radare

A commandline framework for reverse engineering ala *nix-style

Contents

Chapter 1: Introduction

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

--pancake

1.1 History

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

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

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

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

1.2 Overview

Nowadays the project is composed by a set of small utilities that can be used together or independently from the command line:

`radare`

The core of the hexadecimal editor and debugger. Allows to open any kind of file from different IO access like disk, network, kernel plugins, remote devices, debugged processes, ... and handle any of them as if they were a simple plain file.

Implements an advanced command line interface for moving around the file, analyzing data, disassembling, binary patching, data comparision, searching, replacing, scripting with ruby, python, lua and perl, ...

`rabin`

Extracts information from executable binaries like ELF, PE, Java CLASS, MACH-O. It's used from the core to get exported symbols, imports, file information, xrefs, library dependencies, sections, ...

`rasm`

Commandline assembler and disassembler for multiple architectures (intel[32,64], mips, arm,

powerpc, java, msil, ...)

```
$ rasm -a java 'nop'
00
$ rasm -a x86 -d '90'
nop
```

`rasc`

Small utility to prepare buffers or shellcodes for exploiting vulnerabilities. It has an internal hardcoded database of shellcodes and a syscall-proxy interface with some nice helpers like fill-with nops, breakpoints, series of values to find the landing point, etc..

`hasher`

Implementation of a block-based hasher for small text strings or large disks, supporting multiple algorithms like md4, md5, crc16, crc32, sha1, sha256, sha384, sha512, par, xor, xorpair, mod255, hamdist or entropy.

It can be used to check the integrity or track changes between big files, memory dumps or disks.

`radiff`

Binary diffing utility with multiple algorithms implemented inside. Supports byte-level or delta diffing for binary files and code-analysis diffing to find changes in basic code blocks from radare code analysis or IDA ones using the idc2rdb rsc script.

`rsc`

Entrypoint for calling multiple small scripts and utilities that can be used from the shell.

1.3 Getting radare

You can get radare from the website http://radare.nopcode.org/

There are binary packages for multiple operating systems and GNU/Linux distributions (ubuntu, maemo, gentoo, windows, iphone, etc..) But I hardly encourage you to get the sources and compile them yourself to better understand the dependencies and have the source code and examples more accessible.

I try to publish a new stable release every month and sometimes publish nightly tarballs.

But as always the best way to use a software is to go upstream and pull the development repository which in radare is commonly more stable than the 'stable' releases O:)

To do this you will need mercurial (a distributed python-based source code management aliased Hg) and type:

```
$ hg clone http://radare.nopcode.org/hg/radare
```

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

To update your local copy of the repository you will have to type these two commands in the root of the recently created 'radare' directory.

```
$ hg pull
$ hg update
```

If you have local modifications of the source, you can revert them with:

```
$ hg revert --all
```

Or just feed me with a patch

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

1.4 Compilation and portability

Currently the core of radare can be compiled in many systems, and architectures, but the main development is done on GNU/Linux and GCC. But it is known to compile with TCC and SunStudio.

People usually wants to use radare as a debugger for reverse engineering, and this is a bit more restrictive portability issue, so if the debugger is not ported to your favorite platform, please, notify it to me or just disable the debugger layer with --without-debugger in the ./configure stage.

Nowadays the debugger layer can be used on Windows, GNU/Linux (intel32, intel64, mips, arm), FreeBSD, NetBSD, OpenBSD (intel32, intel64) and there are plans for Solaris and OSX. And there are some IO plugins to use gdb, gdbremote or wine as backends.

The current build system is 'waf':

```
$ ./waf distclean
$ ./waf configure
$ ./waf
$ sudo ./waf install
...
```

The old build system based on ACR/GMAKE stills maintained and usable, but don't relay on it because it is aimed to be removed to only use waf.

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

1.5 Commandline flags

The core accepts multiple flags from the command line to change some configuration or start with different options.

Here's the help message:

```
$ radare -h
radare [options] [file]
  -s [offset]      seek to the desired offset (cfg.seek)
  -b [blocksize]   change the block size (512) (cfg.bsize)
  -i [script]      interpret radare or ruby/python/perl/lua script
  -p [project]     load metadata from project file
  -l [plugin.so]   link against a plugin (.so or .dll)
  -e [key=val]     evaluates a configuration string
  -d [program|pid] debug a program. same as --args in gdb
  -f               set block size to fit file size
  -L               list all available plugins
  -w               open file in read-write mode
  -x               dump block in hexa and exit
  -n               do not load ~/.radarerc and ./radarerc
  -v               same as -e cfg.verbose=false
  -V               show version information
  -u               unknown size (no seek limits)
  -h               this help message
```

1.6 Basic usage

Lot of people ping me some times for a sample usage session of radare to help to understand how the shell works and how to perform the most common tasks like disassembling, seeking, binary patching or debugging.

I hardly encourage you to read the rest of this book to help you understand better how everything works and enhace your skills, the learning curve of radare is usually a bit harder at the beggining, but after an hour of using it you will easily understand how most of the things work and how to get them cooperate together :)

For walking thru the binary file you will use three different kind of basic actions: seek, print and alterate.

To 'seek' there's an specific command abreviated as 's' than accepts an expression as argument that can be something like '10', '+0x25' or '[0x100+ptr_table]'. If you are working with block-based files you may prefer to set up the block size to 4K or the size required with the command 'b' and move forward or backward at seeks aligned to the block size using the '>' and '<' commands.

The 'print' command aliased as 'p', accepts a second letter to specify the print mode selected. The most common ones are 'px' for printing in hexadecimal, 'pd' for disassembling.

To 'write' open the file with 'radare -w'. This should be specified while opening the file, or just type 'eval file.write=true' in runtime to reopen the file in read-write-mode. You can use the 'w' command to write strings or 'wx' for hexpair strings:

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

Appending a '?' to the command you will get the help message of it. (p? for example)

Enter the visual mode pressing 'V<enter>', and return to the prompt using the 'q' key.

In the visual mode you should use hjkl keys which are the default ones for scrolling (like left,down,up,right). So entering in cursor mode ('c') you will be able select bytes if using the shift together with HJKL.

In the visual mode you can insert (alterate bytes) pressing 'i' and then <tab> to switch between the hex or string column. Pressing 'q' in hex panel to return into the visual mode.

1.7 Command format

The format of the commands looks something like that:

```
[#][!][cmd] [arg] [@ offset| @@ flags] [> file] [| shell-pipe] [ && ...]
```

Some examples:

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

1.8 Expressions

The expressions are mathematical representations of a 64 bit numeric value which can be displayed in different formats, compared or used at any command as a numeric argument. They support multiple basic arithmetic operations and some binary and boolean ones. The command used to evaluate these math expressions is the '?'. Here there are some examples:

```
[0xB7F9D810]> ? 0x8048000
0x8048000 ; 134512640d ; 1001100000o ; 0000 0000
[0xB7F9D810]> ? 0x8048000+34
0x8048022 ; 134512674d ; 1001100042o ; 0010 0010
[0xB7F9D810]> ? 0x8048000+0x34
0x8048034 ; 134512692d ; 1001100064o ; 0011 0100

[0xB7F9D810]> ? 1+2+3-4*3
0x6 ; 6d ; 6o ; 0000 0110

[0xB7F9D810]> ? [0x8048000]
0x464C457F ; 1179403647d ; 10623042577o ; 0111 1111
```

The supported arithmetic expressions supported are:

```
+ : addition
- : substraction
* : multiply
/ : division
% : modulus
> : shift right
< : shift left
```

The binary expressions should be scapped:

```
\| : logical OR
\& : logical AND
```

The values can be numbers in many formats:

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

There are other special syntaxes for the expressions. Here's for example some of them:

```
$$            ; current seek
$$$           ; size of opcode at current seek
$${file.size} ; file.size (taken from eval variable)
$$j           ; jump address (branch of instruction)
$$f           ; false address (continuation after branch)
$$r           ; data reference from opcode
```

For example:

```
[0x4A13B8C0]> :pd 2
0x4A13B8C0,  mov eax, esp
0x4A13B8C2   call 0x4a13c000

[0x4A13B8C0]> :? $$+$$$
0x4a13b8c2

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

1.9 Rax

The 'rax' utility comes with the radare framework and aims to be a minimalistic expression evaluator for the shell useful for making base conversions easily 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 given.

```
$ rax -h
Usage: rax [-] | [-s] [-e] [int|0x|Fx|.f|.o] [...]
 int   ->  hex           ;  rax 10
 hex   ->  int           ;  rax 0xa
 -int  ->  hex           ;  rax -77
 -hex  ->  int           ;  rax 0xffffffb3
 float ->  hex           ;  rax 3.33f
 hex   ->  float         ;  rax Fx40551ed8
 oct   ->  hex           ;  rax 035
 hex   ->  oct           ;  rax Ox12 (O is a letter)
 bin   ->  hex           ;  rax 1100011b
 hex   ->  bin           ;  rax Bx63
 -e    swap endianness   ;  rax -e 0x33
 -s    swap hex to bin   ;  rax -s 43 4a 50
 -     read data from stdin until eof
```

Some examples:

```
$ rax 0x345
837
$ rax 837
0x345
$ rax 44.44f
Fx8fc23142
$ rax 0xfffffffd
-3
$ rax -3
0xfffffffd
$ rax -s "41 42 43 44"
ABCD
```

1.10 Basic debugger session

To start debugging a program use the '-d' flag and append the PID or the program path with arguments.

```
$ radare -d /bin/ls
```

The debugger will fork and load the 'ls' program in memory stopping the execution in the 'ld.so', so don't expect to see the entrypoint or the mapped libraries at this point. To change this you can define a new 'break entry point' adding 'e dbg.bep=entry' or 'dbg.bep=main' to your .radarerc.

But take care on this, because some malware or programs can execute code before the main.

Now the debugger prompt should appear and if you press 'enter' (null command) the basic view of the process will be displayed with the stack dump, general purpose registers and disassembly from current program counter (eip on intel).

All the debugger commands are handled by a plugin, so the 'system()' interface is hooked by it and you will have to supply them prefixing it with a '!' character.

Here's a list of the most common commands for the debugger:

```
> !help          ; get the help
> !step 3        ; step 3 times
> !bp 0x8048920  ; setup a breakpoint
> !bp -0x8048920 ; remove a breakpoint
> !cont          ; continue process execution
> !contsc        ; continue until syscall
> !fd            ; manipulate file descriptors
> !maps          ; show process maps
> !mp            ; change page protection permissions
> !reg eax=33    ; change a register
```

The easiest way to use the debugger is from the Visual mode, so, you will no need to remember much commands or keep states in your mind.

```
[0xB7F0C8C0]> Visual
```

After entering this command an hexdump of the current eip will be showed. Now press 'p' one time to get into the debugger view. You can press 'p' and 'P' to rotate thru the most commonly used print modes.

Use F6 or 's' to step into and F7 or 'S' to step over.

With the 'c' key you will toggle the cursor mode and being able to select range of bytes to nop them or set breakpoints using the 'F2' key.

In the visual mode you can enter commands with ':' to dump buffer contents like

```
x @ esi
```

To get the help in the visual mode press '?' and for the help of the debugger press '!'.

At this point the most common commands are !reg that can be used to get or set values for the general purpose registers. You can also manipulate the hardware and extended/floating registers.

Chapter 2: Configuration

The core reads ~/.radarerc while starting, so you can setup there some 'eval' commands to set it up in your favorite way.

To avoid parsing this file, use '-n' and to get a cleaner output for using radare in batch mode maybe is better to just drop the verbosity with '-v'.

All the configuration of radare is done with the 'eval' command which allows the user to change some variables from an internal hashtable containing string pairs.

The most common configuration looks like this:

```
$ cat ~/.radarerc
eval scr.color = true
eval dbg.bep   = entry
eval file.id   = true
eval file.flag = true
eval file.analyze = true
```

These configurations can be also defined using the '-e' flag of radare while loading it, so you can setup different initial configurations from the commandline without having to change to rc file.

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

All the configuration is stored in a hash table grouped by different root names ([i]cfg., file., dbg., ..[/i])

To get a list of the configuration variables just type 'eval' or 'e' in the prompt. All the basic commands can be reduced to a single char. You can also list the configuration variables of a single eval configruation group ending the command argument with a dot '.'.

There are two enhaced interfaces to help users to interactively configure this hashtable. One is called 'emenu' and provides a shell for walking through the tree and change variables.

To get a help about this command you can type 'e?':

```
[0x4A13B8C0]> e?
Usage: e[m] key=value
   > ereset              ; reset configuration
   > emenu               ; opens menu for eval
   > e scr.color = true  ; sets color for terminal
```

Note the 'e' of emenu, which stands for 'eval'. In radare, all basic commands can be reduced to a single char, and you can just type 'e?' to get the help of all the 'subcommands' for the basic command.

```
[0xB7EF38C0]> emenu
Menu: (q to quit)
- asm
- cfg
```

```
- child
- cmd
- dbg
- dir
- file
- graph
- scr
- search
- trace
- zoom
>
```

There is a easier eval interface accessible from the Visual mode, just typing 'e' after entering this mode (type 'Visual' command before).

Most of the eval tree is quite stable, so don't expect hard changes on this area.

I encourage you to experiment a bit on this to fit the interface to your needs.

2.1 Colors

The console access is wrapped by an API that permits to show the output of any command as ANSI, w32 console or HTML (more to come ncurses, pango, ...) this allows the core to be flexible enought to run on limited environments like kernels or embedded devices allowing us to get the feedback from the application in our favourite format.

To start, we'll enable the colors by default in our rc file:

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

There's a tree of eval variables in scr.pal. to define the color palette for every attribute printed in console:

```
[0x465D8810]> e scr.pal.
scr.pal.prompt = yellow
scr.pal.default = white
scr.pal.changed = green
scr.pal.jumps = green
scr.pal.calls = green
scr.pal.push = green
scr.pal.trap = red
scr.pal.cmp = yellow
scr.pal.ret = red
scr.pal.nop = gray
scr.pal.metadata = gray
scr.pal.header = green
scr.pal.printable = bwhite
scr.pal.lines0 = white
scr.pal.lines1 = yellow
scr.pal.lines2 = bwhite
scr.pal.address = green
scr.pal.ff = red
scr.pal.00 = white
scr.pal.7f = magenta
```

If you think these default colors are not correct for any reason. Ping me and i'll change it.

2.2 Common configuration variables

Here's a list of the most common eval configuration variables, you can get the complete list using the 'e' command without arguments or just use 'e cfg.' (ending with dot, to list all the configuration

variables of the cfg. space).

`asm.arch`

Defines the architecture to be used while disassembling (pd, pD commands) and analyzing code ('a' command). Currently it handles 'intel32', 'intel64', 'mips', 'arm16', 'arm' 'java', 'csr', 'sparc', 'ppc', 'msil' and 'm68k'.

It is quite simple to add new architectures for disassembling and analyzing code, so there is an interface adapted for the GNU disassembler and others for udis86 or handmade ones.

Setting asm.arch to 'objdump' the disassembly engine will use asm.objdump to disasemble the current block. For the code analysis the core will use the previous architecture defined in asm.arch.

```
[0x4A13B8C0]> e asm.objdump
objdump -m i386 --target=binary -D
[0x4A13B8C0]> e asm.arch
intel
[0x4A13B8C0]> pd 2
      |    0x4A13B8C0,       eip: 89e0            mov eax, esp
      |    0x4A13B8C2            e839070000      call 0x4a13c000       ; 1 = 0x4a13c000
[0x4A13B8C0]> e asm.arch =objdump
[0x4A13B8C0]> pd
      |    0x4A13B8C0, eip
   0:   89 e0                   mov    eax,esp
   2:   e8 39 07 00 00          call   0x740
   7:   89 c7                   mov    edi,eax
   9:   e8 e2 ff ff ff          call   0xfffffff0
   ...
```

This is useful for disassembling files in architectures not supported by radare. You should understand 'objdump' as 'your-own-disassembler'.

`asm.syntax`

Defines the syntax flavour to be used while disassembling. This is currently only targeting the udis86 disassembler for the x86 (32/64 bits). The supported values are 'intel' or 'att'.

`asm.pseudo`

Boolean value that determines which string disassembly engine to use (the native one defined by the architecture) or the one filtered to show pseudocode strings. This is 'eax=ebx' instead of a 'mov eax, ebx' for example.

`asm.section`

Shows or hides section name (based on flags) at the left of the address.

`asm.os`

Defines the target operating system of the binary to analyze. This is automatically defined by 'rabin -rI' and it's useful for switching between the different syscall tables and perform different depending on the OS.

`asm.flags`

If defined to 'true' shows the flags column inside the disassembly.

`asm.lines`

Draw lines at the left of the offset in the dissassemble print format (pd, pD) to graphically represent

jumps and calls inside the current block.

`asm.linesout`

When defined as 'true', also draws the jump lines in the current block that goes ouside of this block.

`asm.linestyle`

Can get 'true' or 'false' values and makes the line analysis be performed from top to bottom if false or bottom to top if true. 'false' is the optimal and default value for readability.

`asm.offset`

Boolean value that shows or hides the offset address of the disassembled opcode.

`asm.profile`

Set how much information is showed to the user on disassembly. Can get the values 'default', 'simple', 'debug' and 'full'.

This eval will modify other asm. variables to change the visualization properties for the disassembler engine. 'simple' asm.profile will show only offset+opcode, and 'debug' will show information about traced opcodes, stack pointer delta, etc..

`asm.trace`

Show tracing information at the left of each opcode (sequence number and counter). This is useful to read execution traces of programs.

`asm.bytes`

Boolean value that shows or hides the bytes of the disassebled opcode.

`dbg.focus`

Can get a boolean value. If true, radare will ignore events from non selected PIDs.

`cfg.bigendian`

Choose the endian flavour 'true' for big, 'false' for little.

`file.id`

When enabled (set it up to '1' or 'true'). Runs rabin -rI after opening the target file and tries to identify the file type and setup the base address and stuff like that.

`file.analyze`

Runs '`.af* @@ sym_`' and '`.af* @ entrypoint`', after resolving the symbols while loading the binary, to determine the maximum information about the code analysis of the program. This will not be used while opening a project file, so it is preloaded. This option requires file.id and file.flag to be true.

`file.flag`

Finds all the information of the target binary and setup flags to point symbols (imports, exports), sections, maps, strings, etc.

This command is commonly used with file.id.

`scr.color`

This boolean variable allows to enable or disable the colorized output

`scr.seek`

This variable accepts an expression, a pointer (eg. eip), etc. radare will automatically seek to make sure its value is always within the limits of the screen.

`cfg.fortunes`

Enables or disables the 'fortune' message at the begging of the program

Chapter 3: Basic commands

The basic set of commands in radare can be mostly grouped by action, and they should be easy to remember and short. This is why they are grouped with a single character, subcommands or related commands are described with a second character. For example '/ foo' for searching plain strings or '/x 90 90' to look for hexpair strings.

The format of the commands looks something like that:

```
[#][!][cmd] [arg] [@ offset| @@ flags] [> file] [| shell-pipe] [ && ...]
```

This is: repeat the described command '#' times.

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

If the command starts with '!' the string is passed to the plugin hadling the current IO (the debugger for example), if no one handles it calls to posix_system() which is a shell escape, you can prefix the command with two '!!'.

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

The [arg] argument depends on the command, but most of them take a number as argument to specify the number of bytes to work on instead of block size. Other commands accept math expressions, or strings.

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

The '@' is used to specify a temporal seek where the command is executed. This is quite useful to not seeking 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:

```
> s 0
> / lib             ; search 'lib' string
> p8 20 @@ hit0_*   ; show 20 hexpairs at each search hit
```

The '>' is used to pipe the output of the command to a file (truncating to 0 if exist)

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

The '|' pipe is used to dump the output of the command to another program.

```
[0x4A13B8C0]> f | grep section | grep text
0x0805f3b0 512 section__text
0x080d24b0 512 section__text_end
```

Using the '&&' chars you can concatenate multiple commands in a single line:

```
> x @ esp && !reg && !bt  ; shows stack, regs and backtrace
```

3.1 Seeking

The command 's' is used to seek. It accepts a math expression as argument that can be composed by shift operations, basic math ones and memory access.

The 's'eek command supports '+-*!' characters as arguments to perform acts on the seek history.

```
[0x4A13B8C0]> s?
Usage: > s 0x128 ; absolute seek
       > s +33   ; relative seek
       > sn      ; seek to next opcode
       > sb      ; seek to opcode branch
       > sc      ; seek to call index (pd)
       > sx N    ; seek to code xref N
       > sX N    ; seek to data reference N
       > sS N    ; seek to section N (fmi: 'S?')
       > s-      ; undo seek
       > s+      ; redo seek
       > s*      ; show seek history
       > .s*     ; flag them all
       > s!      ; reset seek history
```

The '>' and '<' commands are used to seek into the file using a block-aligned base.

```
> >>>         ; seek 3 aligned blocks forward
> 3>          ; 3 times block-seeking
> s +30       ; seek 30 bytes forward from current seek
> s 0x300     ; seek at 0x300
> s [0x400]   ; seek at 4 byte dword at offset 0x400
> s 10+0x80   ; seek at 0x80+10
```

The 'sn' and 'sb' commands uses the code analysis module to determine information about the opcode in the current seek and seek to the next one (sn) or branch where it points (sb).

