `afvn` for *any* type of argument or variable. Or you can simply remove the variable or argument with `afv-` command.

As mentioned before the analysis loop relies heavily on types information while performing variables analysis stages. Thus comes next very important command - `afvt`, which allows you to change the type of variable:

```
[0x00003b92]> afvs
var int local_8h @ rsp+0x8
var int local_10h @ rsp+0x10
var int local_28h @ rsp+0x28
var int local_30h @ rsp+0x30
var int local_32h @ rsp+0x32
var int local_38h @ rsp+0x38
var int local_45h @ rsp+0x45
var int local_46h @ rsp+0x46
var int local_47h @ rsp+0x47
var int local_48h @ rsp+0x48
[0x00003b92]> afvt local_10h char*
[0x00003b92]> afvs
var int local_8h @ rsp+0x8
var char* local_10h @ rsp+0x10
var int local_28h @ rsp+0x28
var int local_30h @ rsp+0x30
var int local_32h @ rsp+0x32
```

```
var int local_38h @ rsp+0x38
var int local_45h @ rsp+0x45
var int local_46h @ rsp+0x46
var int local_47h @ rsp+0x47
var int local_48h @ rsp+0x48
```

Less commonly used feature, which is still under heavy development - distinction between variables being read and written. You can list those being read with `afvR` command and those being written with `afvW` command. Both commands provide a list of the places those operations are performed:

```
[0x00003b92]> afvR
local_48h  0x48ee
local_30h  0x3c93,0x520b,0x52ea,0x532c,0x5400,0x3cfb
local_10h  0x4b53,0x5225,0x53bd,0x50cc
local_8h  0x4d40,0x4d99,0x5221,0x53b9,0x50c8,0x4620
local_28h  0x503a,0x51d8,0x51fa,0x52d3,0x531b
local_38h
local_45h  0x50a1
local_47h
local_46h
local_32h  0x3cb1
[0x00003b92]> afvW
local_48h  0x3adf
local_30h  0x3d3e,0x4868,0x5030
local_10h  0x3d0e,0x5035
local_8h  0x3d13,0x4d39,0x5025
local_28h  0x4d00,0x52dc,0x53af,0x5060,0x507a,0x508b
local_38h  0x486d
local_45h  0x5014,0x5068
local_47h  0x501b
local_46h  0x5083
local_32h
[0x00003b92]>
```

Type inference

The type inference for local variables and arguments is well integrated with the command `afta`.

Let's see an example of this with a simple hello_world binary

```
[0x000007aa]> pdf
|           ;-- main:
/ (fcn) sym.main 157
| sym.main ();
| ; var int local_20h @ rbp-0x20
| ; var int local_1ch @ rbp-0x1c
| ; var int local_18h @ rbp-0x18
| ; var int local_10h @ rbp-0x10
| ; var int local_8h @ rbp-0x8
| ; DATA XREF from entry0 (0x6bd)
| 0x000007aa  push rbp
| 0x000007ab  mov rbp, rsp
| 0x000007ae  sub rsp, 0x20
| 0x000007b2  lea rax, str.Hello          ; 0x8d4 ; "Hello"
| 0x000007b9  mov qword [local_18h], rax
| 0x000007bd  lea rax, str.r2_folks       ; 0x8da ; " r2-folks"
| 0x000007c4  mov qword [local_10h], rax
| 0x000007c8  mov rax, qword [local_18h]
| 0x000007cc  mov rdi, rax
| 0x000007cf  call sym.imp.strlen         ; size_t strlen(const char *s)
```

- After applying `afta`

```
[0x000007aa]> afta
[0x000007aa]> pdf
| ;-- main:
| ;-- rip:
/ (fcn) sym.main 157
| sym.main ();
| ; var size_t local_20h @ rbp-0x20
| ; var size_t size @ rbp-0x1c
| ; var char *src @ rbp-0x18
| ; var char *s2 @ rbp-0x10
| ; var char *dest @ rbp-0x8
| ; DATA XREF from entry0 (0x6bd)
| 0x000007aa  push rbp
| 0x000007ab  mov rbp, rsp
| 0x000007ae  sub rsp, 0x20
| 0x000007b2  lea rax, str.Hello          ; 0x8d4 ; "Hello"
| 0x000007b9  mov qword [src], rax
| 0x000007bd  lea rax, str.r2_folks       ; 0x8da ; " r2-folks"
| 0x000007c4  mov qword [s2], rax
| 0x000007c8  mov rax, qword [src]
| 0x000007cc  mov rdi, rax                ; const char *s
| 0x000007cf  call sym.imp.strlen         ; size_t strlen(const char *s)
```

It also extracts type information from format strings like `printf ("fmt : %s , %u , %d", ...)`, the format specifications are extracted from `anal/d/spec.sdb`

You could create a new profile for specifying a set of format chars depending on different libraries/operating systems/programming languages like this :

```
win=spec
spec.win.u32=unsigned int
```

Then change your default specification to newly created one using this config variable `e anal.spec = win`

For more information about primitive and user-defined types support in radare2 refer to types chapter.

Types

Radare2 supports the C-syntax data types description. Those types are parsed by a C11-compatible parser and stored in the internal SDB, thus are introspectable with `k` command.

Most of the related commands are located in `t` namespace:

```
[0x00000000]> t?
| Usage: t   # cparse types commands
| t                          List all loaded types
| tj                         List all loaded types as json
| t <type>                   Show type in 'pf' syntax
| t*                         List types info in r2 commands
| t- <name>                  Delete types by its name
| t-*                        Remove all types
| tail [filename]            Output the last part of files
| tc [type.name]             List all/given types in C output format
| te[?]                      List all loaded enums
| td[?] <string>             Load types from string
| tf                         List all loaded functions signatures
| tk <sdb-query>             Perform sdb query
| tl[?]                      Show/Link type to an address
| tn[?] [-][addr]            manage noreturn function attributes and marks
| to -                       Open cfg.editor to load types
| to <path>                  Load types from C header file
| toe [type.name]            Open cfg.editor to edit types
| tos <path>                 Load types from parsed Sdb database
| tp  <type> [addr|varname]  cast data at <address> to <type> and print it (XXX: type can contain spaces)
| tpv <type> @ [value]       Show offset formatted for given type
| tpx <type> <hexpairs>      Show value for type with specified byte sequence (XXX: type can contain spaces)
| ts[?]                      Print loaded struct types
| tu[?]                      Print loaded union types
| tx[f?]                     Type xrefs
| tt[?]                      List all loaded typedefs
```

Note that the basic (atomic) types are not those from C standard - not `char`, `_Bool`, or `short`. Because those types can be different from one platform to another, radare2 uses `definite` types like as `int8_t` or `uint64_t` and will convert `int` to `int32_t` or `int64_t` depending on the binary or debuggee platform/compiler.

Basic types can be listed using `t` command. For the structured types you need to use `ts`, for unions use `tu` and for enums — `te`.

```
[0x00000000]> t
char
char *
double
float
gid_t
int
int16_t
int32_t
int64_t
int8_t
long
long long
pid_t
short
size_t
uid_t
uint16_t
uint32_t
uint64_t
uint8_t
unsigned char
unsigned int
unsigned short
void *
```

Loading types

There are three easy ways to define a new type: * Directly from the string using `td` command * From the file using `to <filename>` command * Open an `$EDITOR` to type the definitions in place using `to -`

```
[0x00000000]> "td struct foo {char* a; int b;}"
[0x00000000]> cat ~/radare2-regressions/bins/headers/s3.h
struct S1 {
    int x[3];
    int y[4];
    int z;
};
[0x00000000]> to ~/radare2-regressions/bins/headers/s3.h
[0x00000000]> ts
foo
S1
```

Also note there is a config option to specify include directories for types parsing

```
[0x00000000]> e? dir.types
dir.types: Default path to look for cparse type files
[0x00000000]> e dir.types
/usr/include
```

Printing types

Notice below we have used `ts` command, which basically converts the C type description (or to be precise it's SDB representation) into the sequence of `pf` commands. See more about print format.

The `tp` command uses the `pf` string to print all the members of type at the current offset/given address:

```
[0x00000000]> "td struct foo {char* a; int b;}"
[0x00000000]> wx 68656c6c6f000c000000
[0x00000000]> wz world @ 0x00000010 ; wx 17 @ 0x00000016
[0x00000000]> px
[0x00000000]> ts foo
pf zd a b
[0x00000000]> tp foo
 a : 0x00000000 = "hello"
 b : 0x00000006 = 12
[0x00000000]> tp foo @ 0x00000010
 a : 0x00000010 = "world"
 b : 0x00000016 = 23
```

Also, you could fill your own data into the struct and print it using `tpx` command

```
[0x00000000]> tpx foo 414243440010000000
 a : 0x00000000 = "ABCD"
 b : 0x00000005 = 16
```

Linking Types

The `tp` command just performs a temporary cast. But if we want to link some address or variable with the chosen type, we can use `tl` command to store the relationship in SDB.

```
[0x000051c0]> tl S1 = 0x51cf
[0x000051c0]> tll
(S1)
 x : 0x000051cf = [ 2315619660, 1207959810, 34803085 ]
 y : 0x000051db = [ 2370306049, 4293315645, 3860201471, 4093649307 ]
 z : 0x000051eb = 4464399
```

Moreover, the link will be shown in the disassembly output or visual mode:

```
[0x000051c0 15% 300 /bin/ls]> pd $r @ entry0
 ;-- entry0:
 0x000051c0      xor ebp, ebp
 0x000051c2      mov r9, rdx
 0x000051c5      pop rsi
 0x000051c6      mov rdx, rsp
 0x000051c9      and rsp, 0xfffffffffffffff0
 0x000051cd      push rax
 0x000051ce      push rsp
(S1)
 x : 0x000051cf = [ 2315619660, 1207959810, 34803085 ]
 y : 0x000051db = [ 2370306049, 4293315645, 3860201471, 4093649307 ]
 z : 0x000051eb = 4464399
 0x000051f0      lea rdi, loc._edata         ; 0x21f248
 0x000051f7      push rbp
 0x000051f8      lea rax, loc._edata         ; 0x21f248
 0x000051ff      cmp rax, rdi
 0x00005202      mov rbp, rsp
```

Once the struct is linked, radare2 tries to propagate structure offset in the function at current offset, to run this analysis on whole program or at any targeted functions after all structs are linked you have `aat` command:

```
[0x00000000]> aa?
| aat [fcn]           Analyze all/given function to convert immediate to linked structure offsets (see tl?)
```

Note sometimes the emulation may not be accurate, for example as below :

```
|0x000006da  push rbp
|0x000006db  mov rbp, rsp
|0x000006de  sub rsp, 0x10
|0x000006e2  mov edi, 0x20               ; "@"
|0x000006e7  call sym.imp.malloc         ;  void *malloc(size_t size)
|0x000006ec  mov qword [local_8h], rax
|0x000006f0  mov rax, qword [local_8h]
```

The return value of `malloc` may differ between two emulations, so you have to set the hint for return value manually using `ahr` command, so run `tl` or `aat` command after setting up the return value hint.

```
[0x000006da]> ah?
| ahr val            set hint for return value of a function
```

Structure Immediates

There is one more important aspect of using types in radare2 - using `aht` you can change the immediate in the opcode to the structure offset. Lets see a simple example of [R]SI-relative addressing

```
[0x000052f0]> pd 1
0x000052f0      mov rax, qword [rsi + 8]    ; [0x8:8]=0
```

Here `8` - is some offset in the memory, where `rsi` probably holds some structure pointer. Imagine that we have the following structures

```
[0x000052f0]> "td struct ms { char b[8]; int member1; int member2; };"
[0x000052f0]> "td struct ms1 { uint64_t a; int member1; };"
[0x000052f0]> "td struct ms2 { uint16_t a; int64_t b; int member1; };"
```

Now we need to set the proper structure member offset instead of `8` in this instruction. At first, we need to list available types matching this offset:

```
[0x000052f0]> ahts 8
ms.member1
ms1.member1
```

Note, that `ms2` is not listed, because it has no members with offset `8`. After listing available options we can link it to the chosen offset at the current address:

```
[0x000052f0]> aht ms1.member1
[0x000052f0]> pd 1
0x000052f0      488b4608       mov rax, qword [rsi + ms1.member1]    ; [0x8:8]=0
```

Managing enums

- Printing all fields in enum using `te` command

```
[0x00000000]> "td enum Foo {COW=1,BAR=2};"
[0x00000000]> te Foo
COW = 0x1
BAR = 0x2
```

- Finding matching enum member for given bitfield and vice-versa

```
[0x00000000]> te Foo 0x1
COW
[0x00000000]> teb Foo COW
0x1
```

Internal representation

To see the internal representation of the types you can use `tk` command:

```
[0x000051c0]> tk~S1
S1=struct
struct.S1=x,y,z
struct.S1.x=int32_t,0,3
struct.S1.x.meta=4
struct.S1.y=int32_t,12,4
struct.S1.y.meta=4
struct.S1.z=int32_t,28,0
struct.S1.z.meta=0
[0x000051c0]>
```

Defining primitive types requires an understanding of basic `pf` formats, you can find the whole list of format specifier in `pf??`:

```
-----------------------------------------------------
| format | explanation                               |
|---------------------------------------------------|
|  b     |  byte (unsigned)                          |
|  c     |  char (signed byte)                       |
|  d     |  0x%%08x hexadecimal value (4 bytes)      |
|  f     |  float value (4 bytes)                    |
|  i     |  %%i integer value (4 bytes)              |
|  o     |  0x%%08o octal value (4 byte)             |
|  p     |  pointer reference (2, 4 or 8 bytes)      |
|  q     |  quadword (8 bytes)                       |
|  s     |  32bit pointer to string (4 bytes)        |
|  S     |  64bit pointer to string (8 bytes)        |
|  t     |  UNIX timestamp (4 bytes)                 |
|  T     |  show Ten first bytes of buffer           |
|  u     |  uleb128 (variable length)                |
|  w     |  word (2 bytes unsigned short in hex)     |
|  x     |  0x%%08x hex value and flag (fd @ addr)   |
|  X     |  show formatted hexpairs                  |
|  z     |  \0 terminated string                     |
|  Z     |  \0 terminated wide string                |
-----------------------------------------------------
```

there are basically 3 mandatory keys for defining basic data types: `X=type` `type.X=format_specifier` `type.X.size=size_in_bits` For example, let's define `UNIT`, according to Microsoft documentation `UINT` is just equivalent of standard C `unsigned int` (or `uint32_t` in terms of TCC engine). It will be defined as:

```
UINT=type
type.UINT=d
type.UINT.size=32
```

Now there is an optional entry:

```
X.type.pointto=Y
```

This one may only be used in case of pointer `type.X=p`, one good example is LPFILETIME definition, it is a pointer to `_FILETIME` which happens to be a structure. Assuming that we are targeting only 32-bit windows machine, it will be defined as the following:

```
LPFILETIME=type
type.LPFILETIME=p
type.LPFILETIME.size=32
type.LPFILETIME.pointto=_FILETIME
```

This last field is not mandatory because sometimes the data structure internals will be proprietary, and we will not have a clean representation for it.

There is also one more optional entry:

```
type.UINT.meta=4
```

This entry is for integration with C parser and carries the type class information: integer size, signed/unsigned, etc.

Structures

Those are the basic keys for structs (with just two elements):

```
X=struct
struct.X=a,b
struct.X.a=a_type,a_offset,a_number_of_elements
struct.X.b=b_type,b_offset,b_number_of_elements
```

The first line is used to define a structure called `X`, the second line defines the elements of `X` as comma separated values. After that, we just define each element info.

For example. we can have a struct like this one:

```
struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
}
```

assuming we have `DWORD` defined, the struct will look like this

```
 _FILETIME=struct
struct._FILETIME=dwLowDateTime,dwHighDateTime
struct._FILETIME.dwLowDateTime=DWORD,0,0
struct._FILETIME.dwHighDateTime=DWORD,4,0
```

Note that the number of elements field is used in case of arrays only to identify how many elements are in arrays, other than that it is zero by default.

Unions

Unions are defined exactly like structs the only difference is that you will replace the word `struct` with the word `union`.

Function prototypes

Function prototypes representation is the most detail oriented and the most important one of them all. Actually, this is the one used directly for type matching

```
X=func
func.X.args=NumberOfArgs
func.x.arg0=Arg_type,arg_name
.
.
.
func.X.ret=Return_type
func.X.cc=calling_convention
```

It should be self-explanatory. Let's do strncasecmp as an example for x86 arch for Linux machines. According to man pages, strncasecmp is defined as the following:

```
int strcasecmp(const char *s1, const char *s2, size_t n);
```

When converting it into its sdb representation it will look like the following:

```
strcasecmp=func
func.strcasecmp.args=3
func.strcasecmp.arg0=char *,s1
func.strcasecmp.arg1=char *,s2
func.strcasecmp.arg2=size_t,n
func.strcasecmp.ret=int
func.strcasecmp.cc=cdecl
```

Note that the `.cc` part is optional and if it didn't exist the default calling-convention for your target architecture will be used instead. There is one extra optional key

```
func.x.noreturn=true/false
```

This key is used to mark functions that will not return once called, such as `exit` and `_exit`.

Calling Conventions

Radare2 uses calling conventions to help in identifying function formal arguments and return types. It is used also as a guide for basic function prototype and type propagation.

```
[0x00000000]> afc?
|Usage: afc[agl?]
| afc convention  Manually set calling convention for current function
| afc             Show Calling convention for the Current function
| afc=([cctype])  Select or show default calling convention
| afcr[j]         Show register usage for the current function
| afca            Analyse function for finding the current calling convention
| afcf[j] [name]  Prints return type function(arg1, arg2...), see afij
| afck            List SDB details of call loaded calling conventions
| afcl            List all available calling conventions
| afco path       Open Calling Convention sdb profile from given path
| afcR            Register telescoping using the calling conventions order
[0x00000000]>
```

- To list all available calling conventions for current architecture using `afcl` command

```
[0x00000000]> afcl
amd64
ms
```

- To display function prototype of standard library functions you have `afcf` command

```
[0x00000000]> afcf printf
int printf(const char *format)
[0x00000000]> afcf fgets
char *fgets(char *s, int size, FILE *stream)
```

All this information is loaded via sdb under `/libr/anal/d/cc-[arch]-[bits].sdb`

```
default.cc=amd64

ms=cc
cc.ms.name=ms
cc.ms.arg1=rcx
cc.ms.arg2=rdx
cc.ms.arg3=r8
cc.ms.arg3=r9
cc.ms.argn=stack
cc.ms.ret=rax
```

`cc.x.argi=rax` is used to set the ith argument of this calling convention to register name `rax`

`cc.x.argn=stack` means that all the arguments (or the rest of them in case there was argi for any i as counting number) will be stored in stack from left to right

`cc.x.argn=stack_rev` same as `cc.x.argn=stack` except for it means argument are passed right to left

Virtual Tables

There is a basic support of virtual tables parsing (RTTI and others). The most important thing before you start to perform such kind of analysis is to check if the `anal.cpp.abi` option is set correctly, and change if needed.

All commands to work with virtual tables are located in the `av` namespace. Currently, the support is very basic, allowing you only to inspect parsed tables.

```
|Usage: av[?jr*] C++ vtables and RTTI
| av           search for vtables in data sections and show results
| avj          like av, but as json
| av*          like av, but as r2 commands
| avr[j@addr]  try to parse RTTI at vtable addr (see anal.cpp.abi)
| avra[j]      search for vtables and try to parse RTTI at each of them
```

The main commands here are `av` and `avr`. `av` lists all virtual tables found when r2 opened the file. If you are not happy with the result you may want to try to parse virtual table at a particular address with `avr` command. `avra` performs the search and parsing of all virtual tables in the binary, like r2 does during the file opening.

Syscalls

Radare2 allows manual search for assembly code looking like a syscall operation. For example on ARM platform usually they are represented by the `svc` instruction, on the others can be a different instructions, e.g. `syscall` on x86 PC.

```
[0x0001ece0]> /ad/ svc
...
0x000187c2   # 2: svc 0x76
0x000189ea   # 2: svc 0xa9
0x00018a0e   # 2: svc 0x82
...
```

Syscalls detection is driven by `asm.os`, `asm.bits`, and `asm.arch`. Be sure to setup those configuration options accordingly. You can use `asl` command to check if syscalls' support is set up properly and as you expect. The command lists syscalls supported for your platform.

```
[0x0001ece0]> asl
...
sd_softdevice_enable = 0x80.16
sd_softdevice_disable = 0x80.17
sd_softdevice_is_enabled = 0x80.18
...
```

If you setup ESIL stack with `aei` or `aeim`, you can use `/as` command to search the addresses where particular syscalls were found and list them.

```
[0x0001ece0]> aei
[0x0001ece0]> /as
0x000187c2 sd_ble_gap_disconnect
0x000189ea sd_ble_gatts_sys_attr_set
0x00018a0e sd_ble_gap_sec_info_reply
...
```

To reduce searching time it is possible to restrict the searching range for only executable segments or sections with `/as @e:search.in=io.maps.x`

Using the ESIL emulation radare2 can print syscall arguments in the disassembly output. To enable the linear (but very rough) emulation use `asm.emu` configuration variable:

```
[0x0001ece0]> e asm.emu=true
[0x0001ece0]> s 0x000187c2
[0x000187c2]> pdf~svc
   0x000187c2   svc 0x76  ; 118 = sd_ble_gap_disconnect
[0x000187c2]>
```

In case of executing `aae` (or `aaaa` which calls `aae`) command radare2 will push found syscalls to a special `syscall.` flagspace, which can be useful for automation purpose:

```
[0x000187c2]> fs
0    0 * imports
1    0 * symbols
2 1523 * functions
3  420 * strings
4  183 * syscalls
[0x000187c2]> f~syscall
...
0x000187c2 1 syscall.sd_ble_gap_disconnect.0
0x000189ea 1 syscall.sd_ble_gatts_sys_attr_set
0x00018a0e 1 syscall.sd_ble_gap_sec_info_reply
...
```

It also can be interactively navigated through within HUD mode (`V_`)

```
0> syscall.sd_ble_gap_disconnect
 - 0x000187b2  syscall.sd_ble_gap_disconnect
   0x000187c2  syscall.sd_ble_gap_disconnect.0
   0x00018a16  syscall.sd_ble_gap_disconnect.1
   0x00018b32  syscall.sd_ble_gap_disconnect.2
   0x0002ac36  syscall.sd_ble_gap_disconnect.3
```

When debugging in radare2, you can use `dcs` to continue execution until the next syscall. You can also run `dcs*` to trace all syscalls.

```
[0xf7fb9120]> dcs*
Running child until syscalls:-1
child stopped with signal 133
--> SN 0xf7fd3d5b syscall 45 brk (0xffffffda)
child stopped with signal 133
--> SN 0xf7fd28f3 syscall 384 arch_prctl (0xffffffda 0x3001)
child stopped with signal 133
--> SN 0xf7fc81b2 syscall 33 access (0xffffffda 0xf7fd8bf1)
child stopped with signal 133
```

radare2 also has a syscall name to syscall number utility. You can return the syscall name of a given syscall number or vice versa, without leaving the shell.

```
[0x08048436]> asl 1
exit
[0x08048436]> asl write
4
[0x08048436]> ask write
0x80,4,3,iZi
```

See `as?` for more information about the utility.

Emulation

One of the most important things to remember in reverse engineering is a core difference between static analysis and dynamic analysis. As many already know, static analysis suffers from the path explosion problem, which is impossible to solve even in the most basic way without at least a partial emulation.

Thus many professional reverse engineering tools use code emulation while performing an analysis of binary code, and radare2 is no difference here.

For partial emulation (or imprecise full emulation) radare2 uses its own ESIL intermediate language and virtual machine.

Radare2 supports this kind of partial emulation for all platforms that implement ESIL uplifting (x86/x86_64, ARM, arm64, MIPS, powerpc, sparc, AVR, 8051, Gameboy, …).

One of the most common usages of such emulation is to calculate indirect jumps and conditional jumps.

To see the ESIL representation of the program one can use the `ao` command or enable the `asm.esil` configuration variable, to check if the program uplifted correctly, and to grasp how ESIL works:

```
[0x00001660]> pdf
. (fcn) fcn.00001660 40
   fcn.00001660 ();
     ; CALL XREF from 0x00001713 (entry2.fini)
     0x00001660  lea rdi, obj.__progname      ; 0x207220
     0x00001667  push rbp
     0x00001668  lea rax, obj.__progname      ; 0x207220
     0x0000166f  cmp rax, rdi
     0x00001672  mov rbp, rsp
 .< 0x00001675  je 0x1690
     0x00001677  mov rax, qword [reloc._ITM_deregisterTMCloneTable] ; [0x206fd8:8]=0
     0x0000167e  test rax, rax
. < 0x00001681  je 0x1690
    0x00001683  pop rbp
    0x00001684  jmp rax
``> 0x00001690  pop rbp
`     0x00001691  ret
[0x00001660]> e asm.esil=true
[0x00001660]> pdf
. (fcn) fcn.00001660 40
   fcn.00001660 ();
     ; CALL XREF from 0x00001713 (entry2.fini)
     0x00001660  0x205bb9,rip,+,rdi,=
     0x00001667  rbp,8,rsp,-=,rsp,=[8]
     0x00001668  0x205bb1,rip,+,rax,=
     0x0000166f  rdi,rax,==,$z,zf,=,$b64,cf,=,$p,pf,=,$s,sf,=,$o,of,=
     0x00001672  rsp,rbp,=
 .< 0x00001675  zf,?{,5776,rip,=,}
     0x00001677  0x20595a,rip,+,[8],rax,=
     0x0000167e  0,rax,rax,&,==,$z,zf,=,$p,pf,=,$s,sf,=,$0,cf,=,$0,of,=
. < 0x00001681  zf,?{,5776,rip,=,}
    0x00001683  rsp,[8],rbp,=,8,rsp,+=
    0x00001684  rax,rip,=
``> 0x00001690  rsp,[8],rbp,=,8,rsp,+=
`     0x00001691  rsp,[8],rip,=,8,rsp,+=
```

To manually setup the ESIL imprecise emulation you need to run this command sequence:

- `aei` to initialize ESIL VM
- `aeim` to initialize ESIL VM memory (stack)
- `aeip` to set the initial ESIL VM IP (instruction pointer)
- a sequence of `aer` commands to set the initial register values.

While performing emulation, please remember, that ESIL VM cannot emulate external calls or system calls, along with SIMD instructions. Thus the most common scenario is to emulate only a small chunk of the code, like encryption/decryption, unpacking or calculating something.

After we successfully set up the ESIL VM we can interact with it like with a usual debugging mode. Commands interface for ESIL VM is almost identical to the debugging one:

- `aes` to step (or `s` key in visual mode)
- `aesi` to step over the function calls
- `aesu <address>` to step until some specified address
- `aesue <ESIL expression>` to step until some specified ESIL expression met
- `aec` to continue until break (Ctrl-C), this one is rarely used though, due to the omnipresence of external calls

In visual mode, all of the debugging hotkeys will work also in ESIL emulation mode.

Along with usual emulation, there is a possibility to record and replay mode:

- `aets` to list all current ESIL R&R sessions
- `aets+` to create a new one
- `aesb` to step back in the current ESIL R&R session

More about this operation mode you can read in Reverse Debugging chapter.

Emulation in analysis loop

Apart from the manual emulation mode, it can be used automatically in the analysis loop. For example, the `aaaa` command performs the ESIL emulation stage along with others. To disable or enable its usage you can use `anal.esil` configuration variable. There is one more important option, though setting it might be quite dangerous, especially in the case of malware - `emu.write` which allows ESIL VM to modify memory. Sometimes it is required though, especially in the process of deobfuscating or unpacking code.

