R2 Basics

Maxime Morin

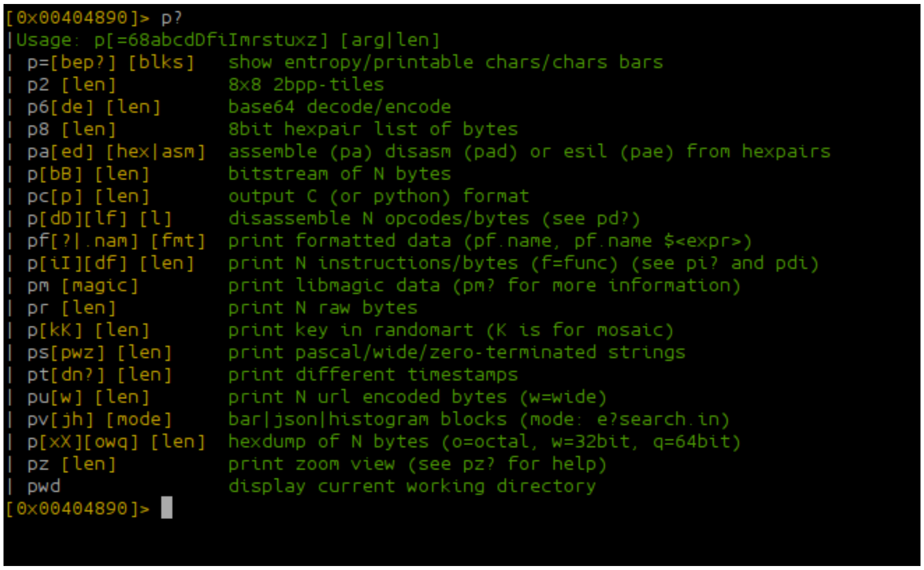
2014-10-22

GroundWork

r2 basics

Each command is associated to a single letter. The rest are subcommands.

- `p?` => print display help on print commands)
- `pd` => print disassembly of the current block size
- `pd?` => print disassembly help
- `pdf` => print disassembly of a function



```
[0x00404890]> p?
|Usage: p[=68abcdDfiImrstuxz] [arg|len]
| p=[bep?] [blks]   show entropy/printable chars/chars bars
| p2 [len]          8x8 2bpp-tiles
| p6[de] [len]      base64 decode/encode
| p8 [len]          8bit hexpair list of bytes
| pa[ed] [hex|asm]  assemble (pa) disasm (pad) or esil (pae) from hexpairs
| p[bB] [len]       bitstream of N bytes
| pc[p] [len]       output C (or python) format
| p[dD][lf] [l]     disassemble N opcodes/bytes (see pd?)
| pf[?|.nam] [fmt]  print formatted data (pf.name, pf.name $<expr>)
| p[iI][df] [len]   print N instructions/bytes (f=func) (see pi? and pdi)
| pm [magic]        print libmagic data (pm? for more information)
| pr [len]          print N raw bytes
| p[kK] [len]       print key in randomart (K is for mosaic)
| ps[pwz] [len]     print pascal/wide/zero-terminated strings
| pt[dn?] [len]     print different timestamps
| pu[w] [len]       print N url encoded bytes (w=wide)
| pv[jh] [mode]     bar|json|histogram blocks (mode: e?search.in)
| p[xX][owq] [len]  hexdump of N bytes (o=octal, w=32bit, q=64bit)
| pz [len]          print zoom view (see pz? for help)
| pwd               display current working directory
[0x00404890]>
```

Figure 1: p? => print display help on print commands)

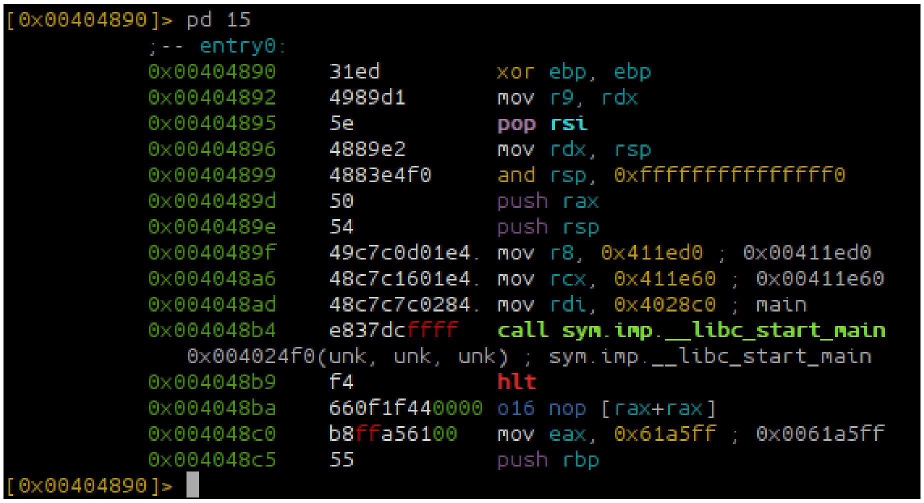


Figure 2: pd => print disassembly of the current block size

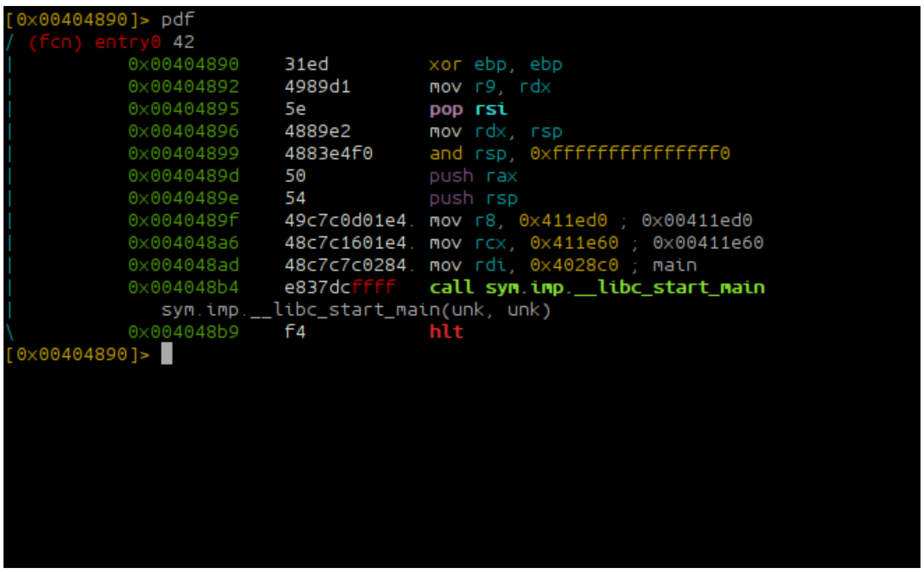


Figure 3: pdf => print disassembly of a function

To get help on any command just append `?` example:

```
[0x00404890]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| wc                list all write changes
| w[1248][+-][n]    increment/decrement byte,word..
| w foobar          write string 'foobar'
| wh r2             whereis/which shell command
| wr 10             write 10 random bytes
| ww foobar         write wide string 'f\x00o\x00o\x00b\x00a\x00r\x00'
| wa push ebp       write opcode, separated by ';' (use '"' around the command)
| waf file          assemble file and write bytes
| wA r 0            alter/modify opcode at current seek (see wA?)
| wb 010203         fill current block with cyclic hexpairs
| wc[ir*?]          write cache undo/commit/reset/list (io.cache)
| wd [off] [n]      duplicate N bytes from offset at current seek (memcpy) (see y?
)
| wx 9090           write two intel nops
| wv eip+34         write 32-64 bit value
| wo? hex           write in block with operation. 'wo?' fmi
| wm f0ff           set binary mask hexpair to be used as cyclic write mask
| ws pstring        write 1 byte for length and then the string
| wf -|file         write contents of file at current offset
| wF -|file         write contents of hexpairs file here
| wp -|file         apply radare patch file. See wp? fmi
| wt file [sz]      write to file (from current seek, blocksize or sz bytes)
[0x00404890]>
```

Figure 4: w? => display help on print commands

More help type `?` to get the main help:

- `man radare2`, `radare2 -h` (same with the other tools)
- `???`: Help on Expressions
- `?$?`: Help on Variables
- `?@?`: Help on Offset

Hashing: Fingerprint for a sample (#)

Hashing is a common method used to uniquely identify malware. The malicious software is run through a hashing program that produces a unique hash that identifies that Malware (like a fingerprint). MD5, SHA1, SHA512 are the most commonly used. The fingerprint will be used for research and sharing instead of sharing the binary. It can also be used for researching over the Internet to see if the file has already been identified.