```
[0x4A13B8C0]> :pd 1
0x4A13B8C0, mov eax, esp
[0x4A13B8C0]> sn              ; seek next opcode
[0x4A13B8C2]> :pd 1
0x4A13B8C2  call 0x4a13c000
[0x4A13B8C2]> sb              ; seek to branch address
[0x4A13C000]> :pd 1
0x4A13C000, push ebp
[0x4A13C000]>
```

To 'query' the math expression you can evaluate them using the '?' command and giving the math operation as argument. And getting the result in hexa, decimal, octal and binary.

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

3.1.1 Undo seek

All the seeks are stored in a linked list as a history of navigation over the file. You can easily go forward backward of the seek history by using the 's-' and 's+' commands.

In visual mode just press 'u' or 'U' to undo or redo inside the seek history.

Here's a seesion example:

```
[0x00000000]> s 0x100
[0x00000100]> s 0x200
[0x00000200]> s-        ; undo last seek done
[0x00000100]>
```

3.2 Block size

The block size is the default view size for radare. All commands will work with this constraint, but you can always temporally change the block size just giving a numeric argument to the print commands for example (px 20)

```
[0xB7F9D810]> b?
Usage: b[f flag]|[size]     ; Change block size
  > b 200 ; set block size to 200
  > bf sym_main && s sym_main
```

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 the one specified by a flag. For example in symbols, the block size of the flag represents the size of the function.

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

3.3 Sections

It is usually on firmware images, bootloaders and binary files to find sections that are loaded in memory at different addresses than the one in the disk.

To solve this issue, radare implements two utilities: 'file.baddr' and 'S'.

The file.baddr specifies the current base address to be used for disassembling and displaying offsets. In the same way all offsets used in expressions are also affected by this eval variable.

For files with more than one base address. The 'S'ection command will do the job. Here's the help message:

```
[0xB7EE8810]> S?
Usage: S len [base [comment]] @ address
 > S                ; list sections
 > S*               ; list sections (in radare commands
 > S=               ; list sections (in visual)
 > S 4096 0x80000 rwx section_text  @ 0x8048000 ; adds new section
 > S 4096 0x80000   ; 4KB of section at current seek with base 0x.
 > S 10K @ 0x300    ; create 10K section at 0x300
 > S -0x300         ; remove this section definition
 > Sc rwx _text     ; add comment to the current section
 > Sb 0x100000      ; change base address
 > St 0x500         ; set end of section at this address
 > Sf 0x100         ; set from address of the current section
```

This command allows you to manage multiple base addresses depending on the current seek, and enables the possibility to add comments to them. So the debugger information can be imported to the core in a simple way, adding information about the page protections of each section and so.

Here's a sample dummy session.

```
[0xB7EEA810]> S 10K
[0xB7EE8810]> s +5K
[0xB7EE8810]> S 20K
[0xB7EE9C10]> s +3K
[0xB7EE9C10]> S 5K
```

We can specify a section in a single line in this way:

```
S [size] [base-address] [comment] @ [from-address]
```

For example:

```
S section_text_end-section_text 0x8048500 r-x section_text @ 0x4300
```

Displaying the sections information:

```
[0xB7EEA810]> S
00 * 0xb7ee8810 - 0xb7eeb010 bs=0x00000000 sz=0x00002800   ; eip
01 * 0xb7ee9c10 - 0xb7eeec10 bs=0x00000000 sz=0x00005000
02 * 0xb7eea810 - 0xb7eebc10 bs=0x00000000 sz=0x00001400

[0xB7EEA810]> S=
00  0xb7ee8810 |#################-------------------------| 0xb7eeb010
01  0xb7ee9c10 |---------#################################| 0xb7eeec10
02  0xb7eea810 |-------------########--------------------| 0xb7eebc10
=>  0xb7eea810 |#----------------------------------------| 0xb7eea874
```

The first three lines are sections and the last one is the current seek representation based on the proportions over them.

The 's'eek command implements a 'sS' (seek to Section) to seek at the beeginging to the section number N. For example: 'sS 1' in this case will seek to 0xb7ee9c10.

To remove a section definition just prefix the from-address of the section with '-':

```
[0xB7EE8810]> S -0xb7ee9c10
[0xB7EE8810]> S
00 . 0xb7ee9c10 - 0xb7eeec10 bs=0x00000000 sz=0x00005000
01 . 0xb7eea810 - 0xb7eebc10 bs=0x00000000 sz=0x00001400
```

After the section definition we can change the parameters of them with the Sf, St, Sc, Sb commands. After this, radare core will automatically setup the file.baddr depending on this section information

3.4 Mapping files

Radare IO allows to virtually map contents of files in the same IO space at random offsets. This is useful to open multiple files in a single view or just to 'emulate' an static environment like if it was in the debugger with the program and all its libraries mapped there.

Using the 'S'ections command you'll be able to define different base address for each library loaded at different offsets.

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

```
[0x00000000]> o?
Usage: o [file] [offset]
 > o /bin/ls                  ; open file
```

```
 > o /lib/libc.so 0xC848000   ; map file at offset
 > o- /lib/libc.so            ; unmap
```

Let's prepare a simple layout:

```
$ rabin -l ./a.out
libc.so.6
$ radare -u ./a.out
[0x00000000]> o /lib/libc.so.6 0x10000000
[0x00000000]> o /lib/ld-2.7.so 0x465f2000
```

NOTE: radare has been started with the -u flag to ignore file size limits and being able to seek on far places like where we have mapped our libs.

Listing mapped files:

```
[0x00000000]> o
0x00000000 0x000018da ./a.out
0x465f2000 0x4660cf28 /lib/ld-2.7.so
0x10000000 0x101370ec /lib/libc.so.6
```

Let's print some strings from ld.so

```
[0x00000000]> pa @ 0x465F0000+ 2469
_rtld_global\x00_dl_make_stack_executable\x00__libc_stack_end\x00__libc_memalign\x00malloc\x00_dl_
\x00__libc_enable_secure\x00_dl_get_tls_static_info\x00calloc\x00_dl_debug_state\x00_dl_argv\x00_d
_init\x00_rtld_global_ro\x00realloc\x00_dl_tls_setup\x00_dl_rtld_di_...
```

To unmap these files just use the 'o-' command giving the mapped file name as argument.

3.5 Print modes

One of the efforts in radare is the way to show the information to the user. This is interpreting the bytes and giving an almost readable output format.

The bytes can be represented as integers, shorts, longs, floats, timestamps, hexpair strings, or things more complex like C structures, disassembly, decompilations, external processors, ..

This is a list of the available print modes listable with 'p?':

```
[0x08049A80]> p?
Available formats:
 a : ascii                   (null)
 A : ascii printable         (null)
 b : binary                  N bytes
 B : LSB Stego analysis      N bytes
 c : C format                N bytes
 d : disassembly N opcodes   bsize bytes
 D : asm.arch disassembler   bsize bytes
 f : float                   4 bytes
 F : windows filetime        8 bytes
 i : integer                 4 bytes
 l : long                    4 bytes
 L : long long               8 bytes
 m : print memory structure  0xHHHH
 o : octal                   N bytes
 O : Zoom out view           entire file
 p : cmd.prompt              (null)
 % : print scrollbar of seek (null)
 r : raw ascii               (null)
 R : reference               (null)
 s : asm shellcode           (null)
```

```
 t : unix timestamp          4 bytes
 T : dos timestamp           4 bytes
 u : URL encoding            (null)
 U : executes cmd.user       (null)
 v : executes cmd.vprompt    (null)
 1 : p16: 16 bit hex word    2 bytes
 3 : p32: 32 bit hex dword   4 bytes
 6 : p64: 64 bit quad-word   8 bytes
 7 : print 7bit block as raw 8bit (null)
 8 : p8:   8 bit hex pair    N byte
 x : hexadecimal byte pairs  N byte
 z : ascii null terminated   (null)
 Z : wide ascii null end     (null)
```

3.5.1 Hexadecimal

User-friendly way:

```
[0x4A13B8C0]> px
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x4A13B8C0, 89e0 e839 0700 0089 c7e8 e2ff ffff 81c3 ...9............
0x4A13B8D0, eea6 0100 8b83 08ff ffff 5a8d 2484 29c2 ..........Z.$.).
```

Hexpairs:

```
[0x4A13B8C0]> p8
89 e0 e8 39 07 00 00 89 c7 e8 e2 ff ff ff 81 c3 ee a6 01 00 8b 83 08 ff ff ff 5a 8d 24 84 29 c2
```

Basic size types governed by endian:

16 bit words

```
[0x4A13B8C0]> p16 4
0xe089
0x39e8
```

32 bit doublewords

```
[0x4A13B8C0]> p32 4
0x39e8e089
[0x4A13B8C0]> e cfg.bigendian
false
[0x4A13B8C0]> e cfg.bigendian = true
[0x4A13B8C0]> p32 4
0x89e0e839
[0x4A13B8C0]>
```

64 bit dwords

```
[0x08049A80]> p8 16
31 ed 5e 89 e1 83 e4 f0 50 54 52 68 60 9e 05 08

[0x08049A80]> p64 16
0x31ed5e89e183e4f0
0x50545268609e0508
```

3.5.2 Date formats

The current supported timestamp print modes are:

```
 F : windows filetime        8 bytes
 t : unix timestamp          4 bytes
 T : dos timestamp           4 bytes
```

For example, you can 'view' the current buffer as timestamps in dos, unix or windows filetime formats:

```
[0x08048000]> eval cfg.bigendian = 0
[0x08048000]> pt 4
30:08:2037 12:25:42 +0000

[0x08048000]> eval cfg.bigendian = 1
[0x08048000]> pt 4
17:05:2007 12:07:27 +0000
```

As you can see, the endianness affects to the print formats. Once printing these filetimes you can grep the results by the year for example:

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

The date format printed can be configured with the 'cfg.datefmt' variable following the `strftime(3)` format.

Extracted from the manpage:

```
 %a  The abbreviated weekday name according to the current locale.
 %A  The full weekday name according to the current locale.
 %b  The abbreviated month name according to the current locale.
 %B  The full month name according to the current locale.
 %c  The preferred date and time representation for the current locale.
 %C  The century number (year/100) as a 2-digit integer. (SU)
 %d  The day of the month as a decimal number (range 01 to 31).
 %e  Like %d, the day of the month as a decimal number, leading spaces
 %E  Modifier: use alternative format, see below. (SU)
 %F  Equivalent to %Y-%m-%d (the ISO 8601 date format). (C99)
 %g  Like %G, but without century, that is, with a 2-digit year (00-99). (TZ)
 %h  Equivalent to %b.  (SU)
 %H  The hour as a decimal number using a 24-hour clock (range 00 to 23).
 %I  The hour as a decimal number using a 12-hour clock (range 01 to 12).
 %j  The day of the year as a decimal number (range 001 to 366).
 %k  The hour (24-hour clock) as a decimal number (range 0 to 23);
 %l  The hour (12-hour clock) as a decimal number (range 1 to 12);
 %m  The month as a decimal number (range 01 to 12).
 %M  The minute as a decimal number (range 00 to 59).
 %n  A newline character. (SU)
 %O  Modifier: use alternative format, see below. (SU)
 %p  Either AM or PM
 %P  Like %p but in lowercase: am or pm
 %r  The time in a.m. or p.m. notation.  In the POSIX this is to %I:%M:%S %p.  (SU)
 %R  The time in 24-hour notation (%H:%M). (SU) For seconds, see %T below.
 %s  The number of seconds since the Epoch (1970-01-01 00:00:00 UTC). (TZ)
 %S  The second as a decimal number (range 00 to 60).
 %t  A tab character. (SU)
 %T  The time in 24-hour notation (%H:%M:%S). (SU)
 %u  The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w.  (SU)
 %w  The day of the week as a decimal, range 0 to 6, Sunday being 0.  See also %u.
 %W  The week number of the current year as a decimal number, range 00 to 53.
 %x  The preferred date representation for the current locale without the time.
 %X  The preferred time representation for the current locale without the date.
 %y  The year as a decimal number without a century (range 00 to 99).
 %Y  The year as a decimal number including the century.
 %z  The time-zone as hour offset from GMT.  (using "%a, %d %b %Y %H:%M:%S %z"). (GNU)
 %Z  The time zone or name or abbreviation.
 %+  The date and time in date(1) format. (TZ) (Not supported in glibc2.)
 %%  A literal % character.
```

3.5.3 Basic types

All basic C types are mapped as print modes for float, integer, long and longlong. If you are interested in a more complex structure or just an array definition see 'print memory' section for more information.

Here's the list of the print (p?) modes for basic C types:

```
 f : float                  4 bytes
 i : integer                4 bytes
 l : long                   4 bytes
 L : long long              8 bytes
```

Let's see some examples:

```
[0x4A13B8C0]> pi 32
57
137
255
195
0
255
141
194

[0x4A13B8C0]> pf
-0.000000
0.000000
-119237.992188
nan
-256878602780814480018744180736.000000
-0.000000
nan
```

3.5.4 Source (asm, C)

```
 c : C format               N bytes
 s : asm shellcode          (null)

[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, 0xff,
0x5a, 0x8d, 0x24, 0x84, 0x29, 0xc2 };

[0xB7F8E810]> ps 32
eip:
.byte 0x89, 0xe0, 0xe8, 0x49, 0x02, 0x00, 0x00, 0x89, 0xc7, 0xe8, 0xe2
.byte 0xff, 0xff, 0xff, 0x81, 0xc3, 0xd6, 0xa7, 0x01, 0x00, 0x8b, 0x83
.byte 0x00, 0xff, 0xff, 0xff, 0x5a, 0x8d, 0x24, 0x84, 0x29, 0xc2
.equ eip_len, 32
```

3.5.5 Strings

Strings are probably one of the most important entrypoints while starting to reverse engineer a program because they are usually referencing information about the functions actions (asserts, debug or info messages, ...).

So it is important for radare to be able to print strings in multiple ways:

```
..p?..
 a : ascii                  (null)
```

```
 A : ascii printable         (null)
 z : ascii null terminated   (null)
 Z : wide ascii null end     (null)
 r : raw ascii               (null)
```

Commands 'pa' and 'pA' are pretty similar, but 'pA' protects your console from strange non-printable characters. These two commands are restricted to the block size, so you will have to manually adjust the block size to get a nicer format. If the analyzed strings are zero-terminated or wide-chars, use 'z' or 'Z'.

Most common strings will be just zero-terminated ones. Here's an example by using the debugger to continue the execution of the 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`. Which is obviously a zero terminated string.

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

Finally, the 'pr' is used to raw print the bytes to stdout. These bytes can be redirected to a file by using the '>' character:

```
[0x4A13B8C0]> pr 20K > file
[0x4A13B8C0]> !!du -h file
20K     file
```

3.5.6 Print memory

It is possible to print various packed data types in a single line using the 'pm' command (print memory). Here's the help and some examples:

```
[0x4A13B8C0]> pm
Usage: pm [times][format] [arg0 arg1]
Example: pm 10xiz pointer length string
 e - temporally swap endian
 f - float value
 b - one byte
 B - show 10 first bytes of buffer
 i - %d integer value (4 byets)
 w - word (16 bit hexa)
 q - quadword (8 bytes)
 p - pointer reference
 x - 0x%08x hexadecimal value
 z - \0 terminated string
 Z - \0 terminated wide string
 s - pointer to string
 t - unix timestamp string
 * - next char is pointer
 . - skip 1 byte
```

The simple use would be like this:

```
[0xB7F08810]> pm xxs @ esp
0xbf8614d4 = 0xb7f22ff4
```

```
0xbf8614d8 = 0xb7f16818
0xbf8614dc = 0xbf8614dc -> 0x00000000 /etc/ld.so.cache
```

This is sometimes useful for looking at the arguments passed to a function, by just giving the 'format memory string' as argument and temporally changing the current seek with the '@' token.

It is also possible to define arrays of structures with 'pm'. Just prefix the format string with a numeric value.

You can also define a name for each field of the structure by giving them as optional arguments after the format string splitted by spaces.

```
[0x4A13B8C0]> pm 2xw pointer type @ esp
0xbf87d160 [0] {
   pointer : 0xbf87d160 = 0x00000001
      type : 0xbf87d164 = 0xd9f3
}
0xbf87d164 [1] {
   pointer : 0xbf87d164 = 0xbf87d9f3
      type : 0xbf87d168 = 0x0000
}
```

If you want to store this information as metadata for the binary file just use the same arguments, but instead of using `pm`, use `Cm`. To store all the metadata stored while analyzing use the 'Ps <filename>' command to save the project and then run `radare -p project-file` to restore the session. Read 'projects' section for more information.

3.5.7 Disassembly

The 'pd' command is the one used to disassemble code, it accepts a numeric value to specify how many opcodes are wanted to be disassembled. The 'pD' one acts in the same way, but using a number-of-bytes instead of counting instructions.

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

If you prefer a smarter disassembly with offset and opcode prefix the 'pd' command with ':'. This is used to temporally drop the verbosity while executing a radare command.

```
[0x4A13B8C0]> pd 1
      |    0x4A13B8C0,       eip: 89e0            mov eax, esp

[0x4A13B8C0]> :pd 1
0x4A13B8C0, mov eax, esp
```

The ',' near the offset determines if the address is aligned to 'cfg.addrmod' (this is 4 by default).

3.5.8 Selecting the architecture

The architecture flavour for the disassembly is defined by the 'asm.arch' eval variable. Here's a list of all the supported architectures:

```
[0xB7F08810]> eval asm.arch = arm

Supported values:
intel
intel16
intel32
intel64
```

```
x86
mips
arm
arm16
java
sparc
ppc
m68k
csr
msil
```

3.5.9 Configuring the disassembler

There are multiple options that can be used to configure the output of the disassembly

```
asm.comments = true    ; show/hide comments
asm.cmtmargin = 27     ; comment margins
asm.cmtlines = 0       ; max number of comment lines (0=unlimit)
asm.offset = true      ; show offsets
asm.reladdr = false    ; show relative addresses
asm.nbytes = 8         ; max number of bytes per opcode
asm.bytes = true       ; show bytes
asm.flags = true       ; show flags
asm.flagsline = false  ; show flags in a new line
asm.functions = true   ; show function closures
asm.lines = true       ; show jump/call lines
asm.nlines = 6         ; max number of jump lines
asm.lineswide = true   ; use wide jump lines
asm.linesout = false   ; show jmp lines that go outside the block
asm.linestyle = false  ; use secondary jump line style
asm.trace = false      ; show opcode trace information
asm.os = linux         ; used for syscall resolution and so
asm.split = true       ; split end blocks by lines
asm.splitall = false   ; split all blocks by lines
asm.size = false       ; show size of opcode
```

3.5.10 Disassembly syntax

The syntax is the flavour of disassembly syntax prefered to be used by the disasm engine.

Actually the x86 disassembler is the more complete one. It's based on udis86 and supports the following syntax flavours:

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

The 'olly' syntax uses the ollydbg disassembler engine. 'intel' and 'att' are the most common ones and 'pseudo' is an experimental pseudocode disassembly, sometimes useful for reading algorithms.

3.6 Zoom

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

The cursor can be used to scroll faster thru the zoom out view and pressing 'z' again to zoom-in where the cursor points.

```
zoom.byte values:
 F : number of 0xFF
 f : number of flags
 c : code (functions)
 s : strings
 t : traces (opcode traces)
 p : number of printable chars
 e : entropy calculation
 * : first byte of block
```

For example. let's see some examples:

```
[0x08049790]> pO
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00001790, 7fc7 0107 0141 b9e9 559b 3b85 f87d 7f89 ff05 .....A..U.;..}....
0x00007730, 04c0 8505 c78b 7555 7dc3 0584 f8b0 8985 8900 ......uU}.........
0x0000D6D0, 8b55 1485 fbff ffff ff50 83d0 6620 2020 6561 .U.......P..f   ea
0x00013670, 6918 7f57 cc74 002e 2400                     i..W.t..$.

[0x08049790]> eval zoom.byte = printable
[0x08049790]> pO
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00001790, 7fc7 0107 0141 b9e9 559b 3b85 f87d 7f89 ff05 .....A..U.;..}....
0x00007730, 04c0 8505 c78b 7555 7dc3 0584 f8b0 8985 8900 ......uU}.........
0x0000D6D0, 8b55 1485 fbff ffff ff50 83d0 6620 2020 6561 .U.......P..f   ea
0x00013670, 6918 7f57 cc74 002e 2400                     i..W.t..$.

[0x08049790]> pO
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00001790, 0202 0304 0505 0505 0505 0505 0505 0605 0505 .................
0x00007730, 0505 0505 0505 0505 0505 0606 0505 0505 0605 .................
0x0000D6D0, 0505 0405 0505 0505 0505 0505 0303 0303 0405 .................
0x00013670, 0403 0405 0404 0304 0303                     ..........

[0x08049790]> eval zoom.byte = flags
[0x08049790]> pO
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00001790, 0b04 1706 0400 0000 0000 0000 0000 0000 0000 .................
0x00007730, 0000 0000 0000 0000 0000 0000 0000 0000 0000 .................
0x0000D6D0, 0000 0000 0000 0000 0000 000d 1416 1413 165b .................[
0x00013670, 1701 0e23 0b67 2705 0f12                     ...#.g'...

