Learning Radare In Practice

Toulouse Hacking Convention

By pancake

Before starting..

WhoAmI

What Am I Doing Here?

- Author and benevolent leader of r2
- Free Software developer
- Working at Nowsecure as a security researcher in the R+D team
- ~20 years doing low level stuff, wifi, bt, vx, n900 flasher, acr, valabind..

Why Radare2?

- It’s free and open-source
- Runs everywhere (Windows, Mac, Android, GNU/Linux, QNX, Haiku, iOS, *BSD, ...)
- Easy to script and extend with plugins
- Embeddable
- Grows fast
- Supports tons of file-formats
- Handles gazillions of architectures
- Easy to modify and extend
- Commandline friendly
- Great community and even better leader
- Collaborative
- It’s mine

Introduction

What Am I Doing Here?

- What is r2?
- How to use the shell
- Analyzing
- Debugging
- Patching
- Scripting

What is Radare2?

- **Reverse Engineering**
  - Analyze Code/Data/..
  - Understanding Programs
- **Low Level Debugging**
  - More close to olly than GDB
  - Multi-platform, and support for remote
- **Forensics**
  - File Systems
  - Memory Dumps
- **Assembler/Disassembler**
  - Several architectures
  - Multiplatform
- **And more!**

History

Radare was born in 2006 (hey this is 12 years!) as a forensic tool to perform manual and interactive carving to recover some deleted files from disk or ram.

It quickly grew adding support for disassembler, debugger, code analyzer, scripting, …

And then I decided to completely rewrite it to fix the maintenance and monolithic design problems.

We organized the RSoC after being rejected in our first try at GSoC, which it rules.

After 8 years coding mostly alone, the community grew a lot and I started switching from developer to maintainer/leader.

r2con

Starting in 2016, in sync with the 10th anniversary of radare2.

- First week(end) of September
- In Barcelona
- No sponsors
- 4 days
- 50e ticket
- Free trainings, talks, hackathons
- R2wars and crackmes competitions
- Friendly environment with chiptune and beers

https://radare.org/con

Tools

Radare2 is composed by some core libraries and a set of tools that use those libraries and plugins.

radare2	**r2pm**	**rarun2**	**ragg2**
rabin2	**radiff2**	**rax2**	**rahash2**
rasm2	**rafind2**	**r2agent**	**rasign2**

Tools

- Quick usage example for every tool:
  - rax2, rabin2, rasm2, ...
- Manpages, inline help

(DEMO)

Libraries

RIO abstracts input-output and layouts

RFS abstracts filesystem and partitions access

RBin parses the structure and detects parameters.

RAsm disassembles the code if any

RAnal analyze and emulate to identify functions and refs

RUtil provide common helper functions

REgg generate payloads ready to be injected

RDiff find differences between two sources

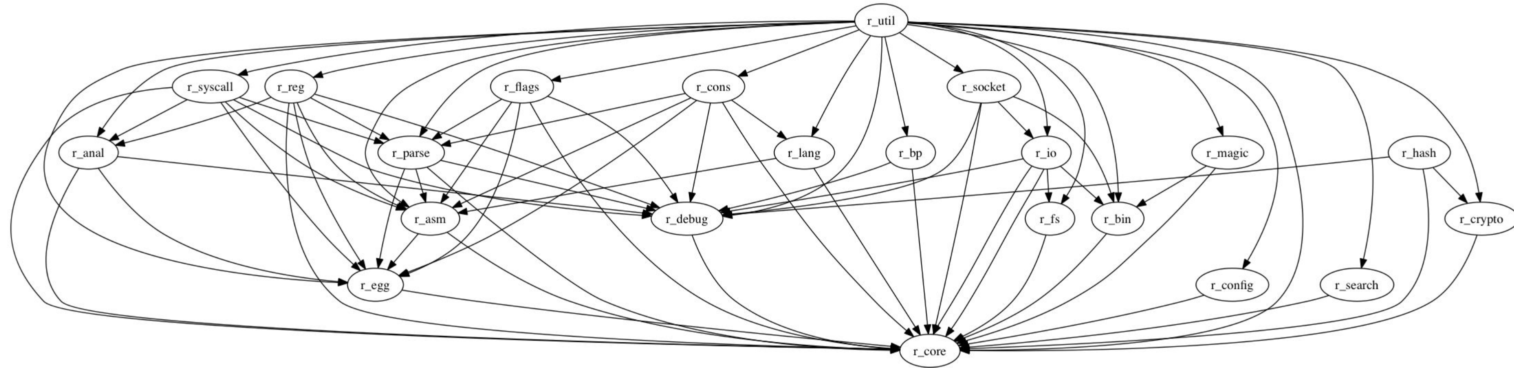
RSearch search patterns, magic headers, binmask, ..

RCore uses them all!

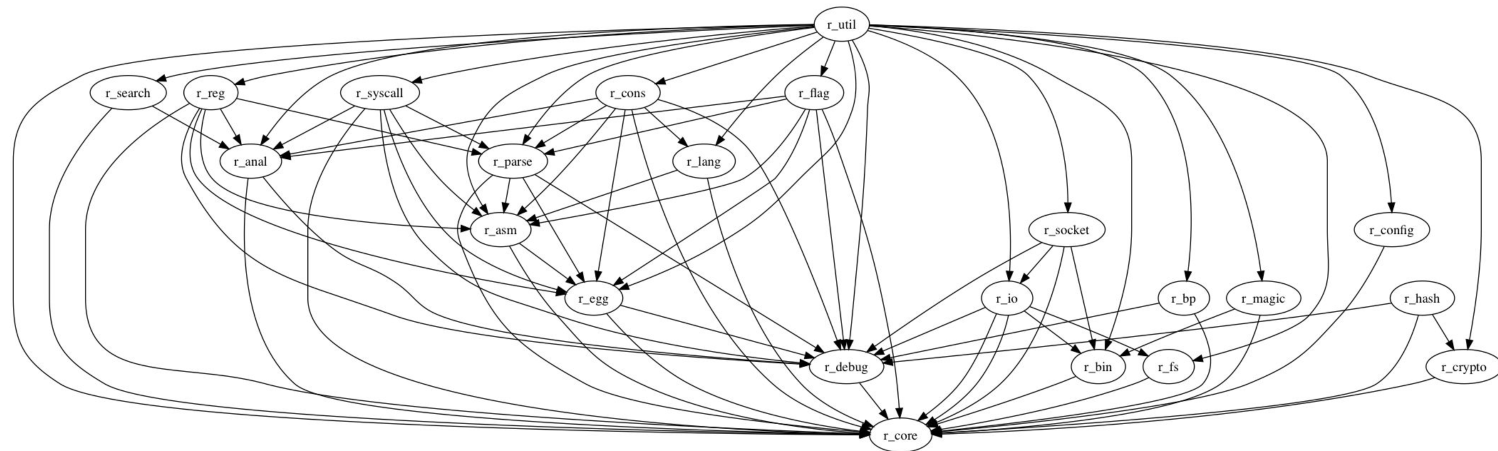
The Framework

```
$ make depgraph.png
```

- 2015



- 2018



Plugins

Almost all of those libraries can be extended with plugins. This means, that r2 codebase is pretty modular and you can do custom builds with your plugins of choice.