To calculate the hash of a program you can either use `r2` or the stand-alone program `rahash2`

Rahash2

- Display list of algorithm available rahash2 -L
- Calculate the sha1 rahash2 -a sha1 program.exe

Radare2

- Display list of algorithm available [0x00404888]>##
- Calculate the sha1 `[0x00404888]>#sha1 $s @ 0 // Compute md5 (#md5) of size of file ($s) at offset 0`

Quick strings fetching

A string in a file is a sequence of characters such as “Abracadabra!”. Searching through the strings can give some information about the functionality of a program.

To quickly display the strings contained in a binary you can use either r2 or rabin2: which is the dedicated command to get information about binaries.

- Rabin2
    - Display strings inside .data section (like gnu strings does) `rabin2 -z file`

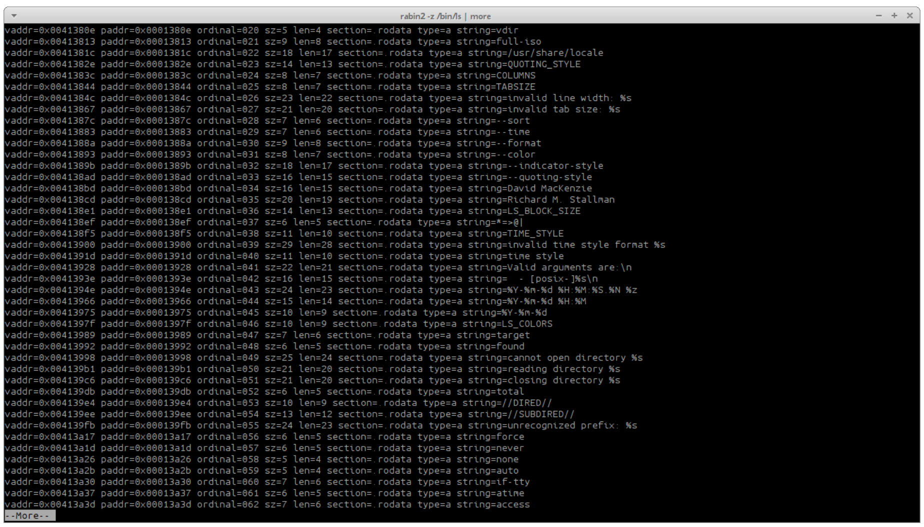


Figure 5: iz

    - Display strings from raw bins `rabin2 -zz file`

- Append `j` to get the result in json format!

```
rabin2 -zj file
```

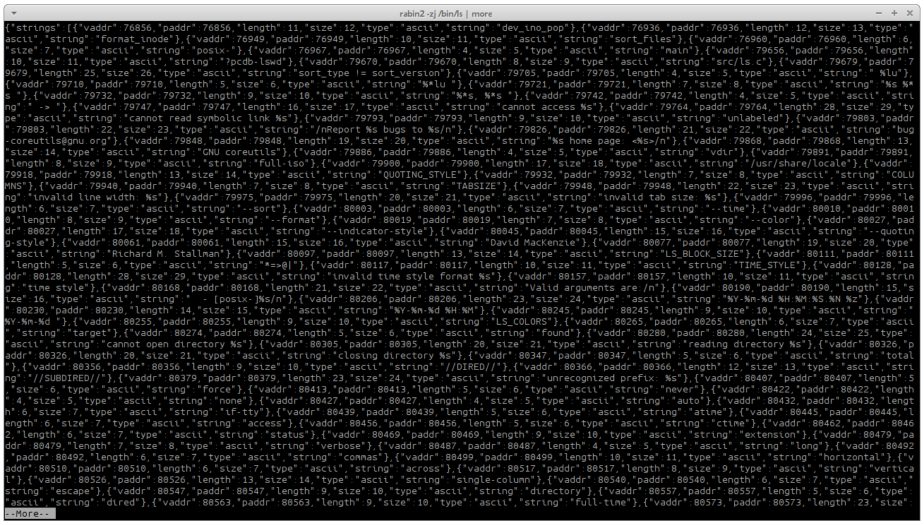


Figure 6: izj

- Radare2
  - Display all strings in r2 `[0x00404888]>izz`

Rabin2 and Radare2 can display both ASCII and Unicode strings: See `type=a` or `type=u`.

Suffix:

```
~ or grep grep/cut interno
|  pipe to program
>  pipe to file
>> concat to a
@  temporal seek
@@ iterator
*  output in commands
j  output in json
?  help
```

Get information about a binary (i?)

We've seen how to parse a binary or any file format to modify or retrieve information at a low-level. You can also retrieve information using info command `i?`:

- Get General information about the binary: `iI` // `rabin2 -I`
- Get Header information `ih` // `rabin2 -H`
- Get Imports: `ii` // `rabin2 -i`
- Get Entrypoints: `ie` // `rabin2 -e`
- Get Exports: `is` // `rabin2 -s`
- Get Relocs: `iR` // `rabin2 -R`
- Get Sections: `iS` // `rabin2 -S`

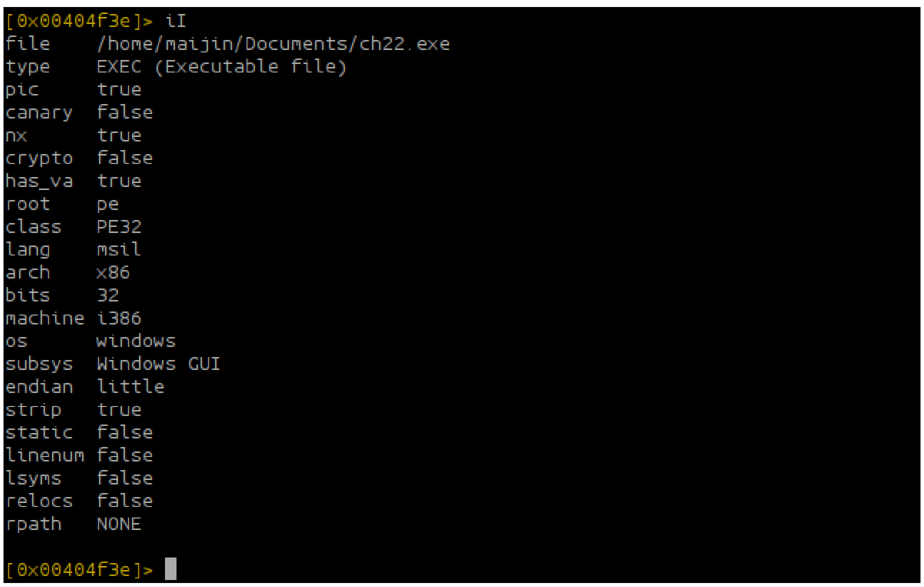


Figure 7: Get General information about the binary: iI/Rabin2 -I

Parse a File format

File Format definition:

> A file format is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in a digital storage medium.