[0x08049790]> eval zoom.byte = FF
[0x08049790]> pO
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00001790, 0000 0000 0000 0001 0000 0001 0000 0000 0200 .................
0x00007730, 0000 0100 0000 0000 0000 0000 0000 0101 0000 .................
0x0000D6D0, 0000 0001 0201 0202 0100 0000 0000 0000 0000 .................
0x00013670, 0000 0000 0002 0000 0000                     ..........
```

In the debugger, the zoom.from and zoom.to eval variables are defined by .!maps* to fit the user code sections of memory of the target process.

BTW you can determine the limits for performing a zoom on a range of bytes of the whole bytespace by using the zoom.from and zoom.to eval variables.

```
[0x465D8810]> e zoom.
zoom.from = 0x08048000
zoom.to = 0x0805f000
zoom.byte = head
```

NOTE: These values (0x8048000-...) are defined by the debugger to limit the zoom view while debugging to only visualize the user maps of the program.

3.7 Flags

The flags are bookmarks at a certain offset in the file that can be stored inside 'flag spaces'. A flag space is something like a namespace for flags. They are used to group flags with similar characteristics or of a certain type. Some example of flagspaces could be [i]sections, registers, symbols, search hits[/i], etc.

To create a flag just type:

```
> f flag_name @ offset
```

You can remove this flag adding '-' at the begginging of the command. Most commands accept '-' as argument-prefix as a way to delete.

```
> f -flag_name
```

To switch/create between flagspaces use the 'fs' command:

```
[0x4A13B8C0]> fs   ; list flag spaces
00   symbols
01   imports
02   sections
03   strings
04   regs
05   maps

> fs symbols
> f         ; list only flags in symbols flagspace
...
> fs *      ; select all flagspaces
```

You can create two flags with the same name with 'fn' or rename them with 'fr'.

Sometimes you'll like to add some flags adding a delta base address to each of them. To do this use the command 'ff' (flag from) which is used to specify this base address. Here's an example:

```
[0x00000000]> f patata
[0x00000000]> ? patata
0x0 ; 0d ; 0o ; 0000 0000
[0x00000000]> ff 0x100
[0x00000000]> f patata
[0x00000000]> ? patata
0x100 ; 256d ; 400o ; 0000 0000
[0x00000000]> ff
0x00000100
[0x00000000]> ff 0         ; reset flag from
```

3.7.1 Flag intersections

The '/' command for searching registers some flags for the hit results. You can use them to draw intersection vectors between these hits and be able to determine block sizes from a header and a footer search keywords.

Here's an example:

```
[0x00000000]> !cat txt
_head
jklsdfjlksaf
_foot
_body
```

```
jeje peeee
_foot
_body
food is lavle
_foot
```

Let's define the header and the footer keywords:

```
[0x00000000]> /k0 _body
[0x00000000]> /k1 _foot
[0x00000000]> /k
00 _body
01 _foot
```

Do the ranged search using keywords 0 and 1:

```
[0x00000000]> /r 0,1
001  0x00000000  hit0_0 _bodyjklsdfjlksaf
002  0x00000015  hit1_1 _foot_bodyjeje p
003  0x0000001c  hit0_2 _bodyjeje peeee-
004  0x0000002f  hit1_3 _foot_bodyfood is
005  0x00000036  hit0_4 _bodyfood is lavle
006  0x0000004b  hit1_5 _foot
```

Perform intersection between hits!

```
[0x00000000]> fi hit0 hit1
hit0_0 (0x00000000) -> hit1_1 (0x00000015)   ; size = 21
hit0_2 (0x0000001c) -> hit1_3 (0x0000002f)   ; size = 19
hit0_4 (0x00000036) -> hit1_5 (0x0000004b)   ; size = 21
```

3.8 Write

Radare can manipulate the file in multiple ways. You can resize the file, move bytes, copy/paste them, insert mode (shifting data to the end of the block or file) or just overwrite some bytes with an address, the contents of a file, a widestring or inline assembling an opcode.

To resize. Use the 'r' command which accepts a numeric argument. Possitive valule sets the new size to the file. A negative one will strip N bytes from the current seek down-sizing the file.

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

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

```
[0x4A13B8C0]> w?
Usage: w[?|*] [argument]
  w  [string]        ; write plain with escaped chars string
  wa [opcode]        ; write assembly using asm.arch and rasm
  wA '[opcode]'      ; write assembly using asm.arch and rsc asm
  wb [hexpair]       ; circulary fill the block with these bytes
  wv [expr]          ; writes 4-8 byte value of expr (use cfg.bigendian)
  ww [string]        ; write wide chars (interlace 00s in string)
  wf [file]          ; write contents of file at current seek
  wF [hexfile]       ; write hexpair contents of file
  wo[xrlaAsmd] [hex] ; operates with hexpairs xor,shiftright,left,add,sub,mul,div
```

Some examples:

```
> wx 12 34 56 @ 0x8048300
> wv 0x8048123 @ 0x8049100
```

```
> wa jmp 0x8048320
```

All write changes are recorded and can be listed or undo-ed using the 'u' command which is explained in the 'undo/redo' section.

3.8.1 Write over with operation

The 'wo' write command accepts multiple kinds of operations that can be applied on the curren block. This is for example a XOR, ADD, SUB, ...

```
[0x4A13B8C0]> wo?
Usage: wo[xrlasmd] [hexpairs]
Example: wox 90    ; xor cur block with 90
Example: woa 02 03 ; add 2, 3 to all bytes of cur block
Supported operations:
  woa  addition        +=
  wos  substraction    -=
  wom  multiply        *=
  wod  divide          /=
  wox  xor             ^=
  woo  or              |=
  woA  and             &=
  wor  shift right    >>=
  wol  shift left     <<=
```

This way it is possible to implement ciphering algorithms using radare core primitives.

A sample session doing a `xor(90) + addition(01 02)`

```
[0x4A13B8C0]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x4A13B8C0, 89e0 e839 0700 0089 c7e8 e2ff ffff ...9.........
0x4A13B8CE  81c3 eea6 0100 8b83 08ff ffff 5a8d ............Z.
0x4A13B8DC, 2484 29c2 528b 8344 0000 008d 7494 $.).R..D....t.
0x4A13B8EA  088d 4c24 0489 e583 e4f0 5050 5556 ..L$......PPUV
0x4A13B8F8, 31ed e8f1 d400 008d 93a4 31ff ff8b 1.........1...
0x4A13B906  2424 ffe7 8db6 0000 0000 e8b2 4f01 $$..........O.
0x4A13B914, 0081 c1a7 a601 0055 89e5 5d8d 814c .......U..]..L
0x4A13B922  0600                              ..

[0x4A13B8C0]> wox 90
[0x4A13B8C0]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x4A13B8C0, 1970 78a9 9790 9019 5778 726f 6f6f .px.....Wxrooo
0x4A13B8CE  1153 7e36 9190 1b13 986f 6f6f ca1d .S~6.....ooo..
0x4A13B8DC, b414 b952 c21b 13d4 9090 901d e404 ...R..........
0x4A13B8EA  981d dcb4 9419 7513 7460 c0c0 c5c6 ......u.t`....
0x4A13B8F8, a17d 7861 4490 901d 0334 a16f 6f1b .}xaD....4.oo.
0x4A13B906  b4b4 6f77 1d26 9090 9090 7822 df91 ..ow.&....x"..
0x4A13B914, 9011 5137 3691 90c5 1975 cd1d 11dc ..Q76....u....
0x4A13B922  9690                              ..

[0x4A13B8C0]> woa 01 02
[0x4A13B8C0]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x4A13B8C0, 1a72 79ab 9892 911b 587a 7371 7071 .ry.....Xzsqpq
0x4A13B8CE  1255 7f38 9292 1c15 9971 7071 cb1f .U.8.....qpq..
0x4A13B8DC, b516 ba54 c31d 14d6 9192 911f e506 ...T..........
0x4A13B8EA  991f ddb6 951b 7615 7562 c1c2 c6c8 ......v.ub....
0x4A13B8F8, a27f 7963 4592 911f 0436 a271 701d ..ycE....6.qp.
0x4A13B906  b5b6 7079 1e28 9192 9192 7924 e093 ..py.(....y$..
0x4A13B914, 9113 5239 3793 91c7 1a77 ce1f 12de ..R97....w....
0x4A13B922  9792                              ..
```

3.9 Undo/redo

The 'u'ndo command is used to undo or redo write changes done on the file.

```
> u?
Usage: > u 3   ; undo write change at index 3
       > u -3  ; redo write change at index 3
       > u     ; list all write changes
```

Here's a sample session working with undo writes:

```
[0x00000000]> wx 90 90 90 @ 0x100
[0x00000100]> u                         ; list changes
00 + 3 00000100: 89 90 c4 => 90 90 90

[0x00000000]> p8 3 @ 0x100
90 90 90
[0x00000000]> u 0
[0x00000000]> p8 3 @ 0x100
89 90 c4
[0x00000000]> u -0
[0x00000000]> p8 3 @ 0x100
90 90 90
```

Note: Read 'undo-seek' for seeking history manipulation.

3.10 Yank/Paste

You can yank/paste bytes in visual mode using the 'y' and 'Y' key bindings that are alias for the 'y' and 'yy' commands of the shell. There is an internal buffer that stores N bytes from the current seek. You can write-back to another seek using the 'yy' one.

```
[0x4A13B8C0]> y?
Usage: y[ft] [length]
 > y 10 @ eip     ; yanks 10 bytes from eip
 > yy @ edi       ; write these bytes where edi points
 > yt [len] dst   ; copy N bytes from here to dst
```

Sample session:

```
> s 0x100    ; seek at 0x100
> y 100      ; yanks 100 bytes from here
> s 0x200    ; seek 0x200
> 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 the following:

```
[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.$.).
[0x4A13B8C0]>
```

3.11 Comparing bytes

You can compare data using the 'c' command that accepts different input formats and compares the input against the bytes in the current seek.

```
> c?
Usage: c[?|d|x|f] [argument]
  c  [string]   - compares a plain with escaped chars string
  cd [offset]   - compare a doubleword from a math expression
  cx [hexpair]  - compare hexpair string
  cf [file]     - compare contents of file at current seek
```

An example of memory comparision:

```
[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]>
```

This is also useful for comparing memory pointers at certain offsets. The variable cfg.bigendian is used to change the value in the proper way to be compared against the contents at the '0x4A13B8C0' offset:

```
[0x4A13B8C0]> cd 0x39e8e089 @ 0x4A13B8C0
Compare 4/4 equal bytes

[0x4A13B8C0]> p8 4
89 e0 e8 39
```

It takes 4 bytes from the current seek (0x4A13B8C0) and compares them to the number given. This number can be an math expressions using flag names and so:

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

We can use the compare command against a file previously dumped to disk from the contents of the current block.

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

Chapter 4: Visual mode

The visual mode is a user-friendlier interface for the commandline prompt of radare which accepts HJKL movement keys, a cursor for selecting bytes and some keybindings to ease the use of the debugger.

In this mode you can change the configuration in a easy way using the 'e' (eval) key. Or just track the flags and walk thru the flagspaces pressing 't'.

To get a help of all the keybindings hooked in visual mode you can press '?':

```
Visual keybindings:
:<cmd>     radare command (vi like)
;          edit or add comment
,.         ',' marks an offset, '.' seeks to mark or eip if no mark
g,G        seek to beggining or end of file
+-*/       +1, -1, +width, -width -> block size
<>         seek block aligned (cursor mode = folder code)
[]         adjust screen width
a,A,=      insert patch assembly, rsc asm or !hack
i          insert mode (tab to switch btw hex,asm,ascii, 'q' to normal)
f,F        seek between flag list (f = forward, F = backward)
t          visual track/browse flagspaces and flags
e          visual eval configuration variables
c          toggle cursor mode
C          toggle scr.color
d          convert cursor selected bytes to ascii, code or hex
m          applies rfile magic on this block
I          invert block (same as pIx or so)
y,Y        yank and Yankee aliases for copy and paste
f,F        go next, previous flag (cursor mode to add/remove)
h,j,k,l    scroll view to left, down, up, right.
J,K        up down scroll one block.
H,L        scroll left, right by 2 bytes (16 bits).
p,P        switch between hex, bin and string formats
x          show xrefs of the current offset
q          exits visual mode

Debugger keybindings:
!          show debugger commands help
F1         commands help
F2         set breakpoint (execute)
F3         set watchpoint (read)
F4         continue until here (!contuh)
F6         continue until syscall (!contsc)
F7         step in debugger user code (!step)
F8         step over in debugger (!stepo)
F9         continue execution (!cont)
F10        continue until user code (!contu)
```

From the visual mode you can toggle the insert and cursor modes with the 'i' and 'c' keys.

4.1 Visual cursor

Pressing lowercase 'c' makes the cursor appear or disappear. The cursor is used to select a range of bytes or just point to a byte to flag it (press 'f' to create a new flag where the cursor points to)

If you select a range of bytes press 'w' and then a byte array to overwrite the selected bytes with the ones you choose in a circular copy way. For example:

```
<select 10 bytes in visual mode>
<press 'w' and then '12 34'>
The 10 bytes selected will become: 12 34 12 34 12 34 12 34 12 34
```

The byte range selection can be used together with the 'd' key to change the data type of the selected bytes into a string, code or a byte array.

That's useful to enhace the disassembly, add metadata or just align the code if there are bytes mixed with code.

In cursor mode you can set the block size by simply moving it to the position you want and pressing '_'. Then block_size = cursor.

4.2 Visual insert

The insert mode allows you to write bytes at nibble-level like most common hexadecimal editors. In this mode you can press '<tab>' to switch between the hexa and ascii columns of the hexadecimal dump.

To get back to the normal mode, just press '<tab>' to switch to the hexadecimal view and press 'q'. (NOTE: if you press 'q' in the ascii view...it will insert a 'q' instead of quit this mode)

There are other keys for inserting and writing data in visual mode. Basically by pressing 'w' key you'll be prompted for an hexpair string or use 'a' for writing assembly where the cursor points.

4.3 Visual xrefs

radare implements many user-friendly features for the visual interface to walk thru the assembly code. One of them is the 'x' key that popups a menu for selecting the xref (data or code) against the current seek and then jump there. In this example, we are displaying the import getenv and displaying the CODE xreferences to this external symbol.

```
[0x08048700]> pd @ imp_getenv
; CODE xref 0x08048e30 (sym_otf_patch+0x1d9)
; CODE xref 0x08048d53 (sym_otf_patch+0xfc)
; CODE xref 0x08048c90 (sym_otf_patch+0x39)
      |    0x08048700,   imp_getenv:
      |    0x08048700           jmp dword near [0x804c00c]
      |    0x08048706           push dword 0x18       ; oeax+0xd
      `==< 0x0804870B           jmp 0x80486c0         ; 1 = section__plt
```

Use the 'sx' and 'sX' command to seek tot he xrefs for code and Xrefs for data indexed by numbers.

All the calls and jumps are numbered (1, 2, 3...) these numbers are the keybindings for seeking there from the visual mode.

```
[0x4A13B8C0]> pd 4
0x4A13B8C0,   eip:   mov eax, esp
0x4A13B8C2           call 0x4a13c000       ; 1 = 0x4a13c000
```

```
0x4A13B8C7           mov edi, eax
0x4A13B8C9           call 0x4a13b8b0       ; 2 = 0x4a13b8b0
```

All the seek history is stored, by pressing 'u' key you will go back in the seek history time :)

Chapter 5: Searching bytes

The search engine of radare is based on the work done by esteve plus multiple features on top of it that allows multiple keyword searching with binary masks and automatic flagging of results.

This powerful command is '/'.

```
[0x00000000]> /?
 / \x7FELF      ; plain string search (supports \x).
 /. [file]      ; search using the token file rules
 /s [string]    ; strip strings matching optional string
 /x A0 B0 43    ; hex byte pair binary search.
 /k# keyword    ; keyword # to search
 /m# FF 0F      ; Binary mask for search '#' (optional)
 /a [opcode]    ; Look for a string in disasembly
 /A             ; Find expanded AES keys from current seek(*)
 /w foobar      ; Search a widechar string (f\0o\0o\0b\0..)
 /r 0,2-10      ; launch range searches 0-10
 /p len         ; search pattern of length = len
 //             ; repeat last search
```

The search is performed from the current seek until the end of the file or 'cfg.limit' if != 0. So in this way you can perform limited searches between two offsets of a file or the process memory.

With radare everything is handled as a file, it doesn't matters if it is a socket, a remote device, the process memory, etc..

5.1 Basic searchs

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

```
$ echo "/ lib" | radare -nv /bin/ls
001  0x00000135  hit0_0 lib/ld-linux.so.2
002  0x00000b71  hit0_1 librt.so.1__gmon_st
003  0x00000bad  hit0_2 libselinux.so.1_ini
004  0x00000bdd  hit0_3 libacl.so.1acl_exte
005  0x00000bfb  hit0_4 libc.so.6_IO_stdin_
006  0x00000f2a  hit0_5 libc_start_maindirf
$
```

As you can see, radare generates a 'hit' flag for each search result found. You you can just use the 'pz' command to visualize the strings at these offsets in this way:

```
[0x00000000]> / ls
...
[0x00000000]> pz @ hit0_0
lib/ld-linux.so.2
```

We can also search wide-char strings (the ones containing zeros between each letter) using the '/w' in this way:

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

It is also possible to mix hexadecimal scape sequences in the search string:

```
$ radare -u /dev/mem
[0x00000000]> / \x7FELF
```

But if you want to perform an hexadecimal search you will probably prefer an hexpair input with '/x':

```
[0x00000000]> /x 7F 45 4C 46
```

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

```
[0x00000000]> fs search
[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 these flags, you can just use the 'f -hit*' command.

Sometimes while working long time in the same file you will need to launch the last search more than once and you will probably prefer to use the '//' command instead of typing all the string again.

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

5.2 Configurating the searchs

The search engine can be configured by the 'eval' interface:

```
[0x08048000]> eval search.
search.from = 0
search.to = 0
search.align = 0
search.flag = true
search.verbose = true
```

The search.[from|to] is used to define the offset range limits for the searches.

'search.align' variable is used to determine that the only 'valid' search hits must have to fit in this alignement. For example. you can use 'e search.align=4' to get only the hits found in 4-byte aligned addresses.

The 'search.flag' boolean variable makes the engine setup flags when finding hits. If the search is stopped by the user with a ^C then a 'search_stop' flag will be added.

5.3 Pattern search

The search command allows you to throw repeated pattern searchs against the IO backend to be able to identify repeated sequences of bytes without specifying them. The only property to perform this search is to manually define the minimum length of these patterns.

Here's an example:

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

The output of the command will show the different patterns found and how many times they are repeated.

5.4 Automatization

The `cmd.hit` eval variable is used to define a command that will be executed when a hit is reached by the search engine. If you want to run more than one command use '&&' or '. script-file-name' for including a file as a script.

For example:

```
[0x08048000]> eval cmd.hit = p8 8
[0x08048000]> / lib
6c 69 62 2f 6c 64 2d 6c
1001  0x00000155  hit0_0 lib/ld-linux
6c 69 62 72 74 2e 73 6f
2002  0x00013a25  hit0_1 librt.so.1c
6c 69 62 63 2e 73 6f 2e
3003  0x00013a61  hit0_2 libc.so.6st
6c 69 62 63 5f 73 74 61
4004  0x00013d6c  hit0_3 libc_start_m
6c 69 62 70 74 68 72 65
5005  0x00013e13  hit0_4 libpthread.s
6c 69 62 2f 6c 64 2d 6c
6006  0x00013e24  hit0_5 lib/ld-linux
6c 69 62 6c 69 73 74 00
7read err at 0x0001542c
007  0x00014f22  hit0_6 liblist.gnu
```

A simple and practical example for using cmd.hit can be for replacing some bytes for another ones, by setting 'wx ..' in cmd.hit. This example shows how to drop the selinux dependency on binaries compiled on selinux-enabled distributions to make the dynamic elf run on other systems without selinux:

```
$ for file in bin/* ; do \
    echo "/ libselinux" | radare -nvwe "cmd.hit=wx 00" $file \
  done
```

This shell command will run radare looking for the string 'libselinux' on the target binary. It ignores the user preferences with '-n', drops verbosity with '-v' and enables write mode with '-w'. Then it setups the 'cmd.hit' variable to run a 'wx 00' command so. it will truncate the 'libselinux' string to be 0length. This way the loader will ignore the loading because of the null-name.

5.5 Backward search

TODO (not yet implemented)

5.6 Multiple keywords

To define multiple keywords you should use the '/k' command which accepts a string with hexa scaped characters. Here's an example of use:

```
[0x08048000]> /k0 lib
[0x08048000]> /k1 rt
[0x08048000]> /k      ; list introduced keywords
00 lib
01 rt
```

To search these two keywords just use the '/r' (ranged search) command:

```
[0x08048000]> /r 0-1
001  0x00000135  hit0_0 lib/ld-linux.so.2
002  0x00000b71  hit0_1 librt.so.1__gmon_st
003  0x00000b74  hit1_2 rt.so.1__gmon_start
...
```

5.7 Binary masks

In the same way you setup keywords to search it is possible to define binary masks for each of them with the '/m' command. Here's an example of use:

```
[0x08048000]> /k0 lib
[0x08048000]> /m0 ff 00 00
[0x08048000]> /m
0 ff 00 00
[0x08048000]> /k
00 lib
```

Now just use '/r 0' to launch the k0 keyword with the associated m0 binary mask and get the 3-byte hit starting by an 'l' because 'il' is ignored by the binary mask.

This case is quite stupid, but if you work with JPEGs or on ARM for example, you can type more fine-grained binary masks to collect some bits from certain headers or just get the opcodes matching a certain conditional.

5.8 Search using rules file

You can specify a list of keywords in a single file with its binary mask and use the search engine to find them.

The file format should be something like this:

```
$ cat token
token:  Library token
        string: lib
        mask:   ff 00 ff

token:  Realtime
        string: rt
        mask:   ff ff
```

Note that tab is used to indent the 'string' and 'mask' tokens. The first line specifies the keyword name which have nothing to do with the search.