Disassemblers, assemblers, header format parsers, filesystems, analyzers, emulators, debuggers, new commands, etc..

Can be installed system wide or in user's home.

What can I inspect?

Targets

R2 can open any file or device via RIO which may access it from the local filesystem or remotely via rap:// http:// r2pipe:// or any other available protocol.

- Executables / Libraries (disasm + analyze)
- Firmwares (carving known headers)
- Filesystems (mount and walk)
- Raw memory dumps (search strings/data)
- PCAP files (emulate BPF rules)
- Debug info (Dwarf/PDB)

UNIX like

R2 is a big project that does a lot of stuff. This is not much unix-like, but it aims to be modular, pluggable and scriptable.

- use of pipes |
- Support for redirections >
- Push into the stdin buffer <
- Use of backticks ‘ like in a posix shell
- Internal filtering ~ (grep, less, …)
- Abstracted IO assumes everything is a file
- pipeable from the shell echo x | r2 -
- Text, JSON and r2 commands output for almost everything
- Simple command structure (mnemonics)
- Auto documented (C, man, ?)
- Almost a posix shell with ls, cp, mkdir, cat, ed..)

UNIX Like

(DEMO)

Documentation

(written in C!)

- If you wanna learn more, or just curious on some specific aspects or usages.
- There are more resources than just this talk.

Documentation

- The whole project is documented in C.
  - Badumtsss
- There are 2 books published in gitbook.
  - Radare explorations
  - Official r2 book based on r1 one
- All commands are documented inline
  - just append the question mark
- Many videos in YouTube
  - All r2con talks are uploaded asap
- Many blog posts and articles on the web

- Join the IRC or Telegram to get human driven help

But First.. A Poll!

(who are you?)

Which is your main OS?

Do you know assembly?

How's your UNIX foo?

Did you used r2 before?

Installation

(always use git)

Stick to your distro packages and don’t complain about bugs or install from Git and get ready for the awesomeness.

How To Install Radare2

There are several binary distributions of radare2

- LiveCD (unmaintained)
- Docker image
- Vagrant (r2pm -i vg)
- OSX package (on every release)
- Windows Installer (and nightly builds)
- BSD || GNU/Linux (Gentoo, ArchLinux, Void, ..)
- Use the Cloud Web user interface (http://cloud.rada.re)
    - Also works in Google Cloud
- Android app and Termux package
- Cydia package (iOS)
- Chat with the @r2bot on Telegram

Installing from Git

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

$ cd radare2; sys/install.sh
```

This will install r2 system-wide using symlinks (faster and handier for development, no make install required after every change, but risky on multiuser shells)

```
$ sys/user.sh

$ export PATH=~/bin:$PATH
```

Side Notes

Notice that r2 build system is based on:

- ACR (auto-conf-replacement)
- Handmade Makefiles

Plugins can be selected with ./configure-plugins

- Random documentation in doc/ directory
- Several useful scripts in sys/
  - sys/static.sh sys/asan.sh ...

Package Management

We can even install r2 via r2pm and get rid of our r2 dir

```
$ r2pm -i radare2

$ rm -rf radare2
```

You can also install other programs, plugins and scripts with it. Everything in your home by default.

r2pm depends on r2 if you didnt cached the setup options, this may be improved in 2.5.

Package Management

Some of the most interesting packages:

- Yara 3
- RetDec decompiler (@nighterman)
- Keystone - assemble instructions
- Unicorn - code emulator
- Native Python bindings
- AGC - Apollo 11 CPU
- Duktape (Embedded javascript)
- Radeco decompiler (@sushant94)
- Baleful (SkUaTeR)
- r2pipe apis for NodeJS, Python, Ruby, C#, ...
- Vala/Vapi/Valabind/Swig/Bokken/…
- r2frida

Package Manager

(Demo)

But Hey!

We have a GUI

You may probably want to also install Cutter

https://rada.re/cutter/

Cutter

Rebranded Iaito originally written by Hugo Teso

- Multiplatform QT5 gui
  - Windows, macOS, Linux/*BSD/Haiku
- Releases in sync with r2 ones
- Looking for contributors

Cutter: Downloading releases

- Binary builds are published every 6 weeks
- In sync with r2 releases
- You can download them from the github releases page
  - Appimage for Linux
  - Dmg for macOS
  - Wizard installer for Windows

https://rada.re/cutter/

Cutter: Installing from Git

```
git clone https://github.com/radareorg/cutter

cd cutter

mkdir build

cd build

qmake ..

make -j4
```

Many Other GUIs

Several GUIs are available, with web technologies, GTK, blessed, etc.. None of them really get enough traction to be maintained (or used) actively.

The CLI will always be superior, but for some users it may be good

Which is Your Favourite UI?

(Yes, that's another poll)

- The CLI
- Visual Mode
- Tiled Panels
- WebUI
- Cutter
- R2Pipe

Mine is CLI+Visual

All new features are always accessible thru the command line interface of radare2.

The testsuite basically tests commands, not api.

The most common actions are integrated into the Visual mode that can be managed using keystrokes instead of typing commands.

CLI is expressive and powerful once you understand the logic behind the commands syntax.

Running r2

```
man r2

r2 -h


rarun2
```

Some r2 Command Line Flags

```
-h           # get help message
-a <arch>    # specify architecture (RAsm Plugin name)
-b <bits>    # specify 8, 16, 32, 64 register size in bits
-c <cmd>     # run command
-i <script> # include/interpret script
-n           # do not load rbin info
-L           # list io plugins
```

Spawning an R2 Shell

The `r2` command is a symlink for `radare2`:

```
$ r2 -           # alias for `radare2 malloc://1024`
$ r2 --          # open r2 without any file opened
$ r2 /bin/ls     # open this file in r2
$ r2 -d ls       # start debugging
```

Files

R2 IO abstract the access to what’s provided by an IO plugin, this layer allows to:

- Load multiple files and map them at virtual addresses
- Define sections to virtualize the memory layout
- Handle write cache to avoid modifying the original files
- Write operations change the target file
- Specify different permissions to each map (rwx)
- Different cache (read, write, pa, va, …)

Use the `o` command to manage the files.

Understanding IO

- Files are represented by a descriptor
- Maps will put a specific region of an fd in virtual addressing
- Bin info can be used to inspect sections, etc
- io.va variable allows us to choose between va and pa
- Maps can be overlapped and priorized.
- oL to list all the plugins
- Difference between maddr, baddr, paddr, vaddr, ...

Basic Commands

Seeking

Change current position, accepts flags, relative offsets, math ops. Use @ for temporal seeks.

Printing

Show current block (b) bytes, instructions, metadata, analysis, ...

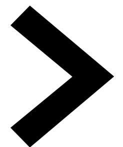
Writing

Write string, hexpairs, file contents, instructions, etc..

In The Shell

Syntax of the commands:

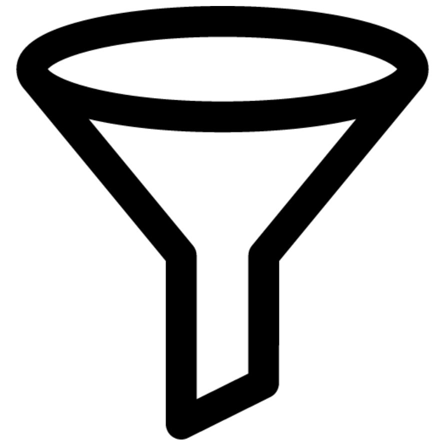
```
> [repeat][command] [args] [@ tmpseek] [; ...] [# comment]