Portable Executable definition:

> The Portable Executable (PE) format is a file format for executable files, object code, DLLs,(. . .) used by Windows operating systems.

- PE101 by Corkami
- Portable Executable header

Pf: print formatted data

```
Usage: pf[.key[.field[=value]]|[ val]]|[times][ [size] format] [arg0 arg1 ...]
Examples:
pf 10xiz pointer length string
pf {array_size}b @ array_base
pf [4]w[7]i     # like pf w..i... pf.            # list all formats
pf.obj xxdz prev next size name
pf.obj          # run stored format
pf.obj.name     # show string inside object
pf.obj.size=33  # set new size
Format chars:
e - temporally swap endian
f - float value (4 bytes)
b - byte (unsigned)
B - resolve enum bitfield (see t?) `pf B (Bitfield_type)arg_name`
c - char (signed byte)
E - resolve enum name (see t?) `pf E (Enum_type)arg_name`
X - show n hexpairs (default n=1) i - %i integer value (4 bytes)
w - word (2 bytes unsigned short in hex)
q - quadword (8 bytes)
p - pointer reference (2, 4 or 8 bytes)
T - show Ten first bytes of buffer
d - 0x%08x hexadecimal value (4 bytes)
D - disassemble one opcode
o - 0x%08o octal value (4 byte)
x - 0x%08x hexadecimal value and flag (fd @ addr)
```

```
X - show formatted hexpairs
z - \0 terminated string
Z - \0 terminated wide string
s - 32bit pointer to string (4 bytes)
S - 64bit pointer to string (8 bytes)
? - data structure `pf ? (struct_type)struct_name`
* - next char is pointer (honors asm.bits)
+ - toggle show flags for each offset
: - skip 4 bytes
. - skip 1 byte
```

1. Look at the structure defined in .h or any valuable documentation about a file format

   ```
   typedef struct _ IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
   } IMAGE_DOS_HEADER, * PIMAGE_DOS_HEADER;
   ```

2. Convert each component type in pf symbol equivalent, for example first is **WORD e__magic;**: * **WORD** is `w`. * **e__ident** should contain the Magic number: A constant numerical or text value used to identify a file format. In PE, this magic number is a magic text ('MZ'), So we can also display/parse it like a string of size 2 `[2]z`.

   ```
   w e_magic` or `[2]z e_magic
   ```

3. Set this new type in pf just using: `pf.dos_header [2]z e_magic`

To try that new type and parse a pe to retrieve the MZ magic:

1. Open an pe file: `r2 *.exe`
2. Do not forget to set the type: `pf.dos_header [2]z e_magic`
3. Run stored format at offset 0 of the elf file and profit: `pf.dos_header @ 0`
4. Retrieve a single value: pf.dos_header.e_magic @ 0

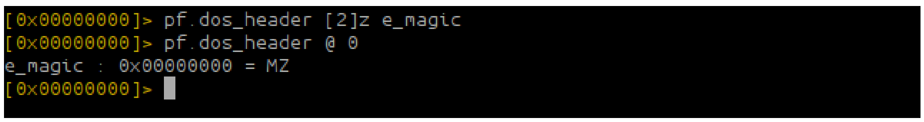


Figure 8: Run stored format at offset 0 of the elf file and profit

The complete dos_header could be done like this:

```
pf.pe_dos_header [2]zwwwwwwwwwwwwww[4]www[10]wx
e_magic e_cblp e_cp e_crlc e_cparhdr e_minalloc
e_maxalloc e_ss e_sp e_csum e_ip e_cs e_lfarlc
e_ovno e_res e_oemid e_oeminfo e_res2 e_lfanew
```

You can contribute on this part to implement PE, ELF and Mach-O in r2. This work has already started for elf and Mach-o, just open a elf file using r2 -nn which means only load the rbin structures and profit.

```
pf.pe_dos_header @ pe_dos_header
pf.pe_nt_image_headers32 @ pe_nt_image_headers32
```

or

```
pf.pe_nt_image_headers64 @ pe_nt_image_headers64
```

Packing

Packer definition:

> Packers are wrappers put around pieces of software to compress and/or encrypt their contents. They can be used by legitimate software to minimise download times and storage space or to protect copyrighted coding, but are commonly used in malware to disguise the contents of malicious files from malware scanners. Runtime packers essentially unpack (i.e. decrypt or decompress) executable files as they run - the first stage is the unwrapping process, and the unpacked file is then loaded into memory and run. A file can be packed numerous times with slight changes to the packing method, or with small and insignificant changes to the file inside, thus producing a final file which appears different from another identical file packed differently.

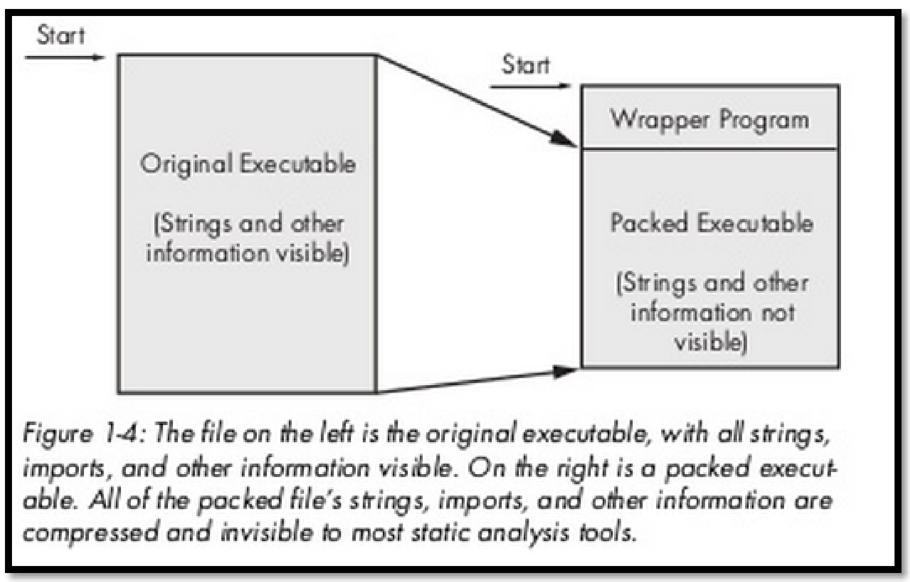


Figure 9: Packer

Detect Packing: Entropy

> Entropy as it relates to digital information is the measurement of randomness in a given set of values (data).

http://www.forensickb.com/2013/03/file-entropy-explained.html

> Entropy can be used is many different way, but quite commonly to detect encryption and compression, since truly random data is not common in typical user data. This is especially true with executable files that have purposely been encrypted with a real-time decryption routine. This prevents an AV engine from seeing "inside" the executable as it sits on the disk in order to detect strings or patterns. It is also very helpful in identifying files that have a high-amount of randomness, which could indicate an encrypted container/volume that may go otherwise unnoticed.