```
[0x08049A80]> /. /tmp/token
Using keyword(Library token,lib,ff 00 ff)
Using keyword(Realtime,rt,ff ff)
Keywords: 2
29 hits found
```

Now you can move to the 'search' flag space and list the hits with the 'f' command.

```
[0x08049A80]> fs search
[0x08049A80]> f
...
```

Use the '/n' command to seek between the hits. Or just 'n' and 'N' keys in visual mode.

5.9 Search in assembly

TODO: pd | grep foo

5.10 Searching AES keys

Thanks to Victor Muoz i have added support to the algorithm he developed to find expanded AES keys. It runs the search from the current seek to the cfg.limit or the end of the file. You can always stop the search pressing ^C.

```
$ sudo radare /dev/mem
[0x00000000]> /A
0 AES keys found
```

Chapter 6: Disassembling

Disassembling in radare is just a way to represent a bunch of bytes. So it is handled as a print mode with the 'p' command.

In the old times when radare core was smaller. The disassembler was handled by an external rsc file, so radare was dumping the current block into a file, and the script was just calling objdump in a proper way to disassemble for intel, arm, etc...

Obviously this is a working solution, but takes too much cpu for repeating just the same task so many times, because there are no caches and the scrolling was absolutely slow.

Nowadays, the disassembler is one of the basics in radare allowing you to choose the architecture flavour and some To disassemble use the 'pd' command.

The 'pd' command accepts a numeric argument to specify how many opcodes of the current block do you want to disassemble. Most of the commands in radare are restricted by the block size. So if you want to disassemble more bytes you should use the 'b' command to specify the new block size.

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

The 'pD' command works like 'pd' but gets the number of bytes instead of the number of opcodes.

The 'pseudo' syntax is closer to the humans, but it can be anoying if you are reading lot of code:

```
[0xB7FB8810]> e asm.syntax=pseudo
[0xB7FB8810]> pd 3
0xB7FB8810,    eax = esp
0xB7FB8812   v call 0xB7FB8A60
0xB7FB8817     edi += eax

[0xB7FB8810]> e asm.syntax=intel
[0xB7FB8810]> pd 3
0xB7FB8810,  mov eax, esp
0xB7FB8812   call 0xb7fb8a60
0xB7FB8817   add edi, eax

[0xB7FB8810]> e asm.syntax=att
[0xB7FB8810]> pd 3
0xB7FB8810,  mov %esp, %eax
0xB7FB8812   call 0xb7fb8a60
0xB7FB8817   add %eax, %edi
[0xB7FB8810]>
```

6.1 Adding metadata

The work on binary files makes the task of taking notes and defining information on top of the file

quite important. Radare offers multiple ways to retrieve and adquire this information from many kind of file types.

Following some *nix principles becomes quite easy to write a small utility in shellscript that using objdump, otool, etc.. to get information from a binary and import it into radare just making echo's of the commands script.

You can have a look on one of the many 'rsc' scripts that are distributed with radare like 'idc2rdb':

```
 $ cat src/rsc/pool/idc2rdb

while(<STDIN>) {
        $str=$_;
        if ($str=~/MakeName[^X]*.([^,]*)[^"]*.([^"]*)/) {
                print "f idc_$2 @ 0x$1\n";
        }
        elsif ($str=~/MakeRptCmt[^X]*.([^,]*)[^"]*.([^"]*)/) {
                $cmt = $2;
                $off = $1;
                $cmt=~s/\\n//g;
                print "CC $cmt @ 0x$off\n";
        }
}
```

This script is called with 'rsc idc2rdb < file.idc > file.rdb'. It reads an IDC file exported from an IDA database and imports the comments and the names of the functions.

We can import the 'file.rdb' using the '.' command of radare (similar to the shell):

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

The command '.' is used to interpret data from external resources like files, programs, etc.. In the same way we can do the same without writing a file.

```
[0x00000000]> .!rsc idc2rdb < file.idc
```

The 'C' command is the one used to manage comments and data conversions. So you can define a range of bytes to be interpreted as code, or a string. It is also possible to define flags and execute code in a certain seek to fetch a comment from an external file or database.

Here's the help:

```
[0x4A13B8C0]> C?
Usage: C[op] [arg] <@ offset>
  CC [-][comment] @ here - add/rm comment
  CF [-][len]  @ here    - add/rm function
  Cx [-][addr] @ here    - add/rm code xref
  CX [-][addr] @ here    - add/rm data xref
  Cm [num] [expr]  ; define memory format (pm?)
  Cc [num]         ; converts num bytes to code
  Cd [num]         ; converts to data bytes
  Cs [num]         ; converts to string
  Cf [num]         ; folds num bytes
  Cu [num]         ; unfolds num bytes
  C*               ; list metadata database
```

For example, if you want to add a comment just type:

```
[0x00000000]> CC this guy seems legit @ 0x8048536
```

You can execute code inside the disassembly just placing a flag and assigning a command to it:

```
[0x00000000]> fc !regs @ eip
```

This way radare will show the registers of the cpu printing the opcode at the address where 'eip' points.

In the same way you can interpret structures or fetch information from external files. If you want to execute more than one command in a single address you will have to type them in a file and use the '.' command as explained before.

```
[0x00000000]> fc . script @ eip
```

The 'C' command allows us to change the type of data. The three basic types are: code (disassembly using asm.arch), data (byte array) or string.

In visual mode is easier to manage this because it is hooked to the 'd' key trying to mean 'data type change'. Use the cursor to select a range of bytes ('c' key to toggle cursor mode and HJKL to move with selection) and then press 'ds' to convert to string.

You can use the Cs command from the shell also:

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

The folding/unfolding is quite premature but the idea comes from the 'folder' concepts in vim. So you can select a range of bytes in the disassembly view and press '<' to fold these bytes in a single line or '>' to unfold them. Just to ease the readability of the code.

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

```
[0x4A13B8C0]> Cm 16 2xi foo bar
[0x4A13B8C0]> pd
           0x4A13B8C0,  eip: (pm 2xi foo bar)
0x4a13b8c0 [0] {
       foo : 0x4a13b8c0 = 0x39e8e089
       bar : 0x4a13b8c4 = -1996488697
}
0x4a13b8c8 [1] {
       foo : 0x4a13b8c8 = 0xffe2e8c7
       bar : 0x4a13b8cc = -1014890497
}
      .==< 0x4A13B927      7600            jbe 0x4a13b8c2        ; 1 = eip+0x69
      `--> 0x4A13B929      8dbc2700000000  lea edi, [edi+0x0]
           0x4A13B930,     55              push ebp
           0x4A13B931      89e5            mov ebp, esp
```

This way it is possible to define structures by just using simple oneliners. See 'print memory' for more information.

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

Chapter 7: Projects

When you are working more than once on the same file you will probably be interested in not losing your comments, flags, xrefs analysis and so.

To solve this problem, radare implements 'project' support which can be specified with the '-p' flag. The project files are stored in '~/.radare/rdb' by default which is configured in 'eval dir.project'.

The 'P' command is the one used inside the core to store and load project files. It also can information about the project file.

These files are just radare scripts with some extra metadata as comments ';'.

If you want to make a full analysis when opening a file try setting 'e file.analyze=true' in your .radarerc. It will run '.af* @@ sym_' and more..

Once the program is analyzed (there is no difference between opening the program as a file or debug it) you can store this information in a project file:

```
$ radare -e file.id=1 -e file.flag=1 -e file.analyze=1 -d rasc
...

[0x4A13B8C0]> P?
 Po [file]  open project
 Ps [file]  save project
 Pi [file]  info
[0x4A13B8C0]> Ps rasc
Project saved

[0x4A13B8C0]> Pi rasc
e file.project = rasc
e dir.project = /home/pancake/.radare/rdb/
; file = /usr/bin/rasc
```

This database is stored in:

```
$ du -hs ~/.radare/rdb/rasc
24K
```

Now you can reopen this project in any directory by typing:

```
$ radare -p rasc
```

And if you prefer you can debug it.

```
$ radare -p rasc -d
```

The path to the filename is stored inside the project file, so you dont have to bother about typing it all the time.

The user will be prompted for re-saving the project before exiting.

Chapter 8: Plugins

Radare can be extended in many ways. The most common is by using stdin/stdout get input from a file an interpret the output of the program execution as radare commands. stderr is used for direct user messaging, because it is not handled by the core and it is directly printed in the terminal.

But with this kind of plugins are not directly interactive, because the communication is one-way from the external program to radare. and the only way to get feedback from radare is by using pipes and files. For example:

```
$ cat interactive.rsc
#!/bin/sh
addr=$1
if [ -z "${addr}" ]; then
   echo "No address given"
   exit 1
fi
echo "p8 4 > tmpfile"
sleep 1
bytes=`cat tmpfile`
echo "wx ${bytes} @ ${addr}+4"
```

What this 'dummy' script does is get an address as argument, read 4 bytes from there, and write them at address+4.

As you see this simple task becomes quite 'ugly' using this concepts, so its better to write a native plugin to get full access to the radare internals

8.1 IO backend

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

The IO backend is implement as IO plugins. They are selected depending on the uri file.

```
# debug this file using the debug io plugin
$ radare dbg:///bin/ls

# allocate 10MB in a malloc buffer
$ radare malloc://10M

# allocate 10MB in a malloc buffer
$ radare malloc://10M

# connect to remote host
$ radare connect://192.168.3.33:9999
```

8.2 IO plugins

IO plugins are the ones used to wrap the open, read, write and 'system' on virtual file systems.

The cool thing of IO plugins is that you can make radare understand that any thing can be handled as a plain file. A socket connection, a remote radare session, a file, a process, a device, a gdb session, etc..

So, when radare reads a block of bytes, is the task of the IO plugin to get these bytes from any place and put them in the internal buffer.

IO plugins are selected while opening a file by its URI. Here'r some examples:

```
# Debugging URIs
$ radare dbg:///bin/ls
$ radare pid://1927

# Remote sessions
$ radare listen://:9999
$ radare connect://localhost:9999

# Virtual buffers
$ radare malloc://1024
```

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

```
$ radare -L
haret       Read WCE memory ( haret://host:port )
debug       Debugs or attach to a process ( dbg://file or pid://PID )
gdb         Debugs/attach with gdb (gdb://file, gdb://PID, gdb://host:port)
gdbx        GDB shell interface 'gdbx://program.exe args' )
shm         shared memory ( shm://key )
mmap        memory mapped device ( mmap://file )
malloc      memory allocation ( malloc://size )
remote      TCP IO ( listen://:port or connect://host:port )
winedbg     Wine Debugger interface ( winedbg://program.exe )
socket      socket stream access ( socket://host:port )
gxemul      GxEmul Debugger interface ( gxemul://program.arm )
posix       plain posix file access
```

8.3 Hack plugins

The hack plugins are just shared libraries that have access to some internal apis of radare. The most important one "radare_cmd" which accepts a command string and returns the string representing the output of the execution.

In this way it is possible to perform any action in the core just formatting command strings and parsing its output.

All language bindings (python, lua, ...) are implemented as hack plugins. See 'scripting' section for detailed information.

8.3.1 Jump hacks

The basic radare distribution comes with two plugins to manipulate jumps (actually only x86) but wouldn't be hard to port it to ARM for example.

These ones are: nj and fj. They stand for 'Negate Jump' and 'Force Jump'.

Here's an example of use:

```
[0x465D8AB7]> :pd 1
0x465D8AB7   ^ jle 0x465D8AA3
[0x465D8AB7]> H nj
0x465D8AB7   ^ jg 0x465D8AA3
```

```
[0x465D8AB7]> H fj
0x465D8AB7   ^ jmp 0x465D8AA3
```

Chapter 9: Scripting

Radare is a very versatile application which supports many kinds of scripting features in different languages.

I have already explained how you can write scripts using radare commands (called 'radare scripts'). Or just interpret the output of external applications as radare commands. This kind of unidirectional scripting is interesting for data adquisition, but probably is a mess if you want to make something more interactive or complex.

For this reason radare have a pluggable interface for scripting languages using the plugin-hack API (See 'language bindings' chapter for more information)

9.1 Radare scripts

Radare scripts are just unidirectional scripts that are parsed in the core from a file or from the output of a program.

This methodology is quite used for automatizing simple tasks or for data adquisition.

```
[0x00000000]> !cat binpatch.rsc
wx 90 90 @ 0x300

[0x00000000]> . file        ; interpret this file
```

You can obviously do the same by interpreting the output of a command:

```
[0x00000000]> .! rsc syms-dbg-flag ${FILE}
```

9.2 Boolean expressions

These expressions can be checked for equality for later make conditional execution of commands.

Here is an example that checks if current eip is 0x8048404 and skips this instruction (!jmp eip+2) if matches.

```
> ? eip == 0x8048404
> ??!jmp eip+2
```

You can check the last comparision result with the '???' command. Which is the substraction of the first part of the expression and the second part of it.

```
> ? 1==1   ; check equality (==)
> ???
0x0
> ? 1==2   ; check equality (==)
> ???
0x1
> ? 1!=2   ; check difference (!=)
> ???
0x0
```

The conditional command is given after the '??' command. Which is the help of the '?' command when no arguments given:

```
[0xB7F9D810]> ??
Usage: ?[?[?]] <expr>
  > ? eip             ; get value of eip flag
  > ? 0x80+44         ; calc math expression
  > ? eip-23          ; ops with flags and numbers
  > ? eip==sym_main   ; compare flags
 The '??' is used for conditional executions after a comparision
  > ? [foo] = 0x44    ; compare memory read with byte
  > ???               ; show result of comparision
  > ?? s +3           ; seek current seek + 3 if equal
```

9.3 Macros

The radare shell support macro definitions and these ones can be used to make up your own set of commands into a macro and then use it from the shell by just giving the name and arguments. You can understand a macro as a function.

Let's see how to define a macro:

```
[0x465D8810]> (?
Usage: (foo\n..cmds..\n)
 Record macros grouping commands
 (foo args\n ..)  ; define a macro
 (-foo)            ; remove a macro
 .(foo)            ; to call it
Argument support:
 (foo x y\n$1 @ $2)     ; define fun with args
 .(foo 128 0x804800) ; call it with args
```

The command to manage macros is '('. The first thing we can do is a hello world:

```
[0x465D8810]> (hello
.. !echo Hello World
.. !echo ===========
.. )
[0x465D8810]> .(hello)
Hello World
===========
[0x465D8810]>
```

Macros supports arguments, and they are referenced with $# expressions.

Here's an example of how to define a simple oneliner function called 'foo' accepting two arguments to be used to print 8bit values from an address.

```
; Create our macro
[0x465D8810]> (dump addr len
.. p8 $1 @ $0)

; List defined macros
[0x465D8810]> (
0 dump: p8 $1 @ $0

; Call the macro
[0x465D8810]> .(dump esp 10)
01 00 00 00 e4 17 e6 bf 00 00

; Remove it!
[0x465D8810]> (-dump)
```

We can define these macros in our ~/.radarerc

```
$ cat ~/.radarerc
(dump addr len
  p8 $1 @ $0)
```

9.4 Language bindings

All language bindings supported by radare to script some actions are implemented as hack plugins.

LUA is probably the cleaner implementation of a language binding for radare, i recommend you to read the source at 'src/plug/hack/lua.c'. Here's the structure to register the plugin:

```
int radare_plugin_type = PLUGIN_TYPE_HACK;
struct plugin_hack_t radare_plugin = {
        .name = "lua",
        .desc = "lua plugin",
        .callback = &lua_hack_cmd
};
```

The 'lua_hack_cmd' accepts a string as argument which is the argument given when calling the plugin from the radare shell:

```
[0x00000000]> H lua my-script.lua
```

If no arguments given, the plugin will loop in a prompt executing the lines given as lua statements.

The same happens with other language bindings like ruby, python or perl.

In the same directory where the plugins are installed, there's a "radare.py" or "radare.lua" which describes the API for that language.

The APIs in radare for language bindings are just wrappers for the basic 'r.cmd()' function handled by the core which is hooked to 'radare_cmd()'.

Here's a small part of radare.py to exemplify this:

```
def flag_get(name):
        return r.cmd("? %s"%name).split(" ")[0].strip()

def flag_set(name, addr=None):
        if addr == None:
                r.cmd("f %s"%name)
        else:
                r.cmd("f %s @ 0xx"%name, addr)

def analyze_opcode(addr=None):
        """
        Returns a hashtable containing the information of the analysis of the opcode in the current
        This is: 'opcode', 'size', 'type', 'bytes', 'offset', 'ref', 'jump' and 'fail'
        """
        if addr == None:
                return __str_to_hash(r.cmd("ao"))
        return __str_to_hash(r.cmd("ao @ 0x%x"%addr))
```

The use of these functions is quite natural:

```
from radare import *

aop = analyze_opcode(flag_get("eip"))
if aop["type"] == "jump":
       print "Jumping to 0x%08x"%aop["jump"]
```

Read the 'scripting' chapter to get a deeper look on this topic.

The clearest example about how to implement a language binding for radare is done in Ruby. Read it at src/plug/hack/ruby.c

9.5 LUA

The LUA language aims to be small, simple and fast dynamic language with a well designed core. This was the first language binding implemented in radare for this obvious reasons, and there are some scripts and API available in 'scripts/'.

The main problem of LUA is the lack of libraries and community, so.. sadly for those copypasta developers it is not a productive language.

TODO:...

9.6 Python

The second scripting language implemented in radare was 'python'. Lot of people ping me for adding support for python scripting. The python interface for C is not as nice as the LUA one, and it is obviously not as optimal as LUA, but it gives a very handy syntax and provides a full-featured list of libraries and modules to extend your script.

Actually in python it is possible to write a radare frontend in GTK+ (for example) just calling this from inside the commandline.

The basics of the scripting for any language is the same. The entrypoint between the language and the core is a 'str=r.cmd(str)' function which accepts a string representing a radare command and returns the output of this command as a string.

The file 'radare.py' implements the API for accessing the raw 'r' module which is only loaded from inside the core. (So you cannot use radare-python scripts outside radare (obviously)).

The file 'radapy.py' implements a pure-python radare-remote server and enables a simple interface for extending the basic IO operations thru the network in python. Read 'networking' section for more information.

9.6.1 Python hello world

to start we will write a small python script for radare to just test some of the features of the API.

```
$ cat hello.py
print "Hello World"
seek(0)
print hex(3)
write("90 90 90")
print hex(3)
quit()

$ echo patata > file             # prepare the dummy file
$ radare -i hello.py -wnv file   # launch the script
Hello World
70 61 74
90 90 90
```

If you want a better interface for writing your scripts inside radare use the scriptedit plugin that depends on GTK+ offering a simple editor with language selector and allows to run scripts from

there.

You can also use radare programatically from the python shell:

```
[0x4A13B8C0]> H python
python> print dir(r)
['__doc__', '__name__', 'cmd', 'eval']
python> print(r.cmd("p8 4"))
89 e0 e8 39
```

9.7 Ruby

Use it like in python by refering a global variable called '$r'.

```
[0x465D8810]> H ruby
Load done
==> Loading radare ruby api... ok
irb(main):001:0> $r
=> #<Radare:0xb703ad38>
irb(main):003:0> print $r.cmd("p8 3 @ esp")
01 00 00
irb(main):004:0>
```

Read radare.rb for more information about the API.

Chapter 10: Rabin

Under this bunny-arabic-like name, radare hides the power of a wonderful tool to handle binary files and get information to show it in the command line or import it into the core.

Rabin is able to handle multiple file formats like Java CLASS, ELF, PE, MACH-O, etc.. and it is able to get symbol import/exports, library dependencies, strings of data sections, xrefs, address of entrypoint, sections, architecture type, etc.

```
$ rabin -h
rabin [options] [bin-file]
 -e        shows entrypoints one per line
 -i        imports (symbols imported from libraries)
 -s        symbols (exports)
 -c        header checksum
 -S        show sections
 -l        linked libraries
 -L [lib]  dlopen library and show address
 -z        search for strings in elf non-executable sections
 -x        show xrefs of symbols (-s/-i/-z required)
 -I        show binary info
 -r        output in radare commands
 -v        be verbose
```

The output of every flag is intended to be easily parseable, they can be combined with -v or -vv for a more readable and verbose human output, and -r for using this output from the radare core. Furtheremore, we can combine -s, -i and -z with -x to get xrefs.

10.1 File identification

The file identification is done through the -I flag, it will output information regarding binary class, encoding, OS, type, etc.

```
$ rabin -I /bin/ls
[Information]
class=ELF32
enconding=2's complement, little endian
os=linux
machine=Intel 80386
arch=intel
type=EXEC (Executable file)
stripped=Yes
static=No
baddr=0x0804800
```

As it was said we can add the -r flag to use all this information in radare:

```
$ rabin -Ir /bin/ls
e file.type = elf
e file.baddr = 0x08048000
e cfg.bigendian = false
e dbg.dwarf = false
```

```
e asm.os = linux
e asm.arch = intel
```

This is automatically done at startup if we append to our configuration file (.radarerc) the eval command "eval file.id = true".

10.2 Entrypoint

The flag "-e" lets us know the program entrypoint.

