Radare from A to Z

extended edition

pancake // r2con2016

@trufae

Introduction

What Am I Doing Here?

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

Why Radare2?

- It’s free and opensource
- Runs everywhere (Windows, Mac, Android, Linux, QNX, iOS, ..)
- Easy to script and extend with plugins
- Embeddable
- Grows fast
- Supports tons of file-formats
- Handles gazillions of architectures
- Easy to hack
- Commandline cowboy-friendly
- Great community and even better leader
- Collaborative

What is Radare2?

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

History

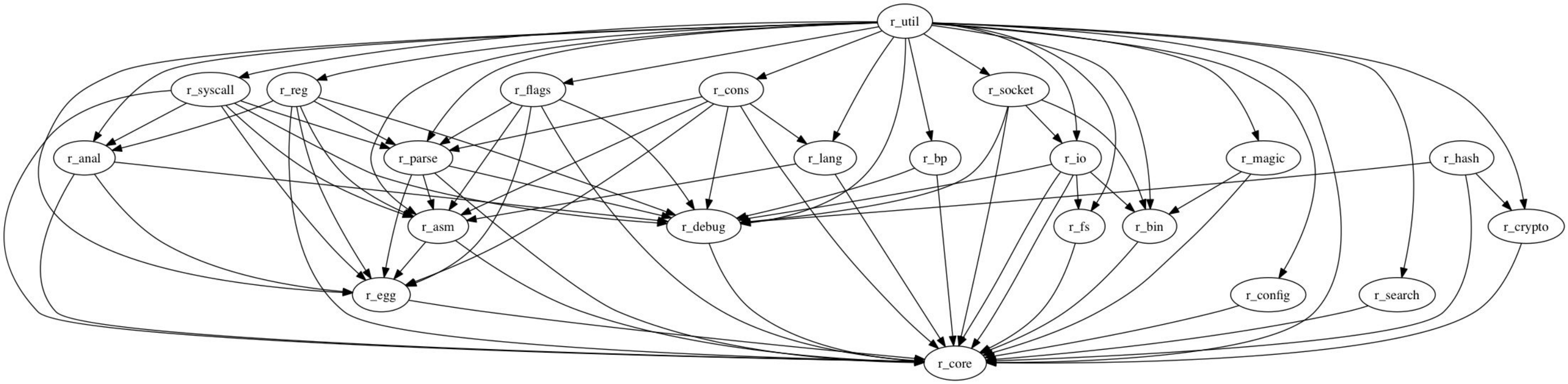
Radare was born in 2006 (hey this is 10 years!) as a forensic tool for performing manual and interactive carving to recover some files from disk or ram.

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

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

The framework

```
$ make depgraph.png
```



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 analize and emulate to identify functions and refs

RUtil provide common helper functions

REgg generate shellcodes ready to be injected

RDiff find differences between two sources

RCore uses them all!

Tools

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

radare2

rabin2

rasm2

r2pm

radiff2

rafind2

rarun2

rax2

r2agent

ragg2

rahash2

rasign2

Tools

- Show how code is organized in r2
- Quick example of every tool.
- Use rax2, rabin2, rasm2, ...
- Manpages

(DEMO)

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
- Firmwares
- Filesystems
- Raw memory dumps

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 |
- Use of redirections >
- Use of backticks ‘
- Internal filtering ~
- Abstracted io to handle everything as a file
- pipeable from the shell
- Textual representations
- Simple command structure
- Autodocumented (man, ?)
- Almost a posix shell with ls, cp, cat, ed..)

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)

PROTIP: Installing radare2 is the recommended method to use it.

How To Install Radare2

There are several binary distributions of radare2

- LiveCD (unmaintained)
- 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)
- Android app and Cydia package (iOS)
- Chat with the @r2bot on Telegram

Coming soon: PPA/Windows from Travis/AppVeyour

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, but risky on a shared machine)

```
$ sys/user.sh
```

Installing from Git

Notice that r2 build system is based on:

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

Plugins can be chosen with ./configure-plugins

- Random documentation in doc/ directory
- Several useful scripts in sys/

Package Management

We can 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

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

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..

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 by default are done in the target
- Specify different permissions to each section (rwx)

Use the `o` command to manage the files.

In The Shell

Syntax of the commands:

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

```
> 3x              # perform 3 hexdumps
> 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
```

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
```

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 hexpairs
px	print hexdump
pxw/pxq	dword/qword dump
pxr	print references
pxe	emojis

Structures

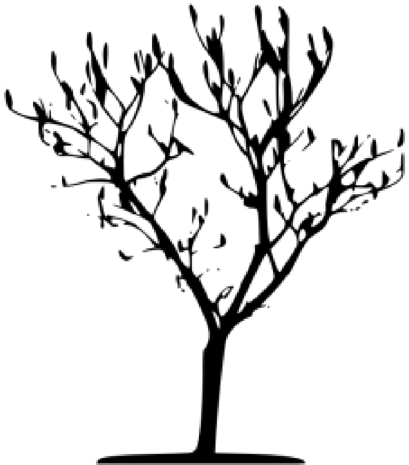
pf - define function signatures

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.

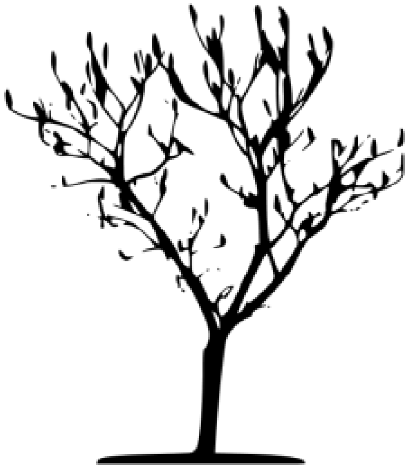
Use pxa to visualize them in colorized hexdump.



Structures

(DEMO)

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



Disassembling

(and printing bytes)

Disassembling
is the “art” of
translating bytes into
meaningful instructions.

rasm2

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

```
$ rasm2 -L

$ rasm2 -a x86 -b 64 nop
```

(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 and

> **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
- wxf

DEMO: patch simple crackme program

Dumping and Restoring

Dump to file

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