- Entropy of this file using: `#entropy $s @ 0`
- Entropy block by block using: `p=`
- Entropy Section using rabin2: `rabin2 -K entropy -S /bin/ls`

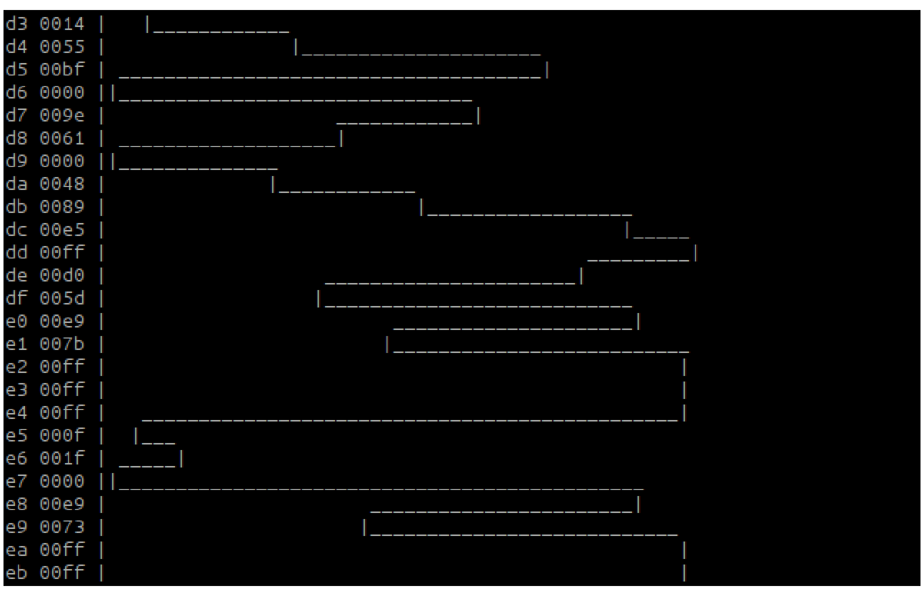


Figure 10: Entropy block by block using: p=

Detect Packing: Yara and signatures

> YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. With YARA you can create descriptions of malware families (or whatever you want to describe) based on textual or binary patterns. Each description, a.k.a rule, consists of a set of strings and a Boolean expression which determine its logic. Let's see an example:

You can apply Yara rule inside r2

To use yara just type: `yara scan`. This command will apply the several rules shipped with radare2 (yara list to display the list of rules). You can use yours using `yara add`

Crypto Algorithm

> Detection of some cryptographic algorithm are implemented:

```
/Ca  Search for AES keys
/Cr  Search for private RSA keys
```

Magic number

> You can search for magic number (constant numerical or text value used to identify a file format or protocol; for files) using `/m`. You can restrict the search from a certain offset using eval variable.

```
e search.from=0 // To set beginning address
e search.to=0x1000 // To set ending address
```

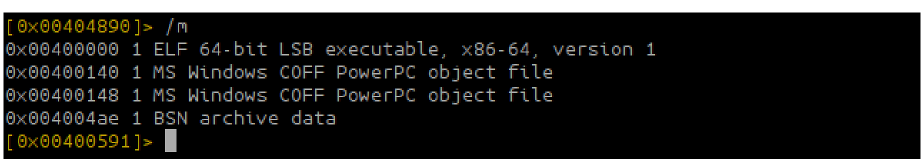


Figure 11: Search for magic using /m

GO GO GO

First steps

To disassemble a program using r2, first open a binary using `r2 file.exe` command.

Radare2 can perform analysis on a binary in order to get function name and so on. You can launch this analysis using `aa` for analyse all or launch the analysis when opening the file directly: `r2 -A file.exe`

Each command is associated to a single letter. The rest are subcommands.

```
px  print hex
pd  print disassembly
pD  print disassembly (takes the number of bytes instead of the number of opcodes.)
pdf print disassembly of a function
pc  output in C
pcp output in Python
afl list functions
axf xref from
axt xref to
s   seek
?d  Describe opcode
wx 9090 write two intel nops
wo? write in block with operation (wox xor, woA and...)
...
```

Basic print commands

One of the key features of radare is displaying information in various formats. The goal is to offer a selection of displaying choices to best interpret binary data.

Binary data can be represented as integers, shorts, longs, floats, timestamps, hexpair strings, or more complex formats like C structures, disassembly, decompilations, external processors, ..

Here's a list of the available print modes listable using `p?`:

```
|Usage: p[=68abcdDfiImrstuxz] [arg|len]
| p=[bep?] [blks]   show entropy/printable chars/chars bars
| p2 [len]          8x8 2bpp-tiles
| p6[de] [len]      base64 decode/encode
| p8 [len]          8bit hexpair list of bytes
| pa[ed] [hex|asm]  assemble (pa) disasm (pad) or esil (pae) from hexpairs
| p[bB] [len]       bitstream of N bytes
| pc[p] [len]       output C (or python) format
| p[dD][lf] [l]     disassemble N opcodes/bytes (see pd?)
| pf[?|.nam] [fmt]  print formatted data (pf.name, pf.name $<expr>)
| p[iI][df] [len]   print N instructions/bytes (f=func) (see pi? and pdi)
| pm [magic]        print libmagic data (pm? for more information)
| pr [len]          print N raw bytes
| p[kK] [len]       print key in randomart (K is for mosaic)
| ps[pwz] [len]     print pascal/wide/zero-terminated strings
| pt[dn?] [len]     print different timestamps
| pu[w] [len]       print N url encoded bytes (w=wide)
| pv[jh] [mode]     bar|json|histogram blocks (mode: e?search.in)
| p[xX][owq] [len]  hexdump of N bytes (o=octal, w=32bit, q=64bit)
| pz [len]          print zoom view (see pz? for help)
```

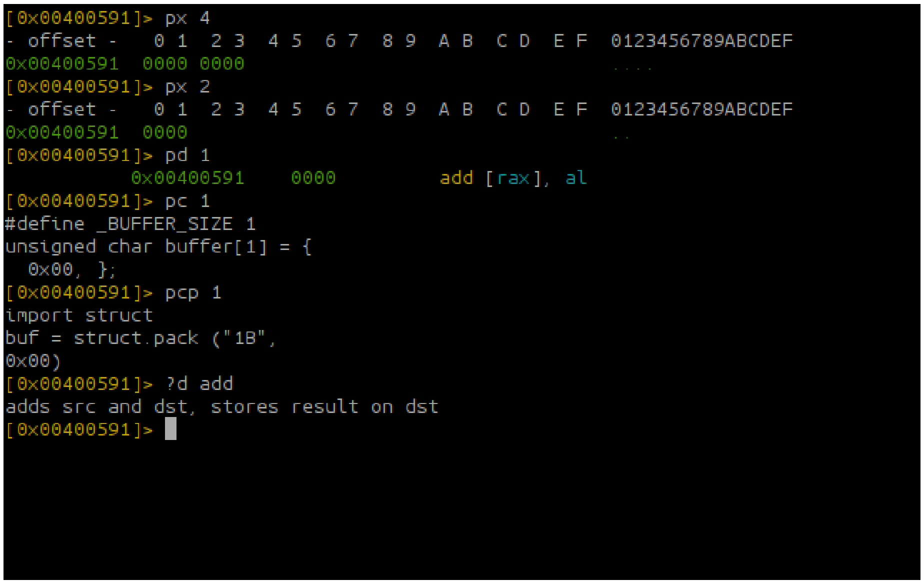


Figure 12: Print commands

Hexadecimal User-friendly way:

```
[0x00404888]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00404888  31ed 4989 d15e 4889 e248 83e4 f050 5449  1.I..^H..H...PTI
0x00404898  c7c0 4024 4100 48c7 c1b0 2341 0048 c7c7  ..@$A.H...#A.H..
0x004048a8  d028 4000 e83f dcff fff4 6690 662e 0f1f  .(@..?....f.f...
```