```
$ rabin -e /bin/ls
0x08049a40
```

Again, if we mix it with -v we get a better human readable output.

```
$ rabin -ev /bin/ls
[Entrypoint]
Memory address: 0x08049a40
```

With -vv we will get more information, in this case the memory location as well as the file offset.

```
$ rabin -evv /bin/ls
[Entrypoint]
Memory address: 0x08049a40
File offset:    0x00001a40
```

Combined with -r radare will create a new flag space called "symbols", and it will add a flag named "entrypoint" which points to the program's entrypoint. Thereupon, radare will seek it.

```
$ rabin -er /bin/ls
fs symbols
f entrypoint @ 0x08049a40
s entrypoint
```

10.3 Imports

Rabin is able to get all the imported objects, as well as their offset at the PLT, this information is quite useful, for example, to recognize wich function is called by a call instruction.

```
$ rabin -i /bin/ls | head
[Imports]
address=0x08049484 offset=0x00001484 bind=GLOBAL type=FUNC name=abort
address=0x08049494 offset=0x00001494 bind=GLOBAL type=FUNC name=__errno_location
address=0x080494a4 offset=0x000014a4 bind=GLOBAL type=FUNC name=sigemptyset
address=0x080494b4 offset=0x000014b4 bind=GLOBAL type=FUNC name=sprintf
address=0x080494c4 offset=0x000014c4 bind=GLOBAL type=FUNC name=localeconv
address=0x080494d4 offset=0x000014d4 bind=GLOBAL type=FUNC name=dirfd
address=0x080494e4 offset=0x000014e4 bind=GLOBAL type=FUNC name=__cxa_atexit
address=0x080494f4 offset=0x000014f4 bind=GLOBAL type=FUNC name=strcoll
address=0x08049504 offset=0x00001504 bind=GLOBAL type=FUNC name=fputs_unlocked
(...)
```

The flag -v will output human readable output.

```
$ rabin -iv /bin/ls
[Imports]
Memory address  File offset     Name
0x08049484      0x00001484      abort
0x08049494      0x00001494      __errno_location
0x080494a4      0x000014a4      sigemptyset
0x080494b4      0x000014b4      sprintf
0x080494c4      0x000014c4      localeconv
0x080494d4      0x000014d4      dirfd
```

```
0x080494e4      0x000014e4      __cxa_atexit
0x080494f4      0x000014f4      strcoll
0x08049504      0x00001504      fputs_unlocked
(...)
```

Combined with -vv, we get two new columns, bind (LOCAL, GLOBAL, etc.) and type (OBJECT, FUNC, SECTION, FILE, etc.)

```
$ rabin -ivv /bin/ls
[Imports]
Memory address  File offset     Bind    Type    Name
0x08049484      0x00001484      GLOBAL  FUNC    abort
0x08049494      0x00001494      GLOBAL  FUNC    __errno_location
0x080494a4      0x000014a4      GLOBAL  FUNC    sigemptyset
0x080494b4      0x000014b4      GLOBAL  FUNC    sprintf
0x080494c4      0x000014c4      GLOBAL  FUNC    localeconv
0x080494d4      0x000014d4      GLOBAL  FUNC    dirfd
0x080494e4      0x000014e4      GLOBAL  FUNC    __cxa_atexit
0x080494f4      0x000014f4      GLOBAL  FUNC    strcoll
0x08049504      0x00001504      GLOBAL  FUNC    fputs_unlocked
(...)
```

Again, with -r we can automatically flag them in radare.

```
$ rabin -ir /bin/ls
```

10.4 Symbols (exports)

In rabin, symbols list works in a very similar way as exports do. With the flag -i it will list all the symbols present in the file in a format that can be parsed easily.

```
$ rabin -s /bin/ls
[Symbols]
address=0x0805e3c0 offset=0x000163c0 size=00000004 bind=GLOBAL type=OBJECT name=stdout
address=0x08059b04 offset=0x00011b04 size=00000004 bind=GLOBAL type=OBJECT name=_IO_stdin_used
address=0x0805e3a4 offset=0x000163a4 size=00000004 bind=GLOBAL type=OBJECT name=stderr
address=0x0805e3a0 offset=0x000163a0 size=00000004 bind=GLOBAL type=OBJECT name=optind
address=0x0805e3c4 offset=0x000163c4 size=00000004 bind=GLOBAL type=OBJECT name=optarg
```

With -v, rabin will print a simpler output.

```
$ rabin -sv /bin/ls
[Symbols]
Memory address  File offset     Name
0x0805e3c0      0x000163c0      stdout
0x08059b04      0x00011b04      _IO_stdin_used
0x0805e3a4      0x000163a4      stderr
0x0805e3a0      0x000163a0      optind
0x0805e3c4      0x000163c4      optarg

5 symbols
```

Using -vv, we will get their size, bind and type too.

```
$ rabin -svv /bin/ls
[Symbols]
Memory address  File offset     Size            Bind    Type    Name
0x0805e3c0      0x000163c0      00000004        GLOBAL  OBJECT  stdout
0x08059b04      0x00011b04      00000004        GLOBAL  OBJECT  _IO_stdin_used
0x0805e3a4      0x000163a4      00000004        GLOBAL  OBJECT  stderr
0x0805e3a0      0x000163a0      00000004        GLOBAL  OBJECT  optind
0x0805e3c4      0x000163c4      00000004        GLOBAL  OBJECT  optarg

5 symbols
```

And, finally, with -r radare core can flag automatically all these symbols and define function and data blocks.

```
$ rabin -sr /bin/ls
fs symbols
b 4 && f sym_stdout @ 0x0805e3c0
b 4 && f sym__IO_stdin_used @ 0x08059b04
b 4 && f sym_stderr @ 0x0805e3a4
b 4 && f sym_optind @ 0x0805e3a0
b 4 && f sym_optarg @ 0x0805e3c4
b 512
5 symbols added
```

10.5 Libraries

Rabin can list the libraries used by a binary with the flag -l.

```
$ rabin -l /bin/ls
[Libraries]
librt.so.1
libselinux.so.1
libacl.so.1
libc.so.6
```

There is another flag related to libraries, -L, it dlopens a library and show us the address where it has been loaded.

```
$ rabin -L /usr/lib/librt.so
0x0805e020 /usr/lib/librt.so
```

10.6 Strings

The -z flag is used to list all the strings located in the section .rodata for ELF binaries, and .text for PE ones.

```
$ rabin -z /bin/ls
[Strings]
address=0x08059b08 offset=0x00011b08 size=00000037 type=A name=Try `%s --help' for more...
address=0x08059b30 offset=0x00011b30 size=00000031 type=A name=Usage: %s [OPTION]... [FILE]...
(...)
```

Using -zv we will get a simpler and more readable output.

```
$ rabin -zv /bin/ls
[Strings]
Memory address  File offset     Name
0x08059b08      0x00011b08      Try `%s --help' for more information.
0x08059b30      0x00011b30      Usage: %s [OPTION]... [FILE]...
(...)
```

Combined with -vv, rabin will look for strings within all non-exectable sections (not only .rodata) and print the string size as well as its encoding (Ascii, Unicode).

```
$ rabin -zvv /bin/ls
[Strings]
Memory address  File offset     Size            Type    Name
0x08048134      0x00000134      00000018        A       /lib/ld-linux.so.2
0x08048154      0x00000154      00000003        A       GNU
0x08048b5d      0x00000b5d      00000010        A       librt.so.1
0x08048b68      0x00000b68      00000014        A       __gmon_start__
0x08048b77      0x00000b77      00000019        A       _Jv_RegisterClasses
0x08048b8b      0x00000b8b      00000013        A       clock_gettime
```

```
0x08048b99      0x00000b99      00000015        A       libselinux.so.1
(...)
```

With -r all this information is converted to radare commands, which will create a flag space called "strings" filled with flags for all those strings. Furtheremore, it will redefine them as strings insted of code.

```
$ rabin -zr /bin/ls
fs strings
b 37 && f str_Try___s___help__for_more_information_ @ 0x08059b08
Cs 37 @ 0x08059b08
b 31 && f str_Usage___s__OPTION______FILE____ @ 0x08059b30
Cs 31 @ 0x08059b30
(...)
```

10.7 Program sections

Rabin give us complete information about the program sections. We can know their index, offset, size, align, type and permissions, as we can see in the next example.

```
$ rabin -Svv /bin/ls
[Sections]
Section index  Memory address  File offset Size      Align       Privileges  Name
00             0x08048000      0x00000000  00000000  0x00000000  ---
01             0x08048134      0x00000134  00000019  0x00000001  r--         .interp
02             0x08048148      0x00000148  00000032  0x00000004  r--         .note.ABI-tag
03             0x08048168      0x00000168  00000808  0x00000004  r--         .hash
04             0x08048490      0x00000490  00000092  0x00000004  r--         .gnu.hash
05             0x080484ec      0x000004ec  00001648  0x00000004  r--         .dynsym
06             0x08048b5c      0x00000b5c  00001127  0x00000001  r--         .dynstr
07             0x08048fc4      0x00000fc4  00000206  0x00000002  r--         .gnu.version
08             0x08049094      0x00001094  00000176  0x00000004  r--         .gnu.version_r
09             0x08049144      0x00001144  00000040  0x00000004  r--         .rel.dyn
10             0x0804916c      0x0000116c  00000728  0x00000004  r--         .rel.plt
11             0x08049444      0x00001444  00000048  0x00000004  r-x         .init
12             0x08049474      0x00001474  00001472  0x00000004  r-x         .plt
13             0x08049a40      0x00001a40  00065692  0x00000010  r-x         .text
14             0x08059adc      0x00011adc  00000028  0x00000004  r-x         .fini
15             0x08059b00      0x00011b00  00015948  0x00000020  r--         .rodata
16             0x0805d94c      0x0001594c  00000044  0x00000004  r--         .eh_frame_hdr
17             0x0805d978      0x00015978  00000156  0x00000004  r--         .eh_frame
18             0x0805e000      0x00016000  00000008  0x00000004  rw-         .ctors
19             0x0805e008      0x00016008  00000008  0x00000004  rw-         .dtors
20             0x0805e010      0x00016010  00000004  0x00000004  rw-         .jcr
21             0x0805e014      0x00016014  00000232  0x00000004  rw-         .dynamic
22             0x0805e0fc      0x000160fc  00000008  0x00000004  rw-         .got
23             0x0805e104      0x00016104  00000376  0x00000004  rw-         .got.plt
24             0x0805e280      0x00016280  00000272  0x00000020  rw-         .data
25             0x0805e390      0x00016390  00001132  0x00000020  rw-         .bss
26             0x0805e390      0x00016390  00000208  0x00000001  ---         .shstrtab

27 sections
```

Also, using -r, radare will flag the beginning and end of each section, as well as comment each one with the previous information.

```
$ rabin -Sr /bin/ls
fs sections
f section_ @ 0x08048000
f section__end @ 0x08048000
CC [00] 0x08048000 size=00000000 align=0x00000000 ---  @ 0x08048000
f section__interp @ 0x08048134
f section__interp_end @ 0x08048147
CC [01] 0x08048134 size=00000019 align=0x00000001 r-- .interp @ 0x08048134
```

```
f section__note_ABI_tag @ 0x08048148
f section__note_ABI_tag_end @ 0x08048168
CC [02] 0x08048148 size=00000032 align=0x00000004 r-- .note.ABI-tag @ 0x08048148
f section__hash @ 0x08048168
f section__hash_end @ 0x08048490
CC [03] 0x08048168 size=00000808 align=0x00000004 r-- .hash @ 0x08048168
f section__gnu_hash @ 0x08048490
f section__gnu_hash_end @ 0x080484ec
CC [04] 0x08048490 size=00000092 align=0x00000004 r-- .gnu.hash @ 0x08048490
f section__dynsym @ 0x080484ec
f section__dynsym_end @ 0x08048b5c
(...)
```

Take care of adding "eval file.flag = true" to .radarerc radare executes rabin -risSz at startup, automatically flaging the file.

Chapter 11: Networking

Radare have some interesting features in the networking area. It can be used as a hexadecimal netcat-like application using the io socket plugin which offers a file-like interface to access a TCP/IP connection.

The radare remote protocol allows to remotelly expand the IO of radare using a TCP connection. There's a pure-python implementation that has been used to implement python-based debuggers or just to offer a radare access to Bochs, vtrace or Immunity debugger for example.

11.1 IO Sockets

The IO plugin called 'socket' generates a virtual file using a malloc-ed buffer which grows when receiving data from the socket and writing data to it in.

```
$ radare socket://av.com:80/
[0x00000000]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00000000, ffff ffff ffff ffff ffff ffff ffff ffff ffff ..................
0x00000012, ffff ffff ffff ffff ffff ffff ffff ffff ffff ..................
...
```

When writing the socket:// plugin redirects it to the socket.

```
[0x00000000]> w GET / HTTP/1.1\r\nHost: av.com\r\n\r\n
[0x00000000]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x00000000, 4854 5450 2f31 2e31 2033 3031 204d 6f76 6564 HTTP/1.1 301 Moved
0x00000012  2050 6572 6d61 6e65 6e74 6c79 0d0a 4461 7465  Permanently..Date
0x00000024, 3a20 4d6f 6e2c 2032 3920 5365 7020 3230 3038 : Mon, 29 Sep 2008
0x00000036  2031 313a 3035 3a34 3320 474d 540d 0a4c 6f63  11:05:43 GMT..Loc
0x00000048, 6174 696f 6e3a 2068 7474 703a 2f2f 7777 772e ation: http://www.
0x0000005A  616c 7461 7669 7374 612e 636f 6d2f 0d0a 436f altavista.com/..Co
0x0000006C, 6e6e 6563 7469 6f6e 3a20 636c 6f73 650d 0a54 nnection: close..T
0x0000007E  7261 6e73 6665 722d 456e 636f 6469 6e67 3a20 ransfer-Encoding:
0x00000090, 6368 756e 6b65 640d 0a43 6f6e 7465 6e74 2d54 chunked..Content-T
0x000000A2  7970 653a 2074 6578 742f 6874 6d6c 3b20 6368 ype: text/html; ch
0x000000B4, 6172 7365 743d 7574 662d 380d 0a0d 0a39 3720 arset=utf-8....97
0x000000C6  2020 2020 0d0a 5468 6520 646f 6375 6d65 6e74     ..The document
0x000000D8, 2068 6173 206d 6f76 6564 203c 4120 4852 4546  has moved <A HREF
0x000000EA  3d22 6874 7470 3a2f 2f77 7777 2e61 6c74 6176 ="http://www.altav
0x000000FC, 6973 7461 2e63 6f6d 2f22 3e68 6572 653c 2f41 ista.com/">here</A
0x0000010E  3e2e 3c50 3e0a 3c21 2d2d 2070 332e 7263 2e72 >.<P>.<!-- p3.rc.r
0x00000120, 6534 2e79 6168 6f6f 2e63 6f6d 2075 6e63 6f6d e4.yahoo.com uncom
0x00000132  7072 6573 7365 642f 6368 756e 6b65 6420 4d6f pressed/chunked Mo
0x00000144, 6e20 5365 7020 3239 2030 343a 3035 3a34 3320 n Sep 29 04:05:43
0x00000156  5044 5420 3230 3038 202d 3e0a 0d0a 0d0a PDT 2008 -->...0..
0x00000168, 0d0a ffff ffff ffff ffff ffff ffff ffff ffff ..................
```

The contents of the file are updated automatically while the socket is feeded by bytes. You can understand this plugin as a raw hexadecimal netcat with a nice interface ;)

All reads from the socket are stored as flags pointing to the last read packet:

```
[0x00000000]> f
0x00000000       512     _sockread_0
0x00000000       512     _sockread_last
```

11.2 Radare remote

The 'io/remote' plugin implements a simple binary protocol for connecting client/server radare implementation and extend the basic IO operations over the network.

An example of use would be:

```
(alice)$ radare listen://:9999
Listening at port 9999

(bob)$ radare connect://alice:9999/dbg:///bin/ls
...
```

Once bob connects to alice using the listen/connect URIs (both handled by the remote plugin) the listening one loads the nested uri and tries to load it "dbg:///bin/ls". Both radares will be working in debugger mode and all the debugger commands will be wrapped by network.

In the same way it is possible to nest multiple socket connections between radare.

11.3 radapy

The radapy is the python implementation for the radare remote protocol. This module is distributed in the scripts/ directory of radare. For better understanding, here's an example:

```
$ cat scripts/radapy-example.py
import radapy
from string import *

PORT = 9999

def fun_system(str):
        print "CURRENT SEEK IS %d"%radapy.offset
        return str

def fun_open(file,flags):
        return str

def fun_seek(off,type):
        return str

def fun_write(buf):
        print "WRITING %d bytes (%s)"%(len(buf),buf)
        return 6

def fun_read(len):
        print "READ %d bytes from %d\n"% (len, radapy.offset)
        str = "patata"
        str = str[radapy.offset:]
        return str

#radapy.handle_cmd_close = fun_close
radapy.handle_cmd_system = fun_system
radapy.handle_cmd_read = fun_read
radapy.handle_cmd_write = fun_write
radapy.size = 10

radapy.listen_tcp (PORT)
```

As you see, you just need to implement the 'read' command and all the rest will mostly work. Here's a shorter implementation for immunity debugger:

```
import immlib
import radapy

def fun_read(len):
    return immlib.Debugger().readMemory(radapy.offset, len)

radapy.handle_cmd_read = fun_read
radapy.listen_tcp ( 9999 )
```

For the other side you just have to connect a radare to this port to get the fun:

```
$ radare connect://127.0.0.1:9999/dbg://immunity
```

11.4 IO thru Syscall proxying

TODO

Chapter 12: Rsc toolset

RSC stands for 'radare scripts' which are a set of scripts accessible thru the 'rsc' command and allow to perform different tasks or provide small utilities that can interact with radare in some way.

12.1 asm/dasm

There are two rsc scripts that emulate 'rasm' to assemble and disassemble single opcodes for multiple architectures from the command line.

```
$ rsc asm 'mov eax,33'
b8 21 00 00 00

$ rsc dasm 'b8 21 00 00 00'
   0:   b8 21 00 00 00          mov    $0x21,%eax
```

If you pay attention to the output you'll notice that it's AT&T syntax and the formatting is the objdump one. Looking the scripts will make you understand that it's using 'gas' and 'nasm' for assembling and objdump for disassembling.

Compare this with rasm:

```
$ rasm 'mov eax,33'
b8 21 00 00 00
$ rasm -d 'b8 21 00 00 00'
mov eax, 0x21
```

12.2 idc2rdb

Use this script to convert IDC scripts exported from an IDA database to import all the metadata into radare.

```
$ rsc idc2rdb < my-database.idc > mydb.radare.script
```

12.3 gokolu

Gokolu is a perl script that strips strings from a program and find them in google code search to try to identify which libraries or files has been linked against the target binary.

Here's an usage example:

```
pancake@flubox:~$ rsc gokolu /bin/ls Usage
The Go*g*e Kode Lurker v0.1
=> Usage: %s [OPTION]... [FILE]...
 12 ftp://alpha.gnu.org/gnu/coreutils/coreutils-4.5.4.tar.bz2
  0 ftp://alpha.gnu.org/gnu/coreutils/coreutils-4.5.3.tar.gz
```

Nice huh? ;)

```
FMI: http://www.openrce.org/blog/view/1001/Gokolu_-_Binary_string_source_identifier
```

Chapter 13: Rasm

The inline assembler/disassembler. Initially 'rasm' was designed to be used for binary patching, just to get the bytes of a certain opcode. Here's the help

```
$ rasm -h
Usage: rasm [-elvV] [-f file] [-s offset] [-a arch] [-d bytes] "opcode"|-
 if 'opcode' is '-' reads from stdin
  -v           enables debug
  -d [bytes]   disassemble from hexpair bytes
  -f [file]    compiles assembly file to 'file'.o
  -s [offset]  offset where this opcode is suposed to be
  -a [arch]    selected architecture (x86, olly, ppc, arm, java, rsc)
  -e           use big endian
  -l           list all supported opcodes and architectures
  -V           show version information
```

The basic 'portable' assembler instructions can be listed with 'rasm -l':

```
$ rasm -l
Usage: rasm [-elvV] [-f file] [-s offset] [-a arch] [-d bytes] "opcode"|-
Architectures:
 olly, x86, ppc, arm, java
Opcodes:
 call [addr]  - call to address
 jmp [addr]   - jump to relative address
 jz  [addr]   - jump if equal
 jnz          - jump if not equal
 trap         - trap into the debugger
 nop          - no operation
 push 33      - push a value or reg in stack
 pop eax      - pop into a register
 int 0x80     - system call interrupt
 ret          - return from subroutine
 ret0         - return 0 from subroutine
 hang         - hang (infinite loop
 mov eax, 33  - asign a value to a register
Directives:
 .zero 23     - fill 23 zeroes
 .org 0x8000  - offset
```

13.1 Assemble

It is quite common to use 'rasm' from the shell. It is a nice utility for copypasting the hexpairs that represent the opcode.