> 3x              # perform 3 hexdumps in the same address
> pd 3 @ entry0   # disasm 3 instructions at entrypoint
> x@rsp;pd@rip    # show stack and code
```



The Internal Grep

As long as r2 is portable, it doesn’t depends on other programs, so there are some basic unix commands, as well as an internal grep/less.

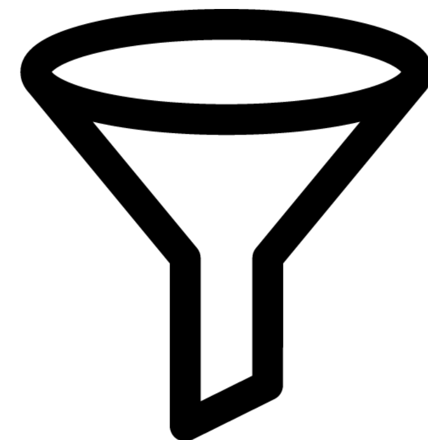
```
> pd~call

> is~test

> is~?

> ~??          # show help
```



Interpreting Scripts

The output of any program, command or script (or contents of a file) can be interpreted as r2 commands. Use the ‘.’.

- . file > interpret file as r2 commands
- .r2cmd* > interpret output of command as r2 commands
- .!bin > run system and interpret output as r2 commands



If the file have an extension it will try to

Run it using #!pipe to make .py, .js, r2pipe

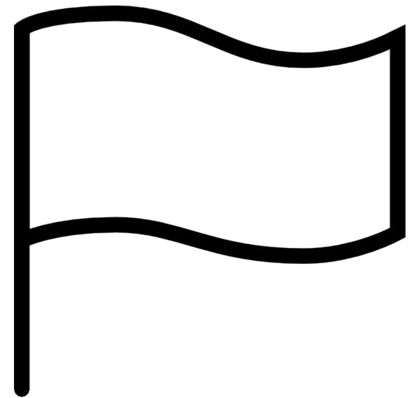
Connection happy.

Flags and Calculations

Flags are used to specify a name for an offset.

Math expressions evaluate those names to retrieve a number.

```
> ? 1+1

> f foo = 1024    vs    f bar @ 1024

> ? foo+123

> ? [123]

> ?v
```



Flags In Disasm

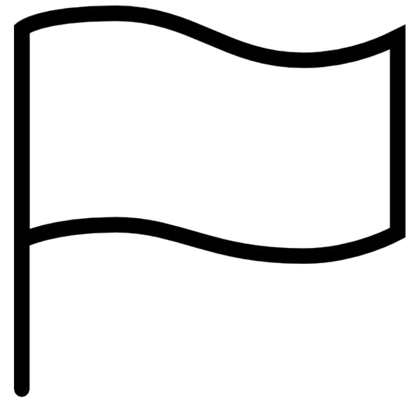
Flags can be displayed in the disasm as labels to show a name for an offset.

But those can be also displayed in the instruction disassembly itself replacing absolute addresses.

```
> e anal.varsub = true
```

Flags must contain a dot, to avoid confusion

With register names.



Help

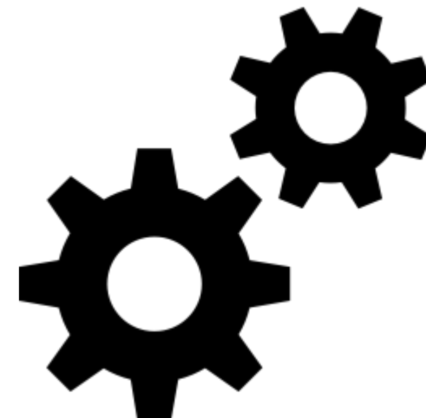
The ? Command is used here for evaluating math expressions, but it have many more functionalities. See that with ???

- prompt the user ?i
- Show only the value ?v
- Resolve nearest flag+ delta ?d
- Conditional execution ??
- Echo messages ?e
- Benchmark commands ?t
- Clippy! ?E

Configuration

Almost everything in r2 can be configured with ‘e’.

```
> e asm.
> e asm.arch=?
> e??rop
> e* > settings.r2
> . settings.r2
```



Printing Bytes

R2 is an block-based hexadecimal editor. Change the block size with the ‘b’ command.

p8	print hex-pairs
px	print hexdump
pxw/pxq	dword/qword dump
pxr	print references (drr)
pxe	emojis
pxa	Show dump map

Structures

pf - define function signatures

Can load include files with the t command.

010 templates can be loaded using 010 python script.

Load the bin with **r2 -nn** to load the struct/headers definitions of the target bin file.

Use **pxa** to visualize them in colorized hexdump.

There’s also support for **Kaitai**

Structures

(DEMO)

- Parse **mach0** header
- Use macho.h
- Use r2 -nn

Disassembling

(and printing bytes)

Disassembling decodes bytes into meaningful instructions.

rasm2

Disassembling and assembling code can be done with **pa/pad** or using the rasm2 command line tool.

```
$ rasm2 -L

$ rasm2 -a x86 -b 64 nop

$ rasm2 -d 90
```

(demo)

Disassembling Code

There are different commands to get the instructions at a specific address.

pd/pD	**disassemble N bytes/instructions.**
pi/pI	**just print the instructions**
pid	**print address, bytes and instruction**
pad	**disassemble given hexpairs**
pa	**assemble instruction**

Disassembling Code

```
> e asm.emu=true      emulates the code with esil
> e asm.emu.str=true show only string refs computed in emu
> agv/agf.            render ascii art or graphviz graph
```

```
Seek History       s- (undo)    s+ (redo)
```

Use u and U keys to go back/forward in the visual seek history.

Patching Code

The ‘w’ command allows us to write stuff

- Open with r2 -w (by default is readonly except debugger)
- VA/PA translations are transparent
- Sometimes we will need to use r2 -nw to patch headers
- The w command also allows to write assembly
- Wx in hexpairs
- Visual CJMP patching
- wxf

DEMO: patch simple crackme program

Dumping, Restoring and Clipboard

Dump to file

```
> pr 1K > file
```

```
> wtf file 1K
```

```
> y 1K
```

Restoring

```
> wf file @ dst
```

```
> yy @ dst
```

Copy

```
> yt 1K @ dst
```

```
- offset -   | 0 1  2 3  4 5  6 7| 01234567
0x100001174 |5548 89e5 4157 4156| UH..AWAV
0x10000117c |4155 4154 5348 81ec| AUATSH..
0x100001184 |3806 0000 4889 f341| 8...H..A
0x10000118c |89fe 488d 85c0 f9ff| ..H.....
0x100001194 |ff48 8985 b8f9 ffff| .H......
0x10000119c |4585 f67f 05e8 5932| E.....Y2
0x1000011a4 |0000 488d 3543 3900| ..H.5C9.
0x1000011ac |0031 ffe8 dc33 0000| .1...3..
0x1000011b4 |41bc 0100 0000 bf01| A.......
0x1000011bc |0000 00e8 7233 0000| ....r3..
0x1000011c4 |85c0 7461 c705 fe42| ..ta...B
0x1000011cc |0000 5000 0000 488d| ..P...H.
0x1000011d4 |3d18 3900 00e8 2e33| =.9....3
0x1000011dc |0000 4885 c074 0f80| ..H..t..
```

Decompilation

Better disassembly

First let's see how we can improve the disassembler output.