To show the process of emulation you can set `asm.emu` variable, which will show calculated register and memory values in disassembly comments:

```
[0x00001660]> e asm.emu=true
[0x00001660]> pdf
. (fcn) fcn.00001660 40
   fcn.00001660 ();
     ; CALL XREF from 0x00001713 (entry2.fini)
     0x00001660  lea rdi, obj.__progname ; 0x207220 ; rdi=0x207220 -> 0x464c457f
     0x00001667  push rbp                ; rsp=0xfffffffffffffff8
     0x00001668  lea rax, obj.__progname ; 0x207220 ; rax=0x207220 -> 0x464c457f
     0x0000166f  cmp rax, rdi            ; zf=0x1 -> 0x2464c45 ; cf=0x0 ; pf=0x1 -> 0x2464c45 ; sf=0x0 ; of=0x0
     0x00001672  mov rbp, rsp            ; rbp=0xfffffffffffffff8
 .< 0x00001675  je 0x1690               ; rip=0x1690 -> 0x1f0fc35d ; likely
     0x00001677  mov rax, qword [reloc._ITM_deregisterTMCloneTable] ; [0x206fd8:8]=0 ; rax=0x0
```

```
     0x0000167e  test rax, rax           ; zf=0x1 -> 0x2464c45 ; pf=0x1 -> 0x2464c45 ; sf=0x0 ; cf=0x0 ; of=0x0
 . < 0x00001681  je 0x1690               ; rip=0x1690 -> 0x1f0fc35d ; likely
     0x00001683  pop rbp                 ; rbp=0xffffffffffffffff -> 0x4c457fff ; rsp=0x0
     0x00001684  jmp rax                 ; rip=0x0 ..
 ``> 0x00001690  pop rbp                 ; rbp=0x10102464c457f ; rsp=0x8 -> 0x464c457f
 `   0x00001691  ret                     ; rip=0x0 ; rsp=0x10 -> 0x3e0003
```

Note here `likely` comments, which indicates that ESIL emulation predicted for particular conditional jump to happen.

Apart from the basic ESIL VM setup, you can change the behavior with other options located in `emu.` and `esil.` configuration namespaces.

For manipulating ESIL working with memory and stack you can use the following options:

- `esil.stack` to enable or disable temporary stack for `asm.emu` mode
- `esil.stack.addr` to set stack address in ESIL VM (like `aeim` command)
- `esil.stack.size` to set stack size in ESIL VM (like `aeim` command)
- `esil.stack.depth` limits the number of PUSH operations into the stack
- `esil.romem` specifies read-only access to the ESIL memory
- `esil.fillstack` and `esil.stack.pattern` allows you to use a various pattern for filling ESIL VM stack upon initialization
- `esil.nonull` when set stops ESIL execution upon NULL pointer read or write.

Symbols

Radare2 automatically parses available imports and exports sections in the binary, moreover, it can load additional debugging information if present. Two main formats are supported: DWARF and PDB (for Windows binaries). Note that, unlike many tools radare2 doesn't rely on Windows API to parse PDB files, thus they can be loaded on any other supported platform - e.g. Linux or OS X.

DWARF debug info loads automatically by default because usually it's stored right in the executable file. PDB is a bit of a different beast - it is always stored as a separate binary, thus the different logic of handling it.

At first, one of the common scenarios is to analyze the file from Windows distribution. In this case, all PDB files are available on the Microsoft server, which is by default is in options. See all pdb options in radare2:

```
pdb.autoload = 0
pdb.extract = 1
pdb.server = https://msdl.microsoft.com/download/symbols
pdb.useragent = Microsoft-Symbol-Server/6.11.0001.402
```

Using the variable `pdb.server` you can change the address where radare2 will try to download the PDB file by the GUID stored in the executable header. You can make use of multiple symbol servers by separating each URL with a semi-colon:

```
e pdb.server = https://msdl.microsoft.com/download/symbols;https://symbols.mozilla.org/
```

On Windows, you can also use local network share paths (UNC paths) as symbol servers.

Usually, there is no reason to change default `pdb.useragent`, but who knows where could it be handy?

Because those PDB files are stored as "cab" archives on the server, `pdb.extract=1` says to automatically extract them.

Note that for the automatic downloading to work you need "cabextract" tool, and wget/curl installed.

Sometimes you don't need to do that from the radare2 itself, thus - two handy rabin2 options:

```
 -P              show debug/pdb information
 -PP             download pdb file for binary
```

where `-PP` automatically downloads the pdb for the selected binary, using those `pdb.*` config options. `-P` will dump the contents of the PDB file, which is useful sometimes for a quick understanding of the symbols stored in it.

Apart from the basic scenario of just opening a file, PDB information can be additionally manipulated by the `id` commands:

```
[0x000051c0]> id?
|Usage: id Debug information
| Output mode:
| '*'               Output in radare commands
| id                Source lines
| idp [file.pdb]    Load pdb file information
| idpi [file.pdb]   Show pdb file information
| idpd              Download pdb file on remote server
```

Where `idpi` is basically the same as `rabin2 -P`. Note that `idp` can be also used not only in the static analysis mode, but also in the debugging mode, even if connected via WinDbg.

For simplifying the loading PDBs, especially for the processes with many linked DLLs, radare2 can autoload all required PDBs automatically - you need just set the `e pdb.autoload=true` option. Then if you load some file in debugging mode in Windows, using `r2 -d file.exe` or `r2 -d 2345` (attach to pid 2345), all related PDB files will be loaded automatically.

DWARF information loading, on the other hand, is completely automated. You don't need to run any commands/change any options:

```
r2 `which rabin2`
[0x00002437 8% 300 /usr/local/bin/rabin2]> pd $r
0x00002437  jne 0x2468                  ;[1]
0x00002439  cmp qword reloc.__cxa_finalize_224, 0
0x00002441  push rbp
0x00002442  mov rbp, rsp
0x00002445  je 0x2453                   ;[2]
0x00002447  lea rdi, obj.__dso_handle   ; 0x207c40 ; "@| "
0x0000244e  call 0x2360                 ;[3]
0x00002453  call sym.deregister_tm_clones ;[4]
0x00002458  mov byte [obj.completed.6991], 1 ; obj.__TMC_END__ ; [0x2082f0:1]=0
0x0000245f  pop rbp
0x00002460  ret
0x00002461  nop dword [rax]
0x00002468  ret
0x0000246a  nop word [rax + rax]
;-- entry1.init:
;-- frame_dummy:
0x00002470  push rbp
0x00002471  mov rbp, rsp
0x00002474  pop rbp
0x00002475  jmp sym.register_tm_clones  ;[5]
;-- blob_version:
0x0000247a  push rbp                    ; ../blob/version.c:18
```

```
0x0000247b  mov rbp, rsp
0x0000247e  sub rsp, 0x10
0x00002482  mov qword [rbp - 8], rdi
0x00002486  mov eax, 0x32              ; ../blob/version.c:24 ; '2'
0x0000248b  test al, al                ; ../blob/version.c:19
0x0000248d  je 0x2498                  ;[6]
0x0000248f  lea rax, str.2.0.1_182_gf1aa3aa4d ; 0x60b8 ; "2.0.1-182-gf1aa3aa4d"
0x00002496  jmp 0x249f                 ;[7]
0x00002498  lea rax, 0x000060cd
0x0000249f  mov rsi, qword [rbp - 8]
0x000024a3  mov r8, rax
0x000024a6  mov ecx, 0x40              ; section_end.ehdr
0x000024ab  mov edx, 0x40c0
0x000024b0  lea rdi, str._s_2.1.0_git__d___linux_x86__d_git._s_n ; 0x60d0 ; "%s 2.1.0-git %d @ linux-x86-%d git.%s\n"
0x000024b7  mov eax, 0
0x000024bc  call 0x2350                ;[8]
0x000024c1  mov eax, 0x66              ; ../blob/version.c:25 ; 'f'
0x000024c6  test al, al
0x000024c8  je 0x24d6                  ;[9]
0x000024ca  lea rdi, str.commit:_f1aa3aa4d2599c1ad60e3ecbe5f4d8261b282385_build:_2017_11_06__12:18:39 ; ../blob/version.c:26 ; 0x60f8 ; "commit: f1aa3aa4
0x000024d1  call sym.imp.puts          ;[?]
0x000024d6  mov eax, 0                 ; ../blob/version.c:28
0x000024db  leave                      ; ../blob/version.c:29
0x000024dc  ret
;-- rabin_show_help:
0x000024dd  push rbp                   ; .//rabin2.c:27
```

As you can see, it loads function names and source line information.

Signatures

Radare2 has its own format of the signatures, allowing to both load/apply and create them on the fly. They are available under the `z` command namespace:

```
[0x00000000]> z?
Usage: z[*j-aof/cs] [args]   # Manage zignatures
| z             show zignatures
| z.            find matching zignatures in current offset
| zb[?][n=5]    search for best match
| z*            show zignatures in radare format
| zq            show zignatures in quiet mode
| zj            show zignatures in json format
| zk            show zignatures in sdb format
| z-zignature   delete zignature
| z-*           delete all zignatures
| za[?]         add zignature
| zg            generate zignatures (alias for zaF)
| zo[?]         manage zignature files
| zf[?]         manage FLIRT signatures
| z/[?]         search zignatures
| zc[?]         compare current zignspace zignatures with another one
| zs[?]         manage zignspaces
| zi            show zignatures matching information
```

To load the created signature file you need to load it from SDB file using `zo` command or from the compressed SDB file using `zoz` command.

To create signature you need to make function first, then you can create it from the function:

```
r2 /bin/ls
[0x000051c0]> aaa # this creates functions, including 'entry0'
[0x000051c0]> zaf entry0 entry
[0x000051c0]> z
entry:
  bytes: 31ed4989d15e4889e24883e4f050544c............48............48............ff..........f4
  graph: cc=1 nbbs=1 edges=0 ebbs=1
  offset: 0x000051c0
[0x000051c0]>
```

As you can see it made a new signature with a name `entry` from a function `entry0`. You can show it in JSON format too, which can be useful for scripting:

```
[0x000051c0]> zj~{}
[
  {
    "name": "entry",
    "bytes": "31ed4989d15e4889e24883e4f050544c............48............48............ff..........f4",
    "graph": {
      "cc": "1",
      "nbbs": "1",
      "edges": "0",
      "ebbs": "1"
    },
    "offset": 20928,
    "refs": [
    ]
  }
]
[0x000051c0]>
```

To remove it just run `z-entry`.

If you want, instead, to save all created signatures, you need to save it into the SDB file using command `zos myentry`.

Then we can apply them. Lets open a file again:

```
r2 /bin/ls
 -- Log On. Hack In. Go Anywhere. Get Everything.
[0x000051c0]> zo myentry
[0x000051c0]> z
```

```
entry:
  bytes: 31ed4989d15e4889e24883e4f050544c............48............48............ff..........f4
  graph: cc=1 nbbs=1 edges=0 ebbs=1
  offset: 0x000051c0
[0x000051c0]>
```

This means that the signatures were successfully loaded from the file `myentry` and now we can search matching functions:

```
[0x000051c0]> z.
[+] searching 0x000051c0 - 0x000052c0
[+] searching function metrics
hits: 1
[0x000051c0]>
```

Note that `z.` command just checks the signatures against the current address. To search signatures across the all file we need to do a bit different thing. There is an important moment though, if we just run it "as is" - it wont find anything:

```
[0x000051c0]> z/
[+] searching 0x0021dfd0 - 0x002203e8
[+] searching function metrics
hits: 0
[0x000051c0]>
```

Note the searching address - this is because we need to adjust the searching range first:

```
[0x000051c0]> e search.in=io.section
[0x000051c0]> z/
[+] searching 0x000038b0 - 0x00015898
[+] searching function metrics
hits: 1
[0x000051c0]>
```

We are setting the search mode to `io.section` (it was `file` by default) to search in the current section (assuming we are currently in the `.text` section of course). Now we can check, what radare2 found for us:

```
[0x000051c0]> pd 5
;-- entry0:
;-- sign.bytes.entry_0:
0x000051c0      31ed           xor ebp, ebp
0x000051c2      4989d1         mov r9, rdx
0x000051c5      5e             pop rsi
0x000051c6      4889e2         mov rdx, rsp
0x000051c9      4883e4f0       and rsp, 0xfffffffffffffff0
[0x000051c0]>
```

Here we can see the comment of `entry0`, which is taken from the ELF parsing, but also the `sign.bytes.entry_0`, which is exactly the result of matching signature.

Signatures configuration stored in the `zign.` config vars' namespace:

```
[0x000051c0]> e? zign.
       zign.autoload: Autoload all zignatures located in ~/.local/share/radare2/zigns
          zign.bytes: Use bytes patterns for matching
   zign.diff.bthresh: Threshold for diffing zign bytes [0, 1] (see zc?)
   zign.diff.gthresh: Threshold for diffing zign graphs [0, 1] (see zc?)
          zign.graph: Use graph metrics for matching
           zign.hash: Use Hash for matching
          zign.maxsz: Maximum zignature length
          zign.mincc: Minimum cyclomatic complexity for matching
          zign.minsz: Minimum zignature length for matching
         zign.offset: Use original offset for matching
         zign.prefix: Default prefix for zignatures matches
           zign.refs: Use references for matching
      zign.threshold: Minimum similarity required for inclusion in zb output
          zign.types: Use types for matching
[0x000051c0]>
```

Finding Best Matches `zb`

Often you know the signature should exist somewhere in a binary but `z/` and `z.` still fail. This is often due to very minor differences between the signature and the function. Maybe the compiler switched two instructions, or your signature is not for the correct function version. In these situations the `zb` commands can still help point you in the right direction by listing near matches.

```
[0x000040a0]> zb?
Usage: zb[r?] [args]  # search for closest matching signatures
| zb [n]           find n closest matching zignatures to function at current offset
| zbr zigname [n]  search for n most similar functions to zigname
```

The `zb` (zign best) command will show the top 5 closest signatures to a function. Each will contain a score between 1.0 and 0.0.

```
[0x0041e390]> s sym.fclose
[0x0040fc10]> zb
0.96032  0.92400 B  0.99664 G   sym.fclose
0.65971  0.35600 B  0.96342 G   sym._nl_expand_alias
0.65770  0.37800 B  0.93740 G   sym.fdopen
0.65112  0.35000 B  0.95225 G   sym.__run_exit_handlers
0.62532  0.34800 B  0.90264 G   sym.__cxa_finalize
```

In the above example, `zb` correctly associated the `sym.fclose` signature to the current function. The `z/` and `z.` command would have failed to match here since both the `Byte` and `Graph` scores are less then 1.0. A 30% separation between the first and second place results is also a good indication of a correct match.

The `zbr` (zign best reverse) accepts a zignature name and attempts to find the closet matching functions. Use an analysis command, like `aa` to find functions first.

```
[0x00401b20]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x00401b20]> zo ./libc.sdb
[0x00401b20]> zbr sym.__libc_malloc 10
0.94873  0.89800 B  0.99946 G   sym.malloc
0.65245  0.40600 B  0.89891 G   sym._mid_memalign
0.59470  0.38600 B  0.80341 G   sym._IO_flush_all_lockp
0.59200  0.28200 B  0.90201 G   sym._IO_file_underflow
0.57802  0.30400 B  0.85204 G   sym.__libc_realloc
```

```
0.57094  0.35200 B  0.78988 G   sym.__calloc
0.56785  0.34000 B  0.79570 G   sym._IO_un_link.part.0
0.56358  0.36200 B  0.76516 G   sym._IO_cleanup
0.56064  0.26000 B  0.86127 G   sym.intel_check_word.constprop.0
0.55726  0.28400 B  0.83051 G   sym.linear_search_fdes
```

Graph commands

When analyzing data it is usually handy to have different ways to represent it in order to get new perspectives to allow the analyst to understand how different parts of the program interact.

Representing basic block edges, function calls, string references as graphs show a very clear view of this information.

Radare2 supports various types of graph available through commands starting with `ag`:

```
[0x00005000]> ag?
|Usage: ag<graphtype><format> [addr]
| Graph commands:
| aga[format]             Data references graph
| agA[format]             Global data references graph
| agc[format]             Function callgraph
| agC[format]             Global callgraph
| agd[format] [fcn addr]  Diff graph
| agf[format]             Basic blocks function graph
| agi[format]             Imports graph
| agr[format]             References graph
| agR[format]             Global references graph
| agx[format]             Cross references graph
| agg[format]             Custom graph
| ag-                     Clear the custom graph
| agn[?] title body       Add a node to the custom graph
| age[?] title1 title2    Add an edge to the custom graph

Output formats:
| <blank>                 Ascii art
| *                       r2 commands
| d                       Graphviz dot
| g                       Graph Modelling Language (gml)
| j                       json ('J' for formatted disassembly)
| k                       SDB key-value
| t                       Tiny ascii art
| v                       Interactive ascii art
| w [path]                Write to path or display graph image (see graph.gv.format and graph.web)
```

The structure of the commands is as follows: `ag <graph type> <output format>`.

For example, `agid` displays the imports graph in dot format, while `aggj` outputs the custom graph in JSON format.

Here's a short description for every output format available:

Ascii Art ** (e.g. `agf`)

Displays the graph directly to stdout using ASCII art to represent blocks and edges.

Warning: displaying large graphs directly to stdout might prove to be computationally expensive and will make r2 not responsive for some time. In case of a doubt, prefer using the interactive view (explained below).

Interactive Ascii Art (e.g. `agfv`)

Displays the ASCII graph in an interactive view similar to `VV` which allows to move the screen, zoom in / zoom out, …

Tiny Ascii Art (e.g. `agft`)

Displays the ASCII graph directly to stdout in tiny mode (which is the same as reaching the maximum zoom out level in the interactive view).

Graphviz dot (e.g. `agfd`)

Prints the dot source code representing the graph, which can be interpreted by programs such as graphviz or online viewers like this

JSON (e.g. `agfj`)

Prints a JSON string representing the graph.

- In case of the `f` format (basic blocks of function), it will have detailed information about the function and will also contain the disassembly of the function (use `J` format for the formatted disassembly.
- In all other cases, it will only have basic information about the nodes of the graph (id, title, body, and edges).

Graph Modelling Language (e.g. `agfg`)

Prints the GML source code representing the graph, which can be interpreted by programs such as yEd

SDB key-value (e.g. `agfk`)

Prints key-value strings representing the graph that was stored by sdb (radare2's string database).

R2 custom graph commands (e.g. `agf*`)

Prints r2 commands that would recreate the desired graph. The commands to construct the graph are `agn [title] [body]` to add a node and `age [title1] [title2]` to add an edge. The `[body]` field can be expressed in base64 to include special formatting (such as newlines).

To easily execute the printed commands, it is possible to prepend a dot to the command (`.agf*`).

Web / image (e.g. `agfw`)

Radare2 will convert the graph to dot format, use the `dot` program to convert it to a `.gif` image and then try to find an already installed viewer on your system (`xdg-open`, `open`, …) and display the graph there.

The extension of the output image can be set with the `graph.extension` config variable. Available extensions are `png, jpg, gif, pdf, ps`.

Note: for particularly large graphs, the most recommended extension is `svg` as it will produce images of much smaller size

If `graph.web` config variable is enabled, radare2 will try to display the graph using the browser (*this feature is experimental and unfinished, and disabled by default.*)

Scripting

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 `` `command` ``. The quoted part will undergo command substitution and the output will be used as an argument of the command line.

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
```

Radare2 also provides quite a few Unix type file processing commands like head, tail, cat, grep and many more. One such command is Uniq, which can be used to filter a file to display only non-duplicate content. So to make a new file with only unique strings, you can do:

```
[0x00404800]> uniq file > uniq_file
```

The head command can be used to see the first N number of lines in the file, similarly tail command allows the last N number of lines to be seen.

```
[0x00404800]> head 3 foodtypes.txt
1 Protein
2 Carbohydrate
3 Fat
[0x00404800]> tail 2 foodtypes.txt
3 Shake
4 Milk
```

The join command could be used to merge two different files with common first field.

```
[0x00404800]> cat foodtypes.txt
1 Protein
2 Carbohydrate
3 Fat
[0x00404800]> cat foods.txt
1 Cheese
2 Potato
3 Butter
[0x00404800]> join foodtypes foods.txt
1 Protein Cheese
2 Carbohydrate Potato
3 Fat Butter
```

Similarly, sorting the content is also possible with the sort command. A typical example could be:

```
[0x00404800]> sort file
eleven
five
five
great
one
one
radare
```

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.

Loops

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]> fs functions
[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.*`. There are also a predefined loop called `@@f`, which runs your command on every functions found by r2:

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

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  xor ebp, ebp
 0x004047d2  mov r9, rdx
 0x004047d5  pop rsi
 0x004047d6  mov rdx, rsp
 0x004047d9  and rsp, 0xfffffffffffffff0
 0x004047dd  push rax
 0x004047de  push rsp
 0x004047df  mov r8, 0x4136c0
 0x004047e6  mov rcx, 0x413660      ; "AWA..AVI..AUI..ATL.%.. "
0A..AVI..AUI.
 0x004047ed  mov rdi, main          ; "AWAVAUATUH..S..H...." @
0
 0x004047f4  call sym.imp.__libc_start_main
 0x004047f9  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). There are other options too (not complete list, check `@@?` for more information): - `@@k sdbquery` - iterate over all offsets returned by that sdbquery - `@@t-` iterate over on all threads (see dp) - `@@b` - iterate over all basic blocks of current function (see afb) - `@@f` - iterate over all functions (see aflq)

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]
```

Beware, @@@ is not compatible with JSON commands.

Macros

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  mov eax, 0x61e627      ; "tab"
0x00404805  push rbp
0x00404806  sub rax, section_end.LOAD1
0x0040480c  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.

```
[0x00404800]
[0x004047d0]> (foo x y; pd $0; s +$1)
[0x004047d0]> .(foo 5 6)
;-- entry0:
0x004047d0      xor ebp, ebp
0x004047d2      mov r9, rdx
0x004047d5      pop rsi
0x004047d6  mov rdx, rsp
0x004047d9  and rsp, 0xfffffffffffffff0
[0x004047d6]>
```

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

Aliases

radare2 also offers aliases which might help you save time by quickly executing your most used commands. They are under `$?`

The general usage of the feature is: `$alias=cmd`

```
[0x00404800]> $disas=pdf
```

The above command will create an alias `disas` for `pdf`. The following command prints the disassembly of the main function.

```
[0x00404800]> $disas @ main
```

Apart from commands, you can also alias a text to be printed, when called.

```
[0x00404800]> $my_alias=$test input
[0x00404800]> $my_alias
test input
```

To undefine alias, use `$alias=`:

```
[0x00404800]> $pmore='b 300;px'
[0x00404800]> $
$pmore
[0x00404800]> $pmore=
[0x00404800]> $
```

A single `$` in the above will list all defined aliases. It's also possible check the aliased command of an alias:

```
[0x00404800]> $pmore?
b 200; px
```

Can we create an alias contains alias ? The answer is yes:

```
[0x00404800]> $pStart='s 0x0;$pmore'
[0x00404800]> $pStart
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00000000  7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF...........
0x00000010  0300 3e00 0100 0000 1014 0000 0000 0000  ..>............
0x00000020  4000 0000 0000 0000 5031 0000 0000 0000  @.......P1......
0x00000030  0000 0000 4000 3800 0d00 4000 1e00 1d00  ....@.8...@.....
0x00000040  0600 0000 0400 0000 4000 0000 0000 0000  ........@.......
0x00000050  4000 0000 0000 0000 4000 0000 0000 0000  @.......@.......
0x00000060  d802 0000 0000 0000 d802 0000 0000 0000  ................
0x00000070  0800 0000 0000 0000 0300 0000 0400 0000  ................
0x00000080  1803 0000 0000 0000 1803 0000 0000 0000  ................
0x00000090  1803 0000 0000 0000 1c00 0000 0000 0000  ................
0x000000a0  1c00 0000 0000 0000 0100 0000 0000 0000  ................
0x000000b0  0100 0000 0400 0000 0000 0000 0000 0000  ................
0x000000c0  0000 0000 0000 0000                      ........
[0x00000000]>
```

R2pipe

The r2pipe api was initially designed for NodeJS in order to support reusing the web's r2.js API from the commandline. The r2pipe module permits interacting with r2 instances in different methods:

- spawn pipes (r2 -0)
- http queries (cloud friendly)
- tcp socket (r2 -c)

```
         pipe spawn async http tcp rap json
nodejs    x     x     x    x    x   -    x
python    x     x     -    x    x   x    x
swift     x     x     x    x    -   -    x
dotnet    x     x     x    x    -   -    -
haskell   x     x     -    x    -   -    x
java      -     x     -    x    -   -    -
golang    x     x     -    -    -   -    x
ruby      x     x     -    -    -   -    x
rust      x     x     -    -    -   -    x
vala      -     x     x    -    -   -    -
erlang    x     x     -    -    -   -    -
newlisp   x     -     -    -    -   -    -
dlang     x     -     -    -    -   -    x
perl      x     -     -    -    -   -    -
```

Examples

Python

```
$ pip install r2pipe
```

```python
import r2pipe

r2 = r2pipe.open("/bin/ls")
r2.cmd('aa')
print(r2.cmd("afl"))
print(r2.cmdj("aflj"))  # evaluates JSONs and returns an object
```

NodeJS

Use this command to install the r2pipe bindings

```
$ npm install r2pipe
```

Here's a sample hello world

```
const r2pipe = require('r2pipe');
r2pipe.open('/bin/ls', (err, res) => {
  if (err) {
  throw err;
  }
  r2.cmd ('af @ entry0', function (o) {
  r2.cmd ("pdf @ entry0", function (o) {
    console.log (o);
    r.quit ()
  });
  });
});
```

Checkout the GIT repository for more examples and details.

https://github.com/radareorg/radare2-r2pipe/blob/master/nodejs/r2pipe/README.md

Go

```
$ r2pm -i r2pipe-go
```

https://github.com/radare/r2pipe-go

```go
package main

import (
  "fmt"
  "github.com/radare/r2pipe-go"
)

func main() {
  r2p, err := r2pipe.NewPipe("/bin/ls")
  if err != nil {
    panic(err)
  }
  defer r2p.Close()
  buf1, err := r2p.Cmd("?E Hello World")
  if err != nil {
    panic(err)
  }
  fmt.Println(buf1)
}
```

Rust

```
$ cat Cargo.toml
...
[dependencies]
r2pipe = "*"
```

```rust
#[macro_use]
extern crate r2pipe;
use r2pipe::R2Pipe;
fn main() {
  let mut r2p = open_pipe!(Some("/bin/ls")).unwrap();
  println!("{:?}", r2p.cmd("?e Hello World"));
  let json = r2p.cmdj("ij").unwrap();
  println!("{}", serde_json::to_string_pretty(&json).unwrap());
  println!("ARCH {}", json["bin"]["arch"]);
  r2p.close();
}
```

Ruby

```
$ gem install r2pipe
```

```ruby
require 'r2pipe'
puts 'r2pipe ruby api demo'
puts '===================='
r2p = R2Pipe.new '/bin/ls'
puts r2p.cmd 'pi 5'
puts r2p.cmd 'pij 1'
puts r2p.json(r2p.cmd 'pij 1')
puts r2p.cmd 'px 64'
r2p.quit
```

Perl

```perl
#!/usr/bin/perl

use R2::Pipe;
use strict;
```

```perl
my $r = R2::Pipe->new ("/bin/ls");
print $r->cmd ("pd 5")."\n";
print $r->cmd ("px 64")."\n";
$r->quit ();
```

Erlang

```erlang
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -smp enable

%% -sname hr
-mode(compile).

-export([main/1]).

main(_Args) ->
  %% adding r2pipe to modulepath, set it to your r2pipe_erl location
  R2pipePATH = filename:dirname(escript:script_name()) ++ "/ebin",
  true = code:add_pathz(R2pipePATH),

  %% initializing the link with r2
  H = r2pipe:init(lpipe),

  %% all work goes here
  io:format("~s", [r2pipe:cmd(H, "i")]).
```

Haskell

```haskell
import R2pipe
import qualified Data.ByteString.Lazy as L

showMainFunction ctx = do
  cmd ctx "s main"
  L.putStr =<< cmd ctx "pD `fl $$`"

main = do
  -- Run r2 locally
  open "/bin/ls" >>= showMainFunction
  -- Connect to r2 via HTTP (e.g. if "r2 -qc=h /bin/ls" is running)
  open "http://127.0.0.1:9090" >>= showMainFunction
```

Dotnet

```csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using r2pipe;

namespace LocalExample {
  class Program {
    static void Main(string[] args) {
#if __MonoCS__
      using(IR2Pipe pipe = new R2Pipe("/bin/ls")) {
#else
      using (IR2Pipe pipe = new R2Pipe(@"C:\Windows\notepad.exe",
        @"C:\radare2\radare2.exe")) {
#endif
        Console.WriteLine("Hello r2! " + pipe.RunCommand("?V"));
        Task<string> async = pipe.RunCommandAsync("?V");
        Console.WriteLine("Hello async r2!" + async.Result);
        QueuedR2Pipe qr2 = new QueuedR2Pipe(pipe);
        qr2.Enqueue(new R2Command("x", (string result) => {
             Console.WriteLine("Result of x:\n {0}", result); }));
        qr2.Enqueue(new R2Command("pi 10", (string result) => {
             Console.WriteLine("Result of pi 10:\n {0}", result); }));
        qr2.ExecuteCommands();
      }
    }
  }
}
```

Java

```java
import org.radare.r2pipe.R2Pipe;

public class Test {
  public static void main (String[] args) {
    try {
      R2Pipe r2p = new R2Pipe ("/bin/ls");
      // new R2Pipe ("http://cloud.rada.re/cmd/", true);
      System.out.println (r2p.cmd ("pd 10"));
      System.out.println (r2p.cmd ("px 32"));
      r2p.quit();
    } catch (Exception e) {
      System.err.println (e);
    }
  }
}
```

Swift

```
if let r2p = R2Pipe(url:nil) {
  r2p.cmd ("?V", closure:{
    (str:String?) in
    if let s = str {
      print ("Version: \(s)");
      exit (0);
    } else {
      debugPrint ("R2PIPE. Error");
      exit (1);
    }
  });
  NSRunLoop.currentRunLoop().run();
} else {
  print ("Needs to run from r2")
}
```

Vala

```
public static int main (string[] args) {
  MainLoop loop = new MainLoop ();
  var r2p = new R2Pipe ("/bin/ls");
  r2p.cmd ("pi 4", (x) => {
    stdout.printf ("Disassembly:\n%s\n", x);
    r2p.cmd ("ie", (x) => {
      stdout.printf ("Entrypoint:\n%s\n", x);
      r2p.cmd ("q");
    });
  });
  ChildWatch.add (r2p.child_pid, (pid, status) => {
    Process.close_pid (pid);
    loop.quit ();
  });
  loop.run ();
  return 0;
}
```

NewLisp

```
(load "r2pipe.lsp")
(println "pd 3:\n" (r2pipe:cmd "pd 3"))
(exit)
```

Dlang

```
import std.stdio;
import r2pipe;

void main() {
   auto r2 = r2pipe.open ();
   writeln ("Hello "~ r2.cmd("?e World"));
   writeln ("Hello "~ r2.cmd("?e Works"));

   string uri = r2.cmdj("ij")["core"]["uri"].str;
   writeln ("Uri: ",uri);
}
```

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 or data structures.