```
$ rasm -a x86 'jmp 0x8048198'
e9 bb 9e fe ff
rasm -a ppc 'jmp 0x8048198'
48 fa 1d 28
```

Rasm is used from radare core to write bytes using 'wa' command. So you can directly an opcode from the radare shell.

It is possible to assemble for x86 (intel syntax), olly (olly syntax), powerpc, arm and java. For the rest of architectures you can use 'rsc asm' that takes $OBJDUMP and $AS to generate the proper output after assembling the given instruction. For the intel syntax, rasm tries to use NASM or GAS. You can use the SYNTAX environment variable to choose your favorite syntax: intel or att.

There are some examples in rasm's source directory to assemble a raw file using rasm from a file describing these opcodes.

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

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

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

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

  ret

$ rasm -f selfstop.rasm
$ ls
selfstop.rasm selfstop.rasm.o
$ echo pd | radare -vn ./selfstop.rasm
  0x00000000,    cursor: 6800800408        push dword 0x8048000
  0x00000005             60                pushad
  0x00000006             b814000000        eax = 0x14
  0x0000000B             cd80              int 0x80
  0x0000000D             89d8              eax = ebx
  0x0000000F             b913000000        ecx = 0x13
  0x00000014,            b825000000        eax = 0x25
  0x00000019             cd80              int 0x80
  0x0000001B             61                popad
  0x0000001C,            c3               ret ;--
  0x0000001C          ; ------------------------------------
  0x0000001D             c3               ret ;--
  0x0000001D          ; ------------------------------------
```

13.2 Disassemble

In the same way as rasm assembler works, giving the '-d' flag you can disassemble an hexpair string:

```
$ rasm -d 'b8 21 00 00 00'
mov eax, 33
```

Chapter 14: Rasc

RASC stands for 'Radare ShellCodes' and aims to be a tool to be used to develop and automatize the use of simple shellcodes and buffer/heap overflow attacks on controlled environments.

14.1 Shellcodes

Rasc contains a database of small shellcodes for multiple operating systems and so..it is useful for fast exploiting on controlled environments. You can get the list with the '-L' flag. Choose it with the '-i' flag.

You can also specify your own shellcode in hexpairs with the '-s' flag or just get it from a raw binary file with the '-S' one.

The return address can be specified with '-a' so you will not have to manually rewrite the return address for multiple tests.

```
$ rasc -h | grep addr
  -a addr@off  set the return address at a specified offset

$ rasc -N 20 -i x86.freebsd.reboot -x -a 0x8048404@2
90 90 04 84 04 08 90 90 90 90 90 90 90 90 90 90 90 90 41 31 c0 50 b0 37 cd 80
```

The supported output formats are:

```
  -c           output in C format
  -e           output in escapped string
  -x           output in hexpairs format
  -X           execute shellcode
```

Some of these shellcodes can be modified by environment variables:

```
 Environment variables to compile shellcodes:
  CMD          Command to execute on execves
  HOST         Host to connect
  PORT         Port to listen or connect

$ rasc -i x86.linux.binsh -x
41 31 c0 50 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 50 53 89 e1 99 b0 0b cd 80

$ rasc -L
arm.linux.binsh        47   Runs /bin/sh
arm.linux.suidsh       67   Setuid and runs /bin/sh
arm.linux.bind        203   Binds /bin/sh to a tcp port
armle.osx.reverse     151   iPhone reverse connect shell to HOST and PORT
dual.linux.binsh       99   x86/ppc MacOSX /bin/sh shellcode
dual.osx.binsh        121   Runs /bin/sh (works also on x86) (dual)
mips.linux.binsh       87   Runs /bin/sh (tested on loongson2f).
ppc.osx.adduser       219   Adds a root user named 'r00t' with no pass.
ppc.osx.binsh         152   Executes /bin/sh
ppc.osx.binsh0         72   Executes /bin/sh (with zeroes)
ppc.osx.bind4444      224   Binds a shell at port 4444
ppc.osx.reboot         28   Reboots the box
```

```
ppc.bsd.binsh          119   Runs /bin/sh
sparc.linux.binsh      216   Runs /bin/sh on sparc/linux
sparc.linux.bind4444   232   Binds a shell at TCP port 4444
ia64.linux.binsh        63   Executes /bin/sh on Intel Itanium
x64.linux.binsh         46   Runs /bin/sh on 64 bits
x86.bsd.binsh           46   Executes /bin/sh
x86.bsd.binsh2          23   Executes /bin/sh
x86.bsd.suidsh          31   Setuid(0) and runs /bin/sh
x86.bsd.bind4444       104   Binds a shell at port 4444
x86.bsdlinux.binsh      38   Dual linux/bsd shellcode runs /bin/sh
x86.freebsd.reboot       7   Reboots target box
x86.freebsd.reverse    126   Reboots target box
x86.linux.adduser       88   Adds user 'x' with password 'y'
x86.linux.bind4444     109   Binds a shell at TCP port 4444
x86.linux.binsh         24   Executes /bin/sh
x86.linux.binsh1        31   Executes /bin/sh
x86.linux.binsh2        36   Executes /bin/sh
x86.linux.binsh3        50   Executes /bin/sh or CMD
x86.linux.udp4444      125   Binds a shell at UDP port 4444
x86.netbsd.binsh        68   Executes /bin/sh
x86.openbsd.binsh       23   Executes /bin/sh
x86.openbsd.bind6969   147   Executes /bin/sh
x86.osx.binsh           45   Executes /bin/sh
x86.osx.binsh2          24   Executes /bin/sh
x86.osx.bind4444       112   Binds a shell at port 4444
x86.solaris.binsh       84   Runs /bin/sh
x86.solaris.binshu      84   Runs /bin/sh (toupper() safe)
x86.solaris.bind4444   120   Binds a shell at port 4444
x86.w32.msg            245   Shows a MessageBox
x86.w32.cmd            164   Runs cmd.exe and ExitThread
x86.w32.adduser        224   Adds user 'x' with password 'y'
x86.w32.bind4444       345   Binds a shell at port 4444
x86.w32.tcp4444        312   Binds a shell at port 4444
```

14.2 Paddings

The shellcode can be easily modified to have different prefixes and suffixes to ease the debugging of the exploit development. These flags are:

```
-A [n]       prefix shellcode with N A's (0x41)
-N [n]       prefix shellcode with N nops (0x90)
-C [n]       suffix shellcode with N traps
-E [n]       prefix with enumeration 01 02 03..
```

14.3 Syscall proxying

TODO

Chapter 15: Analysis

There are different commands to perform data and code analysis and extract information like pointers, string references, basic blocks, extract opcode information, jump information, xrefs, etc..

Those operations are handled by the root 'a'nalyze command:

```
Usage: a[ocdg] [depth]
 ao [nops]    analyze N opcodes
 ab [num]     analyze N code blocks
 af [name]    analyze function
 ac [num]     disasm and analyze N code blocks
 ad [num]     analyze N data blocks
 ag [depth]   graph analyzed code
 as [name]    analyze spcc structure (uses dir.spcc)
 at [args]    analyze opcode traces
 av [nops]    analyze virtual machine (negative resets before)
 ax           analyze xrefs
```

15.1 Code analysis

The code analysis is a common technique used to extract information from the assembly code. Radare stores multiple internal data structures to identify basic blocks, function trees, extract opcode-level information and such.

15.1.1 Functions

We can use rabin to import the function definitions directly from the program

```
[0xB7F14810]> .!rabin -rs $FILE
```

Rabin will mark each function with 'CF <size> @ <fun-addr>', add comments for stack usage, and setup flags for each function symbols.

To make a function analysis use the 'af' command which will analyze the code from the current seek and tries to identify the end of the function. The command will just output some radare commands, Use '.af*' to interpret them.

```
[0xB7F94810]> af @ sym_main
offset = 0x080499b7
label = sym_main
size = 832
blocks = 2
framesize = 0
ncalls = 21
xrefs = 0
args = 3
vars = 5
```

This report shows information about the function analysis. it is useful for scripting, so it is possible

to do function signatures easily to identify functions in static bins from previously captured library signatures. (for example) So you can use these metrics to identify if two functions are the same or not.

Once you read this report it is possible to import this information into the core:

```
[0xB7F93A60]> af*
; from = 0xb7f93a60
; to   = 0xb7f93c0d
CF 430 @ 0xb7f93a60
CC Stack size +40 @ 0xb7f93a71
CC Set var32 @ 0xb7f93a74
...

[0xB7F93A60]> .af*
```

The function analysis stops when reaching calls, use the 'aF' command to analyze the functions recursively. Use it like in 'af'

```
[0xB7F93A60]> .aF
```

You can also analyze all the symbols of a binary using the '@@' operator.

```
[0xB7F94A60]> .af* @@ sym_
```

To read the xref information try with:

```
[0xB7F94A60]> pd 1 @@ imp_printf
; CODE xref 0804923e (sym_main+0x1a2)
0x08048CB8,  imp_printf:
0x08048CB8      v goto dword near [0x805e164]

[0xB7FD9810]> pd 1 @ sym_main+0x1a2
0x0804923E     ^ call 0x8048CB8     ; 1 = imp_printf
```

15.1.2 Basic blocks

A basic block is cosidered a sequence of opcodes ended up by a jump/branch/ret instruction. Radare allows you to determine which kind of basic block splitting you want to use to have a more personalized way to read the code and be able to split the functions as basic blocks for example.

Use these variables to configure how the code analysis should work to determine basic blocks:

```
graph.jmpblocks = true
graph.refblocks = false
graph.callblocks = false
graph.flagblocks = true
```

The 'ab' command will look for a basic block definition starting at current seek:

```
[0xB7FC4810]> ab
offset = 0xb7fc4810
type = head
size = 73
call0 = 0xb7fc4a60
call1 = 0xb7fc4800
call2 = 0xb7fd1a40
n_calls = 3
bytes = 89 e0 e8 49 02 00 00 89 c7 e8 e2 ff ff ff ...
```

The output is quite parsing friendly, so you can easily use the python API to extract this information:

```
[0xB7F14810]> H python
python> block = radare.analyze_block()
python> print "This basic block have %d calls.\n" % radare.block['n_calls']

This basic block have 3 calls.

python> print "Its size is %d bytes.\n" % block['size']

Its size is 74 bytes
```

There's a human friendly version of 'ab' called 'ac' which stands for 'analyze code'. It accepts a numeric argument to determine the depth level when performing the basic block analysis.

```
[0xB7F93A60]> ac 10
0xb7f93a60 (0) -> 0xb7f93ae1, 0xb7f93a99
         0xB7F93A60,       eip:   push ebp
         0xB7F93A61               ebp = esp
         0xB7F93A63               push edi
         ...
         0xB7F93A8F               [ebx+0x2b4] = eax
         0xB7F93A95               test edx, edx
         0xB7F93A97             v jz 0xB7F93AE1      ; 2 = eip+0x7b

0xb7f93ae1 (0) -> 0xb7f93b53, 0xb7f93aeb
         0xB7F93AE1               edx = [ebx+0x2ac]
         0xB7F93AE7               test edx, edx
         0xB7F93AE9             v jz 0xB7F93B53      ; 1

0xb7f93b53 (0) -> 0xb7f93b67, 0xb7f93b5d
         0xB7F93B53               eax = [ebx+0x31c]
         0xB7F93B59               test eax, eax
    .==< 0xB7F93B5B             v jz 0xB7F93B67      ; 1 = eip+0x107

0xb7f93b67 (0) -> 0xb7f93b81, 0xb7f93b71
         0xB7F93B67               eax = [ebx+0x310]
         0xB7F93B6D               test eax, eax
    .==< 0xB7F93B6F             v jz 0xB7F93B81      ; 1 = eip+0x121
         ...
```

The values shown are:

```
address (0) -> true-jump, false-jump
```

15.1.3 Opcodes

In the same way as 'ab' or 'ac' works. Radare exposes an opcode-level analysis with the 'ao' command.

Here's a simple example

```
[0xB7F93A66]> pd 1
0xB7F93A66        eip: v call 0xB7FA87AB

[0xB7F93A66]> ao
index = 0
size = 5
type = call
bytes = e8 40 4d 01 00
offset = 0xb7f93a66
ref = 0x00000000
jump = 0xb7fa87ab
fail = 0xb7f93a6b
```

```
[0xB7F14810]> H python
python> op = analyze_opcode()
python> print "0x%x\n"%op['offset']
0xb7f14810
```

15.2 Opcode traces

The 'at' command is the one used to get information about the executed opcodes while debugging. It registers memory ranges and it is useful to determine which parts of the program are executed under a debugging session.

```
[0xB7F93A60]> at?
Usage: at[*] [addr]
   > at         ; list all traced opcode ranges
   > at-        ; reset the tracing information
   > at*        ; list all traced opcode offsets
   > at [addr]  ; show trace info at address

[0xB7F93A60]> at
0xb7f93810 - 0xb7f93817
0xb7f93a60 - 0xb7f93b8b
0xb7f93b95 - 0xb7f93b9f
0xb7f93ba9 - 0xb7f93bd3
0xb7f93be1 - 0xb7f93c63
0xb7f93c70 - 0xb7f93c80
0xb7f93c93 - 0xb7f93d1e
0xb7f93d23 - 0xb7f93db8
```

To reset this information just type 'at-'. There are API functions to get this information from python, lua, etc..

15.2.1 Opcode emulation

The 'av' stands for 'Analyze' using 'Virtual machine'. It is used to emulate machine code to determine values of registers at a certain part of the program. This is used to resolve register branches and similar stuff.

TODO : actually it is in a very newborn state.

15.3 Data analysis

There are some basic data analysis functions implemented in radare. The most basic one is just a memory parser that tries to identify pointers to flags, strings, linked lists, handled endianness with 'cfg.bigendian' and displays integer values of contained dwords and strings. Here's a simple example from a basic debugger session to analyze the stack to get information about pointers.

```
[0xB7EEC810]> ad @ esp
0xBFBEAA80, int be=0x01000000 le=0x00000001 , (le= 1 )
0xBFBEAA84, int be=0xe4b7bebf le=0xbfbeb7e4
   0xBFBEB7E4, string "/bin/ls"
   0xBFBEB7EC, string "GPG_AGENT_INFO=/tmp/gpg-mJ80Cm/S.gpg-agent:7090:1"
   0xBFBEB81E  string "TERM=xterm"
   0xBFBEB829  string "SHELL=/bin/bash"
   0xBFBEAA88, (NULL)
```

15.3.1 Structures

The 'as' command is used to interpret a buffer of bytes using an spcc program (See 'rsc spcc' for more information). Giving a name of function it firstly edits the C code and later runs 'rsc spcc'

to compile the parser and interpret the current block. The spcc files are stored in ~/.radare/spcc/, so you can edit the structure definition by browsing in this directory.

Here's a sample session:

```
[0xBFBEAA80]> as
Usage: as [?][-][file]
Analyze structure using the spcc descriptor
  > as name   :  create/show structure
  > as -name  :  edit structure
  > as ?      :  list all spcc in dir.spcc

[0xBFBEAA80]> as foo
# .. vi ~/.radare/spcc/foo.spcc
struct foo {
        int id;
        void *next;
        void *prev;
};

void parse(struct spcc *spcc, uchar *buffer) {
        struct foo tmp;
        memcpy(&tmp, buffer, sizeof(struct foo));
        printf("id: %d\nnext: %p\nprev: %p\n",
                tmp.id, tmp.next, tmp.prev);
}
~
~
[0xBFBEAA80]> as foo
id: 1
next: 0xbfbeb7e4
prev: (nil)
```

15.3.2 Spcc

SPCC stands for 'Structure Parser C Compiler'. And it is just a small script that generates a dummy main() to call a user specified function over a block of data from radare. This way it is possible to parse any buffer using just C with all the libraries and includes like a real program will do. Use 'pm' command if you want a oneline and simplified version for reading structures.

```
$ rsc spcc
spcc - structure parser c compiler
Usage: spcc [-ht] [file.spc] ([gcc-flags])
```

The 'rsc spcc' command should be used like gcc against .spc files that are just .c files without main() and having a function called 'void parse(struct spcc *spcc, uchar *buffer)'.

```
$ rsc spcc -t
/*-- test.spcc --*/
struct foo {
        int id;
        void *next;
        void *prev;
};

void parse(struct spcc *spcc, uchar *buffer) {
        struct foo tmp;
        memcpy(&tmp, buffer, sizeof(struct foo));
        printf("id: %d\nnext: %p\nprev: %p\n",
                tmp.id, tmp.next, tmp.prev);
}
```

15.3.3 Trace analysis

The radare debugger stores information about the executed opcodes in a linked list and allows later to get an execution trace of the code with a list of ranged addresses of the executed code and how many times and in which order these instructions has been executed.

This information is managed with the 'at' command. 'Analyze Traces'.

```
[0x4A13C00E]> at
0x4a13b8c0 - 0x4a13b8c7
0x4a13c000 - 0x4a13c048
0x4a13c050 - 0x4a13c066
0x4a1508cb - 0x4a1508cf
```

To get a complete list of all the executed addresses use 'at*'. The first column represents the offset and the second one the number of times it has been executed and the third one the last step counter value for this offset.

```
[0x4A13C00E]> at*
0x4a13b8c0 1 1
0x4a13b8c2 1 2
0x4a13c000 1 3
0x4a13c001 1 4
0x4a13c003 1 5
0x4a13c004 1 6
0x4a13c005 1 7
0x4a13c006 1 8
0x4a13c009 1 9
0x4a1508cb 1 10
0x4a1508ce 1 11
0x4a13c00e 1 12
0x4a13c014 1 13
0x4a13c017 1 14
0x4a13c019 1 15
0x4a13c01f 1 16
0x4a13c025 1 17
0x4a13c02b 1 18
0x4a13c031 1 19
0x4a13c033 1 20
0x4a13c036 1 21
0x4a13c03c 1 22
0x4a13c042 1 23
0x4a13c044 1 24
0x4a13c046 1 25
0x4a13c061 5 54
0x4a13c064 5 55
0x4a13c050 5 56
0x4a13c057 5 57
0x4a13c05a 5 58
0x4a13c05d 5 59
0x4a13c05f 4 53
[0x4A13C00E]>
```

With this log we can easily identify the loops, how many times they have been executed and the execution order of them.

You can also get detailed information of a certain offset:

```
[0x4A13C05A]> !at 0x4a13c064
ffset = 0x4a13c064
opsize = 2
times = 5
count = 55
```

15.4 Graphing code

radare offers a graphical interface for browsing the code graphs. It uses an own-made Vala library called 'grava' that permits the creation of graphs using Gtk+Cairo easily. To open a new window displaying the code analysis graph type: 'ag'.

```
[0x08048923]> ag
... gui ...
```

The grava window permits opening new nodes with the content executing a certain command (!regs, x@esp, ..), and offers a simple interface for debugging allowing to browse the code with the mouse and set breakpoints with the contextual menu over the nodes.

TODO: screenshots

Chapter 16: Gradare

The current graphical frontend of radare is based on a simple Gtk+Vte+Cairo application running a radare application inside the virtual terminal (vte) and feeding it with the output of some scripts called 'grsc' stored in $prefix/share/radare/gradare/ in subdirectories per categories.

```
$ pwd
/usr/share/radare/gradare

$ ls
Config  Debugger  Disassembly  Flags  Hacks  Movement  Search  Shell  Visual

$ ls Debugger/
AttachOrLoad  Continue          Detach  Registers    Status  StepOver      Stop
Breakpoint    ContinueUserCode  Maps    SetRegister  Step    StepUserCode

$ cat Debugger/StepOver
#!/bin/sh
echo S
```

TODO ...

Chapter 17: Rahash

The it is quite easy to calculate a hash checksum of the current block using the '#' command.

Change the block size, seek to the interesting offset and calculate the md5 of it.

```
$ radare /bin/ls
[0x08048000]> s section__text
[0x08049790]> b section__text_end-section__text
[0x08049790]> #md5
d2994c75adaa58392f953a448de5fba7
```

In the same way you can also calculate other hashing algorithms that are supported by 'rahash': md4, md5, crc16, crc32, sha1, sha256, sha384, sha512, par, xor, xorpair, mod255, hamdist, entropy, all.

The '#' command can accept a numeric argument to define the length in bytes to be hashed.

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

17.1 Rahash tool

The rahash tool is the used by radare to realize these calculations. It

```
$ rahash -h
rahash [-action] [-options] [source] [hash-file]
 actions:
  -g           generate (default action)
  -c           check changes between source and hash-file
  -o           shows the contents of the source hash-file
  -A           use all hash algorithms
 options:
  -a [algo]    algorithm to hash (md4, md5, crc16, crc32, sha1, sha256, sha384, sha512, par, xor, x
  -s [string]  hash this string instead of a file
  -S [offset]  seek initial offset to
  -E [offset]  end hashing at offset
  -L [length]  end hashing at length
  -b [size]    sets the block size (default 32KB)
  -f           block size = file size (!!)
  -q           quite output (can be combined with -v)
  -V           show version information
  -v           be verbose
```

It permits the calculation of the hashes from strings or files.

```
$ rahash -a md5 -s 'hello world'
5eb63bbbe01eeed093cb22bb8f5acdc3
```

It is possible to hash the full contents of a file by giving '-f' as argument. But dont do this for large files like disks or so, because rahash stores the buffer in memory before calculating the checksum instead of doing it progressively.

```
$ rahash -a all -f /bin/ls
par:     1
xor:     ae
hamdist: 00
xorpair: 11bf
entropy: 6.08
mod255:  ea
crc16:   41a4
crc32:   d34e458d
md4:     f0bfd80cea21ca98cc48aefef8d71f3e
md5:     f58860f27dd2673111083770c9445099
sha1:    bfb9b77a29318fc6a075323c67af74d5e3071232
sha256:  8c0d752022269a8673dc38ef5d548c991bc7913a43fe3f1d4d076052d6a5f0b6
sha384:  1471bd8b14c2e11b3bcedcaa23209f2b87154e0daedf2f3f23053a598685850318ecb363cf07cf48410d3ed8e9
sha512:  03c63d38b0286e9a6230ffd39a1470419887ea775823d21dc03a2f2b2762a24b496847724296b45e81a5ff607c
```

rahash is designed to work with blocks like radare does. So this way you can generate multiple checksums from a single file, and then make a faster comparision of the blocks to find the part of the file that has changed.