```
> e asm.emu.str=true

> e asm.pseudo=true

> pds

> pdc
```

Decompilers for radare2

Decompiling is not just showing the disassembly in a better way. It requires understanding what the code does, emulating it, removing dead code, and perform several optimizations and mix it with type information to get a C like output.

- **Boomerang** Abandoned
- **Snowman** Supported
- **Retdec** supported
- **Radeco** wip (gsoc)
- **r2dec** Actively maintained

r2dec

Wip and experimental decompiler written in NodeJS + r2pipe.

Developed by @deroad. To install type this:

- **r2pm -ci r2dec**

Using use it:

- **af**
- **#!pipe r2dec**

Decompiler Demo

(DEMO)

- Use r2dec to decompile some functions

r2pm -ci r2dec

User Interface

- WebUI
- Bokken
- Visual Mode
- Visual Panels
- Command line
- R2Pipe
- Colors!

Colors!

```
> e scr.color=true

> e scr.rgb=true

> e scr.truecolor=true

> e scr.utf8=true


> ecr      # Random colors

> eco X    # Color palette

> VE       # visual color theme editor
```

```
[0x100001058]> ecr
[0x100001058]> pd 20
            ;-- main:
            ;-- entry0:
            0x100001058     55              push rbp
            0x100001059     4889e5          mov rbp, rsp
            0x10000105c     4157            push r15
            0x10000105e     4156            push r14
            0x100001060     4155            push r13
            0x100001062     4154            push r12
            0x100001064     53              push rbx
            0x100001065     4881ec480600.   sub rsp, 0x648
            0x10000106c     4889f3          mov rbx, rsi
            0x10000106f     4189fe          mov r14d, edi
            0x100001072     488d85c0f9ff.   lea rax, [rbp - 0x640]
            0x100001079     488985b8f9ff.   mov qword [rbp - 0x648], rax
            0x100001080     4585f6          test r14d, r14d
        ,=< 0x100001083     7f05            jg 0x10000108a
        |   0x100001085     e877330000      call 0x100004401
        |     ^- 0x100004401() ; main
        `-> 0x10000108a     488d357f3a00.   lea rsi, [rip + 0x3a7f]
ng ; section.4.__cstring
            0x100001091     31ff            xor edi, edi
            0x100001093     e806350000      call sym.imp.setlocale
              ^- 0x10000459e() ; sym.imp.setlocale
            0x100001098     41bd01000000    mov r13d, 1
            0x10000109e     bf01000000      mov edi, 1
[0x100001058]>
```

Scripting with r2pipe

It is possible to script r2 using almost any programming language out there. This is possible thanks to r2pipe. A simple interface to run commands and get the output in a string which can be processed as json to avoid parsing issues.

```
import r2pipe

r2 = r2pipe.open("/bib/ls")

print(r2.cmd("pd"))

r2.quit()
```

Visual Mode

Type V and then change the view with ‘p’ and ‘P’