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. Access to the IO system is achieved with `=!`. Most IO plugins provide help with `=!?` or `=!help`. For example:

```
$ r2 -d /bin/ls
...
[0x7fc15afa3cc0]> =!help
Usage: =!cmd args
 =!ptrace   - use ptrace io
 =!mem      - use /proc/pid/mem io if possible
 =!pid      - show targeted pid
 =!pid <#>  - select new pid
```

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   # Debug commands
| db[?]                     Breakpoints commands
| dbt[?]                    Display backtrace based on dbg.btdepth and dbg.btalgo
| dc[?]                     Continue execution
| dd[?]                     File descriptors (!fd in r1)
| de[-sc] [perm] [rm] [e]   Debug with ESIL (see de?)
| dg <file>                 Generate a core-file (WIP)
```

```
| dH [handler]             Transplant process to a new handler
| di[?]                    Show debugger backend information (See dh)
| dk[?]                    List, send, get, set, signal handlers of child
| dL[?]                    List or set debugger handler
| dm[?]                    Show memory maps
| do[?]                    Open process (reload, alias for 'oo')
| doo[args]                Reopen in debug mode with args (alias for 'ood')
| doof[file]               Reopen in debug mode from file (alias for 'oodf')
| doc                      Close debug session
| dp[?]                    List, attach to process or thread id
| dr[?]                    Cpu registers
| ds[?]                    Step, over, source line
| dt[?]                    Display instruction traces
| dw <pid>                 Block prompt until pid dies
| dx[?]                    Inject and run code on target process (See gs)
```

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

Migration from ida, GDB or WinDBG

How to run the program using the debugger

`r2 -d /bin/ls` - start in debugger mode => [video]

How do I attach/detach to running process ? (gdb -p)

`r2 -d <pid>` - attach to process

`r2 ptrace://pid` - same as above, but only for io (not debugger backend hooked)

`[0x7fff6ad90028]> o-225` - close fd=225 (listed in `o~[1]:0`)

`r2 -D gdb gdb://localhost:1234` - attach to gdbserver

How to set args/environment variable/load a specific libraries for the debugging session of radare

Use `rarun2` (`libpath=$PWD:/tmp/lib`, `arg2=hello`, `setenv=FOO=BAR` …) see `rarun2 -h` / `man rarun2`

How to script radare2 ?

`r2 -i <scriptfile> ...` - run a script **after** loading the file => [video]

`r2 -I <scriptfile> ...` - run a script **before** loading the file

`r2 -c $@ | awk $@` - run through awk to get asm from function => [link]

`[0x80480423]> . scriptfile` - interpret this file => [video]

`[0x80480423]> #!c` - enter C repl (see `#!` to list all available RLang plugins) => [video], everything have to be done in a oneliner or a .c file must be passed as an argument.

To get `#!python` and much more, just build radare2-bindings

How to list Source code as in gdb list ?

`CL @ sym.main` - though the feature is highly experimental

shortcuts

Command	IDA Pro	radare2	r2 (visual mode)	GDB	WinDbg
Analysis					
Analysis of everything	`Automatically launched when opening a binary`	`aaa or -A (aaaa or -AA for even experimental analysis)`	`N/A`	N/A	N/A
Navigation					
xref to	`x`	`axt`	`x`	N/A	N/A
xref from	`ctrl + j`	`axf`	`X`	N/A	N/A
xref to graph	?	`agt [offset]`	?	N/A	N/A

Command	IDA Pro	radare2	r2 (visual mode)	GDB	WinDbg
xref from graph	?	agf [offset]	?	N/A	N/A
list functions	alt + 1	afl;is	t	N/A	N/A
listing	alt + 2	pdf	p	N/A	N/A
hex mode	alt + 3	pxa	P	N/A	N/A
imports	alt + 6	ii	:ii	N/A	N/A
exports	alt + 7	is~FUNC	?	N/A	N/A
follow jmp/call	enter	s offset	enter or 0-9	N/A	N/A
undo seek	esc	s-	u	N/A	N/A
redo seek	ctrl+enter	s+	U	N/A	N/A
show graph	space	agv	V	N/A	N/A
Edit					
rename	n	afn	dr	N/A	N/A
graph view	space	agv	V	N/A	N/A
define as data	d	Cd [size]	dd,db,dw,dW	N/A	N/A
define as code	c	C- [size]	d- or du	N/A	N/A
define as undefined	u	C- [size]	d- or du	N/A	N/A
define as string	A	Cs [size]	ds	N/A	N/A
define as struct	Alt+Q	Cf [size]	dF	N/A	N/A
Debugger					
Start Process/ Continue execution	F9	dc	F9	r and c	g
Terminate Process	Ctrl+F2	dk 9	?	kill	q
Detach	?	o-	?	detach	
step into	F7	ds	s	n	t
step into 4 instructions	?	ds 4	F7	n 4	t 4
step over	F8	dso	S	s	p
step until a specific address	?	dsu <addr>	?	s	g <addr>
Run until return	Ctrl+F7	dcr	?	finish	gu
Run until cursor	F4	#249	#249	N/A	N/A
Show Backtrace	?	dbt	?	bt	
display Register	On register Windows	dr all	Shown in Visual mode	info registers	r
display eax	On register Windows	dr?eax	Shown in Visual mode	info registers eax	r rax
display old state of all registers	?	dro	?	?	?
display function addr + N	?	afi $$ - display function information of current offset ($$)	?	?	?
display frame state	?	pxw rbp-rsp@rsp	?	i f	?
How to step until condition is true	?	dsi	?	?	?
Update a register value	?	dr rip=0x456	?	set $rip=0x456	r rip=456
Disassembly					
disassembly forward	N/A	pd	Vp	disas	uf, u
disassembly N instructions	N/A	pd X	Vp	x/i	u <addr> LX
disassembly N (backward)	N/A	pd -X	Vp	disas <a-o> <a>	ub
Information on the bin					
Sections/regions	Menu sections	iS or S (append j for json)	N/A	maint info sections	!address
Load symbol file					
Sections/regions	pdb menu	asm.dwarf.file, pdb.XX)	N/A	add-symbol-file	r
BackTrace					
Stack Trace	N/A	dbt	N/A	bt	k
Stack Trace in Json	N/A	dbtj	N/A		
Partial Backtrace (innermost)	N/A	dbt (dbg.btdepth dbg.btalgo)	N/A	bt	k
Partial Backtrace (outermost)	N/A	dbt (dbg.btdepth dbg.btalgo)	N/A	bt -	
Stacktrace for all threads	N/A	dbt@t	N/A	thread apply all bt	~* k
Breakpoints					
Breakpoint list	Ctrl+Alt+B	db	?	info breakpoints	bl
add breakpoint	F2	db [offset]	F2	break	bp
Threads					
Switch to thread	Thread menu	dp	N/A	thread <N>	~<N>s
Frames					
Frame Numbers	N/A	?	N/A	any bt command	kn
Select Frame	N/A	?	N/A	frame	.frame
Parameters/Locals					
Display parameters	N/A	afv	N/A	info args	dv /t /i /V
Display parameters	N/A	afv	N/A	info locals	dv /t /i /V
Display parameters/locals in json	N/A	afvj	N/A	info locals	dv /t /i /V
list addresses where vars are accessed(R/W)	N/A	afvR/afvW	N/A	?	?
Project Related					
open project		Po [file]		?	
save project	automatic	Ps [file]		?	
show project informations		Pi [file]		?	
Miscellaneous					
Dump byte char array	N/A	pc? (json, C, char, etc.)	Vpppp	x/bc	db
options	option menu	e?	e		

Command	IDA Pro	radare2	r2 (visual mode)	GDB	WinDbg
search	search menu	`/?`	Select the zone with the cursor `c` then `/`		`s`

Equivalent of "set-follow-fork-mode" gdb command

This can be done using 2 commands:

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

Common features

- r2 accepts FLIRT signatures
- r2 can connect to GDB, LLVM and WinDbg
- r2 can write/patch in place
- r2 have fortunes and [s]easter eggs[/s]balls of steel
- r2 can do basic loading of ELF core files from the box and MDMP (Windows minidumps)

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
```

Current state of registers

```
[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
```

Memory Maps

The ability to understand and manipulate the memory maps of a debugged program is important for many different Reverse Engineering tasks. radare2 offers a rich set of commands to handle memory maps in the binary. This includes listing the memory maps of the currently debugged binary, removing memory maps, handling loaded libraries and more.

First, let's see the help message for `dm`, the command which is responsible for handling memory maps:

```
[0x55f2104cf620]> dm?
Usage: dm   # Memory maps commands
| dm                              List memory maps of target process
| dm address size                 Allocate <size> bytes at <address> (anywhere if address is -1) in child process
| dm=                             List memory maps of target process (ascii-art bars)
| dm.                             Show map name of current address
| dm*                             List memmaps in radare commands
| dm- address                     Deallocate memory map of <address>
| dmd[a] [file]                   Dump current (all) debug map region to a file (from-to.dmp) (see Sd)
| dmh[?]                          Show map of heap
| dmi [addr|libname] [symname]    List symbols of target lib
| dmi* [addr|libname] [symname]   List symbols of target lib in radare commands
| dmi.                            List closest symbol to the current address
| dmiv                            Show address of given symbol for given lib
| dmj                             List memmaps in JSON format
| dml <file>                      Load contents of file into the current map region
| dmm[?][j*]                      List modules (libraries, binaries loaded in memory)
| dmp[?] <address> <size> <perms>  Change page at <address> with <size>, protection <perms> (perm)
| dms[?] <id> <mapaddr>           Take memory snapshot
| dms- <id> <mapaddr>             Restore memory snapshot
| dmS [addr|libname] [sectname]   List sections of target lib
| dmS* [addr|libname] [sectname]  List sections of target lib in radare commands
| dmL address size                Allocate <size> bytes at <address> and promote to huge page
```

In this chapter, we'll go over some of the most useful subcommands of `dm` using simple examples. For the following examples, we'll use a simple `helloworld` program for Linux but it'll be the same for every binary.

First things first - open a program in debugging mode:

```
$ r2 -d helloworld
Process with PID 20304 started...
= attach 20304 20304
bin.baddr 0x56136b475000
Using 0x56136b475000
asm.bits 64
[0x7f133f022fb0]>
```

> Note that we passed "helloworld" to radare2 without "./". radare2 will try to find this program in the current directory and then in $PATH, even if no "./" is passed. This is contradictory with UNIX systems, but makes the behaviour consistent for windows users

Let's use `dm` to print the memory maps of the binary we've just opened:

```
[0x7f133f022fb0]> dm
0x0000563a0113a000 - usr   4K s r-x /tmp/helloworld /tmp/helloworld ; map.tmp_helloworld.r_x
```

```
0x0000563a0133a000 - usr   8K s rw- /tmp/helloworld /tmp/helloworld ; map.tmp_helloworld.rw
0x00007f133f022000 * usr 148K s r-x /usr/lib/ld-2.27.so /usr/lib/ld-2.27.so ; map.usr_lib_ld_2.27.so.r_x
0x00007f133f246000 - usr   8K s rw- /usr/lib/ld-2.27.so /usr/lib/ld-2.27.so ; map.usr_lib_ld_2.27.so.rw
0x00007f133f248000 - usr   4K s rw- unk0 unk0 ; map.unk0.rw
0x00007fffd25ce000 - usr 132K s rw- [stack] [stack] ; map.stack_.rw
0x00007fffd25f6000 - usr  12K s r-- [vvar] [vvar] ; map.vvar_.r
0x00007fffd25f9000 - usr   8K s r-x [vdso] [vdso] ; map.vdso_.r_x
0xffffffffff600000 - usr   4K s r-x [vsyscall] [vsyscall] ; map.vsyscall_.r_x
```

For those of you who prefer a more visual way, you can use `dm=` to see the memory maps using an ASCII-art bars. This will be handy when you want to see how these maps are located in the memory.

If you want to know the memory-map you are currently in, use `dm.`:

```
[0x7f133f022fb0]> dm.
0x00007f947eed9000 # 0x00007f947eefe000 * usr   148K s r-x /usr/lib/ld-2.27.so /usr/lib/ld-2.27.so ; map.usr_lib_ld_2.27.so.r_x
```

Using `dmm` we can "List modules (libraries, binaries loaded in memory)", this is quite a handy command to see which modules were loaded.

```
[0x7fa80a19dfb0]> dmm
0x55ca23a4a000 /tmp/helloworld
0x7fa80a19d000 /usr/lib/ld-2.27.so
```

Note that the output of `dm` subcommands, and `dmm` specifically, might be different in various systems and different binaries.

We can see that along with our `helloworld` binary itself, another library was loaded which is `ld-2.27.so`. We don't see `libc` yet and this is because radare2 breaks before `libc` is loaded to memory. Let's use `dcu` (**d**ebug **c**ontinue **u**ntil) to execute our program until the entry point of the program, which radare flags as `entry0`.

```
[0x7fa80a19dfb0]> dcu entry0
Continue until 0x55ca23a4a520 using 1 bpsize
hit breakpoint at: 55ca23a4a518
[0x55ca23a4a520]> dmm
0x55ca23a4a000 /tmp/helloworld
0x7fa809de1000 /usr/lib/libc-2.27.so
0x7fa80a19d000 /usr/lib/ld-2.27.so
```

Now we can see that `libc-2.27.so` was loaded as well, great!

Speaking of `libc`, a popular task for binary exploitation is to find the address of a specific symbol in a library. With this information in hand, you can build, for example, an exploit which uses ROP. This can be achieved using the `dmi` command. So if we want, for example, to find the address of `system()` in the loaded `libc`, we can simply execute the following command:

```
[0x55ca23a4a520]> dmi libc system
514 0x00000000 0x7fa809de1000  LOCAL  FILE    0 system.c
515 0x00043750 0x7fa809e24750  LOCAL  FUNC 1221 do_system
4468 0x001285a0 0x7fa809f095a0 LOCAL  FUNC  100 svcerr_systemerr
5841 0x001285a0 0x7fa809f095a0 LOCAL  FUNC  100 svcerr_systemerr
6427 0x00043d10 0x7fa809e24d10  WEAK  FUNC   45 system
7094 0x00043d10 0x7fa809e24d10 GLBAL  FUNC   45 system
7480 0x001285a0 0x7fa809f095a0 GLBAL  FUNC  100 svcerr_systemerr
```

Similar to the `dm.` command, with `dmi.` you can see the closest symbol to the current address.

Another useful command is to list the sections of a specific library. In the following example we'll list the sections of `ld-2.27.so`:

```
[0x55a7ebf09520]> dmS ld-2.27
[Sections]
00 0x00000000     0 0x00000000     0 ---- ld-2.27.so.
01 0x000001c8    36 0x4652d1c8    36 -r-- ld-2.27.so..note.gnu.build_id
02 0x000001f0   352 0x4652d1f0   352 -r-- ld-2.27.so..hash
03 0x00000350   412 0x4652d350   412 -r-- ld-2.27.so..gnu.hash
04 0x000004f0   816 0x4652d4f0   816 -r-- ld-2.27.so..dynsym
05 0x00000820   548 0x4652d820   548 -r-- ld-2.27.so..dynstr
06 0x00000a44    68 0x4652da44    68 -r-- ld-2.27.so..gnu.version
07 0x00000a88   164 0x4652da88   164 -r-- ld-2.27.so..gnu.version_d
08 0x00000b30  1152 0x4652db30  1152 -r-- ld-2.27.so..rela.dyn
09 0x00000fb0 11497 0x4652dfb0 11497 -r-x ld-2.27.so..text
10 0x0001d0e0 17760 0x4654a0e0 17760 -r-- ld-2.27.so..rodata
11 0x00021640  1716 0x4654e640  1716 -r-- ld-2.27.so..eh_frame_hdr
12 0x00021cf8  9876 0x4654ecf8  9876 -r-- ld-2.27.so..eh_frame
13 0x00024660  2020 0x46751660  2020 -rw- ld-2.27.so..data.rel.ro
14 0x00024e48   336 0x46751e48   336 -rw- ld-2.27.so..dynamic
15 0x00024f98    96 0x46751f98    96 -rw- ld-2.27.so..got
16 0x00025000  3960 0x46752000  3960 -rw- ld-2.27.so..data
17 0x00025f78     0 0x46752f80   376 -rw- ld-2.27.so..bss
18 0x00025f78    17 0x00000000    17 ---- ld-2.27.so..comment
19 0x00025fa0    63 0x00000000    63 ---- ld-2.27.so..gnu.warning.llseek
20 0x00025fe0 13272 0x00000000 13272 ---- ld-2.27.so..symtab
21 0x000293b8  7101 0x00000000  7101 ---- ld-2.27.so..strtab
22 0x0002af75   215 0x00000000   215 ---- ld-2.27.so..shstrtab
```

Heap

radare2's `dm` subcommands can also display a map of the heap which is useful for those who are interested in inspecting the heap and its content. Simply execute `dmh` to show a map of the heap:

```
[0x7fae46236ca6]> dmh
  Malloc chunk @ 0x55a7ecbce250 [size: 0x411][allocated]
  Top chunk @ 0x55a7ecbce660 - [brk_start: 0x55a7ecbce000, brk_end: 0x55a7ecbef000]
```

You can also see a graph layout of the heap:

```
[0x7fae46236ca6]> dmhg
Heap Layout
    .                    .
        Malloc chunk @ 0x55a7ecbce000
      size: 0x251
       fd: 0x0, bk: 0x0
    `                    '
```

```
    . '



  .                        .
      Malloc chunk @ 0x55a7ecbce250
   size: 0x411
    fd: 0x57202c6f6c6c6548, bk: 0xa21646c726f
  `                        '

  . '



.                            .
  Top chunk @ 0x55a7ecbce660
 [brk_start:0x55a7ecbce000, brk_end:0x55a7ecbef000]
`                            '
```

Another heap commands can be found under `dmh`, check `dmh?` for the full list.

```
[0x00000000]> dmh?
|Usage:  dmh # Memory map heap
| dmh                 List chunks in heap segment
| dmh [malloc_state]  List heap chunks of a particular arena
| dmha                List all malloc_state instances in application
| dmhb                Display all parsed Double linked list of main_arena's bins instance
| dmhb [bin_num|bin_num:malloc_state]          Display parsed double linked list of bins instance from a particular arena
| dmhbg [bin_num]     Display double linked list graph of main_arena's bin [Under developemnt]
| dmhc @[chunk_addr]  Display malloc_chunk struct for a given malloc chunk
| dmhf                Display all parsed fastbins of main_arena's fastbinY instance
| dmhf [fastbin_num|fastbin_num:malloc_state]  Display parsed single linked list in fastbinY instance from a particular arena
| dmhg                Display heap graph of heap segment
| dmhg [malloc_state] Display heap graph of a particular arena
| dmhi @[malloc_state]Display heap_info structure/structures for a given arena
| dmhm                List all elements of struct malloc_state of main thread (main_arena)
| dmhm [malloc_state] List all malloc_state instance of a particular arena
| dmht                Display all parsed thead cache bins of main_arena's tcache instance
| dmh?                Show map heap help
```

To print safe-linked lists (glibc >= 2.32) with demangled pointers, the variable `dbg.glibc.demangle` must be true.

Files

The radare2 debugger allows the user to list and manipulate the file descriptors from the target process.

This is a useful feature, which is not found in other debuggers, the functionality is similar to the lsof command line tool, but have extra subcommands to change the seek, close or duplicate them.

So, at any time in the debugging session you can replace the stdio file descriptors to use network sockets created by r2, or replace a network socket connection to hijack it.

This functionality is also available in r2frida by using the dd command prefixed with a backslash. In r2 you may want to see the output of dd? for proper details.

Reverse Debugging

Radare2 has reverse debugger, that can seek the program counter backward. (e.g. reverse-next, reverse-continue in gdb) Firstly you need to save program state at the point that you want to start recording. The syntax for recording is:

```
[0x004028a0]> dts+
```

You can use `dts` commands for recording and managing program states. After recording the states, you can seek pc back and forth to any points after saved address. So after recording, you can try single step back:

```
[0x004028a0]> 2dso
[0x004028a0]> dr rip
0x004028ae
[0x004028a0]> dsb
continue until 0x004028a2
hit breakpoint at: 4028a2
[0x004028a0]> dr rip
0x004028a2
```

When you run `dsb`, reverse debugger restore previous recorded state and execute program from it until desired point.

Or you can also try continue back:

```
[0x004028a0]> db 0x004028a2
[0x004028a0]> 10dso
[0x004028a0]> dr rip
0x004028b9
[0x004028a0]> dcb
[0x004028a0]> dr rip
0x004028a2
```

`dcb` seeks program counter until hit the latest breakpoint. So once set a breakpoint, you can back to it any time.

You can see current recorded program states using `dts`:

```
[0x004028a0]> dts
session: 0   at:0x004028a0   ""
session: 1   at:0x004028c2   ""
```

NOTE: Program records can be saved at any moments. These are diff style format that save only different memory area from previous. It saves memory space rather than entire dump.

And also can add comment:

```
[0x004028c2]> dtsC 0 program start
[0x004028c2]> dtsC 1 decryption start
[0x004028c2]> dts
session: 0   at:0x004028a0   "program start"
session: 1   at:0x004028c2   "decryption start"
```

You can leave notes for each records to keep in your mind. `dsb` and `dcb` commands restore the program state from latest record if there are many records.

Program records can exported to file and of course import it. Export/Import records to/from file:

```
[0x004028c2]> dtst records_for_test
Session saved in records_for_test.session and dump in records_for_test.dump
[0x004028c2]> dtsf records_for_test
session: 0, 0x4028a0 diffs: 0
session: 1, 0x4028c2 diffs: 0
```

Moreover, you can do reverse debugging in ESIL mode. In ESIL mode, program state can be managed by `aets` commands.

```
[0x00404870]> aets+
```

And step back by `aesb`:

```
[0x00404870]> aer rip
0x00404870
[0x00404870]> 5aeso
[0x00404870]> aer rip
0x0040487d
[0x00404870]> aesb
[0x00404870]> aer rip
0x00404879
```

In addition to the native reverse debugging capabilities in radare2, it's also possible to use gdb's remote protocol to reverse debug a target gdbserver that supports it. `=!dsb` and `=!dcb` are available as `dsb` and `dcb` replacementments for this purpose, see remote gdb's documentation for more information.

Windows Messages

On Windows, you can use `dbW` while debugging to set a breakpoint for the message handler of a specific window.

Get a list of the current process windows with `dW` :

```
[0x7ffe885c1164]> dW
.----------------------------------------------------.
| Handle      | PID   | TID    | Class Name         |
)----------------------------------------------------(
| 0x0023038e  | 9432  | 22432  | MSCTFIME UI        |
| 0x0029049e  | 9432  | 22432  | IME                |
| 0x002c048a  | 9432  | 22432  | Edit               |
| 0x000d0474  | 9432  | 22432  | msctls_statusbar32 |
| 0x00070bd6  | 9432  | 22432  | Notepad            |
`----------------------------------------------------'
```

Set the breakpoint with a message type, together with either the window class name or its handle:

```
[0x7ffe885c1164]> dbW WM_KEYDOWN Edit
Breakpoint set.
```

Or

```
[0x7ffe885c1164]> dbW WM_KEYDOWN 0x002c048a
Breakpoint set.
```

If you aren't sure which window you should put a breakpoint on, use `dWi` to identify it with your mouse:

```
[0x7ffe885c1164]> dWi
Move cursor to the window to be identified. Ready? y
Try to get the child? y
.--------------------------------------------.
| Handle      | PID   | TID    | Class Name  |
)--------------------------------------------(
| 0x002c048a  | 9432  | 22432  | Edit        |
`--------------------------------------------'
```

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:  =[:!+-=ghH] [...]   # connect with other instances of r2

remote 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:port        connect to remote host:port (*rap://, raps://, tcp://, udp://, http://)
| =-[fd]                        remove all hosts or host 'fd'
| ==[fd]                        open remote session with host 'fd', 'q' to quit
| =!=                           disable remote cmd mode
| !=!                           enable remote cmd mode

servers:
| .:9000                        start the tcp server (echo x|nc ::1 9090 or curl ::1:9090/cmd/x)
| =:port                        start the rap server (o rap://9999)
| =g[?]                         start the gdbserver
| =h[?]                         start the http webserver
| =H[?]                         start the http webserver (and launch the web browser)

other:
| =&:port                       start rap server in background (same as '&_=h')
| =:host:port cmd               run 'cmd' command on remote server

examples:
| =+tcp://localhost:9090/       connect to: r2 -c.:9090 ./bin
```

```
| =+rap://localhost:9090/       connect to: r2 rap://:9090
| =+http://localhost:9090/cmd/  connect to: r2 -c'=h 9090' bin
| o rap://:9090/                start the rap server on tcp port 9090
```

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).

Debugging with gdbserver

radare2 allows remote debugging over the gdb remote protocol. So you can run a gdbserver and connect to it with radare2 for remote debugging. The syntax for connecting is:

```
$ r2 -d gdb://<host>:<port>
```

Note that the following command does the same, r2 will use the debug plugin specified by the uri if found.