This is useful in forensic tasks, when progressively analyzing memory dumps to find the places where it has changed and then use 'radiff' to get a closer look to these changes.

This is the default work way for rahash. So lets generate a rahash checksumming file and then use it to check if something has changed. The default block size is 32 KBytes. You can change it by using the -b flag.

```
# generate ls.rahash
$ rahash -g -a sha1 /bin/ls ls.rahash
91c9cc53e7c7204027218ba372e9e738
f5547b7cd016678bebc61b4b0ca3a442
5fd03cecbbad68314bc82f2f7db2f6aa

# show values stored in rahash file:
$ rahash -vo ls.hash
file_name  /bin/ls
offt_size  8
endian     0 (little)
version    1
block_size 32768
file_size  92376
fragments  3
file_name  /bin/ls
from       0
to         92376
length     92376
algorithm  md5
algo_size  16
0x00000000 91C9CC53E7C7204027218BA372E9E738
0x00008000 F5547B7CD016678BEBC61B4B0CA3A442
0x00010000 5FD03CECBBAD68314BC82F2F7DB2F6AA

# check if something has changed
$ rahash -c -a sha1 /bin/ls ls.rahash
```

You can also specify some limits when calculating checksummings, so, you can easily tell rahash to start hashing at a certain seek and finish after N bytes or just when reaching another offset.

```
  -S [offset]  seek initial offset to
  -E [offset]  end hashing at offset
  -L [length]  end hashing at length
```

f.example:

```
$ rahash -S 10 -L 20 /bin/ls
4b01adea1951a55cdf05f92cd4b2cf75
```

Chapter 18: Binary diffing

Radiff is the program used in radare to identify changes and delta offsets between two binary files. It implements different algorithms to find and show this information:

```
$ radiff -h
Usage: radiff [-c] [-bgeirp] [file-a] [file-b]
  -b   bytediff (faster but doesnt support displacements)
  -d   use gnu diff as backend (default)
  -e   use erg0ts bdiff (c++) as backend
  -p   use program diff (code analysis diff)
  -s   use rsc symdiff
  -S   use rsc symbytediff
  -r   output radare commands
```

18.1 Diffing at byte-level

The byte-level binary diffing is the fastest and simple one. It will only work on files with the same size and will no detect delta offsets. This is obviously not useful for all the cases, but it will help when analyzing big files or patched ones to be able to identify and extract the patched bytes.

```
$ radiff -b a.bin b.bin
```

18.2 Delta diffing

A better fine grained approach for the binary diffing should be able to detect binary changes with delta offsets.

This kind of diffing is the same algorithm used by the GNU diff utility applied on text based files. The fast hacky approach in radare is done by exporting two binaries files as one hexpair per line ascii files, and then use the GNU diff with these two files to get the delta calculations.

This is the default bindiffing engine use by radiff. You can force the use of this with '-d'.

```
radiff -d /bin/true /bin/false
```

The other approach is done in C++ with better performance and can be used by appending the '-e' flag.

```
radiff -e /bin/true /bin/false
```

18.3 Diffing code graphs

Each code graph generated by the code analysis engine of radare can be stored in memory or disk using the 'R' command of radare.

```
[0xB7F73810]> R?
Usage: R[?] [argument] (TODO)
  R              ; list all RDBs loded in memory
  R [rdb-file]   ; load rdb file into memory
  R -[idx]       ; removes an rdb indexed
```

```
  Rm [range]     ; performs a merge between selected rdbs
  RG [num]       ; graph RDB number 'num'
  Rd [a] [b]     ; rdb diff. should generate a new rdb
```

In the same way it is possible to look for the differences between two different graph analysis of code by using the 'rdb diff' functionality of 'radiff -p' from the shell against two files generated by radare representing the graph structures or internally inside radare with the 'Rd' command.

This command will show the differences between these two graphs like new basic blocks, new edges, differences at byte level of the basic blocks to identify modified branches or so.

To generate an rdb file you just have to save the project using the 'Ps' command. This command will store the project file in ~/.radare/rdb/<project-file>. Take it from there to diff the code analysis with radiff.

TODO: To export an RDB file from IDA use the 'ida2rdb.idc' script that lives in the documentation directory of radare.

18.4 Binary patch

The '-r' flag of `radiff` will make it dump radare commands instead of human readable information. These commands if applied will make the first binary be the same as the second one. For example, have a look on this unit test:

```
$ cat tests/chk/radiff-test.sh
#!/bin/sh
printf "Checking radiff -rd... "
cp /bin/true .
radiff -rd true /bin/false | radare -vnw true
RET=`radiff -d true /bin/false`
if [ -n "${RET}" ]; then
        echo "Failed"
else
        echo "Success"
fi
rm -f true
```

Piping the output of radiff -r into radare opening in read-write mode the original file will modify it to make the contents be the same as the second one.

Chapter 19: Debugger

The debugger in radare is implemented as an IO plugin. It handles two different URIs for creating or attaching to a process: dbg:// and pid://.

There are different backends for multiple architectures and operating systems like GNU/Linux, Windows, MacOSX, (Net,Free,Open)BSD and Solaris.

The process memory is interpreted by radare as a plain file. So all the mapped pages like the program and the libraries can be readed and interpreted as code, structures, etc..

The rest of the communication between radare and the debugger layer is the wrapped system() call that receives a string as argument and executes the given command. The result of the operation is buffered in the output console and this contents can be handled by a scripting language.

This is the reason why radare can handle single and double exclamation marks for calling system().

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

The double exclamation mark tells radare to skip the plugin list to find an IO plugin handling this command to launch it directly to the shell. A single one will walk thru the io plugin list.

The debugger commands are mostly portable between architectures and operating systems. But radare tries to implement them on all the artchitectures and OSs injecting shellcodes, or handling exceptions in a special way. For example in mips there's no stepping feature by hardware, so radare has an own implementation using a mix of code analysis and software breakpoints to bypass this limitation.

To get the basic help of the debugger you can just type '!help':

```
[0x4A13B8C0]> !help
 Information
  info               show debugger and process status
  msg                show last debugger status message
  pid [tid] [action] show pid of the debug process, current tid and childs, or set tid.
  status             show the contents of /proc/pid/status
  signal             show signals handler
  maps[*]            flags the current memory maps (.!rsc maps)
  syms               flags all syms of the debugged program (TODO: get syms from libs too)
  fd[?][-#] [arg]    file descriptors (fd? for help)
  th[?]              threads control command
 Stack analysis
  bt                 backtrace stack frames (use :!bt for user code only)
  st                 analyze stack trace (experimental)
 Memory allocation
  alloc [N]          allocates N bytes (no args = list regions)
  mmap [F] [off]     mmaps a file into a memory region
  free               free allocated memory regions
  imap               map input stream to a memory region (DEPRECATED)
 Loader
```

```
  run [args]          load and start execution
  (un)load            load or unload a program to debug
  kill [-S] [pid]     sends a signal to a process
  {a,de}ttach [pid]   attach or detach target pid
 Flow Control
  jmp [addr]          set program counter
  call [addr]         enters a subroutine
  ret                 emulates a return from subroutine
  skip [N]            skip (N=1) instruction(s)
  step{o,u,bp,ret}    step, step over, step until user code, step until ret
  cont{u,uh,sc,fork}  continue until user code, here, syscall or fork
 Tracing
  trace [N]           trace until bp or eof at N debug level
  tt [size]           touch trace using a swapable bps area
  wtrace              watchpoint trace (see !wp command)
  wp[m|?] [expr]      put a watchpoint (!wp? for help) (wpm for monitor)
 Breakpoints
  bp [addr]           put a breakpoint at addr (no arg = list)
  mp [rwx] [a] [s]    change memory protections at [a]ddress with [s]ize
  ie [-][event]       ignore debug events
 Registers
  [o|d|fp]regs[*]     show registers (o=old, d=diff, fp=fpu, *=radare)
  reg[s|*] [reg[=v]   show get and set registers
  oregs[*]            show old registers information (* for radare)
  dr[rwx-]            DR registers control (dr? for help) (x86 only)
 Other
  dump/restore [N]    dump/restore pages (and registers) to/from disk
  dall                dump from current seek to cfg.limit all available bytes (no !maps required)
  core                dump core of the process
  hack [N]            Make a hack.
  inject [bin]        inject code inside child process (UNSTABLE)
Usage: !<cmd>[?] <args> @ <offset>     ; see eval dbg. fmi
```

19.1 Registers

The registers are part of the user area stored in the context structure used by the scheduler. This structure can be manipulated to get and set values for those registers, and on intel for example, is possible to directly manipulate the DRx hardware registers to setup hardware breakpoints and watchpoints.

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

```
[0x4A13B8C0]> !regs
  eax  0x00000000    esi  0x00000000    eip    0x4a13b8c0
  ebx  0x00000000    edi  0x00000000    oeax   0x0000000b
  ecx  0x00000000    esp  0xbfac9bd0    eflags 0x200286
  edx  0x00000000    ebp  0x00000000    cPazStIdor0 (PSI)
```

The !reg command can be used in different ways:

```
[0x4A13B8C0]> !reg eip         ; get value of 'eip'
0x4a13b8c0

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

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

```
[0x4A13B8C0]> !reg*    ; Appending '*' will show radare commands
fs regs
f oeax @ 0xb
f eax @ 0x0
f ebx @ 0x0
```

```
f ecx @ 0x0
f edx @ 0x0
f ebp @ 0x0
f esi @ 0x0
f edi @ 0x0
f oeip @ 0x0
f eip @ 0x4a13b8c0
f oesp @ 0x0
f esp @ 0xbfac9bd0

[0x4A13B8C0]> .!reg*  ; '.!' interprets the output of this command
```

Note that the 'oeax' stores the original eax value before executing a syscall. This is used by '!contsc' to identify the call, similar to 'strace'.

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

```
[0x4A13B8C0]> !oregs
Mon, 01 Sep 2008 00:22:32 GMT
  eax  0x00000000    esi  0x00000000    eip    0x4a13b8c0
  ebx  0x00000000    edi  0x00000000    oeax   0x0000000b
  ecx  0x00000000    esp  0xbfac9bd0    eflags 0x200386
  edx  0x00000000    ebp  0x00000000    cPazSTIdor0 (PSTI)
[0x4A13B8C0]> !regs
  eax  0xbfac9bd0    esi  0x00000000    eip    0x4a13b8c2
  ebx  0x00000000    edi  0x00000000    oeax   0xffffffff
  ecx  0x00000000    esp  0xbfac9bd0    eflags 0x200386
  edx  0x00000000    ebp  0x00000000    cPazSTIdor0 (PSTI)
```

The values in eax, oeax and eip has changed. And this is noted when enabling scr.color.

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

```
; save registers
[0x4A13B8C0]> !reg* > regs.saved

; restore
[0x4A13B8C0]> . regs.saved
```

In the same way the eflags can be altered by the characters given in this way. Setting to '1' the selected flags:

```
[0x4A13B8C0]> !reg eflags = pst
[0x4A13B8C0]> !reg eflags = azsti
```

You can easily get a string representing the last changes in the registers using the 'dregs' command (diff registers):

```
[0x4A13B8C0]> !dregs
eax = 0xbfac9bd0 (0x00000000)
```

eip register is ignored and oeax is not handled for obvious reasons :)

There's an eval variable called 'trace.cmtregs' that adds a comment after each executed instruction giving information about the changes in the registers. Here's an example

```
[0x4A13C000]> !step 5
[0x4A13C000]> pd 5
   ; 10 esp = 0xbfac9bc8 (0xbfac9bd0)
   0x4A13C000,           55              push ebp
   ; 12 ebp = 0xbfac9bc8 (0x00000000) esp = 0xbfac9bc8 (0xbfac9bcc)
```

```
   0x4A13C001            89e5            mov ebp, esp
   ; 14 ebp = 0xbfac9bc8 (0x00000000) esp = 0xbfac9bc4 (0xbfac9bc8)
   0x4A13C003            57              push edi
   ; 16 esp = 0xbfac9bc0 (0xbfac9bc8)
   0x4A13C004,           56              push esi
   ; 18 esp = 0xbfac9bbc (0xbfac9bc4)
   0x4A13C005      oeip: 53              push ebx
[0x4A13C000]>
```

You can align these comments with 'eval asm.cmtmargin'.

The extended or floating point registers are accessed with the '!fpregs' command. (See floating point registers for more information)

19.1.1 Hardware registers

The hardware registers are only supported on few architectures. In this case only 'intel' 32 and 64 bits. also called 'DRx' on intel.

Radare allows you to manipulate them manually to setup 4 different breakpoints when reading, writing or executing (rwx):

```
[0x4A13C000]> !dr
DR0 0x00000000 x
DR1 0x00000000 x
DR2 0x00000000 x
DR3 0x00000000 x
[0x4A13C000]> !dr?
Usage: !dr[type] [args]
  dr                    - show DR registers
  dr-                   - reset DR registers
  drr [addr]            - set a read watchpoint
  drw [addr]            - set a write watchpoint
  drx [addr]            - set an execution watchpoint
  dr[0-3][rwx] [addr]   - set a rwx wp at a certain DR reg
Use addr=0 to undefine a DR watchpoint
```

For example. To setup a read exception at address 0xBF894404:

```
[0x80480303]> !dr0r 0xBF894404
```

The DR index can be ignored, because it can be automatically handled by radare in the same way:

```
[0x80480303]> !dr 0xBF894404
```

To remove all the values of the hw regs just type:

```
[0x80480303]> !dr-
```

19.1.2 Floating point registers

The floating point and extended registers are accessed thru the '!fpregs' command. This command depends on the cpu and the operating system, so floating point registers are usually stored in a quite weird way at kernel level.

Here's an example on intel with MMX and STX registers:

```
[0xB7F9E810]> !fpregs
 cwd = 0x037f  ; control    swd = 0x0000  ; status
 twd = 0x0000  ; tag        fip = 0x0000  ; eip of fpu opcode
 fcs = 0x0000              foo = 0x0000  ; stack
 fos = 0x0000
```

```
 mm0 = 0000 0000 0000 0000     st0 = 0 (0x00000000)
 mm1 = 0000 0000 0000 0000     st1 = 0 (0x00000000)
 mm2 = 0000 0000 0000 0000     st2 = 0 (0x00000000)
 mm3 = 0000 0000 0000 0000     st3 = 0 (0x00000000)
 mm4 = 0000 0000 0000 0000     st4 = 0 (0x00000000)
 mm5 = 0000 0000 0000 0000     st5 = 0 (0x00000000)
 mm6 = 0000 0000 0000 0000     st6 = 0 (0x00000000)
 mm7 = 0000 0000 0000 0000     st7 = 0 (0x00000000)
```

And the same for mips:

```
[0x2AAA8820]> !fpregs
f00: 0xffffffffffffffff f02: 0xffffffffffffffff
f04: 0xffffffffffffffff f06: 0xffffffffffffffff
f08: 0xffffffffffffffff f10: 0xffffffffffffffff
f12: 0xffffffffffffffff f14: 0xffffffffffffffff
f16: 0xffffffffffffffff f18: 0xffffffffffffffff
f20: 0xffffffffffffffff f22: 0xffffffffffffffff
f24: 0xffffffffffffffff f26: 0xffffffffffffffff
f28: 0xffffffffffffffff f30: 0xffffffffffffffff
```

(On mips the default value for undefined registers is '-1' and not '0'.

You can enable them in the visual debugger view with the 'e dbg.fpregs=true' command. The '!fpregs*' command is also available to export the registers as flags.

19.2 Memory

Apart from the basic core operations done by radare on memory. The debugger allows you to perform some extended actions on it like changing the map protections, fruteforce the memory space to extract section information, maps ranges, etc.

In this section I will explain how these commands works and how we can for example setup a on-read-breakpoint using the `!mp` command.

19.2.1 Memory protections

Another way to setup read/write exceptions is done by changing the protection properties of the memory pages of a program.

These properties must have an aligned size like 4096 bytes or so. This depends on the operating system and the cpu, but it is supported by all the architectures with MMU.

```
[0x4A13C000]> !mp?
Usage: !mp [rwx] [addr] [size]
  > !mp       - lists all memory protection changes
  > !mp --- 0x8048100 4096
  > !mp rwx 0x8048100 4096
- addr and size are aligned to memory (-=%4).
```

Using '!maps' you can get a list of all the mapped regions of the program, but this ones will not refer to the changes done by this command, because they are handled submappings and not all OS allows you a fine-grained control over this.

For this reason, radare handles a list of changes done by this command to allow you to reset the changes done.

Here's an example... you are interested on getting the point where a program tries to write in a buffer at a certain address. The way to catch this is by changing the properties of this page, dropping the write permission and kepping the read one.

```
[0x4A13C000]> !mp r-- 0x8054180 4096
[0x4A13C000]> !cont

.. write exception ..

[0x4A13C000]> !mp rw- 0x8054180 4096 ; restore change
```

19.2.2 Memory pages

You can control different operations over the memory pages of the target process. This is an important task that should be handled by the debugger layer to get information to get the ranges of memory mapped.

The '!maps' command will list all the regions mapped in the target process. For example:

```
[0x4A13B8C0]> !maps
0xbfe15000 - 0xbfe2a000 rw-- 0x00015000 [stack]
0xb7f87000 - 0xb7f88000 r-x- 0x00001000 [vdso]
0x4a155000 - 0x4a157000 rw-- 0x00002000 /lib/ld-2.5.so
0x4a13b000 * 0x4a155000 r-x- 0x0001a000 /lib/ld-2.5.so
0x0805c000 - 0x0805d000 rw-u 0x00001000 /bin/ls
0x08048000 - 0x0805c000 r-xu 0x00014000 /bin/ls
```

The columns are start address, end address, permissions, size and name of region. At the same time all the proper flags are registered in the core named as 'section_foo' and 'section_foo_end'.

This way it is possible to iterate in scripts from these ranges easily. The '*' between the from-to addresses allows you to easily view where you are located

```
-- analyze a section
function opcleaner_section(name)
  print("FROM: "..r.get("section_"..name))
  from = r.get("section_"..name)
  to   = r.get("section_"..name.."_end")
  old_opcode = ''

  print (string.format("Segment "..name.." at 0x%x",from))

  --- ... do the job here ...
end
```

So we can now work on a single segment just giving the section name:

```
opcleaner_section ("_text")
```

We can locate the current seek in the maps by typing '!maps?':

```
[0x4A13B8C0]> !maps?
0x4a13b000 * 0x4a155000 r-x- 0x0001a000 /lib/ld-2.5.so
```

19.2.3 Dumping memory

There are different commands to dump these sections to disk. The '!dump' and '!restore' are used to create a directory called 'dump#' where '#' is a number starting from 0 and is incremented while called multiple times. So you can use '!dump' and '!restore' to go 'forward' and 'backward' of the process status. Because it is also dumping and restoring the register values.

```
[0x4A13B8C0]> !dump
Dump directory: dump0
Dumping BFE15000-BFE2A000.dump  ; 0x00015000  [stack]
Dumping 4A155000-4A157000.dump  ; 0x00002000  /lib/ld-2.5.so
```

```
Dumping 0805C000-0805D000.dump  ; 0x00001000  /bin/ls
Dumping 08048000-0805C000.dump  ; 0x00014000  /bin/ls
Dumping CPU to cpustate.dump...
```

You can also specify the directory name as argument:

```
[0x4A13B8C0]> !dump foo
...
[0x4A13B8C0]> !restore foo
```

The '!dall' command is similar to the previous one, but it is based on the concept that there are no maps sections information. This is useful on some unixes like some BSDs that they have no /proc to get this information. So it reads from 0 to 0xFFFFFFFF looking for readables pages and dumping them to files named 'from-to.bin' in the current directory.

19.2.4 Managing memory

It is also possible to allocate and free new memory regions on the child process at specified or decided by the system. In the following example we are allocating 10 MB in the child process.

```
[0x000000C0]> !alloc 10M
0xb7587000
[0x000000C0]> s 0xb7587000
[0xB7587000]> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0xB7587000, 0000 0000 0000 0000 0000 0000 0000 ..............
0xB758700E  0000 0000 0000 0000 0000 0000 0000 ..............
```

We can now write the contents of a file here:

```
[0xb7587000]> wf program.bin
```

But there's another option for mapping files in memory: '!mmap':

XXX: this is not working

In the same way you can create a core file with '!core'. But this is currently system-specific.

19.3 Run control

The basic life-cycle of a process can be managed with different commands of the debugger.