```
> wt file 1K
```

```
> y 1K
```

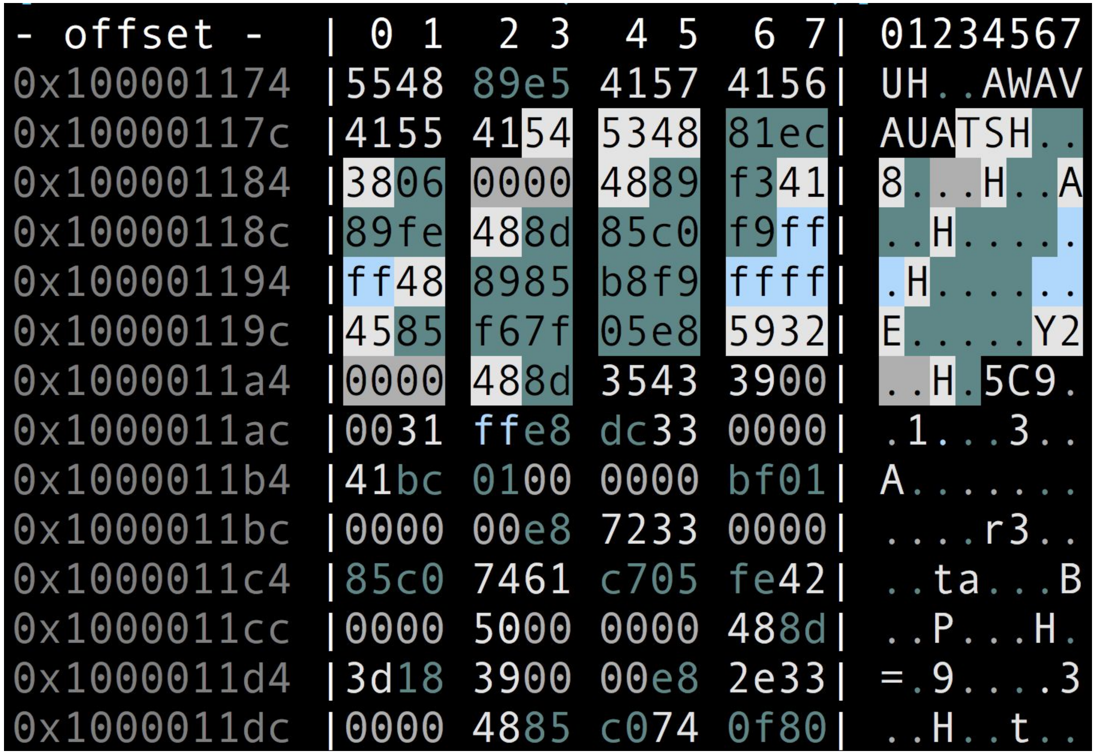
Restoring

```
> 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

There are other disassembler modes in r2.

```
> e asm.emustr=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	unmaintained
Snowman	planned
Retdec	supported, but online
Radeco	wip (gsoc)

Decompiler Demo

(DEMO)

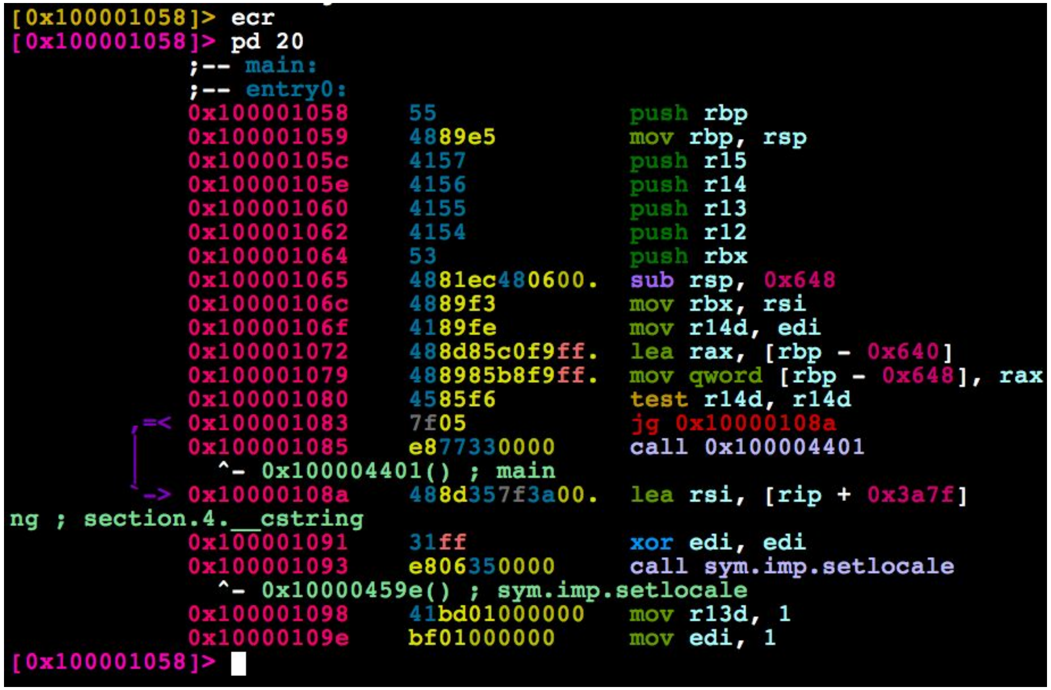
- Use retdec to decompile functions of a program
- Done by nighterman

User Interface

- WebUI
- Bokken
- Visual Mode
- Visual Panels
- Commandline
- R2Pipe
- Colors!

Colors!

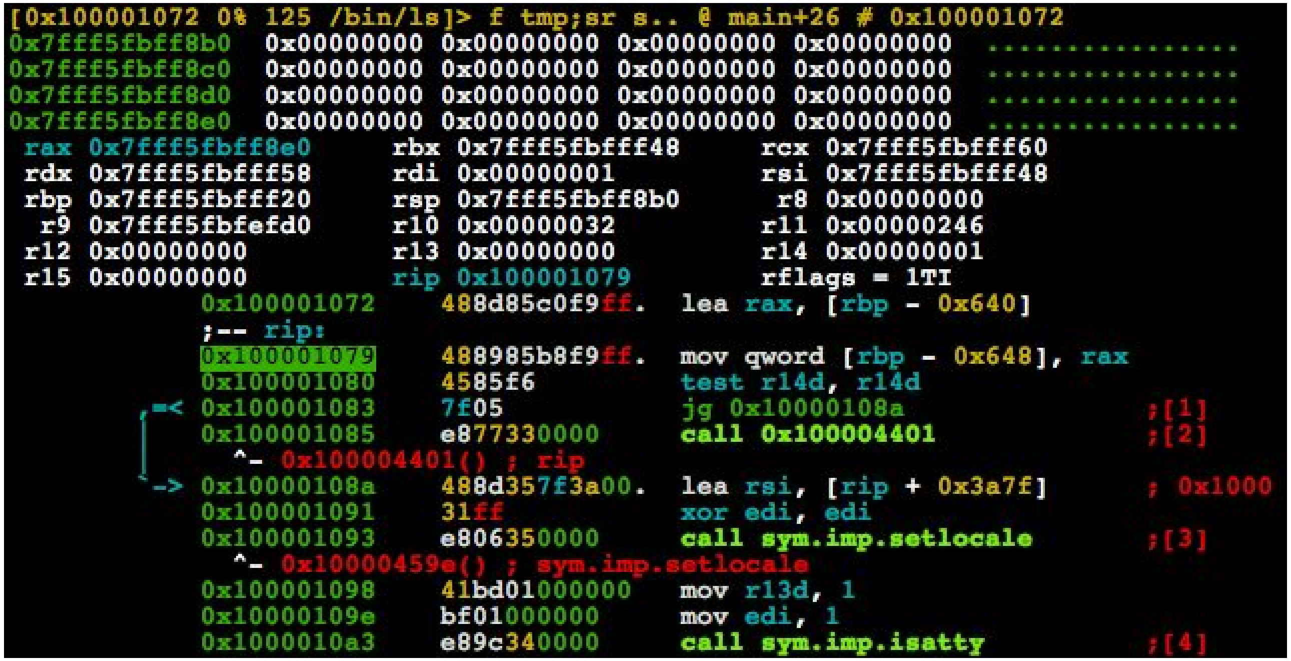
```
> e scr.color=true