```
$ r2 -D gdb gdb://<host>:<port>
```

The debug plugin can be changed at runtime using the dL or Ld commands.

Or if the gdbserver is running in extended mode, you can attach to a process on the host with:

```
$ r2 -d gdb://<host>:<port>/<pid>
```

It is also possible to start debugging after analyzing a file using the `doof` command which rebases the current session's data after opening gdb

```
[0x00404870]> doof gdb://<host>:<port>/<pid>
```

After connecting, you can use the standard r2 debug commands as normal.

radare2 does not yet load symbols from gdbserver, so it needs the binary to be locally present to load symbols from it. In case symbols are not loaded even if the binary is present, you can try specifying the path with `e dbg.exe.path`:

```
$ r2 -e dbg.exe.path=<path> -d gdb://<host>:<port>
```

If symbols are loaded at an incorrect base address, you can try specifying the base address too with `e bin.baddr`:

```
$ r2 -e bin.baddr=<baddr> -e dbg.exe.path=<path> -d gdb://<host>:<port>
```

Usually the gdbserver reports the maximum packet size it supports. Otherwise, radare2 resorts to sensible defaults. But you can specify the maximum packet size with the environment variable `R2_GDB_PKTSZ`. You can also check and set the max packet size during a session with the IO system, `=!`.

```
$ export R2_GDB_PKTSZ=512
$ r2 -d gdb://<host>:<port>
= attach <pid> <tid>
Assuming filepath <path/to/exe>
[0x7ff659d9fcc0]> =!pktsz
packet size: 512 bytes
[0x7ff659d9fcc0]> =!pktsz 64
[0x7ff659d9fcc0]> =!pktsz
packet size: 64 bytes
```

The gdb IO system provides useful commands which might not fit into any standard radare2 commands. You can get a list of these commands with `=!?`. (Remember, `=!` accesses the underlying IO plugin's `system()`).

```
[0x7ff659d9fcc0]> =!?
Usage: =!cmd args
 =!pid             - show targeted pid
 =!pkt s           - send packet 's'
 =!monitor cmd     - hex-encode monitor command and pass to target interpreter
 =!rd              - show reverse debugging availability
 =!dsb             - step backwards
 =!dcb             - continue backwards
 =!detach [pid]    - detach from remote/detach specific pid
 =!inv.reg         - invalidate reg cache
 =!pktsz           - get max packet size used
 =!pktsz bytes     - set max. packet size as 'bytes' bytes
 =!exec_file [pid] - get file which was executed for current/specified pid
```

Note that `=!dsb` and `=!dcb` are only available in special gdbserver implementations such as Mozilla's rr, the default gdbserver doesn't include remote reverse debugging support. Use `=!rd` to print the currently available reverse debugging capabilities.

If you are interested in debugging radare2's interaction with gdbserver you can use `=!monitor set remote-debug 1` to turn on logging of gdb's remote protocol packets in gdbserver's console and `=!monitor set debug 1` to show general debug messages from gdbserver in it's console.

radare2 also provides its own gdbserver implementation:

```
$ r2 -
[0x00000000]> =g?
|Usage:  =[g] [...] # gdb server
| gdbserver:
| =g port file [args]   listen on 'port' debugging 'file' using gdbserver
| =g! port file [args]  same as above, but debug protocol messages (like gdbserver --remote-debug)
```

So you can start it as:

```
$ r2 -
[0x00000000]> =g 8000 /bin/radare2 -
```

And then connect to it like you would to any gdbserver. For example, with radare2:

```
$ r2 -d gdb://localhost:8000
```

WinDBG Kernel-mode Debugging (KD)

The WinDBG KD interface support for r2 allows you to attach to VM running Windows and debug its kernel over a serial port or network.

It is also possible to use the remote GDB interface to connect and debug Windows kernels without depending on Windows capabilities.

Bear in mind that WinDBG KD support is still work-in-progress, and this is just an initial implementation which will get better in time.

Setting Up KD on Windows

For a complete walkthrough, refer to Microsoft's documentation.

Serial Port

Enable KD over a serial port on Windows Vista and higher like this:

```
bcdedit /debug on
bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

Or like this for Windows XP: Open boot.ini and add /debug /debugport=COM1 /baudrate=115200:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Debugging with Cable" /fastdetect /debug /debugport=COM1 /baudrate=57600
```

In case of VMWare

```
    Virtual Machine Settings -> Add -> Serial Port
    Device Status:
    [v] Connect at power on
    Connection:
    [v] Use socket (named pipe)
    [_/tmp/winkd.pipe________]
    From: Server To: Virtual Machine
```

Configure the VirtualBox Machine like this:

```
    Preferences -> Serial Ports -> Port 1

    [v] Enable Serial Port
    Port Number: [_COM1_______[v]]
    Port Mode:   [_Host_Pipe__[v]]
                 [v] Create Pipe
    Port/File Path: [_/tmp/winkd.pipe____]
```

Or just spawn the VM with qemu like this:

```
$ qemu-system-x86_64 -chardev socket,id=serial0,\
     path=/tmp/winkd.pipe,nowait,server \
     -serial chardev:serial0 -hda Windows7-VM.vdi
```

Network

Enable KD over network (KDNet) on Windows 7 or later likes this:

```
bcdedit /debug on
bcdedit /dbgsettings net hostip:w.x.y.z port:n
```

Starting from Windows 8 there is no way to enforce debugging for every boot, but it is possible to always show the advanced boot options, which allows to enable kernel debugging:

```
bcedit /set {globalsettings} advancedoptions true
```

Connecting to KD interface on r2

Serial Port

Radare2 will use the `winkd` io plugin to connect to a socket file created by virtualbox or qemu. Also, the `winkd` debugger plugin and we should specify the x86-32 too. (32 and 64 bit debugging is supported)

```
$ r2 -a x86 -b 32 -D winkd winkd:///tmp/winkd.pipe
```

On Windows you should run the following line:

```
$ radare2 -D winkd winkd://\\.\pipe\com_1
```

Network

```
$ r2 -a x86 -b 32 -d winkd://<hostip>:<port>:w.x.y.z
```

Using KD

When connecting to a KD interface, r2 will send a breakin packet to interrupt the target and we will get stuck here:

```
[0x828997b8]> pd 20
    ;-- eip:
    0x828997b8    cc           int3
    0x828997b9    c20400       ret 4
    0x828997bc    cc           int3
    0x828997bd    90           nop
    0x828997be    c3           ret
    0x828997bf    90           nop
```

In order to skip that trap we will need to change eip and run 'dc' twice:

```
dr eip=eip+1
dc
dr eip=eip+1
dc
```

Now the Windows VM will be interactive again. We will need to kill r2 and attach again to get back to control the kernel.

In addition, the `dp` command can be used to list all processes, and `dpa` or `dp=` to attach to the process. This will display the base address of the process in the physical memory layout.

WinDBG Backend for Windows (DbgEng)

On Windows, radare2 can use `DbgEng.dll` as a debugging backend, allowing it to make use of WinDBG's capabilities, supporting dump files, local and remote user and kernel mode debugging.

You can use the debugging DLLs included on Windows or get the latest version from Microsoft's download page (recommended).

> You cannot use DLLs from the Microsoft Store's `WinDbg Preview` app folder directly as they are not marked as executable for normal users.
>
> radare2 will try to load `dbgeng.dll` from the `_NT_DEBUGGER_EXTENSION_PATH` environment variable before using Windows' default library search path.

Using the plugin

To use the `windbg` plugin, pass the same command-line options as you would for `WinDBG` or `kd` (see Microsoft's documentation), quoting/escaping when necessary:

```
> r2 -d "windbg://-remote tcp:server=Server,port=Socket"
```

```
> r2 -d "windbg://MyProgram.exe \"my arg\""
```

```
> r2 -d "windbg://-k net:port=<n>,key=<MyKey>"
```

```
> r2 -d "windbg://-z MyDumpFile.dmp"
```

You can then debug normally (see `d?` command) or interact with the backend shell directly with the `=!` command:

```
[0x7ffcac9fcea0]> dcu 0x0007ffc98f42190
Continue until 0x7ffc98f42190 using 1 bpsize
ModLoad: 00007ffc`ab6b0000 00007ffc`ab6e0000   C:\WINDOWS\System32\IMM32.DLL
Breakpoint 1 hit
hit breakpoint at: 0x7ffc98f42190

[0x7fffcf232190]> =!k4
Child-SP          RetAddr           Call Site
00000033`73b1f618 00007ff6`c67a861d r_main!r_main_radare2
00000033`73b1f620 00007ff6`c67d0019 radare2!main+0x8d
00000033`73b1f720 00007ff6`c67cfebe radare2!invoke_main+0x39
00000033`73b1f770 00007ff6`c67cfd7e radare2!__scrt_common_main_seh+0x12e
```

Tools

Radare2 is not just the only tool provided by the radare2 project. The rest if chapters in this book are focused on explaining the use of the radare2 tool, this chapter will focus on explaining all the other companion tools that are shipped inside the radare2 project.

All the functionalities provided by the different APIs and plugins have also different tools to allow to use them from the commandline and integrate them with shellscripts easily.

Thanks to the ortogonal design of the framework it is possible to do all the things that r2 is able from different places:

- these companion tools
- native library apis
- scripting with r2pipe
- the r2 shell ## 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.

This is the help message of rax2, this tool can be used in the command-line or interactively (reading the values from stdin), so it can be used as a multi-base calculator.

Inside r2, the functionality of rax2 is available under the ? command. For example:

```
[0x00000000]> ? 3+4
```

As you can see, the numeric expressions can contain mathematical expressions like addition, substraction, .. as well as group operations with parenthesis.

The syntax in which the numbers are represented define the base, for example:

- 3 : decimal, base 10
- 0xface : hexadecimal, base 16
- 0472 : octal, base 8
- 2M : units, 2 megabytes
- …

This is the help message of rax2 -h, which will show you a bunch more syntaxes

```
$ rax2 -h
Usage: rax2 [options] [expr ...]
  =[base]                      ;  rax2 =10 0x46 -> output in base 10
  int     ->  hex              ;  rax2 10
  hex     ->  int              ;  rax2 0xa
  -int    ->  hex              ;  rax2 -77
  -hex    ->  int              ;  rax2 0xffffffb3
  int     ->  bin              ;  rax2 b30
  int     ->  ternary          ;  rax2 t42
  bin     ->  int              ;  rax2 1010d
  ternary ->  int              ;  rax2 1010dt
  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
  ternary ->  hex              ;  rax2 212t
  hex     ->  ternary          ;  rax2 Tx23
  raw     ->  hex              ;  rax2 -S < /binfile
  hex     ->  raw              ;  rax2 -s 414141
  -l                           ;  append newline to output (for -E/-D/-r/..
  -a      show ascii table     ;  rax2 -a
  -b      bin -> str           ;  rax2 -b 01000101 01110110
  -B      str -> bin           ;  rax2 -B hello
  -d      force integer        ;  rax2 -d 3 -> 3 instead of 0x3
  -e      swap endianness      ;  rax2 -e 0x33
  -D      base64 decode        ;
  -E      base64 encode        ;
  -f      floating point       ;  rax2 -f 6.3+2.1
  -F      stdin slurp code hex ;  rax2 -F < shellcode.[c/py/js]
  -h      help                 ;  rax2 -h
  -i      dump as C byte array ;  rax2 -i < bytes
  -k      keep base            ;  rax2 -k 33+3 -> 36
  -K      randomart            ;  rax2 -K 0x34 1020304050
  -L      bin -> hex(bignum)   ;  rax2 -L 111111111 # 0x1ff
  -n      binary number        ;  rax2 -n 0x1234 # 34120000
  -N      binary number        ;  rax2 -N 0x1234 # \x34\x12\x00\x00
  -r      r2 style output      ;  rax2 -r 0x1234
  -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
  -w      signed word          ;  rax2 -w 16 0xffff
  -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        |
|                 |
|                 |
|                 |
|                 |
+-----------------+
```

rafind2

Rafind2 is the command line fronted of the `r_search` library. Which allows you to search for strings, sequences of bytes with binary masks, etc

```
$ rafind2 -h
Usage: rafind2 [-mXnzZhqv] [-a align] [-b sz] [-f/t from/to] [-[e|s|S] str] [-x hex] -|file|dir ..
 -a [align] only accept aligned hits
 -b [size]  set block size
 -e [regex] search for regex matches (can be used multiple times)
 -f [from]  start searching from address 'from'
 -h         show this help
 -i         identify filetype (r2 -nqcpm file)
 -j         output in JSON
 -m         magic search, file-type carver
 -M [str]   set a binary mask to be applied on keywords
 -n         do not stop on read errors
 -r         print using radare commands
 -s [str]   search for a specific string (can be used multiple times)
 -S [str]   search for a specific wide string (can be used multiple times). Assumes str is UTF-8.
 -t [to]    stop search at address 'to'
 -q         quiet - do not show headings (filenames) above matching contents (default for searching a single file)
 -v         print version and exit
 -x [hex]   search for hexpair string (909090) (can be used multiple times)
 -X         show hexdump of search results
 -z         search for zero-terminated strings
 -Z         show string found on each search hit
```

That's how to use it, first we'll search for "lib" inside the `/bin/ls` binary.

```
$ rafind2 -s lib /bin/ls
0x5f9
0x675
0x679
...
$
```

Note that the output is pretty minimal, and shows the offsets where the string `lib` is found. We can then use this output to feed other tools.

Counting results:

```
$ rafind2 -s lib /bin/ls | wc -l
```

Displaying results with context:

```
$ export F=/bin/ls
$ for a in `rafind2 -s lib $F` ; do \
    r2 -ns $a -qc'x 32' $F ; done
0x000005f9  6c69 622f 6479 6c64 .. lib/dyld........
0x00000675  6c69 622f 6c69 6275 .. lib/libutil.dyli
0x00000679  6c69 6275 7469 6c2e .. libutil.dylib...
0x00000683  6c69 6200 000c 0000 .. lib......8......
0x000006a5  6c69 622f 6c69 626e .. lib/libncurses.5
0x000006a9  6c69 626e 6375 7273 .. libncurses.5.4.d
0x000006ba  6c69 6200 0000 0c00 .. lib.......8.....
0x000006dd  6c69 622f 6c69 6253 .. lib/libSystem.B.
0x000006e1  6c69 6253 7973 7465 .. libSystem.B.dyli
0x000006ef  6c69 6200 0000 0000 .. lib......&......
```

rafind2 can also be used as a replacement of `file` to identify the mimetype of a file using the internal magic database of radare2.

```
$ rafind2 -i /bin/ls
0x00000000 1 Mach-O
```

Also works as a `strings` replacement, similar to what you do with rabin2 -z, but without caring about parsing headers and obeying binary sections.

```
$ rafind2 -z /bin/ls| grep http
0x000076e5 %http://www.apple.com/appleca/root.crl0\r
0x00007ae6 https://www.apple.com/appleca/0
0x00007fa9 )http://www.apple.com/certificateauthority0
0x000080ab $http://crl.apple.com/codesigning.crl0
```

Rarun2

Rarun2 is a tool allowing to setup a specified execution environment - redefine stdin/stdout, pipes, change the environment variables and other settings useful to craft the boundary conditions you need to run a binary for debugging.

```
$ rarun2 -h
Usage: rarun2 -v|-t|script.rr2 [directive ..]
```

It takes the text file in key=value format to specify the execution environment. Rarun2 can be used as both separate tool or as a part of radare2. To load the rarun2 profile in radare2 you need to use either `-r` to load the profile from file or `-R` to specify the directive from string.

The format of the profile is very simple. Note the most important keys - `program` and `arg*`

One of the most common usage cases - redirect the output of debugged program in radare2. For this you need to use `stdio`, `stdout`, `stdin`, `input`, and a couple similar keys.

Here is the basic profile example:

```
program=/bin/ls
arg1=/bin
# arg2=hello
# arg3="hello\nworld"
# arg4=:048490184058104849
# arg5=:!ragg2 -p n50 -d 10:0x8048123
# arg6=@arg.txt
# arg7=@300@ABCD # 300 chars filled with ABCD pattern
# system=r2 -
# aslr=no
setenv=FOO=BAR
# unsetenv=FOO
# clearenv=true
```

```
# envfile=environ.txt
timeout=3
# timeoutsig=SIGTERM # or 15
# connect=localhost:8080
# listen=8080
# pty=false
# fork=true
# bits=32
# pid=0
# pidfile=/tmp/foo.pid
# #sleep=0
# #maxfd=0
# #execve=false
# #maxproc=0
# #maxstack=0
# #core=false
# #stdio=blah.txt
# #stderr=foo.txt
# stdout=foo.txt
# stdin=input.txt # or !program to redirect input from another program
# input=input.txt
# chdir=/
# chroot=/mnt/chroot
# libpath=$PWD:/tmp/lib
# r2preload=yes
# preload=/lib/libfoo.so
# setuid=2000
# seteuid=2000
# setgid=2001
# setegid=2001
# nice=5
```

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 and other data. 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 or any format supported by plugins, 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 [-AcdeEghHiIjlLMqrRsSvVxzZ] [-@ at] [-a arch] [-b bits] [-B addr]
              [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-P[-P] pdb]
              [-o str] [-O str] [-k query] [-D lang symname] | file
 -@ [addr]       show section, symbol or import at addr
 -A              list sub-binaries and their arch-bits pairs
 -a [arch]       set arch (x86, arm, .. or <arch>_<bits>)
 -b [bits]       set bits (32, 64 ...)
 -B [addr]       override base address (pie bins)
 -c              list classes
 -C [fmt:C:D]    create [elf,mach0,pe] with Code and Data hexpairs (see -a)
 -d              show debug/dwarf information
 -D lang name    demangle symbol name (-D all for bin.demangle=true)
 -e              entrypoint
 -E              globally exportable symbols
 -f [str]        select sub-bin named str
 -F [binfmt]     force to use that bin plugin (ignore header check)
 -g              same as -SMZIHVResizcld (show all info)
 -G [addr]       load address . offset to header
 -h              this help message
 -H              header fields
 -i              imports (symbols imported from libraries)
 -I              binary info
 -j              output in json
 -k [sdb-query]  run sdb query. for example: '*'
 -K [algo]       calculate checksums (md5, sha1, ..)
 -l              linked libraries
 -L [plugin]     list supported bin plugins or plugin details
 -m [addr]       show source line at addr
 -M              main (show address of main symbol)
 -n [str]        show section, symbol or import named str
 -N [min:max]    force min:max number of chars per string (see -z and -zz)
 -o [str]        output file/folder for write operations (out by default)
 -O [str]        write/extract operations (-O help)
 -p              show physical addresses
 -P              show debug/pdb information
 -PP             download pdb file for binary
 -q              be quiet, just show fewer data
 -qq             show less info (no offset/size for -z for ex.)
 -Q              show load address used by dlopen (non-aslr libs)
 -r              radare output
 -R              relocations
 -s              symbols
 -S              sections
 -u              unfiltered (no rename duplicated symbols/sections)
 -v              display version and quit
 -V              Show binary version information
 -x              extract bins contained in file
 -X [fmt] [f] .. package in fat or zip the given files and bins contained in file
 -z              strings (from data section)
 -zz             strings (from raw bins [e bin.rawstr=1])
 -zzz            dump raw strings to stdout (for huge files)
 -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 type, like its encoding, endianness, class, operating system:

```
$ rabin2 -I /bin/ls
arch     x86
binsz    128456
bintype  elf
bits     64
canary   true
class    ELF64
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
lang     c
linenum  false
lsyms    false
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      true
relocs   false
relro    partial
rpath    NONE
static   false
stripped true
subsys   linux
va       true
```

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

```
$ rabin2 -Ir /bin/ls
e cfg.bigendian=false
e asm.bits=64
e asm.dwarf=true
e bin.lang=c
e file.type=elf
e asm.os=linux
e asm.arch=x86
e asm.pcalign=0
```

Code Entrypoints

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

```
$ rabin2 -e /bin/ls
[Entrypoints]
vaddr=0x00005310 paddr=0x00005310 baddr=0x00000000 laddr=0x00000000 haddr=0x00000018 type=program

1 entrypoints

$ rabin2 -er /bin/ls
fs symbols
f entry0 1 @ 0x00005310
f entry0_haddr 1 @ 0x00000018
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 rabin2 to get a list of imports. An example:

```
$ rabin2 -i /bin/ls
[Imports]
nth vaddr      bind   type   lib name
-------------------------------------
   1 0x000032e0  GLOBAL    FUNC __ctype_toupper_loc
   2 0x000032f0  GLOBAL    FUNC getenv
   3 0x00003300  GLOBAL    FUNC sigprocmask
   4 0x00003310  GLOBAL    FUNC __snprintf_chk
   5 0x00003320  GLOBAL    FUNC raise
   6 0x00000000  GLOBAL    FUNC free
   7 0x00003330  GLOBAL    FUNC abort
   8 0x00003340  GLOBAL    FUNC __errno_location
   9 0x00003350  GLOBAL    FUNC strncmp
  10 0x00000000    WEAK  NOTYPE _ITM_deregisterTMCloneTable
  11 0x00003360  GLOBAL    FUNC localtime_r
  12 0x00003370  GLOBAL    FUNC _exit
  13 0x00003380  GLOBAL    FUNC strcpy
  14 0x00003390  GLOBAL    FUNC __fpending
  15 0x000033a0  GLOBAL    FUNC isatty
  16 0x000033b0  GLOBAL    FUNC sigaction
  17 0x000033c0  GLOBAL    FUNC iswcntrl
  18 0x000033d0  GLOBAL    FUNC wcswidth
  19 0x000033e0  GLOBAL    FUNC localeconv
  20 0x000033f0  GLOBAL    FUNC mbstowcs
  21 0x00003400  GLOBAL    FUNC readlink
...
```

Exports

Rabin2 is able to find exports. For example:

```
$ rabin2 -E /usr/lib/libr_bin.so | head
[Exports]

nth  paddr       vaddr      bind   type size lib name
-----------------------------------------------------
210 0x000ae1f0 0x000ae1f0 GLOBAL   FUNC  200 r_bin_java_print_exceptions_attr_summary
211 0x000afc90 0x000afc90 GLOBAL   FUNC  135 r_bin_java_get_args
212 0x000b18e0 0x000b18e0 GLOBAL   FUNC   35 r_bin_java_get_item_desc_from_bin_cp_list
213 0x00022d90 0x00022d90 GLOBAL   FUNC  204 r_bin_class_add_method
214 0x000ae600 0x000ae600 GLOBAL   FUNC  175 r_bin_java_print_fieldref_cp_summary
215 0x000ad880 0x000ad880 GLOBAL   FUNC  144 r_bin_java_print_constant_value_attr_summary
216 0x000b7330 0x000b7330 GLOBAL   FUNC  679 r_bin_java_print_element_value_summary
217 0x000af170 0x000af170 GLOBAL   FUNC   65 r_bin_java_create_method_fq_str
218 0x00079b00 0x00079b00 GLOBAL   FUNC   15 LZ4_createStreamDecode
```

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]

nth paddr       vaddr      bind   type   size lib name
------------------------------------------------------
110 0x000150a0 0x000150a0 GLOBAL FUNC 56 _obstack_allocated_p
111 0x0001f600 0x0021f600 GLOBAL  OBJ  8 program_name
112 0x0001f620 0x0021f620 GLOBAL  OBJ  8 stderr
113 0x00014f90 0x00014f90 GLOBAL FUNC 21 _obstack_begin_1
114 0x0001f600 0x0021f600   WEAK  OBJ  8 program_invocation_name
115 0x0001f5c0 0x0021f5c0 GLOBAL  OBJ  8 alloc_failed_handler
116 0x0001f5f8 0x0021f5f8 GLOBAL  OBJ  8 optarg
117 0x0001f5e8 0x0021f5e8 GLOBAL  OBJ  8 stdout
118 0x0001f5e0 0x0021f5e0 GLOBAL  OBJ  8 program_short_name
```

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 | head
fs symbols
f sym.obstack_allocated_p 56 0x000150a0
f sym.program_invocation_name 8 0x0021f600
f sym.stderr 8 0x0021f620
f sym.obstack_begin_1 21 0x00014f90
f sym.program_invocation_name 8 0x0021f600
f sym.obstack_alloc_failed_handler 8 0x0021f5c0
f sym.optarg 8 0x0021f5f8
f sym.stdout 8 0x0021f5e8
f sym.program_invocation_short_name 8 0x0021f5e0
```

List Libraries

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

```
$ rabin2 -l `which r2`
[Linked libraries]
libr_core.so
libr_parse.so
libr_search.so
libr_cons.so
libr_config.so
libr_bin.so
libr_debug.so
libr_anal.so
libr_reg.so
libr_bp.so
libr_io.so
libr_fs.so
libr_asm.so
libr_syscall.so
libr_hash.so
libr_magic.so
libr_flag.so
libr_egg.so
libr_crypto.so
libr_util.so
libpthread.so.0
libc.so.6

22 libraries
```

Lets check the output with `ldd` command:

```
$ ldd `which r2`
linux-vdso.so.1 (0x00007fffba38e000)
libr_core.so => /usr/lib64/libr_core.so (0x00007f94b4678000)
libr_parse.so => /usr/lib64/libr_parse.so (0x00007f94b4425000)
libr_search.so => /usr/lib64/libr_search.so (0x00007f94b421f000)
libr_cons.so => /usr/lib64/libr_cons.so (0x00007f94b4000000)
libr_config.so => /usr/lib64/libr_config.so (0x00007f94b3dfa000)
libr_bin.so => /usr/lib64/libr_bin.so (0x00007f94b3afd000)
libr_debug.so => /usr/lib64/libr_debug.so (0x00007f94b38d2000)
libr_anal.so => /usr/lib64/libr_anal.so (0x00007f94b2fbd000)
libr_reg.so => /usr/lib64/libr_reg.so (0x00007f94b2db4000)
libr_bp.so => /usr/lib64/libr_bp.so (0x00007f94b2baf000)
libr_io.so => /usr/lib64/libr_io.so (0x00007f94b2944000)
```

```
libr_fs.so => /usr/lib64/libr_fs.so (0x00007f94b270e000)
libr_asm.so => /usr/lib64/libr_asm.so (0x00007f94b1c69000)
libr_syscall.so => /usr/lib64/libr_syscall.so (0x00007f94b1a63000)
libr_hash.so => /usr/lib64/libr_hash.so (0x00007f94b185a000)
libr_magic.so => /usr/lib64/libr_magic.so (0x00007f94b164d000)
libr_flag.so => /usr/lib64/libr_flag.so (0x00007f94b1446000)
libr_egg.so => /usr/lib64/libr_egg.so (0x00007f94b1236000)
libr_crypto.so => /usr/lib64/libr_crypto.so (0x00007f94b1016000)
libr_util.so => /usr/lib64/libr_util.so (0x00007f94b0d35000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f94b0b15000)
libc.so.6 => /lib64/libc.so.6 (0x00007f94b074d000)
libr_lang.so => /usr/lib64/libr_lang.so (0x00007f94b0546000)
libr_socket.so => /usr/lib64/libr_socket.so (0x00007f94b0339000)
libm.so.6 => /lib64/libm.so.6 (0x00007f94affaf000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f94afdab000)
/lib64/ld-linux-x86-64.so.2 (0x00007f94b4c79000)
libssl.so.1.0.0 => /usr/lib64/libssl.so.1.0.0 (0x00007f94afb3c000)
libcrypto.so.1.0.0 => /usr/lib64/libcrypto.so.1.0.0 (0x00007f94af702000)
libutil.so.1 => /lib64/libutil.so.1 (0x00007f94af4ff000)
libz.so.1 => /lib64/libz.so.1 (0x00007f94af2e8000)
```