Show hexadecimal words dump (32bit)

```
[0x00404888]> pxw
0x00404888  0x8949ed31 0x89485ed1 0xe48348e2 0x495450f0  1.I..^H..H...PTI
0x00404898  0x2440c0c7 0xc7480041 0x4123b0c1 0xc7c74800  ..@$A.H...#A.H..
0x004048a8  0x004028d0 0xffdc3fe8 0x9066f4ff 0x1f0f2e66  .(@..?....f.f...

[0x00404888]> e cfg.bigendian
false

[0x00404888]> e cfg.bigendian = true

[0x00404888]> pxw
0x00404888  0x31ed4989 0xd15e4889 0xe24883e4 0xf0505449  1.I..^H..H...PTI
0x00404898  0xc7c04024 0x410048c7 0xc1b02341 0x0048c7c7  ..@$A.H...#A.H..
0x004048a8  0xd0284000 0xe83fdcff 0xfff46690 0x662e0f1f  .(@..?....f.f...
```

8bit hexpair list of bytes

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

Show hexadecimal quad-words dump (64bit)

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

Date formats The current supported timestamp print modes are:

```
[0x00404888]> pt?
|Usage: pt[dn?]
| pt      print unix time (32 bit cfg.big_endian)
| ptd     print dos time (32 bit cfg.big_endian)
| ptn     print ntfs time (64 bit !cfg.big_endian)
| pt?     show help message
```

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

```
[0x08048000]> eval cfg.bigendian = false
[0x08048000]> pt 4
29:04:32948 23:12:36 +0000
[0x08048000]> eval cfg.bigendian = true
[0x08048000]> pt 4
20:05:13001 09:29:21 +0000
```

As you can see, the endianness affects the print formats. Once you have printed a timestamp you can grep the results by the year for example:

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

The default date format can be configured using the `cfg.datefmt` variable. The field definitions follow the well-known strftime(3) format.

Source (asm, C) Valid print code formats are:

```
pc     C
pcs    string
pcj    json
pcJ    javascript
pcp    python
pcw    words (4 byte)
pcd    dwords (8 byte)

[0xB7F8E810]> pc 32
#define _BUFFER_SIZE 32
unsigned char buffer[_BUFFER_SIZE] = {
0x89, 0xe0, 0xe8, 0x49, 0x02, 0x00, 0x00, 0x89,
0xc7, 0xe8, 0xe2, 0xff, 0xff, 0xff, 0x81, 0xc3,
0xd6, 0xa7, 0x01, 0x00, 0x8b, 0x83, 0x00, 0xff,
0xff, 0xff, 0x5a, 0x8d, 0x24, 0x84, 0x29, 0xc2 };


[0x7fcd6a891630]> pcs
"\x48\x89\xe7\xe8\x68\x39\x00\x00\x49\x89
\xc4\x8b\x05\xef\x16\x22\x00\x5a\x48\x8d
\x24\xc4\x29\xc2\x52\x48\x89\xd6\x49\x89
\xe5\x48\x83\xe4\xf0\x48\x8b\x3d\x06\x1a"
```

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

Therefore radare supports various string formats:

```
[0x00404888]> ps?
|Usage: ps[zpw] [N]
| ps  = print string
| psb = print strings in current block
| psx = show strings with escaped chars
| psz = print zero terminated string
| psp = print pascal string
| psw = print wide string
```

Most strings will be zero-terminated. Here's an example by using the debugger to continue the execution of the program until it executes the 'open' syscall. When we recover the control over the process, we get the arguments passed to the syscall, pointed by %ebx. In the case of the 'open' call, this parameter is a zero terminated string which we can inspect using `psz`.

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

Disassembly The `pd` command is used to disassemble code. It accepts a numeric value to specify how many opcodes should be disassembled. The `pD` command is similar but instead of a number of instructions it decompiles a given number of bytes.

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

```
 [0x00404888]> pd 1
            ;-- entry0:
            0x00404888    31ed         xor ebp, ebp
```

Selecting the architecture The architecture flavour for the disassembly is defined by the `asm.arch` eval variable. You can use `e asm.arch = ?` to list all available architectures.

```
[0xB7F08810]> e asm.arch = ?
```

There are also multiple options that can be used to configure the output of the disassembler, all these options are described using `e? asm.` See also Eval Variable chapter.

The syntax variable is used to influence the flavour of assembly syntax the disassembler engine outputs.

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

You can also check asm.pseudo which is an experimental pseudocode view and asm.esil which outputs ESIL (‘Evaluable Strings Intermediate Language’). It aims to output a human readable representation of every opcode. Those representations can be evaluated in order to emulate the code.

XREF in radare2

Cross references (XREF) can help us determine where certain functions were called from.

In radare2, xref are displayed in disassembly like this:

```
|           ; DATA XREF from 0x080484f0 (sub.printf_4ec)
|           ;-- str.Great:
|           0x08048662     .string "Great" ; len=5
```

You can quickly get the xref using `axt @ str.Great` (find data/code references to this address).

Block size, Values and Flags in radare2

Block Size

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

[0xB7F9D810]> b? Usage: b[f] [arg] b display current block size b+3 increase blocksize by 3 b-16 decrement blocksize by 3 b 33 set block size to 33 b eip+4 numeric argument can be an expression bf foo set block size to flag size bm 1M set max block size

The `b` command is used to change the block size:

```
[0x00000000]> b 0x100   ; block size = 0x100
[0x00000000]> b +16     ;  ... = 0x110
[0x00000000]> b -32     ;  ... = 0xf0
```

The `bf` command is used to change the block size to the one specified by a flag. For example in symbols, the block size of the flag represents the size of the function.

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

You can perform these two operations in a single one (pdf):

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

Values

Values are numbers expressed in various formats:

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

Flags

Flagspaces are groups of flags. Some of them are automatically created by rabin while identifying strings, symbols, sections, etc., and others are updated at runtime like by commands like 'regs' (registers) or 'search' (search results).

Flags are similar to bookmarks. They represent a certain offset in the file. Flags can be grouped in 'flag spaces'. A flag space is something like a namespace for flags. They are used to group flags of similar characteristic or type. Some example of flagspaces could be sections, registers, symbols.

To create a flag just type:

```
[0x4A13B8C0]> f flag_name @ offset
```

You can remove a flag by prefixing its name with -. Most commands accept - as argument-prefix as a way to delete items.

```
[0x4A13B8C0]> f -flag_name
```

To switch between or create new flagspaces use the `fs` command:

```
[0x4A13B8C0]> fs   ; list flag spaces

00   symbols
01   imports
02   sections
03   strings
04   regs
05   maps

[0x4A13B8C0]> fs symbols ; select only flags in symbols flagspace
[0x4A13B8C0]> f          ; list only flags in symbols flagspace
[0x4A13B8C0]> fs *       ; select all flagspaces
```

You can rename flags with `fr`.