> 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]> 
```

Visual Mode

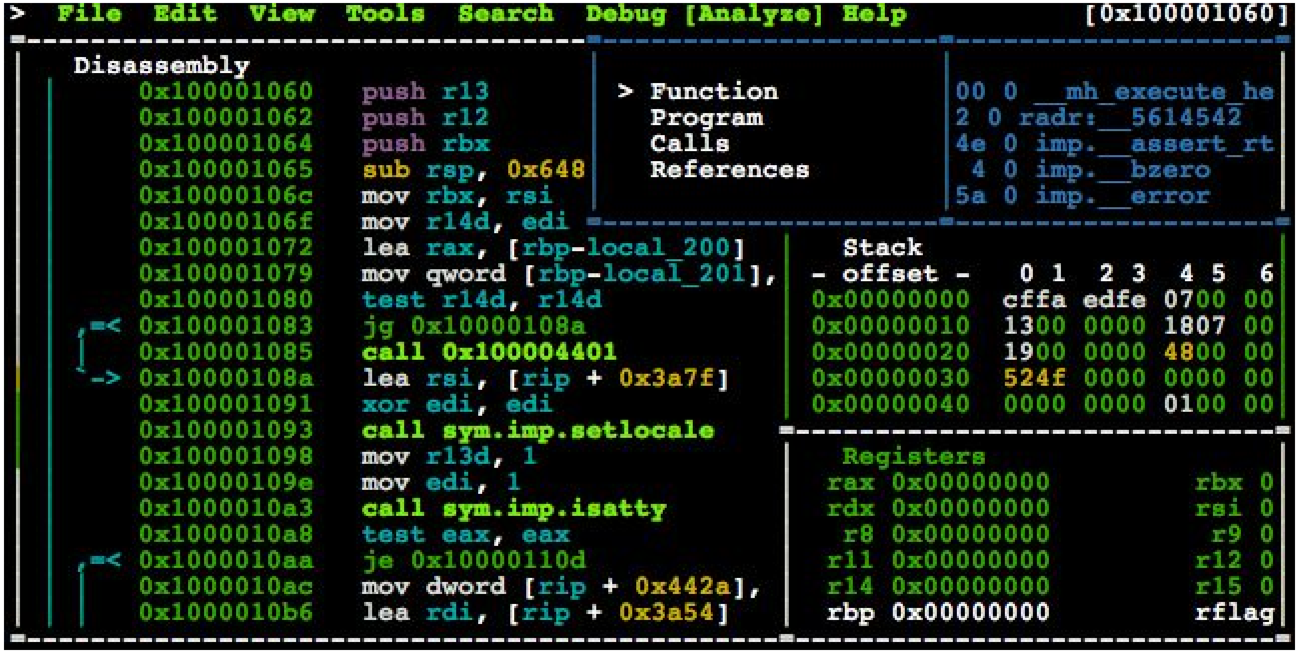
Type V and then change the view with ‘p’ and ‘P’