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
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
000 0x000160f8 0x000160f8  11  12 (.rodata) ascii dev_ino_pop
001 0x00016188 0x00016188  10  11 (.rodata) ascii sort_files
002 0x00016193 0x00016193   6   7 (.rodata) ascii posix-
003 0x0001619a 0x0001619a   4   5 (.rodata) ascii main
004 0x00016250 0x00016250  10  11 (.rodata) ascii ?pcdb-lswd
005 0x00016260 0x00016260  65  66 (.rodata) ascii # Configuration file for dircolors, a utility to help you set the
006 0x000162a2 0x000162a2  72  73 (.rodata) ascii # LS_COLORS environment variable used by GNU ls with the --color option.
007 0x000162eb 0x000162eb  56  57 (.rodata) ascii # Copyright (C) 1996-2018 Free Software Foundation, Inc.
008 0x00016324 0x00016324  70  71 (.rodata) ascii # Copying and distribution of this file, with or without modification,
009 0x0001636b 0x0001636b  76  77 (.rodata) ascii # are permitted provided the copyright notice and this notice are preserved.
```

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 stringsf str.dev_ino_pop 12 @ 0x000160f8
Cs 12 @ 0x000160f8
f str.sort_files 11 @ 0x00016188
Cs 11 @ 0x00016188
f str.posix 7 @ 0x00016193
Cs 7 @ 0x00016193
f str.main 5 @ 0x0001619a
Cs 5 @ 0x0001619a
f str.pcdb_lswd 11 @ 0x00016250
Cs 11 @ 0x00016250
```

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]

nth paddr          size vaddr         vsize perm name
-----------------------------------------------------
00 0x00000000     0 0x00000000     0 ----
01 0x00000238    28 0x00000238    28 -r-- .interp
02 0x00000254    32 0x00000254    32 -r-- .note.ABI_tag
03 0x00000278   176 0x00000278   176 -r-- .gnu.hash
04 0x00000328  3000 0x00000328  3000 -r-- .dynsym
05 0x00000ee0  1412 0x00000ee0  1412 -r-- .dynstr
06 0x00001464   250 0x00001464   250 -r-- .gnu.version
07 0x00001560   112 0x00001560   112 -r-- .gnu.version_r
08 0x000015d0  4944 0x000015d0  4944 -r-- .rela.dyn
09 0x00002920  2448 0x00002920  2448 -r-- .rela.plt
10 0x000032b0    23 0x000032b0    23 -r-x .init
11 0x000032d0  1648 0x000032d0  1648 -r-x .plt
12 0x00003940    24 0x00003940    24 -r-x .plt.got
13 0x00003960 73931 0x00003960 73931 -r-x .text
14 0x00015a2c     9 0x00015a2c     9 -r-x .fini
15 0x00015a40 20201 0x00015a40 20201 -r-- .rodata
16 0x0001a92c  2164 0x0001a92c  2164 -r-- .eh_frame_hdr
17 0x0001b1a0 11384 0x0001b1a0 11384 -r-- .eh_frame
18 0x0001e390     8 0x0021e390     8 -rw- .init_array
19 0x0001e398     8 0x0021e398     8 -rw- .fini_array
20 0x0001e3a0  2616 0x0021e3a0  2616 -rw- .data.rel.ro
21 0x0001edd8   480 0x0021edd8   480 -rw- .dynamic
22 0x0001efb8    56 0x0021efb8    56 -rw- .got
23 0x0001f000   840 0x0021f000   840 -rw- .got.plt
24 0x0001f360   616 0x0021f360   616 -rw- .data
25 0x0001f5c8     0 0x0021f5e0  4824 -rw- .bss
26 0x0001f5c8   232 0x00000000   232 ---- .shstrtab
```

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 | head
fs sections
"f section. 1 0x00000000"
"f section..interp 1 0x000002a8"
"f section..note.gnu.build_id 1 0x000002c4"
"f section..note.ABI_tag 1 0x000002e8"
"f section..gnu.hash 1 0x00000308"
"f section..dynsym 1 0x000003b8"
"f section..dynstr 1 0x00000fb8"
"f section..gnu.version 1 0x00001574"
"f section..gnu.version_r 1 0x00001678"
```

Radiff2

Radiff2 is a tool designed to compare binary files, similar to how regular `diff` compares text files.

```
$ radiff2 -h
Usage: radiff2 [-abBcCdjrspOxuUvV] [-A[A]] [-g sym] [-m graph_mode][-t %] [file] [file]
  -a [arch]  specify architecture plugin to use (x86, arm, ..)
  -A [-A]    run aaa or aaaa after loading each binary (see -C)
  -b [bits]  specify register size for arch (16 (thumb), 32, 64, ..)
  -B         output in binary diff (GDIFF)
  -c         count of changes
  -C         graphdiff code (columns: off-A, match-ratio, off-B) (see -A)
  -d         use delta diffing
  -D         show disasm instead of hexpairs
  -e [k=v]   set eval config var value for all RCore instances
  -g [sym|off1,off2]   graph diff of given symbol, or between two offsets
  -G [cmd]   run an r2 command on every RCore instance created
  -i         diff imports of target files (see -u, -U and -z)
  -j         output in json format
  -n         print bare addresses only (diff.bare=1)
  -m [aditsjJ]  choose the graph output mode
  -O         code diffing with opcode bytes only
  -p         use physical addressing (io.va=0)
  -q         quiet mode (disable colors, reduce output)
  -r         output in radare commands
  -s         compute edit distance (no substitution, Eugene W. Myers' O(ND) diff algorithm)
  -ss        compute Levenshtein edit distance (substitution is allowed, O(N^2))
  -S [name]  sort code diff (name, namelen, addr, size, type, dist) (only for -C or -g)
  -t [0-100] set threshold for code diff (default is 70%)
  -x         show two column hexdump diffing
  -X         show two column hexII diffing
  -u         unified output (---+++)
  -U         unified output using system 'diff'
  -v         show version information
  -V         be verbose (current only for -s)
  -z         diff on extracted strings
  -Z         diff code comparing zignatures

Graph Output formats: (-m [mode])
  <blank/a>  Ascii art
  s          r2 commands
  d          Graphviz dot
  g          Graph Modelling Language (gml)
  j          json
  J          json with disarm
  k          SDB key-value
  t          Tiny ascii art
  i          Interactive ascii art
```

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
fcn.000045e0   24 0x45e0 | UNMATCH  (0.916667) | 0x45f0    24 fcn.000045f0
...
```

Moreover, we can ask radiff2 to perform analysis first - adding `-A` option will run `aaa` on the binaries. And we can specify binaries architecture for this analysis too using

```
$ radiff2 -AC -a x86 /bin/true /bin/false | grep UNMATCH
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
                    sub.fileno_500   86 0x4500 | UNMATCH  (0.965116) | 0x4510    86 sub.fileno_510
                sub.__freading_4c0   59 0x44c0 | UNMATCH  (0.949153) | 0x44d0    59 sub.__freading_4d0
                    sub.fileno_440  120 0x4440 | UNMATCH  (0.200000) | 0x4450   120 sub.fileno_450
                 sub.setlocale_fa0   64 0x3fa0 | UNMATCH  (0.104651) | 0x3fb0    64 sub.setlocale_fb0
                      fcn.00003a50  120 0x3a50 | UNMATCH  (0.125000) | 0x3a60   120 fcn.00003a60
```

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 -md -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 -md -g main /bin/false /bin/true | xdot -` (Notice the order of the arguments) to get the two versions. This is the result:

Figure 23: /bin/true vs /bin/false

Parts in yellow indicate that some offsets do not match. The grey piece means a perfect match. The orange one highlights a strong difference. If you look closely, you will see that the left part of the picture has `mov eax, 0x1; pop rbx; pop rbp; ret`, while the right one has `xor edx, edx; pop rbx; pop rbp; ret`.

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. Its main function is to get bytes corresponding to given machine instruction opcode.

```
$ rasm2 -h
Usage: rasm2 [-ACdDehLBvw] [-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)
 -A           Show Analysis information from given hexpairs
 -b [bits]    Set cpu register size (8, 16, 32, 64) (RASM2_BITS)
 -B           Binary input/output (-l is mandatory for binary input)
 -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
 -E           Display ESIL expression (same input as in -d)
 -f [file]    Read data from file
 -F [in:out]  Specify input and/or output filters (att2intel, x86.pseudo, ...)
 -h, -hh      Show this help, -hh for long
 -i [len]     ignore/skip N bytes of the input buffer
 -j           output in json format
 -k [kernel]  Select operating system (linux, windows, darwin, ..)
 -l [len]     Input/Output length
 -L           List Asm plugins: (a=asm, d=disasm, A=analyze, e=ESIL)
 -o [offset]  Set start address for code (default 0)
 -O [file]    Output file name (rasm2 -Bf a.asm -O a)
 -p           Run SPP over input for assembly
 -q           quiet mode
 -r           output in radare commands
 -s [syntax]  Select syntax (intel, att)
 -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
Environment:
 RASM2_NOPLUGINS  do not load shared plugins (speedup loading)
 RASM2_ARCH       same as rasm2 -a
 RASM2_BITS       same as rasm2 -b
 R_DEBUG          if defined, show error messages and crash signal
```

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
_dAe  8 16       6502        LGPL3   6502/NES/C64/Tamagotchi/T-1000 CPU
_dAe  8          8051        PD      8051 Intel CPU
_dA_  16 32      arc         GPL3    Argonaut RISC Core
a___  16 32 64   arm.as      LGPL3   as ARM Assembler (use ARM_AS environment)
adAe  16 32 64   arm         BSD     Capstone ARM disassembler
_dA_  16 32 64   arm.gnu     GPL3    Acorn RISC Machine CPU
_d__  16 32      arm.winedbg LGPL2   WineDBG's ARM disassembler
adAe  8 16       avr         GPL     AVR Atmel
adAe  16 32 64   bf          LGPL3   Brainfuck (by pancake, nibble) v4.0.0
_dA_  32         chip8       LGPL3   Chip8 disassembler
_dA_  16         cr16        LGPL3   cr16 disassembly plugin
_dA_  32         cris        GPL3    Axis Communications 32-bit embedded processor
adA_  32 64      dalvik      LGPL3   AndroidVM Dalvik
ad__  16         dcpu16      PD      Mojang's DCPU-16
_dA_  32 64      ebc         LGPL3   EFI Bytecode
adAe  16         gb          LGPL3   GameBoy(TM) (z80-like)
_dAe  16         h8300       LGPL3   H8/300 disassembly plugin
_dAe  32         hexagon     LGPL3   Qualcomm Hexagon (QDSP6) V6
_d__  32         hppa        GPL3    HP PA-RISC
_dAe             i4004       LGPL3   Intel 4004 microprocessor
_dA_  8          i8080       BSD     Intel 8080 CPU
adA_  32         java        Apache  Java bytecode
_d__  32         lanai       GPL3    LANAI
_d__  8          lh5801      LGPL3   SHARP LH5801 disassembler
_d__  32         lm32        BSD     disassembly plugin for Lattice Micro 32 ISA
_dA_  16 32      m68k        BSD     Capstone M68K disassembler
_dA_  32         malbolge    LGPL3   Malbolge Ternary VM
_d__  16         mcs96       LGPL3   condrets car
adAe  16 32 64   mips        BSD     Capstone MIPS disassembler
adAe  32 64      mips.gnu    GPL3    MIPS CPU
_dA_  16         msp430      LGPL3   msp430 disassembly plugin
_dA_  32         nios2       GPL3    NIOS II Embedded Processor
_dAe  8          pic         LGPL3   PIC disassembler
_dAe  32 64      ppc         BSD     Capstone PowerPC disassembler
_dA_  32 64      ppc.gnu     GPL3    PowerPC
_d__  32         propeller   LGPL3   propeller disassembly plugin
_dA_  32 64      riscv       GPL     RISC-V
_dAe  32         rsp         LGPL3   Reality Signal Processor
_dAe  32         sh          GPL3    SuperH-4 CPU
_dA_  8 16       snes        LGPL3   SuperNES CPU
```

```
_dAe  32 64      sparc       BSD     Capstone SPARC disassembler
_dA_  32 64      sparc.gnu   GPL3    Scalable Processor Architecture
_d__  16         spc700      LGPL3   spc700, snes' sound-chip
_d__  32         sysz        BSD     SystemZ CPU disassembler
_dA_  32         tms320      LGPLv3  TMS320 DSP family (c54x,c55x,c55x+,c64x)
_d__  32         tricore     GPL3    Siemens TriCore CPU
_dAe  32         v810        LGPL3   v810 disassembly plugin
_dAe  32         v850        LGPL3   v850 disassembly plugin
_dAe  8 32       vax         GPL     VAX
adA_  32         wasm        MIT     WebAssembly (by cgvwzq) v0.1.0
_dA_  32         ws          LGPL3   Whitespace esotheric VM
a___  16 32 64   x86.as      LGPL3   Intel X86 GNU Assembler
_dAe  16 32 64   x86         BSD     Capstone X86 disassembler
a___  16 32 64   x86.nasm    LGPL3   X86 nasm assembler
a___  16 32 64   x86.nz      LGPL3   x86 handmade assembler
_dA_  16         xap         PD      XAP4 RISC (CSR)
_dA_  32         xcore       BSD     Capstone XCore disassembler
_dAe  32         xtensa      GPL3    XTensa CPU
adA_  8          z80         GPL     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

Assembling is the action to take a computer instruction in human readable form (using mnemonics) and convert that into a bunch of bytes that can be executed by a machine.

In radare2, the assembler and disassembler logic is implemented in the r_asm_* API, and can be used with the pa and pad commands from the commandline as well as using `rasm2`.

Rasm2 can be used to quickly copy-paste hexpairs that represent a given machine instruction. The following line is assembling this mov instruction for x86/32.

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

Apart from the specifying the input as an argument, you can also pipe it to rasm2:

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

As you have seen, rasm2 can assemble one or many instructions. In line by separating them with a semicolon `;`, but can also read that from a file, using generic nasm/gas/.. syntax and directives. You can check the rasm2 manpage for more details on this.

The `pa` and `pad` are a subcommands of print, what means they will only print assembly or disassembly. In case you want to actually write the instruction it is required to use `wa` or `wx` commands with the assembly string or bytes appended.

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.

Lets create an assembly file called `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
```

Now we can assemble it in place:

```
[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
```

Visual mode

Assembling also is accessible in radare2 visual mode through pressing `A` key to insert the assembly in the current offset.

The cool thing of writing assembly using the visual assembler interface that the changes are done in memory until you press enter.

So you can check the size of the code and which instructions is overlapping before commiting the changes.

Disassembler

Disassembling is the inverse action of assembling. Rasm2 takes hexpair as an input (but can also take a file in binary form) and show the human readable form.

To do this we can use the `-d` option of rasm2 like this:

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

Rasm2 also have the `-D` flag to show the disassembly like `-d` does, but includes offset and bytes.

In radare2 there are many commands to perform a disassembly from a specific place in memory.

You might be interested in trying if you want different outputs for later parsing with your scripts, or just grep to find what you are looking for:

pd N

Disassemble N instructions

pD N

Disassemble N bytes

pda

Disassemble all instructions (seeking 1 byte, or the minimum alignment instruction size), which can be useful for ROP

pi, pI

Same as `pd` and `pD`, but using a simpler output.

Disassembler Configuration

The assembler and disassembler have many small switches to tweak the output.

Those configurations are available through the `e` command. Here there are the most common ones:

- asm.bytes - show/hide bytes
- asm.offset - show/hide offset
- asm.lines - show/hide lines
- asm.ucase - show disasm in uppercase
- …

Use the `e??asm.` for more details.

ragg2

ragg2 stands for `radare2 egg`, this is the basic block to construct relocatable snippets of code to be used for injection in target processes when doing exploiting.

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

By default it will compile it's own `ragg2` language, but you can also compile C code using GCC or Clang shellcodes depending on the file extension. Lets create C file called `a.c`:

```c
int main() {
    write(1, "Hello World\n", 13);
    return 0;
}
```

```
$ ragg2 -a x86 -b32 a.c
e900000000488d3516000000bf01000000b80400000248c7c20d0000000f0531c0c348656c6c6f20576f726c640a00

$ rasm2 -a x86 -b 32 -D e900000000488d3516000000bf01000000b80400000248c7c20d0000000f0531c0c348656c6c6f20576f726c640a00
0x00000000   5               e900000000  jmp 5
0x00000005   1                       48  dec eax
0x00000006   6             8d3516000000  lea esi, [0x16]
0x0000000c   5               bf01000000  mov edi, 1
0x00000011   5               b804000002  mov eax, 0x2000004
0x00000016   1                       48  dec eax
0x00000017   6             c7c20d000000  mov edx, 0xd
0x0000001d   2                     0f05  syscall
0x0000001f   2                     31c0  xor eax, eax
0x00000021   1                       c3  ret
0x00000022   1                       48  dec eax
0x00000023   2                     656c  insb byte es:[edi], dx
0x00000025   1                       6c  insb byte es:[edi], dx
0x00000026   1                       6f  outsd dx, dword [esi]
0x00000027   3                   20576f  and byte [edi + 0x6f], dl
0x0000002a   2                     726c  jb 0x98
0x0000002c   3                   640a00  or al, byte fs:[eax]
```

Compiling ragg2 example

```
$ cat hello.r
exit@syscall(1);

main@global() {
    exit(2);
}

$ ragg2 -a x86 -b 64 hello.r
48c7c00200000050488b3c2448c7c0010000000f054883c408c3
0x00000000   1                       48  dec eax
```

```
0x00000001   6             c7c002000000  mov eax, 2
0x00000007   1                       50  push eax
0x00000008   1                       48  dec eax
0x00000009   3                   8b3c24  mov edi, dword [esp]
0x0000000c   1                       48  dec eax
0x0000000d   6             c7c001000000  mov eax, 1
0x00000013   2                     0f05  syscall
0x00000015   1                       48  dec eax
0x00000016   3                   83c408  add esp, 8
0x00000019   1                       c3  ret

$ rasm2 -a x86 -b 64 -D 48c7c00200000050488b3c2448c7c0010000000f054883c408c3
0x00000000   7           48c7c002000000  mov rax, 2
0x00000007   1                       50  push rax
0x00000008   4                 488b3c24  mov rdi, qword [rsp]
0x0000000c   7           48c7c001000000  mov rax, 1
0x00000013   2                     0f05  syscall
0x00000015   4                 4883c408  add rsp, 8
0x00000019   1                       c3  ret
```

Tiny binaries

You can create them using the `-F` flag in ragg2, or the `-C` in rabin2.

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.

```
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
```

Attention: All the numbers after `.var` and `.arg` mean the offset with the top of stack, not variable symbols.

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

Math operations

Ragg2 supports local variables assignment by math operating, including

the following operators:

`+ - * / & | ^`

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)`

`if (cond) { body } else { body }`

`break ()` – executes a trap instruction

Comments

Supported syntax for comments are:

`/* multiline comment */'`

`// single line comment`

`# single line comment`

rahash2

The rahash2 tool can be used to compute checksums of files, disk devices or strings. By block or entirely using many different hash algorithms.

This tool is also capable of doing some encoding/decoding operations like base64 and xor encryption.

This is an example usage:

`$ rahash2 -a md5 -s "hello world"`

Note that rahash2 also permits to read from stdin in a stream, so you don't need 4GB of ram to compute the hash of a 4GB file.

Hashing by blocks

When doing forensics, it is useful to compute partial checksums. The reason for that is because you may want to split a huge file into small portions that are easier to identify by contents or regions in the disk.

This will spot the same hash for blocks containing the same contents. For example, if is filled with zeros.

It can also be used to find which blocks have changed between more than one sample dump.

This can be useful when analyzing ram dumps from a virtual machine for example. Use this command for this:

`$ rahash2 -B 1M -b -a sha256 /bin/ls`

Hashing with rabin2

The rabin2 tool parses the binary headers of the files, but it also have the ability to use the rhash plugins to compute checksum of sections in the binary.

```
$ rabin2 -K md5 -S /bin/ls
```

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`:

```
[0x00000000]> ph?
md5
sha1
sha256
sha384
sha512
md4
xor
xorpair
parity
entropy
hamdist
pcprint
mod255
xxhash
adler32
luhn
crc8smbus
crc15can
crc16
crc16hdlc
crc16usb
crc16citt
crc24
crc32
crc32c
crc32ecma267
crc32bzip2
crc32d
crc32mpeg2
crc32posix
crc32q
crc32jamcrc
crc32xfer
crc64
crc64ecma
crc64we
crc64xz
crc64iso
```

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
```

Examples

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

```
$ rahash2 -h
Usage: rahash2 [-rBhLkv] [-b S] [-a A] [-c H] [-E A] [-s S] [-f O] [-t O] [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
 -c hash     compare with this hash
 -e          swap endian (use little endian)
 -E algo     encrypt. Use -S to set key and -I to set IV
 -D algo     decrypt. Use -S to set key and -I to set IV
 -f from     start hashing at given address
 -i num      repeat hash N iterations
 -I iv       use give initialization vector (IV) (hexa or s:string)
 -S seed     use given seed (hexa or s:string) use ^ to prefix (key for -E)
             (- will slurp the key from stdin, the @ prefix points to a file
 -k          show hash using the openssh's randomkey algorithm
 -q          run in quiet mode (-qq to show only the hash)
 -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
 -x hexstr   hash this hexpair string instead of files
 -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: 0x00000000-0x000268c7 md5: 767f0fff116bc6584dbfc1af6fd48fc7
/bin/ls: 0x00000000-0x000268c7 sha1: 404303f3960f196f42f8c2c12970ab0d49e28971
/bin/ls: 0x00000000-0x000268c7 sha256: 74ea05150acf311484bddd19c608aa02e6bf3332a0f0805a4deb278e17396354
/bin/ls: 0x00000000-0x000268c7 sha384: c6f811287514ceeeaabe73b5b2f54545036d6fd3a192ea5d6a1fcd494d46151df4117e1c62de0884cbc174c8db008ed1
/bin/ls: 0x00000000-0x000268c7 sha512: 53e4950a150f06d7922a2ed732060e291bf0e1c2ac20bc72a41b9303e1f2837d50643761030d8b918ed05d12993d9515e1ac46676bc0d15ac9
/bin/ls: 0x00000000-0x000268c7 md4: fdfe7c7118a57c1ff8c88a51b16fc78c
/bin/ls: 0x00000000-0x000268c7 xor: 42
/bin/ls: 0x00000000-0x000268c7 xorpair: d391
/bin/ls: 0x00000000-0x000268c7 parity: 00
/bin/ls: 0x00000000-0x000268c7 entropy: 5.95471783
/bin/ls: 0x00000000-0x000268c7 hamdist: 00
/bin/ls: 0x00000000-0x000268c7 pcprint: 22
/bin/ls: 0x00000000-0x000268c7 mod255: ef
/bin/ls: 0x00000000-0x000268c7 xxhash: 76554666
/bin/ls: 0x00000000-0x000268c7 adler32: 7704fe60
/bin/ls: 0x00000000-0x000268c7 luhn: 01
/bin/ls: 0x00000000-0x000268c7 crc8smbus: 8d
/bin/ls: 0x00000000-0x000268c7 crc15can: 1cd5
/bin/ls: 0x00000000-0x000268c7 crc16: d940
/bin/ls: 0x00000000-0x000268c7 crc16hdlc: 7847
/bin/ls: 0x00000000-0x000268c7 crc16usb: 17bb
/bin/ls: 0x00000000-0x000268c7 crc16citt: 67f7
/bin/ls: 0x00000000-0x000268c7 crc24: 3e7053
/bin/ls: 0x00000000-0x000268c7 crc32: c713f78f
/bin/ls: 0x00000000-0x000268c7 crc32c: 6cfba67c
/bin/ls: 0x00000000-0x000268c7 crc32ecma267: b4c809d6
/bin/ls: 0x00000000-0x000268c7 crc32bzip2: a1884a09
/bin/ls: 0x00000000-0x000268c7 crc32d: d1a9533c
/bin/ls: 0x00000000-0x000268c7 crc32mpeg2: 5e77b5f6
/bin/ls: 0x00000000-0x000268c7 crc32posix: 6ba0dec3
/bin/ls: 0x00000000-0x000268c7 crc32q: 3166085c
/bin/ls: 0x00000000-0x000268c7 crc32jamcrc: 38ec0870
/bin/ls: 0x00000000-0x000268c7 crc32xfer: 7504089d
/bin/ls: 0x00000000-0x000268c7 crc64: b6471d3093d94241
/bin/ls: 0x00000000-0x000268c7 crc64ecma: b6471d3093d94241
/bin/ls: 0x00000000-0x000268c7 crc64we: 8fe37d44a47157bd
/bin/ls: 0x00000000-0x000268c7 crc64xz: ea83e12c719e0d79
/bin/ls: 0x00000000-0x000268c7 crc64iso: d243106d9853221c
```

Plugins

radare2 is implemented on top of a bunch of libraries, almost every of those libraries support plugins to extend the capabilities of the library or add support for different targets.

This section aims to explain what are the plugins, how to write them and use them

Types of plugins

```
$ ls libr/*/p | grep : | awk -F / '{ print $2 }'
anal      # analysis plugins
asm       # assembler/disassembler plugins
bin       # binary format parsing plugins
bp        # breakpoint plugins
core      # core plugins (implement new commands)
crypto    # encrypt/decrypt/hash/...
debug     # debugger backends
egg       # shellcode encoders, etc
fs        # filesystems and partition tables
io        # io plugins
lang      # embedded scripting languages
parse     # disassembler parsing plugins
reg       # arch register logic
```

Listing plugins

Some r2 tools have the `-L` flag to list all the plugins associated to the functionality.

```
rasm2 -L    # list asm plugins
r2 -L       # list io plugins
rabin2 -L   # list bin plugins
rahash2 -L  # list hash/crypto/encoding plugins
```

There are more plugins in r2land, we can list them from inside r2, and this is done by using the `L` suffix.

Those are some of the commands:

```
L          # list core plugins
iL         # list bin plugins
dL         # list debug plugins
mL         # list fs plugins
ph         # print support hash algoriths
```

You can use the `?` as value to get the possible values in the associated eval vars.

```
e asm.arch=?   # list assembler/disassembler plugins
e anal.arch=?  # list analysis plugins
```

Notes

Note there are some inconsistencies that most likely will be fixed in the future radare2 versions.

IO plugins

All access to files, network, debugger and all input/output in general 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.

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<br />
$ r2 pid://1927
```

- Remote sessions

```
$ r2 rap://:1234<br />
$ r2 rap://<host>:1234//bin/ls
```

- Virtual buffers

```
$ r2 malloc://512<br />
shortcut for
$ r2 -
```

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

```
$ r2 -L
rw_  ar       Open ar/lib files [ar|lib]://[file//path] (LGPL3)
rw_  bfdbg    BrainFuck Debugger (bfdbg://path/to/file) (LGPL3)
rwd  bochs    Attach to a BOCHS debugger (LGPL3)
r_d  debug    Native debugger (dbg:///bin/ls dbg://1388 pidof:// waitfor://) (LGPL3) v0.2.0 pancake
rw_  default  open local files using def_mmap:// (LGPL3)
rwd  gdb      Attach to gdbserver, 'qemu -s', gdb://localhost:1234 (LGPL3)
rw_  gprobe   open gprobe connection using gprobe:// (LGPL3)
rw_  gzip     read/write gzipped files (LGPL3)
rw_  http     http get (http://rada.re/) (LGPL3)
rw_  ihex     Intel HEX file (ihex://eeproms.hex) (LGPL)
r__  mach     mach debug io (unsupported in this platform) (LGPL)
rw_  malloc   memory allocation (malloc://1024 hex://cd8090) (LGPL3)
rw_  mmap     open file using mmap:// (LGPL3)
rw_  null     null-plugin (null://23) (LGPL3)
rw_  procpid  /proc/pid/mem io (LGPL3)
rwd  ptrace   ptrace and /proc/pid/mem (if available) io (LGPL3)
rwd  qnx      Attach to QNX pdebug instance, qnx://host:1234 (LGPL3)
rw_  r2k      kernel access API io (r2k://) (LGPL3)
rw_  r2pipe   r2pipe io plugin (MIT)
rw_  r2web    r2web io client (r2web://cloud.rada.re/cmd/) (LGPL3)
rw_  rap      radare network protocol (rap://:port rap://host:port/file) (LGPL3)
rw_  rbuf     RBuffer IO plugin: rbuf:// (LGPL)
rw_  self     read memory from myself using 'self://' (LGPL3)
rw_  shm      shared memory resources (shm://key) (LGPL3)
rw_  sparse   sparse buffer allocation (sparse://1024 sparse://) (LGPL3)
rw_  tcp      load files via TCP (listen or connect) (LGPL3)
rwd  windbg   Attach to a KD debugger (windbg://socket) (LGPL3)
rwd  winedbg  Wine-dbg io and debug.io plugin for r2 (MIT)
rw_  zip      Open zip files [apk|ipa|zip|zipall]://[file//path] (BSD)
```

Implementing a new disassembly plugin

Radare2 has modular architecture, thus adding support for a new architecture is very easy, if you are fluent in C. For various reasons it might be easier to implement it out of the tree. For this we will need to create single C file, called `asm_mycpu.c` and makefile for it.