```
[0x100001072 0% 125 /bin/ls]> f tmp;sr s.. @ main+26 # 0x100001072
0x7fff5fbff8b0  0x00000000 0x00000000 0x00000000 0x00000000  ................
0x7fff5fbff8c0  0x00000000 0x00000000 0x00000000 0x00000000  ................
0x7fff5fbff8d0  0x00000000 0x00000000 0x00000000 0x00000000  ................
0x7fff5fbff8e0  0x00000000 0x00000000 0x00000000 0x00000000  ................
 rax 0x7fff5fbff8e0      rbx 0x7fff5fbfff48      rcx 0x7fff5fbfff60
 rdx 0x7fff5fbfff58      rdi 0x00000001          rsi 0x7fff5fbfff48
 rbp 0x7fff5fbfff20      rsp 0x7fff5fbff8b0       r8 0x00000000
  r9 0x7fff5fbfefd0      r10 0x00000032          r11 0x00000246
 r12 0x00000000          r13 0x00000000          r14 0x00000001
 r15 0x00000000          rip 0x100001079         rflags = 1TI
            0x100001072    488d85c0f9ff.  lea rax, [rbp - 0x640]
            ;-- rip:
            0x100001079    488985b8f9ff.  mov qword [rbp - 0x648], rax
            0x100001080    4585f6         test r14d, r14d
        ,=< 0x100001083    7f05           jg 0x10000108a                ;[1]
        |   0x100001085    e877330000     call 0x100004401              ;[2]
        |     ^- 0x100004401() ; rip
        `-> 0x10000108a    488d357f3a00.  lea rsi, [rip + 0x3a7f]       ; 0x1000
            0x100001091    31ff           xor edi, edi
            0x100001093    e806350000     call sym.imp.setlocale        ;[3]
              ^- 0x10000459e() ; sym.imp.setlocale
            0x100001098    41bd01000000   mov r13d, 1
            0x10000109e    bf01000000     mov edi, 1
            0x1000010a3    e89c340000     call sym.imp.isatty           ;[4]
```

Visual Panels

Press ‘!’ in the Visual mode

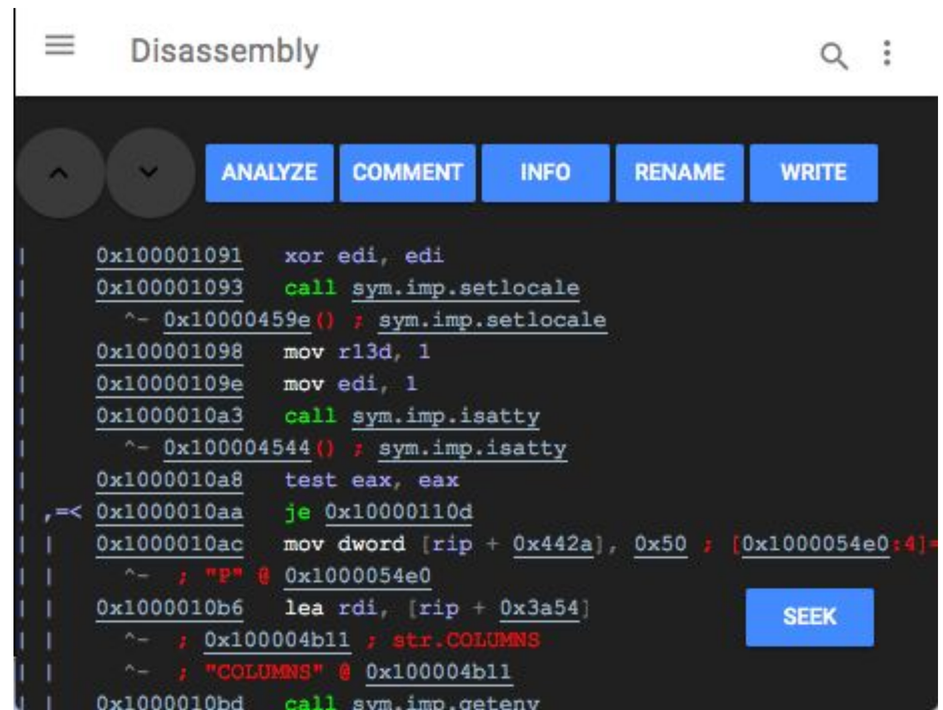
```
> File  Edit  View  Tools  Search  Debug [Analyze] Help            [0x100001060]
 Disassembly
        0x100001060    push r13                  > Function           00 0 __mh_execute_he
        0x100001062    push r12                    Program            2 0 radr:__5614542
        0x100001064    push rbx                    Calls              4e 0 imp.__assert_rt
        0x100001065    sub rsp, 0x648              References          4 0 imp.__bzero
        0x10000106c    mov rbx, rsi                                   5a 0 imp.__error
        0x10000106f    mov r14d, edi
        0x100001072    lea rax, [rbp-local_200]         Stack
        0x100001079    mov qword [rbp-local_201],   - offset -    0 1  2 3  4 5  6
        0x100001080    test r14d, r14d              0x00000000   cffa edfe 0700 00
     ,=< 0x100001083   jg 0x10000108a               0x00000010   1300 0000 1807 00
     |  0x100001085    call 0x100004401             0x00000020   1900 0000 4800 00
     `-> 0x10000108a   lea rsi, [rip + 0x3a7f]      0x00000030   524f 0000 0000 00
        0x100001091    xor edi, edi                 0x00000040   0000 0000 0100 00
        0x100001093    call sym.imp.setlocale
        0x100001098    mov r13d, 1                    Registers
        0x10000109e    mov edi, 1                   rax 0x00000000          rbx 0
        0x1000010a3    call sym.imp.isatty          rdx 0x00000000          rsi 0
        0x1000010a8    test eax, eax                 r8 0x00000000           r9 0
     ,=< 0x1000010aa   je 0x10000110d               r11 0x00000000          r12 0
     |  0x1000010ac    mov dword [rip + 0x442a],    r14 0x00000000          r15 0
     |  0x1000010b6    lea rdi, [rip + 0x3a54]      rbp 0x00000000          rflag
```

Web User Interface

Start the webserver with =h

Launch the browser with =H

See /m /p /t and /enyo



R2 have an embedded webserver

No extra deps required.

Looking for contributors!

Cutter

Rebranded Iaito originally written by Hugo Teso

- Multiplatform QT5 gui
  - Windows, macOS, Linux/*BSD/Haiku
- Looking for contributors
- Actively maintained, in sync with r2 releases

Deeper Look into the Visual Mode

Visualization

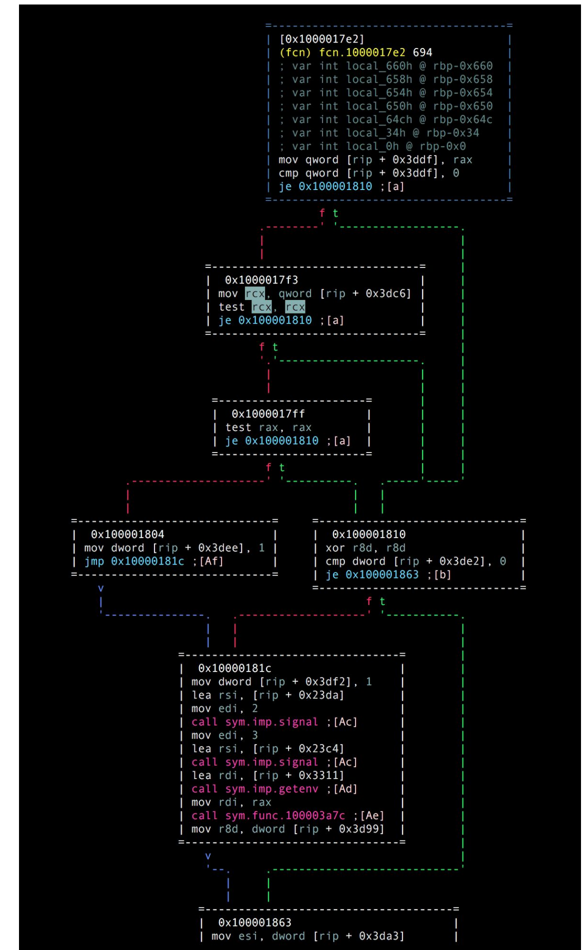
- Toggle Colors (C)
- Highlight stuff with (/)
- Setting new commands on top and right with = and |
- <space> toggle between graph and disasm

```
           0x1000017e9          48833ddf3d00.   cmp qword [rip + 0x3ddf], 0
       ┌─< 0x1000017f1     *    741d            je 0x100001810               ;[2]
       │   0x1000017f3          488b0dc63d00.   mov rcx, qword [rip + 0x3dc6]
       │   0x1000017fa          4885c9          test rcx, rcx
      ┌──< 0x1000017fd          7411            je 0x100001810               ;[2]
      ││   0x1000017ff          4885c0          test rax, rax
     ┌───< 0x100001802          740c            je 0x100001810               ;[2]
     │││   0x100001804          c705ee3d0000.   mov dword [rip + 0x3dee], 1
    ┌────< 0x10000180e          eb0c            jmp 0x10000181c              ;[3]
    │└└└─> 0x100001810          4531c0          xor r8d, r8d
    │      0x100001813          833de23d0000.   cmp dword [rip + 0x3de2], 0
    │  ┌─< 0x10000181a          7447            je 0x100001863               ;[4]
    └────> 0x10000181c          c705f23d0000.   mov dword [rip + 0x3df2], 1
       │   0x100001826          488d35da2300.   lea rsi, [rip + 0x23da]
```

Deeper Look into the Visual Mode

Navigation

- Cursor Mode (Vc)
- Browse
- HUD (V_)
- Resize Hexdump with []
- Add comments (;)
- Undo/Redo seek (u/U)
- Find next/prev hit/func/.. With n/N
- Basic Block Graphs
- Tab to choose panel
- Use = and |
- Highlight words with /
- ? Help for more



Deeper Look into the Visual Mode

Debugging

- Debugger integration
  - Seek to PC (.)
  - Step (s) or StepOver (S)
  - Set breakpoints with ‘b’
- Change stack settings
- Change register values
- Continue until X
- Watchpoints, etc..