```
[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



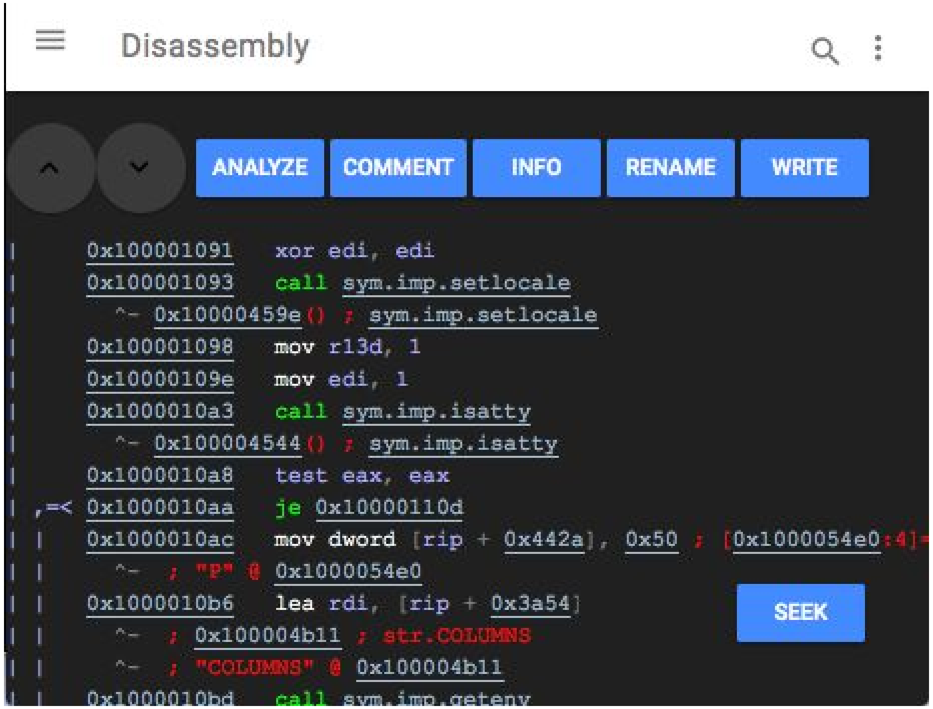
```
>  File  Edit  View  Tools  Search  Debug [Analyze] Help            [0x100001060]
    Disassembly
        0x100001060    push r13               > Function       00 0 __mh_execute_he
        0x100001062    push r12                 Program        2 0 radr:__5614542
        0x100001064    push rbx                 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



```
|     0x100001091   xor edi, edi
|     0x100001093   call sym.imp.setlocale
|       ^- 0x10000459e() ; sym.imp.setlocale
|     0x100001098   mov r13d, 1
|     0x10000109e   mov edi, 1
|     0x1000010a3   call sym.imp.isatty
|       ^- 0x100004544() ; sym.imp.isatty
|     0x1000010a8   test eax, eax
| ,=< 0x1000010aa   je 0x10000110d
| |   0x1000010ac   mov dword [rip + 0x442a], 0x50 ; [0x1000054e0:4]=
| |     ^-  ; "P" @ 0x1000054e0
| |   0x1000010b6   lea rdi, [rip + 0x3a54]
| |     ^-  ; 0x100004b11 ; str.COLUMNS
| |     ^-  ; "COLUMNS" @ 0x100004b11
| |   0x1000010bd   call sym.imp.getenv
```

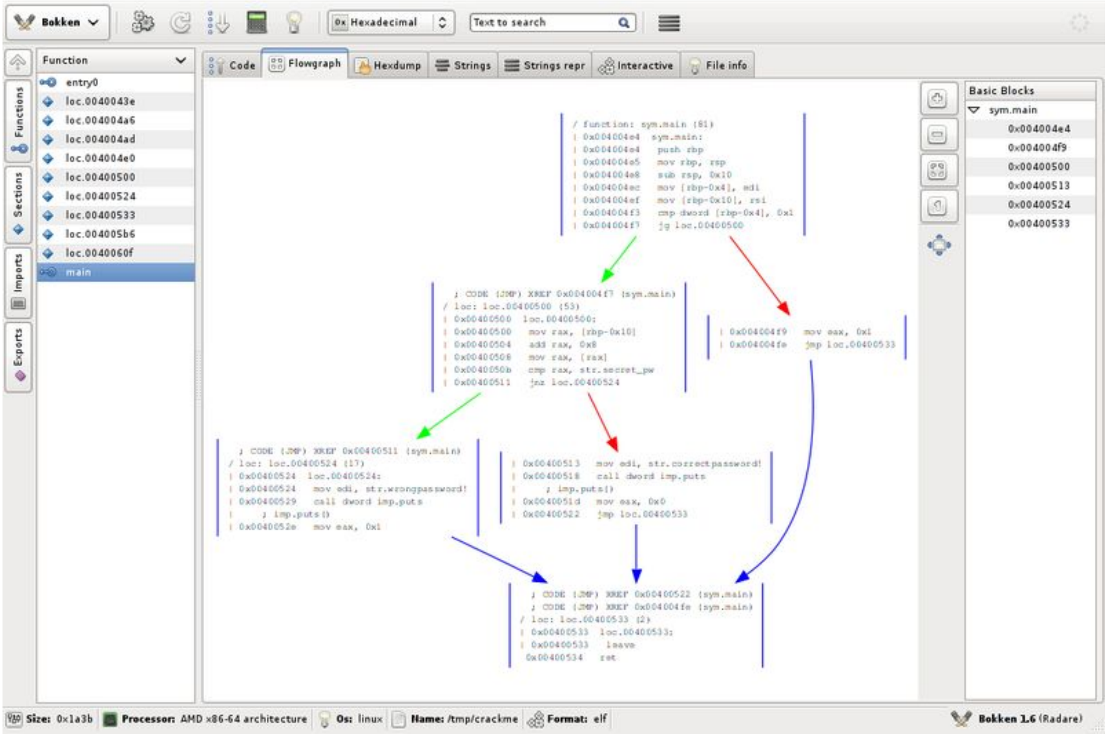
Bokken

Native Python/Gtk GUI

Binaries for Windows

Runs on OSX/Linux too

Author: Hugo Teso



```
/ function: sym.main (81)
| 0x004004e4  sym.main:
| 0x004004e4   push rbp
| 0x004004e5   mov rbp, rsp
| 0x004004e8   sub rsp, 0x10
| 0x004004ec   mov [rbp-0x4], edi
| 0x004004ef   mov [rbp-0x10], rsi
| 0x004004f3   cmp dword [rbp-0x4], 0x1
| 0x004004f7   jg loc.00400500
```

```
  ; CODE (JMP) XREF 0x004004f7 (sym.main)
/ loc: loc.00400500 (53)
| 0x00400500  loc.00400500:
| 0x00400500   mov rax, [rbp-0x10]
| 0x00400504   add rax, 0x8
| 0x00400508   mov rax, [rax]
| 0x0040050b   cmp rax, str.secret_pw
| 0x00400511   jnz loc.00400524
```

```
| 0x004004f9   mov eax, 0x1
| 0x004004fe   jmp loc.00400533
```

```
  ; CODE (JMP) XREF 0x00400511 (sym.main)
/ loc: loc.00400524 (17)
| 0x00400524  loc.00400524:
| 0x00400524   mov edi, str.wrongpassword!
| 0x00400529   call dword imp.puts
|      ; imp.puts()
| 0x0040052e   mov eax, 0x1
```