The key thing of RAsm plugin is a structure

```
RAsmPlugin r_asm_plugin_mycpu = {
    .name = "mycpu",
    .license = "LGPL3",
    .desc = "MYCPU disassembly plugin",
    .arch = "mycpu",
    .bits = 32,
    .endian = R_SYS_ENDIAN_LITTLE,
    .disassemble = &disassemble
};
```

where `.disassemble` is a pointer to disassembly function, which accepts the bytes buffer and length:

```
static int disassemble(RAsm *a, RAsmOp *op, const ut8 *buf, int len)
```

Makefile

```
NAME=asm_snes
R2_PLUGIN_PATH=$(shell r2 -H R2_USER_PLUGINS)
LIBEXT=$(shell r2 -H LIBEXT)
CFLAGS=-g -fPIC $(shell pkg-config --cflags r_anal)
LDFLAGS=-shared $(shell pkg-config --libs r_anal)
OBJS=$(NAME).o
LIB=$(NAME).$(LIBEXT)

all: $(LIB)

clean:
    rm -f $(LIB) $(OBJS)

$(LIB): $(OBJS)
    $(CC) $(CFLAGS) $(LDFLAGS) $(OBJS) -o $(LIB)

install:
    cp -f asm_mycpu.$(SO_EXT) $(R2_PLUGIN_PATH)
```

```
uninstall:
    rm -f $(R2_PLUGIN_PATH)/asm_mycpu.$(SO_EXT)
```

asm_mycpu.c

```
/* radare - LGPL - Copyright 2018 - user */

#include <stdio.h>
#include <string.h>
#include <r_types.h>
#include <r_lib.h>
#include <r_asm.h>

static int disassemble(RAsm *a, RAsmOp *op, const ut8 *buf, int len) {
    struct op_cmd cmd = {
        .instr = "",
        .operands = ""
    };
    if (len < 2) return -1;
    int ret = decode_opcode (buf, len, &cmd);
    if (ret > 0) {
        snprintf (op->buf_asm, R_ASM_BUFSIZE, "%s %s",
              cmd.instr, cmd.operands);
    }
    return op->size = ret;
}

RAsmPlugin r_asm_plugin_mycpu = {
    .name = "mycpu",
    .license = "LGPL3",
    .desc = "MYCPU disassembly plugin",
    .arch = "mycpu",
    .bits = 32,
    .endian = R_SYS_ENDIAN_LITTLE,
    .disassemble = &disassemble
};

#ifndef R2_PLUGIN_INCORE
R_API RLibStruct radare_plugin = {
    .type = R_LIB_TYPE_ASM,
    .data = &r_asm_plugin_mycpu,
    .version = R2_VERSION
};
#endif
```

After compiling radare2 will list this plugin in the output:

```
_d__  _8_32      mycpu        LGPL3   MYCPU
```

Moving plugin into the tree

Pushing a new architecture into the main branch of r2 requires to modify several files in order to make it fit into the way the rest of plugins are built.

List of affected files:

- `plugins.def.cfg` : add the `asm.mycpu` plugin name string in there
- `libr/asm/p/mycpu.mk` : build instructions
- `libr/asm/p/asm_mycpu.c` : implementation
- `libr/include/r_asm.h` : add the struct definition in there

Check out how the NIOS II CPU disassembly plugin was implemented by reading those commits:

Implement RAsm plugin: https://github.com/radareorg/radare2/commit/933dc0ef6ddfe44c88bbb261165bf8f8b531476b

Implement RAnal plugin: https://github.com/radareorg/radare2/commit/ad430f0d52fbe933e0830c49ee607e9b0e4ac8f2

Implementing a new analysis plugin

After implementing disassembly plugin, you might have noticed that output is far from being good - no proper highlighting, no reference lines and so on. This is because radare2 requires every architecture plugin to provide also analysis information about every opcode. At the moment the implementation of disassembly and opcodes analysis is separated between two modules - RAsm and RAnal. Thus we need to write an analysis plugin too. The principle is very similar - you just need to create a C file and corresponding Makefile.

They structure of RAnal plugin looks like

```
RAnalPlugin r_anal_plugin_v810 = {
    .name = "mycpu",
    .desc = "MYCPU code analysis plugin",
    .license = "LGPL3",
    .arch = "mycpu",
    .bits = 32,
    .op = mycpu_op,
    .esil = true,
    .set_reg_profile = set_reg_profile,
};
```

Like with disassembly plugin there is a key function - `mycpu_op` which scans the opcode and builds RAnalOp structure. On the other hand, in this example analysis plugins also performs uplifting to ESIL, which is enabled in `.esil = true` statement. Thus, `mycpu_op` obliged to fill the corresponding RAnalOp ESIL field for the opcodes. Second important thing for ESIL uplifting and emulation - register profile, like in debugger, which is set within `set_reg_profile` function.

Makefile

```
NAME=anal_snes
R2_PLUGIN_PATH=$(shell r2 -H R2_USER_PLUGINS)
LIBEXT=$(shell r2 -H LIBEXT)
CFLAGS=-g -fPIC $(shell pkg-config --cflags r_anal)
LDFLAGS=-shared $(shell pkg-config --libs r_anal)
OBJS=$(NAME).o
LIB=$(NAME).$(LIBEXT)
```

```
all: $(LIB)

clean:
	rm -f $(LIB) $(OBJS)

$(LIB): $(OBJS)
	$(CC) $(CFLAGS) $(LDFLAGS) $(OBJS) -o $(LIB)

install:
	cp -f anal_snes.$(SO_EXT) $(R2_PLUGIN_PATH)

uninstall:
	rm -f $(R2_PLUGIN_PATH)/anal_snes.$(SO_EXT)
```

anal_snes.c:

```
/* radare - LGPL - Copyright 2015 - condret */

#include <string.h>
#include <r_types.h>
#include <r_lib.h>
#include <r_asm.h>
#include <r_anal.h>
#include "snes_op_table.h"

static int snes_anop(RAnal *anal, RAnalOp *op, ut64 addr, const ut8 *data, int len) {
	memset (op, '\0', sizeof (RAnalOp));
	op->size = snes_op[data[0]].len;
	op->addr = addr;
	op->type = R_ANAL_OP_TYPE_UNK;
	switch (data[0]) {
		case 0xea:
			op->type = R_ANAL_OP_TYPE_NOP;
			break;
	}
	return op->size;
}

struct r_anal_plugin_t r_anal_plugin_snes = {
	.name = "snes",
	.desc = "SNES analysis plugin",
	.license = "LGPL3",
	.arch = R_SYS_ARCH_NONE,
	.bits = 16,
	.init = NULL,
	.fini = NULL,
	.op = &snes_anop,
	.set_reg_profile = NULL,
	.fingerprint_bb = NULL,
	.fingerprint_fcn = NULL,
	.diff_bb = NULL,
	.diff_fcn = NULL,
	.diff_eval = NULL
};

#ifndef R2_PLUGIN_INCORE
R_API RLibStruct radare_plugin = {
	.type = R_LIB_TYPE_ANAL,
	.data = &r_anal_plugin_snes,
	.version = R2_VERSION
};
#endif
```

After compiling radare2 will list this plugin in the output:

```
_dA_  _8_16      snes        LGPL3   SuperNES CPU
```

snes_op_table.h: https://github.com/radareorg/radare2/blob/master/libr/asm/arch/snes/snes_op_table.h

Example:

- **6502**: https://github.com/radareorg/radare2/commit/64636e9505f9ca8b408958d3c01ac8e3ce254a9b
- **SNES**: https://github.com/radareorg/radare2/commit/60d6e5a1b9d244c7085b22ae8985d00027624b49

Implementing a new format

To enable virtual addressing

In `info` add `et->has_va = 1;` and `ptr->srwx` with the `R_BIN_SCN_MAP;` attribute

Create a folder with file format name in libr/bin/format

Makefile:

```
NAME=bin_nes
R2_PLUGIN_PATH=$(shell r2 -H R2_USER_PLUGINS)
LIBEXT=$(shell r2 -H LIBEXT)
CFLAGS=-g -fPIC $(shell pkg-config --cflags r_bin)
LDFLAGS=-shared $(shell pkg-config --libs r_bin)
OBJS=$(NAME).o
LIB=$(NAME).$(LIBEXT)

all: $(LIB)

clean:
	rm -f $(LIB) $(OBJS)
```

```
$(LIB): $(OBJS)
	$(CC) $(CFLAGS) $(LDFLAGS) $(OBJS) -o $(LIB)

install:
	cp -f $(NAME).$(SO_EXT) $(R2_PLUGIN_PATH)

uninstall:
	rm -f $(R2_PLUGIN_PATH)/$(NAME).$(SO_EXT)
```

bin_nes.c:

```c
#include <r_util.h>
#include <r_bin.h>

static bool load_buffer(RBinFile *bf, void **bin_obj, RBuffer *b, ut64 loadaddr, Sdb *sdb) {
    ut64 size;
    const ut8 *buf = r_buf_data (b, &size);
    r_return_val_if_fail (buf, false);
    *bin_obj = r_bin_internal_nes_load (buf, size);
    return *bin_obj != NULL;
}

static void destroy(RBinFile *bf) {
    r_bin_free_all_nes_obj (bf->o->bin_obj);
    bf->o->bin_obj = NULL;
}

static bool check_buffer(RBuffer *b) {
    if (!buf || length < 4) return false;
    return (!memcmp (buf, "\x4E\x45\x53\x1A", 4));
}

static RBinInfo* info(RBinFile *arch) {
    RBinInfo \*ret = R_NEW0 (RBinInfo);
    if (!ret) return NULL;

    if (!arch || !arch->buf) {
        free (ret);
        return NULL;
    }
    ret->file = strdup (arch->file);
    ret->type = strdup ("ROM");
    ret->machine = strdup ("Nintendo NES");
    ret->os = strdup ("nes");
    ret->arch = strdup ("6502");
    ret->bits = 8;

    return ret;
}

struct r_bin_plugin_t r_bin_plugin_nes = {
    .name = "nes",
    .desc = "NES",
    .license = "BSD",
    .get_sdb = NULL,
    .load_buffer = &load_buffer,
    .destroy = &destroy,
    .check_buffer = &check_buffer,
    .baddr = NULL,
    .entries = NULL,
    .sections = NULL,
    .info = &info,
};

#ifndef R2_PLUGIN_INCORE
R_API RLibStruct radare_plugin = {
    .type = R_LIB_TYPE_BIN,
    .data = &r_bin_plugin_nes,
    .version = R2_VERSION
};
#endif
```

Some Examples

- XBE - https://github.com/radareorg/radare2/pull/972
- COFF - https://github.com/radareorg/radare2/pull/645
- TE - https://github.com/radareorg/radare2/pull/61
- Zimgz - https://github.com/radareorg/radare2/commit/d1351cf836df3e2e63043a6dc728e880316f00eb
- OMF - https://github.com/radareorg/radare2/commit/44fd8b2555a0446ea759901a94c06f20566bbc40

Write a debugger plugin

- Adding the debugger registers profile into the shlr/gdb/src/core.c
- Adding the registers profile and architecture support in the libr/debug/p/debug_native.c and libr/debug/p/debug_gdb.c
- Add the code to apply the profiles into the function `r_debug_gdb_attach(RDebug *dbg, int pid)`

If you want to add support for the gdb, you can see the register profile in the active gdb session using command `maint print registers`.

More to come..

- Related article: http://radare.today/posts/extending-r2-with-new-plugins/

Some commits related to "Implementing a new architecture"

- Extensa: https://github.com/radareorg/radare2/commit/6f1655c49160fe9a287020537afe0fb8049085d7
- Malbolge: https://github.com/radareorg/radare2/pull/579
- 6502: https://github.com/radareorg/radare2/pull/656
- h8300: https://github.com/radareorg/radare2/pull/664
- GBA: https://github.com/radareorg/radare2/pull/702
- CR16: https://github.com/radareorg/radare2/pull/721/ && 726
- XCore: https://github.com/radareorg/radare2/commit/bb16d1737ca5a471142f16ccfa7d444d2713a54d
- SharpLH5801: https://github.com/neuschaefer/radare2/commit/f4993cca634161ce6f82a64596fce45fe6b818e7
- MSP430: https://github.com/radareorg/radare2/pull/1426
- HP-PA-RISC: https://github.com/radareorg/radare2/commit/f8384feb6ba019b91229adb8fd6e0314b0656f7b
- V810: https://github.com/radareorg/radare2/pull/2899
- TMS320: https://github.com/radareorg/radare2/pull/596

Implementing a new pseudo architecture

This is an simple plugin for z80 that you may use as example:

https://github.com/radareorg/radare2/commit/8ff6a92f65331cf8ad74cd0f44a60c258b137a06

Python plugins

At first, to be able to write a plugins in Python for radare2 you need to install r2lang plugin: `r2pm -i lang-python`. Note - in the following examples there are missing functions of the actual decoding for the sake of readability!

For this you need to do this: 1. `import r2lang` and `from r2lang import R` (for constants) 2. Make a function with 2 subfunctions - `assemble` and `disassemble` and returning plugin structure - for RAsm plugin

```
def mycpu(a):
    def assemble(s):
        return [1, 2, 3, 4]

    def disassemble(memview, addr):
        try:
            opcode = get_opcode(memview) # https://docs.python.org/3/library/stdtypes.html#memoryview
            opstr = optbl[opcode][1]
            return [4, opstr]
        except:
            return [4, "unknown"]
```

3. This structure should contain a pointers to these 2 functions - `assemble` and `disassemble`

```
return {
        "name" : "mycpu",
        "arch" : "mycpu",
        "bits" : 32,
        "endian" : R.R_SYS_ENDIAN_LITTLE,
        "license" : "GPL",
        "desc" : "MYCPU disasm",
        "assemble" : assemble,
        "disassemble" : disassemble,
}
```

4. Make a function with 2 subfunctions - `set_reg_profile` and `op` and returning plugin structure - for RAnal plugin

```
def mycpu_anal(a):
       def set_reg_profile():
        profile = "=PC  pc\n" + \
        "=SP    sp\n" + \
        "gpr    r0  .32 0   0\n" + \
        "gpr    r1  .32 4   0\n" + \
        "gpr    r2  .32 8   0\n" + \
        "gpr    r3  .32 12  0\n" + \
        "gpr    r4  .32 16  0\n" + \
        "gpr    r5  .32 20  0\n" + \
        "gpr    sp  .32 24  0\n" + \
        "gpr    pc  .32 28  0\n"
        return profile

    def op(memview, pc):
        analop = {
            "type" : R.R_ANAL_OP_TYPE_NULL,
            "cycles" : 0,
            "stackop" : 0,
            "stackptr" : 0,
            "ptr" : -1,
            "jump" : -1,
            "addr" : 0,
            "eob" : False,
            "esil" : "",
        }
        try:
            opcode = get_opcode(memview) # https://docs.python.org/3/library/stdtypes.html#memoryview
            esilstr = optbl[opcode][2]
            if optbl[opcode][0] == "J": # it's jump
                analop["type"] = R.R_ANAL_OP_TYPE_JMP
                analop["jump"] = decode_jump(opcode, j_mask)
                esilstr = jump_esil(esilstr, opcode, j_mask)

        except:
            result = analop
        # Don't forget to return proper instruction size!
        return [4, result]
```

5. This structure should contain a pointers to these 2 functions - `set_reg_profile` and `op`

```
    return {
            "name" : "mycpu",
            "arch" : "mycpu",
            "bits" : 32,
            "license" : "GPL",
            "desc" : "MYCPU anal",
            "esil" : 1,
            "set_reg_profile" : set_reg_profile,
            "op" : op,
    }
```

6. Then register those using `r2lang.plugin("asm")` and `r2lang.plugin("anal")` respectively

```
print("Registering MYCPU disasm plugin...")
print(r2lang.plugin("asm", mycpu))
print("Registering MYCPU analysis plugin...")
print(r2lang.plugin("anal", mycpu_anal))
```

You can combine everything in one file and load it using `-i` option:

```
r2 -I mycpu.py some_file.bin
```

Or you can load it from the r2 shell: `#!python mycpu.py`

See also:

- Python
- Javascript

Implementing new format plugin in Python

Note - in the following examples there are missing functions of the actual decoding for the sake of readability!

For this you need to do this: 1. `import r2lang` 2. Make a function with subfunctions: - `load` - `load_bytes` - `destroy` - `check_bytes` - `baddr` - `entries` - `sections` - `imports` - `relocs` - `binsym` - `info`

and returning plugin structure - for RAsm plugin

```
def le_format(a):
    def load(binf):
        return [0]

    def check_bytes(buf):
        try:
            if buf[0] == 77 and buf[1] == 90:
                lx_off, = struct.unpack("<I", buf[0x3c:0x40])
                if buf[lx_off] == 76 and buf[lx_off+1] == 88:
                    return [1]
            return [0]
        except:
            return [0]
```

and so on. Please be sure of the parameters for each function and format of returns. Note, that functions `entries`, `sections`, `imports`, `relocs` returns a list of special formed dictionaries - each with a different type. Other functions return just a list of numerical values, even if single element one. There is a special function, which returns information about the file - `info`:

```
    def info(binf):
        return [{
                "type" : "le",
                "bclass" : "le",
                "rclass" : "le",
                "os" : "OS/2",
                "subsystem" : "CLI",
                "machine" : "IBM",
                "arch" : "x86",
                "has_va" : 0,
                "bits" : 32,
                "big_endian" : 0,
                "dbg_info" : 0,
                }]
```

3. This structure should contain a pointers to the most important functions like `check_bytes`, `load` and `load_bytes`, `entries`, `relocs`, `imports`.

```
    return {
            "name" : "le",
            "desc" : "OS/2 LE/LX format",
            "license" : "GPL",
            "load" : load,
            "load_bytes" : load_bytes,
            "destroy" : destroy,
            "check_bytes" : check_bytes,
            "baddr" : baddr,
            "entries" : entries,
            "sections" : sections,
            "imports" : imports,
            "symbols" : symbols,
            "relocs" : relocs,
            "binsym" : binsym,
            "info" : info,
    }
```

4. Then you need to register it as a file format plugin:

```
print("Registering OS/2 LE/LX plugin...")
print(r2lang.plugin("bin", le_format))
```

Debugging

It is common to have an issues when you write a plugin, especially if you do this for the first time. This is why debugging them is very important. The first step for debugging is to set an environment variable when running radare2 instance:

```
R_DEBUG=yes r2 /bin/ls
Loading /usr/local/lib/radare2/2.2.0-git//bin_xtr_dyldcache.so
Cannot find symbol 'radare_plugin' in library '/usr/local/lib/radare2/2.2.0-git//bin_xtr_dyldcache.so'
Cannot open /usr/local/lib/radare2/2.2.0-git//2.2.0-git
Loading /home/user/.config/radare2/plugins/asm_mips_ks.so
PLUGIN OK 0x55b205ea6070 fcn 0x7f298de08762
Loading /home/user/.config/radare2/plugins/asm_sparc_ks.so
PLUGIN OK 0x55b205ea6070 fcn 0x7f298de08762
Cannot open /home/user/.config/radare2/plugins/pimp
Cannot open /home/user/.config/radare2/plugins/yara
Loading /home/user/.config/radare2/plugins/asm_arm_ks.so
PLUGIN OK 0x55b205ea6070 fcn 0x7f298de08762
Loading /home/user/.config/radare2/plugins/core_yara.so
Module version mismatch /home/user/.config/radare2/plugins/core_yara.so (2.1.0) vs (2.2.0-git)
Loading /home/user/.config/radare2/plugins/asm_ppc_ks.so
PLUGIN OK 0x55b205ea6070 fcn 0x7f298de08762
Loading /home/user/.config/radare2/plugins/lang_python3.so
PLUGIN OK 0x55b205ea5ed0 fcn 0x7f298de08692
Loading /usr/local/lib/radare2/2.2.0-git/bin_xtr_dyldcache.so
Cannot find symbol 'radare_plugin' in library '/usr/local/lib/radare2/2.2.0-git/bin_xtr_dyldcache.so'
Cannot open /usr/local/lib/radare2/2.2.0-git/2.2.0-git
Cannot open directory '/usr/local/lib/radare2-extras/2.2.0-git'
Cannot open directory '/usr/local/lib/radare2-bindings/2.2.0-git'
USER CONFIG loaded from /home/user/.config/radare2/radare2rc
 -- In visual mode press 'c' to toggle the cursor mode. Use tab to navigate
[0x00005520]>
```

Testing the plugin

This plugin is used by rasm2 and r2. You can verify that the plugin is properly loaded with this command:

```
$ rasm2 -L | grep mycpu
_d  mycpu        My CPU disassembler  (LGPL3)
```

Let's open an empty file using the 'mycpu' arch and write some random code there.

```
$ r2 -
 -- I endians swap
[0x00000000]> e asm.arch=mycpu
[0x00000000]> woR
[0x00000000]> pd 10
           0x00000000    888e         mov r8, 14
           0x00000002    b2a5         ifnot r10, r5
           0x00000004    3f67         ret
           0x00000006    7ef6         bl r15, r6
           0x00000008    2701         xor r0, 1
           0x0000000a    9826         mov r2, 6
           0x0000000c    478d         xor r8, 13
           0x0000000e    6b6b         store r6, 11
           0x00000010    1382         add r8, r2
           0x00000012    7f15         ret
```

Yay! it works.. and the mandatory oneliner too!

```
r2 -nqamycpu -cwoR -cpd' 10' -
```

Creating an r2pm package of the plugin

As you remember radare2 has its own packaging manager and we can easily add newly written plugin for everyone to access.

All packages are located in radare2-pm repository, and have very simple text format.

```
R2PM_BEGIN

R2PM_GIT "https://github.com/user/mycpu"
R2PM_DESC "[r2-arch] MYCPU disassembler and analyzer plugins"

R2PM_INSTALL() {
    ${MAKE} clean
    ${MAKE} all || exit 1
    ${MAKE} install R2PM_PLUGDIR="${R2PM_PLUGDIR}"
}

R2PM_UNINSTALL() {
    rm -f "${R2PM_PLUGDIR}/asm_mycpu."*
    rm -f "${R2PM_PLUGDIR}/anal_mycpu."*
}

R2PM_END
```

Then add it in the `/db` directory of radare2-pm repository and send a pull request to the mainline.

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
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x00000568 0x08048568 24  25   .rodata ascii IOLI Crackme Level 0x00\n
1   0x00000581 0x08048581 10  11   .rodata ascii Password:
2   0x0000058f 0x0804858f 6   7    .rodata ascii 250382
3   0x00000596 0x08048596 18  19   .rodata ascii Invalid Password!\n
4   0x000005a9 0x080485a9 15  16   .rodata ascii Password OK :)\n
```

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

```
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x00000568 0x08048568 24  25   .rodata ascii IOLI Crackme Level 0x00\n
```

Here we have the prompt for the password.

```
1   0x00000581 0x08048581 10  11   .rodata ascii Password:
```

This is the error on entering an invalid password.

```
3   0x00000596 0x08048596 18  19   .rodata ascii Invalid Password!\n
```

This is the message on the password being accepted.

```
4   0x000005a9 0x080485a9 15  16   .rodata ascii Password OK :)\n
```

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

```
2   0x0000058f 0x0804858f 6   7    .rodata ascii 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
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x00000528 0x08048528 24  25   .rodata ascii IOLI Crackme Level 0x01\n
1   0x00000541 0x08048541 10  11   .rodata ascii Password:
2   0x0000054f 0x0804854f 18  19   .rodata ascii Invalid Password!\n
3   0x00000562 0x08048562 15  16   .rodata ascii 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 find strings
[0x08048330]> aa
[0x08048330]> pdf@main
            ; DATA XREF from entry0 @ 0x8048347
/ 113: int main (int argc, char **argv, char **envp);
|           ; var int32_t var_4h @ ebp-0x4
|           ; var int32_t var_sp_4h @ esp+0x4
|           0x080483e4      55             push ebp
|           0x080483e5      89e5           mov ebp, esp
|           0x080483e7      83ec18         sub esp, 0x18
|           0x080483ea      83e4f0         and esp, 0xfffffff0
|           0x080483ed      b800000000     mov eax, 0
|           0x080483f2      83c00f         add eax, 0xf                ; 15
|           0x080483f5      83c00f         add eax, 0xf                ; 15
|           0x080483f8      c1e804         shr eax, 4
|           0x080483fb      c1e004         shl eax, 4
|           0x080483fe      29c4           sub esp, eax
|           0x08048400      c70424288504.  mov dword [esp], str.IOLI_Crackme_Level_0x01 ; [0x8048528:4]=0x494c4f49 ; "IOLI Crackme Level 0x01\n"
|           0x08048407      e810ffffff     call sym.imp.printf         ; int printf(const char *format)
|           0x0804840c      c70424418504.  mov dword [esp], str.Password: ; [0x8048541:4]=0x73736150 ; "Password: "
|           0x08048413      e804ffffff     call sym.imp.printf         ; int printf(const char *format)
|           0x08048418      8d45fc         lea eax, [var_4h]
|           0x0804841b      89442404       mov dword [var_sp_4h], eax
|           0x0804841f      c704244c8504.  mov dword [esp], 0x804854c  ; [0x804854c:4]=0x49006425
|           0x08048426      e8e1feffff     call sym.imp.scanf          ; int scanf(const char *format)
|           0x0804842b      817dfc9a1400.  cmp dword [var_4h], 0x149a
```

```
|       ,=< 0x08048432      740e           je 0x8048442
|       |   0x08048434      c704244f8504.  mov dword [esp], str.Invalid_Password ; [0x804854f:4]=0x61766e49 ; "Invalid Password!\n"
|       |   0x0804843b      e8dcfeffff     call sym.imp.printf         ; int printf(const char *format)
|      ,==< 0x08048440      eb0c           jmp 0x804844e
|      |`-> 0x08048442      c70424628504.  mov dword [esp], str.Password_OK_: ; [0x8048562:4]=0x73736150 ; "Password OK :)\n"
|      |    0x08048449      e8cefeffff     call sym.imp.printf         ; int printf(const char *format)
|      |    ; CODE XREF from main @ 0x8048440
|      `--> 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 display 0x149a in another numeric base.

```
[0x08048330]> ? 0x149a
int32   5274
uint32  5274
hex     0x149a
octal   012232
unit    5.2K
segment 0000:049a
string  "\x9a\x14"
fvalue: 5274.0
float:  0.000000f
double: 0.000000
binary  0b0001010010011010
trits   0t21020100
```

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.