```
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fff5fbfff10  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7fff5fbfff20  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7fff5fbfff30  0100 0000 0000 0000 90ff bf5f ff7f 0000  ..........._....
0x7fff5fbfff40  0000 0000 0000 0000 0000 0000 0000 0000  ................
 rax 0x00000000          rbx 0x00000000          rcx 0x00000000
 rdx 0x00000000          rdi 0x100000000         rsi 0x00000001
 rbp 0x7fff5fbfff28      rsp 0x7fff5fbfff10       r8 0x00000000
  r9 0x00000000          r10 0x00000000          r11 0x00000000
 r12 0x00000000          r13 0x00000000          r14 0x00000000
 r15 0x00000000          rip 0x7fff5fc01011      rflags 1TI
      0x7fff5fc01000    5f              pop rdi
      0x7fff5fc01001    6a00            push 0
      0x7fff5fc01003    4889e5          mov rbp, rsp
      0x7fff5fc01006    4883e4f0        and rsp, 0xfffffffffffffff0
      0x7fff5fc0100a    4883ec10        sub rsp, 0x10
      0x7fff5fc0100e    8b7508          mov esi, dword [rbp + 8]
      ;-- rip:
      0x7fff5fc01011    488d5510        lea rdx, [rbp + 0x10]
      0x7fff5fc01015    4c8b051c8b03.   mov r8, qword [rip + 0x38b1c]
      0x7fff5fc0101c    488d0ddddffff.  lea rcx, [rip - 0x23]
      0x7fff5fc01023    4c29c1          sub rcx, r8
      0x7fff5fc01026    4c8d05d3efff.   lea r8, [rip - 0x102d]
      0x7fff5fc0102d    4c8d4df8        lea r9, [rbp - 8]
      0x7fff5fc01031    e840000000      call 0x7fff5fc01076            ;[1]
      0x7fff5fc01036    488b7df8        mov rdi, qword [rbp - 8]
      0x7fff5fc0103a    4883ff00        cmp rdi, 0
   ┌─< 0x7fff5fc0103e    7510            jne 0x7fff5fc01050             ;[2]
   │   0x7fff5fc01040    4889ec          mov rsp, rbp
   │   0x7fff5fc01043    4883c408        add rsp, 8
   │   0x7fff5fc01047    48c7c5000000.   mov rbp, 0
   │   0x7fff5fc0104e    ffe0            jmp rax
   └─> 0x7fff5fc01050    4883c410        add rsp, 0x10
      0x7fff5fc01054    57              push rdi
```

Deeper Look into the Visual Mode

Editing stuff

- Bit Editor (Vd1)
- Increment/Decrement bytes (Cursor + +/- keys)
- Select ranges bytes to copy/paste
- Define flags
- Interactive writes
    - A : rewrite assembly in place
    - I : insert hex/ascii stuff

Color Themes

Color themes are r2 scripts that run ec* commands to change the color palette.

- Portable across windows and *unix
- Supports 16, 256 and truecolor setups
- Supports html output
- Character attributes:
  - Bold, italic, bgcolor, …
- Supports utf8 chars (text width is not constant)

See eco and ec commands for more information

Binary Info

(parsing file formats)

RBin detects file type and parses the internal structures to provide symbolic and other information.

RBin Information

```
$ rabin2 -s
> is
> fs symbols;f
```

Symbols	Relocs	Classes	Entrypoints
Imports	Strings	Demangling	Exports
Sections	Libraries	SourceLines	ExtraInfo

RBin Information

All this info can be exported in JSON by appending a ‘j’.

```
$ r2 -nn /bin/ls
> e scr.hexflags=9999
> pxa
```

(DEMO)

Rebasing Symbols

Check binary information to see if its relocatable by checking the “pic” field in rabin2 -I

Symbols represent public intormation of name=address. This is exported symbols from the binary or library, the imports in the plt, the function information of mach0 binaries, methods in java/dalvik binaries, etc..

Those can be rebased with:

$ rabin2 -B 0x800000 /bin/ls

Imports

The imports are the functions that must be resolved by the runtime linker from the libraries linked to allow the program run.

On windows binaries, imports specify the library where the symbol must be found so its reflected in its name:

$ rabin2 -i

Classes and methods

R2 does name demangling by default. (e bin.demangle=false)

The information of classes and methods can befound in:

- objc metadata
- Class/Dex
- Symbol name with :: separators
- C++ Vtables

Sections

Some of them are mapped and some others don’t. Executables use to provide the information duplicated. One simplified for the loader and another for analysis, exposing swarf information, annotations, etc

> **iS**

> **.iS***

> **S=**

> **S-***

```
[0x100001174]> S=
00* 0x100000e94 |#####################------| 0x10000442d mr-x 0.__text  13.4K
01  0x10000442e |--------------------#------| 0x1000045f6 mr-x 1.__stubs  0456
02  0x1000045f8 |--------------------##-----| 0x100004900 mr-x 2.__stub_helper  0776
03  0x100004900 |---------------------##----| 0x100004af0 mr-x 3.__const  0496
04  0x100004af0 |----------------------###--| 0x100004f69 mr-x 4.__cstring  1.1K
05  0x100004f6c |------------------------#--| 0x100005000 mr-x 5.__unwind_info  0148
06  0x100005000 |------------------------#--| 0x100005028 mrw- 6.__got  0040
07  0x100005028 |------------------------#--| 0x100005038 mrw- 7.__nl_symbol_ptr  0016
08  0x100005038 |------------------------##-| 0x100005298 mrw- 8.__la_symbol_ptr  0608
09  0x1000052a0 |-------------------------##| 0x1000054c8 mrw- 9.__const  0552
10  0x1000054d0 |--------------------------#| 0x1000054f8 mrw- 10.__data  0040
11  0x100005500 |--------------------------#| 0x1000055c0 mrw- 11.__bss  0192
12  0x1000055c0 |--------------------------#| 0x10000564c mrw- 12.__common  0140
=>  0x100001174 |---------------------------| 0x100001274
[0x100001174]>
```

Hashing Sections

Rahash2 allows us to compute a variety of checksums to a portion of a file, a full file or by blocks.

$ rahash2 -a md5 -s “hello world”

$ rahash2 -a all /bin/ls

$ rabin2 -SK md5 /bin/ls

$ rahash2 -L

- Also supports encryption/decryption
- As well as encoding/decoding

Entropy

The entropy is computed by the amount of different values in a specific block of data.

- **Low entropy** = plain/text
- **Middle entropy** = code
- **High entropy** = compressed / encrypted

There are other methods to identify

- **p=e**
- **p=p**
- **p=0**
- ...

Visualizing Big Regions

There are many ways to represent the contents of a buffer in r2. This is, by computing a value that represents each block.

But also, we have analysis maps (each instruction type is rendered with a different color).

These “zoomed” view mode is useful when trying to find a region of interest, that may contain text, nulls, etc..

Strings

Strings can be stored in different places inside the binaries:

- In .rodata section
- Inside the .text (code)
- In headers (interpreter, libraries, symbol names, ..)

Also, we can find strings in a variety of file types:

- Raw memory dump
- Hard disk image
- Known file format
- Debugged process
- Emulated code to find references or construct strings
- Encoded (base64, utf16, ...)
- Encrypted

Strings

So we have different commands depending on that:

```
$ rabin2 -z      # strings from .rodata (default in r2)
$ rabin2 -zz     # strings in full file
$ rabin2 -zzz    # dont map once, useful for huge files like 1TB
```

Radare2 will load the strings by default, which is sometimes not desired, see the following vars:

```
> e bin.strings=false
> e bin.maxstrbuf=32M
```

Scripting

(automation)

Automating actions in r2 using your favourite programming language (or not).

Scripting

- **Shellscript (batch mode)**
  - Use ‘jq’ to parse json output
  - Send commands via stdin
- **Bindings (full api)**
  - Also supports Python, Java, ...
- **Plugins**
  - Loaded from home and system directories
- **R2Pipe scripts**
  - spawn/pipe/http/...
  - C / C++QT / C#/.NET / Erlang / Haskell / Lisp / NodeJS / Python / Perl / Ruby / Rust / Go / Swift / Java / Nim / Perl / Vala...
  - Interpreted with ‘.’ command

Using R2Pipe For Automation

R2 provides a very basic interface to use it based on the cmd() api call which accepts a string with the command and returns the output string.

```
$ pip install r2pipe

$ r2 -qi names.py /bin/ls

$ cat names.py
```

Other uses of r2pipe

R2pipe provides also the ability to expose an API to implement plugins in alternative languages. Right now only for Python and NodeJS. But it can be easily ported to other languages.

- Syscall implementations for ESIL
- IO plugins via r2 r2pipe://”node …”
- Asm plugins via r2 -i asmArch.js
- Bin plugins can be aksi dine via r2pipe

Running r2pipe scripts with the . command

R2pipe Performance

If you are worried about using r2pipe instead of the native API.
It must also care about other aspects like maintainability, portability, stability, etc

- Pipe + JSON parsing is faster than FFI
- Textual representation, easy to debug
- Native language objects and idiomatic access to fields
- There are many different r2pipe backends
  - http is slow
  - rap a bit faster
  - spawn and doing pipes
  - native (dlopen+dlsym(r_core_cmd_str))

R2pipe On BigData

Using async programming with r2pipe allows us to split the user interface to the logic of the program which results in more responsiveness.

R2 can not execute more than one command at a time so if a long process is happening it will queue the rest.

For this cases we must consider splitting the process into smaller operations to avoid huge replies that may fail depending on transport and long operations that will make r2 eat all cpu.

Analyzing Code

(and graphing)

Analyzing code unveils the real code structures that is defined by the instruction listings and find references, function boundaries, local variables, identify types, etc..

Analyzing From The Metal

R2 provides tools for analyzing code at different levels.

ae emulates the instruction (microinstructions)

ao provides information about the current opcode

afb enumerate basic blocks of function

af analyzes the function (or a2f)

ax code/data references/calls

Analyzing the Whole Thing

Many people is used to the IDA way: load the bin, expect all xrefs, functions and strings to magically appear in there.

This is the default behaviour, which can be slow, tedious, and 99% of the time we can solve the problem quicker with direct and manual analysis.

Run `**r2 -A**` or use the **‘aa’** subcommands to achieve this.

- **aa**
- **aaa**
- **aaaa**
- **aaaaa # :D**

Analyzing the Whole Thing

- The proper way to analyze programs is not to rely on the default analysis loops under aaa, but rather understand what each command does and which one fits better to solve the problems you are facing.
- Not all xrefs are usually required, so finding only the ones you are interested in is interesting to save some time.
- anal.from/to can be used to restrict boundaries.
- aab and aac are pretty useful to find all functions and call refs.

Low Level Anal Tweaks

Use the anal hints command to modify instruction behaviours by hand.

> ahs 1 @ 0x100001175

(DEMO) Jump in the middle of instruction

> e asm.middle

Searching for code

We can search for some specific code in a binary or memory.

- **/R [expr]** search for ROP gadgets
- **/r sym.imp.printf** find references to this address
- **/m** search for magic headers
- **Yara** identify crypto algorithms
- **/a [asm]** assemble code and search bytes
- **/A [type]** find instructions of this type
- **/c [code]** find strstr matching instructions
- **/v4 1234** search for this number in memory
- **pxa** disasm all possible instructions
- **e asm.emustr=true** pD $SS @ $S~Hello

Graphing Code

```
[0x100001058]> 0 VV @ entry0 (nodes 42 edges 60 zoom 100%) BB mouse:ca

[0x100001058]
push rbp
mov rbp, rsp
push r15
push r14
push r13
push r12
push rbx
sub rsp, 0x648
mov rbx, rsi
mov r14d, edi
lea rax, [rbp - 0x640]
mov qword [rbp - 0x648], rax
test r14d, r14d
jg 0x10000108a

f t

0x100001085
call 0x100004401

v

0x10000108a
lea rsi, [rip + 0x3a7f]
xor edi, edi
call sym.imp.setlocale
mov r13d, 1
mov edi, 1
call sym.imp.isatty
test eax, eax
je 0x10000110d

t f

0x10000110d
lea rdi, [rip + 0x39fd]
call sym.imp.getenv
test rax, rax
je 0x10000112c

f t

0x1000010ac
mov dword [rip + 0x442a], 0x50
lea rdi, [rip + 0x3a54]
call sym.imp.getenv
test rax, rax
je 0x1000010d6
```

Functions can be rendered as an ascii-art graph using the ‘ag’.

Enter visual mode using the V key

Then press V again (or spacebar) to get the graph view.

Graphing Code

The graph view is the result of the agf command and it permits to:

- Move nodes
- Zoom in/out
- Relayout
- Switch between different graph modes
  - Callgraph
  - Refs graph
  - Control Flow Graph (basic blocks)
  - Change contents of nodes (pds, pd, ..)

Graphing Code

R2 can also use graphviz, xdot or web graph to show this graph to the user, not just in ascii art.

```
> agv
> ag $$ > a.dot
```

Show how to export function and basic block information.

Doing Your Custom Graphs

You can create your own graphs, or write code that spits agn/age commands to render an ascii-art graph.

- See agn/age commands

(DEMO)

Signatures

(and graphing)

Signatures is the "art" of identifying functions by looking at byte patterns.

Preludes

There are many ways to identify functions inside a binary, one of them is using signatures to find the beginning of them. The aap command will run different search patterns depending on arch/bits/os:

- **aap** - function preludes

It is also possible to perform searchs with /x and run a command on each offset:

We can also use strings as signatures and use /

> /x 898989

> pd 5 @@ hit*

Signatures

The signatures define a more fine-grained view of the function. Which excludes the parts of the instruction that can vary depending on compilation time. This is similar to how IDA FLIRT signatures work, and in fact we also support them via the zF command

```
$ r2 -A static-bin

> zg lebin > lebin.r2

> zo lebin.r2

> z/ $$
```

Modern Signatures

Radiff2 allows to find differences in code by trying to find two functions that match and compares them internally.

Zignatures can be defined to follow some specific metrics extracted from the code analysis information.

- Afi
- Calling convention
- Number of arguments
- Number of local variables
- Number of exit points
- Cyclomatic Complexity
- ...

References

The code and data is referenced by ref and xref structs, using the axt command we can inspect them.

Finding references to strings is an important task and r2 have different commands that may help on the analysis.