```
| 0x00400513   mov edi, str.correctpassword!
| 0x00400518   call dword imp.puts
|      ; imp.puts()
| 0x0040051d   mov eax, 0x0
| 0x00400522   jmp loc.00400533
```

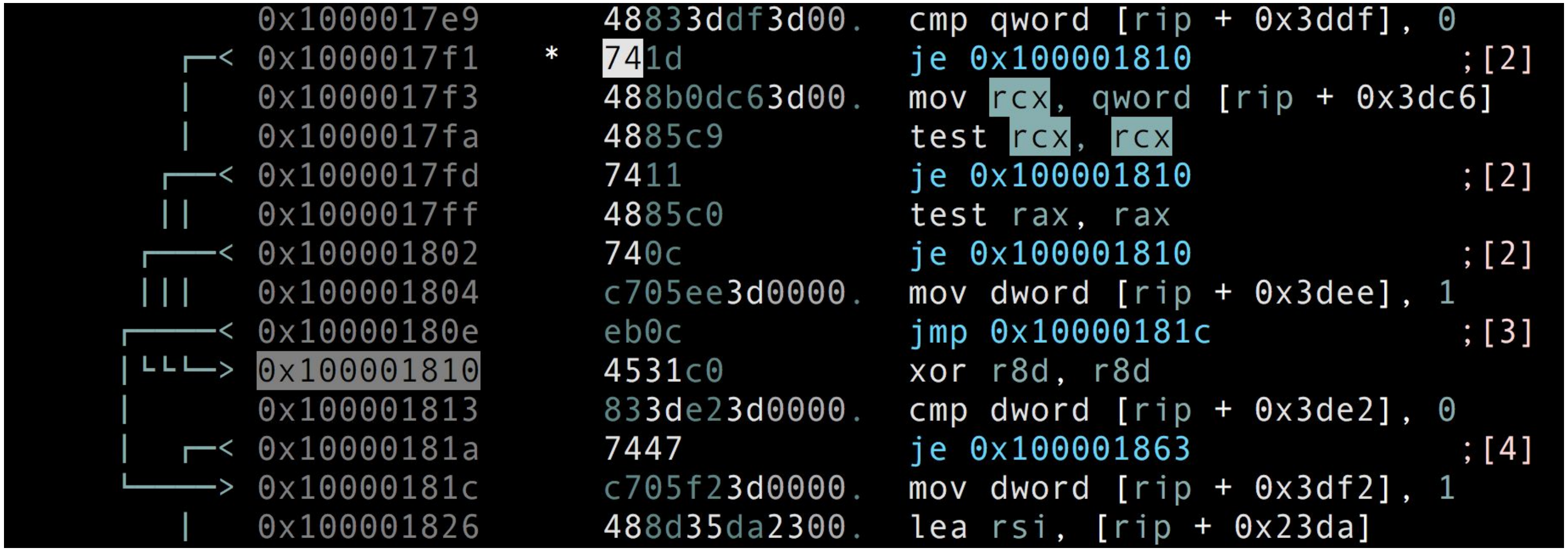
```
  ; CODE (JMP) XREF 0x00400522 (sym.main)
  ; CODE (JMP) XREF 0x004004fe (sym.main)
/ loc: loc.00400533 (2)
| 0x00400533  loc.00400533:
| 0x00400533   leave
 0x00400534   ret
```

Size: 0x1a3b | Processor: AMD x86-64 architecture | Os: linux | Name: /tmp/crackme | Format: elf

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

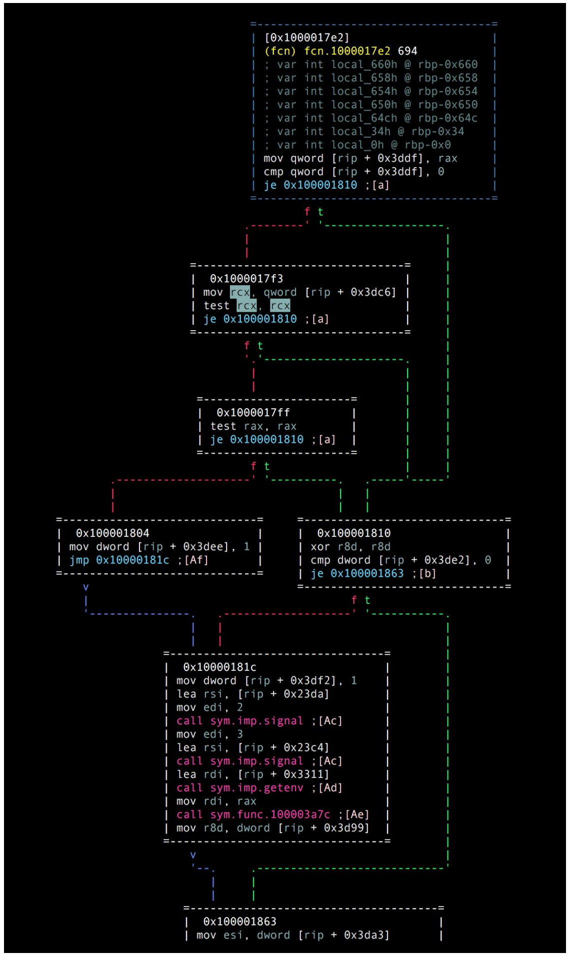


```
        0x1000017e9         48833ddf3d00.  cmp qword [rip + 0x3ddf], 0
   ┌─<  0x1000017f1    *    741d           je 0x100001810                ;[2]
   │    0x1000017f3         488b0dc63d00.  mov rcx, qword [rip + 0x3dc6]
   │    0x1000017fa         4885c9         test rcx, rcx
  ┌──<  0x1000017fd         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)
- HUD (V_)
- Resize Hexdump with []
- Add comments (;)
- Undo/Redo seek (u/U)
- Find next/prev hit/func/.. With n/N
- Basic Block Graphs