Variables

You can also use variables and seeks to build more complex expressions. Here are a few examples:

```
?@?    or stype @@?      ; misc help for '@' (seek), '~' (grep) (see ~??)
?$?            ; show available '$' variables
$$             ; here (current virtual seek)
$l             ; opcode length
$s             ; file size
$j             ; jump address (e.g. jmp 0x10, jz 0x10 => 0x10)
$f             ; jump fail address (e.g. jz 0x10 => next instruction)
$m             ; opcode memory reference (e.g. mov eax,[0x10] => 0x10)

? 1+2 // Do calculus and conversion hex/oct/bin...
```

You can also perform calculus with the `rax2` standalone tool

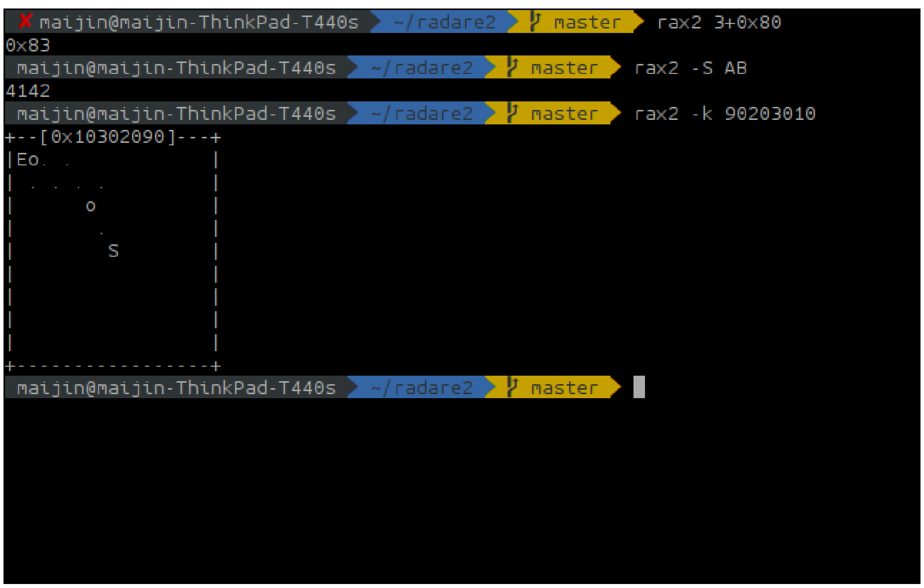


Figure 13: Rax2 commands

Basic Write commands

Radare can manipulate a loaded binary file in multiple ways. You can resize the file, move and copy/paste bytes, insert new bytes (shifting data to the end of the block or file) or simply overwrite bytes at an address, contents of a file, a widestring or even inline assembling an opcode.

To resize use the `r` command which accepts a numeric argument. A positive value sets the new size to the file. A negative one will strip N bytes from the current seek, down-sizing the file.

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

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

```
[0x00404888]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| w foobar     write string 'foobar'
| wh r2        whereis/which shell command
| wr 10        write 10 random bytes
| ww foobar    write wide string 'f\x00o\x00o\x00b\x00a\x00r\x00'
| wa push ebp  write opcode, separated by ';' (use '"' around the command)
| waf file     assemble file and write bytes
| wA r 0       alter/modify opcode at current seek (see wA?)
| wb 010203    fill current block with cyclic hexpairs
| wc[ir*?]     write cache undo/commit/reset/list (io.cache)
| wx 9090      write two intel nops
| wv eip+34    write 32-64 bit value
| wo? hex      write in block with operation. 'wo?' fmi
| wm f0ff      set binary mask hexpair to be used as cyclic write mask
| ws pstring   write 1 byte for length and then the string
| wf -|file    write contents of file at current offset
| wF -|file    write contents of hexpairs file here
| wp -|file    apply radare patch file. See wp? fmi
| wt file [sz] write to file (from current seek, blocksize or sz bytes)
```

Some examples:

```
 [0x00000000]> wx 123456 @ 0x8048300
 [0x00000000]> wv 0x8048123 @ 0x8049100
 [0x00000000]> wa jmp 0x8048320
```

Write over with operation The wo command (write operation) accepts multiple kinds of operations that can be applied on the current block. This is for example a XOR, ADD, SUB...

```
[0x4A13B8C0]> wo?
|Usage: wo[asmdxoArl24] [hexpairs] @ addr[:bsize]
|Example:
|  wox 0x90   ; xor cur block with 0x90
|  wox 90     ; xor cur block with 0x90
|  wox 0x0203 ; xor cur block with 0203
|  woa 02 03  ; add [0203][0203][...] to curblk
|  woe 02 03
|Supported operations:
|  wow  ==  write looped value (alias for 'wb')
|  woa  +=  addition
|  wos  -=  subtraction
|  wom  *=  multiply
|  wod  /=  divide
|  wox  ^=  xor
|  woo  |=  or
|  woA  &=  and
|  woR  random bytes (alias for 'wr $b'
|  wor  >>= shift right
|  wol  <<= shift left
|  wo2  2=  2 byte endian swap
|  wo4  4=  4 byte endian swap
```

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

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

```
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  4889 e7e8 6839 0000 4989 c48b 05ef 1622  H...h9..I......"
0x7fcd6a891640  005a 488d 24c4 29c2 5248 89d6 4989 e548  .ZH.$.).RH..I..H
0x7fcd6a891650  83e4 f048 8b3d 061a 2200 498d 4cd5 1049  ...H.=..".I.L..I
0x7fcd6a891660  8d55 0831 ede8 06e2 0000 488d 15cf e600  .U.1......H.....


[0x7fcd6a891630]> wox 90
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  d819 7778 d919 541b 90ca d81d c2d8 1946  ..wx..T........F
0x7fcd6a891640  1374 60d8 b290 d91d 1dc5 98a1 9090 d81d  .t`.............
0x7fcd6a891650  90dc 197c 9f8f 1490 d81d 95d9 9f8f 1490  ...|............
```

```
0x7fcd6a891660  13d7 9491 9f8f 1490 13ff 9491 9f8f 1490  ................


[0x7fcd6a891630]> woa 01 02
[0x7fcd6a891630]> px
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fcd6a891630  d91b 787a 91cc d91f 1476 61da 1ec7 99a3  ..xz.....va.....
0x7fcd6a891640  91de 1a7e d91f 96db 14d9 9593 1401 9593  ...~............
0x7fcd6a891650  c4da 1a6d e89a d959 9192 9159 1cb1 d959  ...m...Y...Y...Y
0x7fcd6a891660  9192 79cb 81da 1652 81da 1456 a252 7c77  ..y....R...V.R|w
```

Basic search commands

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

```
$ r2 -c "/ lib" -q /bin/ls
Searching 3 bytes from 0x00400000 to 0x0041ae08: 6c 69 62
hits: 9
0x00400239 hit0_0 "lib64/ld-linux-x86-64.so.2"
0x00400f19 hit0_1 "libselinux.so.1"
0x00400fae hit0_2 "librt.so.1"
0x00400fc7 hit0_3 "libacl.so.1"
0x00401004 hit0_4 "libc.so.6"
0x004013ce hit0_5 "libc_start_main"
0x00416542 hit0_6 "libs/"
0x00417160 hit0_7 "lib/xstrtol.c"
0x00417578 hit0_8 "lib"
```

`r2 -q` // quiet mode (no prompt) and quit after -i

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

```
[0x00404888]> / ls
...
[0x00404888]> ps @ hit0_0
lseek
```

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

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

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

```
[0x00000000]> / \x7FELF
```

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

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

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

```
[0x00000000]> f
0x00000135 512 hit0_0
0x00000b71 512 hit0_1
0x00000bad 512 hit0_2
0x00000bdd 512 hit0_3
0x00000bfb 512 hit0_4
0x00000f2a 512 hit0_5
```

To remove these flags, you can just use the `f@-hit*` command.