IOLI 0x02

This is the third one.

```
$ ./crackme0x02
IOLI Crackme Level 0x02
Password: hello
Invalid Password!
```

check it with rabin2.

```
$ rabin2 -z ./crackme0x02
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x00000548 0x08048548 24  25   .rodata ascii IOLI Crackme Level 0x02\n
1   0x00000561 0x08048561 10  11   .rodata ascii Password:
2   0x0000056f 0x0804856f 15  16   .rodata ascii Password OK :)\n
3   0x0000057f 0x0804857f 18  19   .rodata ascii Invalid Password!\n
```

similar to 0x01, no explicity password string here. so it's time to analyze it with r2.

```
[0x08048330]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x08048330]> pdf@main
            ; DATA XREF from entry0 @ 0x8048347
/ 144: int main (int argc, char **argv, char **envp);
|           ; var int32_t var_ch @ ebp-0xc
|           ; var int32_t var_8h @ ebp-0x8
|           ; var int32_t var_4h @ ebp-0x4
|           ; var int32_t var_sp_4h @ esp+0x4
|           0x080483e4      55             push ebp
|           0x080483e5      89e5           mov ebp, esp
|           0x080483e7      83ec18         sub esp, 0x18
|           0x080483ea      83e4f0         and esp, 0xfffffff0
|           0x080483ed      b800000000     mov eax, 0
```

```
|           0x080483f2      83c00f         add eax, 0xf                ; 15
|           0x080483f5      83c00f         add eax, 0xf                ; 15
|           0x080483f8      c1e804         shr eax, 4
|           0x080483fb      c1e004         shl eax, 4
|           0x080483fe      29c4           sub esp, eax
|           0x08048400      c70424488504.  mov dword [esp], str.IOLI_Crackme_Level_0x02 ; [0x8048548:4]=0x494c4f49 ; "IOLI Crackme Level 0x02\n"
|           0x08048407      e810ffffff     call sym.imp.printf         ; int printf(const char *format)
|           0x0804840c      c70424618504.  mov dword [esp], str.Password: ; [0x8048561:4]=0x73736150 ; "Password: "
|           0x08048413      e804ffffff     call sym.imp.printf         ; int printf(const char *format)
|           0x08048418      8d45fc         lea eax, [var_4h]
|           0x0804841b      89442404       mov dword [var_sp_4h], eax
|           0x0804841f      c704246c8504.  mov dword [esp], 0x804856c  ; [0x804856c:4]=0x50006425
|           0x08048426      e8e1feffff     call sym.imp.scanf          ; int scanf(const char *format)
|           0x0804842b      c745f85a0000.  mov dword [var_8h], 0x5a    ; 'Z' ; 90
|           0x08048432      c745f4ec0100.  mov dword [var_ch], 0x1ec   ; 492
|           0x08048439      8b55f4         mov edx, dword [var_ch]
|           0x0804843c      8d45f8         lea eax, [var_8h]
|           0x0804843f      0110           add dword [eax], edx
|           0x08048441      8b45f8         mov eax, dword [var_8h]
|           0x08048444      0faf45f8       imul eax, dword [var_8h]
|           0x08048448      8945f4         mov dword [var_ch], eax
|           0x0804844b      8b45fc         mov eax, dword [var_4h]
|           0x0804844e      3b45f4         cmp eax, dword [var_ch]
|       ,=< 0x08048451      750e           jne 0x8048461
|       |   0x08048453      c704246f8504.  mov dword [esp], str.Password_OK_: ; [0x804856f:4]=0x73736150 ; "Password OK :)\n"
|       |   0x0804845a      e8bdfeffff     call sym.imp.printf         ; int printf(const char *format)
|      ,==< 0x0804845f      eb0c           jmp 0x804846d
|      |`-> 0x08048461      c704247f8504.  mov dword [esp], str.Invalid_Password ; [0x804857f:4]=0x61766e49 ; "Invalid Password!\n"
|      |    0x08048468      e8affeffff     call sym.imp.printf         ; int printf(const char *format)
|      |    ; CODE XREF from main @ 0x804845f
|      `--> 0x0804846d      b800000000     mov eax, 0
|           0x08048472      c9             leave
\           0x08048473      c3             ret
```

with the experience of solving crackme0x02, we first locate the position of `cmp` instruction by using this simple oneliner:

```
[0x08048330]> pdf@main | grep cmp
|           0x0804844e      3b45f4         cmp eax, dword [var_ch]
```

Unfortunately, the variable compared to eax is stored in the stack. we can't check the value of this variable directly. It's a common case in reverse engineerning that we have to derive the value of the variable from the previous sequence. As the amount of code is relatively small, it's possible.

for example:

```
|           0x080483ed      b800000000     mov eax, 0
|           0x080483f2      83c00f         add eax, 0xf                ; 15
|           0x080483f5      83c00f         add eax, 0xf                ; 15
|           0x080483f8      c1e804         shr eax, 4
|           0x080483fb      c1e004         shl eax, 4
|           0x080483fe      29c4           sub esp, eax
```

we can easily get the value of eax. it's 0x16.

It gets hard when the scale of program grows. radare2 provides a pseudo disassembler output in C-like syntax. It may be useful.

```
[0x08048330]> pdc@main
function main () {
    //  4 basic blocks

    loc_0x80483e4:

         //DATA XREF from entry0 @ 0x8048347
       push ebp
       ebp = esp
       esp -= 0x18
       esp &= 0xfffffff0
       eax = 0
       eax += 0xf               //15
       eax += 0xf               //15
       eax >>>= 4
       eax <<<= 4
       esp -= eax
       dword [esp] = "IOLI Crackme Level 0x02\n" //[0x8048548:4]=0x494c4f49 ; str.IOLI_Crackme_Level_0x02 ; const char *format

       int printf("IOLI Crackme Level 0x02\n")
       dword [esp] = "Password: " //[0x8048561:4]=0x73736150 ; str.Password: ; const char *format

       int printf("Password: ")
       eax = var_4h
       dword [var_sp_4h] = eax
       dword [esp] = 0x804856c  //[0x804856c:4]=0x50006425 ; const char *format
       int scanf("%d")
                               //sym.imp.scanf ()
       dword [var_8h] = 0x5a    //'Z' ; 90
       dword [var_ch] = 0x1ec   //492
       edx = dword [var_ch]
       eax = var_8h             //"Z"
       dword [eax] += edx
       eax = dword [var_8h]
       eax = eax * dword [var_8h]
       dword [var_ch] = eax
       eax = dword [var_4h]
       var = eax - dword [var_ch]
       if (var) goto 0x8048461  //likely
       {
        loc_0x8048461:
```

```
            //CODE XREF from main @ 0x8048451
            dword [esp] = s"Invalid Password!\n"//[0x804857f:4]=0x61766e49 ; str.Invalid_Password ; const char *format

            int printf("Invalid ")
        do
        {
             loc_0x804846d:

                //CODE XREF from main @ 0x804845f
                eax = 0
                leave                     //(pstr 0x0804857f) "Invalid Password!\n" ebp ; str.Invalid_Password
                return
            } while (?);
        } while (?);
       }
       return;

}
```

The `pdc` command is unreliable especially in processing loops (while, for, etc.). So I prefer to use the r2dec plugin in r2 repo to generate the pseudo C code. you can install it easily:

```
r2pm install r2dec
```

decompile `main()` with the following command (like `F5` in IDA):

```
[0x08048330]> pdd@main
/* r2dec pseudo code output */
/* ./crackme0x02 @ 0x80483e4 */
#include <stdint.h>

int32_t main (void) {
    uint32_t var_ch;
    int32_t var_8h;
    int32_t var_4h;
    int32_t var_sp_4h;
    eax = 0;
    eax += 0xf;
    eax += 0xf;
    eax >>= 4;
    eax <<= 4;
    printf ("IOLI Crackme Level 0x02\n");
    printf ("Password: ");
    eax = &var_4h;
    *((esp + 4)) = eax;
    scanf (0x804856c);
    var_8h = 0x5a;
    var_ch = 0x1ec;
    edx = 0x1ec;
    eax = &var_8h;
    *(eax) += edx;
    eax = var_8h;
    eax *= var_8h;
    var_ch = eax;
    eax = var_4h;
    if (eax == var_ch) {
        printf ("Password OK :)\n");
    } else {
        printf ("Invalid Password!\n");
    }
    eax = 0;
    return eax;
}
```

It's more human-readable now. To check the string in 0x804856c, we can: * seek * print string

```
[0x08048330]> s 0x804856c
[0x0804856c]> ps
%d
```

it's exactly the format string of `scanf()`. But r2dec does not recognize the second argument (eax) which is a pointer. it points to var_4h and means out input will store in var_4h.

we can easily write out pseudo code here.

```
var_ch = (var_8h + var_ch)^2;
if (var_ch == our_input)
  printf("Password OK :)\n");
```

given the initial status that var_8h is 0x5a, var_ch is 0x1ec, we have var_ch = 338724 (0x52b24):

```
$ rax2 '=10' '(0x5a+0x1ec)*(0x5a+0x1ec)'
338724

$ ./crackme0x02
IOLI Crackme Level 0x02
Password: 338724
Password OK :)
```

and we finish the crackme0x02.

IOLI 0x03

crackme 0x03, let's skip the string check part and analyze it directly.

```
[0x08048360]> aaa
[0x08048360]> pdd@sym.main
```

```
/* r2dec pseudo code output */
/* ./crackme0x03 @ 0x8048498 */
#include <stdint.h>

int32_t main (void) {
    int32_t var_ch;
    int32_t var_8h;
    int32_t var_4h;
    int32_t var_sp_4h;
    eax = 0;
    eax += 0xf;
    eax += 0xf;
    eax >>= 4;
    eax <<= 4;
    printf ("IOLI Crackme Level 0x03\n");
    printf ("Password: ");
    eax = &var_4h;
    scanf (0x8048634, eax);
    var_8h = 0x5a;
    var_ch = 0x1ec;
    edx = 0x1ec;
    eax = &var_8h;
    *(eax) += edx;
    eax = var_8h;
    eax *= var_8h;
    var_ch = eax;
    eax = var_4h;
    test (eax, eax);
    eax = 0;
    return eax;
}
```

It looks straightforward except the function `test(eax, eax)`. This is unusual to call a function with same two parameters , so I speculate that the decompiler has gone wrong. we can check it in disassembly.

```
[0x08048360]> pdf@sym.main
...
           0x080484fc      8945f4         mov dword [var_ch], eax
           0x080484ff      8b45f4         mov eax, dword [var_ch]
           0x08048502      89442404       mov dword [var_sp_4h], eax   ; uint32_t arg_ch
           0x08048506      8b45fc         mov eax, dword [var_4h]
           0x08048509      890424         mov dword [esp], eax         ; int32_t arg_8h
           0x0804850c      e85dffffff     call sym.test
...
```

Here comes the`sym.test`, called with two parameters. One is var_4h (our input from `scanf()`). The other is var_ch. The value of var_ch (as the parameter of `test()`) can be calculated like it did in crackme_0x02. It's 0x52b24. Try it!

```
./crackme0x03
IOLI Crackme Level 0x03
Password: 338724
Password OK!!! :)
```

Take a look at `sym.test`. It's a two path conditional jump which compares two parameters and then do shift. We can guess that shift is most likely the decryption part (shift cipher, e.g. Caesar cipher).

```
/* r2dec pseudo code output */
/* ./crackme0x03 @ 0x804846e */
#include <stdint.h>

int32_t test (int32_t arg_8h, uint32_t arg_ch) {
    eax = arg_8h;
    if (eax != arg_ch) {
        shift ("Lqydolg#Sdvvzrug$");
    } else {
        shift ("Sdvvzrug#RN$$$#=,");
    }
    return eax;
}
```

can also reverse `shift()` to satisfy curiosity.

```
[0x08048360]> pdf@sym.shift
        ; CODE (CALL) XREF 0x08048491 (sym.test)
        ; CODE (CALL) XREF 0x08048483 (sym.test)
/ function: sym.shift (90)
|       0x08048414  sym.shift:
|       0x08048414     55               push ebp
|       0x08048415     89e5             mov ebp, esp
|       0x08048417     81ec98000000     sub esp, 0x98
|       0x0804841d     c7458400000000   mov dword [ebp-0x7c], 0x0  ; this seems to be a counter
|  .    ; CODE (JMP) XREF 0x0804844e (sym.shift)
/ loc: loc.08048424 (74)
|  .    0x08048424  loc.08048424:
|  .--> 0x08048424     8b4508           mov eax, [ebp+0x8] ; ebp+0x8 = strlen(chain)
|  |    0x08048427     890424           mov [esp], eax
|  |    0x0804842a     e811ffffff       call dword imp.strlen
|  |       ; imp.strlen()
|  |    0x0804842f     394584           cmp [ebp-0x7c], eax
|  |,=< 0x08048432     731c             jae loc.08048450
|  ||   0x08048434     8d4588           lea eax, [ebp-0x78]
|  ||   0x08048437     89c2             mov edx, eax
|  ||   0x08048439     035584           add edx, [ebp-0x7c]
|  ||   0x0804843c     8b4584           mov eax, [ebp-0x7c]
|  ||   0x0804843f     034508           add eax, [ebp+0x8]
|  ||   0x08048442     0fb600           movzx eax, byte [eax]
```

```
|  ||   0x08048445     2c03             sub al, 0x3
|  ||   0x08048447     8802             mov [edx], al
|  ||   0x08048449     8d4584           lea eax, [ebp-0x7c]
|  ||   0x0804844c     ff00             inc dword [eax]
|  `==< 0x0804844e     ebd4             jmp loc.08048424
|   |   ; CODE (JMP) XREF 0x08048432 (sym.shift)
/ loc: loc.08048450 (30)
|   |   0x08048450  loc.08048450:
|   `-> 0x08048450     8d4588           lea eax, [ebp-0x78]
|       0x08048453     034584           add eax, [ebp-0x7c]
|       0x08048456     c60000           mov byte [eax], 0x0
|       0x08048459     8d4588           lea eax, [ebp-0x78]
|       0x0804845c     89442404         mov [esp+0x4], eax
|       0x08048460     c70424e8850408   mov dword [esp], 0x80485e8
|       0x08048467     e8e4feffff       call dword imp.printf
|          ; imp.printf()
|       0x0804846c     c9               leave
\       0x0804846d     c3               ret
        ; ------------
```

you can read the assembly code and find the decryption is actually a "sub al, 0x3". we can write a python script for it:

```
print(''.join([chr(ord(i)-0x3) for i in 'SdvvzrugRN$$$']))
print(''.join([chr(ord(i)-0x3) for i in 'LqydolgSdvvzrug$']))
```

the easier way is to **run** the decryption code, that means debug it or emulate it. I used radare2 ESIL emulator but it got stuck when executed `call dword imp.strlen`. And I can't find the usage of hooking function / skip instruction in radare2. The following is an example to show u how to emulate ESIL.

```
[0x08048414]> s 0x08048445      # the 'sub al, 0x03'
[0x08048445]> aei               # init VM
[0x08048445]> aeim              # init memory
[0x08048445]> aeip              # init ip
[0x08048445]> aer eax=0x41      # set eax=0x41 -- 'A'
[0x08048445]> aer               # show current value of regs
oeax = 0x00000000
eax = 0x00000041
ebx = 0x00000000
ecx = 0x00000000
edx = 0x00000000
esi = 0x00000000
edi = 0x00000000
esp = 0x00178000
ebp = 0x00178000
eip = 0x08048445
eflags = 0x00000000
[0x08048445]> V                 # enter Visual mode
# 'p' or 'P' to change visual mode
# I prefer the [xaDvc] mode
# use 's' to step in and 'S' to step over
[0x08048442 [xaDvc]0 0% 265 ./crackme0x03]> diq;?0;f t.. @ sym.shift+46 # 0x8048442
dead at 0x00000000
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00178000  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00178010  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00178020  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00178030  0000 0000 0000 0000 0000 0000 0000 0000  ................
   oeax 0x00000000      eax 0x00000041      ebx 0x00000000      ecx 0x00000000
    edx 0x00000000      esi 0x00000000      edi 0x00000000      esp 0x00178000
    ebp 0x00178000      eip 0x08048445   eflags 0x00000000
        :   0x08048442      0fb600         movzx eax, byte [eax]
        :   ;-- eip:
        :   0x08048445      2c03           sub al, 3
        :   0x08048447      8802           mov byte [edx], al
        :   0x08048449      8d4584         lea eax, [var_7ch]
        :   0x0804844c      ff00           inc dword [eax]
        :=< 0x0804844e      ebd4           jmp 0x8048424
            ; CODE XREF from sym.shift @ 0x8048432
            0x08048450      8d4588         lea eax, [var_78h]
```

By the way, u can also open the file and use write data command to decrypt data.

```
r2 -w ./crackme0x03
[0x08048360]> aaa
[0x08048360]> fs strings
[0x08048360]> f
0x080485ec 18 str.Lqydolg_Sdvvzrug
0x080485fe 18 str.Sdvvzrug_RN
0x08048610 25 str.IOLI_Crackme_Level_0x03
0x08048629 11 str.Password:
[0x08048360]> s str.Lqydolg_Sdvvzrug
[0x080485ec]> wos 0x03 @ str.Lqydolg_Sdvvzrug!0x11
[0x080485ec]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x080485ec  496e 7661 6c69 6420 5061 7373 776f 7264  Invalid Password
0x080485fc  2100 5364 7676 7a72 7567 2352 4e24 2424  !.Sdvvzrug#RN$$$
0x0804860c  233d 2c00 494f 4c49 2043 7261 636b 6d65  #=,.IOLI Crackme
0x0804861c  204c 6576 656c 2030 7830 330a 0050 6173   Level 0x03..Pas
0x0804862c  7377 6f72 643a 2000 2564 0000 0000 0000  sword: .%d......
[0x080485ec]> wos 0x03 @ str.Sdvvzrug_RN!17
[0x080485ec]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x080485ec  496e 7661 6c69 6420 5061 7373 776f 7264  Invalid Password
0x080485fc  2100 5061 7373 776f 7264 204f 4b21 2121  !.Password OK!!!
0x0804860c  203a 2900 494f 4c49 2043 7261 636b 6d65   :).IOLI Crackme
0x0804861c  204c 6576 656c 2030 7830 330a 0050 6173   Level 0x03..Pas
```

```
0x0804862c  7377 6f72 643a 2000 2564 0000 0000 0000  sword: .%d......
[0x080485ec]>
```

IOLI 0x04

0x04

```
[0x080483d0]> pdd@main
/* r2dec pseudo code output */
/* ./crackme0x04 @ 0x8048509 */
#include <stdint.h>

int32_t main (void) {
    int32_t var_78h;
    int32_t var_4h;
    eax = 0;
    eax += 0xf;
    eax += 0xf;
    eax >>= 4;
    eax <<= 4;
    printf ("IOLI Crackme Level 0x04\n");
    printf ("Password: ");
    eax = &var_78h;
    scanf (0x8048682, eax);
    eax = &var_78h;
    check (eax);
    eax = 0;
    return eax;
}
```

Let's enter check.

```
#include <stdint.h>

int32_t check (char * s) {
    char * var_dh;
    uint32_t var_ch;
    uint32_t var_8h;
    int32_t var_4h;
    char * format;
    int32_t var_sp_8h;
    var_8h = 0;
    var_ch = 0;
    do {
        eax = s;
        eax = strlen (eax);
        if (var_ch >= eax) {
            goto label_0;
        }
        eax = var_ch;
        eax += s;
        eax = *(eax);
        var_dh = al;
        eax = &var_4h;
        eax = &var_dh;
        sscanf (eax, eax, 0x8048638);
        edx = var_4h;
        eax = &var_8h;
        *(eax) += edx;
        if (var_8h == 0xf) {
            printf ("Password OK!\n");
            exit (0);
        }
        eax = &var_ch;
        *(eax)++;
    } while (1);
label_0:
    printf ("Password Incorrect!\n");
    return eax;
}
```

manually analyze with both the assembly and pseudo code we can simply write down the C-like code to describe this function:

```
#include <stdint.h>
int32_t check(char *s)
{
    var_ch = 0;
    var_8h = 0;
    for (var_ch = 0; var_ch < strlen(s); ++var_ch)
    {
        var_dh = s[var_ch];
        sscanf(&var_dh, %d, &var_4h);           // read from string[var_ch], store to var_4h
        var_8h += var_4h;
        if(var_8h == 0xf)
            printf("Password OK\n");
    }
    printf("Password Incorrect!\n");
    return 0;
}
```

In short, it calculates the Digit Sum of a number (add a number digit by digit. for example, 96 => 9 + 6 = 15) :

```
./crackme0x04
IOLI Crackme Level 0x04
```

```
Password: 12345
Password OK!

./crackme0x04
IOLI Crackme Level 0x04
Password: 96
Password OK!
```

IOLI 0x05

check again, it uses `scanf()` to get our input and pass it to `check()` as parameter.

```
[0x080483d0]> pdd@main
/* r2dec pseudo code output */
/* ./crackme0x05 @ 0x8048540 */
#include <stdint.h>

int32_t main (void) {
    int32_t var_78h;
    int32_t var_4h;
    eax = 0;
    eax += 0xf;
    eax += 0xf;
    eax >>= 4;
    eax <<= 4;
    printf ("IOLI Crackme Level 0x05\n");
    printf ("Password: ");
    eax = &var_78h;
    scanf (0x80486b2, eax);          // 0x80486b2 is %s
    eax = &var_78h;
    check (eax);
    eax = 0;
    return eax;
}
```

the check() function:

```
/* r2dec pseudo code output */
/* ./crackme0x05 @ 0x80484c8 */
#include <stdint.h>

int32_t check (char * s) {
    char * var_dh;
    uint32_t var_ch;
    uint32_t var_8h;
    int32_t var_4h;
    char * format;
    int32_t var_sp_8h;
    var_8h = 0;
    var_ch = 0;
    do {
        eax = s;
        eax = strlen (eax);
        if (var_ch >= eax) {
            goto label_0;
        }
        eax = var_ch;
        eax += s;
        eax = *(eax);
        var_dh = al;
        eax = &var_4h;
        eax = &var_dh;
        sscanf (eax, eax, 0x8048668);           // 0x8048668 is %d
        edx = var_4h;
        eax = &var_8h;
        *(eax) += edx;
        if (var_8h == 0x10) {
            eax = s;
            parell (eax);
        }
        eax = &var_ch;
        *(eax)++;
    } while (1);
label_0:
    printf ("Password Incorrect!\n");
    return eax;
}
```

The same, we can write our own C-like pseudo code.

```
#include <stdint.h>
int32_t check(char *s)
{
    var_ch = 0;
    var_8h = 0;
    for (var_ch = 0; var_ch < strlen(s); ++var_ch)
    {
        var_dh = s[var_ch];
        sscanf(&var_dh, %d, &var_4h);           // read from string[var_ch], store to var_4h
        var_8h += var_4h;
        if(var_8h == 0x10)
            parell(s);
    }
    printf("Password Incorrect!\n");
```

```
    return 0;
}
```

The if condition becomes `var_8h == 0x10`. In addition, a new function call - `parell(s)` replace the `printf("password OK")`now. The next step is to reverse sym.parell.

```
[0x08048484]> s sym.parell
[0x08048484]> pdd@sym.parell
/* r2dec pseudo code output */
/* ./crackme0x05 @ 0x8048484 */
#include <stdint.h>

uint32_t parell (char * s) {
    int32_t var_4h;
    char * format;
    int32_t var_8h;
    eax = &var_4h;
    eax = s;
    sscanf (eax, eax, 0x8048668);
    eax = var_4h;
    eax &= 1;
    if (eax == 0) {
        printf ("Password OK!\n");
        exit (0);
    }
    return eax;
}
```

the decompiled code looks well except the `sscanf()` function. It can be easily corrected by checking the assembly code.

```
/ 68: sym.parell (int32_t arg_8h);
|           ; var int32_t var_4h @ ebp-0x4
|           ; arg int32_t arg_8h @ ebp+0x8
|           ; var int32_t var_sp_4h @ esp+0x4
|           ; var int32_t var_8h @ esp+0x8
|           0x08048484      55             push ebp
|           0x08048485      89e5           mov ebp, esp
|           0x08048487      83ec18         sub esp, 0x18
|           0x0804848a      8d45fc         lea eax, [var_4h]
|           0x0804848d      89442408       mov dword [var_8h], eax
|           0x08048491      c74424046886.  mov dword [var_sp_4h], 0x8048668 ; [0x8048668:4]=0x50006425 %d
|           0x08048499      8b4508         mov eax, dword [arg_8h]
|           0x0804849c      890424         mov dword [esp], eax
|           0x0804849f      e800ffffff     call sym.imp.sscanf         ; int sscanf(const char *s, const char *format,   ...)
....
```

The `mov dword [esp], eax` is the nearest instruction to sscanf (and it's equivalent to a push instruction). It stores the string 's' to the stack top (arg1). `mov dword [var_sp_4h], 0x8048668` push '%d' as arg2 into stack. var_8h (esp + 0x8) which keeps the address of var_4h is the arg3.

Finally we have the corrected pseudo code:

```
uint32_t parell (char * s) {
    sscanf (s, %d, &var_4h);
    if ((var_4h & 1) == 0) {
        printf ("Password OK!\n");
        exit(0);
    }
    return 0;
}
```

Now there are 2 constraints:

- Digit Sum is 16 (0x10)
- Must be an odd number (1 & number == 0)

The password is at our fingertips now.

```
./crackme0x05
IOLI Crackme Level 0x05
Password: 88
Password OK!

./crackme0x05
IOLI Crackme Level 0x05
Password: 12346
Password OK!
```

we can also use angr to solve it since we have two constraints to the password.