```
> aav

> aae

> /r

> pD $SS @ $S~Hello
```

How Do References Look Like?

References can be on many types:

- Read / Write / Exec
- CALL, JMP, LEA
- Code, Data (Type Of Data)

Some references are implicit in one instruction.

Others are computed by a sequence of instructions.

Some reuse register values, and recursive emulation

Also, hardcoded relative or absolute addresses..

Finding References

References can be on many types:

- Read / Write / Exec
- CALL, JMP, LEA
- Code, Data (Type Of Data)

BinDiffing

(and graphing)

Finding differences between two binaries looking for bugfixes.

Checking differences

Being able to identify what is different from two files is important, there are many ways to do that:

- At byte level
- With delta diffing
- Permit some % of aproximation
- At code level (function, basic block, ..)

Lets try that:

- radiff2 -x fileA fileB
- two column hexdump in r2 (cc $$ @ $$+1)
- DEMO: radiff2 with all the modes
- Creating a patch with -r

Finding the Change

DEMO: Identify what is different between two executables

Create a patch, analyze the changes...

- **radiff2 -r fileA fileB**

Applying the patch:

- **r2 -qwni patch.r2 orig**

Debugging

(and emulation)

R2 supports native debugger for Linux, BSD, XNU and Windows.

But there’s more!

What Is Debugging?

R2 is a low level debugger (not a source debugger).

It provides much more low level information than source debuggers use to provide. Doesn’t competes with GDB/LLDB.

Basic Actions for a debugger are:

ds	**step**	**db**	**breakpoint**	**dr**	**show regs**
dso	**step over**	**dcu**	**continue-until**	**dx**	**code-inject**
dc	**continue**	**dm**	**memory-maps**	**dd**	**file-desc**

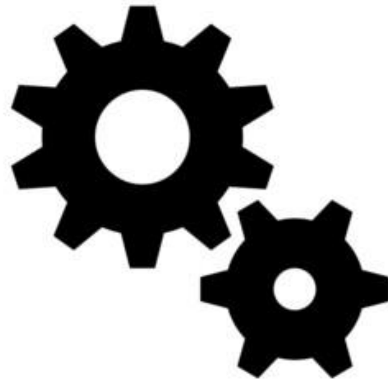
...

The state of the process

The process state is represented by this information:

- **Memory (maps, dm)**
- **Registers**
- **Threads (shared memory, unique regs)**
- **File Descriptors**

This state can be saved and restored with the dmp command.



IO != Debug

R2 have different plugins to interact with external resources like processes.

- IO plugins abstract the access to reading memory
- RDebug shares a link with IO to set breakpoints, memory,.

We can open a process or debug it:

```
$ r2 -d vs r2 dbg://
```

We can also debug ourselves:

```
$ rarun2 r2preload=true program=/bin/cat

$ r2 self://
```

Registers

Retrieved with the dr command

- Store last two states to colorize diffs
- Imported into r2 as flags .dr*
- Special register names for generic ones PC, SP, …
- Change its value with dr regname=value
- Debug registers are accessed with the drx command
- Register profiles define how are they stored

Memory

The memory in the process is organized in maps. Those are virtual regions of memory that can be a map of a file or just allocation for the heap.

Each page have its permissions and sometimes an associated name that allows us to identify which library is in there.

We can change those permissions to force page fault exceptions and emulate

> pxr @ rsp

> dm

> dms (memory snapshots)

ASLR and Rarun2

Rarun2 is a tool that allows us to spawn a process with a specific environment and configuration. It is ideal to construct reproducible runs without much hassle.

ASLR is the ability of the linker to map the binaries on different virtual addresses on each run. Some systems allow to disable this feature and rarun2 can do that.

```
$ rarun2 aslr=no program=./test

$ r2 -e dbg.profile=test.rarun2 -d test
```

Stack and Heap

Stack is where the function frame is stored, we can check local variables values in there.

- Return address stored in the stack
- Reconstruct backtrace with dbt command
- e dbg.btalgo=?
- pxr @ rsp

Heap is dynamically allocated by request of the program and is structured and not lineal like code or stack is.

- dmh command (only available on Linux for now)
- There are several implementations, a single process can have more than one heap, even per-thread.

Threads

A process can raise different signals:

- New thread created (clone)
- New process spawned (fork)
- New library loaded (windows)
- Syscall executed (dcs)
- Signal received (dck / dk / dko)
- Is dead (di)

File Descriptors

The kernel will expose the file descriptors opened by the process. R2 allows to enumerate and do different things by injecting code in the target process.

- open a new file
- Dup2 to replace one file descriptor
- Close a file

This code injection functionality can be useful for other places and its exposed in dx command.

Injecting code

This code injection functionality can be useful for other places and its exposed in dx command.

Inject code to spawn a shell generated by ragg2

```
$ ragg2-cc -a x86 -b 64 -k darwin -x h.c

$ r2 -d ls

> dx e900000000488d3516000000bf01000000b80400000248c7c206..

$ ragg2 -B cc -x
```

Remote Debugging

R2 supports WINDBG, GDB and native remote protocols. But, as long as r2 runs everywhere it is recommended to use it in place.

For example:

```
$ lldbserver /bin/ls
$ r2 -d gdb://localhost:7363/
```

ESIL

ESIL stands for Evaluable Strings Intermediate Language.

A forth-like language (stack based language) using comma as a tokenizer and used for emulating and analyzing code.

Widely used for decrypting malware routines and analyzing shellcodes and other payloads.

```
mov eax, 33       =>       33,eax,=
```

ESIL

The anal plugins provide an esil expression for every instruction that represents what it is doing internally.

This way it is possible to emulate an instruction and get some metrics out of it:

- Which registers are read, or write
- Which memory is accessed
- It is modifying the stack?
- Branch prediction
- ...

ESIL

Esil can be also used to construct search keyword or rules.

And even used with the debugger for assisted and prediction of conditional branches.

Also helpful for software watchpoints emulated with steps + esil emulation to stop before executing the offending instruction.

(DEMO)

ESIL: Demo

Solve a crackme by emulating a function that decrypts a password in memory.

r2frida

Frida + Radare2

FRIDA is an in-process dynamic tracer scriptable with Javascript.

R2FRIDA allows to talk to a frida-server or any process to read/write memory and inject code at runtime.

R2Frida: Installation

Developed by oleavr, mrmacete and pancake (me) at NowSecure.

Provides a handy repl (r2 shell) to use Frida without having to write code snippets. Because it implements the most common actions as r2 commands accessible via the io.cmd interface.

IO plugins can open/read/write/close but also cmd(), this allows the user to talk directly to the io plugin by using the \ or =! Prefix.

```
$ r2pm -ci r2frida
```

R2frida: DEMO

Twitter 280 chars :D

Questions?

\o.

EOF