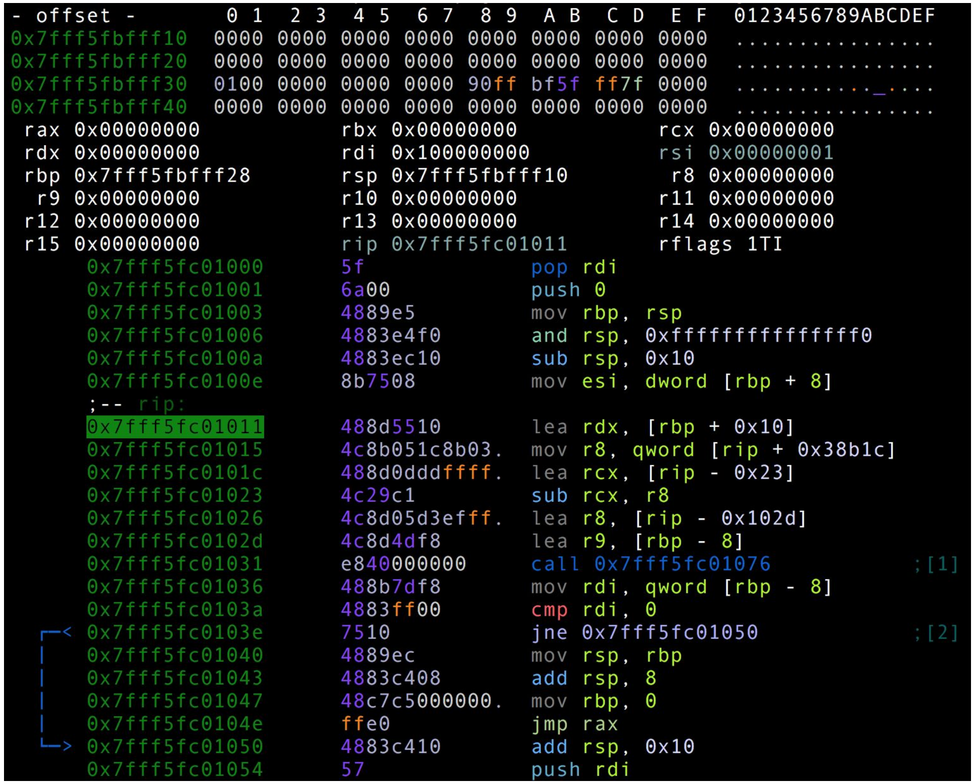


```
                             =-------------------------------------=
                             | [0x1000017e2]                       |
                             | (fcn) fcn.1000017e2 694             |
                             | ; var int local_660h @ rbp-0x660    |
                             | ; var int local_658h @ rbp-0x658    |
                             | ; var int local_654h @ rbp-0x654    |
                             | ; var int local_650h @ rbp-0x650    |
                             | ; var int local_64ch @ rbp-0x64c    |
                             | ; var int local_34h @ rbp-0x34      |
                             | ; var int local_0h @ rbp-0x0        |
                             | mov qword [rip + 0x3ddf], rax       |
                             | cmp qword [rip + 0x3ddf], 0         |
                             | je 0x100001810 ;[a]                 |
                             =-------------------------------------=
                                       f t
                             .--------' '-------------------.
                             |                              |
                             |                              |
                    =--------------------------------=      |
                    |  0x1000017f3                   |      |
                    | mov rcx, qword [rip + 0x3dc6]  |      |
                    | test rcx, rcx                  |      |
                    | je 0x100001810 ;[a]            |      |
                    =--------------------------------=      |
                              f t                           |
                              '.'--------------------.      |
                               |                     |      |
                               |                     |      |
                     =-----------------------=       |      |
                     |  0x1000017ff          |       |      |
                     | test rax, rax         |       |      |
                     | je 0x100001810 ;[a]   |       |      |
                     =-----------------------=       |      |
                               f t                   |      |
          .-------------------' '--------.      .----'------'
          |                              |      |
          |                              |      |
=-------------------------------=    =----------------------------------=
|  0x100001804                  |    |  0x100001810                     |
| mov dword [rip + 0x3dee], 1   |    | xor r8d, r8d                     |
| jmp 0x10000181c ;[Af]         |    | cmp dword [rip + 0x3de2], 0      |
=-------------------------------=    | je 0x100001863 ;[b]              |
    v                                =----------------------------------=
    |                                           f t
    '---------------.  .-----------------------' '------------.
                    |  |                                      |
                    |  |                                      |
              =---------------------------------=             |
              |  0x10000181c                    |             |
              | mov dword [rip + 0x3df2], 1     |             |
              | lea rsi, [rip + 0x23da]         |             |
              | mov edi, 2                      |             |
              | call sym.imp.signal ;[Ac]       |             |
              | mov edi, 3                      |             |
              | lea rsi, [rip + 0x23c4]         |             |
              | call sym.imp.signal ;[Ac]       |             |
              | lea rdi, [rip + 0x3311]         |             |
              | call sym.imp.getenv ;[Ad]       |             |
              | mov rdi, rax                    |             |
              | call sym.func.100003a7c ;[Ae]   |             |
              | mov r8d, dword [rip + 0x3d99]   |             |
              =---------------------------------=             |
                  v                                           |
                  '--.     .----------------------------------'
                     |     |
                     |     |
                 =---------------------------------------=
                 |  0x100001863                          |
                 | mov esi, dword [rip + 0x3da3]         |
```