```
!load         ; reload the process (restart program)
!run          ; start running
```

19.3.1 Stepping

```
!step         ; executes a single instruction
!stepo        ; step over call instructions
!stepret      ; step instructions until function return
!stepu [addr] ; step until address
!stepbp       ; step using breakpoints and code analysis
```

Is interesting to have multiple ways to perform stepping and continuation to be able to bypass different protections like avoid changing the memory adding software breakpoints, just stepping until an address.

You can also step N times in two ways:

```
[0x80484040]> !step 4
[0x80484040]> 4!step
```

19.3.2 Continuations

About continuations radare offers multiple commands for handling process continuations:

```
!cont         ; continue execution
!contu [addr] ; continue until address using a breakpoint
!contuh       ; continue until here
!contsc       ; continue until syscall (strace)
!contfork     ; continue until new child is created
```

The 'contuh' command is quite useful when analyzing loops, because allows you to easily complete a loop without having to manually setup breakpoints.

19.3.3 Skipping opcodes

```
!skip [n]
```

19.3.4 Threads and processes

The list of child processes of the target process can be listed with '!pid' command.

```
$ radare -d /bin/sh

[0x13B8C0]> !run
To cleanly stop the execution, type: "^Z kill -STOP 1527 && fg"

sh-3.2$ ls
pid: 1611. new process created!
 - Use !pid for processes, and !th for threads
CLONE HAS BEEN INVOKED
debug_dispatch_wait: RET = 0 WS(event)=6 INT3_EVENT=2 INT_EVENT=3 CLONE_EVENT=6
=== cont: tid: 1611 event: 6, signal: 19 (SIGSTOP). stop at 0x4ab95eb3

[0x4A13B8C0]> !pid
 pid : 1527 0x4ab95eb3 (stopped)
 pid : 1611 0x4ab95eb3 (stopped)
 `- 1611 : /bin/sh (stopped)
```

The /bin/sh while calling 'ls'. This event has been catched by radare and prompts you again. Now we can choose which process we want to follow. The parent or the child.

```
[0x4A13B8C0]> !pid 1527
Current selected thread id (pid): 1527
```

The same operations can be done by using the '!th' command for the threads.

In the same way you can hackily 'attach' and 'detach' from to a pid.

19.3.5 Touch Tracing

The 'touch trace' is a special tracing engine that was born from an idea of Gadix (Thanks! ;D)

The main idea of '!tt' (which is the assigned command name for this feature) is to fill N bytes of the process memory with software breakpoints while the debugger keeps a copy of the original bytes.

When a breakpoint not swapped by the debugger is catched between this memory range the debugger swaps the original bytes into the process memory and continues the execution. When the program counter stops outside this range, the program memory of the traced program is restored and the debugger keeps the tracing information accessible with the 'at' command explained in

another chapter.

In this way it is possible to create a fast trace, so each instruction will only be tracked one time. So you will be able to generate multiple traces of different parts of the program without a high cpu load and allowing you to easily identify the executed regions of a program with a decent user interaction.

To use this command just give an argument with the size of the tracing area to be swapped and the program will start running

```
[0xB7F75810]> !tt 10K
[0xB7F75810]> at
0xb7f75810 - 0xb7f7585a
0xb7f75860 - 0xb7f75876
0xb7f75a60 - 0xb7f75b8b
0xb7f75b95 - 0xb7f75b9f
...
```

19.4 Breakpoints

The breakpoints in radare are managed by the '!bp' command which is able to handle software and hardware breakpoints for different architectures. On intel there's a specific command called '!dr' to manage the DRX registers and be able to define 4 hardware breakpoints manually for read, write or exec.

The '!bp' command will use software or hardware breakpoints (if supported) depending on the boolean eval value 'dbg.hwbp'.

19.4.1 Software breakpoints

The software breakpoints are the most used and common to be used because of their portability. They are based on invalid, undefined or trap instructions depending on the architecture.

This way the debugger can catch the event and identify if this address is handled by a breakpoint and restore the memory and program counter to make the program flow continue properly.

```
[0xB7F08810]> !bp?
Usage: !bp[?|s|h|*] ([addr|-addr])
  !bp [addr]    add a breakpoint
  !bp -[addr]   remove a breakpoint
  !bp*          remove all breakpoints
  !bps          software breakpoint
  !bph          hardware breakpoint
```

To add a software breakpoint type:

```
[0xB7F75810]> !bp sym_main       ; set breakpoint at flag 'sym_main'
[0xB7F75810]> !cont              ; continue execution
[0x08048910]> !bp -sym_main      ; remove breakpoint
```

You can directly specify which kind of breakpoint you want to use, just using the same syntax for "!bp", but using "!bps" or "!bph" to not let radare decide for you which is the better option. (or just define dbg.hwbp to true or false)

19.4.2 Memory breakpoints

You can setup read/write/exec breakpoints on memory pages by changing their access permissions with the '!mp' command. See 'Memory Protections' section for more information.

19.4.3 Hardware breakpoints

The hardware breakpoints are architecture-dependant and provide a limited but powerful way to make the debugger stop when reading, writing or executing at a certain memory address. On intel there's the '!dr' command to manually modify the DRX registers.

```
[0xB7F08810]> !dr?
Usage: !dr[type] [args]
  dr                    - show DR registers
  dr-                   - reset DR registers
  drr [addr]            - set a read watchpoint
  drw [addr]            - set a write watchpoint
  drx [addr]            - set an execution watchpoint
  dr[0-3][rwx] [addr]   - set a rwx wp at a certain DR reg
Use addr=0 to undefine a DR watchpoint
```

Lets define some hardware breakpoints on intel:

```
[0xB7F08810]> !dr0r 0x8048000 ; DR0 = read breakpoint
[0xB7F08810]> !dr1x 0x8049200 ; DR1 = exec breakpoint
[0xB7F08810]> !dr      ; list DRX reg values
DR0 0x08048000 r
DR1 0x08049200 x
DR2 0x00000000 x
DR3 0x00000000 x

[0xB7F08810]> !dr-     ; reset DRX
[0xB7F08810]> !dr
DR0 0x00000000 x
DR1 0x00000000 x
DR2 0x00000000 x
DR3 0x00000000 x
```

TODO !bph

19.4.4 Watchpoints

Watchpoints are evaluated expressions that makes the '!contwp' command stop stepping when they match. You can setup multiple expressions and logical operations and groupping checking for memory or register values.

```
e cmd.wp = pd 3 @ eip
e dbg.wptrace = false
```

The 'dbg.wptrace' is used to make the watchpoints stop or not the execution when a watchpoint expression matches. The 'cmd.wp' will be executed every time a watchpoint is matched.

```
[0xB7F45A60]> !wp %eip = 0xB7F45A8B
0: %eip = 0xB7F45A8B
[0xB7F45A60]> !contwp
watchpoint 0 matches at 0xb7f45a8b
[0xB7F45A60]> pd 1 @ 0xB7F45A8B
               0xB7F45A8B        eip: 01d0              eax += edx
[0xB7F45A60]> !wp %eax = 0x6fffffff
1: %eax = 0x6fffffff
[0xB7F45A60]> !contwp
watchpoint 1 matches at 0xb7f45abe
[0xB7F45A60]> pd 1 @ 0xb7f45abe
          |    0xB7F45ABE        eip: 29d0              eax -= edx
[0xB7F45A60]> !reg eax
0x6fffffff
[0xB7F45A60]>
```

19.5 Filedescriptors

The file descriptors of a process can be manipulated in multiple ways to get information about the opened files of a program, in which permissions, duplicate them as new file descriptors, close, change the seek, open a file on an already opened filedescriptor or making a TCP connection for example.

All this stuff is system-dependant, so if it is not implemented in your favourite operating system(R) i'll be happy to receive feedback and patches.

It is handled by the '!fd' command:

```
[0x4A13C00E]> !fd?
Usage: !fd[s|d] [-#] [file | host:port]
  !fd                   ; list filedescriptors
  !fdd 2 7              ; dup2(2, 7)
  !fds 3 0x840          ; seek filedescriptor
  !fd -1                ; close stdout
  !fd /etc/motd         ; open file at fd=3
  !fd 127.0.0.1:9999    ; open socket at fd=5
```

Here's the filedescriptor list once a process is created with stdin, stdout and sterr

[0x4A13C00E]> !fd 0 0x00000013 rwC /dev/pts/0 1 0x00000000 rwC /dev/pts/0 2 0x00000000 rwC /dev/pts/0

Now we can close the stdout of the program or just redirect them to a file opening a file on the fd number '1':

```
[0x4A13C00E]> !fd /tmp/stdout.txt  ; open new fd
[0x4A13C00E]> !fd -1               ; close stdout
[0x4A13C00E]> !fdd 3 1             ; stdout = new_fd
[0x4A13C00E]> !fd -3               ; close file
```

19.6 Events

You can even configure radare to handle or ignore multiple types of events. This is useful to avoid stopping when an event is received from the child process.

```
[0xB7FA8810]> !ie?
Usage: !ie [-]<event>
 - enables or disables the 'ignore event'
[0xB7FA8810]> !ie
 0 stop
 0 trap
 0 pipe
 0 alarm
 0 fpe
 0 ill
```

This is usually used to bypass some packer protections that relay on some events. So this way you can skip them just with '!ie stop' or '!ie -stop'.

19.6.1 Event handling

It is possible to configure which events should be ignored by the debugger

```
[0x4A13B8C0]> !ie
 0 stop
 0 trap
```

```
 0 pipe
 0 alarm
 0 fpe
 0 ill

[0x4A13B8C0]> !ie stop
[0x4A13B8C0]> !ie | grep stop
 1 stop
```

The 'ie' command refers to 'ignore events'

19.6.2 Signal handling

You can change visualize and change the signal handlers of the target process:

```
[0x4A13B8C0]> !signal
SIGHUP     (DEFAULT)
SIGINT     (IGNORE)
SIGQUIT    (DEFAULT)
SIGILL     (DEFAULT)
SIGTRAP    (DEFAULT)
SIGABRT    (DEFAULT)
SIGFPE     (DEFAULT)
SIGKILL    (DEFAULT)
SIGBUS     (DEFAULT)
SIGSEGV    (DEFAULT)
SIGSYS     (DEFAULT)
SIGPIPE    (DEFAULT)
SIGALRM    (IGNORE)
SIGTERM    (DEFAULT)
SIGURG     (DEFAULT)
SIGSTOP    (DEFAULT)
SIGTSTP    (DEFAULT)
SIGCONT    (DEFAULT)
SIGCHLD    (DEFAULT)
SIGTTIN    (DEFAULT)
SIGTTOU    (DEFAULT)
SIGIO      (DEFAULT)

[0xB7FA8810]> !sig SIGIO 0x8049350
Signal 29 handled by 0x08049350
[0xB7FA8810]> !sig SIGIO
SIGIO      (0x8049350)
[0xB7FA8810]> !sig SIGIO 0
(DEFAULT)
[0xB7FA8810]> !sig SIGIO
SIGIO      (DEFAULT)
[0xB7FA8810]>
```

Chapter 20: Random stuff

20.1 Debugging brainfuck

The 'bfdbg' IO plugin offers a debugging interface for a brainfuck virtual machine implemented inside the same plugin. Rabin can magically autodetect brainfuck files, so the 'e asm.arch=bf' will be defined if you use 'e file.id=true' in your ~/.radarerc.

The brainfuck disassembler decodes the bf instructions as complex instructions supporting repetitions. The translation would be:

```
+++++   ->   add [ptr], 5
```

Also loops are automatically detected between the '[' ']' opcodes, so the code analysis will show nice jump lines there:

```
[0x00000000]> pd
     0x00000000       3e              > inc [ptr]
     0x00000001       2b2b2b2b2b2b2b. + add [ptr], 9
.--> 0x0000000A  eip: 5b              [ loop {
|    0x0000000B       3c              < dec [ptr]
|    0x0000000C,      2b2b2b2b2b2b2b. + add [ptr], 8
|    0x00000014,      3e              > inc [ptr]
|    0x00000015       2d              - dec [ptr]
`==< 0x00000016       5d              ] }  ; 1 = 0x0000000a
     ...
```

The BF virtual machine supports user input and screen output emulating two virtual buffers located at 0x10000 for the 'input' section and 0x50000 for the 'screen'. Here's a 'hello.bf' example:

```
$ radare bfdbg://hello.bf
File type: Brainfuck
[0x00000000]> !!cat hello.bf
>+++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.+++++++..+++.[-]
>++++++++[<++++>-] <.>++++++++++[<++++++++>-]<-.--------.+++
.------.--------.[-]>++++++++[<++++>- ]<+.[-]++++++++++.

[0x00000000]> pz @ screen

[0x00000000]> !cont
Trap instruction at 0x000000b5

[0x00000000]> pz @ screen
Hello world!
```

You can query the section information using the !maps command:

```
[0x00000000]> !maps
0x00000000 - 0xffffffff rwxu 0xffffffff .code
0x000d0000 - 0x000dffff rw-- 0x0000ffff .data
0x00050000 - 0x00051000 rw-- 0x00001000 .screen
0x00010000 - 0x00011000 rw-- 0x00001000 .input
```

For getting/setting registers use the '!reg' command:

```
[0x00000000]> !reg
  eip  0x000000b5     esp  0x000d0000
  ptr  0x00000000     [ptr]  10 = 0x0a ' '

[0x00000000]> !reg eip 0
[0x00000000]> !reg
  eip  0x00000000     esp  0x000d0000
  ptr  0x00000000     [ptr]  10 = 0x0a ' '
```

The help for all the debugger commands is:

```
[0x00000000]> !help
Brainfuck debugger help:
20!step       ; perform 20 steps
!step         ; perform a step
!stepo        ; step over rep instructions
!maps         ; show registers
!reg          ; show registers
!cont [addr]  ; continue until address or ^C
!trace [addr] ; trace code execution
!contsc       ; continue until write or read syscall
!reg eip 3    ; force program counter
.!reg*        ; adquire register information into core
```

So.. as a fast overview, see that you can step, or step over all repeated instructions, continue the execution until an address, trace the executed opcodes with data information (bytes peeked and poked) or trace until syscall.

Obviously all these commands can be used from the visual mode..so press 'V<enter>' in the radare prompt and use these keys:

```
F6 : continue until syscall
F7 : step instruction
F8 : step over repeated instructions
F9 : continue until trap or user ^C
```

Enjoy!

20.2 Analyze serial protocols

Since radare 1.0 it ships a serial:// IO plugin to connect to a serial port device and work with it like if it was a socket:// one.

Both plugins (socket and serial) are quite similar and based on the concept of an always-growing virtual file containing the received bytes. The plugin sets flags for each read operation from the device.

Let's see an example of how to use it to analyze the protocol of the Symbian's TRK debugger.

1) Install TRK symbian agent into your phone:

```
http://tools.ext.nokia.com/agents/index.htm
```

2) Prepare your laptop

```
$ rfcomm listen hci0 1
```

3) Make your phone connect via bluetooth

```
f.ex: Options -> Connect -> BlueZ(0)
```

4) Attach radare to the newly rfcomm created

```
 $ sudo radare serial:///dev/rfcomm0:9600
```

Once here we can start writing raw commands which are headed and footed by '7E'

```
> wx 7E 00 00 FF 7E
> wx 7E 01 01 FD 7E
> wx 7E 05 02 F8 7E

> x
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF01234567
0x00000000, 7eff 0007 f97e 7220 5379 6d62 6961 6e20 ~....~r Symbian OS start
0x00000018, 6564 c7bf 0463 c24a ffff ffff ffff ffff ed...c.J................
```

20.3 Debugging with bochs and python

radare can be easily adapted to be a frontend for other debuggers. Ero Carrera has written a python interface for instrumentating bochs. Having python embedded on a debugger is just simple to use radare as a frontend to launch scripts.

So, understand this bochs implementation as simple as for the immunity debugger or vtrace. for example :)

The magic resides in the 'radapy.py' python module included in the scripts/ directory of the sources. It should be installed in the /usr/lib/python2.5 or similar directory, so you can type from any python console "import radapy" to test it.

```
$ python
>>> import radapy
```

This module implements a python version of the network protocol of radare to be used together with connect://.

There's also another python module called 'radapy_bochs' which implements the interface of radapy to interact between ero's python-bochs instrumentation api and a remote radare. The port can be configured if we just call 'radapy.listen_tcp(PORT)' from the bochs-python console.

To open the debugger interface in radare we just have to append 'dbg://' to the end of the 'connect://' uri.

20.3.1 Demo

We will need a bochs compiled with the Ero patches to provide the python instrumentation with the CVS version.

```
$ cvs -d:pserver:anonymous@bochs.cvs.sourceforge.net:/cvsroot/bochs login
$ cvs -z3 -d:pserver:anonymous@bochs.cvs.sourceforge.net:/cvsroot/bochs co -P bochs
 ...
$ cd bochs
$ patch -p1 < bochs-python-ero.patch
$ ./configure --enable-debugger --enable-instrumentation=python_hooks --enable-all-optimizations \
 --enable-show-ips --enable-global-pages --enable-fpu --enable-pci --enable-cpu-level=6 \
 --enable-vbe --enable-repeat-speedups --enable-guest2host-tlb --enable-ignore-bad-msr \
 --enable-pae --enable-mtrr --enable-trace-cache --enable-icache --enable-fast-function-calls \
 --enable-host-specific-asms --enable-mmx --enable-sse=4 --enable-sse-extension --enable-sep \
 --enable-x86-debugger
...
$ make
...
```

```
$ sudo make install
...
```

Now we have to open a bochs session and initialize the radapy-bochs interface.

```
$ bochs
00000000000i[APIC?] local apic in  initializing
========================================================================
                    Bochs x86 Emulator 2.3.7.cvs
            Build from CVS snapshot, after release 2.3.7
========================================================================
00000000000i[     ] reading configuration from .bochsrc
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6]
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
bochs-python: environment variable "BOCHS_PYTHON_INIT" not set

>>> import radapy_bochs
Listening at port 9998
```

Now is time to launch our radare against this port!

```
$ radare connect://127.0.0.1:9998/dbg://bochs
Connected to 127.0.0.1:9998
```

We can just enter in visual mode to make the debugging more pleasant, and obviously we need to disassemble in 16 bit. EIP is calculated from CS_base+EIP only for flags.

```
[0x000F05F7]> e asm.arch=intel16
[0x000F05F7]> V

Stack:
   offset   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  0123456789ABCDEF01
0x0000FFC6  d0ff 9c08 00f0 9401 0000 f0ff d50c 0400 00f0 ..................
0x0000FFD8, 9401 0000 0000 0000 0000 0000 9401 0000 f8ff ..................
0x0000FFEA  0000 0100 0073 faff e117 0400 f7d7 9401 0000 .....s............
0x0000FFFC, c1e0 0000 0000 0000 0000 0000                ............
Registers:
  eax  0x0000f000    esi  0x00000000    eip     0x000005f7
  ebx  0x0000fff8    edi  0x00000500    oeax    0x0000f000
  ecx  0x00000000    esp  0x0000ffc6    eflags  0x00000082
  edx  0x00000f00    ebp  0x0000ffc6    cr0     0x60000010
  dr0 0x00000000   dr1 0x00000000   dr2 0x00000000   dr3 0x00000000
```

```
Disassembly:
          0x000F05F7        eip: 53              push bx
          0x000F05F8,            1e              push ds
          0x000F05F9             8b4604          mov ax, [bp+0x4]
          0x000F05FC,            8ed8            mov ds, ax
          0x000F05FE             8b5e06          mov bx, [bp+0x6]
          0x000F0601             8a07            mov al, [bx]
          0x000F0603             1f              pop ds
          0x000F0604,            5b              pop bx
          0x000F0605             5d              pop bp
          0x000F0606             c3              ret
          0x000F0606           ; ------------------------------------
          0x000F0607             55              push bp
          0x000F0608,            89e5            mov bp, sp
          0x000F060A             53              push bx
```

The '!help' will show us the available commands

```
[0x000F05F7]> !help
Bochs-python remote debugger
 !?                      : alias for !help
 !reg [reg] ([value])    : get/set CPU registers
 !regs[*]                : show CPU registers
 !cregs                  : show control registers
 !fpregs                 : show FPU registers
 !st                     : print stack
 !bp [[-]addr]           : breakpoints
 !cont                   : continue execution
 !step [n]               : perforn N steps
 !stepo [n]              : step over
 !mem [physical|linear] : select memory addressing
 !exec [python-expr]     : execute python expression remotely
```

Have fun!

Chapter 21: EOF

This book is a work in progress documentation, so don't take all this information as static and unalterable :)

Readers are welcome to feed me with bug reports, ideas, concepts, patches or any other kind of collaboration.

I hope you enjoyed the reading and find it useful. Keep tuned for updates of this book at:

`http://radare.nopcode.org/get/radare.pdf`

21.1 Greetings

I would like to greet some drugs, drinks and people that shared his life with mine during the development of radare and this book.

- God. aka Flying Spaguetti Monster

- nopcode guys (for the cons and

- Sexy Pandas (let's pwn the plugs!)

- 48bits (keep up the good work)

- Gerardo (ideas and tips for the book)

- pof (for the crackme tutorial and usability tips)

- Esteve (search engine+some code graph stuff)

- Nibble (ELF32/64 and PE parser+some core work)

- revenge (OSX debugger+mach0 work)

- Lia (4teh luf :)

I hope you guys did enjoyed the book, the soft, the hard, the weed and the smoke.

Spread the dword and don't let the bytes grind you down!

Have fun!