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

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

Search in assembly If you want to search for a certain type of opcodes you can either use `/c` or `/a`:

```
/c jmp [esp]    search for asm code
```

```
[0x00404888]> /c jmp qword [rdx]
f hit_0 @ 0x0040e50d   # 2: jmp qword [rdx]
f hit_1 @ 0x00418dbb   # 2: jmp qword [rdx]
f hit_2 @ 0x00418fcb   # 3: jmp qword [rdx]
f hit_3 @ 0x004196ab   # 6: jmp qword [rdx]
f hit_4 @ 0x00419bf3   # 3: jmp qword [rdx]
f hit_5 @ 0x00419c1b   # 3: jmp qword [rdx]
f hit_6 @ 0x00419c43   # 3: jmp qword [rdx]
```

```
/a jmp eax      assemble opcode and search its bytes
```

```
[0x00404888]> /a jmp eax
hits: 1
0x004048e7 hit3_0 ffe00f1f8000000000b8
```

Graph

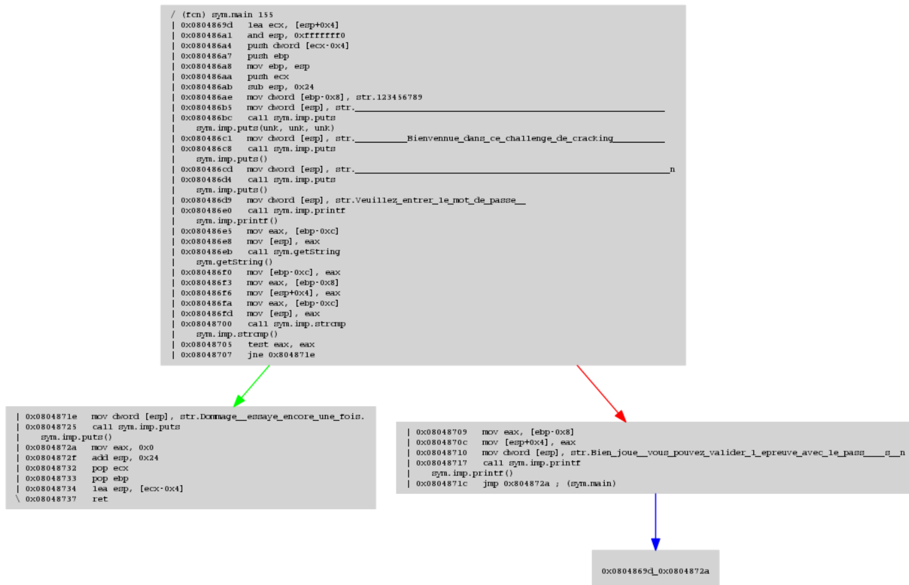


Figure 14: Graphviz ag $$ > a.dot

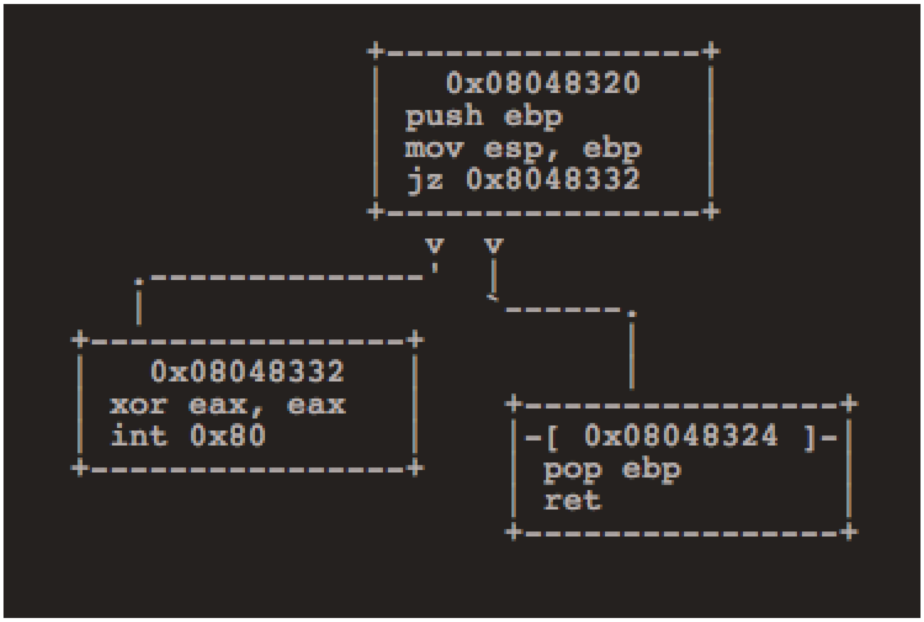


Figure 15: Ascii ART VVV

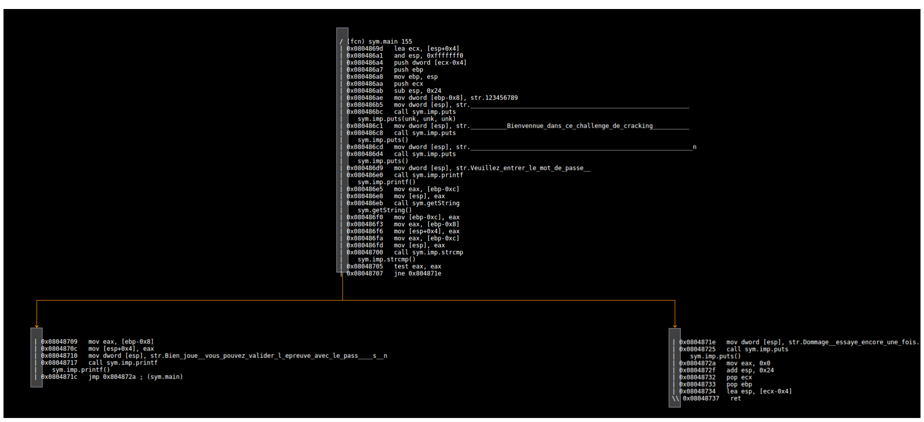


Figure 16: WebView `=H` , agv (display graph in web-ui)

Visual Mode

- `V`: Launch the visual mode
- `?`: To get help in the visual mode
- `d`: Define (define code, indefined, rename function) equivalent of ida rightclick define

- `;`: Add a comment
- `p` or `P`: Switch Visual view
- `_`: HUD
- `u` undo/Back to previous screen