Deeper Look into the Visual Mode

Debugging

- Debugger integration
  - Seek to PC (.)
  - Step (s) or StepOver (S)
  - Set breakpoints with ‘b’
- Change stack settings



```
- 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    488d0ddffff.    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

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

Imports

Sections

Relocs

Strings

Libraries

Classes

Demangling

SourceLines

Entrypoints

Exports

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)

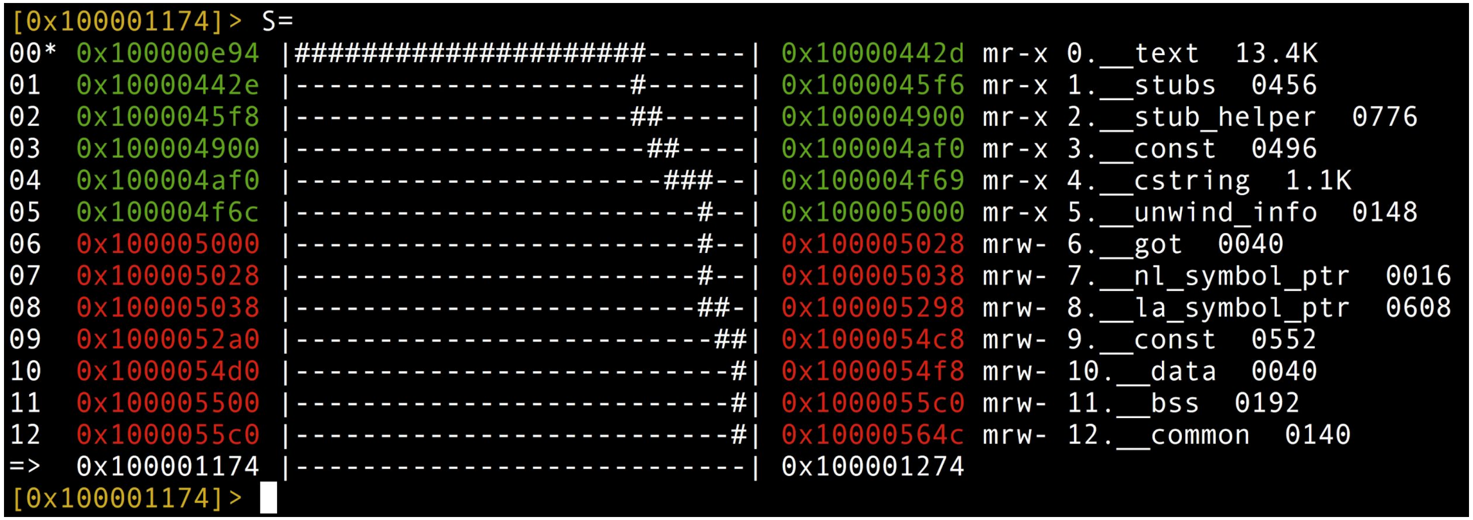
The information of classes and methods can befound in:

- 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**
- ...

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)

The art of 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

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 is the “art” of understanding the purpose of a sequence of instructions.

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**

Low Level Anal Tweaks

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

```
> ahs 1 @ 0x100001175
```

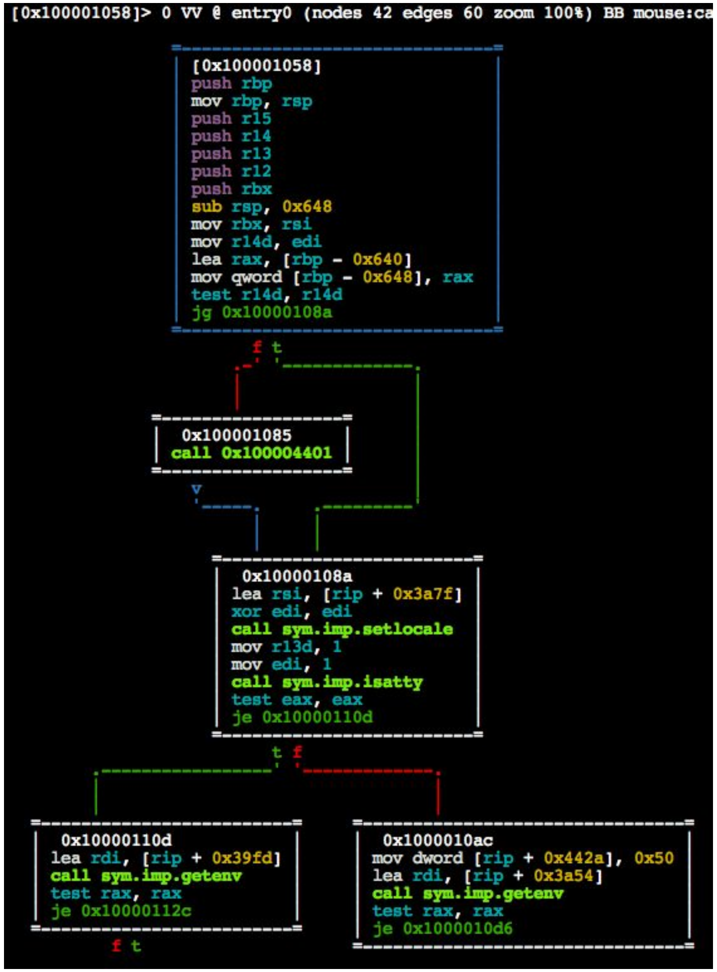
(DEMO) Jump in the middle of instruction

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
    - Overview

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.

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/ $$
```

Future Signatures

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

Ideally the signatures should support graph metrics matching and intermediate language reductions.

- 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
```

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**

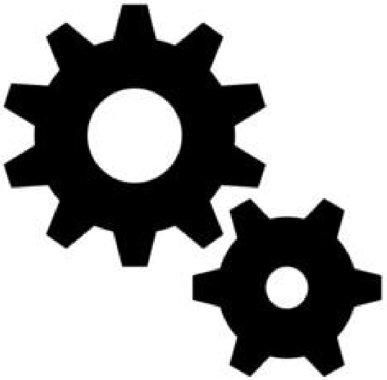
...

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 filedescriptors 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 filedescriptor
- 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)

Development

Development

Show how to find easy bugs in github

How code is structured

Rebuilding for testing doesn’t requires a full rebuild or reinstall. Just type make in the working directory.

- sys/install.sh installs with symlinks
- env.sh to run from build directory
- sys/user.sh to install at user’s home

But First.. A Poll!

(who are you?)

How many know git?

Do you know C?

Do you GDB/LLDB?

Be social!

R2 is WYSIWYF

What You See Is What You Fix.

Many times, bugs in r2 can be identified and fixed faster than opening a browser and describing the issue.

- Learn to use valgrind, asan, lldb/gdb, eprintf
- Report in a proper way
  - Provide reproducer (files, commands, steps)
  - Paste version number (we only fix bugs in master)
  - Show backtraces, valgrind logs, register states, disasm..

Debugging The Bug

Once a crash is found, we first need to identify where is this happening.

- Grep for previous messages
- Analyze the backtrace and variable values
- See register values and offending asm instruction
- Add eprintf messages around the lines to see whats going on

Identify The Kind Of Bug

- Overflow (valgrind/asan will help a lot)
- Null Dereference (memreads to <1000 usually are)
- DoubleFree (MALLOC_OPTIONS=g / valgrind)
- Format String
- Logic bugs (missbehaviour)
- Integer Overflows (negative or huge values)
- Heap exhaustion

Using Grep

Ctags is a nice solution but needs to be regenerated every time we change the code, grep may find where is the code you are looking for faster. A consistent coding style matters.