IOLI 0x06

nearly a routine to check this binary (not complete output in the following):

```
rabin2 -z ./crackme0x06
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x00000738 0x08048738 4   5    .rodata ascii LOLO
1   0x00000740 0x08048740 13  14   .rodata ascii Password OK!\n
2   0x0000074e 0x0804874e 20  21   .rodata ascii Password Incorrect!\n
3   0x00000763 0x08048763 24  25   .rodata ascii IOLI Crackme Level 0x06\n
4   0x0000077c 0x0804877c 10  11   .rodata ascii Password:

rabin2 -I ./crackme0x06
arch     x86
baddr    0x8048000
bintype  elf
bits     32
compiler GCC: (GNU) 3.4.6 (Gentoo 3.4.6-r2, ssp-3.4.6-1.0, pie-8.7.10)
```

```
crypto   false
endian   little
havecode true
lang     c
machine  Intel 80386
maxopsz  16
minopsz  1
os       linux
static   false
va       true
```

and analyze it then decompile main

```
[0x08048400]> pdd@main
/* r2dec pseudo code output */
/* ./crackme0x06 @ 0x8048607 */
#include <stdint.h>

int32_t main (int32_t arg_10h) {
    int32_t var_78h;
    int32_t var_4h;
    // adjusting stack
    eax = 0;
    eax += 0xf;
    eax += 0xf;
    eax >>= 4;
    eax <<= 4;

    // main logic
    printf ("IOLI Crackme Level 0x06\n");
    printf ("Password: ");
    eax = &var_78h;
    scanf (0x8048787, eax);
    eax = arg_10h;
    eax = &var_78h;
    check (eax, arg_10h);
    eax = 0;
    return eax;
}
```

main has 3 arguments `argc, argv, envp`, and this program is compiled with GCC, so the stack should be like this :

```
[esp + 0x10] - envp
[esp + 0x0c] - argv
[esp + 0x08] - argc
[esp + 0x04] - return address
```

enter the `check()` and decompile it. this function is different from 0x05 now. but they still have similar code structure.

```
int32_t check (char * s, int32_t arg_ch) {
    char * var_dh;
    uint32_t var_ch;
    uint32_t var_8h;
    int32_t var_4h;
    char * format;
    int32_t var_sp_8h;
    var_8h = 0;
    var_ch = 0;
    do {
        eax = s;
        eax = strlen (eax);
        if (var_ch >= eax) {
            goto label_0;
        }
        eax = var_ch;
        eax += s;
        eax = *(eax);
        var_dh = al;
        eax = &var_4h;
        eax = &var_dh;
        sscanf (eax, eax, 0x804873d);
        edx = var_4h;
        eax = &var_8h;
        *(eax) += edx;
        if (var_8h == 0x10) {
            eax = arg_ch;
            eax = s;
            parell (eax, arg_ch);
        }
        eax = &var_ch;
        *(eax)++;
    } while (1);
label_0:
    printf ("Password Incorrect!\n");
    return eax;
}
```

Correct the `sscanf` part and `parell` part, both of them were generated incorrectly:

```
int32_t check (char * s, void* envp)
{
    var_ch = 0;
    var_8h = 0;
    for (var_ch = 0; var_ch < strlen(s); ++var_ch)
    {
        var_dh = s[var_ch];
```

```
        sscanf(&var_dh, %d, &var_4h);           // read from string[var_ch], store to var_4h
        var_8h += var_4h;
        if(var_8h == 0x10)
            parell(s, envp);
    }
    printf("Password Incorrect!\n");
    return 0;
}
```

no more info, we have to reverse `parell()` again.

```
#include <stdint.h>

uint32_t parell (char * s, char * arg_ch) {
    sscanf (s, %d, &var_4h);

    if (dummy (var_4h, arg_ch) == 0)
        return 0;

    for (var_bp_8h = 0; var_bp_8h <= 9; ++var_bp_8h){
        if (var_4h & 1 == 0){
            printf("Password OK!\n");
            exit(0);
        }
    }

    return 0;
}
```

well, there is a new check condition in `parell()` – `dummy (var_4h, arg_ch) == 0`. then reverse dummy!

```
[0x080484b4]> pdd@sym.dummy
/* r2dec pseudo code output */
/* ./crackme0x06 @ 0x80484b4 */
#include <stdint.h>

int32_t dummy (char ** s1) {
    int32_t var_8h;
    int32_t var_4h;
    char * s2;
    size_t * n;
    var_4h = 0;
    do {
        eax = 0;
        edx = eax*4;
        eax = s1;
        if (*((edx + eax)) == 0) {
            goto label_0;
        }
        eax = var_4h;
        ecx = eax*4;
        edx = s1;
        eax = &var_4h;
        *(eax)++;
        eax = *((ecx + edx));
        eax = strncmp (eax, 3, "LOLO");
    } while (eax != 0);
    var_8h = 1;
    goto label_1;
label_0:
    var_8h = 0;
label_1:
    eax = 0;
    return eax;
}
```

looks like a loop to process string. we can beautify it.

```
[0x080484b4]> pdd@sym.dummy
/* r2dec pseudo code output */
/* ./crackme0x06 @ 0x80484b4 */
#include <stdint.h>

int32_t dummy (char ** s1) {
    for (var_4h = 0; strncmp(s1[var_4h], "LOLO", 3) != 0; var_4h++){
        if (s1[i] == NULL)
            return 0;
    }
    return 1;
}
```

There are 3 constraints to crackme_0x06:

- Digit Sum
- Odd Number
- should have an environment variable whose name started with "LOL".

```
$ ./crackme0x06
IOLI Crackme Level 0x06
Password: 12346
Password Incorrect!
$ export LOLAA=help
$ ./cracke0x06
IOLI Crackme Level 0x06
Password: 12346
Password OK!
```

IOLI 0x07

a weird "wtf?" string.

```
$ rabin2 -z ./crackme0x07
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x000007a8 0x080487a8 4   5    .rodata ascii LOLO
1   0x000007ad 0x080487ad 20  21   .rodata ascii Password Incorrect!\n
2   0x000007c5 0x080487c5 13  14   .rodata ascii Password OK!\n
3   0x000007d3 0x080487d3 5   6    .rodata ascii wtf?\n
4   0x000007d9 0x080487d9 24  25   .rodata ascii IOLI Crackme Level 0x07\n
5   0x000007f2 0x080487f2 10  11   .rodata ascii Password:
```

again, no password string or compare in `main()`. I put the simplified pseudo code here. var_78h is likely to a char *pointer (string) .

```
#include <stdint.h>
int32_t main (int32_t arg_10h) {
    printf ("IOLI Crackme Level 0x07\n");
    printf ("Password: ");
    scanf (%s, &var_78h);
    return fcn_080485b9 (&var_78h, arg_10h);
}
```

due to the symbol info lost, neither `aa` nor `aaa` show the name of functions. we can double check this in "flagspace". Radare2 use fcn_080485b9 as the function name. It's a common case in reverse engineering that we don't have any symbol info of the binary.

```
[0x080487fd]> fs symbols
[0x080487fd]> f
0x08048400 33 entry0
0x0804867d 92 main
0x080487a4 4 obj._IO_stdin_used
```

decompile the `fcn_080485b9()`:

```
[0x080485b9]> pdfc
            ; CALL XREF from main @ 0x80486d4
/ 118: fcn.080485b9 (char *s, int32_t arg_ch);
|           ; var char *var_dh @ ebp-0xd
|           ; var signed int var_ch  { >= 0xffffffffffffffff} @ ebp-0xc
|           ; var uint32_t var_8h @ ebp-0x8
|           ; var int32_t var_bp_4h @ ebp-0x4
|           ; arg char *s @ ebp+0x8
|           ; arg int32_t arg_ch @ ebp+0xc
|           ; var char *format @ esp+0x4
|           ; var int32_t var_sp_8h @ esp+0x8
|           0x080485b9      55              push ebp
|           0x080485ba      89e5            mov ebp, esp
|           0x080485bc      83ec28          sub esp, 0x28
|           0x080485bf      c745f8000000.   mov dword [var_8h], 0
|           0x080485c6      c745f4000000.   mov dword [var_ch], 0
|           ; CODE XREF from fcn.080485b9 @ 0x8048628
|       .-> 0x080485cd      8b4508          mov eax, dword [s]
|       :   0x080485d0      890424          mov dword [esp], eax        ; const char *s
|       :   0x080485d3      e8d0fdffff      call sym.imp.strlen         ; size_t strlen(const char *s)
|       :   0x080485d8      3945f4          cmp dword [var_ch], eax
|      ,==< 0x080485db      734d            jae 0x804862a
|      |:   0x080485dd      8b45f4          mov eax, dword [var_ch]
|      |:   0x080485e0      034508          add eax, dword [s]
|      |:   0x080485e3      0fb600          movzx eax, byte [eax]
|      |:   0x080485e6      8845f3          mov byte [var_dh], al
|      |:   0x080485e9      8d45fc          lea eax, [var_bp_4h]
|      |:   0x080485ec      89442408        mov dword [var_sp_8h], eax  ;   ...
|      |:   0x080485f0      c7442404c287.   mov dword [format], 0x80487c2 ; [0x80487c2:4]=0x50006425 ; const char *format
|      |:   ;-- eip:
|      |:   0x080485f8      8d45f3          lea eax, [var_dh]
|      |:   0x080485fb      890424          mov dword [esp], eax        ; const char *s
|      |:   0x080485fe      e8c5fdffff      call sym.imp.sscanf         ; int sscanf(const char *s, const char *format,   ...)
|      |:   0x08048603      8b55fc          mov edx, dword [var_bp_4h]
|      |:   0x08048606      8d45f8          lea eax, [var_8h]
|      |:   0x08048609      0110            add dword [eax], edx
|      |:   0x0804860b      837df810        cmp dword [var_8h], 0x10
|     ,===< 0x0804860f      7512            jne 0x8048623
|     ||:   0x08048611      8b450c          mov eax, dword [arg_ch]
|     ||:   0x08048614      89442404        mov dword [format], eax     ; char *arg_ch
|     ||:   0x08048618      8b4508          mov eax, dword [s]
|     ||:   0x0804861b      890424          mov dword [esp], eax        ; char *s
|     ||:   0x0804861e      e81fffffff      call fcn.08048542
|     ||:   ; CODE XREF from fcn.080485b9 @ 0x804860f
|     `---> 0x08048623      8d45f4          lea eax, [var_ch]
|      |:   0x08048626      ff00            inc dword [eax]
|      |`=< 0x08048628      eba3            jmp 0x80485cd
|      |    ; CODE XREF from fcn.080485b9 @ 0x80485db
\      `--> 0x0804862a      e8f5feffff      call fcn.08048524
```

we got familiar with this code structure in the previous challenges (the check function). It's not difficult for us even we don't have the symbol info. you can also use `afn` command to rename the function name if you like.

```
int32_t fcn_080485b9 (char * s, void* envp)
{
    var_ch = 0;
    var_8h = 0;
    for (var_ch = 0; var_ch < strlen(s); ++var_ch)
    {
        var_dh = s[var_ch];
        sscanf(&var_dh, %d, &var_4h);           // read from string[var_ch], store to var_4h
```

```
        var_8h += var_4h;
        if(var_8h == 0x10)
            fcn_08048542(s, envp);
    }
    return fcn_08048524();
}
```

most part of crackme 0x07 is the same with 0x06. and it can be solved by the same password & environment:

```
$ export LOLAA=help
$ ./cracke0x07
IOLI Crackme Level 0x07
Password: 12346
Password OK!
```

wait … where is the 'wtf?'. Often, we would like to find the cross reference (xref) to strings (or data, functions, etc.) in reverse engineering. The related commands in Radare2 are under "ax" namespace:

```
[0x08048400]> f
0x080487a8 5 str.LOLO
0x080487ad 21 str.Password_Incorrect
0x080487c5 14 str.Password_OK
0x080487d3 6 str.wtf
0x080487d9 25 str.IOLI_Crackme_Level_0x07
0x080487f2 11 str.Password:
[0x08048400]> axt 0x80487d3
(nofunc) 0x804865c [DATA] mov dword [esp], str.wtf
[0x08048400]> axF str.wtf
Finding references of flags matching 'str.wtf'...
[0x001eff28-0x001f0000] (nofunc) 0x804865c [DATA] mov dword [esp], str.wtf
Macro 'findstref' removed.
```

the `[DATA] mov dword [esp], str.wtf` at `0x804865c` is an instruction of fcn.080485b9. But the analysis in my PC ignores the remained instructions and only display the incomplete assembly. the range of fcn.080485b9 should be `0x080485b9 ~ 0x0804867c` . we can reset block size and print opcodes.

```
[0x08040000]> s 0x080485b9
[0x080485b9]> b 230
[0x08048400]> pd
...
            0x0804862f      8b450c         mov eax, dword [ebp + 0xc]
            0x08048632      89442404       mov dword [esp + 4], eax
            0x08048636      8b45fc         mov eax, dword [ebp - 4]
            0x08048639      890424         mov dword [esp], eax        ; char **s1
            0x0804863c      e873feffff     call fcn.080484b4
            0x08048641      85c0           test eax, eax
        ,=< 0x08048643      7436           je 0x804867b
        |   0x08048645      c745f4000000.  mov dword [ebp - 0xc], 0
        |   ; CODE XREF from fcn.080485b9 @ +0xc0
       .--> 0x0804864c      837df409       cmp dword [ebp - 0xc], 9
      ,===< 0x08048650      7f29           jg 0x804867b
      |:|   0x08048652      8b45fc         mov eax, dword [ebp - 4]
      |:|   0x08048655      83e001         and eax, 1
      |:|   0x08048658      85c0           test eax, eax
     ,====< 0x0804865a      7518           jne 0x8048674
     ||:|   0x0804865c      c70424d38704.  mov dword [esp], str.wtf    ; [0x80487d3:4]=0x3f667477 ; "wtf?\n" ; const char *format
     ||:|   0x08048663      e850fdffff     call sym.imp.printf         ; int printf(const char *format)
     ||:|   0x08048668      c70424000000.  mov dword [esp], 0          ; int status
     ||:|   0x0804866f      e874fdffff     call sym.imp.exit           ; void exit(int status)
     ||:|   ; CODE XREF from fcn.080485b9 @ +0xa1
     `----> 0x08048674      8d45f4         lea eax, [ebp - 0xc]
      |:|   0x08048677      ff00           inc dword [eax]
      |`==< 0x08048679      ebd1           jmp 0x804864c
      | |   ; CODE XREFS from fcn.080485b9 @ +0x8a, +0x97
      `-`-> 0x0804867b      c9             leave
            0x0804867c      c3             ret
```

`test eax, ea;je 0x804867b` will jump to `leave; ret`, which forever skips the str.wtf part. only use `aa` to analyze this binary can display the whole function.

IOLI 0x08

we can reverse it and find it's similar to 0x07, and use the same password to solve it:

```
$ export LOLAA=help
$ ./cracke0x08
IOLI Crackme Level 0x08
Password: 12346
Password OK!
```

dustri provided a better way to check crackme0x08. 0x07 is the stripped version of 0x08.

```
$ radiff2 -A -C ./crackme0x07 ./crackme0x08
...
             fcn.08048360  23 0x8048360 |   MATCH  (1.000000) | 0x8048360   23 sym._init
 sym.imp.__libc_start_main   6 0x8048388 |   MATCH  (1.000000) | 0x8048388    6 sym.imp.__libc_start_main
            sym.imp.scanf    6 0x8048398 |   MATCH  (1.000000) | 0x8048398    6 sym.imp.scanf
           sym.imp.strlen    6 0x80483a8 |   MATCH  (1.000000) | 0x80483a8    6 sym.imp.strlen
           sym.imp.printf    6 0x80483b8 |   MATCH  (1.000000) | 0x80483b8    6 sym.imp.printf
           sym.imp.sscanf    6 0x80483c8 |   MATCH  (1.000000) | 0x80483c8    6 sym.imp.sscanf
          sym.imp.strncmp    6 0x80483d8 |   MATCH  (1.000000) | 0x80483d8    6 sym.imp.strncmp
             sym.imp.exit    6 0x80483e8 |   MATCH  (1.000000) | 0x80483e8    6 sym.imp.exit
                   entry0   33 0x8048400 |   MATCH  (1.000000) | 0x8048400   33 entry0
             fcn.08048424   33 0x8048424 |   MATCH  (1.000000) | 0x8048424   33 fcn.08048424
             fcn.08048450   47 0x8048450 |   MATCH  (1.000000) | 0x8048450   47 sym.__do_global_dtors_aux
             fcn.08048480   50 0x8048480 |   MATCH  (1.000000) | 0x8048480   50 sym.frame_dummy
             fcn.080484b4  112 0x80484b4 |   MATCH  (1.000000) | 0x80484b4  112 sym.dummy
             fcn.08048524   30 0x8048524 |   MATCH  (1.000000) | 0x8048524   30 sym.che
             fcn.08048542  119 0x8048542 |   MATCH  (1.000000) | 0x8048542  119 sym.parell
```

```
          fcn.080485b9 118 0x80485b9 |   MATCH  (1.000000) | 0x80485b9  118 sym.check
                  main  92 0x804867d |   MATCH  (1.000000) | 0x804867d   92 main
          fcn.08048755   4 0x8048755 |   MATCH  (1.000000) | 0x8048755    4 sym.__i686.get_pc_thunk.bx
          fcn.08048760  35 0x8048760 |   MATCH  (1.000000) | 0x8048760   35 sym.__do_global_ctors_aux
          fcn.0804878d  17 0x804878d |     NEW  (0.000000)
   sym.__libc_csu_init  99 0x80486e0 |     NEW  (0.000000)
   sym.__libc_csu_fini   5 0x8048750 |     NEW  (0.000000)
             sym._fini  26 0x8048784 |     NEW  (0.000000)
```

IOLI 0x09

Hints: crackme0x09 hides the format string (%d and %s), and nothing more than 0x08.

```
$ export LOLA=help
$ ./crackme0x09
IOLI Crackme Level 0x09
Password: 12346
Password OK!
```

Avatao R3v3rs3 4

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 *aaa* 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:

main functions's minimap

> ***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 *<TAB>* and the *<SHIFT><TAB>* 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:

main bb-0c63

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:

```
[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:

main bb-0ca6

There are a few things happening in the next block:

main bb-0cac

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:

main bb-0cac_meta

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.

main bb-0d1d

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:

main bb-0dde

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*:

main bb-0d34

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

main bb-0d65

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:

main bb-0d3d

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:

main bb-0d61

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

main bb-0d4d

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:

main bb-0d6b

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:

main bb-0d6b_meta

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:

main bb-0d75

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:

main bb-0d80

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
```

vmloop bb-0a45

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,

vmloop bb-0a74

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.

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.

vmloop bb-0c4d

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:

vmloop bb-0a74

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 (
>
> $$*)tocurrentseek + 8 * 0x17(*$$
>
> *+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!

vmloop bb-0c04

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*.

vmloop full

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:

vmloop graph

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.

By the way, 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:

vmloop bb-0a80

vmloop bb-0ab6

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:

vmloop bb-0b42

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:

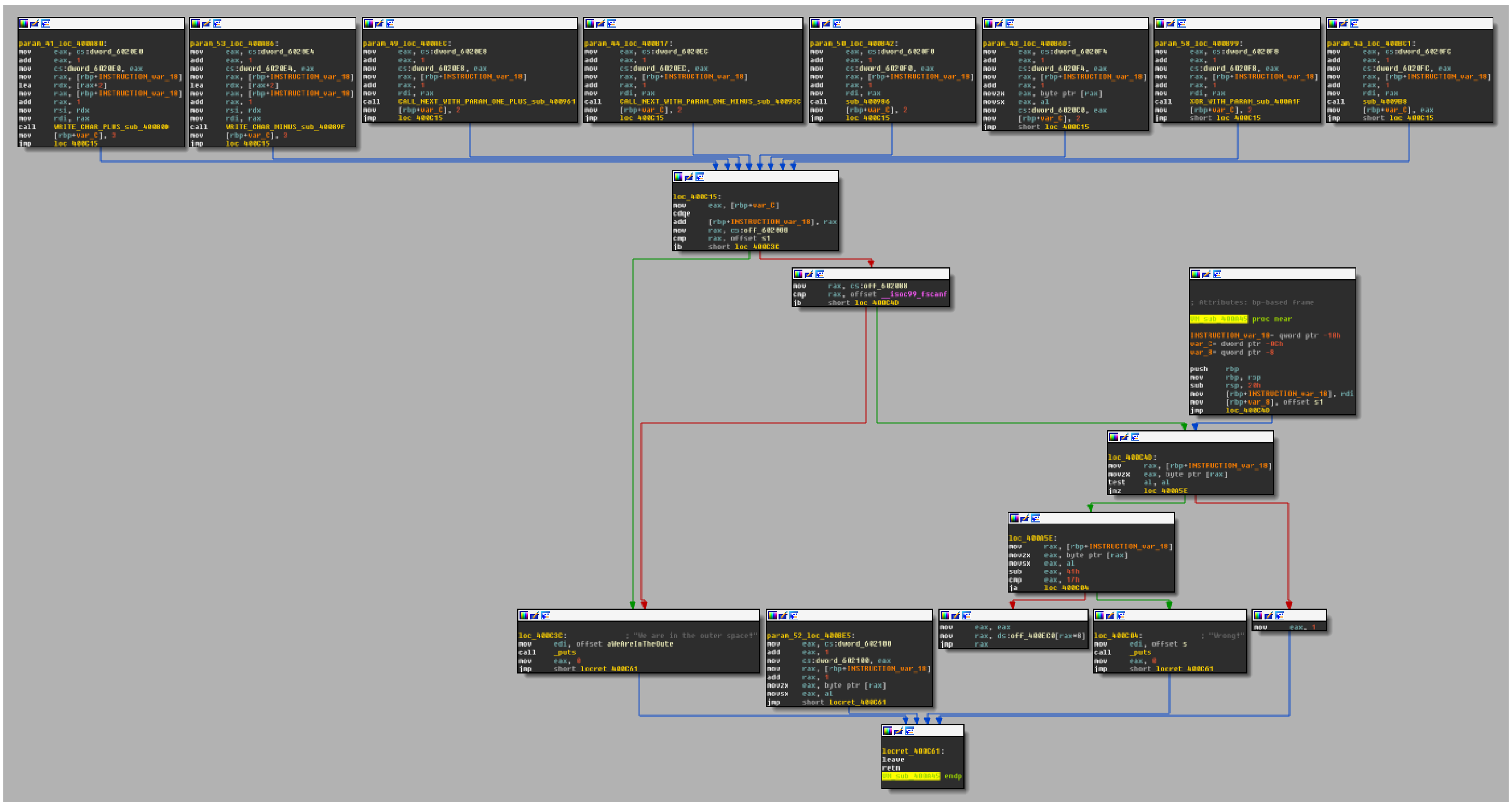

Figure 24: IDA graph

vmloop bb-0aec

And a two parameters example:

vmloop bb-0a80_full

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:

vmloop bb-0bc1

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

vmloop bb-0be5

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:

vmloop bb-0b6d

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 <Enter> 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*!

instr_A minimap

> ***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:

instr_A bb-080d

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".

instr_A switch values

This is the "M" branch:

instr_A switch-M

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:

instr_A switch-P

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:

instr_A switch-C

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 -
```

instr_S graph1

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

instr_S graph2

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

instr_I

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

instr_D

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
```

instr_P

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
```

instr_X

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
```

instr_J

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:

instr_J bb-09e1

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

instr_J bb-09e4

instr_J bb-0a1a

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

instr_J bb-09ef

If it is zero, the function returns 2,

instr_J bb-0a13

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

instr_J bb-09f9

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

instr_J bb-09ff

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

instr_J bb-0a0e

.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 *= -1if (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\x01AMMAP\x01AMuAP\x01AMCAP\x01AMHAP\x01AM AP\x01AMrAP\x01AMeAP\x01AMVAP\x0
```

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

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

Survival Guide

Those are the basic commands you will want to know and use for moving around a binary and getting information about it.

Command	Description
s (tab)	Seek to a different place
x [nbytes]	Hexdump of nbytes, $b by default
aa	Auto analyze
pdf@ funcname	Disassemble function (main, fcn, etc.)
f fcn(Tab)	List functions
f str(Tab)	List strings
fr [flagname] [newname]	Rename flag
psz [offset]~grep	Print strings and grep for one
axF [flag]	Find cross reference for a flag

Flags

Flags are like bookmarks, but they carry some extra information like size, tags or associated flagspace. Use the `f` command to list, set, get them.

Command	Description
f	List flags
fd $$	Describe an offset
fj	Display flags in JSON
fl	Show flag length
fx [flagname]	Show hexdump of flag
fC [name] [comment]	Set flag comment

Flagspaces

Flags are created into a flagspace, by default none is selected, and listing flags will list them all. To display a subset of flags you can use the `fs` command to restrict it.

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

Information

Binary files have information stored inside the headers. The `i` command uses the RBin api and allows us to the same things rabin2 do. Those are the most common ones.

Command	Description
ii	Information on imports
iI	Info on binary
ie	Display entrypoint
iS	Display sections
ir	Display relocations
iz	List strings (izz, izzz)

Print string

There are different ways to represent a string in memory. The `ps` command allows us to print it in utf-16, pascal, zero terminated, .. formats.

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

The visual mode is the standard interactive interface of radare2.

To enter in visual mode use the `v` or `V` command, and then you'll only have to press keys to get the actions happen instead of commands.

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	Toggle asm.pseudo and asm.esil
.	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

Command	Description
i/a/A	(i)nsert hex, (a)ssemble code, visual (A)ssembler
b	Toggle breakpoint
B	Browse evals, symbols, flags, classes, …
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)
C	Toggle (C)olors
R	Randomize color palette (ecr)
tT	Tab related. see also tab
v	Visual code analysis menu
V	(V)iew graph (agv?)
wW	Seek cursor to next/prev word
uU	Undo/redo seek
x	Show xrefs of current func from/to data/code
yY	Copy and paste selection
z	fold/unfold comments in diassembly

Searching

There are many situations where we need to find a value inside a binary or in some specific regions. Use the `e search.in=?` command to choose where the `/` command may search for the given value.

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 a1b2c3	Search for bytes; spaces and uppercase nibbles are allowed, same as /x A1 B2 C3
/x a1..c3	Search for bytes ignoring some nibbles (auto-generates mask, in this example: ff00ff)
/x a1b2:fff3	Search for bytes with mask (specify individual bits)
/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 (Broken)

This feature has broken and not been resolved at the time of writing these words (Nov.16th 2020). check #Issue 6945: META - Project files and #Issue 17034 for more details.

To save your analysis for now, write your own script which records the function name, variable name, etc. for example:

```
vim sample_A.r2

e scr.utf8 = false
s 0x000403ce0
aaa
s fcn.00403130
afn return_delta_to_heapaddr
afvn iter var_04h
...
```

Usable variables in expression

The `?$?` command will display the variables that can be used in any math operation inside the r2 shell. For example, using the `? $$` command to evaluate a number or `?v` to just the value in one format.

All commands in r2 that accept a number supports the use of those variables.

Command	Description
|*here*(*currentvirtualseek*)||$	current non-temporary virtual seek
$?	last comparison value
$alias=value	alias commands (simple macros)
$b	block size
$B	base address (aligned lowest map address)
$f	jump fail address (e.g. jz 0x10 => next instruction)
$fl	flag length (size) at current address (fla; pD $l @ entry0)
$F	current function size
$FB	begin of function
$Fb	address of the current basic block
$Fs	size of the current basic block
$FE	end of function
$FS	function size
$Fj	function jump destination

Command	Description
$Ff	function false destination
$FI	function instructions
c,r	get width and height of terminal
$Cn	get nth call of function
$Dn	get nth data reference in function
$D	current debug map base address ?v $D @ rsp
$DD	current debug map size
$e	1 if end of block, else 0
$j	jump address (e.g. jmp 0x10, jz 0x10 => 0x10)
$Ja	get nth jump of function
$Xn	get nth xref of function
$l	opcode length
$m	opcode memory reference (e.g. mov eax,[0x10] => 0x10)
$M	map address (lowest map address)
$o	here (current disk io offset)
$p	getpid()
$P	pid of children (only in debug)
$s	file size
$S	section offset
$SS	section size
$v	opcode immediate value (e.g. lui a0,0x8010 => 0x8010)
$w	get word size, 4 if asm.bits=32, 8 if 64, …
${ev}	get value of eval config variable
$r{reg}	get value of named register
$k{kv}	get value of an sdb query value
$s{flag}	get size of flag
RNum	$variables usable in math expressions

Authors & Contributors

This book wouldn't be possible without the help of a large list of contributors who have been reviewing, writing and reporting bugs and stuff in the radare2 project as well as in this book.

The radare2 book

This book was started by maijin as a new version of the original radare book written by pancake.

- Old radare1 book http://www.radare.org/get/radare.pdf

Many thanks to everyone who has been involved with the gitbook:

Adrian Studer, Ahmed Mohamed Abd El-MAwgood, Akshay Krishnan R, Andrew Hoog, Anton Kochkov, Antonio Sánchez, Austin Hartzheim, Aswin C (officialcjunior), Bob131, DZ_ruyk, David Tomaschik, Eric, Fangrui Song, Francesco Tamagni, FreeArtMan, Gerardo García Peña, Giuseppe, Grigory Rechistov, Hui Peng, ITAYC0HEN, Itay Cohen, Jeffrey Crowell, John, Judge Dredd (key 6E23685A), Jupiter, Kevin Grandemange, Kevin Laeufer, Luca Di Bartolomeo, Lukas Dresel, Maijin, Michael Scherer, Mike, Nikita Abdullin, Paul, Paweł Łukasik, Peter C, RandomLive, Ren Kimura, Reto Schneider, SchumBlubBlub, SkUaTeR, Solomon, Srimanta Barua, Sushant Dinesh, TDKPS, Thanat0s, Vanellope, Vex Woo, Vorlent, XYlearn, Yuri Slobodyanyuk, ali, aoighost, condret, hdznrrd, izhuer, jvoisin, kij, madblobfish, muzlightbeer, pancake, polym (Tim), puddl3glum, radare, sghctoma, shakreiner, sivaramaaa, taiyu, vane11ope, xarkes.