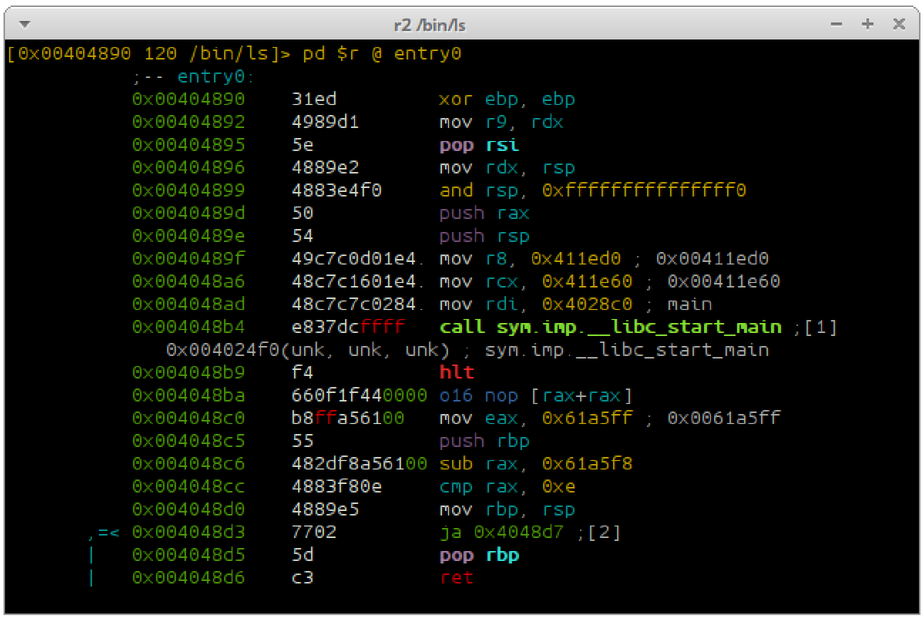


Figure 17: Disassembly Visual Mode

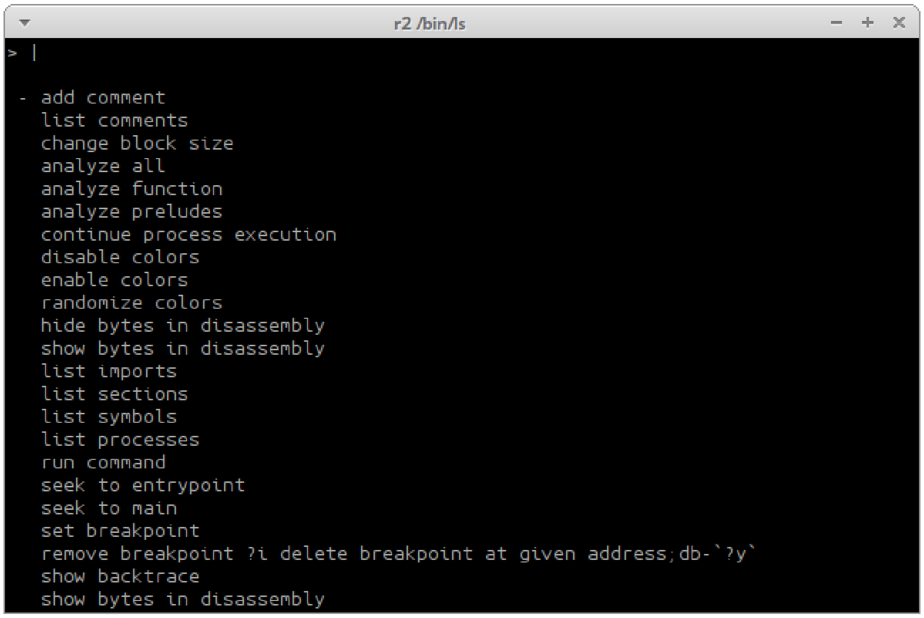


Figure 18: HUD in Visual Mode

Functions in Visual mode You can seek to a symbol or a function typing the number on next to it and get back using `u`, In this example you can type `3` to seek to `sym.imp.printf` symbol

```
|      ;--main:
|          0x08048330    55           push ebp
|          0x08048331    89e5         mov ebp, esp
|          0x08048333    83ec1c       sub esp, 0x1c
|          0x08048336    53           push ebx
|          0x08048337    c745fc00000. mov dword [ebp-0x4], 0x0
|          0x0804833e    c745f800000. mov dword [ebp-0x8], 0x0
|          0x08048345    686c850408   push str._n_tCrackme_1_by_syscalo_n ; str._n_tCrackme_
|          0x0804834a    e861ffffff   call sym.imp.printf ;[3]
```

You can also display a list a function and quickly navigate between them using `v` (Visual code analysis manipulation)

Figure 19: Visual code analysis manipulation

XREF in Visual mode Radare2 implements many user-friendly features for the visual interface to walk thru the assembly code. One of them is the `x` key that popups a menu for selecting the xref (data or code) against the current seek and then jump there. For example when pressing x when looking at those XREF:

```
|   ....--> ; CODE (CALL) XREF from 0x00402b98 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402ba0 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402ba9 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402bd5 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402beb (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402c25 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402c31 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402c40 (fcn.004028d0)
|   ....--> ; CODE (CALL) XREF from 0x00402c51 (fcn.004028d0)
```

After pressing `x`

```
[GOTO XREF]>
[0] CODE (CALL) XREF 0x00402b98 (loc.00402b38)
[1] CODE (CALL) XREF 0x00402ba0 (loc.00402b38)
[2] CODE (CALL) XREF 0x00402ba9 (loc.00402b38)
[3] CODE (CALL) XREF 0x00402bd5 (loc.00402b38)
[4] CODE (CALL) XREF 0x00402beb (loc.00402b38)
[5] CODE (CALL) XREF 0x00402c25 (loc.00402b38)
[6] CODE (CALL) XREF 0x00402c31 (loc.00402b38)
```

```
[7] CODE (CALL) XREF 0x00402c40 (loc.00402b38)
[8] CODE (CALL) XREF 0x00402c51 (loc.00402b38)
[9] CODE (CALL) XREF 0x00402c60 (loc.00402b38)
```

All the calls and jumps are numbered (1, 2, 3...) these numbers are the keybindings for seeking there from the visual mode. All the seek history is stored, by pressing **u** key you will go back in the seek history time :)

Figure 20: XREF in Visual mode

Let's tweak this interface

Eval Variable

All the configuration of radare2 is done with the eval command `e` which allows the user to change some variables from an internal hashtable containing string pairs.

These configurations can be also defined using the -e flag of radare2 while loading it, so you can setup different initial configurations from the command line.

```
radare2 -e scr.color=false file
```

You can also use the rc file: `~/.radare2rc` There are enhanced interfaces to help users to interactively configure this hashtable. One is `Ve` and provides a shell for walking through the tree and change variables. You can also get list of all variables with description `e??`

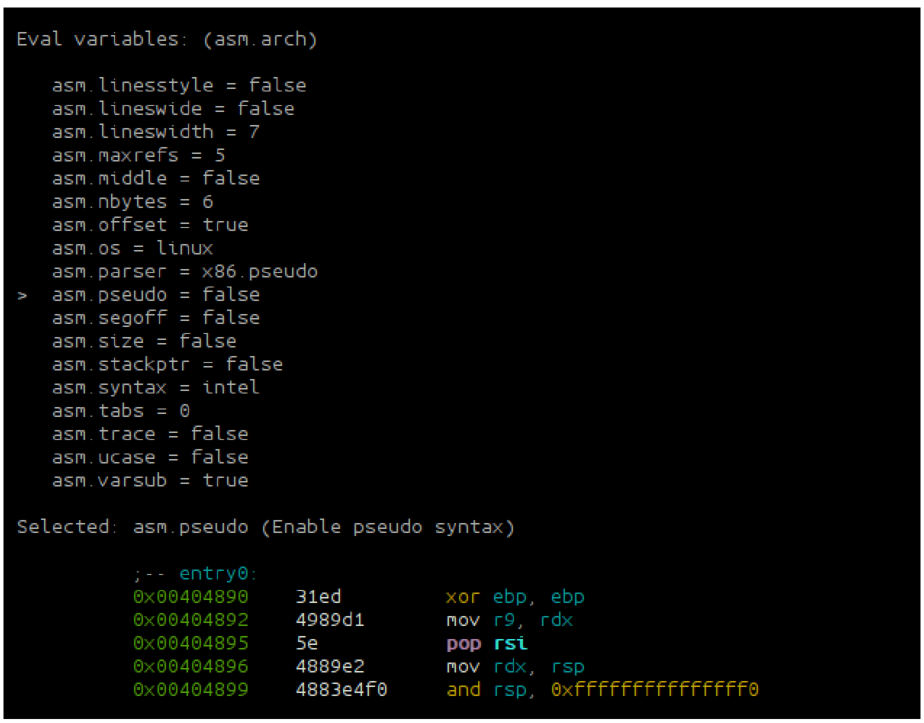


Figure 21: Ve command

- See the state of an eval variable: `e asm.pseudo`
- Set an eval variable: `e asm.pseudo = true`