```
$ git grep  # faster than system’s grep only in repo files
```

^R_API	get public apis
StructName;	typedef definition of that struct
^static	find static methods
fcn(	function definition
fcn (	function calls
“cmd”	where is this r2 command implemented
%llx	find non-portable code
XXX	stuff known to be wrong
TODO	things to be done

Using Git

At first Git is a bit overwhelming for all the new commands that must be learned, but once passed, you will memorize the most common commands.

- **tig**
- **git diff**
- **git cherry-pick**
- **git clean -xdf**
- **git reset --hard @^^**
- **git clone ..**
- **git checkout -b branchname**
- **git add**
- **git commit -a -m …**

Squashing and Rebasing

Merging from master branch while developing in a separate branch is a bad idea because it makes the history messy.

The way to properly rebase your commits into master is:

```
$ git log | grep ^commit | cut -d ‘ ‘ -f 2 > commits

$ git reset --hard @^^^

$ git pull master

$ for a in `tac commits` ; do git cherry-pick $a ; done
```

Pull requests

The way to contribute in Github is via a pull request, which triggers Travis and AppVeyour services to do new builds and run the whole testsuite to find any regression.

- Allows to do commenting in code review.
- Can be done from the web clicking the pencil button

Merge / **Squash**

Regressions Testsuite

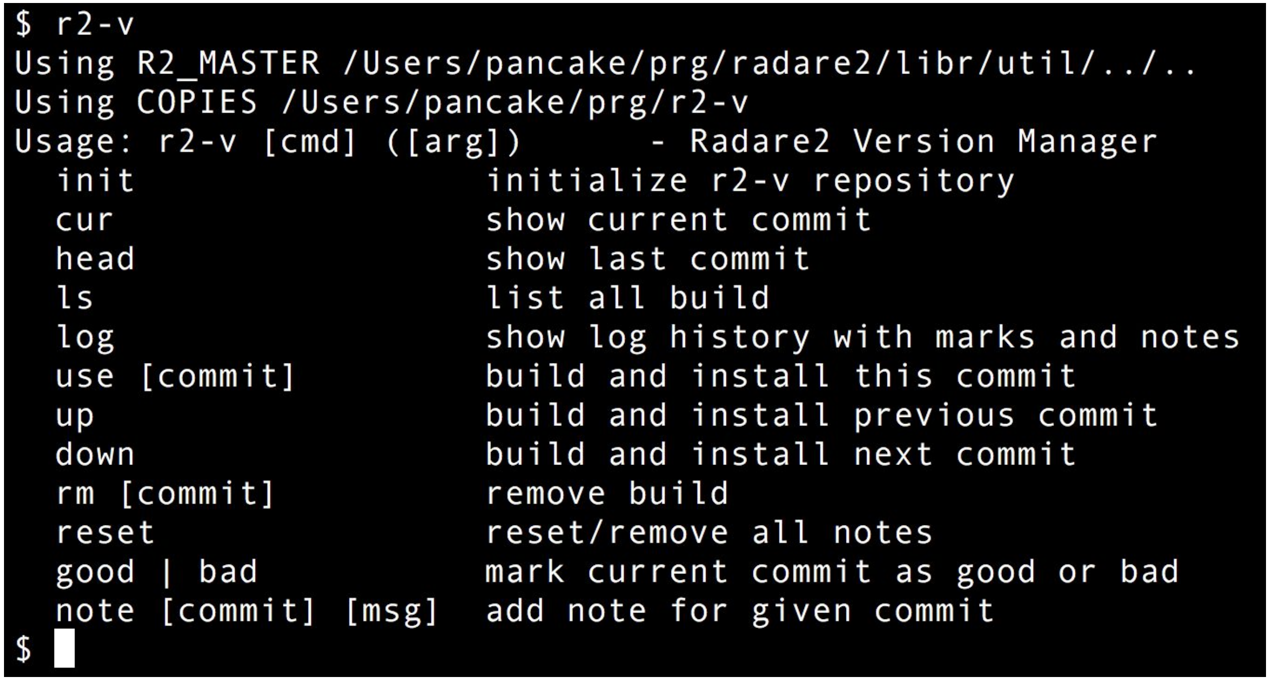
The testsuite is a trivial piece of radare2 that serves as a way to ensure that the behaviour is stable and is not segfaulting.

- Test fuzzed binaries
- Useful for other projects
- Assemble/disassemble instructions
- Run commands and compare output
- The Unit testsuite verifies the C apis

Finding Regression Point

Build every commit of r2 until it is able to run a test script without errors.

$ r2-v



```
$ r2-v
Using R2_MASTER /Users/pancake/prg/radare2/libr/util/../..
Using COPIES /Users/pancake/prg/r2-v
Usage: r2-v [cmd] ([arg])      - Radare2 Version Manager
  init                    initialize r2-v repository
  cur                     show current commit
  head                    show last commit
  ls                      list all build
  log                     show log history with marks and notes
  use [commit]            build and install this commit
  up                      build and install previous commit
  down                    build and install next commit
  rm [commit]             remove build
  reset                   reset/remove all notes
  good | bad              mark current commit as good or bad
  note [commit] [msg]     add note for given commit
$ 
```

Your First Bugfix

Check for the “easy” label in github. One of them would be to fix esil expressions. Let’s check this out:

```
$ r2 /bin/ls

> aae

0x1000042a8 esil_eq: invalid parameters

> s 0x1000042a8

> ao~esil

esil: r15,4,*,,r14d,=
```

Writing A Test

(DEMO)

- Edit any file under t* directories in the r2r repo
- r2r stands for radare2-regressions
- Add a demo test.

Congratulations!
You Are Now A DevRev!

Questions?

\o.

EOF